Using the GNU Compiler Collection

For GCC version 9.1.0

(GCC)

Richard M. Stallman and the GCC Developer Community

Published by:

GNU Press
a division of the
Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301 USA

Website: http://www.gnupress.org
General: press@gnu.org
Orders: sales@gnu.org
Tel 617-542-5942
Fax 617-542-2652

Last printed October 2003 for GCC 3.3.1.
Printed copies are available for $45 each.

Copyright © 1988-2019 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "Funding Free Software", the Front-Cover Texts being (a) (see below), and with the Back-Cover Texts being (b) (see below). A copy of the license is included in the section entitled "GNU Free Documentation License".

(a) The FSF's Front-Cover Text is:

A GNU Manual

(b) The FSF's Back-Cover Text is:

You have freedom to copy and modify this GNU Manual, like GNU software. Copies published by the Free Software Foundation raise funds for GNU development.

Short Contents

Introduction 1
1 Programming Languages Supported by GCC 3
2 Language Standards Supported by GCC 5
3 GCC Command Options 11
4 C Implementation-Defined Behavior 441
5 C++ Implementation-Defined Behavior 449
6 Extensions to the C Language Family 451
7 Extensions to the C++ Language 815
8 GNU Objective-C Features 829
9 Binary Compatibility 845
10 `gcov`—a Test Coverage Program 849
11 `gcov-tool`—an Offline Gcda Profile Processing Tool 863
12 `gcov-dump`—an Offline Gcda and Gcno Profile Dump Tool . . 867
13 Known Causes of Trouble with GCC 869
14 Reporting Bugs 885
15 How To Get Help with GCC 887
16 Contributing to GCC Development 889
Funding Free Software 891
The GNU Project and GNU/Linux 893
GNU General Public License 895
GNU Free Documentation License 907
Contributors to GCC 915
Option Index 933
Keyword Index 959

Table of Contents

Introduction 1

1 Programming Languages Supported by GCC 3

2 Language Standards Supported by GCC 5

2.1 C Language 5
2.2 C++ Language 6
2.3 Objective-C and Objective-C++ Languages 7
2.4 Go Language 8
2.5 HSA Intermediate Language (HSAIL) 8
2.6 D language 8
2.7 References for Other Languages 9

3 GCC Command Options 11

3.1 Option Summary 11
3.2 Options Controlling the Kind of Output 31
3.3 Compiling C++ Programs 37
3.4 Options Controlling C Dialect 37
3.5 Options Controlling C++ Dialect 44
3.6 Options Controlling Objective-C and Objective-C++ Dialects . . 60
3.7 Options to Control Diagnostic Messages Formatting 63
3.8 Options to Request or Suppress Warnings 70
3.9 Options for Debugging Your Program 118
3.10 Options That Control Optimization 125
3.11 Program Instrumentation Options 189
3.12 Options Controlling the Preprocessor 202
3.13 Passing Options to the Assembler 209
3.14 Options for Linking 209
3.15 Options for Directory Search 215
3.16 Options for Code Generation Conventions 218
3.17 GCC Developer Options 228
3.18 Machine-Dependent Options 245
3.18.1 AArch64 Options 245
3.18.1.1 '`-march`' and '`-mcpu`' Feature Modifiers 250
3.18.2 Adapteva Epiphany Options 251
3.18.3 AMD GCN Options 253
3.18.4 ARC Options 254
3.18.5 ARM Options 263
3.18.6 AVR Options 277
3.18.6.1 `EIND` and Devices with More Than 128 Ki Bytes of Flash 282

3.18.6.2 Handling of the RAMPD, RAMPX, RAMPY and RAMPZ Special Function Registers ... 284
3.18.6.3 AVR Built-in Macros ... 284
3.18.7 Blackfin Options ... 286
3.18.8 C6X Options ... 289
3.18.9 CRIS Options ... 290
3.18.10 CR16 Options ... 291
3.18.11 C-SKY Options ... 292
3.18.12 Darwin Options ... 294
3.18.13 DEC Alpha Options ... 298
3.18.14 FR30 Options ... 302
3.18.15 FT32 Options ... 303
3.18.16 FRV Options ... 303
3.18.17 GNU/Linux Options ... 307
3.18.18 H8/300 Options ... 307
3.18.19 HPPA Options ... 308
3.18.20 IA-64 Options ... 311
3.18.21 LM32 Options ... 314
3.18.22 M32C Options ... 315
3.18.23 M32R/D Options ... 315
3.18.24 M680x0 Options ... 317
3.18.25 MCore Options ... 322
3.18.26 MeP Options ... 323
3.18.27 MicroBlaze Options ... 324
3.18.28 MIPS Options ... 326
3.18.29 MMIX Options ... 341
3.18.30 MN10300 Options ... 342
3.18.31 Moxie Options ... 343
3.18.32 MSP430 Options ... 343
3.18.33 NDS32 Options ... 345
3.18.34 Nios II Options ... 346
3.18.35 Nvidia PTX Options ... 351
3.18.36 OpenRISC Options ... 352
3.18.37 PDP-11 Options ... 353
3.18.38 picoChip Options ... 354
3.18.39 PowerPC Options ... 354
3.18.40 RISC-V Options ... 354
3.18.41 RL78 Options ... 356
3.18.42 IBM RS/6000 and PowerPC Options ... 358
3.18.43 RX Options ... 373
3.18.44 S/390 and zSeries Options ... 376
3.18.45 Score Options ... 380
3.18.46 SH Options ... 380
3.18.47 Solaris 2 Options ... 387
3.18.48 SPARC Options ... 387
3.18.49 SPU Options ... 393
3.18.50 Options for System V ... 395
3.18.51 TILE-Gx Options ... 396

3.18.52 TILEPro Options . . . 396
3.18.53 V850 Options . . . 396
3.18.54 VAX Options . . . 399
3.18.55 Visium Options . . . 399
3.18.56 VMS Options . . . 400
3.18.57 VxWorks Options . . . 400
3.18.58 x86 Options . . . 401
3.18.59 x86 Windows Options . . . 425
3.18.60 Xstormy16 Options . . . 426
3.18.61 Xtensa Options . . . 426
3.18.62 zSeries Options . . . 428
3.19 Specifying Subprocesses and the Switches to Pass to Them . . 428
3.20 Environment Variables Affecting GCC . . . 436
3.21 Using Precompiled Headers . . . 438

4 C Implementation-Defined Behavior . . . 441

4.1 Translation . . . 441
4.2 Environment . . . 441
4.3 Identifiers . . . 441
4.4 Characters . . . 442
4.5 Integers . . . 443
4.6 Floating Point . . . 443
4.7 Arrays and Pointers . . . 444
4.8 Hints . . . 445
4.9 Structures, Unions, Enumerations, and Bit-Fields . . . 445
4.10 Qualifiers . . . 446
4.11 Declarators . . . 447
4.12 Statements . . . 447
4.13 Preprocessing Directives . . . 447
4.14 Library Functions . . . 448
4.15 Architecture . . . 448
4.16 Locale-Specific Behavior . . . 448

5 C++ Implementation-Defined Behavior . . . 449

5.1 Conditionally-Supported Behavior . . . 449
5.2 Exception Handling . . . 449

6 Extensions to the C Language Family . . . 451

6.1 Statements and Declarations in Expressions . . . 451
6.2 Locally Declared Labels . . . 453
6.3 Labels as Values . . . 454
6.4 Nested Functions . . . 455
6.5 Nonlocal Gotos . . . 456
6.6 Constructing Function Calls . . . 457
6.7 Referring to a Type with `typeof` . . . 459
6.8 Conditionals with Omitted Operands . . . 460
6.9 128-bit Integers . . . 461

6.10 Double-Word Integers ... 461
6.11 Complex Numbers ... 461
6.12 Additional Floating Types ... 462
6.13 Half-Precision Floating Point ... 463
6.14 Decimal Floating Types ... 464
6.15 Hex Floats ... 464
6.16 Fixed-Point Types ... 464
6.17 Named Address Spaces ... 466
6.17.1 AVR Named Address Spaces ... 466
6.17.2 M32C Named Address Spaces ... 468
6.17.3 RL78 Named Address Spaces ... 468
6.17.4 SPU Named Address Spaces ... 468
6.17.5 x86 Named Address Spaces ... 468
6.18 Arrays of Length Zero ... 468
6.19 Structures with No Members ... 470
6.20 Arrays of Variable Length ... 470
6.21 Macros with a Variable Number of Arguments ... 471
6.22 Slightly Looser Rules for Escaped Newlines ... 472
6.23 Non-Lvalue Arrays May Have Subscripts ... 472
6.24 Arithmetic on `void`- and Function-Pointers ... 472
6.25 Pointer Arguments in Variadic Functions ... 473
6.26 Pointers to Arrays with Qualifiers Work as Expected ... 473
6.27 Non-Constant Initializers ... 473
6.28 Compound Literals ... 473
6.29 Designated Initializers ... 474
6.30 Case Ranges ... 476
6.31 Cast to a Union Type ... 476
6.32 Mixed Declarations and Code ... 477
6.33 Declaring Attributes of Functions ... 477
6.33.1 Common Function Attributes ... 478
6.33.2 AArch64 Function Attributes ... 497
6.33.2.1 Inlining rules ... 499
6.33.3 AMD GCN Function Attributes ... 499
6.33.4 ARC Function Attributes ... 501
6.33.5 ARM Function Attributes ... 502
6.33.6 AVR Function Attributes ... 504
6.33.7 Blackfin Function Attributes ... 505
6.33.8 CR16 Function Attributes ... 506
6.33.9 C-SKY Function Attributes ... 506
6.33.10 Epiphany Function Attributes ... 507
6.33.11 H8/300 Function Attributes ... 507
6.33.12 IA-64 Function Attributes ... 508
6.33.13 M32C Function Attributes ... 508
6.33.14 M32R/D Function Attributes ... 509
6.33.15 m68k Function Attributes ... 510
6.33.16 MCORE Function Attributes ... 510
6.33.17 MeP Function Attributes ... 510
6.33.18 MicroBlaze Function Attributes ... 511

6.33.19 Microsoft Windows Function Attributes 511
6.33.20 MIPS Function Attributes 512
6.33.21 MSP430 Function Attributes 514
6.33.22 NDS32 Function Attributes 516
6.33.23 Nios II Function Attributes 516
6.33.24 Nvidia PTX Function Attributes 517
6.33.25 PowerPC Function Attributes 517
6.33.26 RISC-V Function Attributes 520
6.33.27 RL78 Function Attributes 521
6.33.28 RX Function Attributes 521
6.33.29 S/390 Function Attributes 522
6.33.30 SH Function Attributes 523
6.33.31 SPU Function Attributes 524
6.33.32 Symbian OS Function Attributes 525
6.33.33 V850 Function Attributes 525
6.33.34 Visium Function Attributes 525
6.33.35 x86 Function Attributes 525
6.33.36 Xstormy16 Function Attributes 536
6.34 Specifying Attributes of Variables 536
6.34.1 Common Variable Attributes 536
6.34.2 ARC Variable Attributes 542
6.34.3 AVR Variable Attributes 542
6.34.4 Blackfin Variable Attributes 544
6.34.5 H8/300 Variable Attributes 544
6.34.6 IA-64 Variable Attributes 545
6.34.7 M32R/D Variable Attributes 545
6.34.8 MeP Variable Attributes 545
6.34.9 Microsoft Windows Variable Attributes 546
6.34.10 MSP430 Variable Attributes 547
6.34.11 Nvidia PTX Variable Attributes 547
6.34.12 PowerPC Variable Attributes 547
6.34.13 RL78 Variable Attributes 547
6.34.14 SPU Variable Attributes 547
6.34.15 V850 Variable Attributes 548
6.34.16 x86 Variable Attributes 548
6.34.17 Xstormy16 Variable Attributes 548
6.35 Specifying Attributes of Types 548
6.35.1 Common Type Attributes 549
6.35.2 ARC Type Attributes 556
6.35.3 ARM Type Attributes 556
6.35.4 MeP Type Attributes 557
6.35.5 PowerPC Type Attributes 557
6.35.6 SPU Type Attributes 557
6.35.7 x86 Type Attributes 557
6.36 Label Attributes 557
6.37 Enumerator Attributes 558
6.38 Statement Attributes 559
6.39 Attribute Syntax 559

6.40 Prototypes and Old-Style Function Definitions 563
6.41 C++ Style Comments 564
6.42 Dollar Signs in Identifier Names 564
6.43 The Character `ESC` in Constants 564
6.44 Determining the Alignment of Functions, Types or Variables 564
6.45 An Inline Function is As Fast As a Macro 564
6.46 When is a Volatile Object Accessed? 566
6.47 How to Use Inline Assembly Language in C Code 567
6.47.1 Basic Asm — Assembler Instructions Without Operands 567
6.47.2 Extended Asm - Assembler Instructions with C Expression Operands 569
6.47.2.1 Volatile 571
6.47.2.2 Assembler Template 573
6.47.2.3 Output Operands 574
6.47.2.4 Flag Output Operands 577
6.47.2.5 Input Operands 578
6.47.2.6 Clobbers and Scratch Registers 579
6.47.2.7 Goto Labels 582
6.47.2.8 x86 Operand Modifiers 583
6.47.2.9 x86 Floating-Point `asm` Operands 585
6.47.3 Constraints for `asm` Operands 586
6.47.3.1 Simple Constraints 586
6.47.3.2 Multiple Alternative Constraints 588
6.47.3.3 Constraint Modifier Characters 589
6.47.3.4 Constraints for Particular Machines 590
6.47.4 Controlling Names Used in Assembler Code 620
6.47.5 Variables in Specified Registers 621
6.47.5.1 Defining Global Register Variables 621
6.47.5.2 Specifying Registers for Local Variables 622
6.47.6 Size of an `asm` 623
6.48 Alternate Keywords 624
6.49 Incomplete `enum` Types 624
6.50 Function Names as Strings 624
6.51 Getting the Return or Frame Address of a Function 625
6.52 Using Vector Instructions through Built-in Functions 626
6.53 Support for `offsetof` 630
6.54 Legacy `__sync` Built-in Functions for Atomic Memory Access 630
6.55 Built-in Functions for Memory Model Aware Atomic Operations 632
6.56 Built-in Functions to Perform Arithmetic with Overflow Checking 636
6.57 x86-Specific Memory Model Extensions for Transactional Memory 638
6.58 Object Size Checking Built-in Functions 639
6.59 Other Built-in Functions Provided by GCC 641

6.60 Built-in Functions Specific to Particular Target Machines 656
6.60.1 AArch64 Built-in Functions 657
6.60.2 Alpha Built-in Functions 657
6.60.3 Altera Nios II Built-in Functions 658
6.60.4 ARC Built-in Functions 660
6.60.5 ARC SIMD Built-in Functions 662
6.60.6 ARM iWMMXt Built-in Functions 666
6.60.7 ARM C Language Extensions (ACLE) 668
6.60.8 ARM Floating Point Status and Control Intrinsics 669
6.60.9 ARM ARMv8-M Security Extensions 669
6.60.10 AVR Built-in Functions 669
6.60.11 Blackfin Built-in Functions 671
6.60.12 FR-V Built-in Functions 671
6.60.12.1 Argument Types 671
6.60.12.2 Directly-Mapped Integer Functions 671
6.60.12.3 Directly-Mapped Media Functions 672
6.60.12.4 Raw Read/Write Functions 674
6.60.12.5 Other Built-in Functions 674
6.60.13 MIPS DSP Built-in Functions 675
6.60.14 MIPS Paired-Single Support 679
6.60.15 MIPS Loongson Built-in Functions 679
6.60.15.1 Paired-Single Arithmetic 681
6.60.15.2 Paired-Single Built-in Functions 682
6.60.15.3 MIPS-3D Built-in Functions 683
6.60.16 MIPS SIMD Architecture (MSA) Support 685
6.60.16.1 MIPS SIMD Architecture Built-in Functions 686
6.60.17 Other MIPS Built-in Functions 699
6.60.18 MSP430 Built-in Functions 699
6.60.19 NDS32 Built-in Functions 699
6.60.20 picoChip Built-in Functions 700
6.60.21 Basic PowerPC Built-in Functions 700
6.60.21.1 Basic PowerPC Built-in Functions Available on all Configurations 701
6.60.21.2 Basic PowerPC Built-in Functions Available on ISA 2.05 704
6.60.21.3 Basic PowerPC Built-in Functions Available on ISA 2.06 706
6.60.21.4 Basic PowerPC Built-in Functions Available on ISA 2.07 706
6.60.21.5 Basic PowerPC Built-in Functions Available on ISA 3.0 706
6.60.22 PowerPC AltiVec/VSX Built-in Functions 709
6.60.22.1 PowerPC AltiVec Built-in Functions on ISA 2.05 .. 710
6.60.22.2 PowerPC AltiVec Built-in Functions Available on ISA 2.06 736
6.60.22.3 PowerPC AltiVec Built-in Functions Available on ISA 2.07 744

6.60.22.4 PowerPC AltiVec Built-in Functions Available on ISA 3.0 . . . 750
6.60.23 PowerPC Hardware Transactional Memory Built-in Functions . . . 760
6.60.23.1 PowerPC HTM Low Level Built-in Functions . . . 760
6.60.23.2 PowerPC HTM High Level Inline Functions . . . 763
6.60.24 PowerPC Atomic Memory Operation Functions . . . 764
6.60.25 RX Built-in Functions . . . 765
6.60.26 S/390 System z Built-in Functions . . . 766
6.60.27 SH Built-in Functions . . . 768
6.60.28 SPARC VIS Built-in Functions . . . 769
6.60.29 SPU Built-in Functions . . . 772
6.60.30 TI C6X Built-in Functions . . . 773
6.60.31 TILE-Gx Built-in Functions . . . 773
6.60.32 TILEPro Built-in Functions . . . 774
6.60.33 x86 Built-in Functions . . . 774
6.60.34 x86 Transactional Memory Intrinsics . . . 799
6.60.35 x86 Control-Flow Protection Intrinsics . . . 800
6.61 Format Checks Specific to Particular Target Machines . . . 801
6.61.1 Solaris Format Checks . . . 801
6.61.2 Darwin Format Checks . . . 801
6.62 Pragmas Accepted by GCC . . . 801
6.62.1 AArch64 Pragmas . . . 801
6.62.2 ARM Pragmas . . . 802
6.62.3 M32C Pragmas . . . 802
6.62.4 MeP Pragmas . . . 802
6.62.5 RS/6000 and PowerPC Pragmas . . . 803
6.62.6 S/390 Pragmas . . . 803
6.62.7 Darwin Pragmas . . . 803
6.62.8 Solaris Pragmas . . . 804
6.62.9 Symbol-Renaming Pragmas . . . 804
6.62.10 Structure-Layout Pragmas . . . 805
6.62.11 Weak Pragmas . . . 806
6.62.12 Diagnostic Pragmas . . . 806
6.62.13 Visibility Pragmas . . . 807
6.62.14 Push/Pop Macro Pragmas . . . 808
6.62.15 Function Specific Option Pragmas . . . 808
6.62.16 Loop-Specific Pragmas . . . 809
6.63 Unnamed Structure and Union Fields . . . 809
6.64 Thread-Local Storage . . . 810
6.64.1 ISO/IEC 9899:1999 Edits for Thread-Local Storage . . . 811
6.64.2 ISO/IEC 14882:1998 Edits for Thread-Local Storage . . . 812
6.65 Binary Constants using the '`0b`' Prefix . . . 813

7 Extensions to the C++ Language 815

7.1 When is a Volatile C++ Object Accessed? 815
7.2 Restricting Pointer Aliasing 815
7.3 Vague Linkage 816
7.4 C++ Interface and Implementation Pragmas 817
7.5 Where's the Template? 818
7.6 Extracting the Function Pointer from a Bound Pointer to Member Function 821
7.7 C++-Specific Variable, Function, and Type Attributes 821
7.8 Function Multiversioning 822
7.9 Type Traits 823
7.10 C++ Concepts 826
7.11 Deprecated Features 826
7.12 Backwards Compatibility 827

8 GNU Objective-C Features 829

8.1 GNU Objective-C Runtime API 829
8.1.1 Modern GNU Objective-C Runtime API 829
8.1.2 Traditional GNU Objective-C Runtime API 830
8.2 `+load`: Executing Code before `main` 830
8.2.1 What You Can and Cannot Do in `+load` 831
8.3 Type Encoding 832
8.3.1 Legacy Type Encoding 834
8.3.2 `@encode` 834
8.3.3 Method Signatures 835
8.4 Garbage Collection 835
8.5 Constant String Objects 836
8.6 `compatibility_alias` 837
8.7 Exceptions 837
8.8 Synchronization 839
8.9 Fast Enumeration 839
8.9.1 Using Fast Enumeration 839
8.9.2 C99-Like Fast Enumeration Syntax 839
8.9.3 Fast Enumeration Details 840
8.9.4 Fast Enumeration Protocol 841
8.10 Messaging with the GNU Objective-C Runtime 842
8.10.1 Dynamically Registering Methods 842
8.10.2 Forwarding Hook 842

9 Binary Compatibility 845

10 `gcov`—a Test Coverage Program 849

10.1 Introduction to `gcov` 849
10.2 Invoking `gcov` 849
10.3 Using `gcov` with GCC Optimization 859
10.4 Brief Description of `gcov` Data Files 861
10.5 Data File Relocation to Support Cross-Profiling 861

11 gcov-tool—an Offline Gcda Profile Processing Tool . . . 863
11.1 Introduction to gcov-tool . . . 863
11.2 Invoking gcov-tool . . . 863

12 gcov-dump—an Offline Gcda and Gcno Profile Dump Tool . . . 867
12.1 Introduction to gcov-dump . . . 867
12.2 Invoking gcov-dump . . . 867

13 Known Causes of Trouble with GCC . . . 869
13.1 Actual Bugs We Haven't Fixed Yet . . . 869
13.2 Interoperation . . . 869
13.3 Incompatibilities of GCC . . . 871
13.4 Fixed Header Files . . . 874
13.5 Standard Libraries . . . 874
13.6 Disappointments and Misunderstandings . . . 875
13.7 Common Misunderstandings with GNU C++ . . . 876
13.7.1 Declare *and* Define Static Members . . . 876
13.7.2 Name Lookup, Templates, and Accessing Members of Base Classes . . . 877
13.7.3 Temporaries May Vanish Before You Expect . . . 878
13.7.4 Implicit Copy-Assignment for Virtual Bases . . . 879
13.8 Certain Changes We Don't Want to Make . . . 880
13.9 Warning Messages and Error Messages . . . 883

14 Reporting Bugs . . . 885
14.1 Have You Found a Bug? . . . 885
14.2 How and Where to Report Bugs . . . 885

15 How To Get Help with GCC . . . 887

16 Contributing to GCC Development . . . 889

Funding Free Software . . . 891

The GNU Project and GNU/Linux . . . 893

GNU General Public License . . . 895

GNU Free Documentation License . . . 907
ADDENDUM: How to use this License for your documents . . . 914

Contributors to GCC . . . 915

Option Index **933**

Keyword Index **959**

Introduction

This manual documents how to use the GNU compilers, as well as their features and incompatibilities, and how to report bugs. It corresponds to the compilers (GCC) version 9.1.0. The internals of the GNU compilers, including how to port them to new targets and some information about how to write front ends for new languages, are documented in a separate manual. See Section "Introduction" in *GNU Compiler Collection (GCC) Internals*.

1 Programming Languages Supported by GCC

GCC stands for "GNU Compiler Collection". GCC is an integrated distribution of compilers for several major programming languages. These languages currently include C, C++, Objective-C, Objective-C++, Fortran, Ada, D, Go, and BRIG (HSAIL).

The abbreviation *GCC* has multiple meanings in common use. The current official meaning is "GNU Compiler Collection", which refers generically to the complete suite of tools. The name historically stood for "GNU C Compiler", and this usage is still common when the emphasis is on compiling C programs. Finally, the name is also used when speaking of the *language-independent* component of GCC: code shared among the compilers for all supported languages.

The language-independent component of GCC includes the majority of the optimizers, as well as the "back ends" that generate machine code for various processors.

The part of a compiler that is specific to a particular language is called the "front end". In addition to the front ends that are integrated components of GCC, there are several other front ends that are maintained separately. These support languages such as Mercury, and COBOL. To use these, they must be built together with GCC proper.

Most of the compilers for languages other than C have their own names. The C++ compiler is G++, the Ada compiler is GNAT, and so on. When we talk about compiling one of those languages, we might refer to that compiler by its own name, or as GCC. Either is correct.

Historically, compilers for many languages, including C++ and Fortran, have been implemented as "preprocessors" which emit another high level language such as C. None of the compilers included in GCC are implemented this way; they all generate machine code directly. This sort of preprocessor should not be confused with the *C preprocessor*, which is an integral feature of the C, C++, Objective-C and Objective-C++ languages.

2 Language Standards Supported by GCC

For each language compiled by GCC for which there is a standard, GCC attempts to follow one or more versions of that standard, possibly with some exceptions, and possibly with some extensions.

2.1 C Language

The original ANSI C standard (X3.159-1989) was ratified in 1989 and published in 1990. This standard was ratified as an ISO standard (ISO/IEC 9899:1990) later in 1990. There were no technical differences between these publications, although the sections of the ANSI standard were renumbered and became clauses in the ISO standard. The ANSI standard, but not the ISO standard, also came with a Rationale document. This standard, in both its forms, is commonly known as *C89*, or occasionally as *C90*, from the dates of ratification. To select this standard in GCC, use one of the options '`-ansi`', '`-std=c90`' or '`-std=iso9899:1990`'; to obtain all the diagnostics required by the standard, you should also specify '`-pedantic`' (or '`-pedantic-errors`' if you want them to be errors rather than warnings). See Section 3.4 [Options Controlling C Dialect], page 37.

Errors in the 1990 ISO C standard were corrected in two Technical Corrigenda published in 1994 and 1996. GCC does not support the uncorrected version.

An amendment to the 1990 standard was published in 1995. This amendment added digraphs and `__STDC_VERSION__` to the language, but otherwise concerned the library. This amendment is commonly known as *AMD1*; the amended standard is sometimes known as *C94* or *C95*. To select this standard in GCC, use the option '`-std=iso9899:199409`' (with, as for other standard versions, '`-pedantic`' to receive all required diagnostics).

A new edition of the ISO C standard was published in 1999 as ISO/IEC 9899:1999, and is commonly known as *C99*. (While in development, drafts of this standard version were referred to as *C9X*.) GCC has substantially complete support for this standard version; see `http://gcc.gnu.org/c99status.html` for details. To select this standard, use '`-std=c99`' or '`-std=iso9899:1999`'.

Errors in the 1999 ISO C standard were corrected in three Technical Corrigenda published in 2001, 2004 and 2007. GCC does not support the uncorrected version.

A fourth version of the C standard, known as *C11*, was published in 2011 as ISO/IEC 9899:2011. (While in development, drafts of this standard version were referred to as *C1X*.) GCC has substantially complete support for this standard, enabled with '`-std=c11`' or '`-std=iso9899:2011`'. A version with corrections integrated was prepared in 2017 and published in 2018 as ISO/IEC 9899:2018; it is known as *C17* and is supported with '`-std=c17`' or '`-std=iso9899:2017`'; the corrections are also applied with '`-std=c11`', and the only difference between the options is the value of `__STDC_VERSION__`.

A further version of the C standard, known as *C2X*, is under development; experimental and incomplete support for this is enabled with '`-std=c2x`'.

By default, GCC provides some extensions to the C language that, on rare occasions conflict with the C standard. See Chapter 6 [Extensions to the C Language Family], page 451. Some features that are part of the C99 standard are accepted as extensions in C90 mode, and some features that are part of the C11 standard are accepted as extensions in C90 and C99 modes. Use of the '`-std`' options listed above disables these extensions where they

conflict with the C standard version selected. You may also select an extended version of the C language explicitly with '`-std=gnu90`' (for C90 with GNU extensions), '`-std=gnu99`' (for C99 with GNU extensions) or '`-std=gnu11`' (for C11 with GNU extensions).

The default, if no C language dialect options are given, is '`-std=gnu11`'.

The ISO C standard defines (in clause 4) two classes of conforming implementation. A *conforming hosted implementation* supports the whole standard including all the library facilities; a *conforming freestanding implementation* is only required to provide certain library facilities: those in `<float.h>`, `<limits.h>`, `<stdarg.h>`, and `<stddef.h>`; since AMD1, also those in `<iso646.h>`; since C99, also those in `<stdbool.h>` and `<stdint.h>`; and since C11, also those in `<stdalign.h>` and `<stdnoreturn.h>`. In addition, complex types, added in C99, are not required for freestanding implementations.

The standard also defines two environments for programs, a *freestanding environment*, required of all implementations and which may not have library facilities beyond those required of freestanding implementations, where the handling of program startup and termination are implementation-defined; and a *hosted environment*, which is not required, in which all the library facilities are provided and startup is through a function `int main (void)` or `int main (int, char *[])`. An OS kernel is an example of a program running in a freestanding environment; a program using the facilities of an operating system is an example of a program running in a hosted environment.

GCC aims towards being usable as a conforming freestanding implementation, or as the compiler for a conforming hosted implementation. By default, it acts as the compiler for a hosted implementation, defining `__STDC_HOSTED__` as `1` and presuming that when the names of ISO C functions are used, they have the semantics defined in the standard. To make it act as a conforming freestanding implementation for a freestanding environment, use the option '`-ffreestanding`'; it then defines `__STDC_HOSTED__` to `0` and does not make assumptions about the meanings of function names from the standard library, with exceptions noted below. To build an OS kernel, you may well still need to make your own arrangements for linking and startup. See Section 3.4 [Options Controlling C Dialect], page 37.

GCC does not provide the library facilities required only of hosted implementations, nor yet all the facilities required by C99 of freestanding implementations on all platforms. To use the facilities of a hosted environment, you need to find them elsewhere (for example, in the GNU C library). See Section 13.5 [Standard Libraries], page 874.

Most of the compiler support routines used by GCC are present in '`libgcc`', but there are a few exceptions. GCC requires the freestanding environment provide `memcpy`, `memmove`, `memset` and `memcmp`. Finally, if `__builtin_trap` is used, and the target does not implement the `trap` pattern, then GCC emits a call to `abort`.

For references to Technical Corrigenda, Rationale documents and information concerning the history of C that is available online, see `http://gcc.gnu.org/readings.html`

2.2 C++ Language

GCC supports the original ISO C++ standard published in 1998, and the 2011 and 2014 revisions.

The original ISO C++ standard was published as the ISO standard (ISO/IEC 14882:1998) and amended by a Technical Corrigenda published in 2003 (ISO/IEC 14882:2003). These

standards are referred to as C++98 and C++03, respectively. GCC implements the majority of C++98 (`export` is a notable exception) and most of the changes in C++03. To select this standard in GCC, use one of the options '`-ansi`', '`-std=c++98`', or '`-std=c++03`'; to obtain all the diagnostics required by the standard, you should also specify '`-pedantic`' (or '`-pedantic-errors`' if you want them to be errors rather than warnings).

A revised ISO C++ standard was published in 2011 as ISO/IEC 14882:2011, and is referred to as C++11; before its publication it was commonly referred to as C++0x. C++11 contains several changes to the C++ language, all of which have been implemented in GCC. For details see `https://gcc.gnu.org/projects/cxx-status.html#cxx11`. To select this standard in GCC, use the option '`-std=c++11`'.

Another revised ISO C++ standard was published in 2014 as ISO/IEC 14882:2014, and is referred to as C++14; before its publication it was sometimes referred to as C++1y. C++14 contains several further changes to the C++ language, all of which have been implemented in GCC. For details see `https://gcc.gnu.org/projects/cxx-status.html#cxx14`. To select this standard in GCC, use the option '`-std=c++14`'.

The C++ language was further revised in 2017 and ISO/IEC 14882:2017 was published. This is referred to as C++17, and before publication was often referred to as C++1z. GCC supports all the changes in the new specification. For further details see `https://gcc.gnu.org/projects/cxx-status.html#cxx1z`. Use the option '`-std=c++17`' to select this variant of C++.

More information about the C++ standards is available on the ISO C++ committee's web site at `http://www.open-std.org/jtc1/sc22/wg21/`.

To obtain all the diagnostics required by any of the standard versions described above you should specify '`-pedantic`' or '`-pedantic-errors`', otherwise GCC will allow some non-ISO C++ features as extensions. See Section 3.8 [Warning Options], page 70.

By default, GCC also provides some additional extensions to the C++ language that on rare occasions conflict with the C++ standard. See Section 3.5 [C++ Dialect Options], page 44. Use of the '`-std`' options listed above disables these extensions where they they conflict with the C++ standard version selected. You may also select an extended version of the C++ language explicitly with '`-std=gnu++98`' (for C++98 with GNU extensions), or '`-std=gnu++11`' (for C++11 with GNU extensions), or '`-std=gnu++14`' (for C++14 with GNU extensions), or '`-std=gnu++17`' (for C++17 with GNU extensions).

The default, if no C++ language dialect options are given, is '`-std=gnu++14`'.

2.3 Objective-C and Objective-C++ Languages

GCC supports "traditional" Objective-C (also known as "Objective-C 1.0") and contains support for the Objective-C exception and synchronization syntax. It has also support for a number of "Objective-C 2.0" language extensions, including properties, fast enumeration (only for Objective-C), method attributes and the @optional and @required keywords in protocols. GCC supports Objective-C++ and features available in Objective-C are also available in Objective-C++.

GCC by default uses the GNU Objective-C runtime library, which is part of GCC and is not the same as the Apple/NeXT Objective-C runtime library used on Apple systems. There are a number of differences documented in this manual. The options '`-fgnu-runtime`'

and '-fnext-runtime' allow you to switch between producing output that works with the GNU Objective-C runtime library and output that works with the Apple/NeXT Objective-C runtime library.

There is no formal written standard for Objective-C or Objective-C++. The authoritative manual on traditional Objective-C (1.0) is "Object-Oriented Programming and the Objective-C Language": http://www.gnustep.org/resources/documentation/ObjectivCBook.pdf is the original NeXTstep document.

The Objective-C exception and synchronization syntax (that is, the keywords @try, @throw, @catch, @finally and @synchronized) is supported by GCC and is enabled with the option '-fobjc-exceptions'. The syntax is briefly documented in this manual and in the Objective-C 2.0 manuals from Apple.

The Objective-C 2.0 language extensions and features are automatically enabled; they include properties (via the @property, @synthesize and @dynamic keywords), fast enumeration (not available in Objective-C++), attributes for methods (such as deprecated, noreturn, sentinel, format), the unused attribute for method arguments, the @package keyword for instance variables and the @optional and @required keywords in protocols. You can disable all these Objective-C 2.0 language extensions with the option '-fobjc-std=objc1', which causes the compiler to recognize the same Objective-C language syntax recognized by GCC 4.0, and to produce an error if one of the new features is used.

GCC has currently no support for non-fragile instance variables.

The authoritative manual on Objective-C 2.0 is available from Apple:

- https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

For more information concerning the history of Objective-C that is available online, see http://gcc.gnu.org/readings.html

2.4 Go Language

As of the GCC 4.7.1 release, GCC supports the Go 1 language standard, described at https://golang.org/doc/go1.

2.5 HSA Intermediate Language (HSAIL)

GCC can compile the binary representation (BRIG) of the HSAIL text format as described in HSA Programmer's Reference Manual version 1.0.1. This capability is typically utilized to implement the HSA runtime API's HSAIL finalization extension for a gcc supported processor. HSA standards are freely available at http://www.hsafoundation.com/standards/ .

2.6 D language

GCC supports the D 2.0 programming language. The D language itself is currently defined by its reference implementation and supporting language specification, described at https://dlang.org/spec/spec.html.

2.7 References for Other Languages

See Section "About This Guide" in *GNAT Reference Manual*, for information on standard conformance and compatibility of the Ada compiler.

See Section "Standards" in *The GNU Fortran Compiler*, for details of standards supported by GNU Fortran.

3 GCC Command Options

When you invoke GCC, it normally does preprocessing, compilation, assembly and linking. The "overall options" allow you to stop this process at an intermediate stage. For example, the '`-c`' option says not to run the linker. Then the output consists of object files output by the assembler. See Section 3.2 [Options Controlling the Kind of Output], page 31.

Other options are passed on to one or more stages of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; most of these are not documented here, since you rarely need to use any of them.

Most of the command-line options that you can use with GCC are useful for C programs; when an option is only useful with another language (usually C++), the explanation says so explicitly. If the description for a particular option does not mention a source language, you can use that option with all supported languages.

The usual way to run GCC is to run the executable called `gcc`, or *`machine`*`-gcc` when cross-compiling, or *`machine`*`-gcc-`*`version`* to run a specific version of GCC. When you compile C++ programs, you should invoke GCC as `g++` instead. See Section 3.3 [Compiling C++ Programs], page 37, for information about the differences in behavior between `gcc` and `g++` when compiling C++ programs.

The `gcc` program accepts options and file names as operands. Many options have multi-letter names; therefore multiple single-letter options may *not* be grouped: '`-dv`' is very different from '`-d -v`'.

You can mix options and other arguments. For the most part, the order you use doesn't matter. Order does matter when you use several options of the same kind; for example, if you specify '`-L`' more than once, the directories are searched in the order specified. Also, the placement of the '`-l`' option is significant.

Many options have long names starting with '`-f`' or with '`-W`'—for example, '`-fmove-loop-invariants`', '`-Wformat`' and so on. Most of these have both positive and negative forms; the negative form of '`-ffoo`' is '`-fno-foo`'. This manual documents only one of these two forms, whichever one is not the default.

Some options take one or more arguments typically separated either by a space or by the equals sign ('`=`') from the option name. Unless documented otherwise, an argument can be either numeric or a string. Numeric arguments must typically be small unsigned decimal or hexadecimal integers. Hexadecimal arguments must begin with the '`0x`' prefix. Arguments to options that specify a size threshold of some sort may be arbitrarily large decimal or hexadecimal integers followed by a byte size suffix designating a multiple of bytes such as `kB` and `KiB` for kilobyte and kibibyte, respectively, `MB` and `MiB` for megabyte and mebibyte, `GB` and `GiB` for gigabyte and gigibyte, and so on. Such arguments are designated by *byte-size* in the following text. Refer to the NIST, IEC, and other relevant national and international standards for the full listing and explanation of the binary and decimal byte size prefixes.

See [Option Index], page 933, for an index to GCC's options.

3.1 Option Summary

Here is a summary of all the options, grouped by type. Explanations are in the following sections.

Overall Options

See Section 3.2 [Options Controlling the Kind of Output], page 31.

```
-c -S -E -o file -x language
-v -### --help[=class[,...]] --target-help --version
-pass-exit-codes -pipe -specs=file -wrapper
@file -ffile-prefix-map=old=new
-fplugin=file -fplugin-arg-name=arg
-fdump-ada-spec[-slim] -fada-spec-parent=unit -fdump-go-spec=file
```

C Language Options

See Section 3.4 [Options Controlling C Dialect], page 37.

```
-ansi -std=standard -fgnu89-inline
-fpermitted-flt-eval-methods=standard
-aux-info filename -fallow-parameterless-variadic-functions
-fno-asm -fno-builtin -fno-builtin-function -fgimple
-fhosted -ffreestanding
-fopenacc -fopenacc-dim=geom
-fopenmp -fopenmp-simd
-fms-extensions -fplan9-extensions -fsso-struct=endianness
-fallow-single-precision -fcond-mismatch -flax-vector-conversions
-fsigned-bitfields -fsigned-char
-funsigned-bitfields -funsigned-char
```

C++ Language Options

See Section 3.5 [Options Controlling C++ Dialect], page 44.

```
-fabi-version=n -fno-access-control
-faligned-new=n -fargs-in-order=n -fchar8_t -fcheck-new
-fconstexpr-depth=n -fconstexpr-loop-limit=n
-fconstexpr-ops-limit=n -fno-elide-constructors
-fno-enforce-eh-specs
-fno-gnu-keywords
-fno-implicit-templates
-fno-implicit-inline-templates
-fno-implement-inlines -fms-extensions
-fnew-inheriting-ctors
-fnew-ttp-matching
-fno-nonansi-builtins -fnothrow-opt -fno-operator-names
-fno-optional-diags -fpermissive
-fno-pretty-templates
-frepo -fno-rtti -fsized-deallocation
-ftemplate-backtrace-limit=n
-ftemplate-depth=n
-fno-threadsafe-statics -fuse-cxa-atexit
-fno-weak -nostdinc++
-fvisibility-inlines-hidden
-fvisibility-ms-compat
-fext-numeric-literals
-Wabi=n -Wabi-tag -Wconversion-null -Wctor-dtor-privacy
-Wdelete-non-virtual-dtor -Wdeprecated-copy -Wdeprecated-copy-dtor
-Wliteral-suffix
-Wmultiple-inheritance -Wno-init-list-lifetime
-Wnamespaces -Wnarrowing
-Wpessimizing-move -Wredundant-move
-Wnoexcept -Wnoexcept-type -Wclass-memaccess
-Wnon-virtual-dtor -Wreorder -Wregister
-Weffc++ -Wstrict-null-sentinel -Wtemplates
-Wno-non-template-friend -Wold-style-cast
-Woverloaded-virtual -Wno-pmf-conversions
```

```
-Wno-class-conversion -Wno-terminate
-Wsign-promo -Wvirtual-inheritance
```

Objective-C and Objective-C++ Language Options

See Section 3.6 [Options Controlling Objective-C and Objective-C++ Dialects], page 60.

```
-fconstant-string-class=class-name
-fgnu-runtime -fnext-runtime
-fno-nil-receivers
-fobjc-abi-version=n
-fobjc-call-cxx-cdtors
-fobjc-direct-dispatch
-fobjc-exceptions
-fobjc-gc
-fobjc-nilcheck
-fobjc-std=objc1
-fno-local-ivars
-fivar-visibility=[public|protected|private|package]
-freplace-objc-classes
-fzero-link
-gen-decls
-Wassign-intercept
-Wno-protocol -Wselector
-Wstrict-selector-match
-Wundeclared-selector
```

Diagnostic Message Formatting Options

See Section 3.7 [Options to Control Diagnostic Messages Formatting], page 63.

```
-fmessage-length=n
-fdiagnostics-show-location=[once|every-line]
-fdiagnostics-color=[auto|never|always]
-fdiagnostics-format=[text|json]
-fno-diagnostics-show-option -fno-diagnostics-show-caret
-fno-diagnostics-show-labels -fno-diagnostics-show-line-numbers
-fdiagnostics-minimum-margin-width=width
-fdiagnostics-parseable-fixits -fdiagnostics-generate-patch
-fdiagnostics-show-template-tree -fno-elide-type
-fno-show-column
```

Warning Options

See Section 3.8 [Options to Request or Suppress Warnings], page 70.

```
-fsyntax-only -fmax-errors=n -Wpedantic
-pedantic-errors
-w -Wextra -Wall -Waddress -Waddress-of-packed-member
-Waggregate-return -Waligned-new
-Walloc-zero -Walloc-size-larger-than=byte-size
-Walloca -Walloca-larger-than=byte-size
-Wno-aggressive-loop-optimizations -Warray-bounds -Warray-bounds=n
-Wno-attributes -Wattribute-alias=n
-Wbool-compare -Wbool-operation
-Wno-builtin-declaration-mismatch
-Wno-builtin-macro-redefined -Wc90-c99-compat -Wc99-c11-compat
-Wc++-compat -Wc++11-compat -Wc++14-compat -Wc++17-compat
-Wcast-align -Wcast-align=strict -Wcast-function-type -Wcast-qual
-Wchar-subscripts -Wcatch-value -Wcatch-value=n
-Wclobbered -Wcomment -Wconditionally-supported
-Wconversion -Wcoverage-mismatch -Wno-cpp -Wdangling-else -Wdate-time
```

-Wdelete-incomplete
-Wno-attribute-warning
-Wno-deprecated -Wno-deprecated-declarations -Wno-designated-init
-Wdisabled-optimization
-Wno-discarded-qualifiers -Wno-discarded-array-qualifiers
-Wno-div-by-zero -Wdouble-promotion
-Wduplicated-branches -Wduplicated-cond
-Wempty-body -Wenum-compare -Wno-endif-labels -Wexpansion-to-defined
-Werror -Werror=* -Wextra-semi -Wfatal-errors
-Wfloat-equal -Wformat -Wformat=2
-Wno-format-contains-nul -Wno-format-extra-args
-Wformat-nonliteral -Wformat-overflow=*n*
-Wformat-security -Wformat-signedness -Wformat-truncation=*n*
-Wformat-y2k -Wframe-address
-Wframe-larger-than=*byte-size* -Wno-free-nonheap-object
-Wjump-misses-init
-Whsa -Wif-not-aligned
-Wignored-qualifiers -Wignored-attributes -Wincompatible-pointer-types
-Wimplicit -Wimplicit-fallthrough -Wimplicit-fallthrough=*n*
-Wimplicit-function-declaration -Wimplicit-int
-Winit-self -Winline -Wno-int-conversion -Wint-in-bool-context
-Wno-int-to-pointer-cast -Winvalid-memory-model -Wno-invalid-offsetof
-Winvalid-pch -Wlarger-than=*byte-size*
-Wlogical-op -Wlogical-not-parentheses -Wlong-long
-Wmain -Wmaybe-uninitialized -Wmemset-elt-size -Wmemset-transposed-args
-Wmisleading-indentation -Wmissing-attributes -Wmissing-braces
-Wmissing-field-initializers -Wmissing-format-attribute
-Wmissing-include-dirs -Wmissing-noreturn -Wmissing-profile
-Wno-multichar -Wmultistatement-macros -Wnonnull -Wnonnull-compare
-Wnormalized=[none|id|nfc|nfkc]
-Wnull-dereference -Wodr -Wno-overflow -Wopenmp-simd
-Woverride-init-side-effects -Woverlength-strings
-Wpacked -Wpacked-bitfield-compat -Wpacked-not-aligned -Wpadded
-Wparentheses -Wno-pedantic-ms-format
-Wplacement-new -Wplacement-new=*n*
-Wpointer-arith -Wpointer-compare -Wno-pointer-to-int-cast
-Wno-pragmas -Wno-prio-ctor-dtor -Wredundant-decls
-Wrestrict -Wno-return-local-addr
-Wreturn-type -Wsequence-point -Wshadow -Wno-shadow-ivar
-Wshadow=global, -Wshadow=local, -Wshadow=compatible-local
-Wshift-overflow -Wshift-overflow=*n*
-Wshift-count-negative -Wshift-count-overflow -Wshift-negative-value
-Wsign-compare -Wsign-conversion -Wfloat-conversion
-Wno-scalar-storage-order -Wsizeof-pointer-div
-Wsizeof-pointer-memaccess -Wsizeof-array-argument
-Wstack-protector -Wstack-usage=*byte-size* -Wstrict-aliasing
-Wstrict-aliasing=n -Wstrict-overflow -Wstrict-overflow=*n*
-Wstringop-overflow=*n* -Wstringop-truncation -Wsubobject-linkage
-Wsuggest-attribute=[pure|const|noreturn|format|malloc]
-Wsuggest-final-types
-Wsuggest-final-methods -Wsuggest-override
-Wswitch -Wswitch-bool -Wswitch-default -Wswitch-enum
-Wswitch-unreachable -Wsync-nand
-Wsystem-headers -Wtautological-compare -Wtrampolines -Wtrigraphs
-Wtype-limits -Wundef
-Wuninitialized -Wunknown-pragmas
-Wunsuffixed-float-constants -Wunused -Wunused-function
-Wunused-label -Wunused-local-typedefs -Wunused-macros

-Wunused-parameter -Wno-unused-result
-Wunused-value -Wunused-variable
-Wunused-const-variable -Wunused-const-variable=*n*
-Wunused-but-set-parameter -Wunused-but-set-variable
-Wuseless-cast -Wvariadic-macros -Wvector-operation-performance
-Wvla -Wvla-larger-than=*byte-size* -Wvolatile-register-var
-Wwrite-strings
-Wzero-as-null-pointer-constant

C and Objective-C-only Warning Options

-Wbad-function-cast -Wmissing-declarations
-Wmissing-parameter-type -Wmissing-prototypes -Wnested-externs
-Wold-style-declaration -Wold-style-definition
-Wstrict-prototypes -Wtraditional -Wtraditional-conversion
-Wdeclaration-after-statement -Wpointer-sign

Debugging Options

See Section 3.9 [Options for Debugging Your Program], page 118.

-g -g*level* -gdwarf -gdwarf-*version*
-ggdb -grecord-gcc-switches -gno-record-gcc-switches
-gstabs -gstabs+ -gstrict-dwarf -gno-strict-dwarf
-gas-loc-support -gno-as-loc-support
-gas-locview-support -gno-as-locview-support
-gcolumn-info -gno-column-info
-gstatement-frontiers -gno-statement-frontiers
-gvariable-location-views -gno-variable-location-views
-ginternal-reset-location-views -gno-internal-reset-location-views
-ginline-points -gno-inline-points
-gvms -gxcoff -gxcoff+ -gz[=*type*]
-gsplit-dwarf -gdescribe-dies -gno-describe-dies
-fdebug-prefix-map=*old*=*new* -fdebug-types-section
-fno-eliminate-unused-debug-types
-femit-struct-debug-baseonly -femit-struct-debug-reduced
-femit-struct-debug-detailed[=*spec-list*]
-feliminate-unused-debug-symbols -femit-class-debug-always
-fno-merge-debug-strings -fno-dwarf2-cfi-asm
-fvar-tracking -fvar-tracking-assignments

Optimization Options

See Section 3.10 [Options that Control Optimization], page 125.

-faggressive-loop-optimizations
-falign-functions[=*n*[:*m*:[*n2*[:*m2*]]]]
-falign-jumps[=*n*[:*m*:[*n2*[:*m2*]]]]
-falign-labels[=*n*[:*m*:[*n2*[:*m2*]]]]
-falign-loops[=*n*[:*m*:[*n2*[:*m2*]]]]
-fassociative-math -fauto-profile -fauto-profile[=*path*]
-fauto-inc-dec -fbranch-probabilities
-fbranch-target-load-optimize -fbranch-target-load-optimize2
-fbtr-bb-exclusive -fcaller-saves
-fcombine-stack-adjustments -fconserve-stack
-fcompare-elim -fcprop-registers -fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules
-fcx-limited-range
-fdata-sections -fdce -fdelayed-branch
-fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively
-fdevirtualize-at-ltrans -fdse
-fearly-inlining -fipa-sra -fexpensive-optimizations -ffat-lto-objects
-ffast-math -ffinite-math-only -ffloat-store -fexcess-precision=*style*

-fforward-propagate -ffp-contract=*style* -ffunction-sections
-fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgraphite-identity
-fgcse-sm -fhoist-adjacent-loads -fif-conversion
-fif-conversion2 -findirect-inlining
-finline-functions -finline-functions-called-once -finline-limit=*n*
-finline-small-functions -fipa-cp -fipa-cp-clone
-fipa-bit-cp -fipa-vrp -fipa-pta -fipa-profile -fipa-pure-const
-fipa-reference -fipa-reference-addressable
-fipa-stack-alignment -fipa-icf -fira-algorithm=*algorithm*
-flive-patching=*level*
-fira-region=*region* -fira-hoist-pressure
-fira-loop-pressure -fno-ira-share-save-slots
-fno-ira-share-spill-slots
-fisolate-erroneous-paths-dereference -fisolate-erroneous-paths-attribute
-fivopts -fkeep-inline-functions -fkeep-static-functions
-fkeep-static-consts -flimit-function-alignment -flive-range-shrinkage
-floop-block -floop-interchange -floop-strip-mine
-floop-unroll-and-jam -floop-nest-optimize
-floop-parallelize-all -flra-remat -flto -flto-compression-level
-flto-partition=*alg* -fmerge-all-constants
-fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves
-fmove-loop-invariants -fno-branch-count-reg
-fno-defer-pop -fno-fp-int-builtin-inexact -fno-function-cse
-fno-guess-branch-probability -fno-inline -fno-math-errno -fno-peephole
-fno-peephole2 -fno-printf-return-value -fno-sched-interblock
-fno-sched-spec -fno-signed-zeros
-fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss
-fomit-frame-pointer -foptimize-sibling-calls
-fpartial-inlining -fpeel-loops -fpredictive-commoning
-fprefetch-loop-arrays
-fprofile-correction
-fprofile-use -fprofile-use=*path* -fprofile-values
-fprofile-reorder-functions
-freciprocal-math -free -frename-registers -freorder-blocks
-freorder-blocks-algorithm=*algorithm*
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop -freschedule-modulo-scheduled-loops
-frounding-math -fsave-optimization-record
-fsched2-use-superblocks -fsched-pressure
-fsched-spec-load -fsched-spec-load-dangerous
-fsched-stalled-insns-dep[=*n*] -fsched-stalled-insns[=*n*]
-fsched-group-heuristic -fsched-critical-path-heuristic
-fsched-spec-insn-heuristic -fsched-rank-heuristic
-fsched-last-insn-heuristic -fsched-dep-count-heuristic
-fschedule-fusion
-fschedule-insns -fschedule-insns2 -fsection-anchors
-fselective-scheduling -fselective-scheduling2
-fsel-sched-pipelining -fsel-sched-pipelining-outer-loops
-fsemantic-interposition -fshrink-wrap -fshrink-wrap-separate
-fsignaling-nans
-fsingle-precision-constant -fsplit-ivs-in-unroller -fsplit-loops
-fsplit-paths
-fsplit-wide-types -fssa-backprop -fssa-phiopt
-fstdarg-opt -fstore-merging -fstrict-aliasing
-fthread-jumps -ftracer -ftree-bit-ccp
-ftree-builtin-call-dce -ftree-ccp -ftree-ch
-ftree-coalesce-vars -ftree-copy-prop -ftree-dce -ftree-dominator-opts
-ftree-dse -ftree-forwprop -ftree-fre -fcode-hoisting

```
-ftree-loop-if-convert -ftree-loop-im
-ftree-phiprop -ftree-loop-distribution -ftree-loop-distribute-patterns
-ftree-loop-ivcanon -ftree-loop-linear -ftree-loop-optimize
-ftree-loop-vectorize
-ftree-parallelize-loops=n -ftree-pre -ftree-partial-pre -ftree-pta
-ftree-reassoc -ftree-scev-cprop -ftree-sink -ftree-slsr -ftree-sra
-ftree-switch-conversion -ftree-tail-merge
-ftree-ter -ftree-vectorize -ftree-vrp -funconstrained-commons
-funit-at-a-time -funroll-all-loops -funroll-loops
-funsafe-math-optimizations -funswitch-loops
-fipa-ra -fvariable-expansion-in-unroller -fvect-cost-model -fvpt
-fweb -fwhole-program -fwpa -fuse-linker-plugin
--param name=value -O -O0 -O1 -O2 -O3 -Os -Ofast -Og
```

Program Instrumentation Options

See Section 3.11 [Program Instrumentation Options], page 189.

```
-p -pg -fprofile-arcs --coverage -ftest-coverage
-fprofile-abs-path
-fprofile-dir=path -fprofile-generate -fprofile-generate=path
-fprofile-update=method -fprofile-filter-files=regex
-fprofile-exclude-files=regex
-fsanitize=style -fsanitize-recover -fsanitize-recover=style
-fasan-shadow-offset=number -fsanitize-sections=s1,s2,...
-fsanitize-undefined-trap-on-error -fbounds-check
-fcf-protection=[full|branch|return|none]
-fstack-protector -fstack-protector-all -fstack-protector-strong
-fstack-protector-explicit -fstack-check
-fstack-limit-register=reg -fstack-limit-symbol=sym
-fno-stack-limit -fsplit-stack
-fvtable-verify=[std|preinit|none]
-fvtv-counts -fvtv-debug
-finstrument-functions
-finstrument-functions-exclude-function-list=sym,sym,...
-finstrument-functions-exclude-file-list=file,file,...
```

Preprocessor Options

See Section 3.12 [Options Controlling the Preprocessor], page 202.

```
-Aquestion=answer
-A-question[=answer]
-C -CC -Dmacro[=defn]
-dD -dI -dM -dN -dU
-fdebug-cpp -fdirectives-only -fdollars-in-identifiers
-fexec-charset=charset -fextended-identifiers
-finput-charset=charset -fmacro-prefix-map=old=new
-fno-canonical-system-headers -fpch-deps -fpch-preprocess
-fpreprocessed -ftabstop=width -ftrack-macro-expansion
-fwide-exec-charset=charset -fworking-directory
-H -imacros file -include file
-M -MD -MF -MG -MM -MMD -MP -MQ -MT
-no-integrated-cpp -P -pthread -remap
-traditional -traditional-cpp -trigraphs
-Umacro -undef
-Wp,option -Xpreprocessor option
```

Assembler Options

See Section 3.13 [Passing Options to the Assembler], page 209.

```
-Wa,option -Xassembler option
```

Linker Options

See Section 3.14 [Options for Linking], page 209.

```
object-file-name -fuse-ld=linker -llibrary
-nostartfiles -nodefaultlibs -nolibc -nostdlib
-e entry --entry=entry
-pie -pthread -r -rdynamic
-s -static -static-pie -static-libgcc -static-libstdc++
-static-libasan -static-libtsan -static-liblsan -static-libubsan
-shared -shared-libgcc -symbolic
-T script -Wl,option -Xlinker option
-u symbol -z keyword
```

Directory Options

See Section 3.15 [Options for Directory Search], page 215.

```
-Bprefix -Idir -I-
-idirafter dir
-imacros file -imultilib dir
-iplugindir=dir -iprefix file
-iquote dir -isysroot dir -isystem dir
-iwithprefix dir -iwithprefixbefore dir
-Ldir -no-canonical-prefixes --no-sysroot-suffix
-nostdinc -nostdinc++ --sysroot=dir
```

Code Generation Options

See Section 3.16 [Options for Code Generation Conventions], page 218.

```
-fcall-saved-reg -fcall-used-reg
-ffixed-reg -fexceptions
-fnon-call-exceptions -fdelete-dead-exceptions -funwind-tables
-fasynchronous-unwind-tables
-fno-gnu-unique
-finhibit-size-directive -fno-common -fno-ident
-fpcc-struct-return -fpic -fPIC -fpie -fPIE -fno-plt
-fno-jump-tables
-frecord-gcc-switches
-freg-struct-return -fshort-enums -fshort-wchar
-fverbose-asm -fpack-struct[=n]
-fleading-underscore -ftls-model=model
-fstack-reuse=reuse_level
-ftrampolines -ftrapv -fwrapv
-fvisibility=[default|internal|hidden|protected]
-fstrict-volatile-bitfields -fsync-libcalls
```

Developer Options

See Section 3.17 [GCC Developer Options], page 228.

```
-dletters -dumpspecs -dumpmachine -dumpversion
-dumpfullversion -fchecking -fchecking=n -fdbg-cnt-list
-fdbg-cnt=counter-value-list
-fdisable-ipa-pass_name
-fdisable-rtl-pass_name
-fdisable-rtl-pass-name=range-list
-fdisable-tree-pass_name
-fdisable-tree-pass-name=range-list
-fdump-debug -fdump-earlydebug
-fdump-noaddr -fdump-unnumbered -fdump-unnumbered-links
-fdump-final-insns[=file]
-fdump-ipa-all -fdump-ipa-cgraph -fdump-ipa-inline
-fdump-lang-all
```

-fdump-lang-*switch*
-fdump-lang-*switch-options*
-fdump-lang-*switch-options=filename*
-fdump-passes
-fdump-rtl-*pass* -fdump-rtl-*pass=filename*
-fdump-statistics
-fdump-tree-all
-fdump-tree-*switch*
-fdump-tree-*switch-options*
-fdump-tree-*switch-options=filename*
-fcompare-debug[=*opts*] -fcompare-debug-second
-fenable-*kind-pass*
-fenable-*kind-pass=range-list*
-fira-verbose=*n*
-flto-report -flto-report-wpa -fmem-report-wpa
-fmem-report -fpre-ipa-mem-report -fpost-ipa-mem-report
-fopt-info -fopt-info-*options*[=*file*]
-fprofile-report
-frandom-seed=*string* -fsched-verbose=*n*
-fsel-sched-verbose -fsel-sched-dump-cfg -fsel-sched-pipelining-verbose
-fstats -fstack-usage -ftime-report -ftime-report-details
-fvar-tracking-assignments-toggle -gtoggle
-print-file-name=*library* -print-libgcc-file-name
-print-multi-directory -print-multi-lib -print-multi-os-directory
-print-prog-name=*program* -print-search-dirs -Q
-print-sysroot -print-sysroot-headers-suffix
-save-temps -save-temps=cwd -save-temps=obj -time[=*file*]

Machine-Dependent Options

See Section 3.18 [Machine-Dependent Options], page 245.

AArch64 Options

-mabi=*name* -mbig-endian -mlittle-endian
-mgeneral-regs-only
-mcmodel=tiny -mcmodel=small -mcmodel=large
-mstrict-align -mno-strict-align
-momit-leaf-frame-pointer
-mtls-dialect=desc -mtls-dialect=traditional
-mtls-size=*size*
-mfix-cortex-a53-835769 -mfix-cortex-a53-843419
-mlow-precision-recip-sqrt -mlow-precision-sqrt -mlow-precision-div
-mpc-relative-literal-loads
-msign-return-address=*scope*
-mbranch-protection=*none*|*standard*|*pac-ret*[+*leaf*]|*bti*
-march=*name* -mcpu=*name* -mtune=*name*
-moverride=*string* -mverbose-cost-dump
-mstack-protector-guard=*guard* -mstack-protector-guard-reg=*sysreg*
-mstack-protector-guard-offset=*offset* -mtrack-speculation

Adapteva Epiphany Options

-mhalf-reg-file -mprefer-short-insn-regs
-mbranch-cost=*num* -mcmove -mnops=*num* -msoft-cmpsf
-msplit-lohi -mpost-inc -mpost-modify -mstack-offset=*num*
-mround-nearest -mlong-calls -mshort-calls -msmall16
-mfp-mode=*mode* -mvect-double -max-vect-align=*num*
-msplit-vecmove-early -m1reg-*reg*

AMD GCN Options

-march=*gpu* -mtune=*gpu* -mstack-size=*bytes*

ARC Options

-mbarrel-shifter -mjli-always
-mcpu=*cpu* -mA6 -mARC600 -mA7 -mARC700
-mdpfp -mdpfp-compact -mdpfp-fast -mno-dpfp-lrsr
-mea -mno-mpy -mmul32x16 -mmul64 -matomic
-mnorm -mspfp -mspfp-compact -mspfp-fast -msimd -msoft-float -mswap
-mcrc -mdsp-packa -mdvbf -mlock -mmac-d16 -mmac-24 -mrtsc -mswape
-mtelephony -mxy -misize -mannotate-align -marclinux -marclinux_prof
-mlong-calls -mmedium-calls -msdata -mirq-ctrl-saved
-mrgf-banked-regs -mlpc-width=*width* -G *num*
-mvolatile-cache -mtp-regno=*regno*
-malign-call -mauto-modify-reg -mbbit-peephole -mno-brcc
-mcase-vector-pcrel -mcompact-casesi -mno-cond-exec -mearly-cbranchsi
-mexpand-adddi -mindexed-loads -mlra -mlra-priority-none
-mlra-priority-compact mlra-priority-noncompact -mmillicode
-mmixed-code -mq-class -mRcq -mRcw -msize-level=*level*
-mtune=*cpu* -mmultcost=*num* -mcode-density-frame
-munalign-prob-threshold=*probability* -mmpy-option=*multo*
-mdiv-rem -mcode-density -mll64 -mfpu=*fpu* -mrf16 -mbranch-index

ARM Options

-mapcs-frame -mno-apcs-frame
-mabi=*name*
-mapcs-stack-check -mno-apcs-stack-check
-mapcs-reentrant -mno-apcs-reentrant
-mgeneral-regs-only
-msched-prolog -mno-sched-prolog
-mlittle-endian -mbig-endian
-mbe8 -mbe32
-mfloat-abi=*name*
-mfp16-format=*name* -mthumb-interwork -mno-thumb-interwork
-mcpu=*name* -march=*name* -mfpu=*name*
-mtune=*name* -mprint-tune-info
-mstructure-size-boundary=*n*
-mabort-on-noreturn
-mlong-calls -mno-long-calls
-msingle-pic-base -mno-single-pic-base
-mpic-register=*reg*
-mnop-fun-dllimport
-mpoke-function-name
-mthumb -marm -mflip-thumb
-mtpcs-frame -mtpcs-leaf-frame
-mcaller-super-interworking -mcallee-super-interworking
-mtp=*name* -mtls-dialect=*dialect*
-mword-relocations
-mfix-cortex-m3-ldrd
-munaligned-access
-mneon-for-64bits
-mslow-flash-data
-masm-syntax-unified
-mrestrict-it
-mverbose-cost-dump
-mpure-code
-mcmse

AVR Options

-mmcu=*mcu* -mabsdata -maccumulate-args
-mbranch-cost=*cost*
-mcall-prologues -mgas-isr-prologues -mint8

-mn_flash=*size* -mno-interrupts
-mmain-is-OS_task -mrelax -mrmw -mstrict-X -mtiny-stack
-mfract-convert-truncate
-mshort-calls -nodevicelib
-Waddr-space-convert -Wmisspelled-isr

Blackfin Options

-mcpu=*cpu*[-*sirevision*]
-msim -momit-leaf-frame-pointer -mno-omit-leaf-frame-pointer
-mspecld-anomaly -mno-specld-anomaly -mcsync-anomaly -mno-csync-anomaly
-mlow-64k -mno-low64k -mstack-check-l1 -mid-shared-library
-mno-id-shared-library -mshared-library-id=*n*
-mleaf-id-shared-library -mno-leaf-id-shared-library
-msep-data -mno-sep-data -mlong-calls -mno-long-calls
-mfast-fp -minline-plt -mmulticore -mcorea -mcoreb -msdram
-micplb

C6X Options

-mbig-endian -mlittle-endian -march=*cpu*
-msim -msdata=*sdata-type*

CRIS Options

-mcpu=*cpu* -march=*cpu* -mtune=*cpu*
-mmax-stack-frame=*n* -melinux-stacksize=*n*
-metrax4 -metrax100 -mpdebug -mcc-init -mno-side-effects
-mstack-align -mdata-align -mconst-align
-m32-bit -m16-bit -m8-bit -mno-prologue-epilogue -mno-gotplt
-melf -maout -melinux -mlinux -sim -sim2
-mmul-bug-workaround -mno-mul-bug-workaround

CR16 Options

-mmac
-mcr16cplus -mcr16c
-msim -mint32 -mbit-ops -mdata-model=*model*

C-SKY Options

-march=*arch* -mcpu=*cpu*
-mbig-endian -EB -mlittle-endian -EL
-mhard-float -msoft-float -mfpu=*fpu* -mdouble-float -mfdivdu
-melrw -mistack -mmp -mcp -mcache -msecurity -mtrust
-mdsp -medsp -mvdsp
-mdiv -msmart -mhigh-registers -manchor
-mpushpop -mmultiple-stld -mconstpool -mstack-size -mccrt
-mbranch-cost=*n* -mcse-cc -msched-prolog

Darwin Options

-all_load -allowable_client -arch -arch_errors_fatal
-arch_only -bind_at_load -bundle -bundle_loader
-client_name -compatibility_version -current_version
-dead_strip
-dependency-file -dylib_file -dylinker_install_name
-dynamic -dynamiclib -exported_symbols_list
-filelist -flat_namespace -force_cpusubtype_ALL
-force_flat_namespace -headerpad_max_install_names
-iframework
-image_base -init -install_name -keep_private_externs
-multi_module -multiply_defined -multiply_defined_unused
-noall_load -no_dead_strip_inits_and_terms
-nofixprebinding -nomultidefs -noprebind -noseglinkedit
-pagezero_size -prebind -prebind_all_twolevel_modules

-private_bundle -read_only_relocs -sectalign
-sectobjectsymbols -whyload -seg1addr
-sectcreate -sectobjectsymbols -sectorder
-segaddr -segs_read_only_addr -segs_read_write_addr
-seg_addr_table -seg_addr_table_filename -seglinkedit
-segprot -segs_read_only_addr -segs_read_write_addr
-single_module -static -sub_library -sub_umbrella
-twolevel_namespace -umbrella -undefined
-unexported_symbols_list -weak_reference_mismatches
-whatsloaded -F -gused -gfull -mmacosx-version-min=*version*
-mkernel -mone-byte-bool

DEC Alpha Options

-mno-fp-regs -msoft-float
-mieee -mieee-with-inexact -mieee-conformant
-mfp-trap-mode=*mode* -mfp-rounding-mode=*mode*
-mtrap-precision=*mode* -mbuild-constants
-mcpu=*cpu-type* -mtune=*cpu-type*
-mbwx -mmax -mfix -mcix
-mfloat-vax -mfloat-ieee
-mexplicit-relocs -msmall-data -mlarge-data
-msmall-text -mlarge-text
-mmemory-latency=*time*

FR30 Options

-msmall-model -mno-lsim

FT32 Options

-msim -mlra -mnodiv -mft32b -mcompress -mnopm

FRV Options

-mgpr-32 -mgpr-64 -mfpr-32 -mfpr-64
-mhard-float -msoft-float
-malloc-cc -mfixed-cc -mdword -mno-dword
-mdouble -mno-double
-mmedia -mno-media -mmuladd -mno-muladd
-mfdpic -minline-plt -mgprel-ro -multilib-library-pic
-mlinked-fp -mlong-calls -malign-labels
-mlibrary-pic -macc-4 -macc-8
-mpack -mno-pack -mno-eflags -mcond-move -mno-cond-move
-moptimize-membar -mno-optimize-membar
-mscc -mno-scc -mcond-exec -mno-cond-exec
-mvliw-branch -mno-vliw-branch
-mmulti-cond-exec -mno-multi-cond-exec -mnested-cond-exec
-mno-nested-cond-exec -mtomcat-stats
-mTLS -mtls
-mcpu=*cpu*

GNU/Linux Options

-mglibc -muclibc -mmusl -mbionic -mandroid
-tno-android-cc -tno-android-ld

H8/300 Options

-mrelax -mh -ms -mn -mexr -mno-exr -mint32 -malign-300

HPPA Options

-march=*architecture-type*
-mcaller-copies -mdisable-fpregs -mdisable-indexing
-mfast-indirect-calls -mgas -mgnu-ld -mhp-ld
-mfixed-range=*register-range*

-mjump-in-delay -mlinker-opt -mlong-calls
-mlong-load-store -mno-disable-fpregs
-mno-disable-indexing -mno-fast-indirect-calls -mno-gas
-mno-jump-in-delay -mno-long-load-store
-mno-portable-runtime -mno-soft-float
-mno-space-regs -msoft-float -mpa-risc-1-0
-mpa-risc-1-1 -mpa-risc-2-0 -mportable-runtime
-mschedule=*cpu-type* -mspace-regs -msio -mwsio
-munix=*unix-std* -nolibdld -static -threads

IA-64 Options

-mbig-endian -mlittle-endian -mgnu-as -mgnu-ld -mno-pic
-mvolatile-asm-stop -mregister-names -msdata -mno-sdata
-mconstant-gp -mauto-pic -mfused-madd
-minline-float-divide-min-latency
-minline-float-divide-max-throughput
-mno-inline-float-divide
-minline-int-divide-min-latency
-minline-int-divide-max-throughput
-mno-inline-int-divide
-minline-sqrt-min-latency -minline-sqrt-max-throughput
-mno-inline-sqrt
-mdwarf2-asm -mearly-stop-bits
-mfixed-range=*register-range* -mtls-size=*tls-size*
-mtune=*cpu-type* -milp32 -mlp64
-msched-br-data-spec -msched-ar-data-spec -msched-control-spec
-msched-br-in-data-spec -msched-ar-in-data-spec -msched-in-control-spec
-msched-spec-ldc -msched-spec-control-ldc
-msched-prefer-non-data-spec-insns -msched-prefer-non-control-spec-insns
-msched-stop-bits-after-every-cycle -msched-count-spec-in-critical-path
-msel-sched-dont-check-control-spec -msched-fp-mem-deps-zero-cost
-msched-max-memory-insns-hard-limit -msched-max-memory-insns=*max-insns*

LM32 Options

-mbarrel-shift-enabled -mdivide-enabled -mmultiply-enabled
-msign-extend-enabled -muser-enabled

M32R/D Options

-m32r2 -m32rx -m32r
-mdebug
-malign-loops -mno-align-loops
-missue-rate=*number*
-mbranch-cost=*number*
-mmodel=*code-size-model-type*
-msdata=*sdata-type*
-mno-flush-func -mflush-func=*name*
-mno-flush-trap -mflush-trap=*number*
-G *num*

M32C Options

-mcpu=*cpu* -msim -memregs=*number*

M680x0 Options

-march=*arch* -mcpu=*cpu* -mtune=*tune*
-m68000 -m68020 -m68020-40 -m68020-60 -m68030 -m68040
-m68060 -mcpu32 -m5200 -m5206e -m528x -m5307 -m5407
-mcfv4e -mbitfield -mno-bitfield -mc68000 -mc68020
-mnobitfield -mrtd -mno-rtd -mdiv -mno-div -mshort
-mno-short -mhard-float -m68881 -msoft-float -mpcrel
-malign-int -mstrict-align -msep-data -mno-sep-data

-mshared-library-id=n -mid-shared-library -mno-id-shared-library
-mxgot -mno-xgot -mlong-jump-table-offsets

MCore Options

-mhardlit -mno-hardlit -mdiv -mno-div -mrelax-immediates
-mno-relax-immediates -mwide-bitfields -mno-wide-bitfields
-m4byte-functions -mno-4byte-functions -mcallgraph-data
-mno-callgraph-data -mslow-bytes -mno-slow-bytes -mno-lsim
-mlittle-endian -mbig-endian -m210 -m340 -mstack-increment

MeP Options

-mabsdiff -mall-opts -maverage -mbased=*n* -mbitops
-mc=*n* -mclip -mconfig=*name* -mcop -mcop32 -mcop64 -mivc2
-mdc -mdiv -meb -mel -mio-volatile -ml -mleadz -mm -mminmax
-mmult -mno-opts -mrepeat -ms -msatur -msdram -msim -msimnovec -mtf
-mtiny=*n*

MicroBlaze Options

-msoft-float -mhard-float -msmall-divides -mcpu=*cpu*
-mmemcpy -mxl-soft-mul -mxl-soft-div -mxl-barrel-shift
-mxl-pattern-compare -mxl-stack-check -mxl-gp-opt -mno-clearbss
-mxl-multiply-high -mxl-float-convert -mxl-float-sqrt
-mbig-endian -mlittle-endian -mxl-reorder -mxl-mode-*app-model*
-mpic-data-is-text-relative

MIPS Options

-EL -EB -march=*arch* -mtune=*arch*
-mips1 -mips2 -mips3 -mips4 -mips32 -mips32r2 -mips32r3 -mips32r5
-mips32r6 -mips64 -mips64r2 -mips64r3 -mips64r5 -mips64r6
-mips16 -mno-mips16 -mflip-mips16
-minterlink-compressed -mno-interlink-compressed
-minterlink-mips16 -mno-interlink-mips16
-mabi=*abi* -mabicalls -mno-abicalls
-mshared -mno-shared -mplt -mno-plt -mxgot -mno-xgot
-mgp32 -mgp64 -mfp32 -mfpxx -mfp64 -mhard-float -msoft-float
-mno-float -msingle-float -mdouble-float
-modd-spreg -mno-odd-spreg
-mabs=*mode* -mnan=*encoding*
-mdsp -mno-dsp -mdspr2 -mno-dspr2
-mmcu -mmno-mcu
-meva -mno-eva
-mvirt -mno-virt
-mxpa -mno-xpa
-mcrc -mno-crc
-mginv -mno-ginv
-mmicromips -mno-micromips
-mmsa -mno-msa
-mloongson-mmi -mno-loongson-mmi
-mloongson-ext -mno-loongson-ext
-mloongson-ext2 -mno-loongson-ext2
-mfpu=*fpu-type*
-msmartmips -mno-smartmips
-mpaired-single -mno-paired-single -mdmx -mno-mdmx
-mips3d -mno-mips3d -mmt -mno-mt -mllsc -mno-llsc
-mlong64 -mlong32 -msym32 -mno-sym32
-G*num* -mlocal-sdata -mno-local-sdata
-mextern-sdata -mno-extern-sdata -mgpopt -mno-gopt
-membedded-data -mno-embedded-data
-muninit-const-in-rodata -mno-uninit-const-in-rodata
-mcode-readable=*setting*

-msplit-addresses -mno-split-addresses
-mexplicit-relocs -mno-explicit-relocs
-mcheck-zero-division -mno-check-zero-division
-mdivide-traps -mdivide-breaks
-mload-store-pairs -mno-load-store-pairs
-mmemcpy -mno-memcpy -mlong-calls -mno-long-calls
-mmad -mno-mad -mimadd -mno-imadd -mfused-madd -mno-fused-madd -nocpp
-mfix-24k -mno-fix-24k
-mfix-r4000 -mno-fix-r4000 -mfix-r4400 -mno-fix-r4400
-mfix-r5900 -mno-fix-r5900
-mfix-r10000 -mno-fix-r10000 -mfix-rm7000 -mno-fix-rm7000
-mfix-vr4120 -mno-fix-vr4120
-mfix-vr4130 -mno-fix-vr4130 -mfix-sb1 -mno-fix-sb1
-mflush-func=*func* -mno-flush-func
-mbranch-cost=*num* -mbranch-likely -mno-branch-likely
-mcompact-branches=*policy*
-mfp-exceptions -mno-fp-exceptions
-mvr4130-align -mno-vr4130-align -msynci -mno-synci
-mlxc1-sxc1 -mno-lxc1-sxc1 -mmadd4 -mno-madd4
-mrelax-pic-calls -mno-relax-pic-calls -mmcount-ra-address
-mframe-header-opt -mno-frame-header-opt

MMIX Options

-mlibfuncs -mno-libfuncs -mepsilon -mno-epsilon -mabi=gnu
-mabi=mmixware -mzero-extend -mknuthdiv -mtoplevel-symbols
-melf -mbranch-predict -mno-branch-predict -mbase-addresses
-mno-base-addresses -msingle-exit -mno-single-exit

MN10300 Options

-mmult-bug -mno-mult-bug
-mno-am33 -mam33 -mam33-2 -mam34
-mtune=*cpu-type*
-mreturn-pointer-on-d0
-mno-crt0 -mrelax -mliw -msetlb

Moxie Options

-meb -mel -mmul.x -mno-crt0

MSP430 Options

-msim -masm-hex -mmcu= -mcpu= -mlarge -msmall -mrelax
-mwarn-mcu
-mcode-region= -mdata-region=
-msilicon-errata= -msilicon-errata-warn=
-mhwmult= -minrt

NDS32 Options

-mbig-endian -mlittle-endian
-mreduced-regs -mfull-regs
-mcmov -mno-cmov
-mext-perf -mno-ext-perf
-mext-perf2 -mno-ext-perf2
-mext-string -mno-ext-string
-mv3push -mno-v3push
-m16bit -mno-16bit
-misr-vector-size=*num*
-mcache-block-size=*num*
-march=*arch*
-mcmodel=*code-model*
-mctor-dtor -mrelax

Nios II Options

-G *num* -mgpopt=*option* -mgpopt -mno-gpopt
-mgprel-sec=*regexp* -mr0rel-sec=*regexp*
-mel -meb
-mno-bypass-cache -mbypass-cache
-mno-cache-volatile -mcache-volatile
-mno-fast-sw-div -mfast-sw-div
-mhw-mul -mno-hw-mul -mhw-mulx -mno-hw-mulx -mno-hw-div -mhw-div
-mcustom-*insn*=*N* -mno-custom-*insn*
-mcustom-fpu-cfg=*name*
-mhal -msmallc -msys-crt0=*name* -msys-lib=*name*
-march=*arch* -mbmx -mno-bmx -mcdx -mno-cdx

Nvidia PTX Options

-m32 -m64 -mmainkernel -moptimize

OpenRISC Options

-mboard=*name* -mnewlib -mhard-mul -mhard-div
-msoft-mul -msoft-div
-mcmov -mror -msext -msfimm -mshftimm

PDP-11 Options

-mfpu -msoft-float -mac0 -mno-ac0 -m40 -m45 -m10
-mint32 -mno-int16 -mint16 -mno-int32
-msplit -munix-asm -mdec-asm -mgnu-asm -mlra

picoChip Options

-mae=*ae_type* -mvliw-lookahead=*N*
-msymbol-as-address -mno-inefficient-warnings

PowerPC Options See RS/6000 and PowerPC Options.

RISC-V Options

-mbranch-cost=*N-instruction*
-mplt -mno-plt
-mabi=*ABI-string*
-mfdiv -mno-fdiv
-mdiv -mno-div
-march=*ISA-string*
-mtune=*processor-string*
-mpreferred-stack-boundary=*num*
-msmall-data-limit=*N-bytes*
-msave-restore -mno-save-restore
-mstrict-align -mno-strict-align
-mcmodel=medlow -mcmodel=medany
-mexplicit-relocs -mno-explicit-relocs
-mrelax -mno-relax
-mriscv-attribute -mmo-riscv-attribute

RL78 Options

-msim -mmul=none -mmul=g13 -mmul=g14 -mallregs
-mcpu=g10 -mcpu=g13 -mcpu=g14 -mg10 -mg13 -mg14
-m64bit-doubles -m32bit-doubles -msave-mduc-in-interrupts

RS/6000 and PowerPC Options

-mcpu=*cpu-type*
-mtune=*cpu-type*
-mcmodel=*code-model*
-mpowerpc64
-maltivec -mno-altivec
-mpowerpc-gpopt -mno-powerpc-gpopt

-mpowerpc-gfxopt -mno-powerpc-gfxopt
-mmfcrf -mno-mfcrf -mpopcntb -mno-popcntb -mpopcntd -mno-popcntd
-mfprnd -mno-fprnd
-mcmpb -mno-cmpb -mmfpgpr -mno-mfpgpr -mhard-dfp -mno-hard-dfp
-mfull-toc -mminimal-toc -mno-fp-in-toc -mno-sum-in-toc
-m64 -m32 -mxl-compat -mno-xl-compat -mpe
-malign-power -malign-natural
-msoft-float -mhard-float -mmultiple -mno-multiple
-mupdate -mno-update
-mavoid-indexed-addresses -mno-avoid-indexed-addresses
-mfused-madd -mno-fused-madd -mbit-align -mno-bit-align
-mstrict-align -mno-strict-align -mrelocatable
-mno-relocatable -mrelocatable-lib -mno-relocatable-lib
-mtoc -mno-toc -mlittle -mlittle-endian -mbig -mbig-endian
-mdynamic-no-pic -mswdiv -msingle-pic-base
-mprioritize-restricted-insns=*priority*
-msched-costly-dep=*dependence_type*
-minsert-sched-nops=*scheme*
-mcall-aixdesc -mcall-eabi -mcall-freebsd
-mcall-linux -mcall-netbsd -mcall-openbsd
-mcall-sysv -mcall-sysv-eabi -mcall-sysv-noeabi
-mtraceback=*traceback_type*
-maix-struct-return -msvr4-struct-return
-mabi=*abi-type* -msecure-plt -mbss-plt
-mlongcall -mno-longcall -mpltseq -mno-pltseq
-mblock-move-inline-limit=*num*
-mblock-compare-inline-limit=*num*
-mblock-compare-inline-loop-limit=*num*
-mstring-compare-inline-limit=*num*
-misel -mno-isel
-mvrsave -mno-vrsave
-mmulhw -mno-mulhw
-mdlmzb -mno-dlmzb
-mprototype -mno-prototype
-msim -mmvme -mads -myellowknife -memb -msdata
-msdata=*opt* -mreadonly-in-sdata -mvxworks -G *num*
-mrecip -mrecip=*opt* -mno-recip -mrecip-precision
-mno-recip-precision
-mveclibabi=*type* -mfriz -mno-friz
-mpointers-to-nested-functions -mno-pointers-to-nested-functions
-msave-toc-indirect -mno-save-toc-indirect
-mpower8-fusion -mno-mpower8-fusion -mpower8-vector -mno-power8-vector
-mcrypto -mno-crypto -mhtm -mno-htm
-mquad-memory -mno-quad-memory
-mquad-memory-atomic -mno-quad-memory-atomic
-mcompat-align-parm -mno-compat-align-parm
-mfloat128 -mno-float128 -mfloat128-hardware -mno-float128-hardware
-mgnu-attribute -mno-gnu-attribute
-mstack-protector-guard=*guard* -mstack-protector-guard-reg=*reg*
-mstack-protector-guard-offset=*offset*

RX Options

-m64bit-doubles -m32bit-doubles -fpu -nofpu
-mcpu=
-mbig-endian-data -mlittle-endian-data
-msmall-data
-msim -mno-sim
-mas100-syntax -mno-as100-syntax
-mrelax

-mmax-constant-size=
-mint-register=
-mpid
-mallow-string-insns -mno-allow-string-insns
-mjsr
-mno-warn-multiple-fast-interrupts
-msave-acc-in-interrupts

S/390 and zSeries Options

-mtune=*cpu-type* -march=*cpu-type*
-mhard-float -msoft-float -mhard-dfp -mno-hard-dfp
-mlong-double-64 -mlong-double-128
-mbackchain -mno-backchain -mpacked-stack -mno-packed-stack
-msmall-exec -mno-small-exec -mmvcle -mno-mvcle
-m64 -m31 -mdebug -mno-debug -mesa -mzarch
-mhtm -mvx -mzvector
-mtpf-trace -mno-tpf-trace -mfused-madd -mno-fused-madd
-mwarn-framesize -mwarn-dynamicstack -mstack-size -mstack-guard
-mhotpatch=*halfwords*,*halfwords*

Score Options

-meb -mel
-mnhwloop
-muls
-mmac
-mscore5 -mscore5u -mscore7 -mscore7d

SH Options

-m1 -m2 -m2e
-m2a-nofpu -m2a-single-only -m2a-single -m2a
-m3 -m3e
-m4-nofpu -m4-single-only -m4-single -m4
-m4a-nofpu -m4a-single-only -m4a-single -m4a -m4al
-mb -ml -mdalign -mrelax
-mbigtable -mfmovd -mrenesas -mno-renesas -mnomacsave
-mieee -mno-ieee -mbitops -misize -minline-ic_invalidate -mpadstruct
-mprefergot -musermode -multcost=*number* -mdiv=*strategy*
-mdivsi3_libfunc=*name* -mfixed-range=*register-range*
-maccumulate-outgoing-args
-matomic-model=*atomic-model*
-mbranch-cost=*num* -mzdcbranch -mno-zdcbranch
-mcbranch-force-delay-slot
-mfused-madd -mno-fused-madd -mfsca -mno-fsca -mfsrra -mno-fsrra
-mpretend-cmove -mtas

Solaris 2 Options

-mclear-hwcap -mno-clear-hwcap -mimpure-text -mno-impure-text
-pthreads

SPARC Options

-mcpu=*cpu-type*
-mtune=*cpu-type*
-mcmodel=*code-model*
-mmemory-model=*mem-model*
-m32 -m64 -mapp-regs -mno-app-regs
-mfaster-structs -mno-faster-structs -mflat -mno-flat
-mfpu -mno-fpu -mhard-float -msoft-float
-mhard-quad-float -msoft-quad-float
-mstack-bias -mno-stack-bias
-mstd-struct-return -mno-std-struct-return

-munaligned-doubles -mno-unaligned-doubles
-muser-mode -mno-user-mode
-mv8plus -mno-v8plus -mvis -mno-vis
-mvis2 -mno-vis2 -mvis3 -mno-vis3
-mvis4 -mno-vis4 -mvis4b -mno-vis4b
-mcbcond -mno-cbcond -mfmaf -mno-fmaf -mfsmuld -mno-fsmuld
-mpopc -mno-popc -msubxc -mno-subxc
-mfix-at697f -mfix-ut699 -mfix-ut700 -mfix-gr712rc
-mlra -mno-lra

SPU Options

-mwarn-reloc -merror-reloc
-msafe-dma -munsafe-dma
-mbranch-hints
-msmall-mem -mlarge-mem -mstdmain
-mfixed-range=*register-range*
-mea32 -mea64
-maddress-space-conversion -mno-address-space-conversion
-mcache-size=*cache-size*
-matomic-updates -mno-atomic-updates

System V Options

-Qy -Qn -YP,*paths* -Ym,*dir*

TILE-Gx Options

-mcpu=CPU -m32 -m64 -mbig-endian -mlittle-endian
-mcmodel=*code-model*

TILEPro Options

-mcpu=*cpu* -m32

V850 Options

-mlong-calls -mno-long-calls -mep -mno-ep
-mprolog-function -mno-prolog-function -mspace
-mtda=*n* -msda=*n* -mzda=*n*
-mapp-regs -mno-app-regs
-mdisable-callt -mno-disable-callt
-mv850e2v3 -mv850e2 -mv850e1 -mv850es
-mv850e -mv850 -mv850e3v5
-mloop
-mrelax
-mlong-jumps
-msoft-float
-mhard-float
-mgcc-abi
-mrh850-abi
-mbig-switch

VAX Options

-mg -mgnu -munix

Visium Options

-mdebug -msim -mfpu -mno-fpu -mhard-float -msoft-float
-mcpu=*cpu-type* -mtune=*cpu-type* -msv-mode -muser-mode

VMS Options

-mvms-return-codes -mdebug-main=*prefix* -mmalloc64
-mpointer-size=*size*

VxWorks Options

-mrtp -non-static -Bstatic -Bdynamic
-Xbind-lazy -Xbind-now

x86 Options

-mtune=*cpu-type* -march=*cpu-type*
-mtune-ctrl=*feature-list* -mdump-tune-features -mno-default
-mfpmath=*unit*
-masm=*dialect* -mno-fancy-math-387
-mno-fp-ret-in-387 -m80387 -mhard-float -msoft-float
-mno-wide-multiply -mrtd -malign-double
-mpreferred-stack-boundary=*num*
-mincoming-stack-boundary=*num*
-mcld -mcx16 -msahf -mmovbe -mcrc32
-mrecip -mrecip=*opt*
-mvzeroupper -mprefer-avx128 -mprefer-vector-width=*opt*
-mmmx -msse -msse2 -msse3 -mssse3 -msse4.1 -msse4.2 -msse4 -mavx
-mavx2 -mavx512f -mavx512pf -mavx512er -mavx512cd -mavx512vl
-mavx512bw -mavx512dq -mavx512ifma -mavx512vbmi -msha -maes
-mpclmul -mfsgsbase -mrdrnd -mf16c -mfma -mpconfig -mwbnoinvd
-mptwrite -mprefetchwt1 -mclflushopt -mclwb -mxsavec -mxsaves
-msse4a -m3dnow -m3dnowa -mpopcnt -mabm -mbmi -mtbm -mfma4 -mxop
-madx -mlzcnt -mbmi2 -mfxsr -mxsave -mxsaveopt -mrtm -mhle -mlwp
-mmwaitx -mclzero -mpku -mthreads -mgfni -mvaes -mwaitpkg
-mshstk -mmanual-endbr -mforce-indirect-call -mavx512vbmi2
-mvpclmulqdq -mavx512bitalg -mmovdiri -mmovdir64b -mavx512vpopcntdq
-mavx5124fmaps -mavx512vnni -mavx5124vnniw -mprfchw -mrdpid
-mrdseed -msgx
-mcldemote -mms-bitfields -mno-align-stringops -minline-all-stringops
-minline-stringops-dynamically -mstringop-strategy=*alg*
-mmemcpy-strategy=*strategy* -mmemset-strategy=*strategy*
-mpush-args -maccumulate-outgoing-args -m128bit-long-double
-m96bit-long-double -mlong-double-64 -mlong-double-80 -mlong-double-128
-mregparm=*num* -msseregparm
-mveclibabi=*type* -mvect8-ret-in-mem
-mpc32 -mpc64 -mpc80 -mstackrealign
-momit-leaf-frame-pointer -mno-red-zone -mno-tls-direct-seg-refs
-mcmodel=*code-model* -mabi=*name* -maddress-mode=*mode*
-m32 -m64 -mx32 -m16 -miamcu -mlarge-data-threshold=*num*
-msse2avx -mfentry -mrecord-mcount -mnop-mcount -m8bit-idiv
-minstrument-return=*type* -mfentry-name=*name* -mfentry-section=*name*
-mavx256-split-unaligned-load -mavx256-split-unaligned-store
-malign-data=*type* -mstack-protector-guard=*guard*
-mstack-protector-guard-reg=*reg*
-mstack-protector-guard-offset=*offset*
-mstack-protector-guard-symbol=*symbol*
-mgeneral-regs-only -mcall-ms2sysv-xlogues
-mindirect-branch=*choice* -mfunction-return=*choice*
-mindirect-branch-register

x86 Windows Options

-mconsole -mcygwin -mno-cygwin -mdll
-mnop-fun-dllimport -mthread
-municode -mwin32 -mwindows -fno-set-stack-executable

Xstormy16 Options

-msim

Xtensa Options

-mconst16 -mno-const16
-mfused-madd -mno-fused-madd

```
-mforce-no-pic
-mserialize-volatile -mno-serialize-volatile
-mtext-section-literals -mno-text-section-literals
-mauto-litpools -mno-auto-litpools
-mtarget-align -mno-target-align
-mlongcalls -mno-longcalls
```

zSeries Options See S/390 and zSeries Options.

3.2 Options Controlling the Kind of Output

Compilation can involve up to four stages: preprocessing, compilation proper, assembly and linking, always in that order. GCC is capable of preprocessing and compiling several files either into several assembler input files, or into one assembler input file; then each assembler input file produces an object file, and linking combines all the object files (those newly compiled, and those specified as input) into an executable file.

For any given input file, the file name suffix determines what kind of compilation is done:

file.c C source code that must be preprocessed.

file.i C source code that should not be preprocessed.

file.ii C++ source code that should not be preprocessed.

file.m Objective-C source code. Note that you must link with the '`libobjc`' library to make an Objective-C program work.

file.mi Objective-C source code that should not be preprocessed.

file.mm
file.M Objective-C++ source code. Note that you must link with the '`libobjc`' library to make an Objective-C++ program work. Note that '`.M`' refers to a literal capital M.

file.mii Objective-C++ source code that should not be preprocessed.

file.h C, C++, Objective-C or Objective-C++ header file to be turned into a precompiled header (default), or C, C++ header file to be turned into an Ada spec (via the '`-fdump-ada-spec`' switch).

file.cc
file.cp
file.cxx
file.cpp
file.CPP
file.c++
file.C C++ source code that must be preprocessed. Note that in '`.cxx`', the last two letters must both be literally '`x`'. Likewise, '`.C`' refers to a literal capital C.

file.mm
file.M Objective-C++ source code that must be preprocessed.

file.mii Objective-C++ source code that should not be preprocessed.

file.hh
file.H
file.hp
file.hxx
file.hpp
file.HPP
file.h++
file.tcc C++ header file to be turned into a precompiled header or Ada spec.

file.f
file.for
file.ftn Fixed form Fortran source code that should not be preprocessed.

file.F
file.FOR
file.fpp
file.FPP
file.FTN Fixed form Fortran source code that must be preprocessed (with the traditional preprocessor).

file.f90
file.f95
file.f03
file.f08 Free form Fortran source code that should not be preprocessed.

file.F90
file.F95
file.F03
file.F08 Free form Fortran source code that must be preprocessed (with the traditional preprocessor).

file.go Go source code.

file.brig
BRIG files (binary representation of HSAIL).

file.d D source code.

file.di D interface file.

file.dd D documentation code (Ddoc).

file.ads Ada source code file that contains a library unit declaration (a declaration of a package, subprogram, or generic, or a generic instantiation), or a library unit renaming declaration (a package, generic, or subprogram renaming declaration). Such files are also called *specs*.

file.adb Ada source code file containing a library unit body (a subprogram or package body). Such files are also called *bodies*.

file.s Assembler code.

file.S
file.sx Assembler code that must be preprocessed.

other An object file to be fed straight into linking. Any file name with no recognized suffix is treated this way.

You can specify the input language explicitly with the '-x' option:

-x *language*
Specify explicitly the *language* for the following input files (rather than letting the compiler choose a default based on the file name suffix). This option applies to all following input files until the next '-x' option. Possible values for *language* are:

```
c  c-header  cpp-output
c++  c++-header  c++-cpp-output
objective-c  objective-c-header  objective-c-cpp-output
objective-c++ objective-c++-header objective-c++-cpp-output
assembler  assembler-with-cpp
ada
d
f77  f77-cpp-input f95  f95-cpp-input
go
brig
```

-x none Turn off any specification of a language, so that subsequent files are handled according to their file name suffixes (as they are if '-x' has not been used at all).

If you only want some of the stages of compilation, you can use '-x' (or filename suffixes) to tell gcc where to start, and one of the options '-c', '-S', or '-E' to say where gcc is to stop. Note that some combinations (for example, '-x cpp-output -E') instruct gcc to do nothing at all.

-c Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

By default, the object file name for a source file is made by replacing the suffix '.c', '.i', '.s', etc., with '.o'.

Unrecognized input files, not requiring compilation or assembly, are ignored.

-S Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix '.c', '.i', etc., with '.s'.

Input files that don't require compilation are ignored.

-E Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

Input files that don't require preprocessing are ignored.

-o *file* Place output in file *file*. This applies to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

If '-o' is not specified, the default is to put an executable file in 'a.out', the object file for '*source.suffix*' in '*source*.o', its assembler file in '*source*.s', a

precompiled header file in '*source.suffix*.gch', and all preprocessed C source on standard output.

-v Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-### Like '-v' except the commands are not executed and arguments are quoted unless they contain only alphanumeric characters or ./-_. This is useful for shell scripts to capture the driver-generated command lines.

--help Print (on the standard output) a description of the command-line options understood by gcc. If the '-v' option is also specified then '--help' is also passed on to the various processes invoked by gcc, so that they can display the command-line options they accept. If the '-Wextra' option has also been specified (prior to the '--help' option), then command-line options that have no documentation associated with them are also displayed.

--target-help
Print (on the standard output) a description of target-specific command-line options for each tool. For some targets extra target-specific information may also be printed.

--help={*class*|[^]*qualifier*}[,...]
Print (on the standard output) a description of the command-line options understood by the compiler that fit into all specified classes and qualifiers. These are the supported classes:

'optimizers'
Display all of the optimization options supported by the compiler.

'warnings'
Display all of the options controlling warning messages produced by the compiler.

'target' Display target-specific options. Unlike the '--target-help' option however, target-specific options of the linker and assembler are not displayed. This is because those tools do not currently support the extended '--help=' syntax.

'params' Display the values recognized by the '--param' option.

language Display the options supported for *language*, where *language* is the name of one of the languages supported in this version of GCC.

'common' Display the options that are common to all languages.

These are the supported qualifiers:

'undocumented'
Display only those options that are undocumented.

'joined' Display options taking an argument that appears after an equal sign in the same continuous piece of text, such as: '--help=target'.

'separate'
: Display options taking an argument that appears as a separate word following the original option, such as: '-o output-file'.

Thus for example to display all the undocumented target-specific switches supported by the compiler, use:

```
--help=target,undocumented
```

The sense of a qualifier can be inverted by prefixing it with the '^' character, so for example to display all binary warning options (i.e., ones that are either on or off and that do not take an argument) that have a description, use:

```
--help=warnings,^joined,^undocumented
```

The argument to '--help=' should not consist solely of inverted qualifiers.

Combining several classes is possible, although this usually restricts the output so much that there is nothing to display. One case where it does work, however, is when one of the classes is *target*. For example, to display all the target-specific optimization options, use:

```
--help=target,optimizers
```

The '--help=' option can be repeated on the command line. Each successive use displays its requested class of options, skipping those that have already been displayed. If '--help' is also specified anywhere on the command line then this takes precedence over any '--help=' option.

If the '-Q' option appears on the command line before the '--help=' option, then the descriptive text displayed by '--help=' is changed. Instead of describing the displayed options, an indication is given as to whether the option is enabled, disabled or set to a specific value (assuming that the compiler knows this at the point where the '--help=' option is used).

Here is a truncated example from the ARM port of gcc:

```
% gcc -Q -mabi=2 --help=target -c
The following options are target specific:
-mabi=                                2
-mabort-on-noreturn                   [disabled]
-mapcs                                [disabled]
```

The output is sensitive to the effects of previous command-line options, so for example it is possible to find out which optimizations are enabled at '-O2' by using:

```
-Q -O2 --help=optimizers
```

Alternatively you can discover which binary optimizations are enabled by '-O3' by using:

```
gcc -c -Q -O3 --help=optimizers > /tmp/O3-opts
gcc -c -Q -O2 --help=optimizers > /tmp/O2-opts
diff /tmp/O2-opts /tmp/O3-opts | grep enabled
```

--version
: Display the version number and copyrights of the invoked GCC.

-pass-exit-codes
: Normally the gcc program exits with the code of 1 if any phase of the compiler returns a non-success return code. If you specify '-pass-exit-codes', the gcc

program instead returns with the numerically highest error produced by any phase returning an error indication. The C, C++, and Fortran front ends return 4 if an internal compiler error is encountered.

`-pipe` Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

`-specs=file`
Process *file* after the compiler reads in the standard '`specs`' file, in order to override the defaults which the `gcc` driver program uses when determining what switches to pass to `cc1`, `cc1plus`, `as`, `ld`, etc. More than one '`-specs=file`' can be specified on the command line, and they are processed in order, from left to right. See Section 3.19 [Spec Files], page 428, for information about the format of the *file*.

`-wrapper` Invoke all subcommands under a wrapper program. The name of the wrapper program and its parameters are passed as a comma separated list.

```
gcc -c t.c -wrapper gdb,--args
```

This invokes all subprograms of `gcc` under '`gdb --args`', thus the invocation of `cc1` is '`gdb --args cc1 ...`'.

`-ffile-prefix-map=old=new`
When compiling files residing in directory '*old*', record any references to them in the result of the compilation as if the files resided in directory '*new*' instead. Specifying this option is equivalent to specifying all the individual '`-f*-prefix-map`' options. This can be used to make reproducible builds that are location independent. See also '`-fmacro-prefix-map`' and '`-fdebug-prefix-map`'.

`-fplugin=name.so`
Load the plugin code in file *name*.so, assumed to be a shared object to be dlopen'd by the compiler. The base name of the shared object file is used to identify the plugin for the purposes of argument parsing (See '`-fplugin-arg-name-key=value`' below). Each plugin should define the callback functions specified in the Plugins API.

`-fplugin-arg-name-key=value`
Define an argument called *key* with a value of *value* for the plugin called *name*.

`-fdump-ada-spec[-slim]`
For C and C++ source and include files, generate corresponding Ada specs. See Section "Generating Ada Bindings for C and C++ headers" in *GNAT User's Guide*, which provides detailed documentation on this feature.

`-fada-spec-parent=unit`
In conjunction with '`-fdump-ada-spec[-slim]`' above, generate Ada specs as child units of parent *unit*.

`-fdump-go-spec=file`
For input files in any language, generate corresponding Go declarations in *file*. This generates Go `const`, `type`, `var`, and `func` declarations which may be

a useful way to start writing a Go interface to code written in some other language.

`@file` Read command-line options from *file*. The options read are inserted in place of the original @*file* option. If *file* does not exist, or cannot be read, then the option will be treated literally, and not removed.

Options in *file* are separated by whitespace. A whitespace character may be included in an option by surrounding the entire option in either single or double quotes. Any character (including a backslash) may be included by prefixing the character to be included with a backslash. The *file* may itself contain additional @*file* options; any such options will be processed recursively.

3.3 Compiling C++ Programs

C++ source files conventionally use one of the suffixes '`.C`', '`.cc`', '`.cpp`', '`.CPP`', '`.c++`', '`.cp`', or '`.cxx`'; C++ header files often use '`.hh`', '`.hpp`', '`.H`', or (for shared template code) '`.tcc`'; and preprocessed C++ files use the suffix '`.ii`'. GCC recognizes files with these names and compiles them as C++ programs even if you call the compiler the same way as for compiling C programs (usually with the name `gcc`).

However, the use of `gcc` does not add the C++ library. `g++` is a program that calls GCC and automatically specifies linking against the C++ library. It treats '`.c`', '`.h`' and '`.i`' files as C++ source files instead of C source files unless '`-x`' is used. This program is also useful when precompiling a C header file with a '`.h`' extension for use in C++ compilations. On many systems, `g++` is also installed with the name `c++`.

When you compile C++ programs, you may specify many of the same command-line options that you use for compiling programs in any language; or command-line options meaningful for C and related languages; or options that are meaningful only for C++ programs. See Section 3.4 [Options Controlling C Dialect], page 37, for explanations of options for languages related to C. See Section 3.5 [Options Controlling C++ Dialect], page 44, for explanations of options that are meaningful only for C++ programs.

3.4 Options Controlling C Dialect

The following options control the dialect of C (or languages derived from C, such as C++, Objective-C and Objective-C++) that the compiler accepts:

`-ansi` In C mode, this is equivalent to '`-std=c90`'. In C++ mode, it is equivalent to '`-std=c++98`'.

This turns off certain features of GCC that are incompatible with ISO C90 (when compiling C code), or of standard C++ (when compiling C++ code), such as the `asm` and `typeof` keywords, and predefined macros such as `unix` and `vax` that identify the type of system you are using. It also enables the undesirable and rarely used ISO trigraph feature. For the C compiler, it disables recognition of C++ style '`//`' comments as well as the `inline` keyword.

The alternate keywords `__asm__`, `__extension__`, `__inline__` and `__typeof__` continue to work despite '`-ansi`'. You would not want to use them in an ISO C program, of course, but it is useful to put them in header files that might be

included in compilations done with '-ansi'. Alternate predefined macros such as __unix__ and __vax__ are also available, with or without '-ansi'.

The '-ansi' option does not cause non-ISO programs to be rejected gratuitously. For that, '-Wpedantic' is required in addition to '-ansi'. See Section 3.8 [Warning Options], page 70.

The macro __STRICT_ANSI__ is predefined when the '-ansi' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ISO standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

Functions that are normally built in but do not have semantics defined by ISO C (such as alloca and ffs) are not built-in functions when '-ansi' is used. See Section 6.59 [Other built-in functions provided by GCC], page 641, for details of the functions affected.

-std= Determine the language standard. See Chapter 2 [Language Standards Supported by GCC], page 5, for details of these standard versions. This option is currently only supported when compiling C or C++.

The compiler can accept several base standards, such as 'c90' or 'c++98', and GNU dialects of those standards, such as 'gnu90' or 'gnu++98'. When a base standard is specified, the compiler accepts all programs following that standard plus those using GNU extensions that do not contradict it. For example, '-std=c90' turns off certain features of GCC that are incompatible with ISO C90, such as the asm and typeof keywords, but not other GNU extensions that do not have a meaning in ISO C90, such as omitting the middle term of a ?: expression. On the other hand, when a GNU dialect of a standard is specified, all features supported by the compiler are enabled, even when those features change the meaning of the base standard. As a result, some strict-conforming programs may be rejected. The particular standard is used by '-Wpedantic' to identify which features are GNU extensions given that version of the standard. For example '-std=gnu90 -Wpedantic' warns about C++ style '//' comments, while '-std=gnu99 -Wpedantic' does not.

A value for this option must be provided; possible values are

'c90'
'c89'
'iso9899:1990'
: Support all ISO C90 programs (certain GNU extensions that conflict with ISO C90 are disabled). Same as '-ansi' for C code.

'iso9899:199409'
: ISO C90 as modified in amendment 1.

'c99'
'c9x'
'iso9899:1999'
'iso9899:199x'
: ISO C99. This standard is substantially completely supported, modulo bugs and floating-point issues (mainly but not entirely

relating to optional C99 features from Annexes F and G). See `http://gcc.gnu.org/c99status.html` for more information. The names '`c9x`' and '`iso9899:199x`' are deprecated.

'`c11`'
'`c1x`'
'`iso9899:2011`'
: ISO C11, the 2011 revision of the ISO C standard. This standard is substantially completely supported, modulo bugs, floating-point issues (mainly but not entirely relating to optional C11 features from Annexes F and G) and the optional Annexes K (Bounds-checking interfaces) and L (Analyzability). The name '`c1x`' is deprecated.

'`c17`'
'`c18`'
'`iso9899:2017`'
'`iso9899:2018`'
: ISO C17, the 2017 revision of the ISO C standard (published in 2018). This standard is same as C11 except for corrections of defects (all of which are also applied with '`-std=c11`') and a new value of `__STDC_VERSION__`, and so is supported to the same extent as C11.

'`c2x`'
: The next version of the ISO C standard, still under development. The support for this version is experimental and incomplete.

'`gnu90`'
'`gnu89`'
: GNU dialect of ISO C90 (including some C99 features).

'`gnu99`'
'`gnu9x`'
: GNU dialect of ISO C99. The name '`gnu9x`' is deprecated.

'`gnu11`'
'`gnu1x`'
: GNU dialect of ISO C11. The name '`gnu1x`' is deprecated.

'`gnu17`'
'`gnu18`'
: GNU dialect of ISO C17. This is the default for C code.

'`gnu2x`'
: The next version of the ISO C standard, still under development, plus GNU extensions. The support for this version is experimental and incomplete.

'`c++98`'
'`c++03`'
: The 1998 ISO C++ standard plus the 2003 technical corrigendum and some additional defect reports. Same as '`-ansi`' for C++ code.

'`gnu++98`'
'`gnu++03`'
: GNU dialect of '`-std=c++98`'.

'`c++11`'
'`c++0x`'
: The 2011 ISO C++ standard plus amendments. The name '`c++0x`' is deprecated.

'gnu++11'
'gnu++0x' GNU dialect of '-std=c++11'. The name 'gnu++0x' is deprecated.

'c++14'
'c++1y' The 2014 ISO C++ standard plus amendments. The name 'c++1y' is deprecated.

'gnu++14'
'gnu++1y' GNU dialect of '-std=c++14'. This is the default for C++ code. The name 'gnu++1y' is deprecated.

'c++17'
'c++1z' The 2017 ISO C++ standard plus amendments. The name 'c++1z' is deprecated.

'gnu++17'
'gnu++1z' GNU dialect of '-std=c++17'. The name 'gnu++1z' is deprecated.

'c++2a' The next revision of the ISO C++ standard, tentatively planned for 2020. Support is highly experimental, and will almost certainly change in incompatible ways in future releases.

'gnu++2a' GNU dialect of '-std=c++2a'. Support is highly experimental, and will almost certainly change in incompatible ways in future releases.

-fgnu89-inline

The option '-fgnu89-inline' tells GCC to use the traditional GNU semantics for inline functions when in C99 mode. See Section 6.45 [An Inline Function is As Fast As a Macro], page 564. Using this option is roughly equivalent to adding the gnu_inline function attribute to all inline functions (see Section 6.33 [Function Attributes], page 477).

The option '-fno-gnu89-inline' explicitly tells GCC to use the C99 semantics for inline when in C99 or gnu99 mode (i.e., it specifies the default behavior). This option is not supported in '-std=c90' or '-std=gnu90' mode.

The preprocessor macros __GNUC_GNU_INLINE__ and __GNUC_STDC_INLINE__ may be used to check which semantics are in effect for inline functions. See Section "Common Predefined Macros" in *The C Preprocessor*.

-fpermitted-flt-eval-methods=*style*

ISO/IEC TS 18661-3 defines new permissible values for FLT_EVAL_METHOD that indicate that operations and constants with a semantic type that is an interchange or extended format should be evaluated to the precision and range of that type. These new values are a superset of those permitted under C99/C11, which does not specify the meaning of other positive values of FLT_EVAL_METHOD. As such, code conforming to C11 may not have been written expecting the possibility of the new values.

'-fpermitted-flt-eval-methods' specifies whether the compiler should allow only the values of FLT_EVAL_METHOD specified in C99/C11, or the extended set of values specified in ISO/IEC TS 18661-3.

style is either c11 or ts-18661-3 as appropriate.

The default when in a standards compliant mode ('-std=c11' or similar) is '-fpermitted-flt-eval-methods=c11'. The default when in a GNU dialect ('-std=gnu11' or similar) is '-fpermitted-flt-eval-methods=ts-18661-3'.

-aux-info *filename*

Output to the given filename prototyped declarations for all functions declared and/or defined in a translation unit, including those in header files. This option is silently ignored in any language other than C.

Besides declarations, the file indicates, in comments, the origin of each declaration (source file and line), whether the declaration was implicit, prototyped or unprototyped ('I', 'N' for new or 'O' for old, respectively, in the first character after the line number and the colon), and whether it came from a declaration or a definition ('C' or 'F', respectively, in the following character). In the case of function definitions, a K&R-style list of arguments followed by their declarations is also provided, inside comments, after the declaration.

-fallow-parameterless-variadic-functions

Accept variadic functions without named parameters.

Although it is possible to define such a function, this is not very useful as it is not possible to read the arguments. This is only supported for C as this construct is allowed by C++.

-fno-asm Do not recognize asm, inline or typeof as a keyword, so that code can use these words as identifiers. You can use the keywords __asm__, __inline__ and __typeof__ instead. '-ansi' implies '-fno-asm'.

In C++, this switch only affects the typeof keyword, since asm and inline are standard keywords. You may want to use the '-fno-gnu-keywords' flag instead, which has the same effect. In C99 mode ('-std=c99' or '-std=gnu99'), this switch only affects the asm and typeof keywords, since inline is a standard keyword in ISO C99.

-fno-builtin

-fno-builtin-*function*

Don't recognize built-in functions that do not begin with '__builtin_' as prefix. See Section 6.59 [Other built-in functions provided by GCC], page 641, for details of the functions affected, including those which are not built-in functions when '-ansi' or '-std' options for strict ISO C conformance are used because they do not have an ISO standard meaning.

GCC normally generates special code to handle certain built-in functions more efficiently; for instance, calls to alloca may become single instructions which adjust the stack directly, and calls to memcpy may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library. In addition, when a function is recognized as a built-in function, GCC may use information about that function to warn about problems with calls to that function, or to generate more efficient code, even if the resulting code still contains calls to that function. For example, warnings are given with '-Wformat'

for bad calls to `printf` when `printf` is built in and `strlen` is known not to modify global memory.

With the '`-fno-builtin-function`' option only the built-in function *function* is disabled. *function* must not begin with '`__builtin_`'. If a function is named that is not built-in in this version of GCC, this option is ignored. There is no corresponding '`-fbuiltin-function`' option; if you wish to enable built-in functions selectively when using '`-fno-builtin`' or '`-ffreestanding`', you may define macros such as:

```
#define abs(n)          __builtin_abs ((n))
#define strcpy(d, s)    __builtin_strcpy ((d), (s))
```

`-fgimple`

Enable parsing of function definitions marked with `__GIMPLE`. This is an experimental feature that allows unit testing of GIMPLE passes.

`-fhosted`

Assert that compilation targets a hosted environment. This implies '`-fbuiltin`'. A hosted environment is one in which the entire standard library is available, and in which `main` has a return type of `int`. Examples are nearly everything except a kernel. This is equivalent to '`-fno-freestanding`'.

`-ffreestanding`

Assert that compilation targets a freestanding environment. This implies '`-fno-builtin`'. A freestanding environment is one in which the standard library may not exist, and program startup may not necessarily be at `main`. The most obvious example is an OS kernel. This is equivalent to '`-fno-hosted`'.

See Chapter 2 [Language Standards Supported by GCC], page 5, for details of freestanding and hosted environments.

`-fopenacc`

Enable handling of OpenACC directives `#pragma acc` in C/C++ and `!$acc` in Fortran. When '`-fopenacc`' is specified, the compiler generates accelerated code according to the OpenACC Application Programming Interface v2.0 `https://www.openacc.org`. This option implies '`-pthread`', and thus is only supported on targets that have support for '`-pthread`'.

`-fopenacc-dim=`*geom*

Specify default compute dimensions for parallel offload regions that do not explicitly specify. The *geom* value is a triple of ':'-separated sizes, in order 'gang', 'worker' and, 'vector'. A size can be omitted, to use a target-specific default value.

`-fopenmp` Enable handling of OpenMP directives `#pragma omp` in C/C++ and `!$omp` in Fortran. When '`-fopenmp`' is specified, the compiler generates parallel code according to the OpenMP Application Program Interface v4.5 `https://www.openmp.org`. This option implies '`-pthread`', and thus is only supported on targets that have support for '`-pthread`'. '`-fopenmp`' implies '`-fopenmp-simd`'.

`-fopenmp-simd`
Enable handling of OpenMP's SIMD directives with `#pragma omp` in C/C++ and `!$omp` in Fortran. Other OpenMP directives are ignored.

`-fgnu-tm` When the option '`-fgnu-tm`' is specified, the compiler generates code for the Linux variant of Intel's current Transactional Memory ABI specification document (Revision 1.1, May 6 2009). This is an experimental feature whose interface may change in future versions of GCC, as the official specification changes. Please note that not all architectures are supported for this feature.

For more information on GCC's support for transactional memory, See Section "The GNU Transactional Memory Library" in *GNU Transactional Memory Library*.

Note that the transactional memory feature is not supported with non-call exceptions ('`-fnon-call-exceptions`').

`-fms-extensions`
Accept some non-standard constructs used in Microsoft header files.

In C++ code, this allows member names in structures to be similar to previous types declarations.

```
typedef int UOW;
struct ABC {
  UOW UOW;
};
```

Some cases of unnamed fields in structures and unions are only accepted with this option. See Section 6.63 [Unnamed struct/union fields within structs/unions], page 809, for details.

Note that this option is off for all targets but x86 targets using ms-abi.

`-fplan9-extensions`
Accept some non-standard constructs used in Plan 9 code.

This enables '`-fms-extensions`', permits passing pointers to structures with anonymous fields to functions that expect pointers to elements of the type of the field, and permits referring to anonymous fields declared using a typedef. See Section 6.63 [Unnamed struct/union fields within structs/unions], page 809, for details. This is only supported for C, not C++.

`-fcond-mismatch`
Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void. This option is not supported for C++.

`-flax-vector-conversions`
Allow implicit conversions between vectors with differing numbers of elements and/or incompatible element types. This option should not be used for new code.

`-funsigned-char`
Let the type `char` be unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default.

Ideally, a portable program should always use `signed char` or `unsigned char` when it depends on the signedness of an object. But many programs have been written to use plain `char` and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for. This option, and its inverse, let you make such a program work with the opposite default.

The type `char` is always a distinct type from each of `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

`-fsigned-char`

Let the type `char` be signed, like `signed char`.

Note that this is equivalent to '`-fno-unsigned-char`', which is the negative form of '`-funsigned-char`'. Likewise, the option '`-fno-signed-char`' is equivalent to '`-funsigned-char`'.

`-fsigned-bitfields`
`-funsigned-bitfields`
`-fno-signed-bitfields`
`-fno-unsigned-bitfields`

These options control whether a bit-field is signed or unsigned, when the declaration does not use either `signed` or `unsigned`. By default, such a bit-field is signed, because this is consistent: the basic integer types such as `int` are signed types.

`-fsso-struct=`*endianness*

Set the default scalar storage order of structures and unions to the specified endianness. The accepted values are '`big-endian`', '`little-endian`' and '`native`' for the native endianness of the target (the default). This option is not supported for C++.

Warning: the '`-fsso-struct`' switch causes GCC to generate code that is not binary compatible with code generated without it if the specified endianness is not the native endianness of the target.

3.5 Options Controlling C++ Dialect

This section describes the command-line options that are only meaningful for C++ programs. You can also use most of the GNU compiler options regardless of what language your program is in. For example, you might compile a file '`firstClass.C`' like this:

```
g++ -g -fstrict-enums -O -c firstClass.C
```

In this example, only '`-fstrict-enums`' is an option meant only for C++ programs; you can use the other options with any language supported by GCC.

Some options for compiling C programs, such as '`-std`', are also relevant for C++ programs. See Section 3.4 [Options Controlling C Dialect], page 37.

Here is a list of options that are *only* for compiling C++ programs:

`-fabi-version=`*n*

Use version *n* of the C++ ABI. The default is version 0.

Version 0 refers to the version conforming most closely to the C++ ABI specification. Therefore, the ABI obtained using version 0 will change in different versions of G++ as ABI bugs are fixed.

Version 1 is the version of the C++ ABI that first appeared in G++ 3.2.

Version 2 is the version of the C++ ABI that first appeared in G++ 3.4, and was the default through G++ 4.9.

Version 3 corrects an error in mangling a constant address as a template argument.

Version 4, which first appeared in G++ 4.5, implements a standard mangling for vector types.

Version 5, which first appeared in G++ 4.6, corrects the mangling of attribute const/volatile on function pointer types, decltype of a plain decl, and use of a function parameter in the declaration of another parameter.

Version 6, which first appeared in G++ 4.7, corrects the promotion behavior of C++11 scoped enums and the mangling of template argument packs, const/static_cast, prefix ++ and –, and a class scope function used as a template argument.

Version 7, which first appeared in G++ 4.8, that treats nullptr_t as a builtin type and corrects the mangling of lambdas in default argument scope.

Version 8, which first appeared in G++ 4.9, corrects the substitution behavior of function types with function-cv-qualifiers.

Version 9, which first appeared in G++ 5.2, corrects the alignment of `nullptr_t`.

Version 10, which first appeared in G++ 6.1, adds mangling of attributes that affect type identity, such as ia32 calling convention attributes (e.g. '`stdcall`').

Version 11, which first appeared in G++ 7, corrects the mangling of sizeof... expressions and operator names. For multiple entities with the same name within a function, that are declared in different scopes, the mangling now changes starting with the twelfth occurrence. It also implies '`-fnew-inheriting-ctors`'.

Version 12, which first appeared in G++ 8, corrects the calling conventions for empty classes on the x86_64 target and for classes with only deleted copy/move constructors. It accidentally changes the calling convention for classes with a deleted copy constructor and a trivial move constructor.

Version 13, which first appeared in G++ 8.2, fixes the accidental change in version 12.

See also '`-Wabi`'.

`-fabi-compat-version=`*n*

On targets that support strong aliases, G++ works around mangling changes by creating an alias with the correct mangled name when defining a symbol with an incorrect mangled name. This switch specifies which ABI version to use for the alias.

With '`-fabi-version=0`' (the default), this defaults to 11 (GCC 7 compatibility). If another ABI version is explicitly selected, this defaults to 0. For compatibility with GCC versions 3.2 through 4.9, use '`-fabi-compat-version=2`'.

If this option is not provided but '`-Wabi=`*n*' is, that version is used for compatibility aliases. If this option is provided along with '`-Wabi`' (without the version), the version from this option is used for the warning.

`-fno-access-control`

Turn off all access checking. This switch is mainly useful for working around bugs in the access control code.

`-faligned-new`

Enable support for C++17 `new` of types that require more alignment than `void* ::operator new(std::size_t)` provides. A numeric argument such as `-faligned-new=32` can be used to specify how much alignment (in bytes) is provided by that function, but few users will need to override the default of `alignof(std::max_align_t)`.

This flag is enabled by default for '`-std=c++17`'.

`-fchar8_t`
`-fno-char8_t`

Enable support for `char8_t` as adopted for C++2a. This includes the addition of a new `char8_t` fundamental type, changes to the types of UTF-8 string and character literals, new signatures for user-defined literals, associated standard library updates, and new `__cpp_char8_t` and `__cpp_lib_char8_t` feature test macros.

This option enables functions to be overloaded for ordinary and UTF-8 strings:

```
int f(const char *);    // #1
int f(const char8_t *); // #2
int v1 = f("text");     // Calls #1
int v2 = f(u8"text");   // Calls #2
```

and introduces new signatures for user-defined literals:

```
int operator""_udl1(char8_t);
int v3 = u8'x'_udl1;
int operator""_udl2(const char8_t*, std::size_t);
int v4 = u8"text"_udl2;
template<typename T, T...> int operator""_udl3();
int v5 = u8"text"_udl3;
```

The change to the types of UTF-8 string and character literals introduces incompatibilities with ISO C++11 and later standards. For example, the following code is well-formed under ISO C++11, but is ill-formed when '`-fchar8_t`' is specified.

```
char ca[] = u8"xx";     // error: char-array initialized from wide
                        //        string
const char *cp = u8"xx";// error: invalid conversion from
                        //        'const char8_t*' to 'const char*'
int f(const char*);
auto v = f(u8"xx");     // error: invalid conversion from
                        //        'const char8_t*' to 'const char*'
std::string s{u8"xx"};  // error: no matching function for call to
                        //        'std::basic_string<char>::basic_string()'
using namespace std::literals;
s = u8"xx"s;            // error: conversion from
                        //        'basic_string<char8_t>' to non-scalar
                        //        type 'basic_string<char>' requested
```

`-fcheck-new`

Check that the pointer returned by `operator new` is non-null before attempting to modify the storage allocated. This check is normally unnecessary because

the C++ standard specifies that `operator new` only returns `0` if it is declared `throw()`, in which case the compiler always checks the return value even without this option. In all other cases, when `operator new` has a non-empty exception specification, memory exhaustion is signalled by throwing `std::bad_alloc`. See also '`new (nothrow)`'.

`-fconcepts`

Enable support for the C++ Extensions for Concepts Technical Specification, ISO 19217 (2015), which allows code like

```
template <class T> concept bool Addable = requires (T t) { t + t; };
template <Addable T> T add (T a, T b) { return a + b; }
```

`-fconstexpr-depth=n`

Set the maximum nested evaluation depth for C++11 constexpr functions to *n*. A limit is needed to detect endless recursion during constant expression evaluation. The minimum specified by the standard is 512.

`-fconstexpr-loop-limit=n`

Set the maximum number of iterations for a loop in C++14 constexpr functions to *n*. A limit is needed to detect infinite loops during constant expression evaluation. The default is 262144 (1<<18).

`-fconstexpr-ops-limit=n`

Set the maximum number of operations during a single constexpr evaluation. Even when number of iterations of a single loop is limited with the above limit, if there are several nested loops and each of them has many iterations but still smaller than the above limit, or if in a body of some loop or even outside of a loop too many expressions need to be evaluated, the resulting constexpr evaluation might take too long. The default is 33554432 (1<<25).

`-fdeduce-init-list`

Enable deduction of a template type parameter as `std::initializer_list` from a brace-enclosed initializer list, i.e.

```
template <class T> auto forward(T t) -> decltype (realfn (t))
{
  return realfn (t);
}

void f()
{
  forward({1,2}); // call forward<std::initializer_list<int>>
}
```

This deduction was implemented as a possible extension to the originally proposed semantics for the C++11 standard, but was not part of the final standard, so it is disabled by default. This option is deprecated, and may be removed in a future version of G++.

`-fno-elide-constructors`

The C++ standard allows an implementation to omit creating a temporary that is only used to initialize another object of the same type. Specifying this option disables that optimization, and forces G++ to call the copy constructor in all

cases. This option also causes G++ to call trivial member functions which otherwise would be expanded inline.

In C++17, the compiler is required to omit these temporaries, but this option still affects trivial member functions.

`-fno-enforce-eh-specs`

Don't generate code to check for violation of exception specifications at run time. This option violates the C++ standard, but may be useful for reducing code size in production builds, much like defining `NDEBUG`. This does not give user code permission to throw exceptions in violation of the exception specifications; the compiler still optimizes based on the specifications, so throwing an unexpected exception results in undefined behavior at run time.

`-fextern-tls-init`
`-fno-extern-tls-init`

The C++11 and OpenMP standards allow `thread_local` and `threadprivate` variables to have dynamic (runtime) initialization. To support this, any use of such a variable goes through a wrapper function that performs any necessary initialization. When the use and definition of the variable are in the same translation unit, this overhead can be optimized away, but when the use is in a different translation unit there is significant overhead even if the variable doesn't actually need dynamic initialization. If the programmer can be sure that no use of the variable in a non-defining TU needs to trigger dynamic initialization (either because the variable is statically initialized, or a use of the variable in the defining TU will be executed before any uses in another TU), they can avoid this overhead with the '`-fno-extern-tls-init`' option.

On targets that support symbol aliases, the default is '`-fextern-tls-init`'. On targets that do not support symbol aliases, the default is '`-fno-extern-tls-init`'.

`-fno-gnu-keywords`

Do not recognize `typeof` as a keyword, so that code can use this word as an identifier. You can use the keyword `__typeof__` instead. This option is implied by the strict ISO C++ dialects: '`-ansi`', '`-std=c++98`', '`-std=c++11`', etc.

`-fno-implicit-templates`

Never emit code for non-inline templates that are instantiated implicitly (i.e. by use); only emit code for explicit instantiations. If you use this option, you must take care to structure your code to include all the necessary explicit instantiations to avoid getting undefined symbols at link time. See Section 7.5 [Template Instantiation], page 818, for more information.

`-fno-implicit-inline-templates`

Don't emit code for implicit instantiations of inline templates, either. The default is to handle inlines differently so that compiles with and without optimization need the same set of explicit instantiations.

`-fno-implement-inlines`
To save space, do not emit out-of-line copies of inline functions controlled by `#pragma implementation`. This causes linker errors if these functions are not inlined everywhere they are called.

`-fms-extensions`
Disable Wpedantic warnings about constructs used in MFC, such as implicit int and getting a pointer to member function via non-standard syntax.

`-fnew-inheriting-ctors`
Enable the P0136 adjustment to the semantics of C++11 constructor inheritance. This is part of C++17 but also considered to be a Defect Report against C++11 and C++14. This flag is enabled by default unless '`-fabi-version=10`' or lower is specified.

`-fnew-ttp-matching`
Enable the P0522 resolution to Core issue 150, template template parameters and default arguments: this allows a template with default template arguments as an argument for a template template parameter with fewer template parameters. This flag is enabled by default for '`-std=c++17`'.

`-fno-nonansi-builtins`
Disable built-in declarations of functions that are not mandated by ANSI/ISO C. These include `ffs`, `alloca`, `_exit`, `index`, `bzero`, `conjf`, and other related functions.

`-fnothrow-opt`
Treat a `throw()` exception specification as if it were a `noexcept` specification to reduce or eliminate the text size overhead relative to a function with no exception specification. If the function has local variables of types with non-trivial destructors, the exception specification actually makes the function smaller because the EH cleanups for those variables can be optimized away. The semantic effect is that an exception thrown out of a function with such an exception specification results in a call to `terminate` rather than `unexpected`.

`-fno-operator-names`
Do not treat the operator name keywords `and`, `bitand`, `bitor`, `compl`, `not`, `or` and `xor` as synonyms as keywords.

`-fno-optional-diags`
Disable diagnostics that the standard says a compiler does not need to issue. Currently, the only such diagnostic issued by G++ is the one for a name having multiple meanings within a class.

`-fpermissive`
Downgrade some diagnostics about nonconformant code from errors to warnings. Thus, using '`-fpermissive`' allows some nonconforming code to compile.

`-fno-pretty-templates`
When an error message refers to a specialization of a function template, the compiler normally prints the signature of the template followed by the template arguments and any typedefs or typenames in the signature (e.g. `void f(T) [with`

T = int] rather than void f(int)) so that it's clear which template is involved. When an error message refers to a specialization of a class template, the compiler omits any template arguments that match the default template arguments for that template. If either of these behaviors make it harder to understand the error message rather than easier, you can use '-fno-pretty-templates' to disable them.

-frepo
: Enable automatic template instantiation at link time. This option also implies '-fno-implicit-templates'. See Section 7.5 [Template Instantiation], page 818, for more information.

-fno-rtti
: Disable generation of information about every class with virtual functions for use by the C++ run-time type identification features (dynamic_cast and typeid). If you don't use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but G++ generates it as needed. The dynamic_cast operator can still be used for casts that do not require run-time type information, i.e. casts to void * or to unambiguous base classes.

  Mixing code compiled with '-frtti' with that compiled with '-fno-rtti' may not work. For example, programs may fail to link if a class compiled with '-fno-rtti' is used as a base for a class compiled with '-frtti'.

-fsized-deallocation
: Enable the built-in global declarations

```
void operator delete (void *, std::size_t) noexcept;
void operator delete[] (void *, std::size_t) noexcept;
```

  as introduced in C++14. This is useful for user-defined replacement deallocation functions that, for example, use the size of the object to make deallocation faster. Enabled by default under '-std=c++14' and above. The flag '-Wsized-deallocation' warns about places that might want to add a definition.

-fstrict-enums
: Allow the compiler to optimize using the assumption that a value of enumerated type can only be one of the values of the enumeration (as defined in the C++ standard; basically, a value that can be represented in the minimum number of bits needed to represent all the enumerators). This assumption may not be valid if the program uses a cast to convert an arbitrary integer value to the enumerated type.

-fstrong-eval-order
: Evaluate member access, array subscripting, and shift expressions in left-to-right order, and evaluate assignment in right-to-left order, as adopted for C++17. Enabled by default with '-std=c++17'. '-fstrong-eval-order=some' enables just the ordering of member access and shift expressions, and is the default without '-std=c++17'.

-ftemplate-backtrace-limit=*n*
: Set the maximum number of template instantiation notes for a single warning or error to *n*. The default value is 10.

`-ftemplate-depth=n`

Set the maximum instantiation depth for template classes to *n*. A limit on the template instantiation depth is needed to detect endless recursions during template class instantiation. ANSI/ISO C++ conforming programs must not rely on a maximum depth greater than 17 (changed to 1024 in C++11). The default value is 900, as the compiler can run out of stack space before hitting 1024 in some situations.

`-fno-threadsafe-statics`

Do not emit the extra code to use the routines specified in the C++ ABI for thread-safe initialization of local statics. You can use this option to reduce code size slightly in code that doesn't need to be thread-safe.

`-fuse-cxa-atexit`

Register destructors for objects with static storage duration with the `__cxa_atexit` function rather than the `atexit` function. This option is required for fully standards-compliant handling of static destructors, but only works if your C library supports `__cxa_atexit`.

`-fno-use-cxa-get-exception-ptr`

Don't use the `__cxa_get_exception_ptr` runtime routine. This causes `std::uncaught_exception` to be incorrect, but is necessary if the runtime routine is not available.

`-fvisibility-inlines-hidden`

This switch declares that the user does not attempt to compare pointers to inline functions or methods where the addresses of the two functions are taken in different shared objects.

The effect of this is that GCC may, effectively, mark inline methods with `__attribute__ ((visibility ("hidden")))` so that they do not appear in the export table of a DSO and do not require a PLT indirection when used within the DSO. Enabling this option can have a dramatic effect on load and link times of a DSO as it massively reduces the size of the dynamic export table when the library makes heavy use of templates.

The behavior of this switch is not quite the same as marking the methods as hidden directly, because it does not affect static variables local to the function or cause the compiler to deduce that the function is defined in only one shared object.

You may mark a method as having a visibility explicitly to negate the effect of the switch for that method. For example, if you do want to compare pointers to a particular inline method, you might mark it as having default visibility. Marking the enclosing class with explicit visibility has no effect.

Explicitly instantiated inline methods are unaffected by this option as their linkage might otherwise cross a shared library boundary. See Section 7.5 [Template Instantiation], page 818.

`-fvisibility-ms-compat`

This flag attempts to use visibility settings to make GCC's C++ linkage model compatible with that of Microsoft Visual Studio.

The flag makes these changes to GCC's linkage model:

1. It sets the default visibility to `hidden`, like '`-fvisibility=hidden`'.
2. Types, but not their members, are not hidden by default.
3. The One Definition Rule is relaxed for types without explicit visibility specifications that are defined in more than one shared object: those declarations are permitted if they are permitted when this option is not used.

In new code it is better to use '`-fvisibility=hidden`' and export those classes that are intended to be externally visible. Unfortunately it is possible for code to rely, perhaps accidentally, on the Visual Studio behavior.

Among the consequences of these changes are that static data members of the same type with the same name but defined in different shared objects are different, so changing one does not change the other; and that pointers to function members defined in different shared objects may not compare equal. When this flag is given, it is a violation of the ODR to define types with the same name differently.

`-fno-weak`

Do not use weak symbol support, even if it is provided by the linker. By default, G++ uses weak symbols if they are available. This option exists only for testing, and should not be used by end-users; it results in inferior code and has no benefits. This option may be removed in a future release of G++.

`-nostdinc++`

Do not search for header files in the standard directories specific to C++, but do still search the other standard directories. (This option is used when building the C++ library.)

In addition, these optimization, warning, and code generation options have meanings only for C++ programs:

`-Wabi` (C, Objective-C, C++ and Objective-C++ only)

Warn when G++ it generates code that is probably not compatible with the vendor-neutral C++ ABI. Since G++ now defaults to updating the ABI with each major release, normally '`-Wabi`' will warn only if there is a check added later in a release series for an ABI issue discovered since the initial release. '`-Wabi`' will warn about more things if an older ABI version is selected (with '`-fabi-version=n`').

'`-Wabi`' can also be used with an explicit version number to warn about compatibility with a particular '`-fabi-version`' level, e.g. '`-Wabi=2`' to warn about changes relative to '`-fabi-version=2`'.

If an explicit version number is provided and '`-fabi-compat-version`' is not specified, the version number from this option is used for compatibility aliases. If no explicit version number is provided with this option, but '`-fabi-compat-version`' is specified, that version number is used for ABI warnings.

Although an effort has been made to warn about all such cases, there are probably some cases that are not warned about, even though G++ is generating

incompatible code. There may also be cases where warnings are emitted even though the code that is generated is compatible.

You should rewrite your code to avoid these warnings if you are concerned about the fact that code generated by G++ may not be binary compatible with code generated by other compilers.

Known incompatibilities in '`-fabi-version=2`' (which was the default from GCC 3.4 to 4.9) include:

- A template with a non-type template parameter of reference type was mangled incorrectly:

  ```
  extern int N;
  template <int &> struct S {};
  void n (S<N>) {2}
  ```

  This was fixed in '`-fabi-version=3`'.

- SIMD vector types declared using `__attribute ((vector_size))` were mangled in a non-standard way that does not allow for overloading of functions taking vectors of different sizes.

  The mangling was changed in '`-fabi-version=4`'.

- `__attribute ((const))` and `noreturn` were mangled as type qualifiers, and `decltype` of a plain declaration was folded away.

  These mangling issues were fixed in '`-fabi-version=5`'.

- Scoped enumerators passed as arguments to a variadic function are promoted like unscoped enumerators, causing `va_arg` to complain. On most targets this does not actually affect the parameter passing ABI, as there is no way to pass an argument smaller than `int`.

  Also, the ABI changed the mangling of template argument packs, `const_cast`, `static_cast`, prefix increment/decrement, and a class scope function used as a template argument.

  These issues were corrected in '`-fabi-version=6`'.

- Lambdas in default argument scope were mangled incorrectly, and the ABI changed the mangling of `nullptr_t`.

  These issues were corrected in '`-fabi-version=7`'.

- When mangling a function type with function-cv-qualifiers, the un-qualified function type was incorrectly treated as a substitution candidate.

  This was fixed in '`-fabi-version=8`', the default for GCC 5.1.

- `decltype(nullptr)` incorrectly had an alignment of 1, leading to unaligned accesses. Note that this did not affect the ABI of a function with a `nullptr_t` parameter, as parameters have a minimum alignment.

  This was fixed in '`-fabi-version=9`', the default for GCC 5.2.

- Target-specific attributes that affect the identity of a type, such as ia32 calling conventions on a function type (stdcall, regparm, etc.), did not affect the mangled name, leading to name collisions when function pointers were used as template arguments.

  This was fixed in '`-fabi-version=10`', the default for GCC 6.1.

It also warns about psABI-related changes. The known psABI changes at this point include:

- For SysV/x86-64, unions with `long double` members are passed in memory as specified in psABI. For example:

```
union U {
  long double ld;
  int i;
};
```

  `union U` is always passed in memory.

`-Wabi-tag` (C++ and Objective-C++ only)
Warn when a type with an ABI tag is used in a context that does not have that ABI tag. See Section 7.7 [C++ Attributes], page 821 for more information about ABI tags.

`-Wctor-dtor-privacy` (C++ and Objective-C++ only)
Warn when a class seems unusable because all the constructors or destructors in that class are private, and it has neither friends nor public static member functions. Also warn if there are no non-private methods, and there's at least one private member function that isn't a constructor or destructor.

`-Wdelete-non-virtual-dtor` (C++ and Objective-C++ only)
Warn when `delete` is used to destroy an instance of a class that has virtual functions and non-virtual destructor. It is unsafe to delete an instance of a derived class through a pointer to a base class if the base class does not have a virtual destructor. This warning is enabled by '`-Wall`'.

`-Wdeprecated-copy` (C++ and Objective-C++ only)
Warn that the implicit declaration of a copy constructor or copy assignment operator is deprecated if the class has a user-provided copy constructor or copy assignment operator, in C++11 and up. This warning is enabled by '`-Wextra`'. With '`-Wdeprecated-copy-dtor`', also deprecate if the class has a user-provided destructor.

`-Wno-init-list-lifetime` (C++ and Objective-C++ only)
Do not warn about uses of `std::initializer_list` that are likely to result in dangling pointers. Since the underlying array for an `initializer_list` is handled like a normal C++ temporary object, it is easy to inadvertently keep a pointer to the array past the end of the array's lifetime. For example:

- If a function returns a temporary `initializer_list`, or a local `initializer_list` variable, the array's lifetime ends at the end of the return statement, so the value returned has a dangling pointer.
- If a new-expression creates an `initializer_list`, the array only lives until the end of the enclosing full-expression, so the `initializer_list` in the heap has a dangling pointer.
- When an `initializer_list` variable is assigned from a brace-enclosed initializer list, the temporary array created for the right side of the assignment only lives until the end of the full-expression, so at the next statement the `initializer_list` variable has a dangling pointer.

```
// li's initial underlying array lives as long as li
std::initializer_list<int> li = { 1,2,3 };
// assignment changes li to point to a temporary array
li = { 4, 5 };
// now the temporary is gone and li has a dangling pointer
int i = li.begin()[0] // undefined behavior
```

- When a list constructor stores the `begin` pointer from the `initializer_list` argument, this doesn't extend the lifetime of the array, so if a class variable is constructed from a temporary `initializer_list`, the pointer is left dangling by the end of the variable declaration statement.

`-Wliteral-suffix` (C++ and Objective-C++ only)

Warn when a string or character literal is followed by a ud-suffix which does not begin with an underscore. As a conforming extension, GCC treats such suffixes as separate preprocessing tokens in order to maintain backwards compatibility with code that uses formatting macros from `<inttypes.h>`. For example:

```
#define __STDC_FORMAT_MACROS
#include <inttypes.h>
#include <stdio.h>

int main() {
  int64_t i64 = 123;
  printf("My int64: %" PRId64"\n", i64);
}
```

In this case, `PRId64` is treated as a separate preprocessing token.

Additionally, warn when a user-defined literal operator is declared with a literal suffix identifier that doesn't begin with an underscore. Literal suffix identifiers that don't begin with an underscore are reserved for future standardization.

This warning is enabled by default.

`-Wlto-type-mismatch`

During the link-time optimization warn about type mismatches in global declarations from different compilation units. Requires '`-flto`' to be enabled. Enabled by default.

`-Wno-narrowing` (C++ and Objective-C++ only)

For C++11 and later standards, narrowing conversions are diagnosed by default, as required by the standard. A narrowing conversion from a constant produces an error, and a narrowing conversion from a non-constant produces a warning, but '`-Wno-narrowing`' suppresses the diagnostic. Note that this does not affect the meaning of well-formed code; narrowing conversions are still considered ill-formed in SFINAE contexts.

With '`-Wnarrowing`' in C++98, warn when a narrowing conversion prohibited by C++11 occurs within '`{ }`', e.g.

```
int i = { 2.2 }; // error: narrowing from double to int
```

This flag is included in '`-Wall`' and '`-Wc++11-compat`'.

`-Wnoexcept` (C++ and Objective-C++ only)

Warn when a noexcept-expression evaluates to false because of a call to a function that does not have a non-throwing exception specification (i.e. `throw()` or `noexcept`) but is known by the compiler to never throw an exception.

`-Wnoexcept-type` (C++ and Objective-C++ only)

Warn if the C++17 feature making `noexcept` part of a function type changes the mangled name of a symbol relative to C++14. Enabled by '`-Wabi`' and '`-Wc++17-compat`'.

As an example:

```
template <class T> void f(T t) { t(); };
void g() noexcept;
void h() { f(g); }
```

In C++14, `f` calls `f<void(*)()>`, but in C++17 it calls `f<void(*)()noexcept>`.

`-Wclass-memaccess` (C++ and Objective-C++ only)

Warn when the destination of a call to a raw memory function such as `memset` or `memcpy` is an object of class type, and when writing into such an object might bypass the class non-trivial or deleted constructor or copy assignment, violate const-correctness or encapsulation, or corrupt virtual table pointers. Modifying the representation of such objects may violate invariants maintained by member functions of the class. For example, the call to `memset` below is undefined because it modifies a non-trivial class object and is, therefore, diagnosed. The safe way to either initialize or clear the storage of objects of such types is by using the appropriate constructor or assignment operator, if one is available.

```
std::string str = "abc";
memset (&str, 0, sizeof str);
```

The '`-Wclass-memaccess`' option is enabled by '`-Wall`'. Explicitly casting the pointer to the class object to `void *` or to a type that can be safely accessed by the raw memory function suppresses the warning.

`-Wnon-virtual-dtor` (C++ and Objective-C++ only)

Warn when a class has virtual functions and an accessible non-virtual destructor itself or in an accessible polymorphic base class, in which case it is possible but unsafe to delete an instance of a derived class through a pointer to the class itself or base class. This warning is automatically enabled if '`-Weffc++`' is specified.

`-Wregister` (C++ and Objective-C++ only)

Warn on uses of the `register` storage class specifier, except when it is part of the GNU Section 6.47.5 [Explicit Register Variables], page 621 extension. The use of the `register` keyword as storage class specifier has been deprecated in C++11 and removed in C++17. Enabled by default with '`-std=c++17`'.

`-Wreorder` (C++ and Objective-C++ only)

Warn when the order of member initializers given in the code does not match the order in which they must be executed. For instance:

```
struct A {
  int i;
  int j;
  A(): j (0), i (1) { }
};
```

The compiler rearranges the member initializers for `i` and `j` to match the declaration order of the members, emitting a warning to that effect. This warning is enabled by '`-Wall`'.

`-Wno-pessimizing-move` (C++ and Objective-C++ only)

This warning warns when a call to `std::move` prevents copy elision. A typical scenario when copy elision can occur is when returning in a function with a class return type, when the expression being returned is the name of a non-volatile automatic object, and is not a function parameter, and has the same type as the function return type.

```
struct T {
...
};
T fn()
{
  T t;
  ...
  return std::move (t);
}
```

But in this example, the `std::move` call prevents copy elision.

This warning is enabled by '`-Wall`'.

`-Wno-redundant-move` (C++ and Objective-C++ only)

This warning warns about redundant calls to `std::move`; that is, when a move operation would have been performed even without the `std::move` call. This happens because the compiler is forced to treat the object as if it were an rvalue in certain situations such as returning a local variable, where copy elision isn't applicable. Consider:

```
struct T {
...
};
T fn(T t)
{
  ...
  return std::move (t);
}
```

Here, the `std::move` call is redundant. Because G++ implements Core Issue 1579, another example is:

```
struct T { // convertible to U
...
};
struct U {
...
};
U fn()
{
  T t;
  ...
  return std::move (t);
}
```

In this example, copy elision isn't applicable because the type of the expression being returned and the function return type differ, yet G++ treats the return value as if it were designated by an rvalue.

This warning is enabled by '`-Wextra`'.

`-fext-numeric-literals` (C++ and Objective-C++ only)
: Accept imaginary, fixed-point, or machine-defined literal number suffixes as GNU extensions. When this option is turned off these suffixes are treated as C++11 user-defined literal numeric suffixes. This is on by default for all pre-C++11 dialects and all GNU dialects: '`-std=c++98`', '`-std=gnu++98`', '`-std=gnu++11`', '`-std=gnu++14`'. This option is off by default for ISO C++11 onwards ('`-std=c++11`', ...).

The following '`-W...`' options are not affected by '`-Wall`'.

`-Weffc++` (C++ and Objective-C++ only)
: Warn about violations of the following style guidelines from Scott Meyers' *Effective C++* series of books:

- Define a copy constructor and an assignment operator for classes with dynamically-allocated memory.
- Prefer initialization to assignment in constructors.
- Have `operator=` return a reference to `*this`.
- Don't try to return a reference when you must return an object.
- Distinguish between prefix and postfix forms of increment and decrement operators.
- Never overload `&&`, `||`, or `,`.

This option also enables '`-Wnon-virtual-dtor`', which is also one of the effective C++ recommendations. However, the check is extended to warn about the lack of virtual destructor in accessible non-polymorphic bases classes too.

When selecting this option, be aware that the standard library headers do not obey all of these guidelines; use '`grep -v`' to filter out those warnings.

`-Wstrict-null-sentinel` (C++ and Objective-C++ only)
: Warn about the use of an uncasted `NULL` as sentinel. When compiling only with GCC this is a valid sentinel, as `NULL` is defined to `__null`. Although it is a null pointer constant rather than a null pointer, it is guaranteed to be of the same size as a pointer. But this use is not portable across different compilers.

`-Wno-non-template-friend` (C++ and Objective-C++ only)
: Disable warnings when non-template friend functions are declared within a template. In very old versions of GCC that predate implementation of the ISO standard, declarations such as '`friend int foo(int)`', where the name of the friend is an unqualified-id, could be interpreted as a particular specialization of a template function; the warning exists to diagnose compatibility problems, and is enabled by default.

`-Wold-style-cast` (C++ and Objective-C++ only)
: Warn if an old-style (C-style) cast to a non-void type is used within a C++ program. The new-style casts (`dynamic_cast`, `static_cast`, `reinterpret_cast`, and `const_cast`) are less vulnerable to unintended effects and much easier to search for.

`-Woverloaded-virtual` (C++ and Objective-C++ only)

Warn when a function declaration hides virtual functions from a base class. For example, in:

```
struct A {
  virtual void f();
};

struct B: public A {
  void f(int);
};
```

the `A` class version of `f` is hidden in `B`, and code like:

```
B* b;
b->f();
```

fails to compile.

`-Wno-pmf-conversions` (C++ and Objective-C++ only)

Disable the diagnostic for converting a bound pointer to member function to a plain pointer.

`-Wsign-promo` (C++ and Objective-C++ only)

Warn when overload resolution chooses a promotion from unsigned or enumerated type to a signed type, over a conversion to an unsigned type of the same size. Previous versions of G++ tried to preserve unsignedness, but the standard mandates the current behavior.

`-Wtemplates` (C++ and Objective-C++ only)

Warn when a primary template declaration is encountered. Some coding rules disallow templates, and this may be used to enforce that rule. The warning is inactive inside a system header file, such as the STL, so one can still use the STL. One may also instantiate or specialize templates.

`-Wmultiple-inheritance` (C++ and Objective-C++ only)

Warn when a class is defined with multiple direct base classes. Some coding rules disallow multiple inheritance, and this may be used to enforce that rule. The warning is inactive inside a system header file, such as the STL, so one can still use the STL. One may also define classes that indirectly use multiple inheritance.

`-Wvirtual-inheritance`

Warn when a class is defined with a virtual direct base class. Some coding rules disallow multiple inheritance, and this may be used to enforce that rule. The warning is inactive inside a system header file, such as the STL, so one can still use the STL. One may also define classes that indirectly use virtual inheritance.

`-Wnamespaces`

Warn when a namespace definition is opened. Some coding rules disallow namespaces, and this may be used to enforce that rule. The warning is inactive inside a system header file, such as the STL, so one can still use the STL. One may also use using directives and qualified names.

`-Wno-terminate` (C++ and Objective-C++ only)

Disable the warning about a throw-expression that will immediately result in a call to `terminate`.

`-Wno-class-conversion` (C++ and Objective-C++ only)
Disable the warning about the case when a conversion function converts an object to the same type, to a base class of that type, or to void; such a conversion function will never be called.

3.6 Options Controlling Objective-C and Objective-C++ Dialects

(NOTE: This manual does not describe the Objective-C and Objective-C++ languages themselves. See Chapter 2 [Language Standards Supported by GCC], page 5, for references.)

This section describes the command-line options that are only meaningful for Objective-C and Objective-C++ programs. You can also use most of the language-independent GNU compiler options. For example, you might compile a file '`some_class.m`' like this:

```
gcc -g -fgnu-runtime -O -c some_class.m
```

In this example, '`-fgnu-runtime`' is an option meant only for Objective-C and Objective-C++ programs; you can use the other options with any language supported by GCC.

Note that since Objective-C is an extension of the C language, Objective-C compilations may also use options specific to the C front-end (e.g., '`-Wtraditional`'). Similarly, Objective-C++ compilations may use C++-specific options (e.g., '`-Wabi`').

Here is a list of options that are *only* for compiling Objective-C and Objective-C++ programs:

`-fconstant-string-class=`*`class-name`*
Use *class-name* as the name of the class to instantiate for each literal string specified with the syntax `@"..."`. The default class name is `NXConstantString` if the GNU runtime is being used, and `NSConstantString` if the NeXT runtime is being used (see below). The '`-fconstant-cfstrings`' option, if also present, overrides the '`-fconstant-string-class`' setting and cause `@"..."` literals to be laid out as constant CoreFoundation strings.

`-fgnu-runtime`
Generate object code compatible with the standard GNU Objective-C runtime. This is the default for most types of systems.

`-fnext-runtime`
Generate output compatible with the NeXT runtime. This is the default for NeXT-based systems, including Darwin and Mac OS X. The macro `__NEXT_RUNTIME__` is predefined if (and only if) this option is used.

`-fno-nil-receivers`
Assume that all Objective-C message dispatches (`[receiver message:arg]`) in this translation unit ensure that the receiver is not `nil`. This allows for more efficient entry points in the runtime to be used. This option is only available in conjunction with the NeXT runtime and ABI version 0 or 1.

`-fobjc-abi-version=`*`n`*
Use version *n* of the Objective-C ABI for the selected runtime. This option is currently supported only for the NeXT runtime. In that case, Version 0 is the traditional (32-bit) ABI without support for properties and other Objective-C 2.0 additions. Version 1 is the traditional (32-bit) ABI with support for

properties and other Objective-C 2.0 additions. Version 2 is the modern (64-bit) ABI. If nothing is specified, the default is Version 0 on 32-bit target machines, and Version 2 on 64-bit target machines.

`-fobjc-call-cxx-cdtors`

For each Objective-C class, check if any of its instance variables is a C++ object with a non-trivial default constructor. If so, synthesize a special `- (id) .cxx_construct` instance method which runs non-trivial default constructors on any such instance variables, in order, and then return `self`. Similarly, check if any instance variable is a C++ object with a non-trivial destructor, and if so, synthesize a special `- (void) .cxx_destruct` method which runs all such default destructors, in reverse order.

The `- (id) .cxx_construct` and `- (void) .cxx_destruct` methods thusly generated only operate on instance variables declared in the current Objective-C class, and not those inherited from superclasses. It is the responsibility of the Objective-C runtime to invoke all such methods in an object's inheritance hierarchy. The `- (id) .cxx_construct` methods are invoked by the runtime immediately after a new object instance is allocated; the `- (void) .cxx_destruct` methods are invoked immediately before the runtime deallocates an object instance.

As of this writing, only the NeXT runtime on Mac OS X 10.4 and later has support for invoking the `- (id) .cxx_construct` and `- (void) .cxx_destruct` methods.

`-fobjc-direct-dispatch`

Allow fast jumps to the message dispatcher. On Darwin this is accomplished via the comm page.

`-fobjc-exceptions`

Enable syntactic support for structured exception handling in Objective-C, similar to what is offered by C++. This option is required to use the Objective-C keywords `@try`, `@throw`, `@catch`, `@finally` and `@synchronized`. This option is available with both the GNU runtime and the NeXT runtime (but not available in conjunction with the NeXT runtime on Mac OS X 10.2 and earlier).

`-fobjc-gc`

Enable garbage collection (GC) in Objective-C and Objective-C++ programs. This option is only available with the NeXT runtime; the GNU runtime has a different garbage collection implementation that does not require special compiler flags.

`-fobjc-nilcheck`

For the NeXT runtime with version 2 of the ABI, check for a nil receiver in method invocations before doing the actual method call. This is the default and can be disabled using '`-fno-objc-nilcheck`'. Class methods and super calls are never checked for nil in this way no matter what this flag is set to. Currently this flag does nothing when the GNU runtime, or an older version of the NeXT runtime ABI, is used.

`-fobjc-std=objc1`
: Conform to the language syntax of Objective-C 1.0, the language recognized by GCC 4.0. This only affects the Objective-C additions to the C/C++ language; it does not affect conformance to C/C++ standards, which is controlled by the separate C/C++ dialect option flags. When this option is used with the Objective-C or Objective-C++ compiler, any Objective-C syntax that is not recognized by GCC 4.0 is rejected. This is useful if you need to make sure that your Objective-C code can be compiled with older versions of GCC.

`-freplace-objc-classes`
: Emit a special marker instructing `ld(1)` not to statically link in the resulting object file, and allow `dyld(1)` to load it in at run time instead. This is used in conjunction with the Fix-and-Continue debugging mode, where the object file in question may be recompiled and dynamically reloaded in the course of program execution, without the need to restart the program itself. Currently, Fix-and-Continue functionality is only available in conjunction with the NeXT runtime on Mac OS X 10.3 and later.

`-fzero-link`
: When compiling for the NeXT runtime, the compiler ordinarily replaces calls to `objc_getClass("...")` (when the name of the class is known at compile time) with static class references that get initialized at load time, which improves runtime performance. Specifying the '`-fzero-link`' flag suppresses this behavior and causes calls to `objc_getClass("...")` to be retained. This is useful in Zero-Link debugging mode, since it allows for individual class implementations to be modified during program execution. The GNU runtime currently always retains calls to `objc_get_class("...")` regardless of command-line options.

`-fno-local-ivars`
: By default instance variables in Objective-C can be accessed as if they were local variables from within the methods of the class they're declared in. This can lead to shadowing between instance variables and other variables declared either locally inside a class method or globally with the same name. Specifying the '`-fno-local-ivars`' flag disables this behavior thus avoiding variable shadowing issues.

`-fivar-visibility=`[`public|protected|private|package`]
: Set the default instance variable visibility to the specified option so that instance variables declared outside the scope of any access modifier directives default to the specified visibility.

`-gen-decls`
: Dump interface declarations for all classes seen in the source file to a file named '*`sourcename`*`.decl`'.

`-Wassign-intercept` (Objective-C and Objective-C++ only)
: Warn whenever an Objective-C assignment is being intercepted by the garbage collector.

`-Wno-protocol` (Objective-C and Objective-C++ only)
If a class is declared to implement a protocol, a warning is issued for every method in the protocol that is not implemented by the class. The default behavior is to issue a warning for every method not explicitly implemented in the class, even if a method implementation is inherited from the superclass. If you use the '`-Wno-protocol`' option, then methods inherited from the superclass are considered to be implemented, and no warning is issued for them.

`-Wselector` (Objective-C and Objective-C++ only)
Warn if multiple methods of different types for the same selector are found during compilation. The check is performed on the list of methods in the final stage of compilation. Additionally, a check is performed for each selector appearing in a `@selector(...)` expression, and a corresponding method for that selector has been found during compilation. Because these checks scan the method table only at the end of compilation, these warnings are not produced if the final stage of compilation is not reached, for example because an error is found during compilation, or because the '`-fsyntax-only`' option is being used.

`-Wstrict-selector-match` (Objective-C and Objective-C++ only)
Warn if multiple methods with differing argument and/or return types are found for a given selector when attempting to send a message using this selector to a receiver of type `id` or `Class`. When this flag is off (which is the default behavior), the compiler omits such warnings if any differences found are confined to types that share the same size and alignment.

`-Wundeclared-selector` (Objective-C and Objective-C++ only)
Warn if a `@selector(...)` expression referring to an undeclared selector is found. A selector is considered undeclared if no method with that name has been declared before the `@selector(...)` expression, either explicitly in an `@interface` or `@protocol` declaration, or implicitly in an `@implementation` section. This option always performs its checks as soon as a `@selector(...)` expression is found, while '`-Wselector`' only performs its checks in the final stage of compilation. This also enforces the coding style convention that methods and selectors must be declared before being used.

`-print-objc-runtime-info`
Generate C header describing the largest structure that is passed by value, if any.

3.7 Options to Control Diagnostic Messages Formatting

Traditionally, diagnostic messages have been formatted irrespective of the output device's aspect (e.g. its width, ...). You can use the options described below to control the formatting algorithm for diagnostic messages, e.g. how many characters per line, how often source location information should be reported. Note that some language front ends may not honor these options.

`-fmessage-length=n`

Try to format error messages so that they fit on lines of about *n* characters. If *n* is zero, then no line-wrapping is done; each error message appears on a single line. This is the default for all front ends.

Note - this option also affects the display of the '`#error`' and '`#warning`' preprocessor directives, and the '`deprecated`' function/type/variable attribute. It does not however affect the '`pragma GCC warning`' and '`pragma GCC error`' pragmas.

`-fdiagnostics-show-location=once`

Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit source location information *once*; that is, in case the message is too long to fit on a single physical line and has to be wrapped, the source location won't be emitted (as prefix) again, over and over, in subsequent continuation lines. This is the default behavior.

`-fdiagnostics-show-location=every-line`

Only meaningful in line-wrapping mode. Instructs the diagnostic messages reporter to emit the same source location information (as prefix) for physical lines that result from the process of breaking a message which is too long to fit on a single line.

`-fdiagnostics-color[=WHEN]`
`-fno-diagnostics-color`

Use color in diagnostics. *WHEN* is '`never`', '`always`', or '`auto`'. The default depends on how the compiler has been configured, it can be any of the above *WHEN* options or also '`never`' if `GCC_COLORS` environment variable isn't present in the environment, and '`auto`' otherwise. '`auto`' means to use color only when the standard error is a terminal. The forms '`-fdiagnostics-color`' and '`-fno-diagnostics-color`' are aliases for '`-fdiagnostics-color=always`' and '`-fdiagnostics-color=never`', respectively.

The colors are defined by the environment variable `GCC_COLORS`. Its value is a colon-separated list of capabilities and Select Graphic Rendition (SGR) substrings. SGR commands are interpreted by the terminal or terminal emulator. (See the section in the documentation of your text terminal for permitted values and their meanings as character attributes.) These substring values are integers in decimal representation and can be concatenated with semicolons. Common values to concatenate include '`1`' for bold, '`4`' for underline, '`5`' for blink, '`7`' for inverse, '`39`' for default foreground color, '`30`' to '`37`' for foreground colors, '`90`' to '`97`' for 16-color mode foreground colors, '`38;5;0`' to '`38;5;255`' for 88-color and 256-color modes foreground colors, '`49`' for default background color, '`40`' to '`47`' for background colors, '`100`' to '`107`' for 16-color mode background colors, and '`48;5;0`' to '`48;5;255`' for 88-color and 256-color modes background colors.

The default `GCC_COLORS` is

```
error=01;31:warning=01;35:note=01;36:range1=32:range2=34:locus=01:\
quote=01:fixit-insert=32:fixit-delete=31:\
```

```
diff-filename=01:diff-hunk=32:diff-delete=31:diff-insert=32:\
type-diff=01;32
```

where '01;31' is bold red, '01;35' is bold magenta, '01;36' is bold cyan, '32' is green, '34' is blue, '01' is bold, and '31' is red. Setting GCC_COLORS to the empty string disables colors. Supported capabilities are as follows.

error= SGR substring for error: markers.

warning= SGR substring for warning: markers.

note= SGR substring for note: markers.

range1= SGR substring for first additional range.

range2= SGR substring for second additional range.

locus= SGR substring for location information, 'file:line' or 'file:line:column' etc.

quote= SGR substring for information printed within quotes.

fixit-insert=
SGR substring for fix-it hints suggesting text to be inserted or replaced.

fixit-delete=
SGR substring for fix-it hints suggesting text to be deleted.

diff-filename=
SGR substring for filename headers within generated patches.

diff-hunk=
SGR substring for the starts of hunks within generated patches.

diff-delete=
SGR substring for deleted lines within generated patches.

diff-insert=
SGR substring for inserted lines within generated patches.

type-diff=
SGR substring for highlighting mismatching types within template arguments in the C++ frontend.

-fno-diagnostics-show-option
By default, each diagnostic emitted includes text indicating the command-line option that directly controls the diagnostic (if such an option is known to the diagnostic machinery). Specifying the '-fno-diagnostics-show-option' flag suppresses that behavior.

-fno-diagnostics-show-caret
By default, each diagnostic emitted includes the original source line and a caret '^' indicating the column. This option suppresses this information. The source line is truncated to *n* characters, if the '-fmessage-length=n' option is given. When the output is done to the terminal, the width is limited to the width given by the COLUMNS environment variable or, if not set, to the terminal width.

`-fno-diagnostics-show-labels`

By default, when printing source code (via '`-fdiagnostics-show-caret`'), diagnostics can label ranges of source code with pertinent information, such as the types of expressions:

```
printf ("foo %s bar", long_i + long_j);
             ~~       ~~~~~~~~~~~~~~~
              |              |
              char *         long int
```

This option suppresses the printing of these labels (in the example above, the vertical bars and the "char *" and "long int" text).

`-fno-diagnostics-show-line-numbers`

By default, when printing source code (via '`-fdiagnostics-show-caret`'), a left margin is printed, showing line numbers. This option suppresses this left margin.

`-fdiagnostics-minimum-margin-width=`*`width`*

This option controls the minimum width of the left margin printed by '`-fdiagnostics-show-line-numbers`'. It defaults to 6.

`-fdiagnostics-parseable-fixits`

Emit fix-it hints in a machine-parseable format, suitable for consumption by IDEs. For each fix-it, a line will be printed after the relevant diagnostic, starting with the string "fix-it:". For example:

```
fix-it:"test.c":{45:3-45:21}:"gtk_widget_show_all"
```

The location is expressed as a half-open range, expressed as a count of bytes, starting at byte 1 for the initial column. In the above example, bytes 3 through 20 of line 45 of "test.c" are to be replaced with the given string:

```
00000000011111111112222222222
12345678901234567890123456789
  gtk_widget_showall (dlg);
  ^^^^^^^^^^^^^^^^^^
  gtk_widget_show_all
```

The filename and replacement string escape backslash as "`\\`", tab as "`\t`", newline as "`\n`", double quotes as "`\"`", non-printable characters as octal (e.g. vertical tab as "`\013`").

An empty replacement string indicates that the given range is to be removed. An empty range (e.g. "45:3-45:3") indicates that the string is to be inserted at the given position.

`-fdiagnostics-generate-patch`

Print fix-it hints to stderr in unified diff format, after any diagnostics are printed. For example:

```
--- test.c
+++ test.c
@ -42,5 +42,5 @

 void show_cb(GtkDialog *dlg)
 {
-  gtk_widget_showall(dlg);
+  gtk_widget_show_all(dlg);
```

```
}
```

The diff may or may not be colorized, following the same rules as for diagnostics (see '-fdiagnostics-color').

-fdiagnostics-show-template-tree

In the C++ frontend, when printing diagnostics showing mismatching template types, such as:

```
could not convert 'std::map<int, std::vector<double> >()'
  from 'map<[...],vector<double>>' to 'map<[...],vector<float>>
```

the '-fdiagnostics-show-template-tree' flag enables printing a tree-like structure showing the common and differing parts of the types, such as:

```
map<
  [...],
  vector<
    [double != float]>>
```

The parts that differ are highlighted with color ("double" and "float" in this case).

-fno-elide-type

By default when the C++ frontend prints diagnostics showing mismatching template types, common parts of the types are printed as "[...]" to simplify the error message. For example:

```
could not convert 'std::map<int, std::vector<double> >()'
  from 'map<[...],vector<double>>' to 'map<[...],vector<float>>
```

Specifying the '-fno-elide-type' flag suppresses that behavior. This flag also affects the output of the '-fdiagnostics-show-template-tree' flag.

-fno-show-column

Do not print column numbers in diagnostics. This may be necessary if diagnostics are being scanned by a program that does not understand the column numbers, such as dejagnu.

-fdiagnostics-format=*FORMAT*

Select a different format for printing diagnostics. *FORMAT* is 'text' or 'json'. The default is 'text'.

The 'json' format consists of a top-level JSON array containing JSON objects representing the diagnostics.

The JSON is emitted as one line, without formatting; the examples below have been formatted for clarity.

Diagnostics can have child diagnostics. For example, this error and note:

```
misleading-indentation.c:15:3: warning: this 'if' clause does not
  guard... [-Wmisleading-indentation]
   15 |   if (flag)
      |   ^~
misleading-indentation.c:17:5: note: ...this statement, but the latter
  is misleadingly indented as if it were guarded by the 'if'
   17 |     y = 2;
      |     ^
```

might be printed in JSON form (after formatting) like this:

```
[
    {
        "kind": "warning",
        "locations": [
            {
                "caret": {
                    "column": 3,
                    "file": "misleading-indentation.c",
                    "line": 15
                },
                "finish": {
                    "column": 4,
                    "file": "misleading-indentation.c",
                    "line": 15
                }
            }
        ],
        "message": "this \u2018if\u2019 clause does not guard...",
        "option": "-Wmisleading-indentation",
        "children": [
            {
                "kind": "note",
                "locations": [
                    {
                        "caret": {
                            "column": 5,
                            "file": "misleading-indentation.c",
                            "line": 17
                        }
                    }
                ],
                "message": "...this statement, but the latter is ..."
            }
        ]
    },
    ...
]
```

where the `note` is a child of the `warning`.

A diagnostic has a `kind`. If this is `warning`, then there is an `option` key describing the command-line option controlling the warning.

A diagnostic can contain zero or more locations. Each location has up to three positions within it: a `caret` position and optional `start` and `finish` positions. A location can also have an optional `label` string. For example, this error:

```
bad-binary-ops.c:64:23: error: invalid operands to binary + (have 'S' {aka
   'struct s'} and 'T' {aka 'struct t'})
   64 |   return callee_4a () + callee_4b ();
      |          ~~~~~~~~~~~~ ^ ~~~~~~~~~~~~
      |          |              |
      |          |              T {aka struct t}
      |          S {aka struct s}
```

has three locations. Its primary location is at the "+" token at column 23. It has two secondary locations, describing the left and right-hand sides of the expression, which have labels. It might be printed in JSON form as:

```
    {
        "children": [],
```

```
        "kind": "error",
        "locations": [
            {
                "caret": {
                    "column": 23, "file": "bad-binary-ops.c", "line": 64
                }
            },
            {
                "caret": {
                    "column": 10, "file": "bad-binary-ops.c", "line": 64
                },
                "finish": {
                    "column": 21, "file": "bad-binary-ops.c", "line": 64
                },
                "label": "S {aka struct s}"
            },
            {
                "caret": {
                    "column": 25, "file": "bad-binary-ops.c", "line": 64
                },
                "finish": {
                    "column": 36, "file": "bad-binary-ops.c", "line": 64
                },
                "label": "T {aka struct t}"
            }
        ],
        "message": "invalid operands to binary + ..."
    }
```

If a diagnostic contains fix-it hints, it has a `fixits` array, consisting of half-open intervals, similar to the output of '`-fdiagnostics-parseable-fixits`'. For example, this diagnostic with a replacement fix-it hint:

```
demo.c:8:15: error: 'struct s' has no member named 'colour'; did you
  mean 'color'?
    8 |   return ptr->colour;
      |                ^~~~~~
      |                color
```

might be printed in JSON form as:

```
    {
        "children": [],
        "fixits": [
            {
                "next": {
                    "column": 21,
                    "file": "demo.c",
                    "line": 8
                },
                "start": {
                    "column": 15,
                    "file": "demo.c",
                    "line": 8
                },
                "string": "color"
            }
        ],
        "kind": "error",
        "locations": [
            {
```

```
                    "caret": {
                        "column": 15,
                        "file": "demo.c",
                        "line": 8
                    },
                    "finish": {
                        "column": 20,
                        "file": "demo.c",
                        "line": 8
                    }
                }
            ],
            "message": "\u2018struct s\u2019 has no member named ..."
        }
```

where the fix-it hint suggests replacing the text from `start` up to but not including `next` with `string`'s value. Deletions are expressed via an empty value for `string`, insertions by having `start` equal `next`.

3.8 Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

The following language-independent options do not enable specific warnings but control the kinds of diagnostics produced by GCC.

`-fsyntax-only`
: Check the code for syntax errors, but don't do anything beyond that.

`-fmax-errors=`*n*
: Limits the maximum number of error messages to *n*, at which point GCC bails out rather than attempting to continue processing the source code. If *n* is 0 (the default), there is no limit on the number of error messages produced. If '`-Wfatal-errors`' is also specified, then '`-Wfatal-errors`' takes precedence over this option.

`-w`
: Inhibit all warning messages.

`-Werror`
: Make all warnings into errors.

`-Werror=`
: Make the specified warning into an error. The specifier for a warning is appended; for example '`-Werror=switch`' turns the warnings controlled by '`-Wswitch`' into errors. This switch takes a negative form, to be used to negate '`-Werror`' for specific warnings; for example '`-Wno-error=switch`' makes '`-Wswitch`' warnings not be errors, even when '`-Werror`' is in effect.

  The warning message for each controllable warning includes the option that controls the warning. That option can then be used with '`-Werror=`' and '`-Wno-error=`' as described above. (Printing of the option in the warning message can be disabled using the '`-fno-diagnostics-show-option`' flag.)

  Note that specifying '`-Werror=`'*foo* automatically implies '`-W`'*foo*. However, '`-Wno-error=`'*foo* does not imply anything.

`-Wfatal-errors`

This option causes the compiler to abort compilation on the first error occurred rather than trying to keep going and printing further error messages.

You can request many specific warnings with options beginning with '`-W`', for example '`-Wimplicit`' to request warnings on implicit declarations. Each of these specific warning options also has a negative form beginning '`-Wno-`' to turn off warnings; for example, '`-Wno-implicit`'. This manual lists only one of the two forms, whichever is not the default. For further language-specific options also refer to Section 3.5 [C++ Dialect Options], page 44 and Section 3.6 [Objective-C and Objective-C++ Dialect Options], page 60.

Some options, such as '`-Wall`' and '`-Wextra`', turn on other options, such as '`-Wunused`', which may turn on further options, such as '`-Wunused-value`'. The combined effect of positive and negative forms is that more specific options have priority over less specific ones, independently of their position in the command-line. For options of the same specificity, the last one takes effect. Options enabled or disabled via pragmas (see Section 6.62.12 [Diagnostic Pragmas], page 806) take effect as if they appeared at the end of the command-line.

When an unrecognized warning option is requested (e.g., '`-Wunknown-warning`'), GCC emits a diagnostic stating that the option is not recognized. However, if the '`-Wno-`' form is used, the behavior is slightly different: no diagnostic is produced for '`-Wno-unknown-warning`' unless other diagnostics are being produced. This allows the use of new '`-Wno-`' options with old compilers, but if something goes wrong, the compiler warns that an unrecognized option is present.

`-Wpedantic`
`-pedantic`

Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any '`-std`' option used.

Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few require '`-ansi`' or a '`-std`' option specifying the required version of ISO C). However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are rejected.

'`-Wpedantic`' does not cause warning messages for use of the alternate keywords whose names begin and end with '`__`'. Pedantic warnings are also disabled in the expression that follows `__extension__`. However, only system header files should use these escape routes; application programs should avoid them. See Section 6.48 [Alternate Keywords], page 624.

Some users try to use '`-Wpedantic`' to check programs for strict ISO C conformance. They soon find that it does not do quite what they want: it finds some non-ISO practices, but not all—only those for which ISO C *requires* a diagnostic, and some others for which diagnostics have been added.

A feature to report any failure to conform to ISO C might be useful in some instances, but would require considerable additional work and would be quite

different from '-Wpedantic'. We don't have plans to support such a feature in the near future.

Where the standard specified with '-std' represents a GNU extended dialect of C, such as 'gnu90' or 'gnu99', there is a corresponding *base standard*, the version of ISO C on which the GNU extended dialect is based. Warnings from '-Wpedantic' are given where they are required by the base standard. (It does not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

-pedantic-errors
: Give an error whenever the *base standard* (see '-Wpedantic') requires a diagnostic, in some cases where there is undefined behavior at compile-time and in some other cases that do not prevent compilation of programs that are valid according to the standard. This is not equivalent to '-Werror=pedantic', since there are errors enabled by this option and not enabled by the latter and vice versa.

-Wall
: This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings described in Section 3.5 [C++ Dialect Options], page 44 and Section 3.6 [Objective-C and Objective-C++ Dialect Options], page 60.

  '-Wall' turns on the following warning flags:

  -Waddress
  -Warray-bounds=1 (only with '-O2')
  -Wbool-compare
  -Wbool-operation
  -Wc++11-compat -Wc++14-compat
  -Wcatch-value (C++ and Objective-C++ only)
  -Wchar-subscripts
  -Wcomment
  -Wduplicate-decl-specifier (C and Objective-C only)
  -Wenum-compare (in C/ObjC; this is on by default in C++)
  -Wformat
  -Wint-in-bool-context
  -Wimplicit (C and Objective-C only)
  -Wimplicit-int (C and Objective-C only)
  -Wimplicit-function-declaration (C and Objective-C only)
  -Winit-self (only for C++)
  -Wlogical-not-parentheses
  -Wmain (only for C/ObjC and unless '-ffreestanding')
  -Wmaybe-uninitialized
  -Wmemset-elt-size
  -Wmemset-transposed-args
  -Wmisleading-indentation (only for C/C++)
  -Wmissing-attributes
  -Wmissing-braces (only for C/ObjC)
  -Wmultistatement-macros
  -Wnarrowing (only for C++)
  -Wnonnull
  -Wnonnull-compare

-Wopenmp-simd
-Wparentheses
-Wpessimizing-move (only for C++)
-Wpointer-sign
-Wreorder
-Wrestrict
-Wreturn-type
-Wsequence-point
-Wsign-compare (only in C++)
-Wsizeof-pointer-div
-Wsizeof-pointer-memaccess
-Wstrict-aliasing
-Wstrict-overflow=1
-Wswitch
-Wtautological-compare
-Wtrigraphs
-Wuninitialized
-Wunknown-pragmas
-Wunused-function
-Wunused-label
-Wunused-value
-Wunused-variable
-Wvolatile-register-var

Note that some warning flags are not implied by '-Wall'. Some of them warn about constructions that users generally do not consider questionable, but which occasionally you might wish to check for; others warn about constructions that are necessary or hard to avoid in some cases, and there is no simple way to modify the code to suppress the warning. Some of them are enabled by '-Wextra' but many of them must be enabled individually.

-Wextra This enables some extra warning flags that are not enabled by '-Wall'. (This option used to be called '-W'. The older name is still supported, but the newer name is more descriptive.)

-Wclobbered
-Wcast-function-type
-Wdeprecated-copy (C++ only)
-Wempty-body
-Wignored-qualifiers
-Wimplicit-fallthrough=3
-Wmissing-field-initializers
-Wmissing-parameter-type (C only)
-Wold-style-declaration (C only)
-Woverride-init
-Wsign-compare (C only)
-Wredundant-move (only for C++)
-Wtype-limits
-Wuninitialized
-Wshift-negative-value (in C++03 and in C99 and newer)
-Wunused-parameter (only with '-Wunused' or '-Wall')
-Wunused-but-set-parameter (only with '-Wunused' or '-Wall')

The option '-Wextra' also prints warning messages for the following cases:

- A pointer is compared against integer zero with <, <=, >, or >=.
- (C++ only) An enumerator and a non-enumerator both appear in a conditional expression.

- (C++ only) Ambiguous virtual bases.
- (C++ only) Subscripting an array that has been declared `register`.
- (C++ only) Taking the address of a variable that has been declared `register`.
- (C++ only) A base class is not initialized in the copy constructor of a derived class.

`-Wchar-subscripts`
Warn if an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines. This warning is enabled by '`-Wall`'.

`-Wno-coverage-mismatch`
Warn if feedback profiles do not match when using the '`-fprofile-use`' option. If a source file is changed between compiling with '`-fprofile-generate`' and with '`-fprofile-use`', the files with the profile feedback can fail to match the source file and GCC cannot use the profile feedback information. By default, this warning is enabled and is treated as an error. '`-Wno-coverage-mismatch`' can be used to disable the warning or '`-Wno-error=coverage-mismatch`' can be used to disable the error. Disabling the error for this warning can result in poorly optimized code and is useful only in the case of very minor changes such as bug fixes to an existing code-base. Completely disabling the warning is not recommended.

`-Wno-cpp` (C, Objective-C, C++, Objective-C++ and Fortran only)
Suppress warning messages emitted by `#warning` directives.

`-Wdouble-promotion` (C, C++, Objective-C and Objective-C++ only)
Give a warning when a value of type `float` is implicitly promoted to `double`. CPUs with a 32-bit "single-precision" floating-point unit implement `float` in hardware, but emulate `double` in software. On such a machine, doing computations using `double` values is much more expensive because of the overhead required for software emulation.

It is easy to accidentally do computations with `double` because floating-point literals are implicitly of type `double`. For example, in:

```
float area(float radius)
{
   return 3.14159 * radius * radius;
}
```

the compiler performs the entire computation with `double` because the floating-point literal is a `double`.

`-Wduplicate-decl-specifier` (C and Objective-C only)
Warn if a declaration has duplicate `const`, `volatile`, `restrict` or `_Atomic` specifier. This warning is enabled by '`-Wall`'.

`-Wformat`
`-Wformat=`*n*
Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions

specified in the format string make sense. This includes standard functions, and others specified by format attributes (see Section 6.33 [Function Attributes], page 477), in the `printf`, `scanf`, `strftime` and `strfmon` (an X/Open extension, not in the C standard) families (or other target-specific families). Which functions are checked without format attributes having been specified depends on the standard version selected, and such checks of functions without the attribute specified are disabled by '`-ffreestanding`' or '`-fno-builtin`'.

The formats are checked against the format features supported by GNU libc version 2.2. These include all ISO C90 and C99 features, as well as features from the Single Unix Specification and some BSD and GNU extensions. Other library implementations may not support all these features; GCC does not support warning about features that go beyond a particular library's limitations. However, if '`-Wpedantic`' is used with '`-Wformat`', warnings are given about format features not in the selected standard version (but not for `strfmon` formats, since those are not in any version of the C standard). See Section 3.4 [Options Controlling C Dialect], page 37.

`-Wformat=1`
`-Wformat` Option '`-Wformat`' is equivalent to '`-Wformat=1`', and '`-Wno-format`' is equivalent to '`-Wformat=0`'. Since '`-Wformat`' also checks for null format arguments for several functions, '`-Wformat`' also implies '`-Wnonnull`'. Some aspects of this level of format checking can be disabled by the options: '`-Wno-format-contains-nul`', '`-Wno-format-extra-args`', and '`-Wno-format-zero-length`'. '`-Wformat`' is enabled by '`-Wall`'.

`-Wno-format-contains-nul`
If '`-Wformat`' is specified, do not warn about format strings that contain NUL bytes.

`-Wno-format-extra-args`
If '`-Wformat`' is specified, do not warn about excess arguments to a `printf` or `scanf` format function. The C standard specifies that such arguments are ignored.

Where the unused arguments lie between used arguments that are specified with '`$`' operand number specifications, normally warnings are still given, since the implementation could not know what type to pass to `va_arg` to skip the unused arguments. However, in the case of `scanf` formats, this option suppresses the warning if the unused arguments are all pointers, since the Single Unix Specification says that such unused arguments are allowed.

`-Wformat-overflow`
`-Wformat-overflow=`*`level`*
Warn about calls to formatted input/output functions such as `sprintf` and `vsprintf` that might overflow the destination buffer. When the exact number of bytes written by a format directive cannot be determined at compile-time it is estimated based on heuristics that depend on the *level* argument and on optimization.

While enabling optimization will in most cases improve the accuracy of the warning, it may also result in false positives.

`-Wformat-overflow`
`-Wformat-overflow=1`

Level *1* of '`-Wformat-overflow`' enabled by '`-Wformat`' employs a conservative approach that warns only about calls that most likely overflow the buffer. At this level, numeric arguments to format directives with unknown values are assumed to have the value of one, and strings of unknown length to be empty. Numeric arguments that are known to be bounded to a subrange of their type, or string arguments whose output is bounded either by their directive's precision or by a finite set of string literals, are assumed to take on the value within the range that results in the most bytes on output. For example, the call to `sprintf` below is diagnosed because even with both *a* and *b* equal to zero, the terminating NUL character (`'\0'`) appended by the function to the destination buffer will be written past its end. Increasing the size of the buffer by a single byte is sufficient to avoid the warning, though it may not be sufficient to avoid the overflow.

```
void f (int a, int b)
{
  char buf [13];
  sprintf (buf, "a = %i, b = %i\n", a, b);
}
```

`-Wformat-overflow=2`

Level *2* warns also about calls that might overflow the destination buffer given an argument of sufficient length or magnitude. At level *2*, unknown numeric arguments are assumed to have the minimum representable value for signed types with a precision greater than 1, and the maximum representable value otherwise. Unknown string arguments whose length cannot be assumed to be bounded either by the directive's precision, or by a finite set of string literals they may evaluate to, or the character array they may point to, are assumed to be 1 character long.

At level *2*, the call in the example above is again diagnosed, but this time because with *a* equal to a 32-bit `INT_MIN` the first `%i` directive will write some of its digits beyond the end of the destination buffer. To make the call safe regardless of the values of the two variables, the size of the destination buffer must be increased to at least 34 bytes. GCC includes the minimum size of

the buffer in an informational note following the warning.

An alternative to increasing the size of the destination buffer is to constrain the range of formatted values. The maximum length of string arguments can be bounded by specifying the precision in the format directive. When numeric arguments of format directives can be assumed to be bounded by less than the precision of their type, choosing an appropriate length modifier to the format specifier will reduce the required buffer size. For example, if *a* and *b* in the example above can be assumed to be within the precision of the `short int` type then using either the `%hi` format directive or casting the argument to `short` reduces the maximum required size of the buffer to 24 bytes.

```
void f (int a, int b)
{
  char buf [23];
  sprintf (buf, "a = %hi, b = %i\n", a, (short)b);
}
```

`-Wno-format-zero-length`
: If '`-Wformat`' is specified, do not warn about zero-length formats. The C standard specifies that zero-length formats are allowed.

`-Wformat=2`
: Enable '`-Wformat`' plus additional format checks. Currently equivalent to '`-Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k`'.

`-Wformat-nonliteral`
: If '`-Wformat`' is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a `va_list`.

`-Wformat-security`
: If '`-Wformat`' is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to `printf` and `scanf` functions where the format string is not a string literal and there are no format arguments, as in `printf (foo);`. This may be a security hole if the format string came from untrusted input and contains '`%n`'. (This is currently a subset of what '`-Wformat-nonliteral`' warns about, but in future warnings may be added to '`-Wformat-security`' that are not included in '`-Wformat-nonliteral`'.)

`-Wformat-signedness`
: If '`-Wformat`' is specified, also warn if the format string requires an unsigned argument and the argument is signed and vice versa.

`-Wformat-truncation`
`-Wformat-truncation=`*level*

Warn about calls to formatted input/output functions such as `snprintf` and `vsnprintf` that might result in output truncation. When the exact number of bytes written by a format directive cannot be determined at compile-time it is estimated based on heuristics that depend on the *level* argument and on optimization. While enabling optimization will in most cases improve the accuracy of the warning, it may also result in false positives. Except as noted otherwise, the option uses the same logic '`-Wformat-overflow`'.

`-Wformat-truncation`
`-Wformat-truncation=1`

Level *1* of '`-Wformat-truncation`' enabled by '`-Wformat`' employs a conservative approach that warns only about calls to bounded functions whose return value is unused and that will most likely result in output truncation.

`-Wformat-truncation=2`

Level *2* warns also about calls to bounded functions whose return value is used and that might result in truncation given an argument of sufficient length or magnitude.

`-Wformat-y2k`

If '`-Wformat`' is specified, also warn about `strftime` formats that may yield only a two-digit year.

`-Wnonnull`

Warn about passing a null pointer for arguments marked as requiring a non-null value by the `nonnull` function attribute.

'`-Wnonnull`' is included in '`-Wall`' and '`-Wformat`'. It can be disabled with the '`-Wno-nonnull`' option.

`-Wnonnull-compare`

Warn when comparing an argument marked with the `nonnull` function attribute against null inside the function.

'`-Wnonnull-compare`' is included in '`-Wall`'. It can be disabled with the '`-Wno-nonnull-compare`' option.

`-Wnull-dereference`

Warn if the compiler detects paths that trigger erroneous or undefined behavior due to dereferencing a null pointer. This option is only active when '`-fdelete-null-pointer-checks`' is active, which is enabled by optimizations in most targets. The precision of the warnings depends on the optimization options used.

`-Winit-self` (C, C++, Objective-C and Objective-C++ only)

Warn about uninitialized variables that are initialized with themselves. Note this option can only be used with the '`-Wuninitialized`' option.

For example, GCC warns about `i` being uninitialized in the following snippet only when '`-Winit-self`' has been specified:

```
int f()
{
  int i = i;
  return i;
}
```

This warning is enabled by '`-Wall`' in C++.

`-Wimplicit-int` (C and Objective-C only)

Warn when a declaration does not specify a type. This warning is enabled by '`-Wall`'.

`-Wimplicit-function-declaration` (C and Objective-C only)

Give a warning whenever a function is used before being declared. In C99 mode ('`-std=c99`' or '`-std=gnu99`'), this warning is enabled by default and it is made into an error by '`-pedantic-errors`'. This warning is also enabled by '`-Wall`'.

`-Wimplicit` (C and Objective-C only)

Same as '`-Wimplicit-int`' and '`-Wimplicit-function-declaration`'. This warning is enabled by '`-Wall`'.

`-Wimplicit-fallthrough`

'`-Wimplicit-fallthrough`' is the same as '`-Wimplicit-fallthrough=3`' and '`-Wno-implicit-fallthrough`' is the same as '`-Wimplicit-fallthrough=0`'.

`-Wimplicit-fallthrough=`*n*

Warn when a switch case falls through. For example:

```
switch (cond)
  {
  case 1:
    a = 1;
    break;
  case 2:
    a = 2;
  case 3:
    a = 3;
    break;
  }
```

This warning does not warn when the last statement of a case cannot fall through, e.g. when there is a return statement or a call to function declared with the noreturn attribute. '`-Wimplicit-fallthrough=`' also takes into account control flow statements, such as ifs, and only warns when appropriate. E.g.

```
switch (cond)
  {
  case 1:
    if (i > 3) {
      bar (5);
      break;
    } else if (i < 1) {
      bar (0);
    } else
      return;
  default:
    ...
  }
```

Since there are occasions where a switch case fall through is desirable, GCC provides an attribute, `__attribute__ ((fallthrough))`, that is to be used along with a null statement to suppress this warning that would normally occur:

```
switch (cond)
  {
  case 1:
    bar (0);
    __attribute__ ((fallthrough));
  default:
    ...
  }
```

C++17 provides a standard way to suppress the '`-Wimplicit-fallthrough`' warning using `[[fallthrough]];` instead of the GNU attribute. In C++11 or C++14 users can use `[[gnu::fallthrough]];`, which is a GNU extension. Instead of these attributes, it is also possible to add a fallthrough comment to silence the warning. The whole body of the C or C++ style comment should match the given regular expressions listed below. The option argument *n* specifies what kind of comments are accepted:

- '`-Wimplicit-fallthrough=0`' disables the warning altogether.
- '`-Wimplicit-fallthrough=1`' matches `.*` regular expression, any comment is used as fallthrough comment.
- '`-Wimplicit-fallthrough=2`' case insensitively matches `.*falls?[ \t-]*thr(ough|u).*` regular expression.
- '`-Wimplicit-fallthrough=3`' case sensitively matches one of the following regular expressions:
  - `-fallthrough`
  - `@fallthrough@`
  - `lint -fallthrough[ \t]*`
  - `[ \t.!]*(ELSE,? |INTENTIONAL(LY)? )?`
    `FALL(S | |-)?THR(OUGH|U)[ \t.!]*(-[^\n\r]*)?`
  - `[ \t.!]*(Else,? |Intentional(ly)? )?`
    `Fall((s | |-)[Tt]|t)hr(ough|u)[ \t.!]*(-[^\n\r]*)?`
  - `[ \t.!]*([Ee]lse,? |[Ii]ntentional(ly)? )?`
    `fall(s | |-)?thr(ough|u)[ \t.!]*(-[^\n\r]*)?`

- '-Wimplicit-fallthrough=4' case sensitively matches one of the following regular expressions:
  - -fallthrough
  - @fallthrough@
  - lint -fallthrough[\t]*
  - [\t]*FALLTHR(OUGH|U)[\t]*
- '-Wimplicit-fallthrough=5' doesn't recognize any comments as fallthrough comments, only attributes disable the warning.

The comment needs to be followed after optional whitespace and other comments by case or default keywords or by a user label that precedes some case or default label.

```
switch (cond)
  {
  case 1:
    bar (0);
    /* FALLTHRU */
  default:
    ...
  }
```

The '-Wimplicit-fallthrough=3' warning is enabled by '-Wextra'.

-Wif-not-aligned (C, C++, Objective-C and Objective-C++ only)
Control if warning triggered by the warn_if_not_aligned attribute should be issued. This is enabled by default. Use '-Wno-if-not-aligned' to disable it.

-Wignored-qualifiers (C and C++ only)
Warn if the return type of a function has a type qualifier such as const. For ISO C such a type qualifier has no effect, since the value returned by a function is not an lvalue. For C++, the warning is only emitted for scalar types or void. ISO C prohibits qualified void return types on function definitions, so such return types always receive a warning even without this option.

This warning is also enabled by '-Wextra'.

-Wignored-attributes (C and C++ only)
Warn when an attribute is ignored. This is different from the '-Wattributes' option in that it warns whenever the compiler decides to drop an attribute, not that the attribute is either unknown, used in a wrong place, etc. This warning is enabled by default.

-Wmain Warn if the type of main is suspicious. main should be a function with external linkage, returning int, taking either zero arguments, two, or three arguments of appropriate types. This warning is enabled by default in C++ and is enabled by either '-Wall' or '-Wpedantic'.

-Wmisleading-indentation (C and C++ only)
Warn when the indentation of the code does not reflect the block structure. Specifically, a warning is issued for if, else, while, and for clauses with a guarded statement that does not use braces, followed by an unguarded statement with the same indentation.

In the following example, the call to "bar" is misleadingly indented as if it were guarded by the "if" conditional.

```
if (some_condition ())
  foo ();
  bar ();  /* Gotcha: this is not guarded by the "if".  */
```

In the case of mixed tabs and spaces, the warning uses the '`-ftabstop=`' option to determine if the statements line up (defaulting to 8).

The warning is not issued for code involving multiline preprocessor logic such as the following example.

```
  if (flagA)
    foo (0);
#if SOME_CONDITION_THAT_DOES_NOT_HOLD
  if (flagB)
#endif
    foo (1);
```

The warning is not issued after a `#line` directive, since this typically indicates autogenerated code, and no assumptions can be made about the layout of the file that the directive references.

This warning is enabled by '`-Wall`' in C and C++.

`-Wmissing-attributes`

Warn when a declaration of a function is missing one or more attributes that a related function is declared with and whose absence may adversely affect the correctness or efficiency of generated code. For example, the warning is issued for declarations of aliases that use attributes to specify less restrictive requirements than those of their targets. This typically represents a potential optimization opportunity. By contrast, the '`-Wattribute-alias=2`' option controls warnings issued when the alias is more restrictive than the target, which could lead to incorrect code generation. Attributes considered include `alloc_align`, `alloc_size`, `cold`, `const`, `hot`, `leaf`, `malloc`, `nonnull`, `noreturn`, `nothrow`, `pure`, `returns_nonnull`, and `returns_twice`.

In C++, the warning is issued when an explicit specialization of a primary template declared with attribute `alloc_align`, `alloc_size`, `assume_aligned`, `format`, `format_arg`, `malloc`, or `nonnull` is declared without it. Attributes `deprecated`, `error`, and `warning` suppress the warning. (see Section 6.33 [Function Attributes], page 477).

You can use the `copy` attribute to apply the same set of attributes to a declaration as that on another declaration without explicitly enumerating the attributes. This attribute can be applied to declarations of functions (see Section 6.33.1 [Common Function Attributes], page 478), variables (see Section 6.34.1 [Common Variable Attributes], page 536), or types (see Section 6.35.1 [Common Type Attributes], page 549).

'`-Wmissing-attributes`' is enabled by '`-Wall`'.

For example, since the declaration of the primary function template below makes use of both attribute `malloc` and `alloc_size` the declaration of the explicit specialization of the template is diagnosed because it is missing one of the attributes.

```
template <class T>
T* __attribute__ ((malloc, alloc_size (1)))
allocate (size_t);

template <>
void* __attribute__ ((malloc))   // missing alloc_size
allocate<void> (size_t);
```

`-Wmissing-braces`

Warn if an aggregate or union initializer is not fully bracketed. In the following example, the initializer for `a` is not fully bracketed, but that for `b` is fully bracketed. This warning is enabled by '`-Wall`' in C.

```
int a[2][2] = { 0, 1, 2, 3 };
int b[2][2] = { { 0, 1 }, { 2, 3 } };
```

This warning is enabled by '`-Wall`'.

`-Wmissing-include-dirs` (C, C++, Objective-C and Objective-C++ only)

Warn if a user-supplied include directory does not exist.

`-Wmissing-profile`

Warn if feedback profiles are missing when using the '`-fprofile-use`' option. This option diagnoses those cases where a new function or a new file is added to the user code between compiling with '`-fprofile-generate`' and with '`-fprofile-use`', without regenerating the profiles. In these cases, the profile feedback data files do not contain any profile feedback information for the newly added function or file respectively. Also, in the case when profile count data (.gcda) files are removed, GCC cannot use any profile feedback information. In all these cases, warnings are issued to inform the user that a profile generation step is due. '`-Wno-missing-profile`' can be used to disable the warning. Ignoring the warning can result in poorly optimized code. Completely disabling the warning is not recommended and should be done only when non-existent profile data is justified.

`-Wmultistatement-macros`

Warn about unsafe multiple statement macros that appear to be guarded by a clause such as `if`, `else`, `for`, `switch`, or `while`, in which only the first statement is actually guarded after the macro is expanded.

For example:

```
#define DOIT x++; y++
if (c)
  DOIT;
```

will increment `y` unconditionally, not just when `c` holds. The can usually be fixed by wrapping the macro in a do-while loop:

```
#define DOIT do { x++; y++; } while (0)
if (c)
  DOIT;
```

This warning is enabled by '`-Wall`' in C and C++.

`-Wparentheses`

Warn if parentheses are omitted in certain contexts, such as when there is an assignment in a context where a truth value is expected, or when operators are nested whose precedence people often get confused about.

Also warn if a comparison like `x<=y<=z` appears; this is equivalent to `(x<=y ? 1 : 0) <= z`, which is a different interpretation from that of ordinary mathematical notation.

Also warn for dangerous uses of the GNU extension to `?:` with omitted middle operand. When the condition in the `?:` operator is a boolean expression, the omitted value is always 1. Often programmers expect it to be a value computed inside the conditional expression instead.

For C++ this also warns for some cases of unnecessary parentheses in declarations, which can indicate an attempt at a function call instead of a declaration:

```
{
  // Declares a local variable called mymutex.
  std::unique_lock<std::mutex> (mymutex);
  // User meant std::unique_lock<std::mutex> lock (mymutex);
}
```

This warning is enabled by '`-Wall`'.

`-Wsequence-point`

Warn about code that may have undefined semantics because of violations of sequence point rules in the C and C++ standards.

The C and C++ standards define the order in which expressions in a C/C++ program are evaluated in terms of *sequence points*, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a `&&`, `||`, `? :` or `,` (comma) operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C and C++ standards specify that "Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.". If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

Examples of code with undefined behavior are `a = a++;`, `a[n] = b[n++]` and `a[i++] = i;`. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs.

The C++17 standard will define the order of evaluation of operands in more cases: in particular it requires that the right-hand side of an assignment be

evaluated before the left-hand side, so the above examples are no longer undefined. But this warning will still warn about them, to help people avoid writing code that is undefined in C and earlier revisions of C++.

The standard is worded confusingly, therefore there is some debate over the precise meaning of the sequence point rules in subtle cases. Links to discussions of the problem, including proposed formal definitions, may be found on the GCC readings page, at `http://gcc.gnu.org/readings.html`.

This warning is enabled by '`-Wall`' for C and C++.

`-Wno-return-local-addr`

Do not warn about returning a pointer (or in C++, a reference) to a variable that goes out of scope after the function returns.

`-Wreturn-type`

Warn whenever a function is defined with a return type that defaults to `int`. Also warn about any `return` statement with no return value in a function whose return type is not `void` (falling off the end of the function body is considered returning without a value).

For C only, warn about a `return` statement with an expression in a function whose return type is `void`, unless the expression type is also `void`. As a GNU extension, the latter case is accepted without a warning unless '`-Wpedantic`' is used. Attempting to use the return value of a non-`void` function other than `main` that flows off the end by reaching the closing curly brace that terminates the function is undefined.

Unlike in C, in C++, flowing off the end of a non-`void` function other than `main` results in undefined behavior even when the value of the function is not used.

This warning is enabled by default in C++ and by '`-Wall`' otherwise.

`-Wshift-count-negative`

Warn if shift count is negative. This warning is enabled by default.

`-Wshift-count-overflow`

Warn if shift count >= width of type. This warning is enabled by default.

`-Wshift-negative-value`

Warn if left shifting a negative value. This warning is enabled by '`-Wextra`' in C99 and C++11 modes (and newer).

`-Wshift-overflow`
`-Wshift-overflow=`*n*

Warn about left shift overflows. This warning is enabled by default in C99 and C++11 modes (and newer).

`-Wshift-overflow=1`

This is the warning level of '`-Wshift-overflow`' and is enabled by default in C99 and C++11 modes (and newer). This warning level does not warn about left-shifting 1 into the sign bit. (However, in C, such an overflow is still rejected in contexts where an integer constant expression is required.) No warning is emitted in C++2A mode (and newer), as signed left shifts always wrap.

`-Wshift-overflow=2`
This warning level also warns about left-shifting 1 into the sign bit, unless C++14 mode (or newer) is active.

`-Wswitch` Warn whenever a `switch` statement has an index of enumerated type and lacks a `case` for one or more of the named codes of that enumeration. (The presence of a `default` label prevents this warning.) `case` labels outside the enumeration range also provoke warnings when this option is used (even if there is a `default` label). This warning is enabled by '`-Wall`'.

`-Wswitch-default`
Warn whenever a `switch` statement does not have a `default` case.

`-Wswitch-enum`
Warn whenever a `switch` statement has an index of enumerated type and lacks a `case` for one or more of the named codes of that enumeration. `case` labels outside the enumeration range also provoke warnings when this option is used. The only difference between '`-Wswitch`' and this option is that this option gives a warning about an omitted enumeration code even if there is a `default` label.

`-Wswitch-bool`
Warn whenever a `switch` statement has an index of boolean type and the case values are outside the range of a boolean type. It is possible to suppress this warning by casting the controlling expression to a type other than `bool`. For example:

```
switch ((int) (a == 4))
  {
  ...
  }
```

This warning is enabled by default for C and C++ programs.

`-Wswitch-unreachable`
Warn whenever a `switch` statement contains statements between the controlling expression and the first case label, which will never be executed. For example:

```
switch (cond)
  {
   i = 15;
  ...
   case 5:
  ...
  }
```

'`-Wswitch-unreachable`' does not warn if the statement between the controlling expression and the first case label is just a declaration:

```
switch (cond)
  {
   int i;
  ...
   case 5:
   i = 5;
  ...
  }
```

This warning is enabled by default for C and C++ programs.

`-Wsync-nand` (C and C++ only)
Warn when `__sync_fetch_and_nand` and `__sync_nand_and_fetch` built-in functions are used. These functions changed semantics in GCC 4.4.

`-Wunused-but-set-parameter`
Warn whenever a function parameter is assigned to, but otherwise unused (aside from its declaration).

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

This warning is also enabled by '`-Wunused`' together with '`-Wextra`'.

`-Wunused-but-set-variable`
Warn whenever a local variable is assigned to, but otherwise unused (aside from its declaration). This warning is enabled by '`-Wall`'.

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

This warning is also enabled by '`-Wunused`', which is enabled by '`-Wall`'.

`-Wunused-function`
Warn whenever a static function is declared but not defined or a non-inline static function is unused. This warning is enabled by '`-Wall`'.

`-Wunused-label`
Warn whenever a label is declared but not used. This warning is enabled by '`-Wall`'.

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

`-Wunused-local-typedefs` (C, Objective-C, C++ and Objective-C++ only)
Warn when a typedef locally defined in a function is not used. This warning is enabled by '`-Wall`'.

`-Wunused-parameter`
Warn whenever a function parameter is unused aside from its declaration.

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

`-Wno-unused-result`
Do not warn if a caller of a function marked with attribute `warn_unused_result` (see Section 6.33 [Function Attributes], page 477) does not use its return value. The default is '`-Wunused-result`'.

`-Wunused-variable`
Warn whenever a local or static variable is unused aside from its declaration. This option implies '`-Wunused-const-variable=1`' for C, but not for C++. This warning is enabled by '`-Wall`'.

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

`-Wunused-const-variable`
`-Wunused-const-variable=n`

Warn whenever a constant static variable is unused aside from its declaration. '`-Wunused-const-variable=1`' is enabled by '`-Wunused-variable`' for C, but not for C++. In C this declares variable storage, but in C++ this is not an error since const variables take the place of `#define`s.

To suppress this warning use the `unused` attribute (see Section 6.34 [Variable Attributes], page 536).

`-Wunused-const-variable=1`

This is the warning level that is enabled by '`-Wunused-variable`' for C. It warns only about unused static const variables defined in the main compilation unit, but not about static const variables declared in any header included.

`-Wunused-const-variable=2`

This warning level also warns for unused constant static variables in headers (excluding system headers). This is the warning level of '`-Wunused-const-variable`' and must be explicitly requested since in C++ this isn't an error and in C it might be harder to clean up all headers included.

`-Wunused-value`

Warn whenever a statement computes a result that is explicitly not used. To suppress this warning cast the unused expression to `void`. This includes an expression-statement or the left-hand side of a comma expression that contains no side effects. For example, an expression such as `x[i,j]` causes a warning, while `x[(void)i,j]` does not.

This warning is enabled by '`-Wall`'.

`-Wunused` All the above '`-Wunused`' options combined.

In order to get a warning about an unused function parameter, you must either specify '`-Wextra -Wunused`' (note that '`-Wall`' implies '`-Wunused`'), or separately specify '`-Wunused-parameter`'.

`-Wuninitialized`

Warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a `setjmp` call. In C++, warn if a non-static reference or non-static `const` member appears in a class without constructors.

If you want to warn about code that uses the uninitialized value of the variable in its own initializer, use the '`-Winit-self`' option.

These warnings occur for individual uninitialized or clobbered elements of structure, union or array variables as well as for variables that are uninitialized or clobbered as a whole. They do not occur for variables or elements declared `volatile`. Because these warnings depend on optimization, the exact variables or elements for which there are warnings depends on the precise optimization options and version of GCC used.

Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

`-Winvalid-memory-model`

Warn for invocations of Section 6.55 [__atomic Builtins], page 632, Section 6.54 [__sync Builtins], page 630, and the C11 atomic generic functions with a memory consistency argument that is either invalid for the operation or outside the range of values of the `memory_order` enumeration. For example, since the `__atomic_store` and `__atomic_store_n` built-ins are only defined for the relaxed, release, and sequentially consistent memory orders the following code is diagnosed:

```
void store (int *i)
{
  __atomic_store_n (i, 0, memory_order_consume);
}
```

'`-Winvalid-memory-model`' is enabled by default.

`-Wmaybe-uninitialized`

For an automatic (i.e. local) variable, if there exists a path from the function entry to a use of the variable that is initialized, but there exist some other paths for which the variable is not initialized, the compiler emits a warning if it cannot prove the uninitialized paths are not executed at run time.

These warnings are only possible in optimizing compilation, because otherwise GCC does not keep track of the state of variables.

These warnings are made optional because GCC may not be able to determine when the code is correct in spite of appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y)
    {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
    }
  foo (x);
}
```

If the value of `y` is always 1, 2 or 3, then `x` is always initialized, but GCC doesn't know this. To suppress the warning, you need to provide a default case with assert(0) or similar code.

This option also warns when a non-volatile automatic variable might be changed by a call to `longjmp`. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place that would cause a problem.

Some spurious warnings can be avoided if you declare all the functions you use that never return as `noreturn`. See Section 6.33 [Function Attributes], page 477.

This warning is enabled by '`-Wall`' or '`-Wextra`'.

`-Wunknown-pragmas`

Warn when a `#pragma` directive is encountered that is not understood by GCC. If this command-line option is used, warnings are even issued for unknown pragmas in system header files. This is not the case if the warnings are only enabled by the '`-Wall`' command-line option.

`-Wno-pragmas`

Do not warn about misuses of pragmas, such as incorrect parameters, invalid syntax, or conflicts between pragmas. See also '`-Wunknown-pragmas`'.

`-Wno-prio-ctor-dtor`

Do not warn if a priority from 0 to 100 is used for constructor or destructor. The use of constructor and destructor attributes allow you to assign a priority to the constructor/destructor to control its order of execution before `main` is called or after it returns. The priority values must be greater than 100 as the compiler reserves priority values between 0–100 for the implementation.

`-Wstrict-aliasing`

This option is only active when '`-fstrict-aliasing`' is active. It warns about code that might break the strict aliasing rules that the compiler is using for optimization. The warning does not catch all cases, but does attempt to catch the more common pitfalls. It is included in '`-Wall`'. It is equivalent to '`-Wstrict-aliasing=3`'

`-Wstrict-aliasing=n`

This option is only active when '`-fstrict-aliasing`' is active. It warns about code that might break the strict aliasing rules that the compiler is using for optimization. Higher levels correspond to higher accuracy (fewer false positives). Higher levels also correspond to more effort, similar to the way '`-O`' works. '`-Wstrict-aliasing`' is equivalent to '`-Wstrict-aliasing=3`'.

Level 1: Most aggressive, quick, least accurate. Possibly useful when higher levels do not warn but '`-fstrict-aliasing`' still breaks the code, as it has very few false negatives. However, it has many false positives. Warns for all pointer conversions between possibly incompatible types, even if never dereferenced. Runs in the front end only.

Level 2: Aggressive, quick, not too precise. May still have many false positives (not as many as level 1 though), and few false negatives (but possibly more than level 1). Unlike level 1, it only warns when an address is taken. Warns about incomplete types. Runs in the front end only.

Level 3 (default for '`-Wstrict-aliasing`'): Should have very few false positives and few false negatives. Slightly slower than levels 1 or 2 when optimization is enabled. Takes care of the common pun+dereference pattern in the front end: `*(int*)&some_float`. If optimization is enabled, it also runs in the back end, where it deals with multiple statement cases using flow-sensitive points-to

information. Only warns when the converted pointer is dereferenced. Does not warn about incomplete types.

`-Wstrict-overflow`
`-Wstrict-overflow=n`

This option is only active when signed overflow is undefined. It warns about cases where the compiler optimizes based on the assumption that signed overflow does not occur. Note that it does not warn about all cases where the code might overflow: it only warns about cases where the compiler implements some optimization. Thus this warning depends on the optimization level.

An optimization that assumes that signed overflow does not occur is perfectly safe if the values of the variables involved are such that overflow never does, in fact, occur. Therefore this warning can easily give a false positive: a warning about code that is not actually a problem. To help focus on important issues, several warning levels are defined. No warnings are issued for the use of undefined signed overflow when estimating how many iterations a loop requires, in particular when determining whether a loop will be executed at all.

`-Wstrict-overflow=1`

Warn about cases that are both questionable and easy to avoid. For example the compiler simplifies `x + 1 > x` to `1`. This level of '`-Wstrict-overflow`' is enabled by '`-Wall`'; higher levels are not, and must be explicitly requested.

`-Wstrict-overflow=2`

Also warn about other cases where a comparison is simplified to a constant. For example: `abs (x) >= 0`. This can only be simplified when signed integer overflow is undefined, because `abs (INT_MIN)` overflows to `INT_MIN`, which is less than zero. '`-Wstrict-overflow`' (with no level) is the same as '`-Wstrict-overflow=2`'.

`-Wstrict-overflow=3`

Also warn about other cases where a comparison is simplified. For example: `x + 1 > 1` is simplified to `x > 0`.

`-Wstrict-overflow=4`

Also warn about other simplifications not covered by the above cases. For example: `(x * 10) / 5` is simplified to `x * 2`.

`-Wstrict-overflow=5`

Also warn about cases where the compiler reduces the magnitude of a constant involved in a comparison. For example: `x + 2 > y` is simplified to `x + 1 >= y`. This is reported only at the highest warning level because this simplification applies to many comparisons, so this warning level gives a very large number of false positives.

`-Wstringop-overflow`
`-Wstringop-overflow=type`

Warn for calls to string manipulation functions such as `memcpy` and `strcpy` that are determined to overflow the destination buffer. The optional argument is one greater than the type of Object Size Checking to perform to determine

the size of the destination. See Section 6.58 [Object Size Checking], page 639. The argument is meaningful only for functions that operate on character arrays but not for raw memory functions like `memcpy` which always make use of Object Size type-0. The option also warns for calls that specify a size in excess of the largest possible object or at most `SIZE_MAX / 2` bytes. The option produces the best results with optimization enabled but can detect a small subset of simple buffer overflows even without optimization in calls to the GCC built-in functions like `__builtin_memcpy` that correspond to the standard functions. In any case, the option warns about just a subset of buffer overflows detected by the corresponding overflow checking built-ins. For example, the option will issue a warning for the `strcpy` call below because it copies at least 5 characters (the string `"blue"` including the terminating NUL) into the buffer of size 4.

```
enum Color { blue, purple, yellow };
const char* f (enum Color clr)
{
  static char buf [4];
  const char *str;
  switch (clr)
    {
      case blue: str = "blue"; break;
      case purple: str = "purple"; break;
      case yellow: str = "yellow"; break;
    }

  return strcpy (buf, str);   // warning here
}
```

Option '`-Wstringop-overflow=2`' is enabled by default.

`-Wstringop-overflow`
`-Wstringop-overflow=1`

The '`-Wstringop-overflow=1`' option uses type-zero Object Size Checking to determine the sizes of destination objects. This is the default setting of the option. At this setting the option will not warn for writes past the end of subobjects of larger objects accessed by pointers unless the size of the largest surrounding object is known. When the destination may be one of several objects it is assumed to be the largest one of them. On Linux systems, when optimization is enabled at this setting the option warns for the same code as when the `_FORTIFY_SOURCE` macro is defined to a non-zero value.

`-Wstringop-overflow=2`

The '`-Wstringop-overflow=2`' option uses type-one Object Size Checking to determine the sizes of destination objects. At this setting the option will warn about overflows when writing to members of the largest complete objects whose exact size is known. It will, however, not warn for excessive writes to the same members of unknown objects referenced by pointers since they may point to arrays containing unknown numbers of elements.

`-Wstringop-overflow=3`
The '`-Wstringop-overflow=3`' option uses type-two Object Size Checking to determine the sizes of destination objects. At this setting the option warns about overflowing the smallest object or data member. This is the most restrictive setting of the option that may result in warnings for safe code.

`-Wstringop-overflow=4`
The '`-Wstringop-overflow=4`' option uses type-three Object Size Checking to determine the sizes of destination objects. At this setting the option will warn about overflowing any data members, and when the destination is one of several objects it uses the size of the largest of them to decide whether to issue a warning. Similarly to '`-Wstringop-overflow=3`' this setting of the option may result in warnings for benign code.

`-Wstringop-truncation`
Warn for calls to bounded string manipulation functions such as `strncat`, `strncpy`, and `stpncpy` that may either truncate the copied string or leave the destination unchanged.

In the following example, the call to `strncat` specifies a bound that is less than the length of the source string. As a result, the copy of the source will be truncated and so the call is diagnosed. To avoid the warning use `bufsize - strlen (buf) - 1)` as the bound.

```
void append (char *buf, size_t bufsize)
{
  strncat (buf, ".txt", 3);
}
```

As another example, the following call to `strncpy` results in copying to `d` just the characters preceding the terminating NUL, without appending the NUL to the end. Assuming the result of `strncpy` is necessarily a NUL-terminated string is a common mistake, and so the call is diagnosed. To avoid the warning when the result is not expected to be NUL-terminated, call `memcpy` instead.

```
void copy (char *d, const char *s)
{
  strncpy (d, s, strlen (s));
}
```

In the following example, the call to `strncpy` specifies the size of the destination buffer as the bound. If the length of the source string is equal to or greater than this size the result of the copy will not be NUL-terminated. Therefore, the call is also diagnosed. To avoid the warning, specify `sizeof buf - 1` as the bound and set the last element of the buffer to `NUL`.

```
void copy (const char *s)
{
  char buf[80];
  strncpy (buf, s, sizeof buf);
  ...
}
```

In situations where a character array is intended to store a sequence of bytes with no terminating `NUL` such an array may be annotated with attribute

`nonstring` to avoid this warning. Such arrays, however, are not suitable arguments to functions that expect `NUL`-terminated strings. To help detect accidental misuses of such arrays GCC issues warnings unless it can prove that the use is safe. See Section 6.34.1 [Common Variable Attributes], page 536.

`-Wsuggest-attribute=[pure|const|noreturn|format|cold|malloc]`
Warn for cases where adding an attribute may be beneficial. The attributes currently supported are listed below.

`-Wsuggest-attribute=pure`
`-Wsuggest-attribute=const`
`-Wsuggest-attribute=noreturn`
`-Wmissing-noreturn`
`-Wsuggest-attribute=malloc`
Warn about functions that might be candidates for attributes `pure`, `const` or `noreturn` or `malloc`. The compiler only warns for functions visible in other compilation units or (in the case of `pure` and `const`) if it cannot prove that the function returns normally. A function returns normally if it doesn't contain an infinite loop or return abnormally by throwing, calling `abort` or trapping. This analysis requires option '`-fipa-pure-const`', which is enabled by default at '`-O`' and higher. Higher optimization levels improve the accuracy of the analysis.

`-Wsuggest-attribute=format`
`-Wmissing-format-attribute`
Warn about function pointers that might be candidates for `format` attributes. Note these are only possible candidates, not absolute ones. GCC guesses that function pointers with `format` attributes that are used in assignment, initialization, parameter passing or return statements should have a corresponding `format` attribute in the resulting type. I.e. the left-hand side of the assignment or initialization, the type of the parameter variable, or the return type of the containing function respectively should also have a `format` attribute to avoid the warning.

GCC also warns about function definitions that might be candidates for `format` attributes. Again, these are only possible candidates. GCC guesses that `format` attributes might be appropriate for any function that calls a function like `vprintf` or `vscanf`, but this might not always be the case, and some functions for which `format` attributes are appropriate may not be detected.

`-Wsuggest-attribute=cold`
Warn about functions that might be candidates for `cold` attribute. This is based on static detection and generally will only warn about functions which always leads to a call to another `cold` function such as wrappers of C++ `throw` or fatal error reporting functions leading to `abort`.

`-Wsuggest-final-types`
Warn about types with virtual methods where code quality would be improved if the type were declared with the C++11 `final` specifier, or, if possible, declared in an anonymous namespace. This allows GCC to more aggressively devirtualize the polymorphic calls. This warning is more effective with link time optimization, where the information about the class hierarchy graph is more complete.

`-Wsuggest-final-methods`
Warn about virtual methods where code quality would be improved if the method were declared with the C++11 `final` specifier, or, if possible, its type were declared in an anonymous namespace or with the `final` specifier. This warning is more effective with link-time optimization, where the information about the class hierarchy graph is more complete. It is recommended to first consider suggestions of '`-Wsuggest-final-types`' and then rebuild with new annotations.

`-Wsuggest-override`
Warn about overriding virtual functions that are not marked with the override keyword.

`-Walloc-zero`
Warn about calls to allocation functions decorated with attribute `alloc_size` that specify zero bytes, including those to the built-in forms of the functions `aligned_alloc`, `alloca`, `calloc`, `malloc`, and `realloc`. Because the behavior of these functions when called with a zero size differs among implementations (and in the case of `realloc` has been deprecated) relying on it may result in subtle portability bugs and should be avoided.

`-Walloc-size-larger-than=`*`byte-size`*
Warn about calls to functions decorated with attribute `alloc_size` that attempt to allocate objects larger than the specified number of bytes, or where the result of the size computation in an integer type with infinite precision would exceed the value of '`PTRDIFF_MAX`' on the target. '`-Walloc-size-larger-than=`'`PTRDIFF_MAX`' is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by '`-Wno-alloc-size-larger-than`'. See Section 6.33 [Function Attributes], page 477.

`-Wno-alloc-size-larger-than`
Disable '`-Walloc-size-larger-than=`' warnings. The option is equivalent to '`-Walloc-size-larger-than=`'`SIZE_MAX`' or larger.

`-Walloca` This option warns on all uses of `alloca` in the source.

`-Walloca-larger-than=`*`byte-size`*
This option warns on calls to `alloca` with an integer argument whose value is either zero, or that is not bounded by a controlling predicate that limits its value to at most *byte-size*. It also warns for calls to `alloca` where the bound value is unknown. Arguments of non-integer types are considered unbounded even if they appear to be constrained to the expected range.

For example, a bounded case of `alloca` could be:

```
void func (size_t n)
{
  void *p;
  if (n <= 1000)
    p = alloca (n);
  else
    p = malloc (n);
  f (p);
}
```

In the above example, passing `-Walloca-larger-than=1000` would not issue a warning because the call to `alloca` is known to be at most 1000 bytes. However, if `-Walloca-larger-than=500` were passed, the compiler would emit a warning.

Unbounded uses, on the other hand, are uses of `alloca` with no controlling predicate constraining its integer argument. For example:

```
void func ()
{
  void *p = alloca (n);
  f (p);
}
```

If `-Walloca-larger-than=500` were passed, the above would trigger a warning, but this time because of the lack of bounds checking.

Note, that even seemingly correct code involving signed integers could cause a warning:

```
void func (signed int n)
{
  if (n < 500)
    {
      p = alloca (n);
      f (p);
    }
}
```

In the above example, *n* could be negative, causing a larger than expected argument to be implicitly cast into the `alloca` call.

This option also warns when `alloca` is used in a loop.

'`-Walloca-larger-than=`'`PTRDIFF_MAX`' is enabled by default but is usually only effective when '`-ftree-vrp`' is active (default for '`-O2`' and above).

See also '`-Wvla-larger-than=`'`byte-size`'.

`-Wno-alloca-larger-than`

Disable '`-Walloca-larger-than=`' warnings. The option is equivalent to '`-Walloca-larger-than=`'`SIZE_MAX`' or larger.

`-Warray-bounds`
`-Warray-bounds=`*n*

This option is only active when '`-ftree-vrp`' is active (default for '`-O2`' and above). It warns about subscripts to arrays that are always out of bounds. This warning is enabled by '`-Wall`'.

`-Warray-bounds=1`
This is the warning level of '`-Warray-bounds`' and is enabled by '`-Wall`'; higher levels are not, and must be explicitly requested.

`-Warray-bounds=2`
This warning level also warns about out of bounds access for arrays at the end of a struct and for arrays accessed through pointers. This warning level may give a larger number of false positives and is deactivated by default.

`-Wattribute-alias=n`
`-Wno-attribute-alias`
Warn about declarations using the `alias` and similar attributes whose target is incompatible with the type of the alias. See Section 6.33 [Declaring Attributes of Functions], page 477.

`-Wattribute-alias=1`
The default warning level of the '`-Wattribute-alias`' option diagnoses incompatibilities between the type of the alias declaration and that of its target. Such incompatibilities are typically indicative of bugs.

`-Wattribute-alias=2`
At this level '`-Wattribute-alias`' also diagnoses cases where the attributes of the alias declaration are more restrictive than the attributes applied to its target. These mismatches can potentially result in incorrect code generation. In other cases they may be benign and could be resolved simply by adding the missing attribute to the target. For comparison, see the '`-Wmissing-attributes`' option, which controls diagnostics when the alias declaration is less restrictive than the target, rather than more restrictive.

Attributes considered include `alloc_align`, `alloc_size`, `cold`, `const`, `hot`, `leaf`, `malloc`, `nonnull`, `noreturn`, `nothrow`, `pure`, `returns_nonnull`, and `returns_twice`.

'`-Wattribute-alias`' is equivalent to '`-Wattribute-alias=1`'. This is the default. You can disable these warnings with either '`-Wno-attribute-alias`' or '`-Wattribute-alias=0`'.

`-Wbool-compare`
Warn about boolean expression compared with an integer value different from `true/false`. For instance, the following comparison is always false:

```
int n = 5;
...
if ((n > 1) == 2) { ... }
```

This warning is enabled by '`-Wall`'.

`-Wbool-operation`
Warn about suspicious operations on expressions of a boolean type. For instance, bitwise negation of a boolean is very likely a bug in the program. For C, this warning also warns about incrementing or decrementing a boolean,

which rarely makes sense. (In C++, decrementing a boolean is always invalid. Incrementing a boolean is invalid in C++17, and deprecated otherwise.)

This warning is enabled by '-Wall'.

-Wduplicated-branches
: Warn when an if-else has identical branches. This warning detects cases like

```
if (p != NULL)
  return 0;
else
  return 0;
```

It doesn't warn when both branches contain just a null statement. This warning also warn for conditional operators:

```
int i = x ? *p : *p;
```

-Wduplicated-cond
: Warn about duplicated conditions in an if-else-if chain. For instance, warn for the following code:

```
if (p->q != NULL) { ... }
else if (p->q != NULL) { ... }
```

-Wframe-address
: Warn when the '__builtin_frame_address' or '__builtin_return_address' is called with an argument greater than 0. Such calls may return indeterminate values or crash the program. The warning is included in '-Wall'.

-Wno-discarded-qualifiers (C and Objective-C only)
: Do not warn if type qualifiers on pointers are being discarded. Typically, the compiler warns if a `const char *` variable is passed to a function that takes a `char *` parameter. This option can be used to suppress such a warning.

-Wno-discarded-array-qualifiers (C and Objective-C only)
: Do not warn if type qualifiers on arrays which are pointer targets are being discarded. Typically, the compiler warns if a `const int (*)[]` variable is passed to a function that takes a `int (*)[]` parameter. This option can be used to suppress such a warning.

-Wno-incompatible-pointer-types (C and Objective-C only)
: Do not warn when there is a conversion between pointers that have incompatible types. This warning is for cases not covered by '-Wno-pointer-sign', which warns for pointer argument passing or assignment with different signedness.

-Wno-int-conversion (C and Objective-C only)
: Do not warn about incompatible integer to pointer and pointer to integer conversions. This warning is about implicit conversions; for explicit conversions the warnings '-Wno-int-to-pointer-cast' and '-Wno-pointer-to-int-cast' may be used.

-Wno-div-by-zero
: Do not warn about compile-time integer division by zero. Floating-point division by zero is not warned about, as it can be a legitimate way of obtaining infinities and NaNs.

`-Wsystem-headers`

Print warning messages for constructs found in system header files. Warnings from system headers are normally suppressed, on the assumption that they usually do not indicate real problems and would only make the compiler output harder to read. Using this command-line option tells GCC to emit warnings from system headers as if they occurred in user code. However, note that using '`-Wall`' in conjunction with this option does *not* warn about unknown pragmas in system headers—for that, '`-Wunknown-pragmas`' must also be used.

`-Wtautological-compare`

Warn if a self-comparison always evaluates to true or false. This warning detects various mistakes such as:

```
int i = 1;
...
if (i > i) { ... }
```

This warning also warns about bitwise comparisons that always evaluate to true or false, for instance:

```
if ((a & 16) == 10) { ... }
```

will always be false.

This warning is enabled by '`-Wall`'.

`-Wtrampolines`

Warn about trampolines generated for pointers to nested functions. A trampoline is a small piece of data or code that is created at run time on the stack when the address of a nested function is taken, and is used to call the nested function indirectly. For some targets, it is made up of data only and thus requires no special treatment. But, for most targets, it is made up of code and thus requires the stack to be made executable in order for the program to work properly.

`-Wfloat-equal`

Warn if floating-point values are used in equality comparisons.

The idea behind this is that sometimes it is convenient (for the programmer) to consider floating-point values as approximations to infinitely precise real numbers. If you are doing this, then you need to compute (by analyzing the code, or in some other way) the maximum or likely maximum error that the computation introduces, and allow for it when performing comparisons (and when producing output, but that's a different problem). In particular, instead of testing for equality, you should check to see whether the two values have ranges that overlap; and this is done with the relational operators, so equality comparisons are probably mistaken.

`-Wtraditional` (C and Objective-C only)

Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and/or problematic constructs that should be avoided.

- Macro parameters that appear within string literals in the macro body. In traditional C macro replacement takes place within string literals, but in ISO C it does not.

- In traditional C, some preprocessor directives did not exist. Traditional preprocessors only considered a line to be a directive if the '#' appeared in column 1 on the line. Therefore '-Wtraditional' warns about directives that traditional C understands but ignores because the '#' does not appear as the first character on the line. It also suggests you hide directives like #pragma not understood by traditional C by indenting them. Some traditional implementations do not recognize #elif, so this option suggests avoiding it altogether.
- A function-like macro that appears without arguments.
- The unary plus operator.
- The 'U' integer constant suffix, or the 'F' or 'L' floating-point constant suffixes. (Traditional C does support the 'L' suffix on integer constants.) Note, these suffixes appear in macros defined in the system headers of most modern systems, e.g. the '_MIN'/'_MAX' macros in <limits.h>. Use of these macros in user code might normally lead to spurious warnings, however GCC's integrated preprocessor has enough context to avoid warning in these cases.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type long.
- A non-static function declaration follows a static one. This construct is not accepted by some traditional C compilers.
- The ISO type of an integer constant has a different width or signedness from its traditional type. This warning is only issued if the base of the constant is ten. I.e. hexadecimal or octal values, which typically represent bit patterns, are not warned about.
- Usage of ISO string concatenation is detected.
- Initialization of automatic aggregates.
- Identifier conflicts with labels. Traditional C lacks a separate namespace for labels.
- Initialization of unions. If the initializer is zero, the warning is omitted. This is done under the assumption that the zero initializer in user code appears conditioned on e.g. __STDC__ to avoid missing initializer warnings and relies on default initialization to zero in the traditional C case.
- Conversions by prototypes between fixed/floating-point values and vice versa. The absence of these prototypes when compiling with traditional C causes serious problems. This is a subset of the possible conversion warnings; for the full set use '-Wtraditional-conversion'.
- Use of ISO C style function definitions. This warning intentionally is *not* issued for prototype declarations or variadic functions because these ISO C features appear in your code when using libiberty's traditional C compatibility macros, PARAMS and VPARAMS. This warning is also bypassed for nested functions because that feature is already a GCC extension and thus not relevant to traditional C compatibility.

`-Wtraditional-conversion` (C and Objective-C only)
Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed-point argument except when the same as the default promotion.

`-Wdeclaration-after-statement` (C and Objective-C only)
Warn when a declaration is found after a statement in a block. This construct, known from C++, was introduced with ISO C99 and is by default allowed in GCC. It is not supported by ISO C90. See Section 6.32 [Mixed Declarations], page 477.

`-Wshadow` Warn whenever a local variable or type declaration shadows another variable, parameter, type, class member (in C++), or instance variable (in Objective-C) or whenever a built-in function is shadowed. Note that in C++, the compiler warns if a local variable shadows an explicit typedef, but not if it shadows a struct/class/enum. Same as '`-Wshadow=global`'.

`-Wno-shadow-ivar` (Objective-C only)
Do not warn whenever a local variable shadows an instance variable in an Objective-C method.

`-Wshadow=global`
The default for '`-Wshadow`'. Warns for any (global) shadowing.

`-Wshadow=local`
Warn when a local variable shadows another local variable or parameter. This warning is enabled by '`-Wshadow=global`'.

`-Wshadow=compatible-local`
Warn when a local variable shadows another local variable or parameter whose type is compatible with that of the shadowing variable. In C++, type compatibility here means the type of the shadowing variable can be converted to that of the shadowed variable. The creation of this flag (in addition to '`-Wshadow=local`') is based on the idea that when a local variable shadows another one of incompatible type, it is most likely intentional, not a bug or typo, as shown in the following example:

```
for (SomeIterator i = SomeObj.begin(); i != SomeObj.end(); ++i)
{
  for (int i = 0; i < N; ++i)
  {
    ...
  }
  ...
}
```

Since the two variable `i` in the example above have incompatible types, enabling only '`-Wshadow=compatible-local`' will not emit a warning. Because their types are incompatible, if a programmer accidentally uses one in place of the other, type checking will catch that and emit an error or warning. So not warning (about shadowing) in this case will not lead to undetected bugs. Use

of this flag instead of '`-Wshadow=local`' can possibly reduce the number of warnings triggered by intentional shadowing.

This warning is enabled by '`-Wshadow=local`'.

`-Wlarger-than=`*`byte-size`*

Warn whenever an object is defined whose size exceeds *byte-size*. '`-Wlarger-than=`'`PTRDIFF_MAX`' is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by '`-Wno-larger-than`'.

`-Wno-larger-than`

Disable '`-Wlarger-than=`' warnings. The option is equivalent to '`-Wlarger-than=`'`SIZE_MAX`' or larger.

`-Wframe-larger-than=`*`byte-size`*

Warn if the size of a function frame exceeds *byte-size*. The computation done to determine the stack frame size is approximate and not conservative. The actual requirements may be somewhat greater than *byte-size* even if you do not get a warning. In addition, any space allocated via `alloca`, variable-length arrays, or related constructs is not included by the compiler when determining whether or not to issue a warning. '`-Wframe-larger-than=`'`PTRDIFF_MAX`' is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by '`-Wno-frame-larger-than`'.

`-Wno-frame-larger-than`

Disable '`-Wframe-larger-than=`' warnings. The option is equivalent to '`-Wframe-larger-than=`'`SIZE_MAX`' or larger.

`-Wno-free-nonheap-object`

Do not warn when attempting to free an object that was not allocated on the heap.

`-Wstack-usage=`*`byte-size`*

Warn if the stack usage of a function might exceed *byte-size*. The computation done to determine the stack usage is conservative. Any space allocated via `alloca`, variable-length arrays, or related constructs is included by the compiler when determining whether or not to issue a warning.

The message is in keeping with the output of '`-fstack-usage`'.

- If the stack usage is fully static but exceeds the specified amount, it's:

```
warning: stack usage is 1120 bytes
```

- If the stack usage is (partly) dynamic but bounded, it's:

```
warning: stack usage might be 1648 bytes
```

- If the stack usage is (partly) dynamic and not bounded, it's:

```
warning: stack usage might be unbounded
```

'`-Wstack-usage=`'`PTRDIFF_MAX`' is enabled by default. Warnings controlled by the option can be disabled either by specifying *byte-size* of '`SIZE_MAX`' or more or by '`-Wno-stack-usage`'.

`-Wno-stack-usage`
Disable '`-Wstack-usage=`' warnings. The option is equivalent to '`-Wstack-usage=`'`SIZE_MAX`' or larger.

`-Wunsafe-loop-optimizations`
Warn if the loop cannot be optimized because the compiler cannot assume anything on the bounds of the loop indices. With '`-funsafe-loop-optimizations`' warn if the compiler makes such assumptions.

`-Wno-pedantic-ms-format` (MinGW targets only)
When used in combination with '`-Wformat`' and '`-pedantic`' without GNU extensions, this option disables the warnings about non-ISO `printf` / `scanf` format width specifiers `I32`, `I64`, and `I` used on Windows targets, which depend on the MS runtime.

`-Waligned-new`
Warn about a new-expression of a type that requires greater alignment than the `alignof(std::max_align_t)` but uses an allocation function without an explicit alignment parameter. This option is enabled by '`-Wall`'.

Normally this only warns about global allocation functions, but '`-Waligned-new=all`' also warns about class member allocation functions.

`-Wplacement-new`
`-Wplacement-new=`*n*
Warn about placement new expressions with undefined behavior, such as constructing an object in a buffer that is smaller than the type of the object. For example, the placement new expression below is diagnosed because it attempts to construct an array of 64 integers in a buffer only 64 bytes large.

```
char buf [64];
new (buf) int[64];
```

This warning is enabled by default.

`-Wplacement-new=1`
This is the default warning level of '`-Wplacement-new`'. At this level the warning is not issued for some strictly undefined constructs that GCC allows as extensions for compatibility with legacy code. For example, the following `new` expression is not diagnosed at this level even though it has undefined behavior according to the C++ standard because it writes past the end of the one-element array.

```
struct S { int n, a[1]; };
S *s = (S *)malloc (sizeof *s + 31 * sizeof s->a[0]);
new (s->a)int [32]();
```

`-Wplacement-new=2`
At this level, in addition to diagnosing all the same constructs as at level 1, a diagnostic is also issued for placement new expressions that construct an object in the last member of structure whose type is an array of a single element and whose size is less than the size of the object being constructed. While the previous example would be diagnosed, the following construct makes use of the flexible member array extension to avoid the warning at level 2.

```
struct S { int n, a[]; };
S *s = (S *)malloc (sizeof *s + 32 * sizeof s->a[0]);
new (s->a)int [32]();
```

`-Wpointer-arith`
: Warn about anything that depends on the "size of" a function type or of `void`. GNU C assigns these types a size of 1, for convenience in calculations with `void *` pointers and pointers to functions. In C++, warn also when an arithmetic operation involves `NULL`. This warning is also enabled by '`-Wpedantic`'.

`-Wpointer-compare`
: Warn if a pointer is compared with a zero character constant. This usually means that the pointer was meant to be dereferenced. For example:

```
const char *p = foo ();
if (p == '\0')
  return 42;
```

Note that the code above is invalid in C++11.

This warning is enabled by default.

`-Wtype-limits`
: Warn if a comparison is always true or always false due to the limited range of the data type, but do not warn for constant expressions. For example, warn if an unsigned variable is compared against zero with `<` or `>=`. This warning is also enabled by '`-Wextra`'.

`-Wabsolute-value` (C and Objective-C only)
: Warn for calls to standard functions that compute the absolute value of an argument when a more appropriate standard function is available. For example, calling `abs(3.14)` triggers the warning because the appropriate function to call to compute the absolute value of a double argument is `fabs`. The option also triggers warnings when the argument in a call to such a function has an unsigned type. This warning can be suppressed with an explicit type cast and it is also enabled by '`-Wextra`'.

`-Wcomment`
`-Wcomments`
: Warn whenever a comment-start sequence '`/*`' appears in a '`/*`' comment, or whenever a backslash-newline appears in a '`//`' comment. This warning is enabled by '`-Wall`'.

`-Wtrigraphs`
: Warn if any trigraphs are encountered that might change the meaning of the program. Trigraphs within comments are not warned about, except those that would form escaped newlines.

This option is implied by '`-Wall`'. If '`-Wall`' is not given, this option is still enabled unless trigraphs are enabled. To get trigraph conversion without warnings, but get the other '`-Wall`' warnings, use '`-trigraphs -Wall -Wno-trigraphs`'.

`-Wundef` Warn if an undefined identifier is evaluated in an `#if` directive. Such identifiers are replaced with zero.

`-Wexpansion-to-defined`
: Warn whenever '`defined`' is encountered in the expansion of a macro (including the case where the macro is expanded by an '`#if`' directive). Such usage is not portable. This warning is also enabled by '`-Wpedantic`' and '`-Wextra`'.

`-Wunused-macros`
: Warn about macros defined in the main file that are unused. A macro is *used* if it is expanded or tested for existence at least once. The preprocessor also warns if the macro has not been used at the time it is redefined or undefined.

  Built-in macros, macros defined on the command line, and macros defined in include files are not warned about.

  Note: If a macro is actually used, but only used in skipped conditional blocks, then the preprocessor reports it as unused. To avoid the warning in such a case, you might improve the scope of the macro's definition by, for example, moving it into the first skipped block. Alternatively, you could provide a dummy use with something like:

```
#if defined the_macro_causing_the_warning
#endif
```

`-Wno-endif-labels`
: Do not warn whenever an `#else` or an `#endif` are followed by text. This sometimes happens in older programs with code of the form

```
#if FOO
...
#else FOO
...
#endif FOO
```

  The second and third `FOO` should be in comments. This warning is on by default.

`-Wbad-function-cast` (C and Objective-C only)
: Warn when a function call is cast to a non-matching type. For example, warn if a call to a function returning an integer type is cast to a pointer type.

`-Wc90-c99-compat` (C and Objective-C only)
: Warn about features not present in ISO C90, but present in ISO C99. For instance, warn about use of variable length arrays, `long long` type, `bool` type, compound literals, designated initializers, and so on. This option is independent of the standards mode. Warnings are disabled in the expression that follows `__extension__`.

`-Wc99-c11-compat` (C and Objective-C only)
: Warn about features not present in ISO C99, but present in ISO C11. For instance, warn about use of anonymous structures and unions, `_Atomic` type qualifier, `_Thread_local` storage-class specifier, `_Alignas` specifier, `Alignof` operator, `_Generic` keyword, and so on. This option is independent of the standards mode. Warnings are disabled in the expression that follows `__extension__`.

`-Wc++-compat` (C and Objective-C only)
: Warn about ISO C constructs that are outside of the common subset of ISO C and ISO C++, e.g. request for implicit conversion from `void *` to a pointer to non-`void` type.

`-Wc++11-compat` (C++ and Objective-C++ only)
Warn about C++ constructs whose meaning differs between ISO C++ 1998 and ISO C++ 2011, e.g., identifiers in ISO C++ 1998 that are keywords in ISO C++ 2011. This warning turns on '`-Wnarrowing`' and is enabled by '`-Wall`'.

`-Wc++14-compat` (C++ and Objective-C++ only)
Warn about C++ constructs whose meaning differs between ISO C++ 2011 and ISO C++ 2014. This warning is enabled by '`-Wall`'.

`-Wc++17-compat` (C++ and Objective-C++ only)
Warn about C++ constructs whose meaning differs between ISO C++ 2014 and ISO C++ 2017. This warning is enabled by '`-Wall`'.

`-Wcast-qual`
Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a `const char *` is cast to an ordinary `char *`.

Also warn when making a cast that introduces a type qualifier in an unsafe way. For example, casting `char **` to `const char **` is unsafe, as in this example:

```
/* p is char ** value.  */
const char **q = (const char **) p;
/* Assignment of readonly string to const char * is OK.  */
*q = "string";
/* Now char** pointer points to read-only memory.  */
**p = 'b';
```

`-Wcast-align`
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` on machines where integers can only be accessed at two- or four-byte boundaries.

`-Wcast-align=strict`
Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn if a `char *` is cast to an `int *` regardless of the target machine.

`-Wcast-function-type`
Warn when a function pointer is cast to an incompatible function pointer. In a cast involving function types with a variable argument list only the types of initial arguments that are provided are considered. Any parameter of pointer-type matches any other pointer-type. Any benign differences in integral types are ignored, like `int` vs. `long` on ILP32 targets. Likewise type qualifiers are ignored. The function type `void (*) (void)` is special and matches everything, which can be used to suppress this warning. In a cast involving pointer to member types this warning warns whenever the type cast is changing the pointer to member type. This warning is enabled by '`-Wextra`'.

`-Wwrite-strings`
When compiling C, give string constants the type `const char[`*length*`]` so that copying the address of one into a non-`const char *` pointer produces a warning. These warnings help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using `const` in

declarations and prototypes. Otherwise, it is just a nuisance. This is why we did not make '-Wall' request these warnings.

When compiling C++, warn about the deprecated conversion from string literals to char *. This warning is enabled by default for C++ programs.

-Wcatch-value
-Wcatch-value=*n* (C++ and Objective-C++ only)

Warn about catch handlers that do not catch via reference. With '-Wcatch-value=1' (or '-Wcatch-value' for short) warn about polymorphic class types that are caught by value. With '-Wcatch-value=2' warn about all class types that are caught by value. With '-Wcatch-value=3' warn about all types that are not caught by reference. '-Wcatch-value' is enabled by '-Wall'.

-Wclobbered

Warn for variables that might be changed by longjmp or vfork. This warning is also enabled by '-Wextra'.

-Wconditionally-supported (C++ and Objective-C++ only)

Warn for conditionally-supported (C++11 [intro.defs]) constructs.

-Wconversion

Warn for implicit conversions that may alter a value. This includes conversions between real and integer, like abs (x) when x is double; conversions between signed and unsigned, like unsigned ui = -1; and conversions to smaller types, like sqrtf (M_PI). Do not warn for explicit casts like abs ((int) x) and ui = (unsigned) -1, or if the value is not changed by the conversion like in abs (2.0). Warnings about conversions between signed and unsigned integers can be disabled by using '-Wno-sign-conversion'.

For C++, also warn for confusing overload resolution for user-defined conversions; and conversions that never use a type conversion operator: conversions to void, the same type, a base class or a reference to them. Warnings about conversions between signed and unsigned integers are disabled by default in C++ unless '-Wsign-conversion' is explicitly enabled.

-Wno-conversion-null (C++ and Objective-C++ only)

Do not warn for conversions between NULL and non-pointer types. '-Wconversion-null' is enabled by default.

-Wzero-as-null-pointer-constant (C++ and Objective-C++ only)

Warn when a literal '0' is used as null pointer constant. This can be useful to facilitate the conversion to nullptr in C++11.

-Wsubobject-linkage (C++ and Objective-C++ only)

Warn if a class type has a base or a field whose type uses the anonymous namespace or depends on a type with no linkage. If a type A depends on a type B with no or internal linkage, defining it in multiple translation units would be an ODR violation because the meaning of B is different in each translation unit. If A only appears in a single translation unit, the best way to silence the warning is to give it internal linkage by putting it in an anonymous namespace as well. The compiler doesn't give this warning for types defined in the main .C

file, as those are unlikely to have multiple definitions. '-Wsubobject-linkage' is enabled by default.

-Wdangling-else
: Warn about constructions where there may be confusion to which if statement an else branch belongs. Here is an example of such a case:

```
{
  if (a)
    if (b)
      foo ();
  else
    bar ();
}
```

In C/C++, every else branch belongs to the innermost possible if statement, which in this example is if (b). This is often not what the programmer expected, as illustrated in the above example by indentation the programmer chose. When there is the potential for this confusion, GCC issues a warning when this flag is specified. To eliminate the warning, add explicit braces around the innermost if statement so there is no way the else can belong to the enclosing if. The resulting code looks like this:

```
{
  if (a)
    {
      if (b)
        foo ();
      else
        bar ();
    }
}
```

This warning is enabled by '-Wparentheses'.

-Wdate-time
: Warn when macros __TIME__, __DATE__ or __TIMESTAMP__ are encountered as they might prevent bit-wise-identical reproducible compilations.

-Wdelete-incomplete (C++ and Objective-C++ only)
: Warn when deleting a pointer to incomplete type, which may cause undefined behavior at runtime. This warning is enabled by default.

-Wuseless-cast (C++ and Objective-C++ only)
: Warn when an expression is casted to its own type.

-Wempty-body
: Warn if an empty body occurs in an if, else or do while statement. This warning is also enabled by '-Wextra'.

-Wenum-compare
: Warn about a comparison between values of different enumerated types. In C++ enumerated type mismatches in conditional expressions are also diagnosed and the warning is enabled by default. In C this warning is enabled by '-Wall'.

-Wextra-semi (C++, Objective-C++ only)
: Warn about redundant semicolon after in-class function definition.

`-Wjump-misses-init` (C, Objective-C only)

Warn if a `goto` statement or a `switch` statement jumps forward across the initialization of a variable, or jumps backward to a label after the variable has been initialized. This only warns about variables that are initialized when they are declared. This warning is only supported for C and Objective-C; in C++ this sort of branch is an error in any case.

'`-Wjump-misses-init`' is included in '`-Wc++-compat`'. It can be disabled with the '`-Wno-jump-misses-init`' option.

`-Wsign-compare`

Warn when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. In C++, this warning is also enabled by '`-Wall`'. In C, it is also enabled by '`-Wextra`'.

`-Wsign-conversion`

Warn for implicit conversions that may change the sign of an integer value, like assigning a signed integer expression to an unsigned integer variable. An explicit cast silences the warning. In C, this option is enabled also by '`-Wconversion`'.

`-Wfloat-conversion`

Warn for implicit conversions that reduce the precision of a real value. This includes conversions from real to integer, and from higher precision real to lower precision real values. This option is also enabled by '`-Wconversion`'.

`-Wno-scalar-storage-order`

Do not warn on suspicious constructs involving reverse scalar storage order.

`-Wsized-deallocation` (C++ and Objective-C++ only)

Warn about a definition of an unsized deallocation function

```
void operator delete (void *) noexcept;
void operator delete[] (void *) noexcept;
```

without a definition of the corresponding sized deallocation function

```
void operator delete (void *, std::size_t) noexcept;
void operator delete[] (void *, std::size_t) noexcept;
```

or vice versa. Enabled by '`-Wextra`' along with '`-fsized-deallocation`'.

`-Wsizeof-pointer-div`

Warn for suspicious divisions of two sizeof expressions that divide the pointer size by the element size, which is the usual way to compute the array size but won't work out correctly with pointers. This warning warns e.g. about `sizeof (ptr) / sizeof (ptr[0])` if `ptr` is not an array, but a pointer. This warning is enabled by '`-Wall`'.

`-Wsizeof-pointer-memaccess`

Warn for suspicious length parameters to certain string and memory built-in functions if the argument uses `sizeof`. This warning triggers for example for `memset (ptr, 0, sizeof (ptr));` if `ptr` is not an array, but a pointer, and suggests a possible fix, or about `memcpy (&foo, ptr, sizeof (&foo));`. '`-Wsizeof-pointer-memaccess`' also warns about calls to bounded string copy functions like `strncat` or `strncpy` that specify as the bound a `sizeof` expression of the source array. For example, in the following function the call to

strncat specifies the size of the source string as the bound. That is almost certainly a mistake and so the call is diagnosed.

```
void make_file (const char *name)
{
  char path[PATH_MAX];
  strncpy (path, name, sizeof path - 1);
  strncat (path, ".text", sizeof ".text");
  ...
}
```

The '-Wsizeof-pointer-memaccess' option is enabled by '-Wall'.

-Wsizeof-array-argument
: Warn when the sizeof operator is applied to a parameter that is declared as an array in a function definition. This warning is enabled by default for C and C++ programs.

-Wmemset-elt-size
: Warn for suspicious calls to the memset built-in function, if the first argument references an array, and the third argument is a number equal to the number of elements, but not equal to the size of the array in memory. This indicates that the user has omitted a multiplication by the element size. This warning is enabled by '-Wall'.

-Wmemset-transposed-args
: Warn for suspicious calls to the memset built-in function where the second argument is not zero and the third argument is zero. For example, the call memset (buf, sizeof buf, 0) is diagnosed because memset (buf, 0, sizeof buf) was meant instead. The diagnostic is only emitted if the third argument is a literal zero. Otherwise, if it is an expression that is folded to zero, or a cast of zero to some type, it is far less likely that the arguments have been mistakenly transposed and no warning is emitted. This warning is enabled by '-Wall'.

-Waddress
: Warn about suspicious uses of memory addresses. These include using the address of a function in a conditional expression, such as void func(void); if (func), and comparisons against the memory address of a string literal, such as if (x == "abc"). Such uses typically indicate a programmer error: the address of a function always evaluates to true, so their use in a conditional usually indicate that the programmer forgot the parentheses in a function call; and comparisons against string literals result in unspecified behavior and are not portable in C, so they usually indicate that the programmer intended to use strcmp. This warning is enabled by '-Wall'.

-Waddress-of-packed-member
: Warn when the address of packed member of struct or union is taken, which usually results in an unaligned pointer value. This is enabled by default.

-Wlogical-op
: Warn about suspicious uses of logical operators in expressions. This includes using logical operators in contexts where a bit-wise operator is likely to be expected. Also warns when the operands of a logical operator are the same:

```
extern int a;
if (a < 0 && a < 0) { ... }
```

-Wlogical-not-parentheses

Warn about logical not used on the left hand side operand of a comparison. This option does not warn if the right operand is considered to be a boolean expression. Its purpose is to detect suspicious code like the following:

```
int a;
...
if (!a > 1) { ... }
```

It is possible to suppress the warning by wrapping the LHS into parentheses:

```
if ((!a) > 1) { ... }
```

This warning is enabled by '-Wall'.

-Waggregate-return

Warn if any functions that return structures or unions are defined or called. (In languages where you can return an array, this also elicits a warning.)

-Wno-aggressive-loop-optimizations

Warn if in a loop with constant number of iterations the compiler detects undefined behavior in some statement during one or more of the iterations.

-Wno-attributes

Do not warn if an unexpected __attribute__ is used, such as unrecognized attributes, function attributes applied to variables, etc. This does not stop errors for incorrect use of supported attributes.

-Wno-builtin-declaration-mismatch

Warn if a built-in function is declared with an incompatible signature or as a non-function, or when a built-in function declared with a type that does not include a prototype is called with arguments whose promoted types do not match those expected by the function. When '-Wextra' is specified, also warn when a built-in function that takes arguments is declared without a prototype. The '-Wno-builtin-declaration-mismatch' warning is enabled by default. To avoid the warning include the appropriate header to bring the prototypes of built-in functions into scope.

For example, the call to memset below is diagnosed by the warning because the function expects a value of type size_t as its argument but the type of 32 is int. With '-Wextra', the declaration of the function is diagnosed as well.

```
extern void* memset ();
void f (void *d)
{
  memset (d, '\0', 32);
}
```

-Wno-builtin-macro-redefined

Do not warn if certain built-in macros are redefined. This suppresses warnings for redefinition of __TIMESTAMP__, __TIME__, __DATE__, __FILE__, and __BASE_FILE__.

`-Wstrict-prototypes` (C and Objective-C only)
Warn if a function is declared or defined without specifying the argument types. (An old-style function definition is permitted without a warning if preceded by a declaration that specifies the argument types.)

`-Wold-style-declaration` (C and Objective-C only)
Warn for obsolescent usages, according to the C Standard, in a declaration. For example, warn if storage-class specifiers like `static` are not the first things in a declaration. This warning is also enabled by '`-Wextra`'.

`-Wold-style-definition` (C and Objective-C only)
Warn if an old-style function definition is used. A warning is given even if there is a previous prototype.

`-Wmissing-parameter-type` (C and Objective-C only)
A function parameter is declared without a type specifier in K&R-style functions:

```
void foo(bar) { }
```

This warning is also enabled by '`-Wextra`'.

`-Wmissing-prototypes` (C and Objective-C only)
Warn if a global function is defined without a previous prototype declaration. This warning is issued even if the definition itself provides a prototype. Use this option to detect global functions that do not have a matching prototype declaration in a header file. This option is not valid for C++ because all function declarations provide prototypes and a non-matching declaration declares an overload rather than conflict with an earlier declaration. Use '`-Wmissing-declarations`' to detect missing declarations in C++.

`-Wmissing-declarations`
Warn if a global function is defined without a previous declaration. Do so even if the definition itself provides a prototype. Use this option to detect global functions that are not declared in header files. In C, no warnings are issued for functions with previous non-prototype declarations; use '`-Wmissing-prototypes`' to detect missing prototypes. In C++, no warnings are issued for function templates, or for inline functions, or for functions in anonymous namespaces.

`-Wmissing-field-initializers`
Warn if a structure's initializer has some fields missing. For example, the following code causes such a warning, because `x.h` is implicitly zero:

```
struct s { int f, g, h; };
struct s x = { 3, 4 };
```

This option does not warn about designated initializers, so the following modification does not trigger a warning:

```
struct s { int f, g, h; };
struct s x = { .f = 3, .g = 4 };
```

In C this option does not warn about the universal zero initializer '`{ 0 }`':

```
struct s { int f, g, h; };
struct s x = { 0 };
```

Likewise, in C++ this option does not warn about the empty { } initializer, for example:

```
struct s { int f, g, h; };
s x = { };
```

This warning is included in '-Wextra'. To get other '-Wextra' warnings without this one, use '-Wextra -Wno-missing-field-initializers'.

-Wno-multichar

Do not warn if a multicharacter constant (''FOOF'') is used. Usually they indicate a typo in the user's code, as they have implementation-defined values, and should not be used in portable code.

-Wnormalized=[none|id|nfc|nfkc]

In ISO C and ISO C++, two identifiers are different if they are different sequences of characters. However, sometimes when characters outside the basic ASCII character set are used, you can have two different character sequences that look the same. To avoid confusion, the ISO 10646 standard sets out some *normalization rules* which when applied ensure that two sequences that look the same are turned into the same sequence. GCC can warn you if you are using identifiers that have not been normalized; this option controls that warning.

There are four levels of warning supported by GCC. The default is '-Wnormalized=nfc', which warns about any identifier that is not in the ISO 10646 "C" normalized form, *NFC*. NFC is the recommended form for most uses. It is equivalent to '-Wnormalized'.

Unfortunately, there are some characters allowed in identifiers by ISO C and ISO C++ that, when turned into NFC, are not allowed in identifiers. That is, there's no way to use these symbols in portable ISO C or C++ and have all your identifiers in NFC. '-Wnormalized=id' suppresses the warning for these characters. It is hoped that future versions of the standards involved will correct this, which is why this option is not the default.

You can switch the warning off for all characters by writing '-Wnormalized=none' or '-Wno-normalized'. You should only do this if you are using some other normalization scheme (like "D"), because otherwise you can easily create bugs that are literally impossible to see.

Some characters in ISO 10646 have distinct meanings but look identical in some fonts or display methodologies, especially once formatting has been applied. For instance \u207F, "SUPERSCRIPT LATIN SMALL LETTER N", displays just like a regular n that has been placed in a superscript. ISO 10646 defines the *NFKC* normalization scheme to convert all these into a standard form as well, and GCC warns if your code is not in NFKC if you use '-Wnormalized=nfkc'. This warning is comparable to warning about every identifier that contains the letter O because it might be confused with the digit 0, and so is not the default, but may be useful as a local coding convention if the programming environment cannot be fixed to display these characters distinctly.

-Wno-attribute-warning

Do not warn about usage of functions (see Section 6.33 [Function Attributes], page 477) declared with warning attribute. By default, this warning is enabled. '-Wno-attribute-warning' can be used to disable the warning or

'-Wno-error=attribute-warning' can be used to disable the error when compiled with '-Werror' flag.

-Wno-deprecated
Do not warn about usage of deprecated features. See Section 7.11 [Deprecated Features], page 826.

-Wno-deprecated-declarations
Do not warn about uses of functions (see Section 6.33 [Function Attributes], page 477), variables (see Section 6.34 [Variable Attributes], page 536), and types (see Section 6.35 [Type Attributes], page 548) marked as deprecated by using the deprecated attribute.

-Wno-overflow
Do not warn about compile-time overflow in constant expressions.

-Wno-odr Warn about One Definition Rule violations during link-time optimization. Requires '-flto-odr-type-merging' to be enabled. Enabled by default.

-Wopenmp-simd
Warn if the vectorizer cost model overrides the OpenMP simd directive set by user. The '-fsimd-cost-model=unlimited' option can be used to relax the cost model.

-Woverride-init (C and Objective-C only)
Warn if an initialized field without side effects is overridden when using designated initializers (see Section 6.29 [Designated Initializers], page 474).

This warning is included in '-Wextra'. To get other '-Wextra' warnings without this one, use '-Wextra -Wno-override-init'.

-Woverride-init-side-effects (C and Objective-C only)
Warn if an initialized field with side effects is overridden when using designated initializers (see Section 6.29 [Designated Initializers], page 474). This warning is enabled by default.

-Wpacked Warn if a structure is given the packed attribute, but the packed attribute has no effect on the layout or size of the structure. Such structures may be mis-aligned for little benefit. For instance, in this code, the variable f.x in struct bar is misaligned even though struct bar does not itself have the packed attribute:

```
struct foo {
  int x;
  char a, b, c, d;
} __attribute__((packed));
struct bar {
  char z;
  struct foo f;
};
```

-Wpacked-bitfield-compat
The 4.1, 4.2 and 4.3 series of GCC ignore the packed attribute on bit-fields of type char. This has been fixed in GCC 4.4 but the change can lead to differences in the structure layout. GCC informs you when the offset of such a field has changed in GCC 4.4. For example there is no longer a 4-bit padding between field a and b in this structure:

```
struct foo
{
  char a:4;
  char b:8;
} __attribute__ ((packed));
```

This warning is enabled by default. Use '-Wno-packed-bitfield-compat' to disable this warning.

-Wpacked-not-aligned (C, C++, Objective-C and Objective-C++ only)
Warn if a structure field with explicitly specified alignment in a packed struct or union is misaligned. For example, a warning will be issued on struct S, like, warning: alignment 1 of 'struct S' is less than 8, in this code:

```
struct __attribute__ ((aligned (8))) S8 { char a[8]; };
struct __attribute__ ((packed)) S {
  struct S8 s8;
};
```

This warning is enabled by '-Wall'.

-Wpadded Warn if padding is included in a structure, either to align an element of the structure or to align the whole structure. Sometimes when this happens it is possible to rearrange the fields of the structure to reduce the padding and so make the structure smaller.

-Wredundant-decls
Warn if anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

-Wno-restrict
Warn when an object referenced by a restrict-qualified parameter (or, in C++, a __restrict-qualified parameter) is aliased by another argument, or when copies between such objects overlap. For example, the call to the strcpy function below attempts to truncate the string by replacing its initial characters with the last four. However, because the call writes the terminating NUL into a[4], the copies overlap and the call is diagnosed.

```
void foo (void)
{
  char a[] = "abcd1234";
  strcpy (a, a + 4);
  ...
}
```

The '-Wrestrict' option detects some instances of simple overlap even without optimization but works best at '-O2' and above. It is included in '-Wall'.

-Wnested-externs (C and Objective-C only)
Warn if an extern declaration is encountered within a function.

-Wno-inherited-variadic-ctor
Suppress warnings about use of C++11 inheriting constructors when the base class inherited from has a C variadic constructor; the warning is on by default because the ellipsis is not inherited.

`-Winline` Warn if a function that is declared as inline cannot be inlined. Even with this option, the compiler does not warn about failures to inline functions declared in system headers.

The compiler uses a variety of heuristics to determine whether or not to inline a function. For example, the compiler takes into account the size of the function being inlined and the amount of inlining that has already been done in the current function. Therefore, seemingly insignificant changes in the source program can cause the warnings produced by '`-Winline`' to appear or disappear.

`-Wno-invalid-offsetof` (C++ and Objective-C++ only)
Suppress warnings from applying the `offsetof` macro to a non-POD type. According to the 2014 ISO C++ standard, applying `offsetof` to a non-standard-layout type is undefined. In existing C++ implementations, however, `offsetof` typically gives meaningful results. This flag is for users who are aware that they are writing nonportable code and who have deliberately chosen to ignore the warning about it.

The restrictions on `offsetof` may be relaxed in a future version of the C++ standard.

`-Wint-in-bool-context`
Warn for suspicious use of integer values where boolean values are expected, such as conditional expressions (?:) using non-boolean integer constants in boolean context, like `if (a <= b ? 2 : 3)`. Or left shifting of signed integers in boolean context, like `for (a = 0; 1 << a; a++);`. Likewise for all kinds of multiplications regardless of the data type. This warning is enabled by '`-Wall`'.

`-Wno-int-to-pointer-cast`
Suppress warnings from casts to pointer type of an integer of a different size. In C++, casting to a pointer type of smaller size is an error. '`Wint-to-pointer-cast`' is enabled by default.

`-Wno-pointer-to-int-cast` (C and Objective-C only)
Suppress warnings from casts from a pointer to an integer type of a different size.

`-Winvalid-pch`
Warn if a precompiled header (see Section 3.21 [Precompiled Headers], page 438) is found in the search path but cannot be used.

`-Wlong-long`
Warn if `long long` type is used. This is enabled by either '`-Wpedantic`' or '`-Wtraditional`' in ISO C90 and C++98 modes. To inhibit the warning messages, use '`-Wno-long-long`'.

`-Wvariadic-macros`
Warn if variadic macros are used in ISO C90 mode, or if the GNU alternate syntax is used in ISO C99 mode. This is enabled by either '`-Wpedantic`' or '`-Wtraditional`'. To inhibit the warning messages, use '`-Wno-variadic-macros`'.

`-Wvarargs`
Warn upon questionable usage of the macros used to handle variable arguments like `va_start`. This is default. To inhibit the warning messages, use '`-Wno-varargs`'.

`-Wvector-operation-performance`
Warn if vector operation is not implemented via SIMD capabilities of the architecture. Mainly useful for the performance tuning. Vector operation can be implemented `piecewise`, which means that the scalar operation is performed on every vector element; `in parallel`, which means that the vector operation is implemented using scalars of wider type, which normally is more performance efficient; and `as a single scalar`, which means that vector fits into a scalar type.

`-Wno-virtual-move-assign`
Suppress warnings about inheriting from a virtual base with a non-trivial C++11 move assignment operator. This is dangerous because if the virtual base is reachable along more than one path, it is moved multiple times, which can mean both objects end up in the moved-from state. If the move assignment operator is written to avoid moving from a moved-from object, this warning can be disabled.

`-Wvla` Warn if a variable-length array is used in the code. '`-Wno-vla`' prevents the '`-Wpedantic`' warning of the variable-length array.

`-Wvla-larger-than=`*`byte-size`*
If this option is used, the compiler will warn for declarations of variable-length arrays whose size is either unbounded, or bounded by an argument that allows the array size to exceed *byte-size* bytes. This is similar to how '`-Walloca-larger-than=`'*byte-size* works, but with variable-length arrays.

Note that GCC may optimize small variable-length arrays of a known value into plain arrays, so this warning may not get triggered for such arrays.

'`-Wvla-larger-than=`'`PTRDIFF_MAX`' is enabled by default but is typically only effective when '`-ftree-vrp`' is active (default for '`-O2`' and above).

See also '`-Walloca-larger-than=`*`byte-size`*'.

`-Wno-vla-larger-than`
Disable '`-Wvla-larger-than=`' warnings. The option is equivalent to '`-Wvla-larger-than=`'`SIZE_MAX`' or larger.

`-Wvolatile-register-var`
Warn if a register variable is declared volatile. The volatile modifier does not inhibit all optimizations that may eliminate reads and/or writes to register variables. This warning is enabled by '`-Wall`'.

`-Wdisabled-optimization`
Warn if a requested optimization pass is disabled. This warning does not generally indicate that there is anything wrong with your code; it merely indicates that GCC's optimizers are unable to handle the code effectively. Often, the problem is that your code is too big or too complex; GCC refuses to optimize

programs when the optimization itself is likely to take inordinate amounts of time.

`-Wpointer-sign` (C and Objective-C only)
Warn for pointer argument passing or assignment with different signedness. This option is only supported for C and Objective-C. It is implied by '`-Wall`' and by '`-Wpedantic`', which can be disabled with '`-Wno-pointer-sign`'.

`-Wstack-protector`
This option is only active when '`-fstack-protector`' is active. It warns about functions that are not protected against stack smashing.

`-Woverlength-strings`
Warn about string constants that are longer than the "minimum maximum" length specified in the C standard. Modern compilers generally allow string constants that are much longer than the standard's minimum limit, but very portable programs should avoid using longer strings.

The limit applies *after* string constant concatenation, and does not count the trailing NUL. In C90, the limit was 509 characters; in C99, it was raised to 4095. C++98 does not specify a normative minimum maximum, so we do not diagnose overlength strings in C++.

This option is implied by '`-Wpedantic`', and can be disabled with '`-Wno-overlength-strings`'.

`-Wunsuffixed-float-constants` (C and Objective-C only)
Issue a warning for any floating constant that does not have a suffix. When used together with '`-Wsystem-headers`' it warns about such constants in system header files. This can be useful when preparing code to use with the `FLOAT_CONST_DECIMAL64` pragma from the decimal floating-point extension to C99.

`-Wno-designated-init` (C and Objective-C only)
Suppress warnings when a positional initializer is used to initialize a structure that has been marked with the `designated_init` attribute.

`-Whsa` Issue a warning when HSAIL cannot be emitted for the compiled function or OpenMP construct.

3.9 Options for Debugging Your Program

To tell GCC to emit extra information for use by a debugger, in almost all cases you need only to add '`-g`' to your other options.

GCC allows you to use '`-g`' with '`-O`'. The shortcuts taken by optimized code may occasionally be surprising: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values are already at hand; some statements may execute in different places because they have been moved out of loops. Nevertheless it is possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

If you are not using some other optimization option, consider using '`-Og`' (see Section 3.10 [Optimize Options], page 125) with '`-g`'. With no '`-O`' option at all, some compiler passes

that collect information useful for debugging do not run at all, so that '`-Og`' may result in a better debugging experience.

`-g` Produce debugging information in the operating system's native format (stabs, COFF, XCOFF, or DWARF). GDB can work with this debugging information.

On most systems that use stabs format, '`-g`' enables use of extra debugging information that only GDB can use; this extra information makes debugging work better in GDB but probably makes other debuggers crash or refuse to read the program. If you want to control for certain whether to generate the extra information, use '`-gstabs+`', '`-gstabs`', '`-gxcoff+`', '`-gxcoff`', or '`-gvms`' (see below).

`-ggdb` Produce debugging information for use by GDB. This means to use the most expressive format available (DWARF, stabs, or the native format if neither of those are supported), including GDB extensions if at all possible.

`-gdwarf`
`-gdwarf-`*`version`*

Produce debugging information in DWARF format (if that is supported). The value of *version* may be either 2, 3, 4 or 5; the default version for most targets is 4. DWARF Version 5 is only experimental.

Note that with DWARF Version 2, some ports require and always use some non-conflicting DWARF 3 extensions in the unwind tables.

Version 4 may require GDB 7.0 and '`-fvar-tracking-assignments`' for maximum benefit.

GCC no longer supports DWARF Version 1, which is substantially different than Version 2 and later. For historical reasons, some other DWARF-related options such as '`-fno-dwarf2-cfi-asm`') retain a reference to DWARF Version 2 in their names, but apply to all currently-supported versions of DWARF.

`-gstabs` Produce debugging information in stabs format (if that is supported), without GDB extensions. This is the format used by DBX on most BSD systems. On MIPS, Alpha and System V Release 4 systems this option produces stabs debugging output that is not understood by DBX. On System V Release 4 systems this option requires the GNU assembler.

`-gstabs+` Produce debugging information in stabs format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program.

`-gxcoff` Produce debugging information in XCOFF format (if that is supported). This is the format used by the DBX debugger on IBM RS/6000 systems.

`-gxcoff+` Produce debugging information in XCOFF format (if that is supported), using GNU extensions understood only by the GNU debugger (GDB). The use of these extensions is likely to make other debuggers crash or refuse to read the program, and may cause assemblers other than the GNU assembler (GAS) to fail with an error.

`-gvms` Produce debugging information in Alpha/VMS debug format (if that is supported). This is the format used by DEBUG on Alpha/VMS systems.

`-g`*level*
`-ggdb`*level*
`-gstabs`*level*
`-gxcoff`*level*
`-gvms`*level*

Request debugging information and also use *level* to specify how much information. The default level is 2.

Level 0 produces no debug information at all. Thus, '`-g0`' negates '`-g`'.

Level 1 produces minimal information, enough for making backtraces in parts of the program that you don't plan to debug. This includes descriptions of functions and external variables, and line number tables, but no information about local variables.

Level 3 includes extra information, such as all the macro definitions present in the program. Some debuggers support macro expansion when you use '`-g3`'.

If you use multiple '`-g`' options, with or without level numbers, the last such option is the one that is effective.

'`-gdwarf`' does not accept a concatenated debug level, to avoid confusion with '`-gdwarf-`*level*'. Instead use an additional '`-g`*level*' option to change the debug level for DWARF.

`-feliminate-unused-debug-symbols`
Produce debugging information in stabs format (if that is supported), for only symbols that are actually used.

`-femit-class-debug-always`
Instead of emitting debugging information for a C++ class in only one object file, emit it in all object files using the class. This option should be used only with debuggers that are unable to handle the way GCC normally emits debugging information for classes because using this option increases the size of debugging information by as much as a factor of two.

`-fno-merge-debug-strings`
Direct the linker to not merge together strings in the debugging information that are identical in different object files. Merging is not supported by all assemblers or linkers. Merging decreases the size of the debug information in the output file at the cost of increasing link processing time. Merging is enabled by default.

`-fdebug-prefix-map=`*old*`=`*new*
When compiling files residing in directory '*old*', record debugging information describing them as if the files resided in directory '*new*' instead. This can be used to replace a build-time path with an install-time path in the debug info. It can also be used to change an absolute path to a relative path by using '`.`' for *new*. This can give more reproducible builds, which are location independent, but may require an extra command to tell GDB where to find the source files. See also '`-ffile-prefix-map`'.

`-fvar-tracking`

Run variable tracking pass. It computes where variables are stored at each position in code. Better debugging information is then generated (if the debugging information format supports this information).

It is enabled by default when compiling with optimization ('`-Os`', '`-O`', '`-O2`', ...), debugging information ('`-g`') and the debug info format supports it.

`-fvar-tracking-assignments`

Annotate assignments to user variables early in the compilation and attempt to carry the annotations over throughout the compilation all the way to the end, in an attempt to improve debug information while optimizing. Use of '`-gdwarf-4`' is recommended along with it.

It can be enabled even if var-tracking is disabled, in which case annotations are created and maintained, but discarded at the end. By default, this flag is enabled together with '`-fvar-tracking`', except when selective scheduling is enabled.

`-gsplit-dwarf`

Separate as much DWARF debugging information as possible into a separate output file with the extension '`.dwo`'. This option allows the build system to avoid linking files with debug information. To be useful, this option requires a debugger capable of reading '`.dwo`' files.

`-gdescribe-dies`

Add description attributes to some DWARF DIEs that have no name attribute, such as artificial variables, external references and call site parameter DIEs.

`-gpubnames`

Generate DWARF `.debug_pubnames` and `.debug_pubtypes` sections.

`-ggnu-pubnames`

Generate `.debug_pubnames` and `.debug_pubtypes` sections in a format suitable for conversion into a GDB index. This option is only useful with a linker that can produce GDB index version 7.

`-fdebug-types-section`

When using DWARF Version 4 or higher, type DIEs can be put into their own `.debug_types` section instead of making them part of the `.debug_info` section. It is more efficient to put them in a separate comdat section since the linker can then remove duplicates. But not all DWARF consumers support `.debug_types` sections yet and on some objects `.debug_types` produces larger instead of smaller debugging information.

`-grecord-gcc-switches`
`-gno-record-gcc-switches`

This switch causes the command-line options used to invoke the compiler that may affect code generation to be appended to the DW_AT_producer attribute in DWARF debugging information. The options are concatenated with spaces separating them from each other and from the compiler version. It is enabled by default. See also '`-frecord-gcc-switches`' for another way of storing compiler options into the object file.

`-gstrict-dwarf`
: Disallow using extensions of later DWARF standard version than selected with '`-gdwarf-version`'. On most targets using non-conflicting DWARF extensions from later standard versions is allowed.

`-gno-strict-dwarf`
: Allow using extensions of later DWARF standard version than selected with '`-gdwarf-version`'.

`-gas-loc-support`
: Inform the compiler that the assembler supports `.loc` directives. It may then use them for the assembler to generate DWARF2+ line number tables.

  This is generally desirable, because assembler-generated line-number tables are a lot more compact than those the compiler can generate itself.

  This option will be enabled by default if, at GCC configure time, the assembler was found to support such directives.

`-gno-as-loc-support`
: Force GCC to generate DWARF2+ line number tables internally, if DWARF2+ line number tables are to be generated.

`gas-locview-support`
: Inform the compiler that the assembler supports `view` assignment and reset assertion checking in `.loc` directives.

  This option will be enabled by default if, at GCC configure time, the assembler was found to support them.

`gno-as-locview-support`
: Force GCC to assign view numbers internally, if '`-gvariable-location-views`' are explicitly requested.

`-gcolumn-info`
`-gno-column-info`
: Emit location column information into DWARF debugging information, rather than just file and line. This option is enabled by default.

`-gstatement-frontiers`
`-gno-statement-frontiers`
: This option causes GCC to create markers in the internal representation at the beginning of statements, and to keep them roughly in place throughout compilation, using them to guide the output of `is_stmt` markers in the line number table. This is enabled by default when compiling with optimization ('`-Os`', '`-O`', '`-O2`', ...), and outputting DWARF 2 debug information at the normal level.

`-gvariable-location-views`
`-gvariable-location-views=incompat5`
`-gno-variable-location-views`
: Augment variable location lists with progressive view numbers implied from the line number table. This enables debug information consumers to inspect state at certain points of the program, even if no instructions associated with the

corresponding source locations are present at that point. If the assembler lacks support for view numbers in line number tables, this will cause the compiler to emit the line number table, which generally makes them somewhat less compact. The augmented line number tables and location lists are fully backward-compatible, so they can be consumed by debug information consumers that are not aware of these augmentations, but they won't derive any benefit from them either.

This is enabled by default when outputting DWARF 2 debug information at the normal level, as long as there is assembler support, '`-fvar-tracking-assignments`' is enabled and '`-gstrict-dwarf`' is not. When assembler support is not available, this may still be enabled, but it will force GCC to output internal line number tables, and if '`-ginternal-reset-location-views`' is not enabled, that will most certainly lead to silently mismatching location views.

There is a proposed representation for view numbers that is not backward compatible with the location list format introduced in DWARF 5, that can be enabled with '`-gvariable-location-views=incompat5`'. This option may be removed in the future, is only provided as a reference implementation of the proposed representation. Debug information consumers are not expected to support this extended format, and they would be rendered unable to decode location lists using it.

`-ginternal-reset-location-views`
`-gnointernal-reset-location-views`

Attempt to determine location views that can be omitted from location view lists. This requires the compiler to have very accurate insn length estimates, which isn't always the case, and it may cause incorrect view lists to be generated silently when using an assembler that does not support location view lists. The GNU assembler will flag any such error as a `view number mismatch`. This is only enabled on ports that define a reliable estimation function.

`-ginline-points`
`-gno-inline-points`

Generate extended debug information for inlined functions. Location view tracking markers are inserted at inlined entry points, so that address and view numbers can be computed and output in debug information. This can be enabled independently of location views, in which case the view numbers won't be output, but it can only be enabled along with statement frontiers, and it is only enabled by default if location views are enabled.

`-gz`[=*type*]

Produce compressed debug sections in DWARF format, if that is supported. If *type* is not given, the default type depends on the capabilities of the assembler and linker used. *type* may be one of '`none`' (don't compress debug sections), '`zlib`' (use zlib compression in ELF gABI format), or '`zlib-gnu`' (use zlib compression in traditional GNU format). If the linker doesn't support writing compressed debug sections, the option is rejected. Otherwise, if the assembler does not support them, '`-gz`' is silently ignored when producing object files.

`-femit-struct-debug-baseonly`

Emit debug information for struct-like types only when the base name of the compilation source file matches the base name of file in which the struct is defined.

This option substantially reduces the size of debugging information, but at significant potential loss in type information to the debugger. See '`-femit-struct-debug-reduced`' for a less aggressive option. See '`-femit-struct-debug-detailed`' for more detailed control.

This option works only with DWARF debug output.

`-femit-struct-debug-reduced`

Emit debug information for struct-like types only when the base name of the compilation source file matches the base name of file in which the type is defined, unless the struct is a template or defined in a system header.

This option significantly reduces the size of debugging information, with some potential loss in type information to the debugger. See '`-femit-struct-debug-baseonly`' for a more aggressive option. See '`-femit-struct-debug-detailed`' for more detailed control.

This option works only with DWARF debug output.

`-femit-struct-debug-detailed`[`=`*`spec-list`*]

Specify the struct-like types for which the compiler generates debug information. The intent is to reduce duplicate struct debug information between different object files within the same program.

This option is a detailed version of '`-femit-struct-debug-reduced`' and '`-femit-struct-debug-baseonly`', which serves for most needs.

A specification has the syntax
[`dir:`'|'`ind:`'][`ord:`'|'`gen:`'](`any`'|'`sys`'|'`base`'|'`none`')

The optional first word limits the specification to structs that are used directly (`dir:`) or used indirectly (`ind:`). A struct type is used directly when it is the type of a variable, member. Indirect uses arise through pointers to structs. That is, when use of an incomplete struct is valid, the use is indirect. An example is '`struct one direct; struct two * indirect;`'.

The optional second word limits the specification to ordinary structs (`ord:`) or generic structs (`gen:`). Generic structs are a bit complicated to explain. For C++, these are non-explicit specializations of template classes, or non-template classes within the above. Other programming languages have generics, but '`-femit-struct-debug-detailed`' does not yet implement them.

The third word specifies the source files for those structs for which the compiler should emit debug information. The values '`none`' and '`any`' have the normal meaning. The value '`base`' means that the base of name of the file in which the type declaration appears must match the base of the name of the main compilation file. In practice, this means that when compiling '`foo.c`', debug information is generated for types declared in that file and '`foo.h`', but not other header files. The value '`sys`' means those types satisfying '`base`' or declared in system or compiler headers.

You may need to experiment to determine the best settings for your application.

The default is '`-femit-struct-debug-detailed=all`'.

This option works only with DWARF debug output.

`-fno-dwarf2-cfi-asm`
: Emit DWARF unwind info as compiler generated `.eh_frame` section instead of using GAS `.cfi_*` directives.

`-fno-eliminate-unused-debug-types`
: Normally, when producing DWARF output, GCC avoids producing debug symbol output for types that are nowhere used in the source file being compiled. Sometimes it is useful to have GCC emit debugging information for all types declared in a compilation unit, regardless of whether or not they are actually used in that compilation unit, for example if, in the debugger, you want to cast a value to a type that is not actually used in your program (but is declared). More often, however, this results in a significant amount of wasted space.

3.10 Options That Control Optimization

These options control various sorts of optimizations.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code.

Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

The compiler performs optimization based on the knowledge it has of the program. Compiling multiple files at once to a single output file mode allows the compiler to use information gained from all of the files when compiling each of them.

Not all optimizations are controlled directly by a flag. Only optimizations that have a flag are listed in this section.

Most optimizations are completely disabled at '`-O0`' or if an '`-O`' level is not set on the command line, even if individual optimization flags are specified. Similarly, '`-Og`' suppresses many optimization passes.

Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each '`-O`' level than those listed here. You can invoke GCC with '`-Q --help=optimizers`' to find out the exact set of optimizations that are enabled at each level. See Section 3.2 [Overall Options], page 31, for examples.

`-O`
`-O1`
: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With '`-O`', the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

'`-O`' turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

`-O2` Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to '`-O`', this option increases both compilation time and the performance of the generated code.

'`-O2`' turns on all optimization flags specified by '`-O`'. It also turns on the following optimization flags:

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fcaller-saves
-fcode-hoisting
-fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks
```

```
-fdelete-null-pointer-checks
-fdevirtualize -fdevirtualize-speculatively
-fexpensive-optimizations
-fgcse -fgcse-lm
-fhoist-adjacent-loads
-finline-small-functions
-findirect-inlining
-fipa-bit-cp -fipa-cp -fipa-icf
-fipa-ra -fipa-sra -fipa-vrp
-fisolate-erroneous-paths-dereference
-flra-remat
-foptimize-sibling-calls
-foptimize-strlen
-fpartial-inlining
-fpeephole2
-freorder-blocks-algorithm=stc
-freorder-blocks-and-partition -freorder-functions
-frerun-cse-after-loop
-fschedule-insns -fschedule-insns2
-fsched-interblock -fsched-spec
-fstore-merging
-fstrict-aliasing
-fthread-jumps
-ftree-builtin-call-dce
-ftree-pre
-ftree-switch-conversion -ftree-tail-merge
-ftree-vrp
```

Please note the warning under '`-fgcse`' about invoking '`-O2`' on programs that use computed gotos.

`-O3` Optimize yet more. '`-O3`' turns on all optimizations specified by '`-O2`' and also turns on the following optimization flags:

```
-fgcse-after-reload
-finline-functions
-fipa-cp-clone -floop-interchange
-floop-unroll-and-jam
-fpeel-loops
-fpredictive-commoning
-fsplit-paths
-ftree-loop-distribute-patterns
-ftree-loop-distribution
-ftree-loop-vectorize
-ftree-partial-pre
-ftree-slp-vectorize
-funswitch-loops
-fvect-cost-model
-fversion-loops-for-strides
```

`-O0` Reduce compilation time and make debugging produce the expected results. This is the default.

`-Os` Optimize for size. '`-Os`' enables all '`-O2`' optimizations except those that often increase code size:

```
-falign-functions -falign-jumps
-falign-labels -falign-loops
-fprefetch-loop-arrays -freorder-blocks-algorithm=stc
```

It also enables '-finline-functions', causes the compiler to tune for code size rather than execution speed, and performs further optimizations designed to reduce code size.

-Ofast Disregard strict standards compliance. '-Ofast' enables all '-O3' optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on '-ffast-math' and the Fortran-specific '-fstack-arrays', unless '-fmax-stack-var-size' is specified, and '-fno-protect-parens'.

-Og Optimize debugging experience. '-Og' should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience. It is a better choice than '-O0' for producing debuggable code because some compiler passes that collect debug information are disabled at '-O0'.

Like '-O0', '-Og' completely disables a number of optimization passes so that individual options controlling them have no effect. Otherwise '-Og' enables all '-O1' optimization flags except for those that may interfere with debugging:

```
-fbranch-count-reg -fdelayed-branch
-fif-conversion -fif-conversion2
-finline-functions-called-once
-fmove-loop-invariants -fssa-phiopt
-ftree-bit-ccp -ftree-pta -ftree-sra
```

If you use multiple '-O' options, with or without level numbers, the last such option is the one that is effective.

Options of the form '-f*flag*' specify machine-independent flags. Most flags have both positive and negative forms; the negative form of '-ffoo' is '-fno-foo'. In the table below, only one of the forms is listed—the one you typically use. You can figure out the other form by either removing 'no-' or adding it.

The following options control specific optimizations. They are either activated by '-O' options or are related to ones that are. You can use the following flags in the rare cases when "fine-tuning" of optimizations to be performed is desired.

-fno-defer-pop
For machines that must pop arguments after a function call, always pop the arguments as soon as each function returns. At levels '-O1' and higher, '-fdefer-pop' is the default; this allows the compiler to let arguments accumulate on the stack for several function calls and pop them all at once.

-fforward-propagate
Perform a forward propagation pass on RTL. The pass tries to combine two instructions and checks if the result can be simplified. If loop unrolling is active, two passes are performed and the second is scheduled after loop unrolling.

This option is enabled by default at optimization levels '-O', '-O2', '-O3', '-Os'.

-ffp-contract=*style*
'-ffp-contract=off' disables floating-point expression contraction. '-ffp-contract=fast' enables floating-point expression contraction such as forming of fused multiply-add operations if the target has native support for them. '-ffp-contract=on' enables floating-point expression contraction if

allowed by the language standard. This is currently not implemented and treated equal to '`-ffp-contract=off`'.

The default is '`-ffp-contract=fast`'.

`-fomit-frame-pointer`

Omit the frame pointer in functions that don't need one. This avoids the instructions to save, set up and restore the frame pointer; on many targets it also makes an extra register available.

On some targets this flag has no effect because the standard calling sequence always uses a frame pointer, so it cannot be omitted.

Note that '`-fno-omit-frame-pointer`' doesn't guarantee the frame pointer is used in all functions. Several targets always omit the frame pointer in leaf functions.

Enabled by default at '`-O`' and higher.

`-foptimize-sibling-calls`

Optimize sibling and tail recursive calls.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-foptimize-strlen`

Optimize various standard C string functions (e.g. `strlen`, `strchr` or `strcpy`) and their `_FORTIFY_SOURCE` counterparts into faster alternatives.

Enabled at levels '`-O2`', '`-O3`'.

`-fno-inline`

Do not expand any functions inline apart from those marked with the `always_inline` attribute. This is the default when not optimizing.

Single functions can be exempted from inlining by marking them with the `noinline` attribute.

`-finline-small-functions`

Integrate functions into their callers when their body is smaller than expected function call code (so overall size of program gets smaller). The compiler heuristically decides which functions are simple enough to be worth integrating in this way. This inlining applies to all functions, even those not declared inline.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-findirect-inlining`

Inline also indirect calls that are discovered to be known at compile time thanks to previous inlining. This option has any effect only when inlining itself is turned on by the '`-finline-functions`' or '`-finline-small-functions`' options.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-finline-functions`

Consider all functions for inlining, even if they are not declared inline. The compiler heuristically decides which functions are worth integrating in this way.

If all calls to a given function are integrated, and the function is declared `static`, then the function is normally not output as assembler code in its own right.

Enabled at levels '-O3', '-Os'. Also enabled by '-fprofile-use' and '-fauto-profile'.

-finline-functions-called-once

Consider all static functions called once for inlining into their caller even if they are not marked inline. If a call to a given function is integrated, then the function is not output as assembler code in its own right.

Enabled at levels '-O1', '-O2', '-O3' and '-Os', but not '-Og'.

-fearly-inlining

Inline functions marked by always_inline and functions whose body seems smaller than the function call overhead early before doing '-fprofile-generate' instrumentation and real inlining pass. Doing so makes profiling significantly cheaper and usually inlining faster on programs having large chains of nested wrapper functions.

Enabled by default.

-fipa-sra

Perform interprocedural scalar replacement of aggregates, removal of unused parameters and replacement of parameters passed by reference by parameters passed by value.

Enabled at levels '-O2', '-O3' and '-Os'.

-finline-limit=*n*

By default, GCC limits the size of functions that can be inlined. This flag allows coarse control of this limit. *n* is the size of functions that can be inlined in number of pseudo instructions.

Inlining is actually controlled by a number of parameters, which may be specified individually by using '--param *name=value*'. The '-finline-limit=*n*' option sets some of these parameters as follows:

max-inline-insns-single
: is set to $n/2$.

max-inline-insns-auto
: is set to $n/2$.

See below for a documentation of the individual parameters controlling inlining and for the defaults of these parameters.

Note: there may be no value to '-finline-limit' that results in default behavior.

Note: pseudo instruction represents, in this particular context, an abstract measurement of function's size. In no way does it represent a count of assembly instructions and as such its exact meaning might change from one release to an another.

-fno-keep-inline-dllexport

This is a more fine-grained version of '-fkeep-inline-functions', which applies only to functions that are declared using the dllexport attribute or declspec. See Section 6.33 [Declaring Attributes of Functions], page 477.

`-fkeep-inline-functions`
In C, emit `static` functions that are declared `inline` into the object file, even if the function has been inlined into all of its callers. This switch does not affect functions using the `extern inline` extension in GNU C90. In C++, emit any and all inline functions into the object file.

`-fkeep-static-functions`
Emit `static` functions into the object file, even if the function is never used.

`-fkeep-static-consts`
Emit variables declared `static const` when optimization isn't turned on, even if the variables aren't referenced.

GCC enables this option by default. If you want to force the compiler to check if a variable is referenced, regardless of whether or not optimization is turned on, use the '`-fno-keep-static-consts`' option.

`-fmerge-constants`
Attempt to merge identical constants (string constants and floating-point constants) across compilation units.

This option is the default for optimized compilation if the assembler and linker support it. Use '`-fno-merge-constants`' to inhibit this behavior.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-fmerge-all-constants`
Attempt to merge identical constants and identical variables.

This option implies '`-fmerge-constants`'. In addition to '`-fmerge-constants`' this considers e.g. even constant initialized arrays or initialized constant variables with integral or floating-point types. Languages like C or C++ require each variable, including multiple instances of the same variable in recursive calls, to have distinct locations, so using this option results in non-conforming behavior.

`-fmodulo-sched`
Perform swing modulo scheduling immediately before the first scheduling pass. This pass looks at innermost loops and reorders their instructions by overlapping different iterations.

`-fmodulo-sched-allow-regmoves`
Perform more aggressive SMS-based modulo scheduling with register moves allowed. By setting this flag certain anti-dependences edges are deleted, which triggers the generation of reg-moves based on the life-range analysis. This option is effective only with '`-fmodulo-sched`' enabled.

`-fno-branch-count-reg`
Disable the optimization pass that scans for opportunities to use "decrement and branch" instructions on a count register instead of instruction sequences that decrement a register, compare it against zero, and then branch based upon the result. This option is only meaningful on architectures that support such instructions, which include x86, PowerPC, IA-64 and S/390. Note that the '`-fno-branch-count-reg`' option doesn't remove the decrement and branch

instructions from the generated instruction stream introduced by other optimization passes.

The default is '`-fbranch-count-reg`' at '`-O1`' and higher, except for '`-Og`'.

`-fno-function-cse`

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

The default is '`-ffunction-cse`'

`-fno-zero-initialized-in-bss`

If the target supports a BSS section, GCC by default puts variables that are initialized to zero into BSS. This can save space in the resulting code.

This option turns off this behavior because some programs explicitly rely on variables going to the data section—e.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

The default is '`-fzero-initialized-in-bss`'.

`-fthread-jumps`

Perform optimizations that check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fsplit-wide-types`

When using a type that occupies multiple registers, such as `long long` on a 32-bit system, split the registers apart and allocate them independently. This normally generates better code for those types, but may make debugging more difficult.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-fcse-follow-jumps`

In common subexpression elimination (CSE), scan through jump instructions when the target of the jump is not reached by any other path. For example, when CSE encounters an `if` statement with an `else` clause, CSE follows the jump when the condition tested is false.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fcse-skip-blocks`

This is similar to '`-fcse-follow-jumps`', but causes CSE to follow jumps that conditionally skip over blocks. When CSE encounters a simple `if` statement with no else clause, '`-fcse-skip-blocks`' causes CSE to follow the jump around the body of the `if`.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-frerun-cse-after-loop`
: Re-run common subexpression elimination after loop optimizations are performed.

  Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fgcse`
: Perform a global common subexpression elimination pass. This pass also performs global constant and copy propagation.

  Note: When compiling a program using computed gotos, a GCC extension, you may get better run-time performance if you disable the global common subexpression elimination pass by adding '`-fno-gcse`' to the command line.

  Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fgcse-lm`
: When '`-fgcse-lm`' is enabled, global common subexpression elimination attempts to move loads that are only killed by stores into themselves. This allows a loop containing a load/store sequence to be changed to a load outside the loop, and a copy/store within the loop.

  Enabled by default when '`-fgcse`' is enabled.

`-fgcse-sm`
: When '`-fgcse-sm`' is enabled, a store motion pass is run after global common subexpression elimination. This pass attempts to move stores out of loops. When used in conjunction with '`-fgcse-lm`', loops containing a load/store sequence can be changed to a load before the loop and a store after the loop.

  Not enabled at any optimization level.

`-fgcse-las`
: When '`-fgcse-las`' is enabled, the global common subexpression elimination pass eliminates redundant loads that come after stores to the same memory location (both partial and full redundancies).

  Not enabled at any optimization level.

`-fgcse-after-reload`
: When '`-fgcse-after-reload`' is enabled, a redundant load elimination pass is performed after reload. The purpose of this pass is to clean up redundant spilling.

  Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-faggressive-loop-optimizations`
: This option tells the loop optimizer to use language constraints to derive bounds for the number of iterations of a loop. This assumes that loop code does not invoke undefined behavior by for example causing signed integer overflows or out-of-bound array accesses. The bounds for the number of iterations of a loop are used to guide loop unrolling and peeling and loop exit test optimizations. This option is enabled by default.

`-funconstrained-commons`
: This option tells the compiler that variables declared in common blocks (e.g. Fortran) may later be overridden with longer trailing arrays. This prevents certain optimizations that depend on knowing the array bounds.

`-fcrossjumping`
Perform cross-jumping transformation. This transformation unifies equivalent code and saves code size. The resulting code may or may not perform better than without cross-jumping.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fauto-inc-dec`
Combine increments or decrements of addresses with memory accesses. This pass is always skipped on architectures that do not have instructions to support this. Enabled by default at '`-O`' and higher on architectures that support this.

`-fdce` Perform dead code elimination (DCE) on RTL. Enabled by default at '`-O`' and higher.

`-fdse` Perform dead store elimination (DSE) on RTL. Enabled by default at '`-O`' and higher.

`-fif-conversion`
Attempt to transform conditional jumps into branch-less equivalents. This includes use of conditional moves, min, max, set flags and abs instructions, and some tricks doable by standard arithmetics. The use of conditional execution on chips where it is available is controlled by '`-fif-conversion2`'.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`', but not with '`-Og`'.

`-fif-conversion2`
Use conditional execution (where available) to transform conditional jumps into branch-less equivalents.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`', but not with '`-Og`'.

`-fdeclone-ctor-dtor`
The C++ ABI requires multiple entry points for constructors and destructors: one for a base subobject, one for a complete object, and one for a virtual destructor that calls operator delete afterwards. For a hierarchy with virtual bases, the base and complete variants are clones, which means two copies of the function. With this option, the base and complete variants are changed to be thunks that call a common implementation.

Enabled by '`-Os`'.

`-fdelete-null-pointer-checks`
Assume that programs cannot safely dereference null pointers, and that no code or data element resides at address zero. This option enables simple constant folding optimizations at all optimization levels. In addition, other optimization passes in GCC use this flag to control global dataflow analyses that eliminate useless checks for null pointers; these assume that a memory access to address zero always results in a trap, so that if a pointer is checked after it has already been dereferenced, it cannot be null.

Note however that in some environments this assumption is not true. Use '`-fno-delete-null-pointer-checks`' to disable this optimization for programs that depend on that behavior.

This option is enabled by default on most targets. On Nios II ELF, it defaults to off. On AVR, CR16, and MSP430, this option is completely disabled.

Passes that use the dataflow information are enabled independently at different optimization levels.

`-fdevirtualize`
: Attempt to convert calls to virtual functions to direct calls. This is done both within a procedure and interprocedurally as part of indirect inlining ('`-findirect-inlining`') and interprocedural constant propagation ('`-fipa-cp`'). Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fdevirtualize-speculatively`
: Attempt to convert calls to virtual functions to speculative direct calls. Based on the analysis of the type inheritance graph, determine for a given call the set of likely targets. If the set is small, preferably of size 1, change the call into a conditional deciding between direct and indirect calls. The speculative calls enable more optimizations, such as inlining. When they seem useless after further optimization, they are converted back into original form.

`-fdevirtualize-at-ltrans`
: Stream extra information needed for aggressive devirtualization when running the link-time optimizer in local transformation mode. This option enables more devirtualization but significantly increases the size of streamed data. For this reason it is disabled by default.

`-fexpensive-optimizations`
: Perform a number of minor optimizations that are relatively expensive.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-free` Attempt to remove redundant extension instructions. This is especially helpful for the x86-64 architecture, which implicitly zero-extends in 64-bit registers after writing to their lower 32-bit half.

Enabled for Alpha, AArch64 and x86 at levels '`-O2`', '`-O3`', '`-Os`'.

`-fno-lifetime-dse`
: In C++ the value of an object is only affected by changes within its lifetime: when the constructor begins, the object has an indeterminate value, and any changes during the lifetime of the object are dead when the object is destroyed. Normally dead store elimination will take advantage of this; if your code relies on the value of the object storage persisting beyond the lifetime of the object, you can use this flag to disable this optimization. To preserve stores before the constructor starts (e.g. because your operator new clears the object storage) but still treat the object as dead after the destructor you, can use '`-flifetime-dse=1`'. The default behavior can be explicitly selected with '`-flifetime-dse=2`'. '`-flifetime-dse=0`' is equivalent to '`-fno-lifetime-dse`'.

`-flive-range-shrinkage`
: Attempt to decrease register pressure through register live range shrinkage. This is helpful for fast processors with small or moderate size register sets.

`-fira-algorithm=`*algorithm*

Use the specified coloring algorithm for the integrated register allocator. The *algorithm* argument can be '`priority`', which specifies Chow's priority coloring, or '`CB`', which specifies Chaitin-Briggs coloring. Chaitin-Briggs coloring is not implemented for all architectures, but for those targets that do support it, it is the default because it generates better code.

`-fira-region=`*region*

Use specified regions for the integrated register allocator. The *region* argument should be one of the following:

'`all`' Use all loops as register allocation regions. This can give the best results for machines with a small and/or irregular register set.

'`mixed`' Use all loops except for loops with small register pressure as the regions. This value usually gives the best results in most cases and for most architectures, and is enabled by default when compiling with optimization for speed ('`-O`', '`-O2`', . . .).

'`one`' Use all functions as a single region. This typically results in the smallest code size, and is enabled by default for '`-Os`' or '`-O0`'.

`-fira-hoist-pressure`

Use IRA to evaluate register pressure in the code hoisting pass for decisions to hoist expressions. This option usually results in smaller code, but it can slow the compiler down.

This option is enabled at level '`-Os`' for all targets.

`-fira-loop-pressure`

Use IRA to evaluate register pressure in loops for decisions to move loop invariants. This option usually results in generation of faster and smaller code on machines with large register files (>= 32 registers), but it can slow the compiler down.

This option is enabled at level '`-O3`' for some targets.

`-fno-ira-share-save-slots`

Disable sharing of stack slots used for saving call-used hard registers living through a call. Each hard register gets a separate stack slot, and as a result function stack frames are larger.

`-fno-ira-share-spill-slots`

Disable sharing of stack slots allocated for pseudo-registers. Each pseudo-register that does not get a hard register gets a separate stack slot, and as a result function stack frames are larger.

`-flra-remat`

Enable CFG-sensitive rematerialization in LRA. Instead of loading values of spilled pseudos, LRA tries to rematerialize (recalculate) values if it is profitable.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fdelayed-branch`

If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.

Enabled at levels '-O', '-O2', '-O3', '-Os', but not at '-Og'.

-fschedule-insns

If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating-point instruction is required.

Enabled at levels '-O2', '-O3'.

-fschedule-insns2

Similar to '-fschedule-insns', but requests an additional pass of instruction scheduling after register allocation has been done. This is especially useful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Enabled at levels '-O2', '-O3', '-Os'.

-fno-sched-interblock

Disable instruction scheduling across basic blocks, which is normally enabled when scheduling before register allocation, i.e. with '-fschedule-insns' or at '-O2' or higher.

-fno-sched-spec

Disable speculative motion of non-load instructions, which is normally enabled when scheduling before register allocation, i.e. with '-fschedule-insns' or at '-O2' or higher.

-fsched-pressure

Enable register pressure sensitive insn scheduling before register allocation. This only makes sense when scheduling before register allocation is enabled, i.e. with '-fschedule-insns' or at '-O2' or higher. Usage of this option can improve the generated code and decrease its size by preventing register pressure increase above the number of available hard registers and subsequent spills in register allocation.

-fsched-spec-load

Allow speculative motion of some load instructions. This only makes sense when scheduling before register allocation, i.e. with '-fschedule-insns' or at '-O2' or higher.

-fsched-spec-load-dangerous

Allow speculative motion of more load instructions. This only makes sense when scheduling before register allocation, i.e. with '-fschedule-insns' or at '-O2' or higher.

-fsched-stalled-insns
-fsched-stalled-insns=*n*

Define how many insns (if any) can be moved prematurely from the queue of stalled insns into the ready list during the second scheduling pass. '-fno-sched-stalled-insns' means that no insns are moved prematurely, '-fsched-stalled-insns=0' means there is no limit on how many queued

insns can be moved prematurely. '-fsched-stalled-insns' without a value is equivalent to '-fsched-stalled-insns=1'.

-fsched-stalled-insns-dep
-fsched-stalled-insns-dep=*n*

Define how many insn groups (cycles) are examined for a dependency on a stalled insn that is a candidate for premature removal from the queue of stalled insns. This has an effect only during the second scheduling pass, and only if '-fsched-stalled-insns' is used. '-fno-sched-stalled-insns-dep' is equivalent to '-fsched-stalled-insns-dep=0'. '-fsched-stalled-insns-dep' without a value is equivalent to '-fsched-stalled-insns-dep=1'.

-fsched2-use-superblocks

When scheduling after register allocation, use superblock scheduling. This allows motion across basic block boundaries, resulting in faster schedules. This option is experimental, as not all machine descriptions used by GCC model the CPU closely enough to avoid unreliable results from the algorithm.

This only makes sense when scheduling after register allocation, i.e. with '-fschedule-insns2' or at '-O2' or higher.

-fsched-group-heuristic

Enable the group heuristic in the scheduler. This heuristic favors the instruction that belongs to a schedule group. This is enabled by default when scheduling is enabled, i.e. with '-fschedule-insns' or '-fschedule-insns2' or at '-O2' or higher.

-fsched-critical-path-heuristic

Enable the critical-path heuristic in the scheduler. This heuristic favors instructions on the critical path. This is enabled by default when scheduling is enabled, i.e. with '-fschedule-insns' or '-fschedule-insns2' or at '-O2' or higher.

-fsched-spec-insn-heuristic

Enable the speculative instruction heuristic in the scheduler. This heuristic favors speculative instructions with greater dependency weakness. This is enabled by default when scheduling is enabled, i.e. with '-fschedule-insns' or '-fschedule-insns2' or at '-O2' or higher.

-fsched-rank-heuristic

Enable the rank heuristic in the scheduler. This heuristic favors the instruction belonging to a basic block with greater size or frequency. This is enabled by default when scheduling is enabled, i.e. with '-fschedule-insns' or '-fschedule-insns2' or at '-O2' or higher.

-fsched-last-insn-heuristic

Enable the last-instruction heuristic in the scheduler. This heuristic favors the instruction that is less dependent on the last instruction scheduled. This is enabled by default when scheduling is enabled, i.e. with '-fschedule-insns' or '-fschedule-insns2' or at '-O2' or higher.

`-fsched-dep-count-heuristic`
Enable the dependent-count heuristic in the scheduler. This heuristic favors the instruction that has more instructions depending on it. This is enabled by default when scheduling is enabled, i.e. with '`-fschedule-insns`' or '`-fschedule-insns2`' or at '`-O2`' or higher.

`-freschedule-modulo-scheduled-loops`
Modulo scheduling is performed before traditional scheduling. If a loop is modulo scheduled, later scheduling passes may change its schedule. Use this option to control that behavior.

`-fselective-scheduling`
Schedule instructions using selective scheduling algorithm. Selective scheduling runs instead of the first scheduler pass.

`-fselective-scheduling2`
Schedule instructions using selective scheduling algorithm. Selective scheduling runs instead of the second scheduler pass.

`-fsel-sched-pipelining`
Enable software pipelining of innermost loops during selective scheduling. This option has no effect unless one of '`-fselective-scheduling`' or '`-fselective-scheduling2`' is turned on.

`-fsel-sched-pipelining-outer-loops`
When pipelining loops during selective scheduling, also pipeline outer loops. This option has no effect unless '`-fsel-sched-pipelining`' is turned on.

`-fsemantic-interposition`
Some object formats, like ELF, allow interposing of symbols by the dynamic linker. This means that for symbols exported from the DSO, the compiler cannot perform interprocedural propagation, inlining and other optimizations in anticipation that the function or variable in question may change. While this feature is useful, for example, to rewrite memory allocation functions by a debugging implementation, it is expensive in the terms of code quality. With '`-fno-semantic-interposition`' the compiler assumes that if interposition happens for functions the overwriting function will have precisely the same semantics (and side effects). Similarly if interposition happens for variables, the constructor of the variable will be the same. The flag has no effect for functions explicitly declared inline (where it is never allowed for interposition to change semantics) and for symbols explicitly declared weak.

`-fshrink-wrap`
Emit function prologues only before parts of the function that need it, rather than at the top of the function. This flag is enabled by default at '`-O`' and higher.

`-fshrink-wrap-separate`
Shrink-wrap separate parts of the prologue and epilogue separately, so that those parts are only executed when needed. This option is on by default, but has no effect unless '`-fshrink-wrap`' is also turned on and the target supports this.

`-fcaller-saves`
Enable allocation of values to registers that are clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code.

This option is always enabled by default on certain machines, usually those which have no call-preserved registers to use instead.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fcombine-stack-adjustments`
Tracks stack adjustments (pushes and pops) and stack memory references and then tries to find ways to combine them.

Enabled by default at '`-O1`' and higher.

`-fipa-ra` Use caller save registers for allocation if those registers are not used by any called function. In that case it is not necessary to save and restore them around calls. This is only possible if called functions are part of same compilation unit as current function and they are compiled before it.

Enabled at levels '`-O2`', '`-O3`', '`-Os`', however the option is disabled if generated code will be instrumented for profiling ('`-p`', or '`-pg`') or if callee's register usage cannot be known exactly (this happens on targets that do not expose prologues and epilogues in RTL).

`-fconserve-stack`
Attempt to minimize stack usage. The compiler attempts to use less stack space, even if that makes the program slower. This option implies setting the '`large-stack-frame`' parameter to 100 and the '`large-stack-frame-growth`' parameter to 400.

`-ftree-reassoc`
Perform reassociation on trees. This flag is enabled by default at '`-O`' and higher.

`-fcode-hoisting`
Perform code hoisting. Code hoisting tries to move the evaluation of expressions executed on all paths to the function exit as early as possible. This is especially useful as a code size optimization, but it often helps for code speed as well. This flag is enabled by default at '`-O2`' and higher.

`-ftree-pre`
Perform partial redundancy elimination (PRE) on trees. This flag is enabled by default at '`-O2`' and '`-O3`'.

`-ftree-partial-pre`
Make partial redundancy elimination (PRE) more aggressive. This flag is enabled by default at '`-O3`'.

`-ftree-forwprop`
Perform forward propagation on trees. This flag is enabled by default at '`-O`' and higher.

`-ftree-fre`
Perform full redundancy elimination (FRE) on trees. The difference between FRE and PRE is that FRE only considers expressions that are computed on all paths leading to the redundant computation. This analysis is faster than PRE, though it exposes fewer redundancies. This flag is enabled by default at '`-O`' and higher.

`-ftree-phiprop`
Perform hoisting of loads from conditional pointers on trees. This pass is enabled by default at '`-O`' and higher.

`-fhoist-adjacent-loads`
Speculatively hoist loads from both branches of an if-then-else if the loads are from adjacent locations in the same structure and the target architecture has a conditional move instruction. This flag is enabled by default at '`-O2`' and higher.

`-ftree-copy-prop`
Perform copy propagation on trees. This pass eliminates unnecessary copy operations. This flag is enabled by default at '`-O`' and higher.

`-fipa-pure-const`
Discover which functions are pure or constant. Enabled by default at '`-O`' and higher.

`-fipa-reference`
Discover which static variables do not escape the compilation unit. Enabled by default at '`-O`' and higher.

`-fipa-reference-addressable`
Discover read-only, write-only and non-addressable static variables. Enabled by default at '`-O`' and higher.

`-fipa-stack-alignment`
Reduce stack alignment on call sites if possible. Enabled by default.

`-fipa-pta`
Perform interprocedural pointer analysis and interprocedural modification and reference analysis. This option can cause excessive memory and compile-time usage on large compilation units. It is not enabled by default at any optimization level.

`-fipa-profile`
Perform interprocedural profile propagation. The functions called only from cold functions are marked as cold. Also functions executed once (such as `cold`, `noreturn`, static constructors or destructors) are identified. Cold functions and loop less parts of functions executed once are then optimized for size. Enabled by default at '`-O`' and higher.

`-fipa-cp` Perform interprocedural constant propagation. This optimization analyzes the program to determine when values passed to functions are constants and then optimizes accordingly. This optimization can substantially increase performance if the application has constants passed to functions. This flag is enabled

by default at '-O2', '-Os' and '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-fipa-cp-clone
: Perform function cloning to make interprocedural constant propagation stronger. When enabled, interprocedural constant propagation performs function cloning when externally visible function can be called with constant arguments. Because this optimization can create multiple copies of functions, it may significantly increase code size (see '--param ipcp-unit-growth=*value*'). This flag is enabled by default at '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-fipa-bit-cp
: When enabled, perform interprocedural bitwise constant propagation. This flag is enabled by default at '-O2' and by '-fprofile-use' and '-fauto-profile'. It requires that '-fipa-cp' is enabled.

-fipa-vrp
: When enabled, perform interprocedural propagation of value ranges. This flag is enabled by default at '-O2'. It requires that '-fipa-cp' is enabled.

-fipa-icf
: Perform Identical Code Folding for functions and read-only variables. The optimization reduces code size and may disturb unwind stacks by replacing a function by equivalent one with a different name. The optimization works more effectively with link-time optimization enabled.

  Although the behavior is similar to the Gold Linker's ICF optimization, GCC ICF works on different levels and thus the optimizations are not same - there are equivalences that are found only by GCC and equivalences found only by Gold.

  This flag is enabled by default at '-O2' and '-Os'.

-flive-patching=*level*
: Control GCC's optimizations to produce output suitable for live-patching.

  If the compiler's optimization uses a function's body or information extracted from its body to optimize/change another function, the latter is called an impacted function of the former. If a function is patched, its impacted functions should be patched too.

  The impacted functions are determined by the compiler's interprocedural optimizations. For example, a caller is impacted when inlining a function into its caller, cloning a function and changing its caller to call this new clone, or extracting a function's pureness/constness information to optimize its direct or indirect callers, etc.

  Usually, the more IPA optimizations enabled, the larger the number of impacted functions for each function. In order to control the number of impacted functions and more easily compute the list of impacted function, IPA optimizations can be partially enabled at two different levels.

  The *level* argument should be one of the following:

'inline-clone'

Only enable inlining and cloning optimizations, which includes inlining, cloning, interprocedural scalar replacement of aggregates and partial inlining. As a result, when patching a function, all its callers and its clones' callers are impacted, therefore need to be patched as well.

'-flive-patching=inline-clone' disables the following optimization flags:

```
-fwhole-program -fipa-pta -fipa-reference -fipa-ra
-fipa-icf -fipa-icf-functions -fipa-icf-variables
-fipa-bit-cp -fipa-vrp -fipa-pure-const -fipa-reference-addressable
-fipa-stack-alignment
```

'inline-only-static'

Only enable inlining of static functions. As a result, when patching a static function, all its callers are impacted and so need to be patched as well.

In addition to all the flags that '-flive-patching=inline-clone' disables, '-flive-patching=inline-only-static' disables the following additional optimization flags:

```
-fipa-cp-clone -fipa-sra -fpartial-inlining -fipa-cp
```

When '-flive-patching' is specified without any value, the default value is *inline-clone*.

This flag is disabled by default.

Note that '-flive-patching' is not supported with link-time optimization ('-flto').

-fisolate-erroneous-paths-dereference

Detect paths that trigger erroneous or undefined behavior due to dereferencing a null pointer. Isolate those paths from the main control flow and turn the statement with erroneous or undefined behavior into a trap. This flag is enabled by default at '-O2' and higher and depends on '-fdelete-null-pointer-checks' also being enabled.

-fisolate-erroneous-paths-attribute

Detect paths that trigger erroneous or undefined behavior due to a null value being used in a way forbidden by a returns_nonnull or nonnull attribute. Isolate those paths from the main control flow and turn the statement with erroneous or undefined behavior into a trap. This is not currently enabled, but may be enabled by '-O2' in the future.

-ftree-sink

Perform forward store motion on trees. This flag is enabled by default at '-O' and higher.

-ftree-bit-ccp

Perform sparse conditional bit constant propagation on trees and propagate pointer alignment information. This pass only operates on local scalar variables and is enabled by default at '-O1' and higher, except for '-Og'. It requires that '-ftree-ccp' is enabled.

`-ftree-ccp`
: Perform sparse conditional constant propagation (CCP) on trees. This pass only operates on local scalar variables and is enabled by default at '`-O`' and higher.

`-fssa-backprop`
: Propagate information about uses of a value up the definition chain in order to simplify the definitions. For example, this pass strips sign operations if the sign of a value never matters. The flag is enabled by default at '`-O`' and higher.

`-fssa-phiopt`
: Perform pattern matching on SSA PHI nodes to optimize conditional code. This pass is enabled by default at '`-O1`' and higher, except for '`-Og`'.

`-ftree-switch-conversion`
: Perform conversion of simple initializations in a switch to initializations from a scalar array. This flag is enabled by default at '`-O2`' and higher.

`-ftree-tail-merge`
: Look for identical code sequences. When found, replace one with a jump to the other. This optimization is known as tail merging or cross jumping. This flag is enabled by default at '`-O2`' and higher. The compilation time in this pass can be limited using '`max-tail-merge-comparisons`' parameter and '`max-tail-merge-iterations`' parameter.

`-ftree-dce`
: Perform dead code elimination (DCE) on trees. This flag is enabled by default at '`-O`' and higher.

`-ftree-builtin-call-dce`
: Perform conditional dead code elimination (DCE) for calls to built-in functions that may set `errno` but are otherwise free of side effects. This flag is enabled by default at '`-O2`' and higher if '`-Os`' is not also specified.

`-ftree-dominator-opts`
: Perform a variety of simple scalar cleanups (constant/copy propagation, redundancy elimination, range propagation and expression simplification) based on a dominator tree traversal. This also performs jump threading (to reduce jumps to jumps). This flag is enabled by default at '`-O`' and higher.

`-ftree-dse`
: Perform dead store elimination (DSE) on trees. A dead store is a store into a memory location that is later overwritten by another store without any intervening loads. In this case the earlier store can be deleted. This flag is enabled by default at '`-O`' and higher.

`-ftree-ch`
: Perform loop header copying on trees. This is beneficial since it increases effectiveness of code motion optimizations. It also saves one jump. This flag is enabled by default at '`-O`' and higher. It is not enabled for '`-Os`', since it usually increases code size.

`-ftree-loop-optimize`
Perform loop optimizations on trees. This flag is enabled by default at '`-O`' and higher.

`-ftree-loop-linear`
`-floop-strip-mine`
`-floop-block`
Perform loop nest optimizations. Same as '`-floop-nest-optimize`'. To use this code transformation, GCC has to be configured with '`--with-isl`' to enable the Graphite loop transformation infrastructure.

`-fgraphite-identity`
Enable the identity transformation for graphite. For every SCoP we generate the polyhedral representation and transform it back to gimple. Using '`-fgraphite-identity`' we can check the costs or benefits of the GIMPLE -> GRAPHITE -> GIMPLE transformation. Some minimal optimizations are also performed by the code generator isl, like index splitting and dead code elimination in loops.

`-floop-nest-optimize`
Enable the isl based loop nest optimizer. This is a generic loop nest optimizer based on the Pluto optimization algorithms. It calculates a loop structure optimized for data-locality and parallelism. This option is experimental.

`-floop-parallelize-all`
Use the Graphite data dependence analysis to identify loops that can be parallelized. Parallelize all the loops that can be analyzed to not contain loop carried dependences without checking that it is profitable to parallelize the loops.

`-ftree-coalesce-vars`
While transforming the program out of the SSA representation, attempt to reduce copying by coalescing versions of different user-defined variables, instead of just compiler temporaries. This may severely limit the ability to debug an optimized program compiled with '`-fno-var-tracking-assignments`'. In the negated form, this flag prevents SSA coalescing of user variables. This option is enabled by default if optimization is enabled, and it does very little otherwise.

`-ftree-loop-if-convert`
Attempt to transform conditional jumps in the innermost loops to branch-less equivalents. The intent is to remove control-flow from the innermost loops in order to improve the ability of the vectorization pass to handle these loops. This is enabled by default if vectorization is enabled.

`-ftree-loop-distribution`
Perform loop distribution. This flag can improve cache performance on big loop bodies and allow further loop optimizations, like parallelization or vectorization, to take place. For example, the loop

```
DO I = 1, N
  A(I) = B(I) + C
  D(I) = E(I) * F
ENDDO
```

is transformed to

```
DO I = 1, N
   A(I) = B(I) + C
ENDDO
DO I = 1, N
   D(I) = E(I) * F
ENDDO
```

This flag is enabled by default at '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-ftree-loop-distribute-patterns

Perform loop distribution of patterns that can be code generated with calls to a library. This flag is enabled by default at '-O3', and by '-fprofile-use' and '-fauto-profile'.

This pass distributes the initialization loops and generates a call to memset zero. For example, the loop

```
DO I = 1, N
  A(I) = 0
  B(I) = A(I) + I
ENDDO
```

is transformed to

```
DO I = 1, N
   A(I) = 0
ENDDO
DO I = 1, N
   B(I) = A(I) + I
ENDDO
```

and the initialization loop is transformed into a call to memset zero. This flag is enabled by default at '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-floop-interchange

Perform loop interchange outside of graphite. This flag can improve cache performance on loop nest and allow further loop optimizations, like vectorization, to take place. For example, the loop

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

is transformed to

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

This flag is enabled by default at '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-floop-unroll-and-jam

Apply unroll and jam transformations on feasible loops. In a loop nest this unrolls the outer loop by some factor and fuses the resulting multiple inner loops. This flag is enabled by default at '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

`-ftree-loop-im`
Perform loop invariant motion on trees. This pass moves only invariants that are hard to handle at RTL level (function calls, operations that expand to non-trivial sequences of insns). With '`-funswitch-loops`' it also moves operands of conditions that are invariant out of the loop, so that we can use just trivial invariantness analysis in loop unswitching. The pass also includes store motion.

`-ftree-loop-ivcanon`
Create a canonical counter for number of iterations in loops for which determining number of iterations requires complicated analysis. Later optimizations then may determine the number easily. Useful especially in connection with unrolling.

`-ftree-scev-cprop`
Perform final value replacement. If a variable is modified in a loop in such a way that its value when exiting the loop can be determined using only its initial value and the number of loop iterations, replace uses of the final value by such a computation, provided it is sufficiently cheap. This reduces data dependencies and may allow further simplifications. Enabled by default at '`-O`' and higher.

`-fivopts` Perform induction variable optimizations (strength reduction, induction variable merging and induction variable elimination) on trees.

`-ftree-parallelize-loops=n`
Parallelize loops, i.e., split their iteration space to run in n threads. This is only possible for loops whose iterations are independent and can be arbitrarily reordered. The optimization is only profitable on multiprocessor machines, for loops that are CPU-intensive, rather than constrained e.g. by memory bandwidth. This option implies '`-pthread`', and thus is only supported on targets that have support for '`-pthread`'.

`-ftree-pta`
Perform function-local points-to analysis on trees. This flag is enabled by default at '`-O1`' and higher, except for '`-Og`'.

`-ftree-sra`
Perform scalar replacement of aggregates. This pass replaces structure references with scalars to prevent committing structures to memory too early. This flag is enabled by default at '`-O1`' and higher, except for '`-Og`'.

`-fstore-merging`
Perform merging of narrow stores to consecutive memory addresses. This pass merges contiguous stores of immediate values narrower than a word into fewer wider stores to reduce the number of instructions. This is enabled by default at '`-O2`' and higher as well as '`-Os`'.

`-ftree-ter`
Perform temporary expression replacement during the SSA->normal phase. Single use/single def temporaries are replaced at their use location with their defining expression. This results in non-GIMPLE code, but gives the expanders much more complex trees to work on resulting in better RTL generation. This is enabled by default at '`-O`' and higher.

`-ftree-slsr`
: Perform straight-line strength reduction on trees. This recognizes related expressions involving multiplications and replaces them by less expensive calculations when possible. This is enabled by default at '`-O`' and higher.

`-ftree-vectorize`
: Perform vectorization on trees. This flag enables '`-ftree-loop-vectorize`' and '`-ftree-slp-vectorize`' if not explicitly specified.

`-ftree-loop-vectorize`
: Perform loop vectorization on trees. This flag is enabled by default at '`-O3`' and by '`-ftree-vectorize`', '`-fprofile-use`', and '`-fauto-profile`'.

`-ftree-slp-vectorize`
: Perform basic block vectorization on trees. This flag is enabled by default at '`-O3`' and by '`-ftree-vectorize`', '`-fprofile-use`', and '`-fauto-profile`'.

`-fvect-cost-model=`*`model`*
: Alter the cost model used for vectorization. The *model* argument should be one of '`unlimited`', '`dynamic`' or '`cheap`'. With the '`unlimited`' model the vectorized code-path is assumed to be profitable while with the '`dynamic`' model a runtime check guards the vectorized code-path to enable it only for iteration counts that will likely execute faster than when executing the original scalar loop. The '`cheap`' model disables vectorization of loops where doing so would be cost prohibitive for example due to required runtime checks for data dependence or alignment but otherwise is equal to the '`dynamic`' model. The default cost model depends on other optimization flags and is either '`dynamic`' or '`cheap`'.

`-fsimd-cost-model=`*`model`*
: Alter the cost model used for vectorization of loops marked with the OpenMP simd directive. The *model* argument should be one of '`unlimited`', '`dynamic`', '`cheap`'. All values of *model* have the same meaning as described in '`-fvect-cost-model`' and by default a cost model defined with '`-fvect-cost-model`' is used.

`-ftree-vrp`
: Perform Value Range Propagation on trees. This is similar to the constant propagation pass, but instead of values, ranges of values are propagated. This allows the optimizers to remove unnecessary range checks like array bound checks and null pointer checks. This is enabled by default at '`-O2`' and higher. Null pointer check elimination is only done if '`-fdelete-null-pointer-checks`' is enabled.

`-fsplit-paths`
: Split paths leading to loop backedges. This can improve dead code elimination and common subexpression elimination. This is enabled by default at '`-O3`' and above.

`-fsplit-ivs-in-unroller`
: Enables expression of values of induction variables in later iterations of the unrolled loop using the value in the first iteration. This breaks long dependency chains, thus improving efficiency of the scheduling passes.

A combination of '-fweb' and CSE is often sufficient to obtain the same effect. However, that is not reliable in cases where the loop body is more complicated than a single basic block. It also does not work at all on some architectures due to restrictions in the CSE pass.

This optimization is enabled by default.

-fvariable-expansion-in-unroller
: With this option, the compiler creates multiple copies of some local variables when unrolling a loop, which can result in superior code.

-fpartial-inlining
: Inline parts of functions. This option has any effect only when inlining itself is turned on by the '-finline-functions' or '-finline-small-functions' options.

Enabled at levels '-O2', '-O3', '-Os'.

-fpredictive-commoning
: Perform predictive commoning optimization, i.e., reusing computations (especially memory loads and stores) performed in previous iterations of loops.

This option is enabled at level '-O3'. It is also enabled by '-fprofile-use' and '-fauto-profile'.

-fprefetch-loop-arrays
: If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

This option may generate better or worse code; results are highly dependent on the structure of loops within the source code.

Disabled at level '-Os'.

-fno-printf-return-value
: Do not substitute constants for known return value of formatted output functions such as sprintf, snprintf, vsprintf, and vsnprintf (but not printf of fprintf). This transformation allows GCC to optimize or even eliminate branches based on the known return value of these functions called with arguments that are either constant, or whose values are known to be in a range that makes determining the exact return value possible. For example, when '-fprintf-return-value' is in effect, both the branch and the body of the if statement (but not the call to snprint) can be optimized away when i is a 32-bit or smaller integer because the return value is guaranteed to be at most 8.

```
char buf[9];
if (snprintf (buf, "%08x", i) >= sizeof buf)
  ...
```

The '-fprintf-return-value' option relies on other optimizations and yields best results with '-O2' and above. It works in tandem with the '-Wformat-overflow' and '-Wformat-truncation' options. The '-fprintf-return-value' option is enabled by default.

`-fno-peephole`
`-fno-peephole2`

Disable any machine-specific peephole optimizations. The difference between '`-fno-peephole`' and '`-fno-peephole2`' is in how they are implemented in the compiler; some targets use one, some use the other, a few use both.

'`-fpeephole`' is enabled by default. '`-fpeephole2`' enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fno-guess-branch-probability`

Do not guess branch probabilities using heuristics.

GCC uses heuristics to guess branch probabilities if they are not provided by profiling feedback ('`-fprofile-arcs`'). These heuristics are based on the control flow graph. If some branch probabilities are specified by `__builtin_expect`, then the heuristics are used to guess branch probabilities for the rest of the control flow graph, taking the `__builtin_expect` info into account. The interactions between the heuristics and `__builtin_expect` can be complex, and in some cases, it may be useful to disable the heuristics so that the effects of `__builtin_expect` are easier to understand.

It is also possible to specify expected probability of the expression with `__builtin_expect_with_probability` built-in function.

The default is '`-fguess-branch-probability`' at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-freorder-blocks`

Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-freorder-blocks-algorithm=`*algorithm*

Use the specified algorithm for basic block reordering. The *algorithm* argument can be '`simple`', which does not increase code size (except sometimes due to secondary effects like alignment), or '`stc`', the "software trace cache" algorithm, which tries to put all often executed code together, minimizing the number of branches executed by making extra copies of code.

The default is '`simple`' at levels '`-O`', '`-Os`', and '`stc`' at levels '`-O2`', '`-O3`'.

`-freorder-blocks-and-partition`

In addition to reordering basic blocks in the compiled function, in order to reduce number of taken branches, partitions hot and cold basic blocks into separate sections of the assembly and '`.o`' files, to improve paging and cache locality performance.

This optimization is automatically turned off in the presence of exception handling or unwind tables (on targets using setjump/longjump or target specific scheme), for linkonce sections, for functions with a user-defined section attribute and on any architecture that does not support named sections. When '`-fsplit-stack`' is used this option is not enabled by default (to avoid linker errors), but may be enabled explicitly (if using a working linker).

Enabled for x86 at levels '`-O2`', '`-O3`', '`-Os`'.

`-freorder-functions`

Reorder functions in the object file in order to improve code locality. This is implemented by using special subsections `.text.hot` for most frequently executed functions and `.text.unlikely` for unlikely executed functions. Reordering is done by the linker so object file format must support named sections and linker must place them in a reasonable way.

This option isn't effective unless you either provide profile feedback (see '`-fprofile-arcs`' for details) or manually annotate functions with `hot` or `cold` attributes (see Section 6.33.1 [Common Function Attributes], page 478).

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-fstrict-aliasing`

Allow the compiler to assume the strictest aliasing rules applicable to the language being compiled. For C (and C++), this activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same address as an object of a different type, unless the types are almost the same. For example, an `unsigned int` can alias an `int`, but not a `void*` or a `double`. A character type may alias any other type.

Pay special attention to code like this:

```
union a_union {
  int i;
  double d;
};

int f() {
  union a_union t;
  t.d = 3.0;
  return t.i;
}
```

The practice of reading from a different union member than the one most recently written to (called "type-punning") is common. Even with '`-fstrict-aliasing`', type-punning is allowed, provided the memory is accessed through the union type. So, the code above works as expected. See Section 4.9 [Structures unions enumerations and bit-fields implementation], page 445. However, this code might not:

```
int f() {
  union a_union t;
  int* ip;
  t.d = 3.0;
  ip = &t.i;
  return *ip;
}
```

Similarly, access by taking the address, casting the resulting pointer and dereferencing the result has undefined behavior, even if the cast uses a union type, e.g.:

```
int f() {
  double d = 3.0;
  return ((union a_union *) &d)->i;
}
```

The '`-fstrict-aliasing`' option is enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-falign-functions`
`-falign-functions=n`
`-falign-functions=n:m`
`-falign-functions=n:m:n2`
`-falign-functions=n:m:n2:m2`

Align the start of functions to the next power-of-two greater than *n*, skipping up to *m*-1 bytes. This ensures that at least the first *m* bytes of the function can be fetched by the CPU without crossing an *n*-byte alignment boundary.

If *m* is not specified, it defaults to *n*.

Examples: '`-falign-functions=32`' aligns functions to the next 32-byte boundary, '`-falign-functions=24`' aligns to the next 32-byte boundary only if this can be done by skipping 23 bytes or less, '`-falign-functions=32:7`' aligns to the next 32-byte boundary only if this can be done by skipping 6 bytes or less.

The second pair of *n2*:*m2* values allows you to specify a secondary alignment: '`-falign-functions=64:7:32:3`' aligns to the next 64-byte boundary if this can be done by skipping 6 bytes or less, otherwise aligns to the next 32-byte boundary if this can be done by skipping 2 bytes or less. If *m2* is not specified, it defaults to *n2*.

Some assemblers only support this flag when *n* is a power of two; in that case, it is rounded up.

'`-fno-align-functions`' and '`-falign-functions=1`' are equivalent and mean that functions are not aligned.

If *n* is not specified or is zero, use a machine-dependent default. The maximum allowed *n* option value is 65536.

Enabled at levels '`-O2`', '`-O3`'.

`-flimit-function-alignment`

If this option is enabled, the compiler tries to avoid unnecessarily overaligning functions. It attempts to instruct the assembler to align by the amount specified by '`-falign-functions`', but not to skip more bytes than the size of the function.

`-falign-labels`
`-falign-labels=n`
`-falign-labels=n:m`
`-falign-labels=n:m:n2`
`-falign-labels=n:m:n2:m2`

Align all branch targets to a power-of-two boundary.

Parameters of this option are analogous to the '`-falign-functions`' option. '`-fno-align-labels`' and '`-falign-labels=1`' are equivalent and mean that labels are not aligned.

If '`-falign-loops`' or '`-falign-jumps`' are applicable and are greater than this value, then their values are used instead.

If *n* is not specified or is zero, use a machine-dependent default which is very likely to be '1', meaning no alignment. The maximum allowed *n* option value is 65536.

Enabled at levels '-O2', '-O3'.

-falign-loops
-falign-loops=*n*
-falign-loops=*n*:*m*
-falign-loops=*n*:*m*:*n2*
-falign-loops=*n*:*m*:*n2*:*m2*

Align loops to a power-of-two boundary. If the loops are executed many times, this makes up for any execution of the dummy padding instructions.

Parameters of this option are analogous to the '-falign-functions' option. '-fno-align-loops' and '-falign-loops=1' are equivalent and mean that loops are not aligned. The maximum allowed *n* option value is 65536.

If *n* is not specified or is zero, use a machine-dependent default.

Enabled at levels '-O2', '-O3'.

-falign-jumps
-falign-jumps=*n*
-falign-jumps=*n*:*m*
-falign-jumps=*n*:*m*:*n2*
-falign-jumps=*n*:*m*:*n2*:*m2*

Align branch targets to a power-of-two boundary, for branch targets where the targets can only be reached by jumping. In this case, no dummy operations need be executed.

Parameters of this option are analogous to the '-falign-functions' option. '-fno-align-jumps' and '-falign-jumps=1' are equivalent and mean that loops are not aligned.

If *n* is not specified or is zero, use a machine-dependent default. The maximum allowed *n* option value is 65536.

Enabled at levels '-O2', '-O3'.

-funit-at-a-time

This option is left for compatibility reasons. '-funit-at-a-time' has no effect, while '-fno-unit-at-a-time' implies '-fno-toplevel-reorder' and '-fno-section-anchors'.

Enabled by default.

-fno-toplevel-reorder

Do not reorder top-level functions, variables, and asm statements. Output them in the same order that they appear in the input file. When this option is used, unreferenced static variables are not removed. This option is intended to support existing code that relies on a particular ordering. For new code, it is better to use attributes when possible.

'-ftoplevel-reorder' is the default at '-O1' and higher, and also at '-O0' if '-fsection-anchors' is explicitly requested. Additionally '-fno-toplevel-reorder' implies '-fno-section-anchors'.

`-fweb` Constructs webs as commonly used for register allocation purposes and assign each web individual pseudo register. This allows the register allocation pass to operate on pseudos directly, but also strengthens several other optimization passes, such as CSE, loop optimizer and trivial dead code remover. It can, however, make debugging impossible, since variables no longer stay in a "home register".

Enabled by default with '`-funroll-loops`'.

`-fwhole-program`

Assume that the current compilation unit represents the whole program being compiled. All public functions and variables with the exception of `main` and those merged by attribute `externally_visible` become static functions and in effect are optimized more aggressively by interprocedural optimizers.

This option should not be used in combination with '`-flto`'. Instead relying on a linker plugin should provide safer and more precise information.

`-flto[=n]`

This option runs the standard link-time optimizer. When invoked with source code, it generates GIMPLE (one of GCC's internal representations) and writes it to special ELF sections in the object file. When the object files are linked together, all the function bodies are read from these ELF sections and instantiated as if they had been part of the same translation unit.

To use the link-time optimizer, '`-flto`' and optimization options should be specified at compile time and during the final link. It is recommended that you compile all the files participating in the same link with the same options and also specify those options at link time. For example:

```
gcc -c -O2 -flto foo.c
gcc -c -O2 -flto bar.c
gcc -o myprog -flto -O2 foo.o bar.o
```

The first two invocations to GCC save a bytecode representation of GIMPLE into special ELF sections inside '`foo.o`' and '`bar.o`'. The final invocation reads the GIMPLE bytecode from '`foo.o`' and '`bar.o`', merges the two files into a single internal image, and compiles the result as usual. Since both '`foo.o`' and '`bar.o`' are merged into a single image, this causes all the interprocedural analyses and optimizations in GCC to work across the two files as if they were a single one. This means, for example, that the inliner is able to inline functions in '`bar.o`' into functions in '`foo.o`' and vice-versa.

Another (simpler) way to enable link-time optimization is:

```
gcc -o myprog -flto -O2 foo.c bar.c
```

The above generates bytecode for '`foo.c`' and '`bar.c`', merges them together into a single GIMPLE representation and optimizes them as usual to produce '`myprog`'.

The important thing to keep in mind is that to enable link-time optimizations you need to use the GCC driver to perform the link step. GCC automatically performs link-time optimization if any of the objects involved were compiled with the '`-flto`' command-line option. You can always override the automatic

decision to do link-time optimization by passing '-fno-lto' to the link command.

To make whole program optimization effective, it is necessary to make certain whole program assumptions. The compiler needs to know what functions and variables can be accessed by libraries and runtime outside of the link-time optimized unit. When supported by the linker, the linker plugin (see '-fuse-linker-plugin') passes information to the compiler about used and externally visible symbols. When the linker plugin is not available, '-fwhole-program' should be used to allow the compiler to make these assumptions, which leads to more aggressive optimization decisions.

When a file is compiled with '-flto' without '-fuse-linker-plugin', the generated object file is larger than a regular object file because it contains GIMPLE bytecodes and the usual final code (see '-ffat-lto-objects'. This means that object files with LTO information can be linked as normal object files; if '-fno-lto' is passed to the linker, no interprocedural optimizations are applied. Note that when '-fno-fat-lto-objects' is enabled the compile stage is faster but you cannot perform a regular, non-LTO link on them.

When producing the final binary, GCC only applies link-time optimizations to those files that contain bytecode. Therefore, you can mix and match object files and libraries with GIMPLE bytecodes and final object code. GCC automatically selects which files to optimize in LTO mode and which files to link without further processing.

Generally, options specified at link time override those specified at compile time, although in some cases GCC attempts to infer link-time options from the settings used to compile the input files.

If you do not specify an optimization level option '-O' at link time, then GCC uses the highest optimization level used when compiling the object files. Note that it is generally ineffective to specify an optimization level option only at link time and not at compile time, for two reasons. First, compiling without optimization suppresses compiler passes that gather information needed for effective optimization at link time. Second, some early optimization passes can be performed only at compile time and not at link time.

There are some code generation flags preserved by GCC when generating bytecodes, as they need to be used during the final link. Currently, the following options and their settings are taken from the first object file that explicitly specifies them: '-fPIC', '-fpic', '-fpie', '-fcommon', '-fexceptions', '-fnon-call-exceptions', '-fgnu-tm' and all the '-m' target flags.

Certain ABI-changing flags are required to match in all compilation units, and trying to override this at link time with a conflicting value is ignored. This includes options such as '-freg-struct-return' and '-fpcc-struct-return'.

Other options such as '-ffp-contract', '-fno-strict-overflow', '-fwrapv', '-fno-trapv' or '-fno-strict-aliasing' are passed through to the link stage and merged conservatively for conflicting translation units. Specifically '-fno-strict-overflow', '-fwrapv' and '-fno-trapv' take

precedence; and for example '-ffp-contract=off' takes precedence over '-ffp-contract=fast'. You can override them at link time.

If LTO encounters objects with C linkage declared with incompatible types in separate translation units to be linked together (undefined behavior according to ISO C99 6.2.7), a non-fatal diagnostic may be issued. The behavior is still undefined at run time. Similar diagnostics may be raised for other languages.

Another feature of LTO is that it is possible to apply interprocedural optimizations on files written in different languages:

```
gcc -c -flto foo.c
g++ -c -flto bar.cc
gfortran -c -flto baz.f90
g++ -o myprog -flto -O3 foo.o bar.o baz.o -lgfortran
```

Notice that the final link is done with g++ to get the C++ runtime libraries and '-lgfortran' is added to get the Fortran runtime libraries. In general, when mixing languages in LTO mode, you should use the same link command options as when mixing languages in a regular (non-LTO) compilation.

If object files containing GIMPLE bytecode are stored in a library archive, say 'libfoo.a', it is possible to extract and use them in an LTO link if you are using a linker with plugin support. To create static libraries suitable for LTO, use gcc-ar and gcc-ranlib instead of ar and ranlib; to show the symbols of object files with GIMPLE bytecode, use gcc-nm. Those commands require that ar, ranlib and nm have been compiled with plugin support. At link time, use the flag '-fuse-linker-plugin' to ensure that the library participates in the LTO optimization process:

```
gcc -o myprog -O2 -flto -fuse-linker-plugin a.o b.o -lfoo
```

With the linker plugin enabled, the linker extracts the needed GIMPLE files from 'libfoo.a' and passes them on to the running GCC to make them part of the aggregated GIMPLE image to be optimized.

If you are not using a linker with plugin support and/or do not enable the linker plugin, then the objects inside 'libfoo.a' are extracted and linked as usual, but they do not participate in the LTO optimization process. In order to make a static library suitable for both LTO optimization and usual linkage, compile its object files with '-flto' '-ffat-lto-objects'.

Link-time optimizations do not require the presence of the whole program to operate. If the program does not require any symbols to be exported, it is possible to combine '-flto' and '-fwhole-program' to allow the interprocedural optimizers to use more aggressive assumptions which may lead to improved optimization opportunities. Use of '-fwhole-program' is not needed when linker plugin is active (see '-fuse-linker-plugin').

The current implementation of LTO makes no attempt to generate bytecode that is portable between different types of hosts. The bytecode files are versioned and there is a strict version check, so bytecode files generated in one version of GCC do not work with an older or newer version of GCC.

Link-time optimization does not work well with generation of debugging information on systems other than those using a combination of ELF and DWARF.

If you specify the optional *n*, the optimization and code generation done at link time is executed in parallel using *n* parallel jobs by utilizing an installed `make` program. The environment variable `MAKE` may be used to override the program used. The default value for *n* is 1.

You can also specify '`-flto=jobserver`' to use GNU make's job server mode to determine the number of parallel jobs. This is useful when the Makefile calling GCC is already executing in parallel. You must prepend a '`+`' to the command recipe in the parent Makefile for this to work. This option likely only works if `MAKE` is GNU make.

`-flto-partition=alg`

Specify the partitioning algorithm used by the link-time optimizer. The value is either '`1to1`' to specify a partitioning mirroring the original source files or '`balanced`' to specify partitioning into equally sized chunks (whenever possible) or '`max`' to create new partition for every symbol where possible. Specifying '`none`' as an algorithm disables partitioning and streaming completely. The default value is '`balanced`'. While '`1to1`' can be used as an workaround for various code ordering issues, the '`max`' partitioning is intended for internal testing only. The value '`one`' specifies that exactly one partition should be used while the value '`none`' bypasses partitioning and executes the link-time optimization step directly from the WPA phase.

`-flto-odr-type-merging`

Enable streaming of mangled types names of C++ types and their unification at link time. This increases size of LTO object files, but enables diagnostics about One Definition Rule violations.

`-flto-compression-level=n`

This option specifies the level of compression used for intermediate language written to LTO object files, and is only meaningful in conjunction with LTO mode ('`-flto`'). Valid values are 0 (no compression) to 9 (maximum compression). Values outside this range are clamped to either 0 or 9. If the option is not given, a default balanced compression setting is used.

`-fuse-linker-plugin`

Enables the use of a linker plugin during link-time optimization. This option relies on plugin support in the linker, which is available in gold or in GNU ld 2.21 or newer.

This option enables the extraction of object files with GIMPLE bytecode out of library archives. This improves the quality of optimization by exposing more code to the link-time optimizer. This information specifies what symbols can be accessed externally (by non-LTO object or during dynamic linking). Resulting code quality improvements on binaries (and shared libraries that use hidden visibility) are similar to '`-fwhole-program`'. See '`-flto`' for a description of the effect of this flag and how to use it.

This option is enabled by default when LTO support in GCC is enabled and GCC was configured for use with a linker supporting plugins (GNU ld 2.21 or newer or gold).

`-ffat-lto-objects`

Fat LTO objects are object files that contain both the intermediate language and the object code. This makes them usable for both LTO linking and normal linking. This option is effective only when compiling with '`-flto`' and is ignored at link time.

'`-fno-fat-lto-objects`' improves compilation time over plain LTO, but requires the complete toolchain to be aware of LTO. It requires a linker with linker plugin support for basic functionality. Additionally, `nm`, `ar` and `ranlib` need to support linker plugins to allow a full-featured build environment (capable of building static libraries etc). GCC provides the `gcc-ar`, `gcc-nm`, `gcc-ranlib` wrappers to pass the right options to these tools. With non fat LTO makefiles need to be modified to use them.

Note that modern binutils provide plugin auto-load mechanism. Installing the linker plugin into '`$libdir/bfd-plugins`' has the same effect as usage of the command wrappers (`gcc-ar`, `gcc-nm` and `gcc-ranlib`).

The default is '`-fno-fat-lto-objects`' on targets with linker plugin support.

`-fcompare-elim`

After register allocation and post-register allocation instruction splitting, identify arithmetic instructions that compute processor flags similar to a comparison operation based on that arithmetic. If possible, eliminate the explicit comparison operation.

This pass only applies to certain targets that cannot explicitly represent the comparison operation before register allocation is complete.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-fcprop-registers`

After register allocation and post-register allocation instruction splitting, perform a copy-propagation pass to try to reduce scheduling dependencies and occasionally eliminate the copy.

Enabled at levels '`-O`', '`-O2`', '`-O3`', '`-Os`'.

`-fprofile-correction`

Profiles collected using an instrumented binary for multi-threaded programs may be inconsistent due to missed counter updates. When this option is specified, GCC uses heuristics to correct or smooth out such inconsistencies. By default, GCC emits an error message when an inconsistent profile is detected.

This option is enabled by '`-fauto-profile`'.

`-fprofile-use`
`-fprofile-use=`*`path`*

Enable profile feedback-directed optimizations, and the following optimizations, many of which are generally profitable only with profile feedback available:

```
-fbranch-probabilities -fprofile-values
-funroll-loops -fpeel-loops -ftracer -fvpt
-finline-functions -fipa-cp -fipa-cp-clone -fipa-bit-cp
-fpredictive-commoning -fsplit-loops -funswitch-loops
-fgcse-after-reload -ftree-loop-vectorize -ftree-slp-vectorize
```

```
-fvect-cost-model=dynamic -ftree-loop-distribute-patterns
-fprofile-reorder-functions
```

Before you can use this option, you must first generate profiling information. See Section 3.11 [Instrumentation Options], page 189, for information about the '`-fprofile-generate`' option.

By default, GCC emits an error message if the feedback profiles do not match the source code. This error can be turned into a warning by using '`-Wno-error=coverage-mismatch`'. Note this may result in poorly optimized code. Additionally, by default, GCC also emits a warning message if the feedback profiles do not exist (see '`-Wmissing-profile`').

If *path* is specified, GCC looks at the *path* to find the profile feedback data files. See '`-fprofile-dir`'.

`-fauto-profile`
`-fauto-profile=`*path*

Enable sampling-based feedback-directed optimizations, and the following optimizations, many of which are generally profitable only with profile feedback available:

```
-fbranch-probabilities -fprofile-values
-funroll-loops -fpeel-loops -ftracer -fvpt
-finline-functions -fipa-cp -fipa-cp-clone -fipa-bit-cp
-fpredictive-commoning -fsplit-loops -funswitch-loops
-fgcse-after-reload -ftree-loop-vectorize -ftree-slp-vectorize
-fvect-cost-model=dynamic -ftree-loop-distribute-patterns
-fprofile-correction
```

path is the name of a file containing AutoFDO profile information. If omitted, it defaults to '`fbdata.afdo`' in the current directory.

Producing an AutoFDO profile data file requires running your program with the `perf` utility on a supported GNU/Linux target system. For more information, see `https://perf.wiki.kernel.org/`.

E.g.

```
perf record -e br_inst_retired:near_taken -b -o perf.data \
    -- your_program
```

Then use the `create_gcov` tool to convert the raw profile data to a format that can be used by GCC. You must also supply the unstripped binary for your program to this tool. See `https://github.com/google/autofdo`.

E.g.

```
create_gcov --binary=your_program.unstripped --profile=perf.data \
    --gcov=profile.afdo
```

The following options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness. All must be specifically enabled.

`-ffloat-store`

Do not store floating-point variables in registers, and inhibit other options that might change whether a floating-point value is taken from a register or memory.

This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double`

is supposed to have. Similarly for the x86 architecture. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '-ffloat-store' for such programs, after modifying them to store all pertinent intermediate computations into variables.

-fexcess-precision=*style*

This option allows further control over excess precision on machines where floating-point operations occur in a format with more precision or range than the IEEE standard and interchange floating-point types. By default, '-fexcess-precision=fast' is in effect; this means that operations may be carried out in a wider precision than the types specified in the source if that would result in faster code, and it is unpredictable when rounding to the types specified in the source code takes place. When compiling C, if '-fexcess-precision=standard' is specified then excess precision follows the rules specified in ISO C99; in particular, both casts and assignments cause values to be rounded to their semantic types (whereas '-ffloat-store' only affects assignments). This option is enabled by default for C if a strict conformance option such as '-std=c99' is used. '-ffast-math' enables '-fexcess-precision=fast' by default regardless of whether a strict conformance option is used.

'-fexcess-precision=standard' is not implemented for languages other than C. On the x86, it has no effect if '-mfpmath=sse' or '-mfpmath=sse+387' is specified; in the former case, IEEE semantics apply without excess precision, and in the latter, rounding is unpredictable.

-ffast-math

Sets the options '-fno-math-errno', '-funsafe-math-optimizations', '-ffinite-math-only', '-fno-rounding-math', '-fno-signaling-nans', '-fcx-limited-range' and '-fexcess-precision=fast'.

This option causes the preprocessor macro __FAST_MATH__ to be defined.

This option is not turned on by any '-O' option besides '-Ofast' since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

-fno-math-errno

Do not set errno after calling math functions that are executed with a single instruction, e.g., sqrt. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.

This option is not turned on by any '-O' option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

The default is '-fmath-errno'.

On Darwin systems, the math library never sets `errno`. There is therefore no reason for the compiler to consider the possibility that it might, and '`-fno-math-errno`' is the default.

`-funsafe-math-optimizations`

Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid and (b) may violate IEEE or ANSI standards. When used at link time, it may include libraries or startup files that change the default FPU control word or other similar optimizations.

This option is not turned on by any '`-O`' option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications. Enables '`-fno-signed-zeros`', '`-fno-trapping-math`', '`-fassociative-math`' and '`-freciprocal-math`'.

The default is '`-fno-unsafe-math-optimizations`'.

`-fassociative-math`

Allow re-association of operands in series of floating-point operations. This violates the ISO C and C++ language standard by possibly changing computation result. NOTE: re-ordering may change the sign of zero as well as ignore NaNs and inhibit or create underflow or overflow (and thus cannot be used on code that relies on rounding behavior like `(x + 2**52) - 2**52`. May also reorder floating-point comparisons and thus may not be used when ordered comparisons are required. This option requires that both '`-fno-signed-zeros`' and '`-fno-trapping-math`' be in effect. Moreover, it doesn't make much sense with '`-frounding-math`'. For Fortran the option is automatically enabled when both '`-fno-signed-zeros`' and '`-fno-trapping-math`' are in effect.

The default is '`-fno-associative-math`'.

`-freciprocal-math`

Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations. For example `x / y` can be replaced with `x * (1/y)`, which is useful if `(1/y)` is subject to common subexpression elimination. Note that this loses precision and increases the number of flops operating on the value.

The default is '`-fno-reciprocal-math`'.

`-ffinite-math-only`

Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs or `+`-Infs.

This option is not turned on by any '`-O`' option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions. It may, however, yield faster code for programs that do not require the guarantees of these specifications.

The default is '`-fno-finite-math-only`'.

`-fno-signed-zeros`

Allow optimizations for floating-point arithmetic that ignore the signedness of zero. IEEE arithmetic specifies the behavior of distinct +0.0 and −0.0 values, which then prohibits simplification of expressions such as x+0.0 or 0.0*x (even with '`-ffinite-math-only`'). This option implies that the sign of a zero result isn't significant.

The default is '`-fsigned-zeros`'.

`-fno-trapping-math`

Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation. This option requires that '`-fno-signaling-nans`' be in effect. Setting this option may allow faster code if one relies on "non-stop" IEEE arithmetic, for example.

This option should never be turned on by any '`-O`' option since it can result in incorrect output for programs that depend on an exact implementation of IEEE or ISO rules/specifications for math functions.

The default is '`-ftrapping-math`'.

`-frounding-math`

Disable transformations and optimizations that assume default floating-point rounding behavior. This is round-to-zero for all floating point to integer conversions, and round-to-nearest for all other arithmetic truncations. This option should be specified for programs that change the FP rounding mode dynamically, or that may be executed with a non-default rounding mode. This option disables constant folding of floating-point expressions at compile time (which may be affected by rounding mode) and arithmetic transformations that are unsafe in the presence of sign-dependent rounding modes.

The default is '`-fno-rounding-math`'.

This option is experimental and does not currently guarantee to disable all GCC optimizations that are affected by rounding mode. Future versions of GCC may provide finer control of this setting using C99's `FENV_ACCESS` pragma. This command-line option will be used to specify the default state for `FENV_ACCESS`.

`-fsignaling-nans`

Compile code assuming that IEEE signaling NaNs may generate user-visible traps during floating-point operations. Setting this option disables optimizations that may change the number of exceptions visible with signaling NaNs. This option implies '`-ftrapping-math`'.

This option causes the preprocessor macro `__SUPPORT_SNAN__` to be defined.

The default is '`-fno-signaling-nans`'.

This option is experimental and does not currently guarantee to disable all GCC optimizations that affect signaling NaN behavior.

`-fno-fp-int-builtin-inexact`

Do not allow the built-in functions `ceil`, `floor`, `round` and `trunc`, and their `float` and `long double` variants, to generate code that raises the "inexact"

floating-point exception for noninteger arguments. ISO C99 and C11 allow these functions to raise the "inexact" exception, but ISO/IEC TS 18661-1:2014, the C bindings to IEEE 754-2008, does not allow these functions to do so.

The default is '`-ffp-int-builtin-inexact`', allowing the exception to be raised. This option does nothing unless '`-ftrapping-math`' is in effect.

Even if '`-fno-fp-int-builtin-inexact`' is used, if the functions generate a call to a library function then the "inexact" exception may be raised if the library implementation does not follow TS 18661.

`-fsingle-precision-constant`
: Treat floating-point constants as single precision instead of implicitly converting them to double-precision constants.

`-fcx-limited-range`
: When enabled, this option states that a range reduction step is not needed when performing complex division. Also, there is no checking whether the result of a complex multiplication or division is `NaN + I*NaN`, with an attempt to rescue the situation in that case. The default is '`-fno-cx-limited-range`', but is enabled by '`-ffast-math`'.

This option controls the default setting of the ISO C99 `CX_LIMITED_RANGE` pragma. Nevertheless, the option applies to all languages.

`-fcx-fortran-rules`
: Complex multiplication and division follow Fortran rules. Range reduction is done as part of complex division, but there is no checking whether the result of a complex multiplication or division is `NaN + I*NaN`, with an attempt to rescue the situation in that case.

The default is '`-fno-cx-fortran-rules`'.

The following options control optimizations that may improve performance, but are not enabled by any '`-O`' options. This section includes experimental options that may produce broken code.

`-fbranch-probabilities`
: After running a program compiled with '`-fprofile-arcs`' (see Section 3.11 [Instrumentation Options], page 189), you can compile it a second time using '`-fbranch-probabilities`', to improve optimizations based on the number of times each branch was taken. When a program compiled with '`-fprofile-arcs`' exits, it saves arc execution counts to a file called '*sourcename*`.gcda`' for each source file. The information in this data file is very dependent on the structure of the generated code, so you must use the same source code and the same optimization options for both compilations.

With '`-fbranch-probabilities`', GCC puts a '`REG_BR_PROB`' note on each '`JUMP_INSN`' and '`CALL_INSN`'. These can be used to improve optimization. Currently, they are only used in one place: in '`reorg.c`', instead of guessing which path a branch is most likely to take, the '`REG_BR_PROB`' values are used to exactly determine which path is taken more often.

Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-fprofile-values`

If combined with '`-fprofile-arcs`', it adds code so that some data about values of expressions in the program is gathered.

With '`-fbranch-probabilities`', it reads back the data gathered from profiling values of expressions for usage in optimizations.

Enabled by '`-fprofile-generate`', '`-fprofile-use`', and '`-fauto-profile`'.

`-fprofile-reorder-functions`

Function reordering based on profile instrumentation collects first time of execution of a function and orders these functions in ascending order.

Enabled with '`-fprofile-use`'.

`-fvpt` If combined with '`-fprofile-arcs`', this option instructs the compiler to add code to gather information about values of expressions.

With '`-fbranch-probabilities`', it reads back the data gathered and actually performs the optimizations based on them. Currently the optimizations include specialization of division operations using the knowledge about the value of the denominator.

Enabled with '`-fprofile-use`' and '`-fauto-profile`'.

`-frename-registers`

Attempt to avoid false dependencies in scheduled code by making use of registers left over after register allocation. This optimization most benefits processors with lots of registers. Depending on the debug information format adopted by the target, however, it can make debugging impossible, since variables no longer stay in a "home register".

Enabled by default with '`-funroll-loops`'.

`-fschedule-fusion`

Performs a target dependent pass over the instruction stream to schedule instructions of same type together because target machine can execute them more efficiently if they are adjacent to each other in the instruction flow.

Enabled at levels '`-O2`', '`-O3`', '`-Os`'.

`-ftracer` Perform tail duplication to enlarge superblock size. This transformation simplifies the control flow of the function allowing other optimizations to do a better job.

Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-funroll-loops`

Unroll loops whose number of iterations can be determined at compile time or upon entry to the loop. '`-funroll-loops`' implies '`-frerun-cse-after-loop`', '`-fweb`' and '`-frename-registers`'. It also turns on complete loop peeling (i.e. complete removal of loops with a small constant number of iterations). This option makes code larger, and may or may not make it run faster.

Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-funroll-all-loops`

Unroll all loops, even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly. '`-funroll-all-loops`' implies the same options as '`-funroll-loops`'.

`-fpeel-loops`

Peels loops for which there is enough information that they do not roll much (from profile feedback or static analysis). It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations).

Enabled by '`-O3`', '`-fprofile-use`', and '`-fauto-profile`'.

`-fmove-loop-invariants`

Enables the loop invariant motion pass in the RTL loop optimizer. Enabled at level '`-O1`' and higher, except for '`-Og`'.

`-fsplit-loops`

Split a loop into two if it contains a condition that's always true for one side of the iteration space and false for the other.

Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-funswitch-loops`

Move branches with loop invariant conditions out of the loop, with duplicates of the loop on both branches (modified according to result of the condition).

Enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-fversion-loops-for-strides`

If a loop iterates over an array with a variable stride, create another version of the loop that assumes the stride is always one. For example:

```
for (int i = 0; i < n; ++i)
  x[i * stride] = ...;
```

becomes:

```
if (stride == 1)
  for (int i = 0; i < n; ++i)
    x[i] = ...;
else
  for (int i = 0; i < n; ++i)
    x[i * stride] = ...;
```

This is particularly useful for assumed-shape arrays in Fortran where (for example) it allows better vectorization assuming contiguous accesses. This flag is enabled by default at '`-O3`'. It is also enabled by '`-fprofile-use`' and '`-fauto-profile`'.

`-ffunction-sections`
`-fdata-sections`

Place each function or data item into its own section in the output file if the target supports arbitrary sections. The name of the function or the name of the data item determines the section's name in the output file.

Use these options on systems where the linker can perform optimizations to improve locality of reference in the instruction space. Most systems using the ELF object format have linkers with such optimizations. On AIX, the linker

rearranges sections (CSECTs) based on the call graph. The performance impact varies.

Together with a linker garbage collection (linker '`--gc-sections`' option) these options may lead to smaller statically-linked executables (after stripping).

On ELF/DWARF systems these options do not degenerate the quality of the debug information. There could be issues with other object files/debug info formats.

Only use these options when there are significant benefits from doing so. When you specify these options, the assembler and linker create larger object and executable files and are also slower. These options affect code generation. They prevent optimizations by the compiler and assembler using relative locations inside a translation unit since the locations are unknown until link time. An example of such an optimization is relaxing calls to short call instructions.

`-fbranch-target-load-optimize`
: Perform branch target register load optimization before prologue / epilogue threading. The use of target registers can typically be exposed only during reload, thus hoisting loads out of loops and doing inter-block scheduling needs a separate optimization pass.

`-fbranch-target-load-optimize2`
: Perform branch target register load optimization after prologue / epilogue threading.

`-fbtr-bb-exclusive`
: When performing branch target register load optimization, don't reuse branch target registers within any basic block.

`-fstdarg-opt`
: Optimize the prologue of variadic argument functions with respect to usage of those arguments.

`-fsection-anchors`
: Try to reduce the number of symbolic address calculations by using shared "anchor" symbols to address nearby objects. This transformation can help to reduce the number of GOT entries and GOT accesses on some targets.

For example, the implementation of the following function `foo`:

```
static int a, b, c;
int foo (void) { return a + b + c; }
```

usually calculates the addresses of all three variables, but if you compile it with '`-fsection-anchors`', it accesses the variables from a common anchor point instead. The effect is similar to the following pseudocode (which isn't valid C):

```
int foo (void)
{
  register int *xr = &x;
  return xr[&a - &x] + xr[&b - &x] + xr[&c - &x];
}
```

Not all targets support this option.

`--param` *name=value*

In some places, GCC uses various constants to control the amount of optimization that is done. For example, GCC does not inline functions that contain more than a certain number of instructions. You can control some of these constants on the command line using the '`--param`' option.

The names of specific parameters, and the meaning of the values, are tied to the internals of the compiler, and are subject to change without notice in future releases.

In order to get minimal, maximal and default value of a parameter, one can use '`--help=param -Q`' options.

In each case, the *value* is an integer. The allowable choices for *name* are:

`predictable-branch-outcome`

When branch is predicted to be taken with probability lower than this threshold (in percent), then it is considered well predictable.

`max-rtl-if-conversion-insns`

RTL if-conversion tries to remove conditional branches around a block and replace them with conditionally executed instructions. This parameter gives the maximum number of instructions in a block which should be considered for if-conversion. The compiler will also use other heuristics to decide whether if-conversion is likely to be profitable.

`max-rtl-if-conversion-predictable-cost`
`max-rtl-if-conversion-unpredictable-cost`

RTL if-conversion will try to remove conditional branches around a block and replace them with conditionally executed instructions. These parameters give the maximum permissible cost for the sequence that would be generated by if-conversion depending on whether the branch is statically determined to be predictable or not. The units for this parameter are the same as those for the GCC internal seq_cost metric. The compiler will try to provide a reasonable default for this parameter using the BRANCH_COST target macro.

`max-crossjump-edges`

The maximum number of incoming edges to consider for cross-jumping. The algorithm used by '`-fcrossjumping`' is $O(N^2)$ in the number of edges incoming to each block. Increasing values mean more aggressive optimization, making the compilation time increase with probably small improvement in executable size.

`min-crossjump-insns`

The minimum number of instructions that must be matched at the end of two blocks before cross-jumping is performed on them. This value is ignored in the case where all instructions in the block being cross-jumped from are matched.

`max-grow-copy-bb-insns`
: The maximum code size expansion factor when copying basic blocks instead of jumping. The expansion is relative to a jump instruction.

`max-goto-duplication-insns`
: The maximum number of instructions to duplicate to a block that jumps to a computed goto. To avoid $O(N^2)$ behavior in a number of passes, GCC factors computed gotos early in the compilation process, and unfactors them as late as possible. Only computed jumps at the end of a basic blocks with no more than max-goto-duplication-insns are unfactored.

`max-delay-slot-insn-search`
: The maximum number of instructions to consider when looking for an instruction to fill a delay slot. If more than this arbitrary number of instructions are searched, the time savings from filling the delay slot are minimal, so stop searching. Increasing values mean more aggressive optimization, making the compilation time increase with probably small improvement in execution time.

`max-delay-slot-live-search`
: When trying to fill delay slots, the maximum number of instructions to consider when searching for a block with valid live register information. Increasing this arbitrarily chosen value means more aggressive optimization, increasing the compilation time. This parameter should be removed when the delay slot code is rewritten to maintain the control-flow graph.

`max-gcse-memory`
: The approximate maximum amount of memory that can be allocated in order to perform the global common subexpression elimination optimization. If more memory than specified is required, the optimization is not done.

`max-gcse-insertion-ratio`
: If the ratio of expression insertions to deletions is larger than this value for any expression, then RTL PRE inserts or removes the expression and thus leaves partially redundant computations in the instruction stream.

`max-pending-list-length`
: The maximum number of pending dependencies scheduling allows before flushing the current state and starting over. Large functions with few branches or calls can create excessively large lists which needlessly consume memory and resources.

`max-modulo-backtrack-attempts`
: The maximum number of backtrack attempts the scheduler should make when modulo scheduling a loop. Larger values can exponentially increase compilation time.

`max-inline-insns-single`
: Several parameters control the tree inliner used in GCC. This number sets the maximum number of instructions (counted in GCC's internal representation) in a single function that the tree inliner considers for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++).

`max-inline-insns-auto`
: When you use '`-finline-functions`' (included in '`-O3`'), a lot of functions that would otherwise not be considered for inlining by the compiler are investigated. To those functions, a different (more restrictive) limit compared to functions declared inline can be applied.

`max-inline-insns-small`
: This is bound applied to calls which are considered relevant with '`-finline-small-functions`'.

`max-inline-insns-size`
: This is bound applied to calls which are optimized for size. Small growth may be desirable to anticipate optimization oppurtunities exposed by inlining.

`uninlined-function-insns`
: Number of instructions accounted by inliner for function overhead such as function prologue and epilogue.

`uninlined-function-time`
: Extra time accounted by inliner for function overhead such as time needed to execute function prologue and epilogue

`uninlined-thunk-insns`
`uninlined-thunk-time`
: Same as '`--param uninlined-function-insns`' and '`--param uninlined-function-time`' but applied to function thunks

`inline-min-speedup`
: When estimated performance improvement of caller + callee runtime exceeds this threshold (in percent), the function can be inlined regardless of the limit on '`--param max-inline-insns-single`' and '`--param max-inline-insns-auto`'.

`large-function-insns`
: The limit specifying really large functions. For functions larger than this limit after inlining, inlining is constrained by '`--param large-function-growth`'. This parameter is useful primarily to avoid extreme compilation time caused by non-linear algorithms used by the back end.

`large-function-growth`
: Specifies maximal growth of large function caused by inlining in percents. For example, parameter value 100 limits large function growth to 2.0 times the original size.

`large-unit-insns`
: The limit specifying large translation unit. Growth caused by inlining of units larger than this limit is limited by '`--param inline-unit-growth`'. For small units this might be too tight. For example, consider a unit consisting of function A that is inline and B that just calls A three times. If B is small relative to A, the growth of unit is 300\% and yet such inlining is very sane. For very large units consisting of small inlineable functions, however, the overall unit growth limit is needed to avoid exponential explosion of code size. Thus for smaller units, the size is increased to '`--param large-unit-insns`' before applying '`--param inline-unit-growth`'.

`inline-unit-growth`
: Specifies maximal overall growth of the compilation unit caused by inlining. For example, parameter value 20 limits unit growth to 1.2 times the original size. Cold functions (either marked cold via an attribute or by profile feedback) are not accounted into the unit size.

`ipcp-unit-growth`
: Specifies maximal overall growth of the compilation unit caused by interprocedural constant propagation. For example, parameter value 10 limits unit growth to 1.1 times the original size.

`large-stack-frame`
: The limit specifying large stack frames. While inlining the algorithm is trying to not grow past this limit too much.

`large-stack-frame-growth`
: Specifies maximal growth of large stack frames caused by inlining in percents. For example, parameter value 1000 limits large stack frame growth to 11 times the original size.

`max-inline-insns-recursive`
`max-inline-insns-recursive-auto`
: Specifies the maximum number of instructions an out-of-line copy of a self-recursive inline function can grow into by performing recursive inlining.

  '`--param max-inline-insns-recursive`' applies to functions declared inline. For functions not declared inline, recursive inlining happens only when '`-finline-functions`' (included in '`-O3`') is enabled; '`--param max-inline-insns-recursive-auto`' applies instead.

`max-inline-recursive-depth`
`max-inline-recursive-depth-auto`
: Specifies the maximum recursion depth used for recursive inlining.

  '`--param max-inline-recursive-depth`' applies to functions declared inline. For functions not declared inline, recursive inlin-

ing happens only when '`-finline-functions`' (included in '`-O3`') is enabled; '`--param max-inline-recursive-depth-auto`' applies instead.

`min-inline-recursive-probability`
: Recursive inlining is profitable only for function having deep recursion in average and can hurt for function having little recursion depth by increasing the prologue size or complexity of function body to other optimizers.

  When profile feedback is available (see '`-fprofile-generate`') the actual recursion depth can be guessed from the probability that function recurses via a given call expression. This parameter limits inlining only to call expressions whose probability exceeds the given threshold (in percents).

`early-inlining-insns`
: Specify growth that the early inliner can make. In effect it increases the amount of inlining for code having a large abstraction penalty.

`max-early-inliner-iterations`
: Limit of iterations of the early inliner. This basically bounds the number of nested indirect calls the early inliner can resolve. Deeper chains are still handled by late inlining.

`comdat-sharing-probability`
: Probability (in percent) that C++ inline function with comdat visibility are shared across multiple compilation units.

`profile-func-internal-id`
: A parameter to control whether to use function internal id in profile database lookup. If the value is 0, the compiler uses an id that is based on function assembler name and filename, which makes old profile data more tolerant to source changes such as function reordering etc.

`min-vect-loop-bound`
: The minimum number of iterations under which loops are not vectorized when '`-ftree-vectorize`' is used. The number of iterations after vectorization needs to be greater than the value specified by this option to allow vectorization.

`gcse-cost-distance-ratio`
: Scaling factor in calculation of maximum distance an expression can be moved by GCSE optimizations. This is currently supported only in the code hoisting pass. The bigger the ratio, the more aggressive code hoisting is with simple expressions, i.e., the expressions that have cost less than '`gcse-unrestricted-cost`'. Specifying 0 disables hoisting of simple expressions.

`gcse-unrestricted-cost`
: Cost, roughly measured as the cost of a single typical machine instruction, at which GCSE optimizations do not constrain the dis-

tance an expression can travel. This is currently supported only in the code hoisting pass. The lesser the cost, the more aggressive code hoisting is. Specifying 0 allows all expressions to travel unrestricted distances.

`max-hoist-depth`
: The depth of search in the dominator tree for expressions to hoist. This is used to avoid quadratic behavior in hoisting algorithm. The value of 0 does not limit on the search, but may slow down compilation of huge functions.

`max-tail-merge-comparisons`
: The maximum amount of similar bbs to compare a bb with. This is used to avoid quadratic behavior in tree tail merging.

`max-tail-merge-iterations`
: The maximum amount of iterations of the pass over the function. This is used to limit compilation time in tree tail merging.

`store-merging-allow-unaligned`
: Allow the store merging pass to introduce unaligned stores if it is legal to do so.

`max-stores-to-merge`
: The maximum number of stores to attempt to merge into wider stores in the store merging pass.

`max-unrolled-insns`
: The maximum number of instructions that a loop may have to be unrolled. If a loop is unrolled, this parameter also determines how many times the loop code is unrolled.

`max-average-unrolled-insns`
: The maximum number of instructions biased by probabilities of their execution that a loop may have to be unrolled. If a loop is unrolled, this parameter also determines how many times the loop code is unrolled.

`max-unroll-times`
: The maximum number of unrollings of a single loop.

`max-peeled-insns`
: The maximum number of instructions that a loop may have to be peeled. If a loop is peeled, this parameter also determines how many times the loop code is peeled.

`max-peel-times`
: The maximum number of peelings of a single loop.

`max-peel-branches`
: The maximum number of branches on the hot path through the peeled sequence.

`max-completely-peeled-insns`
: The maximum number of insns of a completely peeled loop.

`max-completely-peel-times`
: The maximum number of iterations of a loop to be suitable for complete peeling.

`max-completely-peel-loop-nest-depth`
: The maximum depth of a loop nest suitable for complete peeling.

`max-unswitch-insns`
: The maximum number of insns of an unswitched loop.

`max-unswitch-level`
: The maximum number of branches unswitched in a single loop.

`lim-expensive`
: The minimum cost of an expensive expression in the loop invariant motion.

`iv-consider-all-candidates-bound`
: Bound on number of candidates for induction variables, below which all candidates are considered for each use in induction variable optimizations. If there are more candidates than this, only the most relevant ones are considered to avoid quadratic time complexity.

`iv-max-considered-uses`
: The induction variable optimizations give up on loops that contain more induction variable uses.

`iv-always-prune-cand-set-bound`
: If the number of candidates in the set is smaller than this value, always try to remove unnecessary ivs from the set when adding a new one.

`avg-loop-niter`
: Average number of iterations of a loop.

`dse-max-object-size`
: Maximum size (in bytes) of objects tracked bytewise by dead store elimination. Larger values may result in larger compilation times.

`dse-max-alias-queries-per-store`
: Maximum number of queries into the alias oracle per store. Larger values result in larger compilation times and may result in more removed dead stores.

`scev-max-expr-size`
: Bound on size of expressions used in the scalar evolutions analyzer. Large expressions slow the analyzer.

`scev-max-expr-complexity`
: Bound on the complexity of the expressions in the scalar evolutions analyzer. Complex expressions slow the analyzer.

`max-tree-if-conversion-phi-args`
: Maximum number of arguments in a PHI supported by TREE if conversion unless the loop is marked with simd pragma.

`vect-max-version-for-alignment-checks`
: The maximum number of run-time checks that can be performed when doing loop versioning for alignment in the vectorizer.

`vect-max-version-for-alias-checks`
: The maximum number of run-time checks that can be performed when doing loop versioning for alias in the vectorizer.

`vect-max-peeling-for-alignment`
: The maximum number of loop peels to enhance access alignment for vectorizer. Value -1 means no limit.

`max-iterations-to-track`
: The maximum number of iterations of a loop the brute-force algorithm for analysis of the number of iterations of the loop tries to evaluate.

`hot-bb-count-ws-permille`
: A basic block profile count is considered hot if it contributes to the given permillage (i.e. 0...1000) of the entire profiled execution.

`hot-bb-frequency-fraction`
: Select fraction of the entry block frequency of executions of basic block in function given basic block needs to have to be considered hot.

`max-predicted-iterations`
: The maximum number of loop iterations we predict statically. This is useful in cases where a function contains a single loop with known bound and another loop with unknown bound. The known number of iterations is predicted correctly, while the unknown number of iterations average to roughly 10. This means that the loop without bounds appears artificially cold relative to the other one.

`builtin-expect-probability`
: Control the probability of the expression having the specified value. This parameter takes a percentage (i.e. 0 ... 100) as input.

`builtin-string-cmp-inline-length`
: The maximum length of a constant string for a builtin string cmp call eligible for inlining.

`align-threshold`
: Select fraction of the maximal frequency of executions of a basic block in a function to align the basic block.

`align-loop-iterations`
: A loop expected to iterate at least the selected number of iterations is aligned.

`tracer-dynamic-coverage`
`tracer-dynamic-coverage-feedback`

This value is used to limit superblock formation once the given percentage of executed instructions is covered. This limits unnecessary code size expansion.

The '`tracer-dynamic-coverage-feedback`' parameter is used only when profile feedback is available. The real profiles (as opposed to statically estimated ones) are much less balanced allowing the threshold to be larger value.

`tracer-max-code-growth`

Stop tail duplication once code growth has reached given percentage. This is a rather artificial limit, as most of the duplicates are eliminated later in cross jumping, so it may be set to much higher values than is the desired code growth.

`tracer-min-branch-ratio`

Stop reverse growth when the reverse probability of best edge is less than this threshold (in percent).

`tracer-min-branch-probability`
`tracer-min-branch-probability-feedback`

Stop forward growth if the best edge has probability lower than this threshold.

Similarly to '`tracer-dynamic-coverage`' two parameters are provided. '`tracer-min-branch-probability-feedback`' is used for compilation with profile feedback and '`tracer-min-branch-probability`' compilation without. The value for compilation with profile feedback needs to be more conservative (higher) in order to make tracer effective.

`stack-clash-protection-guard-size`

Specify the size of the operating system provided stack guard as 2 raised to *num* bytes. Higher values may reduce the number of explicit probes, but a value larger than the operating system provided guard will leave code vulnerable to stack clash style attacks.

`stack-clash-protection-probe-interval`

Stack clash protection involves probing stack space as it is allocated. This param controls the maximum distance between probes into the stack as 2 raised to *num* bytes. Higher values may reduce the number of explicit probes, but a value larger than the operating system provided guard will leave code vulnerable to stack clash style attacks.

`max-cse-path-length`

The maximum number of basic blocks on path that CSE considers.

`max-cse-insns`

The maximum number of instructions CSE processes before flushing.

`ggc-min-expand`

GCC uses a garbage collector to manage its own memory allocation. This parameter specifies the minimum percentage by which the garbage collector's heap should be allowed to expand between collections. Tuning this may improve compilation speed; it has no effect on code generation.

The default is 30% + 70% * (RAM/1GB) with an upper bound of 100% when RAM >= 1GB. If `getrlimit` is available, the notion of "RAM" is the smallest of actual RAM and `RLIMIT_DATA` or `RLIMIT_AS`. If GCC is not able to calculate RAM on a particular platform, the lower bound of 30% is used. Setting this parameter and '`ggc-min-heapsize`' to zero causes a full collection to occur at every opportunity. This is extremely slow, but can be useful for debugging.

`ggc-min-heapsize`

Minimum size of the garbage collector's heap before it begins bothering to collect garbage. The first collection occurs after the heap expands by '`ggc-min-expand`'% beyond '`ggc-min-heapsize`'. Again, tuning this may improve compilation speed, and has no effect on code generation.

The default is the smaller of RAM/8, RLIMIT_RSS, or a limit that tries to ensure that RLIMIT_DATA or RLIMIT_AS are not exceeded, but with a lower bound of 4096 (four megabytes) and an upper bound of 131072 (128 megabytes). If GCC is not able to calculate RAM on a particular platform, the lower bound is used. Setting this parameter very large effectively disables garbage collection. Setting this parameter and '`ggc-min-expand`' to zero causes a full collection to occur at every opportunity.

`max-reload-search-insns`

The maximum number of instruction reload should look backward for equivalent register. Increasing values mean more aggressive optimization, making the compilation time increase with probably slightly better performance.

`max-cselib-memory-locations`

The maximum number of memory locations cselib should take into account. Increasing values mean more aggressive optimization, making the compilation time increase with probably slightly better performance.

`max-sched-ready-insns`

The maximum number of instructions ready to be issued the scheduler should consider at any given time during the first scheduling pass. Increasing values mean more thorough searches, making the compilation time increase with probably little benefit.

`max-sched-region-blocks`
The maximum number of blocks in a region to be considered for interblock scheduling.

`max-pipeline-region-blocks`
The maximum number of blocks in a region to be considered for pipelining in the selective scheduler.

`max-sched-region-insns`
The maximum number of insns in a region to be considered for interblock scheduling.

`max-pipeline-region-insns`
The maximum number of insns in a region to be considered for pipelining in the selective scheduler.

`min-spec-prob`
The minimum probability (in percents) of reaching a source block for interblock speculative scheduling.

`max-sched-extend-regions-iters`
The maximum number of iterations through CFG to extend regions. A value of 0 disables region extensions.

`max-sched-insn-conflict-delay`
The maximum conflict delay for an insn to be considered for speculative motion.

`sched-spec-prob-cutoff`
The minimal probability of speculation success (in percents), so that speculative insns are scheduled.

`sched-state-edge-prob-cutoff`
The minimum probability an edge must have for the scheduler to save its state across it.

`sched-mem-true-dep-cost`
Minimal distance (in CPU cycles) between store and load targeting same memory locations.

`selsched-max-lookahead`
The maximum size of the lookahead window of selective scheduling. It is a depth of search for available instructions.

`selsched-max-sched-times`
The maximum number of times that an instruction is scheduled during selective scheduling. This is the limit on the number of iterations through which the instruction may be pipelined.

`selsched-insns-to-rename`
The maximum number of best instructions in the ready list that are considered for renaming in the selective scheduler.

`sms-min-sc`
: The minimum value of stage count that swing modulo scheduler generates.

`max-last-value-rtl`
: The maximum size measured as number of RTLs that can be recorded in an expression in combiner for a pseudo register as last known value of that register.

`max-combine-insns`
: The maximum number of instructions the RTL combiner tries to combine.

`integer-share-limit`
: Small integer constants can use a shared data structure, reducing the compiler's memory usage and increasing its speed. This sets the maximum value of a shared integer constant.

`ssp-buffer-size`
: The minimum size of buffers (i.e. arrays) that receive stack smashing protection when '`-fstack-protection`' is used.

`min-size-for-stack-sharing`
: The minimum size of variables taking part in stack slot sharing when not optimizing.

`max-jump-thread-duplication-stmts`
: Maximum number of statements allowed in a block that needs to be duplicated when threading jumps.

`max-fields-for-field-sensitive`
: Maximum number of fields in a structure treated in a field sensitive manner during pointer analysis.

`prefetch-latency`
: Estimate on average number of instructions that are executed before prefetch finishes. The distance prefetched ahead is proportional to this constant. Increasing this number may also lead to less streams being prefetched (see '`simultaneous-prefetches`').

`simultaneous-prefetches`
: Maximum number of prefetches that can run at the same time.

`l1-cache-line-size`
: The size of cache line in L1 data cache, in bytes.

`l1-cache-size`
: The size of L1 data cache, in kilobytes.

`l2-cache-size`
: The size of L2 data cache, in kilobytes.

`prefetch-dynamic-strides`
: Whether the loop array prefetch pass should issue software prefetch hints for strides that are non-constant. In some cases this may be

beneficial, though the fact the stride is non-constant may make it hard to predict when there is clear benefit to issuing these hints.

Set to 1 if the prefetch hints should be issued for non-constant strides. Set to 0 if prefetch hints should be issued only for strides that are known to be constant and below '`prefetch-minimum-stride`'.

`prefetch-minimum-stride`
: Minimum constant stride, in bytes, to start using prefetch hints for. If the stride is less than this threshold, prefetch hints will not be issued.

This setting is useful for processors that have hardware prefetchers, in which case there may be conflicts between the hardware prefetchers and the software prefetchers. If the hardware prefetchers have a maximum stride they can handle, it should be used here to improve the use of software prefetchers.

A value of -1 means we don't have a threshold and therefore prefetch hints can be issued for any constant stride.

This setting is only useful for strides that are known and constant.

`loop-interchange-max-num-stmts`
: The maximum number of stmts in a loop to be interchanged.

`loop-interchange-stride-ratio`
: The minimum ratio between stride of two loops for interchange to be profitable.

`min-insn-to-prefetch-ratio`
: The minimum ratio between the number of instructions and the number of prefetches to enable prefetching in a loop.

`prefetch-min-insn-to-mem-ratio`
: The minimum ratio between the number of instructions and the number of memory references to enable prefetching in a loop.

`use-canonical-types`
: Whether the compiler should use the "canonical" type system. Should always be 1, which uses a more efficient internal mechanism for comparing types in C++ and Objective-C++. However, if bugs in the canonical type system are causing compilation failures, set this value to 0 to disable canonical types.

`switch-conversion-max-branch-ratio`
: Switch initialization conversion refuses to create arrays that are bigger than '`switch-conversion-max-branch-ratio`' times the number of branches in the switch.

`max-partial-antic-length`
: Maximum length of the partial antic set computed during the tree partial redundancy elimination optimization ('`-ftree-pre`') when

optimizing at '-O3' and above. For some sorts of source code the enhanced partial redundancy elimination optimization can run away, consuming all of the memory available on the host machine. This parameter sets a limit on the length of the sets that are computed, which prevents the runaway behavior. Setting a value of 0 for this parameter allows an unlimited set length.

`rpo-vn-max-loop-depth`
: Maximum loop depth that is value-numbered optimistically. When the limit hits the innermost *rpo-vn-max-loop-depth* loops and the outermost loop in the loop nest are value-numbered optimistically and the remaining ones not.

`sccvn-max-alias-queries-per-access`
: Maximum number of alias-oracle queries we perform when looking for redundancies for loads and stores. If this limit is hit the search is aborted and the load or store is not considered redundant. The number of queries is algorithmically limited to the number of stores on all paths from the load to the function entry.

`ira-max-loops-num`
: IRA uses regional register allocation by default. If a function contains more loops than the number given by this parameter, only at most the given number of the most frequently-executed loops form regions for regional register allocation.

`ira-max-conflict-table-size`
: Although IRA uses a sophisticated algorithm to compress the conflict table, the table can still require excessive amounts of memory for huge functions. If the conflict table for a function could be more than the size in MB given by this parameter, the register allocator instead uses a faster, simpler, and lower-quality algorithm that does not require building a pseudo-register conflict table.

`ira-loop-reserved-regs`
: IRA can be used to evaluate more accurate register pressure in loops for decisions to move loop invariants (see '-O3'). The number of available registers reserved for some other purposes is given by this parameter. Default of the parameter is the best found from numerous experiments.

`lra-inheritance-ebb-probability-cutoff`
: LRA tries to reuse values reloaded in registers in subsequent insns. This optimization is called inheritance. EBB is used as a region to do this optimization. The parameter defines a minimal fall-through edge probability in percentage used to add BB to inheritance EBB in LRA. The default value was chosen from numerous runs of SPEC2000 on x86-64.

`loop-invariant-max-bbs-in-loop`
: Loop invariant motion can be very expensive, both in compilation time and in amount of needed compile-time memory, with very large loops. Loops with more basic blocks than this parameter won't have loop invariant motion optimization performed on them.

`loop-max-datarefs-for-datadeps`
: Building data dependencies is expensive for very large loops. This parameter limits the number of data references in loops that are considered for data dependence analysis. These large loops are no handled by the optimizations using loop data dependencies.

`max-vartrack-size`
: Sets a maximum number of hash table slots to use during variable tracking dataflow analysis of any function. If this limit is exceeded with variable tracking at assignments enabled, analysis for that function is retried without it, after removing all debug insns from the function. If the limit is exceeded even without debug insns, var tracking analysis is completely disabled for the function. Setting the parameter to zero makes it unlimited.

`max-vartrack-expr-depth`
: Sets a maximum number of recursion levels when attempting to map variable names or debug temporaries to value expressions. This trades compilation time for more complete debug information. If this is set too low, value expressions that are available and could be represented in debug information may end up not being used; setting this higher may enable the compiler to find more complex debug expressions, but compile time and memory use may grow.

`max-debug-marker-count`
: Sets a threshold on the number of debug markers (e.g. begin stmt markers) to avoid complexity explosion at inlining or expanding to RTL. If a function has more such gimple stmts than the set limit, such stmts will be dropped from the inlined copy of a function, and from its RTL expansion.

`min-nondebug-insn-uid`
: Use uids starting at this parameter for nondebug insns. The range below the parameter is reserved exclusively for debug insns created by '`-fvar-tracking-assignments`', but debug insns may get (non-overlapping) uids above it if the reserved range is exhausted.

`ipa-sra-ptr-growth-factor`
: IPA-SRA replaces a pointer to an aggregate with one or more new parameters only when their cumulative size is less or equal to '`ipa-sra-ptr-growth-factor`' times the size of the original pointer parameter.

`sra-max-scalarization-size-Ospeed`
`sra-max-scalarization-size-Osize`
: The two Scalar Reduction of Aggregates passes (SRA and IPA-SRA) aim to replace scalar parts of aggregates with uses of independent scalar variables. These parameters control the maximum size, in storage units, of aggregate which is considered for replacement when compiling for speed ('`sra-max-scalarization-size-Ospeed`') or size ('`sra-max-scalarization-size-Osize`') respectively.

`tm-max-aggregate-size`
: When making copies of thread-local variables in a transaction, this parameter specifies the size in bytes after which variables are saved with the logging functions as opposed to save/restore code sequence pairs. This option only applies when using '`-fgnu-tm`'.

`graphite-max-nb-scop-params`
: To avoid exponential effects in the Graphite loop transforms, the number of parameters in a Static Control Part (SCoP) is bounded. A value of zero can be used to lift the bound. A variable whose value is unknown at compilation time and defined outside a SCoP is a parameter of the SCoP.

`loop-block-tile-size`
: Loop blocking or strip mining transforms, enabled with '`-floop-block`' or '`-floop-strip-mine`', strip mine each loop in the loop nest by a given number of iterations. The strip length can be changed using the '`loop-block-tile-size`' parameter.

`ipa-cp-value-list-size`
: IPA-CP attempts to track all possible values and types passed to a function's parameter in order to propagate them and perform devirtualization. '`ipa-cp-value-list-size`' is the maximum number of values and types it stores per one formal parameter of a function.

`ipa-cp-eval-threshold`
: IPA-CP calculates its own score of cloning profitability heuristics and performs those cloning opportunities with scores that exceed '`ipa-cp-eval-threshold`'.

`ipa-cp-recursion-penalty`
: Percentage penalty the recursive functions will receive when they are evaluated for cloning.

`ipa-cp-single-call-penalty`
: Percentage penalty functions containing a single call to another function will receive when they are evaluated for cloning.

`ipa-max-agg-items`
: IPA-CP is also capable to propagate a number of scalar values passed in an aggregate. '`ipa-max-agg-items`' controls the maximum number of such values per one parameter.

`ipa-cp-loop-hint-bonus`
When IPA-CP determines that a cloning candidate would make the number of iterations of a loop known, it adds a bonus of '`ipa-cp-loop-hint-bonus`' to the profitability score of the candidate.

`ipa-cp-array-index-hint-bonus`
When IPA-CP determines that a cloning candidate would make the index of an array access known, it adds a bonus of '`ipa-cp-array-index-hint-bonus`' to the profitability score of the candidate.

`ipa-max-aa-steps`
During its analysis of function bodies, IPA-CP employs alias analysis in order to track values pointed to by function parameters. In order not spend too much time analyzing huge functions, it gives up and consider all memory clobbered after examining '`ipa-max-aa-steps`' statements modifying memory.

`lto-partitions`
Specify desired number of partitions produced during WHOPR compilation. The number of partitions should exceed the number of CPUs used for compilation.

`lto-min-partition`
Size of minimal partition for WHOPR (in estimated instructions). This prevents expenses of splitting very small programs into too many partitions.

`lto-max-partition`
Size of max partition for WHOPR (in estimated instructions). to provide an upper bound for individual size of partition. Meant to be used only with balanced partitioning.

`lto-max-streaming-parallelism`
Maximal number of parallel processes used for LTO streaming.

`cxx-max-namespaces-for-diagnostic-help`
The maximum number of namespaces to consult for suggestions when C++ name lookup fails for an identifier.

`sink-frequency-threshold`
The maximum relative execution frequency (in percents) of the target block relative to a statement's original block to allow statement sinking of a statement. Larger numbers result in more aggressive statement sinking. A small positive adjustment is applied for statements with memory operands as those are even more profitable so sink.

`max-stores-to-sink`
The maximum number of conditional store pairs that can be sunk. Set to 0 if either vectorization ('`-ftree-vectorize`') or if-conversion ('`-ftree-loop-if-convert`') is disabled.

`allow-store-data-races`
: Allow optimizers to introduce new data races on stores. Set to 1 to allow, otherwise to 0.

`case-values-threshold`
: The smallest number of different values for which it is best to use a jump-table instead of a tree of conditional branches. If the value is 0, use the default for the machine.

`tree-reassoc-width`
: Set the maximum number of instructions executed in parallel in reassociated tree. This parameter overrides target dependent heuristics used by default if has non zero value.

`sched-pressure-algorithm`
: Choose between the two available implementations of '`-fsched-pressure`'. Algorithm 1 is the original implementation and is the more likely to prevent instructions from being reordered. Algorithm 2 was designed to be a compromise between the relatively conservative approach taken by algorithm 1 and the rather aggressive approach taken by the default scheduler. It relies more heavily on having a regular register file and accurate register pressure classes. See '`haifa-sched.c`' in the GCC sources for more details.

  The default choice depends on the target.

`max-slsr-cand-scan`
: Set the maximum number of existing candidates that are considered when seeking a basis for a new straight-line strength reduction candidate.

`asan-globals`
: Enable buffer overflow detection for global objects. This kind of protection is enabled by default if you are using '`-fsanitize=address`' option. To disable global objects protection use '`--param asan-globals=0`'.

`asan-stack`
: Enable buffer overflow detection for stack objects. This kind of protection is enabled by default when using '`-fsanitize=address`'. To disable stack protection use '`--param asan-stack=0`' option.

`asan-instrument-reads`
: Enable buffer overflow detection for memory reads. This kind of protection is enabled by default when using '`-fsanitize=address`'. To disable memory reads protection use '`--param asan-instrument-reads=0`'.

`asan-instrument-writes`
: Enable buffer overflow detection for memory writes. This kind of protection is enabled by default when using

'-fsanitize=address'. To disable memory writes protection use '--param asan-instrument-writes=0' option.

asan-memintrin
: Enable detection for built-in functions. This kind of protection is enabled by default when using '-fsanitize=address'. To disable built-in functions protection use '--param asan-memintrin=0'.

asan-use-after-return
: Enable detection of use-after-return. This kind of protection is enabled by default when using the '-fsanitize=address' option. To disable it use '--param asan-use-after-return=0'.

  Note: By default the check is disabled at run time. To enable it, add detect_stack_use_after_return=1 to the environment variable ASAN_OPTIONS.

asan-instrumentation-with-call-threshold
: If number of memory accesses in function being instrumented is greater or equal to this number, use callbacks instead of inline checks. E.g. to disable inline code use '--param asan-instrumentation-with-call-threshold=0'.

use-after-scope-direct-emission-threshold
: If the size of a local variable in bytes is smaller or equal to this number, directly poison (or unpoison) shadow memory instead of using run-time callbacks.

max-fsm-thread-path-insns
: Maximum number of instructions to copy when duplicating blocks on a finite state automaton jump thread path.

max-fsm-thread-length
: Maximum number of basic blocks on a finite state automaton jump thread path.

max-fsm-thread-paths
: Maximum number of new jump thread paths to create for a finite state automaton.

parloops-chunk-size
: Chunk size of omp schedule for loops parallelized by parloops.

parloops-schedule
: Schedule type of omp schedule for loops parallelized by parloops (static, dynamic, guided, auto, runtime).

parloops-min-per-thread
: The minimum number of iterations per thread of an innermost parallelized loop for which the parallelized variant is preferred over the single threaded one. Note that for a parallelized loop nest the minimum number of iterations of the outermost loop per thread is two.

`max-ssa-name-query-depth`
: Maximum depth of recursion when querying properties of SSA names in things like fold routines. One level of recursion corresponds to following a use-def chain.

`hsa-gen-debug-stores`
: Enable emission of special debug stores within HSA kernels which are then read and reported by libgomp plugin. Generation of these stores is disabled by default, use '`--param hsa-gen-debug-stores=1`' to enable it.

`max-speculative-devirt-maydefs`
: The maximum number of may-defs we analyze when looking for a must-def specifying the dynamic type of an object that invokes a virtual call we may be able to devirtualize speculatively.

`max-vrp-switch-assertions`
: The maximum number of assertions to add along the default edge of a switch statement during VRP.

`unroll-jam-min-percent`
: The minimum percentage of memory references that must be optimized away for the unroll-and-jam transformation to be considered profitable.

`unroll-jam-max-unroll`
: The maximum number of times the outer loop should be unrolled by the unroll-and-jam transformation.

`max-rtl-if-conversion-unpredictable-cost`
: Maximum permissible cost for the sequence that would be generated by the RTL if-conversion pass for a branch that is considered unpredictable.

`max-variable-expansions-in-unroller`
: If '`-fvariable-expansion-in-unroller`' is used, the maximum number of times that an individual variable will be expanded during loop unrolling.

`tracer-min-branch-probability-feedback`
: Stop forward growth if the probability of best edge is less than this threshold (in percent). Used when profile feedback is available.

`partial-inlining-entry-probability`
: Maximum probability of the entry BB of split region (in percent relative to entry BB of the function) to make partial inlining happen.

`max-tracked-strlens`
: Maximum number of strings for which strlen optimization pass will track string lengths.

`gcse-after-reload-partial-fraction`
The threshold ratio for performing partial redundancy elimination after reload.

`gcse-after-reload-critical-fraction`
The threshold ratio of critical edges execution count that permit performing redundancy elimination after reload.

`max-loop-header-insns`
The maximum number of insns in loop header duplicated by the copy loop headers pass.

`vect-epilogues-nomask`
Enable loop epilogue vectorization using smaller vector size.

`slp-max-insns-in-bb`
Maximum number of instructions in basic block to be considered for SLP vectorization.

`avoid-fma-max-bits`
Maximum number of bits for which we avoid creating FMAs.

`sms-loop-average-count-threshold`
A threshold on the average loop count considered by the swing modulo scheduler.

`sms-dfa-history`
The number of cycles the swing modulo scheduler considers when checking conflicts using DFA.

`hot-bb-count-fraction`
Select fraction of the maximal count of repetitions of basic block in program given basic block needs to have to be considered hot (used in non-LTO mode)

`max-inline-insns-recursive-auto`
The maximum number of instructions non-inline function can grow to via recursive inlining.

`graphite-allow-codegen-errors`
Whether codegen errors should be ICEs when '`-fchecking`'.

`sms-max-ii-factor`
A factor for tuning the upper bound that swing modulo scheduler uses for scheduling a loop.

`lra-max-considered-reload-pseudos`
The max number of reload pseudos which are considered during spilling a non-reload pseudo.

`max-pow-sqrt-depth`
Maximum depth of sqrt chains to use when synthesizing exponentiation by a real constant.

`max-dse-active-local-stores`
: Maximum number of active local stores in RTL dead store elimination.

`asan-instrument-allocas`
: Enable asan allocas/VLAs protection.

`max-iterations-computation-cost`
: Bound on the cost of an expression to compute the number of iterations.

`max-isl-operations`
: Maximum number of isl operations, 0 means unlimited.

`graphite-max-arrays-per-scop`
: Maximum number of arrays per scop.

`max-vartrack-reverse-op-size`
: Max. size of loc list for which reverse ops should be added.

`unlikely-bb-count-fraction`
: The minimum fraction of profile runs a given basic block execution count must be not to be considered unlikely.

`tracer-dynamic-coverage-feedback`
: The percentage of function, weighted by execution frequency, that must be covered by trace formation. Used when profile feedback is available.

`max-inline-recursive-depth-auto`
: The maximum depth of recursive inlining for non-inline functions.

`fsm-scale-path-stmts`
: Scale factor to apply to the number of statements in a threading path when comparing to the number of (scaled) blocks.

`fsm-maximum-phi-arguments`
: Maximum number of arguments a PHI may have before the FSM threader will not try to thread through its block.

`uninit-control-dep-attempts`
: Maximum number of nested calls to search for control dependencies during uninitialized variable analysis.

`indir-call-topn-profile`
: Track top N target addresses in indirect-call profile.

`max-once-peeled-insns`
: The maximum number of insns of a peeled loop that rolls only once.

`sra-max-scalarization-size-Osize`
: Maximum size, in storage units, of an aggregate which should be considered for scalarization when compiling for size.

`fsm-scale-path-blocks`
: Scale factor to apply to the number of blocks in a threading path when comparing to the number of (scaled) statements.

`sched-autopref-queue-depth`
: Hardware autoprefetcher scheduler model control flag. Number of lookahead cycles the model looks into; at ' ' only enable instruction sorting heuristic.

`loop-versioning-max-inner-insns`
: The maximum number of instructions that an inner loop can have before the loop versioning pass considers it too big to copy.

`loop-versioning-max-outer-insns`
: The maximum number of instructions that an outer loop can have before the loop versioning pass considers it too big to copy, discounting any instructions in inner loops that directly benefit from versioning.

3.11 Program Instrumentation Options

GCC supports a number of command-line options that control adding run-time instrumentation to the code it normally generates. For example, one purpose of instrumentation is collect profiling statistics for use in finding program hot spots, code coverage analysis, or profile-guided optimizations. Another class of program instrumentation is adding run-time checking to detect programming errors like invalid pointer dereferences or out-of-bounds array accesses, as well as deliberately hostile attacks such as stack smashing or C++ vtable hijacking. There is also a general hook which can be used to implement other forms of tracing or function-level instrumentation for debug or program analysis purposes.

`-p`
`-pg`
: Generate extra code to write profile information suitable for the analysis program `prof` (for '`-p`') or `gprof` (for '`-pg`'). You must use this option when compiling the source files you want data about, and you must also use it when linking.

  You can use the function attribute `no_instrument_function` to suppress profiling of individual functions when compiling with these options. See Section 6.33.1 [Common Function Attributes], page 478.

`-fprofile-arcs`
: Add code so that program flow *arcs* are instrumented. During execution the program records how many times each branch and call is executed and how many times it is taken or returns. On targets that support constructors with priority support, profiling properly handles constructors, destructors and C++ constructors (and destructors) of classes which are used as a type of a global variable.

  When the compiled program exits it saves this data to a file called '*auxname*`.gcda`' for each source file. The data may be used for profile-directed optimizations ('`-fbranch-probabilities`'), or for test coverage analysis ('`-ftest-coverage`'). Each object file's *auxname* is generated from the name

of the output file, if explicitly specified and it is not the final executable, otherwise it is the basename of the source file. In both cases any suffix is removed (e.g. '`foo.gcda`' for input file '`dir/foo.c`', or '`dir/foo.gcda`' for output file specified as '`-o dir/foo.o`'). See Section 10.5 [Cross-profiling], page 861.

`--coverage`

This option is used to compile and link code instrumented for coverage analysis. The option is a synonym for '`-fprofile-arcs`' '`-ftest-coverage`' (when compiling) and '`-lgcov`' (when linking). See the documentation for those options for more details.

- Compile the source files with '`-fprofile-arcs`' plus optimization and code generation options. For test coverage analysis, use the additional '`-ftest-coverage`' option. You do not need to profile every source file in a program.
- Compile the source files additionally with '`-fprofile-abs-path`' to create absolute path names in the '`.gcno`' files. This allows `gcov` to find the correct sources in projects where compilations occur with different working directories.
- Link your object files with '`-lgcov`' or '`-fprofile-arcs`' (the latter implies the former).
- Run the program on a representative workload to generate the arc profile information. This may be repeated any number of times. You can run concurrent instances of your program, and provided that the file system supports locking, the data files will be correctly updated. Unless a strict ISO C dialect option is in effect, `fork` calls are detected and correctly handled without double counting.
- For profile-directed optimizations, compile the source files again with the same optimization and code generation options plus '`-fbranch-probabilities`' (see Section 3.10 [Options that Control Optimization], page 125).
- For test coverage analysis, use `gcov` to produce human readable information from the '`.gcno`' and '`.gcda`' files. Refer to the `gcov` documentation for further information.

With '`-fprofile-arcs`', for each function of your program GCC creates a program flow graph, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

`-ftest-coverage`

Produce a notes file that the `gcov` code-coverage utility (see Chapter 10 [`gcov`—a Test Coverage Program], page 849) can use to show program coverage. Each source file's note file is called '*auxname*`.gcno`'. Refer to the '`-fprofile-arcs`' option above for a description of *auxname* and instructions on how to generate test coverage data. Coverage data matches the source files more closely if you do not optimize.

`-fprofile-abs-path`

Automatically convert relative source file names to absolute path names in the '`.gcno`' files. This allows `gcov` to find the correct sources in projects where compilations occur with different working directories.

`-fprofile-dir=`*`path`*

Set the directory to search for the profile data files in to *path*. This option affects only the profile data generated by '`-fprofile-generate`', '`-ftest-coverage`', '`-fprofile-arcs`' and used by '`-fprofile-use`' and '`-fbranch-probabilities`' and its related options. Both absolute and relative paths can be used. By default, GCC uses the current directory as *path*, thus the profile data file appears in the same directory as the object file. In order to prevent the file name clashing, if the object file name is not an absolute path, we mangle the absolute path of the '*sourcename*`.gcda`' file and use it as the file name of a '`.gcda`' file.

When an executable is run in a massive parallel environment, it is recommended to save profile to different folders. That can be done with variables in *path* that are exported during run-time:

`%p`	process ID.
`%q{VAR}`	value of environment variable *VAR*

`-fprofile-generate`
`-fprofile-generate=`*`path`*

Enable options usually used for instrumenting application to produce profile useful for later recompilation with profile feedback based optimization. You must use '`-fprofile-generate`' both when compiling and when linking your program.

The following options are enabled: '`-fprofile-arcs`', '`-fprofile-values`', '`-finline-functions`', and '`-fipa-bit-cp`'.

If *path* is specified, GCC looks at the *path* to find the profile feedback data files. See '`-fprofile-dir`'.

To optimize the program based on the collected profile information, use '`-fprofile-use`'. See Section 3.10 [Optimize Options], page 125, for more information.

`-fprofile-update=`*`method`*

Alter the update method for an application instrumented for profile feedback based optimization. The *method* argument should be one of '`single`', '`atomic`' or '`prefer-atomic`'. The first one is useful for single-threaded applications, while the second one prevents profile corruption by emitting thread-safe code.

Warning: When an application does not properly join all threads (or creates an detached thread), a profile file can be still corrupted.

Using '`prefer-atomic`' would be transformed either to '`atomic`', when supported by a target, or to '`single`' otherwise. The GCC driver automatically selects '`prefer-atomic`' when '`-pthread`' is present in the command line.

`-fprofile-filter-files=`*`regex`*

Instrument only functions from files where names match any regular expression (separated by a semi-colon).

For example, '`-fprofile-filter-files=main.c;module.*.c`' will instrument only '`main.c`' and all C files starting with 'module'.

`-fprofile-exclude-files=`*`regex`*

Instrument only functions from files where names do not match all the regular expressions (separated by a semi-colon).

For example, '`-fprofile-exclude-files=/usr/*`' will prevent instrumentation of all files that are located in '`/usr/`' folder.

`-fsanitize=address`

Enable AddressSanitizer, a fast memory error detector. Memory access instructions are instrumented to detect out-of-bounds and use-after-free bugs. The option enables '`-fsanitize-address-use-after-scope`'. See `https://github.com/google/sanitizers/wiki/AddressSanitizer` for more details. The run-time behavior can be influenced using the `ASAN_OPTIONS` environment variable. When set to `help=1`, the available options are shown at startup of the instrumented program. See `https://github.com/google/sanitizers/wiki/AddressSanitizerFlags#run-time-flags` for a list of supported options. The option cannot be combined with '`-fsanitize=thread`'.

`-fsanitize=kernel-address`

Enable AddressSanitizer for Linux kernel. See `https://github.com/google/kasan/wiki` for more details.

`-fsanitize=pointer-compare`

Instrument comparison operation (`<`, `<=`, `>`, `>=`) with pointer operands. The option must be combined with either '`-fsanitize=kernel-address`' or '`-fsanitize=address`' The option cannot be combined with '`-fsanitize=thread`'. Note: By default the check is disabled at run time. To enable it, add `detect_invalid_pointer_pairs=2` to the environment variable `ASAN_OPTIONS`. Using `detect_invalid_pointer_pairs=1` detects invalid operation only when both pointers are non-null.

`-fsanitize=pointer-subtract`

Instrument subtraction with pointer operands. The option must be combined with either '`-fsanitize=kernel-address`' or '`-fsanitize=address`' The option cannot be combined with '`-fsanitize=thread`'. Note: By default the check is disabled at run time. To enable it, add `detect_invalid_pointer_pairs=2` to the environment variable `ASAN_OPTIONS`. Using `detect_invalid_pointer_pairs=1` detects invalid operation only when both pointers are non-null.

`-fsanitize=thread`

Enable ThreadSanitizer, a fast data race detector. Memory access instructions are instrumented to detect data race bugs. See `https://github.com/google/sanitizers/wiki#threadsanitizer` for more details. The run-time behavior can be influenced using the `TSAN_OPTIONS` environment variable; see `https://github.com/google/sanitizers/wiki/ThreadSanitizerFlags` for a list of supported options. The option cannot be combined with '`-fsanitize=address`', '`-fsanitize=leak`'.

Note that sanitized atomic builtins cannot throw exceptions when operating on invalid memory addresses with non-call exceptions ('`-fnon-call-exceptions`').

`-fsanitize=leak`

Enable LeakSanitizer, a memory leak detector. This option only matters for linking of executables and the executable is linked against a library that overrides `malloc` and other allocator functions. See `https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer` for more details. The run-time behavior can be influenced using the `LSAN_OPTIONS` environment variable. The option cannot be combined with '`-fsanitize=thread`'.

`-fsanitize=undefined`

Enable UndefinedBehaviorSanitizer, a fast undefined behavior detector. Various computations are instrumented to detect undefined behavior at runtime. Current suboptions are:

`-fsanitize=shift`

This option enables checking that the result of a shift operation is not undefined. Note that what exactly is considered undefined differs slightly between C and C++, as well as between ISO C90 and C99, etc. This option has two suboptions, '`-fsanitize=shift-base`' and '`-fsanitize=shift-exponent`'.

`-fsanitize=shift-exponent`

This option enables checking that the second argument of a shift operation is not negative and is smaller than the precision of the promoted first argument.

`-fsanitize=shift-base`

If the second argument of a shift operation is within range, check that the result of a shift operation is not undefined. Note that what exactly is considered undefined differs slightly between C and C++, as well as between ISO C90 and C99, etc.

`-fsanitize=integer-divide-by-zero`

Detect integer division by zero as well as `INT_MIN / -1` division.

`-fsanitize=unreachable`

With this option, the compiler turns the `__builtin_unreachable` call into a diagnostics message call instead. When reaching the `__builtin_unreachable` call, the behavior is undefined.

`-fsanitize=vla-bound`
This option instructs the compiler to check that the size of a variable length array is positive.

`-fsanitize=null`
This option enables pointer checking. Particularly, the application built with this option turned on will issue an error message when it tries to dereference a NULL pointer, or if a reference (possibly an rvalue reference) is bound to a NULL pointer, or if a method is invoked on an object pointed by a NULL pointer.

`-fsanitize=return`
This option enables return statement checking. Programs built with this option turned on will issue an error message when the end of a non-void function is reached without actually returning a value. This option works in C++ only.

`-fsanitize=signed-integer-overflow`
This option enables signed integer overflow checking. We check that the result of `+`, `*`, and both unary and binary `-` does not overflow in the signed arithmetics. Note, integer promotion rules must be taken into account. That is, the following is not an overflow:

```
signed char a = SCHAR_MAX;
a++;
```

`-fsanitize=bounds`
This option enables instrumentation of array bounds. Various out of bounds accesses are detected. Flexible array members, flexible array member-like arrays, and initializers of variables with static storage are not instrumented.

`-fsanitize=bounds-strict`
This option enables strict instrumentation of array bounds. Most out of bounds accesses are detected, including flexible array members and flexible array member-like arrays. Initializers of variables with static storage are not instrumented.

`-fsanitize=alignment`
This option enables checking of alignment of pointers when they are dereferenced, or when a reference is bound to insufficiently aligned target, or when a method or constructor is invoked on insufficiently aligned object.

`-fsanitize=object-size`
This option enables instrumentation of memory references using the `__builtin_object_size` function. Various out of bounds pointer accesses are detected.

`-fsanitize=float-divide-by-zero`
Detect floating-point division by zero. Unlike other similar options, '`-fsanitize=float-divide-by-zero`' is not enabled by

'-fsanitize=undefined', since floating-point division by zero can be a legitimate way of obtaining infinities and NaNs.

-fsanitize=float-cast-overflow
This option enables floating-point type to integer conversion checking. We check that the result of the conversion does not overflow. Unlike other similar options, '-fsanitize=float-cast-overflow' is not enabled by '-fsanitize=undefined'. This option does not work well with FE_INVALID exceptions enabled.

-fsanitize=nonnull-attribute
This option enables instrumentation of calls, checking whether null values are not passed to arguments marked as requiring a non-null value by the nonnull function attribute.

-fsanitize=returns-nonnull-attribute
This option enables instrumentation of return statements in functions marked with returns_nonnull function attribute, to detect returning of null values from such functions.

-fsanitize=bool
This option enables instrumentation of loads from bool. If a value other than 0/1 is loaded, a run-time error is issued.

-fsanitize=enum
This option enables instrumentation of loads from an enum type. If a value outside the range of values for the enum type is loaded, a run-time error is issued.

-fsanitize=vptr
This option enables instrumentation of C++ member function calls, member accesses and some conversions between pointers to base and derived classes, to verify the referenced object has the correct dynamic type.

-fsanitize=pointer-overflow
This option enables instrumentation of pointer arithmetics. If the pointer arithmetics overflows, a run-time error is issued.

-fsanitize=builtin
This option enables instrumentation of arguments to selected builtin functions. If an invalid value is passed to such arguments, a run-time error is issued. E.g. passing 0 as the argument to __builtin_ctz or __builtin_clz invokes undefined behavior and is diagnosed by this option.

While '-ftrapv' causes traps for signed overflows to be emitted, '-fsanitize=undefined' gives a diagnostic message. This currently works only for the C family of languages.

-fno-sanitize=all
This option disables all previously enabled sanitizers. '-fsanitize=all' is not allowed, as some sanitizers cannot be used together.

`-fasan-shadow-offset=number`

This option forces GCC to use custom shadow offset in AddressSanitizer checks. It is useful for experimenting with different shadow memory layouts in Kernel AddressSanitizer.

`-fsanitize-sections=s1,s2,...`

Sanitize global variables in selected user-defined sections. *si* may contain wildcards.

`-fsanitize-recover`[`=opts`]

'`-fsanitize-recover=`' controls error recovery mode for sanitizers mentioned in comma-separated list of *opts*. Enabling this option for a sanitizer component causes it to attempt to continue running the program as if no error happened. This means multiple runtime errors can be reported in a single program run, and the exit code of the program may indicate success even when errors have been reported. The '`-fno-sanitize-recover=`' option can be used to alter this behavior: only the first detected error is reported and program then exits with a non-zero exit code.

Currently this feature only works for '`-fsanitize=undefined`' (and its suboptions except for '`-fsanitize=unreachable`' and '`-fsanitize=return`'), '`-fsanitize=float-cast-overflow`', '`-fsanitize=float-divide-by-zero`', '`-fsanitize=bounds-strict`', '`-fsanitize=kernel-address`' and '`-fsanitize=address`'. For these sanitizers error recovery is turned on by default, except '`-fsanitize=address`', for which this feature is experimental. '`-fsanitize-recover=all`' and '`-fno-sanitize-recover=all`' is also accepted, the former enables recovery for all sanitizers that support it, the latter disables recovery for all sanitizers that support it.

Even if a recovery mode is turned on the compiler side, it needs to be also enabled on the runtime library side, otherwise the failures are still fatal. The runtime library defaults to `halt_on_error=0` for ThreadSanitizer and UndefinedBehaviorSanitizer, while default value for AddressSanitizer is `halt_on_error=1`. This can be overridden through setting the `halt_on_error` flag in the corresponding environment variable.

Syntax without an explicit *opts* parameter is deprecated. It is equivalent to specifying an *opts* list of:

```
undefined,float-cast-overflow,float-divide-by-zero,bounds-strict
```

`-fsanitize-address-use-after-scope`

Enable sanitization of local variables to detect use-after-scope bugs. The option sets '`-fstack-reuse`' to '`none`'.

`-fsanitize-undefined-trap-on-error`

The '`-fsanitize-undefined-trap-on-error`' option instructs the compiler to report undefined behavior using `__builtin_trap` rather than a `libubsan` library routine. The advantage of this is that the `libubsan` library is not needed and is not linked in, so this is usable even in freestanding environments.

`-fsanitize-coverage=trace-pc`

Enable coverage-guided fuzzing code instrumentation. Inserts a call to `__sanitizer_cov_trace_pc` into every basic block.

`-fsanitize-coverage=trace-cmp`

Enable dataflow guided fuzzing code instrumentation. Inserts a call to `__sanitizer_cov_trace_cmp1`, `__sanitizer_cov_trace_cmp2`, `__sanitizer_cov_trace_cmp4` or `__sanitizer_cov_trace_cmp8` for integral comparison with both operands variable or `__sanitizer_cov_trace_const_cmp1`, `__sanitizer_cov_trace_const_cmp2`, `__sanitizer_cov_trace_const_cmp4` or `__sanitizer_cov_trace_const_cmp8` for integral comparison with one operand constant, `__sanitizer_cov_trace_cmpf` or `__sanitizer_cov_trace_cmpd` for float or double comparisons and `__sanitizer_cov_trace_switch` for switch statements.

`-fcf-protection=[full|branch|return|none]`

Enable code instrumentation of control-flow transfers to increase program security by checking that target addresses of control-flow transfer instructions (such as indirect function call, function return, indirect jump) are valid. This prevents diverting the flow of control to an unexpected target. This is intended to protect against such threats as Return-oriented Programming (ROP), and similarly call/jmp-oriented programming (COP/JOP).

The value `branch` tells the compiler to implement checking of validity of control-flow transfer at the point of indirect branch instructions, i.e. call/jmp instructions. The value `return` implements checking of validity at the point of returning from a function. The value `full` is an alias for specifying both `branch` and `return`. The value `none` turns off instrumentation.

The macro `__CET__` is defined when '`-fcf-protection`' is used. The first bit of `__CET__` is set to 1 for the value `branch` and the second bit of `__CET__` is set to 1 for the `return`.

You can also use the `nocf_check` attribute to identify which functions and calls should be skipped from instrumentation (see Section 6.33 [Function Attributes], page 477).

Currently the x86 GNU/Linux target provides an implementation based on Intel Control-flow Enforcement Technology (CET).

`-fstack-protector`

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call `alloca`, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

`-fstack-protector-all`

Like '`-fstack-protector`' except that all functions are protected.

`-fstack-protector-strong`

Like '`-fstack-protector`' but includes additional functions to be protected — those that have local array definitions, or have references to local frame addresses.

`-fstack-protector-explicit`

Like '`-fstack-protector`' but only protects those functions which have the `stack_protect` attribute.

`-fstack-check`

Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but you only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

Note that this switch does not actually cause checking to be done; the operating system or the language runtime must do that. The switch causes generation of code to ensure that they see the stack being extended.

You can additionally specify a string parameter: '`no`' means no checking, '`generic`' means force the use of old-style checking, '`specific`' means use the best checking method and is equivalent to bare '`-fstack-check`'.

Old-style checking is a generic mechanism that requires no specific target support in the compiler but comes with the following drawbacks:

1. Modified allocation strategy for large objects: they are always allocated dynamically if their size exceeds a fixed threshold. Note this may change the semantics of some code.
2. Fixed limit on the size of the static frame of functions: when it is topped by a particular function, stack checking is not reliable and a warning is issued by the compiler.
3. Inefficiency: because of both the modified allocation strategy and the generic implementation, code performance is hampered.

Note that old-style stack checking is also the fallback method for '`specific`' if no target support has been added in the compiler.

'`-fstack-check=`' is designed for Ada's needs to detect infinite recursion and stack overflows. '`specific`' is an excellent choice when compiling Ada code. It is not generally sufficient to protect against stack-clash attacks. To protect against those you want '`-fstack-clash-protection`'.

`-fstack-clash-protection`

Generate code to prevent stack clash style attacks. When this option is enabled, the compiler will only allocate one page of stack space at a time and each page is accessed immediately after allocation. Thus, it prevents allocations from jumping over any stack guard page provided by the operating system.

Most targets do not fully support stack clash protection. However, on those targets '`-fstack-clash-protection`' will protect dynamic stack allocations. '`-fstack-clash-protection`' may also provide limited protection for static stack allocations if the target supports '`-fstack-check=specific`'.

`-fstack-limit-register=reg`
`-fstack-limit-symbol=sym`
`-fno-stack-limit`

Generate code to ensure that the stack does not grow beyond a certain value, either the value of a register or the address of a symbol. If a larger stack is required, a signal is raised at run time. For most targets, the signal is raised before the stack overruns the boundary, so it is possible to catch the signal without taking special precautions.

For instance, if the stack starts at absolute address '`0x80000000`' and grows downwards, you can use the flags '`-fstack-limit-symbol=__stack_limit`' and '`-Wl,--defsym,__stack_limit=0x7ffe0000`' to enforce a stack limit of 128KB. Note that this may only work with the GNU linker.

You can locally override stack limit checking by using the `no_stack_limit` function attribute (see Section 6.33 [Function Attributes], page 477).

`-fsplit-stack`

Generate code to automatically split the stack before it overflows. The resulting program has a discontiguous stack which can only overflow if the program is unable to allocate any more memory. This is most useful when running threaded programs, as it is no longer necessary to calculate a good stack size to use for each thread. This is currently only implemented for the x86 targets running GNU/Linux.

When code compiled with '`-fsplit-stack`' calls code compiled without '`-fsplit-stack`', there may not be much stack space available for the latter code to run. If compiling all code, including library code, with '`-fsplit-stack`' is not an option, then the linker can fix up these calls so that the code compiled without '`-fsplit-stack`' always has a large stack. Support for this is implemented in the gold linker in GNU binutils release 2.21 and later.

`-fvtable-verify=[std|preinit|none]`

This option is only available when compiling C++ code. It turns on (or off, if using '`-fvtable-verify=none`') the security feature that verifies at run time, for every virtual call, that the vtable pointer through which the call is made is valid for the type of the object, and has not been corrupted or overwritten. If an invalid vtable pointer is detected at run time, an error is reported and execution of the program is immediately halted.

This option causes run-time data structures to be built at program startup, which are used for verifying the vtable pointers. The options '`std`' and '`preinit`' control the timing of when these data structures are built. In both cases the data structures are built before execution reaches `main`. Using '`-fvtable-verify=std`' causes the data structures to be built after shared libraries have been loaded and initialized. '`-fvtable-verify=preinit`' causes them to be built before shared libraries have been loaded and initialized.

If this option appears multiple times in the command line with different values specified, '`none`' takes highest priority over both '`std`' and '`preinit`'; '`preinit`' takes priority over '`std`'.

`-fvtv-debug`

When used in conjunction with '`-fvtable-verify=std`' or '`-fvtable-verify=preinit`', causes debug versions of the runtime functions for the vtable verification feature to be called. This flag also causes the compiler to log information about which vtable pointers it finds for each class. This information is written to a file named '`vtv_set_ptr_data.log`' in the directory named by the environment variable `VTV_LOGS_DIR` if that is defined or the current working directory otherwise.

Note: This feature *appends* data to the log file. If you want a fresh log file, be sure to delete any existing one.

`-fvtv-counts`

This is a debugging flag. When used in conjunction with '`-fvtable-verify=std`' or '`-fvtable-verify=preinit`', this causes the compiler to keep track of the total number of virtual calls it encounters and the number of verifications it inserts. It also counts the number of calls to certain run-time library functions that it inserts and logs this information for each compilation unit. The compiler writes this information to a file named '`vtv_count_data.log`' in the directory named by the environment variable `VTV_LOGS_DIR` if that is defined or the current working directory otherwise. It also counts the size of the vtable pointer sets for each class, and writes this information to '`vtv_class_set_sizes.log`' in the same directory.

Note: This feature *appends* data to the log files. To get fresh log files, be sure to delete any existing ones.

`-finstrument-functions`

Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions are called with the address of the current function and its call site. (On some platforms, `__builtin_return_address` does not work beyond the current function, so the call site information may not be available to the profiling functions otherwise.)

```
void __cyg_profile_func_enter (void *this_fn,
                               void *call_site);
void __cyg_profile_func_exit  (void *this_fn,
                               void *call_site);
```

The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table.

This instrumentation is also done for functions expanded inline in other functions. The profiling calls indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use `extern inline` in your C code, an addressable version of such functions must be provided. (This is normally the case anyway, but if you get lucky and the optimizer always expands the functions inline, you might have gotten away without providing static copies.)

A function may be given the attribute `no_instrument_function`, in which case this instrumentation is not done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory). See Section 6.33.1 [Common Function Attributes], page 478.

`-finstrument-functions-exclude-file-list=`*`file,file,...`*

Set the list of functions that are excluded from instrumentation (see the description of '`-finstrument-functions`'). If the file that contains a function definition matches with one of *file*, then that function is not instrumented. The match is done on substrings: if the *file* parameter is a substring of the file name, it is considered to be a match.

For example:

```
-finstrument-functions-exclude-file-list=/bits/stl,include/sys
```

excludes any inline function defined in files whose pathnames contain '`/bits/stl`' or '`include/sys`'.

If, for some reason, you want to include letter '`,`' in one of *sym*, write '`\,`'. For example, '`-finstrument-functions-exclude-file-list='\,\,tmp''` (note the single quote surrounding the option).

`-finstrument-functions-exclude-function-list=`*`sym,sym,...`*

This is similar to '`-finstrument-functions-exclude-file-list`', but this option sets the list of function names to be excluded from instrumentation. The function name to be matched is its user-visible name, such as `vector<int> blah(const vector<int> &)`, not the internal mangled name (e.g., `_Z4blahRSt6vectorIiSaIiEE`). The match is done on substrings: if the *sym* parameter is a substring of the function name, it is considered to be a match. For C99 and C++ extended identifiers, the function name must be given in UTF-8, not using universal character names.

`-fpatchable-function-entry=`*`N`*`[,`*`M`*`]`

Generate *N* NOPs right at the beginning of each function, with the function entry point before the *M*th NOP. If *M* is omitted, it defaults to `0` so the function entry points to the address just at the first NOP. The NOP instructions reserve extra space which can be used to patch in any desired instrumentation at run time, provided that the code segment is writable. The amount of space is controllable indirectly via the number of NOPs; the NOP instruction used corresponds to the instruction emitted by the internal GCC back-end interface `gen_nop`. This behavior is target-specific and may also depend on the architecture variant and/or other compilation options.

For run-time identification, the starting addresses of these areas, which correspond to their respective function entries minus *M*, are additionally collected in the `__patchable_function_entries` section of the resulting binary.

Note that the value of `__attribute__ ((patchable_function_entry (N,M)))` takes precedence over command-line option '`-fpatchable-function-entry=N,M`'. This can be used to increase

the area size or to remove it completely on a single function. If `N=0`, no pad location is recorded.

The NOP instructions are inserted at—and maybe before, depending on *M*—the function entry address, even before the prologue.

3.12 Options Controlling the Preprocessor

These options control the C preprocessor, which is run on each C source file before actual compilation.

If you use the '`-E`' option, nothing is done except preprocessing. Some of these options make sense only together with '`-E`' because they cause the preprocessor output to be unsuitable for actual compilation.

In addition to the options listed here, there are a number of options to control search paths for include files documented in Section 3.15 [Directory Options], page 215. Options to control preprocessor diagnostics are listed in Section 3.8 [Warning Options], page 70.

`-D` *`name`*
: Predefine *name* as a macro, with definition `1`.

`-D` *`name=definition`*
: The contents of *definition* are tokenized and processed as if they appeared during translation phase three in a '`#define`' directive. In particular, the definition is truncated by embedded newline characters.

  If you are invoking the preprocessor from a shell or shell-like program you may need to use the shell's quoting syntax to protect characters such as spaces that have a meaning in the shell syntax.

  If you wish to define a function-like macro on the command line, write its argument list with surrounding parentheses before the equals sign (if any). Parentheses are meaningful to most shells, so you should quote the option. With `sh` and `csh`, '`-D'name(args...)=definition'`' works.

  '`-D`' and '`-U`' options are processed in the order they are given on the command line. All '`-imacros` *`file`*' and '`-include` *`file`*' options are processed after all '`-D`' and '`-U`' options.

`-U` *`name`*
: Cancel any previous definition of *name*, either built in or provided with a '`-D`' option.

`-include` *`file`*
: Process *file* as if `#include "file"` appeared as the first line of the primary source file. However, the first directory searched for *file* is the preprocessor's working directory *instead of* the directory containing the main source file. If not found there, it is searched for in the remainder of the `#include "..."` search chain as normal.

  If multiple '`-include`' options are given, the files are included in the order they appear on the command line.

`-imacros` *`file`*
: Exactly like '`-include`', except that any output produced by scanning *file* is thrown away. Macros it defines remain defined. This allows you to acquire all the macros from a header without also processing its declarations.

All files specified by '-imacros' are processed before all files specified by '-include'.

-undef Do not predefine any system-specific or GCC-specific macros. The standard predefined macros remain defined.

-pthread Define additional macros required for using the POSIX threads library. You should use this option consistently for both compilation and linking. This option is supported on GNU/Linux targets, most other Unix derivatives, and also on x86 Cygwin and MinGW targets.

-M Instead of outputting the result of preprocessing, output a rule suitable for make describing the dependencies of the main source file. The preprocessor outputs one make rule containing the object file name for that source file, a colon, and the names of all the included files, including those coming from '-include' or '-imacros' command-line options.

Unless specified explicitly (with '-MT' or '-MQ'), the object file name consists of the name of the source file with any suffix replaced with object file suffix and with any leading directory parts removed. If there are many included files then the rule is split into several lines using '\'-newline. The rule has no commands.

This option does not suppress the preprocessor's debug output, such as '-dM'. To avoid mixing such debug output with the dependency rules you should explicitly specify the dependency output file with '-MF', or use an environment variable like DEPENDENCIES_OUTPUT (see Section 3.20 [Environment Variables], page 436). Debug output is still sent to the regular output stream as normal.

Passing '-M' to the driver implies '-E', and suppresses warnings with an implicit '-w'.

-MM Like '-M' but do not mention header files that are found in system header directories, nor header files that are included, directly or indirectly, from such a header.

This implies that the choice of angle brackets or double quotes in an '#include' directive does not in itself determine whether that header appears in '-MM' dependency output.

-MF *file* When used with '-M' or '-MM', specifies a file to write the dependencies to. If no '-MF' switch is given the preprocessor sends the rules to the same place it would send preprocessed output.

When used with the driver options '-MD' or '-MMD', '-MF' overrides the default dependency output file.

If *file* is '-', then the dependencies are written to 'stdout'.

-MG In conjunction with an option such as '-M' requesting dependency generation, '-MG' assumes missing header files are generated files and adds them to the dependency list without raising an error. The dependency filename is taken directly from the #include directive without prepending any path. '-MG' also suppresses preprocessed output, as a missing header file renders this useless.

This feature is used in automatic updating of makefiles.

`-MP` This option instructs CPP to add a phony target for each dependency other than the main file, causing each to depend on nothing. These dummy rules work around errors `make` gives if you remove header files without updating the '`Makefile`' to match.

This is typical output:

```
test.o: test.c test.h

test.h:
```

`-MT` *target*

Change the target of the rule emitted by dependency generation. By default CPP takes the name of the main input file, deletes any directory components and any file suffix such as '`.c`', and appends the platform's usual object suffix. The result is the target.

An '`-MT`' option sets the target to be exactly the string you specify. If you want multiple targets, you can specify them as a single argument to '`-MT`', or use multiple '`-MT`' options.

For example, '`-MT '$(objpfx)foo.o'`' might give

```
$(objpfx)foo.o: foo.c
```

`-MQ` *target*

Same as '`-MT`', but it quotes any characters which are special to Make. '`-MQ '$(objpfx)foo.o'`' gives

```
$$(objpfx)foo.o: foo.c
```

The default target is automatically quoted, as if it were given with '`-MQ`'.

`-MD` '`-MD`' is equivalent to '`-M -MF` *file*', except that '`-E`' is not implied. The driver determines *file* based on whether an '`-o`' option is given. If it is, the driver uses its argument but with a suffix of '`.d`', otherwise it takes the name of the input file, removes any directory components and suffix, and applies a '`.d`' suffix.

If '`-MD`' is used in conjunction with '`-E`', any '`-o`' switch is understood to specify the dependency output file (see [-MF], page 203), but if used without '`-E`', each '`-o`' is understood to specify a target object file.

Since '`-E`' is not implied, '`-MD`' can be used to generate a dependency output file as a side effect of the compilation process.

`-MMD` Like '`-MD`' except mention only user header files, not system header files.

`-fpreprocessed`

Indicate to the preprocessor that the input file has already been preprocessed. This suppresses things like macro expansion, trigraph conversion, escaped newline splicing, and processing of most directives. The preprocessor still recognizes and removes comments, so that you can pass a file preprocessed with '`-C`' to the compiler without problems. In this mode the integrated preprocessor is little more than a tokenizer for the front ends.

'`-fpreprocessed`' is implicit if the input file has one of the extensions '`.i`', '`.ii`' or '`.mi`'. These are the extensions that GCC uses for preprocessed files created by '`-save-temps`'.

`-fdirectives-only`

When preprocessing, handle directives, but do not expand macros.

The option's behavior depends on the '`-E`' and '`-fpreprocessed`' options.

With '`-E`', preprocessing is limited to the handling of directives such as `#define`, `#ifdef`, and `#error`. Other preprocessor operations, such as macro expansion and trigraph conversion are not performed. In addition, the '`-dD`' option is implicitly enabled.

With '`-fpreprocessed`', predefinition of command line and most builtin macros is disabled. Macros such as `__LINE__`, which are contextually dependent, are handled normally. This enables compilation of files previously preprocessed with `-E -fdirectives-only`.

With both '`-E`' and '`-fpreprocessed`', the rules for '`-fpreprocessed`' take precedence. This enables full preprocessing of files previously preprocessed with `-E -fdirectives-only`.

`-fdollars-in-identifiers`

Accept '`$`' in identifiers.

`-fextended-identifiers`

Accept universal character names in identifiers. This option is enabled by default for C99 (and later C standard versions) and C++.

`-fno-canonical-system-headers`

When preprocessing, do not shorten system header paths with canonicalization.

`-ftabstop=`*`width`*

Set the distance between tab stops. This helps the preprocessor report correct column numbers in warnings or errors, even if tabs appear on the line. If the value is less than 1 or greater than 100, the option is ignored. The default is 8.

`-ftrack-macro-expansion`[`=`*`level`*]

Track locations of tokens across macro expansions. This allows the compiler to emit diagnostic about the current macro expansion stack when a compilation error occurs in a macro expansion. Using this option makes the preprocessor and the compiler consume more memory. The *level* parameter can be used to choose the level of precision of token location tracking thus decreasing the memory consumption if necessary. Value '`0`' of *level* de-activates this option. Value '`1`' tracks tokens locations in a degraded mode for the sake of minimal memory overhead. In this mode all tokens resulting from the expansion of an argument of a function-like macro have the same location. Value '`2`' tracks tokens locations completely. This value is the most memory hungry. When this option is given no argument, the default parameter value is '`2`'.

Note that `-ftrack-macro-expansion=2` is activated by default.

`-fmacro-prefix-map=`*`old`*`=`*`new`*

When preprocessing files residing in directory '*`old`*', expand the `__FILE__` and `__BASE_FILE__` macros as if the files resided in directory '*`new`*' instead. This can be used to change an absolute path to a relative path by using '`.`' for

new which can result in more reproducible builds that are location independent. This option also affects `__builtin_FILE()` during compilation. See also '`-ffile-prefix-map`'.

`-fexec-charset=`*charset*
: Set the execution character set, used for string and character constants. The default is UTF-8. *charset* can be any encoding supported by the system's `iconv` library routine.

`-fwide-exec-charset=`*charset*
: Set the wide execution character set, used for wide string and character constants. The default is UTF-32 or UTF-16, whichever corresponds to the width of `wchar_t`. As with '`-fexec-charset`', *charset* can be any encoding supported by the system's `iconv` library routine; however, you will have problems with encodings that do not fit exactly in `wchar_t`.

`-finput-charset=`*charset*
: Set the input character set, used for translation from the character set of the input file to the source character set used by GCC. If the locale does not specify, or GCC cannot get this information from the locale, the default is UTF-8. This can be overridden by either the locale or this command-line option. Currently the command-line option takes precedence if there's a conflict. *charset* can be any encoding supported by the system's `iconv` library routine.

`-fpch-deps`
: When using precompiled headers (see Section 3.21 [Precompiled Headers], page 438), this flag causes the dependency-output flags to also list the files from the precompiled header's dependencies. If not specified, only the precompiled header are listed and not the files that were used to create it, because those files are not consulted when a precompiled header is used.

`-fpch-preprocess`
: This option allows use of a precompiled header (see Section 3.21 [Precompiled Headers], page 438) together with '`-E`'. It inserts a special `#pragma`, `#pragma GCC pch_preprocess "`*filename*`"` in the output to mark the place where the precompiled header was found, and its *filename*. When '`-fpreprocessed`' is in use, GCC recognizes this `#pragma` and loads the PCH.

  This option is off by default, because the resulting preprocessed output is only really suitable as input to GCC. It is switched on by '`-save-temps`'.

  You should not write this `#pragma` in your own code, but it is safe to edit the filename if the PCH file is available in a different location. The filename may be absolute or it may be relative to GCC's current directory.

`-fworking-directory`
: Enable generation of linemarkers in the preprocessor output that let the compiler know the current working directory at the time of preprocessing. When this option is enabled, the preprocessor emits, after the initial linemarker, a second linemarker with the current working directory followed by two slashes. GCC uses this directory, when it's present in the preprocessed input, as the directory emitted as the current working directory in some debugging information

formats. This option is implicitly enabled if debugging information is enabled, but this can be inhibited with the negated form '`-fno-working-directory`'. If the '`-P`' flag is present in the command line, this option has no effect, since no `#line` directives are emitted whatsoever.

`-A` *predicate=answer*

Make an assertion with the predicate *predicate* and answer *answer*. This form is preferred to the older form '`-A` *predicate(answer)*', which is still supported, because it does not use shell special characters.

`-A -`*predicate=answer*

Cancel an assertion with the predicate *predicate* and answer *answer*.

`-C`

Do not discard comments. All comments are passed through to the output file, except for comments in processed directives, which are deleted along with the directive.

You should be prepared for side effects when using '`-C`'; it causes the preprocessor to treat comments as tokens in their own right. For example, comments appearing at the start of what would be a directive line have the effect of turning that line into an ordinary source line, since the first token on the line is no longer a '`#`'.

`-CC`

Do not discard comments, including during macro expansion. This is like '`-C`', except that comments contained within macros are also passed through to the output file where the macro is expanded.

In addition to the side effects of the '`-C`' option, the '`-CC`' option causes all C++-style comments inside a macro to be converted to C-style comments. This is to prevent later use of that macro from inadvertently commenting out the remainder of the source line.

The '`-CC`' option is generally used to support lint comments.

`-P`

Inhibit generation of linemarkers in the output from the preprocessor. This might be useful when running the preprocessor on something that is not C code, and will be sent to a program which might be confused by the linemarkers.

`-traditional`
`-traditional-cpp`

Try to imitate the behavior of pre-standard C preprocessors, as opposed to ISO C preprocessors. See the GNU CPP manual for details.

Note that GCC does not otherwise attempt to emulate a pre-standard C compiler, and these options are only supported with the '`-E`' switch, or when invoking CPP explicitly.

`-trigraphs`

Support ISO C trigraphs. These are three-character sequences, all starting with '`??`', that are defined by ISO C to stand for single characters. For example, '`??/`' stands for '`\`', so '`'??/n'`' is a character constant for a newline.

The nine trigraphs and their replacements are

```
Trigraph:       ??(  ??)  ??<  ??>  ??=  ??/  ??'  ??!  ??-
Replacement:      [    ]    {    }    #    \    ^    |    ~
```

By default, GCC ignores trigraphs, but in standard-conforming modes it converts them. See the '-std' and '-ansi' options.

-remap Enable special code to work around file systems which only permit very short file names, such as MS-DOS.

-H Print the name of each header file used, in addition to other normal activities. Each name is indented to show how deep in the '#include' stack it is. Precompiled header files are also printed, even if they are found to be invalid; an invalid precompiled header file is printed with '...x' and a valid one with '...!' .

-d*letters*

Says to make debugging dumps during compilation as specified by *letters*. The flags documented here are those relevant to the preprocessor. Other *letters* are interpreted by the compiler proper, or reserved for future versions of GCC, and so are silently ignored. If you specify *letters* whose behavior conflicts, the result is undefined. See Section 3.17 [Developer Options], page 228, for more information.

-dM Instead of the normal output, generate a list of '#define' directives for all the macros defined during the execution of the preprocessor, including predefined macros. This gives you a way of finding out what is predefined in your version of the preprocessor. Assuming you have no file 'foo.h', the command

```
touch foo.h; cpp -dM foo.h
```

shows all the predefined macros.

If you use '-dM' without the '-E' option, '-dM' is interpreted as a synonym for '-fdump-rtl-mach'. See Section "Developer Options" in gcc.

-dD Like '-dM' except in two respects: it does *not* include the predefined macros, and it outputs *both* the '#define' directives and the result of preprocessing. Both kinds of output go to the standard output file.

-dN Like '-dD', but emit only the macro names, not their expansions.

-dI Output '#include' directives in addition to the result of preprocessing.

-dU Like '-dD' except that only macros that are expanded, or whose definedness is tested in preprocessor directives, are output; the output is delayed until the use or test of the macro; and '#undef' directives are also output for macros tested but undefined at the time.

-fdebug-cpp

This option is only useful for debugging GCC. When used from CPP or with '-E', it dumps debugging information about location maps. Every token in the output is preceded by the dump of the map its location belongs to.

When used from GCC without '-E', this option has no effect.

`-Wp,option`

You can use '`-Wp,option`' to bypass the compiler driver and pass *option* directly through to the preprocessor. If *option* contains commas, it is split into multiple options at the commas. However, many options are modified, translated or interpreted by the compiler driver before being passed to the preprocessor, and '`-Wp`' forcibly bypasses this phase. The preprocessor's direct interface is undocumented and subject to change, so whenever possible you should avoid using '`-Wp`' and let the driver handle the options instead.

`-Xpreprocessor option`

Pass *option* as an option to the preprocessor. You can use this to supply system-specific preprocessor options that GCC does not recognize.

If you want to pass an option that takes an argument, you must use '`-Xpreprocessor`' twice, once for the option and once for the argument.

`-no-integrated-cpp`

Perform preprocessing as a separate pass before compilation. By default, GCC performs preprocessing as an integrated part of input tokenization and parsing. If this option is provided, the appropriate language front end (`cc1`, `cc1plus`, or `cc1obj` for C, C++, and Objective-C, respectively) is instead invoked twice, once for preprocessing only and once for actual compilation of the preprocessed input. This option may be useful in conjunction with the '`-B`' or '`-wrapper`' options to specify an alternate preprocessor or perform additional processing of the program source between normal preprocessing and compilation.

3.13 Passing Options to the Assembler

You can pass options to the assembler.

`-Wa,option`

Pass *option* as an option to the assembler. If *option* contains commas, it is split into multiple options at the commas.

`-Xassembler option`

Pass *option* as an option to the assembler. You can use this to supply system-specific assembler options that GCC does not recognize.

If you want to pass an option that takes an argument, you must use '`-Xassembler`' twice, once for the option and once for the argument.

3.14 Options for Linking

These options come into play when the compiler links object files into an executable output file. They are meaningless if the compiler is not doing a link step.

`object-file-name`

A file name that does not end in a special recognized suffix is considered to name an object file or library. (Object files are distinguished from libraries by the linker according to the file contents.) If linking is done, these object files are used as input to the linker.

`-c`
`-S`
`-E`
If any of these options is used, then the linker is not run, and object file names should not be used as arguments. See Section 3.2 [Overall Options], page 31.

`-flinker-output=`*type*
This option controls the code generation of the link time optimizer. By default the linker output is determined by the linker plugin automatically. For debugging the compiler and in the case of incremental linking to non-lto object file is desired, it may be useful to control the type manually.

If *type* is '`exec`' the code generation is configured to produce static binary. In this case '`-fpic`' and '`-fpie`' are both disabled.

If *type* is '`dyn`' the code generation is configured to produce shared library. In this case '`-fpic`' or '`-fPIC`' is preserved, but not enabled automatically. This makes it possible to build shared libraries without position independent code on architectures this is possible, i.e. on x86.

If *type* is '`pie`' the code generation is configured to produce '`-fpie`' executable. This result in similar optimizations as '`exec`' except that '`-fpie`' is not disabled if specified at compilation time.

If *type* is '`rel`' the compiler assumes that incremental linking is done. The sections containing intermediate code for link-time optimization are merged, pre-optimized, and output to the resulting object file. In addition, if '`-ffat-lto-objects`' is specified the binary code is produced for future non-lto linking. The object file produced by incremental linking will be smaller than a static library produced from the same object files. At link-time the result of incremental linking will also load faster to compiler than a static library assuming that majority of objects in the library are used.

Finally '`nolto-rel`' configure compiler to for incremental linking where code generation is forced, final binary is produced and the intermediate code for later link-time optimization is stripped. When multiple object files are linked together the resulting code will be optimized better than with link time optimizations disabled (for example, the cross-module inlining will happen), most of benefits of whole program optimizations are however lost.

During the incremental link (by '`-r`') the linker plugin will default to '`rel`'. With current interfaces to GNU Binutils it is however not possible to link incrementally LTO objects and non-LTO objects into a single mixed object file. In the case any of object files in incremental link cannot be used for link-time optimization the linker plugin will output warning and use '`nolto-rel`'. To maintain the whole program optimization it is recommended to link such objects into static library instead. Alternatively it is possible to use H.J. Lu's binutils with support for mixed objects.

`-fuse-ld=bfd`
Use the `bfd` linker instead of the default linker.

`-fuse-ld=gold`
Use the `gold` linker instead of the default linker.

`-fuse-ld=lld`
: Use the LLVM `lld` linker instead of the default linker.

`-l`*library*
`-l` *library*
: Search the library named *library* when linking. (The second alternative with the library as a separate argument is only for POSIX compliance and is not recommended.)

The '`-l`' option is passed directly to the linker by GCC. Refer to your linker documentation for exact details. The general description below applies to the GNU linker.

The linker searches a standard list of directories for the library. The directories searched include several standard system directories plus any that you specify with '`-L`'.

Static libraries are archives of object files, and have file names like '`lib`*library*`.a`'. Some targets also support shared libraries, which typically have names like '`lib`*library*`.so`'. If both static and shared libraries are found, the linker gives preference to linking with the shared library unless the '`-static`' option is used.

It makes a difference where in the command you write this option; the linker searches and processes libraries and object files in the order they are specified. Thus, '`foo.o -lz bar.o`' searches library '`z`' after file '`foo.o`' but before '`bar.o`'. If '`bar.o`' refers to functions in '`z`', those functions may not be loaded.

`-lobjc`
: You need this special case of the '`-l`' option in order to link an Objective-C or Objective-C++ program.

`-nostartfiles`
: Do not use the standard system startup files when linking. The standard system libraries are used normally, unless '`-nostdlib`', '`-nolibc`', or '`-nodefaultlibs`' is used.

`-nodefaultlibs`
: Do not use the standard system libraries when linking. Only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as '`-static-libgcc`' or '`-shared-libgcc`', are ignored. The standard startup files are used normally, unless '`-nostartfiles`' is used.

The compiler may generate calls to `memcmp`, `memset`, `memcpy` and `memmove`. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

`-nolibc`
: Do not use the C library or system libraries tightly coupled with it when linking. Still link with the startup files, '`libgcc`' or toolchain provided language support libraries such as '`libgnat`', '`libgfortran`' or '`libstdc++`' unless options preventing their inclusion are used as well. This typically removes '`-lc`' from the link command line, as well as system libraries that normally go with it and become meaningless when absence of a C library is assumed, for example '`-lpthread`' or '`-lm`' in some configurations. This is intended for bare-board targets when there is indeed no C library available.

`-nostdlib`

Do not use the standard system startup files or libraries when linking. No startup files and only the libraries you specify are passed to the linker, and options specifying linkage of the system libraries, such as '`-static-libgcc`' or '`-shared-libgcc`', are ignored.

The compiler may generate calls to `memcmp`, `memset`, `memcpy` and `memmove`. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

One of the standard libraries bypassed by '`-nostdlib`' and '`-nodefaultlibs`' is '`libgcc.a`', a library of internal subroutines which GCC uses to overcome shortcomings of particular machines, or special needs for some languages. (See Section "Interfacing to GCC Output" in *GNU Compiler Collection (GCC) Internals*, for more discussion of '`libgcc.a`'.) In most cases, you need '`libgcc.a`' even when you want to avoid other standard libraries. In other words, when you specify '`-nostdlib`' or '`-nodefaultlibs`' you should usually specify '`-lgcc`' as well. This ensures that you have no unresolved references to internal GCC library subroutines. (An example of such an internal subroutine is `__main`, used to ensure C++ constructors are called; see Section "`collect2`" in *GNU Compiler Collection (GCC) Internals*.)

`-e` *entry*
`--entry=`*entry*

Specify that the program entry point is *entry*. The argument is interpreted by the linker; the GNU linker accepts either a symbol name or an address.

`-pie` Produce a dynamically linked position independent executable on targets that support it. For predictable results, you must also specify the same set of options used for compilation ('`-fpie`', '`-fPIE`', or model suboptions) when you specify this linker option.

`-no-pie` Don't produce a dynamically linked position independent executable.

`-static-pie`

Produce a static position independent executable on targets that support it. A static position independent executable is similar to a static executable, but can be loaded at any address without a dynamic linker. For predictable results, you must also specify the same set of options used for compilation ('`-fpie`', '`-fPIE`', or model suboptions) when you specify this linker option.

`-pthread` Link with the POSIX threads library. This option is supported on GNU/Linux targets, most other Unix derivatives, and also on x86 Cygwin and MinGW targets. On some targets this option also sets flags for the preprocessor, so it should be used consistently for both compilation and linking.

`-r` Produce a relocatable object as output. This is also known as partial linking.

`-rdynamic`

Pass the flag '`-export-dynamic`' to the ELF linker, on targets that support it. This instructs the linker to add all symbols, not only used ones, to the dynamic symbol table. This option is needed for some uses of `dlopen` or to allow obtaining backtraces from within a program.

`-s` Remove all symbol table and relocation information from the executable.

`-static` On systems that support dynamic linking, this overrides '`-pie`' and prevents linking with the shared libraries. On other systems, this option has no effect.

`-shared` Produce a shared object which can then be linked with other objects to form an executable. Not all systems support this option. For predictable results, you must also specify the same set of options used for compilation ('`-fpic`', '`-fPIC`', or model suboptions) when you specify this linker option.[1]

`-shared-libgcc`
`-static-libgcc`

On systems that provide '`libgcc`' as a shared library, these options force the use of either the shared or static version, respectively. If no shared version of '`libgcc`' was built when the compiler was configured, these options have no effect.

There are several situations in which an application should use the shared '`libgcc`' instead of the static version. The most common of these is when the application wishes to throw and catch exceptions across different shared libraries. In that case, each of the libraries as well as the application itself should use the shared '`libgcc`'.

Therefore, the G++ driver automatically adds '`-shared-libgcc`' whenever you build a shared library or a main executable, because C++ programs typically use exceptions, so this is the right thing to do.

If, instead, you use the GCC driver to create shared libraries, you may find that they are not always linked with the shared '`libgcc`'. If GCC finds, at its configuration time, that you have a non-GNU linker or a GNU linker that does not support option '`--eh-frame-hdr`', it links the shared version of '`libgcc`' into shared libraries by default. Otherwise, it takes advantage of the linker and optimizes away the linking with the shared version of '`libgcc`', linking with the static version of libgcc by default. This allows exceptions to propagate through such shared libraries, without incurring relocation costs at library load time.

However, if a library or main executable is supposed to throw or catch exceptions, you must link it using the G++ driver, or using the option '`-shared-libgcc`', such that it is linked with the shared '`libgcc`'.

`-static-libasan`

When the '`-fsanitize=address`' option is used to link a program, the GCC driver automatically links against '`libasan`'. If '`libasan`' is available as a shared library, and the '`-static`' option is not used, then this links against the shared version of '`libasan`'. The '`-static-libasan`' option directs the GCC driver to link '`libasan`' statically, without necessarily linking other libraries statically.

[1] On some systems, '`gcc -shared`' needs to build supplementary stub code for constructors to work. On multi-libbed systems, '`gcc -shared`' must select the correct support libraries to link against. Failing to supply the correct flags may lead to subtle defects. Supplying them in cases where they are not necessary is innocuous.

`-static-libtsan`
When the '`-fsanitize=thread`' option is used to link a program, the GCC driver automatically links against '`libtsan`'. If '`libtsan`' is available as a shared library, and the '`-static`' option is not used, then this links against the shared version of '`libtsan`'. The '`-static-libtsan`' option directs the GCC driver to link '`libtsan`' statically, without necessarily linking other libraries statically.

`-static-liblsan`
When the '`-fsanitize=leak`' option is used to link a program, the GCC driver automatically links against '`liblsan`'. If '`liblsan`' is available as a shared library, and the '`-static`' option is not used, then this links against the shared version of '`liblsan`'. The '`-static-liblsan`' option directs the GCC driver to link '`liblsan`' statically, without necessarily linking other libraries statically.

`-static-libubsan`
When the '`-fsanitize=undefined`' option is used to link a program, the GCC driver automatically links against '`libubsan`'. If '`libubsan`' is available as a shared library, and the '`-static`' option is not used, then this links against the shared version of '`libubsan`'. The '`-static-libubsan`' option directs the GCC driver to link '`libubsan`' statically, without necessarily linking other libraries statically.

`-static-libstdc++`
When the `g++` program is used to link a C++ program, it normally automatically links against '`libstdc++`'. If '`libstdc++`' is available as a shared library, and the '`-static`' option is not used, then this links against the shared version of '`libstdc++`'. That is normally fine. However, it is sometimes useful to freeze the version of '`libstdc++`' used by the program without going all the way to a fully static link. The '`-static-libstdc++`' option directs the `g++` driver to link '`libstdc++`' statically, without necessarily linking other libraries statically.

`-symbolic`
Bind references to global symbols when building a shared object. Warn about any unresolved references (unless overridden by the link editor option '`-Xlinker -z -Xlinker defs`'). Only a few systems support this option.

`-T` *`script`* Use *script* as the linker script. This option is supported by most systems using the GNU linker. On some targets, such as bare-board targets without an operating system, the '`-T`' option may be required when linking to avoid references to undefined symbols.

`-Xlinker` *`option`*
Pass *option* as an option to the linker. You can use this to supply system-specific linker options that GCC does not recognize.

If you want to pass an option that takes a separate argument, you must use '`-Xlinker`' twice, once for the option and once for the argument. For example, to pass '`-assert definitions`', you must write '`-Xlinker -assert -Xlinker definitions`'. It does not work to write '`-Xlinker "-assert definitions"`', because this passes the entire string as a single argument, which is not what the linker expects.

When using the GNU linker, it is usually more convenient to pass arguments to linker options using the '*option=value*' syntax than as separate arguments. For example, you can specify '`-Xlinker -Map=output.map`' rather than '`-Xlinker -Map -Xlinker output.map`'. Other linkers may not support this syntax for command-line options.

`-Wl,`*`option`*

Pass *option* as an option to the linker. If *option* contains commas, it is split into multiple options at the commas. You can use this syntax to pass an argument to the option. For example, '`-Wl,-Map,output.map`' passes '`-Map output.map`' to the linker. When using the GNU linker, you can also get the same effect with '`-Wl,-Map=output.map`'.

`-u` *`symbol`* Pretend the symbol *symbol* is undefined, to force linking of library modules to define it. You can use '`-u`' multiple times with different symbols to force loading of additional library modules.

`-z` *`keyword`*

'`-z`' is passed directly on to the linker along with the keyword *keyword*. See the section in the documentation of your linker for permitted values and their meanings.

3.15 Options for Directory Search

These options specify directories to search for header files, for libraries and for parts of the compiler:

`-I` *`dir`*
`-iquote` *`dir`*
`-isystem` *`dir`*
`-idirafter` *`dir`*

Add the directory *dir* to the list of directories to be searched for header files during preprocessing. If *dir* begins with '`=`' or `$SYSROOT`, then the '`=`' or `$SYSROOT` is replaced by the sysroot prefix; see '`--sysroot`' and '`-isysroot`'.

Directories specified with '`-iquote`' apply only to the quote form of the directive, `#include "`*`file`*`"`. Directories specified with '`-I`', '`-isystem`', or '`-idirafter`' apply to lookup for both the `#include "`*`file`*`"` and `#include <`*`file`*`>` directives.

You can specify any number or combination of these options on the command line to search for header files in several directories. The lookup order is as follows:

1. For the quote form of the include directive, the directory of the current file is searched first.
2. For the quote form of the include directive, the directories specified by '`-iquote`' options are searched in left-to-right order, as they appear on the command line.
3. Directories specified with '`-I`' options are scanned in left-to-right order.
4. Directories specified with '`-isystem`' options are scanned in left-to-right order.

5. Standard system directories are scanned.
6. Directories specified with '`-idirafter`' options are scanned in left-to-right order.

You can use '`-I`' to override a system header file, substituting your own version, since these directories are searched before the standard system header file directories. However, you should not use this option to add directories that contain vendor-supplied system header files; use '`-isystem`' for that.

The '`-isystem`' and '`-idirafter`' options also mark the directory as a system directory, so that it gets the same special treatment that is applied to the standard system directories.

If a standard system include directory, or a directory specified with '`-isystem`', is also specified with '`-I`', the '`-I`' option is ignored. The directory is still searched but as a system directory at its normal position in the system include chain. This is to ensure that GCC's procedure to fix buggy system headers and the ordering for the `#include_next` directive are not inadvertently changed. If you really need to change the search order for system directories, use the '`-nostdinc`' and/or '`-isystem`' options.

`-I-` Split the include path. This option has been deprecated. Please use '`-iquote`' instead for '`-I`' directories before the '`-I-`' and remove the '`-I-`' option.

Any directories specified with '`-I`' options before '`-I-`' are searched only for headers requested with `#include "file"`; they are not searched for `#include <file>`. If additional directories are specified with '`-I`' options after the '`-I-`', those directories are searched for all '`#include`' directives.

In addition, '`-I-`' inhibits the use of the directory of the current file directory as the first search directory for `#include "file"`. There is no way to override this effect of '`-I-`'.

`-iprefix prefix`
Specify *prefix* as the prefix for subsequent '`-iwithprefix`' options. If the prefix represents a directory, you should include the final '`/`'.

`-iwithprefix dir`
`-iwithprefixbefore dir`
Append *dir* to the prefix specified previously with '`-iprefix`', and add the resulting directory to the include search path. '`-iwithprefixbefore`' puts it in the same place '`-I`' would; '`-iwithprefix`' puts it where '`-idirafter`' would.

`-isysroot dir`
This option is like the '`--sysroot`' option, but applies only to header files (except for Darwin targets, where it applies to both header files and libraries). See the '`--sysroot`' option for more information.

`-imultilib dir`
Use *dir* as a subdirectory of the directory containing target-specific C++ headers.

`-nostdinc`
Do not search the standard system directories for header files. Only the directories explicitly specified with '`-I`', '`-iquote`', '`-isystem`', and/or '`-idirafter`' options (and the directory of the current file, if appropriate) are searched.

`-nostdinc++`
: Do not search for header files in the C++-specific standard directories, but do still search the other standard directories. (This option is used when building the C++ library.)

`-iplugindir=dir`
: Set the directory to search for plugins that are passed by '`-fplugin=name`' instead of '`-fplugin=path/name.so`'. This option is not meant to be used by the user, but only passed by the driver.

`-Ldir`
: Add directory *dir* to the list of directories to be searched for '`-l`'.

`-Bprefix`
: This option specifies where to find the executables, libraries, include files, and data files of the compiler itself.

  The compiler driver program runs one or more of the subprograms `cpp`, `cc1`, `as` and `ld`. It tries *prefix* as a prefix for each program it tries to run, both with and without '`machine/version/`' for the corresponding target machine and compiler version.

  For each subprogram to be run, the compiler driver first tries the '`-B`' prefix, if any. If that name is not found, or if '`-B`' is not specified, the driver tries two standard prefixes, '`/usr/lib/gcc/`' and '`/usr/local/lib/gcc/`'. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your `PATH` environment variable.

  The compiler checks to see if the path provided by '`-B`' refers to a directory, and if necessary it adds a directory separator character at the end of the path.

  '`-B`' prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into '`-L`' options for the linker. They also apply to include files in the preprocessor, because the compiler translates these options into '`-isystem`' options for the preprocessor. In this case, the compiler appends '`include`' to the prefix.

  The runtime support file '`libgcc.a`' can also be searched for using the '`-B`' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means.

  Another way to specify a prefix much like the '`-B`' prefix is to use the environment variable `GCC_EXEC_PREFIX`. See Section 3.20 [Environment Variables], page 436.

  As a special kludge, if the path provided by '`-B`' is '`[dir/]stageN/`', where *N* is a number in the range 0 to 9, then it is replaced by '`[dir/]include`'. This is to help with boot-strapping the compiler.

`-no-canonical-prefixes`
: Do not expand any symbolic links, resolve references to '`/../`' or '`/./`', or make the path absolute when generating a relative prefix.

`--sysroot=dir`
: Use *dir* as the logical root directory for headers and libraries. For example, if the compiler normally searches for headers in '`/usr/include`' and libraries in '`/usr/lib`', it instead searches '`dir/usr/include`' and '`dir/usr/lib`'.

If you use both this option and the '-isysroot' option, then the '--sysroot' option applies to libraries, but the '-isysroot' option applies to header files.

The GNU linker (beginning with version 2.16) has the necessary support for this option. If your linker does not support this option, the header file aspect of '--sysroot' still works, but the library aspect does not.

--no-sysroot-suffix

For some targets, a suffix is added to the root directory specified with '--sysroot', depending on the other options used, so that headers may for example be found in '*dir*/*suffix*/usr/include' instead of '*dir*/usr/include'. This option disables the addition of such a suffix.

3.16 Options for Code Generation Conventions

These machine-independent options control the interface conventions used in code generation.

Most of them have both positive and negative forms; the negative form of '-ffoo' is '-fno-foo'. In the table below, only one of the forms is listed—the one that is not the default. You can figure out the other form by either removing 'no-' or adding it.

-fstack-reuse=*reuse-level*

This option controls stack space reuse for user declared local/auto variables and compiler generated temporaries. *reuse_level* can be 'all', 'named_vars', or 'none'. 'all' enables stack reuse for all local variables and temporaries, 'named_vars' enables the reuse only for user defined local variables with names, and 'none' disables stack reuse completely. The default value is 'all'. The option is needed when the program extends the lifetime of a scoped local variable or a compiler generated temporary beyond the end point defined by the language. When a lifetime of a variable ends, and if the variable lives in memory, the optimizing compiler has the freedom to reuse its stack space with other temporaries or scoped local variables whose live range does not overlap with it. Legacy code extending local lifetime is likely to break with the stack reuse optimization.

For example,

```
int *p;
{
  int local1;

  p = &local1;
  local1 = 10;
  ....
}
{
   int local2;
   local2 = 20;
   ...
}

if (*p == 10)  // out of scope use of local1
  {
```

```
}
```

Another example:

```
struct A
{
    A(int k) : i(k), j(k) { }
    int i;
    int j;
};

A *ap;

void foo(const A& ar)
{
   ap = &ar;
}

void bar()
{
   foo(A(10)); // temp object's lifetime ends when foo returns

   {
     A a(20);
     ....
   }
   ap->i+= 10;  // ap references out of scope temp whose space
                // is reused with a. What is the value of ap->i?
}
```

The lifetime of a compiler generated temporary is well defined by the C++ standard. When a lifetime of a temporary ends, and if the temporary lives in memory, the optimizing compiler has the freedom to reuse its stack space with other temporaries or scoped local variables whose live range does not overlap with it. However some of the legacy code relies on the behavior of older compilers in which temporaries' stack space is not reused, the aggressive stack reuse can lead to runtime errors. This option is used to control the temporary stack reuse optimization.

`-ftrapv` This option generates traps for signed overflow on addition, subtraction, multiplication operations. The options '`-ftrapv`' and '`-fwrapv`' override each other, so using '`-ftrapv`' '`-fwrapv`' on the command-line results in '`-fwrapv`' being effective. Note that only active options override, so using '`-ftrapv`' '`-fwrapv`' '`-fno-wrapv`' on the command-line results in '`-ftrapv`' being effective.

`-fwrapv` This option instructs the compiler to assume that signed arithmetic overflow of addition, subtraction and multiplication wraps around using twos-complement representation. This flag enables some optimizations and disables others. The options '`-ftrapv`' and '`-fwrapv`' override each other, so using '`-ftrapv`' '`-fwrapv`' on the command-line results in '`-fwrapv`' being effective. Note that only active options override, so using '`-ftrapv`' '`-fwrapv`' '`-fno-wrapv`' on the command-line results in '`-ftrapv`' being effective.

`-fwrapv-pointer`
This option instructs the compiler to assume that pointer arithmetic overflow on addition and subtraction wraps around using twos-complement representation. This flag disables some optimizations which assume pointer overflow is invalid.

`-fstrict-overflow`
This option implies '`-fno-wrapv`' '`-fno-wrapv-pointer`' and when negated implies '`-fwrapv`' '`-fwrapv-pointer`'.

`-fexceptions`
Enable exception handling. Generates extra code needed to propagate exceptions. For some targets, this implies GCC generates frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution. If you do not specify this option, GCC enables it by default for languages like C++ that normally require exception handling, and disables it for languages like C that do not normally require it. However, you may need to enable this option when compiling C code that needs to interoperate properly with exception handlers written in C++. You may also wish to disable this option if you are compiling older C++ programs that don't use exception handling.

`-fnon-call-exceptions`
Generate code that allows trapping instructions to throw exceptions. Note that this requires platform-specific runtime support that does not exist everywhere. Moreover, it only allows *trapping* instructions to throw exceptions, i.e. memory references or floating-point instructions. It does not allow exceptions to be thrown from arbitrary signal handlers such as `SIGALRM`.

`-fdelete-dead-exceptions`
Consider that instructions that may throw exceptions but don't otherwise contribute to the execution of the program can be optimized away. This option is enabled by default for the Ada front end, as permitted by the Ada language specification. Optimization passes that cause dead exceptions to be removed are enabled independently at different optimization levels.

`-funwind-tables`
Similar to '`-fexceptions`', except that it just generates any needed static data, but does not affect the generated code in any other way. You normally do not need to enable this option; instead, a language processor that needs this handling enables it on your behalf.

`-fasynchronous-unwind-tables`
Generate unwind table in DWARF format, if supported by target machine. The table is exact at each instruction boundary, so it can be used for stack unwinding from asynchronous events (such as debugger or garbage collector).

`-fno-gnu-unique`
On systems with recent GNU assembler and C library, the C++ compiler uses the `STB_GNU_UNIQUE` binding to make sure that definitions of template static data members and static local variables in inline functions are unique even in the presence of `RTLD_LOCAL`; this is necessary to avoid problems with a library

used by two different `RTLD_LOCAL` plugins depending on a definition in one of them and therefore disagreeing with the other one about the binding of the symbol. But this causes `dlclose` to be ignored for affected DSOs; if your program relies on reinitialization of a DSO via `dlclose` and `dlopen`, you can use '`-fno-gnu-unique`'.

`-fpcc-struct-return`

Return "short" `struct` and `union` values in memory like longer ones, rather than in registers. This convention is less efficient, but it has the advantage of allowing intercallability between GCC-compiled files and files compiled with other compilers, particularly the Portable C Compiler (pcc).

The precise convention for returning structures in memory depends on the target configuration macros.

Short structures and unions are those whose size and alignment match that of some integer type.

Warning: code compiled with the '`-fpcc-struct-return`' switch is not binary compatible with code compiled with the '`-freg-struct-return`' switch. Use it to conform to a non-default application binary interface.

`-freg-struct-return`

Return `struct` and `union` values in registers when possible. This is more efficient for small structures than '`-fpcc-struct-return`'.

If you specify neither '`-fpcc-struct-return`' nor '`-freg-struct-return`', GCC defaults to whichever convention is standard for the target. If there is no standard convention, GCC defaults to '`-fpcc-struct-return`', except on targets where GCC is the principal compiler. In those cases, we can choose the standard, and we chose the more efficient register return alternative.

Warning: code compiled with the '`-freg-struct-return`' switch is not binary compatible with code compiled with the '`-fpcc-struct-return`' switch. Use it to conform to a non-default application binary interface.

`-fshort-enums`

Allocate to an `enum` type only as many bytes as it needs for the declared range of possible values. Specifically, the `enum` type is equivalent to the smallest integer type that has enough room.

Warning: the '`-fshort-enums`' switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-fshort-wchar`

Override the underlying type for `wchar_t` to be `short unsigned int` instead of the default for the target. This option is useful for building programs to run under WINE.

Warning: the '`-fshort-wchar`' switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-fno-common`

In C code, this option controls the placement of global variables defined without an initializer, known as *tentative definitions* in the C standard. Tentative definitions are distinct from declarations of a variable with the `extern` keyword, which do not allocate storage.

Unix C compilers have traditionally allocated storage for uninitialized global variables in a common block. This allows the linker to resolve all tentative definitions of the same variable in different compilation units to the same object, or to a non-tentative definition. This is the behavior specified by '`-fcommon`', and is the default for GCC on most targets. On the other hand, this behavior is not required by ISO C, and on some targets may carry a speed or code size penalty on variable references.

The '`-fno-common`' option specifies that the compiler should instead place uninitialized global variables in the BSS section of the object file. This inhibits the merging of tentative definitions by the linker so you get a multiple-definition error if the same variable is defined in more than one compilation unit. Compiling with '`-fno-common`' is useful on targets for which it provides better performance, or if you wish to verify that the program will work on other systems that always treat uninitialized variable definitions this way.

`-fno-ident`

Ignore the `#ident` directive.

`-finhibit-size-directive`

Don't output a `.size` assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling '`crtstuff.c`'; you should not need to use it for anything else.

`-fverbose-asm`

Put extra commentary information in the generated assembly code to make it more readable. This option is generally only of use to those who actually need to read the generated assembly code (perhaps while debugging the compiler itself).

'`-fno-verbose-asm`', the default, causes the extra information to be omitted and is useful when comparing two assembler files.

The added comments include:

- information on the compiler version and command-line options,
- the source code lines associated with the assembly instructions, in the form FILENAME:LINENUMBER:CONTENT OF LINE,
- hints on which high-level expressions correspond to the various assembly instruction operands.

For example, given this C source file:

```
int test (int n)
{
  int i;
  int total = 0;
```

```
  for (i = 0; i < n; i++)
    total += i * i;

  return total;
}
```

compiling to (x86_64) assembly via '-S' and emitting the result direct to stdout via '-o' '-'

```
gcc -S test.c -fverbose-asm -Os -o -
```

gives output similar to this:

```
.file "test.c"
# GNU C11 (GCC) version 7.0.0 20160809 (experimental) (x86_64-pc-linux-gnu)
  [...snip...]
# options passed:
  [...snip...]

.text
.globl test
.type test, @function
test:
.LFB0:
.cfi_startproc
# test.c:4:   int total = 0;
xorl %eax, %eax # <retval>
# test.c:6:   for (i = 0; i < n; i++)
xorl %edx, %edx # i
.L2:
# test.c:6:   for (i = 0; i < n; i++)
cmpl %edi, %edx # n, i
jge .L5 #,
# test.c:7:     total += i * i;
movl %edx, %ecx # i, tmp92
imull %edx, %ecx # i, tmp92
# test.c:6:   for (i = 0; i < n; i++)
incl %edx # i
# test.c:7:     total += i * i;
addl %ecx, %eax # tmp92, <retval>
jmp .L2 #
.L5:
# test.c:10: }
ret
.cfi_endproc
.LFE0:
.size test, .-test
.ident "GCC: (GNU) 7.0.0 20160809 (experimental)"
.section .note.GNU-stack,"",@progbits
```

The comments are intended for humans rather than machines and hence the precise format of the comments is subject to change.

`-frecord-gcc-switches`

This switch causes the command line used to invoke the compiler to be recorded into the object file that is being created. This switch is only implemented on some targets and the exact format of the recording is target and binary file format dependent, but it usually takes the form of a section containing ASCII text. This switch is related to the '-fverbose-asm' switch, but that switch

only records information in the assembler output file as comments, so it never reaches the object file. See also '`-grecord-gcc-switches`' for another way of storing compiler options into the object file.

`-fpic` Generate position-independent code (PIC) suitable for use in a shared library, if supported for the target machine. Such code accesses all constant addresses through a global offset table (GOT). The dynamic loader resolves the GOT entries when the program starts (the dynamic loader is not part of GCC; it is part of the operating system). If the GOT size for the linked executable exceeds a machine-specific maximum size, you get an error message from the linker indicating that '`-fpic`' does not work; in that case, recompile with '`-fPIC`' instead. (These maximums are 8k on the SPARC, 28k on AArch64 and 32k on the m68k and RS/6000. The x86 has no such limit.)

Position-independent code requires special support, and therefore works only on certain machines. For the x86, GCC supports PIC for System V but not for the Sun 386i. Code generated for the IBM RS/6000 is always position-independent.

When this flag is set, the macros `__pic__` and `__PIC__` are defined to 1.

`-fPIC` If supported for the target machine, emit position-independent code, suitable for dynamic linking and avoiding any limit on the size of the global offset table. This option makes a difference on AArch64, m68k, PowerPC and SPARC.

Position-independent code requires special support, and therefore works only on certain machines.

When this flag is set, the macros `__pic__` and `__PIC__` are defined to 2.

`-fpie`
`-fPIE` These options are similar to '`-fpic`' and '`-fPIC`', but the generated position-independent code can be only linked into executables. Usually these options are used to compile code that will be linked using the '`-pie`' GCC option.

'`-fpie`' and '`-fPIE`' both define the macros `__pie__` and `__PIE__`. The macros have the value 1 for '`-fpie`' and 2 for '`-fPIE`'.

`-fno-plt` Do not use the PLT for external function calls in position-independent code. Instead, load the callee address at call sites from the GOT and branch to it. This leads to more efficient code by eliminating PLT stubs and exposing GOT loads to optimizations. On architectures such as 32-bit x86 where PLT stubs expect the GOT pointer in a specific register, this gives more register allocation freedom to the compiler. Lazy binding requires use of the PLT; with '`-fno-plt`' all external symbols are resolved at load time.

Alternatively, the function attribute `noplt` can be used to avoid calls through the PLT for specific external functions.

In position-dependent code, a few targets also convert calls to functions that are marked to not use the PLT to use the GOT instead.

`-fno-jump-tables`
Do not use jump tables for switch statements even where it would be more efficient than other code generation strategies. This option is of use in conjunction with '`-fpic`' or '`-fPIC`' for building code that forms part of a dynamic linker

and cannot reference the address of a jump table. On some targets, jump tables do not require a GOT and this option is not needed.

`-ffixed-reg`

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the `REGISTER_NAMES` macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-used-reg`

Treat the register named *reg* as an allocable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way do not save and restore the register *reg*.

It is an error to use this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

`-fcall-saved-reg`

Treat the register named *reg* as an allocable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way save and restore the register *reg* if they use it.

It is an error to use this flag with the frame pointer or stack pointer. Use of this flag for other registers that have fixed pervasive roles in the machine's execution model produces disastrous results.

A different sort of disaster results from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

`-fpack-struct[=n]`

Without a value specified, pack all structure members together without holes. When a value is specified (which must be a small power of two), pack structure members according to this value, representing the maximum alignment (that is, objects with default alignment requirements larger than this are output potentially unaligned at the next fitting location.

Warning: the '`-fpack-struct`' switch causes GCC to generate code that is not binary compatible with code generated without that switch. Additionally, it makes the code suboptimal. Use it to conform to a non-default application binary interface.

`-fleading-underscore`

This option and its counterpart, '`-fno-leading-underscore`', forcibly change the way C symbols are represented in the object file. One use is to help link with legacy assembly code.

Warning: the '`-fleading-underscore`' switch causes GCC to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface. Not all targets provide complete support for this switch.

`-ftls-model=model`

Alter the thread-local storage model to be used (see Section 6.64 [Thread-Local], page 810). The *model* argument should be one of '`global-dynamic`', '`local-dynamic`', '`initial-exec`' or '`local-exec`'. Note that the choice is subject to optimization: the compiler may use a more efficient model for symbols not visible outside of the translation unit, or if '`-fpic`' is not given on the command line.

The default without '`-fpic`' is '`initial-exec`'; with '`-fpic`' the default is '`global-dynamic`'.

`-ftrampolines`

For targets that normally need trampolines for nested functions, always generate them instead of using descriptors. Otherwise, for targets that do not need them, like for example HP-PA or IA-64, do nothing.

A trampoline is a small piece of code that is created at run time on the stack when the address of a nested function is taken, and is used to call the nested function indirectly. Therefore, it requires the stack to be made executable in order for the program to work properly.

'`-fno-trampolines`' is enabled by default on a language by language basis to let the compiler avoid generating them, if it computes that this is safe, and replace them with descriptors. Descriptors are made up of data only, but the generated code must be prepared to deal with them. As of this writing, '`-fno-trampolines`' is enabled by default only for Ada.

Moreover, code compiled with '`-ftrampolines`' and code compiled with '`-fno-trampolines`' are not binary compatible if nested functions are present. This option must therefore be used on a program-wide basis and be manipulated with extreme care.

`-fvisibility=[default|internal|hidden|protected]`

Set the default ELF image symbol visibility to the specified option—all symbols are marked with this unless overridden within the code. Using this feature can very substantially improve linking and load times of shared object libraries, produce more optimized code, provide near-perfect API export and prevent symbol clashes. It is **strongly** recommended that you use this in any shared objects you distribute.

Despite the nomenclature, '`default`' always means public; i.e., available to be linked against from outside the shared object. '`protected`' and '`internal`' are pretty useless in real-world usage so the only other commonly used option is '`hidden`'. The default if '`-fvisibility`' isn't specified is '`default`', i.e., make every symbol public.

A good explanation of the benefits offered by ensuring ELF symbols have the correct visibility is given by "How To Write Shared Libraries" by Ulrich

Drepper (which can be found at `https://www.akkadia.org/drepper/`)—however a superior solution made possible by this option to marking things hidden when the default is public is to make the default hidden and mark things public. This is the norm with DLLs on Windows and with '`-fvisibility=hidden`' and `__attribute__ ((visibility("default")))` instead of `__declspec(dllexport)` you get almost identical semantics with identical syntax. This is a great boon to those working with cross-platform projects.

For those adding visibility support to existing code, you may find `#pragma GCC visibility` of use. This works by you enclosing the declarations you wish to set visibility for with (for example) `#pragma GCC visibility push(hidden)` and `#pragma GCC visibility pop`. Bear in mind that symbol visibility should be viewed **as part of the API interface contract** and thus all new code should always specify visibility when it is not the default; i.e., declarations only for use within the local DSO should **always** be marked explicitly as hidden as so to avoid PLT indirection overheads—making this abundantly clear also aids readability and self-documentation of the code. Note that due to ISO C++ specification requirements, `operator new` and `operator delete` must always be of default visibility.

Be aware that headers from outside your project, in particular system headers and headers from any other library you use, may not be expecting to be compiled with visibility other than the default. You may need to explicitly say `#pragma GCC visibility push(default)` before including any such headers.

`extern` declarations are not affected by '`-fvisibility`', so a lot of code can be recompiled with '`-fvisibility=hidden`' with no modifications. However, this means that calls to `extern` functions with no explicit visibility use the PLT, so it is more effective to use `__attribute ((visibility))` and/or `#pragma GCC visibility` to tell the compiler which `extern` declarations should be treated as hidden.

Note that '`-fvisibility`' does affect C++ vague linkage entities. This means that, for instance, an exception class that is be thrown between DSOs must be explicitly marked with default visibility so that the '`type_info`' nodes are unified between the DSOs.

An overview of these techniques, their benefits and how to use them is at `http://gcc.gnu.org/wiki/Visibility`.

`-fstrict-volatile-bitfields`

This option should be used if accesses to volatile bit-fields (or other structure fields, although the compiler usually honors those types anyway) should use a single access of the width of the field's type, aligned to a natural alignment if possible. For example, targets with memory-mapped peripheral registers might require all such accesses to be 16 bits wide; with this flag you can declare all peripheral bit-fields as `unsigned short` (assuming short is 16 bits on these targets) to force GCC to use 16-bit accesses instead of, perhaps, a more efficient 32-bit access.

If this option is disabled, the compiler uses the most efficient instruction. In the previous example, that might be a 32-bit load instruction, even though that accesses bytes that do not contain any portion of the bit-field, or memory-mapped registers unrelated to the one being updated.

In some cases, such as when the `packed` attribute is applied to a structure field, it may not be possible to access the field with a single read or write that is correctly aligned for the target machine. In this case GCC falls back to generating multiple accesses rather than code that will fault or truncate the result at run time.

Note: Due to restrictions of the C/C++11 memory model, write accesses are not allowed to touch non bit-field members. It is therefore recommended to define all bits of the field's type as bit-field members.

The default value of this option is determined by the application binary interface for the target processor.

`-fsync-libcalls`

This option controls whether any out-of-line instance of the `__sync` family of functions may be used to implement the C++11 `__atomic` family of functions.

The default value of this option is enabled, thus the only useful form of the option is '`-fno-sync-libcalls`'. This option is used in the implementation of the '`libatomic`' runtime library.

3.17 GCC Developer Options

This section describes command-line options that are primarily of interest to GCC developers, including options to support compiler testing and investigation of compiler bugs and compile-time performance problems. This includes options that produce debug dumps at various points in the compilation; that print statistics such as memory use and execution time; and that print information about GCC's configuration, such as where it searches for libraries. You should rarely need to use any of these options for ordinary compilation and linking tasks.

Many developer options that cause GCC to dump output to a file take an optional '`=`*`filename`*' suffix. You can specify '`stdout`' or '`-`' to dump to standard output, and '`stderr`' for standard error.

If '`=`*`filename`*' is omitted, a default dump file name is constructed by concatenating the base dump file name, a pass number, phase letter, and pass name. The base dump file name is the name of output file produced by the compiler if explicitly specified and not an executable; otherwise it is the source file name. The pass number is determined by the order passes are registered with the compiler's pass manager. This is generally the same as the order of execution, but passes registered by plugins, target-specific passes, or passes that are otherwise registered late are numbered higher than the pass named '`final`', even if they are executed earlier. The phase letter is one of '`i`' (inter-procedural analysis), '`l`' (language-specific), '`r`' (RTL), or '`t`' (tree). The files are created in the directory of the output file.

`-dletters`
`-fdump-rtl-pass`
`-fdump-rtl-pass=filename`

Says to make debugging dumps during compilation at times specified by *letters*. This is used for debugging the RTL-based passes of the compiler.

Some '`-dletters`' switches have different meaning when '`-E`' is used for preprocessing. See Section 3.12 [Preprocessor Options], page 202, for information about preprocessor-specific dump options.

Debug dumps can be enabled with a '`-fdump-rtl`' switch or some '`-d`' option *letters*. Here are the possible letters for use in *pass* and *letters*, and their meanings:

`-fdump-rtl-alignments`
: Dump after branch alignments have been computed.

`-fdump-rtl-asmcons`
: Dump after fixing rtl statements that have unsatisfied in/out constraints.

`-fdump-rtl-auto_inc_dec`
: Dump after auto-inc-dec discovery. This pass is only run on architectures that have auto inc or auto dec instructions.

`-fdump-rtl-barriers`
: Dump after cleaning up the barrier instructions.

`-fdump-rtl-bbpart`
: Dump after partitioning hot and cold basic blocks.

`-fdump-rtl-bbro`
: Dump after block reordering.

`-fdump-rtl-btl1`
`-fdump-rtl-btl2`
: '`-fdump-rtl-btl1`' and '`-fdump-rtl-btl2`' enable dumping after the two branch target load optimization passes.

`-fdump-rtl-bypass`
: Dump after jump bypassing and control flow optimizations.

`-fdump-rtl-combine`
: Dump after the RTL instruction combination pass.

`-fdump-rtl-compgotos`
: Dump after duplicating the computed gotos.

`-fdump-rtl-ce1`
`-fdump-rtl-ce2`
`-fdump-rtl-ce3`
: '`-fdump-rtl-ce1`', '`-fdump-rtl-ce2`', and '`-fdump-rtl-ce3`' enable dumping after the three if conversion passes.

`-fdump-rtl-cprop_hardreg`
: Dump after hard register copy propagation.

`-fdump-rtl-csa`
: Dump after combining stack adjustments.

`-fdump-rtl-cse1`
`-fdump-rtl-cse2`
: '`-fdump-rtl-cse1`' and '`-fdump-rtl-cse2`' enable dumping after the two common subexpression elimination passes.

`-fdump-rtl-dce`
: Dump after the standalone dead code elimination passes.

`-fdump-rtl-dbr`
: Dump after delayed branch scheduling.

`-fdump-rtl-dce1`
`-fdump-rtl-dce2`
: '`-fdump-rtl-dce1`' and '`-fdump-rtl-dce2`' enable dumping after the two dead store elimination passes.

`-fdump-rtl-eh`
: Dump after finalization of EH handling code.

`-fdump-rtl-eh_ranges`
: Dump after conversion of EH handling range regions.

`-fdump-rtl-expand`
: Dump after RTL generation.

`-fdump-rtl-fwprop1`
`-fdump-rtl-fwprop2`
: '`-fdump-rtl-fwprop1`' and '`-fdump-rtl-fwprop2`' enable dumping after the two forward propagation passes.

`-fdump-rtl-gcse1`
`-fdump-rtl-gcse2`
: '`-fdump-rtl-gcse1`' and '`-fdump-rtl-gcse2`' enable dumping after global common subexpression elimination.

`-fdump-rtl-init-regs`
: Dump after the initialization of the registers.

`-fdump-rtl-initvals`
: Dump after the computation of the initial value sets.

`-fdump-rtl-into_cfglayout`
: Dump after converting to cfglayout mode.

`-fdump-rtl-ira`
: Dump after iterated register allocation.

`-fdump-rtl-jump`
: Dump after the second jump optimization.

`-fdump-rtl-loop2`
: '`-fdump-rtl-loop2`' enables dumping after the rtl loop optimization passes.

`-fdump-rtl-mach`
Dump after performing the machine dependent reorganization pass, if that pass exists.

`-fdump-rtl-mode_sw`
Dump after removing redundant mode switches.

`-fdump-rtl-rnreg`
Dump after register renumbering.

`-fdump-rtl-outof_cfglayout`
Dump after converting from cfglayout mode.

`-fdump-rtl-peephole2`
Dump after the peephole pass.

`-fdump-rtl-postreload`
Dump after post-reload optimizations.

`-fdump-rtl-pro_and_epilogue`
Dump after generating the function prologues and epilogues.

`-fdump-rtl-sched1`
`-fdump-rtl-sched2`
'`-fdump-rtl-sched1`' and '`-fdump-rtl-sched2`' enable dumping after the basic block scheduling passes.

`-fdump-rtl-ree`
Dump after sign/zero extension elimination.

`-fdump-rtl-seqabstr`
Dump after common sequence discovery.

`-fdump-rtl-shorten`
Dump after shortening branches.

`-fdump-rtl-sibling`
Dump after sibling call optimizations.

`-fdump-rtl-split1`
`-fdump-rtl-split2`
`-fdump-rtl-split3`
`-fdump-rtl-split4`
`-fdump-rtl-split5`
These options enable dumping after five rounds of instruction splitting.

`-fdump-rtl-sms`
Dump after modulo scheduling. This pass is only run on some architectures.

`-fdump-rtl-stack`
Dump after conversion from GCC's "flat register file" registers to the x87's stack-like registers. This pass is only run on x86 variants.

`-fdump-rtl-subreg1`
`-fdump-rtl-subreg2`
: '`-fdump-rtl-subreg1`' and '`-fdump-rtl-subreg2`' enable dumping after the two subreg expansion passes.

`-fdump-rtl-unshare`
: Dump after all rtl has been unshared.

`-fdump-rtl-vartrack`
: Dump after variable tracking.

`-fdump-rtl-vregs`
: Dump after converting virtual registers to hard registers.

`-fdump-rtl-web`
: Dump after live range splitting.

`-fdump-rtl-regclass`
`-fdump-rtl-subregs_of_mode_init`
`-fdump-rtl-subregs_of_mode_finish`
`-fdump-rtl-dfinit`
`-fdump-rtl-dfinish`
: These dumps are defined but always produce empty files.

`-da`
`-fdump-rtl-all`
: Produce all the dumps listed above.

`-dA`
: Annotate the assembler output with miscellaneous debugging information.

`-dD`
: Dump all macro definitions, at the end of preprocessing, in addition to normal output.

`-dH`
: Produce a core dump whenever an error occurs.

`-dp`
: Annotate the assembler output with a comment indicating which pattern and alternative is used. The length and cost of each instruction are also printed.

`-dP`
: Dump the RTL in the assembler output as a comment before each instruction. Also turns on '`-dp`' annotation.

`-dx`
: Just generate RTL for a function instead of compiling it. Usually used with '`-fdump-rtl-expand`'.

`-fdump-debug`
: Dump debugging information generated during the debug generation phase.

`-fdump-earlydebug`
: Dump debugging information generated during the early debug generation phase.

`-fdump-noaddr`
: When doing debugging dumps, suppress address output. This makes it more feasible to use diff on debugging dumps for compiler invocations with different

compiler binaries and/or different text / bss / data / heap / stack / dso start locations.

`-freport-bug`
Collect and dump debug information into a temporary file if an internal compiler error (ICE) occurs.

`-fdump-unnumbered`
When doing debugging dumps, suppress instruction numbers and address output. This makes it more feasible to use diff on debugging dumps for compiler invocations with different options, in particular with and without '`-g`'.

`-fdump-unnumbered-links`
When doing debugging dumps (see '`-d`' option above), suppress instruction numbers for the links to the previous and next instructions in a sequence.

`-fdump-ipa-`*`switch`*
`-fdump-ipa-`*`switch-options`*
Control the dumping at various stages of inter-procedural analysis language tree to a file. The file name is generated by appending a switch specific suffix to the source file name, and the file is created in the same directory as the output file. The following dumps are possible:

'`all`' Enables all inter-procedural analysis dumps.

'`cgraph`' Dumps information about call-graph optimization, unused function removal, and inlining decisions.

'`inline`' Dump after function inlining.

Additionally, the options '`-optimized`', '`-missed`', '`-note`', and '`-all`' can be provided, with the same meaning as for '`-fopt-info`', defaulting to '`-optimized`'.

For example, '`-fdump-ipa-inline-optimized-missed`' will emit information on callsites that were inlined, along with callsites that were not inlined.

By default, the dump will contain messages about successful optimizations (equivalent to '`-optimized`') together with low-level details about the analysis.

`-fdump-lang-all`
`-fdump-lang-`*`switch`*
`-fdump-lang-`*`switch-options`*
`-fdump-lang-`*`switch-options=filename`*
Control the dumping of language-specific information. The *options* and *filename* portions behave as described in the '`-fdump-tree`' option. The following *switch* values are accepted:

'`all`'
Enable all language-specific dumps.

'`class`' Dump class hierarchy information. Virtual table information is emitted unless "`slim`" is specified. This option is applicable to C++ only.

'raw' Dump the raw internal tree data. This option is applicable to C++ only.

-fdump-passes
Print on 'stderr' the list of optimization passes that are turned on and off by the current command-line options.

-fdump-statistics-*option*
Enable and control dumping of pass statistics in a separate file. The file name is generated by appending a suffix ending in '.statistics' to the source file name, and the file is created in the same directory as the output file. If the '-*option*' form is used, '-stats' causes counters to be summed over the whole compilation unit while '-details' dumps every event as the passes generate them. The default with no option is to sum counters for each function compiled.

-fdump-tree-all
-fdump-tree-*switch*
-fdump-tree-*switch*-*options*
-fdump-tree-*switch*-*options*=*filename*
Control the dumping at various stages of processing the intermediate language tree to a file. If the '-*options*' form is used, *options* is a list of '-' separated options which control the details of the dump. Not all options are applicable to all dumps; those that are not meaningful are ignored. The following options are available

'address' Print the address of each node. Usually this is not meaningful as it changes according to the environment and source file. Its primary use is for tying up a dump file with a debug environment.

'asmname' If DECL_ASSEMBLER_NAME has been set for a given decl, use that in the dump instead of DECL_NAME. Its primary use is ease of use working backward from mangled names in the assembly file.

'slim' When dumping front-end intermediate representations, inhibit dumping of members of a scope or body of a function merely because that scope has been reached. Only dump such items when they are directly reachable by some other path.

When dumping pretty-printed trees, this option inhibits dumping the bodies of control structures.

When dumping RTL, print the RTL in slim (condensed) form instead of the default LISP-like representation.

'raw' Print a raw representation of the tree. By default, trees are pretty-printed into a C-like representation.

'details' Enable more detailed dumps (not honored by every dump option). Also include information from the optimization passes.

'stats' Enable dumping various statistics about the pass (not honored by every dump option).

'blocks' Enable showing basic block boundaries (disabled in raw dumps).

'graph' For each of the other indicated dump files ('-fdump-rtl-*pass*'), dump a representation of the control flow graph suitable for viewing with GraphViz to '*file.passid.pass*.dot'. Each function in the file is pretty-printed as a subgraph, so that GraphViz can render them all in a single plot.

This option currently only works for RTL dumps, and the RTL is always dumped in slim form.

'vops' Enable showing virtual operands for every statement.

'lineno' Enable showing line numbers for statements.

'uid' Enable showing the unique ID (DECL_UID) for each variable.

'verbose' Enable showing the tree dump for each statement.

'eh' Enable showing the EH region number holding each statement.

'scev' Enable showing scalar evolution analysis details.

'optimized'
Enable showing optimization information (only available in certain passes).

'missed' Enable showing missed optimization information (only available in certain passes).

'note' Enable other detailed optimization information (only available in certain passes).

'all' Turn on all options, except 'raw', 'slim', 'verbose' and 'lineno'.

'optall' Turn on all optimization options, i.e., 'optimized', 'missed', and 'note'.

To determine what tree dumps are available or find the dump for a pass of interest follow the steps below.

1. Invoke GCC with '-fdump-passes' and in the 'stderr' output look for a code that corresponds to the pass you are interested in. For example, the codes tree-evrp, tree-vrp1, and tree-vrp2 correspond to the three Value Range Propagation passes. The number at the end distinguishes distinct invocations of the same pass.
2. To enable the creation of the dump file, append the pass code to the '-fdump-' option prefix and invoke GCC with it. For example, to enable the dump from the Early Value Range Propagation pass, invoke GCC with the '-fdump-tree-evrp' option. Optionally, you may specify the name of the dump file. If you don't specify one, GCC creates as described below.
3. Find the pass dump in a file whose name is composed of three components separated by a period: the name of the source file GCC was invoked to compile, a numeric suffix indicating the pass number followed by the letter 't' for tree passes (and the letter 'r' for RTL passes), and finally the pass code. For example, the Early VRP pass dump might be in a file named 'myfile.c.038t.evrp' in the current working directory. Note that the

numeric codes are not stable and may change from one version of GCC to another.

`-fopt-info`
`-fopt-info-`*`options`*
`-fopt-info-`*`options=filename`*

Controls optimization dumps from various optimization passes. If the '`-`*`options`*' form is used, *options* is a list of '`-`' separated option keywords to select the dump details and optimizations.

The *options* can be divided into three groups:

1. options describing what kinds of messages should be emitted,
2. options describing the verbosity of the dump, and
3. options describing which optimizations should be included.

The options from each group can be freely mixed as they are non-overlapping. However, in case of any conflicts, the later options override the earlier options on the command line.

The following options control which kinds of messages should be emitted:

'`optimized`'
: Print information when an optimization is successfully applied. It is up to a pass to decide which information is relevant. For example, the vectorizer passes print the source location of loops which are successfully vectorized.

'`missed`'
: Print information about missed optimizations. Individual passes control which information to include in the output.

'`note`'
: Print verbose information about optimizations, such as certain transformations, more detailed messages about decisions etc.

'`all`'
: Print detailed optimization information. This includes '`optimized`', '`missed`', and '`note`'.

The following option controls the dump verbosity:

'`internals`'
: By default, only "high-level" messages are emitted. This option enables additional, more detailed, messages, which are likely to only be of interest to GCC developers.

One or more of the following option keywords can be used to describe a group of optimizations:

'`ipa`'
: Enable dumps from all interprocedural optimizations.

'`loop`'
: Enable dumps from all loop optimizations.

'`inline`'
: Enable dumps from all inlining optimizations.

'`omp`'
: Enable dumps from all OMP (Offloading and Multi Processing) optimizations.

'`vec`'
: Enable dumps from all vectorization optimizations.

'optall' Enable dumps from all optimizations. This is a superset of the optimization groups listed above.

If *options* is omitted, it defaults to 'optimized-optall', which means to dump messages about successful optimizations from all the passes, omitting messages that are treated as "internals".

If the *filename* is provided, then the dumps from all the applicable optimizations are concatenated into the *filename*. Otherwise the dump is output onto 'stderr'. Though multiple '-fopt-info' options are accepted, only one of them can include a *filename*. If other filenames are provided then all but the first such option are ignored.

Note that the output *filename* is overwritten in case of multiple translation units. If a combined output from multiple translation units is desired, 'stderr' should be used instead.

In the following example, the optimization info is output to 'stderr':

```
gcc -O3 -fopt-info
```

This example:

```
gcc -O3 -fopt-info-missed=missed.all
```

outputs missed optimization report from all the passes into 'missed.all', and this one:

```
gcc -O2 -ftree-vectorize -fopt-info-vec-missed
```

prints information about missed optimization opportunities from vectorization passes on 'stderr'. Note that '-fopt-info-vec-missed' is equivalent to '-fopt-info-missed-vec'. The order of the optimization group names and message types listed after '-fopt-info' does not matter.

As another example,

```
gcc -O3 -fopt-info-inline-optimized-missed=inline.txt
```

outputs information about missed optimizations as well as optimized locations from all the inlining passes into 'inline.txt'.

Finally, consider:

```
gcc -fopt-info-vec-missed=vec.miss -fopt-info-loop-optimized=loop.opt
```

Here the two output filenames 'vec.miss' and 'loop.opt' are in conflict since only one output file is allowed. In this case, only the first option takes effect and the subsequent options are ignored. Thus only 'vec.miss' is produced which contains dumps from the vectorizer about missed opportunities.

-fsave-optimization-record

Write a SRCFILE.opt-record.json.gz file detailing what optimizations were performed, for those optimizations that support '-fopt-info'.

This option is experimental and the format of the data within the compressed JSON file is subject to change.

It is roughly equivalent to a machine-readable version of '-fopt-info-all', as a collection of messages with source file, line number and column number, with the following additional data for each message:

- the execution count of the code being optimized, along with metadata about whether this was from actual profile data, or just an estimate, allowing consumers to prioritize messages by code hotness,
- the function name of the code being optimized, where applicable,
- the "inlining chain" for the code being optimized, so that when a function is inlined into several different places (which might themselves be inlined), the reader can distinguish between the copies,
- objects identifying those parts of the message that refer to expressions, statements or symbol-table nodes, which of these categories they are, and, when available, their source code location,
- the GCC pass that emitted the message, and
- the location in GCC's own code from which the message was emitted

Additionally, some messages are logically nested within other messages, reflecting implementation details of the optimization passes.

`-fsched-verbose=n`

On targets that use instruction scheduling, this option controls the amount of debugging output the scheduler prints to the dump files.

For *n* greater than zero, '`-fsched-verbose`' outputs the same information as '`-fdump-rtl-sched1`' and '`-fdump-rtl-sched2`'. For *n* greater than one, it also output basic block probabilities, detailed ready list information and unit/insn info. For *n* greater than two, it includes RTL at abort point, control-flow and regions info. And for *n* over four, '`-fsched-verbose`' also includes dependence info.

`-fenable-`*`kind-pass`*
`-fdisable-`*`kind-pass=range-list`*

This is a set of options that are used to explicitly disable/enable optimization passes. These options are intended for use for debugging GCC. Compiler users should use regular options for enabling/disabling passes instead.

`-fdisable-ipa-`*`pass`*

Disable IPA pass *pass*. *pass* is the pass name. If the same pass is statically invoked in the compiler multiple times, the pass name should be appended with a sequential number starting from 1.

`-fdisable-rtl-`*`pass`*
`-fdisable-rtl-`*`pass=range-list`*

Disable RTL pass *pass*. *pass* is the pass name. If the same pass is statically invoked in the compiler multiple times, the pass name should be appended with a sequential number starting from 1. *range-list* is a comma-separated list of function ranges or assembler names. Each range is a number pair separated by a colon. The range is inclusive in both ends. If the range is trivial, the number pair can be simplified as a single number. If the function's call graph node's *uid* falls within one of the specified ranges, the *pass* is disabled for that function. The *uid* is shown in the function

header of a dump file, and the pass names can be dumped by using option '-fdump-passes'.

-fdisable-tree-*pass*
-fdisable-tree-*pass*=*range-list*
Disable tree pass *pass*. See '-fdisable-rtl' for the description of option arguments.

-fenable-ipa-*pass*
Enable IPA pass *pass*. *pass* is the pass name. If the same pass is statically invoked in the compiler multiple times, the pass name should be appended with a sequential number starting from 1.

-fenable-rtl-*pass*
-fenable-rtl-*pass*=*range-list*
Enable RTL pass *pass*. See '-fdisable-rtl' for option argument description and examples.

-fenable-tree-*pass*
-fenable-tree-*pass*=*range-list*
Enable tree pass *pass*. See '-fdisable-rtl' for the description of option arguments.

Here are some examples showing uses of these options.

```
# disable ccp1 for all functions
   -fdisable-tree-ccp1
# disable complete unroll for function whose cgraph node uid is 1
   -fenable-tree-cunroll=1
# disable gcse2 for functions at the following ranges [1,1],
# [300,400], and [400,1000]
# disable gcse2 for functions foo and foo2
   -fdisable-rtl-gcse2=foo,foo2
# disable early inlining
   -fdisable-tree-einline
# disable ipa inlining
   -fdisable-ipa-inline
# enable tree full unroll
   -fenable-tree-unroll
```

-fchecking
-fchecking=*n*
Enable internal consistency checking. The default depends on the compiler configuration. '-fchecking=2' enables further internal consistency checking that might affect code generation.

-frandom-seed=*string*
This option provides a seed that GCC uses in place of random numbers in generating certain symbol names that have to be different in every compiled file. It is also used to place unique stamps in coverage data files and the object files that produce them. You can use the '-frandom-seed' option to produce reproducibly identical object files.

The *string* can either be a number (decimal, octal or hex) or an arbitrary string (in which case it's converted to a number by computing CRC32).

The *string* should be different for every file you compile.

`-save-temps`
`-save-temps=cwd`

Store the usual "temporary" intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling '`foo.c`' with '`-c -save-temps`' produces files '`foo.i`' and '`foo.s`', as well as '`foo.o`'. This creates a preprocessed '`foo.i`' output file even though the compiler now normally uses an integrated preprocessor.

When used in combination with the '`-x`' command-line option, '`-save-temps`' is sensible enough to avoid over writing an input source file with the same extension as an intermediate file. The corresponding intermediate file may be obtained by renaming the source file before using '`-save-temps`'.

If you invoke GCC in parallel, compiling several different source files that share a common base name in different subdirectories or the same source file compiled for multiple output destinations, it is likely that the different parallel compilers will interfere with each other, and overwrite the temporary files. For instance:

```
gcc -save-temps -o outdir1/foo.o indir1/foo.c&
gcc -save-temps -o outdir2/foo.o indir2/foo.c&
```

may result in '`foo.i`' and '`foo.o`' being written to simultaneously by both compilers.

`-save-temps=obj`

Store the usual "temporary" intermediate files permanently. If the '`-o`' option is used, the temporary files are based on the object file. If the '`-o`' option is not used, the '`-save-temps=obj`' switch behaves like '`-save-temps`'.

For example:

```
gcc -save-temps=obj -c foo.c
gcc -save-temps=obj -c bar.c -o dir/xbar.o
gcc -save-temps=obj foobar.c -o dir2/yfoobar
```

creates '`foo.i`', '`foo.s`', '`dir/xbar.i`', '`dir/xbar.s`', '`dir2/yfoobar.i`', '`dir2/yfoobar.s`', and '`dir2/yfoobar.o`'.

`-time`[`=`*file*]

Report the CPU time taken by each subprocess in the compilation sequence. For C source files, this is the compiler proper and assembler (plus the linker if linking is done).

Without the specification of an output file, the output looks like this:

```
# cc1 0.12 0.01
# as 0.00 0.01
```

The first number on each line is the "user time", that is time spent executing the program itself. The second number is "system time", time spent executing operating system routines on behalf of the program. Both numbers are in seconds.

With the specification of an output file, the output is appended to the named file, and it looks like this:

```
0.12 0.01 cc1 options
0.00 0.01 as options
```

The "user time" and the "system time" are moved before the program name, and the options passed to the program are displayed, so that one can later tell what file was being compiled, and with which options.

`-fdump-final-insns[=file]`
: Dump the final internal representation (RTL) to *file*. If the optional argument is omitted (or if *file* is `.`), the name of the dump file is determined by appending `.gkd` to the compilation output file name.

`-fcompare-debug[=opts]`
: If no error occurs during compilation, run the compiler a second time, adding *opts* and '`-fcompare-debug-second`' to the arguments passed to the second compilation. Dump the final internal representation in both compilations, and print an error if they differ.

If the equal sign is omitted, the default '`-gtoggle`' is used.

The environment variable `GCC_COMPARE_DEBUG`, if defined, non-empty and nonzero, implicitly enables '`-fcompare-debug`'. If `GCC_COMPARE_DEBUG` is defined to a string starting with a dash, then it is used for *opts*, otherwise the default '`-gtoggle`' is used.

'`-fcompare-debug=`', with the equal sign but without *opts*, is equivalent to '`-fno-compare-debug`', which disables the dumping of the final representation and the second compilation, preventing even `GCC_COMPARE_DEBUG` from taking effect.

To verify full coverage during '`-fcompare-debug`' testing, set `GCC_COMPARE_DEBUG` to say '`-fcompare-debug-not-overridden`', which GCC rejects as an invalid option in any actual compilation (rather than preprocessing, assembly or linking). To get just a warning, setting `GCC_COMPARE_DEBUG` to '`-w%n-fcompare-debug not overridden`' will do.

`-fcompare-debug-second`
: This option is implicitly passed to the compiler for the second compilation requested by '`-fcompare-debug`', along with options to silence warnings, and omitting other options that would cause the compiler to produce output to files or to standard output as a side effect. Dump files and preserved temporary files are renamed so as to contain the `.gk` additional extension during the second compilation, to avoid overwriting those generated by the first.

When this option is passed to the compiler driver, it causes the *first* compilation to be skipped, which makes it useful for little other than debugging the compiler proper.

`-gtoggle`
: Turn off generation of debug info, if leaving out this option generates it, or turn it on at level 2 otherwise. The position of this argument in the command line does not matter; it takes effect after all other options are processed, and it does so only once, no matter how many times it is given. This is mainly intended to be used with '`-fcompare-debug`'.

`-fvar-tracking-assignments-toggle`
: Toggle '`-fvar-tracking-assignments`', in the same way that '`-gtoggle`' toggles '`-g`'.

`-Q`
: Makes the compiler print out each function name as it is compiled, and print some statistics about each pass when it finishes.

`-ftime-report`
: Makes the compiler print some statistics about the time consumed by each pass when it finishes.

`-ftime-report-details`
: Record the time consumed by infrastructure parts separately for each pass.

`-fira-verbose=`*n*
: Control the verbosity of the dump file for the integrated register allocator. The default value is 5. If the value *n* is greater or equal to 10, the dump output is sent to stderr using the same format as *n* minus 10.

`-flto-report`
: Prints a report with internal details on the workings of the link-time optimizer. The contents of this report vary from version to version. It is meant to be useful to GCC developers when processing object files in LTO mode (via '`-flto`').

  Disabled by default.

`-flto-report-wpa`
: Like '`-flto-report`', but only print for the WPA phase of Link Time Optimization.

`-fmem-report`
: Makes the compiler print some statistics about permanent memory allocation when it finishes.

`-fmem-report-wpa`
: Makes the compiler print some statistics about permanent memory allocation for the WPA phase only.

`-fpre-ipa-mem-report`
`-fpost-ipa-mem-report`
: Makes the compiler print some statistics about permanent memory allocation before or after interprocedural optimization.

`-fprofile-report`
: Makes the compiler print some statistics about consistency of the (estimated) profile and effect of individual passes.

`-fstack-usage`
: Makes the compiler output stack usage information for the program, on a per-function basis. The filename for the dump is made by appending '`.su`' to the *auxname*. *auxname* is generated from the name of the output file, if explicitly specified and it is not an executable, otherwise it is the basename of the source file. An entry is made up of three fields:

  - The name of the function.

- A number of bytes.
- One or more qualifiers: `static`, `dynamic`, `bounded`.

The qualifier `static` means that the function manipulates the stack statically: a fixed number of bytes are allocated for the frame on function entry and released on function exit; no stack adjustments are otherwise made in the function. The second field is this fixed number of bytes.

The qualifier `dynamic` means that the function manipulates the stack dynamically: in addition to the static allocation described above, stack adjustments are made in the body of the function, for example to push/pop arguments around function calls. If the qualifier `bounded` is also present, the amount of these adjustments is bounded at compile time and the second field is an upper bound of the total amount of stack used by the function. If it is not present, the amount of these adjustments is not bounded at compile time and the second field only represents the bounded part.

`-fstats`
: Emit statistics about front-end processing at the end of the compilation. This option is supported only by the C++ front end, and the information is generally only useful to the G++ development team.

`-fdbg-cnt-list`
: Print the name and the counter upper bound for all debug counters.

`-fdbg-cnt=`*`counter-value-list`*
: Set the internal debug counter lower and upper bound. *counter-value-list* is a comma-separated list of *name*:*lower_bound*:*upper_bound* tuples which sets the lower and the upper bound of each debug counter *name*. The *lower_bound* is optional and is zero initialized if not set. All debug counters have the initial upper bound of `UINT_MAX`; thus `dbg_cnt` returns true always unless the upper bound is set by this option. For example, with '`-fdbg-cnt=dce:2:4,tail_call:10`', `dbg_cnt(dce)` returns true only for third and fourth invocation. For `dbg_cnt(tail_call)` true is returned for first 10 invocations.

`-print-file-name=`*`library`*
: Print the full absolute name of the library file *library* that would be used when linking—and don't do anything else. With this option, GCC does not compile or link anything; it just prints the file name.

`-print-multi-directory`
: Print the directory name corresponding to the multilib selected by any other switches present in the command line. This directory is supposed to exist in `GCC_EXEC_PREFIX`.

`-print-multi-lib`
: Print the mapping from multilib directory names to compiler switches that enable them. The directory name is separated from the switches by '`;`', and each switch starts with an '`@`' instead of the '`-`', without spaces between multiple switches. This is supposed to ease shell processing.

`-print-multi-os-directory`
: Print the path to OS libraries for the selected multilib, relative to some '`lib`' subdirectory. If OS libraries are present in the '`lib`' subdirectory and no mul-

tilibs are used, this is usually just '.', if OS libraries are present in 'lib*suffix*' sibling directories this prints e.g. '../lib64', '../lib' or '../lib32', or if OS libraries are present in 'lib/*subdir*' subdirectories it prints e.g. 'amd64', 'sparcv9' or 'ev6'.

-print-multiarch
: Print the path to OS libraries for the selected multiarch, relative to some 'lib' subdirectory.

-print-prog-name=*program*
: Like '-print-file-name', but searches for a program such as cpp.

-print-libgcc-file-name
: Same as '-print-file-name=libgcc.a'.

This is useful when you use '-nostdlib' or '-nodefaultlibs' but you do want to link with 'libgcc.a'. You can do:

```
gcc -nostdlib files... `gcc -print-libgcc-file-name`
```

-print-search-dirs
: Print the name of the configured installation directory and a list of program and library directories gcc searches—and don't do anything else.

This is useful when gcc prints the error message 'installation problem, cannot exec cpp0: No such file or directory'. To resolve this you either need to put 'cpp0' and the other compiler components where gcc expects to find them, or you can set the environment variable GCC_EXEC_PREFIX to the directory where you installed them. Don't forget the trailing '/'. See Section 3.20 [Environment Variables], page 436.

-print-sysroot
: Print the target sysroot directory that is used during compilation. This is the target sysroot specified either at configure time or using the '--sysroot' option, possibly with an extra suffix that depends on compilation options. If no target sysroot is specified, the option prints nothing.

-print-sysroot-headers-suffix
: Print the suffix added to the target sysroot when searching for headers, or give an error if the compiler is not configured with such a suffix—and don't do anything else.

-dumpmachine
: Print the compiler's target machine (for example, 'i686-pc-linux-gnu')—and don't do anything else.

-dumpversion
: Print the compiler version (for example, 3.0, 6.3.0 or 7)—and don't do anything else. This is the compiler version used in filesystem paths and specs. Depending on how the compiler has been configured it can be just a single number (major version), two numbers separated by a dot (major and minor version) or three numbers separated by dots (major, minor and patchlevel version).

`-dumpfullversion`
: Print the full compiler version—and don't do anything else. The output is always three numbers separated by dots, major, minor and patchlevel version.

`-dumpspecs`
: Print the compiler's built-in specs—and don't do anything else. (This is used when GCC itself is being built.) See Section 3.19 [Spec Files], page 428.

3.18 Machine-Dependent Options

Each target machine supported by GCC can have its own options—for example, to allow you to compile for a particular processor variant or ABI, or to control optimizations specific to that machine. By convention, the names of machine-specific options start with '`-m`'.

Some configurations of the compiler also support additional target-specific options, usually for compatibility with other compilers on the same platform.

3.18.1 AArch64 Options

These options are defined for AArch64 implementations:

`-mabi=`*`name`*
: Generate code for the specified data model. Permissible values are '`ilp32`' for SysV-like data model where int, long int and pointers are 32 bits, and '`lp64`' for SysV-like data model where int is 32 bits, but long int and pointers are 64 bits.

  The default depends on the specific target configuration. Note that the LP64 and ILP32 ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.

`-mbig-endian`
: Generate big-endian code. This is the default when GCC is configured for an '`aarch64_be-*-*`' target.

`-mgeneral-regs-only`
: Generate code which uses only the general-purpose registers. This will prevent the compiler from using floating-point and Advanced SIMD registers but will not impose any restrictions on the assembler.

`-mlittle-endian`
: Generate little-endian code. This is the default when GCC is configured for an '`aarch64-*-*`' but not an '`aarch64_be-*-*`' target.

`-mcmodel=tiny`
: Generate code for the tiny code model. The program and its statically defined symbols must be within 1MB of each other. Programs can be statically or dynamically linked.

`-mcmodel=small`
: Generate code for the small code model. The program and its statically defined symbols must be within 4GB of each other. Programs can be statically or dynamically linked. This is the default code model.

`-mcmodel=large`
: Generate code for the large code model. This makes no assumptions about addresses and sizes of sections. Programs can be statically linked only.

`-mstrict-align`
`-mno-strict-align`
: Avoid or allow generating memory accesses that may not be aligned on a natural object boundary as described in the architecture specification.

`-momit-leaf-frame-pointer`
`-mno-omit-leaf-frame-pointer`
: Omit or keep the frame pointer in leaf functions. The former behavior is the default.

`-mstack-protector-guard=`*`guard`*
`-mstack-protector-guard-reg=`*`reg`*
`-mstack-protector-guard-offset=`*`offset`*
: Generate stack protection code using canary at *guard*. Supported locations are '`global`' for a global canary or '`sysreg`' for a canary in an appropriate system register.

  With the latter choice the options '`-mstack-protector-guard-reg=`*`reg`*' and '`-mstack-protector-guard-offset=`*`offset`*' furthermore specify which system register to use as base register for reading the canary, and from what offset from that base register. There is no default register or offset as this is entirely for use within the Linux kernel.

`-mstack-protector-guard=`*`guard`*
`-mstack-protector-guard-reg=`*`reg`*
`-mstack-protector-guard-offset=`*`offset`*
: Generate stack protection code using canary at *guard*. Supported locations are '`global`' for a global canary or '`sysreg`' for a canary in an appropriate system register.

  With the latter choice the options '`-mstack-protector-guard-reg=`*`reg`*' and '`-mstack-protector-guard-offset=`*`offset`*' furthermore specify which system register to use as base register for reading the canary, and from what offset from that base register. There is no default register or offset as this is entirely for use within the Linux kernel.

`-mtls-dialect=desc`
: Use TLS descriptors as the thread-local storage mechanism for dynamic accesses of TLS variables. This is the default.

`-mtls-dialect=traditional`
: Use traditional TLS as the thread-local storage mechanism for dynamic accesses of TLS variables.

`-mtls-size=`*`size`*
: Specify bit size of immediate TLS offsets. Valid values are 12, 24, 32, 48. This option requires binutils 2.26 or newer.

`-mfix-cortex-a53-835769`
`-mno-fix-cortex-a53-835769`
Enable or disable the workaround for the ARM Cortex-A53 erratum number 835769. This involves inserting a NOP instruction between memory instructions and 64-bit integer multiply-accumulate instructions.

`-mfix-cortex-a53-843419`
`-mno-fix-cortex-a53-843419`
Enable or disable the workaround for the ARM Cortex-A53 erratum number 843419. This erratum workaround is made at link time and this will only pass the corresponding flag to the linker.

`-mlow-precision-recip-sqrt`
`-mno-low-precision-recip-sqrt`
Enable or disable the reciprocal square root approximation. This option only has an effect if '`-ffast-math`' or '`-funsafe-math-optimizations`' is used as well. Enabling this reduces precision of reciprocal square root results to about 16 bits for single precision and to 32 bits for double precision.

`-mlow-precision-sqrt`
`-mno-low-precision-sqrt`
Enable or disable the square root approximation. This option only has an effect if '`-ffast-math`' or '`-funsafe-math-optimizations`' is used as well. Enabling this reduces precision of square root results to about 16 bits for single precision and to 32 bits for double precision. If enabled, it implies '`-mlow-precision-recip-sqrt`'.

`-mlow-precision-div`
`-mno-low-precision-div`
Enable or disable the division approximation. This option only has an effect if '`-ffast-math`' or '`-funsafe-math-optimizations`' is used as well. Enabling this reduces precision of division results to about 16 bits for single precision and to 32 bits for double precision.

`-mtrack-speculation`
`-mno-track-speculation`
Enable or disable generation of additional code to track speculative execution through conditional branches. The tracking state can then be used by the compiler when expanding calls to `__builtin_speculation_safe_copy` to permit a more efficient code sequence to be generated.

`-march=`*`name`*
Specify the name of the target architecture and, optionally, one or more feature modifiers. This option has the form '`-march=`*`arch`*{`+`[`no`]*`feature`*}*'.

The permissible values for *arch* are '`armv8-a`', '`armv8.1-a`', '`armv8.2-a`', '`armv8.3-a`', '`armv8.4-a`', '`armv8.5-a`' or *native*.

The value '`armv8.5-a`' implies '`armv8.4-a`' and enables compiler support for the ARMv8.5-A architecture extensions.

The value '`armv8.4-a`' implies '`armv8.3-a`' and enables compiler support for the ARMv8.4-A architecture extensions.

The value '`armv8.3-a`' implies '`armv8.2-a`' and enables compiler support for the ARMv8.3-A architecture extensions.

The value '`armv8.2-a`' implies '`armv8.1-a`' and enables compiler support for the ARMv8.2-A architecture extensions.

The value '`armv8.1-a`' implies '`armv8-a`' and enables compiler support for the ARMv8.1-A architecture extension. In particular, it enables the '`+crc`', '`+lse`', and '`+rdma`' features.

The value '`native`' is available on native AArch64 GNU/Linux and causes the compiler to pick the architecture of the host system. This option has no effect if the compiler is unable to recognize the architecture of the host system,

The permissible values for *feature* are listed in the sub-section on ['`-march`' and '`-mcpu`' Feature Modifiers], page 250. Where conflicting feature modifiers are specified, the right-most feature is used.

GCC uses *name* to determine what kind of instructions it can emit when generating assembly code. If '`-march`' is specified without either of '`-mtune`' or '`-mcpu`' also being specified, the code is tuned to perform well across a range of target processors implementing the target architecture.

`-mtune=`*`name`*

Specify the name of the target processor for which GCC should tune the performance of the code. Permissible values for this option are: '`generic`', '`cortex-a35`', '`cortex-a53`', '`cortex-a55`', '`cortex-a57`', '`cortex-a72`', '`cortex-a73`', '`cortex-a75`', '`cortex-a76`', '`ares`', '`exynos-m1`', '`emag`', '`falkor`', '`neoverse-e1`','`neoverse-n1`','`qdf24xx`', '`saphira`', '`phecda`', '`xgene1`', '`vulcan`', '`octeontx`', '`octeontx81`', '`octeontx83`', '`thunderx`', '`thunderxt88`', '`thunderxt88p1`', '`thunderxt81`', '`tsv110`', '`thunderxt83`', '`thunderx2t99`', '`cortex-a57.cortex-a53`', '`cortex-a72.cortex-a53`', '`cortex-a73.cortex-a35`', '`cortex-a73.cortex-a53`', '`cortex-a75.cortex-a55`', '`cortex-a76.cortex-a55`' '`native`'.

The values '`cortex-a57.cortex-a53`', '`cortex-a72.cortex-a53`', '`cortex-a73.cortex-a35`', '`cortex-a73.cortex-a53`', '`cortex-a75.cortex-a55`', '`cortex-a76.cortex-a55`' specify that GCC should tune for a big.LITTLE system.

Additionally on native AArch64 GNU/Linux systems the value '`native`' tunes performance to the host system. This option has no effect if the compiler is unable to recognize the processor of the host system.

Where none of '`-mtune=`', '`-mcpu=`' or '`-march=`' are specified, the code is tuned to perform well across a range of target processors.

This option cannot be suffixed by feature modifiers.

`-mcpu=`*`name`*

Specify the name of the target processor, optionally suffixed by one or more feature modifiers. This option has the form '`-mcpu=`*`cpu`*{`+`[`no`]*`feature`*}*', where the permissible values for *cpu* are the same as those available for '`-mtune`'. The permissible values for *feature* are documented in the sub-section on ['`-march`' and '`-mcpu`' Feature Modifiers], page 250. Where conflicting feature modifiers are specified, the right-most feature is used.

GCC uses *name* to determine what kind of instructions it can emit when generating assembly code (as if by '`-march`') and to determine the target processor for which to tune for performance (as if by '`-mtune`'). Where this option is used in conjunction with '`-march`' or '`-mtune`', those options take precedence over the appropriate part of this option.

`-moverride=`*`string`*

Override tuning decisions made by the back-end in response to a '`-mtune=`' switch. The syntax, semantics, and accepted values for *string* in this option are not guaranteed to be consistent across releases.

This option is only intended to be useful when developing GCC.

`-mverbose-cost-dump`

Enable verbose cost model dumping in the debug dump files. This option is provided for use in debugging the compiler.

`-mpc-relative-literal-loads`
`-mno-pc-relative-literal-loads`

Enable or disable PC-relative literal loads. With this option literal pools are accessed using a single instruction and emitted after each function. This limits the maximum size of functions to 1MB. This is enabled by default for '`-mcmodel=tiny`'.

`-msign-return-address=`*`scope`*

Select the function scope on which return address signing will be applied. Permissible values are '`none`', which disables return address signing, '`non-leaf`', which enables pointer signing for functions which are not leaf functions, and '`all`', which enables pointer signing for all functions. The default value is '`none`'. This option has been deprecated by -mbranch-protection.

`-mbranch-protection=`*`none`*`|`*`standard`*`|`*`pac-ret`*`[+`*`leaf`*`]|`*`bti`*

Select the branch protection features to use. '`none`' is the default and turns off all types of branch protection. '`standard`' turns on all types of branch protection features. If a feature has additional tuning options, then '`standard`' sets it to its standard level. '`pac-ret[+`*`leaf`*`]`' turns on return address signing to its standard level: signing functions that save the return address to memory (non-leaf functions will practically always do this) using the a-key. The optional argument '`leaf`' can be used to extend the signing to include leaf functions. '`bti`' turns on branch target identification mechanism.

`-msve-vector-bits=`*`bits`*

Specify the number of bits in an SVE vector register. This option only has an effect when SVE is enabled.

GCC supports two forms of SVE code generation: "vector-length agnostic" output that works with any size of vector register and "vector-length specific" output that allows GCC to make assumptions about the vector length when it is useful for optimization reasons. The possible values of '`bits`' are: '`scalable`', '`128`', '`256`', '`512`', '`1024`' and '`2048`'. Specifying '`scalable`' selects vector-length agnostic output. At present '`-msve-vector-bits=128`' also generates vector-length agnostic output. All other values generate vector-length specific

code. The behavior of these values may change in future releases and no value except '`scalable`' should be relied on for producing code that is portable across different hardware SVE vector lengths.

The default is '`-msve-vector-bits=scalable`', which produces vector-length agnostic code.

3.18.1.1 '-march' and '-mcpu' Feature Modifiers

Feature modifiers used with '`-march`' and '`-mcpu`' can be any of the following and their inverses '`no`*feature*':

'`crc`' Enable CRC extension. This is on by default for '`-march=armv8.1-a`'.

'`crypto`' Enable Crypto extension. This also enables Advanced SIMD and floating-point instructions.

'`fp`' Enable floating-point instructions. This is on by default for all possible values for options '`-march`' and '`-mcpu`'.

'`simd`' Enable Advanced SIMD instructions. This also enables floating-point instructions. This is on by default for all possible values for options '`-march`' and '`-mcpu`'.

'`sve`' Enable Scalable Vector Extension instructions. This also enables Advanced SIMD and floating-point instructions.

'`lse`' Enable Large System Extension instructions. This is on by default for '`-march=armv8.1-a`'.

'`rdma`' Enable Round Double Multiply Accumulate instructions. This is on by default for '`-march=armv8.1-a`'.

'`fp16`' Enable FP16 extension. This also enables floating-point instructions.

'`fp16fml`' Enable FP16 fmla extension. This also enables FP16 extensions and floating-point instructions. This option is enabled by default for '`-march=armv8.4-a`'. Use of this option with architectures prior to Armv8.2-A is not supported.

'`rcpc`' Enable the RcPc extension. This does not change code generation from GCC, but is passed on to the assembler, enabling inline asm statements to use instructions from the RcPc extension.

'`dotprod`' Enable the Dot Product extension. This also enables Advanced SIMD instructions.

'`aes`' Enable the Armv8-a aes and pmull crypto extension. This also enables Advanced SIMD instructions.

'`sha2`' Enable the Armv8-a sha2 crypto extension. This also enables Advanced SIMD instructions.

'`sha3`' Enable the sha512 and sha3 crypto extension. This also enables Advanced SIMD instructions. Use of this option with architectures prior to Armv8.2-A is not supported.

'sm4' Enable the sm3 and sm4 crypto extension. This also enables Advanced SIMD instructions. Use of this option with architectures prior to Armv8.2-A is not supported.

'profile' Enable the Statistical Profiling extension. This option is only to enable the extension at the assembler level and does not affect code generation.

'rng' Enable the Armv8.5-a Random Number instructions. This option is only to enable the extension at the assembler level and does not affect code generation.

'memtag' Enable the Armv8.5-a Memory Tagging Extensions. This option is only to enable the extension at the assembler level and does not affect code generation.

'sb' Enable the Armv8-a Speculation Barrier instruction. This option is only to enable the extension at the assembler level and does not affect code generation. This option is enabled by default for '-march=armv8.5-a'.

'ssbs' Enable the Armv8-a Speculative Store Bypass Safe instruction. This option is only to enable the extension at the assembler level and does not affect code generation. This option is enabled by default for '-march=armv8.5-a'.

'predres' Enable the Armv8-a Execution and Data Prediction Restriction instructions. This option is only to enable the extension at the assembler level and does not affect code generation. This option is enabled by default for '-march=armv8.5-a'.

Feature 'crypto' implies 'aes', 'sha2', and 'simd', which implies 'fp'. Conversely, 'nofp' implies 'nosimd', which implies 'nocrypto', 'noaes' and 'nosha2'.

3.18.2 Adapteva Epiphany Options

These '-m' options are defined for Adapteva Epiphany:

-mhalf-reg-file
Don't allocate any register in the range r32...r63. That allows code to run on hardware variants that lack these registers.

-mprefer-short-insn-regs
Preferentially allocate registers that allow short instruction generation. This can result in increased instruction count, so this may either reduce or increase overall code size.

-mbranch-cost=*num*
Set the cost of branches to roughly *num* "simple" instructions. This cost is only a heuristic and is not guaranteed to produce consistent results across releases.

-mcmove Enable the generation of conditional moves.

-mnops=*num*
Emit *num* NOPs before every other generated instruction.

-mno-soft-cmpsf
For single-precision floating-point comparisons, emit an fsub instruction and test the flags. This is faster than a software comparison, but can get incorrect results in the presence of NaNs, or when two different small numbers are compared such that their difference is calculated as zero. The default is '-msoft-cmpsf', which uses slower, but IEEE-compliant, software comparisons.

`-mstack-offset=num`

Set the offset between the top of the stack and the stack pointer. E.g., a value of 8 means that the eight bytes in the range `sp+0...sp+7` can be used by leaf functions without stack allocation. Values other than '`8`' or '`16`' are untested and unlikely to work. Note also that this option changes the ABI; compiling a program with a different stack offset than the libraries have been compiled with generally does not work. This option can be useful if you want to evaluate if a different stack offset would give you better code, but to actually use a different stack offset to build working programs, it is recommended to configure the toolchain with the appropriate '`--with-stack-offset=num`' option.

`-mno-round-nearest`

Make the scheduler assume that the rounding mode has been set to truncating. The default is '`-mround-nearest`'.

`-mlong-calls`

If not otherwise specified by an attribute, assume all calls might be beyond the offset range of the `b` / `bl` instructions, and therefore load the function address into a register before performing a (otherwise direct) call. This is the default.

`-mshort-calls`

If not otherwise specified by an attribute, assume all direct calls are in the range of the `b` / `bl` instructions, so use these instructions for direct calls. The default is '`-mlong-calls`'.

`-msmall16`

Assume addresses can be loaded as 16-bit unsigned values. This does not apply to function addresses for which '`-mlong-calls`' semantics are in effect.

`-mfp-mode=mode`

Set the prevailing mode of the floating-point unit. This determines the floating-point mode that is provided and expected at function call and return time. Making this mode match the mode you predominantly need at function start can make your programs smaller and faster by avoiding unnecessary mode switches.

mode can be set to one the following values:

'`caller`' Any mode at function entry is valid, and retained or restored when the function returns, and when it calls other functions. This mode is useful for compiling libraries or other compilation units you might want to incorporate into different programs with different prevailing FPU modes, and the convenience of being able to use a single object file outweighs the size and speed overhead for any extra mode switching that might be needed, compared with what would be needed with a more specific choice of prevailing FPU mode.

'`truncate`'

This is the mode used for floating-point calculations with truncating (i.e. round towards zero) rounding mode. That includes conversion from floating point to integer.

'round-nearest'
This is the mode used for floating-point calculations with round-to-nearest-or-even rounding mode.

'int' This is the mode used to perform integer calculations in the FPU, e.g. integer multiply, or integer multiply-and-accumulate.

The default is '-mfp-mode=caller'

-mno-split-lohi
-mno-postinc
-mno-postmodify
Code generation tweaks that disable, respectively, splitting of 32-bit loads, generation of post-increment addresses, and generation of post-modify addresses. The defaults are 'msplit-lohi', '-mpost-inc', and '-mpost-modify'.

-mnovect-double
Change the preferred SIMD mode to SImode. The default is '-mvect-double', which uses DImode as preferred SIMD mode.

-max-vect-align=*num*
The maximum alignment for SIMD vector mode types. *num* may be 4 or 8. The default is 8. Note that this is an ABI change, even though many library function interfaces are unaffected if they don't use SIMD vector modes in places that affect size and/or alignment of relevant types.

-msplit-vecmove-early
Split vector moves into single word moves before reload. In theory this can give better register allocation, but so far the reverse seems to be generally the case.

-m1reg-*reg*
Specify a register to hold the constant −1, which makes loading small negative constants and certain bitmasks faster. Allowable values for *reg* are 'r43' and 'r63', which specify use of that register as a fixed register, and 'none', which means that no register is used for this purpose. The default is '-m1reg-none'.

3.18.3 AMD GCN Options

These options are defined specifically for the AMD GCN port.

-march=*gpu*
-mtune=*gpu*
Set architecture type or tuning for *gpu*. Supported values for *gpu* are

'fiji' Compile for GCN3 Fiji devices (gfx803).

'gfx900' Compile for GCN5 Vega 10 devices (gfx900).

-mstack-size=*bytes*
Specify how many *bytes* of stack space will be requested for each GPU thread (wave-front). Beware that there may be many threads and limited memory available. The size of the stack allocation may also have an impact on run-time performance. The default is 32KB when using OpenACC or OpenMP, and 1MB otherwise.

3.18.4 ARC Options

The following options control the architecture variant for which code is being compiled:

`-mbarrel-shifter`
: Generate instructions supported by barrel shifter. This is the default unless '`-mcpu=ARC601`' or '`-mcpu=ARCEM`' is in effect.

`-mjli-always`
: Force to call a function using jli_s instruction. This option is valid only for ARCv2 architecture.

`-mcpu=`*cpu*
: Set architecture type, register usage, and instruction scheduling parameters for *cpu*. There are also shortcut alias options available for backward compatibility and convenience. Supported values for *cpu* are

  '`arc600`' Compile for ARC600. Aliases: '`-mA6`', '`-mARC600`'.

  '`arc601`' Compile for ARC601. Alias: '`-mARC601`'.

  '`arc700`' Compile for ARC700. Aliases: '`-mA7`', '`-mARC700`'. This is the default when configured with '`--with-cpu=arc700`'.

  '`arcem`' Compile for ARC EM.

  '`archs`' Compile for ARC HS.

  '`em`' Compile for ARC EM CPU with no hardware extensions.

  '`em4`' Compile for ARC EM4 CPU.

  '`em4_dmips`'
  Compile for ARC EM4 DMIPS CPU.

  '`em4_fpus`'
  Compile for ARC EM4 DMIPS CPU with the single-precision floating-point extension.

  '`em4_fpuda`'
  Compile for ARC EM4 DMIPS CPU with single-precision floating-point and double assist instructions.

  '`hs`' Compile for ARC HS CPU with no hardware extensions except the atomic instructions.

  '`hs34`' Compile for ARC HS34 CPU.

  '`hs38`' Compile for ARC HS38 CPU.

  '`hs38_linux`'
  Compile for ARC HS38 CPU with all hardware extensions on.

  '`arc600_norm`'
  Compile for ARC 600 CPU with `norm` instructions enabled.

  '`arc600_mul32x16`'
  Compile for ARC 600 CPU with `norm` and 32x16-bit multiply instructions enabled.

'arc600_mul64'
Compile for ARC 600 CPU with norm and mul64-family instructions enabled.

'arc601_norm'
Compile for ARC 601 CPU with norm instructions enabled.

'arc601_mul32x16'
Compile for ARC 601 CPU with norm and 32x16-bit multiply instructions enabled.

'arc601_mul64'
Compile for ARC 601 CPU with norm and mul64-family instructions enabled.

'nps400' Compile for ARC 700 on NPS400 chip.

'em_mini' Compile for ARC EM minimalist configuration featuring reduced register set.

-mdpfp
-mdpfp-compact
Generate double-precision FPX instructions, tuned for the compact implementation.

-mdpfp-fast
Generate double-precision FPX instructions, tuned for the fast implementation.

-mno-dpfp-lrsr
Disable lr and sr instructions from using FPX extension aux registers.

-mea Generate extended arithmetic instructions. Currently only divaw, adds, subs, and sat16 are supported. This is always enabled for '-mcpu=ARC700'.

-mno-mpy Do not generate mpy-family instructions for ARC700. This option is deprecated.

-mmul32x16
Generate 32x16-bit multiply and multiply-accumulate instructions.

-mmul64 Generate mul64 and mulu64 instructions. Only valid for '-mcpu=ARC600'.

-mnorm Generate norm instructions. This is the default if '-mcpu=ARC700' is in effect.

-mspfp
-mspfp-compact
Generate single-precision FPX instructions, tuned for the compact implementation.

-mspfp-fast
Generate single-precision FPX instructions, tuned for the fast implementation.

-msimd Enable generation of ARC SIMD instructions via target-specific builtins. Only valid for '-mcpu=ARC700'.

-msoft-float
This option ignored; it is provided for compatibility purposes only. Software floating-point code is emitted by default, and this default can overridden by

FPX options; '-mspfp', '-mspfp-compact', or '-mspfp-fast' for single precision, and '-mdpfp', '-mdpfp-compact', or '-mdpfp-fast' for double precision.

-mswap Generate swap instructions.

-matomic This enables use of the locked load/store conditional extension to implement atomic memory built-in functions. Not available for ARC 6xx or ARC EM cores.

-mdiv-rem
Enable div and rem instructions for ARCv2 cores.

-mcode-density
Enable code density instructions for ARC EM. This option is on by default for ARC HS.

-mll64 Enable double load/store operations for ARC HS cores.

-mtp-regno=*regno*
Specify thread pointer register number.

-mmpy-option=*multo*
Compile ARCv2 code with a multiplier design option. You can specify the option using either a string or numeric value for *multo*. 'wlh1' is the default value. The recognized values are:

'0'
'none' No multiplier available.

'1'
'w' 16x16 multiplier, fully pipelined. The following instructions are enabled: mpyw and mpyuw.

'2'
'wlh1' 32x32 multiplier, fully pipelined (1 stage). The following instructions are additionally enabled: mpy, mpyu, mpym, mpymu, and mpy_s.

'3'
'wlh2' 32x32 multiplier, fully pipelined (2 stages). The following instructions are additionally enabled: mpy, mpyu, mpym, mpymu, and mpy_s.

'4'
'wlh3' Two 16x16 multipliers, blocking, sequential. The following instructions are additionally enabled: mpy, mpyu, mpym, mpymu, and mpy_s.

'5'
'wlh4' One 16x16 multiplier, blocking, sequential. The following instructions are additionally enabled: mpy, mpyu, mpym, mpymu, and mpy_s.

'6'
'wlh5' One 32x4 multiplier, blocking, sequential. The following instructions are additionally enabled: mpy, mpyu, mpym, mpymu, and mpy_s.

'7'
'plus_dmpy'
ARC HS SIMD support.

'8'
'plus_macd'
ARC HS SIMD support.

'9'
'plus_qmacw'
ARC HS SIMD support.

This option is only available for ARCv2 cores.

-mfpu=*fpu*

Enables support for specific floating-point hardware extensions for ARCv2 cores. Supported values for *fpu* are:

'fpus' Enables support for single-precision floating-point hardware extensions.

'fpud' Enables support for double-precision floating-point hardware extensions. The single-precision floating-point extension is also enabled. Not available for ARC EM.

'fpuda' Enables support for double-precision floating-point hardware extensions using double-precision assist instructions. The single-precision floating-point extension is also enabled. This option is only available for ARC EM.

'fpuda_div'
Enables support for double-precision floating-point hardware extensions using double-precision assist instructions. The single-precision floating-point, square-root, and divide extensions are also enabled. This option is only available for ARC EM.

'fpuda_fma'
Enables support for double-precision floating-point hardware extensions using double-precision assist instructions. The single-precision floating-point and fused multiply and add hardware extensions are also enabled. This option is only available for ARC EM.

'fpuda_all'
Enables support for double-precision floating-point hardware extensions using double-precision assist instructions. All single-precision floating-point hardware extensions are also enabled. This option is only available for ARC EM.

'fpus_div'
Enables support for single-precision floating-point, square-root and divide hardware extensions.

'fpud_div'
Enables support for double-precision floating-point, square-root and divide hardware extensions. This option includes option 'fpus_div'. Not available for ARC EM.

'`fpus_fma`'
Enables support for single-precision floating-point and fused multiply and add hardware extensions.

'`fpud_fma`'
Enables support for double-precision floating-point and fused multiply and add hardware extensions. This option includes option '`fpus_fma`'. Not available for ARC EM.

'`fpus_all`'
Enables support for all single-precision floating-point hardware extensions.

'`fpud_all`'
Enables support for all single- and double-precision floating-point hardware extensions. Not available for ARC EM.

`-mirq-ctrl-saved=`*`register-range, blink, lp_count`*
Specifies general-purposes registers that the processor automatically saves/restores on interrupt entry and exit. *register-range* is specified as two registers separated by a dash. The register range always starts with `r0`, the upper limit is `fp` register. *blink* and *lp_count* are optional. This option is only valid for ARC EM and ARC HS cores.

`-mrgf-banked-regs=`*`number`*
Specifies the number of registers replicated in second register bank on entry to fast interrupt. Fast interrupts are interrupts with the highest priority level P0. These interrupts save only PC and STATUS32 registers to avoid memory transactions during interrupt entry and exit sequences. Use this option when you are using fast interrupts in an ARC V2 family processor. Permitted values are 4, 8, 16, and 32.

`-mlpc-width=`*`width`*
Specify the width of the `lp_count` register. Valid values for *width* are 8, 16, 20, 24, 28 and 32 bits. The default width is fixed to 32 bits. If the width is less than 32, the compiler does not attempt to transform loops in your program to use the zero-delay loop mechanism unless it is known that the `lp_count` register can hold the required loop-counter value. Depending on the width specified, the compiler and run-time library might continue to use the loop mechanism for various needs. This option defines macro `__ARC_LPC_WIDTH__` with the value of *width*.

`-mrf16` This option instructs the compiler to generate code for a 16-entry register file. This option defines the `__ARC_RF16__` preprocessor macro.

`-mbranch-index`
Enable use of `bi` or `bih` instructions to implement jump tables.

The following options are passed through to the assembler, and also define preprocessor macro symbols.

`-mdsp-packa`
: Passed down to the assembler to enable the DSP Pack A extensions. Also sets the preprocessor symbol `__Xdsp_packa`. This option is deprecated.

`-mdvbf`
: Passed down to the assembler to enable the dual Viterbi butterfly extension. Also sets the preprocessor symbol `__Xdvbf`. This option is deprecated.

`-mlock`
: Passed down to the assembler to enable the locked load/store conditional extension. Also sets the preprocessor symbol `__Xlock`.

`-mmac-d16`
: Passed down to the assembler. Also sets the preprocessor symbol `__Xxmac_d16`. This option is deprecated.

`-mmac-24`
: Passed down to the assembler. Also sets the preprocessor symbol `__Xxmac_24`. This option is deprecated.

`-mrtsc`
: Passed down to the assembler to enable the 64-bit time-stamp counter extension instruction. Also sets the preprocessor symbol `__Xrtsc`. This option is deprecated.

`-mswape`
: Passed down to the assembler to enable the swap byte ordering extension instruction. Also sets the preprocessor symbol `__Xswape`.

`-mtelephony`
: Passed down to the assembler to enable dual- and single-operand instructions for telephony. Also sets the preprocessor symbol `__Xtelephony`. This option is deprecated.

`-mxy`
: Passed down to the assembler to enable the XY memory extension. Also sets the preprocessor symbol `__Xxy`.

The following options control how the assembly code is annotated:

`-misize`
: Annotate assembler instructions with estimated addresses.

`-mannotate-align`
: Explain what alignment considerations lead to the decision to make an instruction short or long.

The following options are passed through to the linker:

`-marclinux`
: Passed through to the linker, to specify use of the `arclinux` emulation. This option is enabled by default in tool chains built for `arc-linux-uclibc` and `arceb-linux-uclibc` targets when profiling is not requested.

`-marclinux_prof`
: Passed through to the linker, to specify use of the `arclinux_prof` emulation. This option is enabled by default in tool chains built for `arc-linux-uclibc` and `arceb-linux-uclibc` targets when profiling is requested.

The following options control the semantics of generated code:

`-mlong-calls`
: Generate calls as register indirect calls, thus providing access to the full 32-bit address range.

`-mmedium-calls`
: Don't use less than 25-bit addressing range for calls, which is the offset available for an unconditional branch-and-link instruction. Conditional execution of function calls is suppressed, to allow use of the 25-bit range, rather than the 21-bit range with conditional branch-and-link. This is the default for tool chains built for `arc-linux-uclibc` and `arceb-linux-uclibc` targets.

`-G` *num*
: Put definitions of externally-visible data in a small data section if that data is no bigger than *num* bytes. The default value of *num* is 4 for any ARC configuration, or 8 when we have double load/store operations.

`-mno-sdata`
: Do not generate sdata references. This is the default for tool chains built for `arc-linux-uclibc` and `arceb-linux-uclibc` targets.

`-mvolatile-cache`
: Use ordinarily cached memory accesses for volatile references. This is the default.

`-mno-volatile-cache`
: Enable cache bypass for volatile references.

The following options fine tune code generation:

`-malign-call`
: Do alignment optimizations for call instructions.

`-mauto-modify-reg`
: Enable the use of pre/post modify with register displacement.

`-mbbit-peephole`
: Enable bbit peephole2.

`-mno-brcc`
: This option disables a target-specific pass in '`arc_reorg`' to generate compare-and-branch (`brcc`) instructions. It has no effect on generation of these instructions driven by the combiner pass.

`-mcase-vector-pcrel`
: Use PC-relative switch case tables to enable case table shortening. This is the default for '`-Os`'.

`-mcompact-casesi`
: Enable compact `casesi` pattern. This is the default for '`-Os`', and only available for ARCv1 cores. This option is deprecated.

`-mno-cond-exec`
: Disable the ARCompact-specific pass to generate conditional execution instructions.

  Due to delay slot scheduling and interactions between operand numbers, literal sizes, instruction lengths, and the support for conditional execution, the target-independent pass to generate conditional execution is often lacking, so the ARC port has kept a special pass around that tries to find more conditional execution

generation opportunities after register allocation, branch shortening, and delay slot scheduling have been done. This pass generally, but not always, improves performance and code size, at the cost of extra compilation time, which is why there is an option to switch it off. If you have a problem with call instructions exceeding their allowable offset range because they are conditionalized, you should consider using '`-mmedium-calls`' instead.

`-mearly-cbranchsi`
: Enable pre-reload use of the `cbranchsi` pattern.

`-mexpand-adddi`
: Expand `adddi3` and `subdi3` at RTL generation time into `add.f`, `adc` etc. This option is deprecated.

`-mindexed-loads`
: Enable the use of indexed loads. This can be problematic because some optimizers then assume that indexed stores exist, which is not the case.

`-mlra`
: Enable Local Register Allocation. This is still experimental for ARC, so by default the compiler uses standard reload (i.e. '`-mno-lra`').

`-mlra-priority-none`
: Don't indicate any priority for target registers.

`-mlra-priority-compact`
: Indicate target register priority for r0..r3 / r12..r15.

`-mlra-priority-noncompact`
: Reduce target register priority for r0..r3 / r12..r15.

`-mmillicode`
: When optimizing for size (using '`-Os`'), prologues and epilogues that have to save or restore a large number of registers are often shortened by using call to a special function in libgcc; this is referred to as a *millicode* call. As these calls can pose performance issues, and/or cause linking issues when linking in a nonstandard way, this option is provided to turn on or off millicode call generation.

`-mcode-density-frame`
: This option enable the compiler to emit `enter` and `leave` instructions. These instructions are only valid for CPUs with code-density feature.

`-mmixed-code`
: Tweak register allocation to help 16-bit instruction generation. This generally has the effect of decreasing the average instruction size while increasing the instruction count.

`-mq-class`
: Enable '`q`' instruction alternatives. This is the default for '`-Os`'.

`-mRcq`
: Enable '`Rcq`' constraint handling. Most short code generation depends on this. This is the default.

`-mRcw`
: Enable '`Rcw`' constraint handling. Most ccfsm condexec mostly depends on this. This is the default.

`-msize-level=level`

Fine-tune size optimization with regards to instruction lengths and alignment. The recognized values for *level* are:

'0' No size optimization. This level is deprecated and treated like '1'.

'1' Short instructions are used opportunistically.

'2' In addition, alignment of loops and of code after barriers are dropped.

'3' In addition, optional data alignment is dropped, and the option '`Os`' is enabled.

This defaults to '3' when '`-Os`' is in effect. Otherwise, the behavior when this is not set is equivalent to level '1'.

`-mtune=cpu`

Set instruction scheduling parameters for *cpu*, overriding any implied by '`-mcpu=`'.

Supported values for *cpu* are

'`ARC600`' Tune for ARC600 CPU.

'`ARC601`' Tune for ARC601 CPU.

'`ARC700`' Tune for ARC700 CPU with standard multiplier block.

'`ARC700-xmac`'
Tune for ARC700 CPU with XMAC block.

'`ARC725D`' Tune for ARC725D CPU.

'`ARC750D`' Tune for ARC750D CPU.

`-mmultcost=num`

Cost to assume for a multiply instruction, with '4' being equal to a normal instruction.

`-munalign-prob-threshold=probability`

Set probability threshold for unaligning branches. When tuning for '`ARC700`' and optimizing for speed, branches without filled delay slot are preferably emitted unaligned and long, unless profiling indicates that the probability for the branch to be taken is below *probability*. See Section 10.5 [Cross-profiling], page 861. The default is (REG_BR_PROB_BASE/2), i.e. 5000.

The following options are maintained for backward compatibility, but are now deprecated and will be removed in a future release:

`-margonaut`

Obsolete FPX.

`-mbig-endian`
`-EB`

Compile code for big-endian targets. Use of these options is now deprecated. Big-endian code is supported by configuring GCC to build `arceb-elf32` and `arceb-linux-uclibc` targets, for which big endian is the default.

`-mlittle-endian`
`-EL` Compile code for little-endian targets. Use of these options is now deprecated. Little-endian code is supported by configuring GCC to build `arc-elf32` and `arc-linux-uclibc` targets, for which little endian is the default.

`-mbarrel_shifter`
Replaced by '`-mbarrel-shifter`'.

`-mdpfp_compact`
Replaced by '`-mdpfp-compact`'.

`-mdpfp_fast`
Replaced by '`-mdpfp-fast`'.

`-mdsp_packa`
Replaced by '`-mdsp-packa`'.

`-mEA` Replaced by '`-mea`'.

`-mmac_24` Replaced by '`-mmac-24`'.

`-mmac_d16`
Replaced by '`-mmac-d16`'.

`-mspfp_compact`
Replaced by '`-mspfp-compact`'.

`-mspfp_fast`
Replaced by '`-mspfp-fast`'.

`-mtune=`*cpu*
Values '`arc600`', '`arc601`', '`arc700`' and '`arc700-xmac`' for *cpu* are replaced by '`ARC600`', '`ARC601`', '`ARC700`' and '`ARC700-xmac`' respectively.

`-multcost=`*num*
Replaced by '`-mmultcost`'.

3.18.5 ARM Options

These '`-m`' options are defined for the ARM port:

`-mabi=`*name*
Generate code for the specified ABI. Permissible values are: '`apcs-gnu`', '`atpcs`', '`aapcs`', '`aapcs-linux`' and '`iwmmxt`'.

`-mapcs-frame`
Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying '`-fomit-frame-pointer`' with this option causes the stack frames not to be generated for leaf functions. The default is '`-mno-apcs-frame`'. This option is deprecated.

`-mapcs` This is a synonym for '`-mapcs-frame`' and is deprecated.

`-mthumb-interwork`
Generate code that supports calling between the ARM and Thumb instruction sets. Without this option, on pre-v5 architectures, the two

instruction sets cannot be reliably used inside one program. The default is '`-mno-thumb-interwork`', since slightly larger code is generated when '`-mthumb-interwork`' is specified. In AAPCS configurations this option is meaningless.

`-mno-sched-prolog`
: Prevent the reordering of instructions in the function prologue, or the merging of those instruction with the instructions in the function's body. This means that all functions start with a recognizable set of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start of functions inside an executable piece of code. The default is '`-msched-prolog`'.

`-mfloat-abi=`*name*
: Specifies which floating-point ABI to use. Permissible values are: '`soft`', '`softfp`' and '`hard`'.

  Specifying '`soft`' causes GCC to generate output containing library calls for floating-point operations. '`softfp`' allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions. '`hard`' allows generation of floating-point instructions and uses FPU-specific calling conventions.

  The default depends on the specific target configuration. Note that the hard-float and soft-float ABIs are not link-compatible; you must compile your entire program with the same ABI, and link with a compatible set of libraries.

`-mgeneral-regs-only`
: Generate code which uses only the general-purpose registers. This will prevent the compiler from using floating-point and Advanced SIMD registers but will not impose any restrictions on the assembler.

`-mlittle-endian`
: Generate code for a processor running in little-endian mode. This is the default for all standard configurations.

`-mbig-endian`
: Generate code for a processor running in big-endian mode; the default is to compile code for a little-endian processor.

`-mbe8`
`-mbe32`
: When linking a big-endian image select between BE8 and BE32 formats. The option has no effect for little-endian images and is ignored. The default is dependent on the selected target architecture. For ARMv6 and later architectures the default is BE8, for older architectures the default is BE32. BE32 format has been deprecated by ARM.

`-march=`*name*[`+extension...`]
: This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. This option can be used in conjunction with or instead of the '`-mcpu=`' option.

Permissible names are: ‘armv4t’, ‘armv5t’, ‘armv5te’, ‘armv6’, ‘armv6j’, ‘armv6k’, ‘armv6kz’, ‘armv6t2’, ‘armv6z’, ‘armv6zk’, ‘armv7’, ‘armv7-a’, ‘armv7ve’, ‘armv8-a’, ‘armv8.1-a’, ‘armv8.2-a’, ‘armv8.3-a’, ‘armv8.4-a’, ‘armv8.5-a’, ‘armv7-r’, ‘armv8-r’, ‘armv6-m’, ‘armv6s-m’, ‘armv7-m’, ‘armv7e-m’, ‘armv8-m.base’, ‘armv8-m.main’, ‘iwmmxt’ and ‘iwmmxt2’.

Additionally, the following architectures, which lack support for the Thumb execution state, are recognized but support is deprecated: ‘armv4’.

Many of the architectures support extensions. These can be added by appending ‘+*extension*’ to the architecture name. Extension options are processed in order and capabilities accumulate. An extension will also enable any necessary base extensions upon which it depends. For example, the ‘+crypto’ extension will always enable the ‘+simd’ extension. The exception to the additive construction is for extensions that are prefixed with ‘+no...’: these extensions disable the specified option and any other extensions that may depend on the presence of that extension.

For example, ‘-march=armv7-a+simd+nofp+vfpv4’ is equivalent to writing ‘-march=armv7-a+vfpv4’ since the ‘+simd’ option is entirely disabled by the ‘+nofp’ option that follows it.

Most extension names are generically named, but have an effect that is dependent upon the architecture to which it is applied. For example, the ‘+simd’ option can be applied to both ‘armv7-a’ and ‘armv8-a’ architectures, but will enable the original ARMv7-A Advanced SIMD (Neon) extensions for ‘armv7-a’ and the ARMv8-A variant for ‘armv8-a’.

The table below lists the supported extensions for each architecture. Architectures not mentioned do not support any extensions.

‘armv5te’
‘armv6’
‘armv6j’
‘armv6k’
‘armv6kz’
‘armv6t2’
‘armv6z’
‘armv6zk’

‘+fp’ The VFPv2 floating-point instructions. The extension ‘+vfpv2’ can be used as an alias for this extension.

‘+nofp’ Disable the floating-point instructions.

‘armv7’ The common subset of the ARMv7-A, ARMv7-R and ARMv7-M architectures.

‘+fp’ The VFPv3 floating-point instructions, with 16 double-precision registers. The extension ‘+vfpv3-d16’ can be used as an alias for this extension. Note that floating-point is not supported by the base ARMv7-M architecture, but is compatible with both the ARMv7-A and ARMv7-R architectures.

‘+nofp’ Disable the floating-point instructions.

‘armv7-a’

‘+mp’ The multiprocessing extension.

‘+sec’ The security extension.

‘+fp’ The VFPv3 floating-point instructions, with 16 double-precision registers. The extension ‘+vfpv3-d16’ can be used as an alias for this extension.

‘+simd’ The Advanced SIMD (Neon) v1 and the VFPv3 floating-point instructions. The extensions ‘+neon’ and ‘+neon-vfpv3’ can be used as aliases for this extension.

‘+vfpv3’ The VFPv3 floating-point instructions, with 32 double-precision registers.

‘+vfpv3-d16-fp16’
The VFPv3 floating-point instructions, with 16 double-precision registers and the half-precision floating-point conversion operations.

‘+vfpv3-fp16’
The VFPv3 floating-point instructions, with 32 double-precision registers and the half-precision floating-point conversion operations.

‘+vfpv4-d16’
The VFPv4 floating-point instructions, with 16 double-precision registers.

‘+vfpv4’ The VFPv4 floating-point instructions, with 32 double-precision registers.

‘+neon-fp16’
The Advanced SIMD (Neon) v1 and the VFPv3 floating-point instructions, with the half-precision floating-point conversion operations.

‘+neon-vfpv4’
The Advanced SIMD (Neon) v2 and the VFPv4 floating-point instructions.

‘+nosimd’ Disable the Advanced SIMD instructions (does not disable floating point).

‘+nofp’ Disable the floating-point and Advanced SIMD instructions.

‘armv7ve’ The extended version of the ARMv7-A architecture with support for virtualization.

'+fp' The VFPv4 floating-point instructions, with 16 double-precision registers. The extension '+vfpv4-d16' can be used as an alias for this extension.

'+simd' The Advanced SIMD (Neon) v2 and the VFPv4 floating-point instructions. The extension '+neon-vfpv4' can be used as an alias for this extension.

'+vfpv3-d16'
The VFPv3 floating-point instructions, with 16 double-precision registers.

'+vfpv3' The VFPv3 floating-point instructions, with 32 double-precision registers.

'+vfpv3-d16-fp16'
The VFPv3 floating-point instructions, with 16 double-precision registers and the half-precision floating-point conversion operations.

'+vfpv3-fp16'
The VFPv3 floating-point instructions, with 32 double-precision registers and the half-precision floating-point conversion operations.

'+vfpv4-d16'
The VFPv4 floating-point instructions, with 16 double-precision registers.

'+vfpv4' The VFPv4 floating-point instructions, with 32 double-precision registers.

'+neon' The Advanced SIMD (Neon) v1 and the VFPv3 floating-point instructions. The extension '+neon-vfpv3' can be used as an alias for this extension.

'+neon-fp16'
The Advanced SIMD (Neon) v1 and the VFPv3 floating-point instructions, with the half-precision floating-point conversion operations.

'+nosimd' Disable the Advanced SIMD instructions (does not disable floating point).

'+nofp' Disable the floating-point and Advanced SIMD instructions.

'armv8-a'

'+crc' The Cyclic Redundancy Check (CRC) instructions.

'+simd' The ARMv8-A Advanced SIMD and floating-point instructions.

‘+crypto’ The cryptographic instructions.

‘+nocrypto’
Disable the cryptographic instructions.

‘+nofp’ Disable the floating-point, Advanced SIMD and cryptographic instructions.

‘+sb’ Speculation Barrier Instruction.

‘+predres’
Execution and Data Prediction Restriction Instructions.

‘armv8.1-a’

‘+simd’ The ARMv8.1-A Advanced SIMD and floating-point instructions.

‘+crypto’ The cryptographic instructions. This also enables the Advanced SIMD and floating-point instructions.

‘+nocrypto’
Disable the cryptographic instructions.

‘+nofp’ Disable the floating-point, Advanced SIMD and cryptographic instructions.

‘+sb’ Speculation Barrier Instruction.

‘+predres’
Execution and Data Prediction Restriction Instructions.

‘armv8.2-a’
‘armv8.3-a’

‘+fp16’ The half-precision floating-point data processing instructions. This also enables the Advanced SIMD and floating-point instructions.

‘+fp16fml’
The half-precision floating-point fmla extension. This also enables the half-precision floating-point extension and Advanced SIMD and floating-point instructions.

‘+simd’ The ARMv8.1-A Advanced SIMD and floating-point instructions.

‘+crypto’ The cryptographic instructions. This also enables the Advanced SIMD and floating-point instructions.

‘+dotprod’
Enable the Dot Product extension. This also enables Advanced SIMD instructions.

‘+nocrypto’
Disable the cryptographic extension.

‘+nofp’ Disable the floating-point, Advanced SIMD and cryptographic instructions.

‘+sb’ Speculation Barrier Instruction.

‘+predres’ Execution and Data Prediction Restriction Instructions.

‘armv8.4-a’

‘+fp16’ The half-precision floating-point data processing instructions. This also enables the Advanced SIMD and floating-point instructions as well as the Dot Product extension and the half-precision floating-point fmla extension.

‘+simd’ The ARMv8.3-A Advanced SIMD and floating-point instructions as well as the Dot Product extension.

‘+crypto’ The cryptographic instructions. This also enables the Advanced SIMD and floating-point instructions as well as the Dot Product extension.

‘+nocrypto’ Disable the cryptographic extension.

‘+nofp’ Disable the floating-point, Advanced SIMD and cryptographic instructions.

‘+sb’ Speculation Barrier Instruction.

‘+predres’ Execution and Data Prediction Restriction Instructions.

‘armv8.5-a’

‘+fp16’ The half-precision floating-point data processing instructions. This also enables the Advanced SIMD and floating-point instructions as well as the Dot Product extension and the half-precision floating-point fmla extension.

‘+simd’ The ARMv8.3-A Advanced SIMD and floating-point instructions as well as the Dot Product extension.

‘+crypto’ The cryptographic instructions. This also enables the Advanced SIMD and floating-point instructions as well as the Dot Product extension.

‘+nocrypto’ Disable the cryptographic extension.

‘+nofp’ Disable the floating-point, Advanced SIMD and cryptographic instructions.

‘armv7-r’

‘+fp.sp’ The single-precision VFPv3 floating-point instructions. The extension ‘+vfpv3xd’ can be used as an alias for this extension.

‘+fp’ The VFPv3 floating-point instructions with 16 double-precision registers. The extension +vfpv3-d16 can be used as an alias for this extension.

‘+vfpv3xd-d16-fp16’ The single-precision VFPv3 floating-point instructions with 16 double-precision registers and the half-precision floating-point conversion operations.

‘+vfpv3-d16-fp16’ The VFPv3 floating-point instructions with 16 double-precision registers and the half-precision floating-point conversion operations.

‘+nofp’ Disable the floating-point extension.

‘+idiv’ The ARM-state integer division instructions.

‘+noidiv’ Disable the ARM-state integer division extension.

‘armv7e-m’

‘+fp’ The single-precision VFPv4 floating-point instructions.

‘+fpv5’ The single-precision FPv5 floating-point instructions.

‘+fp.dp’ The single- and double-precision FPv5 floating-point instructions.

‘+nofp’ Disable the floating-point extensions.

‘armv8-m.main’

‘+dsp’ The DSP instructions.

‘+nodsp’ Disable the DSP extension.

‘+fp’ The single-precision floating-point instructions.

‘+fp.dp’ The single- and double-precision floating-point instructions.

‘+nofp’ Disable the floating-point extension.

‘armv8-r’

‘+crc’ The Cyclic Redundancy Check (CRC) instructions.

‘+fp.sp’ The single-precision FPv5 floating-point instructions.

‘+simd’ The ARMv8-A Advanced SIMD and floating-point instructions.

‘+crypto’ The cryptographic instructions.

'+nocrypto'
Disable the cryptographic instructions.

'+nofp' Disable the floating-point, Advanced SIMD and cryptographic instructions.

'-march=native' causes the compiler to auto-detect the architecture of the build computer. At present, this feature is only supported on GNU/Linux, and not all architectures are recognized. If the auto-detect is unsuccessful the option has no effect.

-mtune=*name*

This option specifies the name of the target ARM processor for which GCC should tune the performance of the code. For some ARM implementations better performance can be obtained by using this option. Permissible names are: 'arm7tdmi', 'arm7tdmi-s', 'arm710t', 'arm720t', 'arm740t', 'strongarm', 'strongarm110', 'strongarm1100', 0'strongarm1110', 'arm8', 'arm810', 'arm9', 'arm9e', 'arm920', 'arm920t', 'arm922t', 'arm946e-s', 'arm966e-s', 'arm968e-s', 'arm926ej-s', 'arm940t', 'arm9tdmi', 'arm10tdmi', 'arm1020t', 'arm1026ej-s', 'arm10e', 'arm1020e', 'arm1022e', 'arm1136j-s', 'arm1136jf-s', 'mpcore', 'mpcorenovfp', 'arm1156t2-s', 'arm1156t2f-s', 'arm1176jz-s', 'arm1176jzf-s', 'generic-armv7-a', 'cortex-a5', 'cortex-a7', 'cortex-a8', 'cortex-a9', 'cortex-a12', 'cortex-a15', 'cortex-a17', 'cortex-a32', 'cortex-a35', 'cortex-a53', 'cortex-a55', 'cortex-a57', 'cortex-a72', 'cortex-a73', 'cortex-a75', 'cortex-a76', 'ares', 'cortex-r4', 'cortex-r4f', 'cortex-r5', 'cortex-r7', 'cortex-r8', 'cortex-r52', 'cortex-m0', 'cortex-m0plus', 'cortex-m1', 'cortex-m3', 'cortex-m4', 'cortex-m7', 'cortex-m23', 'cortex-m33', 'cortex-m1.small-multiply', 'cortex-m0.small-multiply', 'cortex-m0plus.small-multiply', 'exynos-m1', 'marvell-pj4', 'neoverse-n1', 'xscale', 'iwmmxt', 'iwmmxt2', 'ep9312', 'fa526', 'fa626', 'fa606te', 'fa626te', 'fmp626', 'fa726te', 'xgene1'.

Additionally, this option can specify that GCC should tune the performance of the code for a big.LITTLE system. Permissible names are: 'cortex-a15.cortex-a7', 'cortex-a17.cortex-a7', 'cortex-a57.cortex-a53', 'cortex-a72.cortex-a53', 'cortex-a72.cortex-a35', 'cortex-a73.cortex-a53', 'cortex-a75.cortex-a55', 'cortex-a76.cortex-a55'.

'-mtune=generic-*arch*' specifies that GCC should tune the performance for a blend of processors within architecture *arch*. The aim is to generate code that run well on the current most popular processors, balancing between optimizations that benefit some CPUs in the range, and avoiding performance pitfalls of other CPUs. The effects of this option may change in future GCC versions as CPU models come and go.

'-mtune' permits the same extension options as '-mcpu', but the extension options do not affect the tuning of the generated code.

'-mtune=native' causes the compiler to auto-detect the CPU of the build computer. At present, this feature is only supported on GNU/Linux, and not all

architectures are recognized. If the auto-detect is unsuccessful the option has no effect.

`-mcpu=name[+extension...]`

This specifies the name of the target ARM processor. GCC uses this name to derive the name of the target ARM architecture (as if specified by '`-march`') and the ARM processor type for which to tune for performance (as if specified by '`-mtune`'). Where this option is used in conjunction with '`-march`' or '`-mtune`', those options take precedence over the appropriate part of this option.

Many of the supported CPUs implement optional architectural extensions. Where this is so the architectural extensions are normally enabled by default. If implementations that lack the extension exist, then the extension syntax can be used to disable those extensions that have been omitted. For floating-point and Advanced SIMD (Neon) instructions, the settings of the options '`-mfloat-abi`' and '`-mfpu`' must also be considered: floating-point and Advanced SIMD instructions will only be used if '`-mfloat-abi`' is not set to '`soft`'; and any setting of '`-mfpu`' other than '`auto`' will override the available floating-point and SIMD extension instructions.

For example, '`cortex-a9`' can be found in three major configurations: integer only, with just a floating-point unit or with floating-point and Advanced SIMD. The default is to enable all the instructions, but the extensions '`+nosimd`' and '`+nofp`' can be used to disable just the SIMD or both the SIMD and floating-point instructions respectively.

Permissible names for this option are the same as those for '`-mtune`'.

The following extension options are common to the listed CPUs:

'`+nodsp`' Disable the DSP instructions on '`cortex-m33`'.

'`+nofp`' Disables the floating-point instructions on '`arm9e`', '`arm946e-s`', '`arm966e-s`', '`arm968e-s`', '`arm10e`', '`arm1020e`', '`arm1022e`', '`arm926ej-s`', '`arm1026ej-s`', '`cortex-r5`', '`cortex-r7`', '`cortex-r8`', '`cortex-m4`', '`cortex-m7`' and '`cortex-m33`'. Disables the floating-point and SIMD instructions on '`generic-armv7-a`', '`cortex-a5`', '`cortex-a7`', '`cortex-a8`', '`cortex-a9`', '`cortex-a12`', '`cortex-a15`', '`cortex-a17`', '`cortex-a15.cortex-a7`', '`cortex-a17.cortex-a7`', '`cortex-a32`', '`cortex-a35`', '`cortex-a53`' and '`cortex-a55`'.

'`+nofp.dp`'
Disables the double-precision component of the floating-point instructions on '`cortex-r5`', '`cortex-r7`', '`cortex-r8`', '`cortex-r52`' and '`cortex-m7`'.

'`+nosimd`' Disables the SIMD (but not floating-point) instructions on '`generic-armv7-a`', '`cortex-a5`', '`cortex-a7`' and '`cortex-a9`'.

'`+crypto`' Enables the cryptographic instructions on '`cortex-a32`', '`cortex-a35`', '`cortex-a53`', '`cortex-a55`', '`cortex-a57`', '`cortex-a72`', '`cortex-a73`', '`cortex-a75`', '`exynos-m1`',

'xgene1', 'cortex-a57.cortex-a53', 'cortex-a72.cortex-a53', 'cortex-a73.cortex-a35', 'cortex-a73.cortex-a53' and 'cortex-a75.cortex-a55'.

Additionally the 'generic-armv7-a' pseudo target defaults to VFPv3 with 16 double-precision registers. It supports the following extension options: 'mp', 'sec', 'vfpv3-d16', 'vfpv3', 'vfpv3-d16-fp16', 'vfpv3-fp16', 'vfpv4-d16', 'vfpv4', 'neon', 'neon-vfpv3', 'neon-fp16', 'neon-vfpv4'. The meanings are the same as for the extensions to '-march=armv7-a'.

'-mcpu=generic-*arch*' is also permissible, and is equivalent to '-march=*arch* -mtune=generic-*arch*'. See '-mtune' for more information.

'-mcpu=native' causes the compiler to auto-detect the CPU of the build computer. At present, this feature is only supported on GNU/Linux, and not all architectures are recognized. If the auto-detect is unsuccessful the option has no effect.

-mfpu=*name*

This specifies what floating-point hardware (or hardware emulation) is available on the target. Permissible names are: 'auto', 'vfpv2', 'vfpv3', 'vfpv3-fp16', 'vfpv3-d16', 'vfpv3-d16-fp16', 'vfpv3xd', 'vfpv3xd-fp16', 'neon-vfpv3', 'neon-fp16', 'vfpv4', 'vfpv4-d16', 'fpv4-sp-d16', 'neon-vfpv4', 'fpv5-d16', 'fpv5-sp-d16', 'fp-armv8', 'neon-fp-armv8' and 'crypto-neon-fp-armv8'. Note that 'neon' is an alias for 'neon-vfpv3' and 'vfp' is an alias for 'vfpv2'.

The setting 'auto' is the default and is special. It causes the compiler to select the floating-point and Advanced SIMD instructions based on the settings of '-mcpu' and '-march'.

If the selected floating-point hardware includes the NEON extension (e.g. '-mfpu=neon'), note that floating-point operations are not generated by GCC's auto-vectorization pass unless '-funsafe-math-optimizations' is also specified. This is because NEON hardware does not fully implement the IEEE 754 standard for floating-point arithmetic (in particular denormal values are treated as zero), so the use of NEON instructions may lead to a loss of precision.

You can also set the fpu name at function level by using the target("fpu=") function attributes (see Section 6.33.5 [ARM Function Attributes], page 502) or pragmas (see Section 6.62.15 [Function Specific Option Pragmas], page 808).

-mfp16-format=*name*

Specify the format of the __fp16 half-precision floating-point type. Permissible names are 'none', 'ieee', and 'alternative'; the default is 'none', in which case the __fp16 type is not defined. See Section 6.13 [Half-Precision], page 463, for more information.

-mstructure-size-boundary=*n*

The sizes of all structures and unions are rounded up to a multiple of the number of bits set by this option. Permissible values are 8, 32 and 64. The default value varies for different toolchains. For the COFF targeted toolchain the default value is 8. A value of 64 is only allowed if the underlying ABI supports it.

Specifying a larger number can produce faster, more efficient code, but can also increase the size of the program. Different values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with another value, if they exchange information using structures or unions.

This option is deprecated.

`-mabort-on-noreturn`
: Generate a call to the function `abort` at the end of a `noreturn` function. It is executed if the function tries to return.

`-mlong-calls`
`-mno-long-calls`
: Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function lies outside of the 64-megabyte addressing range of the offset-based version of subroutine call instruction.

Even if this switch is enabled, not all function calls are turned into long calls. The heuristic is that static functions, functions that have the `short_call` attribute, functions that are inside the scope of a `#pragma no_long_calls` directive, and functions whose definitions have already been compiled within the current compilation unit are not turned into long calls. The exceptions to this rule are that weak function definitions, functions with the `long_call` attribute or the `section` attribute, and functions that are within the scope of a `#pragma long_calls` directive are always turned into long calls.

This feature is not enabled by default. Specifying '`-mno-long-calls`' restores the default behavior, as does placing the function calls within the scope of a `#pragma long_calls_off` directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.

`-msingle-pic-base`
: Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The runtime system is responsible for initializing this register with an appropriate value before execution begins.

`-mpic-register=`*reg*
: Specify the register to be used for PIC addressing. For standard PIC base case, the default is any suitable register determined by compiler. For single PIC base case, the default is '`R9`' if target is EABI based or stack-checking is enabled, otherwise the default is '`R10`'.

`-mpic-data-is-text-relative`
: Assume that the displacement between the text and data segments is fixed at static link time. This permits using PC-relative addressing operations to access data known to be in the data segment. For non-VxWorks RTP targets, this option is enabled by default. When disabled on such targets, it will enable '`-msingle-pic-base`' by default.

`-mpoke-function-name`

Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this:

```
t0
    .ascii "arm_poke_function_name", 0
    .align
t1
    .word 0xff000000 + (t1 - t0)
arm_poke_function_name
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
```

When performing a stack backtrace, code can inspect the value of `pc` stored at `fp + 0`. If the trace function then looks at location `pc - 12` and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length `((pc[-3]) & 0xff000000)`.

`-mthumb`
`-marm`

Select between generating code that executes in ARM and Thumb states. The default for most configurations is to generate code that executes in ARM state, but the default can be changed by configuring GCC with the '`--with-mode=`'*state* configure option.

You can also override the ARM and Thumb mode for each function by using the `target("thumb")` and `target("arm")` function attributes (see Section 6.33.5 [ARM Function Attributes], page 502) or pragmas (see Section 6.62.15 [Function Specific Option Pragmas], page 808).

`-mflip-thumb`

Switch ARM/Thumb modes on alternating functions. This option is provided for regression testing of mixed Thumb/ARM code generation, and is not intended for ordinary use in compiling code.

`-mtpcs-frame`

Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all non-leaf functions. (A leaf function is one that does not call any other functions.) The default is '`-mno-tpcs-frame`'.

`-mtpcs-leaf-frame`

Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all leaf functions. (A leaf function is one that does not call any other functions.) The default is '`-mno-apcs-leaf-frame`'.

`-mcallee-super-interworking`

Gives all externally visible functions in the file being compiled an ARM instruction set header which switches to Thumb mode before executing the rest of the function. This allows these functions to be called from non-interworking code. This option is not valid in AAPCS configurations because interworking is enabled by default.

`-mcaller-super-interworking`

Allows calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. There is a small overhead in the cost of executing a function pointer if this option is enabled. This option is not valid in AAPCS configurations because interworking is enabled by default.

`-mtp=`*name*

Specify the access model for the thread local storage pointer. The valid models are '`soft`', which generates calls to `__aeabi_read_tp`, '`cp15`', which fetches the thread pointer from `cp15` directly (supported in the arm6k architecture), and '`auto`', which uses the best available method for the selected processor. The default setting is '`auto`'.

`-mtls-dialect=`*dialect*

Specify the dialect to use for accessing thread local storage. Two *dialect*s are supported—'`gnu`' and '`gnu2`'. The '`gnu`' dialect selects the original GNU scheme for supporting local and global dynamic TLS models. The '`gnu2`' dialect selects the GNU descriptor scheme, which provides better performance for shared libraries. The GNU descriptor scheme is compatible with the original scheme, but does require new assembler, linker and library support. Initial and local exec TLS models are unaffected by this option and always use the original scheme.

`-mword-relocations`

Only generate absolute relocations on word-sized values (i.e. R_ARM_ABS32). This is enabled by default on targets (uClinux, SymbianOS) where the runtime loader imposes this restriction, and when '`-fpic`' or '`-fPIC`' is specified. This option conflicts with '`-mslow-flash-data`'.

`-mfix-cortex-m3-ldrd`

Some Cortex-M3 cores can cause data corruption when `ldrd` instructions with overlapping destination and base registers are used. This option avoids generating these instructions. This option is enabled by default when '`-mcpu=cortex-m3`' is specified.

`-munaligned-access`
`-mno-unaligned-access`

Enables (or disables) reading and writing of 16- and 32- bit values from addresses that are not 16- or 32- bit aligned. By default unaligned access is disabled for all pre-ARMv6, all ARMv6-M and for ARMv8-M Baseline architectures, and enabled for all other architectures. If unaligned access is not enabled then words in packed data structures are accessed a byte at a time.

The ARM attribute `Tag_CPU_unaligned_access` is set in the generated object file to either true or false, depending upon the setting of this option. If unaligned access is enabled then the preprocessor symbol `__ARM_FEATURE_UNALIGNED` is also defined.

`-mneon-for-64bits`

Enables using Neon to handle scalar 64-bits operations. This is disabled by default since the cost of moving data from core registers to Neon is high.

`-mslow-flash-data`
: Assume loading data from flash is slower than fetching instruction. Therefore literal load is minimized for better performance. This option is only supported when compiling for ARMv7 M-profile and off by default. It conflicts with '`-mword-relocations`'.

`-masm-syntax-unified`
: Assume inline assembler is using unified asm syntax. The default is currently off which implies divided syntax. This option has no impact on Thumb2. However, this may change in future releases of GCC. Divided syntax should be considered deprecated.

`-mrestrict-it`
: Restricts generation of IT blocks to conform to the rules of ARMv8-A. IT blocks can only contain a single 16-bit instruction from a select set of instructions. This option is on by default for ARMv8-A Thumb mode.

`-mprint-tune-info`
: Print CPU tuning information as comment in assembler file. This is an option used only for regression testing of the compiler and not intended for ordinary use in compiling code. This option is disabled by default.

`-mverbose-cost-dump`
: Enable verbose cost model dumping in the debug dump files. This option is provided for use in debugging the compiler.

`-mpure-code`
: Do not allow constant data to be placed in code sections. Additionally, when compiling for ELF object format give all text sections the ELF processor-specific section attribute `SHF_ARM_PURECODE`. This option is only available when generating non-pic code for M-profile targets with the MOVT instruction.

`-mcmse`
: Generate secure code as per the "ARMv8-M Security Extensions: Requirements on Development Tools Engineering Specification", which can be found on `http://infocenter.arm.com/help/topic/com.arm.doc.ecm0359818/ECM0359818_armv8m_security_extensions_reqs_on_dev_tools_1_0.pdf`.

3.18.6 AVR Options

These options are defined for AVR implementations:

`-mmcu=`*mcu*
: Specify Atmel AVR instruction set architectures (ISA) or MCU type.

  The default for this option is '`avr2`'.

  GCC supports the following AVR devices and ISAs:

  `avr2`
  : "Classic" devices with up to 8 KiB of program memory.
  mcu = `attiny22`, `attiny26`, `at90c8534`, `at90s2313`, `at90s2323`, `at90s2333`, `at90s2343`, `at90s4414`, `at90s4433`, `at90s4434`, `at90s8515`, `at90s8535`.

  `avr25`
  : "Classic" devices with up to 8 KiB of program memory and with the `MOVW` instruction.

mcu = ata5272, ata6616c, attiny13, attiny13a, attiny2313, attiny2313a, attiny24, attiny24a, attiny25, attiny261, attiny261a, attiny43u, attiny4313, attiny44, attiny44a, attiny441, attiny45, attiny461, attiny461a, attiny48, attiny828, attiny84, attiny84a, attiny841, attiny85, attiny861, attiny861a, attiny87, attiny88, at86rf401.

avr3 "Classic" devices with 16 KiB up to 64 KiB of program memory.
mcu = at43usb355, at76c711.

avr31 "Classic" devices with 128 KiB of program memory.
mcu = atmega103, at43usb320.

avr35 "Classic" devices with 16 KiB up to 64 KiB of program memory and with the MOVW instruction.
mcu = ata5505, ata6617c, ata664251, atmega16u2, atmega32u2, atmega8u2, attiny1634, attiny167, at90usb162, at90usb82.

avr4 "Enhanced" devices with up to 8 KiB of program memory.
mcu = ata6285, ata6286, ata6289, ata6612c, atmega48, atmega48a, atmega48p, atmega48pa, atmega48pb, atmega8, atmega8a, atmega8hva, atmega8515, atmega8535, atmega88, atmega88a, atmega88p, atmega88pa, atmega88pb, at90pwm1, at90pwm2, at90pwm2b, at90pwm3, at90pwm3b, at90pwm81.

avr5 "Enhanced" devices with 16 KiB up to 64 KiB of program memory.
mcu = ata5702m322, ata5782, ata5790, ata5790n, ata5791, ata5795, ata5831, ata6613c, ata6614q, ata8210, ata8510, atmega16, atmega16a, atmega16hva, atmega16hva2, atmega16hvb, atmega16hvbrevb, atmega16m1, atmega16u4, atmega161, atmega162, atmega163, atmega164a, atmega164p, atmega164pa, atmega165, atmega165a, atmega165p, atmega165pa, atmega168, atmega168a, atmega168p, atmega168pa, atmega168pb, atmega169, atmega169a, atmega169p, atmega169pa, atmega32, atmega32a, atmega32c1, atmega32hvb, atmega32hvbrevb, atmega32m1, atmega32u4, atmega32u6, atmega323, atmega324a, atmega324p, atmega324pa, atmega325, atmega325a, atmega325p, atmega325pa, atmega3250, atmega3250a, atmega3250p, atmega3250pa, atmega328, atmega328p, atmega328pb, atmega329, atmega329a, atmega329p, atmega329pa, atmega3290, atmega3290a, atmega3290p, atmega3290pa, atmega406, atmega64, atmega64a, atmega64c1, atmega64hve, atmega64hve2, atmega64m1, atmega64rfr2, atmega640, atmega644, atmega644a, atmega644p, atmega644pa, atmega644rfr2, atmega645, atmega645a, atmega645p, atmega6450, atmega6450a, atmega6450p, atmega649, atmega649a, atmega649p, atmega6490, atmega6490a, atmega6490p, at90can32, at90can64, at90pwm161, at90pwm216,

at90pwm316, at90scr100, at90usb646, at90usb647, at94k, m3000.

avr51 "Enhanced" devices with 128 KiB of program memory.
mcu = atmega128, atmega128a, atmega128rfa1, atmega128rfr2, atmega1280, atmega1281, atmega1284, atmega1284p, atmega1284rfr2, at90can128, at90usb1286, at90usb1287.

avr6 "Enhanced" devices with 3-byte PC, i.e. with more than 128 KiB of program memory.
mcu = atmega256rfr2, atmega2560, atmega2561, atmega2564rfr2.

avrxmega2
"XMEGA" devices with more than 8 KiB and up to 64 KiB of program memory.
mcu = atxmega16a4, atxmega16a4u, atxmega16c4, atxmega16d4, atxmega16e5, atxmega32a4, atxmega32a4u, atxmega32c3, atxmega32c4, atxmega32d3, atxmega32d4, atxmega32e5, atxmega8e5.

avrxmega3
"XMEGA" devices with up to 64 KiB of combined program memory and RAM, and with program memory visible in the RAM address space.
mcu = attiny1614, attiny1616, attiny1617, attiny212, attiny214, attiny3214, attiny3216, attiny3217, attiny412, attiny414, attiny416, attiny417, attiny814, attiny816, attiny817.

avrxmega4
"XMEGA" devices with more than 64 KiB and up to 128 KiB of program memory.
mcu = atxmega64a3, atxmega64a3u, atxmega64a4u, atxmega64b1, atxmega64b3, atxmega64c3, atxmega64d3, atxmega64d4.

avrxmega5
"XMEGA" devices with more than 64 KiB and up to 128 KiB of program memory and more than 64 KiB of RAM.
mcu = atxmega64a1, atxmega64a1u.

avrxmega6
"XMEGA" devices with more than 128 KiB of program memory.
mcu = atxmega128a3, atxmega128a3u, atxmega128b1, atxmega128b3, atxmega128c3, atxmega128d3, atxmega128d4, atxmega192a3, atxmega192a3u, atxmega192c3, atxmega192d3, atxmega256a3, atxmega256a3b, atxmega256a3bu, atxmega256a3u, atxmega256c3, atxmega256d3, atxmega384c3, atxmega384d3.

`avrxmega7`
: "XMEGA" devices with more than 128 KiB of program memory and more than 64 KiB of RAM.
mcu = `atxmega128a1`, `atxmega128a1u`, `atxmega128a4u`.

`avrtiny`
: "TINY" Tiny core devices with 512 B up to 4 KiB of program memory.
mcu = `attiny10`, `attiny20`, `attiny4`, `attiny40`, `attiny5`, `attiny9`.

`avr1`
: This ISA is implemented by the minimal AVR core and supported for assembler only.
mcu = `attiny11`, `attiny12`, `attiny15`, `attiny28`, `at90s1200`.

`-mabsdata`
: Assume that all data in static storage can be accessed by LDS / STS instructions. This option has only an effect on reduced Tiny devices like ATtiny40. See also the `absdata` Section 6.34.3 [AVR Variable Attributes], page 542.

`-maccumulate-args`
: Accumulate outgoing function arguments and acquire/release the needed stack space for outgoing function arguments once in function prologue/epilogue. Without this option, outgoing arguments are pushed before calling a function and popped afterwards.

Popping the arguments after the function call can be expensive on AVR so that accumulating the stack space might lead to smaller executables because arguments need not be removed from the stack after such a function call.

This option can lead to reduced code size for functions that perform several calls to functions that get their arguments on the stack like calls to printf-like functions.

`-mbranch-cost=`*cost*
: Set the branch costs for conditional branch instructions to *cost*. Reasonable values for *cost* are small, non-negative integers. The default branch cost is 0.

`-mcall-prologues`
: Functions prologues/epilogues are expanded as calls to appropriate subroutines. Code size is smaller.

`-mgas-isr-prologues`
: Interrupt service routines (ISRs) may use the `__gcc_isr` pseudo instruction supported by GNU Binutils. If this option is on, the feature can still be disabled for individual ISRs by means of the Section 6.33.6 [`no_gccisr`], page 504 function attribute. This feature is activated per default if optimization is on (but not with '`-Og`', see Section 3.10 [Optimize Options], page 125), and if GNU Binutils support PR21683 (`https://sourceware.org/PR21683`).

`-mint8`
: Assume `int` to be 8-bit integer. This affects the sizes of all types: a `char` is 1 byte, an `int` is 1 byte, a `long` is 2 bytes, and `long long` is 4 bytes. Please note that this option does not conform to the C standards, but it results in smaller code size.

`-mmain-is-OS_task`

Do not save registers in `main`. The effect is the same like attaching attribute Section 6.33.6 [`OS_task`], page 504 to `main`. It is activated per default if optimization is on.

`-mn-flash=num`

Assume that the flash memory has a size of *num* times 64 KiB.

`-mno-interrupts`

Generated code is not compatible with hardware interrupts. Code size is smaller.

`-mrelax` Try to replace `CALL` resp. `JMP` instruction by the shorter `RCALL` resp. `RJMP` instruction if applicable. Setting '`-mrelax`' just adds the '`--mlink-relax`' option to the assembler's command line and the '`--relax`' option to the linker's command line.

Jump relaxing is performed by the linker because jump offsets are not known before code is located. Therefore, the assembler code generated by the compiler is the same, but the instructions in the executable may differ from instructions in the assembler code.

Relaxing must be turned on if linker stubs are needed, see the section on `EIND` and linker stubs below.

`-mrmw` Assume that the device supports the Read-Modify-Write instructions `XCH`, `LAC`, `LAS` and `LAT`.

`-mshort-calls`

Assume that `RJMP` and `RCALL` can target the whole program memory.

This option is used internally for multilib selection. It is not an optimization option, and you don't need to set it by hand.

`-msp8` Treat the stack pointer register as an 8-bit register, i.e. assume the high byte of the stack pointer is zero. In general, you don't need to set this option by hand.

This option is used internally by the compiler to select and build multilibs for architectures `avr2` and `avr25`. These architectures mix devices with and without `SPH`. For any setting other than '`-mmcu=avr2`' or '`-mmcu=avr25`' the compiler driver adds or removes this option from the compiler proper's command line, because the compiler then knows if the device or architecture has an 8-bit stack pointer and thus no `SPH` register or not.

`-mstrict-X`

Use address register `X` in a way proposed by the hardware. This means that `X` is only used in indirect, post-increment or pre-decrement addressing.

Without this option, the `X` register may be used in the same way as `Y` or `Z` which then is emulated by additional instructions. For example, loading a value with `X+const` addressing with a small non-negative `const < 64` to a register *Rn* is performed as

```
adiw r26, const   ; X += const
ld   Rn, X        ; Rn = *X
sbiw r26, const   ; X -= const
```

`-mtiny-stack`
: Only change the lower 8 bits of the stack pointer.

`-mfract-convert-truncate`
: Allow to use truncation instead of rounding towards zero for fractional fixed-point types.

`-nodevicelib`
: Don't link against AVR-LibC's device specific library `lib<mcu>.a`.

`-Waddr-space-convert`
: Warn about conversions between address spaces in the case where the resulting address space is not contained in the incoming address space.

`-Wmisspelled-isr`
: Warn if the ISR is misspelled, i.e. without `__vector` prefix. Enabled by default.

3.18.6.1 `EIND` and Devices with More Than 128 Ki Bytes of Flash

Pointers in the implementation are 16 bits wide. The address of a function or label is represented as word address so that indirect jumps and calls can target any code address in the range of 64 Ki words.

In order to facilitate indirect jump on devices with more than 128 Ki bytes of program memory space, there is a special function register called `EIND` that serves as most significant part of the target address when `EICALL` or `EIJMP` instructions are used.

Indirect jumps and calls on these devices are handled as follows by the compiler and are subject to some limitations:

- The compiler never sets `EIND`.
- The compiler uses `EIND` implicitly in `EICALL/EIJMP` instructions or might read `EIND` directly in order to emulate an indirect call/jump by means of a `RET` instruction.
- The compiler assumes that `EIND` never changes during the startup code or during the application. In particular, `EIND` is not saved/restored in function or interrupt service routine prologue/epilogue.
- For indirect calls to functions and computed goto, the linker generates *stubs*. Stubs are jump pads sometimes also called *trampolines*. Thus, the indirect call/jump jumps to such a stub. The stub contains a direct jump to the desired address.
- Linker relaxation must be turned on so that the linker generates the stubs correctly in all situations. See the compiler option '`-mrelax`' and the linker option '`--relax`'. There are corner cases where the linker is supposed to generate stubs but aborts without relaxation and without a helpful error message.
- The default linker script is arranged for code with `EIND = 0`. If code is supposed to work for a setup with `EIND != 0`, a custom linker script has to be used in order to place the sections whose name start with `.trampolines` into the segment where `EIND` points to.
- The startup code from libgcc never sets `EIND`. Notice that startup code is a blend of code from libgcc and AVR-LibC. For the impact of AVR-LibC on `EIND`, see the AVR-LibC user manual (`http://nongnu.org/avr-libc/user-manual/`).

- It is legitimate for user-specific startup code to set up `EIND` early, for example by means of initialization code located in section `.init3`. Such code runs prior to general startup code that initializes RAM and calls constructors, but after the bit of startup code from AVR-LibC that sets `EIND` to the segment where the vector table is located.

```
#include <avr/io.h>

static void
__attribute__((section(".init3"),naked,used,no_instrument_function))
init3_set_eind (void)
{
  __asm volatile ("ldi r24,pm_hh8(__trampolines_start)\n\t"
                  "out %i0,r24" :: "n" (&EIND) : "r24","memory");
}
```

  The `__trampolines_start` symbol is defined in the linker script.

- Stubs are generated automatically by the linker if the following two conditions are met:
  - The address of a label is taken by means of the `gs` modifier (short for *generate stubs*) like so:

    ```
    LDI r24, lo8(gs(func))
    LDI r25, hi8(gs(func))
    ```

  - The final location of that label is in a code segment *outside* the segment where the stubs are located.
- The compiler emits such `gs` modifiers for code labels in the following situations:
  - Taking address of a function or code label.
  - Computed goto.
  - If prologue-save function is used, see '`-mcall-prologues`' command-line option.
  - Switch/case dispatch tables. If you do not want such dispatch tables you can specify the '`-fno-jump-tables`' command-line option.
  - C and C++ constructors/destructors called during startup/shutdown.
  - If the tools hit a `gs()` modifier explained above.
- Jumping to non-symbolic addresses like so is *not* supported:

```
int main (void)
{
    /* Call function at word address 0x2 */
    return ((int(*)(void)) 0x2)();
}
```

  Instead, a stub has to be set up, i.e. the function has to be called through a symbol (`func_4` in the example):

```
int main (void)
{
    extern int func_4 (void);

    /* Call function at byte address 0x4 */
    return func_4();
```

```
}
```

and the application be linked with '`-Wl,--defsym,func_4=0x4`'. Alternatively, `func_4` can be defined in the linker script.

3.18.6.2 Handling of the RAMPD, RAMPX, RAMPY and RAMPZ Special Function Registers

Some AVR devices support memories larger than the 64 KiB range that can be accessed with 16-bit pointers. To access memory locations outside this 64 KiB range, the content of a `RAMP` register is used as high part of the address: The `X`, `Y`, `Z` address register is concatenated with the `RAMPX`, `RAMPY`, `RAMPZ` special function register, respectively, to get a wide address. Similarly, `RAMPD` is used together with direct addressing.

- The startup code initializes the `RAMP` special function registers with zero.
- If a [AVR Named Address Spaces], page 466 other than generic or `__flash` is used, then `RAMPZ` is set as needed before the operation.
- If the device supports RAM larger than 64 KiB and the compiler needs to change `RAMPZ` to accomplish an operation, `RAMPZ` is reset to zero after the operation.
- If the device comes with a specific `RAMP` register, the ISR prologue/epilogue saves/restores that SFR and initializes it with zero in case the ISR code might (implicitly) use it.
- RAM larger than 64 KiB is not supported by GCC for AVR targets. If you use inline assembler to read from locations outside the 16-bit address range and change one of the `RAMP` registers, you must reset it to zero after the access.

3.18.6.3 AVR Built-in Macros

GCC defines several built-in macros so that the user code can test for the presence or absence of features. Almost any of the following built-in macros are deduced from device capabilities and thus triggered by the '`-mmcu=`' command-line option.

For even more AVR-specific built-in macros see [AVR Named Address Spaces], page 466 and Section 6.60.10 [AVR Built-in Functions], page 669.

`__AVR_ARCH__`

Build-in macro that resolves to a decimal number that identifies the architecture and depends on the '`-mmcu=`*`mcu`*' option. Possible values are:

`2`, `25`, `3`, `31`, `35`, `4`, `5`, `51`, `6`

for *mcu*=`avr2`, `avr25`, `avr3`, `avr31`, `avr35`, `avr4`, `avr5`, `avr51`, `avr6`,

respectively and

`100`, `102`, `103`, `104`, `105`, `106`, `107`

for *mcu*=`avrtiny`, `avrxmega2`, `avrxmega3`, `avrxmega4`, `avrxmega5`, `avrxmega6`, `avrxmega7`, respectively. If *mcu* specifies a device, this built-in macro is set accordingly. For example, with '`-mmcu=atmega8`' the macro is defined to `4`.

`__AVR_`*`Device`*`__`

Setting '`-mmcu=`*`device`*' defines this built-in macro which reflects the device's name. For example, '`-mmcu=atmega8`' defines the built-in macro `__AVR_ATmega8__`, '`-mmcu=attiny261a`' defines `__AVR_ATtiny261A__`, etc.

The built-in macros' names follow the scheme `__AVR_Device__` where *Device* is the device name as from the AVR user manual. The difference between *Device* in the built-in macro and *device* in '`-mmcu=device`' is that the latter is always lowercase.

If *device* is not a device but only a core architecture like '`avr51`', this macro is not defined.

`__AVR_DEVICE_NAME__`
Setting '`-mmcu=device`' defines this built-in macro to the device's name. For example, with '`-mmcu=atmega8`' the macro is defined to `atmega8`.

If *device* is not a device but only a core architecture like '`avr51`', this macro is not defined.

`__AVR_XMEGA__`
The device / architecture belongs to the XMEGA family of devices.

`__AVR_HAVE_ELPM__`
The device has the `ELPM` instruction.

`__AVR_HAVE_ELPMX__`
The device has the `ELPM Rn,Z` and `ELPM Rn,Z+` instructions.

`__AVR_HAVE_MOVW__`
The device has the `MOVW` instruction to perform 16-bit register-register moves.

`__AVR_HAVE_LPMX__`
The device has the `LPM Rn,Z` and `LPM Rn,Z+` instructions.

`__AVR_HAVE_MUL__`
The device has a hardware multiplier.

`__AVR_HAVE_JMP_CALL__`
The device has the `JMP` and `CALL` instructions. This is the case for devices with more than 8 KiB of program memory.

`__AVR_HAVE_EIJMP_EICALL__`
`__AVR_3_BYTE_PC__`
The device has the `EIJMP` and `EICALL` instructions. This is the case for devices with more than 128 KiB of program memory. This also means that the program counter (PC) is 3 bytes wide.

`__AVR_2_BYTE_PC__`
The program counter (PC) is 2 bytes wide. This is the case for devices with up to 128 KiB of program memory.

`__AVR_HAVE_8BIT_SP__`
`__AVR_HAVE_16BIT_SP__`
The stack pointer (SP) register is treated as 8-bit respectively 16-bit register by the compiler. The definition of these macros is affected by '`-mtiny-stack`'.

`__AVR_HAVE_SPH__`
`__AVR_SP8__`
The device has the SPH (high part of stack pointer) special function register or has an 8-bit stack pointer, respectively. The definition of these macros is

affected by '-mmcu=' and in the cases of '-mmcu=avr2' and '-mmcu=avr25' also by '-msp8'.

__AVR_HAVE_RAMPD__
__AVR_HAVE_RAMPX__
__AVR_HAVE_RAMPY__
__AVR_HAVE_RAMPZ__
: The device has the RAMPD, RAMPX, RAMPY, RAMPZ special function register, respectively.

__NO_INTERRUPTS__
: This macro reflects the '-mno-interrupts' command-line option.

__AVR_ERRATA_SKIP__
__AVR_ERRATA_SKIP_JMP_CALL__
: Some AVR devices (AT90S8515, ATmega103) must not skip 32-bit instructions because of a hardware erratum. Skip instructions are SBRS, SBRC, SBIS, SBIC and CPSE. The second macro is only defined if __AVR_HAVE_JMP_CALL__ is also set.

__AVR_ISA_RMW__
: The device has Read-Modify-Write instructions (XCH, LAC, LAS and LAT).

__AVR_SFR_OFFSET__=*offset*
: Instructions that can address I/O special function registers directly like IN, OUT, SBI, etc. may use a different address as if addressed by an instruction to access RAM like LD or STS. This offset depends on the device architecture and has to be subtracted from the RAM address in order to get the respective I/O address.

__AVR_SHORT_CALLS__
: The '-mshort-calls' command line option is set.

__AVR_PM_BASE_ADDRESS__=*addr*
: Some devices support reading from flash memory by means of LD* instructions. The flash memory is seen in the data address space at an offset of __AVR_PM_BASE_ADDRESS__. If this macro is not defined, this feature is not available. If defined, the address space is linear and there is no need to put .rodata into RAM. This is handled by the default linker description file, and is currently available for avrtiny and avrxmega3. Even more convenient, there is no need to use address spaces like __flash or features like attribute progmem and pgm_read_*.

__WITH_AVRLIBC__
: The compiler is configured to be used together with AVR-Libc. See the '--with-avrlibc' configure option.

3.18.7 Blackfin Options

-mcpu=*cpu*[-*sirevision*]
: Specifies the name of the target Blackfin processor. Currently, *cpu* can be one of 'bf512', 'bf514', 'bf516', 'bf518', 'bf522', 'bf523', 'bf524', 'bf525', 'bf526', 'bf527', 'bf531', 'bf532', 'bf533', 'bf534', 'bf536', 'bf537', 'bf538',

'bf539', 'bf542', 'bf544', 'bf547', 'bf548', 'bf549', 'bf542m', 'bf544m', 'bf547m', 'bf548m', 'bf549m', 'bf561', 'bf592'.

The optional *sirevision* specifies the silicon revision of the target Blackfin processor. Any workarounds available for the targeted silicon revision are enabled. If *sirevision* is 'none', no workarounds are enabled. If *sirevision* is 'any', all workarounds for the targeted processor are enabled. The __SILICON_REVISION__ macro is defined to two hexadecimal digits representing the major and minor numbers in the silicon revision. If *sirevision* is 'none', the __SILICON_REVISION__ is not defined. If *sirevision* is 'any', the __SILICON_REVISION__ is defined to be 0xffff. If this optional *sirevision* is not used, GCC assumes the latest known silicon revision of the targeted Blackfin processor.

GCC defines a preprocessor macro for the specified *cpu*. For the 'bfin-elf' toolchain, this option causes the hardware BSP provided by libgloss to be linked in if '-msim' is not given.

Without this option, 'bf532' is used as the processor by default.

Note that support for 'bf561' is incomplete. For 'bf561', only the preprocessor macro is defined.

-msim
: Specifies that the program will be run on the simulator. This causes the simulator BSP provided by libgloss to be linked in. This option has effect only for 'bfin-elf' toolchain. Certain other options, such as '-mid-shared-library' and '-mfdpic', imply '-msim'.

-momit-leaf-frame-pointer
: Don't keep the frame pointer in a register for leaf functions. This avoids the instructions to save, set up and restore frame pointers and makes an extra register available in leaf functions.

-mspecld-anomaly
: When enabled, the compiler ensures that the generated code does not contain speculative loads after jump instructions. If this option is used, __WORKAROUND_SPECULATIVE_LOADS is defined.

-mno-specld-anomaly
: Don't generate extra code to prevent speculative loads from occurring.

-mcsync-anomaly
: When enabled, the compiler ensures that the generated code does not contain CSYNC or SSYNC instructions too soon after conditional branches. If this option is used, __WORKAROUND_SPECULATIVE_SYNCS is defined.

-mno-csync-anomaly
: Don't generate extra code to prevent CSYNC or SSYNC instructions from occurring too soon after a conditional branch.

-mlow64k
: When enabled, the compiler is free to take advantage of the knowledge that the entire program fits into the low 64k of memory.

-mno-low64k
: Assume that the program is arbitrarily large. This is the default.

`-mstack-check-l1`
: Do stack checking using information placed into L1 scratchpad memory by the uClinux kernel.

`-mid-shared-library`
: Generate code that supports shared libraries via the library ID method. This allows for execute in place and shared libraries in an environment without virtual memory management. This option implies '`-fPIC`'. With a '`bfin-elf`' target, this option implies '`-msim`'.

`-mno-id-shared-library`
: Generate code that doesn't assume ID-based shared libraries are being used. This is the default.

`-mleaf-id-shared-library`
: Generate code that supports shared libraries via the library ID method, but assumes that this library or executable won't link against any other ID shared libraries. That allows the compiler to use faster code for jumps and calls.

`-mno-leaf-id-shared-library`
: Do not assume that the code being compiled won't link against any ID shared libraries. Slower code is generated for jump and call insns.

`-mshared-library-id=n`
: Specifies the identification number of the ID-based shared library being compiled. Specifying a value of 0 generates more compact code; specifying other values forces the allocation of that number to the current library but is no more space- or time-efficient than omitting this option.

`-msep-data`
: Generate code that allows the data segment to be located in a different area of memory from the text segment. This allows for execute in place in an environment without virtual memory management by eliminating relocations against the text section.

`-mno-sep-data`
: Generate code that assumes that the data segment follows the text segment. This is the default.

`-mlong-calls`
`-mno-long-calls`
: Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function lies outside of the 24-bit addressing range of the offset-based version of subroutine call instruction.

  This feature is not enabled by default. Specifying '`-mno-long-calls`' restores the default behavior. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.

`-mfast-fp`
: Link with the fast floating-point library. This library relaxes some of the IEEE floating-point standard's rules for checking inputs against Not-a-Number (NAN), in the interest of performance.

`-minline-plt`
: Enable inlining of PLT entries in function calls to functions that are not known to bind locally. It has no effect without '`-mfdpic`'.

`-mmulticore`
: Build a standalone application for multicore Blackfin processors. This option causes proper start files and link scripts supporting multicore to be used, and defines the macro `__BFIN_MULTICORE`. It can only be used with '`-mcpu=bf561`[`-`*`sirevision`*]'.

  This option can be used with '`-mcorea`' or '`-mcoreb`', which selects the one-application-per-core programming model. Without '`-mcorea`' or '`-mcoreb`', the single-application/dual-core programming model is used. In this model, the main function of Core B should be named as `coreb_main`.

  If this option is not used, the single-core application programming model is used.

`-mcorea`
: Build a standalone application for Core A of BF561 when using the one-application-per-core programming model. Proper start files and link scripts are used to support Core A, and the macro `__BFIN_COREA` is defined. This option can only be used in conjunction with '`-mmulticore`'.

`-mcoreb`
: Build a standalone application for Core B of BF561 when using the one-application-per-core programming model. Proper start files and link scripts are used to support Core B, and the macro `__BFIN_COREB` is defined. When this option is used, `coreb_main` should be used instead of `main`. This option can only be used in conjunction with '`-mmulticore`'.

`-msdram`
: Build a standalone application for SDRAM. Proper start files and link scripts are used to put the application into SDRAM, and the macro `__BFIN_SDRAM` is defined. The loader should initialize SDRAM before loading the application.

`-micplb`
: Assume that ICPLBs are enabled at run time. This has an effect on certain anomaly workarounds. For Linux targets, the default is to assume ICPLBs are enabled; for standalone applications the default is off.

3.18.8 C6X Options

`-march=`*`name`*
: This specifies the name of the target architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. Permissible names are: '`c62x`', '`c64x`', '`c64x+`', '`c67x`', '`c67x+`', '`c674x`'.

`-mbig-endian`
: Generate code for a big-endian target.

`-mlittle-endian`
: Generate code for a little-endian target. This is the default.

`-msim`
: Choose startup files and linker script suitable for the simulator.

`-msdata=default`
: Put small global and static data in the `.neardata` section, which is pointed to by register `B14`. Put small uninitialized global and static data in the `.bss`

section, which is adjacent to the `.neardata` section. Put small read-only data into the `.rodata` section. The corresponding sections used for large pieces of data are `.fardata`, `.far` and `.const`.

`-msdata=all`
: Put all data, not just small objects, into the sections reserved for small data, and use addressing relative to the `B14` register to access them.

`-msdata=none`
: Make no use of the sections reserved for small data, and use absolute addresses to access all data. Put all initialized global and static data in the `.fardata` section, and all uninitialized data in the `.far` section. Put all constant data into the `.const` section.

3.18.9 CRIS Options

These options are defined specifically for the CRIS ports.

`-march=`*`architecture-type`*
`-mcpu=`*`architecture-type`*
: Generate code for the specified architecture. The choices for *architecture-type* are '`v3`', '`v8`' and '`v10`' for respectively ETRAX 4, ETRAX 100, and ETRAX 100 LX. Default is '`v0`' except for cris-axis-linux-gnu, where the default is '`v10`'.

`-mtune=`*`architecture-type`*
: Tune to *architecture-type* everything applicable about the generated code, except for the ABI and the set of available instructions. The choices for *architecture-type* are the same as for '`-march=`*`architecture-type`*'.

`-mmax-stack-frame=`*`n`*
: Warn when the stack frame of a function exceeds *n* bytes.

`-metrax4`
`-metrax100`
: The options '`-metrax4`' and '`-metrax100`' are synonyms for '`-march=v3`' and '`-march=v8`' respectively.

`-mmul-bug-workaround`
`-mno-mul-bug-workaround`
: Work around a bug in the `muls` and `mulu` instructions for CPU models where it applies. This option is active by default.

`-mpdebug` Enable CRIS-specific verbose debug-related information in the assembly code. This option also has the effect of turning off the '`#NO_APP`' formatted-code indicator to the assembler at the beginning of the assembly file.

`-mcc-init`
: Do not use condition-code results from previous instruction; always emit compare and test instructions before use of condition codes.

`-mno-side-effects`
: Do not emit instructions with side effects in addressing modes other than post-increment.

`-mstack-align`
`-mno-stack-align`
`-mdata-align`
`-mno-data-align`
`-mconst-align`
`-mno-const-align`
These options ('`no-`' options) arrange (eliminate arrangements) for the stack frame, individual data and constants to be aligned for the maximum single data access size for the chosen CPU model. The default is to arrange for 32-bit alignment. ABI details such as structure layout are not affected by these options.

`-m32-bit`
`-m16-bit`
`-m8-bit` Similar to the stack- data- and const-align options above, these options arrange for stack frame, writable data and constants to all be 32-bit, 16-bit or 8-bit aligned. The default is 32-bit alignment.

`-mno-prologue-epilogue`
`-mprologue-epilogue`
With '`-mno-prologue-epilogue`', the normal function prologue and epilogue which set up the stack frame are omitted and no return instructions or return sequences are generated in the code. Use this option only together with visual inspection of the compiled code: no warnings or errors are generated when call-saved registers must be saved, or storage for local variables needs to be allocated.

`-mno-gotplt`
`-mgotplt` With '`-fpic`' and '`-fPIC`', don't generate (do generate) instruction sequences that load addresses for functions from the PLT part of the GOT rather than (traditional on other architectures) calls to the PLT. The default is '`-mgotplt`'.

`-melf` Legacy no-op option only recognized with the cris-axis-elf and cris-axis-linux-gnu targets.

`-mlinux` Legacy no-op option only recognized with the cris-axis-linux-gnu target.

`-sim` This option, recognized for the cris-axis-elf, arranges to link with input-output functions from a simulator library. Code, initialized data and zero-initialized data are allocated consecutively.

`-sim2` Like '`-sim`', but pass linker options to locate initialized data at 0x40000000 and zero-initialized data at 0x80000000.

3.18.10 CR16 Options

These options are defined specifically for the CR16 ports.

`-mmac` Enable the use of multiply-accumulate instructions. Disabled by default.

`-mcr16cplus`
`-mcr16c` Generate code for CR16C or CR16C+ architecture. CR16C+ architecture is default.

`-msim` Links the library libsim.a which is in compatible with simulator. Applicable to ELF compiler only.

`-mint32` Choose integer type as 32-bit wide.

`-mbit-ops`
Generates `sbit/cbit` instructions for bit manipulations.

`-mdata-model=`*`model`*
Choose a data model. The choices for *model* are '`near`', '`far`' or '`medium`'. '`medium`' is default. However, '`far`' is not valid with '`-mcr16c`', as the CR16C architecture does not support the far data model.

3.18.11 C-SKY Options

GCC supports these options when compiling for C-SKY V2 processors.

`-march=`*`arch`*
Specify the C-SKY target architecture. Valid values for *arch* are: '`ck801`', '`ck802`', '`ck803`', '`ck807`', and '`ck810`'. The default is '`ck810`'.

`-mcpu=`*`cpu`*
Specify the C-SKY target processor. Valid values for *cpu* are: '`ck801`', '`ck801t`', '`ck802`', '`ck802t`', '`ck802j`', '`ck803`', '`ck803h`', '`ck803t`', '`ck803ht`', '`ck803f`', '`ck803fh`', '`ck803e`', '`ck803eh`', '`ck803et`', '`ck803eht`', '`ck803ef`', '`ck803efh`', '`ck803ft`', '`ck803eft`', '`ck803efht`', '`ck803r1`', '`ck803hr1`', '`ck803tr1`', '`ck803htr1`', '`ck803fr1`', '`ck803fhr1`', '`ck803er1`', '`ck803ehr1`', '`ck803etr1`', '`ck803ehtr1`', '`ck803efr1`', '`ck803efhr1`', '`ck803ftr1`', '`ck803eftr1`', '`ck803efhtr1`', '`ck803s`', '`ck803st`', '`ck803se`', '`ck803sf`', '`ck803sef`', '`ck803seft`', '`ck807e`', '`ck807ef`', '`ck807`', '`ck807f`', '`ck810e`', '`ck810et`', '`ck810ef`', '`ck810eft`', '`ck810`', '`ck810v`', '`ck810f`', '`ck810t`', '`ck810fv`', '`ck810tv`', '`ck810ft`', and '`ck810ftv`'.

`-mbig-endian`
`-EB`
`-mlittle-endian`
`-EL`
Select big- or little-endian code. The default is little-endian.

`-mhard-float`
`-msoft-float`
Select hardware or software floating-point implementations. The default is soft float.

`-mdouble-float`
`-mno-double-float`
When '`-mhard-float`' is in effect, enable generation of double-precision float instructions. This is the default except when compiling for CK803.

`-mfdivdu`
`-mno-fdivdu`
When '`-mhard-float`' is in effect, enable generation of `frecipd`, `fsqrtd`, and `fdivd` instructions. This is the default except when compiling for CK803.

`-mfpu=fpu`
Select the floating-point processor. This option can only be used with '`-mhard-float`'. Values for *fpu* are '`fpv2_sf`' (equivalent to '`-mno-double-float -mno-fdivdu`'), '`fpv2`' ('`-mdouble-float -mno-divdu`'), and '`fpv2_divd`' ('`-mdouble-float -mdivdu`').

`-melrw`
`-mno-elrw`
Enable the extended `lrw` instruction. This option defaults to on for CK801 and off otherwise.

`-mistack`
`-mno-istack`
Enable interrupt stack instructions; the default is off.

The '`-mistack`' option is required to handle the `interrupt` and `isr` function attributes (see Section 6.33.9 [C-SKY Function Attributes], page 506).

`-mmp` Enable multiprocessor instructions; the default is off.

`-mcp` Enable coprocessor instructions; the default is off.

`-mcache` Enable coprocessor instructions; the default is off.

`-msecurity`
Enable C-SKY security instructions; the default is off.

`-mtrust` Enable C-SKY trust instructions; the default is off.

`-mdsp`
`-medsp`
`-mvdsp` Enable C-SKY DSP, Enhanced DSP, or Vector DSP instructions, respectively. All of these options default to off.

`-mdiv`
`-mno-div` Generate divide instructions. Default is off.

`-msmart`
`-mno-smart`
Generate code for Smart Mode, using only registers numbered 0-7 to allow use of 16-bit instructions. This option is ignored for CK801 where this is the required behavior, and it defaults to on for CK802. For other targets, the default is off.

`-mhigh-registers`
`-mno-high-registers`
Generate code using the high registers numbered 16-31. This option is not supported on CK801, CK802, or CK803, and is enabled by default for other processors.

`-manchor`
`-mno-anchor`
Generate code using global anchor symbol addresses.

`-mpushpop`
`-mno-pushpop`
Generate code using `push` and `pop` instructions. This option defaults to on.

`-mmultiple-stld`
`-mstm`
`-mno-multiple-stld`
`-mno-stm` Generate code using `stm` and `ldm` instructions. This option isn't supported on CK801 but is enabled by default on other processors.

`-mconstpool`
`-mno-constpool`
Create constant pools in the compiler instead of deferring it to the assembler. This option is the default and required for correct code generation on CK801 and CK802, and is optional on other processors.

`-mstack-size`
`-mno-stack-size`
Emit `.stack_size` directives for each function in the assembly output. This option defaults to off.

`-mccrt`
`-mno-ccrt`
Generate code for the C-SKY compiler runtime instead of libgcc. This option defaults to off.

`-mbranch-cost=n`
Set the branch costs to roughly `n` instructions. The default is 1.

`-msched-prolog`
`-mno-sched-prolog`
Permit scheduling of function prologue and epilogue sequences. Using this option can result in code that is not compliant with the C-SKY V2 ABI prologue requirements and that cannot be debugged or backtraced. It is disabled by default.

3.18.12 Darwin Options

These options are defined for all architectures running the Darwin operating system.

FSF GCC on Darwin does not create "fat" object files; it creates an object file for the single architecture that GCC was built to target. Apple's GCC on Darwin does create "fat" files if multiple '`-arch`' options are used; it does so by running the compiler or linker multiple times and joining the results together with '`lipo`'.

The subtype of the file created (like '`ppc7400`' or '`ppc970`' or '`i686`') is determined by the flags that specify the ISA that GCC is targeting, like '`-mcpu`' or '`-march`'. The '`-force_cpusubtype_ALL`' option can be used to override this.

The Darwin tools vary in their behavior when presented with an ISA mismatch. The assembler, '`as`', only permits instructions to be used that are valid for the subtype of the file it is generating, so you cannot put 64-bit instructions in a '`ppc750`' object file. The linker for shared libraries, '`/usr/bin/libtool`', fails and prints an error if asked to create a shared library with a less restrictive subtype than its input files (for instance, trying to put a '`ppc970`' object file in a '`ppc7400`' library). The linker for executables, `ld`, quietly gives the executable the most restrictive subtype of any of its input files.

`-F`*dir* Add the framework directory *dir* to the head of the list of directories to be searched for header files. These directories are interleaved with those specified by '`-I`' options and are scanned in a left-to-right order.

A framework directory is a directory with frameworks in it. A framework is a directory with a '`Headers`' and/or '`PrivateHeaders`' directory contained directly in it that ends in '`.framework`'. The name of a framework is the name of this directory excluding the '`.framework`'. Headers associated with the framework are found in one of those two directories, with '`Headers`' being searched first. A subframework is a framework directory that is in a framework's '`Frameworks`' directory. Includes of subframework headers can only appear in a header of a framework that contains the subframework, or in a sibling subframework header. Two subframeworks are siblings if they occur in the same framework. A subframework should not have the same name as a framework; a warning is issued if this is violated. Currently a subframework cannot have subframeworks; in the future, the mechanism may be extended to support this. The standard frameworks can be found in '`/System/Library/Frameworks`' and '`/Library/Frameworks`'. An example include looks like `#include <Framework/header.h>`, where '`Framework`' denotes the name of the framework and '`header.h`' is found in the '`PrivateHeaders`' or '`Headers`' directory.

`-iframework`*dir*
Like '`-F`' except the directory is a treated as a system directory. The main difference between this '`-iframework`' and '`-F`' is that with '`-iframework`' the compiler does not warn about constructs contained within header files found via *dir*. This option is valid only for the C family of languages.

`-gused` Emit debugging information for symbols that are used. For stabs debugging format, this enables '`-feliminate-unused-debug-symbols`'. This is by default ON.

`-gfull` Emit debugging information for all symbols and types.

`-mmacosx-version-min=`*version*
The earliest version of MacOS X that this executable will run on is *version*. Typical values of *version* include `10.1`, `10.2`, and `10.3.9`.

If the compiler was built to use the system's headers by default, then the default for this option is the system version on which the compiler is running, otherwise the default is to make choices that are compatible with as many systems and code bases as possible.

`-mkernel` Enable kernel development mode. The '`-mkernel`' option sets '`-static`', '`-fno-common`', '`-fno-use-cxa-atexit`', '`-fno-exceptions`', '`-fno-non-call-exceptions`', '`-fapple-kext`', '`-fno-weak`' and '`-fno-rtti`' where applicable. This mode also sets '`-mno-altivec`', '`-msoft-float`', '`-fno-builtin`' and '`-mlong-branch`' for PowerPC targets.

`-mone-byte-bool`
: Override the defaults for `bool` so that `sizeof(bool)==1`. By default `sizeof(bool)` is `4` when compiling for Darwin/PowerPC and `1` when compiling for Darwin/x86, so this option has no effect on x86.

  Warning: The '`-mone-byte-bool`' switch causes GCC to generate code that is not binary compatible with code generated without that switch. Using this switch may require recompiling all other modules in a program, including system libraries. Use this switch to conform to a non-default data model.

`-mfix-and-continue`
`-ffix-and-continue`
`-findirect-data`
: Generate code suitable for fast turnaround development, such as to allow GDB to dynamically load '`.o`' files into already-running programs. '`-findirect-data`' and '`-ffix-and-continue`' are provided for backwards compatibility.

`-all_load`
: Loads all members of static archive libraries. See man ld(1) for more information.

`-arch_errors_fatal`
: Cause the errors having to do with files that have the wrong architecture to be fatal.

`-bind_at_load`
: Causes the output file to be marked such that the dynamic linker will bind all undefined references when the file is loaded or launched.

`-bundle` Produce a Mach-o bundle format file. See man ld(1) for more information.

`-bundle_loader` *`executable`*
: This option specifies the *executable* that will load the build output file being linked. See man ld(1) for more information.

`-dynamiclib`
: When passed this option, GCC produces a dynamic library instead of an executable when linking, using the Darwin '`libtool`' command.

`-force_cpusubtype_ALL`
: This causes GCC's output file to have the '`ALL`' subtype, instead of one controlled by the '`-mcpu`' or '`-march`' option.

-allowable_client *client_name*
-client_name
-compatibility_version
-current_version
-dead_strip
-dependency-file
-dylib_file
-dylinker_install_name
-dynamic
-exported_symbols_list
-filelist
-flat_namespace
-force_flat_namespace
-headerpad_max_install_names
-image_base
-init
-install_name
-keep_private_externs
-multi_module
-multiply_defined
-multiply_defined_unused
-noall_load
-no_dead_strip_inits_and_terms
-nofixprebinding
-nomultidefs
-noprebind
-noseglinkedit
-pagezero_size
-prebind
-prebind_all_twolevel_modules
-private_bundle
-read_only_relocs
-sectalign
-sectobjectsymbols
-whyload
-seg1addr
-sectcreate
-sectobjectsymbols
-sectorder
-segaddr
-segs_read_only_addr

`-segs_read_write_addr`
`-seg_addr_table`
`-seg_addr_table_filename`
`-seglinkedit`
`-segprot`
`-segs_read_only_addr`
`-segs_read_write_addr`
`-single_module`
`-static`
`-sub_library`
`-sub_umbrella`
`-twolevel_namespace`
`-umbrella`
`-undefined`
`-unexported_symbols_list`
`-weak_reference_mismatches`
`-whatsloaded`

These options are passed to the Darwin linker. The Darwin linker man page describes them in detail.

3.18.13 DEC Alpha Options

These '`-m`' options are defined for the DEC Alpha implementations:

`-mno-soft-float`
`-msoft-float`

Use (do not use) the hardware floating-point instructions for floating-point operations. When '`-msoft-float`' is specified, functions in '`libgcc.a`' are used to perform floating-point operations. Unless they are replaced by routines that emulate the floating-point operations, or compiled in such a way as to call such emulations routines, these routines issue floating-point operations. If you are compiling for an Alpha without floating-point operations, you must ensure that the library is built so as not to call them.

Note that Alpha implementations without floating-point operations are required to have floating-point registers.

`-mfp-reg`
`-mno-fp-regs`

Generate code that uses (does not use) the floating-point register set. '`-mno-fp-regs`' implies '`-msoft-float`'. If the floating-point register set is not used, floating-point operands are passed in integer registers as if they were integers and floating-point results are passed in `$0` instead of `$f0`. This is a non-standard calling sequence, so any function with a floating-point argument or return value called by code compiled with '`-mno-fp-regs`' must also be compiled with that option.

A typical use of this option is building a kernel that does not use, and hence need not save and restore, any floating-point registers.

`-mieee` The Alpha architecture implements floating-point hardware optimized for maximum performance. It is mostly compliant with the IEEE floating-point standard. However, for full compliance, software assistance is required. This option generates code fully IEEE-compliant code *except* that the *inexact-flag* is not maintained (see below). If this option is turned on, the preprocessor macro `_IEEE_FP` is defined during compilation. The resulting code is less efficient but is able to correctly support denormalized numbers and exceptional IEEE values such as not-a-number and plus/minus infinity. Other Alpha compilers call this option '`-ieee_with_no_inexact`'.

`-mieee-with-inexact`
This is like '`-mieee`' except the generated code also maintains the IEEE *inexact-flag*. Turning on this option causes the generated code to implement fully-compliant IEEE math. In addition to `_IEEE_FP`, `_IEEE_FP_EXACT` is defined as a preprocessor macro. On some Alpha implementations the resulting code may execute significantly slower than the code generated by default. Since there is very little code that depends on the *inexact-flag*, you should normally not specify this option. Other Alpha compilers call this option '`-ieee_with_inexact`'.

`-mfp-trap-mode=`*`trap-mode`*
This option controls what floating-point related traps are enabled. Other Alpha compilers call this option '`-fptm` *`trap-mode`*'. The trap mode can be set to one of four values:

- '`n`' This is the default (normal) setting. The only traps that are enabled are the ones that cannot be disabled in software (e.g., division by zero trap).
- '`u`' In addition to the traps enabled by '`n`', underflow traps are enabled as well.
- '`su`' Like '`u`', but the instructions are marked to be safe for software completion (see Alpha architecture manual for details).
- '`sui`' Like '`su`', but inexact traps are enabled as well.

`-mfp-rounding-mode=`*`rounding-mode`*
Selects the IEEE rounding mode. Other Alpha compilers call this option '`-fprm` *`rounding-mode`*'. The *rounding-mode* can be one of:

- '`n`' Normal IEEE rounding mode. Floating-point numbers are rounded towards the nearest machine number or towards the even machine number in case of a tie.
- '`m`' Round towards minus infinity.
- '`c`' Chopped rounding mode. Floating-point numbers are rounded towards zero.
- '`d`' Dynamic rounding mode. A field in the floating-point control register (*fpcr*, see Alpha architecture reference manual) controls the rounding mode in effect. The C library initializes this register for rounding towards plus infinity. Thus, unless your program modifies the *fpcr*, '`d`' corresponds to round towards plus infinity.

`-mtrap-precision=`*`trap-precision`*

In the Alpha architecture, floating-point traps are imprecise. This means without software assistance it is impossible to recover from a floating trap and program execution normally needs to be terminated. GCC can generate code that can assist operating system trap handlers in determining the exact location that caused a floating-point trap. Depending on the requirements of an application, different levels of precisions can be selected:

'`p`' Program precision. This option is the default and means a trap handler can only identify which program caused a floating-point exception.

'`f`' Function precision. The trap handler can determine the function that caused a floating-point exception.

'`i`' Instruction precision. The trap handler can determine the exact instruction that caused a floating-point exception.

Other Alpha compilers provide the equivalent options called '`-scope_safe`' and '`-resumption_safe`'.

`-mieee-conformant`

This option marks the generated code as IEEE conformant. You must not use this option unless you also specify '`-mtrap-precision=i`' and either '`-mfp-trap-mode=su`' or '`-mfp-trap-mode=sui`'. Its only effect is to emit the line '`.eflag 48`' in the function prologue of the generated assembly file.

`-mbuild-constants`

Normally GCC examines a 32- or 64-bit integer constant to see if it can construct it from smaller constants in two or three instructions. If it cannot, it outputs the constant as a literal and generates code to load it from the data segment at run time.

Use this option to require GCC to construct *all* integer constants using code, even if it takes more instructions (the maximum is six).

You typically use this option to build a shared library dynamic loader. Itself a shared library, it must relocate itself in memory before it can find the variables and constants in its own data segment.

`-mbwx`
`-mno-bwx`
`-mcix`
`-mno-cix`
`-mfix`
`-mno-fix`
`-mmax`
`-mno-max` Indicate whether GCC should generate code to use the optional BWX, CIX, FIX and MAX instruction sets. The default is to use the instruction sets supported by the CPU type specified via '`-mcpu=`' option or that of the CPU on which GCC was built if none is specified.

`-mfloat-vax`
`-mfloat-ieee`
: Generate code that uses (does not use) VAX F and G floating-point arithmetic instead of IEEE single and double precision.

`-mexplicit-relocs`
`-mno-explicit-relocs`
: Older Alpha assemblers provided no way to generate symbol relocations except via assembler macros. Use of these macros does not allow optimal instruction scheduling. GNU binutils as of version 2.12 supports a new syntax that allows the compiler to explicitly mark which relocations should apply to which instructions. This option is mostly useful for debugging, as GCC detects the capabilities of the assembler when it is built and sets the default accordingly.

`-msmall-data`
`-mlarge-data`
: When '`-mexplicit-relocs`' is in effect, static data is accessed via *gp-relative* relocations. When '`-msmall-data`' is used, objects 8 bytes long or smaller are placed in a *small data area* (the `.sdata` and `.sbss` sections) and are accessed via 16-bit relocations off of the `$gp` register. This limits the size of the small data area to 64KB, but allows the variables to be directly accessed via a single instruction.

  The default is '`-mlarge-data`'. With this option the data area is limited to just below 2GB. Programs that require more than 2GB of data must use `malloc` or `mmap` to allocate the data in the heap instead of in the program's data segment.

  When generating code for shared libraries, '`-fpic`' implies '`-msmall-data`' and '`-fPIC`' implies '`-mlarge-data`'.

`-msmall-text`
`-mlarge-text`
: When '`-msmall-text`' is used, the compiler assumes that the code of the entire program (or shared library) fits in 4MB, and is thus reachable with a branch instruction. When '`-msmall-data`' is used, the compiler can assume that all local symbols share the same `$gp` value, and thus reduce the number of instructions required for a function call from 4 to 1.

  The default is '`-mlarge-text`'.

`-mcpu=`*cpu_type*
: Set the instruction set and instruction scheduling parameters for machine type *cpu_type*. You can specify either the '`EV`' style name or the corresponding chip number. GCC supports scheduling parameters for the EV4, EV5 and EV6 family of processors and chooses the default values for the instruction set from the processor you specify. If you do not specify a processor type, GCC defaults to the processor on which the compiler was built.

  Supported values for *cpu_type* are

  '`ev4`'
  '`ev45`'
  '`21064`' Schedules as an EV4 and has no instruction set extensions.

'ev5'
'21164' Schedules as an EV5 and has no instruction set extensions.

'ev56'
'21164a' Schedules as an EV5 and supports the BWX extension.

'pca56'
'21164pc'
'21164PC' Schedules as an EV5 and supports the BWX and MAX extensions.

'ev6'
'21264' Schedules as an EV6 and supports the BWX, FIX, and MAX extensions.

'ev67'
'21264a' Schedules as an EV6 and supports the BWX, CIX, FIX, and MAX extensions.

Native toolchains also support the value '`native`', which selects the best architecture option for the host processor. '`-mcpu=native`' has no effect if GCC does not recognize the processor.

`-mtune=`*`cpu_type`*
Set only the instruction scheduling parameters for machine type *cpu_type*. The instruction set is not changed.

Native toolchains also support the value '`native`', which selects the best architecture option for the host processor. '`-mtune=native`' has no effect if GCC does not recognize the processor.

`-mmemory-latency=`*`time`*
Sets the latency the scheduler should assume for typical memory references as seen by the application. This number is highly dependent on the memory access patterns used by the application and the size of the external cache on the machine.

Valid options for *time* are

'*number*' A decimal number representing clock cycles.

'L1'
'L2'
'L3'
'main' The compiler contains estimates of the number of clock cycles for "typical" EV4 & EV5 hardware for the Level 1, 2 & 3 caches (also called Dcache, Scache, and Bcache), as well as to main memory. Note that L3 is only valid for EV5.

3.18.14 FR30 Options

These options are defined specifically for the FR30 port.

`-msmall-model`
Use the small address space model. This can produce smaller code, but it does assume that all symbolic values and addresses fit into a 20-bit range.

`-mno-lsim`
: Assume that runtime support has been provided and so there is no need to include the simulator library ('`libsim.a`') on the linker command line.

3.18.15 FT32 Options

These options are defined specifically for the FT32 port.

`-msim`
: Specifies that the program will be run on the simulator. This causes an alternate runtime startup and library to be linked. You must not use this option when generating programs that will run on real hardware; you must provide your own runtime library for whatever I/O functions are needed.

`-mlra`
: Enable Local Register Allocation. This is still experimental for FT32, so by default the compiler uses standard reload.

`-mnodiv`
: Do not use div and mod instructions.

`-mft32b`
: Enable use of the extended instructions of the FT32B processor.

`-mcompress`
: Compress all code using the Ft32B code compression scheme.

`-mnopm`
: Do not generate code that reads program memory.

3.18.16 FRV Options

`-mgpr-32`
: Only use the first 32 general-purpose registers.

`-mgpr-64`
: Use all 64 general-purpose registers.

`-mfpr-32`
: Use only the first 32 floating-point registers.

`-mfpr-64`
: Use all 64 floating-point registers.

`-mhard-float`
: Use hardware instructions for floating-point operations.

`-msoft-float`
: Use library routines for floating-point operations.

`-malloc-cc`
: Dynamically allocate condition code registers.

`-mfixed-cc`
: Do not try to dynamically allocate condition code registers, only use `icc0` and `fcc0`.

`-mdword`
: Change ABI to use double word insns.

`-mno-dword`
: Do not use double word instructions.

`-mdouble`
: Use floating-point double instructions.

`-mno-double`
: Do not use floating-point double instructions.

`-mmedia`
: Use media instructions.

`-mno-media`
: Do not use media instructions.

`-mmuladd`
: Use multiply and add/subtract instructions.

`-mno-muladd`
: Do not use multiply and add/subtract instructions.

`-mfdpic`
: Select the FDPIC ABI, which uses function descriptors to represent pointers to functions. Without any PIC/PIE-related options, it implies '`-fPIE`'. With '`-fpic`' or '`-fpie`', it assumes GOT entries and small data are within a 12-bit range from the GOT base address; with '`-fPIC`' or '`-fPIE`', GOT offsets are computed with 32 bits. With a '`bfin-elf`' target, this option implies '`-msim`'.

`-minline-plt`
: Enable inlining of PLT entries in function calls to functions that are not known to bind locally. It has no effect without '`-mfdpic`'. It's enabled by default if optimizing for speed and compiling for shared libraries (i.e., '`-fPIC`' or '`-fpic`'), or when an optimization option such as '`-O3`' or above is present in the command line.

`-mTLS`
: Assume a large TLS segment when generating thread-local code.

`-mtls`
: Do not assume a large TLS segment when generating thread-local code.

`-mgprel-ro`
: Enable the use of `GPREL` relocations in the FDPIC ABI for data that is known to be in read-only sections. It's enabled by default, except for '`-fpic`' or '`-fpie`': even though it may help make the global offset table smaller, it trades 1 instruction for 4. With '`-fPIC`' or '`-fPIE`', it trades 3 instructions for 4, one of which may be shared by multiple symbols, and it avoids the need for a GOT entry for the referenced symbol, so it's more likely to be a win. If it is not, '`-mno-gprel-ro`' can be used to disable it.

`-multilib-library-pic`
: Link with the (library, not FD) pic libraries. It's implied by '`-mlibrary-pic`', as well as by '`-fPIC`' and '`-fpic`' without '`-mfdpic`'. You should never have to use it explicitly.

`-mlinked-fp`
: Follow the EABI requirement of always creating a frame pointer whenever a stack frame is allocated. This option is enabled by default and can be disabled with '`-mno-linked-fp`'.

`-mlong-calls`
: Use indirect addressing to call functions outside the current compilation unit. This allows the functions to be placed anywhere within the 32-bit address space.

`-malign-labels`
: Try to align labels to an 8-byte boundary by inserting NOPs into the previous packet. This option only has an effect when VLIW packing is enabled. It doesn't create new packets; it merely adds NOPs to existing ones.

`-mlibrary-pic`
: Generate position-independent EABI code.

`-macc-4`
: Use only the first four media accumulator registers.

`-macc-8`
: Use all eight media accumulator registers.

`-mpack`
: Pack VLIW instructions.

`-mno-pack`
: Do not pack VLIW instructions.

`-mno-eflags`
: Do not mark ABI switches in e_flags.

`-mcond-move`
: Enable the use of conditional-move instructions (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-cond-move`
: Disable the use of conditional-move instructions.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mscc`
: Enable the use of conditional set instructions (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-scc`
: Disable the use of conditional set instructions.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mcond-exec`
: Enable the use of conditional execution (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-cond-exec`
: Disable the use of conditional execution.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mvliw-branch`
: Run a pass to pack branches into VLIW instructions (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-vliw-branch`
: Do not run a pass to pack branches into VLIW instructions.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mmulti-cond-exec`
: Enable optimization of `&&` and `||` in conditional execution (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-multi-cond-exec`
: Disable optimization of `&&` and `||` in conditional execution.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mnested-cond-exec`
: Enable nested conditional execution optimizations (default).

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-mno-nested-cond-exec`
: Disable nested conditional execution optimizations.

  This switch is mainly for debugging the compiler and will likely be removed in a future version.

`-moptimize-membar`
: This switch removes redundant `membar` instructions from the compiler-generated code. It is enabled by default.

`-mno-optimize-membar`
: This switch disables the automatic removal of redundant `membar` instructions from the generated code.

`-mtomcat-stats`
: Cause gas to print out tomcat statistics.

`-mcpu=cpu`
: Select the processor type for which to generate code. Possible values are '`frv`', '`fr550`', '`tomcat`', '`fr500`', '`fr450`', '`fr405`', '`fr400`', '`fr300`' and '`simple`'.

3.18.17 GNU/Linux Options

These '`-m`' options are defined for GNU/Linux targets:

`-mglibc`
: Use the GNU C library. This is the default except on '`*-*-linux-*uclibc*`', '`*-*-linux-*musl*`' and '`*-*-linux-*android*`' targets.

`-muclibc`
: Use uClibc C library. This is the default on '`*-*-linux-*uclibc*`' targets.

`-mmusl`
: Use the musl C library. This is the default on '`*-*-linux-*musl*`' targets.

`-mbionic`
: Use Bionic C library. This is the default on '`*-*-linux-*android*`' targets.

`-mandroid`
: Compile code compatible with Android platform. This is the default on '`*-*-linux-*android*`' targets.

  When compiling, this option enables '`-mbionic`', '`-fPIC`', '`-fno-exceptions`' and '`-fno-rtti`' by default. When linking, this option makes the GCC driver pass Android-specific options to the linker. Finally, this option causes the preprocessor macro `__ANDROID__` to be defined.

`-tno-android-cc`
: Disable compilation effects of '`-mandroid`', i.e., do not enable '`-mbionic`', '`-fPIC`', '`-fno-exceptions`' and '`-fno-rtti`' by default.

`-tno-android-ld`
: Disable linking effects of '`-mandroid`', i.e., pass standard Linux linking options to the linker.

3.18.18 H8/300 Options

These '`-m`' options are defined for the H8/300 implementations:

`-mrelax`
: Shorten some address references at link time, when possible; uses the linker option '`-relax`'. See Section "`ld` and the H8/300" in *Using ld*, for a fuller description.

`-mh`
: Generate code for the H8/300H.

`-ms`
: Generate code for the H8S.

`-mn`
: Generate code for the H8S and H8/300H in the normal mode. This switch must be used either with '`-mh`' or '`-ms`'.

`-ms2600`
: Generate code for the H8S/2600. This switch must be used with '`-ms`'.

`-mexr`
: Extended registers are stored on stack before execution of function with monitor attribute. Default option is '`-mexr`'. This option is valid only for H8S targets.

`-mno-exr`
: Extended registers are not stored on stack before execution of function with monitor attribute. Default option is '`-mno-exr`'. This option is valid only for H8S targets.

`-mint32` Make `int` data 32 bits by default.

`-malign-300`
On the H8/300H and H8S, use the same alignment rules as for the H8/300. The default for the H8/300H and H8S is to align longs and floats on 4-byte boundaries. '`-malign-300`' causes them to be aligned on 2-byte boundaries. This option has no effect on the H8/300.

3.18.19 HPPA Options

These '`-m`' options are defined for the HPPA family of computers:

`-march=`*`architecture-type`*
Generate code for the specified architecture. The choices for *architecture-type* are '`1.0`' for PA 1.0, '`1.1`' for PA 1.1, and '`2.0`' for PA 2.0 processors. Refer to '`/usr/lib/sched.models`' on an HP-UX system to determine the proper architecture option for your machine. Code compiled for lower numbered architectures runs on higher numbered architectures, but not the other way around.

`-mpa-risc-1-0`
`-mpa-risc-1-1`
`-mpa-risc-2-0`
Synonyms for '`-march=1.0`', '`-march=1.1`', and '`-march=2.0`' respectively.

`-mcaller-copies`
The caller copies function arguments passed by hidden reference. This option should be used with care as it is not compatible with the default 32-bit runtime. However, only aggregates larger than eight bytes are passed by hidden reference and the option provides better compatibility with OpenMP.

`-mjump-in-delay`
This option is ignored and provided for compatibility purposes only.

`-mdisable-fpregs`
Prevent floating-point registers from being used in any manner. This is necessary for compiling kernels that perform lazy context switching of floating-point registers. If you use this option and attempt to perform floating-point operations, the compiler aborts.

`-mdisable-indexing`
Prevent the compiler from using indexing address modes. This avoids some rather obscure problems when compiling MIG generated code under MACH.

`-mno-space-regs`
Generate code that assumes the target has no space registers. This allows GCC to generate faster indirect calls and use unscaled index address modes.

Such code is suitable for level 0 PA systems and kernels.

`-mfast-indirect-calls`
Generate code that assumes calls never cross space boundaries. This allows GCC to emit code that performs faster indirect calls.

This option does not work in the presence of shared libraries or nested functions.

`-mfixed-range=register-range`
: Generate code treating the given register range as fixed registers. A fixed register is one that the register allocator cannot use. This is useful when compiling kernel code. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma.

`-mlong-load-store`
: Generate 3-instruction load and store sequences as sometimes required by the HP-UX 10 linker. This is equivalent to the '`+k`' option to the HP compilers.

`-mportable-runtime`
: Use the portable calling conventions proposed by HP for ELF systems.

`-mgas`
: Enable the use of assembler directives only GAS understands.

`-mschedule=cpu-type`
: Schedule code according to the constraints for the machine type *cpu-type*. The choices for *cpu-type* are '`700`' '`7100`', '`7100LC`', '`7200`', '`7300`' and '`8000`'. Refer to '`/usr/lib/sched.models`' on an HP-UX system to determine the proper scheduling option for your machine. The default scheduling is '`8000`'.

`-mlinker-opt`
: Enable the optimization pass in the HP-UX linker. Note this makes symbolic debugging impossible. It also triggers a bug in the HP-UX 8 and HP-UX 9 linkers in which they give bogus error messages when linking some programs.

`-msoft-float`
: Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all HPPA targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

  '`-msoft-float`' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile '`libgcc.a`', the library that comes with GCC, with '`-msoft-float`' in order for this to work.

`-msio`
: Generate the predefine, `_SIO`, for server IO. The default is '`-mwsio`'. This generates the predefines, `__hp9000s700`, `__hp9000s700__` and `_WSIO`, for workstation IO. These options are available under HP-UX and HI-UX.

`-mgnu-ld`
: Use options specific to GNU `ld`. This passes '`-shared`' to `ld` when building a shared library. It is the default when GCC is configured, explicitly or implicitly, with the GNU linker. This option does not affect which `ld` is called; it only changes what parameters are passed to that `ld`. The `ld` that is called is determined by the '`--with-ld`' configure option, GCC's program search path, and finally by the user's `PATH`. The linker used by GCC can be printed using '`which 'gcc -print-prog-name=ld'`'. This option is only available on the 64-bit HP-UX GCC, i.e. configured with '`hppa*64*-*-hpux*`'.

`-mhp-ld`
: Use options specific to HP `ld`. This passes '`-b`' to `ld` when building a shared library and passes '`+Accept TypeMismatch`' to `ld` on all links. It is the default

when GCC is configured, explicitly or implicitly, with the HP linker. This option does not affect which `ld` is called; it only changes what parameters are passed to that `ld`. The `ld` that is called is determined by the '`--with-ld`' configure option, GCC's program search path, and finally by the user's `PATH`. The linker used by GCC can be printed using '`which 'gcc -print-prog-name=ld'`'. This option is only available on the 64-bit HP-UX GCC, i.e. configured with '`hppa*64*-*-hpux*`'.

`-mlong-calls`

Generate code that uses long call sequences. This ensures that a call is always able to reach linker generated stubs. The default is to generate long calls only when the distance from the call site to the beginning of the function or translation unit, as the case may be, exceeds a predefined limit set by the branch type being used. The limits for normal calls are 7,600,000 and 240,000 bytes, respectively for the PA 2.0 and PA 1.X architectures. Sibcalls are always limited at 240,000 bytes.

Distances are measured from the beginning of functions when using the '`-ffunction-sections`' option, or when using the '`-mgas`' and '`-mno-portable-runtime`' options together under HP-UX with the SOM linker.

It is normally not desirable to use this option as it degrades performance. However, it may be useful in large applications, particularly when partial linking is used to build the application.

The types of long calls used depends on the capabilities of the assembler and linker, and the type of code being generated. The impact on systems that support long absolute calls, and long pic symbol-difference or pc-relative calls should be relatively small. However, an indirect call is used on 32-bit ELF systems in pic code and it is quite long.

`-munix=`*`unix-std`*

Generate compiler predefines and select a startfile for the specified UNIX standard. The choices for *unix-std* are '`93`', '`95`' and '`98`'. '`93`' is supported on all HP-UX versions. '`95`' is available on HP-UX 10.10 and later. '`98`' is available on HP-UX 11.11 and later. The default values are '`93`' for HP-UX 10.00, '`95`' for HP-UX 10.10 though to 11.00, and '`98`' for HP-UX 11.11 and later.

'`-munix=93`' provides the same predefines as GCC 3.3 and 3.4. '`-munix=95`' provides additional predefines for `XOPEN_UNIX` and `_XOPEN_SOURCE_EXTENDED`, and the startfile '`unix95.o`'. '`-munix=98`' provides additional predefines for `_XOPEN_UNIX`, `_XOPEN_SOURCE_EXTENDED`, `_INCLUDE__STDC_A1_SOURCE` and `_INCLUDE_XOPEN_SOURCE_500`, and the startfile '`unix98.o`'.

It is *important* to note that this option changes the interfaces for various library routines. It also affects the operational behavior of the C library. Thus, *extreme* care is needed in using this option.

Library code that is intended to operate with more than one UNIX standard must test, set and restore the variable `__xpg4_extended_mask` as appropriate. Most GNU software doesn't provide this capability.

`-nolibdld`
: Suppress the generation of link options to search libdld.sl when the '`-static`' option is specified on HP-UX 10 and later.

`-static`
: The HP-UX implementation of setlocale in libc has a dependency on libdld.sl. There isn't an archive version of libdld.sl. Thus, when the '`-static`' option is specified, special link options are needed to resolve this dependency.

  On HP-UX 10 and later, the GCC driver adds the necessary options to link with libdld.sl when the '`-static`' option is specified. This causes the resulting binary to be dynamic. On the 64-bit port, the linkers generate dynamic binaries by default in any case. The '`-nolibdld`' option can be used to prevent the GCC driver from adding these link options.

`-threads`
: Add support for multithreading with the *dce thread* library under HP-UX. This option sets flags for both the preprocessor and linker.

3.18.20 IA-64 Options

These are the '`-m`' options defined for the Intel IA-64 architecture.

`-mbig-endian`
: Generate code for a big-endian target. This is the default for HP-UX.

`-mlittle-endian`
: Generate code for a little-endian target. This is the default for AIX5 and GNU/Linux.

`-mgnu-as`
`-mno-gnu-as`
: Generate (or don't) code for the GNU assembler. This is the default.

`-mgnu-ld`
`-mno-gnu-ld`
: Generate (or don't) code for the GNU linker. This is the default.

`-mno-pic`
: Generate code that does not use a global pointer register. The result is not position independent code, and violates the IA-64 ABI.

`-mvolatile-asm-stop`
`-mno-volatile-asm-stop`
: Generate (or don't) a stop bit immediately before and after volatile asm statements.

`-mregister-names`
`-mno-register-names`
: Generate (or don't) '`in`', '`loc`', and '`out`' register names for the stacked registers. This may make assembler output more readable.

`-mno-sdata`
`-msdata`
: Disable (or enable) optimizations that use the small data section. This may be useful for working around optimizer bugs.

`-mconstant-gp`
: Generate code that uses a single constant global pointer value. This is useful when compiling kernel code.

`-mauto-pic`
: Generate code that is self-relocatable. This implies '`-mconstant-gp`'. This is useful when compiling firmware code.

`-minline-float-divide-min-latency`
: Generate code for inline divides of floating-point values using the minimum latency algorithm.

`-minline-float-divide-max-throughput`
: Generate code for inline divides of floating-point values using the maximum throughput algorithm.

`-mno-inline-float-divide`
: Do not generate inline code for divides of floating-point values.

`-minline-int-divide-min-latency`
: Generate code for inline divides of integer values using the minimum latency algorithm.

`-minline-int-divide-max-throughput`
: Generate code for inline divides of integer values using the maximum throughput algorithm.

`-mno-inline-int-divide`
: Do not generate inline code for divides of integer values.

`-minline-sqrt-min-latency`
: Generate code for inline square roots using the minimum latency algorithm.

`-minline-sqrt-max-throughput`
: Generate code for inline square roots using the maximum throughput algorithm.

`-mno-inline-sqrt`
: Do not generate inline code for `sqrt`.

`-mfused-madd`
`-mno-fused-madd`
: Do (don't) generate code that uses the fused multiply/add or multiply/subtract instructions. The default is to use these instructions.

`-mno-dwarf2-asm`
`-mdwarf2-asm`
: Don't (or do) generate assembler code for the DWARF line number debugging info. This may be useful when not using the GNU assembler.

`-mearly-stop-bits`
`-mno-early-stop-bits`
: Allow stop bits to be placed earlier than immediately preceding the instruction that triggered the stop bit. This can improve instruction scheduling, but does not always do so.

`-mfixed-range=`*`register-range`*
: Generate code treating the given register range as fixed registers. A fixed register is one that the register allocator cannot use. This is useful when compiling

kernel code. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma.

`-mtls-size=`*`tls-size`*
Specify bit size of immediate TLS offsets. Valid values are 14, 22, and 64.

`-mtune=`*`cpu-type`*
Tune the instruction scheduling for a particular CPU, Valid values are '`itanium`', '`itanium1`', '`merced`', '`itanium2`', and '`mckinley`'.

`-milp32`
`-mlp64` Generate code for a 32-bit or 64-bit environment. The 32-bit environment sets int, long and pointer to 32 bits. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits. These are HP-UX specific flags.

`-mno-sched-br-data-spec`
`-msched-br-data-spec`
(Dis/En)able data speculative scheduling before reload. This results in generation of `ld.a` instructions and the corresponding check instructions (`ld.c` / `chk.a`). The default setting is disabled.

`-msched-ar-data-spec`
`-mno-sched-ar-data-spec`
(En/Dis)able data speculative scheduling after reload. This results in generation of `ld.a` instructions and the corresponding check instructions (`ld.c` / `chk.a`). The default setting is enabled.

`-mno-sched-control-spec`
`-msched-control-spec`
(Dis/En)able control speculative scheduling. This feature is available only during region scheduling (i.e. before reload). This results in generation of the `ld.s` instructions and the corresponding check instructions `chk.s`. The default setting is disabled.

`-msched-br-in-data-spec`
`-mno-sched-br-in-data-spec`
(En/Dis)able speculative scheduling of the instructions that are dependent on the data speculative loads before reload. This is effective only with '`-msched-br-data-spec`' enabled. The default setting is enabled.

`-msched-ar-in-data-spec`
`-mno-sched-ar-in-data-spec`
(En/Dis)able speculative scheduling of the instructions that are dependent on the data speculative loads after reload. This is effective only with '`-msched-ar-data-spec`' enabled. The default setting is enabled.

`-msched-in-control-spec`
`-mno-sched-in-control-spec`
(En/Dis)able speculative scheduling of the instructions that are dependent on the control speculative loads. This is effective only with '`-msched-control-spec`' enabled. The default setting is enabled.

`-mno-sched-prefer-non-data-spec-insns`
`-msched-prefer-non-data-spec-insns`
If enabled, data-speculative instructions are chosen for schedule only if there are no other choices at the moment. This makes the use of the data speculation much more conservative. The default setting is disabled.

`-mno-sched-prefer-non-control-spec-insns`
`-msched-prefer-non-control-spec-insns`
If enabled, control-speculative instructions are chosen for schedule only if there are no other choices at the moment. This makes the use of the control speculation much more conservative. The default setting is disabled.

`-mno-sched-count-spec-in-critical-path`
`-msched-count-spec-in-critical-path`
If enabled, speculative dependencies are considered during computation of the instructions priorities. This makes the use of the speculation a bit more conservative. The default setting is disabled.

`-msched-spec-ldc`
Use a simple data speculation check. This option is on by default.

`-msched-control-spec-ldc`
Use a simple check for control speculation. This option is on by default.

`-msched-stop-bits-after-every-cycle`
Place a stop bit after every cycle when scheduling. This option is on by default.

`-msched-fp-mem-deps-zero-cost`
Assume that floating-point stores and loads are not likely to cause a conflict when placed into the same instruction group. This option is disabled by default.

`-msel-sched-dont-check-control-spec`
Generate checks for control speculation in selective scheduling. This flag is disabled by default.

`-msched-max-memory-insns=`*`max-insns`*
Limit on the number of memory insns per instruction group, giving lower priority to subsequent memory insns attempting to schedule in the same instruction group. Frequently useful to prevent cache bank conflicts. The default value is 1.

`-msched-max-memory-insns-hard-limit`
Makes the limit specified by '`msched-max-memory-insns`' a hard limit, disallowing more than that number in an instruction group. Otherwise, the limit is "soft", meaning that non-memory operations are preferred when the limit is reached, but memory operations may still be scheduled.

3.18.21 LM32 Options

These '`-m`' options are defined for the LatticeMico32 architecture:

`-mbarrel-shift-enabled`
Enable barrel-shift instructions.

`-mdivide-enabled`
Enable divide and modulus instructions.

`-mmultiply-enabled`
Enable multiply instructions.

`-msign-extend-enabled`
Enable sign extend instructions.

`-muser-enabled`
Enable user-defined instructions.

3.18.22 M32C Options

`-mcpu=name`
Select the CPU for which code is generated. *name* may be one of '`r8c`' for the R8C/Tiny series, '`m16c`' for the M16C (up to /60) series, '`m32cm`' for the M16C/80 series, or '`m32c`' for the M32C/80 series.

`-msim` Specifies that the program will be run on the simulator. This causes an alternate runtime library to be linked in which supports, for example, file I/O. You must not use this option when generating programs that will run on real hardware; you must provide your own runtime library for whatever I/O functions are needed.

`-memregs=number`
Specifies the number of memory-based pseudo-registers GCC uses during code generation. These pseudo-registers are used like real registers, so there is a tradeoff between GCC's ability to fit the code into available registers, and the performance penalty of using memory instead of registers. Note that all modules in a program must be compiled with the same value for this option. Because of that, you must not use this option with GCC's default runtime libraries.

3.18.23 M32R/D Options

These '`-m`' options are defined for Renesas M32R/D architectures:

`-m32r2` Generate code for the M32R/2.

`-m32rx` Generate code for the M32R/X.

`-m32r` Generate code for the M32R. This is the default.

`-mmodel=small`
Assume all objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and assume all subroutines are reachable with the `bl` instruction. This is the default.

The addressability of a particular object can be set with the `model` attribute.

`-mmodel=medium`
Assume objects may be anywhere in the 32-bit address space (the compiler generates `seth/add3` instructions to load their addresses), and assume all subroutines are reachable with the `bl` instruction.

`-mmodel=large`
: Assume objects may be anywhere in the 32-bit address space (the compiler generates `seth/add3` instructions to load their addresses), and assume subroutines may not be reachable with the `bl` instruction (the compiler generates the much slower `seth/add3/jl` instruction sequence).

`-msdata=none`
: Disable use of the small data area. Variables are put into one of `.data`, `.bss`, or `.rodata` (unless the `section` attribute has been specified). This is the default.

  The small data area consists of sections `.sdata` and `.sbss`. Objects may be explicitly put in the small data area with the `section` attribute using one of these sections.

`-msdata=sdata`
: Put small global and static data in the small data area, but do not generate special code to reference them.

`-msdata=use`
: Put small global and static data in the small data area, and generate special instructions to reference them.

`-G` *num*
: Put global and static objects less than or equal to *num* bytes into the small data or BSS sections instead of the normal data or BSS sections. The default value of *num* is 8. The '`-msdata`' option must be set to one of '`sdata`' or '`use`' for this option to have any effect.

  All modules should be compiled with the same '`-G` *num*' value. Compiling with different values of *num* may or may not work; if it doesn't the linker gives an error message—incorrect code is not generated.

`-mdebug`
: Makes the M32R-specific code in the compiler display some statistics that might help in debugging programs.

`-malign-loops`
: Align all loops to a 32-byte boundary.

`-mno-align-loops`
: Do not enforce a 32-byte alignment for loops. This is the default.

`-missue-rate=`*number*
: Issue *number* instructions per cycle. *number* can only be 1 or 2.

`-mbranch-cost=`*number*
: *number* can only be 1 or 2. If it is 1 then branches are preferred over conditional code, if it is 2, then the opposite applies.

`-mflush-trap=`*number*
: Specifies the trap number to use to flush the cache. The default is 12. Valid numbers are between 0 and 15 inclusive.

`-mno-flush-trap`
: Specifies that the cache cannot be flushed by using a trap.

`-mflush-func=name`

Specifies the name of the operating system function to call to flush the cache. The default is '`_flush_cache`', but a function call is only used if a trap is not available.

`-mno-flush-func`

Indicates that there is no OS function for flushing the cache.

3.18.24 M680x0 Options

These are the '`-m`' options defined for M680x0 and ColdFire processors. The default settings depend on which architecture was selected when the compiler was configured; the defaults for the most common choices are given below.

`-march=arch`

Generate code for a specific M680x0 or ColdFire instruction set architecture. Permissible values of *arch* for M680x0 architectures are: '`68000`', '`68010`', '`68020`', '`68030`', '`68040`', '`68060`' and '`cpu32`'. ColdFire architectures are selected according to Freescale's ISA classification and the permissible values are: '`isaa`', '`isaaplus`', '`isab`' and '`isac`'.

GCC defines a macro `__mcf`*arch*`__` whenever it is generating code for a ColdFire target. The *arch* in this macro is one of the '`-march`' arguments given above.

When used together, '`-march`' and '`-mtune`' select code that runs on a family of similar processors but that is optimized for a particular microarchitecture.

`-mcpu=cpu`

Generate code for a specific M680x0 or ColdFire processor. The M680x0 *cpu*s are: '`68000`', '`68010`', '`68020`', '`68030`', '`68040`', '`68060`', '`68302`', '`68332`' and '`cpu32`'. The ColdFire *cpu*s are given by the table below, which also classifies the CPUs into families:

Family	**'-mcpu' arguments**
'51'	'51' '51ac' '51ag' '51cn' '51em' '51je' '51jf' '51jg' '51jm' '51mm' '51qe' '51qm'
'5206'	'5202' '5204' '5206'
'5206e'	'5206e'
'5208'	'5207' '5208'
'5211a'	'5210a' '5211a'
'5213'	'5211' '5212' '5213'
'5216'	'5214' '5216'
'52235'	'52230' '52231' '52232' '52233' '52234' '52235'
'5225'	'5224' '5225'
'52259'	'52252' '52254' '52255' '52256' '52258' '52259'
'5235'	'5232' '5233' '5234' '5235' '523x'
'5249'	'5249'
'5250'	'5250'
'5271'	'5270' '5271'
'5272'	'5272'
'5275'	'5274' '5275'
'5282'	'5280' '5281' '5282' '528x'

'53017'	'53011' '53012' '53013' '53014' '53015' '53016' '53017'
'5307'	'5307'
'5329'	'5327' '5328' '5329' '532x'
'5373'	'5372' '5373' '537x'
'5407'	'5407'
'5475'	'5470' '5471' '5472' '5473' '5474' '5475' '547x' '5480' '5481' '5482' '5483' '5484' '5485'

'-mcpu=*cpu*' overrides '-march=*arch*' if *arch* is compatible with *cpu*. Other combinations of '-mcpu' and '-march' are rejected.

GCC defines the macro __mcf_cpu_*cpu* when ColdFire target *cpu* is selected. It also defines __mcf_family_*family*, where the value of *family* is given by the table above.

-mtune=*tune*

Tune the code for a particular microarchitecture within the constraints set by '-march' and '-mcpu'. The M680x0 microarchitectures are: '68000', '68010', '68020', '68030', '68040', '68060' and 'cpu32'. The ColdFire microarchitectures are: 'cfv1', 'cfv2', 'cfv3', 'cfv4' and 'cfv4e'.

You can also use '-mtune=68020-40' for code that needs to run relatively well on 68020, 68030 and 68040 targets. '-mtune=68020-60' is similar but includes 68060 targets as well. These two options select the same tuning decisions as '-m68020-40' and '-m68020-60' respectively.

GCC defines the macros __m*arch* and __m*arch*__ when tuning for 680x0 architecture *arch*. It also defines m*arch* unless either '-ansi' or a non-GNU '-std' option is used. If GCC is tuning for a range of architectures, as selected by '-mtune=68020-40' or '-mtune=68020-60', it defines the macros for every architecture in the range.

GCC also defines the macro __m*uarch*__ when tuning for ColdFire microarchitecture *uarch*, where *uarch* is one of the arguments given above.

-m68000
-mc68000

Generate output for a 68000. This is the default when the compiler is configured for 68000-based systems. It is equivalent to '-march=68000'.

Use this option for microcontrollers with a 68000 or EC000 core, including the 68008, 68302, 68306, 68307, 68322, 68328 and 68356.

-m68010

Generate output for a 68010. This is the default when the compiler is configured for 68010-based systems. It is equivalent to '-march=68010'.

-m68020
-mc68020

Generate output for a 68020. This is the default when the compiler is configured for 68020-based systems. It is equivalent to '-march=68020'.

-m68030

Generate output for a 68030. This is the default when the compiler is configured for 68030-based systems. It is equivalent to '-march=68030'.

-m68040

Generate output for a 68040. This is the default when the compiler is configured for 68040-based systems. It is equivalent to '-march=68040'.

This option inhibits the use of 68881/68882 instructions that have to be emulated by software on the 68040. Use this option if your 68040 does not have code to emulate those instructions.

`-m68060` Generate output for a 68060. This is the default when the compiler is configured for 68060-based systems. It is equivalent to '`-march=68060`'.

This option inhibits the use of 68020 and 68881/68882 instructions that have to be emulated by software on the 68060. Use this option if your 68060 does not have code to emulate those instructions.

`-mcpu32` Generate output for a CPU32. This is the default when the compiler is configured for CPU32-based systems. It is equivalent to '`-march=cpu32`'.

Use this option for microcontrollers with a CPU32 or CPU32+ core, including the 68330, 68331, 68332, 68333, 68334, 68336, 68340, 68341, 68349 and 68360.

`-m5200` Generate output for a 520X ColdFire CPU. This is the default when the compiler is configured for 520X-based systems. It is equivalent to '`-mcpu=5206`', and is now deprecated in favor of that option.

Use this option for microcontroller with a 5200 core, including the MCF5202, MCF5203, MCF5204 and MCF5206.

`-m5206e` Generate output for a 5206e ColdFire CPU. The option is now deprecated in favor of the equivalent '`-mcpu=5206e`'.

`-m528x` Generate output for a member of the ColdFire 528X family. The option is now deprecated in favor of the equivalent '`-mcpu=528x`'.

`-m5307` Generate output for a ColdFire 5307 CPU. The option is now deprecated in favor of the equivalent '`-mcpu=5307`'.

`-m5407` Generate output for a ColdFire 5407 CPU. The option is now deprecated in favor of the equivalent '`-mcpu=5407`'.

`-mcfv4e` Generate output for a ColdFire V4e family CPU (e.g. 547x/548x). This includes use of hardware floating-point instructions. The option is equivalent to '`-mcpu=547x`', and is now deprecated in favor of that option.

`-m68020-40`
Generate output for a 68040, without using any of the new instructions. This results in code that can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68040.

The option is equivalent to '`-march=68020`' '`-mtune=68020-40`'.

`-m68020-60`
Generate output for a 68060, without using any of the new instructions. This results in code that can run relatively efficiently on either a 68020/68881 or a 68030 or a 68040. The generated code does use the 68881 instructions that are emulated on the 68060.

The option is equivalent to '`-march=68020`' '`-mtune=68020-60`'.

`-mhard-float`
`-m68881` Generate floating-point instructions. This is the default for 68020 and above, and for ColdFire devices that have an FPU. It defines the macro `__HAVE_68881__` on M680x0 targets and `__mcffpu__` on ColdFire targets.

`-msoft-float`
Do not generate floating-point instructions; use library calls instead. This is the default for 68000, 68010, and 68832 targets. It is also the default for ColdFire devices that have no FPU.

`-mdiv`
`-mno-div` Generate (do not generate) ColdFire hardware divide and remainder instructions. If '`-march`' is used without '`-mcpu`', the default is "on" for ColdFire architectures and "off" for M680x0 architectures. Otherwise, the default is taken from the target CPU (either the default CPU, or the one specified by '`-mcpu`'). For example, the default is "off" for '`-mcpu=5206`' and "on" for '`-mcpu=5206e`'.

GCC defines the macro `__mcfhwdiv__` when this option is enabled.

`-mshort` Consider type `int` to be 16 bits wide, like `short int`. Additionally, parameters passed on the stack are also aligned to a 16-bit boundary even on targets whose API mandates promotion to 32-bit.

`-mno-short`
Do not consider type `int` to be 16 bits wide. This is the default.

`-mnobitfield`
`-mno-bitfield`
Do not use the bit-field instructions. The '`-m68000`', '`-mcpu32`' and '`-m5200`' options imply '`-mnobitfield`'.

`-mbitfield`
Do use the bit-field instructions. The '`-m68020`' option implies '`-mbitfield`'. This is the default if you use a configuration designed for a 68020.

`-mrtd` Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `rtd` instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code is generated for calls to those functions.

In addition, seriously incorrect code results if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The `rtd` instruction is supported by the 68010, 68020, 68030, 68040, 68060 and CPU32 processors, but not by the 68000 or 5200.

The default is '`-mno-rtd`'.

`-malign-int`
`-mno-align-int`

Control whether GCC aligns `int`, `long`, `long long`, `float`, `double`, and `long double` variables on a 32-bit boundary ('`-malign-int`') or a 16-bit boundary ('`-mno-align-int`'). Aligning variables on 32-bit boundaries produces code that runs somewhat faster on processors with 32-bit busses at the expense of more memory.

Warning: if you use the '`-malign-int`' switch, GCC aligns structures containing the above types differently than most published application binary interface specifications for the m68k.

`-mpcrel` Use the pc-relative addressing mode of the 68000 directly, instead of using a global offset table. At present, this option implies '`-fpic`', allowing at most a 16-bit offset for pc-relative addressing. '`-fPIC`' is not presently supported with '`-mpcrel`', though this could be supported for 68020 and higher processors.

`-mno-strict-align`
`-mstrict-align`

Do not (do) assume that unaligned memory references are handled by the system.

`-msep-data`

Generate code that allows the data segment to be located in a different area of memory from the text segment. This allows for execute-in-place in an environment without virtual memory management. This option implies '`-fPIC`'.

`-mno-sep-data`

Generate code that assumes that the data segment follows the text segment. This is the default.

`-mid-shared-library`

Generate code that supports shared libraries via the library ID method. This allows for execute-in-place and shared libraries in an environment without virtual memory management. This option implies '`-fPIC`'.

`-mno-id-shared-library`

Generate code that doesn't assume ID-based shared libraries are being used. This is the default.

`-mshared-library-id=n`

Specifies the identification number of the ID-based shared library being compiled. Specifying a value of 0 generates more compact code; specifying other values forces the allocation of that number to the current library, but is no more space- or time-efficient than omitting this option.

`-mxgot`
`-mno-xgot`

When generating position-independent code for ColdFire, generate code that works if the GOT has more than 8192 entries. This code is larger and slower than code generated without this option. On M680x0 processors, this option is not needed; '`-fPIC`' suffices.

GCC normally uses a single instruction to load values from the GOT. While this is relatively efficient, it only works if the GOT is smaller than about 64k. Anything larger causes the linker to report an error such as:

```
relocation truncated to fit: R_68K_GOT16O foobar
```

If this happens, you should recompile your code with '`-mxgot`'. It should then work with very large GOTs. However, code generated with '`-mxgot`' is less efficient, since it takes 4 instructions to fetch the value of a global symbol.

Note that some linkers, including newer versions of the GNU linker, can create multiple GOTs and sort GOT entries. If you have such a linker, you should only need to use '`-mxgot`' when compiling a single object file that accesses more than 8192 GOT entries. Very few do.

These options have no effect unless GCC is generating position-independent code.

`-mlong-jump-table-offsets`
: Use 32-bit offsets in `switch` tables. The default is to use 16-bit offsets.

3.18.25 MCore Options

These are the '`-m`' options defined for the Motorola M*Core processors.

`-mhardlit`
`-mno-hardlit`
: Inline constants into the code stream if it can be done in two instructions or less.

`-mdiv`
`-mno-div` Use the divide instruction. (Enabled by default).

`-mrelax-immediate`
`-mno-relax-immediate`
: Allow arbitrary-sized immediates in bit operations.

`-mwide-bitfields`
`-mno-wide-bitfields`
: Always treat bit-fields as `int`-sized.

`-m4byte-functions`
`-mno-4byte-functions`
: Force all functions to be aligned to a 4-byte boundary.

`-mcallgraph-data`
`-mno-callgraph-data`
: Emit callgraph information.

`-mslow-bytes`
`-mno-slow-bytes`
: Prefer word access when reading byte quantities.

`-mlittle-endian`
`-mbig-endian`
: Generate code for a little-endian target.

`-m210`
`-m340` Generate code for the 210 processor.

`-mno-lsim`
Assume that runtime support has been provided and so omit the simulator library ('`libsim.a)`' from the linker command line.

`-mstack-increment=size`
Set the maximum amount for a single stack increment operation. Large values can increase the speed of programs that contain functions that need a large amount of stack space, but they can also trigger a segmentation fault if the stack is extended too much. The default value is 0x1000.

3.18.26 MeP Options

`-mabsdiff`
Enables the `abs` instruction, which is the absolute difference between two registers.

`-mall-opts`
Enables all the optional instructions—average, multiply, divide, bit operations, leading zero, absolute difference, min/max, clip, and saturation.

`-maverage`
Enables the `ave` instruction, which computes the average of two registers.

`-mbased=n`
Variables of size *n* bytes or smaller are placed in the `.based` section by default. Based variables use the `$tp` register as a base register, and there is a 128-byte limit to the `.based` section.

`-mbitops` Enables the bit operation instructions—bit test (`btstm`), set (`bsetm`), clear (`bclrm`), invert (`bnotm`), and test-and-set (`tas`).

`-mc=name` Selects which section constant data is placed in. *name* may be '`tiny`', '`near`', or '`far`'.

`-mclip` Enables the `clip` instruction. Note that '`-mclip`' is not useful unless you also provide '`-mminmax`'.

`-mconfig=name`
Selects one of the built-in core configurations. Each MeP chip has one or more modules in it; each module has a core CPU and a variety of coprocessors, optional instructions, and peripherals. The `MeP-Integrator` tool, not part of GCC, provides these configurations through this option; using this option is the same as using all the corresponding command-line options. The default configuration is '`default`'.

`-mcop` Enables the coprocessor instructions. By default, this is a 32-bit coprocessor. Note that the coprocessor is normally enabled via the '`-mconfig=`' option.

`-mcop32` Enables the 32-bit coprocessor's instructions.

`-mcop64` Enables the 64-bit coprocessor's instructions.

`-mivc2` Enables IVC2 scheduling. IVC2 is a 64-bit VLIW coprocessor.

`-mdc` Causes constant variables to be placed in the `.near` section.

`-mdiv` Enables the `div` and `divu` instructions.

`-meb` Generate big-endian code.

`-mel` Generate little-endian code.

`-mio-volatile`
Tells the compiler that any variable marked with the `io` attribute is to be considered volatile.

`-ml` Causes variables to be assigned to the `.far` section by default.

`-mleadz` Enables the `leadz` (leading zero) instruction.

`-mm` Causes variables to be assigned to the `.near` section by default.

`-mminmax` Enables the `min` and `max` instructions.

`-mmult` Enables the multiplication and multiply-accumulate instructions.

`-mno-opts`
Disables all the optional instructions enabled by '`-mall-opts`'.

`-mrepeat` Enables the `repeat` and `erepeat` instructions, used for low-overhead looping.

`-ms` Causes all variables to default to the `.tiny` section. Note that there is a 65536-byte limit to this section. Accesses to these variables use the `%gp` base register.

`-msatur` Enables the saturation instructions. Note that the compiler does not currently generate these itself, but this option is included for compatibility with other tools, like `as`.

`-msdram` Link the SDRAM-based runtime instead of the default ROM-based runtime.

`-msim` Link the simulator run-time libraries.

`-msimnovec`
Link the simulator runtime libraries, excluding built-in support for reset and exception vectors and tables.

`-mtf` Causes all functions to default to the `.far` section. Without this option, functions default to the `.near` section.

`-mtiny=n` Variables that are *n* bytes or smaller are allocated to the `.tiny` section. These variables use the `$gp` base register. The default for this option is 4, but note that there's a 65536-byte limit to the `.tiny` section.

3.18.27 MicroBlaze Options

`-msoft-float`
Use software emulation for floating point (default).

`-mhard-float`
Use hardware floating-point instructions.

`-mmemcpy` Do not optimize block moves, use `memcpy`.

`-mno-clearbss`
This option is deprecated. Use '`-fno-zero-initialized-in-bss`' instead.

`-mcpu=`*`cpu-type`*
Use features of, and schedule code for, the given CPU. Supported values are in the format '`v`*X*`.`*YY*`.`*Z*', where *X* is a major version, *YY* is the minor version, and *Z* is compatibility code. Example values are '`v3.00.a`', '`v4.00.b`', '`v5.00.a`', '`v5.00.b`', '`v6.00.a`'.

`-mxl-soft-mul`
Use software multiply emulation (default).

`-mxl-soft-div`
Use software emulation for divides (default).

`-mxl-barrel-shift`
Use the hardware barrel shifter.

`-mxl-pattern-compare`
Use pattern compare instructions.

`-msmall-divides`
Use table lookup optimization for small signed integer divisions.

`-mxl-stack-check`
This option is deprecated. Use '`-fstack-check`' instead.

`-mxl-gp-opt`
Use GP-relative `.sdata`/`.sbss` sections.

`-mxl-multiply-high`
Use multiply high instructions for high part of 32x32 multiply.

`-mxl-float-convert`
Use hardware floating-point conversion instructions.

`-mxl-float-sqrt`
Use hardware floating-point square root instruction.

`-mbig-endian`
Generate code for a big-endian target.

`-mlittle-endian`
Generate code for a little-endian target.

`-mxl-reorder`
Use reorder instructions (swap and byte reversed load/store).

`-mxl-mode-`*`app-model`*
Select application model *app-model*. Valid models are

'`executable`'
normal executable (default), uses startup code '`crt0.o`'.

'-mpic-data-is-text-relative'
: Assume that the displacement between the text and data segments is fixed at static link time. This allows data to be referenced by offset from start of text address instead of GOT since PC-relative addressing is not supported.

'xmdstub' for use with Xilinx Microprocessor Debugger (XMD) based software intrusive debug agent called xmdstub. This uses startup file 'crt1.o' and sets the start address of the program to 0x800.

'bootstrap'
: for applications that are loaded using a bootloader. This model uses startup file 'crt2.o' which does not contain a processor reset vector handler. This is suitable for transferring control on a processor reset to the bootloader rather than the application.

'novectors'
: for applications that do not require any of the MicroBlaze vectors. This option may be useful for applications running within a monitoring application. This model uses 'crt3.o' as a startup file.

Option '-xl-mode-*app-model*' is a deprecated alias for '-mxl-mode-*app-model*'.

3.18.28 MIPS Options

-EB Generate big-endian code.

-EL Generate little-endian code. This is the default for 'mips*el-*-*' configurations.

-march=*arch*
: Generate code that runs on *arch*, which can be the name of a generic MIPS ISA, or the name of a particular processor. The ISA names are: 'mips1', 'mips2', 'mips3', 'mips4', 'mips32', 'mips32r2', 'mips32r3', 'mips32r5', 'mips32r6', 'mips64', 'mips64r2', 'mips64r3', 'mips64r5' and 'mips64r6'. The processor names are: '4kc', '4km', '4kp', '4ksc', '4kec', '4kem', '4kep', '4ksd', '5kc', '5kf', '20kc', '24kc', '24kf2_1', '24kf1_1', '24kec', '24kef2_1', '24kef1_1', '34kc', '34kf2_1', '34kf1_1', '34kn', '74kc', '74kf2_1', '74kf1_1', '74kf3_2', '1004kc', '1004kf2_1', '1004kf1_1', 'i6400', 'i6500', 'interaptiv', 'loongson2e', 'loongson2f', 'loongson3a', 'gs464', 'gs464e', 'gs264e', 'm4k', 'm14k', 'm14kc', 'm14ke', 'm14kec', 'm5100', 'm5101', 'octeon', 'octeon+', 'octeon2', 'octeon3', 'orion', 'p5600', 'p6600', 'r2000', 'r3000', 'r3900', 'r4000', 'r4400', 'r4600', 'r4650', 'r4700', 'r5900', 'r6000', 'r8000', 'rm7000', 'rm9000', 'r10000', 'r12000', 'r14000', 'r16000', 'sb1', 'sr71000', 'vr4100', 'vr4111', 'vr4120', 'vr4130', 'vr4300', 'vr5000', 'vr5400', 'vr5500', 'xlr' and 'xlp'. The special value 'from-abi' selects the most compatible architecture for the selected ABI (that is, 'mips1' for 32-bit ABIs and 'mips3' for 64-bit ABIs).

The native Linux/GNU toolchain also supports the value '`native`', which selects the best architecture option for the host processor. '`-march=native`' has no effect if GCC does not recognize the processor.

In processor names, a final '`000`' can be abbreviated as '`k`' (for example, '`-march=r2k`'). Prefixes are optional, and '`vr`' may be written '`r`'.

Names of the form '`nf2_1`' refer to processors with FPUs clocked at half the rate of the core, names of the form '`nf1_1`' refer to processors with FPUs clocked at the same rate as the core, and names of the form '`nf3_2`' refer to processors with FPUs clocked a ratio of 3:2 with respect to the core. For compatibility reasons, '`nf`' is accepted as a synonym for '`nf2_1`' while '`nx`' and '`bfx`' are accepted as synonyms for '`nf1_1`'.

GCC defines two macros based on the value of this option. The first is `_MIPS_ARCH`, which gives the name of target architecture, as a string. The second has the form `_MIPS_ARCH_`*foo*, where *foo* is the capitalized value of `_MIPS_ARCH`. For example, '`-march=r2000`' sets `_MIPS_ARCH` to `"r2000"` and defines the macro `_MIPS_ARCH_R2000`.

Note that the `_MIPS_ARCH` macro uses the processor names given above. In other words, it has the full prefix and does not abbreviate '`000`' as '`k`'. In the case of '`from-abi`', the macro names the resolved architecture (either `"mips1"` or `"mips3"`). It names the default architecture when no '`-march`' option is given.

`-mtune=`*arch*

Optimize for *arch*. Among other things, this option controls the way instructions are scheduled, and the perceived cost of arithmetic operations. The list of *arch* values is the same as for '`-march`'.

When this option is not used, GCC optimizes for the processor specified by '`-march`'. By using '`-march`' and '`-mtune`' together, it is possible to generate code that runs on a family of processors, but optimize the code for one particular member of that family.

'`-mtune`' defines the macros `_MIPS_TUNE` and `_MIPS_TUNE_`*foo*, which work in the same way as the '`-march`' ones described above.

`-mips1` Equivalent to '`-march=mips1`'.

`-mips2` Equivalent to '`-march=mips2`'.

`-mips3` Equivalent to '`-march=mips3`'.

`-mips4` Equivalent to '`-march=mips4`'.

`-mips32` Equivalent to '`-march=mips32`'.

`-mips32r3`

Equivalent to '`-march=mips32r3`'.

`-mips32r5`

Equivalent to '`-march=mips32r5`'.

`-mips32r6`

Equivalent to '`-march=mips32r6`'.

`-mips64` Equivalent to '`-march=mips64`'.

`-mips64r2`
Equivalent to '`-march=mips64r2`'.

`-mips64r3`
Equivalent to '`-march=mips64r3`'.

`-mips64r5`
Equivalent to '`-march=mips64r5`'.

`-mips64r6`
Equivalent to '`-march=mips64r6`'.

`-mips16`
`-mno-mips16`
Generate (do not generate) MIPS16 code. If GCC is targeting a MIPS32 or MIPS64 architecture, it makes use of the MIPS16e ASE.

MIPS16 code generation can also be controlled on a per-function basis by means of `mips16` and `nomips16` attributes. See Section 6.33 [Function Attributes], page 477, for more information.

`-mflip-mips16`
Generate MIPS16 code on alternating functions. This option is provided for regression testing of mixed MIPS16/non-MIPS16 code generation, and is not intended for ordinary use in compiling user code.

`-minterlink-compressed`
`-mno-interlink-compressed`
Require (do not require) that code using the standard (uncompressed) MIPS ISA be link-compatible with MIPS16 and microMIPS code, and vice versa.

For example, code using the standard ISA encoding cannot jump directly to MIPS16 or microMIPS code; it must either use a call or an indirect jump. '`-minterlink-compressed`' therefore disables direct jumps unless GCC knows that the target of the jump is not compressed.

`-minterlink-mips16`
`-mno-interlink-mips16`
Aliases of '`-minterlink-compressed`' and '`-mno-interlink-compressed`'. These options predate the microMIPS ASE and are retained for backwards compatibility.

`-mabi=32`
`-mabi=o64`
`-mabi=n32`
`-mabi=64`
`-mabi=eabi`
Generate code for the given ABI.

Note that the EABI has a 32-bit and a 64-bit variant. GCC normally generates 64-bit code when you select a 64-bit architecture, but you can use '`-mgp32`' to get 32-bit code instead.

For information about the O64 ABI, see `http://gcc.gnu.org/projects/mipso64-abi.html`.

GCC supports a variant of the o32 ABI in which floating-point registers are 64 rather than 32 bits wide. You can select this combination with '`-mabi=32`' '`-mfp64`'. This ABI relies on the `mthc1` and `mfhc1` instructions and is therefore only supported for MIPS32R2, MIPS32R3 and MIPS32R5 processors.

The register assignments for arguments and return values remain the same, but each scalar value is passed in a single 64-bit register rather than a pair of 32-bit registers. For example, scalar floating-point values are returned in '`$f0`' only, not a '`$f0`'/'`$f1`' pair. The set of call-saved registers also remains the same in that the even-numbered double-precision registers are saved.

Two additional variants of the o32 ABI are supported to enable a transition from 32-bit to 64-bit registers. These are FPXX ('`-mfpxx`') and FP64A ('`-mfp64`' '`-mno-odd-spreg`'). The FPXX extension mandates that all code must execute correctly when run using 32-bit or 64-bit registers. The code can be interlinked with either FP32 or FP64, but not both. The FP64A extension is similar to the FP64 extension but forbids the use of odd-numbered single-precision registers. This can be used in conjunction with the `FRE` mode of FPUs in MIPS32R5 processors and allows both FP32 and FP64A code to interlink and run in the same process without changing FPU modes.

`-mabicalls`
`-mno-abicalls`

Generate (do not generate) code that is suitable for SVR4-style dynamic objects. '`-mabicalls`' is the default for SVR4-based systems.

`-mshared`
`-mno-shared`

Generate (do not generate) code that is fully position-independent, and that can therefore be linked into shared libraries. This option only affects '`-mabicalls`'.

All '`-mabicalls`' code has traditionally been position-independent, regardless of options like '`-fPIC`' and '`-fpic`'. However, as an extension, the GNU toolchain allows executables to use absolute accesses for locally-binding symbols. It can also use shorter GP initialization sequences and generate direct calls to locally-defined functions. This mode is selected by '`-mno-shared`'.

'`-mno-shared`' depends on binutils 2.16 or higher and generates objects that can only be linked by the GNU linker. However, the option does not affect the ABI of the final executable; it only affects the ABI of relocatable objects. Using '`-mno-shared`' generally makes executables both smaller and quicker.

'`-mshared`' is the default.

`-mplt`
`-mno-plt` Assume (do not assume) that the static and dynamic linkers support PLTs and copy relocations. This option only affects '`-mno-shared -mabicalls`'. For the n64 ABI, this option has no effect without '`-msym32`'.

You can make '`-mplt`' the default by configuring GCC with '`--with-mips-plt`'. The default is '`-mno-plt`' otherwise.

`-mxgot`
`-mno-xgot`

Lift (do not lift) the usual restrictions on the size of the global offset table.

GCC normally uses a single instruction to load values from the GOT. While this is relatively efficient, it only works if the GOT is smaller than about 64k. Anything larger causes the linker to report an error such as:

```
relocation truncated to fit: R_MIPS_GOT16 foobar
```

If this happens, you should recompile your code with '`-mxgot`'. This works with very large GOTs, although the code is also less efficient, since it takes three instructions to fetch the value of a global symbol.

Note that some linkers can create multiple GOTs. If you have such a linker, you should only need to use '`-mxgot`' when a single object file accesses more than 64k's worth of GOT entries. Very few do.

These options have no effect unless GCC is generating position independent code.

`-mgp32` Assume that general-purpose registers are 32 bits wide.

`-mgp64` Assume that general-purpose registers are 64 bits wide.

`-mfp32` Assume that floating-point registers are 32 bits wide.

`-mfp64` Assume that floating-point registers are 64 bits wide.

`-mfpxx` Do not assume the width of floating-point registers.

`-mhard-float`

Use floating-point coprocessor instructions.

`-msoft-float`

Do not use floating-point coprocessor instructions. Implement floating-point calculations using library calls instead.

`-mno-float`

Equivalent to '`-msoft-float`', but additionally asserts that the program being compiled does not perform any floating-point operations. This option is presently supported only by some bare-metal MIPS configurations, where it may select a special set of libraries that lack all floating-point support (including, for example, the floating-point `printf` formats). If code compiled with '`-mno-float`' accidentally contains floating-point operations, it is likely to suffer a link-time or run-time failure.

`-msingle-float`

Assume that the floating-point coprocessor only supports single-precision operations.

`-mdouble-float`

Assume that the floating-point coprocessor supports double-precision operations. This is the default.

`-modd-spreg`
`-mno-odd-spreg`

Enable the use of odd-numbered single-precision floating-point registers for the o32 ABI. This is the default for processors that are known to support these registers. When using the o32 FPXX ABI, '`-mno-odd-spreg`' is set by default.

`-mabs=2008`
`-mabs=legacy`

These options control the treatment of the special not-a-number (NaN) IEEE 754 floating-point data with the `abs.`*`fmt`* and `neg.`*`fmt`* machine instructions.

By default or when '`-mabs=legacy`' is used the legacy treatment is selected. In this case these instructions are considered arithmetic and avoided where correct operation is required and the input operand might be a NaN. A longer sequence of instructions that manipulate the sign bit of floating-point datum manually is used instead unless the '`-ffinite-math-only`' option has also been specified.

The '`-mabs=2008`' option selects the IEEE 754-2008 treatment. In this case these instructions are considered non-arithmetic and therefore operating correctly in all cases, including in particular where the input operand is a NaN. These instructions are therefore always used for the respective operations.

`-mnan=2008`
`-mnan=legacy`

These options control the encoding of the special not-a-number (NaN) IEEE 754 floating-point data.

The '`-mnan=legacy`' option selects the legacy encoding. In this case quiet NaNs (qNaNs) are denoted by the first bit of their trailing significand field being 0, whereas signaling NaNs (sNaNs) are denoted by the first bit of their trailing significand field being 1.

The '`-mnan=2008`' option selects the IEEE 754-2008 encoding. In this case qNaNs are denoted by the first bit of their trailing significand field being 1, whereas sNaNs are denoted by the first bit of their trailing significand field being 0.

The default is '`-mnan=legacy`' unless GCC has been configured with '`--with-nan=2008`'.

`-mllsc`
`-mno-llsc`

Use (do not use) '`ll`', '`sc`', and '`sync`' instructions to implement atomic memory built-in functions. When neither option is specified, GCC uses the instructions if the target architecture supports them.

'`-mllsc`' is useful if the runtime environment can emulate the instructions and '`-mno-llsc`' can be useful when compiling for nonstandard ISAs. You can make either option the default by configuring GCC with '`--with-llsc`' and '`--without-llsc`' respectively. '`--with-llsc`' is the default for some configurations; see the installation documentation for details.

`-mdsp`
`-mno-dsp` Use (do not use) revision 1 of the MIPS DSP ASE. See Section 6.60.13 [MIPS DSP Built-in Functions], page 675. This option defines the preprocessor macro `__mips_dsp`. It also defines `__mips_dsp_rev` to 1.

`-mdspr2`
`-mno-dspr2`
Use (do not use) revision 2 of the MIPS DSP ASE. See Section 6.60.13 [MIPS DSP Built-in Functions], page 675. This option defines the preprocessor macros `__mips_dsp` and `__mips_dspr2`. It also defines `__mips_dsp_rev` to 2.

`-msmartmips`
`-mno-smartmips`
Use (do not use) the MIPS SmartMIPS ASE.

`-mpaired-single`
`-mno-paired-single`
Use (do not use) paired-single floating-point instructions. See Section 6.60.14 [MIPS Paired-Single Support], page 679. This option requires hardware floating-point support to be enabled.

`-mdmx`
`-mno-mdmx`
Use (do not use) MIPS Digital Media Extension instructions. This option can only be used when generating 64-bit code and requires hardware floating-point support to be enabled.

`-mips3d`
`-mno-mips3d`
Use (do not use) the MIPS-3D ASE. See Section 6.60.15.3 [MIPS-3D Built-in Functions], page 683. The option '`-mips3d`' implies '`-mpaired-single`'.

`-mmicromips`
`-mno-micromips`
Generate (do not generate) microMIPS code.

MicroMIPS code generation can also be controlled on a per-function basis by means of `micromips` and `nomicromips` attributes. See Section 6.33 [Function Attributes], page 477, for more information.

`-mmt`
`-mno-mt` Use (do not use) MT Multithreading instructions.

`-mmcu`
`-mno-mcu` Use (do not use) the MIPS MCU ASE instructions.

`-meva`
`-mno-eva` Use (do not use) the MIPS Enhanced Virtual Addressing instructions.

`-mvirt`
`-mno-virt`
Use (do not use) the MIPS Virtualization (VZ) instructions.

`-mxpa`
`-mno-xpa` Use (do not use) the MIPS eXtended Physical Address (XPA) instructions.

`-mcrc`
`-mno-crc` Use (do not use) the MIPS Cyclic Redundancy Check (CRC) instructions.

`-mginv`
`-mno-ginv`
Use (do not use) the MIPS Global INValidate (GINV) instructions.

`-mloongson-mmi`
`-mno-loongson-mmi`
Use (do not use) the MIPS Loongson MultiMedia extensions Instructions (MMI).

`-mloongson-ext`
`-mno-loongson-ext`
Use (do not use) the MIPS Loongson EXTensions (EXT) instructions.

`-mloongson-ext2`
`-mno-loongson-ext2`
Use (do not use) the MIPS Loongson EXTensions r2 (EXT2) instructions.

`-mlong64` Force `long` types to be 64 bits wide. See '`-mlong32`' for an explanation of the default and the way that the pointer size is determined.

`-mlong32` Force `long`, `int`, and pointer types to be 32 bits wide.

The default size of `int`s, `long`s and pointers depends on the ABI. All the supported ABIs use 32-bit `int`s. The n64 ABI uses 64-bit `long`s, as does the 64-bit EABI; the others use 32-bit `long`s. Pointers are the same size as `long`s, or the same size as integer registers, whichever is smaller.

`-msym32`
`-mno-sym32`
Assume (do not assume) that all symbols have 32-bit values, regardless of the selected ABI. This option is useful in combination with '`-mabi=64`' and '`-mno-abicalls`' because it allows GCC to generate shorter and faster references to symbolic addresses.

`-G` *num* Put definitions of externally-visible data in a small data section if that data is no bigger than *num* bytes. GCC can then generate more efficient accesses to the data; see '`-mgpopt`' for details.

The default '`-G`' option depends on the configuration.

`-mlocal-sdata`
`-mno-local-sdata`
Extend (do not extend) the '`-G`' behavior to local data too, such as to static variables in C. '`-mlocal-sdata`' is the default for all configurations.

If the linker complains that an application is using too much small data, you might want to try rebuilding the less performance-critical parts with '`-mno-local-sdata`'. You might also want to build large libraries with '`-mno-local-sdata`', so that the libraries leave more room for the main program.

`-mextern-sdata`
`-mno-extern-sdata`

Assume (do not assume) that externally-defined data is in a small data section if the size of that data is within the '`-G`' limit. '`-mextern-sdata`' is the default for all configurations.

If you compile a module *Mod* with '`-mextern-sdata`' '`-G` *num*' '`-mgpopt`', and *Mod* references a variable *Var* that is no bigger than *num* bytes, you must make sure that *Var* is placed in a small data section. If *Var* is defined by another module, you must either compile that module with a high-enough '`-G`' setting or attach a `section` attribute to *Var*'s definition. If *Var* is common, you must link the application with a high-enough '`-G`' setting.

The easiest way of satisfying these restrictions is to compile and link every module with the same '`-G`' option. However, you may wish to build a library that supports several different small data limits. You can do this by compiling the library with the highest supported '`-G`' setting and additionally using '`-mno-extern-sdata`' to stop the library from making assumptions about externally-defined data.

`-mgpopt`
`-mno-gpopt`

Use (do not use) GP-relative accesses for symbols that are known to be in a small data section; see '`-G`', '`-mlocal-sdata`' and '`-mextern-sdata`'. '`-mgpopt`' is the default for all configurations.

'`-mno-gpopt`' is useful for cases where the `$gp` register might not hold the value of `_gp`. For example, if the code is part of a library that might be used in a boot monitor, programs that call boot monitor routines pass an unknown value in `$gp`. (In such situations, the boot monitor itself is usually compiled with '`-G0`'.)

'`-mno-gpopt`' implies '`-mno-local-sdata`' and '`-mno-extern-sdata`'.

`-membedded-data`
`-mno-embedded-data`

Allocate variables to the read-only data section first if possible, then next in the small data section if possible, otherwise in data. This gives slightly slower code than the default, but reduces the amount of RAM required when executing, and thus may be preferred for some embedded systems.

`-muninit-const-in-rodata`
`-mno-uninit-const-in-rodata`

Put uninitialized `const` variables in the read-only data section. This option is only meaningful in conjunction with '`-membedded-data`'.

`-mcode-readable=`*setting*

Specify whether GCC may generate code that reads from executable sections. There are three possible settings:

`-mcode-readable=yes`

Instructions may freely access executable sections. This is the default setting.

`-mcode-readable=pcrel`
: MIPS16 PC-relative load instructions can access executable sections, but other instructions must not do so. This option is useful on 4KSc and 4KSd processors when the code TLBs have the Read Inhibit bit set. It is also useful on processors that can be configured to have a dual instruction/data SRAM interface and that, like the M4K, automatically redirect PC-relative loads to the instruction RAM.

`-mcode-readable=no`
: Instructions must not access executable sections. This option can be useful on targets that are configured to have a dual instruction/data SRAM interface but that (unlike the M4K) do not automatically redirect PC-relative loads to the instruction RAM.

`-msplit-addresses`
`-mno-split-addresses`
: Enable (disable) use of the `%hi()` and `%lo()` assembler relocation operators. This option has been superseded by '`-mexplicit-relocs`' but is retained for backwards compatibility.

`-mexplicit-relocs`
`-mno-explicit-relocs`
: Use (do not use) assembler relocation operators when dealing with symbolic addresses. The alternative, selected by '`-mno-explicit-relocs`', is to use assembler macros instead.

  '`-mexplicit-relocs`' is the default if GCC was configured to use an assembler that supports relocation operators.

`-mcheck-zero-division`
`-mno-check-zero-division`
: Trap (do not trap) on integer division by zero.

  The default is '`-mcheck-zero-division`'.

`-mdivide-traps`
`-mdivide-breaks`
: MIPS systems check for division by zero by generating either a conditional trap or a break instruction. Using traps results in smaller code, but is only supported on MIPS II and later. Also, some versions of the Linux kernel have a bug that prevents trap from generating the proper signal (`SIGFPE`). Use '`-mdivide-traps`' to allow conditional traps on architectures that support them and '`-mdivide-breaks`' to force the use of breaks.

  The default is usually '`-mdivide-traps`', but this can be overridden at configure time using '`--with-divide=breaks`'. Divide-by-zero checks can be completely disabled using '`-mno-check-zero-division`'.

`-mload-store-pairs`
`-mno-load-store-pairs`
: Enable (disable) an optimization that pairs consecutive load or store instructions to enable load/store bonding. This option is enabled by default but only takes effect when the selected architecture is known to support bonding.

`-mmemcpy`
`-mno-memcpy`
: Force (do not force) the use of `memcpy` for non-trivial block moves. The default is '`-mno-memcpy`', which allows GCC to inline most constant-sized copies.

`-mlong-calls`
`-mno-long-calls`
: Disable (do not disable) use of the `jal` instruction. Calling functions using `jal` is more efficient but requires the caller and callee to be in the same 256 megabyte segment.

  This option has no effect on abicalls code. The default is '`-mno-long-calls`'.

`-mmad`
`-mno-mad`
: Enable (disable) use of the `mad`, `madu` and `mul` instructions, as provided by the R4650 ISA.

`-mimadd`
`-mno-imadd`
: Enable (disable) use of the `madd` and `msub` integer instructions. The default is '`-mimadd`' on architectures that support `madd` and `msub` except for the 74k architecture where it was found to generate slower code.

`-mfused-madd`
`-mno-fused-madd`
: Enable (disable) use of the floating-point multiply-accumulate instructions, when they are available. The default is '`-mfused-madd`'.

  On the R8000 CPU when multiply-accumulate instructions are used, the intermediate product is calculated to infinite precision and is not subject to the FCSR Flush to Zero bit. This may be undesirable in some circumstances. On other processors the result is numerically identical to the equivalent computation using separate multiply, add, subtract and negate instructions.

`-nocpp`
: Tell the MIPS assembler to not run its preprocessor over user assembler files (with a '`.s`' suffix) when assembling them.

`-mfix-24k`
`-mno-fix-24k`
: Work around the 24K E48 (lost data on stores during refill) errata. The workarounds are implemented by the assembler rather than by GCC.

`-mfix-r4000`
`-mno-fix-r4000`
: Work around certain R4000 CPU errata:

  - A double-word or a variable shift may give an incorrect result if executed immediately after starting an integer division.

- A double-word or a variable shift may give an incorrect result if executed while an integer multiplication is in progress.
- An integer division may give an incorrect result if started in a delay slot of a taken branch or a jump.

`-mfix-r4400`
`-mno-fix-r4400`

Work around certain R4400 CPU errata:

- A double-word or a variable shift may give an incorrect result if executed immediately after starting an integer division.

`-mfix-r10000`
`-mno-fix-r10000`

Work around certain R10000 errata:

- `ll/sc` sequences may not behave atomically on revisions prior to 3.0. They may deadlock on revisions 2.6 and earlier.

This option can only be used if the target architecture supports branch-likely instructions. '`-mfix-r10000`' is the default when '`-march=r10000`' is used; '`-mno-fix-r10000`' is the default otherwise.

`-mfix-r5900`
`-mno-fix-r5900`

Do not attempt to schedule the preceding instruction into the delay slot of a branch instruction placed at the end of a short loop of six instructions or fewer and always schedule a `nop` instruction there instead. The short loop bug under certain conditions causes loops to execute only once or twice, due to a hardware bug in the R5900 chip. The workaround is implemented by the assembler rather than by GCC.

`-mfix-rm7000`
`-mno-fix-rm7000`

Work around the RM7000 `dmult/dmultu` errata. The workarounds are implemented by the assembler rather than by GCC.

`-mfix-vr4120`
`-mno-fix-vr4120`

Work around certain VR4120 errata:

- `dmultu` does not always produce the correct result.
- `div` and `ddiv` do not always produce the correct result if one of the operands is negative.

The workarounds for the division errata rely on special functions in '`libgcc.a`'. At present, these functions are only provided by the `mips64vr*-elf` configurations.

Other VR4120 errata require a NOP to be inserted between certain pairs of instructions. These errata are handled by the assembler, not by GCC itself.

`-mfix-vr4130`

Work around the VR4130 `mflo/mfhi` errata. The workarounds are implemented by the assembler rather than by GCC, although GCC avoids using `mflo` and

`mfhi` if the VR4130 `macc`, `macchi`, `dmacc` and `dmacchi` instructions are available instead.

`-mfix-sb1`
`-mno-fix-sb1`
: Work around certain SB-1 CPU core errata. (This flag currently works around the SB-1 revision 2 "F1" and "F2" floating-point errata.)

`-mr10k-cache-barrier=`*`setting`*
: Specify whether GCC should insert cache barriers to avoid the side effects of speculation on R10K processors.

In common with many processors, the R10K tries to predict the outcome of a conditional branch and speculatively executes instructions from the "taken" branch. It later aborts these instructions if the predicted outcome is wrong. However, on the R10K, even aborted instructions can have side effects.

This problem only affects kernel stores and, depending on the system, kernel loads. As an example, a speculatively-executed store may load the target memory into cache and mark the cache line as dirty, even if the store itself is later aborted. If a DMA operation writes to the same area of memory before the "dirty" line is flushed, the cached data overwrites the DMA-ed data. See the R10K processor manual for a full description, including other potential problems.

One workaround is to insert cache barrier instructions before every memory access that might be speculatively executed and that might have side effects even if aborted. '`-mr10k-cache-barrier=`*`setting`*' controls GCC's implementation of this workaround. It assumes that aborted accesses to any byte in the following regions does not have side effects:

1. the memory occupied by the current function's stack frame;
2. the memory occupied by an incoming stack argument;
3. the memory occupied by an object with a link-time-constant address.

It is the kernel's responsibility to ensure that speculative accesses to these regions are indeed safe.

If the input program contains a function declaration such as:

```
void foo (void);
```

then the implementation of `foo` must allow `j foo` and `jal foo` to be executed speculatively. GCC honors this restriction for functions it compiles itself. It expects non-GCC functions (such as hand-written assembly code) to do the same.

The option has three forms:

`-mr10k-cache-barrier=load-store`
: Insert a cache barrier before a load or store that might be speculatively executed and that might have side effects even if aborted.

`-mr10k-cache-barrier=store`
: Insert a cache barrier before a store that might be speculatively executed and that might have side effects even if aborted.

`-mr10k-cache-barrier=none`
Disable the insertion of cache barriers. This is the default setting.

`-mflush-func=`*func*
`-mno-flush-func`
Specifies the function to call to flush the I and D caches, or to not call any such function. If called, the function must take the same arguments as the common `_flush_func`, that is, the address of the memory range for which the cache is being flushed, the size of the memory range, and the number 3 (to flush both caches). The default depends on the target GCC was configured for, but commonly is either `_flush_func` or `__cpu_flush`.

`mbranch-cost=`*num*
Set the cost of branches to roughly *num* "simple" instructions. This cost is only a heuristic and is not guaranteed to produce consistent results across releases. A zero cost redundantly selects the default, which is based on the '`-mtune`' setting.

`-mbranch-likely`
`-mno-branch-likely`
Enable or disable use of Branch Likely instructions, regardless of the default for the selected architecture. By default, Branch Likely instructions may be generated if they are supported by the selected architecture. An exception is for the MIPS32 and MIPS64 architectures and processors that implement those architectures; for those, Branch Likely instructions are not be generated by default because the MIPS32 and MIPS64 architectures specifically deprecate their use.

`-mcompact-branches=never`
`-mcompact-branches=optimal`
`-mcompact-branches=always`
These options control which form of branches will be generated. The default is '`-mcompact-branches=optimal`'.

The '`-mcompact-branches=never`' option ensures that compact branch instructions will never be generated.

The '`-mcompact-branches=always`' option ensures that a compact branch instruction will be generated if available. If a compact branch instruction is not available, a delay slot form of the branch will be used instead.

This option is supported from MIPS Release 6 onwards.

The '`-mcompact-branches=optimal`' option will cause a delay slot branch to be used if one is available in the current ISA and the delay slot is successfully filled. If the delay slot is not filled, a compact branch will be chosen if one is available.

`-mfp-exceptions`
`-mno-fp-exceptions`
Specifies whether FP exceptions are enabled. This affects how FP instructions are scheduled for some processors. The default is that FP exceptions are enabled.

For instance, on the SB-1, if FP exceptions are disabled, and we are emitting 64-bit code, then we can use both FP pipes. Otherwise, we can only use one FP pipe.

`-mvr4130-align`
`-mno-vr4130-align`

The VR4130 pipeline is two-way superscalar, but can only issue two instructions together if the first one is 8-byte aligned. When this option is enabled, GCC aligns pairs of instructions that it thinks should execute in parallel.

This option only has an effect when optimizing for the VR4130. It normally makes code faster, but at the expense of making it bigger. It is enabled by default at optimization level '`-O3`'.

`-msynci`
`-mno-synci`

Enable (disable) generation of `synci` instructions on architectures that support it. The `synci` instructions (if enabled) are generated when `__builtin__ _clear_cache` is compiled.

This option defaults to '`-mno-synci`', but the default can be overridden by configuring GCC with '`--with-synci`'.

When compiling code for single processor systems, it is generally safe to use `synci`. However, on many multi-core (SMP) systems, it does not invalidate the instruction caches on all cores and may lead to undefined behavior.

`-mrelax-pic-calls`
`-mno-relax-pic-calls`

Try to turn PIC calls that are normally dispatched via register `$25` into direct calls. This is only possible if the linker can resolve the destination at link time and if the destination is within range for a direct call.

'`-mrelax-pic-calls`' is the default if GCC was configured to use an assembler and a linker that support the `.reloc` assembly directive and '`-mexplicit-relocs`' is in effect. With '`-mno-explicit-relocs`', this optimization can be performed by the assembler and the linker alone without help from the compiler.

`-mmcount-ra-address`
`-mno-mcount-ra-address`

Emit (do not emit) code that allows `_mcount` to modify the calling function's return address. When enabled, this option extends the usual `_mcount` interface with a new *ra-address* parameter, which has type `intptr_t *` and is passed in register `$12`. `_mcount` can then modify the return address by doing both of the following:

- Returning the new address in register `$31`.
- Storing the new address in `*ra-address`, if *ra-address* is nonnull.

The default is '`-mno-mcount-ra-address`'.

`-mframe-header-opt`
`-mno-frame-header-opt`
Enable (disable) frame header optimization in the o32 ABI. When using the o32 ABI, calling functions will allocate 16 bytes on the stack for the called function to write out register arguments. When enabled, this optimization will suppress the allocation of the frame header if it can be determined that it is unused.

This optimization is off by default at all optimization levels.

`-mlxc1-sxc1`
`-mno-lxc1-sxc1`
When applicable, enable (disable) the generation of `lwxc1`, `swxc1`, `ldxc1`, `sdxc1` instructions. Enabled by default.

`-mmadd4`
`-mno-madd4`
When applicable, enable (disable) the generation of 4-operand `madd.s`, `madd.d` and related instructions. Enabled by default.

3.18.29 MMIX Options

These options are defined for the MMIX:

`-mlibfuncs`
`-mno-libfuncs`
Specify that intrinsic library functions are being compiled, passing all values in registers, no matter the size.

`-mepsilon`
`-mno-epsilon`
Generate floating-point comparison instructions that compare with respect to the `rE` epsilon register.

`-mabi=mmixware`
`-mabi=gnu`
Generate code that passes function parameters and return values that (in the called function) are seen as registers `$0` and up, as opposed to the GNU ABI which uses global registers `$231` and up.

`-mzero-extend`
`-mno-zero-extend`
When reading data from memory in sizes shorter than 64 bits, use (do not use) zero-extending load instructions by default, rather than sign-extending ones.

`-mknuthdiv`
`-mno-knuthdiv`
Make the result of a division yielding a remainder have the same sign as the divisor. With the default, '`-mno-knuthdiv`', the sign of the remainder follows the sign of the dividend. Both methods are arithmetically valid, the latter being almost exclusively used.

`-mtoplevel-symbols`
`-mno-toplevel-symbols`
Prepend (do not prepend) a '`:`' to all global symbols, so the assembly code can be used with the `PREFIX` assembly directive.

`-melf` Generate an executable in the ELF format, rather than the default '`mmo`' format used by the `mmix` simulator.

`-mbranch-predict`
`-mno-branch-predict`
Use (do not use) the probable-branch instructions, when static branch prediction indicates a probable branch.

`-mbase-addresses`
`-mno-base-addresses`
Generate (do not generate) code that uses *base addresses*. Using a base address automatically generates a request (handled by the assembler and the linker) for a constant to be set up in a global register. The register is used for one or more base address requests within the range 0 to 255 from the value held in the register. The generally leads to short and fast code, but the number of different data items that can be addressed is limited. This means that a program that uses lots of static data may require '`-mno-base-addresses`'.

`-msingle-exit`
`-mno-single-exit`
Force (do not force) generated code to have a single exit point in each function.

3.18.30 MN10300 Options

These '`-m`' options are defined for Matsushita MN10300 architectures:

`-mmult-bug`
Generate code to avoid bugs in the multiply instructions for the MN10300 processors. This is the default.

`-mno-mult-bug`
Do not generate code to avoid bugs in the multiply instructions for the MN10300 processors.

`-mam33` Generate code using features specific to the AM33 processor.

`-mno-am33`
Do not generate code using features specific to the AM33 processor. This is the default.

`-mam33-2` Generate code using features specific to the AM33/2.0 processor.

`-mam34` Generate code using features specific to the AM34 processor.

`-mtune=`*`cpu-type`*
Use the timing characteristics of the indicated CPU type when scheduling instructions. This does not change the targeted processor type. The CPU type must be one of '`mn10300`', '`am33`', '`am33-2`' or '`am34`'.

`-mreturn-pointer-on-d0`
When generating a function that returns a pointer, return the pointer in both `a0` and `d0`. Otherwise, the pointer is returned only in `a0`, and attempts to call such functions without a prototype result in errors. Note that this option is on by default; use '`-mno-return-pointer-on-d0`' to disable it.

`-mno-crt0`
Do not link in the C run-time initialization object file.

`-mrelax` Indicate to the linker that it should perform a relaxation optimization pass to shorten branches, calls and absolute memory addresses. This option only has an effect when used on the command line for the final link step.

This option makes symbolic debugging impossible.

`-mliw` Allow the compiler to generate *Long Instruction Word* instructions if the target is the '`AM33`' or later. This is the default. This option defines the preprocessor macro `__LIW__`.

`-mno-liw` Do not allow the compiler to generate *Long Instruction Word* instructions. This option defines the preprocessor macro `__NO_LIW__`.

`-msetlb` Allow the compiler to generate the *SETLB* and *Lcc* instructions if the target is the '`AM33`' or later. This is the default. This option defines the preprocessor macro `__SETLB__`.

`-mno-setlb`
Do not allow the compiler to generate *SETLB* or *Lcc* instructions. This option defines the preprocessor macro `__NO_SETLB__`.

3.18.31 Moxie Options

`-meb` Generate big-endian code. This is the default for '`moxie-*-*`' configurations.

`-mel` Generate little-endian code.

`-mmul.x` Generate mul.x and umul.x instructions. This is the default for '`moxiebox-*-*`' configurations.

`-mno-crt0`
Do not link in the C run-time initialization object file.

3.18.32 MSP430 Options

These options are defined for the MSP430:

`-masm-hex`
Force assembly output to always use hex constants. Normally such constants are signed decimals, but this option is available for testsuite and/or aesthetic purposes.

`-mmcu=` Select the MCU to target. This is used to create a C preprocessor symbol based upon the MCU name, converted to upper case and pre- and post-fixed with '`__`'. This in turn is used by the '`msp430.h`' header file to select an MCU-specific supplementary header file.

The option also sets the ISA to use. If the MCU name is one that is known to only support the 430 ISA then that is selected, otherwise the 430X ISA is selected. A generic MCU name of '`msp430`' can also be used to select the 430 ISA. Similarly the generic '`msp430x`' MCU name selects the 430X ISA.

In addition an MCU-specific linker script is added to the linker command line. The script's name is the name of the MCU with '`.ld`' appended. Thus specifying '`-mmcu=xxx`' on the `gcc` command line defines the C preprocessor symbol `__XXX__` and cause the linker to search for a script called '`xxx.ld`'.

This option is also passed on to the assembler.

`-mwarn-mcu`
`-mno-warn-mcu`

This option enables or disables warnings about conflicts between the MCU name specified by the '`-mmcu`' option and the ISA set by the '`-mcpu`' option and/or the hardware multiply support set by the '`-mhwmult`' option. It also toggles warnings about unrecognized MCU names. This option is on by default.

`-mcpu=` Specifies the ISA to use. Accepted values are '`msp430`', '`msp430x`' and '`msp430xv2`'. This option is deprecated. The '`-mmcu=`' option should be used to select the ISA.

`-msim` Link to the simulator runtime libraries and linker script. Overrides any scripts that would be selected by the '`-mmcu=`' option.

`-mlarge` Use large-model addressing (20-bit pointers, 32-bit `size_t`).

`-msmall` Use small-model addressing (16-bit pointers, 16-bit `size_t`).

`-mrelax` This option is passed to the assembler and linker, and allows the linker to perform certain optimizations that cannot be done until the final link.

`mhwmult=` Describes the type of hardware multiply supported by the target. Accepted values are '`none`' for no hardware multiply, '`16bit`' for the original 16-bit-only multiply supported by early MCUs. '`32bit`' for the 16/32-bit multiply supported by later MCUs and '`f5series`' for the 16/32-bit multiply supported by F5-series MCUs. A value of '`auto`' can also be given. This tells GCC to deduce the hardware multiply support based upon the MCU name provided by the '`-mmcu`' option. If no '`-mmcu`' option is specified or if the MCU name is not recognized then no hardware multiply support is assumed. `auto` is the default setting.

Hardware multiplies are normally performed by calling a library routine. This saves space in the generated code. When compiling at '`-O3`' or higher however the hardware multiplier is invoked inline. This makes for bigger, but faster code.

The hardware multiply routines disable interrupts whilst running and restore the previous interrupt state when they finish. This makes them safe to use inside interrupt handlers as well as in normal code.

`-minrt` Enable the use of a minimum runtime environment - no static initializers or constructors. This is intended for memory-constrained devices. The compiler

includes special symbols in some objects that tell the linker and runtime which code fragments are required.

`-mcode-region=`
`-mdata-region=`
These options tell the compiler where to place functions and data that do not have one of the `lower`, `upper`, `either` or `section` attributes. Possible values are `lower`, `upper`, `either` or `any`. The first three behave like the corresponding attribute. The fourth possible value - `any` - is the default. It leaves placement entirely up to the linker script and how it assigns the standard sections (`.text`, `.data`, etc) to the memory regions.

`-msilicon-errata=`
This option passes on a request to assembler to enable the fixes for the named silicon errata.

`-msilicon-errata-warn=`
This option passes on a request to the assembler to enable warning messages when a silicon errata might need to be applied.

3.18.33 NDS32 Options

These options are defined for NDS32 implementations:

`-mbig-endian`
Generate code in big-endian mode.

`-mlittle-endian`
Generate code in little-endian mode.

`-mreduced-regs`
Use reduced-set registers for register allocation.

`-mfull-regs`
Use full-set registers for register allocation.

`-mcmov` Generate conditional move instructions.

`-mno-cmov`
Do not generate conditional move instructions.

`-mext-perf`
Generate performance extension instructions.

`-mno-ext-perf`
Do not generate performance extension instructions.

`-mext-perf2`
Generate performance extension 2 instructions.

`-mno-ext-perf2`
Do not generate performance extension 2 instructions.

`-mext-string`
Generate string extension instructions.

`-mno-ext-string`
: Do not generate string extension instructions.

`-mv3push` Generate v3 push25/pop25 instructions.

`-mno-v3push`
: Do not generate v3 push25/pop25 instructions.

`-m16-bit` Generate 16-bit instructions.

`-mno-16-bit`
: Do not generate 16-bit instructions.

`-misr-vector-size=`*num*
: Specify the size of each interrupt vector, which must be 4 or 16.

`-mcache-block-size=`*num*
: Specify the size of each cache block, which must be a power of 2 between 4 and 512.

`-march=`*arch*
: Specify the name of the target architecture.

`-mcmodel=`*code-model*
: Set the code model to one of

  ‘`small`’ All the data and read-only data segments must be within 512KB addressing space. The text segment must be within 16MB addressing space.

  ‘`medium`’ The data segment must be within 512KB while the read-only data segment can be within 4GB addressing space. The text segment should be still within 16MB addressing space.

  ‘`large`’ All the text and data segments can be within 4GB addressing space.

`-mctor-dtor`
: Enable constructor/destructor feature.

`-mrelax` Guide linker to relax instructions.

3.18.34 Nios II Options

These are the options defined for the Altera Nios II processor.

`-G` *num* Put global and static objects less than or equal to *num* bytes into the small data or BSS sections instead of the normal data or BSS sections. The default value of *num* is 8.

`-mgpopt=`*option*
`-mgpopt`
`-mno-gpopt`
: Generate (do not generate) GP-relative accesses. The following *option* names are recognized:

  ‘`none`’ Do not generate GP-relative accesses.

'`local`' Generate GP-relative accesses for small data objects that are not external, weak, or uninitialized common symbols. Also use GP-relative addressing for objects that have been explicitly placed in a small data section via a `section` attribute.

'`global`' As for '`local`', but also generate GP-relative accesses for small data objects that are external, weak, or common. If you use this option, you must ensure that all parts of your program (including libraries) are compiled with the same '`-G`' setting.

'`data`' Generate GP-relative accesses for all data objects in the program. If you use this option, the entire data and BSS segments of your program must fit in 64K of memory and you must use an appropriate linker script to allocate them within the addressable range of the global pointer.

'`all`' Generate GP-relative addresses for function pointers as well as data pointers. If you use this option, the entire text, data, and BSS segments of your program must fit in 64K of memory and you must use an appropriate linker script to allocate them within the addressable range of the global pointer.

'`-mgpopt`' is equivalent to '`-mgpopt=local`', and '`-mno-gpopt`' is equivalent to '`-mgpopt=none`'.

The default is '`-mgpopt`' except when '`-fpic`' or '`-fPIC`' is specified to generate position-independent code. Note that the Nios II ABI does not permit GP-relative accesses from shared libraries.

You may need to specify '`-mno-gpopt`' explicitly when building programs that include large amounts of small data, including large GOT data sections. In this case, the 16-bit offset for GP-relative addressing may not be large enough to allow access to the entire small data section.

`-mgprel-sec=`*regexp*

This option specifies additional section names that can be accessed via GP-relative addressing. It is most useful in conjunction with `section` attributes on variable declarations (see Section 6.34.1 [Common Variable Attributes], page 536) and a custom linker script. The *regexp* is a POSIX Extended Regular Expression.

This option does not affect the behavior of the '`-G`' option, and the specified sections are in addition to the standard `.sdata` and `.sbss` small-data sections that are recognized by '`-mgpopt`'.

`-mr0rel-sec=`*regexp*

This option specifies names of sections that can be accessed via a 16-bit offset from `r0`; that is, in the low 32K or high 32K of the 32-bit address space. It is most useful in conjunction with `section` attributes on variable declarations (see Section 6.34.1 [Common Variable Attributes], page 536) and a custom linker script. The *regexp* is a POSIX Extended Regular Expression.

In contrast to the use of GP-relative addressing for small data, zero-based addressing is never generated by default and there are no conventional section

names used in standard linker scripts for sections in the low or high areas of memory.

`-mel`
`-meb` Generate little-endian (default) or big-endian (experimental) code, respectively.

`-march=arch`
This specifies the name of the target Nios II architecture. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. Permissible names are: '`r1`', '`r2`'.

The preprocessor macro `__nios2_arch__` is available to programs, with value 1 or 2, indicating the targeted ISA level.

`-mbypass-cache`
`-mno-bypass-cache`
Force all load and store instructions to always bypass cache by using I/O variants of the instructions. The default is not to bypass the cache.

`-mno-cache-volatile`
`-mcache-volatile`
Volatile memory access bypass the cache using the I/O variants of the load and store instructions. The default is not to bypass the cache.

`-mno-fast-sw-div`
`-mfast-sw-div`
Do not use table-based fast divide for small numbers. The default is to use the fast divide at '`-O3`' and above.

`-mno-hw-mul`
`-mhw-mul`
`-mno-hw-mulx`
`-mhw-mulx`
`-mno-hw-div`
`-mhw-div` Enable or disable emitting `mul`, `mulx` and `div` family of instructions by the compiler. The default is to emit `mul` and not emit `div` and `mulx`.

`-mbmx`
`-mno-bmx`
`-mcdx`
`-mno-cdx` Enable or disable generation of Nios II R2 BMX (bit manipulation) and CDX (code density) instructions. Enabling these instructions also requires '`-march=r2`'. Since these instructions are optional extensions to the R2 architecture, the default is not to emit them.

`-mcustom-insn=N`
`-mno-custom-insn`
Each '`-mcustom-insn=N`' option enables use of a custom instruction with encoding *N* when generating code that uses *insn*. For example, '`-mcustom-fadds=253`' generates custom instruction 253 for single-precision floating-point add operations instead of the default behavior of using a library call.

The following values of *insn* are supported. Except as otherwise noted, floating-point operations are expected to be implemented with normal IEEE 754 semantics and correspond directly to the C operators or the equivalent GCC built-in functions (see Section 6.59 [Other Builtins], page 641).

Single-precision floating point:

'`fadds`', '`fsubs`', '`fdivs`', '`fmuls`'
: Binary arithmetic operations.

'`fnegs`' Unary negation.

'`fabss`' Unary absolute value.

'`fcmpeqs`', '`fcmpges`', '`fcmpgts`', '`fcmples`', '`fcmplts`', '`fcmpnes`'
: Comparison operations.

'`fmins`', '`fmaxs`'
: Floating-point minimum and maximum. These instructions are only generated if '`-ffinite-math-only`' is specified.

'`fsqrts`' Unary square root operation.

'`fcoss`', '`fsins`', '`ftans`', '`fatans`', '`fexps`', '`flogs`'
: Floating-point trigonometric and exponential functions. These instructions are only generated if '`-funsafe-math-optimizations`' is also specified.

Double-precision floating point:

'`faddd`', '`fsubd`', '`fdivd`', '`fmuld`'
: Binary arithmetic operations.

'`fnegd`' Unary negation.

'`fabsd`' Unary absolute value.

'`fcmpeqd`', '`fcmpged`', '`fcmpgtd`', '`fcmpled`', '`fcmpltd`', '`fcmpned`'
: Comparison operations.

'`fmind`', '`fmaxd`'
: Double-precision minimum and maximum. These instructions are only generated if '`-ffinite-math-only`' is specified.

'`fsqrtd`' Unary square root operation.

'`fcosd`', '`fsind`', '`ftand`', '`fatand`', '`fexpd`', '`flogd`'
: Double-precision trigonometric and exponential functions. These instructions are only generated if '`-funsafe-math-optimizations`' is also specified.

Conversions:

'`fextsd`' Conversion from single precision to double precision.

'`ftruncds`'
: Conversion from double precision to single precision.

'fixsi', 'fixsu', 'fixdi', 'fixdu'
: Conversion from floating point to signed or unsigned integer types, with truncation towards zero.

'round'
: Conversion from single-precision floating point to signed integer, rounding to the nearest integer and ties away from zero. This corresponds to the `__builtin_lroundf` function when '`-fno-math-errno`' is used.

'floatis', 'floatus', 'floatid', 'floatud'
: Conversion from signed or unsigned integer types to floating-point types.

In addition, all of the following transfer instructions for internal registers X and Y must be provided to use any of the double-precision floating-point instructions. Custom instructions taking two double-precision source operands expect the first operand in the 64-bit register X. The other operand (or only operand of a unary operation) is given to the custom arithmetic instruction with the least significant half in source register *src1* and the most significant half in *src2*. A custom instruction that returns a double-precision result returns the most significant 32 bits in the destination register and the other half in 32-bit register Y. GCC automatically generates the necessary code sequences to write register X and/or read register Y when double-precision floating-point instructions are used.

'fwrx'
: Write *src1* into the least significant half of X and *src2* into the most significant half of X.

'fwry'
: Write *src1* into Y.

'frdxhi', 'frdxlo'
: Read the most or least (respectively) significant half of X and store it in *dest*.

'frdy'
: Read the value of Y and store it into *dest*.

Note that you can gain more local control over generation of Nios II custom instructions by using the `target("custom-insn=N")` and `target("no-custom-insn")` function attributes (see Section 6.33 [Function Attributes], page 477) or pragmas (see Section 6.62.15 [Function Specific Option Pragmas], page 808).

`-mcustom-fpu-cfg=name`
: This option enables a predefined, named set of custom instruction encodings (see '`-mcustom-insn`' above). Currently, the following sets are defined:

'`-mcustom-fpu-cfg=60-1`' is equivalent to:

```
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-fsingle-precision-constant
```

'`-mcustom-fpu-cfg=60-2`' is equivalent to:

```
-mcustom-fmuls=252
-mcustom-fadds=253
```

```
-mcustom-fsubs=254
-mcustom-fdivs=255
-fsingle-precision-constant
```

'-mcustom-fpu-cfg=72-3' is equivalent to:

```
-mcustom-floatus=243
-mcustom-fixsi=244
-mcustom-floatis=245
-mcustom-fcmpgts=246
-mcustom-fcmples=249
-mcustom-fcmpeqs=250
-mcustom-fcmpnes=251
-mcustom-fmuls=252
-mcustom-fadds=253
-mcustom-fsubs=254
-mcustom-fdivs=255
-fsingle-precision-constant
```

Custom instruction assignments given by individual '-mcustom-*insn*=' options override those given by '-mcustom-fpu-cfg=', regardless of the order of the options on the command line.

Note that you can gain more local control over selection of a FPU configuration by using the `target("custom-fpu-cfg=name")` function attribute (see Section 6.33 [Function Attributes], page 477) or pragma (see Section 6.62.15 [Function Specific Option Pragmas], page 808).

These additional '-m' options are available for the Altera Nios II ELF (bare-metal) target:

-mhal
: Link with HAL BSP. This suppresses linking with the GCC-provided C runtime startup and termination code, and is typically used in conjunction with '-msys-crt0=' to specify the location of the alternate startup code provided by the HAL BSP.

-msmallc
: Link with a limited version of the C library, '-lsmallc', rather than Newlib.

-msys-crt0=*startfile*
: *startfile* is the file name of the startfile (crt0) to use when linking. This option is only useful in conjunction with '-mhal'.

-msys-lib=*systemlib*
: *systemlib* is the library name of the library that provides low-level system calls required by the C library, e.g. `read` and `write`. This option is typically used to link with a library provided by a HAL BSP.

3.18.35 Nvidia PTX Options

These options are defined for Nvidia PTX:

-m32
-m64
: Generate code for 32-bit or 64-bit ABI.

-misa=*ISA-string*
: Generate code for given the specified PTX ISA (e.g. 'sm_35'). ISA strings must be lower-case. Valid ISA strings include 'sm_30' and 'sm_35'. The default ISA is sm_30.

`-mmainkernel`
: Link in code for a __main kernel. This is for stand-alone instead of offloading execution.

`-moptimize`
: Apply partitioned execution optimizations. This is the default when any level of optimization is selected.

`-msoft-stack`
: Generate code that does not use `.local` memory directly for stack storage. Instead, a per-warp stack pointer is maintained explicitly. This enables variable-length stack allocation (with variable-length arrays or `alloca`), and when global memory is used for underlying storage, makes it possible to access automatic variables from other threads, or with atomic instructions. This code generation variant is used for OpenMP offloading, but the option is exposed on its own for the purpose of testing the compiler; to generate code suitable for linking into programs using OpenMP offloading, use option '`-mgomp`'.

`-muniform-simt`
: Switch to code generation variant that allows to execute all threads in each warp, while maintaining memory state and side effects as if only one thread in each warp was active outside of OpenMP SIMD regions. All atomic operations and calls to runtime (malloc, free, vprintf) are conditionally executed (iff current lane index equals the master lane index), and the register being assigned is copied via a shuffle instruction from the master lane. Outside of SIMD regions lane 0 is the master; inside, each thread sees itself as the master. Shared memory array `int __nvptx_uni[]` stores all-zeros or all-ones bitmasks for each warp, indicating current mode (0 outside of SIMD regions). Each thread can bitwise-and the bitmask at position `tid.y` with current lane index to compute the master lane index.

`-mgomp`
: Generate code for use in OpenMP offloading: enables '`-msoft-stack`' and '`-muniform-simt`' options, and selects corresponding multilib variant.

3.18.36 OpenRISC Options

These options are defined for OpenRISC:

`-mboard=`*`name`*
: Configure a board specific runtime. This will be passed to the linker for newlib board library linking. The default is `or1ksim`.

`-mnewlib`
: For compatibility, it's always newlib for elf now.

`-mhard-div`
: Generate code for hardware which supports divide instructions. This is the default.

`-mhard-mul`
: Generate code for hardware which supports multiply instructions. This is the default.

`-mcmov`
: Generate code for hardware which supports the conditional move (`l.cmov`) instruction.

`-mror` Generate code for hardware which supports rotate right instructions.

`-msext` Generate code for hardware which supports sign-extension instructions.

`-msfimm` Generate code for hardware which supports set flag immediate (`l.sf*i`) instructions.

`-mshftimm`
Generate code for hardware which supports shift immediate related instructions (i.e. `l.srai`, `l.srli`, `l.slli`, `1.rori`). Note, to enable generation of the `l.rori` instruction the '`-mror`' flag must also be specified.

`-msoft-div`
Generate code for hardware which requires divide instruction emulation.

`-msoft-mul`
Generate code for hardware which requires multiply instruction emulation.

3.18.37 PDP-11 Options

These options are defined for the PDP-11:

`-mfpu` Use hardware FPP floating point. This is the default. (FIS floating point on the PDP-11/40 is not supported.) Implies -m45.

`-msoft-float`
Do not use hardware floating point.

`-mac0` Return floating-point results in ac0 (fr0 in Unix assembler syntax).

`-mno-ac0` Return floating-point results in memory. This is the default.

`-m40` Generate code for a PDP-11/40. Implies -msoft-float -mno-split.

`-m45` Generate code for a PDP-11/45. This is the default.

`-m10` Generate code for a PDP-11/10. Implies -msoft-float -mno-split.

`-mint16`
`-mno-int32`
Use 16-bit `int`. This is the default.

`-mint32`
`-mno-int16`
Use 32-bit `int`.

`-msplit` Target has split instruction and data space. Implies -m45.

`-munix-asm`
Use Unix assembler syntax.

`-mdec-asm`
Use DEC assembler syntax.

`-mgnu-asm`
Use GNU assembler syntax. This is the default.

`-mlra` Use the new LRA register allocator. By default, the old "reload" allocator is used.

3.18.38 picoChip Options

These '-m' options are defined for picoChip implementations:

-mae=*ae_type*

Set the instruction set, register set, and instruction scheduling parameters for array element type *ae_type*. Supported values for *ae_type* are 'ANY', 'MUL', and 'MAC'.

'-mae=ANY' selects a completely generic AE type. Code generated with this option runs on any of the other AE types. The code is not as efficient as it would be if compiled for a specific AE type, and some types of operation (e.g., multiplication) do not work properly on all types of AE.

'-mae=MUL' selects a MUL AE type. This is the most useful AE type for compiled code, and is the default.

'-mae=MAC' selects a DSP-style MAC AE. Code compiled with this option may suffer from poor performance of byte (char) manipulation, since the DSP AE does not provide hardware support for byte load/stores.

-msymbol-as-address

Enable the compiler to directly use a symbol name as an address in a load/store instruction, without first loading it into a register. Typically, the use of this option generates larger programs, which run faster than when the option isn't used. However, the results vary from program to program, so it is left as a user option, rather than being permanently enabled.

-mno-inefficient-warnings

Disables warnings about the generation of inefficient code. These warnings can be generated, for example, when compiling code that performs byte-level memory operations on the MAC AE type. The MAC AE has no hardware support for byte-level memory operations, so all byte load/stores must be synthesized from word load/store operations. This is inefficient and a warning is generated to indicate that you should rewrite the code to avoid byte operations, or to target an AE type that has the necessary hardware support. This option disables these warnings.

3.18.39 PowerPC Options

These are listed under See Section 3.18.42 [RS/6000 and PowerPC Options], page 358.

3.18.40 RISC-V Options

These command-line options are defined for RISC-V targets:

-mbranch-cost=*n*

Set the cost of branches to roughly *n* instructions.

-mplt
-mno-plt When generating PIC code, do or don't allow the use of PLTs. Ignored for non-PIC. The default is '-mplt'.

-mabi=*ABI-string*

Specify integer and floating-point calling convention. *ABI-string* contains two parts: the size of integer types and the registers used for floating-point types.

For example '`-march=rv64ifd -mabi=lp64d`' means that '`long`' and pointers are 64-bit (implicitly defining '`int`' to be 32-bit), and that floating-point values up to 64 bits wide are passed in F registers. Contrast this with '`-march=rv64ifd -mabi=lp64f`', which still allows the compiler to generate code that uses the F and D extensions but only allows floating-point values up to 32 bits long to be passed in registers; or '`-march=rv64ifd -mabi=lp64`', in which no floating-point arguments will be passed in registers.

The default for this argument is system dependent, users who want a specific calling convention should specify one explicitly. The valid calling conventions are: '`ilp32`', '`ilp32f`', '`ilp32d`', '`lp64`', '`lp64f`', and '`lp64d`'. Some calling conventions are impossible to implement on some ISAs: for example, '`-march=rv32if -mabi=ilp32d`' is invalid because the ABI requires 64-bit values be passed in F registers, but F registers are only 32 bits wide. There is also the '`ilp32e`' ABI that can only be used with the '`rv32e`' architecture. This ABI is not well specified at present, and is subject to change.

`-mfdiv`
`-mno-fdiv`

Do or don't use hardware floating-point divide and square root instructions. This requires the F or D extensions for floating-point registers. The default is to use them if the specified architecture has these instructions.

`-mdiv`
`-mno-div` Do or don't use hardware instructions for integer division. This requires the M extension. The default is to use them if the specified architecture has these instructions.

`-march=`*`ISA-string`*

Generate code for given RISC-V ISA (e.g. '`rv64im`'). ISA strings must be lower-case. Examples include '`rv64i`', '`rv32g`', '`rv32e`', and '`rv32imaf`'.

`-mtune=`*`processor-string`*

Optimize the output for the given processor, specified by microarchitecture name. Permissible values for this option are: '`rocket`', '`sifive-3-series`', '`sifive-5-series`', '`sifive-7-series`', and '`size`'.

When '`-mtune=`' is not specified, the default is '`rocket`'.

The '`size`' choice is not intended for use by end-users. This is used when '`-Os`' is specified. It overrides the instruction cost info provided by '`-mtune=`', but does not override the pipeline info. This helps reduce code size while still giving good performance.

`-mpreferred-stack-boundary=`*`num`*

Attempt to keep the stack boundary aligned to a 2 raised to *num* byte boundary. If '`-mpreferred-stack-boundary`' is not specified, the default is 4 (16 bytes or 128-bits).

Warning: If you use this switch, then you must build all modules with the same value, including any libraries. This includes the system libraries and startup modules.

`-msmall-data-limit=n`
: Put global and static data smaller than *n* bytes into a special section (on some targets).

`-msave-restore`
`-mno-save-restore`
: Do or don't use smaller but slower prologue and epilogue code that uses library function calls. The default is to use fast inline prologues and epilogues.

`-mstrict-align`
`-mno-strict-align`
: Do not or do generate unaligned memory accesses. The default is set depending on whether the processor we are optimizing for supports fast unaligned access or not.

`-mcmodel=medlow`
: Generate code for the medium-low code model. The program and its statically defined symbols must lie within a single 2 GiB address range and must lie between absolute addresses −2 GiB and +2 GiB. Programs can be statically or dynamically linked. This is the default code model.

`-mcmodel=medany`
: Generate code for the medium-any code model. The program and its statically defined symbols must be within any single 2 GiB address range. Programs can be statically or dynamically linked.

`-mexplicit-relocs`
`-mno-exlicit-relocs`
: Use or do not use assembler relocation operators when dealing with symbolic addresses. The alternative is to use assembler macros instead, which may limit optimization.

`-mrelax`
`-mno-relax`
: Take advantage of linker relaxations to reduce the number of instructions required to materialize symbol addresses. The default is to take advantage of linker relaxations.

`-memit-attribute`
`-mno-emit-attribute`
: Emit (do not emit) RISC-V attribute to record extra information into ELF objects. This feature requires at least binutils 2.32.

3.18.41 RL78 Options

`-msim` Links in additional target libraries to support operation within a simulator.

`-mmul=none`
`-mmul=g10`
`-mmul=g13`
`-mmul=g14`
`-mmul=rl78`

Specifies the type of hardware multiplication and division support to be used. The simplest is `none`, which uses software for both multiplication and division. This is the default. The `g13` value is for the hardware multiply/divide peripheral found on the RL78/G13 (S2 core) targets. The `g14` value selects the use of the multiplication and division instructions supported by the RL78/G14 (S3 core) parts. The value `rl78` is an alias for `g14` and the value `mg10` is an alias for `none`.

In addition a C preprocessor macro is defined, based upon the setting of this option. Possible values are: `__RL78_MUL_NONE__`, `__RL78_MUL_G13__` or `__RL78_MUL_G14__`.

`-mcpu=g10`
`-mcpu=g13`
`-mcpu=g14`
`-mcpu=rl78`

Specifies the RL78 core to target. The default is the G14 core, also known as an S3 core or just RL78. The G13 or S2 core does not have multiply or divide instructions, instead it uses a hardware peripheral for these operations. The G10 or S1 core does not have register banks, so it uses a different calling convention.

If this option is set it also selects the type of hardware multiply support to use, unless this is overridden by an explicit '`-mmul=none`' option on the command line. Thus specifying '`-mcpu=g13`' enables the use of the G13 hardware multiply peripheral and specifying '`-mcpu=g10`' disables the use of hardware multiplications altogether.

Note, although the RL78/G14 core is the default target, specifying '`-mcpu=g14`' or '`-mcpu=rl78`' on the command line does change the behavior of the toolchain since it also enables G14 hardware multiply support. If these options are not specified on the command line then software multiplication routines will be used even though the code targets the RL78 core. This is for backwards compatibility with older toolchains which did not have hardware multiply and divide support.

In addition a C preprocessor macro is defined, based upon the setting of this option. Possible values are: `__RL78_G10__`, `__RL78_G13__` or `__RL78_G14__`.

`-mg10`
`-mg13`
`-mg14`
`-mrl78`

These are aliases for the corresponding '`-mcpu=`' option. They are provided for backwards compatibility.

`-mallregs`
: Allow the compiler to use all of the available registers. By default registers `r24..r31` are reserved for use in interrupt handlers. With this option enabled these registers can be used in ordinary functions as well.

`-m64bit-doubles`
`-m32bit-doubles`
: Make the `double` data type be 64 bits ('`-m64bit-doubles`') or 32 bits ('`-m32bit-doubles`') in size. The default is '`-m32bit-doubles`'.

`-msave-mduc-in-interrupts`
`-mno-save-mduc-in-interrupts`
: Specifies that interrupt handler functions should preserve the MDUC registers. This is only necessary if normal code might use the MDUC registers, for example because it performs multiplication and division operations. The default is to ignore the MDUC registers as this makes the interrupt handlers faster. The target option -mg13 needs to be passed for this to work as this feature is only available on the G13 target (S2 core). The MDUC registers will only be saved if the interrupt handler performs a multiplication or division operation or it calls another function.

3.18.42 IBM RS/6000 and PowerPC Options

These '`-m`' options are defined for the IBM RS/6000 and PowerPC:

`-mpowerpc-gpopt`
`-mno-powerpc-gpopt`
`-mpowerpc-gfxopt`
`-mno-powerpc-gfxopt`
`-mpowerpc64`
`-mno-powerpc64`
`-mmfcrf`
`-mno-mfcrf`
`-mpopcntb`
`-mno-popcntb`
`-mpopcntd`
`-mno-popcntd`
`-mfprnd`
`-mno-fprnd`
`-mcmpb`
`-mno-cmpb`
`-mmfpgpr`
`-mno-mfpgpr`
`-mhard-dfp`
`-mno-hard-dfp`
: You use these options to specify which instructions are available on the processor you are using. The default value of these options is determined when configuring GCC. Specifying the '`-mcpu=`*`cpu_type`*' overrides the specification of these options. We recommend you use the '`-mcpu=`*`cpu_type`*' option rather than the options listed above.

Specifying '-mpowerpc-gpopt' allows GCC to use the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root. Specifying '-mpowerpc-gfxopt' allows GCC to use the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

The '-mmfcrf' option allows GCC to generate the move from condition register field instruction implemented on the POWER4 processor and other processors that support the PowerPC V2.01 architecture. The '-mpopcntb' option allows GCC to generate the popcount and double-precision FP reciprocal estimate instruction implemented on the POWER5 processor and other processors that support the PowerPC V2.02 architecture. The '-mpopcntd' option allows GCC to generate the popcount instruction implemented on the POWER7 processor and other processors that support the PowerPC V2.06 architecture. The '-mfprnd' option allows GCC to generate the FP round to integer instructions implemented on the POWER5+ processor and other processors that support the PowerPC V2.03 architecture. The '-mcmpb' option allows GCC to generate the compare bytes instruction implemented on the POWER6 processor and other processors that support the PowerPC V2.05 architecture. The '-mmfpgpr' option allows GCC to generate the FP move to/from general-purpose register instructions implemented on the POWER6X processor and other processors that support the extended PowerPC V2.05 architecture. The '-mhard-dfp' option allows GCC to generate the decimal floating-point instructions implemented on some POWER processors.

The '-mpowerpc64' option allows GCC to generate the additional 64-bit instructions that are found in the full PowerPC64 architecture and to treat GPRs as 64-bit, doubleword quantities. GCC defaults to '-mno-powerpc64'.

-mcpu=*cpu_type*

Set architecture type, register usage, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '401', '403', '405', '405fp', '440', '440fp', '464', '464fp', '476', '476fp', '505', '601', '602', '603', '603e', '604', '604e', '620', '630', '740', '7400', '7450', '750', '801', '821', '823', '860', '970', '8540', 'a2', 'e300c2', 'e300c3', 'e500mc', 'e500mc64', 'e5500', 'e6500', 'ec603e', 'G3', 'G4', 'G5', 'titan', 'power3', 'power4', 'power5', 'power5+', 'power6', 'power6x', 'power7', 'power8', 'power9', 'powerpc', 'powerpc64', 'powerpc64le', 'rs64', and 'native'.

'-mcpu=powerpc', '-mcpu=powerpc64', and '-mcpu=powerpc64le' specify pure 32-bit PowerPC (either endian), 64-bit big endian PowerPC and 64-bit little endian PowerPC architecture machine types, with an appropriate, generic processor model assumed for scheduling purposes.

Specifying 'native' as cpu type detects and selects the architecture option that corresponds to the host processor of the system performing the compilation. '-mcpu=native' has no effect if GCC does not recognize the processor.

The other options specify a specific processor. Code generated under those options runs best on that processor, and may not run at all on others.

The '-mcpu' options automatically enable or disable the following options:

```
-maltivec -mfprnd -mhard-float -mmfcrf -mmultiple
-mpopcntb -mpopcntd -mpowerpc64
-mpowerpc-gpopt -mpowerpc-gfxopt
-mmulhw -mdlmzb -mmfpgpr -mvsx
-mcrypto -mhtm -mpower8-fusion -mpower8-vector
-mquad-memory -mquad-memory-atomic -mfloat128 -mfloat128-hardware
```

The particular options set for any particular CPU varies between compiler versions, depending on what setting seems to produce optimal code for that CPU; it doesn't necessarily reflect the actual hardware's capabilities. If you wish to set an individual option to a particular value, you may specify it after the '`-mcpu`' option, like '`-mcpu=970 -mno-altivec`'.

On AIX, the '`-maltivec`' and '`-mpowerpc64`' options are not enabled or disabled by the '`-mcpu`' option at present because AIX does not have full support for these options. You may still enable or disable them individually if you're sure it'll work in your environment.

`-mtune=`*`cpu_type`*

Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the architecture type or register usage, as '`-mcpu=`*`cpu_type`*' does. The same values for *cpu_type* are used for '`-mtune`' as for '`-mcpu`'. If both are specified, the code generated uses the architecture and registers set by '`-mcpu`', but the scheduling parameters set by '`-mtune`'.

`-mcmodel=small`

Generate PowerPC64 code for the small model: The TOC is limited to 64k.

`-mcmodel=medium`

Generate PowerPC64 code for the medium model: The TOC and other static data may be up to a total of 4G in size. This is the default for 64-bit Linux.

`-mcmodel=large`

Generate PowerPC64 code for the large model: The TOC may be up to 4G in size. Other data and code is only limited by the 64-bit address space.

`-maltivec`
`-mno-altivec`

Generate code that uses (does not use) AltiVec instructions, and also enable the use of built-in functions that allow more direct access to the AltiVec instruction set. You may also need to set '`-mabi=altivec`' to adjust the current ABI with AltiVec ABI enhancements.

When '`-maltivec`' is used, the element order for AltiVec intrinsics such as `vec_splat`, `vec_extract`, and `vec_insert` match array element order corresponding to the endianness of the target. That is, element zero identifies the leftmost element in a vector register when targeting a big-endian platform, and identifies the rightmost element in a vector register when targeting a little-endian platform.

`-mvrsave`
`-mno-vrsave`

Generate VRSAVE instructions when generating AltiVec code.

`-msecure-plt`
: Generate code that allows `ld` and `ld.so` to build executables and shared libraries with non-executable `.plt` and `.got` sections. This is a PowerPC 32-bit SYSV ABI option.

`-mbss-plt`
: Generate code that uses a BSS `.plt` section that `ld.so` fills in, and requires `.plt` and `.got` sections that are both writable and executable. This is a PowerPC 32-bit SYSV ABI option.

`-misel`
`-mno-isel`
: This switch enables or disables the generation of ISEL instructions.

`-mvsx`
`-mno-vsx`
: Generate code that uses (does not use) vector/scalar (VSX) instructions, and also enable the use of built-in functions that allow more direct access to the VSX instruction set.

`-mcrypto`
`-mno-crypto`
: Enable the use (disable) of the built-in functions that allow direct access to the cryptographic instructions that were added in version 2.07 of the PowerPC ISA.

`-mhtm`
`-mno-htm`
: Enable (disable) the use of the built-in functions that allow direct access to the Hardware Transactional Memory (HTM) instructions that were added in version 2.07 of the PowerPC ISA.

`-mpower8-fusion`
`-mno-power8-fusion`
: Generate code that keeps (does not keeps) some integer operations adjacent so that the instructions can be fused together on power8 and later processors.

`-mpower8-vector`
`-mno-power8-vector`
: Generate code that uses (does not use) the vector and scalar instructions that were added in version 2.07 of the PowerPC ISA. Also enable the use of built-in functions that allow more direct access to the vector instructions.

`-mquad-memory`
`-mno-quad-memory`
: Generate code that uses (does not use) the non-atomic quad word memory instructions. The '`-mquad-memory`' option requires use of 64-bit mode.

`-mquad-memory-atomic`
`-mno-quad-memory-atomic`
: Generate code that uses (does not use) the atomic quad word memory instructions. The '`-mquad-memory-atomic`' option requires use of 64-bit mode.

`-mfloat128`
`-mno-float128`

Enable/disable the *__float128* keyword for IEEE 128-bit floating point and use either software emulation for IEEE 128-bit floating point or hardware instructions.

The VSX instruction set ('`-mvsx`', '`-mcpu=power7`', '`-mcpu=power8`'), or '`-mcpu=power9`' must be enabled to use the IEEE 128-bit floating point support. The IEEE 128-bit floating point support only works on PowerPC Linux systems.

The default for '`-mfloat128`' is enabled on PowerPC Linux systems using the VSX instruction set, and disabled on other systems.

If you use the ISA 3.0 instruction set ('`-mpower9-vector`' or '`-mcpu=power9`') on a 64-bit system, the IEEE 128-bit floating point support will also enable the generation of ISA 3.0 IEEE 128-bit floating point instructions. Otherwise, if you do not specify to generate ISA 3.0 instructions or you are targeting a 32-bit big endian system, IEEE 128-bit floating point will be done with software emulation.

`-mfloat128-hardware`
`-mno-float128-hardware`

Enable/disable using ISA 3.0 hardware instructions to support the *__float128* data type.

The default for '`-mfloat128-hardware`' is enabled on PowerPC Linux systems using the ISA 3.0 instruction set, and disabled on other systems.

`-m32`
`-m64`

Generate code for 32-bit or 64-bit environments of Darwin and SVR4 targets (including GNU/Linux). The 32-bit environment sets int, long and pointer to 32 bits and generates code that runs on any PowerPC variant. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits, and generates code for PowerPC64, as for '`-mpowerpc64`'.

`-mfull-toc`
`-mno-fp-in-toc`
`-mno-sum-in-toc`
`-mminimal-toc`

Modify generation of the TOC (Table Of Contents), which is created for every executable file. The '`-mfull-toc`' option is selected by default. In that case, GCC allocates at least one TOC entry for each unique non-automatic variable reference in your program. GCC also places floating-point constants in the TOC. However, only 16,384 entries are available in the TOC.

If you receive a linker error message that saying you have overflowed the available TOC space, you can reduce the amount of TOC space used with the '`-mno-fp-in-toc`' and '`-mno-sum-in-toc`' options. '`-mno-fp-in-toc`' prevents GCC from putting floating-point constants in the TOC and '`-mno-sum-in-toc`' forces GCC to generate code to calculate the sum of an address and a constant at run time instead of putting that sum into the TOC. You may specify one

or both of these options. Each causes GCC to produce very slightly slower and larger code at the expense of conserving TOC space.

If you still run out of space in the TOC even when you specify both of these options, specify '`-mminimal-toc`' instead. This option causes GCC to make only one TOC entry for every file. When you specify this option, GCC produces code that is slower and larger but which uses extremely little TOC space. You may wish to use this option only on files that contain less frequently-executed code.

`-maix64`
`-maix32`
: Enable 64-bit AIX ABI and calling convention: 64-bit pointers, 64-bit `long` type, and the infrastructure needed to support them. Specifying '`-maix64`' implies '`-mpowerpc64`', while '`-maix32`' disables the 64-bit ABI and implies '`-mno-powerpc64`'. GCC defaults to '`-maix32`'.

`-mxl-compat`
`-mno-xl-compat`
: Produce code that conforms more closely to IBM XL compiler semantics when using AIX-compatible ABI. Pass floating-point arguments to prototyped functions beyond the register save area (RSA) on the stack in addition to argument FPRs. Do not assume that most significant double in 128-bit long double value is properly rounded when comparing values and converting to double. Use XL symbol names for long double support routines.

The AIX calling convention was extended but not initially documented to handle an obscure K&R C case of calling a function that takes the address of its arguments with fewer arguments than declared. IBM XL compilers access floating-point arguments that do not fit in the RSA from the stack when a subroutine is compiled without optimization. Because always storing floating-point arguments on the stack is inefficient and rarely needed, this option is not enabled by default and only is necessary when calling subroutines compiled by IBM XL compilers without optimization.

`-mpe`
: Support *IBM RS/6000 SP Parallel Environment* (PE). Link an application written to use message passing with special startup code to enable the application to run. The system must have PE installed in the standard location ('`/usr/lpp/ppe.poe/`'), or the '`specs`' file must be overridden with the '`-specs=`' option to specify the appropriate directory location. The Parallel Environment does not support threads, so the '`-mpe`' option and the '`-pthread`' option are incompatible.

`-malign-natural`
`-malign-power`
: On AIX, 32-bit Darwin, and 64-bit PowerPC GNU/Linux, the option '`-malign-natural`' overrides the ABI-defined alignment of larger types, such as floating-point doubles, on their natural size-based boundary. The option '`-malign-power`' instructs GCC to follow the ABI-specified alignment rules. GCC defaults to the standard alignment defined in the ABI.

On 64-bit Darwin, natural alignment is the default, and '`-malign-power`' is not supported.

`-msoft-float`
`-mhard-float`
: Generate code that does not use (uses) the floating-point register set. Software floating-point emulation is provided if you use the '`-msoft-float`' option, and pass the option to GCC when linking.

`-mmultiple`
`-mno-multiple`
: Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions. These instructions are generated by default on POWER systems, and not generated on PowerPC systems. Do not use '`-mmultiple`' on little-endian PowerPC systems, since those instructions do not work when the processor is in little-endian mode. The exceptions are PPC740 and PPC750 which permit these instructions in little-endian mode.

`-mupdate`
`-mno-update`
: Generate code that uses (does not use) the load or store instructions that update the base register to the address of the calculated memory location. These instructions are generated by default. If you use '`-mno-update`', there is a small window between the time that the stack pointer is updated and the address of the previous frame is stored, which means code that walks the stack frame across interrupts or signals may get corrupted data.

`-mavoid-indexed-addresses`
`-mno-avoid-indexed-addresses`
: Generate code that tries to avoid (not avoid) the use of indexed load or store instructions. These instructions can incur a performance penalty on Power6 processors in certain situations, such as when stepping through large arrays that cross a 16M boundary. This option is enabled by default when targeting Power6 and disabled otherwise.

`-mfused-madd`
`-mno-fused-madd`
: Generate code that uses (does not use) the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used. The machine-dependent '`-mfused-madd`' option is now mapped to the machine-independent '`-ffp-contract=fast`' option, and '`-mno-fused-madd`' is mapped to '`-ffp-contract=off`'.

`-mmulhw`
`-mno-mulhw`
: Generate code that uses (does not use) the half-word multiply and multiply-accumulate instructions on the IBM 405, 440, 464 and 476 processors. These instructions are generated by default when targeting those processors.

`-mdlmzb`
`-mno-dlmzb`
: Generate code that uses (does not use) the string-search '`dlmzb`' instruction on the IBM 405, 440, 464 and 476 processors. This instruction is generated by default when targeting those processors.

`-mno-bit-align`
`-mbit-align`

On System V.4 and embedded PowerPC systems do not (do) force structures and unions that contain bit-fields to be aligned to the base type of the bit-field.

For example, by default a structure containing nothing but 8 `unsigned` bit-fields of length 1 is aligned to a 4-byte boundary and has a size of 4 bytes. By using '`-mno-bit-align`', the structure is aligned to a 1-byte boundary and is 1 byte in size.

`-mno-strict-align`
`-mstrict-align`

On System V.4 and embedded PowerPC systems do not (do) assume that unaligned memory references are handled by the system.

`-mrelocatable`
`-mno-relocatable`

Generate code that allows (does not allow) a static executable to be relocated to a different address at run time. A simple embedded PowerPC system loader should relocate the entire contents of `.got2` and 4-byte locations listed in the `.fixup` section, a table of 32-bit addresses generated by this option. For this to work, all objects linked together must be compiled with '`-mrelocatable`' or '`-mrelocatable-lib`'. '`-mrelocatable`' code aligns the stack to an 8-byte boundary.

`-mrelocatable-lib`
`-mno-relocatable-lib`

Like '`-mrelocatable`', '`-mrelocatable-lib`' generates a `.fixup` section to allow static executables to be relocated at run time, but '`-mrelocatable-lib`' does not use the smaller stack alignment of '`-mrelocatable`'. Objects compiled with '`-mrelocatable-lib`' may be linked with objects compiled with any combination of the '`-mrelocatable`' options.

`-mno-toc`
`-mtoc`

On System V.4 and embedded PowerPC systems do not (do) assume that register 2 contains a pointer to a global area pointing to the addresses used in the program.

`-mlittle`
`-mlittle-endian`

On System V.4 and embedded PowerPC systems compile code for the processor in little-endian mode. The '`-mlittle-endian`' option is the same as '`-mlittle`'.

`-mbig`
`-mbig-endian`

On System V.4 and embedded PowerPC systems compile code for the processor in big-endian mode. The '`-mbig-endian`' option is the same as '`-mbig`'.

`-mdynamic-no-pic`

On Darwin and Mac OS X systems, compile code so that it is not relocatable, but that its external references are relocatable. The resulting code is suitable for applications, but not shared libraries.

`-msingle-pic-base`
Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The runtime system is responsible for initializing this register with an appropriate value before execution begins.

`-mprioritize-restricted-insns=`*`priority`*
This option controls the priority that is assigned to dispatch-slot restricted instructions during the second scheduling pass. The argument *priority* takes the value '`0`', '`1`', or '`2`' to assign no, highest, or second-highest (respectively) priority to dispatch-slot restricted instructions.

`-msched-costly-dep=`*`dependence_type`*
This option controls which dependences are considered costly by the target during instruction scheduling. The argument *dependence_type* takes one of the following values:

'`no`' No dependence is costly.

'`all`' All dependences are costly.

'`true_store_to_load`'
A true dependence from store to load is costly.

'`store_to_load`'
Any dependence from store to load is costly.

number Any dependence for which the latency is greater than or equal to *number* is costly.

`-minsert-sched-nops=`*`scheme`*
This option controls which NOP insertion scheme is used during the second scheduling pass. The argument *scheme* takes one of the following values:

'`no`' Don't insert NOPs.

'`pad`' Pad with NOPs any dispatch group that has vacant issue slots, according to the scheduler's grouping.

'`regroup_exact`'
Insert NOPs to force costly dependent insns into separate groups. Insert exactly as many NOPs as needed to force an insn to a new group, according to the estimated processor grouping.

number Insert NOPs to force costly dependent insns into separate groups. Insert *number* NOPs to force an insn to a new group.

`-mcall-sysv`
On System V.4 and embedded PowerPC systems compile code using calling conventions that adhere to the March 1995 draft of the System V Application Binary Interface, PowerPC processor supplement. This is the default unless you configured GCC using '`powerpc-*-eabiaix`'.

`-mcall-sysv-eabi`
`-mcall-eabi`
Specify both '`-mcall-sysv`' and '`-meabi`' options.

`-mcall-sysv-noeabi`
Specify both '`-mcall-sysv`' and '`-mno-eabi`' options.

`-mcall-aixdesc`
On System V.4 and embedded PowerPC systems compile code for the AIX operating system.

`-mcall-linux`
On System V.4 and embedded PowerPC systems compile code for the Linux-based GNU system.

`-mcall-freebsd`
On System V.4 and embedded PowerPC systems compile code for the FreeBSD operating system.

`-mcall-netbsd`
On System V.4 and embedded PowerPC systems compile code for the NetBSD operating system.

`-mcall-openbsd`
On System V.4 and embedded PowerPC systems compile code for the OpenBSD operating system.

`-mtraceback=`*traceback_type*
Select the type of traceback table. Valid values for *traceback_type* are '`full`', '`part`', and '`no`'.

`-maix-struct-return`
Return all structures in memory (as specified by the AIX ABI).

`-msvr4-struct-return`
Return structures smaller than 8 bytes in registers (as specified by the SVR4 ABI).

`-mabi=`*abi-type*
Extend the current ABI with a particular extension, or remove such extension. Valid values are '`altivec`', '`no-altivec`', '`ibmlongdouble`', '`ieeelongdouble`', '`elfv1`', '`elfv2`'.

`-mabi=ibmlongdouble`
Change the current ABI to use IBM extended-precision long double. This is not likely to work if your system defaults to using IEEE extended-precision long double. If you change the long double type from IEEE extended-precision, the compiler will issue a warning unless you use the '`-Wno-psabi`' option. Requires '`-mlong-double-128`' to be enabled.

`-mabi=ieeelongdouble`
Change the current ABI to use IEEE extended-precision long double. This is not likely to work if your system defaults to using IBM extended-precision long double. If you change the long double type from IBM extended-precision, the compiler will issue a warning unless you use the '`-Wno-psabi`' option. Requires '`-mlong-double-128`' to be enabled.

`-mabi=elfv1`
: Change the current ABI to use the ELFv1 ABI. This is the default ABI for big-endian PowerPC 64-bit Linux. Overriding the default ABI requires special system support and is likely to fail in spectacular ways.

`-mabi=elfv2`
: Change the current ABI to use the ELFv2 ABI. This is the default ABI for little-endian PowerPC 64-bit Linux. Overriding the default ABI requires special system support and is likely to fail in spectacular ways.

`-mgnu-attribute`
`-mno-gnu-attribute`
: Emit .gnu_attribute assembly directives to set tag/value pairs in a .gnu.attributes section that specify ABI variations in function parameters or return values.

`-mprototype`
`-mno-prototype`
: On System V.4 and embedded PowerPC systems assume that all calls to variable argument functions are properly prototyped. Otherwise, the compiler must insert an instruction before every non-prototyped call to set or clear bit 6 of the condition code register (`CR`) to indicate whether floating-point values are passed in the floating-point registers in case the function takes variable arguments. With '`-mprototype`', only calls to prototyped variable argument functions set or clear the bit.

`-msim`
: On embedded PowerPC systems, assume that the startup module is called '`sim-crt0.o`' and that the standard C libraries are '`libsim.a`' and '`libc.a`'. This is the default for '`powerpc-*-eabisim`' configurations.

`-mmvme`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libmvme.a`' and '`libc.a`'.

`-mads`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libads.a`' and '`libc.a`'.

`-myellowknife`
: On embedded PowerPC systems, assume that the startup module is called '`crt0.o`' and the standard C libraries are '`libyk.a`' and '`libc.a`'.

`-mvxworks`
: On System V.4 and embedded PowerPC systems, specify that you are compiling for a VxWorks system.

`-memb`
: On embedded PowerPC systems, set the `PPC_EMB` bit in the ELF flags header to indicate that '`eabi`' extended relocations are used.

`-meabi`
`-mno-eabi`
: On System V.4 and embedded PowerPC systems do (do not) adhere to the Embedded Applications Binary Interface (EABI), which is a set of modifications to the System V.4 specifications. Selecting '`-meabi`' means that the stack is

aligned to an 8-byte boundary, a function `__eabi` is called from `main` to set up the EABI environment, and the '`-msdata`' option can use both `r2` and `r13` to point to two separate small data areas. Selecting '`-mno-eabi`' means that the stack is aligned to a 16-byte boundary, no EABI initialization function is called from `main`, and the '`-msdata`' option only uses `r13` to point to a single small data area. The '`-meabi`' option is on by default if you configured GCC using one of the '`powerpc*-*-eabi*`' options.

`-msdata=eabi`
On System V.4 and embedded PowerPC systems, put small initialized `const` global and static data in the `.sdata2` section, which is pointed to by register `r2`. Put small initialized non-`const` global and static data in the `.sdata` section, which is pointed to by register `r13`. Put small uninitialized global and static data in the `.sbss` section, which is adjacent to the `.sdata` section. The '`-msdata=eabi`' option is incompatible with the '`-mrelocatable`' option. The '`-msdata=eabi`' option also sets the '`-memb`' option.

`-msdata=sysv`
On System V.4 and embedded PowerPC systems, put small global and static data in the `.sdata` section, which is pointed to by register `r13`. Put small uninitialized global and static data in the `.sbss` section, which is adjacent to the `.sdata` section. The '`-msdata=sysv`' option is incompatible with the '`-mrelocatable`' option.

`-msdata=default`
`-msdata` On System V.4 and embedded PowerPC systems, if '`-meabi`' is used, compile code the same as '`-msdata=eabi`', otherwise compile code the same as '`-msdata=sysv`'.

`-msdata=data`
On System V.4 and embedded PowerPC systems, put small global data in the `.sdata` section. Put small uninitialized global data in the `.sbss` section. Do not use register `r13` to address small data however. This is the default behavior unless other '`-msdata`' options are used.

`-msdata=none`
`-mno-sdata`
On embedded PowerPC systems, put all initialized global and static data in the `.data` section, and all uninitialized data in the `.bss` section.

`-mreadonly-in-sdata`
Put read-only objects in the `.sdata` section as well. This is the default.

`-mblock-move-inline-limit=`*num*
Inline all block moves (such as calls to `memcpy` or structure copies) less than or equal to *num* bytes. The minimum value for *num* is 32 bytes on 32-bit targets and 64 bytes on 64-bit targets. The default value is target-specific.

`-mblock-compare-inline-limit=`*num*
Generate non-looping inline code for all block compares (such as calls to `memcmp` or structure compares) less than or equal to *num* bytes. If *num* is 0, all inline

expansion (non-loop and loop) of block compare is disabled. The default value is target-specific.

`-mblock-compare-inline-loop-limit=num`
: Generate an inline expansion using loop code for all block compares that are less than or equal to *num* bytes, but greater than the limit for non-loop inline block compare expansion. If the block length is not constant, at most *num* bytes will be compared before `memcmp` is called to compare the remainder of the block. The default value is target-specific.

`-mstring-compare-inline-limit=num`
: Compare at most *num* string bytes with inline code. If the difference or end of string is not found at the end of the inline compare a call to `strcmp` or `strncmp` will take care of the rest of the comparison. The default is 64 bytes.

`-G num`
: On embedded PowerPC systems, put global and static items less than or equal to *num* bytes into the small data or BSS sections instead of the normal data or BSS section. By default, *num* is 8. The '`-G num`' switch is also passed to the linker. All modules should be compiled with the same '`-G num`' value.

`-mregnames`
`-mno-regnames`
: On System V.4 and embedded PowerPC systems do (do not) emit register names in the assembly language output using symbolic forms.

`-mlongcall`
`-mno-longcall`
: By default assume that all calls are far away so that a longer and more expensive calling sequence is required. This is required for calls farther than 32 megabytes (33,554,432 bytes) from the current location. A short call is generated if the compiler knows the call cannot be that far away. This setting can be overridden by the `shortcall` function attribute, or by `#pragma longcall(0)`.

  Some linkers are capable of detecting out-of-range calls and generating glue code on the fly. On these systems, long calls are unnecessary and generate slower code. As of this writing, the AIX linker can do this, as can the GNU linker for PowerPC/64. It is planned to add this feature to the GNU linker for 32-bit PowerPC systems as well.

  On PowerPC64 ELFv2 and 32-bit PowerPC systems with newer GNU linkers, GCC can generate long calls using an inline PLT call sequence (see '`-mpltseq`'). PowerPC with '`-mbss-plt`' and PowerPC64 ELFv1 (big-endian) do not support inline PLT calls.

  On Darwin/PPC systems, `#pragma longcall` generates `jbsr callee, L42`, plus a *branch island* (glue code). The two target addresses represent the callee and the branch island. The Darwin/PPC linker prefers the first address and generates a `bl callee` if the PPC `bl` instruction reaches the callee directly; otherwise, the linker generates `bl L42` to call the branch island. The branch island is appended to the body of the calling function; it computes the full 32-bit address of the callee and jumps to it.

On Mach-O (Darwin) systems, this option directs the compiler emit to the glue for every direct call, and the Darwin linker decides whether to use or discard it.

In the future, GCC may ignore all longcall specifications when the linker is known to generate glue.

`-mpltseq`
`-mno-pltseq`

Implement (do not implement) -fno-plt and long calls using an inline PLT call sequence that supports lazy linking and long calls to functions in dlopen'd shared libraries. Inline PLT calls are only supported on PowerPC64 ELFv2 and 32-bit PowerPC systems with newer GNU linkers, and are enabled by default if the support is detected when configuring GCC, and, in the case of 32-bit PowerPC, if GCC is configured with '`--enable-secureplt`'. '`-mpltseq`' code and '`-mbss-plt`' 32-bit PowerPC relocatable objects may not be linked together.

`-mtls-markers`
`-mno-tls-markers`

Mark (do not mark) calls to `__tls_get_addr` with a relocation specifying the function argument. The relocation allows the linker to reliably associate function call with argument setup instructions for TLS optimization, which in turn allows GCC to better schedule the sequence.

`-mrecip`
`-mno-recip`

This option enables use of the reciprocal estimate and reciprocal square root estimate instructions with additional Newton-Raphson steps to increase precision instead of doing a divide or square root and divide for floating-point arguments. You should use the '`-ffast-math`' option when using '`-mrecip`' (or at least '`-funsafe-math-optimizations`', '`-ffinite-math-only`', '`-freciprocal-math`' and '`-fno-trapping-math`'). Note that while the throughput of the sequence is generally higher than the throughput of the non-reciprocal instruction, the precision of the sequence can be decreased by up to 2 ulp (i.e. the inverse of 1.0 equals 0.99999994) for reciprocal square roots.

`-mrecip=`*opt*

This option controls which reciprocal estimate instructions may be used. *opt* is a comma-separated list of options, which may be preceded by a `!` to invert the option:

'`all`' Enable all estimate instructions.

'`default`' Enable the default instructions, equivalent to '`-mrecip`'.

'`none`' Disable all estimate instructions, equivalent to '`-mno-recip`'.

'`div`' Enable the reciprocal approximation instructions for both single and double precision.

'`divf`' Enable the single-precision reciprocal approximation instructions.

'divd' Enable the double-precision reciprocal approximation instructions.

'rsqrt' Enable the reciprocal square root approximation instructions for both single and double precision.

'rsqrtf' Enable the single-precision reciprocal square root approximation instructions.

'rsqrtd' Enable the double-precision reciprocal square root approximation instructions.

So, for example, '-mrecip=all,!rsqrtd' enables all of the reciprocal estimate instructions, except for the FRSQRTE, XSRSQRTEDP, and XVRSQRTEDP instructions which handle the double-precision reciprocal square root calculations.

-mrecip-precision
-mno-recip-precision
Assume (do not assume) that the reciprocal estimate instructions provide higher-precision estimates than is mandated by the PowerPC ABI. Selecting '-mcpu=power6', '-mcpu=power7' or '-mcpu=power8' automatically selects '-mrecip-precision'. The double-precision square root estimate instructions are not generated by default on low-precision machines, since they do not provide an estimate that converges after three steps.

-mveclibabi=*type*
Specifies the ABI type to use for vectorizing intrinsics using an external library. The only type supported at present is 'mass', which specifies to use IBM's Mathematical Acceleration Subsystem (MASS) libraries for vectorizing intrinsics using external libraries. GCC currently emits calls to acosd2, acosf4, acoshd2, acoshf4, asind2, asinf4, asinhd2, asinhf4, atan2d2, atan2f4, atand2, atanf4, atanhd2, atanhf4, cbrtd2, cbrtf4, cosd2, cosf4, coshd2, coshf4, erfcd2, erfcf4, erfd2, erff4, exp2d2, exp2f4, expd2, expf4, expm1d2, expm1f4, hypotd2, hypotf4, lgammad2, lgammaf4, log10d2, log10f4, log1pd2, log1pf4, log2d2, log2f4, logd2, logf4, powd2, powf4, sind2, sinf4, sinhd2, sinhf4, sqrtd2, sqrtf4, tand2, tanf4, tanhd2, and tanhf4 when generating code for power7. Both '-ftree-vectorize' and '-funsafe-math-optimizations' must also be enabled. The MASS libraries must be specified at link time.

-mfriz
-mno-friz
Generate (do not generate) the friz instruction when the '-funsafe-math-optimizations' option is used to optimize rounding of floating-point values to 64-bit integer and back to floating point. The friz instruction does not return the same value if the floating-point number is too large to fit in an integer.

-mpointers-to-nested-functions
-mno-pointers-to-nested-functions
Generate (do not generate) code to load up the static chain register (r11) when calling through a pointer on AIX and 64-bit Linux systems where a function

pointer points to a 3-word descriptor giving the function address, TOC value to be loaded in register `r2`, and static chain value to be loaded in register `r11`. The '`-mpointers-to-nested-functions`' is on by default. You cannot call through pointers to nested functions or pointers to functions compiled in other languages that use the static chain if you use '`-mno-pointers-to-nested-functions`'.

`-msave-toc-indirect`
`-mno-save-toc-indirect`
: Generate (do not generate) code to save the TOC value in the reserved stack location in the function prologue if the function calls through a pointer on AIX and 64-bit Linux systems. If the TOC value is not saved in the prologue, it is saved just before the call through the pointer. The '`-mno-save-toc-indirect`' option is the default.

`-mcompat-align-parm`
`-mno-compat-align-parm`
: Generate (do not generate) code to pass structure parameters with a maximum alignment of 64 bits, for compatibility with older versions of GCC.

Older versions of GCC (prior to 4.9.0) incorrectly did not align a structure parameter on a 128-bit boundary when that structure contained a member requiring 128-bit alignment. This is corrected in more recent versions of GCC. This option may be used to generate code that is compatible with functions compiled with older versions of GCC.

The '`-mno-compat-align-parm`' option is the default.

`-mstack-protector-guard=`*`guard`*
`-mstack-protector-guard-reg=`*`reg`*
`-mstack-protector-guard-offset=`*`offset`*
`-mstack-protector-guard-symbol=`*`symbol`*
: Generate stack protection code using canary at *guard*. Supported locations are '`global`' for global canary or '`tls`' for per-thread canary in the TLS block (the default with GNU libc version 2.4 or later).

With the latter choice the options '`-mstack-protector-guard-reg=`*`reg`*' and '`-mstack-protector-guard-offset=`*`offset`*' furthermore specify which register to use as base register for reading the canary, and from what offset from that base register. The default for those is as specified in the relevant ABI. '`-mstack-protector-guard-symbol=`*`symbol`*' overrides the offset with a symbol reference to a canary in the TLS block.

3.18.43 RX Options

These command-line options are defined for RX targets:

`-m64bit-doubles`
`-m32bit-doubles`
: Make the `double` data type be 64 bits ('`-m64bit-doubles`') or 32 bits ('`-m32bit-doubles`') in size. The default is '`-m32bit-doubles`'. *Note* RX floating-point hardware only works on 32-bit values, which is why the default is '`-m32bit-doubles`'.

`-fpu`
`-nofpu` Enables ('`-fpu`') or disables ('`-nofpu`') the use of RX floating-point hardware. The default is enabled for the RX600 series and disabled for the RX200 series.

Floating-point instructions are only generated for 32-bit floating-point values, however, so the FPU hardware is not used for doubles if the '`-m64bit-doubles`' option is used.

Note If the '`-fpu`' option is enabled then '`-funsafe-math-optimizations`' is also enabled automatically. This is because the RX FPU instructions are themselves unsafe.

`-mcpu=`*`name`*
Selects the type of RX CPU to be targeted. Currently three types are supported, the generic '`RX600`' and '`RX200`' series hardware and the specific '`RX610`' CPU. The default is '`RX600`'.

The only difference between '`RX600`' and '`RX610`' is that the '`RX610`' does not support the `MVTIPL` instruction.

The '`RX200`' series does not have a hardware floating-point unit and so '`-nofpu`' is enabled by default when this type is selected.

`-mbig-endian-data`
`-mlittle-endian-data`
Store data (but not code) in the big-endian format. The default is '`-mlittle-endian-data`', i.e. to store data in the little-endian format.

`-msmall-data-limit=`*`N`*
Specifies the maximum size in bytes of global and static variables which can be placed into the small data area. Using the small data area can lead to smaller and faster code, but the size of area is limited and it is up to the programmer to ensure that the area does not overflow. Also when the small data area is used one of the RX's registers (usually `r13`) is reserved for use pointing to this area, so it is no longer available for use by the compiler. This could result in slower and/or larger code if variables are pushed onto the stack instead of being held in this register.

Note, common variables (variables that have not been initialized) and constants are not placed into the small data area as they are assigned to other sections in the output executable.

The default value is zero, which disables this feature. Note, this feature is not enabled by default with higher optimization levels ('`-O2`' etc) because of the potentially detrimental effects of reserving a register. It is up to the programmer to experiment and discover whether this feature is of benefit to their program. See the description of the '`-mpid`' option for a description of how the actual register to hold the small data area pointer is chosen.

`-msim`
`-mno-sim` Use the simulator runtime. The default is to use the libgloss board-specific runtime.

`-mas100-syntax`
`-mno-as100-syntax`
When generating assembler output use a syntax that is compatible with Renesas's AS100 assembler. This syntax can also be handled by the GAS assembler, but it has some restrictions so it is not generated by default.

`-mmax-constant-size=N`
Specifies the maximum size, in bytes, of a constant that can be used as an operand in a RX instruction. Although the RX instruction set does allow constants of up to 4 bytes in length to be used in instructions, a longer value equates to a longer instruction. Thus in some circumstances it can be beneficial to restrict the size of constants that are used in instructions. Constants that are too big are instead placed into a constant pool and referenced via register indirection.

The value *N* can be between 0 and 4. A value of 0 (the default) or 4 means that constants of any size are allowed.

`-mrelax` Enable linker relaxation. Linker relaxation is a process whereby the linker attempts to reduce the size of a program by finding shorter versions of various instructions. Disabled by default.

`-mint-register=N`
Specify the number of registers to reserve for fast interrupt handler functions. The value *N* can be between 0 and 4. A value of 1 means that register `r13` is reserved for the exclusive use of fast interrupt handlers. A value of 2 reserves `r13` and `r12`. A value of 3 reserves `r13`, `r12` and `r11`, and a value of 4 reserves `r13` through `r10`. A value of 0, the default, does not reserve any registers.

`-msave-acc-in-interrupts`
Specifies that interrupt handler functions should preserve the accumulator register. This is only necessary if normal code might use the accumulator register, for example because it performs 64-bit multiplications. The default is to ignore the accumulator as this makes the interrupt handlers faster.

`-mpid`
`-mno-pid` Enables the generation of position independent data. When enabled any access to constant data is done via an offset from a base address held in a register. This allows the location of constant data to be determined at run time without requiring the executable to be relocated, which is a benefit to embedded applications with tight memory constraints. Data that can be modified is not affected by this option.

Note, using this feature reserves a register, usually `r13`, for the constant data base address. This can result in slower and/or larger code, especially in complicated functions.

The actual register chosen to hold the constant data base address depends upon whether the '`-msmall-data-limit`' and/or the '`-mint-register`' command-line options are enabled. Starting with register `r13` and proceeding downwards, registers are allocated first to satisfy the requirements of '`-mint-register`', then '`-mpid`' and finally '`-msmall-data-limit`'. Thus it is possible for the

small data area register to be r8 if both '-mint-register=4' and '-mpid' are specified on the command line.

By default this feature is not enabled. The default can be restored via the '-mno-pid' command-line option.

-mno-warn-multiple-fast-interrupts
-mwarn-multiple-fast-interrupts
: Prevents GCC from issuing a warning message if it finds more than one fast interrupt handler when it is compiling a file. The default is to issue a warning for each extra fast interrupt handler found, as the RX only supports one such interrupt.

-mallow-string-insns
-mno-allow-string-insns
: Enables or disables the use of the string manipulation instructions SMOVF, SCMPU, SMOVB, SMOVU, SUNTIL SWHILE and also the RMPA instruction. These instructions may prefetch data, which is not safe to do if accessing an I/O register. (See section 12.2.7 of the RX62N Group User's Manual for more information).

The default is to allow these instructions, but it is not possible for GCC to reliably detect all circumstances where a string instruction might be used to access an I/O register, so their use cannot be disabled automatically. Instead it is reliant upon the programmer to use the '-mno-allow-string-insns' option if their program accesses I/O space.

When the instructions are enabled GCC defines the C preprocessor symbol __RX_ALLOW_STRING_INSNS__, otherwise it defines the symbol __RX_DISALLOW_STRING_INSNS__.

-mjsr
-mno-jsr
: Use only (or not only) JSR instructions to access functions. This option can be used when code size exceeds the range of BSR instructions. Note that '-mno-jsr' does not mean to not use JSR but instead means that any type of branch may be used.

Note: The generic GCC command-line option '-ffixed-*reg*' has special significance to the RX port when used with the interrupt function attribute. This attribute indicates a function intended to process fast interrupts. GCC ensures that it only uses the registers r10, r11, r12 and/or r13 and only provided that the normal use of the corresponding registers have been restricted via the '-ffixed-*reg*' or '-mint-register' command-line options.

3.18.44 S/390 and zSeries Options

These are the '-m' options defined for the S/390 and zSeries architecture.

-mhard-float
-msoft-float
: Use (do not use) the hardware floating-point instructions and registers for floating-point operations. When '-msoft-float' is specified, functions in 'libgcc.a' are used to perform floating-point operations. When

'-mhard-float' is specified, the compiler generates IEEE floating-point instructions. This is the default.

-mhard-dfp
-mno-hard-dfp
Use (do not use) the hardware decimal-floating-point instructions for decimal-floating-point operations. When '-mno-hard-dfp' is specified, functions in 'libgcc.a' are used to perform decimal-floating-point operations. When '-mhard-dfp' is specified, the compiler generates decimal-floating-point hardware instructions. This is the default for '-march=z9-ec' or higher.

-mlong-double-64
-mlong-double-128
These switches control the size of long double type. A size of 64 bits makes the long double type equivalent to the double type. This is the default.

-mbackchain
-mno-backchain
Store (do not store) the address of the caller's frame as backchain pointer into the callee's stack frame. A backchain may be needed to allow debugging using tools that do not understand DWARF call frame information. When '-mno-packed-stack' is in effect, the backchain pointer is stored at the bottom of the stack frame; when '-mpacked-stack' is in effect, the backchain is placed into the topmost word of the 96/160 byte register save area.

In general, code compiled with '-mbackchain' is call-compatible with code compiled with '-mmo-backchain'; however, use of the backchain for debugging purposes usually requires that the whole binary is built with '-mbackchain'. Note that the combination of '-mbackchain', '-mpacked-stack' and '-mhard-float' is not supported. In order to build a linux kernel use '-msoft-float'.

The default is to not maintain the backchain.

-mpacked-stack
-mno-packed-stack
Use (do not use) the packed stack layout. When '-mno-packed-stack' is specified, the compiler uses the all fields of the 96/160 byte register save area only for their default purpose; unused fields still take up stack space. When '-mpacked-stack' is specified, register save slots are densely packed at the top of the register save area; unused space is reused for other purposes, allowing for more efficient use of the available stack space. However, when '-mbackchain' is also in effect, the topmost word of the save area is always used to store the backchain, and the return address register is always saved two words below the backchain.

As long as the stack frame backchain is not used, code generated with '-mpacked-stack' is call-compatible with code generated with '-mno-packed-stack'. Note that some non-FSF releases of GCC 2.95 for S/390 or zSeries generated code that uses the stack frame backchain at run time, not just for debugging purposes. Such code is not call-compatible with code compiled with '-mpacked-stack'. Also, note that the combination of

'-mbackchain', '-mpacked-stack' and '-mhard-float' is not supported. In order to build a linux kernel use '-msoft-float'.

The default is to not use the packed stack layout.

-msmall-exec
-mno-small-exec
: Generate (or do not generate) code using the bras instruction to do subroutine calls. This only works reliably if the total executable size does not exceed 64k. The default is to use the basr instruction instead, which does not have this limitation.

-m64
-m31
: When '-m31' is specified, generate code compliant to the GNU/Linux for S/390 ABI. When '-m64' is specified, generate code compliant to the GNU/Linux for zSeries ABI. This allows GCC in particular to generate 64-bit instructions. For the 's390' targets, the default is '-m31', while the 's390x' targets default to '-m64'.

-mzarch
-mesa
: When '-mzarch' is specified, generate code using the instructions available on z/Architecture. When '-mesa' is specified, generate code using the instructions available on ESA/390. Note that '-mesa' is not possible with '-m64'. When generating code compliant to the GNU/Linux for S/390 ABI, the default is '-mesa'. When generating code compliant to the GNU/Linux for zSeries ABI, the default is '-mzarch'.

-mhtm
-mno-htm
: The '-mhtm' option enables a set of builtins making use of instructions available with the transactional execution facility introduced with the IBM zEnterprise EC12 machine generation Section 6.60.26 [S/390 System z Built-in Functions], page 766. '-mhtm' is enabled by default when using '-march=zEC12'.

-mvx
-mno-vx
: When '-mvx' is specified, generate code using the instructions available with the vector extension facility introduced with the IBM z13 machine generation. This option changes the ABI for some vector type values with regard to alignment and calling conventions. In case vector type values are being used in an ABI-relevant context a GAS '.gnu_attribute' command will be added to mark the resulting binary with the ABI used. '-mvx' is enabled by default when using '-march=z13'.

-mzvector
-mno-zvector
: The '-mzvector' option enables vector language extensions and builtins using instructions available with the vector extension facility introduced with the IBM z13 machine generation. This option adds support for 'vector' to be used as a keyword to define vector type variables and arguments. 'vector' is only available when GNU extensions are enabled. It will not be expanded when requesting strict standard compliance e.g. with '-std=c99'. In addition to the GCC low-level builtins '-mzvector' enables a set of builtins added for

compatibility with AltiVec-style implementations like Power and Cell. In order to make use of these builtins the header file '`vecintrin.h`' needs to be included. '`-mzvector`' is disabled by default.

`-mmvcle`
`-mno-mvcle`
Generate (or do not generate) code using the `mvcle` instruction to perform block moves. When '`-mno-mvcle`' is specified, use a `mvc` loop instead. This is the default unless optimizing for size.

`-mdebug`
`-mno-debug`
Print (or do not print) additional debug information when compiling. The default is to not print debug information.

`-march=`*`cpu-type`*
Generate code that runs on *cpu-type*, which is the name of a system representing a certain processor type. Possible values for *cpu-type* are '`z900`'/'`arch5`', '`z990`'/'`arch6`', '`z9-109`', '`z9-ec`'/'`arch7`', '`z10`'/'`arch8`', '`z196`'/'`arch9`', '`zEC12`', '`z13`'/'`arch11`', '`z14`'/'`arch12`', and '`native`'.

The default is '`-march=z900`'.

Specifying '`native`' as cpu type can be used to select the best architecture option for the host processor. '`-march=native`' has no effect if GCC does not recognize the processor.

`-mtune=`*`cpu-type`*
Tune to *cpu-type* everything applicable about the generated code, except for the ABI and the set of available instructions. The list of *cpu-type* values is the same as for '`-march`'. The default is the value used for '`-march`'.

`-mtpf-trace`
`-mno-tpf-trace`
Generate code that adds (does not add) in TPF OS specific branches to trace routines in the operating system. This option is off by default, even when compiling for the TPF OS.

`-mfused-madd`
`-mno-fused-madd`
Generate code that uses (does not use) the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used.

`-mwarn-framesize=`*`framesize`*
Emit a warning if the current function exceeds the given frame size. Because this is a compile-time check it doesn't need to be a real problem when the program runs. It is intended to identify functions that most probably cause a stack overflow. It is useful to be used in an environment with limited stack size e.g. the linux kernel.

`-mwarn-dynamicstack`
Emit a warning if the function calls `alloca` or uses dynamically-sized arrays. This is generally a bad idea with a limited stack size.

`-mstack-guard=`*`stack-guard`*
`-mstack-size=`*`stack-size`*

If these options are provided the S/390 back end emits additional instructions in the function prologue that trigger a trap if the stack size is *stack-guard* bytes above the *stack-size* (remember that the stack on S/390 grows downward). If the *stack-guard* option is omitted the smallest power of 2 larger than the frame size of the compiled function is chosen. These options are intended to be used to help debugging stack overflow problems. The additionally emitted code causes only little overhead and hence can also be used in production-like systems without greater performance degradation. The given values have to be exact powers of 2 and *stack-size* has to be greater than *stack-guard* without exceeding 64k. In order to be efficient the extra code makes the assumption that the stack starts at an address aligned to the value given by *stack-size*. The *stack-guard* option can only be used in conjunction with *stack-size*.

`-mhotpatch=`*`pre-halfwords,post-halfwords`*

If the hotpatch option is enabled, a "hot-patching" function prologue is generated for all functions in the compilation unit. The funtion label is prepended with the given number of two-byte NOP instructions (*pre-halfwords*, maximum 1000000). After the label, 2 * *post-halfwords* bytes are appended, using the largest NOP like instructions the architecture allows (maximum 1000000).

If both arguments are zero, hotpatching is disabled.

This option can be overridden for individual functions with the `hotpatch` attribute.

3.18.45 Score Options

These options are defined for Score implementations:

`-meb` Compile code for big-endian mode. This is the default.

`-mel` Compile code for little-endian mode.

`-mnhwloop`

Disable generation of `bcnz` instructions.

`-muls` Enable generation of unaligned load and store instructions.

`-mmac` Enable the use of multiply-accumulate instructions. Disabled by default.

`-mscore5` Specify the SCORE5 as the target architecture.

`-mscore5u`

Specify the SCORE5U of the target architecture.

`-mscore7` Specify the SCORE7 as the target architecture. This is the default.

`-mscore7d`

Specify the SCORE7D as the target architecture.

3.18.46 SH Options

These '`-m`' options are defined for the SH implementations:

`-m1` Generate code for the SH1.

`-m2` Generate code for the SH2.

`-m2e` Generate code for the SH2e.

`-m2a-nofpu`
Generate code for the SH2a without FPU, or for a SH2a-FPU in such a way that the floating-point unit is not used.

`-m2a-single-only`
Generate code for the SH2a-FPU, in such a way that no double-precision floating-point operations are used.

`-m2a-single`
Generate code for the SH2a-FPU assuming the floating-point unit is in single-precision mode by default.

`-m2a` Generate code for the SH2a-FPU assuming the floating-point unit is in double-precision mode by default.

`-m3` Generate code for the SH3.

`-m3e` Generate code for the SH3e.

`-m4-nofpu`
Generate code for the SH4 without a floating-point unit.

`-m4-single-only`
Generate code for the SH4 with a floating-point unit that only supports single-precision arithmetic.

`-m4-single`
Generate code for the SH4 assuming the floating-point unit is in single-precision mode by default.

`-m4` Generate code for the SH4.

`-m4-100` Generate code for SH4-100.

`-m4-100-nofpu`
Generate code for SH4-100 in such a way that the floating-point unit is not used.

`-m4-100-single`
Generate code for SH4-100 assuming the floating-point unit is in single-precision mode by default.

`-m4-100-single-only`
Generate code for SH4-100 in such a way that no double-precision floating-point operations are used.

`-m4-200` Generate code for SH4-200.

`-m4-200-nofpu`
Generate code for SH4-200 without in such a way that the floating-point unit is not used.

`-m4-200-single`
: Generate code for SH4-200 assuming the floating-point unit is in single-precision mode by default.

`-m4-200-single-only`
: Generate code for SH4-200 in such a way that no double-precision floating-point operations are used.

`-m4-300` Generate code for SH4-300.

`-m4-300-nofpu`
: Generate code for SH4-300 without in such a way that the floating-point unit is not used.

`-m4-300-single`
: Generate code for SH4-300 in such a way that no double-precision floating-point operations are used.

`-m4-300-single-only`
: Generate code for SH4-300 in such a way that no double-precision floating-point operations are used.

`-m4-340` Generate code for SH4-340 (no MMU, no FPU).

`-m4-500` Generate code for SH4-500 (no FPU). Passes '`-isa=sh4-nofpu`' to the assembler.

`-m4a-nofpu`
: Generate code for the SH4al-dsp, or for a SH4a in such a way that the floating-point unit is not used.

`-m4a-single-only`
: Generate code for the SH4a, in such a way that no double-precision floating-point operations are used.

`-m4a-single`
: Generate code for the SH4a assuming the floating-point unit is in single-precision mode by default.

`-m4a` Generate code for the SH4a.

`-m4al` Same as '`-m4a-nofpu`', except that it implicitly passes '`-dsp`' to the assembler. GCC doesn't generate any DSP instructions at the moment.

`-mb` Compile code for the processor in big-endian mode.

`-ml` Compile code for the processor in little-endian mode.

`-mdalign` Align doubles at 64-bit boundaries. Note that this changes the calling conventions, and thus some functions from the standard C library do not work unless you recompile it first with '`-mdalign`'.

`-mrelax` Shorten some address references at link time, when possible; uses the linker option '`-relax`'.

`-mbigtable`
: Use 32-bit offsets in `switch` tables. The default is to use 16-bit offsets.

`-mbitops` Enable the use of bit manipulation instructions on SH2A.

`-mfmovd` Enable the use of the instruction `fmovd`. Check '`-mdalign`' for alignment constraints.

`-mrenesas`
Comply with the calling conventions defined by Renesas.

`-mno-renesas`
Comply with the calling conventions defined for GCC before the Renesas conventions were available. This option is the default for all targets of the SH toolchain.

`-mnomacsave`
Mark the `MAC` register as call-clobbered, even if '`-mrenesas`' is given.

`-mieee`
`-mno-ieee`
Control the IEEE compliance of floating-point comparisons, which affects the handling of cases where the result of a comparison is unordered. By default '`-mieee`' is implicitly enabled. If '`-ffinite-math-only`' is enabled '`-mno-ieee`' is implicitly set, which results in faster floating-point greater-equal and less-equal comparisons. The implicit settings can be overridden by specifying either '`-mieee`' or '`-mno-ieee`'.

`-minline-ic_invalidate`
Inline code to invalidate instruction cache entries after setting up nested function trampolines. This option has no effect if '`-musermode`' is in effect and the selected code generation option (e.g. '`-m4`') does not allow the use of the `icbi` instruction. If the selected code generation option does not allow the use of the `icbi` instruction, and '`-musermode`' is not in effect, the inlined code manipulates the instruction cache address array directly with an associative write. This not only requires privileged mode at run time, but it also fails if the cache line had been mapped via the TLB and has become unmapped.

`-misize` Dump instruction size and location in the assembly code.

`-mpadstruct`
This option is deprecated. It pads structures to multiple of 4 bytes, which is incompatible with the SH ABI.

`-matomic-model=`*`model`*
Sets the model of atomic operations and additional parameters as a comma separated list. For details on the atomic built-in functions see Section 6.55 [__atomic Builtins], page 632. The following models and parameters are supported:

'`none`' Disable compiler generated atomic sequences and emit library calls for atomic operations. This is the default if the target is not `sh*-*-linux*`.

'`soft-gusa`'
Generate GNU/Linux compatible gUSA software atomic sequences for the atomic built-in functions. The generated atomic sequences

require additional support from the interrupt/exception handling code of the system and are only suitable for SH3* and SH4* single-core systems. This option is enabled by default when the target is `sh*-*-linux*` and SH3* or SH4*. When the target is SH4A, this option also partially utilizes the hardware atomic instructions `movli.l` and `movco.l` to create more efficient code, unless '`strict`' is specified.

'`soft-tcb`'
: Generate software atomic sequences that use a variable in the thread control block. This is a variation of the gUSA sequences which can also be used on SH1* and SH2* targets. The generated atomic sequences require additional support from the interrupt/exception handling code of the system and are only suitable for single-core systems. When using this model, the '`gbr-offset=`' parameter has to be specified as well.

'`soft-imask`'
: Generate software atomic sequences that temporarily disable interrupts by setting `SR.IMASK = 1111`. This model works only when the program runs in privileged mode and is only suitable for single-core systems. Additional support from the interrupt/exception handling code of the system is not required. This model is enabled by default when the target is `sh*-*-linux*` and SH1* or SH2*.

'`hard-llcs`'
: Generate hardware atomic sequences using the `movli.l` and `movco.l` instructions only. This is only available on SH4A and is suitable for multi-core systems. Since the hardware instructions support only 32 bit atomic variables access to 8 or 16 bit variables is emulated with 32 bit accesses. Code compiled with this option is also compatible with other software atomic model interrupt/exception handling systems if executed on an SH4A system. Additional support from the interrupt/exception handling code of the system is not required for this model.

'`gbr-offset=`'
: This parameter specifies the offset in bytes of the variable in the thread control block structure that should be used by the generated atomic sequences when the '`soft-tcb`' model has been selected. For other models this parameter is ignored. The specified value must be an integer multiple of four and in the range 0-1020.

'`strict`'
: This parameter prevents mixed usage of multiple atomic models, even if they are compatible, and makes the compiler generate atomic sequences of the specified model only.

`-mtas`
: Generate the `tas.b` opcode for `__atomic_test_and_set`. Notice that depending on the particular hardware and software configuration this can degrade overall performance due to the operand cache line flushes that are implied by

the `tas.b` instruction. On multi-core SH4A processors the `tas.b` instruction must be used with caution since it can result in data corruption for certain cache configurations.

`-mprefergot`
When generating position-independent code, emit function calls using the Global Offset Table instead of the Procedure Linkage Table.

`-musermode`
`-mno-usermode`
Don't allow (allow) the compiler generating privileged mode code. Specifying '`-musermode`' also implies '`-mno-inline-ic_invalidate`' if the inlined code would not work in user mode. '`-musermode`' is the default when the target is `sh*-*-linux*`. If the target is SH1* or SH2* '`-musermode`' has no effect, since there is no user mode.

`-multcost=`*number*
Set the cost to assume for a multiply insn.

`-mdiv=`*strategy*
Set the division strategy to be used for integer division operations. *strategy* can be one of:

'`call-div1`'
Calls a library function that uses the single-step division instruction `div1` to perform the operation. Division by zero calculates an unspecified result and does not trap. This is the default except for SH4, SH2A and SHcompact.

'`call-fp`' Calls a library function that performs the operation in double precision floating point. Division by zero causes a floating-point exception. This is the default for SHcompact with FPU. Specifying this for targets that do not have a double precision FPU defaults to `call-div1`.

'`call-table`'
Calls a library function that uses a lookup table for small divisors and the `div1` instruction with case distinction for larger divisors. Division by zero calculates an unspecified result and does not trap. This is the default for SH4. Specifying this for targets that do not have dynamic shift instructions defaults to `call-div1`.

When a division strategy has not been specified the default strategy is selected based on the current target. For SH2A the default strategy is to use the `divs` and `divu` instructions instead of library function calls.

`-maccumulate-outgoing-args`
Reserve space once for outgoing arguments in the function prologue rather than around each call. Generally beneficial for performance and size. Also needed for unwinding to avoid changing the stack frame around conditional code.

`-mdivsi3_libfunc=name`
: Set the name of the library function used for 32-bit signed division to *name*. This only affects the name used in the '`call`' division strategies, and the compiler still expects the same sets of input/output/clobbered registers as if this option were not present.

`-mfixed-range=register-range`
: Generate code treating the given register range as fixed registers. A fixed register is one that the register allocator cannot use. This is useful when compiling kernel code. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma.

`-mbranch-cost=num`
: Assume *num* to be the cost for a branch instruction. Higher numbers make the compiler try to generate more branch-free code if possible. If not specified the value is selected depending on the processor type that is being compiled for.

`-mzdcbranch`
`-mno-zdcbranch`
: Assume (do not assume) that zero displacement conditional branch instructions `bt` and `bf` are fast. If '`-mzdcbranch`' is specified, the compiler prefers zero displacement branch code sequences. This is enabled by default when generating code for SH4 and SH4A. It can be explicitly disabled by specifying '`-mno-zdcbranch`'.

`-mcbranch-force-delay-slot`
: Force the usage of delay slots for conditional branches, which stuffs the delay slot with a `nop` if a suitable instruction cannot be found. By default this option is disabled. It can be enabled to work around hardware bugs as found in the original SH7055.

`-mfused-madd`
`-mno-fused-madd`
: Generate code that uses (does not use) the floating-point multiply and accumulate instructions. These instructions are generated by default if hardware floating point is used. The machine-dependent '`-mfused-madd`' option is now mapped to the machine-independent '`-ffp-contract=fast`' option, and '`-mno-fused-madd`' is mapped to '`-ffp-contract=off`'.

`-mfsca`
`-mno-fsca`
: Allow or disallow the compiler to emit the `fsca` instruction for sine and cosine approximations. The option '`-mfsca`' must be used in combination with '`-funsafe-math-optimizations`'. It is enabled by default when generating code for SH4A. Using '`-mno-fsca`' disables sine and cosine approximations even if '`-funsafe-math-optimizations`' is in effect.

`-mfsrra`
`-mno-fsrra`
: Allow or disallow the compiler to emit the `fsrra` instruction for reciprocal square root approximations. The option '`-mfsrra`' must be used in combina-

tion with '-funsafe-math-optimizations' and '-ffinite-math-only'. It is enabled by default when generating code for SH4A. Using '-mno-fsrra' disables reciprocal square root approximations even if '-funsafe-math-optimizations' and '-ffinite-math-only' are in effect.

-mpretend-cmove
: Prefer zero-displacement conditional branches for conditional move instruction patterns. This can result in faster code on the SH4 processor.

-mfdpic
: Generate code using the FDPIC ABI.

3.18.47 Solaris 2 Options

These '-m' options are supported on Solaris 2:

-mclear-hwcap
: '-mclear-hwcap' tells the compiler to remove the hardware capabilities generated by the Solaris assembler. This is only necessary when object files use ISA extensions not supported by the current machine, but check at runtime whether or not to use them.

-mimpure-text
: '-mimpure-text', used in addition to '-shared', tells the compiler to not pass '-z text' to the linker when linking a shared object. Using this option, you can link position-dependent code into a shared object.

  '-mimpure-text' suppresses the "relocations remain against allocatable but non-writable sections" linker error message. However, the necessary relocations trigger copy-on-write, and the shared object is not actually shared across processes. Instead of using '-mimpure-text', you should compile all source code with '-fpic' or '-fPIC'.

These switches are supported in addition to the above on Solaris 2:

-pthreads
: This is a synonym for '-pthread'.

3.18.48 SPARC Options

These '-m' options are supported on the SPARC:

-mno-app-regs
-mapp-regs
: Specify '-mapp-regs' to generate output using the global registers 2 through 4, which the SPARC SVR4 ABI reserves for applications. Like the global register 1, each global register 2 through 4 is then treated as an allocable register that is clobbered by function calls. This is the default.

  To be fully SVR4 ABI-compliant at the cost of some performance loss, specify '-mno-app-regs'. You should compile libraries and system software with this option.

-mflat
-mno-flat
: With '-mflat', the compiler does not generate save/restore instructions and uses a "flat" or single register window model. This model is compatible with

the regular register window model. The local registers and the input registers (0–5) are still treated as "call-saved" registers and are saved on the stack as needed.

With '`-mno-flat`' (the default), the compiler generates save/restore instructions (except for leaf functions). This is the normal operating mode.

`-mfpu`
`-mhard-float`
Generate output containing floating-point instructions. This is the default.

`-mno-fpu`
`-msoft-float`
Generate output containing library calls for floating point. **Warning:** the requisite libraries are not available for all SPARC targets. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation. The embedded targets '`sparc-*-aout`' and '`sparclite-*-*`' do provide software floating-point support.

'`-msoft-float`' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile '`libgcc.a`', the library that comes with GCC, with '`-msoft-float`' in order for this to work.

`-mhard-quad-float`
Generate output containing quad-word (long double) floating-point instructions.

`-msoft-quad-float`
Generate output containing library calls for quad-word (long double) floating-point instructions. The functions called are those specified in the SPARC ABI. This is the default.

As of this writing, there are no SPARC implementations that have hardware support for the quad-word floating-point instructions. They all invoke a trap handler for one of these instructions, and then the trap handler emulates the effect of the instruction. Because of the trap handler overhead, this is much slower than calling the ABI library routines. Thus the '`-msoft-quad-float`' option is the default.

`-mno-unaligned-doubles`
`-munaligned-doubles`
Assume that doubles have 8-byte alignment. This is the default.

With '`-munaligned-doubles`', GCC assumes that doubles have 8-byte alignment only if they are contained in another type, or if they have an absolute address. Otherwise, it assumes they have 4-byte alignment. Specifying this option avoids some rare compatibility problems with code generated by other compilers. It is not the default because it results in a performance loss, especially for floating-point code.

`-muser-mode`
`-mno-user-mode`
Do not generate code that can only run in supervisor mode. This is relevant only for the `casa` instruction emitted for the LEON3 processor. This is the default.

`-mfaster-structs`
`-mno-faster-structs`
With '`-mfaster-structs`', the compiler assumes that structures should have 8-byte alignment. This enables the use of pairs of `ldd` and `std` instructions for copies in structure assignment, in place of twice as many `ld` and `st` pairs. However, the use of this changed alignment directly violates the SPARC ABI. Thus, it's intended only for use on targets where the developer acknowledges that their resulting code is not directly in line with the rules of the ABI.

`-mstd-struct-return`
`-mno-std-struct-return`
With '`-mstd-struct-return`', the compiler generates checking code in functions returning structures or unions to detect size mismatches between the two sides of function calls, as per the 32-bit ABI.

The default is '`-mno-std-struct-return`'. This option has no effect in 64-bit mode.

`-mlra`
`-mno-lra` Enable Local Register Allocation. This is the default for SPARC since GCC 7 so '`-mno-lra`' needs to be passed to get old Reload.

`-mcpu=`*cpu_type*
Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`v7`', '`cypress`', '`v8`', '`supersparc`', '`hypersparc`', '`leon`', '`leon3`', '`leon3v7`', '`sparclite`', '`f930`', '`f934`', '`sparclite86x`', '`sparclet`', '`tsc701`', '`v9`', '`ultrasparc`', '`ultrasparc3`', '`niagara`', '`niagara2`', '`niagara3`', '`niagara4`', '`niagara7`' and '`m8`'.

Native Solaris and GNU/Linux toolchains also support the value '`native`', which selects the best architecture option for the host processor. '`-mcpu=native`' has no effect if GCC does not recognize the processor.

Default instruction scheduling parameters are used for values that select an architecture and not an implementation. These are '`v7`', '`v8`', '`sparclite`', '`sparclet`', '`v9`'.

Here is a list of each supported architecture and their supported implementations.

v7	cypress, leon3v7
v8	supersparc, hypersparc, leon, leon3
sparclite	f930, f934, sparclite86x
sparclet	tsc701

v9 ultrasparc, ultrasparc3, niagara, niagara2, niagara3, niagara4, niagara7, m8

By default (unless configured otherwise), GCC generates code for the V7 variant of the SPARC architecture. With '`-mcpu=cypress`', the compiler additionally optimizes it for the Cypress CY7C602 chip, as used in the SPARCStation/SPARCServer 3xx series. This is also appropriate for the older SPARCStation 1, 2, IPX etc.

With '`-mcpu=v8`', GCC generates code for the V8 variant of the SPARC architecture. The only difference from V7 code is that the compiler emits the integer multiply and integer divide instructions which exist in SPARC-V8 but not in SPARC-V7. With '`-mcpu=supersparc`', the compiler additionally optimizes it for the SuperSPARC chip, as used in the SPARCStation 10, 1000 and 2000 series.

With '`-mcpu=sparclite`', GCC generates code for the SPARClite variant of the SPARC architecture. This adds the integer multiply, integer divide step and scan (`ffs`) instructions which exist in SPARClite but not in SPARC-V7. With '`-mcpu=f930`', the compiler additionally optimizes it for the Fujitsu MB86930 chip, which is the original SPARClite, with no FPU. With '`-mcpu=f934`', the compiler additionally optimizes it for the Fujitsu MB86934 chip, which is the more recent SPARClite with FPU.

With '`-mcpu=sparclet`', GCC generates code for the SPARClet variant of the SPARC architecture. This adds the integer multiply, multiply/accumulate, integer divide step and scan (`ffs`) instructions which exist in SPARClet but not in SPARC-V7. With '`-mcpu=tsc701`', the compiler additionally optimizes it for the TEMIC SPARClet chip.

With '`-mcpu=v9`', GCC generates code for the V9 variant of the SPARC architecture. This adds 64-bit integer and floating-point move instructions, 3 additional floating-point condition code registers and conditional move instructions. With '`-mcpu=ultrasparc`', the compiler additionally optimizes it for the Sun UltraSPARC I/II/IIi chips. With '`-mcpu=ultrasparc3`', the compiler additionally optimizes it for the Sun UltraSPARC III/III+/IIIi/IIIi+/IV/IV+ chips. With '`-mcpu=niagara`', the compiler additionally optimizes it for Sun UltraSPARC T1 chips. With '`-mcpu=niagara2`', the compiler additionally optimizes it for Sun UltraSPARC T2 chips. With '`-mcpu=niagara3`', the compiler additionally optimizes it for Sun UltraSPARC T3 chips. With '`-mcpu=niagara4`', the compiler additionally optimizes it for Sun UltraSPARC T4 chips. With '`-mcpu=niagara7`', the compiler additionally optimizes it for Oracle SPARC M7 chips. With '`-mcpu=m8`', the compiler additionally optimizes it for Oracle M8 chips.

`-mtune=`*`cpu_type`*

Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the instruction set or register set that the option '`-mcpu=`*`cpu_type`*' does.

The same values for '`-mcpu=`*`cpu_type`*' can be used for '`-mtune=`*`cpu_type`*', but the only useful values are those that select a particular CPU implementation.

Those are 'cypress', 'supersparc', 'hypersparc', 'leon', 'leon3', 'leon3v7', 'f930', 'f934', 'sparclite86x', 'tsc701', 'ultrasparc', 'ultrasparc3', 'niagara', 'niagara2', 'niagara3', 'niagara4', 'niagara7' and 'm8'. With native Solaris and GNU/Linux toolchains, 'native' can also be used.

-mv8plus
-mno-v8plus
: With '-mv8plus', GCC generates code for the SPARC-V8+ ABI. The difference from the V8 ABI is that the global and out registers are considered 64 bits wide. This is enabled by default on Solaris in 32-bit mode for all SPARC-V9 processors.

-mvis
-mno-vis
: With '-mvis', GCC generates code that takes advantage of the UltraSPARC Visual Instruction Set extensions. The default is '-mno-vis'.

-mvis2
-mno-vis2
: With '-mvis2', GCC generates code that takes advantage of version 2.0 of the UltraSPARC Visual Instruction Set extensions. The default is '-mvis2' when targeting a cpu that supports such instructions, such as UltraSPARC-III and later. Setting '-mvis2' also sets '-mvis'.

-mvis3
-mno-vis3
: With '-mvis3', GCC generates code that takes advantage of version 3.0 of the UltraSPARC Visual Instruction Set extensions. The default is '-mvis3' when targeting a cpu that supports such instructions, such as niagara-3 and later. Setting '-mvis3' also sets '-mvis2' and '-mvis'.

-mvis4
-mno-vis4
: With '-mvis4', GCC generates code that takes advantage of version 4.0 of the UltraSPARC Visual Instruction Set extensions. The default is '-mvis4' when targeting a cpu that supports such instructions, such as niagara-7 and later. Setting '-mvis4' also sets '-mvis3', '-mvis2' and '-mvis'.

-mvis4b
-mno-vis4b
: With '-mvis4b', GCC generates code that takes advantage of version 4.0 of the UltraSPARC Visual Instruction Set extensions, plus the additional VIS instructions introduced in the Oracle SPARC Architecture 2017. The default is '-mvis4b' when targeting a cpu that supports such instructions, such as m8 and later. Setting '-mvis4b' also sets '-mvis4', '-mvis3', '-mvis2' and '-mvis'.

-mcbcond
-mno-cbcond
: With '-mcbcond', GCC generates code that takes advantage of the UltraSPARC Compare-and-Branch-on-Condition instructions. The default is '-mcbcond' when targeting a CPU that supports such instructions, such as Niagara-4 and later.

`-mfmaf`
`-mno-fmaf`
With '`-mfmaf`', GCC generates code that takes advantage of the UltraSPARC Fused Multiply-Add Floating-point instructions. The default is '`-mfmaf`' when targeting a CPU that supports such instructions, such as Niagara-3 and later.

`-mfsmuld`
`-mno-fsmuld`
With '`-mfsmuld`', GCC generates code that takes advantage of the Floating-point Multiply Single to Double (FsMULd) instruction. The default is '`-mfsmuld`' when targeting a CPU supporting the architecture versions V8 or V9 with FPU except '`-mcpu=leon`'.

`-mpopc`
`-mno-popc`
With '`-mpopc`', GCC generates code that takes advantage of the UltraSPARC Population Count instruction. The default is '`-mpopc`' when targeting a CPU that supports such an instruction, such as Niagara-2 and later.

`-msubxc`
`-mno-subxc`
With '`-msubxc`', GCC generates code that takes advantage of the UltraSPARC Subtract-Extended-with-Carry instruction. The default is '`-msubxc`' when targeting a CPU that supports such an instruction, such as Niagara-7 and later.

`-mfix-at697f`
Enable the documented workaround for the single erratum of the Atmel AT697F processor (which corresponds to erratum #13 of the AT697E processor).

`-mfix-ut699`
Enable the documented workarounds for the floating-point errata and the data cache nullify errata of the UT699 processor.

`-mfix-ut700`
Enable the documented workaround for the back-to-back store errata of the UT699E/UT700 processor.

`-mfix-gr712rc`
Enable the documented workaround for the back-to-back store errata of the GR712RC processor.

These '`-m`' options are supported in addition to the above on SPARC-V9 processors in 64-bit environments:

`-m32`
`-m64` Generate code for a 32-bit or 64-bit environment. The 32-bit environment sets int, long and pointer to 32 bits. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits.

`-mcmodel=`*`which`*
Set the code model to one of

'medlow' The Medium/Low code model: 64-bit addresses, programs must be linked in the low 32 bits of memory. Programs can be statically or dynamically linked.

'medmid' The Medium/Middle code model: 64-bit addresses, programs must be linked in the low 44 bits of memory, the text and data segments must be less than 2GB in size and the data segment must be located within 2GB of the text segment.

'medany' The Medium/Anywhere code model: 64-bit addresses, programs may be linked anywhere in memory, the text and data segments must be less than 2GB in size and the data segment must be located within 2GB of the text segment.

'embmedany'
The Medium/Anywhere code model for embedded systems: 64-bit addresses, the text and data segments must be less than 2GB in size, both starting anywhere in memory (determined at link time). The global register %g4 points to the base of the data segment. Programs are statically linked and PIC is not supported.

-mmemory-model=*mem-model*
Set the memory model in force on the processor to one of

'default' The default memory model for the processor and operating system.

'rmo' Relaxed Memory Order

'pso' Partial Store Order

'tso' Total Store Order

'sc' Sequential Consistency

These memory models are formally defined in Appendix D of the SPARC-V9 architecture manual, as set in the processor's PSTATE.MM field.

-mstack-bias
-mno-stack-bias
With '-mstack-bias', GCC assumes that the stack pointer, and frame pointer if present, are offset by −2047 which must be added back when making stack frame references. This is the default in 64-bit mode. Otherwise, assume no such offset is present.

3.18.49 SPU Options

These '-m' options are supported on the SPU:

-mwarn-reloc
-merror-reloc
The loader for SPU does not handle dynamic relocations. By default, GCC gives an error when it generates code that requires a dynamic relocation. '-mno-error-reloc' disables the error, '-mwarn-reloc' generates a warning instead.

`-msafe-dma`
`-munsafe-dma`
: Instructions that initiate or test completion of DMA must not be reordered with respect to loads and stores of the memory that is being accessed. With '`-munsafe-dma`' you must use the `volatile` keyword to protect memory accesses, but that can lead to inefficient code in places where the memory is known to not change. Rather than mark the memory as volatile, you can use '`-msafe-dma`' to tell the compiler to treat the DMA instructions as potentially affecting all memory.

`-mbranch-hints`
: By default, GCC generates a branch hint instruction to avoid pipeline stalls for always-taken or probably-taken branches. A hint is not generated closer than 8 instructions away from its branch. There is little reason to disable them, except for debugging purposes, or to make an object a little bit smaller.

`-msmall-mem`
`-mlarge-mem`
: By default, GCC generates code assuming that addresses are never larger than 18 bits. With '`-mlarge-mem`' code is generated that assumes a full 32-bit address.

`-mstdmain`
: By default, GCC links against startup code that assumes the SPU-style main function interface (which has an unconventional parameter list). With '`-mstdmain`', GCC links your program against startup code that assumes a C99-style interface to `main`, including a local copy of `argv` strings.

`-mfixed-range=`*`register-range`*
: Generate code treating the given register range as fixed registers. A fixed register is one that the register allocator cannot use. This is useful when compiling kernel code. A register range is specified as two registers separated by a dash. Multiple register ranges can be specified separated by a comma.

`-mea32`
`-mea64`
: Compile code assuming that pointers to the PPU address space accessed via the `__ea` named address space qualifier are either 32 or 64 bits wide. The default is 32 bits. As this is an ABI-changing option, all object code in an executable must be compiled with the same setting.

`-maddress-space-conversion`
`-mno-address-space-conversion`
: Allow/disallow treating the `__ea` address space as superset of the generic address space. This enables explicit type casts between `__ea` and generic pointer as well as implicit conversions of generic pointers to `__ea` pointers. The default is to allow address space pointer conversions.

`-mcache-size=`*`cache-size`*
: This option controls the version of libgcc that the compiler links to an executable and selects a software-managed cache for accessing variables in the `__ea` address

space with a particular cache size. Possible options for *cache-size* are '8', '16', '32', '64' and '128'. The default cache size is 64KB.

-matomic-updates
-mno-atomic-updates
This option controls the version of libgcc that the compiler links to an executable and selects whether atomic updates to the software-managed cache of PPU-side variables are used. If you use atomic updates, changes to a PPU variable from SPU code using the `__ea` named address space qualifier do not interfere with changes to other PPU variables residing in the same cache line from PPU code. If you do not use atomic updates, such interference may occur; however, writing back cache lines is more efficient. The default behavior is to use atomic updates.

-mdual-nops
-mdual-nops=*n*
By default, GCC inserts NOPs to increase dual issue when it expects it to increase performance. *n* can be a value from 0 to 10. A smaller *n* inserts fewer NOPs. 10 is the default, 0 is the same as '-mno-dual-nops'. Disabled with '-Os'.

-mhint-max-nops=*n*
Maximum number of NOPs to insert for a branch hint. A branch hint must be at least 8 instructions away from the branch it is affecting. GCC inserts up to *n* NOPs to enforce this, otherwise it does not generate the branch hint.

-mhint-max-distance=*n*
The encoding of the branch hint instruction limits the hint to be within 256 instructions of the branch it is affecting. By default, GCC makes sure it is within 125.

-msafe-hints
Work around a hardware bug that causes the SPU to stall indefinitely. By default, GCC inserts the `hbrp` instruction to make sure this stall won't happen.

3.18.50 Options for System V

These additional options are available on System V Release 4 for compatibility with other compilers on those systems:

-G
Create a shared object. It is recommended that '-symbolic' or '-shared' be used instead.

-Qy
Identify the versions of each tool used by the compiler, in a `.ident` assembler directive in the output.

-Qn
Refrain from adding `.ident` directives to the output file (this is the default).

-YP,*dirs*
Search the directories *dirs*, and no others, for libraries specified with '-l'.

-Ym,*dir*
Look in the directory *dir* to find the M4 preprocessor. The assembler uses this option.

3.18.51 TILE-Gx Options

These '-m' options are supported on the TILE-Gx:

`-mcmodel=small`
: Generate code for the small model. The distance for direct calls is limited to 500M in either direction. PC-relative addresses are 32 bits. Absolute addresses support the full address range.

`-mcmodel=large`
: Generate code for the large model. There is no limitation on call distance, pc-relative addresses, or absolute addresses.

`-mcpu=name`
: Selects the type of CPU to be targeted. Currently the only supported type is 'tilegx'.

`-m32`
`-m64`
: Generate code for a 32-bit or 64-bit environment. The 32-bit environment sets int, long, and pointer to 32 bits. The 64-bit environment sets int to 32 bits and long and pointer to 64 bits.

`-mbig-endian`
`-mlittle-endian`
: Generate code in big/little endian mode, respectively.

3.18.52 TILEPro Options

These '-m' options are supported on the TILEPro:

`-mcpu=name`
: Selects the type of CPU to be targeted. Currently the only supported type is 'tilepro'.

`-m32`
: Generate code for a 32-bit environment, which sets int, long, and pointer to 32 bits. This is the only supported behavior so the flag is essentially ignored.

3.18.53 V850 Options

These '-m' options are defined for V850 implementations:

`-mlong-calls`
`-mno-long-calls`
: Treat all calls as being far away (near). If calls are assumed to be far away, the compiler always loads the function's address into a register, and calls indirect through the pointer.

`-mno-ep`
`-mep`
: Do not optimize (do optimize) basic blocks that use the same index pointer 4 or more times to copy pointer into the `ep` register, and use the shorter `sld` and `sst` instructions. The '-mep' option is on by default if you optimize.

`-mno-prolog-function`
`-mprolog-function`
: Do not use (do use) external functions to save and restore registers at the prologue and epilogue of a function. The external functions are slower, but use

less code space if more than one function saves the same number of registers. The '-mprolog-function' option is on by default if you optimize.

-mspace Try to make the code as small as possible. At present, this just turns on the '-mep' and '-mprolog-function' options.

-mtda=*n* Put static or global variables whose size is *n* bytes or less into the tiny data area that register ep points to. The tiny data area can hold up to 256 bytes in total (128 bytes for byte references).

-msda=*n* Put static or global variables whose size is *n* bytes or less into the small data area that register gp points to. The small data area can hold up to 64 kilobytes.

-mzda=*n* Put static or global variables whose size is *n* bytes or less into the first 32 kilobytes of memory.

-mv850 Specify that the target processor is the V850.

-mv850e3v5
Specify that the target processor is the V850E3V5. The preprocessor constant __v850e3v5__ is defined if this option is used.

-mv850e2v4
Specify that the target processor is the V850E3V5. This is an alias for the '-mv850e3v5' option.

-mv850e2v3
Specify that the target processor is the V850E2V3. The preprocessor constant __v850e2v3__ is defined if this option is used.

-mv850e2 Specify that the target processor is the V850E2. The preprocessor constant __v850e2__ is defined if this option is used.

-mv850e1 Specify that the target processor is the V850E1. The preprocessor constants __v850e1__ and __v850e__ are defined if this option is used.

-mv850es Specify that the target processor is the V850ES. This is an alias for the '-mv850e1' option.

-mv850e Specify that the target processor is the V850E. The preprocessor constant __v850e__ is defined if this option is used.

If neither '-mv850' nor '-mv850e' nor '-mv850e1' nor '-mv850e2' nor '-mv850e2v3' nor '-mv850e3v5' are defined then a default target processor is chosen and the relevant '__v850*__' preprocessor constant is defined.

The preprocessor constants __v850 and __v851__ are always defined, regardless of which processor variant is the target.

-mdisable-callt
-mno-disable-callt
This option suppresses generation of the CALLT instruction for the v850e, v850e1, v850e2, v850e2v3 and v850e3v5 flavors of the v850 architecture.

This option is enabled by default when the RH850 ABI is in use (see '-mrh850-abi'), and disabled by default when the GCC ABI is in use. If CALLT instructions are being generated then the C preprocessor symbol __V850_CALLT__ is defined.

`-mrelax`
`-mno-relax`
Pass on (or do not pass on) the '`-mrelax`' command-line option to the assembler.

`-mlong-jumps`
`-mno-long-jumps`
Disable (or re-enable) the generation of PC-relative jump instructions.

`-msoft-float`
`-mhard-float`
Disable (or re-enable) the generation of hardware floating point instructions. This option is only significant when the target architecture is '`V850E2V3`' or higher. If hardware floating point instructions are being generated then the C preprocessor symbol `__FPU_OK__` is defined, otherwise the symbol `__NO_FPU__` is defined.

`-mloop` Enables the use of the e3v5 LOOP instruction. The use of this instruction is not enabled by default when the e3v5 architecture is selected because its use is still experimental.

`-mrh850-abi`
`-mghs` Enables support for the RH850 version of the V850 ABI. This is the default. With this version of the ABI the following rules apply:

- Integer sized structures and unions are returned via a memory pointer rather than a register.
- Large structures and unions (more than 8 bytes in size) are passed by value.
- Functions are aligned to 16-bit boundaries.
- The '`-m8byte-align`' command-line option is supported.
- The '`-mdisable-callt`' command-line option is enabled by default. The '`-mno-disable-callt`' command-line option is not supported.

When this version of the ABI is enabled the C preprocessor symbol `__V850_RH850_ABI__` is defined.

`-mgcc-abi`
Enables support for the old GCC version of the V850 ABI. With this version of the ABI the following rules apply:

- Integer sized structures and unions are returned in register `r10`.
- Large structures and unions (more than 8 bytes in size) are passed by reference.
- Functions are aligned to 32-bit boundaries, unless optimizing for size.
- The '`-m8byte-align`' command-line option is not supported.
- The '`-mdisable-callt`' command-line option is supported but not enabled by default.

When this version of the ABI is enabled the C preprocessor symbol `__V850_GCC_ABI__` is defined.

`-m8byte-align`
`-mno-8byte-align`
: Enables support for `double` and `long long` types to be aligned on 8-byte boundaries. The default is to restrict the alignment of all objects to at most 4-bytes. When '`-m8byte-align`' is in effect the C preprocessor symbol `__V850_8BYTE_ALIGN__` is defined.

`-mbig-switch`
: Generate code suitable for big switch tables. Use this option only if the assembler/linker complain about out of range branches within a switch table.

`-mapp-regs`
: This option causes r2 and r5 to be used in the code generated by the compiler. This setting is the default.

`-mno-app-regs`
: This option causes r2 and r5 to be treated as fixed registers.

3.18.54 VAX Options

These '`-m`' options are defined for the VAX:

`-munix` Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the VAX cannot handle across long ranges.

`-mgnu` Do output those jump instructions, on the assumption that the GNU assembler is being used.

`-mg` Output code for G-format floating-point numbers instead of D-format.

3.18.55 Visium Options

`-mdebug` A program which performs file I/O and is destined to run on an MCM target should be linked with this option. It causes the libraries libc.a and libdebug.a to be linked. The program should be run on the target under the control of the GDB remote debugging stub.

`-msim` A program which performs file I/O and is destined to run on the simulator should be linked with option. This causes libraries libc.a and libsim.a to be linked.

`-mfpu`
`-mhard-float`
: Generate code containing floating-point instructions. This is the default.

`-mno-fpu`
`-msoft-float`
: Generate code containing library calls for floating-point.

  '`-msoft-float`' changes the calling convention in the output file; therefore, it is only useful if you compile *all* of a program with this option. In particular, you need to compile '`libgcc.a`', the library that comes with GCC, with '`-msoft-float`' in order for this to work.

`-mcpu=cpu_type`
: Set the instruction set, register set, and instruction scheduling parameters for machine type *cpu_type*. Supported values for *cpu_type* are '`mcm`', '`gr5`' and '`gr6`'.

  '`mcm`' is a synonym of '`gr5`' present for backward compatibility.

  By default (unless configured otherwise), GCC generates code for the GR5 variant of the Visium architecture.

  With '`-mcpu=gr6`', GCC generates code for the GR6 variant of the Visium architecture. The only difference from GR5 code is that the compiler will generate block move instructions.

`-mtune=cpu_type`
: Set the instruction scheduling parameters for machine type *cpu_type*, but do not set the instruction set or register set that the option '`-mcpu=cpu_type`' would.

`-msv-mode`
: Generate code for the supervisor mode, where there are no restrictions on the access to general registers. This is the default.

`-muser-mode`
: Generate code for the user mode, where the access to some general registers is forbidden: on the GR5, registers r24 to r31 cannot be accessed in this mode; on the GR6, only registers r29 to r31 are affected.

3.18.56 VMS Options

These '`-m`' options are defined for the VMS implementations:

`-mvms-return-codes`
: Return VMS condition codes from `main`. The default is to return POSIX-style condition (e.g. error) codes.

`-mdebug-main=prefix`
: Flag the first routine whose name starts with *prefix* as the main routine for the debugger.

`-mmalloc64`
: Default to 64-bit memory allocation routines.

`-mpointer-size=size`
: Set the default size of pointers. Possible options for *size* are '`32`' or '`short`' for 32 bit pointers, '`64`' or '`long`' for 64 bit pointers, and '`no`' for supporting only 32 bit pointers. The later option disables `pragma pointer_size`.

3.18.57 VxWorks Options

The options in this section are defined for all VxWorks targets. Options specific to the target hardware are listed with the other options for that target.

`-mrtp`
: GCC can generate code for both VxWorks kernels and real time processes (RTPs). This option switches from the former to the latter. It also defines the preprocessor macro `__RTP__`.

`-non-static`
: Link an RTP executable against shared libraries rather than static libraries. The options '`-static`' and '`-shared`' can also be used for RTPs (see Section 3.14 [Link Options], page 209); '`-static`' is the default.

`-Bstatic`
`-Bdynamic`
: These options are passed down to the linker. They are defined for compatibility with Diab.

`-Xbind-lazy`
: Enable lazy binding of function calls. This option is equivalent to '`-Wl,-z,now`' and is defined for compatibility with Diab.

`-Xbind-now`
: Disable lazy binding of function calls. This option is the default and is defined for compatibility with Diab.

3.18.58 x86 Options

These '`-m`' options are defined for the x86 family of computers.

`-march=`*`cpu-type`*
: Generate instructions for the machine type *cpu-type*. In contrast to '`-mtune=`*`cpu-type`*', which merely tunes the generated code for the specified *cpu-type*, '`-march=`*`cpu-type`*' allows GCC to generate code that may not run at all on processors other than the one indicated. Specifying '`-march=`*`cpu-type`*' implies '`-mtune=`*`cpu-type`*'.

  The choices for *cpu-type* are:

  '`native`'
  : This selects the CPU to generate code for at compilation time by determining the processor type of the compiling machine. Using '`-march=native`' enables all instruction subsets supported by the local machine (hence the result might not run on different machines). Using '`-mtune=native`' produces code optimized for the local machine under the constraints of the selected instruction set.

  '`x86-64`'
  : A generic CPU with 64-bit extensions.

  '`i386`'
  : Original Intel i386 CPU.

  '`i486`'
  : Intel i486 CPU. (No scheduling is implemented for this chip.)

  '`i586`'
  '`pentium`'
  : Intel Pentium CPU with no MMX support.

  '`lakemont`'
  : Intel Lakemont MCU, based on Intel Pentium CPU.

  '`pentium-mmx`'
  : Intel Pentium MMX CPU, based on Pentium core with MMX instruction set support.

  '`pentiumpro`'
  : Intel Pentium Pro CPU.

'i686' When used with '-march', the Pentium Pro instruction set is used, so the code runs on all i686 family chips. When used with '-mtune', it has the same meaning as 'generic'.

'pentium2'
Intel Pentium II CPU, based on Pentium Pro core with MMX instruction set support.

'pentium3'
'pentium3m'
Intel Pentium III CPU, based on Pentium Pro core with MMX and SSE instruction set support.

'pentium-m'
Intel Pentium M; low-power version of Intel Pentium III CPU with MMX, SSE and SSE2 instruction set support. Used by Centrino notebooks.

'pentium4'
'pentium4m'
Intel Pentium 4 CPU with MMX, SSE and SSE2 instruction set support.

'prescott'
Improved version of Intel Pentium 4 CPU with MMX, SSE, SSE2 and SSE3 instruction set support.

'nocona' Improved version of Intel Pentium 4 CPU with 64-bit extensions, MMX, SSE, SSE2 and SSE3 instruction set support.

'core2' Intel Core 2 CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support.

'nehalem' Intel Nehalem CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2 and POPCNT instruction set support.

'westmere'
Intel Westmere CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AES and PCLMUL instruction set support.

'sandybridge'
Intel Sandy Bridge CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AES and PCLMUL instruction set support.

'ivybridge'
Intel Ivy Bridge CPU with 64-bit extensions, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AES, PCLMUL, FSGSBASE, RDRND and F16C instruction set support.

'haswell' Intel Haswell CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2,

AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2 and F16C instruction set support.

'broadwell'
: Intel Broadwell CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX and PREFETCHW instruction set support.

'skylake'
: Intel Skylake CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC and XSAVES instruction set support.

'bonnell'
: Intel Bonnell CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support.

'silvermont'
: Intel Silvermont CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AES, PCLMUL and RDRND instruction set support.

'goldmont'
: Intel Goldmont CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AES, PCLMUL, RDRND, XSAVE, XSAVEOPT and FSGSBASE instruction set support.

'goldmont-plus'
: Intel Goldmont Plus CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AES, PCLMUL, RDRND, XSAVE, XSAVEOPT, FSGSBASE, PTWRITE, RDPID, SGX and UMIP instruction set support.

'tremont'
: Intel Tremont CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AES, PCLMUL, RDRND, XSAVE, XSAVEOPT, FSGSBASE, PTWRITE, RDPID, SGX, UMIP, GFNI-SSE, CLWB and ENCLV instruction set support.

'knl'
: Intel Knight's Landing CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, AVX512F, AVX512PF, AVX512ER and AVX512CD instruction set support.

'knm'
: Intel Knights Mill CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, AVX512F, AVX512PF, AVX512ER, AVX512CD, AVX5124VNNIW, AVX5124FMAPS and AVX512VPOPCNTDQ instruction set support.

‘`skylake-avx512`’
Intel Skylake Server CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, PKU, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC, XSAVES, AVX512F, CLWB, AVX512VL, AVX512BW, AVX512DQ and AVX512CD instruction set support.

‘`cannonlake`’
Intel Cannonlake Server CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, PKU, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC, XSAVES, AVX512F, AVX512VL, AVX512BW, AVX512DQ, AVX512CD, AVX512VBMI, AVX512IFMA, SHA and UMIP instruction set support.

‘`icelake-client`’
Intel Icelake Client CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, PKU, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC, XSAVES, AVX512F, AVX512VL, AVX512BW, AVX512DQ, AVX512CD, AVX512VBMI, AVX512IFMA, SHA, CLWB, UMIP, RDPID, GFNI, AVX512VBMI2, AVX512VPOPCNTDQ, AVX512BITALG, AVX512VNNI, VPCLMULQDQ, VAES instruction set support.

‘`icelake-server`’
Intel Icelake Server CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, PKU, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC, XSAVES, AVX512F, AVX512VL, AVX512BW, AVX512DQ, AVX512CD, AVX512VBMI, AVX512IFMA, SHA, CLWB, UMIP, RDPID, GFNI, AVX512VBMI2, AVX512VPOPCNTDQ, AVX512BITALG, AVX512VNNI, VPCLMULQDQ, VAES, PCONFIG and WBNOINVD instruction set support.

‘`cascadelake`’
Intel Cascadelake CPU with 64-bit extensions, MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, PKU, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2, F16C, RDSEED, ADCX, PREFETCHW, CLFLUSHOPT, XSAVEC, XSAVES, AVX512F, CLWB, AVX512VL, AVX512BW, AVX512DQ, AVX512CD and AVX512VNNI instruction set support.

‘`k6`’ AMD K6 CPU with MMX instruction set support.

'k6-2'
'k6-3' Improved versions of AMD K6 CPU with MMX and 3DNow! instruction set support.

'athlon'
'athlon-tbird'
AMD Athlon CPU with MMX, 3dNOW!, enhanced 3DNow! and SSE prefetch instructions support.

'athlon-4'
'athlon-xp'
'athlon-mp'
Improved AMD Athlon CPU with MMX, 3DNow!, enhanced 3DNow! and full SSE instruction set support.

'k8'
'opteron'
'athlon64'
'athlon-fx'
Processors based on the AMD K8 core with x86-64 instruction set support, including the AMD Opteron, Athlon 64, and Athlon 64 FX processors. (This supersets MMX, SSE, SSE2, 3DNow!, enhanced 3DNow! and 64-bit instruction set extensions.)

'k8-sse3'
'opteron-sse3'
'athlon64-sse3'
Improved versions of AMD K8 cores with SSE3 instruction set support.

'amdfam10'
'barcelona'
CPUs based on AMD Family 10h cores with x86-64 instruction set support. (This supersets MMX, SSE, SSE2, SSE3, SSE4A, 3DNow!, enhanced 3DNow!, ABM and 64-bit instruction set extensions.)

'bdver1' CPUs based on AMD Family 15h cores with x86-64 instruction set support. (This supersets FMA4, AVX, XOP, LWP, AES, PCL_MUL, CX16, MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM and 64-bit instruction set extensions.)

'bdver2' AMD Family 15h core based CPUs with x86-64 instruction set support. (This supersets BMI, TBM, F16C, FMA, FMA4, AVX, XOP, LWP, AES, PCL_MUL, CX16, MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM and 64-bit instruction set extensions.)

'bdver3' AMD Family 15h core based CPUs with x86-64 instruction set support. (This supersets BMI, TBM, F16C, FMA, FMA4, FSGSBASE, AVX, XOP, LWP, AES, PCL_MUL, CX16, MMX, SSE,

SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM and 64-bit instruction set extensions.

'bdver4' AMD Family 15h core based CPUs with x86-64 instruction set support. (This supersets BMI, BMI2, TBM, F16C, FMA, FMA4, FSGSBASE, AVX, AVX2, XOP, LWP, AES, PCL_MUL, CX16, MOVBE, MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM and 64-bit instruction set extensions.

'znver1' AMD Family 17h core based CPUs with x86-64 instruction set support. (This supersets BMI, BMI2, F16C, FMA, FSGSBASE, AVX, AVX2, ADCX, RDSEED, MWAITX, SHA, CLZERO, AES, PCL_MUL, CX16, MOVBE, MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM, XSAVEC, XSAVES, CLFLUSHOPT, POPCNT, and 64-bit instruction set extensions.

'znver2' AMD Family 17h core based CPUs with x86-64 instruction set support. (This supersets BMI, BMI2, ,CLWB, F16C, FMA, FSGSBASE, AVX, AVX2, ADCX, RDSEED, MWAITX, SHA, CLZERO, AES, PCL_MUL, CX16, MOVBE, MMX, SSE, SSE2, SSE3, SSE4A, SSSE3, SSE4.1, SSE4.2, ABM, XSAVEC, XSAVES, CLFLUSHOPT, POPCNT, and 64-bit instruction set extensions.)

'btver1' CPUs based on AMD Family 14h cores with x86-64 instruction set support. (This supersets MMX, SSE, SSE2, SSE3, SSSE3, SSE4A, CX16, ABM and 64-bit instruction set extensions.)

'btver2' CPUs based on AMD Family 16h cores with x86-64 instruction set support. This includes MOVBE, F16C, BMI, AVX, PCL_MUL, AES, SSE4.2, SSE4.1, CX16, ABM, SSE4A, SSSE3, SSE3, SSE2, SSE, MMX and 64-bit instruction set extensions.

'winchip-c6'
IDT WinChip C6 CPU, dealt in same way as i486 with additional MMX instruction set support.

'winchip2'
IDT WinChip 2 CPU, dealt in same way as i486 with additional MMX and 3DNow! instruction set support.

'c3' VIA C3 CPU with MMX and 3DNow! instruction set support. (No scheduling is implemented for this chip.)

'c3-2' VIA C3-2 (Nehemiah/C5XL) CPU with MMX and SSE instruction set support. (No scheduling is implemented for this chip.)

'c7' VIA C7 (Esther) CPU with MMX, SSE, SSE2 and SSE3 instruction set support. (No scheduling is implemented for this chip.)

'samuel-2'
VIA Eden Samuel 2 CPU with MMX and 3DNow! instruction set support. (No scheduling is implemented for this chip.)

'nehemiah'
VIA Eden Nehemiah CPU with MMX and SSE instruction set support. (No scheduling is implemented for this chip.)

'esther' VIA Eden Esther CPU with MMX, SSE, SSE2 and SSE3 instruction set support. (No scheduling is implemented for this chip.)

'eden-x2' VIA Eden X2 CPU with x86-64, MMX, SSE, SSE2 and SSE3 instruction set support. (No scheduling is implemented for this chip.)

'eden-x4' VIA Eden X4 CPU with x86-64, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX and AVX2 instruction set support. (No scheduling is implemented for this chip.)

'nano' Generic VIA Nano CPU with x86-64, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support. (No scheduling is implemented for this chip.)

'nano-1000'
VIA Nano 1xxx CPU with x86-64, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support. (No scheduling is implemented for this chip.)

'nano-2000'
VIA Nano 2xxx CPU with x86-64, MMX, SSE, SSE2, SSE3 and SSSE3 instruction set support. (No scheduling is implemented for this chip.)

'nano-3000'
VIA Nano 3xxx CPU with x86-64, MMX, SSE, SSE2, SSE3, SSSE3 and SSE4.1 instruction set support. (No scheduling is implemented for this chip.)

'nano-x2' VIA Nano Dual Core CPU with x86-64, MMX, SSE, SSE2, SSE3, SSSE3 and SSE4.1 instruction set support. (No scheduling is implemented for this chip.)

'nano-x4' VIA Nano Quad Core CPU with x86-64, MMX, SSE, SSE2, SSE3, SSSE3 and SSE4.1 instruction set support. (No scheduling is implemented for this chip.)

'geode' AMD Geode embedded processor with MMX and 3DNow! instruction set support.

-mtune=*cpu-type*

Tune to *cpu-type* everything applicable about the generated code, except for the ABI and the set of available instructions. While picking a specific *cpu-type* schedules things appropriately for that particular chip, the compiler does not generate any code that cannot run on the default machine type unless you use a '-march=*cpu-type*' option. For example, if GCC is configured for i686-pc-linux-gnu then '-mtune=pentium4' generates code that is tuned for Pentium 4 but still runs on i686 machines.

The choices for *cpu-type* are the same as for '-march'. In addition, '-mtune' supports 2 extra choices for *cpu-type*:

'`generic`' Produce code optimized for the most common IA32/AMD64/EM64T processors. If you know the CPU on which your code will run, then you should use the corresponding '`-mtune`' or '`-march`' option instead of '`-mtune=generic`'. But, if you do not know exactly what CPU users of your application will have, then you should use this option.

As new processors are deployed in the marketplace, the behavior of this option will change. Therefore, if you upgrade to a newer version of GCC, code generation controlled by this option will change to reflect the processors that are most common at the time that version of GCC is released.

There is no '`-march=generic`' option because '`-march`' indicates the instruction set the compiler can use, and there is no generic instruction set applicable to all processors. In contrast, '`-mtune`' indicates the processor (or, in this case, collection of processors) for which the code is optimized.

'`intel`' Produce code optimized for the most current Intel processors, which are Haswell and Silvermont for this version of GCC. If you know the CPU on which your code will run, then you should use the corresponding '`-mtune`' or '`-march`' option instead of '`-mtune=intel`'. But, if you want your application performs better on both Haswell and Silvermont, then you should use this option.

As new Intel processors are deployed in the marketplace, the behavior of this option will change. Therefore, if you upgrade to a newer version of GCC, code generation controlled by this option will change to reflect the most current Intel processors at the time that version of GCC is released.

There is no '`-march=intel`' option because '`-march`' indicates the instruction set the compiler can use, and there is no common instruction set applicable to all processors. In contrast, '`-mtune`' indicates the processor (or, in this case, collection of processors) for which the code is optimized.

`-mcpu=`*`cpu-type`*

A deprecated synonym for '`-mtune`'.

`-mfpmath=`*`unit`*

Generate floating-point arithmetic for selected unit *unit*. The choices for *unit* are:

'`387`' Use the standard 387 floating-point coprocessor present on the majority of chips and emulated otherwise. Code compiled with this option runs almost everywhere. The temporary results are computed in 80-bit precision instead of the precision specified by the type, resulting in slightly different results compared to most of other chips. See '`-ffloat-store`' for more detailed description.

This is the default choice for non-Darwin x86-32 targets.

'sse' Use scalar floating-point instructions present in the SSE instruction set. This instruction set is supported by Pentium III and newer chips, and in the AMD line by Athlon-4, Athlon XP and Athlon MP chips. The earlier version of the SSE instruction set supports only single-precision arithmetic, thus the double and extended-precision arithmetic are still done using 387. A later version, present only in Pentium 4 and AMD x86-64 chips, supports double-precision arithmetic too.

For the x86-32 compiler, you must use '-march=*cpu-type*', '-msse' or '-msse2' switches to enable SSE extensions and make this option effective. For the x86-64 compiler, these extensions are enabled by default.

The resulting code should be considerably faster in the majority of cases and avoid the numerical instability problems of 387 code, but may break some existing code that expects temporaries to be 80 bits.

This is the default choice for the x86-64 compiler, Darwin x86-32 targets, and the default choice for x86-32 targets with the SSE2 instruction set when '-ffast-math' is enabled.

'sse,387'
'sse+387'
'both' Attempt to utilize both instruction sets at once. This effectively doubles the amount of available registers, and on chips with separate execution units for 387 and SSE the execution resources too. Use this option with care, as it is still experimental, because the GCC register allocator does not model separate functional units well, resulting in unstable performance.

-masm=*dialect*
Output assembly instructions using selected *dialect*. Also affects which dialect is used for basic asm (see Section 6.47.1 [Basic Asm], page 567) and extended asm (see Section 6.47.2 [Extended Asm], page 569). Supported choices (in dialect order) are 'att' or 'intel'. The default is 'att'. Darwin does not support 'intel'.

-mieee-fp
-mno-ieee-fp
Control whether or not the compiler uses IEEE floating-point comparisons. These correctly handle the case where the result of a comparison is unordered.

-m80387
-mhard-float
Generate output containing 80387 instructions for floating point.

-mno-80387
-msoft-float
Generate output containing library calls for floating point.

Warning: the requisite libraries are not part of GCC. Normally the facilities of the machine's usual C compiler are used, but this cannot be done directly in cross-compilation. You must make your own arrangements to provide suitable library functions for cross-compilation.

On machines where a function returns floating-point results in the 80387 register stack, some floating-point opcodes may be emitted even if '`-msoft-float`' is used.

`-mno-fp-ret-in-387`

Do not use the FPU registers for return values of functions.

The usual calling convention has functions return values of types `float` and `double` in an FPU register, even if there is no FPU. The idea is that the operating system should emulate an FPU.

The option '`-mno-fp-ret-in-387`' causes such values to be returned in ordinary CPU registers instead.

`-mno-fancy-math-387`

Some 387 emulators do not support the `sin`, `cos` and `sqrt` instructions for the 387. Specify this option to avoid generating those instructions. This option is overridden when '`-march`' indicates that the target CPU always has an FPU and so the instruction does not need emulation. These instructions are not generated unless you also use the '`-funsafe-math-optimizations`' switch.

`-malign-double`
`-mno-align-double`

Control whether GCC aligns `double`, `long double`, and `long long` variables on a two-word boundary or a one-word boundary. Aligning `double` variables on a two-word boundary produces code that runs somewhat faster on a Pentium at the expense of more memory.

On x86-64, '`-malign-double`' is enabled by default.

Warning: if you use the '`-malign-double`' switch, structures containing the above types are aligned differently than the published application binary interface specifications for the x86-32 and are not binary compatible with structures in code compiled without that switch.

`-m96bit-long-double`
`-m128bit-long-double`

These switches control the size of `long double` type. The x86-32 application binary interface specifies the size to be 96 bits, so '`-m96bit-long-double`' is the default in 32-bit mode.

Modern architectures (Pentium and newer) prefer `long double` to be aligned to an 8- or 16-byte boundary. In arrays or structures conforming to the ABI, this is not possible. So specifying '`-m128bit-long-double`' aligns `long double` to a 16-byte boundary by padding the `long double` with an additional 32-bit zero.

In the x86-64 compiler, '`-m128bit-long-double`' is the default choice as its ABI specifies that `long double` is aligned on 16-byte boundary.

Notice that neither of these options enable any extra precision over the x87 standard of 80 bits for a `long double`.

Warning: if you override the default value for your target ABI, this changes the size of structures and arrays containing `long double` variables, as well as modifying the function calling convention for functions taking `long double`. Hence they are not binary-compatible with code compiled without that switch.

`-mlong-double-64`
`-mlong-double-80`
`-mlong-double-128`

These switches control the size of `long double` type. A size of 64 bits makes the `long double` type equivalent to the `double` type. This is the default for 32-bit Bionic C library. A size of 128 bits makes the `long double` type equivalent to the `__float128` type. This is the default for 64-bit Bionic C library.

Warning: if you override the default value for your target ABI, this changes the size of structures and arrays containing `long double` variables, as well as modifying the function calling convention for functions taking `long double`. Hence they are not binary-compatible with code compiled without that switch.

`-malign-data=`*type*

Control how GCC aligns variables. Supported values for *type* are '`compat`' uses increased alignment value compatible uses GCC 4.8 and earlier, '`abi`' uses alignment value as specified by the psABI, and '`cacheline`' uses increased alignment value to match the cache line size. '`compat`' is the default.

`-mlarge-data-threshold=`*threshold*

When '`-mcmodel=medium`' is specified, data objects larger than *threshold* are placed in the large data section. This value must be the same across all objects linked into the binary, and defaults to 65535.

`-mrtd` Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `ret` *num* instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

You can specify that an individual function is called with this calling sequence with the function attribute `stdcall`. You can also override the '`-mrtd`' option by using the function attribute `cdecl`. See Section 6.33 [Function Attributes], page 477.

Warning: this calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code is generated for calls to those functions.

In addition, seriously incorrect code results if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

`-mregparm=num`

Control how many registers are used to pass integer arguments. By default, no registers are used to pass arguments, and at most 3 registers can be used. You can control this behavior for a specific function by using the function attribute `regparm`. See Section 6.33 [Function Attributes], page 477.

Warning: if you use this switch, and *num* is nonzero, then you must build all modules with the same value, including any libraries. This includes the system libraries and startup modules.

`-msseregparm`

Use SSE register passing conventions for float and double arguments and return values. You can control this behavior for a specific function by using the function attribute `sseregparm`. See Section 6.33 [Function Attributes], page 477.

Warning: if you use this switch then you must build all modules with the same value, including any libraries. This includes the system libraries and startup modules.

`-mvect8-ret-in-mem`

Return 8-byte vectors in memory instead of MMX registers. This is the default on Solaris 8 and 9 and VxWorks to match the ABI of the Sun Studio compilers until version 12. Later compiler versions (starting with Studio 12 Update 1) follow the ABI used by other x86 targets, which is the default on Solaris 10 and later. *Only* use this option if you need to remain compatible with existing code produced by those previous compiler versions or older versions of GCC.

`-mpc32`
`-mpc64`
`-mpc80`

Set 80387 floating-point precision to 32, 64 or 80 bits. When '`-mpc32`' is specified, the significands of results of floating-point operations are rounded to 24 bits (single precision); '`-mpc64`' rounds the significands of results of floating-point operations to 53 bits (double precision) and '`-mpc80`' rounds the significands of results of floating-point operations to 64 bits (extended double precision), which is the default. When this option is used, floating-point operations in higher precisions are not available to the programmer without setting the FPU control word explicitly.

Setting the rounding of floating-point operations to less than the default 80 bits can speed some programs by 2% or more. Note that some mathematical libraries assume that extended-precision (80-bit) floating-point operations are enabled by default; routines in such libraries could suffer significant loss of accuracy, typically through so-called "catastrophic cancellation", when this option is used to set the precision to less than extended precision.

`-mstackrealign`

Realign the stack at entry. On the x86, the '`-mstackrealign`' option generates an alternate prologue and epilogue that realigns the run-time stack if necessary. This supports mixing legacy codes that keep 4-byte stack alignment with modern codes that keep 16-byte stack alignment for SSE compatibility. See also the attribute `force_align_arg_pointer`, applicable to individual functions.

`-mpreferred-stack-boundary=num`

Attempt to keep the stack boundary aligned to a 2 raised to *num* byte boundary. If '`-mpreferred-stack-boundary`' is not specified, the default is 4 (16 bytes or 128 bits).

Warning: When generating code for the x86-64 architecture with SSE extensions disabled, '`-mpreferred-stack-boundary=3`' can be used to keep the stack boundary aligned to 8 byte boundary. Since x86-64 ABI require 16 byte stack alignment, this is ABI incompatible and intended to be used in controlled environment where stack space is important limitation. This option leads to wrong code when functions compiled with 16 byte stack alignment (such as functions from a standard library) are called with misaligned stack. In this case, SSE instructions may lead to misaligned memory access traps. In addition, variable arguments are handled incorrectly for 16 byte aligned objects (including x87 long double and __int128), leading to wrong results. You must build all modules with '`-mpreferred-stack-boundary=3`', including any libraries. This includes the system libraries and startup modules.

`-mincoming-stack-boundary=num`

Assume the incoming stack is aligned to a 2 raised to *num* byte boundary. If '`-mincoming-stack-boundary`' is not specified, the one specified by '`-mpreferred-stack-boundary`' is used.

On Pentium and Pentium Pro, `double` and `long double` values should be aligned to an 8-byte boundary (see '`-malign-double`') or suffer significant run time performance penalties. On Pentium III, the Streaming SIMD Extension (SSE) data type `__m128` may not work properly if it is not 16-byte aligned.

To ensure proper alignment of this values on the stack, the stack boundary must be as aligned as that required by any value stored on the stack. Further, every function must be generated such that it keeps the stack aligned. Thus calling a function compiled with a higher preferred stack boundary from a function compiled with a lower preferred stack boundary most likely misaligns the stack. It is recommended that libraries that use callbacks always use the default setting.

This extra alignment does consume extra stack space, and generally increases code size. Code that is sensitive to stack space usage, such as embedded systems and operating system kernels, may want to reduce the preferred alignment to '`-mpreferred-stack-boundary=2`'.

`-mmmx`
`-msse`
`-msse2`
`-msse3`
`-mssse3`
`-msse4`
`-msse4a`
`-msse4.1`
`-msse4.2`
`-mavx`
`-mavx2`

```
-mavx512f
-mavx512pf
-mavx512er
-mavx512cd
-mavx512vl
-mavx512bw
-mavx512dq
-mavx512ifma
-mavx512vbmi
-msha
-maes
-mpclmul
-mclflushopt
-mclwb
-mfsgsbase
-mptwrite
-mrdrnd
-mf16c
-mfma
-mpconfig
-mwbnoinvd
-mfma4
-mprfchw
-mrdpid
-mprefetchwt1
-mrdseed
-msgx
-mxop
-mlwp
-m3dnow
-m3dnowa
-mpopcnt
-mabm
-madx
-mbmi
-mbmi2
-mlzcnt
-mfxsr
-mxsave
-mxsaveopt
-mxsavec
-mxsaves
-mrtm
-mhle
-mtbm
-mmwaitx
-mclzero
-mpku
-mavx512vbmi2
-mgfni
-mvaes
```

`-mwaitpkg`
`-mvpclmulqdq`
`-mavx512bitalg`
`-mmovdiri`
`-mmovdir64b`
`-mavx512vpopcntdq`
`-mavx5124fmaps`
`-mavx512vnni`
`-mavx5124vnniw`
`-mcldemote`

These switches enable the use of instructions in the MMX, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4A, SSE4.1, SSE4.2, AVX, AVX2, AVX512F, AVX512PF, AVX512ER, AVX512CD, AVX512VL, AVX512BW, AVX512DQ, AVX512IFMA, AVX512VBMI, SHA, AES, PCLMUL, CLFLUSHOPT, CLWB, FSGSBASE, PTWRITE, RDRND, F16C, FMA, PCONFIG, WBNOINVD, FMA4, PREFETCHW, RDPID, PREFETCHWT1, RDSEED, SGX, XOP, LWP, 3DNow!, enhanced 3DNow!, POPCNT, ABM, ADX, BMI, BMI2, LZCNT, FXSR, XSAVE, XSAVEOPT, XSAVEC, XSAVES, RTM, HLE, TBM, MWAITX, CLZERO, PKU, AVX512VBMI2, GFNI, VAES, WAITPKG, VPCLMULQDQ, AVX512BITALG, MOVDIRI, MOVDIR64B, AVX512VPOPCNTDQ, AVX5124FMAPS, AVX512VNNI, AVX5124VNNIW, or CLDEMOTE extended instruction sets. Each has a corresponding '`-mno-`' option to disable use of these instructions.

These extensions are also available as built-in functions: see Section 6.60.33 [x86 Built-in Functions], page 774, for details of the functions enabled and disabled by these switches.

To generate SSE/SSE2 instructions automatically from floating-point code (as opposed to 387 instructions), see '`-mfpmath=sse`'.

GCC depresses SSEx instructions when '`-mavx`' is used. Instead, it generates new AVX instructions or AVX equivalence for all SSEx instructions when needed.

These options enable GCC to use these extended instructions in generated code, even without '`-mfpmath=sse`'. Applications that perform run-time CPU detection must compile separate files for each supported architecture, using the appropriate flags. In particular, the file containing the CPU detection code should be compiled without these options.

`-mdump-tune-features`

This option instructs GCC to dump the names of the x86 performance tuning features and default settings. The names can be used in '`-mtune-ctrl=`*`feature-list`*'.

`-mtune-ctrl=`*`feature-list`*

This option is used to do fine grain control of x86 code generation features. *feature-list* is a comma separated list of *feature* names. See also '`-mdump-tune-features`'. When specified, the *feature* is turned on if it is not preceded with '`^`', otherwise, it is turned off. '`-mtune-ctrl=`*`feature-list`*' is intended to be used by GCC developers. Using it may lead to code paths

not covered by testing and can potentially result in compiler ICEs or runtime errors.

`-mno-default`
This option instructs GCC to turn off all tunable features. See also '`-mtune-ctrl=feature-list`' and '`-mdump-tune-features`'.

`-mcld` This option instructs GCC to emit a `cld` instruction in the prologue of functions that use string instructions. String instructions depend on the DF flag to select between autoincrement or autodecrement mode. While the ABI specifies the DF flag to be cleared on function entry, some operating systems violate this specification by not clearing the DF flag in their exception dispatchers. The exception handler can be invoked with the DF flag set, which leads to wrong direction mode when string instructions are used. This option can be enabled by default on 32-bit x86 targets by configuring GCC with the '`--enable-cld`' configure option. Generation of `cld` instructions can be suppressed with the '`-mno-cld`' compiler option in this case.

`-mvzeroupper`
This option instructs GCC to emit a `vzeroupper` instruction before a transfer of control flow out of the function to minimize the AVX to SSE transition penalty as well as remove unnecessary `zeroupper` intrinsics.

`-mprefer-avx128`
This option instructs GCC to use 128-bit AVX instructions instead of 256-bit AVX instructions in the auto-vectorizer.

`-mprefer-vector-width=opt`
This option instructs GCC to use *opt*-bit vector width in instructions instead of default on the selected platform.

'`none`' No extra limitations applied to GCC other than defined by the selected platform.

'`128`' Prefer 128-bit vector width for instructions.

'`256`' Prefer 256-bit vector width for instructions.

'`512`' Prefer 512-bit vector width for instructions.

`-mcx16` This option enables GCC to generate `CMPXCHG16B` instructions in 64-bit code to implement compare-and-exchange operations on 16-byte aligned 128-bit objects. This is useful for atomic updates of data structures exceeding one machine word in size. The compiler uses this instruction to implement Section 6.54 [__sync Builtins], page 630. However, for Section 6.55 [__atomic Builtins], page 632 operating on 128-bit integers, a library call is always used.

`-msahf` This option enables generation of `SAHF` instructions in 64-bit code. Early Intel Pentium 4 CPUs with Intel 64 support, prior to the introduction of Pentium 4 G1 step in December 2005, lacked the `LAHF` and `SAHF` instructions which are supported by AMD64. These are load and store instructions, respectively, for certain status flags. In 64-bit mode, the `SAHF` instruction is used to optimize `fmod`, `drem`, and `remainder` built-in functions; see Section 6.59 [Other Builtins], page 641 for details.

`-mmovbe` This option enables use of the `movbe` instruction to implement `__builtin_bswap32` and `__builtin_bswap64`.

`-mshstk` The '`-mshstk`' option enables shadow stack built-in functions from x86 Control-flow Enforcement Technology (CET).

`-mcrc32` This option enables built-in functions `__builtin_ia32_crc32qi`, `__builtin_ia32_crc32hi`, `__builtin_ia32_crc32si` and `__builtin_ia32_crc32di` to generate the `crc32` machine instruction.

`-mrecip` This option enables use of `RCPSS` and `RSQRTSS` instructions (and their vectorized variants `RCPPS` and `RSQRTPS`) with an additional Newton-Raphson step to increase precision instead of `DIVSS` and `SQRTSS` (and their vectorized variants) for single-precision floating-point arguments. These instructions are generated only when '`-funsafe-math-optimizations`' is enabled together with '`-ffinite-math-only`' and '`-fno-trapping-math`'. Note that while the throughput of the sequence is higher than the throughput of the non-reciprocal instruction, the precision of the sequence can be decreased by up to 2 ulp (i.e. the inverse of 1.0 equals 0.99999994).

Note that GCC implements `1.0f/sqrtf(x)` in terms of `RSQRTSS` (or `RSQRTPS`) already with '`-ffast-math`' (or the above option combination), and doesn't need '`-mrecip`'.

Also note that GCC emits the above sequence with additional Newton-Raphson step for vectorized single-float division and vectorized `sqrtf(x)` already with '`-ffast-math`' (or the above option combination), and doesn't need '`-mrecip`'.

`-mrecip=`*opt*

This option controls which reciprocal estimate instructions may be used. *opt* is a comma-separated list of options, which may be preceded by a '`!`' to invert the option:

'`all`' Enable all estimate instructions.

'`default`' Enable the default instructions, equivalent to '`-mrecip`'.

'`none`' Disable all estimate instructions, equivalent to '`-mno-recip`'.

'`div`' Enable the approximation for scalar division.

'`vec-div`' Enable the approximation for vectorized division.

'`sqrt`' Enable the approximation for scalar square root.

'`vec-sqrt`'
Enable the approximation for vectorized square root.

So, for example, '`-mrecip=all,!sqrt`' enables all of the reciprocal approximations, except for square root.

`-mveclibabi=`*type*

Specifies the ABI type to use for vectorizing intrinsics using an external library. Supported values for *type* are '`svml`' for the Intel short vector math library and '`acml`' for the AMD math core library. To use this option, both

'-ftree-vectorize' and '-funsafe-math-optimizations' have to be enabled, and an SVML or ACML ABI-compatible library must be specified at link time.

GCC currently emits calls to vmldExp2, vmldLn2, vmldLog102, vmldPow2, vmldTanh2, vmldTan2, vmldAtan2, vmldAtanh2, vmldCbrt2, vmldSinh2, vmldSin2, vmldAsinh2, vmldAsin2, vmldCosh2, vmldCos2, vmldAcosh2, vmldAcos2, vmlsExp4, vmlsLn4, vmlsLog104, vmlsPow4, vmlsTanh4, vmlsTan4, vmlsAtan4, vmlsAtanh4, vmlsCbrt4, vmlsSinh4, vmlsSin4, vmlsAsinh4, vmlsAsin4, vmlsCosh4, vmlsCos4, vmlsAcosh4 and vmlsAcos4 for corresponding function type when '-mveclibabi=svml' is used, and __vrd2_sin, __vrd2_cos, __vrd2_exp, __vrd2_log, __vrd2_log2, __vrd2_log10, __vrs4_sinf, __vrs4_cosf, __vrs4_expf, __vrs4_logf, __vrs4_log2f, __vrs4_log10f and __vrs4_powf for the corresponding function type when '-mveclibabi=acml' is used.

-mabi=*name*

Generate code for the specified calling convention. Permissible values are 'sysv' for the ABI used on GNU/Linux and other systems, and 'ms' for the Microsoft ABI. The default is to use the Microsoft ABI when targeting Microsoft Windows and the SysV ABI on all other systems. You can control this behavior for specific functions by using the function attributes ms_abi and sysv_abi. See Section 6.33 [Function Attributes], page 477.

-mforce-indirect-call

Force all calls to functions to be indirect. This is useful when using Intel Processor Trace where it generates more precise timing information for function calls.

-mmanual-endbr

Insert ENDBR instruction at function entry only via the cf_check function attribute. This is useful when used with the option '-fcf-protection=branch' to control ENDBR insertion at the function entry.

-mcall-ms2sysv-xlogues

Due to differences in 64-bit ABIs, any Microsoft ABI function that calls a System V ABI function must consider RSI, RDI and XMM6-15 as clobbered. By default, the code for saving and restoring these registers is emitted inline, resulting in fairly lengthy prologues and epilogues. Using '-mcall-ms2sysv-xlogues' emits prologues and epilogues that use stubs in the static portion of libgcc to perform these saves and restores, thus reducing function size at the cost of a few extra instructions.

-mtls-dialect=*type*

Generate code to access thread-local storage using the 'gnu' or 'gnu2' conventions. 'gnu' is the conservative default; 'gnu2' is more efficient, but it may add compile- and run-time requirements that cannot be satisfied on all systems.

-mpush-args
-mno-push-args

Use PUSH operations to store outgoing parameters. This method is shorter and usually equally fast as method using SUB/MOV operations and is enabled

by default. In some cases disabling it may improve performance because of improved scheduling and reduced dependencies.

`-maccumulate-outgoing-args`
: If enabled, the maximum amount of space required for outgoing arguments is computed in the function prologue. This is faster on most modern CPUs because of reduced dependencies, improved scheduling and reduced stack usage when the preferred stack boundary is not equal to 2. The drawback is a notable increase in code size. This switch implies '`-mno-push-args`'.

`-mthreads`
: Support thread-safe exception handling on MinGW. Programs that rely on thread-safe exception handling must compile and link all code with the '`-mthreads`' option. When compiling, '`-mthreads`' defines '`-D_MT`'; when linking, it links in a special thread helper library '`-lmingwthrd`' which cleans up per-thread exception-handling data.

`-mms-bitfields`
`-mno-ms-bitfields`
: Enable/disable bit-field layout compatible with the native Microsoft Windows compiler.

If `packed` is used on a structure, or if bit-fields are used, it may be that the Microsoft ABI lays out the structure differently than the way GCC normally does. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

This option is enabled by default for Microsoft Windows targets. This behavior can also be controlled locally by use of variable or type attributes. For more information, see Section 6.34.16 [x86 Variable Attributes], page 548 and Section 6.35.7 [x86 Type Attributes], page 557.

The Microsoft structure layout algorithm is fairly simple with the exception of the bit-field packing. The padding and alignment of members of structures and whether a bit-field can straddle a storage-unit boundary are determine by these rules:

1. Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.
2. Every data object has an alignment requirement. The alignment requirement for all data except structures, unions, and arrays is either the size of the object or the current packing size (specified with either the `aligned` attribute or the `pack` pragma), whichever is less. For structures, unions, and arrays, the alignment requirement is the largest alignment requirement of its members. Every object is allocated an offset so that:

   ```
   offset % alignment_requirement == 0
   ```

3. Adjacent bit-fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit-field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit-fields.

MSVC interprets zero-length bit-fields in the following ways:

1. If a zero-length bit-field is inserted between two bit-fields that are normally coalesced, the bit-fields are not coalesced.

   For example:

   ```
   struct
    {
      unsigned long bf_1 : 12;
      unsigned long : 0;
      unsigned long bf_2 : 12;
    } t1;
   ```

   The size of `t1` is 8 bytes with the zero-length bit-field. If the zero-length bit-field were removed, `t1`'s size would be 4 bytes.

2. If a zero-length bit-field is inserted after a bit-field, `foo`, and the alignment of the zero-length bit-field is greater than the member that follows it, `bar`, `bar` is aligned as the type of the zero-length bit-field.

   For example:

   ```
   struct
    {
      char foo : 4;
      short : 0;
      char bar;
    } t2;

   struct
    {
      char foo : 4;
      short : 0;
      double bar;
    } t3;
   ```

   For `t2`, `bar` is placed at offset 2, rather than offset 1. Accordingly, the size of `t2` is 4. For `t3`, the zero-length bit-field does not affect the alignment of `bar` or, as a result, the size of the structure.

   Taking this into account, it is important to note the following:

   1. If a zero-length bit-field follows a normal bit-field, the type of the zero-length bit-field may affect the alignment of the structure as whole. For example, `t2` has a size of 4 bytes, since the zero-length bit-field follows a normal bit-field, and is of type short.
   2. Even if a zero-length bit-field is not followed by a normal bit-field, it may still affect the alignment of the structure:

      ```
      struct
       {
         char foo : 6;
         long : 0;
       } t4;
      ```

      Here, `t4` takes up 4 bytes.

3. Zero-length bit-fields following non-bit-field members are ignored:

   ```
   struct
    {
      char foo;
   ```

```
  long : 0;
  char bar;
} t5;
```

Here, `t5` takes up 2 bytes.

`-mno-align-stringops`
: Do not align the destination of inlined string operations. This switch reduces code size and improves performance in case the destination is already aligned, but GCC doesn't know about it.

`-minline-all-stringops`
: By default GCC inlines string operations only when the destination is known to be aligned to least a 4-byte boundary. This enables more inlining and increases code size, but may improve performance of code that depends on fast `memcpy`, `strlen`, and `memset` for short lengths.

`-minline-stringops-dynamically`
: For string operations of unknown size, use run-time checks with inline code for small blocks and a library call for large blocks.

`-mstringop-strategy=`*alg*
: Override the internal decision heuristic for the particular algorithm to use for inlining string operations. The allowed values for *alg* are:

  '`rep_byte`'
  '`rep_4byte`'
  '`rep_8byte`'
  : Expand using i386 `rep` prefix of the specified size.

  '`byte_loop`'
  '`loop`'
  '`unrolled_loop`'
  : Expand into an inline loop.

  '`libcall`' Always use a library call.

`-mmemcpy-strategy=`*strategy*
: Override the internal decision heuristic to decide if `__builtin_memcpy` should be inlined and what inline algorithm to use when the expected size of the copy operation is known. *strategy* is a comma-separated list of *alg*:*max_size*:*dest_align* triplets. *alg* is specified in '`-mstringop-strategy`', *max_size* specifies the max byte size with which inline algorithm *alg* is allowed. For the last triplet, the *max_size* must be `-1`. The *max_size* of the triplets in the list must be specified in increasing order. The minimal byte size for *alg* is `0` for the first triplet and *`max_size`* `+ 1` of the preceding range.

`-mmemset-strategy=`*strategy*
: The option is similar to '`-mmemcpy-strategy=`' except that it is to control `__builtin_memset` expansion.

`-momit-leaf-frame-pointer`
: Don't keep the frame pointer in a register for leaf functions. This avoids the instructions to save, set up, and restore frame pointers and makes an extra register

available in leaf functions. The option '`-fomit-leaf-frame-pointer`' removes the frame pointer for leaf functions, which might make debugging harder.

`-mtls-direct-seg-refs`
`-mno-tls-direct-seg-refs`
: Controls whether TLS variables may be accessed with offsets from the TLS segment register (`%gs` for 32-bit, `%fs` for 64-bit), or whether the thread base pointer must be added. Whether or not this is valid depends on the operating system, and whether it maps the segment to cover the entire TLS area.

  For systems that use the GNU C Library, the default is on.

`-msse2avx`
`-mno-sse2avx`
: Specify that the assembler should encode SSE instructions with VEX prefix. The option '`-mavx`' turns this on by default.

`-mfentry`
`-mno-fentry`
: If profiling is active ('`-pg`'), put the profiling counter call before the prologue. Note: On x86 architectures the attribute `ms_hook_prologue` isn't possible at the moment for '`-mfentry`' and '`-pg`'.

`-mrecord-mcount`
`-mno-record-mcount`
: If profiling is active ('`-pg`'), generate a __mcount_loc section that contains pointers to each profiling call. This is useful for automatically patching and out calls.

`-mnop-mcount`
`-mno-nop-mcount`
: If profiling is active ('`-pg`'), generate the calls to the profiling functions as NOPs. This is useful when they should be patched in later dynamically. This is likely only useful together with '`-mrecord-mcount`'.

`-minstrument-return=`*type*
: Instrument function exit in -pg -mfentry instrumented functions with call to specified function. This only instruments true returns ending with ret, but not sibling calls ending with jump. Valid types are *none* to not instrument, *call* to generate a call to __return__, or *nop5* to generate a 5 byte nop.

`-mrecord-return`
`-mno-record-return`
: Generate a __return_loc section pointing to all return instrumentation code.

`-mfentry-name=`*name*
: Set name of __fentry__ symbol called at function entry for -pg -mfentry functions.

`-mfentry-section=`*name*
: Set name of section to record -mrecord-mcount calls (default __mcount_loc).

`-mskip-rax-setup`
`-mno-skip-rax-setup`

When generating code for the x86-64 architecture with SSE extensions disabled, '`-mskip-rax-setup`' can be used to skip setting up RAX register when there are no variable arguments passed in vector registers.

Warning: Since RAX register is used to avoid unnecessarily saving vector registers on stack when passing variable arguments, the impacts of this option are callees may waste some stack space, misbehave or jump to a random location. GCC 4.4 or newer don't have those issues, regardless the RAX register value.

`-m8bit-idiv`
`-mno-8bit-idiv`

On some processors, like Intel Atom, 8-bit unsigned integer divide is much faster than 32-bit/64-bit integer divide. This option generates a run-time check. If both dividend and divisor are within range of 0 to 255, 8-bit unsigned integer divide is used instead of 32-bit/64-bit integer divide.

`-mavx256-split-unaligned-load`
`-mavx256-split-unaligned-store`

Split 32-byte AVX unaligned load and store.

`-mstack-protector-guard=`*`guard`*
`-mstack-protector-guard-reg=`*`reg`*
`-mstack-protector-guard-offset=`*`offset`*

Generate stack protection code using canary at *guard*. Supported locations are '`global`' for global canary or '`tls`' for per-thread canary in the TLS block (the default). This option has effect only when '`-fstack-protector`' or '`-fstack-protector-all`' is specified.

With the latter choice the options '`-mstack-protector-guard-reg=`*`reg`*' and '`-mstack-protector-guard-offset=`*`offset`*' furthermore specify which segment register (`%fs` or `%gs`) to use as base register for reading the canary, and from what offset from that base register. The default for those is as specified in the relevant ABI.

`-mgeneral-regs-only`

Generate code that uses only the general-purpose registers. This prevents the compiler from using floating-point, vector, mask and bound registers.

`-mindirect-branch=`*`choice`*

Convert indirect call and jump with *choice*. The default is '`keep`', which keeps indirect call and jump unmodified. '`thunk`' converts indirect call and jump to call and return thunk. '`thunk-inline`' converts indirect call and jump to inlined call and return thunk. '`thunk-extern`' converts indirect call and jump to external call and return thunk provided in a separate object file. You can control this behavior for a specific function by using the function attribute `indirect_branch`. See Section 6.33 [Function Attributes], page 477.

Note that '`-mcmodel=large`' is incompatible with '`-mindirect-branch=thunk`' and '`-mindirect-branch=thunk-extern`' since the thunk function may not be reachable in the large code model.

Note that '-mindirect-branch=thunk-extern' is incompatible with '-fcf-protection=branch' since the external thunk cannot be modified to disable control-flow check.

-mfunction-return=*choice*

Convert function return with *choice*. The default is 'keep', which keeps function return unmodified. 'thunk' converts function return to call and return thunk. 'thunk-inline' converts function return to inlined call and return thunk. 'thunk-extern' converts function return to external call and return thunk provided in a separate object file. You can control this behavior for a specific function by using the function attribute function_return. See Section 6.33 [Function Attributes], page 477.

Note that '-mcmodel=large' is incompatible with '-mfunction-return=thunk' and '-mfunction-return=thunk-extern' since the thunk function may not be reachable in the large code model.

-mindirect-branch-register

Force indirect call and jump via register.

These '-m' switches are supported in addition to the above on x86-64 processors in 64-bit environments.

-m32
-m64
-mx32
-m16
-miamcu

Generate code for a 16-bit, 32-bit or 64-bit environment. The '-m32' option sets int, long, and pointer types to 32 bits, and generates code that runs on any i386 system.

The '-m64' option sets int to 32 bits and long and pointer types to 64 bits, and generates code for the x86-64 architecture. For Darwin only the '-m64' option also turns off the '-fno-pic' and '-mdynamic-no-pic' options.

The '-mx32' option sets int, long, and pointer types to 32 bits, and generates code for the x86-64 architecture.

The '-m16' option is the same as '-m32', except for that it outputs the .code16gcc assembly directive at the beginning of the assembly output so that the binary can run in 16-bit mode.

The '-miamcu' option generates code which conforms to Intel MCU psABI. It requires the '-m32' option to be turned on.

-mno-red-zone

Do not use a so-called "red zone" for x86-64 code. The red zone is mandated by the x86-64 ABI; it is a 128-byte area beyond the location of the stack pointer that is not modified by signal or interrupt handlers and therefore can be used for temporary data without adjusting the stack pointer. The flag '-mno-red-zone' disables this red zone.

`-mcmodel=small`
Generate code for the small code model: the program and its symbols must be linked in the lower 2 GB of the address space. Pointers are 64 bits. Programs can be statically or dynamically linked. This is the default code model.

`-mcmodel=kernel`
Generate code for the kernel code model. The kernel runs in the negative 2 GB of the address space. This model has to be used for Linux kernel code.

`-mcmodel=medium`
Generate code for the medium model: the program is linked in the lower 2 GB of the address space. Small symbols are also placed there. Symbols with sizes larger than '`-mlarge-data-threshold`' are put into large data or BSS sections and can be located above 2GB. Programs can be statically or dynamically linked.

`-mcmodel=large`
Generate code for the large model. This model makes no assumptions about addresses and sizes of sections.

`-maddress-mode=long`
Generate code for long address mode. This is only supported for 64-bit and x32 environments. It is the default address mode for 64-bit environments.

`-maddress-mode=short`
Generate code for short address mode. This is only supported for 32-bit and x32 environments. It is the default address mode for 32-bit and x32 environments.

3.18.59 x86 Windows Options

These additional options are available for Microsoft Windows targets:

`-mconsole`
This option specifies that a console application is to be generated, by instructing the linker to set the PE header subsystem type required for console applications. This option is available for Cygwin and MinGW targets and is enabled by default on those targets.

`-mdll` This option is available for Cygwin and MinGW targets. It specifies that a DLL—a dynamic link library—is to be generated, enabling the selection of the required runtime startup object and entry point.

`-mnop-fun-dllimport`
This option is available for Cygwin and MinGW targets. It specifies that the `dllimport` attribute should be ignored.

`-mthread` This option is available for MinGW targets. It specifies that MinGW-specific thread support is to be used.

`-municode`
This option is available for MinGW-w64 targets. It causes the `UNICODE` preprocessor macro to be predefined, and chooses Unicode-capable runtime startup code.

`-mwin32` This option is available for Cygwin and MinGW targets. It specifies that the typical Microsoft Windows predefined macros are to be set in the pre-processor, but does not influence the choice of runtime library/startup code.

`-mwindows`
This option is available for Cygwin and MinGW targets. It specifies that a GUI application is to be generated by instructing the linker to set the PE header subsystem type appropriately.

`-fno-set-stack-executable`
This option is available for MinGW targets. It specifies that the executable flag for the stack used by nested functions isn't set. This is necessary for binaries running in kernel mode of Microsoft Windows, as there the User32 API, which is used to set executable privileges, isn't available.

`-fwritable-relocated-rdata`
This option is available for MinGW and Cygwin targets. It specifies that relocated-data in read-only section is put into the `.data` section. This is a necessary for older runtimes not supporting modification of `.rdata` sections for pseudo-relocation.

`-mpe-aligned-commons`
This option is available for Cygwin and MinGW targets. It specifies that the GNU extension to the PE file format that permits the correct alignment of COMMON variables should be used when generating code. It is enabled by default if GCC detects that the target assembler found during configuration supports the feature.

See also under Section 3.18.58 [x86 Options], page 401 for standard options.

3.18.60 Xstormy16 Options

These options are defined for Xstormy16:

`-msim` Choose startup files and linker script suitable for the simulator.

3.18.61 Xtensa Options

These options are supported for Xtensa targets:

`-mconst16`
`-mno-const16`
Enable or disable use of `CONST16` instructions for loading constant values. The `CONST16` instruction is currently not a standard option from Tensilica. When enabled, `CONST16` instructions are always used in place of the standard `L32R` instructions. The use of `CONST16` is enabled by default only if the `L32R` instruction is not available.

`-mfused-madd`
`-mno-fused-madd`
Enable or disable use of fused multiply/add and multiply/subtract instructions in the floating-point option. This has no effect if the floating-point option

is not also enabled. Disabling fused multiply/add and multiply/subtract instructions forces the compiler to use separate instructions for the multiply and add/subtract operations. This may be desirable in some cases where strict IEEE 754-compliant results are required: the fused multiply add/subtract instructions do not round the intermediate result, thereby producing results with *more* bits of precision than specified by the IEEE standard. Disabling fused multiply add/subtract instructions also ensures that the program output is not sensitive to the compiler's ability to combine multiply and add/subtract operations.

`-mserialize-volatile`
`-mno-serialize-volatile`
: When this option is enabled, GCC inserts `MEMW` instructions before `volatile` memory references to guarantee sequential consistency. The default is '`-mserialize-volatile`'. Use '`-mno-serialize-volatile`' to omit the `MEMW` instructions.

`-mforce-no-pic`
: For targets, like GNU/Linux, where all user-mode Xtensa code must be position-independent code (PIC), this option disables PIC for compiling kernel code.

`-mtext-section-literals`
`-mno-text-section-literals`
: These options control the treatment of literal pools. The default is '`-mno-text-section-literals`', which places literals in a separate section in the output file. This allows the literal pool to be placed in a data RAM/ROM, and it also allows the linker to combine literal pools from separate object files to remove redundant literals and improve code size. With '`-mtext-section-literals`', the literals are interspersed in the text section in order to keep them as close as possible to their references. This may be necessary for large assembly files. Literals for each function are placed right before that function.

`-mauto-litpools`
`-mno-auto-litpools`
: These options control the treatment of literal pools. The default is '`-mno-auto-litpools`', which places literals in a separate section in the output file unless '`-mtext-section-literals`' is used. With '`-mauto-litpools`' the literals are interspersed in the text section by the assembler. Compiler does not produce explicit `.literal` directives and loads literals into registers with `MOVI` instructions instead of `L32R` to let the assembler do relaxation and place literals as necessary. This option allows assembler to create several literal pools per function and assemble very big functions, which may not be possible with '`-mtext-section-literals`'.

`-mtarget-align`
`-mno-target-align`
: When this option is enabled, GCC instructs the assembler to automatically align instructions to reduce branch penalties at the expense of some code density. The

assembler attempts to widen density instructions to align branch targets and the instructions following call instructions. If there are not enough preceding safe density instructions to align a target, no widening is performed. The default is '`-mtarget-align`'. These options do not affect the treatment of auto-aligned instructions like `LOOP`, which the assembler always aligns, either by widening density instructions or by inserting NOP instructions.

`-mlongcalls`
`-mno-longcalls`
When this option is enabled, GCC instructs the assembler to translate direct calls to indirect calls unless it can determine that the target of a direct call is in the range allowed by the call instruction. This translation typically occurs for calls to functions in other source files. Specifically, the assembler translates a direct `CALL` instruction into an `L32R` followed by a `CALLX` instruction. The default is '`-mno-longcalls`'. This option should be used in programs where the call target can potentially be out of range. This option is implemented in the assembler, not the compiler, so the assembly code generated by GCC still shows direct call instructions—look at the disassembled object code to see the actual instructions. Note that the assembler uses an indirect call for every cross-file call, not just those that really are out of range.

3.18.62 zSeries Options

These are listed under See Section 3.18.44 [S/390 and zSeries Options], page 376.

3.19 Specifying Subprocesses and the Switches to Pass to Them

`gcc` is a driver program. It performs its job by invoking a sequence of other programs to do the work of compiling, assembling and linking. GCC interprets its command-line parameters and uses these to deduce which programs it should invoke, and which command-line options it ought to place on their command lines. This behavior is controlled by *spec strings*. In most cases there is one spec string for each program that GCC can invoke, but a few programs have multiple spec strings to control their behavior. The spec strings built into GCC can be overridden by using the '`-specs=`' command-line switch to specify a spec file.

Spec files are plain-text files that are used to construct spec strings. They consist of a sequence of directives separated by blank lines. The type of directive is determined by the first non-whitespace character on the line, which can be one of the following:

`%command` Issues a *command* to the spec file processor. The commands that can appear here are:

`%include <file>`
Search for *file* and insert its text at the current point in the specs file.

`%include_noerr <file>`
Just like '`%include`', but do not generate an error message if the include file cannot be found.

`%rename` *old_name new_name*
: Rename the spec string *old_name* to *new_name*.

`*[`*spec_name*`]:`
: This tells the compiler to create, override or delete the named spec string. All lines after this directive up to the next directive or blank line are considered to be the text for the spec string. If this results in an empty string then the spec is deleted. (Or, if the spec did not exist, then nothing happens.) Otherwise, if the spec does not currently exist a new spec is created. If the spec does exist then its contents are overridden by the text of this directive, unless the first character of that text is the '`+`' character, in which case the text is appended to the spec.

`[`*suffix*`]:`
: Creates a new '`[`*suffix*`] spec`' pair. All lines after this directive and up to the next directive or blank line are considered to make up the spec string for the indicated suffix. When the compiler encounters an input file with the named suffix, it processes the spec string in order to work out how to compile that file. For example:

```
.ZZ:
z-compile -input %i
```

This says that any input file whose name ends in '`.ZZ`' should be passed to the program '`z-compile`', which should be invoked with the command-line switch '`-input`' and with the result of performing the '`%i`' substitution. (See below.)

As an alternative to providing a spec string, the text following a suffix directive can be one of the following:

`@`*language*
: This says that the suffix is an alias for a known *language*. This is similar to using the '`-x`' command-line switch to GCC to specify a language explicitly. For example:

```
.ZZ:
@c++
```

Says that .ZZ files are, in fact, C++ source files.

`#`*name*
: This causes an error messages saying:

```
name compiler not installed on this system.
```

GCC already has an extensive list of suffixes built into it. This directive adds an entry to the end of the list of suffixes, but since the list is searched from the end backwards, it is effectively possible to override earlier entries using this technique.

GCC has the following spec strings built into it. Spec files can override these strings or create their own. Note that individual targets can also add their own spec strings to this list.

```
asm          Options to pass to the assembler
asm_final    Options to pass to the assembler post-processor
cpp          Options to pass to the C preprocessor
cc1          Options to pass to the C compiler
cc1plus      Options to pass to the C++ compiler
```

```
endfile      Object files to include at the end of the link
link         Options to pass to the linker
lib          Libraries to include on the command line to the linker
libgcc       Decides which GCC support library to pass to the linker
linker       Sets the name of the linker
predefines   Defines to be passed to the C preprocessor
signed_char  Defines to pass to CPP to say whether char is signed
             by default
startfile    Object files to include at the start of the link
```

Here is a small example of a spec file:

```
%rename lib                 old_lib

*lib:
--start-group -lgcc -lc -leval1 --end-group %(old_lib)
```

This example renames the spec called '`lib`' to '`old_lib`' and then overrides the previous definition of '`lib`' with a new one. The new definition adds in some extra command-line options before including the text of the old definition.

Spec strings are a list of command-line options to be passed to their corresponding program. In addition, the spec strings can contain '`%`'-prefixed sequences to substitute variable text or to conditionally insert text into the command line. Using these constructs it is possible to generate quite complex command lines.

Here is a table of all defined '`%`'-sequences for spec strings. Note that spaces are not generated automatically around the results of expanding these sequences. Therefore you can concatenate them together or combine them with constant text in a single argument.

`%%` Substitute one '`%`' into the program name or argument.

`%i` Substitute the name of the input file being processed.

`%b` Substitute the basename of the input file being processed. This is the substring up to (and not including) the last period and not including the directory.

`%B` This is the same as '`%b`', but include the file suffix (text after the last period).

`%d` Marks the argument containing or following the '`%d`' as a temporary file name, so that that file is deleted if GCC exits successfully. Unlike '`%g`', this contributes no text to the argument.

`%g`*`suffix`* Substitute a file name that has suffix *suffix* and is chosen once per compilation, and mark the argument in the same way as '`%d`'. To reduce exposure to denial-of-service attacks, the file name is now chosen in a way that is hard to predict even when previously chosen file names are known. For example, '`%g.s ... %g.o ... %g.s`' might turn into '`ccUVUUAU.s ccXYAXZ12.o ccUVUUAU.s`'. *suffix* matches the regexp '`[.A-Za-z]*`' or the special string '`%O`', which is treated exactly as if '`%O`' had been preprocessed. Previously, '`%g`' was simply substituted with a file name chosen once per compilation, without regard to any appended suffix (which was therefore treated just like ordinary text), making such attacks more likely to succeed.

`%u`*`suffix`* Like '`%g`', but generates a new temporary file name each time it appears instead of once per compilation.

`%U`*suffix* Substitutes the last file name generated with '`%u`*suffix*', generating a new one if there is no such last file name. In the absence of any '`%u`*suffix*', this is just like '`%g`*suffix*', except they don't share the same suffix *space*, so '`%g.s ... %U.s ... %g.s ... %U.s`' involves the generation of two distinct file names, one for each '`%g.s`' and another for each '`%U.s`'. Previously, '`%U`' was simply substituted with a file name chosen for the previous '`%u`', without regard to any appended suffix.

`%j`*suffix* Substitutes the name of the `HOST_BIT_BUCKET`, if any, and if it is writable, and if '`-save-temps`' is not used; otherwise, substitute the name of a temporary file, just like '`%u`'. This temporary file is not meant for communication between processes, but rather as a junk disposal mechanism.

`%|`*suffix*
`%m`*suffix* Like '`%g`', except if '`-pipe`' is in effect. In that case '`%|`' substitutes a single dash and '`%m`' substitutes nothing at all. These are the two most common ways to instruct a program that it should read from standard input or write to standard output. If you need something more elaborate you can use an '`%{pipe:X}`' construct: see for example '`f/lang-specs.h`'.

`%.`*SUFFIX* Substitutes *.SUFFIX* for the suffixes of a matched switch's args when it is subsequently output with '`%*`'. *SUFFIX* is terminated by the next space or %.

`%w` Marks the argument containing or following the '`%w`' as the designated output file of this compilation. This puts the argument into the sequence of arguments that '`%o`' substitutes.

`%o` Substitutes the names of all the output files, with spaces automatically placed around them. You should write spaces around the '`%o`' as well or the results are undefined. '`%o`' is for use in the specs for running the linker. Input files whose names have no recognized suffix are not compiled at all, but they are included among the output files, so they are linked.

`%O` Substitutes the suffix for object files. Note that this is handled specially when it immediately follows '`%g, %u, or %U`', because of the need for those to form complete file names. The handling is such that '`%O`' is treated exactly as if it had already been substituted, except that '`%g, %u, and %U`' do not currently support additional *suffix* characters following '`%O`' as they do following, for example, '`.o`'.

`%p` Substitutes the standard macro predefinitions for the current target machine. Use this when running `cpp`.

`%P` Like '`%p`', but puts '`__`' before and after the name of each predefined macro, except for macros that start with '`__`' or with '`_`*L*', where *L* is an uppercase letter. This is for ISO C.

`%I` Substitute any of '`-iprefix`' (made from `GCC_EXEC_PREFIX`), '`-isysroot`' (made from `TARGET_SYSTEM_ROOT`), '`-isystem`' (made from `COMPILER_PATH` and '`-B`' options) and '`-imultilib`' as necessary.

`%s` Current argument is the name of a library or startup file of some sort. Search for that file in a standard list of directories and substitute the full name found. The current working directory is included in the list of directories scanned.

`%T` Current argument is the name of a linker script. Search for that file in the current list of directories to scan for libraries. If the file is located insert a '`--script`' option into the command line followed by the full path name found. If the file is not found then generate an error message. Note: the current working directory is not searched.

`%e`*str* Print *str* as an error message. *str* is terminated by a newline. Use this when inconsistent options are detected.

`%(`*name*`)` Substitute the contents of spec string *name* at this point.

`%x{`*option*`}`
Accumulate an option for '`%X`'.

`%X` Output the accumulated linker options specified by '`-Wl`' or a '`%x`' spec string.

`%Y` Output the accumulated assembler options specified by '`-Wa`'.

`%Z` Output the accumulated preprocessor options specified by '`-Wp`'.

`%a` Process the `asm` spec. This is used to compute the switches to be passed to the assembler.

`%A` Process the `asm_final` spec. This is a spec string for passing switches to an assembler post-processor, if such a program is needed.

`%l` Process the `link` spec. This is the spec for computing the command line passed to the linker. Typically it makes use of the '`%L %G %S %D and %E`' sequences.

`%D` Dump out a '`-L`' option for each directory that GCC believes might contain startup files. If the target supports multilibs then the current multilib directory is prepended to each of these paths.

`%L` Process the `lib` spec. This is a spec string for deciding which libraries are included on the command line to the linker.

`%G` Process the `libgcc` spec. This is a spec string for deciding which GCC support library is included on the command line to the linker.

`%S` Process the `startfile` spec. This is a spec for deciding which object files are the first ones passed to the linker. Typically this might be a file named '`crt0.o`'.

`%E` Process the `endfile` spec. This is a spec string that specifies the last object files that are passed to the linker.

`%C` Process the `cpp` spec. This is used to construct the arguments to be passed to the C preprocessor.

`%1` Process the `cc1` spec. This is used to construct the options to be passed to the actual C compiler (`cc1`).

`%2` Process the `cc1plus` spec. This is used to construct the options to be passed to the actual C++ compiler (`cc1plus`).

`%*` Substitute the variable part of a matched option. See below. Note that each comma in the substituted string is replaced by a single space.

`%<S` Remove all occurrences of `-S` from the command line. Note—this command is position dependent. '`%`' commands in the spec string before this one see `-S`, '`%`' commands in the spec string after this one do not.

`%:function(args)`
Call the named function *function*, passing it *args*. *args* is first processed as a nested spec string, then split into an argument vector in the usual fashion. The function returns a string which is processed as if it had appeared literally as part of the current spec.

The following built-in spec functions are provided:

`getenv` The `getenv` spec function takes two arguments: an environment variable name and a string. If the environment variable is not defined, a fatal error is issued. Otherwise, the return value is the value of the environment variable concatenated with the string. For example, if `TOPDIR` is defined as '`/path/to/top`', then:

```
%:getenv(TOPDIR /include)
```

expands to '`/path/to/top/include`'.

`if-exists`
The `if-exists` spec function takes one argument, an absolute pathname to a file. If the file exists, `if-exists` returns the pathname. Here is a small example of its usage:

```
*startfile:
crt0%O%s %:if-exists(crti%O%s) crtbegin%O%s
```

`if-exists-else`
The `if-exists-else` spec function is similar to the `if-exists` spec function, except that it takes two arguments. The first argument is an absolute pathname to a file. If the file exists, `if-exists-else` returns the pathname. If it does not exist, it returns the second argument. This way, `if-exists-else` can be used to select one file or another, based on the existence of the first. Here is a small example of its usage:

```
*startfile:
crt0%O%s %:if-exists(crti%O%s) \
%:if-exists-else(crtbeginT%O%s crtbegin%O%s)
```

`replace-outfile`
The `replace-outfile` spec function takes two arguments. It looks for the first argument in the outfiles array and replaces it with the second argument. Here is a small example of its usage:

```
%{fgnu-runtime:%:replace-outfile(-lobjc -lobjc-gnu)}
```

`remove-outfile`
The `remove-outfile` spec function takes one argument. It looks for the first argument in the outfiles array and removes it. Here is a small example its usage:

```
%:remove-outfile(-lm)
```

`pass-through-libs`
: The `pass-through-libs` spec function takes any number of arguments. It finds any '`-l`' options and any non-options ending in '`.a`' (which it assumes are the names of linker input library archive files) and returns a result containing all the found arguments each prepended by '`-plugin-opt=-pass-through=`' and joined by spaces. This list is intended to be passed to the LTO linker plugin.

```
%:pass-through-libs(%G %L %G)
```

`print-asm-header`
: The `print-asm-header` function takes no arguments and simply prints a banner like:

```
Assembler options
=================

Use "-Wa,OPTION" to pass "OPTION" to the assembler.
```

It is used to separate compiler options from assembler options in the '`--target-help`' output.

`%{S}` Substitutes the `-S` switch, if that switch is given to GCC. If that switch is not specified, this substitutes nothing. Note that the leading dash is omitted when specifying this option, and it is automatically inserted if the substitution is performed. Thus the spec string '`%{foo}`' matches the command-line option '`-foo`' and outputs the command-line option '`-foo`'.

`%W{S}` Like %{S} but mark last argument supplied within as a file to be deleted on failure.

`%{S*}` Substitutes all the switches specified to GCC whose names start with `-S`, but which also take an argument. This is used for switches like '`-o`', '`-D`', '`-I`', etc. GCC considers '`-o foo`' as being one switch whose name starts with '`o`'. %{o*} substitutes this text, including the space. Thus two arguments are generated.

`%{S*&T*}` Like %{S*}, but preserve order of `S` and `T` options (the order of `S` and `T` in the spec is not significant). There can be any number of ampersand-separated variables; for each the wild card is optional. Useful for CPP as '`%{D*&U*&A*}`'.

`%{S:X}` Substitutes `X`, if the '`-S`' switch is given to GCC.

`%{!S:X}` Substitutes `X`, if the '`-S`' switch is *not* given to GCC.

`%{S*:X}` Substitutes `X` if one or more switches whose names start with `-S` are specified to GCC. Normally `X` is substituted only once, no matter how many such switches appeared. However, if `%*` appears somewhere in `X`, then `X` is substituted once for each matching switch, with the `%*` replaced by the part of that switch matching the `*`.

If `%*` appears as the last part of a spec sequence then a space is added after the end of the last substitution. If there is more text in the sequence, however, then a space is not generated. This allows the `%*` substitution to be used as part of a larger string. For example, a spec string like this:

```
%{mcu=*:--script=%*/memory.ld}
```

when matching an option like '`-mcu=newchip`' produces:

```
--script=newchip/memory.ld
```

`%{.S:X}` Substitutes `X`, if processing a file with suffix `S`.

`%{!.S:X}` Substitutes `X`, if *not* processing a file with suffix `S`.

`%{,S:X}` Substitutes `X`, if processing a file for language `S`.

`%{!,S:X}` Substitutes `X`, if not processing a file for language `S`.

`%{S|P:X}` Substitutes `X` if either `-S` or `-P` is given to GCC. This may be combined with '`!`', '`.`', '`,`', and `*` sequences as well, although they have a stronger binding than the '`|`'. If `%*` appears in `X`, all of the alternatives must be starred, and only the first matching alternative is substituted.

For example, a spec string like this:

```
%{.c:-foo} %{!.c:-bar} %{.c|d:-baz} %{!.c|d:-boggle}
```

outputs the following command-line options from the following input command-line options:

```
fred.c        -foo -baz
jim.d         -bar -boggle
-d fred.c     -foo -baz -boggle
-d jim.d      -bar -baz -boggle
```

`%{S:X; T:Y; :D}`

If `S` is given to GCC, substitutes `X`; else if `T` is given to GCC, substitutes `Y`; else substitutes `D`. There can be as many clauses as you need. This may be combined with `.`, `,`, `!`, `|`, and `*` as needed.

The switch matching text `S` in a '`%{S}`', '`%{S:X}`' or similar construct can use a backslash to ignore the special meaning of the character following it, thus allowing literal matching of a character that is otherwise specially treated. For example, '`%{std=iso9899\:1999:X}`' substitutes `X` if the '`-std=iso9899:1999`' option is given.

The conditional text `X` in a '`%{S:X}`' or similar construct may contain other nested '`%`' constructs or spaces, or even newlines. They are processed as usual, as described above. Trailing white space in `X` is ignored. White space may also appear anywhere on the left side of the colon in these constructs, except between `.` or `*` and the corresponding word.

The '`-O`', '`-f`', '`-m`', and '`-W`' switches are handled specifically in these constructs. If another value of '`-O`' or the negated form of a '`-f`', '`-m`', or '`-W`' switch is found later in the command line, the earlier switch value is ignored, except with {`S*`} where `S` is just one letter, which passes all matching options.

The character '`|`' at the beginning of the predicate text is used to indicate that a command should be piped to the following command, but only if '`-pipe`' is specified.

It is built into GCC which switches take arguments and which do not. (You might think it would be useful to generalize this to allow each compiler's spec to say which switches take arguments. But this cannot be done in a consistent fashion. GCC cannot even decide which input files have been specified without knowing which switches take arguments, and it must know which input files to compile in order to tell which compilers to run).

GCC also knows implicitly that arguments starting in '`-l`' are to be treated as compiler output files, and passed to the linker in their proper position among the other output files.

3.20 Environment Variables Affecting GCC

This section describes several environment variables that affect how GCC operates. Some of them work by specifying directories or prefixes to use when searching for various kinds of files. Some are used to specify other aspects of the compilation environment.

Note that you can also specify places to search using options such as '-B', '-I' and '-L' (see Section 3.15 [Directory Options], page 215). These take precedence over places specified using environment variables, which in turn take precedence over those specified by the configuration of GCC. See Section "Controlling the Compilation Driver 'gcc'" in *GNU Compiler Collection (GCC) Internals*.

LANG
LC_CTYPE
LC_MESSAGES
LC_ALL

These environment variables control the way that GCC uses localization information which allows GCC to work with different national conventions. GCC inspects the locale categories LC_CTYPE and LC_MESSAGES if it has been configured to do so. These locale categories can be set to any value supported by your installation. A typical value is 'en_GB.UTF-8' for English in the United Kingdom encoded in UTF-8.

The LC_CTYPE environment variable specifies character classification. GCC uses it to determine the character boundaries in a string; this is needed for some multibyte encodings that contain quote and escape characters that are otherwise interpreted as a string end or escape.

The LC_MESSAGES environment variable specifies the language to use in diagnostic messages.

If the LC_ALL environment variable is set, it overrides the value of LC_CTYPE and LC_MESSAGES; otherwise, LC_CTYPE and LC_MESSAGES default to the value of the LANG environment variable. If none of these variables are set, GCC defaults to traditional C English behavior.

TMPDIR

If TMPDIR is set, it specifies the directory to use for temporary files. GCC uses temporary files to hold the output of one stage of compilation which is to be used as input to the next stage: for example, the output of the preprocessor, which is the input to the compiler proper.

GCC_COMPARE_DEBUG

Setting GCC_COMPARE_DEBUG is nearly equivalent to passing '-fcompare-debug' to the compiler driver. See the documentation of this option for more details.

GCC_EXEC_PREFIX

If GCC_EXEC_PREFIX is set, it specifies a prefix to use in the names of the subprograms executed by the compiler. No slash is added when this prefix is combined with the name of a subprogram, but you can specify a prefix that ends with a slash if you wish.

If GCC_EXEC_PREFIX is not set, GCC attempts to figure out an appropriate prefix to use based on the pathname it is invoked with.

If GCC cannot find the subprogram using the specified prefix, it tries looking in the usual places for the subprogram.

The default value of GCC_EXEC_PREFIX is '*prefix*/lib/gcc/' where *prefix* is the prefix to the installed compiler. In many cases *prefix* is the value of prefix when you ran the 'configure' script.

Other prefixes specified with '-B' take precedence over this prefix.

This prefix is also used for finding files such as 'crt0.o' that are used for linking.

In addition, the prefix is used in an unusual way in finding the directories to search for header files. For each of the standard directories whose name normally begins with '/usr/local/lib/gcc' (more precisely, with the value of GCC_INCLUDE_DIR), GCC tries replacing that beginning with the specified prefix to produce an alternate directory name. Thus, with '-Bfoo/', GCC searches 'foo/bar' just before it searches the standard directory '/usr/local/lib/bar'. If a standard directory begins with the configured *prefix* then the value of *prefix* is replaced by GCC_EXEC_PREFIX when looking for header files.

COMPILER_PATH
: The value of COMPILER_PATH is a colon-separated list of directories, much like PATH. GCC tries the directories thus specified when searching for subprograms, if it cannot find the subprograms using GCC_EXEC_PREFIX.

LIBRARY_PATH
: The value of LIBRARY_PATH is a colon-separated list of directories, much like PATH. When configured as a native compiler, GCC tries the directories thus specified when searching for special linker files, if it cannot find them using GCC_EXEC_PREFIX. Linking using GCC also uses these directories when searching for ordinary libraries for the '-l' option (but directories specified with '-L' come first).

LANG
: This variable is used to pass locale information to the compiler. One way in which this information is used is to determine the character set to be used when character literals, string literals and comments are parsed in C and C++. When the compiler is configured to allow multibyte characters, the following values for LANG are recognized:

  'C-JIS' Recognize JIS characters.

  'C-SJIS' Recognize SJIS characters.

  'C-EUCJP' Recognize EUCJP characters.

  If LANG is not defined, or if it has some other value, then the compiler uses mblen and mbtowc as defined by the default locale to recognize and translate multibyte characters.

Some additional environment variables affect the behavior of the preprocessor.

CPATH
C_INCLUDE_PATH
CPLUS_INCLUDE_PATH
OBJC_INCLUDE_PATH
: Each variable's value is a list of directories separated by a special character, much like PATH, in which to look for header files. The special character, PATH_

SEPARATOR, is target-dependent and determined at GCC build time. For Microsoft Windows-based targets it is a semicolon, and for almost all other targets it is a colon.

CPATH specifies a list of directories to be searched as if specified with '-I', but after any paths given with '-I' options on the command line. This environment variable is used regardless of which language is being preprocessed.

The remaining environment variables apply only when preprocessing the particular language indicated. Each specifies a list of directories to be searched as if specified with '-isystem', but after any paths given with '-isystem' options on the command line.

In all these variables, an empty element instructs the compiler to search its current working directory. Empty elements can appear at the beginning or end of a path. For instance, if the value of CPATH is :/special/include, that has the same effect as '-I. -I/special/include'.

DEPENDENCIES_OUTPUT

If this variable is set, its value specifies how to output dependencies for Make based on the non-system header files processed by the compiler. System header files are ignored in the dependency output.

The value of DEPENDENCIES_OUTPUT can be just a file name, in which case the Make rules are written to that file, guessing the target name from the source file name. Or the value can have the form '*file target*', in which case the rules are written to file *file* using *target* as the target name.

In other words, this environment variable is equivalent to combining the options '-MM' and '-MF' (see Section 3.12 [Preprocessor Options], page 202), with an optional '-MT' switch too.

SUNPRO_DEPENDENCIES

This variable is the same as DEPENDENCIES_OUTPUT (see above), except that system header files are not ignored, so it implies '-M' rather than '-MM'. However, the dependence on the main input file is omitted. See Section 3.12 [Preprocessor Options], page 202.

SOURCE_DATE_EPOCH

If this variable is set, its value specifies a UNIX timestamp to be used in replacement of the current date and time in the __DATE__ and __TIME__ macros, so that the embedded timestamps become reproducible.

The value of SOURCE_DATE_EPOCH must be a UNIX timestamp, defined as the number of seconds (excluding leap seconds) since 01 Jan 1970 00:00:00 represented in ASCII; identical to the output of 'date +%s' on GNU/Linux and other systems that support the %s extension in the date command.

The value should be a known timestamp such as the last modification time of the source or package and it should be set by the build process.

3.21 Using Precompiled Headers

Often large projects have many header files that are included in every source file. The time the compiler takes to process these header files over and over again can account for nearly

all of the time required to build the project. To make builds faster, GCC allows you to *precompile* a header file.

To create a precompiled header file, simply compile it as you would any other file, if necessary using the '`-x`' option to make the driver treat it as a C or C++ header file. You may want to use a tool like `make` to keep the precompiled header up-to-date when the headers it contains change.

A precompiled header file is searched for when `#include` is seen in the compilation. As it searches for the included file (see Section "Search Path" in *The C Preprocessor*) the compiler looks for a precompiled header in each directory just before it looks for the include file in that directory. The name searched for is the name specified in the `#include` with '`.gch`' appended. If the precompiled header file cannot be used, it is ignored.

For instance, if you have `#include "all.h"`, and you have '`all.h.gch`' in the same directory as '`all.h`', then the precompiled header file is used if possible, and the original header is used otherwise.

Alternatively, you might decide to put the precompiled header file in a directory and use '`-I`' to ensure that directory is searched before (or instead of) the directory containing the original header. Then, if you want to check that the precompiled header file is always used, you can put a file of the same name as the original header in this directory containing an `#error` command.

This also works with '`-include`'. So yet another way to use precompiled headers, good for projects not designed with precompiled header files in mind, is to simply take most of the header files used by a project, include them from another header file, precompile that header file, and '`-include`' the precompiled header. If the header files have guards against multiple inclusion, they are skipped because they've already been included (in the precompiled header).

If you need to precompile the same header file for different languages, targets, or compiler options, you can instead make a *directory* named like '`all.h.gch`', and put each precompiled header in the directory, perhaps using '`-o`'. It doesn't matter what you call the files in the directory; every precompiled header in the directory is considered. The first precompiled header encountered in the directory that is valid for this compilation is used; they're searched in no particular order.

There are many other possibilities, limited only by your imagination, good sense, and the constraints of your build system.

A precompiled header file can be used only when these conditions apply:

- Only one precompiled header can be used in a particular compilation.
- A precompiled header cannot be used once the first C token is seen. You can have preprocessor directives before a precompiled header; you cannot include a precompiled header from inside another header.
- The precompiled header file must be produced for the same language as the current compilation. You cannot use a C precompiled header for a C++ compilation.
- The precompiled header file must have been produced by the same compiler binary as the current compilation is using.
- Any macros defined before the precompiled header is included must either be defined in the same way as when the precompiled header was generated, or must not affect the

precompiled header, which usually means that they don't appear in the precompiled header at all.

The '-D' option is one way to define a macro before a precompiled header is included; using a #define can also do it. There are also some options that define macros implicitly, like '-O' and '-Wdeprecated'; the same rule applies to macros defined this way.

- If debugging information is output when using the precompiled header, using '-g' or similar, the same kind of debugging information must have been output when building the precompiled header. However, a precompiled header built using '-g' can be used in a compilation when no debugging information is being output.
- The same '-m' options must generally be used when building and using the precompiled header. See Section 3.18 [Submodel Options], page 245, for any cases where this rule is relaxed.
- Each of the following options must be the same when building and using the precompiled header:

  ```
  -fexceptions
  ```

- Some other command-line options starting with '-f', '-p', or '-O' must be defined in the same way as when the precompiled header was generated. At present, it's not clear which options are safe to change and which are not; the safest choice is to use exactly the same options when generating and using the precompiled header. The following are known to be safe:

  ```
  -fmessage-length= -fpreprocessed -fsched-interblock
  -fsched-spec -fsched-spec-load -fsched-spec-load-dangerous
  -fsched-verbose=number -fschedule-insns -fvisibility=
  -pedantic-errors
  ```

For all of these except the last, the compiler automatically ignores the precompiled header if the conditions aren't met. If you find an option combination that doesn't work and doesn't cause the precompiled header to be ignored, please consider filing a bug report, see Chapter 14 [Bugs], page 885.

If you do use differing options when generating and using the precompiled header, the actual behavior is a mixture of the behavior for the options. For instance, if you use '-g' to generate the precompiled header but not when using it, you may or may not get debugging information for routines in the precompiled header.

4 C Implementation-Defined Behavior

A conforming implementation of ISO C is required to document its choice of behavior in each of the areas that are designated "implementation defined". The following lists all such areas, along with the section numbers from the ISO/IEC 9899:1990, ISO/IEC 9899:1999 and ISO/IEC 9899:2011 standards. Some areas are only implementation-defined in one version of the standard.

Some choices depend on the externally determined ABI for the platform (including standard character encodings) which GCC follows; these are listed as "determined by ABI" below. See Chapter 9 [Binary Compatibility], page 845, and `http://gcc.gnu.org/readings.html`. Some choices are documented in the preprocessor manual. See Section "Implementation-defined behavior" in *The C Preprocessor*. Some choices are made by the library and operating system (or other environment when compiling for a freestanding environment); refer to their documentation for details.

4.1 Translation

- *How a diagnostic is identified (C90 3.7, C99 and C11 3.10, C90, C99 and C11 5.1.1.3).*

  Diagnostics consist of all the output sent to stderr by GCC.
- *Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (C90, C99 and C11 5.1.1.2).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.

4.2 Environment

The behavior of most of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

- *The mapping between physical source file multibyte characters and the source character set in translation phase 1 (C90, C99 and C11 5.1.1.2).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.

4.3 Identifiers

- *Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (C99 and C11 6.4.2).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.
- *The number of significant initial characters in an identifier (C90 6.1.2, C90, C99 and C11 5.2.4.1, C99 and C11 6.4.2).*

  For internal names, all characters are significant. For external names, the number of significant characters are defined by the linker; for almost all targets, all characters are significant.
- *Whether case distinctions are significant in an identifier with external linkage (C90 6.1.2).*

  This is a property of the linker. C99 and C11 require that case distinctions are always significant in identifiers with external linkage and systems without this property are not supported by GCC.

4.4 Characters

- *The number of bits in a byte (C90 3.4, C99 and C11 3.6).*

  Determined by ABI.
- *The values of the members of the execution character set (C90, C99 and C11 5.2.1).*

  Determined by ABI.
- *The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (C90, C99 and C11 5.2.2).*

  Determined by ABI.
- *The value of a* `char` *object into which has been stored any character other than a member of the basic execution character set (C90 6.1.2.5, C99 and C11 6.2.5).*

  Determined by ABI.
- *Which of* `signed char` *or* `unsigned char` *has the same range, representation, and behavior as "plain"* `char` *(C90 6.1.2.5, C90 6.2.1.1, C99 and C11 6.2.5, C99 and C11 6.3.1.1).*

  Determined by ABI. The options '`-funsigned-char`' and '`-fsigned-char`' change the default. See Section 3.4 [Options Controlling C Dialect], page 37.
- *The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (C90 6.1.3.4, C99 and C11 6.4.4.4, C90, C99 and C11 5.1.1.2).*

  Determined by ABI.
- *The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (C90 6.1.3.4, C99 and C11 6.4.4.4).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.
- *The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (C90 6.1.3.4, C99 and C11 6.4.4.4).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.
- *The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (C90 6.1.3.4, C99 and C11 6.4.4.4).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.
- *Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (C11 6.4.5).*

  Such tokens may not be concatenated.
- *The current locale used to convert a wide string literal into corresponding wide character codes (C90 6.1.4, C99 and C11 6.4.5).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.
- *The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (C90 6.1.4, C99 and C11 6.4.5).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*.

- *The encoding of any of* `wchar_t`, `char16_t`, *and* `char32_t` *where the corresponding standard encoding macro (*`__STDC_ISO_10646__`, `__STDC_UTF_16__`, *or* `__STDC_UTF_32__`*) is not defined (C11 6.10.8.2).*

  See Section "Implementation-defined behavior" in *The C Preprocessor*. `char16_t` and `char32_t` literals are always encoded in UTF-16 and UTF-32 respectively.

4.5 Integers

- *Any extended integer types that exist in the implementation (C99 and C11 6.2.5).*

  GCC does not support any extended integer types.

- *Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (C99 and C11 6.2.6.2).*

  GCC supports only two's complement integer types, and all bit patterns are ordinary values.

- *The rank of any extended integer type relative to another extended integer type with the same precision (C99 and C11 6.3.1.1).*

  GCC does not support any extended integer types.

- *The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (C90 6.2.1.2, C99 and C11 6.3.1.3).*

  For conversion to a type of width N, the value is reduced modulo 2^N to be within range of the type; no signal is raised.

- *The results of some bitwise operations on signed integers (C90 6.3, C99 and C11 6.5).*

  Bitwise operators act on the representation of the value including both the sign and value bits, where the sign bit is considered immediately above the highest-value value bit. Signed '`>>`' acts on negative numbers by sign extension.

  As an extension to the C language, GCC does not use the latitude given in C99 and C11 only to treat certain aspects of signed '`<<`' as undefined. However, '`-fsanitize=shift`' (and '`-fsanitize=undefined`') will diagnose such cases. They are also diagnosed where constant expressions are required.

- *The sign of the remainder on integer division (C90 6.3.5).*

  GCC always follows the C99 and C11 requirement that the result of division is truncated towards zero.

4.6 Floating Point

- *The accuracy of the floating-point operations and of the library functions in* `<math.h>` *and* `<complex.h>` *that return floating-point results (C90, C99 and C11 5.2.4.2.2).*

  The accuracy is unknown.

- *The rounding behaviors characterized by non-standard values of* `FLT_ROUNDS` *(C90, C99 and C11 5.2.4.2.2).*

  GCC does not use such values.

- *The evaluation methods characterized by non-standard negative values of* `FLT_EVAL_METHOD` *(C99 and C11 5.2.4.2.2).*

  GCC does not use such values.
- *The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (C90 6.2.1.3, C99 and C11 6.3.1.4).*

  C99 Annex F is followed.
- *The direction of rounding when a floating-point number is converted to a narrower floating-point number (C90 6.2.1.4, C99 and C11 6.3.1.5).*

  C99 Annex F is followed.
- *How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (C90 6.1.3.1, C99 and C11 6.4.4.2).*

  C99 Annex F is followed.
- *Whether and how floating expressions are contracted when not disallowed by the* `FP_CONTRACT` *pragma (C99 and C11 6.5).*

  Expressions are currently only contracted if '`-ffp-contract=fast`', '`-funsafe-math-optimizations`' or '`-ffast-math`' are used. This is subject to change.
- *The default state for the* `FENV_ACCESS` *pragma (C99 and C11 7.6.1).*

  This pragma is not implemented, but the default is to "off" unless '`-frounding-math`' is used in which case it is "on".
- *Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (C99 and C11 7.6, C99 and C11 7.12).*

  This is dependent on the implementation of the C library, and is not defined by GCC itself.
- *The default state for the* `FP_CONTRACT` *pragma (C99 and C11 7.12.2).*

  This pragma is not implemented. Expressions are currently only contracted if '`-ffp-contract=fast`', '`-funsafe-math-optimizations`' or '`-ffast-math`' are used. This is subject to change.
- *Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (C99 F.9).*

  This is dependent on the implementation of the C library, and is not defined by GCC itself.
- *Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (C99 F.9).*

  This is dependent on the implementation of the C library, and is not defined by GCC itself.

4.7 Arrays and Pointers

- *The result of converting a pointer to an integer or vice versa (C90 6.3.4, C99 and C11 6.3.2.3).*

A cast from pointer to integer discards most-significant bits if the pointer representation is larger than the integer type, sign-extends[1] if the pointer representation is smaller than the integer type, otherwise the bits are unchanged.

A cast from integer to pointer discards most-significant bits if the pointer representation is smaller than the integer type, extends according to the signedness of the integer type if the pointer representation is larger than the integer type, otherwise the bits are unchanged.

When casting from pointer to integer and back again, the resulting pointer must reference the same object as the original pointer, otherwise the behavior is undefined. That is, one may not use integer arithmetic to avoid the undefined behavior of pointer arithmetic as proscribed in C99 and C11 6.5.6/8.

- *The size of the result of subtracting two pointers to elements of the same array (C90 6.3.6, C99 and C11 6.5.6).*

  The value is as specified in the standard and the type is determined by the ABI.

4.8 Hints

- *The extent to which suggestions made by using the* `register` *storage-class specifier are effective (C90 6.5.1, C99 and C11 6.7.1).*

  The `register` specifier affects code generation only in these ways:

  - When used as part of the register variable extension, see Section 6.47.5 [Explicit Register Variables], page 621.
  - When '`-O0`' is in use, the compiler allocates distinct stack memory for all variables that do not have the `register` storage-class specifier; if `register` is specified, the variable may have a shorter lifespan than the code would indicate and may never be placed in memory.
  - On some rare x86 targets, `setjmp` doesn't save the registers in all circumstances. In those cases, GCC doesn't allocate any variables in registers unless they are marked `register`.

- *The extent to which suggestions made by using the inline function specifier are effective (C99 and C11 6.7.4).*

  GCC will not inline any functions if the '`-fno-inline`' option is used or if '`-O0`' is used. Otherwise, GCC may still be unable to inline a function for many reasons; the '`-Winline`' option may be used to determine if a function has not been inlined and why not.

4.9 Structures, Unions, Enumerations, and Bit-Fields

- *A member of a union object is accessed using a member of a different type (C90 6.3.2.3).*

  The relevant bytes of the representation of the object are treated as an object of the type used for the access. See [Type-punning], page 151. This may be a trap representation.

- *Whether a "plain"* `int` *bit-field is treated as a* `signed int` *bit-field or as an* `unsigned int` *bit-field (C90 6.5.2, C90 6.5.2.1, C99 and C11 6.7.2, C99 and C11 6.7.2.1).*

[1] Future versions of GCC may zero-extend, or use a target-defined `ptr_extend` pattern. Do not rely on sign extension.

By default it is treated as `signed int` but this may be changed by the '`-funsigned-bitfields`' option.

- *Allowable bit-field types other than* `_Bool`, `signed int`, *and* `unsigned int` *(C99 and C11 6.7.2.1).*

  Other integer types, such as `long int`, and enumerated types are permitted even in strictly conforming mode.

- *Whether atomic types are permitted for bit-fields (C11 6.7.2.1).*

  Atomic types are not permitted for bit-fields.

- *Whether a bit-field can straddle a storage-unit boundary (C90 6.5.2.1, C99 and C11 6.7.2.1).*

  Determined by ABI.

- *The order of allocation of bit-fields within a unit (C90 6.5.2.1, C99 and C11 6.7.2.1).*

  Determined by ABI.

- *The alignment of non-bit-field members of structures (C90 6.5.2.1, C99 and C11 6.7.2.1).*

  Determined by ABI.

- *The integer type compatible with each enumerated type (C90 6.5.2.2, C99 and C11 6.7.2.2).*

  Normally, the type is `unsigned int` if there are no negative values in the enumeration, otherwise `int`. If '`-fshort-enums`' is specified, then if there are negative values it is the first of `signed char`, `short` and `int` that can represent all the values, otherwise it is the first of `unsigned char`, `unsigned short` and `unsigned int` that can represent all the values.

  On some targets, '`-fshort-enums`' is the default; this is determined by the ABI.

4.10 Qualifiers

- *What constitutes an access to an object that has volatile-qualified type (C90 6.5.3, C99 and C11 6.7.3).*

  Such an object is normally accessed by pointers and used for accessing hardware. In most expressions, it is intuitively obvious what is a read and what is a write. For example

  ```
  volatile int *dst = somevalue;
  volatile int *src = someothervalue;
  *dst = *src;
  ```

  will cause a read of the volatile object pointed to by *src* and store the value into the volatile object pointed to by *dst*. There is no guarantee that these reads and writes are atomic, especially for objects larger than `int`.

  However, if the volatile storage is not being modified, and the value of the volatile storage is not used, then the situation is less obvious. For example

  ```
  volatile int *src = somevalue;
  *src;
  ```

  According to the C standard, such an expression is an rvalue whose type is the unqualified version of its original type, i.e. `int`. Whether GCC interprets this as a read of the

volatile object being pointed to or only as a request to evaluate the expression for its side effects depends on this type.

If it is a scalar type, or on most targets an aggregate type whose only member object is of a scalar type, or a union type whose member objects are of scalar types, the expression is interpreted by GCC as a read of the volatile object; in the other cases, the expression is only evaluated for its side effects.

4.11 Declarators

- *The maximum number of declarators that may modify an arithmetic, structure or union type (C90 6.5.4).*

  GCC is only limited by available memory.

4.12 Statements

- *The maximum number of* `case` *values in a* `switch` *statement (C90 6.6.4.2).*

  GCC is only limited by available memory.

4.13 Preprocessing Directives

See Section "Implementation-defined behavior" in *The C Preprocessor*, for details of these aspects of implementation-defined behavior.

- *The locations within* `#pragma` *directives where header name preprocessing tokens are recognized (C11 6.4, C11 6.4.7).*
- *How sequences in both forms of header names are mapped to headers or external source file names (C90 6.1.7, C99 and C11 6.4.7).*
- *Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (C90 6.8.1, C99 and C11 6.10.1).*
- *Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (C90 6.8.1, C99 and C11 6.10.1).*
- *The places that are searched for an included '*`<>`*' delimited header, and how the places are specified or the header is identified (C90 6.8.2, C99 and C11 6.10.2).*
- *How the named source file is searched for in an included '*`""`*' delimited header (C90 6.8.2, C99 and C11 6.10.2).*
- *The method by which preprocessing tokens (possibly resulting from macro expansion) in a* `#include` *directive are combined into a header name (C90 6.8.2, C99 and C11 6.10.2).*
- *The nesting limit for* `#include` *processing (C90 6.8.2, C99 and C11 6.10.2).*
- *Whether the '*`#`*' operator inserts a '\' character before the '\' character that begins a universal character name in a character constant or string literal (C99 and C11 6.10.3.2).*
- *The behavior on each recognized non-*`STDC` `#pragma` *directive (C90 6.8.6, C99 and C11 6.10.6).*

See Section "Pragmas" in *The C Preprocessor*, for details of pragmas accepted by GCC on all targets. See Section 6.62 [Pragmas Accepted by GCC], page 801, for details of target-specific pragmas.

- *The definitions for* `__DATE__` *and* `__TIME__` *when respectively, the date and time of translation are not available (C90 6.8.8, C99 6.10.8, C11 6.10.8.1).*

4.14 Library Functions

The behavior of most of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

- *The null pointer constant to which the macro* `NULL` *expands (C90 7.1.6, C99 7.17, C11 7.19).*

  In `<stddef.h>`, `NULL` expands to `((void *)0)`. GCC does not provide the other headers which define `NULL` and some library implementations may use other definitions in those headers.

4.15 Architecture

- *The values or expressions assigned to the macros specified in the headers* `<float.h>`, `<limits.h>`, *and* `<stdint.h>` *(C90, C99 and C11 5.2.4.2, C99 7.18.2, C99 7.18.3, C11 7.20.2, C11 7.20.3).*

  Determined by ABI.

- *The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (C11 6.2.4).*

  Such accesses are supported, subject to the same requirements for synchronization for concurrent accesses as for concurrent accesses to any object.

- *The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (C99 and C11 6.2.6.1).*

  Determined by ABI.

- *Whether any extended alignments are supported and the contexts in which they are supported (C11 6.2.8).*

  Extended alignments up to 2^{28} (bytes) are supported for objects of automatic storage duration. Alignments supported for objects of static and thread storage duration are determined by the ABI.

- *Valid alignment values other than those returned by an* `_Alignof` *expression for fundamental types, if any (C11 6.2.8).*

  Valid alignments are powers of 2 up to and including 2^{28}.

- *The value of the result of the* `sizeof` *and* `_Alignof` *operators (C90 6.3.3.4, C99 and C11 6.5.3.4).*

  Determined by ABI.

4.16 Locale-Specific Behavior

The behavior of these points are dependent on the implementation of the C library, and are not defined by GCC itself.

5 C++ Implementation-Defined Behavior

A conforming implementation of ISO C++ is required to document its choice of behavior in each of the areas that are designated "implementation defined". The following lists all such areas, along with the section numbers from the ISO/IEC 14882:1998 and ISO/IEC 14882:2003 standards. Some areas are only implementation-defined in one version of the standard.

Some choices depend on the externally determined ABI for the platform (including standard character encodings) which GCC follows; these are listed as "determined by ABI" below. See Chapter 9 [Binary Compatibility], page 845, and `http://gcc.gnu.org/readings.html`. Some choices are documented in the preprocessor manual. See Section "Implementation-defined behavior" in *The C Preprocessor*. Some choices are documented in the corresponding document for the C language. See Chapter 4 [C Implementation], page 441. Some choices are made by the library and operating system (or other environment when compiling for a freestanding environment); refer to their documentation for details.

5.1 Conditionally-Supported Behavior

Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support (C++0x 1.4).

- *Whether an argument of class type with a non-trivial copy constructor or destructor can be passed to ... (C++0x 5.2.2).*

  Such argument passing is supported, using the same pass-by-invisible-reference approach used for normal function arguments of such types.

5.2 Exception Handling

- *In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before std::terminate() is called (C++98 15.5.1).*

  The stack is not unwound before std::terminate is called.

c Copyright (C) 1988-2019 Free Software Foundation, Inc.

6 Extensions to the C Language Family

GNU C provides several language features not found in ISO standard C. (The '`-pedantic`' option directs GCC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is always defined under GCC.

These extensions are available in C and Objective-C. Most of them are also available in C++. See Chapter 7 [Extensions to the C++ Language], page 815, for extensions that apply *only* to C++.

Some features that are in ISO C99 but not C90 or C++ are also, as extensions, accepted by GCC in C90 mode and in C++.

6.1 Statements and Declarations in Expressions

A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression.

Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

The last thing in the compound statement should be an expression followed by a semicolon; the value of this subexpression serves as the value of the entire construct. (If you use some other kind of statement last within the braces, the construct has type `void`, and thus effectively no value.)

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here taken as `int`), you can avoid this problem by defining the macro as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Note that introducing variable declarations (as we do in `maxint`) can cause variable shadowing, so while this example using the `max` macro produces correct results:

```
int _a = 1, _b = 2, c;
c = max (_a, _b);
```

this example using maxint will not:

```
int _a = 1, _b = 2, c;
c = maxint (_a, _b);
```

This problem may for instance occur when we use this pattern recursively, like so:

```
#define maxint3(a, b, c) \
  ({int _a = (a), _b = (b), _c = (c); maxint (maxint (_a, _b), _c); })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit-field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` or `__auto_type` (see Section 6.7 [Typeof], page 459).

In G++, the result value of a statement expression undergoes array and function pointer decay, and is returned by value to the enclosing expression. For instance, if `A` is a class, then

```
A a;

({a;}).Foo ()
```

constructs a temporary `A` object to hold the result of the statement expression, and that is used to invoke `Foo`. Therefore the `this` pointer observed by `Foo` is not the address of `a`.

In a statement expression, any temporaries created within a statement are destroyed at that statement's end. This makes statement expressions inside macros slightly different from function calls. In the latter case temporaries introduced during argument evaluation are destroyed at the end of the statement that includes the function call. In the statement expression case they are destroyed during the statement expression. For instance,

```
#define macro(a)  ({__typeof__(a) b = (a); b + 3; })
template<typename T> T function(T a) { T b = a; return b + 3; }

void foo ()
{
  macro (X ());
  function (X ());
}
```

has different places where temporaries are destroyed. For the `macro` case, the temporary `X` is destroyed just after the initialization of `b`. In the `function` case that temporary is destroyed when the function returns.

These considerations mean that it is probably a bad idea to use statement expressions of this form in header files that are designed to work with C++. (Note that some versions of the GNU C Library contained header files using statement expressions that lead to precisely this bug.)

Jumping into a statement expression with `goto` or using a `switch` statement outside the statement expression with a `case` or `default` label inside the statement expression is not permitted. Jumping into a statement expression with a computed `goto` (see Section 6.3 [Labels as Values], page 454) has undefined behavior. Jumping out of a statement expression is permitted, but if the statement expression is part of a larger expression then it is unspecified which other subexpressions of that expression have been evaluated except where the language definition requires certain subexpressions to be evaluated before or after the statement expression. A `break` or `continue` statement inside of a statement expression used in `while`, `do` or `for` loop or `switch` statement condition or `for` statement init or increment expressions jumps to an outer loop or `switch` statement if any (otherwise it is an error), rather than to the loop or `switch` statement in whose condition or init or increment expression it appears. In any case, as with a function call, the evaluation of a statement

expression is not interleaved with the evaluation of other parts of the containing expression. For example,

```
foo (), (({ bar1 (); goto a; 0; }) + bar2 ()), baz();
```

calls `foo` and `bar1` and does not call `baz` but may or may not call `bar2`. If `bar2` is called, it is called after `foo` and before `bar1`.

6.2 Locally Declared Labels

GCC allows you to declare *local labels* in any nested block scope. A local label is just like an ordinary label, but you can only reference it (with a `goto` statement, or by taking its address) within the block in which it is declared.

A local label declaration looks like this:

```
__label__ label;
```

or

```
__label__ label1, label2, /* ... */;
```

Local label declarations must come at the beginning of the block, before any ordinary declarations or statements.

The label declaration defines the label *name*, but does not define the label itself. You must do this in the usual way, with `label:`, within the statements of the statement expression.

The local label feature is useful for complex macros. If a macro contains nested loops, a `goto` can be useful for breaking out of them. However, an ordinary label whose scope is the whole function cannot be used: if the macro can be expanded several times in one function, the label is multiply defined in that function. A local label avoids this problem. For example:

```
#define SEARCH(value, array, target)              \
do {                                              \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { (value) = i; goto found; }              \
  (value) = -1;                                   \
 found:;                                          \
} while (0)
```

This could also be written using a statement expression:

```
#define SEARCH(array, target)                     \
({                                                \
  __label__ found;                                \
  typeof (target) _SEARCH_target = (target);      \
  typeof (*(array)) *_SEARCH_array = (array);     \
  int i, j;                                       \
  int value;                                      \
  for (i = 0; i < max; i++)                       \
    for (j = 0; j < max; j++)                     \
      if (_SEARCH_array[i][j] == _SEARCH_target)  \
        { value = i; goto found; }                \
```

```
  value = -1;                                   \
 found:                                         \
  value;                                        \
})
```

Local label declarations also make the labels they declare visible to nested functions, if there are any. See Section 6.4 [Nested Functions], page 455, for details.

6.3 Labels as Values

You can get the address of a label defined in the current function (or a containing function) with the unary operator '`&&`'. The value has type `void *`. This value is a constant and can be used wherever a constant of that type is valid. For example:

```
void *ptr;
/* ... */
ptr = &&foo;
```

To use these values, you need to be able to jump to one. This is done with the computed goto statement[1], `goto *`*`exp`*`;`. For example,

```
goto *ptr;
```

Any expression of type `void *` is allowed.

One way of using these constants is in initializing a static array that serves as a jump table:

```
static void *array[] = { &&foo, &&bar, &&hack };
```

Then you can select a label with indexing, like this:

```
goto *array[i];
```

Note that this does not check whether the subscript is in bounds—array indexing in C never does that.

Such an array of label values serves a purpose much like that of the `switch` statement. The `switch` statement is cleaner, so use that rather than an array unless the problem does not fit a `switch` statement very well.

Another use of label values is in an interpreter for threaded code. The labels within the interpreter function can be stored in the threaded code for super-fast dispatching.

You may not use this mechanism to jump to code in a different function. If you do that, totally unpredictable things happen. The best way to avoid this is to store the label address only in automatic variables and never pass it as an argument.

An alternate way to write the above example is

```
static const int array[] = { &&foo - &&foo, &&bar - &&foo,
                             &&hack - &&foo };
goto *(&&foo + array[i]);
```

This is more friendly to code living in shared libraries, as it reduces the number of dynamic relocations that are needed, and by consequence, allows the data to be read-only. This alternative with label differences is not supported for the AVR target, please use the first approach for AVR programs.

The `&&foo` expressions for the same label might have different values if the containing function is inlined or cloned. If a program relies on them being always the same,

[1] The analogous feature in Fortran is called an assigned goto, but that name seems inappropriate in C, where one can do more than simply store label addresses in label variables.

`__attribute__((__noinline__,__noclone__))` should be used to prevent inlining and cloning. If `&&foo` is used in a static variable initializer, inlining and cloning is forbidden.

6.4 Nested Functions

A *nested function* is a function defined inside another function. Nested functions are supported as an extension in GNU C, but are not supported by GNU C++.

The nested function's name is local to the block where it is defined. For example, here we define a nested function named `square`, and call it twice:

```
foo (double a, double b)
{
  double square (double z) { return z * z; }

  return square (a) + square (b);
}
```

The nested function can access all the variables of the containing function that are visible at the point of its definition. This is called *lexical scoping*. For example, here we show a nested function which uses an inherited variable named `offset`:

```
bar (int *array, int offset, int size)
{
  int access (int *array, int index)
    { return array[index + offset]; }
  int i;
  /* ... */
  for (i = 0; i < size; i++)
    /* ... */ access (array, i) /* ... */
}
```

Nested function definitions are permitted within functions in the places where variable definitions are allowed; that is, in any block, mixed with the other declarations and statements in the block.

It is possible to call the nested function from outside the scope of its name by storing its address or passing the address to another function:

```
hack (int *array, int size)
{
  void store (int index, int value)
    { array[index] = value; }

  intermediate (store, size);
}
```

Here, the function `intermediate` receives the address of `store` as an argument. If `intermediate` calls `store`, the arguments given to `store` are used to store into `array`. But this technique works only so long as the containing function (`hack`, in this example) does not exit.

If you try to call the nested function through its address after the containing function exits, all hell breaks loose. If you try to call it after a containing scope level exits, and if it refers to some of the variables that are no longer in scope, you may be lucky, but it's not wise to take the risk. If, however, the nested function does not refer to anything that has gone out of scope, you should be safe.

GCC implements taking the address of a nested function using a technique called *trampolines*. This technique was described in *Lexical Closures for C++* (Thomas M. Breuel, USENIX C++ Conference Proceedings, October 17-21, 1988).

A nested function can jump to a label inherited from a containing function, provided the label is explicitly declared in the containing function (see Section 6.2 [Local Labels], page 453). Such a jump returns instantly to the containing function, exiting the nested function that did the `goto` and any intermediate functions as well. Here is an example:

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  int i;
  /* ... */
  for (i = 0; i < size; i++)
    /* ... */ access (array, i) /* ... */
  /* ... */
  return 0;

 /* Control comes here from access
   if it detects an error.   */
 failure:
  return -1;
}
```

A nested function always has no linkage. Declaring one with `extern` or `static` is erroneous. If you need to declare the nested function before its definition, use `auto` (which is otherwise meaningless for function declarations).

```
bar (int *array, int offset, int size)
{
  __label__ failure;
  auto int access (int *, int);
  /* ... */
  int access (int *array, int index)
    {
      if (index > size)
        goto failure;
      return array[index + offset];
    }
  /* ... */
}
```

6.5 Nonlocal Gotos

GCC provides the built-in functions `__builtin_setjmp` and `__builtin_longjmp` which are similar to, but not interchangeable with, the C library functions `setjmp` and `longjmp`. The built-in versions are used internally by GCC's libraries to implement exception handling on some targets. You should use the standard C library functions declared in `<setjmp.h>` in user code instead of the builtins.

The built-in versions of these functions use GCC's normal mechanisms to save and restore registers using the stack on function entry and exit. The jump buffer argument *buf* holds only the information needed to restore the stack frame, rather than the entire set of saved register values.

An important caveat is that GCC arranges to save and restore only those registers known to the specific architecture variant being compiled for. This can make `__builtin_setjmp` and `__builtin_longjmp` more efficient than their library counterparts in some cases, but it can also cause incorrect and mysterious behavior when mixing with code that uses the full register set.

You should declare the jump buffer argument *buf* to the built-in functions as:

```
#include <stdint.h>
intptr_t buf[5];
```

`int __builtin_setjmp` (*intptr_t* ***buf***) [Built-in Function]

This function saves the current stack context in *buf*. `__builtin_setjmp` returns 0 when returning directly, and 1 when returning from `__builtin_longjmp` using the same *buf*.

`void __builtin_longjmp` (*intptr_t* ***buf***, *int* `val`) [Built-in Function]

This function restores the stack context in *buf*, saved by a previous call to `__builtin_setjmp`. After `__builtin_longjmp` is finished, the program resumes execution as if the matching `__builtin_setjmp` returns the value *val*, which must be 1.

Because `__builtin_longjmp` depends on the function return mechanism to restore the stack context, it cannot be called from the same function calling `__builtin_setjmp` to initialize *buf*. It can only be called from a function called (directly or indirectly) from the function calling `__builtin_setjmp`.

6.6 Constructing Function Calls

Using the built-in functions described below, you can record the arguments a function received, and call another function with the same arguments, without knowing the number or types of the arguments.

You can also record the return value of that function call, and later return that value, without knowing what data type the function tried to return (as long as your caller expects that data type).

However, these built-in functions may interact badly with some sophisticated features or other extensions of the language. It is, therefore, not recommended to use them outside very simple functions acting as mere forwarders for their arguments.

`void * __builtin_apply_args ()` [Built-in Function]

This built-in function returns a pointer to data describing how to perform a call with the same arguments as are passed to the current function.

The function saves the arg pointer register, structure value address, and all registers that might be used to pass arguments to a function into a block of memory allocated on the stack. Then it returns the address of that block.

`void * __builtin_apply` (*void* (**function*)(), *void *arguments*, *size_t size*) [Built-in Function]

This built-in function invokes *function* with a copy of the parameters described by *arguments* and *size*.

The value of *arguments* should be the value returned by `__builtin_apply_args`. The argument *size* specifies the size of the stack argument data, in bytes.

This function returns a pointer to data describing how to return whatever value is returned by *function*. The data is saved in a block of memory allocated on the stack.

It is not always simple to compute the proper value for *size*. The value is used by `__builtin_apply` to compute the amount of data that should be pushed on the stack and copied from the incoming argument area.

`void __builtin_return` (*void *result*) [Built-in Function]

This built-in function returns the value described by *result* from the containing function. You should specify, for *result*, a value returned by `__builtin_apply`.

`__builtin_va_arg_pack ()` [Built-in Function]

This built-in function represents all anonymous arguments of an inline function. It can be used only in inline functions that are always inlined, never compiled as a separate function, such as those using `__attribute__ ((__always_inline__))` or `__attribute__ ((__gnu_inline__))` extern inline functions. It must be only passed as last argument to some other function with variable arguments. This is useful for writing small wrapper inlines for variable argument functions, when using preprocessor macros is undesirable. For example:

```
extern int myprintf (FILE *f, const char *format, ...);
extern inline __attribute__ ((__gnu_inline__)) int
myprintf (FILE *f, const char *format, ...)
{
  int r = fprintf (f, "myprintf: ");
  if (r < 0)
    return r;
  int s = fprintf (f, format, __builtin_va_arg_pack ());
  if (s < 0)
    return s;
  return r + s;
}
```

`size_t __builtin_va_arg_pack_len ()` [Built-in Function]

This built-in function returns the number of anonymous arguments of an inline function. It can be used only in inline functions that are always inlined, never compiled as a separate function, such as those using `__attribute__ ((__always_inline__))` or `__attribute__ ((__gnu_inline__))` extern inline functions. For example following does link- or run-time checking of open arguments for optimized code:

```
#ifdef __OPTIMIZE__
extern inline __attribute__((__gnu_inline__)) int
myopen (const char *path, int oflag, ...)
{
  if (__builtin_va_arg_pack_len () > 1)
    warn_open_too_many_arguments ();

  if (__builtin_constant_p (oflag))
```

```
    {
      if ((oflag & O_CREAT) != 0 && __builtin_va_arg_pack_len () < 1)
        {
          warn_open_missing_mode ();
          return __open_2 (path, oflag);
        }
      return open (path, oflag, __builtin_va_arg_pack ());
    }

  if (__builtin_va_arg_pack_len () < 1)
    return __open_2 (path, oflag);

  return open (path, oflag, __builtin_va_arg_pack ());
}
#endif
```

6.7 Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of pointers to functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ISO C programs, write `__typeof__` instead of `typeof`. See Section 6.48 [Alternate Keywords], page 624.

A `typeof` construct can be used anywhere a typedef name can be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

The operand of `typeof` is evaluated for its side effects if and only if it is an expression of variably modified type or the name of such a type.

`typeof` is often useful in conjunction with statement expressions (see Section 6.1 [Statement Exprs], page 451). Here is how the two together can be used to define a safe "maximum" macro which operates on any arithmetic type and evaluates each of its arguments exactly once:

```
#define max(a,b) \
  ({ typeof (a) _a = (a); \
      typeof (b) _b = (b); \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Some more examples of the use of `typeof`:

- This declares `y` with the type of what `x` points to.
  ```
  typeof (*x) y;
  ```
- This declares `y` as an array of such values.
  ```
  typeof (*x) y[4];
  ```
- This declares `y` as an array of pointers to characters:
  ```
  typeof (typeof (char *)[4]) y;
  ```
  It is equivalent to the following traditional C declaration:
  ```
  char *y[4];
  ```
  To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, rewrite it with these macros:
  ```
  #define pointer(T)  typeof(T *)
  #define array(T, N) typeof(T [N])
  ```
  Now the declaration can be rewritten this way:
  ```
  array (pointer (char), 4) y;
  ```
  Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

In GNU C, but not GNU C++, you may also declare the type of a variable as `__auto_type`. In that case, the declaration must declare only one variable, whose declarator must just be an identifier, the declaration must be initialized, and the type of the variable is determined by the initializer; the name of the variable is not in scope until after the initializer. (In C++, you should use C++11 `auto` for this purpose.) Using `__auto_type`, the "maximum" macro above could be written as:

```
#define max(a,b) \
  ({ __auto_type _a = (a); \
      __auto_type _b = (b); \
    _a > _b ? _a : _b; })
```

Using `__auto_type` instead of `typeof` has two advantages:

- Each argument to the macro appears only once in the expansion of the macro. This prevents the size of the macro expansion growing exponentially when calls to such macros are nested inside arguments of such macros.
- If the argument to the macro has variably modified type, it is evaluated only once when using `__auto_type`, but twice if `typeof` is used.

6.8 Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

6.9 128-bit Integers

As an extension the integer scalar type `__int128` is supported for targets which have an integer mode wide enough to hold 128 bits. Simply write `__int128` for a signed 128-bit integer, or `unsigned __int128` for an unsigned 128-bit integer. There is no support in GCC for expressing an integer constant of type `__int128` for targets with `long long` integer less than 128 bits wide.

6.10 Double-Word Integers

ISO C99 and ISO C++11 support data types for integers that are at least 64 bits wide, and as an extension GCC supports them in C90 and C++98 modes. Simply write `long long int` for a signed integer, or `unsigned long long int` for an unsigned integer. To make an integer constant of type `long long int`, add the suffix '`LL`' to the integer. To make an integer constant of type `unsigned long long int`, add the suffix '`ULL`' to the integer.

You can use these types in arithmetic like any other integer types. Addition, subtraction, and bitwise boolean operations on these types are open-coded on all types of machines. Multiplication is open-coded if the machine supports a fullword-to-doubleword widening multiply instruction. Division and shifts are open-coded only on machines that provide special support. The operations that are not open-coded use special library routines that come with GCC.

There may be pitfalls when you use `long long` types for function arguments without function prototypes. If a function expects type `int` for its argument, and you pass a value of type `long long int`, confusion results because the caller and the subroutine disagree about the number of bytes for the argument. Likewise, if the function expects `long long int` and you pass `int`. The best way to avoid such problems is to use prototypes.

6.11 Complex Numbers

ISO C99 supports complex floating data types, and as an extension GCC supports them in C90 mode and in C++. GCC also supports complex integer data types which are not part of ISO C99. You can declare complex types using the keyword `_Complex`. As an extension, the older GNU keyword `__complex__` is also supported.

For example, '`_Complex double x;`' declares `x` as a variable whose real part and imaginary part are both of type `double`. '`_Complex short int y;`' declares `y` to have real and imaginary parts of type `short int`; this is not likely to be useful, but it shows that the set of complex types is complete.

To write a constant with a complex data type, use the suffix '`i`' or '`j`' (either one; they are equivalent). For example, `2.5fi` has type `_Complex float` and `3i` has type `_Complex int`. Such a constant always has a pure imaginary value, but you can form any complex value you like by adding one to a real constant. This is a GNU extension; if you have an ISO C99 conforming C library (such as the GNU C Library), and want to construct complex constants of floating type, you should include `<complex.h>` and use the macros `I` or `_Complex_I` instead.

The ISO C++14 library also defines the '`i`' suffix, so C++14 code that includes the '`<complex>`' header cannot use '`i`' for the GNU extension. The '`j`' suffix still has the GNU meaning.

To extract the real part of a complex-valued expression *exp*, write `__real__` *exp*. Likewise, use `__imag__` to extract the imaginary part. This is a GNU extension; for values of floating type, you should use the ISO C99 functions `crealf`, `creal`, `creall`, `cimagf`, `cimag` and `cimagl`, declared in `<complex.h>` and also provided as built-in functions by GCC.

The operator '~' performs complex conjugation when used on a value with a complex type. This is a GNU extension; for values of floating type, you should use the ISO C99 functions `conjf`, `conj` and `conjl`, declared in `<complex.h>` and also provided as built-in functions by GCC.

GCC can allocate complex automatic variables in a noncontiguous fashion; it's even possible for the real part to be in a register while the imaginary part is on the stack (or vice versa). Only the DWARF debug info format can represent this, so use of DWARF is recommended. If you are using the stabs debug info format, GCC describes a noncontiguous complex variable as if it were two separate variables of noncomplex type. If the variable's actual name is `foo`, the two fictitious variables are named `foo$real` and `foo$imag`. You can examine and set these two fictitious variables with your debugger.

6.12 Additional Floating Types

ISO/IEC TS 18661-3:2015 defines C support for additional floating types `_Floatn` and `_Floatnx`, and GCC supports these type names; the set of types supported depends on the target architecture. These types are not supported when compiling C++. Constants with these types use suffixes `fn` or `Fn` and `fnx` or `Fnx`. These type names can be used together with `_Complex` to declare complex types.

As an extension, GNU C and GNU C++ support additional floating types, which are not supported by all targets.

- `__float128` is available on i386, x86_64, IA-64, and hppa HP-UX, as well as on PowerPC GNU/Linux targets that enable the vector scalar (VSX) instruction set. `__float128` supports the 128-bit floating type. On i386, x86_64, PowerPC, and IA-64 other than HP-UX, `__float128` is an alias for `_Float128`. On hppa and IA-64 HP-UX, `__float128` is an alias for `long double`.
- `__float80` is available on the i386, x86_64, and IA-64 targets, and supports the 80-bit (`XFmode`) floating type. It is an alias for the type name `_Float64x` on these targets.
- `__ibm128` is available on PowerPC targets, and provides access to the IBM extended double format which is the current format used for `long double`. When `long double` transitions to `__float128` on PowerPC in the future, `__ibm128` will remain for use in conversions between the two types.

Support for these additional types includes the arithmetic operators: add, subtract, multiply, divide; unary arithmetic operators; relational operators; equality operators; and conversions to and from integer and other floating types. Use a suffix '`w`' or '`W`' in a literal constant of type `__float80` or type `__ibm128`. Use a suffix '`q`' or '`Q`' for `_float128`.

In order to use `_Float128`, `__float128`, and `__ibm128` on PowerPC Linux systems, you must use the '`-mfloat128`' option. It is expected in future versions of GCC that `_Float128` and `__float128` will be enabled automatically.

The `_Float128` type is supported on all systems where `__float128` is supported or where `long double` has the IEEE binary128 format. The `_Float64x` type is supported on

all systems where `__float128` is supported. The `_Float32` type is supported on all systems supporting IEEE binary32; the `_Float64` and `_Float32x` types are supported on all systems supporting IEEE binary64. The `_Float16` type is supported on AArch64 systems by default, and on ARM systems when the IEEE format for 16-bit floating-point types is selected with '`-mfp16-format=ieee`'. GCC does not currently support `_Float128x` on any systems.

On the i386, x86_64, IA-64, and HP-UX targets, you can declare complex types using the corresponding internal complex type, `XCmode` for `__float80` type and `TCmode` for `__float128` type:

```
typedef _Complex float __attribute__((mode(TC))) _Complex128;
typedef _Complex float __attribute__((mode(XC))) _Complex80;
```

On the PowerPC Linux VSX targets, you can declare complex types using the corresponding internal complex type, `KCmode` for `__float128` type and `ICmode` for `__ibm128` type:

```
typedef _Complex float __attribute__((mode(KC))) _Complex_float128;
typedef _Complex float __attribute__((mode(IC))) _Complex_ibm128;
```

6.13 Half-Precision Floating Point

On ARM and AArch64 targets, GCC supports half-precision (16-bit) floating point via the `__fp16` type defined in the ARM C Language Extensions. On ARM systems, you must enable this type explicitly with the '`-mfp16-format`' command-line option in order to use it.

ARM targets support two incompatible representations for half-precision floating-point values. You must choose one of the representations and use it consistently in your program.

Specifying '`-mfp16-format=ieee`' selects the IEEE 754-2008 format. This format can represent normalized values in the range of 2^{-14} to 65504. There are 11 bits of significand precision, approximately 3 decimal digits.

Specifying '`-mfp16-format=alternative`' selects the ARM alternative format. This representation is similar to the IEEE format, but does not support infinities or NaNs. Instead, the range of exponents is extended, so that this format can represent normalized values in the range of 2^{-14} to 131008.

The GCC port for AArch64 only supports the IEEE 754-2008 format, and does not require use of the '`-mfp16-format`' command-line option.

The `__fp16` type may only be used as an argument to intrinsics defined in `<arm_fp16.h>`, or as a storage format. For purposes of arithmetic and other operations, `__fp16` values in C or C++ expressions are automatically promoted to `float`.

The ARM target provides hardware support for conversions between `__fp16` and `float` values as an extension to VFP and NEON (Advanced SIMD), and from ARMv8-A provides hardware support for conversions between `__fp16` and `double` values. GCC generates code using these hardware instructions if you compile with options to select an FPU that provides them; for example, '`-mfpu=neon-fp16 -mfloat-abi=softfp`', in addition to the '`-mfp16-format`' option to select a half-precision format.

Language-level support for the `__fp16` data type is independent of whether GCC generates code using hardware floating-point instructions. In cases where hardware support is not specified, GCC implements conversions between `__fp16` and other types as library calls.

It is recommended that portable code use the `_Float16` type defined by ISO/IEC TS 18661-3:2015. See Section 6.12 [Floating Types], page 462.

6.14 Decimal Floating Types

As an extension, GNU C supports decimal floating types as defined in the N1312 draft of ISO/IEC WDTR24732. Support for decimal floating types in GCC will evolve as the draft technical report changes. Calling conventions for any target might also change. Not all targets support decimal floating types.

The decimal floating types are `_Decimal32`, `_Decimal64`, and `_Decimal128`. They use a radix of ten, unlike the floating types `float`, `double`, and `long double` whose radix is not specified by the C standard but is usually two.

Support for decimal floating types includes the arithmetic operators add, subtract, multiply, divide; unary arithmetic operators; relational operators; equality operators; and conversions to and from integer and other floating types. Use a suffix '`df`' or '`DF`' in a literal constant of type `_Decimal32`, '`dd`' or '`DD`' for `_Decimal64`, and '`dl`' or '`DL`' for `_Decimal128`.

GCC support of decimal float as specified by the draft technical report is incomplete:

- When the value of a decimal floating type cannot be represented in the integer type to which it is being converted, the result is undefined rather than the result value specified by the draft technical report.
- GCC does not provide the C library functionality associated with '`math.h`', '`fenv.h`', '`stdio.h`', '`stdlib.h`', and '`wchar.h`', which must come from a separate C library implementation. Because of this the GNU C compiler does not define macro `__STDC_DEC_FP__` to indicate that the implementation conforms to the technical report.

Types `_Decimal32`, `_Decimal64`, and `_Decimal128` are supported by the DWARF debug information format.

6.15 Hex Floats

ISO C99 and ISO C++17 support floating-point numbers written not only in the usual decimal notation, such as `1.55e1`, but also numbers such as `0x1.fp3` written in hexadecimal format. As a GNU extension, GCC supports this in C90 mode (except in some cases when strictly conforming) and in C++98, C++11 and C++14 modes. In that format the '`0x`' hex introducer and the '`p`' or '`P`' exponent field are mandatory. The exponent is a decimal number that indicates the power of 2 by which the significant part is multiplied. Thus '`0x1.f`' is $1\frac{15}{16}$, '`p3`' multiplies it by 8, and the value of `0x1.fp3` is the same as `1.55e1`.

Unlike for floating-point numbers in the decimal notation the exponent is always required in the hexadecimal notation. Otherwise the compiler would not be able to resolve the ambiguity of, e.g., `0x1.f`. This could mean `1.0f` or `1.9375` since '`f`' is also the extension for floating-point constants of type `float`.

6.16 Fixed-Point Types

As an extension, GNU C supports fixed-point types as defined in the N1169 draft of ISO/IEC DTR 18037. Support for fixed-point types in GCC will evolve as the draft technical report changes. Calling conventions for any target might also change. Not all targets support fixed-point types.

The fixed-point types are `short _Fract`, `_Fract`, `long _Fract`, `long long _Fract`, `unsigned short _Fract`, `unsigned _Fract`, `unsigned long _Fract`, `unsigned long long _Fract`, `_Sat short _Fract`, `_Sat _Fract`, `_Sat long _Fract`, `_Sat long long _Fract`, `_Sat unsigned short _Fract`, `_Sat unsigned _Fract`, `_Sat unsigned long _Fract`, `_Sat unsigned long long _Fract`, `short _Accum`, `_Accum`, `long _Accum`, `long long _Accum`, `unsigned short _Accum`, `unsigned _Accum`, `unsigned long _Accum`, `unsigned long long _Accum`, `_Sat short _Accum`, `_Sat _Accum`, `_Sat long _Accum`, `_Sat long long _Accum`, `_Sat unsigned short _Accum`, `_Sat unsigned _Accum`, `_Sat unsigned long _Accum`, `_Sat unsigned long long _Accum`.

Fixed-point data values contain fractional and optional integral parts. The format of fixed-point data varies and depends on the target machine.

Support for fixed-point types includes:

- prefix and postfix increment and decrement operators (`++`, `--`)
- unary arithmetic operators (`+`, `-`, `!`)
- binary arithmetic operators (`+`, `-`, `*`, `/`)
- binary shift operators (`<<`, `>>`)
- relational operators (`<`, `<=`, `>=`, `>`)
- equality operators (`==`, `!=`)
- assignment operators (`+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`)
- conversions to and from integer, floating-point, or fixed-point types

Use a suffix in a fixed-point literal constant:

- '`hr`' or '`HR`' for `short _Fract` and `_Sat short _Fract`
- '`r`' or '`R`' for `_Fract` and `_Sat _Fract`
- '`lr`' or '`LR`' for `long _Fract` and `_Sat long _Fract`
- '`llr`' or '`LLR`' for `long long _Fract` and `_Sat long long _Fract`
- '`uhr`' or '`UHR`' for `unsigned short _Fract` and `_Sat unsigned short _Fract`
- '`ur`' or '`UR`' for `unsigned _Fract` and `_Sat unsigned _Fract`
- '`ulr`' or '`ULR`' for `unsigned long _Fract` and `_Sat unsigned long _Fract`
- '`ullr`' or '`ULLR`' for `unsigned long long _Fract` and `_Sat unsigned long long _Fract`
- '`hk`' or '`HK`' for `short _Accum` and `_Sat short _Accum`
- '`k`' or '`K`' for `_Accum` and `_Sat _Accum`
- '`lk`' or '`LK`' for `long _Accum` and `_Sat long _Accum`
- '`llk`' or '`LLK`' for `long long _Accum` and `_Sat long long _Accum`
- '`uhk`' or '`UHK`' for `unsigned short _Accum` and `_Sat unsigned short _Accum`
- '`uk`' or '`UK`' for `unsigned _Accum` and `_Sat unsigned _Accum`
- '`ulk`' or '`ULK`' for `unsigned long _Accum` and `_Sat unsigned long _Accum`
- '`ullk`' or '`ULLK`' for `unsigned long long _Accum` and `_Sat unsigned long long _Accum`

GCC support of fixed-point types as specified by the draft technical report is incomplete:

- Pragmas to control overflow and rounding behaviors are not implemented.

Fixed-point types are supported by the DWARF debug information format.

6.17 Named Address Spaces

As an extension, GNU C supports named address spaces as defined in the N1275 draft of ISO/IEC DTR 18037. Support for named address spaces in GCC will evolve as the draft technical report changes. Calling conventions for any target might also change. At present, only the AVR, SPU, M32C, RL78, and x86 targets support address spaces other than the generic address space.

Address space identifiers may be used exactly like any other C type qualifier (e.g., `const` or `volatile`). See the N1275 document for more details.

6.17.1 AVR Named Address Spaces

On the AVR target, there are several address spaces that can be used in order to put read-only data into the flash memory and access that data by means of the special instructions `LPM` or `ELPM` needed to read from flash.

Devices belonging to `avrtiny` and `avrxmega3` can access flash memory by means of `LD*` instructions because the flash memory is mapped into the RAM address space. There is *no need* for language extensions like `__flash` or attribute Section 6.34.3 [`progmem`], page 542. The default linker description files for these devices cater for that feature and `.rodata` stays in flash: The compiler just generates `LD*` instructions, and the linker script adds core specific offsets to all `.rodata` symbols: `0x4000` in the case of `avrtiny` and `0x8000` in the case of `avrxmega3`. See Section 3.18.6 [AVR Options], page 277 for a list of respective devices.

For devices not in `avrtiny` or `avrxmega3`, any data including read-only data is located in RAM (the generic address space) because flash memory is not visible in the RAM address space. In order to locate read-only data in flash memory *and* to generate the right instructions to access this data without using (inline) assembler code, special address spaces are needed.

`__flash` The `__flash` qualifier locates data in the `.progmem.data` section. Data is read using the `LPM` instruction. Pointers to this address space are 16 bits wide.

`__flash1`
`__flash2`
`__flash3`
`__flash4`
`__flash5` These are 16-bit address spaces locating data in section `.progmem`*N*`.data` where *N* refers to address space `__flash`*N*. The compiler sets the `RAMPZ` segment register appropriately before reading data by means of the `ELPM` instruction.

`__memx` This is a 24-bit address space that linearizes flash and RAM: If the high bit of the address is set, data is read from RAM using the lower two bytes as RAM address. If the high bit of the address is clear, data is read from flash with `RAMPZ` set according to the high byte of the address. See Section 6.60.10 [`__builtin_avr_flash_segment`], page 669.

Objects in this address space are located in `.progmemx.data`.

Example

```
char my_read (const __flash char ** p)
{
```

```
    /* p is a pointer to RAM that points to a pointer to flash.
       The first indirection of p reads that flash pointer
       from RAM and the second indirection reads a char from this
       flash address.  */

    return **p;
}

/* Locate array[] in flash memory */
const __flash int array[] = { 3, 5, 7, 11, 13, 17, 19 };

int i = 1;

int main (void)
{
   /* Return 17 by reading from flash memory */
   return array[array[i]];
}
```

For each named address space supported by avr-gcc there is an equally named but uppercase built-in macro defined. The purpose is to facilitate testing if respective address space support is available or not:

```
#ifdef __FLASH
const __flash int var = 1;

int read_var (void)
{
    return var;
}
#else
#include <avr/pgmspace.h> /* From AVR-LibC */

const int var PROGMEM = 1;

int read_var (void)
{
    return (int) pgm_read_word (&var);
}
#endif /* __FLASH */
```

Notice that attribute Section 6.34.3 [progmem], page 542 locates data in flash but accesses to these data read from generic address space, i.e. from RAM, so that you need special accessors like `pgm_read_byte` from AVR-LibC (`http://nongnu.org/avr-libc/user-manual/`) together with attribute `progmem`.

Limitations and caveats

- Reading across the 64 KiB section boundary of the `__flash` or `__flash`*N* address spaces shows undefined behavior. The only address space that supports reading across the 64 KiB flash segment boundaries is `__memx`.
- If you use one of the `__flash`*N* address spaces you must arrange your linker script to locate the `.progmem`*N*`.data` sections according to your needs.
- Any data or pointers to the non-generic address spaces must be qualified as `const`, i.e. as read-only data. This still applies if the data in one of these address spaces like software version number or calibration lookup table are intended to be changed after load time by, say, a boot loader. In this case the right qualification is `const volatile`

so that the compiler must not optimize away known values or insert them as immediates into operands of instructions.

- The following code initializes a variable `pfoo` located in static storage with a 24-bit address:

```
extern const __memx char foo;
const __memx void *pfoo = &foo;
```

- On the reduced Tiny devices like ATtiny40, no address spaces are supported. Just use vanilla C / C++ code without overhead as outlined above. Attribute `progmem` is supported but works differently, see Section 6.34.3 [AVR Variable Attributes], page 542.

6.17.2 M32C Named Address Spaces

On the M32C target, with the R8C and M16C CPU variants, variables qualified with `__far` are accessed using 32-bit addresses in order to access memory beyond the first 64 Ki bytes. If `__far` is used with the M32CM or M32C CPU variants, it has no effect.

6.17.3 RL78 Named Address Spaces

On the RL78 target, variables qualified with `__far` are accessed with 32-bit pointers (20-bit addresses) rather than the default 16-bit addresses. Non-far variables are assumed to appear in the topmost 64 KiB of the address space.

6.17.4 SPU Named Address Spaces

On the SPU target variables may be declared as belonging to another address space by qualifying the type with the `__ea` address space identifier:

```
extern int __ea i;
```

The compiler generates special code to access the variable `i`. It may use runtime library support, or generate special machine instructions to access that address space.

6.17.5 x86 Named Address Spaces

On the x86 target, variables may be declared as being relative to the `%fs` or `%gs` segments.

`__seg_fs`
`__seg_gs` The object is accessed with the respective segment override prefix.

The respective segment base must be set via some method specific to the operating system. Rather than require an expensive system call to retrieve the segment base, these address spaces are not considered to be subspaces of the generic (flat) address space. This means that explicit casts are required to convert pointers between these address spaces and the generic address space. In practice the application should cast to `uintptr_t` and apply the segment base offset that it installed previously.

The preprocessor symbols `__SEG_FS` and `__SEG_GS` are defined when these address spaces are supported.

6.18 Arrays of Length Zero

Declaring zero-length arrays is allowed in GNU C as an extension. A zero-length array can be useful as the last element of a structure that is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

struct line *thisline = (struct line *)
  malloc (sizeof (struct line) + this_length);
thisline->length = this_length;
```

Although the size of a zero-length array is zero, an array member of this kind may increase the size of the enclosing type as a result of tail padding. The offset of a zero-length array member from the beginning of the enclosing structure is the same as the offset of an array with one or more elements of the same type. The alignment of a zero-length array is the same as the alignment of its elements.

Declaring zero-length arrays in other contexts, including as interior members of structure objects or as non-member objects, is discouraged. Accessing elements of zero-length arrays declared in such contexts is undefined and may be diagnosed.

In the absence of the zero-length array extension, in ISO C90 the `contents` array in the example above would typically be declared to have a single element. Unlike a zero-length array which only contributes to the size of the enclosing structure for the purposes of alignment, a one-element array always occupies at least as much space as a single object of the type. Although using one-element arrays this way is discouraged, GCC handles accesses to trailing one-element array members analogously to zero-length arrays.

The preferred mechanism to declare variable-length types like `struct line` above is the ISO C99 *flexible array member*, with slightly different syntax and semantics:

- Flexible array members are written as `contents[]` without the `0`.
- Flexible array members have incomplete type, and so the `sizeof` operator may not be applied. As a quirk of the original implementation of zero-length arrays, `sizeof` evaluates to zero.
- Flexible array members may only appear as the last member of a `struct` that is otherwise non-empty.
- A structure containing a flexible array member, or a union containing such a structure (possibly recursively), may not be a member of a structure or an element of an array. (However, these uses are permitted by GCC as extensions.)

Non-empty initialization of zero-length arrays is treated like any case where there are more initializer elements than the array holds, in that a suitable warning about "excess elements in array" is given, and the excess elements (all of them, in this case) are ignored.

GCC allows static initialization of flexible array members. This is equivalent to defining a new structure containing the original structure followed by an array of sufficient size to contain the data. E.g. in the following, `f1` is constructed as if it were declared like `f2`.

```
struct f1 {
  int x; int y[];
} f1 = { 1, { 2, 3, 4 } };

struct f2 {
  struct f1 f1; int data[3];
} f2 = { { 1 }, { 2, 3, 4 } };
```

The convenience of this extension is that `f1` has the desired type, eliminating the need to consistently refer to `f2.f1`.

This has symmetry with normal static arrays, in that an array of unknown size is also written with `[]`.

Of course, this extension only makes sense if the extra data comes at the end of a top-level object, as otherwise we would be overwriting data at subsequent offsets. To avoid undue complication and confusion with initialization of deeply nested arrays, we simply disallow any non-empty initialization except when the structure is the top-level object. For example:

```
struct foo { int x; int y[]; };
struct bar { struct foo z; };

struct foo a = { 1, { 2, 3, 4 } };        // Valid.
struct bar b = { { 1, { 2, 3, 4 } } };    // Invalid.
struct bar c = { { 1, { } } };            // Valid.
struct foo d[1] = { { 1, { 2, 3, 4 } } };  // Invalid.
```

6.19 Structures with No Members

GCC permits a C structure to have no members:

```
struct empty {
};
```

The structure has size zero. In C++, empty structures are part of the language. G++ treats empty structures as if they had a single member of type `char`.

6.20 Arrays of Variable Length

Variable-length automatic arrays are allowed in ISO C99, and as an extension GCC accepts them in C90 mode and in C++. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the block scope containing the declaration exits. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; you get an error message for it.

As an extension, GCC accepts variable-length arrays as a member of a structure or a union. For example:

```
void
foo (int n)
{
  struct S { int x[n]; };
}
```

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends, unless you also use `alloca` in this scope.

You can also use variable-length arrays as arguments to functions:

```
struct entry
tester (int len, char data[len][len])
{
  /* ... */
}
```

The length of an array is computed once when the storage is allocated and is remembered for the scope of the array in case you access it with `sizeof`.

If you want to pass the array first and the length afterward, you can use a forward declaration in the parameter list—another GNU extension.

```
struct entry
tester (int len; char data[len][len], int len)
{
  /* ... */
}
```

The '`int len`' before the semicolon is a *parameter forward declaration*, and it serves the purpose of making the name `len` known when the declaration of `data` is parsed.

You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type. ISO C99 does not support parameter forward declarations.

6.21 Macros with a Variable Number of Arguments.

In the ISO C standard of 1999, a macro can be declared to accept a variable number of arguments much as a function can. The syntax for defining the macro is similar to that of a function. Here is an example:

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
```

Here '`...`' is a *variable argument*. In the invocation of such a macro, it represents the zero or more tokens until the closing parenthesis that ends the invocation, including any commas. This set of tokens replaces the identifier `__VA_ARGS__` in the macro body wherever it appears. See the CPP manual for more information.

GCC has long supported variadic macros, and used a different syntax that allowed you to give a name to the variable arguments just like any other argument. Here is an example:

```
#define debug(format, args...) fprintf (stderr, format, args)
```

This is in all ways equivalent to the ISO C example above, but arguably more readable and descriptive.

GNU CPP has two further variadic macro extensions, and permits them to be used with either of the above forms of macro definition.

In standard C, you are not allowed to leave the variable argument out entirely; but you are allowed to pass an empty argument. For example, this invocation is invalid in ISO C, because there is no comma after the string:

```
debug ("A message")
```

GNU CPP permits you to completely omit the variable arguments in this way. In the above examples, the compiler would complain, though since the expansion of the macro still has the extra comma after the format string.

To help solve this problem, CPP behaves specially for variable arguments used with the token paste operator, '`##`'. If instead you write

```
#define debug(format, ...) fprintf (stderr, format, ## __VA_ARGS__)
```

and if the variable arguments are omitted or empty, the '`##`' operator causes the preprocessor to remove the comma before it. If you do provide some variable arguments in your macro invocation, GNU CPP does not complain about the paste operation and instead places the variable arguments after the comma. Just like any other pasted macro argument, these arguments are not macro expanded.

6.22 Slightly Looser Rules for Escaped Newlines

The preprocessor treatment of escaped newlines is more relaxed than that specified by the C90 standard, which requires the newline to immediately follow a backslash. GCC's implementation allows whitespace in the form of spaces, horizontal and vertical tabs, and form feeds between the backslash and the subsequent newline. The preprocessor issues a warning, but treats it as a valid escaped newline and combines the two lines to form a single logical line. This works within comments and tokens, as well as between tokens. Comments are *not* treated as whitespace for the purposes of this relaxation, since they have not yet been replaced with spaces.

6.23 Non-Lvalue Arrays May Have Subscripts

In ISO C99, arrays that are not lvalues still decay to pointers, and may be subscripted, although they may not be modified or used after the next sequence point and the unary '`&`' operator may not be applied to them. As an extension, GNU C allows such arrays to be subscripted in C90 mode, though otherwise they do not decay to pointers outside C99 mode. For example, this is valid in GNU C though not valid in C90:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

6.24 Arithmetic on `void`- and Function-Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The option '`-Wpointer-arith`' requests a warning if these extensions are used.

6.25 Pointer Arguments in Variadic Functions

Standard C requires that pointer types used with `va_arg` in functions with variable argument lists either must be compatible with that of the actual argument, or that one type must be a pointer to `void` and the other a pointer to a character type. GNU C implements the POSIX XSI extension that additionally permits the use of `va_arg` with a pointer type to receive arguments of any other pointer type.

In particular, in GNU C '`va_arg (ap, void *)`' can safely be used to consume an argument of any pointer type.

6.26 Pointers to Arrays with Qualifiers Work as Expected

In GNU C, pointers to arrays with qualifiers work similar to pointers to other qualified types. For example, a value of type `int (*)[5]` can be used to initialize a variable of type `const int (*)[5]`. These types are incompatible in ISO C because the `const` qualifier is formally attached to the element type of the array and not the array itself.

```
extern void
transpose (int N, int M, double out[M][N], const double in[N][M]);
double x[3][2];
double y[2][3];
...
transpose(3, 2, y, x);
```

6.27 Non-Constant Initializers

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  /* ... */
}
```

6.28 Compound Literals

A compound literal looks like a cast of a brace-enclosed aggregate initializer list. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. Unlike the result of a cast, a compound literal is an lvalue. ISO C99 and later support compound literals. As an extension, GCC supports compound literals also in C90 mode and in C++, although as explained below, the C++ semantics are somewhat different.

Usually, the specified type of a compound literal is a structure. Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a `struct foo` with a compound literal:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
```

```
}
```

You can also construct an array, though this is dangerous in C++, as explained below. If all the elements of the compound literal are (made up of) simple constant expressions suitable for use in initializers of objects of static storage duration, then the compound literal can be coerced to a pointer to its first element and used in such an initializer, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Compound literals for scalar types and union types are also allowed. In the following example the variable `i` is initialized to the value `2`, the result of incrementing the unnamed object created by the compound literal.

```
int i = ++(int) { 1 };
```

As a GNU extension, GCC allows initialization of objects with static storage duration by compound literals (which is not possible in ISO C99 because the initializer is not a constant). It is handled as if the object were initialized only with the brace-enclosed list if the types of the compound literal and the object match. The elements of the compound literal must be constant. If the object being initialized has array type of unknown size, the size is determined by the size of the compound literal.

```
static struct foo x = (struct foo) {1, 'a', 'b'};
static int y[] = (int []) {1, 2, 3};
static int z[] = (int [3]) {1};
```

The above lines are equivalent to the following:

```
static struct foo x = {1, 'a', 'b'};
static int y[] = {1, 2, 3};
static int z[] = {1, 0, 0};
```

In C, a compound literal designates an unnamed object with static or automatic storage duration. In C++, a compound literal designates a temporary object that only lives until the end of its full-expression. As a result, well-defined C code that takes the address of a subobject of a compound literal can be undefined in C++, so G++ rejects the conversion of a temporary array to a pointer. For instance, if the array compound literal example above appeared inside a function, any subsequent use of `foo` in C++ would have undefined behavior because the lifetime of the array ends after the declaration of `foo`.

As an optimization, G++ sometimes gives array compound literals longer lifetimes: when the array either appears outside a function or has a `const`-qualified type. If `foo` and its initializer had elements of type `char *const` rather than `char *`, or if `foo` were a global variable, the array would have static storage duration. But it is probably safest just to avoid the use of array compound literals in C++ code.

6.29 Designated Initializers

Standard C90 requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialized.

In ISO C99 you can give the elements in any order, specifying the array indices or structure field names they apply to, and GNU C allows this as an extension in C90 mode as well. This extension is not implemented in GNU C++.

To specify an array index, write '`[index] =`' before the element value. For example,

```
int a[6] = { [4] = 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values must be constant expressions, even if the array being initialized is automatic.

An alternative syntax for this that has been obsolete since GCC 2.5 but GCC still accepts is to write '`[index]`' before the element value, with no '`=`'.

To initialize a range of elements to the same value, write '`[first ... last] = value`'. This is a GNU extension. For example,

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

If the value in it has side effects, the side effects happen only once, not for each initialized field by the range initializer.

Note that the length of the array is the highest value specified plus one.

In a structure initializer, specify the name of a field to initialize with '`.fieldname =`' before the element value. For example, given the following structure,

```
struct point { int x, y; };
```

the following initialization

```
struct point p = { .y = yvalue, .x = xvalue };
```

is equivalent to

```
struct point p = { xvalue, yvalue };
```

Another syntax that has the same meaning, obsolete since GCC 2.5, is '`fieldname:`', as shown here:

```
struct point p = { y: yvalue, x: xvalue };
```

Omitted fields are implicitly initialized the same as for objects that have static storage duration.

The '`[index]`' or '`.fieldname`' is known as a *designator*. You can also use a designator (or the obsolete colon syntax) when initializing a union, to specify which element of the union should be used. For example,

```
union foo { int i; double d; };

union foo f = { .d = 4 };
```

converts 4 to a `double` to store it in the union using the second element. By contrast, casting 4 to type `union foo` stores it into the union as the integer `i`, since it is an integer. See Section 6.31 [Cast to Union], page 476.

You can combine this technique of naming elements with ordinary C initialization of successive elements. Each initializer element that does not have a designator applies to the next consecutive element of the array or structure. For example,

```
int a[6] = { [1] = v1, v2, [4] = v4 };
```

is equivalent to

```
int a[6] = { 0, v1, v2, 0, v4, 0 };
```

Labeling the elements of an array initializer is especially useful when the indices are characters or belong to an `enum` type. For example:

```
int whitespace[256]
  = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
      ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

You can also write a series of '`.fieldname`' and '`[index]`' designators before an '`=`' to specify a nested subobject to initialize; the list is taken relative to the subobject corresponding to the closest surrounding brace pair. For example, with the '`struct point`' declaration above:

```
struct point ptarray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };
```

If the same field is initialized multiple times, or overlapping fields of a union are initialized, the value from the last initialization is used. When a field of a union is itself a structure, the entire structure from the last field initialized is used. If any previous initializer has side effect, it is unspecified whether the side effect happens or not. Currently, GCC discards the side-effecting initializer expressions and issues a warning.

6.30 Case Ranges

You can specify a range of consecutive values in a single `case` label, like this:

```
case low ... high:
```

This has the same effect as the proper number of individual `case` labels, one for each integer value from *low* to *high*, inclusive.

This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

Be careful: Write spaces around the `...`, for otherwise it may be parsed wrong when you use it with integer values. For example, write this:

```
case 1 ... 5:
```

rather than this:

```
case 1...5:
```

6.31 Cast to a Union Type

A cast to a union type is a C extension not available in C++. It looks just like ordinary casts with the constraint that the type specified is a union type. You can specify the type either with the `union` keyword or with a `typedef` name that refers to a union. The result of a cast to a union is a temporary rvalue of the union type with a member whose type matches that of the operand initialized to the value of the operand. The effect of a cast to a union is similar to a compound literal except that it yields an rvalue like standard casts do. See Section 6.28 [Compound Literals], page 473.

Expressions that may be cast to the union type are those whose type matches at least one of the members of the union. Thus, given the following union and variables:

```
union foo { int i; double d; };
int x;
double y;
union foo z;
```

both `x` and `y` can be cast to type `union foo` and the following assignments

```
z = (union foo) x;
z = (union foo) y;
```

are shorthand equivalents of these

```
z = (union foo) { .i = x };
z = (union foo) { .d = y };
```

However, `(union foo) FLT_MAX;` is not a valid cast because the union has no member of type `float`.

Using the cast as the right-hand side of an assignment to a variable of union type is equivalent to storing in a member of the union with the same type

```
union foo u;
/* ... */
u = (union foo) x  ≡  u.i = x
u = (union foo) y  ≡  u.d = y
```

You can also use the union cast as a function argument:

```
void hack (union foo);
/* ... */
hack ((union foo) x);
```

6.32 Mixed Declarations and Code

ISO C99 and ISO C++ allow declarations and code to be freely mixed within compound statements. As an extension, GNU C also allows this in C90 mode. For example, you could do:

```
int i;
/* ... */
i++;
int j = i + 2;
```

Each identifier is visible from where it is declared until the end of the enclosing block.

6.33 Declaring Attributes of Functions

In GNU C and C++, you can use function attributes to specify certain function properties that may help the compiler optimize calls or check code more carefully for correctness. For example, you can use attributes to specify that a function never returns (`noreturn`), returns a value depending only on the values of its arguments (`const`), or has `printf`-style arguments (`format`).

You can also use attributes to control memory placement, code generation options or call/return conventions within the function being annotated. Many of these attributes are target-specific. For example, many targets support attributes for defining interrupt handler functions, which typically must follow special register usage and return conventions. Such attributes are described in the subsection for each target. However, a considerable number of attributes are supported by most, if not all targets. Those are described in the Section 6.33.1 [Common Function Attributes], page 478 section.

Function attributes are introduced by the `__attribute__` keyword in the declaration of a function, followed by an attribute specification enclosed in double parentheses. You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following one attribute specification with another. See Section 6.39 [Attribute Syntax], page 559, for the exact rules on attribute syntax and placement. Compatible attribute specifications on distinct declarations of the same function are merged. An attribute specification that is not compatible with attributes already applied to a declaration of the same function is ignored with a warning.

Some function attributes take one or more arguments that refer to the function's parameters by their positions within the function parameter list. Such attribute arguments are referred to as *positional arguments*. Unless specified otherwise, positional arguments that specify properties of parameters with pointer types can also specify the same properties of the implicit C++ `this` argument in non-static member functions, and of parameters of reference to a pointer type. For ordinary functions, position one refers to the first parameter on the list. In C++ non-static member functions, position one refers to the implicit `this` pointer. The same restrictions and effects apply to function attributes used with ordinary functions or C++ member functions.

GCC also supports attributes on variable declarations (see Section 6.34 [Variable Attributes], page 536), labels (see Section 6.36 [Label Attributes], page 557), enumerators (see Section 6.37 [Enumerator Attributes], page 558), statements (see Section 6.38 [Statement Attributes], page 559), and types (see Section 6.35 [Type Attributes], page 548).

There is some overlap between the purposes of attributes and pragmas (see Section 6.62 [Pragmas Accepted by GCC], page 801). It has been found convenient to use `__attribute__` to achieve a natural attachment of attributes to their corresponding declarations, whereas `#pragma` is of use for compatibility with other compilers or constructs that do not naturally form part of the grammar.

In addition to the attributes documented here, GCC plugins may provide their own attributes.

6.33.1 Common Function Attributes

The following attributes are supported on most targets.

`alias ("target")`

The `alias` attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () { /* Do something. */; }
void f () __attribute__ ((weak, alias ("__f")));
```

defines '`f`' to be a weak alias for '`__f`'. In C++, the mangled name for the target must be used. It is an error if '`__f`' is not defined in the same translation unit.

This attribute requires assembler and object file support, and may not be available on all targets.

`aligned`
`aligned (alignment)`

The `aligned` attribute specifies a minimum alignment for the first instruction of the function, measured in bytes. When specified, *alignment* must be an integer constant power of 2. Specifying no *alignment* argument implies the ideal alignment for the target. The `__alignof__` operator can be used to determine what that is (see Section 6.44 [Alignment], page 564). The attribute has no effect when a definition for the function is not provided in the same translation unit.

The attribute cannot be used to decrease the alignment of a function previously declared with a more restrictive alignment; only to increase it. Attempts to do otherwise are diagnosed. Some targets specify a minimum default alignment for functions that is greater than 1. On such targets, specifying a less restrictive

alignment is silently ignored. Using the attribute overrides the effect of the '`-falign-functions`' (see Section 3.10 [Optimize Options], page 125) option for this function.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in the system linker and/or object file format. On some systems, the linker is only able to arrange for functions to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) See your linker documentation for further information.

The `aligned` attribute can also be used for variables and fields (see Section 6.34 [Variable Attributes], page 536.)

`alloc_align (`*position*`)`

The `alloc_align` attribute may be applied to a function that returns a pointer and takes at least one argument of an integer or enumerated type. It indicates that the returned pointer is aligned on a boundary given by the function argument at *position*. Meaningful alignments are powers of 2 greater than one. GCC uses this information to improve pointer alignment analysis.

The function parameter denoting the allocated alignment is specified by one constant integer argument whose number is the argument of the attribute. Argument numbering starts at one.

For instance,

```
void* my_memalign (size_t, size_t) __attribute__ ((alloc_align (1)));
```

declares that `my_memalign` returns memory with minimum alignment given by parameter 1.

`alloc_size (`*position*`)`
`alloc_size (`*position-1*`, `*position-2*`)`

The `alloc_size` attribute may be applied to a function that returns a pointer and takes at least one argument of an integer or enumerated type. It indicates that the returned pointer points to memory whose size is given by the function argument at *position-1*, or by the product of the arguments at *position-1* and *position-2*. Meaningful sizes are positive values less than `PTRDIFF_MAX`. GCC uses this information to improve the results of `__builtin_object_size`.

The function parameter(s) denoting the allocated size are specified by one or two integer arguments supplied to the attribute. The allocated size is either the value of the single function argument specified or the product of the two function arguments specified. Argument numbering starts at one for ordinary functions, and at two for C++ non-static member functions.

For instance,

```
void* my_calloc (size_t, size_t) __attribute__ ((alloc_size (1, 2)));
void* my_realloc (void*, size_t) __attribute__ ((alloc_size (2)));
```

declares that `my_calloc` returns memory of the size given by the product of parameter 1 and 2 and that `my_realloc` returns memory of the size given by parameter 2.

`always_inline`

Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function independent of any

restrictions that otherwise apply to inlining. Failure to inline such a function is diagnosed as an error. Note that if such a function is called indirectly the compiler may or may not inline it depending on optimization level and a failure to inline an indirect call may or may not be diagnosed.

`artificial`
This attribute is useful for small inline wrappers that if possible should appear during debugging as a unit. Depending on the debug info format it either means marking the function as artificial or using the caller location for all instructions within the inlined body.

`assume_aligned (alignment)`
`assume_aligned (alignment, offset)`
The `assume_aligned` attribute may be applied to a function that returns a pointer. It indicates that the returned pointer is aligned on a boundary given by *alignment*. If the attribute has two arguments, the second argument is misalignment *offset*. Meaningful values of *alignment* are powers of 2 greater than one. Meaningful values of *offset* are greater than zero and less than *alignment*.

For instance

```
void* my_alloc1 (size_t) __attribute__((assume_aligned (16)));
void* my_alloc2 (size_t) __attribute__((assume_aligned (32, 8)));
```

declares that `my_alloc1` returns 16-byte aligned pointers and that `my_alloc2` returns a pointer whose value modulo 32 is equal to 8.

`cold` The `cold` attribute on functions is used to inform the compiler that the function is unlikely to be executed. The function is optimized for size rather than speed and on many targets it is placed into a special subsection of the text section so all cold functions appear close together, improving code locality of non-cold parts of program. The paths leading to calls of cold functions within code are marked as unlikely by the branch prediction mechanism. It is thus useful to mark functions used to handle unlikely conditions, such as `perror`, as cold to improve optimization of hot functions that do call marked functions in rare occasions.

When profile feedback is available, via '`-fprofile-use`', cold functions are automatically detected and this attribute is ignored.

`const` Calls to functions whose return value is not affected by changes to the observable state of the program and that have no observable effects on such state other than to return a value may lend themselves to optimizations such as common subexpression elimination. Declaring such functions with the `const` attribute allows GCC to avoid emitting some calls in repeated invocations of the function with the same argument values.

For example,

```
int square (int) __attribute__ ((const));
```

tells GCC that subsequent calls to function `square` with the same argument value can be replaced by the result of the first call regardless of the statements in between.

The `const` attribute prohibits a function from reading objects that affect its return value between successive invocations. However, functions declared with

the attribute can safely read objects that do not change their return value, such as non-volatile constants.

The `const` attribute imposes greater restrictions on a function's definition than the similar `pure` attribute. Declaring the same function with both the `const` and the `pure` attribute is diagnosed. Because a const function cannot have any observable side effects it does not make sense for it to return `void`. Declaring such a function is diagnosed.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const` if the pointed-to data might change between successive invocations of the function. In general, since a function cannot distinguish data that might change from data that cannot, const functions should never take pointer or, in C++, reference arguments. Likewise, a function that calls a non-const function usually must not be const itself.

`constructor`
`destructor`
`constructor (`*priority*`)`
`destructor (`*priority*`)`

The `constructor` attribute causes the function to be called automatically before execution enters `main ()`. Similarly, the `destructor` attribute causes the function to be called automatically after `main ()` completes or `exit ()` is called. Functions with these attributes are useful for initializing data that is used implicitly during the execution of the program.

On some targets the attributes also accept an integer argument to specify a priority to control the order in which constructor and destructor functions are run. A constructor with a smaller priority number runs before a constructor with a larger priority number; the opposite relationship holds for destructors. So, if you have a constructor that allocates a resource and a destructor that deallocates the same resource, both functions typically have the same priority. The priorities for constructor and destructor functions are the same as those specified for namespace-scope C++ objects (see Section 7.7 [C++ Attributes], page 821). However, at present, the order in which constructors for C++ objects with static storage duration and functions decorated with attribute `constructor` are invoked is unspecified. In mixed declarations, attribute `init_priority` can be used to impose a specific ordering.

Using the argument forms of the `constructor` and `destructor` attributes on targets where the feature is not supported is rejected with an error.

`copy`
`copy (`*function*`)`

The `copy` attribute applies the set of attributes with which *function* has been declared to the declaration of the function to which the attribute is applied. The attribute is designed for libraries that define aliases or function resolvers that are expected to specify the same set of attributes as their targets. The `copy` attribute can be used with functions, variables, or types. However, the kind of symbol to which the attribute is applied (either function or variable) must match the kind of symbol to which the argument refers. The `copy` attribute copies

only syntactic and semantic attributes but not attributes that affect a symbol's linkage or visibility such as `alias`, `visibility`, or `weak`. The `deprecated` attribute is also not copied. See Section 6.35.1 [Common Type Attributes], page 549. See Section 6.34.1 [Common Variable Attributes], page 536.

For example, the *StrongAlias* macro below makes use of the `alias` and `copy` attributes to define an alias named *alloc* for function *allocate* declared with attributes *alloc_size*, *malloc*, and *nothrow*. Thanks to the `__typeof__` operator the alias has the same type as the target function. As a result of the `copy` attribute the alias also shares the same attributes as the target.

```
#define StrongAlias(TagetFunc, AliasDecl)   \
  extern __typeof__ (TargetFunc) AliasDecl  \
    __attribute__ ((alias (#TargetFunc), copy (TargetFunc)));

extern __attribute__ ((alloc_size (1), malloc, nothrow))
  void* allocate (size_t);
StrongAlias (allocate, alloc);
```

`deprecated`
`deprecated (`*msg*`)`

The `deprecated` attribute results in a warning if the function is used anywhere in the source file. This is useful when identifying functions that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated function, to enable users to easily find further information about why the function is deprecated, or what they should do instead. Note that the warnings only occurs for uses:

```
int old_fn () __attribute__ ((deprecated));
int old_fn ();
int (*fn_ptr)() = old_fn;
```

results in a warning on line 3 but not line 2. The optional *msg* argument, which must be a string, is printed in the warning if present.

The `deprecated` attribute can also be used for variables and types (see Section 6.34 [Variable Attributes], page 536, see Section 6.35 [Type Attributes], page 548.)

The message attached to the attribute is affected by the setting of the '`-fmessage-length`' option.

`error ("`*message*`")`
`warning ("`*message*`")`

If the `error` or `warning` attribute is used on a function declaration and a call to such a function is not eliminated through dead code elimination or other optimizations, an error or warning (respectively) that includes *message* is diagnosed. This is useful for compile-time checking, especially together with `__builtin_constant_p` and inline functions where checking the inline function arguments is not possible through `extern char [(condition) ? 1 : -1];` tricks.

While it is possible to leave the function undefined and thus invoke a link failure (to define the function with a message in `.gnu.warning*` section), when using these attributes the problem is diagnosed earlier and with exact location of the call even in presence of inline functions or when not emitting debugging information.

`externally_visible`

This attribute, attached to a global variable or function, nullifies the effect of the '`-fwhole-program`' command-line option, so the object remains visible outside the current compilation unit.

If '`-fwhole-program`' is used together with '`-flto`' and `gold` is used as the linker plugin, `externally_visible` attributes are automatically added to functions (not variable yet due to a current `gold` issue) that are accessed outside of LTO objects according to resolution file produced by `gold`. For other linkers that cannot generate resolution file, explicit `externally_visible` attributes are still necessary.

`flatten` Generally, inlining into a function is limited. For a function marked with this attribute, every call inside this function is inlined, if possible. Functions declared with attribute `noinline` and similar are not inlined. Whether the function itself is considered for inlining depends on its size and the current inlining parameters.

`format (`*`archetype, string-index, first-to-check`*`)`

The `format` attribute specifies that a function takes `printf`, `scanf`, `strftime` or `strfmon` style arguments that should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter *archetype* determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime`, `gnu_printf`, `gnu_scanf`, `gnu_strftime` or `strfmon`. (You can also use `__printf__`, `__scanf__`, `__strftime__` or `__strfmon__`.) On MinGW targets, `ms_printf`, `ms_scanf`, and `ms_strftime` are also present. *archetype* values such as `printf` refer to the formats accepted by the system's C runtime library, while values prefixed with '`gnu_`' always refer to the formats accepted by the GNU C Library. On Microsoft Windows targets, values prefixed with '`ms_`' refer to the formats accepted by the '`msvcrt.dll`' library. The parameter *string-index* specifies which argument is the format string argument (starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency. For `strftime` formats, the third parameter is required to be zero. Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two, not one, when giving values for *string-index* and *first-to-check*.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions that take format strings as arguments, so that GCC can check the calls to these functions

for errors. The compiler always (unless '-ffreestanding' or '-fno-builtin' is used) checks formats for the standard library functions printf, fprintf, sprintf, scanf, fscanf, sscanf, strftime, vprintf, vfprintf and vsprintf whenever such warnings are requested (using '-Wformat'), so there is no need to modify the header file 'stdio.h'. In C99 mode, the functions snprintf, vsnprintf, vscanf, vfscanf and vsscanf are also checked. Except in strictly conforming C standard modes, the X/Open function strfmon is also checked as are printf_unlocked and fprintf_unlocked. See Section 3.4 [Options Controlling C Dialect], page 37.

For Objective-C dialects, NSString (or __NSString__) is recognized in the same context. Declarations including these format attributes are parsed for correct syntax, however the result of checking of such format strings is not yet defined, and is not carried out by this version of the compiler.

The target may also provide additional types of format checks. See Section 6.61 [Format Checks Specific to Particular Target Machines], page 801.

format_arg (*string-index*)

The format_arg attribute specifies that a function takes one or more format strings for a printf, scanf, strftime or strfmon style function and modifies it (for example, to translate it into another language), so the result can be passed to a printf, scanf, strftime or strfmon style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). Multiple format_arg attributes may be applied to the same function, each designating a distinct parameter as a format string. For example, the declaration:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

causes the compiler to check the arguments in calls to a printf, scanf, strftime or strfmon type function, whose format string argument is a call to the my_dgettext function, for consistency with the format string argument my_format. If the format_arg attribute had not been specified, all the compiler could tell in such calls to format functions would be that the format string argument is not constant; this would generate a warning when '-Wformat-nonliteral' is used, but the calls could not be checked without the attribute.

In calls to a function declared with more than one format_arg attribute, each with a distinct argument value, the corresponding actual function arguments are checked against all format strings designated by the attributes. This capability is designed to support the GNU ngettext family of functions.

The parameter *string-index* specifies which argument is the format string argument (starting from one). Since non-static C++ methods have an implicit this argument, the arguments of such methods should be counted from two.

The format_arg attribute allows you to identify your own functions that modify format strings, so that GCC can check the calls to printf, scanf, strftime or strfmon type function whose operands are a call to one of your own function. The compiler always treats gettext, dgettext, and dcgettext in this manner

except when strict ISO C support is requested by '`-ansi`' or an appropriate '`-std`' option, or '`-ffreestanding`' or '`-fno-builtin`' is used. See Section 3.4 [Options Controlling C Dialect], page 37.

For Objective-C dialects, the `format-arg` attribute may refer to an `NSString` reference for compatibility with the `format` attribute above.

The target may also allow additional types in `format-arg` attributes. See Section 6.61 [Format Checks Specific to Particular Target Machines], page 801.

`gnu_inline`

This attribute should be used with a function that is also declared with the `inline` keyword. It directs GCC to treat the function as if it were defined in gnu90 mode even when compiling in C99 or gnu99 mode.

If the function is declared `extern`, then this definition of the function is used only for inlining. In no case is the function compiled as a standalone function, not even if you take its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it. This has almost the effect of a macro. The way to use this is to put a function definition in a header file with this attribute, and put another copy of the function, without `extern`, in a library file. The definition in the header file causes most calls to the function to be inlined. If any uses of the function remain, they refer to the single copy in the library. Note that the two definitions of the functions need not be precisely the same, although if they do not have the same effect your program may behave oddly.

In C, if the function is neither `extern` nor `static`, then the function is compiled as a standalone function, as well as being inlined where possible.

This is how GCC traditionally handled functions declared `inline`. Since ISO C99 specifies a different semantics for `inline`, this function attribute is provided as a transition measure and as a useful feature in its own right. This attribute is available in GCC 4.1.3 and later. It is available if either of the preprocessor macros `__GNUC_GNU_INLINE__` or `__GNUC_STDC_INLINE__` are defined. See Section 6.45 [An Inline Function is As Fast As a Macro], page 564.

In C++, this attribute does not depend on `extern` in any way, but it still requires the `inline` keyword to enable its special behavior.

`hot`

The `hot` attribute on a function is used to inform the compiler that the function is a hot spot of the compiled program. The function is optimized more aggressively and on many targets it is placed into a special subsection of the text section so all hot functions appear close together, improving locality.

When profile feedback is available, via '`-fprofile-use`', hot functions are automatically detected and this attribute is ignored.

`ifunc ("`*`resolver`*`")`

The `ifunc` attribute is used to mark a function as an indirect function using the STT_GNU_IFUNC symbol type extension to the ELF standard. This allows the resolution of the symbol value to be determined dynamically at load time, and an optimized version of the routine to be selected for the particular processor or other system characteristics determined then. To use this attribute, first define

the implementation functions available, and a resolver function that returns a pointer to the selected implementation function. The implementation functions' declarations must match the API of the function being implemented. The resolver should be declared to be a function taking no arguments and returning a pointer to a function of the same type as the implementation. For example:

```
void *my_memcpy (void *dst, const void *src, size_t len)
{
  ...
  return dst;
}

static void * (*resolve_memcpy (void))(void *, const void *, size_t)
{
  return my_memcpy; // we will just always select this routine
}
```

The exported header file declaring the function the user calls would contain:

```
extern void *memcpy (void *, const void *, size_t);
```

allowing the user to call `memcpy` as a regular function, unaware of the actual implementation. Finally, the indirect function needs to be defined in the same translation unit as the resolver function:

```
void *memcpy (void *, const void *, size_t)
     __attribute__ ((ifunc ("resolve_memcpy")));
```

In C++, the `ifunc` attribute takes a string that is the mangled name of the resolver function. A C++ resolver for a non-static member function of class `C` should be declared to return a pointer to a non-member function taking pointer to `C` as the first argument, followed by the same arguments as of the implementation function. G++ checks the signatures of the two functions and issues a '`-Wattribute-alias`' warning for mismatches. To suppress a warning for the necessary cast from a pointer to the implementation member function to the type of the corresponding non-member function use the '`-Wno-pmf-conversions`' option. For example:

```
class S
{
private:
  int debug_impl (int);
  int optimized_impl (int);

  typedef int Func (S*, int);

  static Func* resolver ();
public:

  int interface (int);
};

int S::debug_impl (int) { /* ... */ }
int S::optimized_impl (int) { /* ... */ }

S::Func* S::resolver ()
{
  int (S::*pimpl) (int)
    = getenv ("DEBUG") ? &S::debug_impl : &S::optimized_impl;
```

```
  // Cast triggers -Wno-pmf-conversions.
  return reinterpret_cast<Func*>(pimpl);
}

int S::interface (int) __attribute__ ((ifunc ("_ZN1S8resolverEv")));
```

Indirect functions cannot be weak. Binutils version 2.20.1 or higher and GNU C Library version 2.11.1 are required to use this feature.

`interrupt`
`interrupt_handler`

Many GCC back ends support attributes to indicate that a function is an interrupt handler, which tells the compiler to generate function entry and exit sequences that differ from those from regular functions. The exact syntax and behavior are target-specific; refer to the following subsections for details.

`leaf`

Calls to external functions with this attribute must return to the current compilation unit only by return or by exception handling. In particular, a leaf function is not allowed to invoke callback functions passed to it from the current compilation unit, directly call functions exported by the unit, or `longjmp` into the unit. Leaf functions might still call functions from other compilation units and thus they are not necessarily leaf in the sense that they contain no function calls at all.

The attribute is intended for library functions to improve dataflow analysis. The compiler takes the hint that any data not escaping the current compilation unit cannot be used or modified by the leaf function. For example, the `sin` function is a leaf function, but `qsort` is not.

Note that leaf functions might indirectly run a signal handler defined in the current compilation unit that uses static variables. Similarly, when lazy symbol resolution is in effect, leaf functions might invoke indirect functions whose resolver function or implementation function is defined in the current compilation unit and uses static variables. There is no standard-compliant way to write such a signal handler, resolver function, or implementation function, and the best that you can do is to remove the `leaf` attribute or mark all such static variables `volatile`. Lastly, for ELF-based systems that support symbol interposition, care should be taken that functions defined in the current compilation unit do not unexpectedly interpose other symbols based on the defined standards mode and defined feature test macros; otherwise an inadvertent callback would be added.

The attribute has no effect on functions defined within the current compilation unit. This is to allow easy merging of multiple compilation units into one, for example, by using the link-time optimization. For this reason the attribute is not allowed on types to annotate indirect calls.

`malloc`

This tells the compiler that a function is `malloc`-like, i.e., that the pointer P returned by the function cannot alias any other pointer valid when the function returns, and moreover no pointers to valid objects occur in any storage addressed by P.

Using this attribute can improve optimization. Compiler predicts that a function with the attribute returns non-null in most cases. Functions like `malloc` and `calloc` have this property because they return a pointer to uninitialized or zeroed-out storage. However, functions like `realloc` do not have this property, as they can return a pointer to storage containing pointers.

`no_icf`
This function attribute prevents a functions from being merged with another semantically equivalent function.

`no_instrument_function`
If any of '`-finstrument-functions`', '`-p`', or '`-pg`' are given, profiling function calls are generated at entry and exit of most user-compiled functions. Functions with this attribute are not so instrumented.

`no_profile_instrument_function`
The `no_profile_instrument_function` attribute on functions is used to inform the compiler that it should not process any profile feedback based optimization code instrumentation.

`no_reorder`
Do not reorder functions or variables marked `no_reorder` against each other or top level assembler statements the executable. The actual order in the program will depend on the linker command line. Static variables marked like this are also not removed. This has a similar effect as the '`-fno-toplevel-reorder`' option, but only applies to the marked symbols.

`no_sanitize ("`*sanitize_option*`")`
The `no_sanitize` attribute on functions is used to inform the compiler that it should not do sanitization of all options mentioned in *sanitize_option*. A list of values acceptable by '`-fsanitize`' option can be provided.

```
void __attribute__ ((no_sanitize ("alignment", "object-size")))
f () { /* Do something. */; }
void __attribute__ ((no_sanitize ("alignment,object-size")))
g () { /* Do something. */; }
```

`no_sanitize_address`
`no_address_safety_analysis`
The `no_sanitize_address` attribute on functions is used to inform the compiler that it should not instrument memory accesses in the function when compiling with the '`-fsanitize=address`' option. The `no_address_safety_analysis` is a deprecated alias of the `no_sanitize_address` attribute, new code should use `no_sanitize_address`.

`no_sanitize_thread`
The `no_sanitize_thread` attribute on functions is used to inform the compiler that it should not instrument memory accesses in the function when compiling with the '`-fsanitize=thread`' option.

`no_sanitize_undefined`
The `no_sanitize_undefined` attribute on functions is used to inform the compiler that it should not check for undefined behavior in the function when compiling with the '`-fsanitize=undefined`' option.

`no_split_stack`
: If '`-fsplit-stack`' is given, functions have a small prologue which decides whether to split the stack. Functions with the `no_split_stack` attribute do not have that prologue, and thus may run with only a small amount of stack space available.

`no_stack_limit`
: This attribute locally overrides the '`-fstack-limit-register`' and '`-fstack-limit-symbol`' command-line options; it has the effect of disabling stack limit checking in the function it applies to.

`noclone`
: This function attribute prevents a function from being considered for cloning—a mechanism that produces specialized copies of functions and which is (currently) performed by interprocedural constant propagation.

`noinline`
: This function attribute prevents a function from being considered for inlining. If the function does not have side effects, there are optimizations other than inlining that cause function calls to be optimized away, although the function call is live. To keep such calls from being optimized away, put

```
asm ("");
```

(see Section 6.47.2 [Extended Asm], page 569) in the called function, to serve as a special side effect.

`noipa`
: Disable interprocedural optimizations between the function with this attribute and its callers, as if the body of the function is not available when optimizing callers and the callers are unavailable when optimizing the body. This attribute implies `noinline`, `noclone` and `no_icf` attributes. However, this attribute is not equivalent to a combination of other attributes, because its purpose is to suppress existing and future optimizations employing interprocedural analysis, including those that do not have an attribute suitable for disabling them individually. This attribute is supported mainly for the purpose of testing the compiler.

`nonnull`
`nonnull (`*`arg-index`*`, ...)`
: The `nonnull` attribute may be applied to a function that takes at least one argument of a pointer type. It indicates that the referenced arguments must be non-null pointers. For instance, the declaration:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
        __attribute__((nonnull (1, 2)));
```

causes the compiler to check that, in calls to `my_memcpy`, arguments *dest* and *src* are non-null. If the compiler determines that a null pointer is passed in an argument slot marked as non-null, and the '`-Wnonnull`' option is enabled, a warning is issued. See Section 3.8 [Warning Options], page 70. Unless disabled by the '`-fno-delete-null-pointer-checks`' option the compiler may also perform optimizations based on the knowledge that certain function arguments cannot be null. In addition, the '`-fisolate-erroneous-paths-attribute`' option can be specified to have GCC transform calls with null arguments to non-null functions into traps. See Section 3.10 [Optimize Options], page 125.

If no *arg-index* is given to the `nonnull` attribute, all pointer arguments are marked as non-null. To illustrate, the following declaration is equivalent to the previous example:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
        __attribute__((nonnull));
```

`noplt` The `noplt` attribute is the counterpart to option '`-fno-plt`'. Calls to functions marked with this attribute in position-independent code do not use the PLT.

```
/* Externally defined function foo.  */
int foo () __attribute__ ((noplt));

int
main (/* ... */)
{
  /* ... */
  foo ();
  /* ... */
}
```

The `noplt` attribute on function `foo` tells the compiler to assume that the function `foo` is externally defined and that the call to `foo` must avoid the PLT in position-independent code.

In position-dependent code, a few targets also convert calls to functions that are marked to not use the PLT to use the GOT instead.

`noreturn` A few standard library functions, such as `abort` and `exit`, cannot return. GCC knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
fatal (/* ... */)
{
  /* ... */ /* Print error message. */ /* ... */
  exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

The `noreturn` keyword does not affect the exceptional path when that applies: a `noreturn`-marked function may still return to the caller by throwing an exception or calling `longjmp`.

In order to preserve backtraces, GCC will never turn calls to `noreturn` functions into tail calls.

Do not assume that registers saved by the calling function are restored before calling the `noreturn` function.

It does not make sense for a `noreturn` function to have a return type other than `void`.

`nothrow` The `nothrow` attribute is used to inform the compiler that a function cannot throw an exception. For example, most functions in the standard C library can be guaranteed not to throw an exception with the notable exceptions of `qsort` and `bsearch` that take function pointer arguments.

`optimize (level, ...)`
`optimize (string, ...)`

The `optimize` attribute is used to specify that a function is to be compiled with different optimization options than specified on the command line. Valid arguments are constant non-negative integers and strings. Each numeric argument specifies an optimization *level*. Each *string* argument consists of one or more comma-separated substrings. Each substring that begins with the letter `O` refers to an optimization option such as '`-O0`' or '`-Os`'. Other substrings are taken as suffixes to the `-f` prefix jointly forming the name of an optimization option. See Section 3.10 [Optimize Options], page 125.

'`#pragma GCC optimize`' can be used to set optimization options for more than one function. See Section 6.62.15 [Function Specific Option Pragmas], page 808, for details about the pragma.

Providing multiple strings as arguments separated by commas to specify multiple options is equivalent to separating the option suffixes with a comma ('`,`') within a single string. Spaces are not permitted within the strings.

Not every optimization option that starts with the *-f* prefix specified by the attribute necessarily has an effect on the function. The `optimize` attribute should be used for debugging purposes only. It is not suitable in production code.

`patchable_function_entry`

In case the target's text segment can be made writable at run time by any means, padding the function entry with a number of NOPs can be used to provide a universal tool for instrumentation.

The `patchable_function_entry` function attribute can be used to change the number of NOPs to any desired value. The two-value syntax is the same as for the command-line switch '`-fpatchable-function-entry=N,M`', generating *N* NOPs, with the function entry point before the *M*th NOP instruction. *M* defaults to 0 if omitted e.g. function entry point is before the first NOP.

If patchable function entries are enabled globally using the command-line option '`-fpatchable-function-entry=N,M`', then you must disable instrumentation on all functions that are part of the instrumentation framework with the attribute `patchable_function_entry (0)` to prevent recursion.

`pure`

Calls to functions that have no observable effects on the state of the program other than to return a value may lend themselves to optimizations such as common subexpression elimination. Declaring such functions with the `pure` attribute allows GCC to avoid emitting some calls in repeated invocations of the function with the same argument values.

The `pure` attribute prohibits a function from modifying the state of the program that is observable by means other than inspecting the function's return value.

However, functions declared with the `pure` attribute can safely read any non-volatile objects, and modify the value of objects in a way that does not affect their return value or the observable state of the program.

For example,

```
int hash (char *) __attribute__ ((pure));
```

tells GCC that subsequent calls to the function `hash` with the same string can be replaced by the result of the first call provided the state of the program observable by `hash`, including the contents of the array itself, does not change in between. Even though `hash` takes a non-const pointer argument it must not modify the array it points to, or any other object whose value the rest of the program may depend on. However, the caller may safely change the contents of the array between successive calls to the function (doing so disables the optimization). The restriction also applies to member objects referenced by the `this` pointer in C++ non-static member functions.

Some common examples of pure functions are `strlen` or `memcmp`. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between consecutive calls (such as the standard C `feof` function in a multithreading environment).

The `pure` attribute imposes similar but looser restrictions on a function's definition than the `const` attribute: `pure` allows the function to read any non-volatile memory, even if it changes in between successive invocations of the function. Declaring the same function with both the `pure` and the `const` attribute is diagnosed. Because a pure function cannot have any observable side effects it does not make sense for such a function to return `void`. Declaring such a function is diagnosed.

`returns_nonnull`

The `returns_nonnull` attribute specifies that the function return value should be a non-null pointer. For instance, the declaration:

```
extern void *
mymalloc (size_t len) __attribute__((returns_nonnull));
```

lets the compiler optimize callers based on the knowledge that the return value will never be null.

`returns_twice`

The `returns_twice` attribute tells the compiler that a function may return more than one time. The compiler ensures that all registers are dead before calling such a function and emits a warning about the variables that may be clobbered after the second return from the function. Examples of such functions are `setjmp` and `vfork`. The `longjmp`-like counterpart of such function, if any, might need to be marked with the `noreturn` attribute.

`section ("`*`section-name`*`")`

Normally, the compiler places the code it generates in the `text` section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function `foobar` in the `bar` section.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`sentinel`
`sentinel (`*position*`)`

This function attribute indicates that an argument in a call to the function is expected to be an explicit `NULL`. The attribute is only valid on variadic functions. By default, the sentinel is expected to be the last argument of the function call. If the optional *position* argument is specified to the attribute, the sentinel must be located at *position* counting backwards from the end of the argument list.

```
__attribute__ ((sentinel))
is equivalent to
__attribute__ ((sentinel(0)))
```

The attribute is automatically set with a position of 0 for the built-in functions `execl` and `execlp`. The built-in function `execle` has the attribute set with a position of 1.

A valid `NULL` in this context is defined as zero with any object pointer type. If your system defines the `NULL` macro with an integer type then you need to add an explicit cast. During installation GCC replaces the system `<stddef.h>` header with a copy that redefines NULL appropriately.

The warnings for missing or incorrect sentinels are enabled with '`-Wformat`'.

`simd`
`simd("`*mask*`")`

This attribute enables creation of one or more function versions that can process multiple arguments using SIMD instructions from a single invocation. Specifying this attribute allows compiler to assume that such versions are available at link time (provided in the same or another translation unit). Generated versions are target-dependent and described in the corresponding Vector ABI document. For x86_64 target this document can be found here (`https://sourceware.org/glibc/wiki/libmvec?action=AttachFile&do=view&target=V`

The optional argument *mask* may have the value `notinbranch` or `inbranch`, and instructs the compiler to generate non-masked or masked clones correspondingly. By default, all clones are generated.

If the attribute is specified and `#pragma omp declare simd` is present on a declaration and the '`-fopenmp`' or '`-fopenmp-simd`' switch is specified, then the attribute is ignored.

`stack_protect`

This attribute adds stack protection code to the function if flags '`-fstack-protector`', '`-fstack-protector-strong`' or '`-fstack-protector-explicit`' are set.

`target (string, ...)`

Multiple target back ends implement the `target` attribute to specify that a function is to be compiled with different target options than specified on the command line. One or more strings can be provided as arguments. Each string consists of one or more comma-separated suffixes to the `-m` prefix jointly forming the name of a machine-dependent option. See Section 3.18 [Machine-Dependent Options], page 245.

The `target` attribute can be used for instance to have a function compiled with a different ISA (instruction set architecture) than the default. '`#pragma GCC target`' can be used to specify target-specific options for more than one function. See Section 6.62.15 [Function Specific Option Pragmas], page 808, for details about the pragma.

For instance, on an x86, you could declare one function with the `target("sse4.1,arch=core2")` attribute and another with `target("sse4a,arch=amdfam10")`. This is equivalent to compiling the first function with '`-msse4.1`' and '`-march=core2`' options, and the second function with '`-msse4a`' and '`-march=amdfam10`' options. It is up to you to make sure that a function is only invoked on a machine that supports the particular ISA it is compiled for (for example by using `cpuid` on x86 to determine what feature bits and architecture family are used).

```
int core2_func (void) __attribute__ ((__target__ ("arch=core2")));
int sse3_func (void) __attribute__ ((__target__ ("sse3")));
```

Providing multiple strings as arguments separated by commas to specify multiple options is equivalent to separating the option suffixes with a comma ('`,`') within a single string. Spaces are not permitted within the strings.

The options supported are specific to each target; refer to Section 6.33.35 [x86 Function Attributes], page 525, Section 6.33.25 [PowerPC Function Attributes], page 517, Section 6.33.5 [ARM Function Attributes], page 502, Section 6.33.2 [AArch64 Function Attributes], page 497, Section 6.33.23 [Nios II Function Attributes], page 516, and Section 6.33.29 [S/390 Function Attributes], page 522 for details.

`target_clones (options)`

The `target_clones` attribute is used to specify that a function be cloned into multiple versions compiled with different target options than specified on the command line. The supported options and restrictions are the same as for `target` attribute.

For instance, on an x86, you could compile a function with `target_clones("sse4.1,avx")`. GCC creates two function clones, one compiled with '`-msse4.1`' and another with '`-mavx`'.

On a PowerPC, you can compile a function with `target_clones("cpu=power9,default")`. GCC will create two function clones, one compiled with '`-mcpu=power9`' and another with the default options. GCC must be configured to use GLIBC 2.23 or newer in order to use the `target_clones` attribute.

It also creates a resolver function (see the `ifunc` attribute above) that dynamically selects a clone suitable for current architecture. The resolver is created only if there is a usage of a function with `target_clones` attribute.

`unused` This attribute, attached to a function, means that the function is meant to be possibly unused. GCC does not produce a warning for this function.

`used` This attribute, attached to a function, means that code must be emitted for the function even if it appears that the function is not referenced. This is useful, for example, when the function is referenced only in inline assembly.

When applied to a member function of a C++ class template, the attribute also means that the function is instantiated if the class itself is instantiated.

`visibility ("visibility_type")`

This attribute affects the linkage of the declaration to which it is attached. It can be applied to variables (see Section 6.34.1 [Common Variable Attributes], page 536) and types (see Section 6.35.1 [Common Type Attributes], page 549) as well as functions.

There are four supported *visibility_type* values: default, hidden, protected or internal visibility.

```
void __attribute__ ((visibility ("protected")))
f () { /* Do something. */; }
int i __attribute__ ((visibility ("hidden")));
```

The possible values of *visibility_type* correspond to the visibility settings in the ELF gABI.

`default` Default visibility is the normal case for the object file format. This value is available for the visibility attribute to override other options that may change the assumed visibility of entities.

On ELF, default visibility means that the declaration is visible to other modules and, in shared libraries, means that the declared entity may be overridden.

On Darwin, default visibility means that the declaration is visible to other modules.

Default visibility corresponds to "external linkage" in the language.

`hidden` Hidden visibility indicates that the entity declared has a new form of linkage, which we call "hidden linkage". Two declarations of an object with hidden linkage refer to the same object if they are in the same shared object.

`internal` Internal visibility is like hidden visibility, but with additional processor specific semantics. Unless otherwise specified by the psABI, GCC defines internal visibility to mean that a function is *never* called from another module. Compare this with hidden functions which, while they cannot be referenced directly by other modules, can be referenced indirectly via function pointers. By indicating that a function cannot be called from outside the module, GCC may for instance omit the load of a PIC register since it is known that the calling function loaded the correct value.

`protected`

Protected visibility is like default visibility except that it indicates that references within the defining module bind to the definition in that module. That is, the declared entity cannot be overridden by another module.

All visibilities are supported on many, but not all, ELF targets (supported when the assembler supports the '`.visibility`' pseudo-op). Default visibility is supported everywhere. Hidden visibility is supported on Darwin targets.

The visibility attribute should be applied only to declarations that would otherwise have external linkage. The attribute should be applied consistently, so that the same entity should not be declared with different settings of the attribute.

In C++, the visibility attribute applies to types as well as functions and objects, because in C++ types have linkage. A class must not have greater visibility than its non-static data member types and bases, and class members default to the visibility of their class. Also, a declaration without explicit visibility is limited to the visibility of its type.

In C++, you can mark member functions and static member variables of a class with the visibility attribute. This is useful if you know a particular method or static member variable should only be used from one shared object; then you can mark it hidden while the rest of the class has default visibility. Care must be taken to avoid breaking the One Definition Rule; for example, it is usually not useful to mark an inline method as hidden without marking the whole class as hidden.

A C++ namespace declaration can also have the visibility attribute.

```
namespace nspace1 __attribute__ ((visibility ("protected")))
{ /* Do something. */; }
```

This attribute applies only to the particular namespace body, not to other definitions of the same namespace; it is equivalent to using '`#pragma GCC visibility`' before and after the namespace definition (see Section 6.62.13 [Visibility Pragmas], page 807).

In C++, if a template argument has limited visibility, this restriction is implicitly propagated to the template instantiation. Otherwise, template instantiations and specializations default to the visibility of their template.

If both the template and enclosing class have explicit visibility, the visibility from the template is used.

`warn_unused_result`

The `warn_unused_result` attribute causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as `realloc`.

```
int fn () __attribute__ ((warn_unused_result));
int foo ()
{
  if (fn () < 0) return -1;
  fn ();
  return 0;
```

```
}
```

results in warning on line 5.

`weak` The `weak` attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions that can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

`weakref`
`weakref ("target")`

The `weakref` attribute marks a declaration as a weak reference. Without arguments, it should be accompanied by an `alias` attribute naming the target symbol. Optionally, the *target* may be given as an argument to `weakref` itself. In either case, `weakref` implicitly marks the declaration as `weak`. Without a *target*, given as an argument to `weakref` or to `alias`, `weakref` is equivalent to `weak`.

```
static int x() __attribute__ ((weakref ("y")));
/* is equivalent to... */
static int x() __attribute__ ((weak, weakref, alias ("y")));
/* and to... */
static int x() __attribute__ ((weakref));
static int x() __attribute__ ((alias ("y")));
```

A weak reference is an alias that does not by itself require a definition to be given for the target symbol. If the target symbol is only referenced through weak references, then it becomes a `weak` undefined symbol. If it is directly referenced, however, then such strong references prevail, and a definition is required for the symbol, not necessarily in the same translation unit.

The effect is equivalent to moving all references to the alias to a separate translation unit, renaming the alias to the aliased symbol, declaring it as weak, compiling the two separate translation units and performing a link with relocatable output (ie: `ld -r`) on them.

At present, a declaration to which `weakref` is attached can only be `static`.

6.33.2 AArch64 Function Attributes

The following target-specific function attributes are available for the AArch64 target. For the most part, these options mirror the behavior of similar command-line options (see Section 3.18.1 [AArch64 Options], page 245), but on a per-function basis.

`general-regs-only`

Indicates that no floating-point or Advanced SIMD registers should be used when generating code for this function. If the function explicitly uses floating-point code, then the compiler gives an error. This is the same behavior as that of the command-line option '`-mgeneral-regs-only`'.

`fix-cortex-a53-835769`

Indicates that the workaround for the Cortex-A53 erratum 835769 should be applied to this function. To explicitly disable the workaround for this function specify the negated form: `no-fix-cortex-a53-835769`. This corresponds to

the behavior of the command line options '-mfix-cortex-a53-835769' and '-mno-fix-cortex-a53-835769'.

cmodel= Indicates that code should be generated for a particular code model for this function. The behavior and permissible arguments are the same as for the command line option '-mcmodel='.

strict-align
no-strict-align
strict-align indicates that the compiler should not assume that unaligned memory references are handled by the system. To allow the compiler to assume that aligned memory references are handled by the system, the inverse attribute no-strict-align can be specified. The behavior is same as for the command-line option '-mstrict-align' and '-mno-strict-align'.

omit-leaf-frame-pointer
Indicates that the frame pointer should be omitted for a leaf function call. To keep the frame pointer, the inverse attribute no-omit-leaf-frame-pointer can be specified. These attributes have the same behavior as the command-line options '-momit-leaf-frame-pointer' and '-mno-omit-leaf-frame-pointer'.

tls-dialect=
Specifies the TLS dialect to use for this function. The behavior and permissible arguments are the same as for the command-line option '-mtls-dialect='.

arch= Specifies the architecture version and architectural extensions to use for this function. The behavior and permissible arguments are the same as for the '-march=' command-line option.

tune= Specifies the core for which to tune the performance of this function. The behavior and permissible arguments are the same as for the '-mtune=' command-line option.

cpu= Specifies the core for which to tune the performance of this function and also whose architectural features to use. The behavior and valid arguments are the same as for the '-mcpu=' command-line option.

sign-return-address
Select the function scope on which return address signing will be applied. The behavior and permissible arguments are the same as for the command-line option '-msign-return-address='. The default value is none. This attribute is deprecated. The branch-protection attribute should be used instead.

branch-protection
Select the function scope on which branch protection will be applied. The behavior and permissible arguments are the same as for the command-line option '-mbranch-protection='. The default value is none.

The above target attributes can be specified as follows:

```
__attribute__((target("attr-string")))
int
f (int a)
```

```
{
  return a + 5;
}
```

where *attr-string* is one of the attribute strings specified above.

Additionally, the architectural extension string may be specified on its own. This can be used to turn on and off particular architectural extensions without having to specify a particular architecture version or core. Example:

```
__attribute__((target("+crc+nocrypto")))
int
foo (int a)
{
  return a + 5;
}
```

In this example `target("+crc+nocrypto")` enables the `crc` extension and disables the `crypto` extension for the function `foo` without modifying an existing '`-march=`' or '`-mcpu`' option.

Multiple target function attributes can be specified by separating them with a comma. For example:

```
__attribute__((target("arch=armv8-a+crc+crypto,tune=cortex-a53")))
int
foo (int a)
{
  return a + 5;
}
```

is valid and compiles function `foo` for ARMv8-A with `crc` and `crypto` extensions and tunes it for `cortex-a53`.

6.33.2.1 Inlining rules

Specifying target attributes on individual functions or performing link-time optimization across translation units compiled with different target options can affect function inlining rules:

In particular, a caller function can inline a callee function only if the architectural features available to the callee are a subset of the features available to the caller. For example: A function `foo` compiled with '`-march=armv8-a+crc`', or tagged with the equivalent `arch=armv8-a+crc` attribute, can inline a function `bar` compiled with '`-march=armv8-a+nocrc`' because the all the architectural features that function `bar` requires are available to function `foo`. Conversely, function `bar` cannot inline function `foo`.

Additionally inlining a function compiled with '`-mstrict-align`' into a function compiled without `-mstrict-align` is not allowed. However, inlining a function compiled without '`-mstrict-align`' into a function compiled with '`-mstrict-align`' is allowed.

Note that CPU tuning options and attributes such as the '`-mcpu=`', '`-mtune=`' do not inhibit inlining unless the CPU specified by the '`-mcpu=`' option or the `cpu=` attribute conflicts with the architectural feature rules specified above.

6.33.3 AMD GCN Function Attributes

These function attributes are supported by the AMD GCN back end:

`amdgpu_hsa_kernel`

This attribute indicates that the corresponding function should be compiled as a kernel function, that is an entry point that can be invoked from the host via the HSA runtime library. By default functions are only callable only from other GCN functions.

This attribute is implicitly applied to any function named `main`, using default parameters.

Kernel functions may return an integer value, which will be written to a conventional place within the HSA "kernargs" region.

The attribute parameters configure what values are passed into the kernel function by the GPU drivers, via the initial register state. Some values are used by the compiler, and therefore forced on. Enabling other options may break assumptions in the compiler and/or run-time libraries.

`private_segment_buffer`
: Set `enable_sgpr_private_segment_buffer` flag. Always on (required to locate the stack).

`dispatch_ptr`
: Set `enable_sgpr_dispatch_ptr` flag. Always on (required to locate the launch dimensions).

`queue_ptr`
: Set `enable_sgpr_queue_ptr` flag. Always on (required to convert address spaces).

`kernarg_segment_ptr`
: Set `enable_sgpr_kernarg_segment_ptr` flag. Always on (required to locate the kernel arguments, "kernargs").

`dispatch_id`
: Set `enable_sgpr_dispatch_id` flag.

`flat_scratch_init`
: Set `enable_sgpr_flat_scratch_init` flag.

`private_segment_size`
: Set `enable_sgpr_private_segment_size` flag.

`grid_workgroup_count_X`
: Set `enable_sgpr_grid_workgroup_count_x` flag. Always on (required to use OpenACC/OpenMP).

`grid_workgroup_count_Y`
: Set `enable_sgpr_grid_workgroup_count_y` flag.

`grid_workgroup_count_Z`
: Set `enable_sgpr_grid_workgroup_count_z` flag.

`workgroup_id_X`
: Set `enable_sgpr_workgroup_id_x` flag.

`workgroup_id_Y`
: Set `enable_sgpr_workgroup_id_y` flag.

`workgroup_id_Z`
: Set `enable_sgpr_workgroup_id_z` flag.

`workgroup_info`
: Set `enable_sgpr_workgroup_info` flag.

`private_segment_wave_offset`
: Set `enable_sgpr_private_segment_wave_byte_offset` flag. Always on (required to locate the stack).

`work_item_id_X`
: Set `enable_vgpr_workitem_id` parameter. Always on (can't be disabled).

`work_item_id_Y`
: Set `enable_vgpr_workitem_id` parameter. Always on (required to enable vectorization.)

`work_item_id_Z`
: Set `enable_vgpr_workitem_id` parameter. Always on (required to use OpenACC/OpenMP).

6.33.4 ARC Function Attributes

These function attributes are supported by the ARC back end:

`interrupt`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

On the ARC, you must specify the kind of interrupt to be handled in a parameter to the interrupt attribute like this:

```
void f () __attribute__ ((interrupt ("ilink1")));
```

Permissible values for this parameter are: `ilink1` and `ilink2`.

`long_call`
`medium_call`
`short_call`
: These attributes specify how a particular function is called. These attributes override the '`-mlong-calls`' and '`-mmedium-calls`' (see Section 3.18.4 [ARC Options], page 254) command-line switches and `#pragma long_calls` settings.

For ARC, a function marked with the `long_call` attribute is always called using register-indirect jump-and-link instructions, thereby enabling the called function to be placed anywhere within the 32-bit address space. A function marked with the `medium_call` attribute will always be close enough to be called with an unconditional branch-and-link instruction, which has a 25-bit offset from the call site. A function marked with the `short_call` attribute will always be close enough to be called with a conditional branch-and-link instruction, which has a 21-bit offset from the call site.

`jli_always`
: Forces a particular function to be called using `jli` instruction. The `jli` instruction makes use of a table stored into `.jlitab` section, which holds the location of the functions which are addressed using this instruction.

`jli_fixed`
: Identical like the above one, but the location of the function in the `jli` table is known and given as an attribute parameter.

`secure_call`
: This attribute allows one to mark secure-code functions that are callable from normal mode. The location of the secure call function into the `sjli` table needs to be passed as argument.

6.33.5 ARM Function Attributes

These function attributes are supported for ARM targets:

`general-regs-only`
: Indicates that no floating-point or Advanced SIMD registers should be used when generating code for this function. If the function explicitly uses floating-point code, then the compiler gives an error. This is the same behavior as that of the command-line option '`-mgeneral-regs-only`'.

`interrupt`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

  You can specify the kind of interrupt to be handled by adding an optional parameter to the interrupt attribute like this:

```
void f () __attribute__ ((interrupt ("IRQ")));
```

  Permissible values for this parameter are: `IRQ`, `FIQ`, `SWI`, `ABORT` and `UNDEF`.

  On ARMv7-M the interrupt type is ignored, and the attribute means the function may be called with a word-aligned stack pointer.

`isr`
: Use this attribute on ARM to write Interrupt Service Routines. This is an alias to the `interrupt` attribute above.

`long_call`
`short_call`
: These attributes specify how a particular function is called. These attributes override the '`-mlong-calls`' (see Section 3.18.5 [ARM Options], page 263) command-line switch and `#pragma long_calls` settings. For ARM, the `long_call` attribute indicates that the function might be far away from the call site and require a different (more expensive) calling sequence. The `short_call` attribute always places the offset to the function from the call site into the '`BL`' instruction directly.

`naked`
: This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler.

Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`pcs`

The `pcs` attribute can be used to control the calling convention used for a function on ARM. The attribute takes an argument that specifies the calling convention to use.

When compiling using the AAPCS ABI (or a variant of it) then valid values for the argument are `"aapcs"` and `"aapcs-vfp"`. In order to use a variant other than `"aapcs"` then the compiler must be permitted to use the appropriate co-processor registers (i.e., the VFP registers must be available in order to use `"aapcs-vfp"`). For example,

```
/* Argument passed in r0, and result returned in r0+r1.  */
double f2d (float) __attribute__((pcs("aapcs")));
```

Variadic functions always use the `"aapcs"` calling convention and the compiler rejects attempts to specify an alternative.

`target (`*options*`)`

As discussed in Section 6.33.1 [Common Function Attributes], page 478, this attribute allows specification of target-specific compilation options.

On ARM, the following options are allowed:

'`thumb`' Force code generation in the Thumb (T16/T32) ISA, depending on the architecture level.

'`arm`' Force code generation in the ARM (A32) ISA.

Functions from different modes can be inlined in the caller's mode.

'`fpu=`' Specifies the fpu for which to tune the performance of this function. The behavior and permissible arguments are the same as for the '`-mfpu=`' command-line option.

'`arch=`' Specifies the architecture version and architectural extensions to use for this function. The behavior and permissible arguments are the same as for the '`-march=`' command-line option.

The above target attributes can be specified as follows:

```
__attribute__((target("arch=armv8-a+crc")))
int
f (int a)
{
  return a + 5;
}
```

Additionally, the architectural extension string may be specified on its own. This can be used to turn on and off particular architectural extensions without having to specify a particular architecture version or core. Example:

```
__attribute__((target("+crc+nocrypto")))
int
```

```
foo (int a)
{
  return a + 5;
}
```

In this example `target("+crc+nocrypto")` enables the `crc` extension and disables the `crypto` extension for the function `foo` without modifying an existing '`-march=`' or '`-mcpu`' option.

6.33.6 AVR Function Attributes

These function attributes are supported by the AVR back end:

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

On the AVR, the hardware globally disables interrupts when an interrupt is executed. The first instruction of an interrupt handler declared with this attribute is a `SEI` instruction to re-enable interrupts. See also the `signal` function attribute that does not insert a `SEI` instruction. If both `signal` and `interrupt` are specified for the same function, `signal` is silently ignored.

`naked`

This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`no_gccisr`

Do not use `__gcc_isr` pseudo instructions in a function with the `interrupt` or `signal` attribute aka. interrupt service routine (ISR). Use this attribute if the preamble of the ISR prologue should always read

```
push  __zero_reg__
push  __tmp_reg__
in    __tmp_reg__, __SREG__
push  __tmp_reg__
clr   __zero_reg__
```

and accordingly for the postamble of the epilogue — no matter whether the mentioned registers are actually used in the ISR or not. Situations where you might want to use this attribute include:

- Code that (effectively) clobbers bits of `SREG` other than the `I`-flag by writing to the memory location of `SREG`.
- Code that uses inline assembler to jump to a different function which expects (parts of) the prologue code as outlined above to be present.

To disable `__gcc_isr` generation for the whole compilation unit, there is option '`-mno-gas-isr-prologues`', see Section 3.18.6 [AVR Options], page 277.

`OS_main`
`OS_task` On AVR, functions with the `OS_main` or `OS_task` attribute do not save/restore any call-saved register in their prologue/epilogue.

The `OS_main` attribute can be used when there *is guarantee* that interrupts are disabled at the time when the function is entered. This saves resources when the stack pointer has to be changed to set up a frame for local variables.

The `OS_task` attribute can be used when there is *no guarantee* that interrupts are disabled at that time when the function is entered like for, e.g. task functions in a multi-threading operating system. In that case, changing the stack pointer register is guarded by save/clear/restore of the global interrupt enable flag.

The differences to the `naked` function attribute are:

- `naked` functions do not have a return instruction whereas `OS_main` and `OS_task` functions have a `RET` or `RETI` return instruction.
- `naked` functions do not set up a frame for local variables or a frame pointer whereas `OS_main` and `OS_task` do this as needed.

`signal` Use this attribute on the AVR to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

See also the `interrupt` function attribute.

The AVR hardware globally disables interrupts when an interrupt is executed. Interrupt handler functions defined with the `signal` attribute do not re-enable interrupts. It is save to enable interrupts in a `signal` handler. This "save" only applies to the code generated by the compiler and not to the IRQ layout of the application which is responsibility of the application.

If both `signal` and `interrupt` are specified for the same function, `signal` is silently ignored.

6.33.7 Blackfin Function Attributes

These function attributes are supported by the Blackfin back end:

`exception_handler`
Use this attribute on the Blackfin to indicate that the specified function is an exception handler. The compiler generates function entry and exit sequences suitable for use in an exception handler when this attribute is present.

`interrupt_handler`
Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`kspisusp` When used together with `interrupt_handler`, `exception_handler` or `nmi_handler`, code is generated to load the stack pointer from the USP register in the function prologue.

`l1_text` This attribute specifies a function to be placed into L1 Instruction SRAM. The function is put into a specific section named `.l1.text`. With '`-mfdpic`', function calls with a such function as the callee or caller uses inlined PLT.

`l2` This attribute specifies a function to be placed into L2 SRAM. The function is put into a specific section named `.l2.text`. With '`-mfdpic`', callers of such functions use an inlined PLT.

`longcall`
`shortcall`
The `longcall` attribute indicates that the function might be far away from the call site and require a different (more expensive) calling sequence. The `shortcall` attribute indicates that the function is always close enough for the shorter calling sequence to be used. These attributes override the '`-mlongcall`' switch.

`nesting` Use this attribute together with `interrupt_handler`, `exception_handler` or `nmi_handler` to indicate that the function entry code should enable nested interrupts or exceptions.

`nmi_handler`
Use this attribute on the Blackfin to indicate that the specified function is an NMI handler. The compiler generates function entry and exit sequences suitable for use in an NMI handler when this attribute is present.

`saveall` Use this attribute to indicate that all registers except the stack pointer should be saved in the prologue regardless of whether they are used or not.

6.33.8 CR16 Function Attributes

These function attributes are supported by the CR16 back end:

`interrupt`
Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

6.33.9 C-SKY Function Attributes

These function attributes are supported by the C-SKY back end:

`interrupt`
`isr` Use these attributes to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when either of these attributes are present.

Use of these options requires the '`-mistack`' command-line option to enable support for the necessary interrupt stack instructions. They are ignored with a warning otherwise. See Section 3.18.11 [C-SKY Options], page 292.

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

6.33.10 Epiphany Function Attributes

These function attributes are supported by the Epiphany back end:

`disinterrupt`
: This attribute causes the compiler to emit instructions to disable interrupts for the duration of the given function.

`forwarder_section`
: This attribute modifies the behavior of an interrupt handler. The interrupt handler may be in external memory which cannot be reached by a branch instruction, so generate a local memory trampoline to transfer control. The single parameter identifies the section where the trampoline is placed.

`interrupt`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. It may also generate a special section with code to initialize the interrupt vector table.

  On Epiphany targets one or more optional parameters can be added like this:

```
void __attribute__ ((interrupt ("dma0, dma1"))) universal_dma_handler ();
```

  Permissible values for these parameters are: `reset`, `software_exception`, `page_miss`, `timer0`, `timer1`, `message`, `dma0`, `dma1`, `wand` and `swi`. Multiple parameters indicate that multiple entries in the interrupt vector table should be initialized for this function, i.e. for each parameter *name*, a jump to the function is emitted in the section ivt_entry_*name*. The parameter(s) may be omitted entirely, in which case no interrupt vector table entry is provided.

  Note that interrupts are enabled inside the function unless the `disinterrupt` attribute is also specified.

  The following examples are all valid uses of these attributes on Epiphany targets:

```
void __attribute__ ((interrupt)) universal_handler ();
void __attribute__ ((interrupt ("dma1"))) dma1_handler ();
void __attribute__ ((interrupt ("dma0, dma1")))
  universal_dma_handler ();
void __attribute__ ((interrupt ("timer0"), disinterrupt))
  fast_timer_handler ();
void __attribute__ ((interrupt ("dma0, dma1"),
                     forwarder_section ("tramp")))
  external_dma_handler ();
```

`long_call`
`short_call`
: These attributes specify how a particular function is called. These attributes override the '`-mlong-calls`' (see Section 3.18.2 [Adapteva Epiphany Options], page 251) command-line switch and `#pragma long_calls` settings.

6.33.11 H8/300 Function Attributes

These function attributes are available for H8/300 targets:

`function_vector`
: Use this attribute on the H8/300, H8/300H, and H8S to indicate that the specified function should be called through the function vector. Calling a function through the function vector reduces code size; however, the function vector has a limited size (maximum 128 entries on the H8/300 and 64 entries on the H8/300H and H8S) and shares space with the interrupt vector.

`interrupt_handler`
: Use this attribute on the H8/300, H8/300H, and H8S to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`saveall`
: Use this attribute on the H8/300, H8/300H, and H8S to indicate that all registers except the stack pointer should be saved in the prologue regardless of whether they are used or not.

6.33.12 IA-64 Function Attributes

These function attributes are supported on IA-64 targets:

`syscall_linkage`
: This attribute is used to modify the IA-64 calling convention by marking all input registers as live at all function exits. This makes it possible to restart a system call after an interrupt without having to save/restore the input registers. This also prevents kernel data from leaking into application code.

`version_id`
: This IA-64 HP-UX attribute, attached to a global variable or function, renames a symbol to contain a version string, thus allowing for function level versioning. HP-UX system header files may use function level versioning for some system calls.

```
extern int foo () __attribute__((version_id ("20040821")));
```

Calls to `foo` are mapped to calls to `foo{20040821}`.

6.33.13 M32C Function Attributes

These function attributes are supported by the M32C back end:

`bank_switch`
: When added to an interrupt handler with the M32C port, causes the prologue and epilogue to use bank switching to preserve the registers rather than saving them on the stack.

`fast_interrupt`
: Use this attribute on the M32C port to indicate that the specified function is a fast interrupt handler. This is just like the `interrupt` attribute, except that `freit` is used to return instead of `reit`.

`function_vector`
: On M16C/M32C targets, the `function_vector` attribute declares a special page subroutine call function. Use of this attribute reduces the code size by 2

bytes for each call generated to the subroutine. The argument to the attribute is the vector number entry from the special page vector table which contains the 16 low-order bits of the subroutine's entry address. Each vector table has special page number (18 to 255) that is used in `jsrs` instructions. Jump addresses of the routines are generated by adding 0x0F0000 (in case of M16C targets) or 0xFF0000 (in case of M32C targets), to the 2-byte addresses set in the vector table. Therefore you need to ensure that all the special page vector routines should get mapped within the address range 0x0F0000 to 0x0FFFFF (for M16C) and 0xFF0000 to 0xFFFFFF (for M32C).

In the following example 2 bytes are saved for each call to function `foo`.

```
void foo (void) __attribute__((function_vector(0x18)));
void foo (void)
{
}

void bar (void)
{
    foo();
}
```

If functions are defined in one file and are called in another file, then be sure to write this declaration in both files.

This attribute is ignored for R8C target.

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

6.33.14 M32R/D Function Attributes

These function attributes are supported by the M32R/D back end:

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`model (`*`model-name`*`)`

On the M32R/D, use this attribute to set the addressability of an object, and of the code generated for a function. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and are callable with the `bl` instruction.

Medium model objects may live anywhere in the 32-bit address space (the compiler generates `seth/add3` instructions to load their addresses), and are callable with the `bl` instruction.

Large model objects may live anywhere in the 32-bit address space (the compiler generates `seth/add3` instructions to load their addresses), and may not

be reachable with the `bl` instruction (the compiler generates the much slower `seth/add3/jl` instruction sequence).

6.33.15 m68k Function Attributes

These function attributes are supported by the m68k back end:

`interrupt`
`interrupt_handler`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. Either name may be used.

`interrupt_thread`
: Use this attribute on fido, a subarchitecture of the m68k, to indicate that the specified function is an interrupt handler that is designed to run as a thread. The compiler omits generate prologue/epilogue sequences and replaces the return instruction with a `sleep` instruction. This attribute is available only on fido.

6.33.16 MCORE Function Attributes

These function attributes are supported by the MCORE back end:

`naked`
: This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

6.33.17 MeP Function Attributes

These function attributes are supported by the MeP back end:

`disinterrupt`
: On MeP targets, this attribute causes the compiler to emit instructions to disable interrupts for the duration of the given function.

`interrupt`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`near`
: This attribute causes the compiler to assume the called function is close enough to use the normal calling convention, overriding the '`-mtf`' command-line option.

`far`
: On MeP targets this causes the compiler to use a calling convention that assumes the called function is too far away for the built-in addressing modes.

`vliw`
: The `vliw` attribute tells the compiler to emit instructions in VLIW mode instead of core mode. Note that this attribute is not allowed unless a VLIW coprocessor has been configured and enabled through command-line options.

6.33.18 MicroBlaze Function Attributes

These function attributes are supported on MicroBlaze targets:

`save_volatiles`

Use this attribute to indicate that the function is an interrupt handler. All volatile registers (in addition to non-volatile registers) are saved in the function prologue. If the function is a leaf function, only volatiles used by the function are saved. A normal function return is generated instead of a return from interrupt.

`break_handler`

Use this attribute to indicate that the specified function is a break handler. The compiler generates function entry and exit sequences suitable for use in an break handler when this attribute is present. The return from `break_handler` is done through the `rtbd` instead of `rtsd`.

```
void f () __attribute__ ((break_handler));
```

`interrupt_handler`
`fast_interrupt`

These attributes indicate that the specified function is an interrupt handler. Use the `fast_interrupt` attribute to indicate handlers used in low-latency interrupt mode, and `interrupt_handler` for interrupts that do not use low-latency handlers. In both cases, GCC emits appropriate prologue code and generates a return from the handler using `rtid` instead of `rtsd`.

6.33.19 Microsoft Windows Function Attributes

The following attributes are available on Microsoft Windows and Symbian OS targets.

`dllexport`

On Microsoft Windows targets and Symbian OS targets the `dllexport` attribute causes the compiler to provide a global pointer to a pointer in a DLL, so that it can be referenced with the `dllimport` attribute. On Microsoft Windows targets, the pointer name is formed by combining `_imp__` and the function or variable name.

You can use `__declspec(dllexport)` as a synonym for `__attribute__ ((dllexport))` for compatibility with other compilers.

On systems that support the `visibility` attribute, this attribute also implies "default" visibility. It is an error to explicitly specify any other visibility.

GCC's default behavior is to emit all inline functions with the `dllexport` attribute. Since this can cause object file-size bloat, you can use '`-fno-keep-inline-dllexport`', which tells GCC to ignore the attribute for inlined functions unless the '`-fkeep-inline-functions`' flag is used instead.

The attribute is ignored for undefined symbols.

When applied to C++ classes, the attribute marks defined non-inlined member functions and static data members as exports. Static consts initialized in-class are not marked unless they are also defined out-of-class.

For Microsoft Windows targets there are alternative methods for including the symbol in the DLL's export table such as using a '`.def`' file with an `EXPORTS` section or, with GNU ld, using the '`--export-all`' linker flag.

`dllimport`

On Microsoft Windows and Symbian OS targets, the `dllimport` attribute causes the compiler to reference a function or variable via a global pointer to a pointer that is set up by the DLL exporting the symbol. The attribute implies `extern`. On Microsoft Windows targets, the pointer name is formed by combining `_imp__` and the function or variable name.

You can use `__declspec(dllimport)` as a synonym for `__attribute__ ((dllimport))` for compatibility with other compilers.

On systems that support the `visibility` attribute, this attribute also implies "default" visibility. It is an error to explicitly specify any other visibility.

Currently, the attribute is ignored for inlined functions. If the attribute is applied to a symbol *definition*, an error is reported. If a symbol previously declared `dllimport` is later defined, the attribute is ignored in subsequent references, and a warning is emitted. The attribute is also overridden by a subsequent declaration as `dllexport`.

When applied to C++ classes, the attribute marks non-inlined member functions and static data members as imports. However, the attribute is ignored for virtual methods to allow creation of vtables using thunks.

On the SH Symbian OS target the `dllimport` attribute also has another affect—it can cause the vtable and run-time type information for a class to be exported. This happens when the class has a dllimported constructor or a non-inline, non-pure virtual function and, for either of those two conditions, the class also has an inline constructor or destructor and has a key function that is defined in the current translation unit.

For Microsoft Windows targets the use of the `dllimport` attribute on functions is not necessary, but provides a small performance benefit by eliminating a thunk in the DLL. The use of the `dllimport` attribute on imported variables can be avoided by passing the '`--enable-auto-import`' switch to the GNU linker. As with functions, using the attribute for a variable eliminates a thunk in the DLL.

One drawback to using this attribute is that a pointer to a *variable* marked as `dllimport` cannot be used as a constant address. However, a pointer to a *function* with the `dllimport` attribute can be used as a constant initializer; in this case, the address of a stub function in the import lib is referenced. On Microsoft Windows targets, the attribute can be disabled for functions by setting the '`-mnop-fun-dllimport`' flag.

6.33.20 MIPS Function Attributes

These function attributes are supported by the MIPS back end:

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable

for use in an interrupt handler when this attribute is present. An optional argument is supported for the interrupt attribute which allows the interrupt mode to be described. By default GCC assumes the external interrupt controller (EIC) mode is in use, this can be explicitly set using `eic`. When interrupts are non-masked then the requested Interrupt Priority Level (IPL) is copied to the current IPL which has the effect of only enabling higher priority interrupts. To use vectored interrupt mode use the argument `vector=[sw0|sw1|hw0|hw1|hw2|hw3|hw4|hw5]`, this will change the behavior of the non-masked interrupt support and GCC will arrange to mask all interrupts from sw0 up to and including the specified interrupt vector.

You can use the following attributes to modify the behavior of an interrupt handler:

`use_shadow_register_set`
: Assume that the handler uses a shadow register set, instead of the main general-purpose registers. An optional argument `intstack` is supported to indicate that the shadow register set contains a valid stack pointer.

`keep_interrupts_masked`
: Keep interrupts masked for the whole function. Without this attribute, GCC tries to reenable interrupts for as much of the function as it can.

`use_debug_exception_return`
: Return using the `deret` instruction. Interrupt handlers that don't have this attribute return using `eret` instead.

You can use any combination of these attributes, as shown below:

```
void __attribute__ ((interrupt)) v0 ();
void __attribute__ ((interrupt, use_shadow_register_set)) v1 ();
void __attribute__ ((interrupt, keep_interrupts_masked)) v2 ();
void __attribute__ ((interrupt, use_debug_exception_return)) v3 ();
void __attribute__ ((interrupt, use_shadow_register_set,
                     keep_interrupts_masked)) v4 ();
void __attribute__ ((interrupt, use_shadow_register_set,
                     use_debug_exception_return)) v5 ();
void __attribute__ ((interrupt, keep_interrupts_masked,
                     use_debug_exception_return)) v6 ();
void __attribute__ ((interrupt, use_shadow_register_set,
                     keep_interrupts_masked,
                     use_debug_exception_return)) v7 ();
void __attribute__ ((interrupt("eic"))) v8 ();
void __attribute__ ((interrupt("vector=hw3"))) v9 ();
```

`long_call`
`short_call`
`near`
`far`
: These attributes specify how a particular function is called on MIPS. The attributes override the '`-mlong-calls`' (see Section 3.18.28 [MIPS Options], page 326) command-line switch. The `long_call` and `far` attributes are synonyms, and cause the compiler to always call the function by first loading

its address into a register, and then using the contents of that register. The `short_call` and `near` attributes are synonyms, and have the opposite effect; they specify that non-PIC calls should be made using the more efficient `jal` instruction.

`mips16`
`nomips16`

On MIPS targets, you can use the `mips16` and `nomips16` function attributes to locally select or turn off MIPS16 code generation. A function with the `mips16` attribute is emitted as MIPS16 code, while MIPS16 code generation is disabled for functions with the `nomips16` attribute. These attributes override the '`-mips16`' and '`-mno-mips16`' options on the command line (see Section 3.18.28 [MIPS Options], page 326).

When compiling files containing mixed MIPS16 and non-MIPS16 code, the preprocessor symbol `__mips16` reflects the setting on the command line, not that within individual functions. Mixed MIPS16 and non-MIPS16 code may interact badly with some GCC extensions such as `__builtin_apply` (see Section 6.6 [Constructing Calls], page 457).

`micromips, MIPS`
`nomicromips, MIPS`

On MIPS targets, you can use the `micromips` and `nomicromips` function attributes to locally select or turn off microMIPS code generation. A function with the `micromips` attribute is emitted as microMIPS code, while microMIPS code generation is disabled for functions with the `nomicromips` attribute. These attributes override the '`-mmicromips`' and '`-mno-micromips`' options on the command line (see Section 3.18.28 [MIPS Options], page 326).

When compiling files containing mixed microMIPS and non-microMIPS code, the preprocessor symbol `__mips_micromips` reflects the setting on the command line, not that within individual functions. Mixed microMIPS and non-microMIPS code may interact badly with some GCC extensions such as `__builtin_apply` (see Section 6.6 [Constructing Calls], page 457).

`nocompression`

On MIPS targets, you can use the `nocompression` function attribute to locally turn off MIPS16 and microMIPS code generation. This attribute overrides the '`-mips16`' and '`-mmicromips`' options on the command line (see Section 3.18.28 [MIPS Options], page 326).

6.33.21 MSP430 Function Attributes

These function attributes are supported by the MSP430 back end:

`critical` Critical functions disable interrupts upon entry and restore the previous interrupt state upon exit. Critical functions cannot also have the `naked`, `reentrant` or `interrupt` attributes.

The MSP430 hardware ensures that interrupts are disabled on entry to `interrupt` functions, and restores the previous interrupt state on exit. The `critical` attribute is therefore redundant on `interrupt` functions.

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

You can provide an argument to the interrupt attribute which specifies a name or number. If the argument is a number it indicates the slot in the interrupt vector table (0 - 31) to which this handler should be assigned. If the argument is a name it is treated as a symbolic name for the vector slot. These names should match up with appropriate entries in the linker script. By default the names `watchdog` for vector 26, `nmi` for vector 30 and `reset` for vector 31 are recognized.

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`reentrant`

Reentrant functions disable interrupts upon entry and enable them upon exit. Reentrant functions cannot also have the `naked` or `critical` attributes. They can have the `interrupt` attribute.

`wakeup` This attribute only applies to interrupt functions. It is silently ignored if applied to a non-interrupt function. A wakeup interrupt function will rouse the processor from any low-power state that it might be in when the function exits.

`lower`
`upper`
`either` On the MSP430 target these attributes can be used to specify whether the function or variable should be placed into low memory, high memory, or the placement should be left to the linker to decide. The attributes are only significant if compiling for the MSP430X architecture.

The attributes work in conjunction with a linker script that has been augmented to specify where to place sections with a `.lower` and a `.upper` prefix. So, for example, as well as placing the `.data` section, the script also specifies the placement of a `.lower.data` and a `.upper.data` section. The intention is that `lower` sections are placed into a small but easier to access memory region and the upper sections are placed into a larger, but slower to access, region.

The `either` attribute is special. It tells the linker to place the object into the corresponding `lower` section if there is room for it. If there is insufficient room then the object is placed into the corresponding `upper` section instead. Note that the placement algorithm is not very sophisticated. It does not attempt to find an optimal packing of the `lower` sections. It just makes one pass over the objects and does the best that it can. Using the '`-ffunction-sections`' and '`-fdata-sections`' command-line options can help the packing, however, since they produce smaller, easier to pack regions.

6.33.22 NDS32 Function Attributes

These function attributes are supported by the NDS32 back end:

`exception`
: Use this attribute on the NDS32 target to indicate that the specified function is an exception handler. The compiler will generate corresponding sections for use in an exception handler.

`interrupt`
: On NDS32 target, this attribute indicates that the specified function is an interrupt handler. The compiler generates corresponding sections for use in an interrupt handler. You can use the following attributes to modify the behavior:

  `nested` This interrupt service routine is interruptible.

  `not_nested`
  : This interrupt service routine is not interruptible.

  `nested_ready`
  : This interrupt service routine is interruptible after `PSW.GIE` (global interrupt enable) is set. This allows interrupt service routine to finish some short critical code before enabling interrupts.

  `save_all` The system will help save all registers into stack before entering interrupt handler.

  `partial_save`
  : The system will help save caller registers into stack before entering interrupt handler.

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`reset` Use this attribute on the NDS32 target to indicate that the specified function is a reset handler. The compiler will generate corresponding sections for use in a reset handler. You can use the following attributes to provide extra exception handling:

  `nmi` Provide a user-defined function to handle NMI exception.

  `warm` Provide a user-defined function to handle warm reset exception.

6.33.23 Nios II Function Attributes

These function attributes are supported by the Nios II back end:

`target (`*`options`*`)`
: As discussed in Section 6.33.1 [Common Function Attributes], page 478, this attribute allows specification of target-specific compilation options.

  When compiling for Nios II, the following options are allowed:

'`custom-insn=N`'
'`no-custom-insn`'
Each '`custom-insn=N`' attribute locally enables use of a custom instruction with encoding *N* when generating code that uses *insn*. Similarly, '`no-custom-insn`' locally inhibits use of the custom instruction *insn*. These target attributes correspond to the '`-mcustom-insn=N`' and '`-mno-custom-insn`' command-line options, and support the same set of *insn* keywords. See Section 3.18.34 [Nios II Options], page 346, for more information.

'`custom-fpu-cfg=name`'
This attribute corresponds to the '`-mcustom-fpu-cfg=name`' command-line option, to select a predefined set of custom instructions named *name*. See Section 3.18.34 [Nios II Options], page 346, for more information.

6.33.24 Nvidia PTX Function Attributes

These function attributes are supported by the Nvidia PTX back end:

`kernel` This attribute indicates that the corresponding function should be compiled as a kernel function, which can be invoked from the host via the CUDA RT library. By default functions are only callable only from other PTX functions.

Kernel functions must have `void` return type.

6.33.25 PowerPC Function Attributes

These function attributes are supported by the PowerPC back end:

`longcall`
`shortcall`
The `longcall` attribute indicates that the function might be far away from the call site and require a different (more expensive) calling sequence. The `shortcall` attribute indicates that the function is always close enough for the shorter calling sequence to be used. These attributes override both the '`-mlongcall`' switch and the `#pragma longcall` setting.

See Section 3.18.42 [RS/6000 and PowerPC Options], page 358, for more information on whether long calls are necessary.

`target (options)`
As discussed in Section 6.33.1 [Common Function Attributes], page 478, this attribute allows specification of target-specific compilation options.

On the PowerPC, the following options are allowed:

'`altivec`'
'`no-altivec`'
Generate code that uses (does not use) AltiVec instructions. In 32-bit code, you cannot enable AltiVec instructions unless '`-mabi=altivec`' is used on the command line.

'cmpb'
'no-cmpb' Generate code that uses (does not use) the compare bytes instruction implemented on the POWER6 processor and other processors that support the PowerPC V2.05 architecture.

'dlmzb'
'no-dlmzb'
Generate code that uses (does not use) the string-search 'dlmzb' instruction on the IBM 405, 440, 464 and 476 processors. This instruction is generated by default when targeting those processors.

'fprnd'
'no-fprnd'
Generate code that uses (does not use) the FP round to integer instructions implemented on the POWER5+ processor and other processors that support the PowerPC V2.03 architecture.

'hard-dfp'
'no-hard-dfp'
Generate code that uses (does not use) the decimal floating-point instructions implemented on some POWER processors.

'isel'
'no-isel' Generate code that uses (does not use) ISEL instruction.

'mfcrf'
'no-mfcrf'
Generate code that uses (does not use) the move from condition register field instruction implemented on the POWER4 processor and other processors that support the PowerPC V2.01 architecture.

'mfpgpr'
'no-mfpgpr'
Generate code that uses (does not use) the FP move to/from general purpose register instructions implemented on the POWER6X processor and other processors that support the extended PowerPC V2.05 architecture.

'mulhw'
'no-mulhw'
Generate code that uses (does not use) the half-word multiply and multiply-accumulate instructions on the IBM 405, 440, 464 and 476 processors. These instructions are generated by default when targeting those processors.

'multiple'
'no-multiple'
Generate code that uses (does not use) the load multiple word instructions and the store multiple word instructions.

'update'
'no-update'
Generate code that uses (does not use) the load or store instructions that update the base register to the address of the calculated memory location.

'popcntb'
'no-popcntb'
Generate code that uses (does not use) the popcount and double-precision FP reciprocal estimate instruction implemented on the POWER5 processor and other processors that support the PowerPC V2.02 architecture.

'popcntd'
'no-popcntd'
Generate code that uses (does not use) the popcount instruction implemented on the POWER7 processor and other processors that support the PowerPC V2.06 architecture.

'powerpc-gfxopt'
'no-powerpc-gfxopt'
Generate code that uses (does not use) the optional PowerPC architecture instructions in the Graphics group, including floating-point select.

'powerpc-gpopt'
'no-powerpc-gpopt'
Generate code that uses (does not use) the optional PowerPC architecture instructions in the General Purpose group, including floating-point square root.

'recip-precision'
'no-recip-precision'
Assume (do not assume) that the reciprocal estimate instructions provide higher-precision estimates than is mandated by the PowerPC ABI.

'string'
'no-string'
Generate code that uses (does not use) the load string instructions and the store string word instructions to save multiple registers and do small block moves.

'vsx'
'no-vsx' Generate code that uses (does not use) vector/scalar (VSX) instructions, and also enable the use of built-in functions that allow more direct access to the VSX instruction set. In 32-bit code, you cannot enable VSX or AltiVec instructions unless '-mabi=altivec' is used on the command line.

'friz'
'no-friz' Generate (do not generate) the friz instruction when the '-funsafe-math-optimizations' option is used to optimize rounding a floating-point value to 64-bit integer and back to floating point. The friz instruction does not return the same value if the floating-point number is too large to fit in an integer.

'avoid-indexed-addresses'
'no-avoid-indexed-addresses'
Generate code that tries to avoid (not avoid) the use of indexed load or store instructions.

'paired'
'no-paired'
Generate code that uses (does not use) the generation of PAIRED simd instructions.

'longcall'
'no-longcall'
Generate code that assumes (does not assume) that all calls are far away so that a longer more expensive calling sequence is required.

'cpu=*CPU*' Specify the architecture to generate code for when compiling the function. If you select the target("cpu=power7") attribute when generating 32-bit code, VSX and AltiVec instructions are not generated unless you use the '-mabi=altivec' option on the command line.

'tune=*TUNE*'
Specify the architecture to tune for when compiling the function. If you do not specify the target("tune=*TUNE*") attribute and you do specify the target("cpu=*CPU*") attribute, compilation tunes for the *CPU* architecture, and not the default tuning specified on the command line.

On the PowerPC, the inliner does not inline a function that has different target options than the caller, unless the callee has a subset of the target options of the caller.

6.33.26 RISC-V Function Attributes

These function attributes are supported by the RISC-V back end:

naked This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic asm statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended asm or a mixture of basic asm and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

You can specify the kind of interrupt to be handled by adding an optional parameter to the interrupt attribute like this:

```
void f (void) __attribute__ ((interrupt ("user")));
```

Permissible values for this parameter are `user`, `supervisor`, and `machine`. If there is no parameter, then it defaults to `machine`.

6.33.27 RL78 Function Attributes

These function attributes are supported by the RL78 back end:

`interrupt`
`brk_interrupt`

These attributes indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

Use `brk_interrupt` instead of `interrupt` for handlers intended to be used with the `BRK` opcode (i.e. those that must end with `RETB` instead of `RETI`).

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

6.33.28 RX Function Attributes

These function attributes are supported by the RX back end:

`fast_interrupt`

Use this attribute on the RX port to indicate that the specified function is a fast interrupt handler. This is just like the `interrupt` attribute, except that `freit` is used to return instead of `reit`.

`interrupt`

Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

On RX and RL78 targets, you may specify one or more vector numbers as arguments to the attribute, as well as naming an alternate table name. Parameters are handled sequentially, so one handler can be assigned to multiple entries in multiple tables. One may also pass the magic string `"$default"` which causes the function to be used for any unfilled slots in the current table.

This example shows a simple assignment of a function to one vector in the default table (note that preprocessor macros may be used for chip-specific symbolic vector names):

```
void __attribute__ ((interrupt (5))) txd1_handler ();
```

This example assigns a function to two slots in the default table (using preprocessor macros defined elsewhere) and makes it the default for the `dct` table:

```
void __attribute__ ((interrupt (RXD1_VECT,RXD2_VECT,"dct","$default")))
txd1_handler ();
```

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`vector` This RX attribute is similar to the `interrupt` attribute, including its parameters, but does not make the function an interrupt-handler type function (i.e. it retains the normal C function calling ABI). See the `interrupt` attribute for a description of its arguments.

6.33.29 S/390 Function Attributes

These function attributes are supported on the S/390:

`hotpatch (`*`halfwords-before-function-label,halfwords-after-function-label`*`)`

On S/390 System z targets, you can use this function attribute to make GCC generate a "hot-patching" function prologue. If the '`-mhotpatch=`' command-line option is used at the same time, the `hotpatch` attribute takes precedence. The first of the two arguments specifies the number of halfwords to be added before the function label. A second argument can be used to specify the number of halfwords to be added after the function label. For both arguments the maximum allowed value is 1000000.

If both arguments are zero, hotpatching is disabled.

`target (`*`options`*`)`

As discussed in Section 6.33.1 [Common Function Attributes], page 478, this attribute allows specification of target-specific compilation options.

On S/390, the following options are supported:

'`arch=`'

'`tune=`'

'stack-guard='
'stack-size='
'branch-cost='
'warn-framesize='
'backchain'
'no-backchain'
'hard-dfp'
'no-hard-dfp'
'hard-float'
'soft-float'
'htm'
'no-htm'

'vx'
'no-vx'

'packed-stack'
'no-packed-stack'
'small-exec'
'no-small-exec'
'mvcle'
'no-mvcle'
'warn-dynamicstack'
'no-warn-dynamicstack'

The options work exactly like the S/390 specific command line options (without the prefix '-m') except that they do not change any feature macros. For example,

```
target("no-vx")
```

does not undefine the __VEC__ macro.

6.33.30 SH Function Attributes

These function attributes are supported on the SH family of processors:

`function_vector`

On SH2A targets, this attribute declares a function to be called using the TBR relative addressing mode. The argument to this attribute is the entry number of the same function in a vector table containing all the TBR relative addressable functions. For correct operation the TBR must be setup accordingly to point to the start of the vector table before any functions with this attribute are invoked. Usually a good place to do the initialization is the startup routine. The TBR relative vector table can have at max 256 function entries. The jumps to these functions are generated using a SH2A specific, non delayed branch instruction JSR/N @(disp8,TBR). You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

In an application, for a function being called once, this attribute saves at least 8 bytes of code; and if other successive calls are being made to the same function, it saves 2 bytes of code per each of these calls.

`interrupt_handler`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`nosave_low_regs`
: Use this attribute on SH targets to indicate that an `interrupt_handler` function should not save and restore registers R0..R7. This can be used on SH3* and SH4* targets that have a second R0..R7 register bank for non-reentrant interrupt handlers.

`renesas`
: On SH targets this attribute specifies that the function or struct follows the Renesas ABI.

`resbank`
: On the SH2A target, this attribute enables the high-speed register saving and restoration using a register bank for `interrupt_handler` routines. Saving to the bank is performed automatically after the CPU accepts an interrupt that uses a register bank.

  The nineteen 32-bit registers comprising general register R0 to R14, control register GBR, and system registers MACH, MACL, and PR and the vector table address offset are saved into a register bank. Register banks are stacked in first-in last-out (FILO) sequence. Restoration from the bank is executed by issuing a RESBANK instruction.

`sp_switch`
: Use this attribute on the SH to indicate an `interrupt_handler` function should switch to an alternate stack. It expects a string argument that names a global variable holding the address of the alternate stack.

```
void *alt_stack;
void f () __attribute__ ((interrupt_handler,
                          sp_switch ("alt_stack")));
```

`trap_exit`
: Use this attribute on the SH for an `interrupt_handler` to return using `trapa` instead of `rte`. This attribute expects an integer argument specifying the trap number to be used.

`trapa_handler`
: On SH targets this function attribute is similar to `interrupt_handler` but it does not save and restore all registers.

6.33.31 SPU Function Attributes

These function attributes are supported by the SPU back end:

`naked`
: This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

6.33.32 Symbian OS Function Attributes

See Section 6.33.19 [Microsoft Windows Function Attributes], page 511, for discussion of the `dllexport` and `dllimport` attributes.

6.33.33 V850 Function Attributes

The V850 back end supports these function attributes:

`interrupt`
`interrupt_handler`
: Use these attributes to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when either attribute is present.

6.33.34 Visium Function Attributes

These function attributes are supported by the Visium back end:

`interrupt`
: Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

6.33.35 x86 Function Attributes

These function attributes are supported by the x86 back end:

`cdecl`
: On the x86-32 targets, the `cdecl` attribute causes the compiler to assume that the calling function pops off the stack space used to pass arguments. This is useful to override the effects of the '`-mrtd`' switch.

`fastcall`
: On x86-32 targets, the `fastcall` attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

`thiscall`
: On x86-32 targets, the `thiscall` attribute causes the compiler to pass the first argument (if of integral type) in the register ECX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack. The `thiscall` attribute is intended for C++ non-static member functions. As a GCC extension, this calling convention can be used for C functions and for static member methods.

`ms_abi`
`sysv_abi`
: On 32-bit and 64-bit x86 targets, you can use an ABI attribute to indicate which calling convention should be used for a function. The `ms_abi` attribute tells the compiler to use the Microsoft ABI, while the `sysv_abi` attribute tells the compiler to use the ABI used on GNU/Linux and other systems. The default is to use the Microsoft ABI when targeting Windows. On all other systems, the default is the x86/AMD ABI.

Note, the `ms_abi` attribute for Microsoft Windows 64-bit targets currently requires the '`-maccumulate-outgoing-args`' option.

`callee_pop_aggregate_return (number)`
On x86-32 targets, you can use this attribute to control how aggregates are returned in memory. If the caller is responsible for popping the hidden pointer together with the rest of the arguments, specify *number* equal to zero. If callee is responsible for popping the hidden pointer, specify *number* equal to one.

The default x86-32 ABI assumes that the callee pops the stack for hidden pointer. However, on x86-32 Microsoft Windows targets, the compiler assumes that the caller pops the stack for hidden pointer.

`ms_hook_prologue`
On 32-bit and 64-bit x86 targets, you can use this function attribute to make GCC generate the "hot-patching" function prologue used in Win32 API functions in Microsoft Windows XP Service Pack 2 and newer.

`naked` This attribute allows the compiler to construct the requisite function declaration, while allowing the body of the function to be assembly code. The specified function will not have prologue/epilogue sequences generated by the compiler. Only basic `asm` statements can safely be included in naked functions (see Section 6.47.1 [Basic Asm], page 567). While using extended `asm` or a mixture of basic `asm` and C code may appear to work, they cannot be depended upon to work reliably and are not supported.

`regparm (number)`
On x86-32 targets, the `regparm` attribute causes the compiler to pass arguments number one to *number* if they are of integral type in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments continue to be passed all of their arguments on the stack.

Beware that on some ELF systems this attribute is unsuitable for global functions in shared libraries with lazy binding (which is the default). Lazy binding sends the first call via resolving code in the loader, which might assume EAX, EDX and ECX can be clobbered, as per the standard calling conventions. Solaris 8 is affected by this. Systems with the GNU C Library version 2.1 or higher and FreeBSD are believed to be safe since the loaders there save EAX, EDX and ECX. (Lazy binding can be disabled with the linker or the loader if desired, to avoid the problem.)

`sseregparm`
On x86-32 targets with SSE support, the `sseregparm` attribute causes the compiler to pass up to 3 floating-point arguments in SSE registers instead of on the stack. Functions that take a variable number of arguments continue to pass all of their floating-point arguments on the stack.

`force_align_arg_pointer`
On x86 targets, the `force_align_arg_pointer` attribute may be applied to individual function definitions, generating an alternate prologue and epilogue that realigns the run-time stack if necessary. This supports mixing legacy codes

that run with a 4-byte aligned stack with modern codes that keep a 16-byte stack for SSE compatibility.

`stdcall` On x86-32 targets, the `stdcall` attribute causes the compiler to assume that the called function pops off the stack space used to pass arguments, unless it takes a variable number of arguments.

`no_caller_saved_registers`
Use this attribute to indicate that the specified function has no caller-saved registers. That is, all registers are callee-saved. For example, this attribute can be used for a function called from an interrupt handler. The compiler generates proper function entry and exit sequences to save and restore any modified registers, except for the EFLAGS register. Since GCC doesn't preserve SSE, MMX nor x87 states, the GCC option '`-mgeneral-regs-only`' should be used to compile functions with `no_caller_saved_registers` attribute.

`interrupt`
Use this attribute to indicate that the specified function is an interrupt handler or an exception handler (depending on parameters passed to the function, explained further). The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present. The `IRET` instruction, instead of the `RET` instruction, is used to return from interrupt handlers. All registers, except for the EFLAGS register which is restored by the `IRET` instruction, are preserved by the compiler. Since GCC doesn't preserve SSE, MMX nor x87 states, the GCC option '`-mgeneral-regs-only`' should be used to compile interrupt and exception handlers.

Any interruptible-without-stack-switch code must be compiled with '`-mno-red-zone`' since interrupt handlers can and will, because of the hardware design, touch the red zone.

An interrupt handler must be declared with a mandatory pointer argument:

```
struct interrupt_frame;

__attribute__ ((interrupt))
void
f (struct interrupt_frame *frame)
{
}
```

and you must define `struct interrupt_frame` as described in the processor's manual.

Exception handlers differ from interrupt handlers because the system pushes an error code on the stack. An exception handler declaration is similar to that for an interrupt handler, but with a different mandatory function signature. The compiler arranges to pop the error code off the stack before the `IRET` instruction.

```
#ifdef __x86_64__
typedef unsigned long long int uword_t;
#else
typedef unsigned int uword_t;
#endif

struct interrupt_frame;
```

```
__attribute__ ((interrupt))
void
f (struct interrupt_frame *frame, uword_t error_code)
{
  ...
}
```

Exception handlers should only be used for exceptions that push an error code; you should use an interrupt handler in other cases. The system will crash if the wrong kind of handler is used.

`target (options)`

As discussed in Section 6.33.1 [Common Function Attributes], page 478, this attribute allows specification of target-specific compilation options.

On the x86, the following options are allowed:

'`3dnow`'
'`no-3dnow`'
: Enable/disable the generation of the 3DNow! instructions.

'`3dnowa`'
'`no-3dnowa`'
: Enable/disable the generation of the enhanced 3DNow! instructions.

'`abm`'
'`no-abm`' Enable/disable the generation of the advanced bit instructions.

'`adx`'
'`no-adx`' Enable/disable the generation of the ADX instructions.

'`aes`'
'`no-aes`' Enable/disable the generation of the AES instructions.

'`avx`'
'`no-avx`' Enable/disable the generation of the AVX instructions.

'`avx2`'
'`no-avx2`' Enable/disable the generation of the AVX2 instructions.

'`avx5124fmaps`'
'`no-avx5124fmaps`'
: Enable/disable the generation of the AVX5124FMAPS instructions.

'`avx5124vnniw`'
'`no-avx5124vnniw`'
: Enable/disable the generation of the AVX5124VNNIW instructions.

'`avx512bitalg`'
'`no-avx512bitalg`'
: Enable/disable the generation of the AVX512BITALG instructions.

‘avx512bw’
‘no-avx512bw’
Enable/disable the generation of the AVX512BW instructions.

‘avx512cd’
‘no-avx512cd’
Enable/disable the generation of the AVX512CD instructions.

‘avx512dq’
‘no-avx512dq’
Enable/disable the generation of the AVX512DQ instructions.

‘avx512er’
‘no-avx512er’
Enable/disable the generation of the AVX512ER instructions.

‘avx512f’
‘no-avx512f’
Enable/disable the generation of the AVX512F instructions.

‘avx512ifma’
‘no-avx512ifma’
Enable/disable the generation of the AVX512IFMA instructions.

‘avx512pf’
‘no-avx512pf’
Enable/disable the generation of the AVX512PF instructions.

‘avx512vbmi’
‘no-avx512vbmi’
Enable/disable the generation of the AVX512VBMI instructions.

‘avx512vbmi2’
‘no-avx512vbmi2’
Enable/disable the generation of the AVX512VBMI2 instructions.

‘avx512vl’
‘no-avx512vl’
Enable/disable the generation of the AVX512VL instructions.

‘avx512vnni’
‘no-avx512vnni’
Enable/disable the generation of the AVX512VNNI instructions.

‘avx512vpopcntdq’
‘no-avx512vpopcntdq’
Enable/disable the generation of the AVX512VPOPCNTDQ instructions.

‘bmi’
‘no-bmi’ Enable/disable the generation of the BMI instructions.

‘bmi2’
‘no-bmi2’ Enable/disable the generation of the BMI2 instructions.

'cldemote'
'no-cldemote'
Enable/disable the generation of the CLDEMOTE instructions.

'clflushopt'
'no-clflushopt'
Enable/disable the generation of the CLFLUSHOPT instructions.

'clwb'
'no-clwb' Enable/disable the generation of the CLWB instructions.

'clzero'
'no-clzero'
Enable/disable the generation of the CLZERO instructions.

'crc32'
'no-crc32'
Enable/disable the generation of the CRC32 instructions.

'cx16'
'no-cx16' Enable/disable the generation of the CMPXCHG16B instructions.

'default' See Section 7.8 [Function Multiversioning], page 822, where it is used to specify the default function version.

'f16c'
'no-f16c' Enable/disable the generation of the F16C instructions.

'fma'
'no-fma' Enable/disable the generation of the FMA instructions.

'fma4'
'no-fma4' Enable/disable the generation of the FMA4 instructions.

'fsgsbase'
'no-fsgsbase'
Enable/disable the generation of the FSGSBASE instructions.

'fxsr'
'no-fxsr' Enable/disable the generation of the FXSR instructions.

'gfni'
'no-gfni' Enable/disable the generation of the GFNI instructions.

'hle'
'no-hle' Enable/disable the generation of the HLE instruction prefixes.

'lwp'
'no-lwp' Enable/disable the generation of the LWP instructions.

'lzcnt'
'no-lzcnt'
Enable/disable the generation of the LZCNT instructions.

'mmx'
'no-mmx' Enable/disable the generation of the MMX instructions.

‘movbe’
‘no-movbe’
Enable/disable the generation of the MOVBE instructions.

‘movdir64b’
‘no-movdir64b’
Enable/disable the generation of the MOVDIR64B instructions.

‘movdiri’
‘no-movdiri’
Enable/disable the generation of the MOVDIRI instructions.

‘mwaitx’
‘no-mwaitx’
Enable/disable the generation of the MWAITX instructions.

‘pclmul’
‘no-pclmul’
Enable/disable the generation of the PCLMUL instructions.

‘pconfig’
‘no-pconfig’
Enable/disable the generation of the PCONFIG instructions.

‘pku’
‘no-pku’ Enable/disable the generation of the PKU instructions.

‘popcnt’
‘no-popcnt’
Enable/disable the generation of the POPCNT instruction.

‘prefetchwt1’
‘no-prefetchwt1’
Enable/disable the generation of the PREFETCHWT1 instructions.

‘prfchw’
‘no-prfchw’
Enable/disable the generation of the PREFETCHW instruction.

‘ptwrite’
‘no-ptwrite’
Enable/disable the generation of the PTWRITE instructions.

‘rdpid’
‘no-rdpid’
Enable/disable the generation of the RDPID instructions.

‘rdrnd’
‘no-rdrnd’
Enable/disable the generation of the RDRND instructions.

‘rdseed’
‘no-rdseed’
Enable/disable the generation of the RDSEED instructions.

‘rtm’
‘no-rtm’ Enable/disable the generation of the RTM instructions.

‘sahf’
‘no-sahf’ Enable/disable the generation of the SAHF instructions.

‘sgx’
‘no-sgx’ Enable/disable the generation of the SGX instructions.

‘sha’
‘no-sha’ Enable/disable the generation of the SHA instructions.

‘shstk’
‘no-shstk’
Enable/disable the shadow stack built-in functions from CET.

‘sse’
‘no-sse’ Enable/disable the generation of the SSE instructions.

‘sse2’
‘no-sse2’ Enable/disable the generation of the SSE2 instructions.

‘sse3’
‘no-sse3’ Enable/disable the generation of the SSE3 instructions.

‘sse4’
‘no-sse4’ Enable/disable the generation of the SSE4 instructions (both SSE4.1 and SSE4.2).

‘sse4.1’
‘no-sse4.1’
Enable/disable the generation of the sse4.1 instructions.

‘sse4.2’
‘no-sse4.2’
Enable/disable the generation of the sse4.2 instructions.

‘sse4a’
‘no-sse4a’
Enable/disable the generation of the SSE4A instructions.

‘ssse3’
‘no-ssse3’
Enable/disable the generation of the SSSE3 instructions.

‘tbm’
‘no-tbm’ Enable/disable the generation of the TBM instructions.

‘vaes’
‘no-vaes’ Enable/disable the generation of the VAES instructions.

‘vpclmulqdq’
‘no-vpclmulqdq’
Enable/disable the generation of the VPCLMULQDQ instructions.

'waitpkg'
'no-waitpkg'
Enable/disable the generation of the WAITPKG instructions.

'wbnoinvd'
'no-wbnoinvd'
Enable/disable the generation of the WBNOINVD instructions.

'xop'
'no-xop' Enable/disable the generation of the XOP instructions.

'xsave'
'no-xsave'
Enable/disable the generation of the XSAVE instructions.

'xsavec'
'no-xsavec'
Enable/disable the generation of the XSAVEC instructions.

'xsaveopt'
'no-xsaveopt'
Enable/disable the generation of the XSAVEOPT instructions.

'xsaves'
'no-xsaves'
Enable/disable the generation of the XSAVES instructions.

'cld'
'no-cld' Enable/disable the generation of the CLD before string moves.

'fancy-math-387'
'no-fancy-math-387'
Enable/disable the generation of the `sin`, `cos`, and `sqrt` instructions on the 387 floating-point unit.

'ieee-fp'
'no-ieee-fp'
Enable/disable the generation of floating point that depends on IEEE arithmetic.

'inline-all-stringops'
'no-inline-all-stringops'
Enable/disable inlining of string operations.

'inline-stringops-dynamically'
'no-inline-stringops-dynamically'
Enable/disable the generation of the inline code to do small string operations and calling the library routines for large operations.

'align-stringops'
'no-align-stringops'
Do/do not align destination of inlined string operations.

'`recip`'
'`no-recip`'
Enable/disable the generation of RCPSS, RCPPS, RSQRTSS and RSQRTPS instructions followed an additional Newton-Raphson step instead of doing a floating-point division.

'`arch=ARCH`'
Specify the architecture to generate code for in compiling the function.

'`tune=TUNE`'
Specify the architecture to tune for in compiling the function.

'`fpmath=FPMATH`'
Specify which floating-point unit to use. You must specify the `target("fpmath=sse,387")` option as `target("fpmath=sse+387")` because the comma would separate different options.

'`indirect_branch("choice")`'
On x86 targets, the `indirect_branch` attribute causes the compiler to convert indirect call and jump with *choice*. '`keep`' keeps indirect call and jump unmodified. '`thunk`' converts indirect call and jump to call and return thunk. '`thunk-inline`' converts indirect call and jump to inlined call and return thunk. '`thunk-extern`' converts indirect call and jump to external call and return thunk provided in a separate object file.

'`function_return("choice")`'
On x86 targets, the `function_return` attribute causes the compiler to convert function return with *choice*. '`keep`' keeps function return unmodified. '`thunk`' converts function return to call and return thunk. '`thunk-inline`' converts function return to inlined call and return thunk. '`thunk-extern`' converts function return to external call and return thunk provided in a separate object file.

'`nocf_check`'
The `nocf_check` attribute on a function is used to inform the compiler that the function's prologue should not be instrumented when compiled with the '`-fcf-protection=branch`' option. The compiler assumes that the function's address is a valid target for a control-flow transfer.

The `nocf_check` attribute on a type of pointer to function is used to inform the compiler that a call through the pointer should not be instrumented when compiled with the '`-fcf-protection=branch`' option. The compiler assumes that the function's address from the pointer is a valid target for a control-flow transfer. A direct function call through a function name is assumed to be a safe call thus direct calls are not instrumented by the compiler.

The `nocf_check` attribute is applied to an object's type. In case of assignment of a function address or a function pointer to another pointer, the attribute is not carried over from the right-hand object's type; the type of left-hand object stays unchanged. The compiler checks for `nocf_check` attribute mismatch and reports a warning in case of mismatch.

```
{
int foo (void) __attribute__(nocf_check);
void (*foo1)(void) __attribute__(nocf_check);
void (*foo2)(void);

/* foo's address is assumed to be valid.  */
int
foo (void)

  /* This call site is not checked for control-flow
     validity.  */
  (*foo1)();

  /* A warning is issued about attribute mismatch.  */
  foo1 = foo2;

  /* This call site is still not checked.  */
  (*foo1)();

  /* This call site is checked.  */
  (*foo2)();

  /* A warning is issued about attribute mismatch.  */
  foo2 = foo1;

  /* This call site is still checked.  */
  (*foo2)();

  return 0;
}
```

'`cf_check`'

The `cf_check` attribute on a function is used to inform the compiler that ENDBR instruction should be placed at the function entry when '`-fcf-protection=branch`' is enabled.

'`indirect_return`'

The `indirect_return` attribute can be applied to a function, as well as variable or type of function pointer to inform the compiler that the function may return via indirect branch.

'`fentry_name("name")`'

On x86 targets, the `fentry_name` attribute sets the function to call on function entry when function instrumentation is enabled with '`-pg -mfentry`'. When *name* is nop then a 5 byte nop sequence is generated.

'`fentry_section("name")`'
On x86 targets, the `fentry_section` attribute sets the name of the section to record function entry instrumentation calls in when enabled with '`-pg -mrecord-mcount`'

On the x86, the inliner does not inline a function that has different target options than the caller, unless the callee has a subset of the target options of the caller. For example a function declared with `target("sse3")` can inline a function with `target("sse2")`, since `-msse3` implies `-msse2`.

6.33.36 Xstormy16 Function Attributes

These function attributes are supported by the Xstormy16 back end:

`interrupt`
Use this attribute to indicate that the specified function is an interrupt handler. The compiler generates function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

6.34 Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special properties of variables, function parameters, or structure, union, and, in C++, class members. This `__attribute__` keyword is followed by an attribute specification enclosed in double parentheses. Some attributes are currently defined generically for variables. Other attributes are defined for variables on particular target systems. Other attributes are available for functions (see Section 6.33 [Function Attributes], page 477), labels (see Section 6.36 [Label Attributes], page 557), enumerators (see Section 6.37 [Enumerator Attributes], page 558), statements (see Section 6.38 [Statement Attributes], page 559), and for types (see Section 6.35 [Type Attributes], page 548). Other front ends might define more attributes (see Chapter 7 [Extensions to the C++ Language], page 815).

See Section 6.39 [Attribute Syntax], page 559, for details of the exact syntax for using attributes.

6.34.1 Common Variable Attributes

The following attributes are supported on most targets.

`aligned`
`aligned (alignment)`
The `aligned` attribute specifies a minimum alignment for the variable or structure field, measured in bytes. When specified, *alignment* must be an integer constant power of 2. Specifying no *alignment* argument implies the maximum alignment for the target, which is often, but by no means always, 8 or 16 bytes.

For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. On a 68040, this could be used in conjunction with an `asm` expression to access the `move16` instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member, which forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the default alignment for the target architecture you are compiling for. The default alignment is sufficient for all scalar types, but may not be enough for all vector types on a target that supports vector operations. The default alignment is fixed for a particular target ABI.

GCC also provides a target specific macro `__BIGGEST_ALIGNMENT__`, which is the largest alignment ever used for any data type on the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned (__BIGGEST_ALIGNMENT__)));
```

The compiler automatically sets the alignment for the declared variable or field to `__BIGGEST_ALIGNMENT__`. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way. Note that the value of `__BIGGEST_ALIGNMENT__` may change depending on command-line options.

When used on a struct, or struct member, the `aligned` attribute can only increase the alignment; in order to decrease it, the `packed` attribute must be specified as well. When used as part of a typedef, the `aligned` attribute can both increase and decrease alignment, and specifying the `packed` attribute generates a warning.

Note that the effectiveness of `aligned` attributes for static variables may be limited by inherent limitations in the system linker and/or object file format. On some systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8-byte alignment, then specifying `aligned(16)` in an `__attribute__` still only provides you with 8-byte alignment. See your linker documentation for further information.

Stack variables are not affected by linker restrictions; GCC can properly align them on any target.

The `aligned` attribute can also be used for functions (see Section 6.33.1 [Common Function Attributes], page 478.)

`warn_if_not_aligned (`*alignment*`)`

This attribute specifies a threshold for the structure field, measured in bytes. If the structure field is aligned below the threshold, a warning will be issued. For example, the declaration:

```
struct foo
{
  int i1;
  int i2;
```

```
  unsigned long long x __attribute__ ((warn_if_not_aligned (16)));
};
```

causes the compiler to issue an warning on `struct foo`, like '`warning: alignment 8 of 'struct foo' is less than 16`'. The compiler also issues a warning, like '`warning: 'x' offset 8 in 'struct foo' isn't aligned to 16`', when the structure field has the misaligned offset:

```
struct __attribute__ ((aligned (16))) foo
{
  int i1;
  int i2;
  unsigned long long x __attribute__ ((warn_if_not_aligned (16)));
};
```

This warning can be disabled by '`-Wno-if-not-aligned`'. The `warn_if_not_aligned` attribute can also be used for types (see Section 6.35.1 [Common Type Attributes], page 549.)

`alloc_size (`*`position`*`)`
`alloc_size (`*`position-1`*`, `*`position-2`*`)`

The `alloc_size` variable attribute may be applied to the declaration of a pointer to a function that returns a pointer and takes at least one argument of an integer type. It indicates that the returned pointer points to an object whose size is given by the function argument at *position-1*, or by the product of the arguments at *position-1* and *position-2*. Meaningful sizes are positive values less than `PTRDIFF_MAX`. Other sizes are disagnosed when detected. GCC uses this information to improve the results of `__builtin_object_size`.

For instance, the following declarations

```
typedef __attribute__ ((alloc_size (1, 2))) void*
  (*calloc_ptr) (size_t, size_t);
typedef __attribute__ ((alloc_size (1))) void*
  (*malloc_ptr) (size_t);
```

specify that `calloc_ptr` is a pointer of a function that, like the standard C function `calloc`, returns an object whose size is given by the product of arguments 1 and 2, and similarly, that `malloc_ptr`, like the standard C function `malloc`, returns an object whose size is given by argument 1 to the function.

`cleanup (`*`cleanup_function`*`)`

The `cleanup` attribute runs a function when the variable goes out of scope. This attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.

If '`-fexceptions`' is enabled, then *cleanup_function* is run during the stack unwinding that happens during the processing of the exception. Note that the `cleanup` attribute does not allow the exception to be caught, only to perform an action. It is undefined what happens if *cleanup_function* does not return normally.

`common`
`nocommon`

The `common` attribute requests GCC to place a variable in "common" storage. The `nocommon` attribute requests the opposite—to allocate space for it directly.

These attributes override the default chosen by the '`-fno-common`' and '`-fcommon`' flags respectively.

`copy`
`copy (variable)`

The `copy` attribute applies the set of attributes with which *variable* has been declared to the declaration of the variable to which the attribute is applied. The attribute is designed for libraries that define aliases that are expected to specify the same set of attributes as the aliased symbols. The `copy` attribute can be used with variables, functions or types. However, the kind of symbol to which the attribute is applied (either varible or function) must match the kind of symbol to which the argument refers. The `copy` attribute copies only syntactic and semantic attributes but not attributes that affect a symbol's linkage or visibility such as `alias`, `visibility`, or `weak`. The `deprecated` attribute is also not copied. See Section 6.33.1 [Common Function Attributes], page 478. See Section 6.35.1 [Common Type Attributes], page 549.

`deprecated`
`deprecated (msg)`

The `deprecated` attribute results in a warning if the variable is used anywhere in the source file. This is useful when identifying variables that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated variable, to enable users to easily find further information about why the variable is deprecated, or what they should do instead. Note that the warning only occurs for uses:

```
extern int old_var __attribute__ ((deprecated));
extern int old_var;
int new_fn () { return old_var; }
```

results in a warning on line 3 but not line 2. The optional *msg* argument, which must be a string, is printed in the warning if present.

The `deprecated` attribute can also be used for functions and types (see Section 6.33.1 [Common Function Attributes], page 478, see Section 6.35.1 [Common Type Attributes], page 549).

The message attached to the attribute is affected by the setting of the '`-fmessage-length`' option.

`mode (mode)`

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating-point type according to its width.

See Section "Machine Modes" in *GNU Compiler Collection (GCC) Internals*, for a list of the possible keywords for *mode*. You may also specify a mode of `byte` or `__byte__` to indicate the mode corresponding to a one-byte integer, `word` or `__word__` for the mode of a one-word integer, and `pointer` or `__pointer__` for the mode used to represent pointers.

`nonstring`

The `nonstring` variable attribute specifies that an object or member declaration with type array of `char`, `signed char`, or `unsigned char`, or pointer to

such a type is intended to store character arrays that do not necessarily contain a terminating NUL. This is useful in detecting uses of such arrays or pointers with functions that expect NUL-terminated strings, and to avoid warnings when such an array or pointer is used as an argument to a bounded string manipulation function such as strncpy. For example, without the attribute, GCC will issue a warning for the strncpy call below because it may truncate the copy without appending the terminating NUL character. Using the attribute makes it possible to suppress the warning. However, when the array is declared with the attribute the call to strlen is diagnosed because when the array doesn't contain a NUL-terminated string the call is undefined. To copy, compare, of search non-string character arrays use the memcpy, memcmp, memchr, and other functions that operate on arrays of bytes. In addition, calling strnlen and strndup with such arrays is safe provided a suitable bound is specified, and not diagnosed.

```
struct Data
{
  char name [32] __attribute__ ((nonstring));
};

int f (struct Data *pd, const char *s)
{
  strncpy (pd->name, s, sizeof pd->name);
  ...
  return strlen (pd->name);   // unsafe, gets a warning
}
```

packed The packed attribute specifies that a structure member should have the smallest possible alignment—one bit for a bit-field and one byte otherwise, unless a larger value is specified with the aligned attribute. The attribute does not apply to non-member objects.

For example in the structure below, the member array x is packed so that it immediately follows a with no intervening padding:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

Note: The 4.1, 4.2 and 4.3 series of GCC ignore the packed attribute on bit-fields of type char. This has been fixed in GCC 4.4 but the change can lead to differences in the structure layout. See the documentation of '-Wpacked-bitfield-compat' for more information.

section ("*section-name*")

Normally, the compiler places the objects it generates in sections like data and bss. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The section attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
```

```
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA")));

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the `section` attribute with *global* variables and not *local* variables, as shown in the example.

You may use the `section` attribute with initialized or uninitialized global variables but the linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the `common` (or `bss`) section and can be multiply "defined". Using the `section` attribute changes what section the variable goes into and may cause the linker to issue an error if an uninitialized variable has multiple definitions. You can force a variable to be initialized with the '`-fno-common`' flag or the `nocommon` attribute.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`tls_model ("tls_model")`

The `tls_model` attribute sets thread-local storage model (see Section 6.64 [Thread-Local], page 810) of a particular `__thread` variable, overriding '`-ftls-model=`' command-line switch on a per-variable basis. The *tls_model* argument should be one of `global-dynamic`, `local-dynamic`, `initial-exec` or `local-exec`.

Not all targets support this attribute.

`unused`

This attribute, attached to a variable, means that the variable is meant to be possibly unused. GCC does not produce a warning for this variable.

`used`

This attribute, attached to a variable with static storage, means that the variable must be emitted even if it appears that the variable is not referenced.

When applied to a static data member of a C++ class template, the attribute also means that the member is instantiated if the class itself is instantiated.

`vector_size (bytes)`

This attribute specifies the vector size for the type of the declared variable, measured in bytes. The type to which it applies is known as the *base type*. The *bytes* argument must be a positive power-of-two multiple of the base type size. For example, the declaration:

```
int foo __attribute__ ((vector_size (16)));
```

causes the compiler to set the mode for `foo`, to be 16 bytes, divided into `int` sized units. Assuming a 32-bit `int`, `foo`'s type is a vector of four units of four bytes each, and the corresponding mode of `foo` is `V4SI`. See Section 6.52 [Vector Extensions], page 626 for details of manipulating vector variables.

This attribute is only applicable to integral and floating scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

Aggregates with this attribute are invalid, even if they are of the same size as a corresponding scalar. For example, the declaration:

```
struct S { int a; };
struct S  __attribute__ ((vector_size (16))) foo;
```

is invalid even if the size of the structure is the same as the size of the `int`.

`visibility ("visibility_type")`
: This attribute affects the linkage of the declaration to which it is attached. The `visibility` attribute is described in Section 6.33.1 [Common Function Attributes], page 478.

`weak`
: The `weak` attribute is described in Section 6.33.1 [Common Function Attributes], page 478.

6.34.2 ARC Variable Attributes

`aux`
: The `aux` attribute is used to directly access the ARC's auxiliary register space from C. The auxilirary register number is given via attribute argument.

6.34.3 AVR Variable Attributes

`progmem`
: The `progmem` attribute is used on the AVR to place read-only data in the non-volatile program memory (flash). The `progmem` attribute accomplishes this by putting respective variables into a section whose name starts with `.progmem`.

This attribute works similar to the `section` attribute but adds additional checking.

- Ordinary AVR cores with 32 general purpose registers:

  `progmem` affects the location of the data but not how this data is accessed. In order to read data located with the `progmem` attribute (inline) assembler must be used.

```
/* Use custom macros from AVR-LibC (http://nongnu.org/avr-libc/user-manual/) */
#include <avr/pgmspace.h>

/* Locate var in flash memory */
const int var[2] PROGMEM = { 1, 2 };

int read_var (int i)
{
    /* Access var[] by accessor macro from avr/pgmspace.h */
    return (int) pgm_read_word (& var[i]);
}
```

  AVR is a Harvard architecture processor and data and read-only data normally resides in the data memory (RAM).

See also the [AVR Named Address Spaces], page 466 section for an alternate way to locate and access data in flash memory.

- AVR cores with flash memory visible in the RAM address range:

  On such devices, there is no need for attribute `progmem` or [`__flash`], page 466 qualifier at all. Just use standard C / C++. The compiler will generate `LD*` instructions. As flash memory is visible in the RAM address range, and the default linker script does *not* locate `.rodata` in RAM, no special features are needed in order not to waste RAM for read-only data or to read from flash. You might even get slightly better performance by avoiding `progmem` and `__flash`. This applies to devices from families `avrtiny` and `avrxmega3`, see Section 3.18.6 [AVR Options], page 277 for an overview.

- Reduced AVR Tiny cores like ATtiny40:

  The compiler adds `0x4000` to the addresses of objects and declarations in `progmem` and locates the objects in flash memory, namely in section `.progmem.data`. The offset is needed because the flash memory is visible in the RAM address space starting at address `0x4000`.

  Data in `progmem` can be accessed by means of ordinary C code, no special functions or macros are needed.

```
/* var is located in flash memory */
extern const int var[2] __attribute__((progmem));

int read_var (int i)
{
    return var[i];
}
```

  Please notice that on these devices, there is no need for `progmem` at all.

`io`

`io (addr)` Variables with the `io` attribute are used to address memory-mapped peripherals in the io address range. If an address is specified, the variable is assigned that address, and the value is interpreted as an address in the data address space. Example:

```
volatile int porta __attribute__((io (0x22)));
```

The address specified in the address in the data address range.

Otherwise, the variable it is not assigned an address, but the compiler will still use in/out instructions where applicable, assuming some other module assigns an address in the io address range. Example:

```
extern volatile int porta __attribute__((io));
```

`io_low`

`io_low (addr)`

This is like the `io` attribute, but additionally it informs the compiler that the object lies in the lower half of the I/O area, allowing the use of `cbi`, `sbi`, `sbic` and `sbis` instructions.

`address`
`address (addr)`

Variables with the `address` attribute are used to address memory-mapped peripherals that may lie outside the io address range.

```
volatile int porta __attribute__((address (0x600)));
```

`absdata` Variables in static storage and with the `absdata` attribute can be accessed by the `LDS` and `STS` instructions which take absolute addresses.

- This attribute is only supported for the reduced AVR Tiny core like ATtiny40.
- You must make sure that respective data is located in the address range `0x40...0xbf` accessible by `LDS` and `STS`. One way to achieve this as an appropriate linker description file.
- If the location does not fit the address range of `LDS` and `STS`, there is currently (Binutils 2.26) just an unspecific warning like

  ```
  module.c:(.text+0x1c): warning: internal error: out
  of range error
  ```

See also the '`-mabsdata`' Section 3.18.6 [AVR Options], page 277.

6.34.4 Blackfin Variable Attributes

Three attributes are currently defined for the Blackfin.

`l1_data`
`l1_data_A`
`l1_data_B`

Use these attributes on the Blackfin to place the variable into L1 Data SRAM. Variables with `l1_data` attribute are put into the specific section named `.l1.data`. Those with `l1_data_A` attribute are put into the specific section named `.l1.data.A`. Those with `l1_data_B` attribute are put into the specific section named `.l1.data.B`.

`l2` Use this attribute on the Blackfin to place the variable into L2 SRAM. Variables with `l2` attribute are put into the specific section named `.l2.data`.

6.34.5 H8/300 Variable Attributes

These variable attributes are available for H8/300 targets:

`eightbit_data`

Use this attribute on the H8/300, H8/300H, and H8S to indicate that the specified variable should be placed into the eight-bit data section. The compiler generates more efficient code for certain operations on data in the eight-bit data area. Note the eight-bit data area is limited to 256 bytes of data.

You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

`tiny_data`

Use this attribute on the H8/300H and H8S to indicate that the specified variable should be placed into the tiny data section. The compiler generates more

efficient code for loads and stores on data in the tiny data section. Note the tiny data area is limited to slightly under 32KB of data.

6.34.6 IA-64 Variable Attributes

The IA-64 back end supports the following variable attribute:

`model (model-name)`

On IA-64, use this attribute to set the addressability of an object. At present, the only supported identifier for *model-name* is `small`, indicating addressability via "small" (22-bit) addresses (so that their addresses can be loaded with the `addl` instruction). Caveat: such addressing is by definition not position independent and hence this attribute must not be used for objects defined by shared libraries.

6.34.7 M32R/D Variable Attributes

One attribute is currently defined for the M32R/D.

`model (model-name)`

Use this attribute on the M32R/D to set the addressability of an object. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction).

Medium and large model objects may live anywhere in the 32-bit address space (the compiler generates `seth/add3` instructions to load their addresses).

6.34.8 MeP Variable Attributes

The MeP target has a number of addressing modes and busses. The `near` space spans the standard memory space's first 16 megabytes (24 bits). The `far` space spans the entire 32-bit memory space. The `based` space is a 128-byte region in the memory space that is addressed relative to the `$tp` register. The `tiny` space is a 65536-byte region relative to the `$gp` register. In addition to these memory regions, the MeP target has a separate 16-bit control bus which is specified with `cb` attributes.

`based` Any variable with the `based` attribute is assigned to the `.based` section, and is accessed with relative to the `$tp` register.

`tiny` Likewise, the `tiny` attribute assigned variables to the `.tiny` section, relative to the `$gp` register.

`near` Variables with the `near` attribute are assumed to have addresses that fit in a 24-bit addressing mode. This is the default for large variables (`-mtiny=4` is the default) but this attribute can override `-mtiny=` for small variables, or override `-ml`.

`far` Variables with the `far` attribute are addressed using a full 32-bit address. Since this covers the entire memory space, this allows modules to make no assumptions about where variables might be stored.

io
io (*addr*) Variables with the io attribute are used to address memory-mapped peripherals. If an address is specified, the variable is assigned that address, else it is not assigned an address (it is assumed some other module assigns an address). Example:

```
int timer_count __attribute__((io(0x123)));
```

cb
cb (*addr*) Variables with the cb attribute are used to access the control bus, using special instructions. addr indicates the control bus address. Example:

```
int cpu_clock __attribute__((cb(0x123)));
```

6.34.9 Microsoft Windows Variable Attributes

You can use these attributes on Microsoft Windows targets. Section 6.34.16 [x86 Variable Attributes], page 548 for additional Windows compatibility attributes available on all x86 targets.

dllimport
dllexport
The dllimport and dllexport attributes are described in Section 6.33.19 [Microsoft Windows Function Attributes], page 511.

selectany
The selectany attribute causes an initialized global variable to have link-once semantics. When multiple definitions of the variable are encountered by the linker, the first is selected and the remainder are discarded. Following usage by the Microsoft compiler, the linker is told *not* to warn about size or content differences of the multiple definitions.

Although the primary usage of this attribute is for POD types, the attribute can also be applied to global C++ objects that are initialized by a constructor. In this case, the static initialization and destruction code for the object is emitted in each translation defining the object, but the calls to the constructor and destructor are protected by a link-once guard variable.

The selectany attribute is only available on Microsoft Windows targets. You can use __declspec (selectany) as a synonym for __attribute__ ((selectany)) for compatibility with other compilers.

shared On Microsoft Windows, in addition to putting variable definitions in a named section, the section can also be shared among all running copies of an executable or DLL. For example, this small program defines shared data by putting it in a named section shared and marking the section shareable:

```
int foo __attribute__((section ("shared"), shared)) = 0;

int
main()
{
  /* Read and write foo.  All running
     copies see the same value.  */
  return 0;
}
```

You may only use the `shared` attribute along with `section` attribute with a fully-initialized global definition because of the way linkers work. See `section` attribute for more information.

The `shared` attribute is only available on Microsoft Windows.

6.34.10 MSP430 Variable Attributes

`noinit` Any data with the `noinit` attribute will not be initialised by the C runtime startup code, or the program loader. Not initialising data in this way can reduce program startup times.

`persistent`
Any variable with the `persistent` attribute will not be initialised by the C runtime startup code. Instead its value will be set once, when the application is loaded, and then never initialised again, even if the processor is reset or the program restarts. Persistent data is intended to be placed into FLASH RAM, where its value will be retained across resets. The linker script being used to create the application should ensure that persistent data is correctly placed.

`lower`
`upper`
`either` These attributes are the same as the MSP430 function attributes of the same name (see Section 6.33.21 [MSP430 Function Attributes], page 514). These attributes can be applied to both functions and variables.

6.34.11 Nvidia PTX Variable Attributes

These variable attributes are supported by the Nvidia PTX back end:

`shared` Use this attribute to place a variable in the `.shared` memory space. This memory space is private to each cooperative thread array; only threads within one thread block refer to the same instance of the variable. The runtime does not initialize variables in this memory space.

6.34.12 PowerPC Variable Attributes

Three attributes currently are defined for PowerPC configurations: `altivec`, `ms_struct` and `gcc_struct`.

For full documentation of the struct attributes please see the documentation in Section 6.34.16 [x86 Variable Attributes], page 548.

For documentation of `altivec` attribute please see the documentation in Section 6.35.5 [PowerPC Type Attributes], page 557.

6.34.13 RL78 Variable Attributes

The RL78 back end supports the `saddr` variable attribute. This specifies placement of the corresponding variable in the SADDR area, which can be accessed more efficiently than the default memory region.

6.34.14 SPU Variable Attributes

The SPU supports the `spu_vector` attribute for variables. For documentation of this attribute please see the documentation in Section 6.35.6 [SPU Type Attributes], page 557.

6.34.15 V850 Variable Attributes

These variable attributes are supported by the V850 back end:

`sda` Use this attribute to explicitly place a variable in the small data area, which can hold up to 64 kilobytes.

`tda` Use this attribute to explicitly place a variable in the tiny data area, which can hold up to 256 bytes in total.

`zda` Use this attribute to explicitly place a variable in the first 32 kilobytes of memory.

6.34.16 x86 Variable Attributes

Two attributes are currently defined for x86 configurations: `ms_struct` and `gcc_struct`.

`ms_struct`
`gcc_struct`

If `packed` is used on a structure, or if bit-fields are used, it may be that the Microsoft ABI lays out the structure differently than the way GCC normally does. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

The `ms_struct` and `gcc_struct` attributes correspond to the '`-mms-bitfields`' and '`-mno-ms-bitfields`' command-line options, respectively; see Section 3.18.58 [x86 Options], page 401, for details of how structure layout is affected. See Section 6.35.7 [x86 Type Attributes], page 557, for information about the corresponding attributes on types.

6.34.17 Xstormy16 Variable Attributes

One attribute is currently defined for xstormy16 configurations: `below100`.

`below100`

If a variable has the `below100` attribute (`BELOW100` is allowed also), GCC places the variable in the first 0x100 bytes of memory and use special opcodes to access it. Such variables are placed in either the `.bss_below100` section or the `.data_below100` section.

6.35 Specifying Attributes of Types

The keyword `__attribute__` allows you to specify various special properties of types. Some type attributes apply only to structure and union types, and in C++, also class types, while others can apply to any type defined via a `typedef` declaration. Unless otherwise specified, the same restrictions and effects apply to attributes regardless of whether a type is a trivial structure or a C++ class with user-defined constructors, destructors, or a copy assignment.

Other attributes are defined for functions (see Section 6.33 [Function Attributes], page 477), labels (see Section 6.36 [Label Attributes], page 557), enumerators (see Section 6.37 [Enumerator Attributes], page 558), statements (see Section 6.38 [Statement Attributes], page 559), and for variables (see Section 6.34 [Variable Attributes], page 536).

The `__attribute__` keyword is followed by an attribute specification enclosed in double parentheses.

You may specify type attributes in an enum, struct or union type declaration or definition by placing them immediately after the `struct`, `union` or `enum` keyword. You can also place them just past the closing curly brace of the definition, but this is less preferred because logically the type should be fully defined at the closing brace.

You can also include type attributes in a `typedef` declaration. See Section 6.39 [Attribute Syntax], page 559, for details of the exact syntax for using attributes.

6.35.1 Common Type Attributes

The following type attributes are supported on most targets.

`aligned`
`aligned (`*alignment*`)`

The `aligned` attribute specifies a minimum alignment (in bytes) for variables of the specified type. When specified, *alignment* must be a power of 2. Specifying no *alignment* argument implies the maximum alignment for the target, which is often, but by no means always, 8 or 16 bytes. For example, the declarations:

```
struct __attribute__ ((aligned (8))) S { short f[3]; };
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

force the compiler to ensure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` is allocated and aligned *at least* on a 8-byte boundary. On a SPARC, having all variables of type `struct S` aligned to 8-byte boundaries allows the compiler to use the `ldd` and `std` (doubleword load and store) instructions when copying one variable of type `struct S` to another, thus improving run-time efficiency.

Note that the alignment of any given `struct` or `union` type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the `struct` or `union` in question. This means that you *can* effectively adjust the alignment of a `struct` or `union` type by attaching an `aligned` attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire `struct` or `union` type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given `struct` or `union` type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct __attribute__ ((aligned)) S { short f[3]; };
```

Whenever you leave out the alignment factor in an `aligned` attribute specification, the compiler automatically sets the alignment for the type to the largest alignment that is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables that have types that you have aligned this way.

In the example above, if the size of each `short` is 2 bytes, then the size of the entire `struct S` type is 6 bytes. The smallest power of two that is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that although you can ask the compiler to select a time-efficient alignment for a given type and then declare only individual stand-alone objects of that type, the compiler's ability to select a time-efficient alignment is primarily useful only when you plan to create arrays of variables having the relevant (efficiently aligned) type. If you declare or use arrays of variables of an efficiently-aligned type, then it is likely that your program also does pointer arithmetic (or subscripting, which amounts to the same thing) on pointers to the relevant type, and the code that the compiler generates for these pointer arithmetic operations is often more efficient for efficiently-aligned types than for other types.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8-byte alignment, then specifying `aligned (16)` in an `__attribute__` still only provides you with 8-byte alignment. See your linker documentation for further information.

When used on a struct, or struct member, the `aligned` attribute can only increase the alignment; in order to decrease it, the `packed` attribute must be specified as well. When used as part of a typedef, the `aligned` attribute can both increase and decrease alignment, and specifying the `packed` attribute generates a warning.

`warn_if_not_aligned (`*alignment*`)`

This attribute specifies a threshold for the structure field, measured in bytes. If the structure field is aligned below the threshold, a warning will be issued. For example, the declaration:

```
typedef unsigned long long __u64
   __attribute__((aligned (4), warn_if_not_aligned (8)));

struct foo
{
  int i1;
  int i2;
  __u64 x;
};
```

causes the compiler to issue an warning on `struct foo`, like '`warning: alignment 4 of 'struct foo' is less than 8`'. It is used to define `struct foo` in such a way that `struct foo` has the same layout and the structure field `x` has the same alignment when `__u64` is aligned at either 4 or 8 bytes. Align `struct foo` to 8 bytes:

```
struct __attribute__ ((aligned (8))) foo
{
  int i1;
  int i2;
  __u64 x;
```

```
};
```

silences the warning. The compiler also issues a warning, like '`warning: 'x' offset 12 in 'struct foo' isn't aligned to 8`', when the structure field has the misaligned offset:

```
struct __attribute__ ((aligned (8))) foo
{
  int i1;
  int i2;
  int i3;
  __u64 x;
};
```

This warning can be disabled by '`-Wno-if-not-aligned`'.

`alloc_size (`*`position`*`)`
`alloc_size (`*`position-1`*`, `*`position-2`*`)`

The `alloc_size` type attribute may be applied to the definition of a type of a function that returns a pointer and takes at least one argument of an integer type. It indicates that the returned pointer points to an object whose size is given by the function argument at *position-1*, or by the product of the arguments at *position-1* and *position-2*. Meaningful sizes are positive values less than `PTRDIFF_MAX`. Other sizes are disagnosed when detected. GCC uses this information to improve the results of `__builtin_object_size`.

For instance, the following declarations

```
typedef __attribute__ ((alloc_size (1, 2))) void*
  calloc_type (size_t, size_t);
typedef __attribute__ ((alloc_size (1))) void*
  malloc_type (size_t);
```

specify that `calloc_type` is a type of a function that, like the standard C function `calloc`, returns an object whose size is given by the product of arguments 1 and 2, and that `malloc_type`, like the standard C function `malloc`, returns an object whose size is given by argument 1 to the function.

`copy`
`copy (`*`expression`*`)`

The `copy` attribute applies the set of attributes with which the type of the *expression* has been declared to the declaration of the type to which the attribute is applied. The attribute is designed for libraries that define aliases that are expected to specify the same set of attributes as the aliased symbols. The `copy` attribute can be used with types, variables, or functions. However, the kind of symbol to which the attribute is applied (either varible or function) must match the kind of symbol to which the argument refers. The `copy` attribute copies only syntactic and semantic attributes but not attributes that affect a symbol's linkage or visibility such as `alias`, `visibility`, or `weak`. The `deprecated` attribute is also not copied. See Section 6.33.1 [Common Function Attributes], page 478. See Section 6.34.1 [Common Variable Attributes], page 536.

For example, suppose `struct A` below is defined in some third party library header to have the alignment requirement `N` and to force a warning whenever a variable of the type is not so aligned due to attribute `packed`. Specifying the `copy` attribute on the definition on the unrelated `struct B` has the effect of

copying all relevant attributes from the type referenced by the pointer expression to `struct B`.

```
struct __attribute__ ((aligned (N), warn_if_not_aligned (N)))
A { /* ... */ };
struct __attribute__ ((copy ( (struct A *)0)) B { /* ... */ };
```

`deprecated`
`deprecated (msg)`

The `deprecated` attribute results in a warning if the type is used anywhere in the source file. This is useful when identifying types that are expected to be removed in a future version of a program. If possible, the warning also includes the location of the declaration of the deprecated type, to enable users to easily find further information about why the type is deprecated, or what they should do instead. Note that the warnings only occur for uses and then only if the type is being applied to an identifier that itself is not being declared as deprecated.

```
typedef int T1 __attribute__ ((deprecated));
T1 x;
typedef T1 T2;
T2 y;
typedef T1 T3 __attribute__ ((deprecated));
T3 z __attribute__ ((deprecated));
```

results in a warning on line 2 and 3 but not lines 4, 5, or 6. No warning is issued for line 4 because T2 is not explicitly deprecated. Line 5 has no warning because T3 is explicitly deprecated. Similarly for line 6. The optional *msg* argument, which must be a string, is printed in the warning if present. Control characters in the string will be replaced with escape sequences, and if the '`-fmessage-length`' option is set to 0 (its default value) then any newline characters will be ignored.

The `deprecated` attribute can also be used for functions and variables (see Section 6.33 [Function Attributes], page 477, see Section 6.34 [Variable Attributes], page 536.)

The message attached to the attribute is affected by the setting of the '`-fmessage-length`' option.

`designated_init`

This attribute may only be applied to structure types. It indicates that any initialization of an object of this type must use designated initializers rather than positional initializers. The intent of this attribute is to allow the programmer to indicate that a structure's layout may change, and that therefore relying on positional initialization will result in future breakage.

GCC emits warnings based on this attribute by default; use '`-Wno-designated-init`' to suppress them.

`may_alias`

Accesses through pointers to types with this attribute are not subject to type-based alias analysis, but are instead assumed to be able to alias any other type of objects. In the context of section 6.5 paragraph 7 of the C99 standard, an lvalue expression dereferencing such a pointer is treated like having a character type. See '`-fstrict-aliasing`' for more information on aliasing issues. This

extension exists to support some vector APIs, in which pointers to one vector type are permitted to alias pointers to a different vector type.

Note that an object of a type with this attribute does not have any special semantics.

Example of use:

```
typedef short __attribute__ ((__may_alias__)) short_a;

int
main (void)
{
  int a = 0x12345678;
  short_a *b = (short_a *) &a;

  b[1] = 0;

  if (a == 0x12345678)
    abort();

  exit(0);
}
```

If you replaced `short_a` with `short` in the variable declaration, the above program would abort when compiled with '`-fstrict-aliasing`', which is on by default at '`-O2`' or above.

`mode (`*mode*`)`

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating-point type according to its width.

See Section "Machine Modes" in *GNU Compiler Collection (GCC) Internals*, for a list of the possible keywords for *mode*. You may also specify a mode of `byte` or `__byte__` to indicate the mode corresponding to a one-byte integer, `word` or `__word__` for the mode of a one-word integer, and `pointer` or `__pointer__` for the mode used to represent pointers.

`packed`

This attribute, attached to a `struct`, `union`, or C++ `class` type definition, specifies that each of its members (other than zero-width bit-fields) is placed to minimize the memory required. This is equivalent to specifying the `packed` attribute on each of the members.

When attached to an `enum` definition, the `packed` attribute indicates that the smallest integral type should be used. Specifying the '`-fshort-enums`' flag on the command line is equivalent to specifying the `packed` attribute on all `enum` definitions.

In the following example `struct my_packed_struct`'s members are packed closely together, but the internal layout of its `s` member is not packed—to do that, `struct my_unpacked_struct` needs to be packed too.

```
struct my_unpacked_struct
 {
    char c;
    int i;
 };
```

```
struct __attribute__ ((__packed__)) my_packed_struct
  {
     char c;
     int  i;
     struct my_unpacked_struct s;
  };
```

You may only specify the `packed` attribute on the definition of an `enum`, `struct`, `union`, or `class`, not on a `typedef` that does not also define the enumerated type, structure, union, or class.

`scalar_storage_order ("endianness")`

When attached to a `union` or a `struct`, this attribute sets the storage order, aka endianness, of the scalar fields of the type, as well as the array fields whose component is scalar. The supported endiannesses are `big-endian` and `little-endian`. The attribute has no effects on fields which are themselves a `union`, a `struct` or an array whose component is a `union` or a `struct`, and it is possible for these fields to have a different scalar storage order than the enclosing type.

This attribute is supported only for targets that use a uniform default scalar storage order (fortunately, most of them), i.e. targets that store the scalars either all in big-endian or all in little-endian.

Additional restrictions are enforced for types with the reverse scalar storage order with regard to the scalar storage order of the target:

- Taking the address of a scalar field of a `union` or a `struct` with reverse scalar storage order is not permitted and yields an error.
- Taking the address of an array field, whose component is scalar, of a `union` or a `struct` with reverse scalar storage order is permitted but yields a warning, unless '`-Wno-scalar-storage-order`' is specified.
- Taking the address of a `union` or a `struct` with reverse scalar storage order is permitted.

These restrictions exist because the storage order attribute is lost when the address of a scalar or the address of an array with scalar component is taken, so storing indirectly through this address generally does not work. The second case is nevertheless allowed to be able to perform a block copy from or to the array.

Moreover, the use of type punning or aliasing to toggle the storage order is not supported; that is to say, a given scalar object cannot be accessed through distinct types that assign a different storage order to it.

`transparent_union`

This attribute, attached to a `union` type definition, indicates that any function parameter having that union type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent union type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding

argument can be any pointer expression. If the union member type is a pointer, qualifiers like `const` on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of the first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the `wait` function must accept either a value of type `int *` to comply with POSIX, or a value of type `union wait *` to comply with the 4.1BSD interface. If `wait`'s parameter were `void *`, `wait` would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, `<sys/wait.h>` might define the interface as follows:

```
typedef union __attribute__ ((__transparent_union__))
  {
    int *__ip;
    union wait *__up;
  } wait_status_ptr_t;

pid_t wait (wait_status_ptr_t);
```

This interface allows either `int *` or `union wait *` arguments to be passed, using the `int *` calling convention. The program can call `wait` with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, `wait`'s implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
  return waitpid (-1, p.__ip, 0);
}
```

`unused` When attached to a type (including a `union` or a `struct`), this attribute means that variables of that type are meant to appear possibly unused. GCC does not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

`vector_size (`*bytes*`)`

This attribute specifies the vector size for the type, measured in bytes. The type to which it applies is known as the *base type*. The *bytes* argument must be a positive power-of-two multiple of the base type size. For example, the following declarations:

```
typedef __attribute__ ((vector_size (32))) int int_vec32_t ;
typedef __attribute__ ((vector_size (32))) int* int_vec32_ptr_t;
typedef __attribute__ ((vector_size (32))) int int_vec32_arr3_t[3];
```

define `int_vec32_t` to be a 32-byte vector type composed of `int` sized units. With `int` having a size of 4 bytes, the type defines a vector of eight units,

four bytes each. The mode of variables of type `int_vec32_t` is `V8SI`. `int_vec32_ptr_t` is then defined to be a pointer to such a vector type, and `int_vec32_arr3_t` to be an array of three such vectors. See Section 6.52 [Vector Extensions], page 626 for details of manipulating objects of vector types.

This attribute is only applicable to integral and floating scalar types. In function declarations the attribute applies to the function return type.

For example, the following:

```
__attribute__ ((vector_size (16))) float get_flt_vec16 (void);
```

declares `get_flt_vec16` to be a function returning a 16-byte vector with the base type `float`.

`visibility`

In C++, attribute visibility (see Section 6.33 [Function Attributes], page 477) can also be applied to class, struct, union and enum types. Unlike other type attributes, the attribute must appear between the initial keyword and the name of the type; it cannot appear after the body of the type.

Note that the type visibility is applied to vague linkage entities associated with the class (vtable, typeinfo node, etc.). In particular, if a class is thrown as an exception in one shared object and caught in another, the class must have default visibility. Otherwise the two shared objects are unable to use the same typeinfo node and exception handling will break.

To specify multiple attributes, separate them by commas within the double parentheses: for example, '`__attribute__ ((aligned (16), packed))`'.

6.35.2 ARC Type Attributes

Declaring objects with `uncached` allows you to exclude data-cache participation in load and store operations on those objects without involving the additional semantic implications of `volatile`. The `.di` instruction suffix is used for all loads and stores of data declared `uncached`.

6.35.3 ARM Type Attributes

On those ARM targets that support `dllimport` (such as Symbian OS), you can use the `notshared` attribute to indicate that the virtual table and other similar data for a class should not be exported from a DLL. For example:

```
class __declspec(notshared) C {
public:
  __declspec(dllimport) C();
  virtual void f();
}

__declspec(dllexport)
C::C() {}
```

In this code, `C::C` is exported from the current DLL, but the virtual table for `C` is not exported. (You can use `__attribute__` instead of `__declspec` if you prefer, but most Symbian OS code uses `__declspec`.)

6.35.4 MeP Type Attributes

Many of the MeP variable attributes may be applied to types as well. Specifically, the `based`, `tiny`, `near`, and `far` attributes may be applied to either. The `io` and `cb` attributes may not be applied to types.

6.35.5 PowerPC Type Attributes

Three attributes currently are defined for PowerPC configurations: `altivec`, `ms_struct` and `gcc_struct`.

For full documentation of the `ms_struct` and `gcc_struct` attributes please see the documentation in Section 6.35.7 [x86 Type Attributes], page 557.

The `altivec` attribute allows one to declare AltiVec vector data types supported by the AltiVec Programming Interface Manual. The attribute requires an argument to specify one of three vector types: `vector__`, `pixel__` (always followed by unsigned short), and `bool__` (always followed by unsigned).

```
__attribute__((altivec(vector__)))
__attribute__((altivec(pixel__))) unsigned short
__attribute__((altivec(bool__))) unsigned
```

These attributes mainly are intended to support the `__vector`, `__pixel`, and `__bool` AltiVec keywords.

6.35.6 SPU Type Attributes

The SPU supports the `spu_vector` attribute for types. This attribute allows one to declare vector data types supported by the Sony/Toshiba/IBM SPU Language Extensions Specification. It is intended to support the `__vector` keyword.

6.35.7 x86 Type Attributes

Two attributes are currently defined for x86 configurations: `ms_struct` and `gcc_struct`.

`ms_struct`
`gcc_struct`

If `packed` is used on a structure, or if bit-fields are used it may be that the Microsoft ABI packs them differently than GCC normally packs them. Particularly when moving packed data between functions compiled with GCC and the native Microsoft compiler (either via function call or as data in a file), it may be necessary to access either format.

The `ms_struct` and `gcc_struct` attributes correspond to the '`-mms-bitfields`' and '`-mno-ms-bitfields`' command-line options, respectively; see Section 3.18.58 [x86 Options], page 401, for details of how structure layout is affected. See Section 6.34.16 [x86 Variable Attributes], page 548, for information about the corresponding attributes on variables.

6.36 Label Attributes

GCC allows attributes to be set on C labels. See Section 6.39 [Attribute Syntax], page 559, for details of the exact syntax for using attributes. Other attributes are available for functions (see Section 6.33 [Function Attributes], page 477), variables (see Section 6.34 [Variable Attributes], page 536), enumerators (see Section 6.37 [Enumerator Attributes], page 558),

statements (see Section 6.38 [Statement Attributes], page 559), and for types (see Section 6.35 [Type Attributes], page 548).

This example uses the `cold` label attribute to indicate the `ErrorHandling` branch is unlikely to be taken and that the `ErrorHandling` label is unused:

```
   asm goto ("some asm" : : : : NoError);

/* This branch (the fall-through from the asm) is less commonly used */
ErrorHandling:
   __attribute__((cold, unused)); /* Semi-colon is required here */
   printf("error\n");
   return 0;

NoError:
   printf("no error\n");
   return 1;
```

`unused` This feature is intended for program-generated code that may contain unused labels, but which is compiled with '`-Wall`'. It is not normally appropriate to use in it human-written code, though it could be useful in cases where the code that jumps to the label is contained within an `#ifdef` conditional.

`hot` The `hot` attribute on a label is used to inform the compiler that the path following the label is more likely than paths that are not so annotated. This attribute is used in cases where `__builtin_expect` cannot be used, for instance with computed goto or `asm goto`.

`cold` The `cold` attribute on labels is used to inform the compiler that the path following the label is unlikely to be executed. This attribute is used in cases where `__builtin_expect` cannot be used, for instance with computed goto or `asm goto`.

6.37 Enumerator Attributes

GCC allows attributes to be set on enumerators. See Section 6.39 [Attribute Syntax], page 559, for details of the exact syntax for using attributes. Other attributes are available for functions (see Section 6.33 [Function Attributes], page 477), variables (see Section 6.34 [Variable Attributes], page 536), labels (see Section 6.36 [Label Attributes], page 557), statements (see Section 6.38 [Statement Attributes], page 559), and for types (see Section 6.35 [Type Attributes], page 548).

This example uses the `deprecated` enumerator attribute to indicate the `oldval` enumerator is deprecated:

```
enum E {
  oldval __attribute__((deprecated)),
  newval
};

int
fn (void)
{
  return oldval;
}
```

`deprecated`

The `deprecated` attribute results in a warning if the enumerator is used anywhere in the source file. This is useful when identifying enumerators that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated enumerator, to enable users to easily find further information about why the enumerator is deprecated, or what they should do instead. Note that the warnings only occurs for uses.

6.38 Statement Attributes

GCC allows attributes to be set on null statements. See Section 6.39 [Attribute Syntax], page 559, for details of the exact syntax for using attributes. Other attributes are available for functions (see Section 6.33 [Function Attributes], page 477), variables (see Section 6.34 [Variable Attributes], page 536), labels (see Section 6.36 [Label Attributes], page 557), enumerators (see Section 6.37 [Enumerator Attributes], page 558), and for types (see Section 6.35 [Type Attributes], page 548).

This example uses the `fallthrough` statement attribute to indicate that the '`-Wimplicit-fallthrough`' warning should not be emitted:

```
switch (cond)
  {
  case 1:
    bar (1);
    __attribute__((fallthrough));
  case 2:
    ...
  }
```

`fallthrough`

The `fallthrough` attribute with a null statement serves as a fallthrough statement. It hints to the compiler that a statement that falls through to another case label, or user-defined label in a switch statement is intentional and thus the '`-Wimplicit-fallthrough`' warning must not trigger. The fallthrough attribute may appear at most once in each attribute list, and may not be mixed with other attributes. It can only be used in a switch statement (the compiler will issue an error otherwise), after a preceding statement and before a logically succeeding case label, or user-defined label.

6.39 Attribute Syntax

This section describes the syntax with which `__attribute__` may be used, and the constructs to which attribute specifiers bind, for the C language. Some details may vary for C++ and Objective-C. Because of infelicities in the grammar for attributes, some forms described here may not be successfully parsed in all cases.

There are some problems with the semantics of attributes in C++. For example, there are no manglings for attributes, although they may affect code generation, so problems may arise when attributed types are used in conjunction with templates or overloading. Similarly, `typeid` does not distinguish between types with different attributes. Support for attributes in C++ may be restricted in future to attributes on declarations only, but not on nested declarators.

See Section 6.33 [Function Attributes], page 477, for details of the semantics of attributes applying to functions. See Section 6.34 [Variable Attributes], page 536, for details of the semantics of attributes applying to variables. See Section 6.35 [Type Attributes], page 548, for details of the semantics of attributes applying to structure, union and enumerated types. See Section 6.36 [Label Attributes], page 557, for details of the semantics of attributes applying to labels. See Section 6.37 [Enumerator Attributes], page 558, for details of the semantics of attributes applying to enumerators. See Section 6.38 [Statement Attributes], page 559, for details of the semantics of attributes applying to statements.

An *attribute specifier* is of the form `__attribute__ ((attribute-list))`. An *attribute list* is a possibly empty comma-separated sequence of *attributes*, where each attribute is one of the following:

- Empty. Empty attributes are ignored.
- An attribute name (which may be an identifier such as `unused`, or a reserved word such as `const`).
- An attribute name followed by a parenthesized list of parameters for the attribute. These parameters take one of the following forms:
  - An identifier. For example, `mode` attributes use this form.
  - An identifier followed by a comma and a non-empty comma-separated list of expressions. For example, `format` attributes use this form.
  - A possibly empty comma-separated list of expressions. For example, `format_arg` attributes use this form with the list being a single integer constant expression, and `alias` attributes use this form with the list being a single string constant.

An *attribute specifier list* is a sequence of one or more attribute specifiers, not separated by any other tokens.

You may optionally specify attribute names with '`__`' preceding and following the name. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use the attribute name `__noreturn__` instead of `noreturn`.

Label Attributes

In GNU C, an attribute specifier list may appear after the colon following a label, other than a `case` or `default` label. GNU C++ only permits attributes on labels if the attribute specifier is immediately followed by a semicolon (i.e., the label applies to an empty statement). If the semicolon is missing, C++ label attributes are ambiguous, as it is permissible for a declaration, which could begin with an attribute list, to be labelled in C++. Declarations cannot be labelled in C90 or C99, so the ambiguity does not arise there.

Enumerator Attributes

In GNU C, an attribute specifier list may appear as part of an enumerator. The attribute goes after the enumeration constant, before `=`, if present. The optional attribute in the enumerator appertains to the enumeration constant. It is not possible to place the attribute after the constant expression, if present.

Statement Attributes

In GNU C, an attribute specifier list may appear as part of a null statement. The attribute goes before the semicolon.

Type Attributes

An attribute specifier list may appear as part of a `struct`, `union` or `enum` specifier. It may go either immediately after the `struct`, `union` or `enum` keyword, or after the closing brace. The former syntax is preferred. Where attribute specifiers follow the closing brace, they are considered to relate to the structure, union or enumerated type defined, not to any enclosing declaration the type specifier appears in, and the type defined is not complete until after the attribute specifiers.

All other attributes

Otherwise, an attribute specifier appears as part of a declaration, counting declarations of unnamed parameters and type names, and relates to that declaration (which may be nested in another declaration, for example in the case of a parameter declaration), or to a particular declarator within a declaration. Where an attribute specifier is applied to a parameter declared as a function or an array, it should apply to the function or array rather than the pointer to which the parameter is implicitly converted, but this is not yet correctly implemented.

Any list of specifiers and qualifiers at the start of a declaration may contain attribute specifiers, whether or not such a list may in that context contain storage class specifiers. (Some attributes, however, are essentially in the nature of storage class specifiers, and only make sense where storage class specifiers may be used; for example, `section`.) There is one necessary limitation to this syntax: the first old-style parameter declaration in a function definition cannot begin with an attribute specifier, because such an attribute applies to the function instead by syntax described below (which, however, is not yet implemented in this case). In some other cases, attribute specifiers are permitted by this grammar but not yet supported by the compiler. All attribute specifiers in this place relate to the declaration as a whole. In the obsolescent usage where a type of `int` is implied by the absence of type specifiers, such a list of specifiers and qualifiers may be an attribute specifier list with no other specifiers or qualifiers.

At present, the first parameter in a function prototype must have some type specifier that is not an attribute specifier; this resolves an ambiguity in the interpretation of `void f(int (__attribute__((foo)) x))`, but is subject to change. At present, if the parentheses of a function declarator contain only attributes then those attributes are ignored, rather than yielding an error or warning or implying a single parameter of type int, but this is subject to change.

An attribute specifier list may appear immediately before a declarator (other than the first) in a comma-separated list of declarators in a declaration of more than one identifier using a single list of specifiers and qualifiers. Such attribute specifiers apply only to the identifier before whose declarator they appear. For example, in

```
__attribute__((noreturn)) void d0 (void),
    __attribute__((format(printf, 1, 2))) d1 (const char *, ...),
     d2 (void);
```

the `noreturn` attribute applies to all the functions declared; the `format` attribute only applies to `d1`.

An attribute specifier list may appear immediately before the comma, `=` or semicolon terminating the declaration of an identifier other than a function definition. Such attribute specifiers apply to the declared object or function. Where an assembler name for an object or function is specified (see Section 6.47.4 [Asm Labels], page 620), the attribute must follow the `asm` specification.

An attribute specifier list may, in future, be permitted to appear after the declarator in a function definition (before any old-style parameter declarations or the function body).

Attribute specifiers may be mixed with type qualifiers appearing inside the `[]` of a parameter array declarator, in the C99 construct by which such qualifiers are applied to the pointer to which the array is implicitly converted. Such attribute specifiers apply to the pointer, not to the array, but at present this is not implemented and they are ignored.

An attribute specifier list may appear at the start of a nested declarator. At present, there are some limitations in this usage: the attributes correctly apply to the declarator, but for most individual attributes the semantics this implies are not implemented. When attribute specifiers follow the `*` of a pointer declarator, they may be mixed with any type qualifiers present. The following describes the formal semantics of this syntax. It makes the most sense if you are familiar with the formal specification of declarators in the ISO C standard.

Consider (as in C99 subclause 6.7.5 paragraph 4) a declaration `T D1`, where `T` contains declaration specifiers that specify a type *Type* (such as `int`) and `D1` is a declarator that contains an identifier *ident*. The type specified for *ident* for derived declarators whose type does not include an attribute specifier is as in the ISO C standard.

If `D1` has the form `(` *`attribute-specifier-list`* `D )`, and the declaration `T D` specifies the type "*derived-declarator-type-list Type*" for *ident*, then `T D1` specifies the type "*derived-declarator-type-list attribute-specifier-list Type*" for *ident*.

If `D1` has the form `*` *`type-qualifier-and-attribute-specifier-list`* `D`, and the declaration `T D` specifies the type "*derived-declarator-type-list Type*" for *ident*, then `T D1` specifies the type "*derived-declarator-type-list type-qualifier-and-attribute-specifier-list* pointer to *Type*" for *ident*.

For example,

```
void (__attribute__((noreturn)) ****f) (void);
```

specifies the type "pointer to pointer to pointer to pointer to non-returning function returning `void`". As another example,

```
char *__attribute__((aligned(8))) *f;
```

specifies the type "pointer to 8-byte-aligned pointer to `char`". Note again that this does not work with most attributes; for example, the usage of '`aligned`' and '`noreturn`' attributes given above is not yet supported.

For compatibility with existing code written for compiler versions that did not implement attributes on nested declarators, some laxity is allowed in the placing of attributes. If an attribute that only applies to types is applied to a declaration, it is treated as applying to the type of that declaration. If an attribute that only applies to declarations is applied to the type of a declaration, it is treated as applying to that declaration; and, for compatibility

with code placing the attributes immediately before the identifier declared, such an attribute applied to a function return type is treated as applying to the function type, and such an attribute applied to an array element type is treated as applying to the array type. If an attribute that only applies to function types is applied to a pointer-to-function type, it is treated as applying to the pointer target type; if such an attribute is applied to a function return type that is not a pointer-to-function type, it is treated as applying to the function type.

6.40 Prototypes and Old-Style Function Definitions

GNU C extends ISO C to allow a function prototype to override a later old-style non-prototype definition. Consider the following example:

```
/* Use prototypes unless the compiler is old-fashioned.  */
#ifdef __STDC__
#define P(x) x
#else
#define P(x) ()
#endif

/* Prototype function declaration.  */
int isroot P((uid_t));

/* Old-style function definition.  */
int
isroot (x)   /* ??? lossage here ??? */
     uid_t x;
{
  return x == 0;
}
```

Suppose the type `uid_t` happens to be `short`. ISO C does not allow this example, because subword arguments in old-style non-prototype definitions are promoted. Therefore in this example the function definition's argument is really an `int`, which does not match the prototype argument type of `short`.

This restriction of ISO C makes it hard to write code that is portable to traditional C compilers, because the programmer does not know whether the `uid_t` type is `short`, `int`, or `long`. Therefore, in cases like these GNU C allows a prototype to override a later old-style definition. More precisely, in GNU C, a function prototype argument type overrides the argument type specified by a later old-style definition if the former type is the same as the latter type before promotion. Thus in GNU C the above example is equivalent to the following:

```
int isroot (uid_t);

int
isroot (uid_t x)
{
  return x == 0;
}
```

GNU C++ does not support old-style function definitions, so this extension is irrelevant.

6.41 C++ Style Comments

In GNU C, you may use C++ style comments, which start with '//' and continue until the end of the line. Many other C implementations allow such comments, and they are included in the 1999 C standard. However, C++ style comments are not recognized if you specify an '-std' option specifying a version of ISO C before C99, or '-ansi' (equivalent to '-std=c90').

6.42 Dollar Signs in Identifier Names

In GNU C, you may normally use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers. However, dollar signs in identifiers are not supported on a few target machines, typically because the target assembler does not allow them.

6.43 The Character ESC in Constants

You can use the sequence '\e' in a string or character constant to stand for the ASCII character ESC.

6.44 Determining the Alignment of Functions, Types or Variables

The keyword __alignof__ determines the alignment requirement of a function, object, or a type, or the minimum alignment usually required by a type. Its syntax is just like sizeof and C11 _Alignof.

For example, if the target machine requires a double value to be aligned on an 8-byte boundary, then __alignof__ (double) is 8. This is true on many RISC machines. On more traditional machine designs, __alignof__ (double) is 4 or even 2.

Some machines never actually require alignment; they allow references to any data type even at an odd address. For these machines, __alignof__ reports the smallest alignment that GCC gives the data type, usually as mandated by the target ABI.

If the operand of __alignof__ is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified by attribute aligned (see Section 6.34.1 [Common Variable Attributes], page 536). For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of __alignof__ (foo1.y) is 1, even though its actual alignment is probably 2 or 4, the same as __alignof__ (int). It is an error to ask for the alignment of an incomplete type other than void.

If the operand of the __alignof__ expression is a function, the expression evaluates to the alignment of the function which may be specified by attribute aligned (see Section 6.33.1 [Common Function Attributes], page 478).

6.45 An Inline Function is As Fast As a Macro

By declaring a function inline, you can direct GCC to make calls to that function faster. One way GCC can achieve this is to integrate that function's code into the code for its callers.

This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. You can also direct GCC to try to integrate all "simple enough" functions into their callers with the option '`-finline-functions`'.

GCC implements three different semantics of declaring a function inline. One is available with '`-std=gnu89`' or '`-fgnu89-inline`' or when `gnu_inline` attribute is present on all inline declarations, another when '`-std=c99`', '`-std=gnu99`' or an option for a later C version is used (without '`-fgnu89-inline`'), and the third is used when compiling C++.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
static inline int
inc (int *a)
{
  return (*a)++;
}
```

If you are writing a header file to be included in ISO C90 programs, write `__inline__` instead of `inline`. See Section 6.48 [Alternate Keywords], page 624.

The three types of inlining behave similarly in two important cases: when the `inline` keyword is used on a `static` function, like the example above, and when a function is first declared without using the `inline` keyword and then is defined with `inline`, like this:

```
extern int inc (int *a);
inline int
inc (int *a)
{
  return (*a)++;
}
```

In both of these common cases, the program behaves the same as if you had not used the `inline` keyword, except for its speed.

When a function is both inline and `static`, if all calls to the function are integrated into the caller, and the function's address is never used, then the function's own assembler code is never referenced. In this case, GCC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that cannot be inlined.

Note that certain usages in a function definition can make it unsuitable for inline substitution. Among these usages are: variadic functions, use of `alloca`, use of computed goto (see Section 6.3 [Labels as Values], page 454), use of nonlocal goto, use of nested functions, use of `setjmp`, use of `__builtin_longjmp` and use of `__builtin_return` or `__builtin_apply_args`. Using '`-Winline`' warns when a function marked `inline` could not be substituted, and gives the reason for the failure.

As required by ISO C++, GCC considers member functions defined within the body of a class to be marked inline even if they are not explicitly declared with the `inline` keyword. You can override this with '`-fno-default-inline`'; see Section 3.5 [Options Controlling C++ Dialect], page 44.

GCC does not inline any functions when not optimizing unless you specify the '`always_inline`' attribute for the function, like this:

```
/* Prototype. */
inline void foo (const char) __attribute__((always_inline));
```

The remainder of this section is specific to GNU C90 inlining.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file causes most calls to the function to be inlined. If any uses of the function remain, they refer to the single copy in the library.

6.46 When is a Volatile Object Accessed?

C has the concept of volatile objects. These are normally accessed by pointers and used for accessing hardware or inter-thread communication. The standard encourages compilers to refrain from optimizations concerning accesses to volatile objects, but leaves it implementation defined as to what constitutes a volatile access. The minimum requirement is that at a sequence point all previous accesses to volatile objects have stabilized and no subsequent accesses have occurred. Thus an implementation is free to reorder and combine volatile accesses that occur between sequence points, but cannot do so for accesses across a sequence point. The use of volatile does not allow you to violate the restriction on updating objects multiple times between two sequence points.

Accesses to non-volatile objects are not ordered with respect to volatile accesses. You cannot use a volatile object as a memory barrier to order a sequence of writes to non-volatile memory. For instance:

```
int *ptr = something;
volatile int vobj;
*ptr = something;
vobj = 1;
```

Unless **ptr* and *vobj* can be aliased, it is not guaranteed that the write to **ptr* occurs by the time the update of *vobj* happens. If you need this guarantee, you must use a stronger memory barrier such as:

```
int *ptr = something;
volatile int vobj;
*ptr = something;
asm volatile ("" : : : "memory");
vobj = 1;
```

A scalar volatile object is read when it is accessed in a void context:

```
volatile int *src = somevalue;
*src;
```

Such expressions are rvalues, and GCC implements this as a read of the volatile object being pointed to.

Assignments are also expressions and have an rvalue. However when assigning to a scalar volatile, the volatile object is not reread, regardless of whether the assignment expression's rvalue is used or not. If the assignment's rvalue is used, the value is that assigned to the volatile object. For instance, there is no read of *vobj* in all the following cases:

```
int obj;
volatile int vobj;
vobj = something;
obj = vobj = something;
obj ? vobj = onething : vobj = anotherthing;
obj = (something, vobj = anotherthing);
```

If you need to read the volatile object after an assignment has occurred, you must use a separate expression with an intervening sequence point.

As bit-fields are not individually addressable, volatile bit-fields may be implicitly read when written to, or when adjacent bit-fields are accessed. Bit-field operations may be optimized such that adjacent bit-fields are only partially accessed, if they straddle a storage unit boundary. For these reasons it is unwise to use volatile bit-fields to access hardware.

6.47 How to Use Inline Assembly Language in C Code

The `asm` keyword allows you to embed assembler instructions within C code. GCC provides two forms of inline `asm` statements. A *basic* `asm` statement is one with no operands (see Section 6.47.1 [Basic Asm], page 567), while an *extended* `asm` statement (see Section 6.47.2 [Extended Asm], page 569) includes one or more operands. The extended form is preferred for mixing C and assembly language within a function, but to include assembly language at top level you must use basic `asm`.

You can also use the `asm` keyword to override the assembler name for a C symbol, or to place a C variable in a specific register.

6.47.1 Basic Asm — Assembler Instructions Without Operands

A basic `asm` statement has the following syntax:

```
asm asm-qualifiers ( AssemblerInstructions )
```

The `asm` keyword is a GNU extension. When writing code that can be compiled with '`-ansi`' and the various '`-std`' options, use `__asm__` instead of `asm` (see Section 6.48 [Alternate Keywords], page 624).

Qualifiers

`volatile` The optional `volatile` qualifier has no effect. All basic `asm` blocks are implicitly volatile.

`inline` If you use the `inline` qualifier, then for inlining purposes the size of the `asm` statement is taken as the smallest size possible (see Section 6.47.6 [Size of an asm], page 623).

Parameters

AssemblerInstructions

This is a literal string that specifies the assembler code. The string can contain any instructions recognized by the assembler, including directives. GCC does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler input.

You may place multiple assembler instructions together in a single `asm` string, separated by the characters normally used in assembly code for the system. A combination that works in most places is a newline to break the line, plus a tab character (written as '`\n\t`'). Some assemblers allow semicolons as a line separator. However, note that some assembler dialects use semicolons to start a comment.

Remarks

Using extended `asm` (see Section 6.47.2 [Extended Asm], page 569) typically produces smaller, safer, and more efficient code, and in most cases it is a better solution than basic `asm`. However, there are two situations where only basic `asm` can be used:

- Extended `asm` statements have to be inside a C function, so to write inline assembly language at file scope ("top-level"), outside of C functions, you must use basic `asm`. You can use this technique to emit assembler directives, define assembly language macros that can be invoked elsewhere in the file, or write entire functions in assembly language. Basic `asm` statements outside of functions may not use any qualifiers.
- Functions declared with the `naked` attribute also require basic `asm` (see Section 6.33 [Function Attributes], page 477).

Safely accessing C data and calling functions from basic `asm` is more complex than it may appear. To access C data, it is better to use extended `asm`.

Do not expect a sequence of `asm` statements to remain perfectly consecutive after compilation. If certain instructions need to remain consecutive in the output, put them in a single multi-instruction `asm` statement. Note that GCC's optimizers can move `asm` statements relative to other code, including across jumps.

`asm` statements may not perform jumps into other `asm` statements. GCC does not know about these jumps, and therefore cannot take account of them when deciding how to optimize. Jumps from `asm` to C labels are only supported in extended `asm`.

Under certain circumstances, GCC may duplicate (or remove duplicates of) your assembly code when optimizing. This can lead to unexpected duplicate symbol errors during compilation if your assembly code defines symbols or labels.

Warning: The C standards do not specify semantics for `asm`, making it a potential source of incompatibilities between compilers. These incompatibilities may not produce compiler warnings/errors.

GCC does not parse basic `asm`'s *AssemblerInstructions*, which means there is no way to communicate to the compiler what is happening inside them. GCC has no visibility of symbols in the `asm` and may discard them as unreferenced. It also does not know about side effects of the assembler code, such as modifications to memory or registers. Unlike

some compilers, GCC assumes that no changes to general purpose registers occur. This assumption may change in a future release.

To avoid complications from future changes to the semantics and the compatibility issues between compilers, consider replacing basic `asm` with extended `asm`. See How to convert from basic asm to extended asm (`https://gcc.gnu.org/wiki/ConvertBasicAsmToExtended`) for information about how to perform this conversion.

The compiler copies the assembler instructions in a basic `asm` verbatim to the assembly language output file, without processing dialects or any of the '`%`' operators that are available with extended `asm`. This results in minor differences between basic `asm` strings and extended `asm` templates. For example, to refer to registers you might use '`%eax`' in basic `asm` and '`%%eax`' in extended `asm`.

On targets such as x86 that support multiple assembler dialects, all basic `asm` blocks use the assembler dialect specified by the '`-masm`' command-line option (see Section 3.18.58 [x86 Options], page 401). Basic `asm` provides no mechanism to provide different assembler strings for different dialects.

For basic `asm` with non-empty assembler string GCC assumes the assembler block does not change any general purpose registers, but it may read or write any globally accessible variable.

Here is an example of basic `asm` for i386:

```
/* Note that this code will not compile with -masm=intel */
#define DebugBreak() asm("int $3")
```

6.47.2 Extended Asm - Assembler Instructions with C Expression Operands

With extended `asm` you can read and write C variables from assembler and perform jumps from assembler code to C labels. Extended `asm` syntax uses colons ('`:`') to delimit the operand parameters after the assembler template:

```
asm asm-qualifiers ( AssemblerTemplate
                   : OutputOperands
                   [ : InputOperands
                   [ : Clobbers ] ])

asm asm-qualifiers ( AssemblerTemplate
                     :
                     : InputOperands
                     : Clobbers
                     : GotoLabels)
```

where in the last form, *asm-qualifiers* contains `goto` (and in the first form, not).

The `asm` keyword is a GNU extension. When writing code that can be compiled with '`-ansi`' and the various '`-std`' options, use `__asm__` instead of `asm` (see Section 6.48 [Alternate Keywords], page 624).

Qualifiers

`volatile` The typical use of extended `asm` statements is to manipulate input values to produce output values. However, your `asm` statements may also produce side

effects. If so, you may need to use the `volatile` qualifier to disable certain optimizations. See [Volatile], page 571.

`inline` If you use the `inline` qualifier, then for inlining purposes the size of the `asm` statement is taken as the smallest size possible (see Section 6.47.6 [Size of an asm], page 623).

`goto` This qualifier informs the compiler that the `asm` statement may perform a jump to one of the labels listed in the *GotoLabels*. See [GotoLabels], page 582.

Parameters

AssemblerTemplate
This is a literal string that is the template for the assembler code. It is a combination of fixed text and tokens that refer to the input, output, and goto parameters. See [AssemblerTemplate], page 573.

OutputOperands
A comma-separated list of the C variables modified by the instructions in the *AssemblerTemplate*. An empty list is permitted. See [OutputOperands], page 574.

InputOperands
A comma-separated list of C expressions read by the instructions in the *AssemblerTemplate*. An empty list is permitted. See [InputOperands], page 578.

Clobbers A comma-separated list of registers or other values changed by the *AssemblerTemplate*, beyond those listed as outputs. An empty list is permitted. See [Clobbers and Scratch Registers], page 579.

GotoLabels
When you are using the `goto` form of `asm`, this section contains the list of all C labels to which the code in the *AssemblerTemplate* may jump. See [GotoLabels], page 582.

`asm` statements may not perform jumps into other `asm` statements, only to the listed *GotoLabels*. GCC's optimizers do not know about other jumps; therefore they cannot take account of them when deciding how to optimize.

The total number of input + output + goto operands is limited to 30.

Remarks

The `asm` statement allows you to include assembly instructions directly within C code. This may help you to maximize performance in time-sensitive code or to access assembly instructions that are not readily available to C programs.

Note that extended `asm` statements must be inside a function. Only basic `asm` may be outside functions (see Section 6.47.1 [Basic Asm], page 567). Functions declared with the `naked` attribute also require basic `asm` (see Section 6.33 [Function Attributes], page 477).

While the uses of `asm` are many and varied, it may help to think of an `asm` statement as a series of low-level instructions that convert input parameters to output parameters. So a simple (if not particularly useful) example for i386 using `asm` might look like this:

```
int src = 1;
int dst;

asm ("mov %1, %0\n\t"
    "add $1, %0"
    : "=r" (dst)
    : "r" (src));

printf("%d\n", dst);
```

This code copies `src` to `dst` and add 1 to `dst`.

6.47.2.1 Volatile

GCC's optimizers sometimes discard `asm` statements if they determine there is no need for the output variables. Also, the optimizers may move code out of loops if they believe that the code will always return the same result (i.e. none of its input values change between calls). Using the `volatile` qualifier disables these optimizations. `asm` statements that have no output operands, including `asm goto` statements, are implicitly volatile.

This i386 code demonstrates a case that does not use (or require) the `volatile` qualifier. If it is performing assertion checking, this code uses `asm` to perform the validation. Otherwise, `dwRes` is unreferenced by any code. As a result, the optimizers can discard the `asm` statement, which in turn removes the need for the entire `DoCheck` routine. By omitting the `volatile` qualifier when it isn't needed you allow the optimizers to produce the most efficient code possible.

```
void DoCheck(uint32_t dwSomeValue)
{
   uint32_t dwRes;

   // Assumes dwSomeValue is not zero.
   asm ("bsfl %1,%0"
     : "=r" (dwRes)
     : "r" (dwSomeValue)
     : "cc");

   assert(dwRes > 3);
}
```

The next example shows a case where the optimizers can recognize that the input (`dwSomeValue`) never changes during the execution of the function and can therefore move the `asm` outside the loop to produce more efficient code. Again, using the `volatile` qualifier disables this type of optimization.

```
void do_print(uint32_t dwSomeValue)
{
   uint32_t dwRes;

   for (uint32_t x=0; x < 5; x++)
   {
      // Assumes dwSomeValue is not zero.
```

```
        asm ("bsfl %1,%0"
          : "=r" (dwRes)
          : "r" (dwSomeValue)
          : "cc");

        printf("%u: %u %u\n", x, dwSomeValue, dwRes);
      }
    }
```

The following example demonstrates a case where you need to use the `volatile` qualifier. It uses the x86 `rdtsc` instruction, which reads the computer's time-stamp counter. Without the `volatile` qualifier, the optimizers might assume that the `asm` block will always return the same value and therefore optimize away the second call.

```
uint64_t msr;

asm volatile ( "rdtsc\n\t"    // Returns the time in EDX:EAX.
        "shl $32, %%rdx\n\t"  // Shift the upper bits left.
        "or %%rdx, %0"        // 'Or' in the lower bits.
        : "=a" (msr)
        :
        : "rdx");

printf("msr: %llx\n", msr);

// Do other work...

// Reprint the timestamp
asm volatile ( "rdtsc\n\t"    // Returns the time in EDX:EAX.
        "shl $32, %%rdx\n\t"  // Shift the upper bits left.
        "or %%rdx, %0"        // 'Or' in the lower bits.
        : "=a" (msr)
        :
        : "rdx");

printf("msr: %llx\n", msr);
```

GCC's optimizers do not treat this code like the non-volatile code in the earlier examples. They do not move it out of loops or omit it on the assumption that the result from a previous call is still valid.

Note that the compiler can move even `volatile` `asm` instructions relative to other code, including across jump instructions. For example, on many targets there is a system register that controls the rounding mode of floating-point operations. Setting it with a `volatile` `asm` statement, as in the following PowerPC example, does not work reliably.

```
asm volatile("mtfsf 255, %0" : : "f" (fpenv));
sum = x + y;
```

The compiler may move the addition back before the `volatile` `asm` statement. To make it work as expected, add an artificial dependency to the `asm` by referencing a variable in the subsequent code, for example:

```
asm volatile ("mtfsf 255,%1" : "=X" (sum) : "f" (fpenv));
sum = x + y;
```

Under certain circumstances, GCC may duplicate (or remove duplicates of) your assembly code when optimizing. This can lead to unexpected duplicate symbol errors during compilation if your `asm` code defines symbols or labels. Using '`%=`' (see [AssemblerTemplate], page 573) may help resolve this problem.

6.47.2.2 Assembler Template

An assembler template is a literal string containing assembler instructions. The compiler replaces tokens in the template that refer to inputs, outputs, and goto labels, and then outputs the resulting string to the assembler. The string can contain any instructions recognized by the assembler, including directives. GCC does not parse the assembler instructions themselves and does not know what they mean or even whether they are valid assembler input. However, it does count the statements (see Section 6.47.6 [Size of an asm], page 623).

You may place multiple assembler instructions together in a single `asm` string, separated by the characters normally used in assembly code for the system. A combination that works in most places is a newline to break the line, plus a tab character to move to the instruction field (written as '`\n\t`'). Some assemblers allow semicolons as a line separator. However, note that some assembler dialects use semicolons to start a comment.

Do not expect a sequence of `asm` statements to remain perfectly consecutive after compilation, even when you are using the `volatile` qualifier. If certain instructions need to remain consecutive in the output, put them in a single multi-instruction `asm` statement.

Accessing data from C programs without using input/output operands (such as by using global symbols directly from the assembler template) may not work as expected. Similarly, calling functions directly from an assembler template requires a detailed understanding of the target assembler and ABI.

Since GCC does not parse the assembler template, it has no visibility of any symbols it references. This may result in GCC discarding those symbols as unreferenced unless they are also listed as input, output, or goto operands.

Special format strings

In addition to the tokens described by the input, output, and goto operands, these tokens have special meanings in the assembler template:

'`%%`' Outputs a single '`%`' into the assembler code.

'`%=`' Outputs a number that is unique to each instance of the `asm` statement in the entire compilation. This option is useful when creating local labels and referring to them multiple times in a single template that generates multiple assembler instructions.

'`%{`'
'`%|`'
'`%}`' Outputs '`{`', '`|`', and '`}`' characters (respectively) into the assembler code. When unescaped, these characters have special meaning to indicate multiple assembler dialects, as described below.

Multiple assembler dialects in asm templates

On targets such as x86, GCC supports multiple assembler dialects. The '-masm' option controls which dialect GCC uses as its default for inline assembler. The target-specific documentation for the '-masm' option contains the list of supported dialects, as well as the default dialect if the option is not specified. This information may be important to understand, since assembler code that works correctly when compiled using one dialect will likely fail if compiled using another. See Section 3.18.58 [x86 Options], page 401.

If your code needs to support multiple assembler dialects (for example, if you are writing public headers that need to support a variety of compilation options), use constructs of this form:

```
{ dialect0 | dialect1 | dialect2... }
```

This construct outputs dialect0 when using dialect #0 to compile the code, dialect1 for dialect #1, etc. If there are fewer alternatives within the braces than the number of dialects the compiler supports, the construct outputs nothing.

For example, if an x86 compiler supports two dialects ('att', 'intel'), an assembler template such as this:

```
"bt{l %[Offset],%[Base] | %[Base],%[Offset]}; jc %l2"
```

is equivalent to one of

```
"btl %[Offset],%[Base] ; jc %l2"    /* att dialect */
"bt %[Base],%[Offset]; jc %l2"      /* intel dialect */
```

Using that same compiler, this code:

```
"xchg{l}\t{%%}ebx, %1"
```

corresponds to either

```
"xchgl\t%%ebx, %1"                  /* att dialect */
"xchg\tebx, %1"                     /* intel dialect */
```

There is no support for nesting dialect alternatives.

6.47.2.3 Output Operands

An asm statement has zero or more output operands indicating the names of C variables modified by the assembler code.

In this i386 example, old (referred to in the template string as %0) and *Base (as %1) are outputs and Offset (%2) is an input:

```
bool old;

__asm__ ("btsl %2,%1\n\t" // Turn on zero-based bit #Offset in Base.
         "sbb %0,%0"      // Use the CF to calculate old.
   : "=r" (old), "+rm" (*Base)
   : "Ir" (Offset)
   : "cc");

return old;
```

Operands are separated by commas. Each operand has this format:

[[*asmSymbolicName*]] *constraint* (*cvariablename*)

asmSymbolicName

Specifies a symbolic name for the operand. Reference the name in the assembler template by enclosing it in square brackets (i.e. '`%[Value]`'). The scope of the name is the `asm` statement that contains the definition. Any valid C variable name is acceptable, including names already defined in the surrounding code. No two operands within the same `asm` statement can use the same symbolic name.

When not using an *asmSymbolicName*, use the (zero-based) position of the operand in the list of operands in the assembler template. For example if there are three output operands, use '`%0`' in the template to refer to the first, '`%1`' for the second, and '`%2`' for the third.

constraint A string constant specifying constraints on the placement of the operand; See Section 6.47.3 [Constraints], page 586, for details.

Output constraints must begin with either '`=`' (a variable overwriting an existing value) or '`+`' (when reading and writing). When using '`=`', do not assume the location contains the existing value on entry to the `asm`, except when the operand is tied to an input; see [Input Operands], page 578.

After the prefix, there must be one or more additional constraints (see Section 6.47.3 [Constraints], page 586) that describe where the value resides. Common constraints include '`r`' for register and '`m`' for memory. When you list more than one possible location (for example, `"=rm"`), the compiler chooses the most efficient one based on the current context. If you list as many alternates as the `asm` statement allows, you permit the optimizers to produce the best possible code. If you must use a specific register, but your Machine Constraints do not provide sufficient control to select the specific register you want, local register variables may provide a solution (see Section 6.47.5.2 [Local Register Variables], page 622).

cvariablename

Specifies a C lvalue expression to hold the output, typically a variable name. The enclosing parentheses are a required part of the syntax.

When the compiler selects the registers to use to represent the output operands, it does not use any of the clobbered registers (see [Clobbers and Scratch Registers], page 579).

Output operand expressions must be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. For output expressions that are not directly addressable (for example a bit-field), the constraint must allow a register. In that case, GCC uses the register as the output of the `asm`, and then stores that register into the output.

Operands using the '`+`' constraint modifier count as two operands (that is, both as input and output) towards the total maximum of 30 operands per `asm` statement.

Use the '`&`' constraint modifier (see Section 6.47.3.3 [Modifiers], page 589) on all output operands that must not overlap an input. Otherwise, GCC may allocate the output operand in the same register as an unrelated input operand, on the assumption that the assembler code consumes its inputs before producing outputs. This assumption may be false if the assembler code actually consists of more than one instruction.

The same problem can occur if one output parameter (*a*) allows a register constraint and another output parameter (*b*) allows a memory constraint. The code generated by GCC to access the memory address in *b* can contain registers which *might* be shared by *a*, and GCC considers those registers to be inputs to the asm. As above, GCC assumes that such input registers are consumed before any outputs are written. This assumption may result in incorrect behavior if the `asm` statement writes to *a* before using *b*. Combining the '`&`' modifier with the register constraint on *a* ensures that modifying *a* does not affect the address referenced by *b*. Otherwise, the location of *b* is undefined if *a* is modified before using *b*.

`asm` supports operand modifiers on operands (for example '`%k2`' instead of simply '`%2`'). Typically these qualifiers are hardware dependent. The list of supported modifiers for x86 is found at [x86Operandmodifiers], page 583.

If the C code that follows the `asm` makes no use of any of the output operands, use `volatile` for the `asm` statement to prevent the optimizers from discarding the `asm` statement as unneeded (see [Volatile], page 571).

This code makes no use of the optional *asmSymbolicName*. Therefore it references the first output operand as `%0` (were there a second, it would be `%1`, etc). The number of the first input operand is one greater than that of the last output operand. In this i386 example, that makes `Mask` referenced as `%1`:

```
uint32_t Mask = 1234;
uint32_t Index;

  asm ("bsfl %1, %0"
     : "=r" (Index)
     : "r" (Mask)
     : "cc");
```

That code overwrites the variable `Index` ('`=`'), placing the value in a register ('`r`'). Using the generic '`r`' constraint instead of a constraint for a specific register allows the compiler to pick the register to use, which can result in more efficient code. This may not be possible if an assembler instruction requires a specific register.

The following i386 example uses the *asmSymbolicName* syntax. It produces the same result as the code above, but some may consider it more readable or more maintainable since reordering index numbers is not necessary when adding or removing operands. The names `aIndex` and `aMask` are only used in this example to emphasize which names get used where. It is acceptable to reuse the names `Index` and `Mask`.

```
uint32_t Mask = 1234;
uint32_t Index;

  asm ("bsfl %[aMask], %[aIndex]"
     : [aIndex] "=r" (Index)
     : [aMask] "r" (Mask)
     : "cc");
```

Here are some more examples of output operands.

```
uint32_t c = 1;
uint32_t d;
```

```
uint32_t *e = &c;

asm ("mov %[e], %[d]"
   : [d] "=rm" (d)
   : [e] "rm" (*e));
```

Here, `d` may either be in a register or in memory. Since the compiler might already have the current value of the `uint32_t` location pointed to by `e` in a register, you can enable it to choose the best location for `d` by specifying both constraints.

6.47.2.4 Flag Output Operands

Some targets have a special register that holds the "flags" for the result of an operation or comparison. Normally, the contents of that register are either unmodifed by the asm, or the `asm` statement is considered to clobber the contents.

On some targets, a special form of output operand exists by which conditions in the flags register may be outputs of the asm. The set of conditions supported are target specific, but the general rule is that the output variable must be a scalar integer, and the value is boolean. When supported, the target defines the preprocessor symbol `__GCC_ASM_FLAG_OUTPUTS__`.

Because of the special nature of the flag output operands, the constraint may not include alternatives.

Most often, the target has only one flags register, and thus is an implied operand of many instructions. In this case, the operand should not be referenced within the assembler template via `%0` etc, as there's no corresponding text in the assembly language.

x86 family The flag output constraints for the x86 family are of the form '`=@cc`*cond*' where *cond* is one of the standard conditions defined in the ISA manual for `jcc` or `set`*cc*.

`a`	"above" or unsigned greater than
`ae`	"above or equal" or unsigned greater than or equal
`b`	"below" or unsigned less than
`be`	"below or equal" or unsigned less than or equal
`c`	carry flag set
`e` `z`	"equal" or zero flag set
`g`	signed greater than
`ge`	signed greater than or equal
`l`	signed less than
`le`	signed less than or equal
`o`	overflow flag set
`p`	parity flag set
`s`	sign flag set

`na`
`nae`
`nb`
`nbe`
`nc`
`ne`
`ng`
`nge`
`nl`
`nle`
`no`
`np`
`ns`
`nz` "not" *flag*, or inverted versions of those above

6.47.2.5 Input Operands

Input operands make values from C variables and expressions available to the assembly code.

Operands are separated by commas. Each operand has this format:

[`[`*asmSymbolicName*`]`] *constraint* `(`*cexpression*`)`

asmSymbolicName
: Specifies a symbolic name for the operand. Reference the name in the assembler template by enclosing it in square brackets (i.e. '`%[Value]`'). The scope of the name is the `asm` statement that contains the definition. Any valid C variable name is acceptable, including names already defined in the surrounding code. No two operands within the same `asm` statement can use the same symbolic name.

  When not using an *asmSymbolicName*, use the (zero-based) position of the operand in the list of operands in the assembler template. For example if there are two output operands and three inputs, use '`%2`' in the template to refer to the first input operand, '`%3`' for the second, and '`%4`' for the third.

constraint
: A string constant specifying constraints on the placement of the operand; See Section 6.47.3 [Constraints], page 586, for details.

  Input constraint strings may not begin with either '`=`' or '`+`'. When you list more than one possible location (for example, '`"irm"`'), the compiler chooses the most efficient one based on the current context. If you must use a specific register, but your Machine Constraints do not provide sufficient control to select the specific register you want, local register variables may provide a solution (see Section 6.47.5.2 [Local Register Variables], page 622).

  Input constraints can also be digits (for example, `"0"`). This indicates that the specified input must be in the same place as the output constraint at the (zero-based) index in the output constraint list. When using *asmSymbolicName* syntax for the output operands, you may use these names (enclosed in brackets '`[]`') instead of digits.

cexpression

This is the C variable or expression being passed to the `asm` statement as input. The enclosing parentheses are a required part of the syntax.

When the compiler selects the registers to use to represent the input operands, it does not use any of the clobbered registers (see [Clobbers and Scratch Registers], page 579).

If there are no output operands but there are input operands, place two consecutive colons where the output operands would go:

```
__asm__ ("some instructions"
   : /* No outputs. */
   : "r" (Offset / 8));
```

Warning: Do *not* modify the contents of input-only operands (except for inputs tied to outputs). The compiler assumes that on exit from the `asm` statement these operands contain the same values as they had before executing the statement. It is *not* possible to use clobbers to inform the compiler that the values in these inputs are changing. One common work-around is to tie the changing input variable to an output variable that never gets used. Note, however, that if the code that follows the `asm` statement makes no use of any of the output operands, the GCC optimizers may discard the `asm` statement as unneeded (see [Volatile], page 571).

`asm` supports operand modifiers on operands (for example '`%k2`' instead of simply '`%2`'). Typically these qualifiers are hardware dependent. The list of supported modifiers for x86 is found at [x86Operandmodifiers], page 583.

In this example using the fictitious `combine` instruction, the constraint `"0"` for input operand 1 says that it must occupy the same location as output operand 0. Only input operands may use numbers in constraints, and they must each refer to an output operand. Only a number (or the symbolic assembler name) in the constraint can guarantee that one operand is in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they are in the same place in the generated assembler code.

```
asm ("combine %2, %0"
   : "=r" (foo)
   : "0" (foo), "g" (bar));
```

Here is an example using symbolic names.

```
asm ("cmoveq %1, %2, %[result]"
   : [result] "=r"(result)
   : "r" (test), "r" (new), "[result]" (old));
```

6.47.2.6 Clobbers and Scratch Registers

While the compiler is aware of changes to entries listed in the output operands, the inline `asm` code may modify more than just the outputs. For example, calculations may require additional registers, or the processor may overwrite a register as a side effect of a particular assembler instruction. In order to inform the compiler of these changes, list them in the clobber list. Clobber list items are either register names or the special clobbers (listed below). Each clobber list item is a string constant enclosed in double quotes and separated by commas.

Clobber descriptions may not in any way overlap with an input or output operand. For example, you may not have an operand describing a register class with one member when

listing that register in the clobber list. Variables declared to live in specific registers (see Section 6.47.5 [Explicit Register Variables], page 621) and used as `asm` input or output operands must have no part mentioned in the clobber description. In particular, there is no way to specify that input operands get modified without also specifying them as output operands.

When the compiler selects which registers to use to represent input and output operands, it does not use any of the clobbered registers. As a result, clobbered registers are available for any use in the assembler code.

Another restriction is that the clobber list should not contain the stack pointer register. This is because the compiler requires the value of the stack pointer to be the same after an `asm` statement as it was on entry to the statement. However, previous versions of GCC did not enforce this rule and allowed the stack pointer to appear in the list, with unclear semantics. This behavior is deprecated and listing the stack pointer may become an error in future versions of GCC.

Here is a realistic example for the VAX showing the use of clobbered registers:

```
asm volatile ("movc3 %0, %1, %2"
                    : /* No outputs. */
                    : "g" (from), "g" (to), "g" (count)
                    : "r0", "r1", "r2", "r3", "r4", "r5", "memory");
```

Also, there are two special clobber arguments:

`"cc"` The `"cc"` clobber indicates that the assembler code modifies the flags register. On some machines, GCC represents the condition codes as a specific hardware register; `"cc"` serves to name this register. On other machines, condition code handling is different, and specifying `"cc"` has no effect. But it is valid no matter what the target.

`"memory"` The `"memory"` clobber tells the compiler that the assembly code performs memory reads or writes to items other than those listed in the input and output operands (for example, accessing the memory pointed to by one of the input parameters). To ensure memory contains correct values, GCC may need to flush specific register values to memory before executing the `asm`. Further, the compiler does not assume that any values read from memory before an `asm` remain unchanged after that `asm`; it reloads them as needed. Using the `"memory"` clobber effectively forms a read/write memory barrier for the compiler.

Note that this clobber does not prevent the *processor* from doing speculative reads past the `asm` statement. To prevent that, you need processor-specific fence instructions.

Flushing registers to memory has performance implications and may be an issue for time-sensitive code. You can provide better information to GCC to avoid this, as shown in the following examples. At a minimum, aliasing rules allow GCC to know what memory *doesn't* need to be flushed.

Here is a fictitious sum of squares instruction, that takes two pointers to floating point values in memory and produces a floating point register output. Notice that `x`, and `y` both appear twice in the `asm` parameters, once to specify memory accessed, and once to specify a base register used by the `asm`. You won't normally be wasting a register by doing this as

GCC can use the same register for both purposes. However, it would be foolish to use both `%1` and `%3` for `x` in this `asm` and expect them to be the same. In fact, `%3` may well not be a register. It might be a symbolic memory reference to the object pointed to by `x`.

```
asm ("sumsq %0, %1, %2"
     : "+f" (result)
     : "r" (x), "r" (y), "m" (*x), "m" (*y));
```

Here is a fictitious `*z++ = *x++ * *y++` instruction. Notice that the `x`, `y` and `z` pointer registers must be specified as input/output because the `asm` modifies them.

```
asm ("vecmul %0, %1, %2"
     : "+r" (z), "+r" (x), "+r" (y), "=m" (*z)
     : "m" (*x), "m" (*y));
```

An x86 example where the string memory argument is of unknown length.

```
asm("repne scasb"
    : "=c" (count), "+D" (p)
    : "m" (*(const char (*)[]) p), "0" (-1), "a" (0));
```

If you know the above will only be reading a ten byte array then you could instead use a memory input like: `"m" (*(const char (*)[10]) p)`.

Here is an example of a PowerPC vector scale implemented in assembly, complete with vector and condition code clobbers, and some initialized offset registers that are unchanged by the `asm`.

```
void
dscal (size_t n, double *x, double alpha)
{
  asm ("/* lots of asm here */"
       : "+m" (*(double (*)[n]) x), "+&r" (n), "+b" (x)
       : "d" (alpha), "b" (32), "b" (48), "b" (64),
         "b" (80), "b" (96), "b" (112)
       : "cr0",
         "vs32","vs33","vs34","vs35","vs36","vs37","vs38","vs39",
         "vs40","vs41","vs42","vs43","vs44","vs45","vs46","vs47");
}
```

Rather than allocating fixed registers via clobbers to provide scratch registers for an `asm` statement, an alternative is to define a variable and make it an early-clobber output as with `a2` and `a3` in the example below. This gives the compiler register allocator more freedom. You can also define a variable and make it an output tied to an input as with `a0` and `a1`, tied respectively to `ap` and `lda`. Of course, with tied outputs your `asm` can't use the input value after modifying the output register since they are one and the same register. What's more, if you omit the early-clobber on the output, it is possible that GCC might allocate the same register to another of the inputs if GCC could prove they had the same value on entry to the `asm`. This is why `a1` has an early-clobber. Its tied input, `lda` might conceivably be known to have the value 16 and without an early-clobber share the same register as `%11`. On the other hand, `ap` can't be the same as any of the other inputs, so an early-clobber on `a0` is not needed. It is also not desirable in this case. An early-clobber on `a0` would cause GCC to allocate a separate register for the `"m" (*(const double (*)[]) ap)` input. Note that tying an input to an output is the way to set up an initialized temporary register modified by an `asm` statement. An input not tied to an output is assumed by GCC to be unchanged, for example `"b" (16)` below sets up `%11` to 16, and GCC might use that register in following code if the value 16 happened to be needed. You can even use a normal `asm` output for a scratch if all inputs that might share the same register are consumed before

the scratch is used. The VSX registers clobbered by the `asm` statement could have used this technique except for GCC's limit on the number of `asm` parameters.

```
static void
dgemv_kernel_4x4 (long n, const double *ap, long lda,
                  const double *x, double *y, double alpha)
{
  double *a0;
  double *a1;
  double *a2;
  double *a3;

  __asm__
    (
     /* lots of asm here */
     "#n=%1 ap=%8=%12 lda=%13 x=%7=%10 y=%0=%2 alpha=%9 o16=%11\n"
     "#a0=%3 a1=%4 a2=%5 a3=%6"
     :
       "+m" (*(double (*)[n]) y),
       "+&r" (n), // 1
       "+b" (y), // 2
       "=b" (a0), // 3
       "=&b" (a1), // 4
       "=&b" (a2), // 5
       "=&b" (a3) // 6
     :
       "m" (*(const double (*)[n]) x),
       "m" (*(const double (*)[]) ap),
       "d" (alpha), // 9
       "r" (x), // 10
       "b" (16), // 11
       "3" (ap), // 12
       "4" (lda) // 13
     :
       "cr0",
       "vs32","vs33","vs34","vs35","vs36","vs37",
       "vs40","vs41","vs42","vs43","vs44","vs45","vs46","vs47"
     );
}
```

6.47.2.7 Goto Labels

`asm goto` allows assembly code to jump to one or more C labels. The *GotoLabels* section in an `asm goto` statement contains a comma-separated list of all C labels to which the assembler code may jump. GCC assumes that `asm` execution falls through to the next statement (if this is not the case, consider using the `__builtin_unreachable` intrinsic after the `asm` statement). Optimization of `asm goto` may be improved by using the `hot` and `cold` label attributes (see Section 6.36 [Label Attributes], page 557).

An `asm goto` statement cannot have outputs. This is due to an internal restriction of the compiler: control transfer instructions cannot have outputs. If the assembler code does modify anything, use the `"memory"` clobber to force the optimizers to flush all register values to memory and reload them if necessary after the `asm` statement.

Also note that an `asm goto` statement is always implicitly considered volatile.

To reference a label in the assembler template, prefix it with '`%l`' (lowercase '`L`') followed by its (zero-based) position in *GotoLabels* plus the number of input operands. For example,

if the `asm` has three inputs and references two labels, refer to the first label as '`%l3`' and the second as '`%l4`').

Alternately, you can reference labels using the actual C label name enclosed in brackets. For example, to reference a label named `carry`, you can use '`%l[carry]`'. The label must still be listed in the *GotoLabels* section when using this approach.

Here is an example of `asm goto` for i386:

```
asm goto (
    "btl %1, %0\n\t"
    "jc %l2"
    : /* No outputs. */
    : "r" (p1), "r" (p2)
    : "cc"
    : carry);

return 0;

carry:
return 1;
```

The following example shows an `asm goto` that uses a memory clobber.

```
int frob(int x)
{
  int y;
  asm goto ("frob %%r5, %1; jc %l[error]; mov (%2), %%r5"
            : /* No outputs. */
            : "r"(x), "r"(&y)
            : "r5", "memory"
            : error);
  return y;
error:
  return -1;
}
```

6.47.2.8 x86 Operand Modifiers

References to input, output, and goto operands in the assembler template of extended `asm` statements can use modifiers to affect the way the operands are formatted in the code output to the assembler. For example, the following code uses the '`h`' and '`b`' modifiers for x86:

```
uint16_t  num;
asm volatile ("xchg %h0, %b0" : "+a" (num) );
```

These modifiers generate this assembler code:

```
xchg %ah, %al
```

The rest of this discussion uses the following code for illustrative purposes.

```
int main()
{
```

```
      int iInt = 1;

   top:

      asm volatile goto ("some assembler instructions here"
      : /* No outputs. */
      : "q" (iInt), "X" (sizeof(unsigned char) + 1), "i" (42)
      : /* No clobbers. */
      : top);
   }
```

With no modifiers, this is what the output from the operands would be for the 'att' and 'intel' dialects of assembler:

Operand	'att'	'intel'
%0	%eax	eax
%1	$2	2
%3	$.L3	OFFSET FLAT:.L3

The table below shows the list of supported modifiers and their effects.

Modifier	**Description**	**Operand**	'att'	'intel'
a	Print an absolute memory reference.	%A0	*%rax	rax
b	Print the QImode name of the register.	%b0	%al	al
c	Require a constant operand and print the constant expression with no punctuation.	%c1	2	2
E	Print the address in Double Integer (DImode) mode (8 bytes) when the target is 64-bit. Otherwise mode is unspecified (VOIDmode).	%E1	%(rax)	[rax]
h	Print the QImode name for a "high" register.	%h0	%ah	ah
H	Add 8 bytes to an offsettable memory reference. Useful when accessing the high 8 bytes of SSE values. For a memref in (%rax), it generates	%H0	8(%rax)	8[rax]
k	Print the SImode name of the register.	%k0	%eax	eax
l	Print the label name with no punctuation.	%l3	.L3	.L3
p	Print raw symbol name (without syntax-specific prefixes).	%p2	42	42
P	If used for a function, print the PLT suffix and generate PIC code. For example, emit foo@PLT instead of 'foo' for the function foo(). If used for a constant, drop all syntax-specific prefixes and issue the bare constant. See p above.			

q	Print the DImode name of the register.	%q0	%rax	rax
w	Print the HImode name of the register.	%w0	%ax	ax
z	Print the opcode suffix for the size of the current integer operand (one of b/w/l/q).	%z0	l	

V is a special modifier which prints the name of the full integer register without %.

6.47.2.9 x86 Floating-Point asm Operands

On x86 targets, there are several rules on the usage of stack-like registers in the operands of an asm. These rules apply only to the operands that are stack-like registers:

1. Given a set of input registers that die in an asm, it is necessary to know which are implicitly popped by the asm, and which must be explicitly popped by GCC.

   An input register that is implicitly popped by the asm must be explicitly clobbered, unless it is constrained to match an output operand.

2. For any input register that is implicitly popped by an asm, it is necessary to know how to adjust the stack to compensate for the pop. If any non-popped input is closer to the top of the reg-stack than the implicitly popped register, it would not be possible to know what the stack looked like—it's not clear how the rest of the stack "slides up".

   All implicitly popped input registers must be closer to the top of the reg-stack than any input that is not implicitly popped.

   It is possible that if an input dies in an asm, the compiler might use the input register for an output reload. Consider this example:

   ```
   asm ("foo" : "=t" (a) : "f" (b));
   ```

   This code says that input b is not popped by the asm, and that the asm pushes a result onto the reg-stack, i.e., the stack is one deeper after the asm than it was before. But, it is possible that reload may think that it can use the same register for both the input and the output.

   To prevent this from happening, if any input operand uses the 'f' constraint, all output register constraints must use the '&' early-clobber modifier.

   The example above is correctly written as:

   ```
   asm ("foo" : "=&t" (a) : "f" (b));
   ```

3. Some operands need to be in particular places on the stack. All output operands fall in this category—GCC has no other way to know which registers the outputs appear in unless you indicate this in the constraints.

   Output operands must specifically indicate which register an output appears in after an asm. '=f' is not allowed: the operand constraints must select a class with a single register.

4. Output operands may not be "inserted" between existing stack registers. Since no 387 opcode uses a read/write operand, all output operands are dead before the asm, and are pushed by the asm. It makes no sense to push anywhere but the top of the reg-stack.

   Output operands must start at the top of the reg-stack: output operands may not "skip" a register.

5. Some asm statements may need extra stack space for internal calculations. This can be guaranteed by clobbering stack registers unrelated to the inputs and outputs.

This asm takes one input, which is internally popped, and produces two outputs.

```
asm ("fsincos" : "=t" (cos), "=u" (sin) : "0" (inp));
```

This asm takes two inputs, which are popped by the fyl2xp1 opcode, and replaces them with one output. The st(1) clobber is necessary for the compiler to know that fyl2xp1 pops both inputs.

```
asm ("fyl2xp1" : "=t" (result) : "0" (x), "u" (y) : "st(1)");
```

6.47.3 Constraints for asm Operands

Here are specific details on what constraint letters you can use with asm operands. Constraints can say whether an operand may be in a register, and which kinds of register; whether the operand can be a memory reference, and which kinds of address; whether the operand may be an immediate constant, and which possible values it may have. Constraints can also require two operands to match. Side-effects aren't allowed in operands of inline asm, unless '<' or '>' constraints are used, because there is no guarantee that the side effects will happen exactly once in an instruction that can update the addressing register.

6.47.3.1 Simple Constraints

The simplest kind of constraint is a string full of letters, each of which describes one kind of operand that is permitted. Here are the letters that are allowed:

whitespace

Whitespace characters are ignored and can be inserted at any position except the first. This enables each alternative for different operands to be visually aligned in the machine description even if they have different number of constraints and modifiers.

'm' A memory operand is allowed, with any kind of address that the machine supports in general. Note that the letter used for the general memory constraint can be re-defined by a back end using the TARGET_MEM_CONSTRAINT macro.

'o' A memory operand is allowed, but only if the address is *offsettable*. This means that adding a small integer (actually, the width in bytes of the operand, as determined by its machine mode) may be added to the address and the result is also a valid memory address.

For example, an address which is constant is offsettable; so is an address that is the sum of a register and a constant (as long as a slightly larger constant is also within the range of address-offsets supported by the machine); but an autoincrement or autodecrement address is not offsettable. More complicated indirect/indexed addresses may or may not be offsettable depending on the other addressing modes that the machine supports.

Note that in an output operand which can be matched by another operand, the constraint letter 'o' is valid only when accompanied by both '<' (if the target machine has predecrement addressing) and '>' (if the target machine has preincrement addressing).

'V' A memory operand that is not offsettable. In other words, anything that would fit the 'm' constraint but not the 'o' constraint.

'<' A memory operand with autodecrement addressing (either predecrement or postdecrement) is allowed. In inline `asm` this constraint is only allowed if the operand is used exactly once in an instruction that can handle the side effects. Not using an operand with '<' in constraint string in the inline `asm` pattern at all or using it in multiple instructions isn't valid, because the side effects wouldn't be performed or would be performed more than once. Furthermore, on some targets the operand with '<' in constraint string must be accompanied by special instruction suffixes like `%U0` instruction suffix on PowerPC or `%P0` on IA-64.

'>' A memory operand with autoincrement addressing (either preincrement or postincrement) is allowed. In inline `asm` the same restrictions as for '<' apply.

'r' A register operand is allowed provided that it is in a general register.

'i' An immediate integer operand (one with constant value) is allowed. This includes symbolic constants whose values will be known only at assembly time or later.

'n' An immediate integer operand with a known numeric value is allowed. Many systems cannot support assembly-time constants for operands less than a word wide. Constraints for these operands should use 'n' rather than 'i'.

'I', 'J', 'K', ... 'P'
Other letters in the range 'I' through 'P' may be defined in a machine-dependent fashion to permit immediate integer operands with explicit integer values in specified ranges. For example, on the 68000, 'I' is defined to stand for the range of values 1 to 8. This is the range permitted as a shift count in the shift instructions.

'E' An immediate floating operand (expression code `const_double`) is allowed, but only if the target floating point format is the same as that of the host machine (on which the compiler is running).

'F' An immediate floating operand (expression code `const_double` or `const_vector`) is allowed.

'G', 'H' 'G' and 'H' may be defined in a machine-dependent fashion to permit immediate floating operands in particular ranges of values.

's' An immediate integer operand whose value is not an explicit integer is allowed.

This might appear strange; if an insn allows a constant operand with a value not known at compile time, it certainly must allow any known value. So why use 's' instead of 'i'? Sometimes it allows better code to be generated.

For example, on the 68000 in a fullword instruction it is possible to use an immediate operand; but if the immediate value is between −128 and 127, better code results from loading the value into a register and using the register. This is because the load into the register can be done with a 'moveq' instruction. We arrange for this to happen by defining the letter 'K' to mean "any integer outside the range −128 to 127", and then specifying 'Ks' in the operand constraints.

'g' Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

'X' Any operand whatsoever is allowed.

'0', '1', '2', ... '9'
An operand that matches the specified operand number is allowed. If a digit is used together with letters within the same alternative, the digit should come last.

This number is allowed to be more than a single digit. If multiple digits are encountered consecutively, they are interpreted as a single decimal integer. There is scant chance for ambiguity, since to-date it has never been desirable that '10' be interpreted as matching either operand 1 *or* operand 0. Should this be desired, one can use multiple alternatives instead.

This is called a *matching constraint* and what it really means is that the assembler has only a single operand that fills two roles which `asm` distinguishes. For example, an add instruction uses two input operands and an output operand, but on most CISC machines an add instruction really has only two operands, one of them an input-output operand:

```
addl #35,r12
```

Matching constraints are used in these circumstances. More precisely, the two operands that match must include one input-only operand and one output-only operand. Moreover, the digit must be a smaller number than the number of the operand that uses it in the constraint.

'p' An operand that is a valid memory address is allowed. This is for "load address" and "push address" instructions.

'p' in the constraint must be accompanied by `address_operand` as the predicate in the `match_operand`. This predicate interprets the mode specified in the `match_operand` as the mode of the memory reference for which the address would be valid.

other-letters
Other letters can be defined in machine-dependent fashion to stand for particular classes of registers or other arbitrary operand types. 'd', 'a' and 'f' are defined on the 68000/68020 to stand for data, address and floating point registers.

6.47.3.2 Multiple Alternative Constraints

Sometimes a single instruction has multiple alternative sets of possible operands. For example, on the 68000, a logical-or instruction can combine register or an immediate value into memory, or it can combine any kind of operand into a register; but it cannot combine one memory location into another.

These constraints are represented as multiple alternatives. An alternative can be described by a series of letters for each operand. The overall constraint for an operand is made from the letters for this operand from the first alternative, a comma, the letters for this operand from the second alternative, a comma, and so on until the last alternative. All operands for a single instruction must have the same number of alternatives.

So the first alternative for the 68000's logical-or could be written as `"+m" (output) : "ir" (input)`. The second could be `"+r" (output): "irm" (input)`. However, the fact that two memory locations cannot be used in a single instruction prevents simply using `"+rm" (output) : "irm" (input)`. Using multi-alternatives, this might be written as `"+m,r" (output) : "ir,irm" (input)`. This describes all the available alternatives to the compiler, allowing it to choose the most efficient one for the current conditions.

There is no way within the template to determine which alternative was chosen. However you may be able to wrap your `asm` statements with builtins such as `__builtin_constant_p` to achieve the desired results.

6.47.3.3 Constraint Modifier Characters

Here are constraint modifier characters.

'`=`' Means that this operand is written to by this instruction: the previous value is discarded and replaced by new data.

'`+`' Means that this operand is both read and written by the instruction.

When the compiler fixes up the operands to satisfy the constraints, it needs to know which operands are read by the instruction and which are written by it. '`=`' identifies an operand which is only written; '`+`' identifies an operand that is both read and written; all other operands are assumed to only be read.

If you specify '`=`' or '`+`' in a constraint, you put it in the first character of the constraint string.

'`&`' Means (in a particular alternative) that this operand is an *earlyclobber* operand, which is written before the instruction is finished using the input operands. Therefore, this operand may not lie in a register that is read by the instruction or as part of any memory address.

'`&`' applies only to the alternative in which it is written. In constraints with multiple alternatives, sometimes one alternative requires '`&`' while others do not. See, for example, the '`movdf`' insn of the 68000.

A operand which is read by the instruction can be tied to an earlyclobber operand if its only use as an input occurs before the early result is written. Adding alternatives of this form often allows GCC to produce better code when only some of the read operands can be affected by the earlyclobber. See, for example, the '`mulsi3`' insn of the ARM.

Furthermore, if the *earlyclobber* operand is also a read/write operand, then that operand is written only after it's used.

'`&`' does not obviate the need to write '`=`' or '`+`'. As *earlyclobber* operands are always written, a read-only *earlyclobber* operand is ill-formed and will be rejected by the compiler.

'`%`' Declares the instruction to be commutative for this operand and the following operand. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. '`%`' applies to all alternatives and must appear as the first character in the constraint. Only read-only operands can use '`%`'.

GCC can only handle one commutative pair in an asm; if you use more, the compiler may fail. Note that you need not use the modifier if the two alternatives are strictly identical; this would only waste time in the reload pass.

6.47.3.4 Constraints for Particular Machines

Whenever possible, you should use the general-purpose constraint letters in `asm` arguments, since they will convey meaning more readily to people reading your code. Failing that, use the constraint letters that usually have very similar meanings across architectures. The most commonly used constraints are '`m`' and '`r`' (for memory and general-purpose registers respectively; see Section 6.47.3.1 [Simple Constraints], page 586), and '`I`', usually the letter indicating the most common immediate-constant format.

Each architecture defines additional constraints. These constraints are used by the compiler itself for instruction generation, as well as for `asm` statements; therefore, some of the constraints are not particularly useful for `asm`. Here is a summary of some of the machine-dependent constraints available on some particular machines; it includes both constraints that are useful for `asm` and constraints that aren't. The compiler source file mentioned in the table heading for each architecture is the definitive reference for the meanings of that architecture's constraints.

AArch64 family—'`config/aarch64/constraints.md`'

`k`	The stack pointer register (`SP`)
`w`	Floating point register, Advanced SIMD vector register or SVE vector register
`Upl`	One of the low eight SVE predicate registers (`P0` to `P7`)
`Upa`	Any of the SVE predicate registers (`P0` to `P15`)
`I`	Integer constant that is valid as an immediate operand in an `ADD` instruction
`J`	Integer constant that is valid as an immediate operand in a `SUB` instruction (once negated)
`K`	Integer constant that can be used with a 32-bit logical instruction
`L`	Integer constant that can be used with a 64-bit logical instruction
`M`	Integer constant that is valid as an immediate operand in a 32-bit `MOV` pseudo instruction. The `MOV` may be assembled to one of several different machine instructions depending on the value
`N`	Integer constant that is valid as an immediate operand in a 64-bit `MOV` pseudo instruction
`S`	An absolute symbolic address or a label reference
`Y`	Floating point constant zero
`Z`	Integer constant zero
`Ush`	The high part (bits 12 and upwards) of the pc-relative address of a symbol within 4GB of the instruction

`Q` A memory address which uses a single base register with no offset

`Ump` A memory address suitable for a load/store pair instruction in SI, DI, SF and DF modes

AMD GCN —'`config/gcn/constraints.md`'

`I` Immediate integer in the range −16 to 64

`J` Immediate 16-bit signed integer

`Kf` Immediate constant −1

`L` Immediate 15-bit unsigned integer

`A` Immediate constant that can be inlined in an instruction encoding: integer −16..64, or float 0.0, +/−0.5, +/−1.0, +/−2.0, +/−4.0, 1.0/(2.0*PI)

`B` Immediate 32-bit signed integer that can be attached to an instruction encoding

`C` Immediate 32-bit integer in range −16..4294967295 (i.e. 32-bit unsigned integer or '`A`' constraint)

`DA` Immediate 64-bit constant that can be split into two '`A`' constants

`DB` Immediate 64-bit constant that can be split into two '`B`' constants

`U` Any `unspec`

`Y` Any `symbol_ref` or `label_ref`

`v` VGPR register

`Sg` SGPR register

`SD` SGPR registers valid for instruction destinations, including VCC, M0 and EXEC

`SS` SGPR registers valid for instruction sources, including VCC, M0, EXEC and SCC

`Sm` SGPR registers valid as a source for scalar memory instructions (excludes M0 and EXEC)

`Sv` SGPR registers valid as a source or destination for vector instructions (excludes EXEC)

`ca` All condition registers: SCC, VCCZ, EXECZ

`cs` Scalar condition register: SCC

`cV` Vector condition register: VCC, VCC_LO, VCC_HI

`e` EXEC register (EXEC_LO and EXEC_HI)

`RB` Memory operand with address space suitable for `buffer_*` instructions

`RF` Memory operand with address space suitable for `flat_*` instructions

`RS` Memory operand with address space suitable for `s_*` instructions

`RL` Memory operand with address space suitable for `ds_*` LDS instructions

`RG` Memory operand with address space suitable for `ds_*` GDS instructions

`RD` Memory operand with address space suitable for any `ds_*` instructions

`RM` Memory operand with address space suitable for `global_*` instructions

ARC —'`config/arc/constraints.md`'

`q` Registers usable in ARCompact 16-bit instructions: `r0-r3`, `r12-r15`. This constraint can only match when the '`-mq`' option is in effect.

`e` Registers usable as base-regs of memory addresses in ARCompact 16-bit memory instructions: `r0-r3`, `r12-r15`, `sp`. This constraint can only match when the '`-mq`' option is in effect.

`D` ARC FPX (dpfp) 64-bit registers. `D0`, `D1`.

`I` A signed 12-bit integer constant.

`Cal` constant for arithmetic/logical operations. This might be any constant that can be put into a long immediate by the assmbler or linker without involving a PIC relocation.

`K` A 3-bit unsigned integer constant.

`L` A 6-bit unsigned integer constant.

`CnL` One's complement of a 6-bit unsigned integer constant.

`CmL` Two's complement of a 6-bit unsigned integer constant.

`M` A 5-bit unsigned integer constant.

`O` A 7-bit unsigned integer constant.

`P` A 8-bit unsigned integer constant.

`H` Any const_double value.

ARM family—'`config/arm/constraints.md`'

`h` In Thumb state, the core registers `r8-r15`.

`k` The stack pointer register.

`l` In Thumb State the core registers `r0-r7`. In ARM state this is an alias for the `r` constraint.

`t` VFP floating-point registers `s0-s31`. Used for 32 bit values.

`w` VFP floating-point registers `d0-d31` and the appropriate subset `d0-d15` based on command line options. Used for 64 bit values only. Not valid for Thumb1.

`y` The iWMMX co-processor registers.

`z` The iWMMX GR registers.

`G` The floating-point constant 0.0

`I` Integer that is valid as an immediate operand in a data processing instruction. That is, an integer in the range 0 to 255 rotated by a multiple of 2

`J` Integer in the range −4095 to 4095

`K` Integer that satisfies constraint '`I`' when inverted (ones complement)

`L` Integer that satisfies constraint '`I`' when negated (twos complement)

`M` Integer in the range 0 to 32

`Q` A memory reference where the exact address is in a single register ("`m`" is preferable for `asm` statements)

`R` An item in the constant pool

`S` A symbol in the text segment of the current file

`Uv` A memory reference suitable for VFP load/store insns (reg+constant offset)

`Uy` A memory reference suitable for iWMMXt load/store instructions.

`Uq` A memory reference suitable for the ARMv4 ldrsb instruction.

AVR family—'`config/avr/constraints.md`'

`l` Registers from r0 to r15

`a` Registers from r16 to r23

`d` Registers from r16 to r31

`w` Registers from r24 to r31. These registers can be used in '`adiw`' command

`e` Pointer register (r26–r31)

`b` Base pointer register (r28–r31)

`q` Stack pointer register (SPH:SPL)

`t` Temporary register r0

`x` Register pair X (r27:r26)

`y` Register pair Y (r29:r28)

`z` Register pair Z (r31:r30)

`I` Constant greater than −1, less than 64

`J` Constant greater than −64, less than 1

`K`	Constant integer 2
`L`	Constant integer 0
`M`	Constant that fits in 8 bits
`N`	Constant integer −1
`O`	Constant integer 8, 16, or 24
`P`	Constant integer 1
`G`	A floating point constant 0.0
`Q`	A memory address based on Y or Z pointer with displacement.

Blackfin family—'`config/bfin/constraints.md`'

`a`	P register
`d`	D register
`z`	A call clobbered P register.
`q`*n*	A single register. If *n* is in the range 0 to 7, the corresponding D register. If it is `A`, then the register P0.
`D`	Even-numbered D register
`W`	Odd-numbered D register
`e`	Accumulator register.
`A`	Even-numbered accumulator register.
`B`	Odd-numbered accumulator register.
`b`	I register
`v`	B register
`f`	M register
`c`	Registers used for circular buffering, i.e. I, B, or L registers.
`C`	The CC register.
`t`	LT0 or LT1.
`k`	LC0 or LC1.
`u`	LB0 or LB1.
`x`	Any D, P, B, M, I or L register.
`y`	Additional registers typically used only in prologues and epilogues: RETS, RETN, RETI, RETX, RETE, ASTAT, SEQSTAT and USP.
`w`	Any register except accumulators or CC.
`Ksh`	Signed 16 bit integer (in the range −32768 to 32767)
`Kuh`	Unsigned 16 bit integer (in the range 0 to 65535)

`Ks7` Signed 7 bit integer (in the range −64 to 63)

`Ku7` Unsigned 7 bit integer (in the range 0 to 127)

`Ku5` Unsigned 5 bit integer (in the range 0 to 31)

`Ks4` Signed 4 bit integer (in the range −8 to 7)

`Ks3` Signed 3 bit integer (in the range −3 to 4)

`Ku3` Unsigned 3 bit integer (in the range 0 to 7)

`P`*n* Constant *n*, where *n* is a single-digit constant in the range 0 to 4.

`PA` An integer equal to one of the MACFLAG_XXX constants that is suitable for use with either accumulator.

`PB` An integer equal to one of the MACFLAG_XXX constants that is suitable for use only with accumulator A1.

`M1` Constant 255.

`M2` Constant 65535.

`J` An integer constant with exactly a single bit set.

`L` An integer constant with all bits set except exactly one.

`H`

`Q` Any SYMBOL_REF.

CR16 Architecture—'`config/cr16/cr16.h`'

`b` Registers from r0 to r14 (registers without stack pointer)

`t` Register from r0 to r11 (all 16-bit registers)

`p` Register from r12 to r15 (all 32-bit registers)

`I` Signed constant that fits in 4 bits

`J` Signed constant that fits in 5 bits

`K` Signed constant that fits in 6 bits

`L` Unsigned constant that fits in 4 bits

`M` Signed constant that fits in 32 bits

`N` Check for 64 bits wide constants for add/sub instructions

`G` Floating point constant that is legal for store immediate

C-SKY—'`config/csky/constraints.md`'

`a` The mini registers r0 - r7.

`b` The low registers r0 - r15.

`c` C register.

`y` HI and LO registers.

`l` LO register.

`h` HI register.

`v` Vector registers.

`z` Stack pointer register (SP).

Epiphany—'`config/epiphany/constraints.md`'

`U16` An unsigned 16-bit constant.

`K` An unsigned 5-bit constant.

`L` A signed 11-bit constant.

`Cm1` A signed 11-bit constant added to −1. Can only match when the '`-m1reg-`*`reg`*' option is active.

`Cl1` Left-shift of −1, i.e., a bit mask with a block of leading ones, the rest being a block of trailing zeroes. Can only match when the '`-m1reg-`*`reg`*' option is active.

`Cr1` Right-shift of −1, i.e., a bit mask with a trailing block of ones, the rest being zeroes. Or to put it another way, one less than a power of two. Can only match when the '`-m1reg-`*`reg`*' option is active.

`Cal` Constant for arithmetic/logical operations. This is like `i`, except that for position independent code, no symbols / expressions needing relocations are allowed.

`Csy` Symbolic constant for call/jump instruction.

`Rcs` The register class usable in short insns. This is a register class constraint, and can thus drive register allocation. This constraint won't match unless '`-mprefer-short-insn-regs`' is in effect.

`Rsc` The the register class of registers that can be used to hold a sibcall call address. I.e., a caller-saved register.

`Rct` Core control register class.

`Rgs` The register group usable in short insns. This constraint does not use a register class, so that it only passively matches suitable registers, and doesn't drive register allocation.

`Rra` Matches the return address if it can be replaced with the link register.

`Rcc` Matches the integer condition code register.

`Sra` Matches the return address if it is in a stack slot.

`Cfm` Matches control register values to switch fp mode, which are encapsulated in `UNSPEC_FP_MODE`.

FRV—'`config/frv/frv.h`'

`a` Register in the class `ACC_REGS` (`acc0` to `acc7`).

`b` Register in the class `EVEN_ACC_REGS` (`acc0` to `acc7`).

`c` Register in the class `CC_REGS` (`fcc0` to `fcc3` and `icc0` to `icc3`).

`d` Register in the class `GPR_REGS` (`gr0` to `gr63`).

`e` Register in the class `EVEN_REGS` (`gr0` to `gr63`). Odd registers are excluded not in the class but through the use of a machine mode larger than 4 bytes.

`f` Register in the class `FPR_REGS` (`fr0` to `fr63`).

`h` Register in the class `FEVEN_REGS` (`fr0` to `fr63`). Odd registers are excluded not in the class but through the use of a machine mode larger than 4 bytes.

`l` Register in the class `LR_REG` (the `lr` register).

`q` Register in the class `QUAD_REGS` (`gr2` to `gr63`). Register numbers not divisible by 4 are excluded not in the class but through the use of a machine mode larger than 8 bytes.

`t` Register in the class `ICC_REGS` (`icc0` to `icc3`).

`u` Register in the class `FCC_REGS` (`fcc0` to `fcc3`).

`v` Register in the class `ICR_REGS` (`cc4` to `cc7`).

`w` Register in the class `FCR_REGS` (`cc0` to `cc3`).

`x` Register in the class `QUAD_FPR_REGS` (`fr0` to `fr63`). Register numbers not divisible by 4 are excluded not in the class but through the use of a machine mode larger than 8 bytes.

`z` Register in the class `SPR_REGS` (`lcr` and `lr`).

`A` Register in the class `QUAD_ACC_REGS` (`acc0` to `acc7`).

`B` Register in the class `ACCG_REGS` (`accg0` to `accg7`).

`C` Register in the class `CR_REGS` (`cc0` to `cc7`).

`G` Floating point constant zero

`I` 6-bit signed integer constant

`J` 10-bit signed integer constant

`L` 16-bit signed integer constant

`M` 16-bit unsigned integer constant

`N` 12-bit signed integer constant that is negative—i.e. in the range of -2048 to -1

`O` Constant zero

`P` 12-bit signed integer constant that is greater than zero—i.e. in the range of 1 to 2047.

FT32—'`config/ft32/constraints.md`'

`A` An absolute address

`B`	An offset address
`W`	A register indirect memory operand
`e`	An offset address.
`f`	An offset address.
`O`	The constant zero or one
`I`	A 16-bit signed constant (−32768 . . . 32767)
`w`	A bitfield mask suitable for bext or bins
`x`	An inverted bitfield mask suitable for bext or bins
`L`	A 16-bit unsigned constant, multiple of 4 (0 . . . 65532)
`S`	A 20-bit signed constant (−524288 . . . 524287)
`b`	A constant for a bitfield width (1 . . . 16)
`KA`	A 10-bit signed constant (−512 . . . 511)

Hewlett-Packard PA-RISC—'`config/pa/pa.h`'

`a`	General register 1
`f`	Floating point register
`q`	Shift amount register
`x`	Floating point register (deprecated)
`y`	Upper floating point register (32-bit), floating point register (64-bit)
`Z`	Any register
`I`	Signed 11-bit integer constant
`J`	Signed 14-bit integer constant
`K`	Integer constant that can be deposited with a `zdepi` instruction
`L`	Signed 5-bit integer constant
`M`	Integer constant 0
`N`	Integer constant that can be loaded with a `ldil` instruction
`O`	Integer constant whose value plus one is a power of 2
`P`	Integer constant that can be used for `and` operations in `depi` and `extru` instructions
`S`	Integer constant 31
`U`	Integer constant 63
`G`	Floating-point constant 0.0
`A`	A `lo_sum` data-linkage-table memory operand

`Q` A memory operand that can be used as the destination operand of an integer store instruction

`R` A scaled or unscaled indexed memory operand

`T` A memory operand for floating-point loads and stores

`W` A register indirect memory operand

Intel IA-64—'`config/ia64/ia64.h`'

`a` General register `r0` to `r3` for `addl` instruction

`b` Branch register

`c` Predicate register ('`c`' as in "conditional")

`d` Application register residing in M-unit

`e` Application register residing in I-unit

`f` Floating-point register

`m` Memory operand. If used together with '`<`' or '`>`', the operand can have postincrement and postdecrement which require printing with '`%Pn`' on IA-64.

`G` Floating-point constant 0.0 or 1.0

`I` 14-bit signed integer constant

`J` 22-bit signed integer constant

`K` 8-bit signed integer constant for logical instructions

`L` 8-bit adjusted signed integer constant for compare pseudo-ops

`M` 6-bit unsigned integer constant for shift counts

`N` 9-bit signed integer constant for load and store postincrements

`O` The constant zero

`P` 0 or −1 for `dep` instruction

`Q` Non-volatile memory for floating-point loads and stores

`R` Integer constant in the range 1 to 4 for `shladd` instruction

`S` Memory operand except postincrement and postdecrement. This is now roughly the same as '`m`' when not used together with '`<`' or '`>`'.

M32C—'`config/m32c/m32c.c`'

`Rsp`
`Rfb`
`Rsb` '`$sp`', '`$fb`', '`$sb`'.

`Rcr` Any control register, when they're 16 bits wide (nothing if control registers are 24 bits wide)

`Rcl`	Any control register, when they're 24 bits wide.

`R0w`
`R1w`
`R2w`
`R3w`
	$r0, $r1, $r2, $r3.

`R02`	$r0 or $r2, or $r2r0 for 32 bit values.
`R13`	$r1 or $r3, or $r3r1 for 32 bit values.
`Rdi`	A register that can hold a 64 bit value.
`Rhl`	$r0 or $r1 (registers with addressable high/low bytes)
`R23`	$r2 or $r3
`Raa`	Address registers
`Raw`	Address registers when they're 16 bits wide.
`Ral`	Address registers when they're 24 bits wide.
`Rqi`	Registers that can hold QI values.
`Rad`	Registers that can be used with displacements ($a0, $a1, $sb).
`Rsi`	Registers that can hold 32 bit values.
`Rhi`	Registers that can hold 16 bit values.
`Rhc`	Registers chat can hold 16 bit values, including all control registers.
`Rra`	$r0 through R1, plus $a0 and $a1.
`Rfl`	The flags register.
`Rmm`	The memory-based pseudo-registers $mem0 through $mem15.
`Rpi`	Registers that can hold pointers (16 bit registers for r8c, m16c; 24 bit registers for m32cm, m32c).
`Rpa`	Matches multiple registers in a PARALLEL to form a larger register. Used to match function return values.
`Is3`	$-8 \ldots 7$
`IS1`	$-128 \ldots 127$
`IS2`	$-32768 \ldots 32767$
`IU2`	$0 \ldots 65535$
`In4`	$-8 \ldots -1$ or $1 \ldots 8$
`In5`	$-16 \ldots -1$ or $1 \ldots 16$
`In6`	$-32 \ldots -1$ or $1 \ldots 32$
`IM2`	$-65536 \ldots -1$
`Ilb`	An 8 bit value with exactly one bit set.

`Ilw` A 16 bit value with exactly one bit set.

`Sd` The common src/dest memory addressing modes.

`Sa` Memory addressed using $a0 or $a1.

`Si` Memory addressed with immediate addresses.

`Ss` Memory addressed using the stack pointer ($sp).

`Sf` Memory addressed using the frame base register ($fb).

`Ss` Memory addressed using the small base register ($sb).

`S1` $r1h

MicroBlaze—'`config/microblaze/constraints.md`'

`d` A general register (`r0` to `r31`).

`z` A status register (`rmsr`, `$fcc1` to `$fcc7`).

MIPS—'`config/mips/constraints.md`'

`d` A general-purpose register. This is equivalent to `r` unless generating MIPS16 code, in which case the MIPS16 register set is used.

`f` A floating-point register (if available).

`h` Formerly the `hi` register. This constraint is no longer supported.

`l` The `lo` register. Use this register to store values that are no bigger than a word.

`x` The concatenated `hi` and `lo` registers. Use this register to store doubleword values.

`c` A register suitable for use in an indirect jump. This will always be `$25` for '`-mabicalls`'.

`v` Register `$3`. Do not use this constraint in new code; it is retained only for compatibility with glibc.

`y` Equivalent to `r`; retained for backwards compatibility.

`z` A floating-point condition code register.

`I` A signed 16-bit constant (for arithmetic instructions).

`J` Integer zero.

`K` An unsigned 16-bit constant (for logic instructions).

`L` A signed 32-bit constant in which the lower 16 bits are zero. Such constants can be loaded using `lui`.

`M` A constant that cannot be loaded using `lui`, `addiu` or `ori`.

`N` A constant in the range −65535 to −1 (inclusive).

`O` A signed 15-bit constant.

`P` A constant in the range 1 to 65535 (inclusive).

`G` Floating-point zero.

`R` An address that can be used in a non-macro load or store.

`ZC` A memory operand whose address is formed by a base register and offset that is suitable for use in instructions with the same addressing mode as `ll` and `sc`.

`ZD` An address suitable for a `prefetch` instruction, or for any other instruction with the same addressing mode as `prefetch`.

Motorola 680x0—'`config/m68k/constraints.md`'

`a` Address register

`d` Data register

`f` 68881 floating-point register, if available

`I` Integer in the range 1 to 8

`J` 16-bit signed number

`K` Signed number whose magnitude is greater than 0x80

`L` Integer in the range −8 to −1

`M` Signed number whose magnitude is greater than 0x100

`N` Range 24 to 31, rotatert:SI 8 to 1 expressed as rotate

`O` 16 (for rotate using swap)

`P` Range 8 to 15, rotatert:HI 8 to 1 expressed as rotate

`R` Numbers that mov3q can handle

`G` Floating point constant that is not a 68881 constant

`S` Operands that satisfy 'm' when -mpcrel is in effect

`T` Operands that satisfy 's' when -mpcrel is not in effect

`Q` Address register indirect addressing mode

`U` Register offset addressing

`W` const_call_operand

`Cs` symbol_ref or const

`Ci` const_int

`C0` const_int 0

`Cj` Range of signed numbers that don't fit in 16 bits

`Cmvq` Integers valid for mvq

`Capsw` Integers valid for a moveq followed by a swap

`Cmvz` Integers valid for mvz

`Cmvs` Integers valid for mvs

`Ap` push_operand

`Ac` Non-register operands allowed in clr

Moxie—'`config/moxie/constraints.md`'

`A` An absolute address

`B` An offset address

`W` A register indirect memory operand

`I` A constant in the range of 0 to 255.

`N` A constant in the range of 0 to −255.

MSP430–'`config/msp430/constraints.md`'

`R12` Register R12.

`R13` Register R13.

`K` Integer constant 1.

`L` Integer constant -1^20..1^19.

`M` Integer constant 1-4.

`Ya` Memory references which do not require an extended MOVX instruction.

`Yl` Memory reference, labels only.

`Ys` Memory reference, stack only.

NDS32—'`config/nds32/constraints.md`'

`w` LOW register class $r0 to $r7 constraint for V3/V3M ISA.

`l` LOW register class $r0 to $r7.

`d` MIDDLE register class $r0 to $r11, $r16 to $r19.

`h` HIGH register class $r12 to $r14, $r20 to $r31.

`t` Temporary assist register $ta (i.e. $r15).

`k` Stack register $sp.

`Iu03` Unsigned immediate 3-bit value.

`In03` Negative immediate 3-bit value in the range of −7–0.

`Iu04` Unsigned immediate 4-bit value.

`Is05` Signed immediate 5-bit value.

`Iu05` Unsigned immediate 5-bit value.

`In05` Negative immediate 5-bit value in the range of −31–0.

`Ip05` Unsigned immediate 5-bit value for movpi45 instruction with range 16–47.

`Iu06` Unsigned immediate 6-bit value constraint for addri36.sp instruction.

`Iu08` Unsigned immediate 8-bit value.

`Iu09` Unsigned immediate 9-bit value.

`Is10` Signed immediate 10-bit value.

`Is11` Signed immediate 11-bit value.

`Is15` Signed immediate 15-bit value.

`Iu15` Unsigned immediate 15-bit value.

`Ic15` A constant which is not in the range of imm15u but ok for bclr instruction.

`Ie15` A constant which is not in the range of imm15u but ok for bset instruction.

`It15` A constant which is not in the range of imm15u but ok for btgl instruction.

`Ii15` A constant whose compliment value is in the range of imm15u and ok for bitci instruction.

`Is16` Signed immediate 16-bit value.

`Is17` Signed immediate 17-bit value.

`Is19` Signed immediate 19-bit value.

`Is20` Signed immediate 20-bit value.

`Ihig` The immediate value that can be simply set high 20-bit.

`Izeb` The immediate value 0xff.

`Izeh` The immediate value 0xffff.

`Ixls` The immediate value 0x01.

`Ix11` The immediate value 0x7ff.

`Ibms` The immediate value with power of 2.

`Ifex` The immediate value with power of 2 minus 1.

`U33` Memory constraint for 333 format.

`U45` Memory constraint for 45 format.

`U37` Memory constraint for 37 format.

Nios II family—'`config/nios2/constraints.md`'

`I` Integer that is valid as an immediate operand in an instruction taking a signed 16-bit number. Range −32768 to 32767.

`J` Integer that is valid as an immediate operand in an instruction taking an unsigned 16-bit number. Range 0 to 65535.

`K` Integer that is valid as an immediate operand in an instruction taking only the upper 16-bits of a 32-bit number. Range 32-bit numbers with the lower 16-bits being 0.

`L` Integer that is valid as an immediate operand for a shift instruction. Range 0 to 31.

`M` Integer that is valid as an immediate operand for only the value 0. Can be used in conjunction with the format modifier `z` to use `r0` instead of `0` in the assembly output.

`N` Integer that is valid as an immediate operand for a custom instruction opcode. Range 0 to 255.

`P` An immediate operand for R2 andchi/andci instructions.

`S` Matches immediates which are addresses in the small data section and therefore can be added to `gp` as a 16-bit immediate to re-create their 32-bit value.

`U` Matches constants suitable as an operand for the rdprs and cache instructions.

`v` A memory operand suitable for Nios II R2 load/store exclusive instructions.

`w` A memory operand suitable for load/store IO and cache instructions.

OpenRISC—'`config/or1k/constraints.md`'

`I` Integer that is valid as an immediate operand in an instruction taking a signed 16-bit number. Range −32768 to 32767.

`K` Integer that is valid as an immediate operand in an instruction taking an unsigned 16-bit number. Range 0 to 65535.

`M` Signed 16-bit constant shifted left 16 bits. (Used with `l.movhi`)

`O` Zero

PDP-11—'`config/pdp11/constraints.md`'

`a` Floating point registers AC0 through AC3. These can be loaded from/to memory with a single instruction.

`d` Odd numbered general registers (R1, R3, R5). These are used for 16-bit multiply operations.

`D` A memory reference that is encoded within the opcode, but not auto-increment or auto-decrement.

`f` Any of the floating point registers (AC0 through AC5).

`G` Floating point constant 0.

`h` Floating point registers AC4 and AC5. These cannot be loaded from/to memory with a single instruction.

`I` An integer constant that fits in 16 bits.

`J` An integer constant whose low order 16 bits are zero.

`K` An integer constant that does not meet the constraints for codes '`I`' or '`J`'.

`L` The integer constant 1.

`M` The integer constant −1.

`N` The integer constant 0.

`O` Integer constants 0 through 3; shifts by these amounts are handled as multiple single-bit shifts rather than a single variable-length shift.

`Q` A memory reference which requires an additional word (address or offset) after the opcode.

`R` A memory reference that is encoded within the opcode.

PowerPC and IBM RS6000—'`config/rs6000/constraints.md`'

`b` Address base register

`d` Floating point register (containing 64-bit value)

`f` Floating point register (containing 32-bit value)

`v` Altivec vector register

`wa` Any VSX register if the '`-mvsx`' option was used or NO_REGS.

When using any of the register constraints (`wa`, `wd`, `wf`, `wg`, `wh`, `wi`, `wj`, `wk`, `wl`, `wm`, `wo`, `wp`, `wq`, `ws`, `wt`, `wu`, `wv`, `ww`, or `wy`) that take VSX registers, you must use `%x<n>` in the template so that the correct register is used. Otherwise the register number output in the assembly file will be incorrect if an Altivec register is an operand of a VSX instruction that expects VSX register numbering.

```
asm ("xvadddp %x0,%x1,%x2"
     : "=wa" (v1)
     : "wa" (v2), "wa" (v3));
```

is correct, but:

```
asm ("xvadddp %0,%1,%2"
     : "=wa" (v1)
     : "wa" (v2), "wa" (v3));
```

is not correct.

If an instruction only takes Altivec registers, you do not want to use `%x<n>`.

```
asm ("xsaddqp %0,%1,%2"
     : "=v" (v1)
     : "v" (v2), "v" (v3));
```

is correct because the `xsaddqp` instruction only takes Altivec registers, while:

```
asm ("xsaddqp %x0,%x1,%x2"
     : "=v" (v1)
     : "v" (v2), "v" (v3));
```

is incorrect.

`wb` Altivec register if '`-mcpu=power9`' is used or NO_REGS.

`wd` VSX vector register to hold vector double data or NO_REGS.

`we` VSX register if the '`-mcpu=power9`' and '`-m64`' options were used or NO_REGS.

`wf` VSX vector register to hold vector float data or NO_REGS.

`wg` If '`-mmfpgpr`' was used, a floating point register or NO_REGS.

`wh` Floating point register if direct moves are available, or NO_REGS.

`wi` FP or VSX register to hold 64-bit integers for VSX insns or NO_REGS.

`wj` FP or VSX register to hold 64-bit integers for direct moves or NO_REGS.

`wk` FP or VSX register to hold 64-bit doubles for direct moves or NO_REGS.

`wl` Floating point register if the LFIWAX instruction is enabled or NO_REGS.

`wm` VSX register if direct move instructions are enabled, or NO_REGS.

`wn` No register (NO_REGS).

`wo` VSX register to use for ISA 3.0 vector instructions, or NO_REGS.

`wp` VSX register to use for IEEE 128-bit floating point TFmode, or NO_REGS.

`wq` VSX register to use for IEEE 128-bit floating point, or NO_REGS.

`wr` General purpose register if 64-bit instructions are enabled or NO_REGS.

`ws` VSX vector register to hold scalar double values or NO_REGS.

`wt` VSX vector register to hold 128 bit integer or NO_REGS.

`wu` Altivec register to use for float/32-bit int loads/stores or NO_REGS.

`wv` Altivec register to use for double loads/stores or NO_REGS.

`ww` FP or VSX register to perform float operations under '`-mvsx`' or NO_REGS.

`wx` Floating point register if the STFIWX instruction is enabled or NO_REGS.

`wy` FP or VSX register to perform ISA 2.07 float ops or NO_REGS.

`wz` Floating point register if the LFIWZX instruction is enabled or NO_REGS.

`wA` Address base register if 64-bit instructions are enabled or NO_REGS.

`wB` Signed 5-bit constant integer that can be loaded into an altivec register.

`wD` Int constant that is the element number of the 64-bit scalar in a vector.

`wE` Vector constant that can be loaded with the XXSPLTIB instruction.

`wF` Memory operand suitable for power8 GPR load fusion

`wG` Memory operand suitable for TOC fusion memory references.

`wH` Altivec register if '`-mvsx-small-integer`'.

`wI` Floating point register if '`-mvsx-small-integer`'.

`wJ` FP register if '`-mvsx-small-integer`' and '`-mpower9-vector`'.

`wK` Altivec register if '`-mvsx-small-integer`' and '`-mpower9-vector`'.

`wL` Int constant that is the element number that the MFVSRLD instruction. targets.

`wM` Match vector constant with all 1's if the XXLORC instruction is available.

`wO` A memory operand suitable for the ISA 3.0 vector d-form instructions.

`wQ` A memory address that will work with the `lq` and `stq` instructions.

`wS` Vector constant that can be loaded with XXSPLTIB & sign extension.

`h` '`VRSAVE`', '`CTR`', or '`LINK`' register

`c` '`CTR`' register

`l` '`LINK`' register

`x` '`CR`' register (condition register) number 0

`y` '`CR`' register (condition register)

`z` '`XER[CA]`' carry bit (part of the XER register)

`I` Signed 16-bit constant

`J` Unsigned 16-bit constant shifted left 16 bits (use '`L`' instead for `SImode` constants)

`K` Unsigned 16-bit constant

`L` Signed 16-bit constant shifted left 16 bits

`M` Constant larger than 31

`N` Exact power of 2

`O` Zero

`P` Constant whose negation is a signed 16-bit constant

`G` Floating point constant that can be loaded into a register with one instruction per word

`H` Integer/Floating point constant that can be loaded into a register using three instructions

`m` Memory operand. Normally, `m` does not allow addresses that update the base register. If '`<`' or '`>`' constraint is also used, they are allowed and therefore on PowerPC targets in that case it is only safe to use '`m<>`' in an `asm` statement if that `asm` statement accesses the operand exactly once. The `asm` statement must also use '`%U<`*opno*`>`' as a placeholder for the "update" flag in the corresponding load or store instruction. For example:

```
asm ("st%U0 %1,%0" : "=m<>" (mem) : "r" (val));
```

is correct but:

```
asm ("st %1,%0" : "=m<>" (mem) : "r" (val));
```

is not.

`es` A "stable" memory operand; that is, one which does not include any automodification of the base register. This used to be useful when '`m`' allowed automodification of the base register, but as those are now only allowed when '`<`' or '`>`' is used, '`es`' is basically the same as '`m`' without '`<`' and '`>`'.

`Q` Memory operand that is an offset from a register (it is usually better to use '`m`' or '`es`' in `asm` statements)

`Z` Memory operand that is an indexed or indirect from a register (it is usually better to use '`m`' or '`es`' in `asm` statements)

`R` AIX TOC entry

`a` Address operand that is an indexed or indirect from a register ('`p`' is preferable for `asm` statements)

`U` System V Release 4 small data area reference

`W` Vector constant that does not require memory

`j` Vector constant that is all zeros.

RL78—'`config/rl78/constraints.md`'

`Int3` An integer constant in the range 1 . . . 7.

`Int8` An integer constant in the range 0 . . . 255.

`J` An integer constant in the range −255 . . . 0

`K`	The integer constant 1.
`L`	The integer constant -1.
`M`	The integer constant 0.
`N`	The integer constant 2.
`O`	The integer constant -2.
`P`	An integer constant in the range 1 . . . 15.
`Qbi`	The built-in compare types–eq, ne, gtu, ltu, geu, and leu.
`Qsc`	The synthetic compare types–gt, lt, ge, and le.
`Wab`	A memory reference with an absolute address.
`Wbc`	A memory reference using `BC` as a base register, with an optional offset.
`Wca`	A memory reference using `AX`, `BC`, `DE`, or `HL` for the address, for calls.
`Wcv`	A memory reference using any 16-bit register pair for the address, for calls.
`Wd2`	A memory reference using `DE` as a base register, with an optional offset.
`Wde`	A memory reference using `DE` as a base register, without any offset.
`Wfr`	Any memory reference to an address in the far address space.
`Wh1`	A memory reference using `HL` as a base register, with an optional one-byte offset.
`Whb`	A memory reference using `HL` as a base register, with `B` or `C` as the index register.
`Whl`	A memory reference using `HL` as a base register, without any offset.
`Ws1`	A memory reference using `SP` as a base register, with an optional one-byte offset.
`Y`	Any memory reference to an address in the near address space.
`A`	The `AX` register.
`B`	The `BC` register.
`D`	The `DE` register.
`R`	`A` through `L` registers.
`S`	The `SP` register.
`T`	The `HL` register.
`Z08W`	The 16-bit `R8` register.
`Z10W`	The 16-bit `R10` register.

`Zint` The registers reserved for interrupts (`R24` to `R31`).

`a` The `A` register.

`b` The `B` register.

`c` The `C` register.

`d` The `D` register.

`e` The `E` register.

`h` The `H` register.

`l` The `L` register.

`v` The virtual registers.

`w` The `PSW` register.

`x` The `X` register.

RISC-V—'`config/riscv/constraints.md`'

`f` A floating-point register (if availiable).

`I` An I-type 12-bit signed immediate.

`J` Integer zero.

`K` A 5-bit unsigned immediate for CSR access instructions.

`A` An address that is held in a general-purpose register.

RX—'`config/rx/constraints.md`'

`Q` An address which does not involve register indirect addressing or pre/post increment/decrement addressing.

`Symbol` A symbol reference.

`Int08` A constant in the range −256 to 255, inclusive.

`Sint08` A constant in the range −128 to 127, inclusive.

`Sint16` A constant in the range −32768 to 32767, inclusive.

`Sint24` A constant in the range −8388608 to 8388607, inclusive.

`Uint04` A constant in the range 0 to 15, inclusive.

S/390 and zSeries—'`config/s390/s390.h`'

`a` Address register (general purpose register except r0)

`c` Condition code register

`d` Data register (arbitrary general purpose register)

`f` Floating-point register

`I` Unsigned 8-bit constant (0–255)

`J` Unsigned 12-bit constant (0–4095)

`K` Signed 16-bit constant (−32768–32767)

`L` Value appropriate as displacement.

`(0..4095)`
for short displacement

`(−524288..524287)`
for long displacement

`M` Constant integer with a value of 0x7fffffff.

`N` Multiple letter constraint followed by 4 parameter letters.

`0..9:` number of the part counting from most to least significant

`H,Q:` mode of the part

`D,S,H:` mode of the containing operand

`0,F:` value of the other parts (F—all bits set)

The constraint matches if the specified part of a constant has a value different from its other parts.

`Q` Memory reference without index register and with short displacement.

`R` Memory reference with index register and short displacement.

`S` Memory reference without index register but with long displacement.

`T` Memory reference with index register and long displacement.

`U` Pointer with short displacement.

`W` Pointer with long displacement.

`Y` Shift count operand.

SPARC—'`config/sparc/sparc.h`'

`f` Floating-point register on the SPARC-V8 architecture and lower floating-point register on the SPARC-V9 architecture.

`e` Floating-point register. It is equivalent to '`f`' on the SPARC-V8 architecture and contains both lower and upper floating-point registers on the SPARC-V9 architecture.

`c` Floating-point condition code register.

`d` Lower floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.

`b` Floating-point register. It is only valid on the SPARC-V9 architecture when the Visual Instruction Set is available.

`h` 64-bit global or out register for the SPARC-V8+ architecture.

`C` The constant all-ones, for floating-point.

`A` Signed 5-bit constant

`D` A vector constant

`I` Signed 13-bit constant

`J` Zero

`K` 32-bit constant with the low 12 bits clear (a constant that can be loaded with the `sethi` instruction)

`L` A constant in the range supported by `movcc` instructions (11-bit signed immediate)

`M` A constant in the range supported by `movrcc` instructions (10-bit signed immediate)

`N` Same as '`K`', except that it verifies that bits that are not in the lower 32-bit range are all zero. Must be used instead of '`K`' for modes wider than `SImode`

`O` The constant 4096

`G` Floating-point zero

`H` Signed 13-bit constant, sign-extended to 32 or 64 bits

`P` The constant -1

`Q` Floating-point constant whose integral representation can be moved into an integer register using a single sethi instruction

`R` Floating-point constant whose integral representation can be moved into an integer register using a single mov instruction

`S` Floating-point constant whose integral representation can be moved into an integer register using a high/lo_sum instruction sequence

`T` Memory address aligned to an 8-byte boundary

`U` Even register

`W` Memory address for '`e`' constraint registers

`w` Memory address with only a base register

`Y` Vector zero

SPU—'`config/spu/spu.h`'

`a` An immediate which can be loaded with the il/ila/ilh/ilhu instructions. const_int is treated as a 64 bit value.

`c` An immediate for and/xor/or instructions. const_int is treated as a 64 bit value.

`d` An immediate for the `iohl` instruction. const_int is treated as a 64 bit value.

`f` An immediate which can be loaded with `fsmbi`.

`A` An immediate which can be loaded with the il/ila/ilh/ilhu instructions. const_int is treated as a 32 bit value.

`B` An immediate for most arithmetic instructions. const_int is treated as a 32 bit value.

`C` An immediate for and/xor/or instructions. const_int is treated as a 32 bit value.

`D` An immediate for the `iohl` instruction. const_int is treated as a 32 bit value.

`I` A constant in the range [−64, 63] for shift/rotate instructions.

`J` An unsigned 7-bit constant for conversion/nop/channel instructions.

`K` A signed 10-bit constant for most arithmetic instructions.

`M` A signed 16 bit immediate for `stop`.

`N` An unsigned 16-bit constant for `iohl` and `fsmbi`.

`O` An unsigned 7-bit constant whose 3 least significant bits are 0.

`P` An unsigned 3-bit constant for 16-byte rotates and shifts

`R` Call operand, reg, for indirect calls

`S` Call operand, symbol, for relative calls.

`T` Call operand, const_int, for absolute calls.

`U` An immediate which can be loaded with the il/ila/ilh/ilhu instructions. const_int is sign extended to 128 bit.

`W` An immediate for shift and rotate instructions. const_int is treated as a 32 bit value.

`Y` An immediate for and/xor/or instructions. const_int is sign extended as a 128 bit.

`Z` An immediate for the `iohl` instruction. const_int is sign extended to 128 bit.

TI C6X family—'`config/c6x/constraints.md`'

`a` Register file A (A0–A31).

`b` Register file B (B0–B31).

`A` Predicate registers in register file A (A0–A2 on C64X and higher, A1 and A2 otherwise).

`B` Predicate registers in register file B (B0–B2).

`C` A call-used register in register file B (B0–B9, B16–B31).

`Da` Register file A, excluding predicate registers (A3–A31, plus A0 if not C64X or higher).

`Db` Register file B, excluding predicate registers (B3–B31).

`Iu4` Integer constant in the range 0 . . . 15.

`Iu5` Integer constant in the range 0 . . . 31.

`In5` Integer constant in the range −31 . . . 0.

`Is5` Integer constant in the range −16 . . . 15.

`I5x` Integer constant that can be the operand of an ADDA or a SUBA insn.

`IuB` Integer constant in the range 0 . . . 65535.

`IsB` Integer constant in the range −32768 . . . 32767.

`IsC` Integer constant in the range -2^{20} . . . $2^{20} - 1$.

`Jc` Integer constant that is a valid mask for the clr instruction.

`Js` Integer constant that is a valid mask for the set instruction.

`Q` Memory location with A base register.

`R` Memory location with B base register.

`Z` Register B14 (aka DP).

TILE-Gx—'`config/tilegx/constraints.md`'

`R00`
`R01`
`R02`
`R03`
`R04`
`R05`
`R06`
`R07`
`R08`
`R09`
`R10` Each of these represents a register constraint for an individual register, from r0 to r10.

`I` Signed 8-bit integer constant.

`J` Signed 16-bit integer constant.

`K` Unsigned 16-bit integer constant.

`L` Integer constant that fits in one signed byte when incremented by one (−129 . . . 126).

`m` Memory operand. If used together with '`<`' or '`>`', the operand can have postincrement which requires printing with '`%In`' and '`%in`' on TILE-Gx. For example:

```
asm ("st_add %I0,%1,%i0" : "=m<>" (*mem) : "r" (val));
```

`M` A bit mask suitable for the BFINS instruction.

`N` Integer constant that is a byte tiled out eight times.

`O` The integer zero constant.

`P` Integer constant that is a sign-extended byte tiled out as four shorts.

`Q` Integer constant that fits in one signed byte when incremented (−129 . . . 126), but excluding -1.

`S` Integer constant that has all 1 bits consecutive and starting at bit 0.

`T` A 16-bit fragment of a got, tls, or pc-relative reference.

`U` Memory operand except postincrement. This is roughly the same as '`m`' when not used together with '`<`' or '`>`'.

`W` An 8-element vector constant with identical elements.

`Y` A 4-element vector constant with identical elements.

`Z0` The integer constant 0xffffffff.

`Z1` The integer constant 0xffffffff00000000.

TILEPro—'`config/tilepro/constraints.md`'

`R00`
`R01`
`R02`
`R03`
`R04`
`R05`
`R06`
`R07`
`R08`
`R09`
`R10` Each of these represents a register constraint for an individual register, from r0 to r10.

`I` Signed 8-bit integer constant.

`J` Signed 16-bit integer constant.

`K` Nonzero integer constant with low 16 bits zero.

`L` Integer constant that fits in one signed byte when incremented by one (−129 . . . 126).

`m` Memory operand. If used together with '`<`' or '`>`', the operand can have postincrement which requires printing with '`%In`' and '`%in`' on TILEPro. For example:

```
asm ("swadd %I0,%1,%i0" : "=m<>" (mem) : "r" (val));
```

`M` A bit mask suitable for the MM instruction.

`N` Integer constant that is a byte tiled out four times.

`O` The integer zero constant.

`P` Integer constant that is a sign-extended byte tiled out as two shorts.

`Q` Integer constant that fits in one signed byte when incremented ($-129 \ldots 126$), but excluding -1.

`T` A symbolic operand, or a 16-bit fragment of a got, tls, or pc-relative reference.

`U` Memory operand except postincrement. This is roughly the same as '`m`' when not used together with '`<`' or '`>`'.

`W` A 4-element vector constant with identical elements.

`Y` A 2-element vector constant with identical elements.

Visium—'`config/visium/constraints.md`'

`b` EAM register `mdb`

`c` EAM register `mdc`

`f` Floating point register

`l` General register, but not `r29`, `r30` and `r31`

`t` Register `r1`

`u` Register `r2`

`v` Register `r3`

`G` Floating-point constant 0.0

`J` Integer constant in the range 0 .. 65535 (16-bit immediate)

`K` Integer constant in the range 1 .. 31 (5-bit immediate)

`L` Integer constant in the range -65535 .. -1 (16-bit negative immediate)

`M` Integer constant -1

`O` Integer constant 0

`P` Integer constant 32

x86 family—'`config/i386/constraints.md`'

`R` Legacy register—the eight integer registers available on all i386 processors (`a`, `b`, `c`, `d`, `si`, `di`, `bp`, `sp`).

`q` Any register accessible as `rl`. In 32-bit mode, `a`, `b`, `c`, and `d`; in 64-bit mode, any integer register.

`Q` Any register accessible as `rh`: `a`, `b`, `c`, and `d`.

`a` The `a` register.

`b` The `b` register.

`c` The `c` register.

`d` The `d` register.

`S` The `si` register.

`D` The `di` register.

`A` The `a` and `d` registers. This class is used for instructions that return double word results in the `ax:dx` register pair. Single word values will be allocated either in `ax` or `dx`. For example on i386 the following implements `rdtsc`:

```
unsigned long long rdtsc (void)
{
  unsigned long long tick;
  __asm__ __volatile__("rdtsc":"=A"(tick));
  return tick;
}
```

This is not correct on x86-64 as it would allocate tick in either `ax` or `dx`. You have to use the following variant instead:

```
unsigned long long rdtsc (void)
{
  unsigned int tickl, tickh;
  __asm__ __volatile__("rdtsc":"=a"(tickl),"=d"(tickh));
  return ((unsigned long long)tickh << 32)|tickl;
}
```

`U` The call-clobbered integer registers.

`f` Any 80387 floating-point (stack) register.

`t` Top of 80387 floating-point stack (`%st(0)`).

`u` Second from top of 80387 floating-point stack (`%st(1)`).

`y` Any MMX register.

`x` Any SSE register.

`v` Any EVEX encodable SSE register (`%xmm0-%xmm31`).

`Yz` First SSE register (`%xmm0`).

`I` Integer constant in the range 0 . . . 31, for 32-bit shifts.

`J` Integer constant in the range 0 . . . 63, for 64-bit shifts.

`K` Signed 8-bit integer constant.

`L` `0xFF` or `0xFFFF`, for andsi as a zero-extending move.

`M` 0, 1, 2, or 3 (shifts for the `lea` instruction).

`N` Unsigned 8-bit integer constant (for `in` and `out` instructions).

`G` Standard 80387 floating point constant.

`C` SSE constant zero operand.

`e` 32-bit signed integer constant, or a symbolic reference known to fit that range (for immediate operands in sign-extending x86-64 instructions).

`We` 32-bit signed integer constant, or a symbolic reference known to fit that range (for sign-extending conversion operations that require non-`VOIDmode` immediate operands).

`Wz` 32-bit unsigned integer constant, or a symbolic reference known to fit that range (for zero-extending conversion operations that require non-`VOIDmode` immediate operands).

`Wd` 128-bit integer constant where both the high and low 64-bit word satisfy the `e` constraint.

`Z` 32-bit unsigned integer constant, or a symbolic reference known to fit that range (for immediate operands in zero-extending x86-64 instructions).

`Tv` VSIB address operand.

`Ts` Address operand without segment register.

Xstormy16—'`config/stormy16/stormy16.h`'

`a` Register r0.

`b` Register r1.

`c` Register r2.

`d` Register r8.

`e` Registers r0 through r7.

`t` Registers r0 and r1.

`y` The carry register.

`z` Registers r8 and r9.

`I` A constant between 0 and 3 inclusive.

`J` A constant that has exactly one bit set.

`K` A constant that has exactly one bit clear.

`L` A constant between 0 and 255 inclusive.

`M` A constant between −255 and 0 inclusive.

`N` A constant between −3 and 0 inclusive.

`O` A constant between 1 and 4 inclusive.

`P` A constant between −4 and −1 inclusive.

`Q` A memory reference that is a stack push.

`R` A memory reference that is a stack pop.

`S` A memory reference that refers to a constant address of known value.

`T` The register indicated by Rx (not implemented yet).

U	A constant that is not between 2 and 15 inclusive.
Z	The constant 0.

Xtensa—'`config/xtensa/constraints.md`'

a	General-purpose 32-bit register
b	One-bit boolean register
A	MAC16 40-bit accumulator register
I	Signed 12-bit integer constant, for use in MOVI instructions
J	Signed 8-bit integer constant, for use in ADDI instructions
K	Integer constant valid for BccI instructions
L	Unsigned constant valid for BccUI instructions

6.47.4 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator. It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols, or reference registers.

Assembler names for data:

This sample shows how to specify the assembler name for data:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C variable, this feature allows you to define names for the linker that do not start with an underscore.

GCC does not support using this feature with a non-static local variable since such variables do not have assembler names. If you are trying to put the variable in a particular register, see Section 6.47.5 [Explicit Register Variables], page 621.

Assembler names for functions:

To specify the assembler name for functions, write a declaration for the function before its definition and put `asm` there, like this:

```
int func (int x, int y) asm ("MYFUNC");

int func (int x, int y)
{
   /* ... */
```

This specifies that the name to be used for the function `func` in the assembler code should be `MYFUNC`.

6.47.5 Variables in Specified Registers

GNU C allows you to associate specific hardware registers with C variables. In almost all cases, allowing the compiler to assign registers produces the best code. However under certain unusual circumstances, more precise control over the variable storage is required.

Both global and local variables can be associated with a register. The consequences of performing this association are very different between the two, as explained in the sections below.

6.47.5.1 Defining Global Register Variables

You can define a global register variable and associate it with a specified register like this:

```
register int *foo asm ("r12");
```

Here `r12` is the name of the register that should be used. Note that this is the same syntax used for defining local register variables, but for a global variable the declaration appears outside a function. The `register` keyword is required, and cannot be combined with `static`. The register name must be a valid register name for the target platform.

Do not use type qualifiers such as `const` and `volatile`, as the outcome may be contrary to expectations. In particular, using the `volatile` qualifier does not fully prevent the compiler from optimizing accesses to the register.

Registers are a scarce resource on most systems and allowing the compiler to manage their usage usually results in the best code. However, under special circumstances it can make sense to reserve some globally. For example this may be useful in programs such as programming language interpreters that have a couple of global variables that are accessed very often.

After defining a global register variable, for the current compilation unit:

- If the register is a call-saved register, call ABI is affected: the register will not be restored in function epilogue sequences after the variable has been assigned. Therefore, functions cannot safely return to callers that assume standard ABI.
- Conversely, if the register is a call-clobbered register, making calls to functions that use standard ABI may lose contents of the variable. Such calls may be created by the compiler even if none are evident in the original program, for example when libgcc functions are used to make up for unavailable instructions.
- Accesses to the variable may be optimized as usual and the register remains available for allocation and use in any computations, provided that observable values of the variable are not affected.
- If the variable is referenced in inline assembly, the type of access must be provided to the compiler via constraints (see Section 6.47.3 [Constraints], page 586). Accesses from basic asms are not supported.

Note that these points *only* apply to code that is compiled with the definition. The behavior of code that is merely linked in (for example code from libraries) is not affected.

If you want to recompile source files that do not actually use your global register variable so they do not use the specified register for any other purpose, you need not actually add the global register declaration to their source code. It suffices to specify the compiler option '`-ffixed-`*`reg`*' (see Section 3.16 [Code Gen Options], page 218) to reserve the register.

Declaring the variable

Global register variables cannot have initial values, because an executable file has no means to supply initial contents for a register.

When selecting a register, choose one that is normally saved and restored by function calls on your machine. This ensures that code which is unaware of this reservation (such as library routines) will restore it before returning.

On machines with register windows, be sure to choose a global register that is not affected magically by the function call mechanism.

Using the variable

When calling routines that are not aware of the reservation, be cautious if those routines call back into code which uses them. As an example, if you call the system library version of `qsort`, it may clobber your registers during execution, but (if you have selected appropriate registers) it will restore them before returning. However it will *not* restore them before calling `qsort`'s comparison function. As a result, global values will not reliably be available to the comparison function unless the `qsort` function itself is rebuilt.

Similarly, it is not safe to access the global register variables from signal handlers or from more than one thread of control. Unless you recompile them specially for the task at hand, the system library routines may temporarily use the register for other things. Furthermore, since the register is not reserved exclusively for the variable, accessing it from handlers of asynchronous signals may observe unrelated temporary values residing in the register.

On most machines, `longjmp` restores to each global register variable the value it had at the time of the `setjmp`. On some machines, however, `longjmp` does not change the value of global register variables. To be portable, the function that called `setjmp` should make other arrangements to save the values of the global register variables, and to restore them in a `longjmp`. This way, the same thing happens regardless of what `longjmp` does.

6.47.5.2 Specifying Registers for Local Variables

You can define a local register variable and associate it with a specified register like this:

```
register int *foo asm ("r12");
```

Here `r12` is the name of the register that should be used. Note that this is the same syntax used for defining global register variables, but for a local variable the declaration appears within a function. The `register` keyword is required, and cannot be combined with `static`. The register name must be a valid register name for the target platform.

Do not use type qualifiers such as `const` and `volatile`, as the outcome may be contrary to expectations. In particular, when the `const` qualifier is used, the compiler may substitute the variable with its initializer in `asm` statements, which may cause the corresponding operand to appear in a different register.

As with global register variables, it is recommended that you choose a register that is normally saved and restored by function calls on your machine, so that calls to library routines will not clobber it.

The only supported use for this feature is to specify registers for input and output operands when calling Extended `asm` (see Section 6.47.2 [Extended Asm], page 569). This may be necessary if the constraints for a particular machine don't provide sufficient control

to select the desired register. To force an operand into a register, create a local variable and specify the register name after the variable's declaration. Then use the local variable for the `asm` operand and specify any constraint letter that matches the register:

```
register int *p1 asm ("r0") = ...;
register int *p2 asm ("r1") = ...;
register int *result asm ("r0");
asm ("sysint" : "=r" (result) : "0" (p1), "r" (p2));
```

Warning: In the above example, be aware that a register (for example `r0`) can be call-clobbered by subsequent code, including function calls and library calls for arithmetic operators on other variables (for example the initialization of `p2`). In this case, use temporary variables for expressions between the register assignments:

```
int t1 = ...;
register int *p1 asm ("r0") = ...;
register int *p2 asm ("r1") = t1;
register int *result asm ("r0");
asm ("sysint" : "=r" (result) : "0" (p1), "r" (p2));
```

Defining a register variable does not reserve the register. Other than when invoking the Extended `asm`, the contents of the specified register are not guaranteed. For this reason, the following uses are explicitly *not* supported. If they appear to work, it is only happenstance, and may stop working as intended due to (seemingly) unrelated changes in surrounding code, or even minor changes in the optimization of a future version of gcc:

- Passing parameters to or from Basic `asm`
- Passing parameters to or from Extended `asm` without using input or output operands.
- Passing parameters to or from routines written in assembler (or other languages) using non-standard calling conventions.

Some developers use Local Register Variables in an attempt to improve gcc's allocation of registers, especially in large functions. In this case the register name is essentially a hint to the register allocator. While in some instances this can generate better code, improvements are subject to the whims of the allocator/optimizers. Since there are no guarantees that your improvements won't be lost, this usage of Local Register Variables is discouraged.

On the MIPS platform, there is related use for local register variables with slightly different characteristics (see Section "Defining coprocessor specifics for MIPS targets" in *GNU Compiler Collection (GCC) Internals*).

6.47.6 Size of an `asm`

Some targets require that GCC track the size of each instruction used in order to generate correct code. Because the final length of the code produced by an `asm` statement is only known by the assembler, GCC must make an estimate as to how big it will be. It does this by counting the number of instructions in the pattern of the `asm` and multiplying that by the length of the longest instruction supported by that processor. (When working out the number of instructions, it assumes that any occurrence of a newline or of whatever statement separator character is supported by the assembler — typically '`;`' — indicates the end of an instruction.)

Normally, GCC's estimate is adequate to ensure that correct code is generated, but it is possible to confuse the compiler if you use pseudo instructions or assembler macros that expand into multiple real instructions, or if you use assembler directives that expand to

more space in the object file than is needed for a single instruction. If this happens then the assembler may produce a diagnostic saying that a label is unreachable.

This size is also used for inlining decisions. If you use `asm inline` instead of just `asm`, then for inlining purposes the size of the asm is taken as the minimum size, ignoring how many instructions GCC thinks it is.

6.48 Alternate Keywords

'`-ansi`' and the various '`-std`' options disable certain keywords. This causes trouble when you want to use GNU C extensions, or a general-purpose header file that should be usable by all programs, including ISO C programs. The keywords `asm`, `typeof` and `inline` are not available in programs compiled with '`-ansi`' or '`-std`' (although `inline` can be used in a program compiled with '`-std=c99`' or '`-std=c11`'). The ISO C99 keyword `restrict` is only available when '`-std=gnu99`' (which will eventually be the default) or '`-std=c99`' (or the equivalent '`-std=iso9899:1999`'), or an option for a later standard version, is used.

The way to solve these problems is to put '`__`' at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, and `__inline__` instead of `inline`.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

'`-pedantic`' and other options cause warnings for many GNU C extensions. You can prevent such warnings within one expression by writing `__extension__` before the expression. `__extension__` has no effect aside from this.

6.49 Incomplete `enum` Types

You can define an `enum` tag without specifying its possible values. This results in an incomplete type, much like what you get if you write `struct foo` without describing the elements. A later declaration that does specify the possible values completes the type.

You cannot allocate variables or storage using the type while it is incomplete. However, you can work with pointers to that type.

This extension may not be very useful, but it makes the handling of `enum` more consistent with the way `struct` and `union` are handled.

This extension is not supported by GNU C++.

6.50 Function Names as Strings

GCC provides three magic constants that hold the name of the current function as a string. In C++11 and later modes, all three are treated as constant expressions and can be used in `constexpr` constexts. The first of these constants is `__func__`, which is part of the C99 standard:

The identifier `__func__` is implicitly declared by the translator as if, immediately following the opening brace of each function definition, the declaration

```
static const char __func__[] = "function-name";
```

appeared, where function-name is the name of the lexically-enclosing function. This name is the unadorned name of the function. As an extension, at file (or, in C++, namespace scope), `__func__` evaluates to the empty string.

`__FUNCTION__` is another name for `__func__`, provided for backward compatibility with old versions of GCC.

In C, `__PRETTY_FUNCTION__` is yet another name for `__func__`, except that at file (or, in C++, namespace scope), it evaluates to the string `"top level"`. In addition, in C++, `__PRETTY_FUNCTION__` contains the signature of the function as well as its bare name. For example, this program:

```
extern "C" int printf (const char *, ...);

class a {
 public:
  void sub (int i)
    {
      printf ("__FUNCTION__ = %s\n", __FUNCTION__);
      printf ("__PRETTY_FUNCTION__ = %s\n", __PRETTY_FUNCTION__);
    }
};

int
main (void)
{
  a ax;
  ax.sub (0);
  return 0;
}
```

gives this output:

```
__FUNCTION__ = sub
__PRETTY_FUNCTION__ = void a::sub(int)
```

These identifiers are variables, not preprocessor macros, and may not be used to initialize `char` arrays or be concatenated with string literals.

6.51 Getting the Return or Frame Address of a Function

These functions may be used to get information about the callers of a function.

`void * __builtin_return_address` (*unsigned int* `level`) [Built-in Function]

This function returns the return address of the current function, or of one of its callers. The *level* argument is number of frames to scan up the call stack. A value of `0` yields the return address of the current function, a value of `1` yields the return address of the caller of the current function, and so forth. When inlining the expected behavior is that the function returns the address of the function that is returned to. To work around this behavior use the `noinline` function attribute.

The *level* argument must be a constant integer.

On some machines it may be impossible to determine the return address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function returns `0` or a random value. In addition, `__builtin_frame_address` may be used to determine if the top of the stack has been reached.

Additional post-processing of the returned value may be needed, see `__builtin_extract_return_addr`.

Calling this function with a nonzero argument can have unpredictable effects, including crashing the calling program. As a result, calls that are considered unsafe are diagnosed when the '`-Wframe-address`' option is in effect. Such calls should only be made in debugging situations.

`void * __builtin_extract_return_addr` (*void* *`addr`) [Built-in Function]

The address as returned by `__builtin_return_address` may have to be fed through this function to get the actual encoded address. For example, on the 31-bit S/390 platform the highest bit has to be masked out, or on SPARC platforms an offset has to be added for the true next instruction to be executed.

If no fixup is needed, this function simply passes through *addr*.

`void * __builtin_frob_return_address` (*void* *`addr`) [Built-in Function]

This function does the reverse of `__builtin_extract_return_addr`.

`void * __builtin_frame_address` (*unsigned int* `level`) [Built-in Function]

This function is similar to `__builtin_return_address`, but it returns the address of the function frame rather than the return address of the function. Calling `__builtin_frame_address` with a value of `0` yields the frame address of the current function, a value of `1` yields the frame address of the caller of the current function, and so forth.

The frame is the area on the stack that holds local variables and saved registers. The frame address is normally the address of the first word pushed on to the stack by the function. However, the exact definition depends upon the processor and the calling convention. If the processor has a dedicated frame pointer register, and the function has a frame, then `__builtin_frame_address` returns the value of the frame pointer register.

On some machines it may be impossible to determine the frame address of any function other than the current one; in such cases, or when the top of the stack has been reached, this function returns `0` if the first frame pointer is properly initialized by the startup code.

Calling this function with a nonzero argument can have unpredictable effects, including crashing the calling program. As a result, calls that are considered unsafe are diagnosed when the '`-Wframe-address`' option is in effect. Such calls should only be made in debugging situations.

6.52 Using Vector Instructions through Built-in Functions

On some targets, the instruction set contains SIMD vector instructions which operate on multiple values contained in one large register at the same time. For example, on the x86 the MMX, 3DNow! and SSE extensions can be used this way.

The first step in using these extensions is to provide the necessary data types. This should be done using an appropriate `typedef`:

```
typedef int v4si __attribute__ ((vector_size (16)));
```

The `int` type specifies the *base type*, while the attribute specifies the vector size for the variable, measured in bytes. For example, the declaration above causes the compiler to set

the mode for the `v4si` type to be 16 bytes wide and divided into `int` sized units. For a 32-bit `int` this means a vector of 4 units of 4 bytes, and the corresponding mode of `foo` is V4SI.

The `vector_size` attribute is only applicable to integral and floating scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct. Only sizes that are positive power-of-two multiples of the base type size are currently allowed.

All the basic integer types can be used as base types, both as signed and as unsigned: `char`, `short`, `int`, `long`, `long long`. In addition, `float` and `double` can be used to build floating-point vector types.

Specifying a combination that is not valid for the current architecture causes GCC to synthesize the instructions using a narrower mode. For example, if you specify a variable of type `V4SI` and your architecture does not allow for this specific SIMD type, GCC produces code that uses 4 `SIs`.

The types defined in this manner can be used with a subset of normal C operations. Currently, GCC allows using the following operators on these types: `+, -, *, /, unary minus, ^, |, &, ~, %`.

The operations behave like C++ `valarrays`. Addition is defined as the addition of the corresponding elements of the operands. For example, in the code below, each of the 4 elements in *a* is added to the corresponding 4 elements in *b* and the resulting vector is stored in *c*.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;

c = a + b;
```

Subtraction, multiplication, division, and the logical operations operate in a similar manner. Likewise, the result of using the unary minus or complement operators on a vector type is a vector whose elements are the negative or complemented values of the corresponding elements in the operand.

It is possible to use shifting operators `<<`, `>>` on integer-type vectors. The operation is defined as following: `{a0, a1, ..., an} >> {b0, b1, ..., bn} == {a0 >> b0, a1 >> b1, ..., an >> bn}`. Vector operands must have the same number of elements.

For convenience, it is allowed to use a binary vector operation where one operand is a scalar. In that case the compiler transforms the scalar operand into a vector where each element is the scalar from the operation. The transformation happens only if the scalar could be safely converted to the vector-element type. Consider the following code.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a, b, c;
long l;

a = b + 1;    /* a = b + {1,1,1,1}; */
a = 2 * b;    /* a = {2,2,2,2} * b; */

a = l + a;    /* Error, cannot convert long to int. */
```

Vectors can be subscripted as if the vector were an array with the same number of elements and base type. Out of bound accesses invoke undefined behavior at run time. Warnings for out of bound accesses for vector subscription can be enabled with '`-Warray-bounds`'.

Vector comparison is supported with standard comparison operators: `==, !=, <, <=, >, >=`. Comparison operands can be vector expressions of integer-type or real-type. Comparison between integer-type vectors and real-type vectors are not supported. The result of the comparison is a vector of the same width and number of elements as the comparison operands with a signed integral element type.

Vectors are compared element-wise producing 0 when comparison is false and -1 (constant of the appropriate type where all bits are set) otherwise. Consider the following example.

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a = {1,2,3,4};
v4si b = {3,2,1,4};
v4si c;

c = a >  b;     /* The result would be {0, 0,-1, 0}  */
c = a == b;     /* The result would be {0,-1, 0,-1}  */
```

In C++, the ternary operator `?:` is available. `a?b:c`, where `b` and `c` are vectors of the same type and `a` is an integer vector with the same number of elements of the same size as `b` and `c`, computes all three arguments and creates a vector `{a[0]?b[0]:c[0], a[1]?b[1]:c[1], ...}`. Note that unlike in OpenCL, `a` is thus interpreted as `a != 0` and not `a < 0`. As in the case of binary operations, this syntax is also accepted when one of `b` or `c` is a scalar that is then transformed into a vector. If both `b` and `c` are scalars and the type of `true?b:c` has the same size as the element type of `a`, then `b` and `c` are converted to a vector type whose elements have this type and with the same number of elements as `a`.

In C++, the logic operators `!, &&, ||` are available for vectors. `!v` is equivalent to `v == 0`, `a && b` is equivalent to `a!=0 & b!=0` and `a || b` is equivalent to `a!=0 | b!=0`. For mixed operations between a scalar `s` and a vector `v`, `s && v` is equivalent to `s?v!=0:0` (the evaluation is short-circuit) and `v && s` is equivalent to `v!=0 & (s?-1:0)`.

Vector shuffling is available using functions `__builtin_shuffle (vec, mask)` and `__builtin_shuffle (vec0, vec1, mask)`. Both functions construct a permutation of elements from one or two vectors and return a vector of the same type as the input vector(s). The *mask* is an integral vector with the same width (W) and element count (N) as the output vector.

The elements of the input vectors are numbered in memory ordering of *vec0* beginning at 0 and *vec1* beginning at N. The elements of *mask* are considered modulo N in the single-operand case and modulo $2 * N$ in the two-operand case.

Consider the following example,

```
typedef int v4si __attribute__ ((vector_size (16)));

v4si a = {1,2,3,4};
v4si b = {5,6,7,8};
v4si mask1 = {0,1,1,3};
v4si mask2 = {0,4,2,5};
v4si res;

res = __builtin_shuffle (a, mask1);       /* res is {1,2,2,4}  */
```

```
res = __builtin_shuffle (a, b, mask2);    /* res is {1,5,3,6}  */
```

Note that `__builtin_shuffle` is intentionally semantically compatible with the OpenCL `shuffle` and `shuffle2` functions.

You can declare variables and use them in function calls and returns, as well as in assignments and some casts. You can specify a vector type as a return type for a function. Vector types can also be used as function arguments. It is possible to cast from one vector type to another, provided they are of the same size (in fact, you can also cast vectors to and from other datatypes of the same size).

You cannot operate between vectors of different lengths or different signedness without a cast.

Vector conversion is available using the `__builtin_convertvector (vec, vectype)` function. *vec* must be an expression with integral or floating vector type and *vectype* an integral or floating vector type with the same number of elements. The result has *vectype* type and value of a C cast of every element of *vec* to the element type of *vectype*.

Consider the following example,

```
typedef int v4si __attribute__ ((vector_size (16)));
typedef float v4sf __attribute__ ((vector_size (16)));
typedef double v4df __attribute__ ((vector_size (32)));
typedef unsigned long long v4di __attribute__ ((vector_size (32)));

v4si a = {1,-2,3,-4};
v4sf b = {1.5f,-2.5f,3.f,7.f};
v4di c = {1ULL,5ULL,0ULL,10ULL};
v4sf d = __builtin_convertvector (a, v4sf); /* d is {1.f,-2.f,3.f,-4.f} */
/* Equivalent of:
   v4sf d = { (float)a[0], (float)a[1], (float)a[2], (float)a[3] }; */
v4df e = __builtin_convertvector (a, v4df); /* e is {1.,-2.,3.,-4.} */
v4df f = __builtin_convertvector (b, v4df); /* f is {1.5,-2.5,3.,7.} */
v4si g = __builtin_convertvector (f, v4si); /* g is {1,-2,3,7} */
v4si h = __builtin_convertvector (c, v4si); /* h is {1,5,0,10} */
```

Sometimes it is desirable to write code using a mix of generic vector operations (for clarity) and machine-specific vector intrinsics (to access vector instructions that are not exposed via generic built-ins). On x86, intrinsic functions for integer vectors typically use the same vector type `__m128i` irrespective of how they interpret the vector, making it necessary to cast their arguments and return values from/to other vector types. In C, you can make use of a `union` type:

```
#include <immintrin.h>

typedef unsigned char u8x16 __attribute__ ((vector_size (16)));
typedef unsigned int  u32x4 __attribute__ ((vector_size (16)));

typedef union {
        __m128i mm;
        u8x16   u8;
        u32x4   u32;
} v128;
```

for variables that can be used with both built-in operators and x86 intrinsics:

```
v128 x, y = { 0 };
memcpy (&x, ptr, sizeof x);
y.u8  += 0x80;
```

```
x.mm  = _mm_adds_epu8 (x.mm, y.mm);
x.u32 &= 0xffffff;

/* Instead of a variable, a compound literal may be used to pass the
   return value of an intrinsic call to a function expecting the union: */
v128 foo (v128);
x = foo ((v128) {_mm_adds_epu8 (x.mm, y.mm)});
```

6.53 Support for `offsetof`

GCC implements for both C and C++ a syntactic extension to implement the `offsetof` macro.

```
primary:
        "__builtin_offsetof" "(" typename "," offsetof_member_designator ")"

offsetof_member_designator:
          identifier
        | offsetof_member_designator "." identifier
        | offsetof_member_designator "[" expr "]"
```

This extension is sufficient such that

```
#define offsetof(type, member)  __builtin_offsetof (type, member)
```

is a suitable definition of the `offsetof` macro. In C++, *type* may be dependent. In either case, *member* may consist of a single identifier, or a sequence of member accesses and array references.

6.54 Legacy `__sync` Built-in Functions for Atomic Memory Access

The following built-in functions are intended to be compatible with those described in the *Intel Itanium Processor-specific Application Binary Interface*, section 7.4. As such, they depart from normal GCC practice by not using the '`__builtin_`' prefix and also by being overloaded so that they work on multiple types.

The definition given in the Intel documentation allows only for the use of the types `int`, `long`, `long long` or their unsigned counterparts. GCC allows any scalar type that is 1, 2, 4 or 8 bytes in size other than the C type `_Bool` or the C++ type `bool`. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

These functions are implemented in terms of the '`__atomic`' builtins (see Section 6.55 [__atomic Builtins], page 632). They should not be used for new code which should use the '`__atomic`' builtins instead.

Not all operations are supported by all target processors. If a particular operation cannot be implemented on the target processor, a warning is generated and a call to an external function is generated. The external function carries the same name as the built-in version, with an additional suffix '`_n`' where *n* is the size of the data type.

In most cases, these built-in functions are considered a *full barrier*. That is, no memory operand is moved across the operation, either forward or backward. Further, instructions are issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

All of the routines are described in the Intel documentation to take "an optional list of variables protected by the memory barrier". It's not clear what is meant by that; it could mean that *only* the listed variables are protected, or it could mean a list of additional variables to be protected. The list is ignored by GCC which treats it as empty. GCC interprets an empty list as meaning that all globally accessible variables should be protected.

type `__sync_fetch_and_add (`*type* `*ptr,` *type* `value, ...)`
type `__sync_fetch_and_sub (`*type* `*ptr,` *type* `value, ...)`
type `__sync_fetch_and_or (`*type* `*ptr,` *type* `value, ...)`
type `__sync_fetch_and_and (`*type* `*ptr,` *type* `value, ...)`
type `__sync_fetch_and_xor (`*type* `*ptr,` *type* `value, ...)`
type `__sync_fetch_and_nand (`*type* `*ptr,` *type* `value, ...)`

These built-in functions perform the operation suggested by the name, and returns the value that had previously been in memory. That is, operations on integer operands have the following semantics. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ tmp = *ptr; *ptr op= value; return tmp; }
{ tmp = *ptr; *ptr = ~(tmp & value); return tmp; }   // nand
```

The object pointed to by the first argument must be of integer or pointer type. It must not be a boolean type.

Note: GCC 4.4 and later implement `__sync_fetch_and_nand` as `*ptr = ~(tmp & value)` instead of `*ptr = ~tmp & value`.

type `__sync_add_and_fetch (`*type* `*ptr,` *type* `value, ...)`
type `__sync_sub_and_fetch (`*type* `*ptr,` *type* `value, ...)`
type `__sync_or_and_fetch (`*type* `*ptr,` *type* `value, ...)`
type `__sync_and_and_fetch (`*type* `*ptr,` *type* `value, ...)`
type `__sync_xor_and_fetch (`*type* `*ptr,` *type* `value, ...)`
type `__sync_nand_and_fetch (`*type* `*ptr,` *type* `value, ...)`

These built-in functions perform the operation suggested by the name, and return the new value. That is, operations on integer operands have the following semantics. Operations on pointer operands are performed as if the operand's type were `uintptr_t`.

```
{ *ptr op= value; return *ptr; }
{ *ptr = ~(*ptr & value); return *ptr; }   // nand
```

The same constraints on arguments apply as for the corresponding `__sync_op_and_fetch` built-in functions.

Note: GCC 4.4 and later implement `__sync_nand_and_fetch` as `*ptr = ~(*ptr & value)` instead of `*ptr = ~*ptr & value`.

`bool __sync_bool_compare_and_swap (`*type* `*ptr,` *type* `oldval,` *type* `newval, ...)`
type `__sync_val_compare_and_swap (`*type* `*ptr,` *type* `oldval,` *type* `newval, ...)`

These built-in functions perform an atomic compare and swap. That is, if the current value of `*`*ptr* is *oldval*, then write *newval* into `*`*ptr*.

The "bool" version returns `true` if the comparison is successful and *newval* is written. The "val" version returns the contents of `*`*ptr* before the operation.

`__sync_synchronize (...)`
This built-in function issues a full memory barrier.

type `__sync_lock_test_and_set (`*type* `*ptr,` *type* `value, ...)`
This built-in function, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes *value* into `*`*ptr*, and returns the previous contents of `*`*ptr*.

Many targets have only minimal support for such locks, and do not support a full exchange operation. In this case, a target may support reduced functionality here by which the *only* valid value to store is the immediate constant 1. The exact value actually stored in `*`*ptr* is implementation defined.

This built-in function is not a full barrier, but rather an *acquire barrier*. This means that references after the operation cannot move to (or be speculated to) before the operation, but previous memory stores may not be globally visible yet, and previous memory loads may not yet be satisfied.

`void __sync_lock_release (`*type* `*ptr, ...)`
This built-in function releases the lock acquired by `__sync_lock_test_and_set`. Normally this means writing the constant 0 to `*`*ptr*.

This built-in function is not a full barrier, but rather a *release barrier*. This means that all previous memory stores are globally visible, and all previous memory loads have been satisfied, but following memory reads are not prevented from being speculated to before the barrier.

6.55 Built-in Functions for Memory Model Aware Atomic Operations

The following built-in functions approximately match the requirements for the C++11 memory model. They are all identified by being prefixed with '`__atomic`' and most are overloaded so that they work with multiple types.

These functions are intended to replace the legacy '`__sync`' builtins. The main difference is that the memory order that is requested is a parameter to the functions. New code should always use the '`__atomic`' builtins rather than the '`__sync`' builtins.

Note that the '`__atomic`' builtins assume that programs will conform to the C++11 memory model. In particular, they assume that programs are free of data races. See the C++11 standard for detailed requirements.

The '`__atomic`' builtins can be used with any integral scalar or pointer type that is 1, 2, 4, or 8 bytes in length. 16-byte integral types are also allowed if '`__int128`' (see Section 6.9 [__int128], page 461) is supported by the architecture.

The four non-arithmetic functions (load, store, exchange, and compare_exchange) all have a generic version as well. This generic version works on any data type. It uses the lock-free built-in function if the specific data type size makes that possible; otherwise, an external call is left to be resolved at run time. This external call is the same format with the addition of a '`size_t`' parameter inserted as the first parameter indicating the size of the object being pointed to. All objects must be the same size.

There are 6 different memory orders that can be specified. These map to the C++11 memory orders with the same names, see the C++11 standard or the GCC wiki on atomic

synchronization (`http://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync`) for detailed definitions. Individual targets may also support additional memory orders for use on specific architectures. Refer to the target documentation for details of these.

An atomic operation can both constrain code motion and be mapped to hardware instructions for synchronization between threads (e.g., a fence). To which extent this happens is controlled by the memory orders, which are listed here in approximately ascending order of strength. The description of each memory order is only meant to roughly illustrate the effects and is not a specification; see the C++11 memory model for precise semantics.

`__ATOMIC_RELAXED`
: Implies no inter-thread ordering constraints.

`__ATOMIC_CONSUME`
: This is currently implemented using the stronger `__ATOMIC_ACQUIRE` memory order because of a deficiency in C++11's semantics for `memory_order_consume`.

`__ATOMIC_ACQUIRE`
: Creates an inter-thread happens-before constraint from the release (or stronger) semantic store to this acquire load. Can prevent hoisting of code to before the operation.

`__ATOMIC_RELEASE`
: Creates an inter-thread happens-before constraint to acquire (or stronger) semantic loads that read from this release store. Can prevent sinking of code to after the operation.

`__ATOMIC_ACQ_REL`
: Combines the effects of both `__ATOMIC_ACQUIRE` and `__ATOMIC_RELEASE`.

`__ATOMIC_SEQ_CST`
: Enforces total ordering with all other `__ATOMIC_SEQ_CST` operations.

Note that in the C++11 memory model, *fences* (e.g., '`__atomic_thread_fence`') take effect in combination with other atomic operations on specific memory locations (e.g., atomic loads); operations on specific memory locations do not necessarily affect other operations in the same way.

Target architectures are encouraged to provide their own patterns for each of the atomic built-in functions. If no target is provided, the original non-memory model set of '`__sync`' atomic built-in functions are used, along with any required synchronization fences surrounding it in order to achieve the proper behavior. Execution in this case is subject to the same restrictions as those built-in functions.

If there is no pattern or mechanism to provide a lock-free instruction sequence, a call is made to an external routine with the same parameters to be resolved at run time.

When implementing patterns for these built-in functions, the memory order parameter can be ignored as long as the pattern implements the most restrictive `__ATOMIC_SEQ_CST` memory order. Any of the other memory orders execute correctly with this memory order but they may not execute as efficiently as they could with a more appropriate implementation of the relaxed requirements.

Note that the C++11 standard allows for the memory order parameter to be determined at run time rather than at compile time. These built-in functions map any run-time value

to `__ATOMIC_SEQ_CST` rather than invoke a runtime library call or inline a switch statement. This is standard compliant, safe, and the simplest approach for now.

The memory order parameter is a signed int, but only the lower 16 bits are reserved for the memory order. The remainder of the signed int is reserved for target use and should be 0. Use of the predefined atomic values ensures proper usage.

type `__atomic_load_n` (*type* **ptr*, *int memorder*) [Built-in Function]

This built-in function implements an atomic load operation. It returns the contents of `*ptr`.

The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, `__ATOMIC_ACQUIRE`, and `__ATOMIC_CONSUME`.

`void __atomic_load` (*type* **ptr*, *type* **ret*, *int memorder*) [Built-in Function]

This is the generic version of an atomic load. It returns the contents of `*ptr` in `*ret`.

`void __atomic_store_n` (*type* **ptr*, *type val*, *int memorder*) [Built-in Function]

This built-in function implements an atomic store operation. It writes `val` into `*ptr`.

The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, and `__ATOMIC_RELEASE`.

`void __atomic_store` (*type* **ptr*, *type* **val*, *int memorder*) [Built-in Function]

This is the generic version of an atomic store. It stores the value of `*val` into `*ptr`.

type `__atomic_exchange_n` (*type* **ptr*, *type val*, *int memorder*) [Built-in Function]

This built-in function implements an atomic exchange operation. It writes *val* into `*ptr`, and returns the previous contents of `*ptr`.

The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, `__ATOMIC_ACQUIRE`, `__ATOMIC_RELEASE`, and `__ATOMIC_ACQ_REL`.

`void __atomic_exchange` (*type* **ptr*, *type* **val*, *type* **ret*, *int memorder*) [Built-in Function]

This is the generic version of an atomic exchange. It stores the contents of `*val` into `*ptr`. The original value of `*ptr` is copied into `*ret`.

`bool __atomic_compare_exchange_n` (*type* **ptr*, *type* **expected*, *type desired*, *bool weak*, *int success_memorder*, *int failure_memorder*) [Built-in Function]

This built-in function implements an atomic compare and exchange operation. This compares the contents of `*ptr` with the contents of `*expected`. If equal, the operation is a *read-modify-write* operation that writes *desired* into `*ptr`. If they are not equal, the operation is a *read* and the current contents of `*ptr` are written into `*expected`. *weak* is `true` for weak compare_exchange, which may fail spuriously, and `false` for the strong variation, which never fails spuriously. Many targets only offer the strong variation and ignore the parameter. When in doubt, use the strong variation.

If *desired* is written into `*ptr` then `true` is returned and memory is affected according to the memory order specified by *success_memorder*. There are no restrictions on what memory order can be used here.

Otherwise, `false` is returned and memory is affected according to *failure_memorder*. This memory order cannot be `__ATOMIC_RELEASE` nor `__ATOMIC_ACQ_REL`. It also cannot be a stronger order than that specified by *success_memorder*.

`bool __atomic_compare_exchange (type *ptr, type *expected, type *desired, bool weak, int success_memorder, int failure_memorder)` [Built-in Function]

This built-in function implements the generic version of `__atomic_compare_exchange`. The function is virtually identical to `__atomic_compare_exchange_n`, except the desired value is also a pointer.

`type __atomic_add_fetch (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_sub_fetch (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_and_fetch (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_xor_fetch (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_or_fetch (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_nand_fetch (type *ptr, type val, int memorder)` [Built-in Function]

These built-in functions perform the operation suggested by the name, and return the result of the operation. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ *ptr op= val; return *ptr; }
{ *ptr = ~(*ptr & val); return *ptr; } // nand
```

The object pointed to by the first argument must be of integer or pointer type. It must not be a boolean type. All memory orders are valid.

`type __atomic_fetch_add (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_fetch_sub (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_fetch_and (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_fetch_xor (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_fetch_or (type *ptr, type val, int memorder)` [Built-in Function]
`type __atomic_fetch_nand (type *ptr, type val, int memorder)` [Built-in Function]

These built-in functions perform the operation suggested by the name, and return the value that had previously been in `*ptr`. Operations on pointer arguments are performed as if the operands were of the `uintptr_t` type. That is, they are not scaled by the size of the type to which the pointer points.

```
{ tmp = *ptr; *ptr op= val; return tmp; }
{ tmp = *ptr; *ptr = ~(*ptr & val); return tmp; } // nand
```

The same constraints on arguments apply as for the corresponding `__atomic_op_fetch` built-in functions. All memory orders are valid.

`bool __atomic_test_and_set (void *ptr, int memorder)` [Built-in Function]

This built-in function performs an atomic test-and-set operation on the byte at `*ptr`. The byte is set to some implementation defined nonzero "set" value and the return value is `true` if and only if the previous contents were "set". It should be only used for operands of type `bool` or `char`. For other types only part of the value may be set.

All memory orders are valid.

`void __atomic_clear` (*bool *ptr, int memorder*) [Built-in Function]

This built-in function performs an atomic clear operation on `*ptr`. After the operation, `*ptr` contains 0. It should be only used for operands of type `bool` or `char` and in conjunction with `__atomic_test_and_set`. For other types it may only clear partially. If the type is not `bool` prefer using `__atomic_store`.

The valid memory order variants are `__ATOMIC_RELAXED`, `__ATOMIC_SEQ_CST`, and `__ATOMIC_RELEASE`.

`void __atomic_thread_fence` (*int memorder*) [Built-in Function]

This built-in function acts as a synchronization fence between threads based on the specified memory order.

All memory orders are valid.

`void __atomic_signal_fence` (*int memorder*) [Built-in Function]

This built-in function acts as a synchronization fence between a thread and signal handlers based in the same thread.

All memory orders are valid.

`bool __atomic_always_lock_free` (*size_t size, void *ptr*) [Built-in Function]

This built-in function returns `true` if objects of *size* bytes always generate lock-free atomic instructions for the target architecture. *size* must resolve to a compile-time constant and the result also resolves to a compile-time constant.

ptr is an optional pointer to the object that may be used to determine alignment. A value of 0 indicates typical alignment should be used. The compiler may also ignore this parameter.

```
if (__atomic_always_lock_free (sizeof (long long), 0))
```

`bool __atomic_is_lock_free` (*size_t size, void *ptr*) [Built-in Function]

This built-in function returns `true` if objects of *size* bytes always generate lock-free atomic instructions for the target architecture. If the built-in function is not known to be lock-free, a call is made to a runtime routine named `__atomic_is_lock_free`.

ptr is an optional pointer to the object that may be used to determine alignment. A value of 0 indicates typical alignment should be used. The compiler may also ignore this parameter.

6.56 Built-in Functions to Perform Arithmetic with Overflow Checking

The following built-in functions allow performing simple arithmetic operations together with checking whether the operations overflowed.

`bool __builtin_add_overflow` (*type1 a, type2 b, type3 *res*) [Built-in Function]

`bool __builtin_sadd_overflow` (*int a, int b, int *res*) [Built-in Function]

`bool __builtin_saddl_overflow` (*long int a, long int b, long int *res*) [Built-in Function]

`bool __builtin_saddll_overflow` (*long long int a, long long int b, long long int *res*) [Built-in Function]

`bool __builtin_uadd_overflow` (*unsigned int a, unsigned int b, unsigned int *res*) [Built-in Function]

`bool __builtin_uaddl_overflow` (*unsigned long int a, unsigned long int b, unsigned long int *res*) [Built-in Function]

`bool __builtin_uaddll_overflow` (*unsigned long long int a, unsigned long long int b, unsigned long long int *res*) [Built-in Function]

These built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type the third pointer argument points to and stored there. If the stored result is equal to the infinite precision result, the built-in functions return `false`, otherwise they return `true`. As the addition is performed in infinite signed precision, these built-in functions have fully defined behavior for all argument values.

The first built-in function allows arbitrary integral types for operands and the result type must be pointer to some integral type other than enumerated or boolean type, the rest of the built-in functions have explicit integer types.

The compiler will attempt to use hardware instructions to implement these built-in functions where possible, like conditional jump on overflow after addition, conditional jump on carry etc.

`bool __builtin_sub_overflow` (*type1 a, type2 b, type3 *res*) [Built-in Function]

`bool __builtin_ssub_overflow` (*int a, int b, int *res*) [Built-in Function]

`bool __builtin_ssubl_overflow` (*long int a, long int b, long int *res*) [Built-in Function]

`bool __builtin_ssubll_overflow` (*long long int a, long long int b, long long int *res*) [Built-in Function]

`bool __builtin_usub_overflow` (*unsigned int a, unsigned int b, unsigned int *res*) [Built-in Function]

`bool __builtin_usubl_overflow` (*unsigned long int a, unsigned long int b, unsigned long int *res*) [Built-in Function]

`bool __builtin_usubll_overflow` (*unsigned long long int a, unsigned long long int b, unsigned long long int *res*) [Built-in Function]

These built-in functions are similar to the add overflow checking built-in functions above, except they perform subtraction, subtract the second argument from the first one, instead of addition.

`bool __builtin_mul_overflow` (*type1 a, type2 b, type3 *res*) [Built-in Function]

`bool __builtin_smul_overflow` (*int a, int b, int *res*) [Built-in Function]

`bool __builtin_smull_overflow` (*long int a, long int b, long int *res*) [Built-in Function]

`bool __builtin_smulll_overflow` (*long long int a, long long int b, long long int *res*) [Built-in Function]

`bool __builtin_umul_overflow` (*unsigned int a, unsigned int b, unsigned int *res*) [Built-in Function]

`bool __builtin_umull_overflow` (*unsigned long int a, unsigned long int b, unsigned long int *res*) [Built-in Function]

`bool __builtin_umulll_overflow` (*unsigned long long int a, unsigned long long int b, unsigned long long int *res*) [Built-in Function]

These built-in functions are similar to the add overflow checking built-in functions above, except they perform multiplication, instead of addition.

The following built-in functions allow checking if simple arithmetic operation would overflow.

`bool __builtin_add_overflow_p` (*type1 a, type2 b, type3 c*) [Built-in Function]

`bool __builtin_sub_overflow_p` (*type1 a, type2 b, type3 c*) [Built-in Function]

`bool __builtin_mul_overflow_p` (*type1 a, type2 b, type3 c*) [Built-in Function]

These built-in functions are similar to `__builtin_add_overflow`, `__builtin_sub_overflow`, or `__builtin_mul_overflow`, except that they don't store the result of the arithmetic operation anywhere and the last argument is not a pointer, but some expression with integral type other than enumerated or boolean type.

The built-in functions promote the first two operands into infinite precision signed type and perform addition on those promoted operands. The result is then cast to the type of the third argument. If the cast result is equal to the infinite precision result, the built-in functions return `false`, otherwise they return `true`. The value of the third argument is ignored, just the side effects in the third argument are evaluated, and no integral argument promotions are performed on the last argument. If the third argument is a bit-field, the type used for the result cast has the precision and signedness of the given bit-field, rather than precision and signedness of the underlying type.

For example, the following macro can be used to portably check, at compile-time, whether or not adding two constant integers will overflow, and perform the addition only when it is known to be safe and not to trigger a '`-Woverflow`' warning.

```
#define INT_ADD_OVERFLOW_P(a, b) \
   __builtin_add_overflow_p (a, b, (__typeof__ ((a) + (b))) 0)

enum {
    A = INT_MAX, B = 3,
    C = INT_ADD_OVERFLOW_P (A, B) ? 0 : A + B,
    D = __builtin_add_overflow_p (1, SCHAR_MAX, (signed char) 0)
};
```

The compiler will attempt to use hardware instructions to implement these built-in functions where possible, like conditional jump on overflow after addition, conditional jump on carry etc.

6.57 x86-Specific Memory Model Extensions for Transactional Memory

The x86 architecture supports additional memory ordering flags to mark critical sections for hardware lock elision. These must be specified in addition to an existing memory order to atomic intrinsics.

`__ATOMIC_HLE_ACQUIRE`
: Start lock elision on a lock variable. Memory order must be `__ATOMIC_ACQUIRE` or stronger.

`__ATOMIC_HLE_RELEASE`
: End lock elision on a lock variable. Memory order must be `__ATOMIC_RELEASE` or stronger.

When a lock acquire fails, it is required for good performance to abort the transaction quickly. This can be done with a `_mm_pause`.

```
#include <immintrin.h> // For _mm_pause

int lockvar;

/* Acquire lock with lock elision */
while (__atomic_exchange_n(&lockvar, 1, __ATOMIC_ACQUIRE|__ATOMIC_HLE_ACQUIRE))
    _mm_pause(); /* Abort failed transaction */
...
/* Free lock with lock elision */
__atomic_store_n(&lockvar, 0, __ATOMIC_RELEASE|__ATOMIC_HLE_RELEASE);
```

6.58 Object Size Checking Built-in Functions

GCC implements a limited buffer overflow protection mechanism that can prevent some buffer overflow attacks by determining the sizes of objects into which data is about to be written and preventing the writes when the size isn't sufficient. The built-in functions described below yield the best results when used together and when optimization is enabled. For example, to detect object sizes across function boundaries or to follow pointer assignments through non-trivial control flow they rely on various optimization passes enabled with '`-O2`'. However, to a limited extent, they can be used without optimization as well.

`size_t __builtin_object_size` (*const void * ptr, int type*) [Built-in Function]

is a built-in construct that returns a constant number of bytes from *ptr* to the end of the object *ptr* pointer points to (if known at compile time). To determine the sizes of dynamically allocated objects the function relies on the allocation functions called to obtain the storage to be declared with the `alloc_size` attribute (see Section 6.33.1 [Common Function Attributes], page 478). `__builtin_object_size` never evaluates its arguments for side effects. If there are any side effects in them, it returns `(size_t) -1` for *type* 0 or 1 and `(size_t) 0` for *type* 2 or 3. If there are multiple objects *ptr* can point to and all of them are known at compile time, the returned number is the maximum of remaining byte counts in those objects if *type* & 2 is 0 and minimum if nonzero. If it is not possible to determine which objects *ptr* points to at compile time, `__builtin_object_size` should return `(size_t) -1` for *type* 0 or 1 and `(size_t) 0` for *type* 2 or 3.

type is an integer constant from 0 to 3. If the least significant bit is clear, objects are whole variables, if it is set, a closest surrounding subobject is considered the object a pointer points to. The second bit determines if maximum or minimum of remaining bytes is computed.

```
struct V { char buf1[10]; int b; char buf2[10]; } var;
char *p = &var.buf1[1], *q = &var.b;
```

```
/* Here the object p points to is var.  */
assert (__builtin_object_size (p, 0) == sizeof (var) - 1);
/* The subobject p points to is var.buf1.  */
assert (__builtin_object_size (p, 1) == sizeof (var.buf1) - 1);
/* The object q points to is var.  */
assert (__builtin_object_size (q, 0)
        == (char *) (&var + 1) - (char *) &var.b);
/* The subobject q points to is var.b.  */
assert (__builtin_object_size (q, 1) == sizeof (var.b));
```

There are built-in functions added for many common string operation functions, e.g., for `memcpy` `__builtin___memcpy_chk` built-in is provided. This built-in has an additional last argument, which is the number of bytes remaining in the object the *dest* argument points to or `(size_t) -1` if the size is not known.

The built-in functions are optimized into the normal string functions like `memcpy` if the last argument is `(size_t) -1` or if it is known at compile time that the destination object will not be overflowed. If the compiler can determine at compile time that the object will always be overflowed, it issues a warning.

The intended use can be e.g.

```
#undef memcpy
#define bos0(dest) __builtin_object_size (dest, 0)
#define memcpy(dest, src, n) \
  __builtin___memcpy_chk (dest, src, n, bos0 (dest))

char *volatile p;
char buf[10];
/* It is unknown what object p points to, so this is optimized
   into plain memcpy - no checking is possible.  */
memcpy (p, "abcde", n);
/* Destination is known and length too.  It is known at compile
   time there will be no overflow.  */
memcpy (&buf[5], "abcde", 5);
/* Destination is known, but the length is not known at compile time.
   This will result in __memcpy_chk call that can check for overflow
   at run time.  */
memcpy (&buf[5], "abcde", n);
/* Destination is known and it is known at compile time there will
   be overflow.  There will be a warning and __memcpy_chk call that
   will abort the program at run time.  */
memcpy (&buf[6], "abcde", 5);
```

Such built-in functions are provided for `memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat` and `strncat`.

There are also checking built-in functions for formatted output functions.

```
int __builtin___sprintf_chk (char *s, int flag, size_t os, const char *fmt, ...);
int __builtin___snprintf_chk (char *s, size_t maxlen, int flag, size_t os,
                              const char *fmt, ...);
int __builtin___vsprintf_chk (char *s, int flag, size_t os, const char *fmt,
                              va_list ap);
int __builtin___vsnprintf_chk (char *s, size_t maxlen, int flag, size_t os,
                               const char *fmt, va_list ap);
```

The added *flag* argument is passed unchanged to `__sprintf_chk` etc. functions and can contain implementation specific flags on what additional security measures the checking function might take, such as handling `%n` differently.

The *os* argument is the object size *s* points to, like in the other built-in functions. There is a small difference in the behavior though, if *os* is `(size_t) -1`, the built-in functions are optimized into the non-checking functions only if *flag* is 0, otherwise the checking function is called with *os* argument set to `(size_t) -1`.

In addition to this, there are checking built-in functions `__builtin___printf_chk`, `__builtin___vprintf_chk`, `__builtin___fprintf_chk` and `__builtin___vfprintf_chk`. These have just one additional argument, *flag*, right before format string *fmt*. If the compiler is able to optimize them to `fputc` etc. functions, it does, otherwise the checking function is called and the *flag* argument passed to it.

6.59 Other Built-in Functions Provided by GCC

GCC provides a large number of built-in functions other than the ones mentioned above. Some of these are for internal use in the processing of exceptions or variable-length argument lists and are not documented here because they may change from time to time; we do not recommend general use of these functions.

The remaining functions are provided for optimization purposes.

With the exception of built-ins that have library equivalents such as the standard C library functions discussed below, or that expand to library calls, GCC built-in functions are always expanded inline and thus do not have corresponding entry points and their address cannot be obtained. Attempting to use them in an expression other than a function call results in a compile-time error.

GCC includes built-in versions of many of the functions in the standard C library. These functions come in two forms: one whose names start with the `__builtin_` prefix, and the other without. Both forms have the same type (including prototype), the same address (when their address is taken), and the same meaning as the C library functions even if you specify the '`-fno-builtin`' option see Section 3.4 [C Dialect Options], page 37). Many of these functions are only optimized in certain cases; if they are not optimized in a particular case, a call to the library function is emitted.

Outside strict ISO C mode ('`-ansi`', '`-std=c90`', '`-std=c99`' or '`-std=c11`'), the functions `_exit`, `alloca`, `bcmp`, `bzero`, `dcgettext`, `dgettext`, `dremf`, `dreml`, `drem`, `exp10f`, `exp10l`, `exp10`, `ffsll`, `ffsl`, `ffs`, `fprintf_unlocked`, `fputs_unlocked`, `gammaf`, `gammal`, `gamma`, `gammaf_r`, `gammal_r`, `gamma_r`, `gettext`, `index`, `isascii`, `j0f`, `j0l`, `j0`, `j1f`, `j1l`, `j1`, `jnf`, `jnl`, `jn`, `lgammaf_r`, `lgammal_r`, `lgamma_r`, `mempcpy`, `pow10f`, `pow10l`, `pow10`, `printf_unlocked`, `rindex`, `scalbf`, `scalbl`, `scalb`, `signbit`, `signbitf`, `signbitl`, `signbitd32`, `signbitd64`, `signbitd128`, `significandf`, `significandl`, `significand`, `sincosf`, `sincosl`, `sincos`, `stpcpy`, `stpncpy`, `strcasecmp`, `strdup`, `strfmon`, `strncasecmp`, `strndup`, `strnlen`, `toascii`, `y0f`, `y0l`, `y0`, `y1f`, `y1l`, `y1`, `ynf`, `ynl` and `yn` may be handled as built-in functions. All these functions have corresponding versions prefixed with `__builtin_`, which may be used even in strict C90 mode.

The ISO C99 functions `_Exit`, `acoshf`, `acoshl`, `acosh`, `asinhf`, `asinhl`, `asinh`, `atanhf`, `atanhl`, `atanh`, `cabsf`, `cabsl`, `cabs`, `cacosf`, `cacoshf`, `cacoshl`, `cacosh`, `cacosl`, `cacos`, `cargf`, `cargl`, `carg`, `casinf`, `casinhf`, `casinhl`, `casinh`, `casinl`, `casin`, `catanf`, `catanhf`, `catanhl`, `catanh`, `catanl`, `catan`, `cbrtf`, `cbrtl`, `cbrt`, `ccosf`, `ccoshf`, `ccoshl`, `ccosh`, `ccosl`, `ccos`, `cexpf`, `cexpl`, `cexp`, `cimagf`, `cimagl`, `cimag`, `clogf`, `clogl`, `clog`, `conjf`, `conjl`, `conj`, `copysignf`, `copysignl`, `copysign`, `cpowf`, `cpowl`, `cpow`, `cprojf`,

cprojl, cproj, crealf, creall, creal, csinf, csinhf, csinhl, csinh, csinl, csin, csqrtf, csqrtl, csqrt, ctanf, ctanhf, ctanhl, ctanh, ctanl, ctan, erfcf, erfcl, erfc, erff, erfl, erf, exp2f, exp2l, exp2, expm1f, expm1l, expm1, fdimf, fdiml, fdim, fmaf, fmal, fmaxf, fmaxl, fmax, fma, fminf, fminl, fmin, hypotf, hypotl, hypot, ilogbf, ilogbl, ilogb, imaxabs, isblank, iswblank, lgammaf, lgammal, lgamma, llabs, llrintf, llrintl, llrint, llroundf, llroundl, llround, log1pf, log1pl, log1p, log2f, log2l, log2, logbf, logbl, logb, lrintf, lrintl, lrint, lroundf, lroundl, lround, nearbyintf, nearbyintl, nearbyint, nextafterf, nextafterl, nextafter, nexttowardf, nexttowardl, nexttoward, remainderf, remainderl, remainder, remquof, remquol, remquo, rintf, rintl, rint, roundf, roundl, round, scalblnf, scalblnl, scalbln, scalbnf, scalbnl, scalbn, snprintf, tgammaf, tgammal, tgamma, truncf, truncl, trunc, vfscanf, vscanf, vsnprintf and vsscanf are handled as built-in functions except in strict ISO C90 mode ('-ansi' or '-std=c90').

There are also built-in versions of the ISO C99 functions acosf, acosl, asinf, asinl, atan2f, atan2l, atanf, atanl, ceilf, ceill, cosf, coshf, coshl, cosl, expf, expl, fabsf, fabsl, floorf, floorl, fmodf, fmodl, frexpf, frexpl, ldexpf, ldexpl, log10f, log10l, logf, logl, modfl, modf, powf, powl, sinf, sinhf, sinhl, sinl, sqrtf, sqrtl, tanf, tanhf, tanhl and tanl that are recognized in any mode since ISO C90 reserves these names for the purpose to which ISO C99 puts them. All these functions have corresponding versions prefixed with __builtin_.

There are also built-in functions __builtin_fabsf*n*, __builtin_fabsf*n*x, __builtin_copysignf*n* and __builtin_copysignf*n*x, corresponding to the TS 18661-3 functions fabsf*n*, fabsf*n*x, copysignf*n* and copysignf*n*x, for supported types _Float*n* and _Float*n*x.

There are also GNU extension functions clog10, clog10f and clog10l which names are reserved by ISO C99 for future use. All these functions have versions prefixed with __builtin_.

The ISO C94 functions iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, iswxdigit, towlower and towupper are handled as built-in functions except in strict ISO C90 mode ('-ansi' or '-std=c90').

The ISO C90 functions abort, abs, acos, asin, atan2, atan, calloc, ceil, cosh, cos, exit, exp, fabs, floor, fmod, fprintf, fputs, frexp, fscanf, isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper, labs, ldexp, log10, log, malloc, memchr, memcmp, memcpy, memset, modf, pow, printf, putchar, puts, scanf, sinh, sin, snprintf, sprintf, sqrt, sscanf, strcat, strchr, strcmp, strcpy, strcspn, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, tanh, tan, vfprintf, vprintf and vsprintf are all recognized as built-in functions unless '-fno-builtin' is specified (or '-fno-builtin-*function*' is specified for an individual function). All of these functions have corresponding versions prefixed with __builtin_.

GCC provides built-in versions of the ISO C99 floating-point comparison macros that avoid raising exceptions for unordered operands. They have the same names as the standard macros (isgreater, isgreaterequal, isless, islessequal, islessgreater, and isunordered) , with __builtin_ prefixed. We intend for a library implementor to be able to simply #define each standard macro to its built-in equivalent. In the same fashion,

GCC provides `fpclassify`, `isfinite`, `isinf_sign`, `isnormal` and `signbit` built-ins used with `__builtin_` prefixed. The `isinf` and `isnan` built-in functions appear both with and without the `__builtin_` prefix.

`void *__builtin_alloca` (*size_t size*) [Built-in Function]

The `__builtin_alloca` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The object is aligned on the default stack alignment boundary for the target determined by the `__BIGGEST_ALIGNMENT__` macro. The `__builtin_alloca` function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends just before the calling function returns to its caller. This is so even when `__builtin_alloca` is called within a nested block.

For example, the following function allocates eight objects of `n` bytes each on the stack, storing a pointer to each in consecutive elements of the array `a`. It then passes the array to function `g` which can safely use the storage pointed to by each of the array elements.

```
void f (unsigned n)
{
  void *a [8];
  for (int i = 0; i != 8; ++i)
    a [i] = __builtin_alloca (n);

  g (a, n);   // safe
}
```

Since the `__builtin_alloca` function doesn't validate its argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca` function is provided to make it possible to allocate on the stack arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer similar functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Section 6.20 [Variable Length], page 470, for details.

`void *__builtin_alloca_with_align` (*size_t size, size_t alignment*) [Built-in Function]

The `__builtin_alloca_with_align` function must be called at block scope. The function allocates an object *size* bytes large on the stack of the calling function. The allocated object is aligned on the boundary specified by the argument *alignment* whose unit is given in bits (not bytes). The *size* argument must be positive and not exceed the stack size limit. The *alignment* argument must be a constant integer expression that evaluates to a power of 2 greater than or equal to `CHAR_BIT` and less than some unspecified maximum. Invocations with other values are rejected with an error indicating the valid bounds. The function returns a pointer to the first byte of the allocated object. The lifetime of the allocated object ends at the end of the block in which the function was called. The allocated storage is released no later than just before the calling function returns to its caller, but may be released at the end of the block in which the function was called.

For example, in the following function the call to `g` is unsafe because when `overalign` is non-zero, the space allocated by `__builtin_alloca_with_align` may have been released at the end of the `if` statement in which it was called.

```
void f (unsigned n, bool overalign)
{
  void *p;
  if (overalign)
    p = __builtin_alloca_with_align (n, 64 /* bits */);
  else
    p = __builtin_alloc (n);

  g (p, n);   // unsafe
}
```

Since the `__builtin_alloca_with_align` function doesn't validate its *size* argument it is the responsibility of its caller to make sure the argument doesn't cause it to exceed the stack size limit. The `__builtin_alloca_with_align` function is provided to make it possible to allocate on the stack overaligned arrays of bytes with an upper bound that may be computed at run time. Since C99 Variable Length Arrays offer the same functionality under a portable, more convenient, and safer interface they are recommended instead, in both C99 and C++ programs where GCC provides them as an extension. See Section 6.20 [Variable Length], page 470, for details.

`void *__builtin_alloca_with_align_and_max` (*size_t size, size_t alignment, size_t max_size*) [Built-in Function]

Similar to `__builtin_alloca_with_align` but takes an extra argument specifying an upper bound for *size* in case its value cannot be computed at compile time, for use by '`-fstack-usage`', '`-Wstack-usage`' and '`-Walloca-larger-than`'. *max_size* must be a constant integer expression, it has no effect on code generation and no attempt is made to check its compatibility with *size*.

`bool __builtin_has_attribute` (*type-or-expression, attribute*) [Built-in Function]

The `__builtin_has_attribute` function evaluates to an integer constant expression equal to `true` if the symbol or type referenced by the *type-or-expression* argument has been declared with the *attribute* referenced by the second argument. For an *type-or-expression* argument that does not reference a symbol, since attributes do not apply to expressions the built-in consider the type of the argument. Neither argument is evaluated. The *type-or-expression* argument is subject to the same restrictions as the argument to `typeof` (see Section 6.7 [Typeof], page 459). The *attribute* argument is an attribute name optionally followed by a comma-separated list of arguments enclosed in parentheses. Both forms of attribute names—with and without double leading and trailing underscores—are recognized. See Section 6.39 [Attribute Syntax], page 559, for details. When no attribute arguments are specified for an attribute that expects one or more arguments the function returns `true` if *type-or-expression* has been declared with the attribute regardless of the attribute argument values. Arguments provided for an attribute that expects some are validated and matched up to the provided number. The function returns `true` if all provided arguments match. For example, the first call to the function below evaluates to `true` because `x`

is declared with the `aligned` attribute but the second call evaluates to `false` because `x` is declared `aligned (8)` and not `aligned (4)`.

```
__attribute__ ((aligned (8))) int x;
_Static_assert (__builtin_has_attribute (x, aligned), "aligned");
_Static_assert (!__builtin_has_attribute (x, aligned (4)), "aligned (4)");
```

Due to a limitation the `__builtin_has_attribute` function returns `false` for the `mode` attribute even if the type or variable referenced by the *type-or-expression* argument was declared with one. The function is also not supported with labels, and in C with enumerators.

Note that unlike the `__has_attribute` preprocessor operator which is suitable for use in `#if` preprocessing directives `__builtin_has_attribute` is an intrinsic function that is not recognized in such contexts.

type `__builtin_speculation_safe_value` (*type val*, *type failval*) [Built-in Function]

This built-in function can be used to help mitigate against unsafe speculative execution. *type* may be any integral type or any pointer type.

1. If the CPU is not speculatively executing the code, then *val* is returned.
2. If the CPU is executing speculatively then either:
   - The function may cause execution to pause until it is known that the code is no-longer being executed speculatively (in which case *val* can be returned, as above); or
   - The function may use target-dependent speculation tracking state to cause *failval* to be returned when it is known that speculative execution has incorrectly predicted a conditional branch operation.

The second argument, *failval*, is optional and defaults to zero if omitted.

GCC defines the preprocessor macro `__HAVE_BUILTIN_SPECULATION_SAFE_VALUE` for targets that have been updated to support this builtin.

The built-in function can be used where a variable appears to be used in a safe way, but the CPU, due to speculative execution may temporarily ignore the bounds checks. Consider, for example, the following function:

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return array[untrusted_index];
  return 0;
}
```

If the function is called repeatedly with `untrusted_index` less than the limit of 500, then a branch predictor will learn that the block of code that returns a value stored in `array` will be executed. If the function is subsequently called with an out-of-range value it will still try to execute that block of code first until the CPU determines that the prediction was incorrect (the CPU will unwind any incorrect operations at that point). However, depending on how the result of the function is used, it might be possible to leave traces in the cache that can reveal what was stored at the out-of-bounds location. The built-in function can be used to provide some protection against leaking data in this way by changing the code to:

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return array[__builtin_speculation_safe_value (untrusted_index)];
  return 0;
}
```

The built-in function will either cause execution to stall until the conditional branch has been fully resolved, or it may permit speculative execution to continue, but using 0 instead of `untrusted_value` if that exceeds the limit.

If accessing any memory location is potentially unsafe when speculative execution is incorrect, then the code can be rewritten as

```
int array[500];
int f (unsigned untrusted_index)
{
  if (untrusted_index < 500)
    return *__builtin_speculation_safe_value (&array[untrusted_index], NULL);
  return 0;
}
```

which will cause a `NULL` pointer to be used for the unsafe case.

`int __builtin_types_compatible_p (`*type1*`, `*type2*`)` [Built-in Function]

You can use the built-in function `__builtin_types_compatible_p` to determine whether two types are the same.

This built-in function returns 1 if the unqualified versions of the types *type1* and *type2* (which are types, not expressions) are compatible, 0 otherwise. The result of this built-in function can be used in integer constant expressions.

This built-in function ignores top level qualifiers (e.g., `const`, `volatile`). For example, `int` is equivalent to `const int`.

The type `int[]` and `int[5]` are compatible. On the other hand, `int` and `char *` are not compatible, even if the size of their types, on the particular architecture are the same. Also, the amount of pointer indirection is taken into account when determining similarity. Consequently, `short *` is not similar to `short **`. Furthermore, two types that are typedefed are considered compatible if their underlying types are compatible.

An `enum` type is not considered to be compatible with another `enum` type even if both are compatible with the same integer type; this is what the C standard specifies. For example, `enum {foo, bar}` is not similar to `enum {hot, dog}`.

You typically use this function in code whose execution varies depending on the arguments' types. For example:

```
#define foo(x)                                                  \
  ({                                                           \
    typeof (x) tmp = (x);                                       \
    if (__builtin_types_compatible_p (typeof (x), long double)) \
      tmp = foo_long_double (tmp);                              \
    else if (__builtin_types_compatible_p (typeof (x), double)) \
      tmp = foo_double (tmp);                                   \
    else if (__builtin_types_compatible_p (typeof (x), float))  \
      tmp = foo_float (tmp);                                    \
    else                                                        \
      abort ();                                                 \
```

```
    tmp;                                                    \
  })
```

Note: This construct is only available for C.

type `__builtin_call_with_static_chain` (*call_exp*, *pointer_exp*) [Built-in Function]

The *call_exp* expression must be a function call, and the *pointer_exp* expression must be a pointer. The *pointer_exp* is passed to the function call in the target's static chain location. The result of builtin is the result of the function call.

Note: This builtin is only available for C. This builtin can be used to call Go closures from C.

type `__builtin_choose_expr` (*const_exp*, *exp1*, *exp2*) [Built-in Function]

You can use the built-in function `__builtin_choose_expr` to evaluate code depending on the value of a constant expression. This built-in function returns *exp1* if *const_exp*, which is an integer constant expression, is nonzero. Otherwise it returns *exp2*.

This built-in function is analogous to the '`? :`' operator in C, except that the expression returned has its type unaltered by promotion rules. Also, the built-in function does not evaluate the expression that is not chosen. For example, if *const_exp* evaluates to `true`, *exp2* is not evaluated even if it has side effects.

This built-in function can return an lvalue if the chosen argument is an lvalue.

If *exp1* is returned, the return type is the same as *exp1*'s type. Similarly, if *exp2* is returned, its return type is the same as *exp2*.

Example:

```
#define foo(x)                                                  \
  __builtin_choose_expr (                                       \
    __builtin_types_compatible_p (typeof (x), double),          \
    foo_double (x),                                             \
    __builtin_choose_expr (                                     \
      __builtin_types_compatible_p (typeof (x), float),         \
      foo_float (x),                                            \
      /* The void expression results in a compile-time error   \
         when assigning the result to something.  */          \
      (void)0))
```

Note: This construct is only available for C. Furthermore, the unused expression (*exp1* or *exp2* depending on the value of *const_exp*) may still generate syntax errors. This may change in future revisions.

type `__builtin_tgmath` (*functions*, *arguments*) [Built-in Function]

The built-in function `__builtin_tgmath`, available only for C and Objective-C, calls a function determined according to the rules of `<tgmath.h>` macros. It is intended to be used in implementations of that header, so that expansions of macros from that header only expand each of their arguments once, to avoid problems when calls to such macros are nested inside the arguments of other calls to such macros; in addition, it results in better diagnostics for invalid calls to `<tgmath.h>` macros than implementations using other GNU C language features. For example, the `pow` type-generic macro might be defined as:

```
#define pow(a, b) __builtin_tgmath (powf, pow, powl, \
                                    cpowf, cpow, cpowl, a, b)
```

The arguments to `__builtin_tgmath` are at least two pointers to functions, followed by the arguments to the type-generic macro (which will be passed as arguments to the selected function). All the pointers to functions must be pointers to prototyped functions, none of which may have variable arguments, and all of which must have the same number of parameters; the number of parameters of the first function determines how many arguments to `__builtin_tgmath` are interpreted as function pointers, and how many as the arguments to the called function.

The types of the specified functions must all be different, but related to each other in the same way as a set of functions that may be selected between by a macro in `<tgmath.h>`. This means that the functions are parameterized by a floating-point type t, different for each such function. The function return types may all be the same type, or they may be t for each function, or they may be the real type corresponding to t for each function (if some of the types t are complex). Likewise, for each parameter position, the type of the parameter in that position may always be the same type, or may be t for each function (this case must apply for at least one parameter position), or may be the real type corresponding to t for each function.

The standard rules for `<tgmath.h>` macros are used to find a common type u from the types of the arguments for parameters whose types vary between the functions; complex integer types (a GNU extension) are treated like `_Complex double` for this purpose (or `_Complex _Float64` if all the function return types are the same `_Floatn` or `_Floatnx` type). If the function return types vary, or are all the same integer type, the function called is the one for which t is u, and it is an error if there is no such function. If the function return types are all the same floating-point type, the type-generic macro is taken to be one of those from TS 18661 that rounds the result to a narrower type; if there is a function for which t is u, it is called, and otherwise the first function, if any, for which t has at least the range and precision of u is called, and it is an error if there is no such function.

type `__builtin_complex` (*real*, *imag*) [Built-in Function]

The built-in function `__builtin_complex` is provided for use in implementing the ISO C11 macros `CMPLXF`, `CMPLX` and `CMPLXL`. *real* and *imag* must have the same type, a real binary floating-point type, and the result has the corresponding complex type with real and imaginary parts *real* and *imag*. Unlike '*real* + I * *imag*', this works even when infinities, NaNs and negative zeros are involved.

`int __builtin_constant_p` (*exp*) [Built-in Function]

You can use the built-in function `__builtin_constant_p` to determine if a value is known to be constant at compile time and hence that GCC can perform constant-folding on expressions involving that value. The argument of the function is the value to test. The function returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant. A return of 0 does not indicate that the value is *not* a constant, but merely that GCC cannot prove it is a constant with the specified value of the '`-O`' option.

You typically use this function in an embedded application where memory is a critical resource. If you have some complex calculation, you may want it to be folded if it involves constants, but need to call a function if it does not. For example:

```
#define Scale_Value(X)      \
  (__builtin_constant_p (X) \
  ? ((X) * SCALE + OFFSET) : Scale (X))
```

You may use this built-in function in either a macro or an inline function. However, if you use it in an inlined function and pass an argument of the function as the argument to the built-in, GCC never returns 1 when you call the inline function with a string constant or compound literal (see Section 6.28 [Compound Literals], page 473) and does not return 1 when you pass a constant numeric value to the inline function unless you specify the '`-O`' option.

You may also use `__builtin_constant_p` in initializers for static data. For instance, you can write

```
static const int table[] = {
   __builtin_constant_p (EXPRESSION) ? (EXPRESSION) : -1,
   /* ... */
};
```

This is an acceptable initializer even if *EXPRESSION* is not a constant expression, including the case where `__builtin_constant_p` returns 1 because *EXPRESSION* can be folded to a constant but *EXPRESSION* contains operands that are not otherwise permitted in a static initializer (for example, `0 && foo ()`). GCC must be more conservative about evaluating the built-in in this case, because it has no opportunity to perform optimization.

`bool __builtin_is_constant_evaluated` (*void*) [Built-in Function]

The `__builtin_is_constant_evaluated` function is available only in C++. The built-in is intended to be used by implementations of the `std::is_constant_evaluated` C++ function. Programs should make use of the latter function rather than invoking the built-in directly.

The main use case of the built-in is to determine whether a `constexpr` function is being called in a `constexpr` context. A call to the function evaluates to a core constant expression with the value `true` if and only if it occurs within the evaluation of an expression or conversion that is manifestly constant-evaluated as defined in the C++ standard. Manifestly constant-evaluated contexts include constant-expressions, the conditions of `constexpr if` statements, constraint-expressions, and initializers of variables usable in constant expressions. For more details refer to the latest revision of the C++ standard.

`long __builtin_expect` (*long* `exp`, *long* `c`) [Built-in Function]

You may use `__builtin_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this ('`-fprofile-arcs`'), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of *exp*, which should be an integral expression. The semantics of the built-in are that it is expected that *exp* == *c*. For example:

```
if (__builtin_expect (x, 0))
  foo ();
```

indicates that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for *exp*, you should use constructions such as

```
if (__builtin_expect (ptr != NULL, 1))
  foo (*ptr);
```

when testing pointer or floating-point values.

For the purposes of branch prediction optimizations, the probability that a `__builtin_expect` expression is `true` is controlled by GCC's `builtin-expect-probability` parameter, which defaults to 90%. You can also use `__builtin_expect_with_probability` to explicitly assign a probability value to individual expressions.

`long __builtin_expect_with_probability` [Built-in Function]
(long *exp*, long *c*, double *probability*)

This function has the same semantics as `__builtin_expect`, but the caller provides the expected probability that *exp* == *c*. The last argument, *probability*, is a floating-point value in the range 0.0 to 1.0, inclusive. The *probability* argument must be constant floating-point expression.

`void __builtin_trap` (*void*) [Built-in Function]

This function causes the program to exit abnormally. GCC implements this function by using a target-dependent mechanism (such as intentionally executing an illegal instruction) or by calling `abort`. The mechanism used may vary from release to release so you should not rely on any particular implementation.

`void __builtin_unreachable` (*void*) [Built-in Function]

If control flow reaches the point of the `__builtin_unreachable`, the program is undefined. It is useful in situations where the compiler cannot deduce the unreachability of the code.

One such case is immediately following an `asm` statement that either never terminates, or one that transfers control elsewhere and never returns. In this example, without the `__builtin_unreachable`, GCC issues a warning that control reaches the end of a non-void function. It also generates code to return after the `asm`.

```
int f (int c, int v)
{
  if (c)
    {
      return v;
    }
  else
    {
      asm("jmp error_handler");
      __builtin_unreachable ();
    }
}
```

Because the `asm` statement unconditionally transfers control out of the function, control never reaches the end of the function body. The `__builtin_unreachable` is in fact unreachable and communicates this fact to the compiler.

Another use for `__builtin_unreachable` is following a call a function that never returns but that is not declared `__attribute__((noreturn))`, as in this example:

```
void function_that_never_returns (void);

int g (int c)
{
  if (c)
    {
      return 1;
    }
  else
    {
      function_that_never_returns ();
      __builtin_unreachable ();
    }
}
```

`void * __builtin_assume_aligned` (*const void *exp, size_t align, ...*) [Built-in Function]

This function returns its first argument, and allows the compiler to assume that the returned pointer is at least *align* bytes aligned. This built-in can have either two or three arguments, if it has three, the third argument should have integer type, and if it is nonzero means misalignment offset. For example:

```
void *x = __builtin_assume_aligned (arg, 16);
```

means that the compiler can assume `x`, set to `arg`, is at least 16-byte aligned, while:

```
void *x = __builtin_assume_aligned (arg, 32, 8);
```

means that the compiler can assume for `x`, set to `arg`, that `(char *) x - 8` is 32-byte aligned.

`int __builtin_LINE ()` [Built-in Function]

This function is the equivalent of the preprocessor `__LINE__` macro and returns a constant integer expression that evaluates to the line number of the invocation of the built-in. When used as a C++ default argument for a function *F*, it returns the line number of the call to *F*.

`const char * __builtin_FUNCTION ()` [Built-in Function]

This function is the equivalent of the `__FUNCTION__` symbol and returns an address constant pointing to the name of the function from which the built-in was invoked, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the name of *F*'s caller or the empty string if the call was not made at function scope.

`const char * __builtin_FILE ()` [Built-in Function]

This function is the equivalent of the preprocessor `__FILE__` macro and returns an address constant pointing to the file name containing the invocation of the built-in, or the empty string if the invocation is not at function scope. When used as a C++ default argument for a function *F*, it returns the file name of the call to *F* or the empty string if the call was not made at function scope.

For example, in the following, each call to function `foo` will print a line similar to `"file.c:123: foo: message"` with the name of the file and the line number of the `printf` call, the name of the function `foo`, followed by the word `message`.

```
const char*
function (const char *func = __builtin_FUNCTION ())
{
  return func;
}

void foo (void)
{
  printf ("%s:%i: %s: message\n", file (), line (), function ());
}
```

`void __builtin___clear_cache` (*void *begin, void *end*) [Built-in Function]

This function is used to flush the processor's instruction cache for the region of memory between *begin* inclusive and *end* exclusive. Some targets require that the instruction cache be flushed, after modifying memory containing code, in order to obtain deterministic behavior.

If the target does not require instruction cache flushes, `__builtin___clear_cache` has no effect. Otherwise either instructions are emitted in-line to clear the instruction cache or a call to the `__clear_cache` function in libgcc is made.

`void __builtin_prefetch` (*const void *addr, ...*) [Built-in Function]

This function is used to minimize cache-miss latency by moving data into a cache before it is accessed. You can insert calls to `__builtin_prefetch` into code for which you know addresses of data in memory that is likely to be accessed soon. If the target supports them, data prefetch instructions are generated. If the prefetch is done early enough before the access then the data will be in the cache by the time it is accessed.

The value of *addr* is the address of the memory to prefetch. There are two optional arguments, *rw* and *locality*. The value of *rw* is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero, the default, means that the prefetch is preparing for a read. The value *locality* must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. The default is three.

```
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    __builtin_prefetch (&a[i+j], 1, 1);
    __builtin_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if *addr* is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` does not fault if `p->next` is not a valid address, but evaluation faults if `p` is not a valid address.

If the target does not support data prefetch, the address expression is evaluated if it includes side effects but no other code is generated and GCC does not issue a warning.

`__builtin_object_size (const void * ptr, int type)` [Built-in Functionsize_t]
Returns the size of an object pointed to by *ptr*. See Section 6.58 [Object Size Checking], page 639, for a detailed description of the function.

`double __builtin_huge_val (void)` [Built-in Function]
Returns a positive infinity, if supported by the floating-point format, else `DBL_MAX`. This function is suitable for implementing the ISO C macro `HUGE_VAL`.

`float __builtin_huge_valf (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except the return type is `float`.

`long double __builtin_huge_vall (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except the return type is `long double`.

`_Floatn __builtin_huge_valfn (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except the return type is `_Floatn`.

`_Floatnx __builtin_huge_valfnx (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except the return type is `_Floatnx`.

`int __builtin_fpclassify (int, int, int, int, int, ...)` [Built-in Function]
This built-in implements the C99 fpclassify functionality. The first five int arguments should be the target library's notion of the possible FP classes and are used for return values. They must be constant values and they must appear in this order: `FP_NAN`, `FP_INFINITE`, `FP_NORMAL`, `FP_SUBNORMAL` and `FP_ZERO`. The ellipsis is for exactly one floating-point value to classify. GCC treats the last argument as type-generic, which means it does not do default promotion from float to double.

`double __builtin_inf (void)` [Built-in Function]
Similar to `__builtin_huge_val`, except a warning is generated if the target floating-point format does not support infinities.

`_Decimal32 __builtin_infd32 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal32`.

`_Decimal64 __builtin_infd64 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal64`.

`_Decimal128 __builtin_infd128 (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Decimal128`.

`float __builtin_inff (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `float`. This function is suitable for implementing the ISO C99 macro `INFINITY`.

`long double __builtin_infl (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `long double`.

`_Floatn __builtin_inffn (void)` [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Floatn`.

`_Floatn __builtin_inffnx` (*void*) [Built-in Function]
Similar to `__builtin_inf`, except the return type is `_Floatnx`.

`int __builtin_isinf_sign` (...) [Built-in Function]
Similar to `isinf`, except the return value is -1 for an argument of `-Inf` and 1 for an argument of `+Inf`. Note while the parameter list is an ellipsis, this function only accepts exactly one floating-point argument. GCC treats this parameter as type-generic, which means it does not do default promotion from float to double.

`double __builtin_nan` (*const char *str*) [Built-in Function]
This is an implementation of the ISO C99 function `nan`.

Since ISO C99 defines this function in terms of `strtod`, which we do not implement, a description of the parsing is in order. The string is parsed as by `strtol`; that is, the base is recognized by leading '`0`' or '`0x`' prefixes. The number parsed is placed in the significand such that the least significant bit of the number is at the least significant bit of the significand. The number is truncated to fit the significand field provided. The significand is forced to be a quiet NaN.

This function, if given a string literal all of which would have been consumed by `strtol`, is evaluated early enough that it is considered a compile-time constant.

`_Decimal32 __builtin_nand32` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal32`.

`_Decimal64 __builtin_nand64` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal64`.

`_Decimal128 __builtin_nand128` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Decimal128`.

`float __builtin_nanf` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `float`.

`long double __builtin_nanl` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `long double`.

`_Floatn __builtin_nanfn` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Floatn`.

`_Floatnx __builtin_nanfnx` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the return type is `_Floatnx`.

`double __builtin_nans` (*const char *str*) [Built-in Function]
Similar to `__builtin_nan`, except the significand is forced to be a signaling NaN. The `nans` function is proposed by WG14 N965.

`float __builtin_nansf` (*const char *str*) [Built-in Function]
Similar to `__builtin_nans`, except the return type is `float`.

`long double __builtin_nansl` (*const char *str*) [Built-in Function]
Similar to `__builtin_nans`, except the return type is `long double`.

`_Floatn __builtin_nansfn` (*const char *str*) [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Floatn`.

`_Floatnx __builtin_nansfnx` (*const char *str*) [Built-in Function]
Similar to `__builtin_nans`, except the return type is `_Floatnx`.

`int __builtin_ffs` (*int x*) [Built-in Function]
Returns one plus the index of the least significant 1-bit of *x*, or if *x* is zero, returns zero.

`int __builtin_clz` (*unsigned int x*) [Built-in Function]
Returns the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, the result is undefined.

`int __builtin_ctz` (*unsigned int x*) [Built-in Function]
Returns the number of trailing 0-bits in *x*, starting at the least significant bit position. If *x* is 0, the result is undefined.

`int __builtin_clrsb` (*int x*) [Built-in Function]
Returns the number of leading redundant sign bits in *x*, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.

`int __builtin_popcount` (*unsigned int x*) [Built-in Function]
Returns the number of 1-bits in *x*.

`int __builtin_parity` (*unsigned int x*) [Built-in Function]
Returns the parity of *x*, i.e. the number of 1-bits in *x* modulo 2.

`int __builtin_ffsl` (*long*) [Built-in Function]
Similar to `__builtin_ffs`, except the argument type is `long`.

`int __builtin_clzl` (*unsigned long*) [Built-in Function]
Similar to `__builtin_clz`, except the argument type is `unsigned long`.

`int __builtin_ctzl` (*unsigned long*) [Built-in Function]
Similar to `__builtin_ctz`, except the argument type is `unsigned long`.

`int __builtin_clrsbl` (*long*) [Built-in Function]
Similar to `__builtin_clrsb`, except the argument type is `long`.

`int __builtin_popcountl` (*unsigned long*) [Built-in Function]
Similar to `__builtin_popcount`, except the argument type is `unsigned long`.

`int __builtin_parityl` (*unsigned long*) [Built-in Function]
Similar to `__builtin_parity`, except the argument type is `unsigned long`.

`int __builtin_ffsll` (*long long*) [Built-in Function]
Similar to `__builtin_ffs`, except the argument type is `long long`.

`int __builtin_clzll` (*unsigned long long*) [Built-in Function]
Similar to `__builtin_clz`, except the argument type is `unsigned long long`.

`int __builtin_ctzll` (*unsigned long long*) [Built-in Function]
Similar to `__builtin_ctz`, except the argument type is `unsigned long long`.

`int __builtin_clrsbll` (*long long*) [Built-in Function]
Similar to `__builtin_clrsb`, except the argument type is `long long`.

`int __builtin_popcountll` (*unsigned long long*) [Built-in Function]
Similar to `__builtin_popcount`, except the argument type is `unsigned long long`.

`int __builtin_parityll` (*unsigned long long*) [Built-in Function]
Similar to `__builtin_parity`, except the argument type is `unsigned long long`.

`double __builtin_powi` (*double, int*) [Built-in Function]
Returns the first argument raised to the power of the second. Unlike the `pow` function no guarantees about precision and rounding are made.

`float __builtin_powif` (*float, int*) [Built-in Function]
Similar to `__builtin_powi`, except the argument and return types are `float`.

`long double __builtin_powil` (*long double, int*) [Built-in Function]
Similar to `__builtin_powi`, except the argument and return types are `long double`.

`uint16_t __builtin_bswap16` (*uint16_t x*) [Built-in Function]
Returns *x* with the order of the bytes reversed; for example, `0xaabb` becomes `0xbbaa`. Byte here always means exactly 8 bits.

`uint32_t __builtin_bswap32` (*uint32_t x*) [Built-in Function]
Similar to `__builtin_bswap16`, except the argument and return types are 32 bit.

`uint64_t __builtin_bswap64` (*uint64_t x*) [Built-in Function]
Similar to `__builtin_bswap32`, except the argument and return types are 64 bit.

`Pmode __builtin_extend_pointer` (*void * x*) [Built-in Function]
On targets where the user visible pointer size is smaller than the size of an actual hardware address this function returns the extended user pointer. Targets where this is true included ILP32 mode on x86_64 or Aarch64. This function is mainly useful when writing inline assembly code.

`int __builtin_goacc_parlevel_id` (*int x*) [Built-in Function]
Returns the openacc gang, worker or vector id depending on whether *x* is 0, 1 or 2.

`int __builtin_goacc_parlevel_size` (*int x*) [Built-in Function]
Returns the openacc gang, worker or vector size depending on whether *x* is 0, 1 or 2.

6.60 Built-in Functions Specific to Particular Target Machines

On some target machines, GCC supports many built-in functions specific to those machines. Generally these generate calls to specific machine instructions, but allow the compiler to schedule those calls.

6.60.1 AArch64 Built-in Functions

These built-in functions are available for the AArch64 family of processors.

```
unsigned int __builtin_aarch64_get_fpcr ()
void __builtin_aarch64_set_fpcr (unsigned int)
unsigned int __builtin_aarch64_get_fpsr ()
void __builtin_aarch64_set_fpsr (unsigned int)
```

6.60.2 Alpha Built-in Functions

These built-in functions are available for the Alpha family of processors, depending on the command-line switches used.

The following built-in functions are always available. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_implver (void)
long __builtin_alpha_rpcc (void)
long __builtin_alpha_amask (long)
long __builtin_alpha_cmpbge (long, long)
long __builtin_alpha_extbl (long, long)
long __builtin_alpha_extwl (long, long)
long __builtin_alpha_extll (long, long)
long __builtin_alpha_extql (long, long)
long __builtin_alpha_extwh (long, long)
long __builtin_alpha_extlh (long, long)
long __builtin_alpha_extqh (long, long)
long __builtin_alpha_insbl (long, long)
long __builtin_alpha_inswl (long, long)
long __builtin_alpha_insll (long, long)
long __builtin_alpha_insql (long, long)
long __builtin_alpha_inswh (long, long)
long __builtin_alpha_inslh (long, long)
long __builtin_alpha_insqh (long, long)
long __builtin_alpha_mskbl (long, long)
long __builtin_alpha_mskwl (long, long)
long __builtin_alpha_mskll (long, long)
long __builtin_alpha_mskql (long, long)
long __builtin_alpha_mskwh (long, long)
long __builtin_alpha_msklh (long, long)
long __builtin_alpha_mskqh (long, long)
long __builtin_alpha_umulh (long, long)
long __builtin_alpha_zap (long, long)
long __builtin_alpha_zapnot (long, long)
```

The following built-in functions are always with '`-mmax`' or '`-mcpu=`*cpu*' where *cpu* is `pca56` or later. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_pklb (long)
long __builtin_alpha_pkwb (long)
long __builtin_alpha_unpkbl (long)
long __builtin_alpha_unpkbw (long)
long __builtin_alpha_minub8 (long, long)
long __builtin_alpha_minsb8 (long, long)
long __builtin_alpha_minuw4 (long, long)
long __builtin_alpha_minsw4 (long, long)
long __builtin_alpha_maxub8 (long, long)
long __builtin_alpha_maxsb8 (long, long)
long __builtin_alpha_maxuw4 (long, long)
long __builtin_alpha_maxsw4 (long, long)
```

```
long __builtin_alpha_perr (long, long)
```

The following built-in functions are always with '-mcix' or '-mcpu=*cpu*' where *cpu* is ev67 or later. They all generate the machine instruction that is part of the name.

```
long __builtin_alpha_cttz (long)
long __builtin_alpha_ctlz (long)
long __builtin_alpha_ctpop (long)
```

The following built-in functions are available on systems that use the OSF/1 PALcode. Normally they invoke the rduniq and wruniq PAL calls, but when invoked with '-mtls-kernel', they invoke rdval and wrval.

```
void *__builtin_thread_pointer (void)
void __builtin_set_thread_pointer (void *)
```

6.60.3 Altera Nios II Built-in Functions

These built-in functions are available for the Altera Nios II family of processors.

The following built-in functions are always available. They all generate the machine instruction that is part of the name.

```
int __builtin_ldbio (volatile const void *)
int __builtin_ldbuio (volatile const void *)
int __builtin_ldhio (volatile const void *)
int __builtin_ldhuio (volatile const void *)
int __builtin_ldwio (volatile const void *)
void __builtin_stbio (volatile void *, int)
void __builtin_sthio (volatile void *, int)
void __builtin_stwio (volatile void *, int)
void __builtin_sync (void)
int __builtin_rdctl (int)
int __builtin_rdprs (int, int)
void __builtin_wrctl (int, int)
void __builtin_flushd (volatile void *)
void __builtin_flushda (volatile void *)
int __builtin_wrpie (int);
void __builtin_eni (int);
int __builtin_ldex (volatile const void *)
int __builtin_stex (volatile void *, int)
int __builtin_ldsex (volatile const void *)
int __builtin_stsex (volatile void *, int)
```

The following built-in functions are always available. They all generate a Nios II Custom Instruction. The name of the function represents the types that the function takes and returns. The letter before the n is the return type or void if absent. The n represents the first parameter to all the custom instructions, the custom instruction number. The two letters after the n represent the up to two parameters to the function.

The letters represent the following data types:

<no letter>
: void for return type and no parameter for parameter types.

i
: int for return type and parameter type

`f` `float` for return type and parameter type

`p` `void *` for return type and parameter type

And the function names are:

```
void __builtin_custom_n (void)
void __builtin_custom_ni (int)
void __builtin_custom_nf (float)
void __builtin_custom_np (void *)
void __builtin_custom_nii (int, int)
void __builtin_custom_nif (int, float)
void __builtin_custom_nip (int, void *)
void __builtin_custom_nfi (float, int)
void __builtin_custom_nff (float, float)
void __builtin_custom_nfp (float, void *)
void __builtin_custom_npi (void *, int)
void __builtin_custom_npf (void *, float)
void __builtin_custom_npp (void *, void *)
int __builtin_custom_in (void)
int __builtin_custom_ini (int)
int __builtin_custom_inf (float)
int __builtin_custom_inp (void *)
int __builtin_custom_inii (int, int)
int __builtin_custom_inif (int, float)
int __builtin_custom_inip (int, void *)
int __builtin_custom_infi (float, int)
int __builtin_custom_inff (float, float)
int __builtin_custom_infp (float, void *)
int __builtin_custom_inpi (void *, int)
int __builtin_custom_inpf (void *, float)
int __builtin_custom_inpp (void *, void *)
float __builtin_custom_fn (void)
float __builtin_custom_fni (int)
float __builtin_custom_fnf (float)
float __builtin_custom_fnp (void *)
float __builtin_custom_fnii (int, int)
float __builtin_custom_fnif (int, float)
float __builtin_custom_fnip (int, void *)
float __builtin_custom_fnfi (float, int)
float __builtin_custom_fnff (float, float)
float __builtin_custom_fnfp (float, void *)
float __builtin_custom_fnpi (void *, int)
float __builtin_custom_fnpf (void *, float)
float __builtin_custom_fnpp (void *, void *)
void * __builtin_custom_pn (void)
void * __builtin_custom_pni (int)
void * __builtin_custom_pnf (float)
void * __builtin_custom_pnp (void *)
```

```
void * __builtin_custom_pnii (int, int)
void * __builtin_custom_pnif (int, float)
void * __builtin_custom_pnip (int, void *)
void * __builtin_custom_pnfi (float, int)
void * __builtin_custom_pnff (float, float)
void * __builtin_custom_pnfp (float, void *)
void * __builtin_custom_pnpi (void *, int)
void * __builtin_custom_pnpf (void *, float)
void * __builtin_custom_pnpp (void *, void *)
```

6.60.4 ARC Built-in Functions

The following built-in functions are provided for ARC targets. The built-ins generate the corresponding assembly instructions. In the examples given below, the generated code often requires an operand or result to be in a register. Where necessary further code will be generated to ensure this is true, but for brevity this is not described in each case.

Note: Using a built-in to generate an instruction not supported by a target may cause problems. At present the compiler is not guaranteed to detect such misuse, and as a result an internal compiler error may be generated.

`int __builtin_arc_aligned` (*void *val*, *int alignval*) [Built-in Function]

Return 1 if *val* is known to have the byte alignment given by *alignval*, otherwise return 0. Note that this is different from

```
__alignof__(*(char *)val) >= alignval
```

because __alignof__ sees only the type of the dereference, whereas __builtin_arc_align uses alignment information from the pointer as well as from the pointed-to type. The information available will depend on optimization level.

`void __builtin_arc_brk` (*void*) [Built-in Function]

Generates

```
brk
```

`unsigned int __builtin_arc_core_read` (*unsigned int regno*) [Built-in Function]

The operand is the number of a register to be read. Generates:

```
mov  dest, rregno
```

where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_core_write` (*unsigned int regno*, *unsigned int val*) [Built-in Function]

The first operand is the number of a register to be written, the second operand is a compile time constant to write into that register. Generates:

```
mov  rregno, val
```

`int __builtin_arc_divaw` (*int a*, *int b*) [Built-in Function]

Only available if either '`-mcpu=ARC700`' or '`-meA`' is set. Generates:

```
divaw  dest, a, b
```

where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_flag` (*unsigned int* `a`) [Built-in Function]

Generates

```
flag  a
```

`unsigned int __builtin_arc_lr` (*unsigned int* `auxr`) [Built-in Function]

The operand, *auxv*, is the address of an auxiliary register and must be a compile time constant. Generates:

```
lr  dest, [auxr]
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_mul64` (*int* `a`, *int* `b`) [Built-in Function]

Only available with '`-mmul64`'. Generates:

```
mul64  a, b
```

`void __builtin_arc_mulu64` (*unsigned int* `a`, *unsigned int* `b`) [Built-in Function]

Only available with '`-mmul64`'. Generates:

```
mulu64  a, b
```

`void __builtin_arc_nop` (*void*) [Built-in Function]

Generates:

```
nop
```

`int __builtin_arc_norm` (*int* `src`) [Built-in Function]

Only valid if the '`norm`' instruction is available through the '`-mnorm`' option or by default with '`-mcpu=ARC700`'. Generates:

```
norm  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`short int __builtin_arc_normw` (*short int* `src`) [Built-in Function]

Only valid if the '`normw`' instruction is available through the '`-mnorm`' option or by default with '`-mcpu=ARC700`'. Generates:

```
normw  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_rtie` (*void*) [Built-in Function]

Generates:

```
rtie
```

`void __builtin_arc_sleep` (*int* `a` [Built-in Function]

Generates:

```
sleep  a
```

`void __builtin_arc_sr` (*unsigned int* `auxr`, *unsigned int* `val`) [Built-in Function]

The first argument, *auxv*, is the address of an auxiliary register, the second argument, *val*, is a compile time constant to be written to the register. Generates:

```
sr  auxr, [val]
```

`int __builtin_arc_swap (int src)` [Built-in Function]

Only valid with '`-mswap`'. Generates:

```
swap  dest, src
```

Where the value in *dest* will be the result returned from the built-in.

`void __builtin_arc_swi (void)` [Built-in Function]

Generates:

```
swi
```

`void __builtin_arc_sync (void)` [Built-in Function]

Only available with '`-mcpu=ARC700`'. Generates:

```
sync
```

`void __builtin_arc_trap_s (unsigned int c)` [Built-in Function]

Only available with '`-mcpu=ARC700`'. Generates:

```
trap_s  c
```

`void __builtin_arc_unimp_s (void)` [Built-in Function]

Only available with '`-mcpu=ARC700`'. Generates:

```
unimp_s
```

The instructions generated by the following builtins are not considered as candidates for scheduling. They are not moved around by the compiler during scheduling, and thus can be expected to appear where they are put in the C code:

```
__builtin_arc_brk()
__builtin_arc_core_read()
__builtin_arc_core_write()
__builtin_arc_flag()
__builtin_arc_lr()
__builtin_arc_sleep()
__builtin_arc_sr()
__builtin_arc_swi()
```

6.60.5 ARC SIMD Built-in Functions

SIMD builtins provided by the compiler can be used to generate the vector instructions. This section describes the available builtins and their usage in programs. With the '`-msimd`' option, the compiler provides 128-bit vector types, which can be specified using the `vector_size` attribute. The header file '`arc-simd.h`' can be included to use the following predefined types:

```
typedef int __v4si   __attribute__((vector_size(16)));
typedef short __v8hi __attribute__((vector_size(16)));
```

These types can be used to define 128-bit variables. The built-in functions listed in the following section can be used on these variables to generate the vector operations.

For all builtins, `__builtin_arc_someinsn`, the header file '`arc-simd.h`' also provides equivalent macros called `_someinsn` that can be used for programming ease and improved readability. The following macros for DMA control are also provided:

```
#define _setup_dma_in_channel_reg _vdiwr
#define _setup_dma_out_channel_reg _vdowr
```

The following is a complete list of all the SIMD built-ins provided for ARC, grouped by calling signature.

The following take two `__v8hi` arguments and return a `__v8hi` result:

```
__v8hi __builtin_arc_vaddaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vaddw (__v8hi, __v8hi)
__v8hi __builtin_arc_vand (__v8hi, __v8hi)
__v8hi __builtin_arc_vandaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vavb (__v8hi, __v8hi)
__v8hi __builtin_arc_vavrb (__v8hi, __v8hi)
__v8hi __builtin_arc_vbic (__v8hi, __v8hi)
__v8hi __builtin_arc_vbicaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vdifaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vdifw (__v8hi, __v8hi)
__v8hi __builtin_arc_veqw (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264f (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264ft (__v8hi, __v8hi)
__v8hi __builtin_arc_vh264fw (__v8hi, __v8hi)
__v8hi __builtin_arc_vlew (__v8hi, __v8hi)
__v8hi __builtin_arc_vltw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmaxaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmaxw (__v8hi, __v8hi)
__v8hi __builtin_arc_vminaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vminw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr1aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr1w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr2aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr2w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr3aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr3w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr4aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr4w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr5aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr5w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr6aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr6w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr7aw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmr7w (__v8hi, __v8hi)
__v8hi __builtin_arc_vmrb (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulfaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulfw (__v8hi, __v8hi)
__v8hi __builtin_arc_vmulw (__v8hi, __v8hi)
__v8hi __builtin_arc_vnew (__v8hi, __v8hi)
__v8hi __builtin_arc_vor (__v8hi, __v8hi)
```

```
__v8hi __builtin_arc_vsubaw (__v8hi, __v8hi)
__v8hi __builtin_arc_vsubw (__v8hi, __v8hi)
__v8hi __builtin_arc_vsummw (__v8hi, __v8hi)
__v8hi __builtin_arc_vvc1f (__v8hi, __v8hi)
__v8hi __builtin_arc_vvc1ft (__v8hi, __v8hi)
__v8hi __builtin_arc_vxor (__v8hi, __v8hi)
__v8hi __builtin_arc_vxoraw (__v8hi, __v8hi)
```

The following take one `__v8hi` and one `int` argument and return a `__v8hi` result:

```
__v8hi __builtin_arc_vbaddw (__v8hi, int)
__v8hi __builtin_arc_vbmaxw (__v8hi, int)
__v8hi __builtin_arc_vbminw (__v8hi, int)
__v8hi __builtin_arc_vbmulaw (__v8hi, int)
__v8hi __builtin_arc_vbmulfw (__v8hi, int)
__v8hi __builtin_arc_vbmulw (__v8hi, int)
__v8hi __builtin_arc_vbrsubw (__v8hi, int)
__v8hi __builtin_arc_vbsubw (__v8hi, int)
```

The following take one `__v8hi` argument and one `int` argument which must be a 3-bit compile time constant indicating a register number I0-I7. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vasrw (__v8hi, const int)
__v8hi __builtin_arc_vsr8 (__v8hi, const int)
__v8hi __builtin_arc_vsr8aw (__v8hi, const int)
```

The following take one `__v8hi` argument and one `int` argument which must be a 6-bit compile time constant. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vasrpwbi (__v8hi, const int)
__v8hi __builtin_arc_vasrrpwbi (__v8hi, const int)
__v8hi __builtin_arc_vasrrwi (__v8hi, const int)
__v8hi __builtin_arc_vasrsrwi (__v8hi, const int)
__v8hi __builtin_arc_vasrwi (__v8hi, const int)
__v8hi __builtin_arc_vsr8awi (__v8hi, const int)
__v8hi __builtin_arc_vsr8i (__v8hi, const int)
```

The following take one `__v8hi` argument and one `int` argument which must be a 8-bit compile time constant. They return a `__v8hi` result.

```
__v8hi __builtin_arc_vd6tapf (__v8hi, const int)
__v8hi __builtin_arc_vmvaw (__v8hi, const int)
__v8hi __builtin_arc_vmvw (__v8hi, const int)
__v8hi __builtin_arc_vmvzw (__v8hi, const int)
```

The following take two `int` arguments, the second of which which must be a 8-bit compile time constant. They return a `__v8hi` result:

```
__v8hi __builtin_arc_vmovaw (int, const int)
__v8hi __builtin_arc_vmovw (int, const int)
__v8hi __builtin_arc_vmovzw (int, const int)
```

The following take a single `__v8hi` argument and return a `__v8hi` result:

```
__v8hi __builtin_arc_vabsaw (__v8hi)
__v8hi __builtin_arc_vabsw (__v8hi)
```

```
__v8hi __builtin_arc_vaddsuw (__v8hi)
__v8hi __builtin_arc_vexch1 (__v8hi)
__v8hi __builtin_arc_vexch2 (__v8hi)
__v8hi __builtin_arc_vexch4 (__v8hi)
__v8hi __builtin_arc_vsignw (__v8hi)
__v8hi __builtin_arc_vupbaw (__v8hi)
__v8hi __builtin_arc_vupbw (__v8hi)
__v8hi __builtin_arc_vupsbaw (__v8hi)
__v8hi __builtin_arc_vupsbw (__v8hi)
```

The following take two `int` arguments and return no result:

```
void __builtin_arc_vdirun (int, int)
void __builtin_arc_vdorun (int, int)
```

The following take two `int` arguments and return no result. The first argument must a 3-bit compile time constant indicating one of the DR0-DR7 DMA setup channels:

```
void __builtin_arc_vdiwr (const int, int)
void __builtin_arc_vdowr (const int, int)
```

The following take an `int` argument and return no result:

```
void __builtin_arc_vendrec (int)
void __builtin_arc_vrec (int)
void __builtin_arc_vrecrun (int)
void __builtin_arc_vrun (int)
```

The following take a `__v8hi` argument and two `int` arguments and return a `__v8hi` result. The second argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the third argument must be an 8-bit compile time constant.

Note: Although the equivalent hardware instructions do not take an SIMD register as an operand, these builtins overwrite the relevant bits of the `__v8hi` register provided as the first argument with the value loaded from the `[Ib, u8]` location in the SDM.

```
__v8hi __builtin_arc_vld32 (__v8hi, const int, const int)
__v8hi __builtin_arc_vld32wh (__v8hi, const int, const int)
__v8hi __builtin_arc_vld32wl (__v8hi, const int, const int)
__v8hi __builtin_arc_vld64 (__v8hi, const int, const int)
```

The following take two `int` arguments and return a `__v8hi` result. The first argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the second argument must be an 8-bit compile time constant.

```
__v8hi __builtin_arc_vld128 (const int, const int)
__v8hi __builtin_arc_vld64w (const int, const int)
```

The following take a `__v8hi` argument and two `int` arguments and return no result. The second argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the third argument must be an 8-bit compile time constant.

```
void __builtin_arc_vst128 (__v8hi, const int, const int)
void __builtin_arc_vst64 (__v8hi, const int, const int)
```

The following take a `__v8hi` argument and three `int` arguments and return no result. The second argument must be a 3-bit compile-time constant, identifying the 16-bit sub-

register to be stored, the third argument must be a 3-bit compile time constants, indicating one the registers I0-I7, and the fourth argument must be an 8-bit compile time constant.

```
void __builtin_arc_vst16_n (__v8hi, const int, const int, const int)
void __builtin_arc_vst32_n (__v8hi, const int, const int, const int)
```

6.60.6 ARM iWMMXt Built-in Functions

These built-in functions are available for the ARM family of processors when the '-mcpu=iwmmxt' switch is used:

```
typedef int v2si __attribute__ ((vector_size (8)));
typedef short v4hi __attribute__ ((vector_size (8)));
typedef char v8qi __attribute__ ((vector_size (8)));

int __builtin_arm_getwcgr0 (void)
void __builtin_arm_setwcgr0 (int)
int __builtin_arm_getwcgr1 (void)
void __builtin_arm_setwcgr1 (int)
int __builtin_arm_getwcgr2 (void)
void __builtin_arm_setwcgr2 (int)
int __builtin_arm_getwcgr3 (void)
void __builtin_arm_setwcgr3 (int)
int __builtin_arm_textrmsb (v8qi, int)
int __builtin_arm_textrmsh (v4hi, int)
int __builtin_arm_textrmsw (v2si, int)
int __builtin_arm_textrmub (v8qi, int)
int __builtin_arm_textrmuh (v4hi, int)
int __builtin_arm_textrmuw (v2si, int)
v8qi __builtin_arm_tinsrb (v8qi, int, int)
v4hi __builtin_arm_tinsrh (v4hi, int, int)
v2si __builtin_arm_tinsrw (v2si, int, int)
long long __builtin_arm_tmia (long long, int, int)
long long __builtin_arm_tmiabb (long long, int, int)
long long __builtin_arm_tmiabt (long long, int, int)
long long __builtin_arm_tmiaph (long long, int, int)
long long __builtin_arm_tmiatb (long long, int, int)
long long __builtin_arm_tmiatt (long long, int, int)
int __builtin_arm_tmovmskb (v8qi)
int __builtin_arm_tmovmskh (v4hi)
int __builtin_arm_tmovmskw (v2si)
long long __builtin_arm_waccb (v8qi)
long long __builtin_arm_wacch (v4hi)
long long __builtin_arm_waccw (v2si)
v8qi __builtin_arm_waddb (v8qi, v8qi)
v8qi __builtin_arm_waddbss (v8qi, v8qi)
v8qi __builtin_arm_waddbus (v8qi, v8qi)
v4hi __builtin_arm_waddh (v4hi, v4hi)
v4hi __builtin_arm_waddhss (v4hi, v4hi)
v4hi __builtin_arm_waddhus (v4hi, v4hi)
v2si __builtin_arm_waddw (v2si, v2si)
v2si __builtin_arm_waddwss (v2si, v2si)
v2si __builtin_arm_waddwus (v2si, v2si)
v8qi __builtin_arm_walign (v8qi, v8qi, int)
long long __builtin_arm_wand(long long, long long)
long long __builtin_arm_wandn (long long, long long)
v8qi __builtin_arm_wavg2b (v8qi, v8qi)
v8qi __builtin_arm_wavg2br (v8qi, v8qi)
v4hi __builtin_arm_wavg2h (v4hi, v4hi)
```

```
v4hi __builtin_arm_wavg2hr (v4hi, v4hi)
v8qi __builtin_arm_wcmpeqb (v8qi, v8qi)
v4hi __builtin_arm_wcmpeqh (v4hi, v4hi)
v2si __builtin_arm_wcmpeqw (v2si, v2si)
v8qi __builtin_arm_wcmpgtsb (v8qi, v8qi)
v4hi __builtin_arm_wcmpgtsh (v4hi, v4hi)
v2si __builtin_arm_wcmpgtsw (v2si, v2si)
v8qi __builtin_arm_wcmpgtub (v8qi, v8qi)
v4hi __builtin_arm_wcmpgtuh (v4hi, v4hi)
v2si __builtin_arm_wcmpgtuw (v2si, v2si)
long long __builtin_arm_wmacs (long long, v4hi, v4hi)
long long __builtin_arm_wmacsz (v4hi, v4hi)
long long __builtin_arm_wmacu (long long, v4hi, v4hi)
long long __builtin_arm_wmacuz (v4hi, v4hi)
v4hi __builtin_arm_wmadds (v4hi, v4hi)
v4hi __builtin_arm_wmaddu (v4hi, v4hi)
v8qi __builtin_arm_wmaxsb (v8qi, v8qi)
v4hi __builtin_arm_wmaxsh (v4hi, v4hi)
v2si __builtin_arm_wmaxsw (v2si, v2si)
v8qi __builtin_arm_wmaxub (v8qi, v8qi)
v4hi __builtin_arm_wmaxuh (v4hi, v4hi)
v2si __builtin_arm_wmaxuw (v2si, v2si)
v8qi __builtin_arm_wminsb (v8qi, v8qi)
v4hi __builtin_arm_wminsh (v4hi, v4hi)
v2si __builtin_arm_wminsw (v2si, v2si)
v8qi __builtin_arm_wminub (v8qi, v8qi)
v4hi __builtin_arm_wminuh (v4hi, v4hi)
v2si __builtin_arm_wminuw (v2si, v2si)
v4hi __builtin_arm_wmulsm (v4hi, v4hi)
v4hi __builtin_arm_wmulul (v4hi, v4hi)
v4hi __builtin_arm_wmulum (v4hi, v4hi)
long long __builtin_arm_wor (long long, long long)
v2si __builtin_arm_wpackdss (long long, long long)
v2si __builtin_arm_wpackdus (long long, long long)
v8qi __builtin_arm_wpackhss (v4hi, v4hi)
v8qi __builtin_arm_wpackhus (v4hi, v4hi)
v4hi __builtin_arm_wpackwss (v2si, v2si)
v4hi __builtin_arm_wpackwus (v2si, v2si)
long long __builtin_arm_wrord (long long, long long)
long long __builtin_arm_wrordi (long long, int)
v4hi __builtin_arm_wrorh (v4hi, long long)
v4hi __builtin_arm_wrorhi (v4hi, int)
v2si __builtin_arm_wrorw (v2si, long long)
v2si __builtin_arm_wrorwi (v2si, int)
v2si __builtin_arm_wsadb (v2si, v8qi, v8qi)
v2si __builtin_arm_wsadbz (v8qi, v8qi)
v2si __builtin_arm_wsadh (v2si, v4hi, v4hi)
v2si __builtin_arm_wsadhz (v4hi, v4hi)
v4hi __builtin_arm_wshufh (v4hi, int)
long long __builtin_arm_wslld (long long, long long)
long long __builtin_arm_wslldi (long long, int)
v4hi __builtin_arm_wsllh (v4hi, long long)
v4hi __builtin_arm_wsllhi (v4hi, int)
v2si __builtin_arm_wsllw (v2si, long long)
v2si __builtin_arm_wsllwi (v2si, int)
long long __builtin_arm_wsrad (long long, long long)
long long __builtin_arm_wsradi (long long, int)
v4hi __builtin_arm_wsrah (v4hi, long long)
```

```
v4hi __builtin_arm_wsrahi (v4hi, int)
v2si __builtin_arm_wsraw (v2si, long long)
v2si __builtin_arm_wsrawi (v2si, int)
long long __builtin_arm_wsrld (long long, long long)
long long __builtin_arm_wsrldi (long long, int)
v4hi __builtin_arm_wsrlh (v4hi, long long)
v4hi __builtin_arm_wsrlhi (v4hi, int)
v2si __builtin_arm_wsrlw (v2si, long long)
v2si __builtin_arm_wsrlwi (v2si, int)
v8qi __builtin_arm_wsubb (v8qi, v8qi)
v8qi __builtin_arm_wsubbss (v8qi, v8qi)
v8qi __builtin_arm_wsubbus (v8qi, v8qi)
v4hi __builtin_arm_wsubh (v4hi, v4hi)
v4hi __builtin_arm_wsubhss (v4hi, v4hi)
v4hi __builtin_arm_wsubhus (v4hi, v4hi)
v2si __builtin_arm_wsubw (v2si, v2si)
v2si __builtin_arm_wsubwss (v2si, v2si)
v2si __builtin_arm_wsubwus (v2si, v2si)
v4hi __builtin_arm_wunpckehsb (v8qi)
v2si __builtin_arm_wunpckehsh (v4hi)
long long __builtin_arm_wunpckehsw (v2si)
v4hi __builtin_arm_wunpckehub (v8qi)
v2si __builtin_arm_wunpckehuh (v4hi)
long long __builtin_arm_wunpckehuw (v2si)
v4hi __builtin_arm_wunpckelsb (v8qi)
v2si __builtin_arm_wunpckelsh (v4hi)
long long __builtin_arm_wunpckelsw (v2si)
v4hi __builtin_arm_wunpckelub (v8qi)
v2si __builtin_arm_wunpckeluh (v4hi)
long long __builtin_arm_wunpckeluw (v2si)
v8qi __builtin_arm_wunpckihb (v8qi, v8qi)
v4hi __builtin_arm_wunpckihh (v4hi, v4hi)
v2si __builtin_arm_wunpckihw (v2si, v2si)
v8qi __builtin_arm_wunpckilb (v8qi, v8qi)
v4hi __builtin_arm_wunpckilh (v4hi, v4hi)
v2si __builtin_arm_wunpckilw (v2si, v2si)
long long __builtin_arm_wxor (long long, long long)
long long __builtin_arm_wzero ()
```

6.60.7 ARM C Language Extensions (ACLE)

GCC implements extensions for C as described in the ARM C Language Extensions (ACLE) specification, which can be found at `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0053c/IHI0053C_acle_2_0.pdf`.

As a part of ACLE, GCC implements extensions for Advanced SIMD as described in the ARM C Language Extensions Specification. The complete list of Advanced SIMD intrinsics can be found at `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0073a/IHI0073A_arm_neon_intrinsics_ref.pdf`. The built-in intrinsics for the Advanced SIMD extension are available when NEON is enabled.

Currently, ARM and AArch64 back ends do not support ACLE 2.0 fully. Both back ends support CRC32 intrinsics and the ARM back end supports the Coprocessor intrinsics, all from '`arm_acle.h`'. The ARM back end's 16-bit floating-point Advanced SIMD intrinsics currently comply to ACLE v1.1. AArch64's back end does not have support for 16-bit floating point Advanced SIMD intrinsics yet.

See Section 3.18.5 [ARM Options], page 263 and Section 3.18.1 [AArch64 Options], page 245 for more information on the availability of extensions.

6.60.8 ARM Floating Point Status and Control Intrinsics

These built-in functions are available for the ARM family of processors with floating-point unit.

```
unsigned int __builtin_arm_get_fpscr ()
void __builtin_arm_set_fpscr (unsigned int)
```

6.60.9 ARM ARMv8-M Security Extensions

GCC implements the ARMv8-M Security Extensions as described in the ARMv8-M Security Extensions: Requirements on Development Tools Engineering Specification, which can be found at `http://infocenter.arm.com/help/topic/com.arm.doc.ecm0359818/ECM0359818_armv8m_security_extensions_reqs_on_dev_tools_1_0.pdf`.

As part of the Security Extensions GCC implements two new function attributes: `cmse_nonsecure_entry` and `cmse_nonsecure_call`.

As part of the Security Extensions GCC implements the intrinsics below. FPTR is used here to mean any function pointer type.

```
cmse_address_info_t cmse_TT (void *)
cmse_address_info_t cmse_TT_fptr (FPTR)
cmse_address_info_t cmse_TTT (void *)
cmse_address_info_t cmse_TTT_fptr (FPTR)
cmse_address_info_t cmse_TTA (void *)
cmse_address_info_t cmse_TTA_fptr (FPTR)
cmse_address_info_t cmse_TTAT (void *)
cmse_address_info_t cmse_TTAT_fptr (FPTR)
void * cmse_check_address_range (void *, size_t, int)
typeof(p) cmse_nsfptr_create (FPTR p)
intptr_t cmse_is_nsfptr (FPTR)
int cmse_nonsecure_caller (void)
```

6.60.10 AVR Built-in Functions

For each built-in function for AVR, there is an equally named, uppercase built-in macro defined. That way users can easily query if or if not a specific built-in is implemented or not. For example, if `__builtin_avr_nop` is available the macro `__BUILTIN_AVR_NOP` is defined to `1` and undefined otherwise.

`void __builtin_avr_nop (void)`
`void __builtin_avr_sei (void)`
`void __builtin_avr_cli (void)`
`void __builtin_avr_sleep (void)`
`void __builtin_avr_wdr (void)`
`unsigned char __builtin_avr_swap (unsigned char)`
`unsigned int __builtin_avr_fmul (unsigned char, unsigned char)`
`int __builtin_avr_fmuls (char, char)`
`int __builtin_avr_fmulsu (char, unsigned char)`

These built-in functions map to the respective machine instruction, i.e. `nop`, `sei`, `cli`, `sleep`, `wdr`, `swap`, `fmul`, `fmuls` resp. `fmulsu`. The three `fmul*` built-ins are implemented as library call if no hardware multiplier is available.

`void __builtin_avr_delay_cycles (unsigned long ticks)`

Delay execution for *ticks* cycles. Note that this built-in does not take into account the effect of interrupts that might increase delay time. *ticks* must be a compile-time integer constant; delays with a variable number of cycles are not supported.

`char __builtin_avr_flash_segment (const __memx void*)`

This built-in takes a byte address to the 24-bit [AVR Named Address Spaces], page 466 `__memx` and returns the number of the flash segment (the 64 KiB chunk) where the address points to. Counting starts at `0`. If the address does not point to flash memory, return `-1`.

`uint8_t __builtin_avr_insert_bits (uint32_t map, uint8_t bits, uint8_t val)`

Insert bits from *bits* into *val* and return the resulting value. The nibbles of *map* determine how the insertion is performed: Let X be the n-th nibble of *map*

1. If X is `0xf`, then the n-th bit of *val* is returned unaltered.
2. If X is in the range 0...7, then the n-th result bit is set to the X-th bit of *bits*
3. If X is in the range 8...`0xe`, then the n-th result bit is undefined.

One typical use case for this built-in is adjusting input and output values to non-contiguous port layouts. Some examples:

```
// same as val, bits is unused
__builtin_avr_insert_bits (0xffffffff, bits, val)

// same as bits, val is unused
__builtin_avr_insert_bits (0x76543210, bits, val)

// same as rotating bits by 4
__builtin_avr_insert_bits (0x32107654, bits, 0)

// high nibble of result is the high nibble of val
// low nibble of result is the low nibble of bits
__builtin_avr_insert_bits (0xffff3210, bits, val)

// reverse the bit order of bits
__builtin_avr_insert_bits (0x01234567, bits, 0)
```

`void __builtin_avr_nops (unsigned count)`

Insert *count* `NOP` instructions. The number of instructions must be a compile-time integer constant.

There are many more AVR-specific built-in functions that are used to implement the ISO/IEC TR 18037 "Embedded C" fixed-point functions of section 7.18a.6. You don't need to use these built-ins directly. Instead, use the declarations as supplied by the `stdfix.h` header with GNU-C99:

```
#include <stdfix.h>

// Re-interpret the bit representation of unsigned 16-bit
// integer uval as Q-format 0.16 value.
unsigned fract get_bits (uint_ur_t uval)
{
    return urbits (uval);
}
```

6.60.11 Blackfin Built-in Functions

Currently, there are two Blackfin-specific built-in functions. These are used for generating `CSYNC` and `SSYNC` machine insns without using inline assembly; by using these built-in functions the compiler can automatically add workarounds for hardware errata involving these instructions. These functions are named as follows:

```
void __builtin_bfin_csync (void)
void __builtin_bfin_ssync (void)
```

6.60.12 FR-V Built-in Functions

GCC provides many FR-V-specific built-in functions. In general, these functions are intended to be compatible with those described by *FR-V Family, Softune C/C++ Compiler Manual (V6), Fujitsu Semiconductor*. The two exceptions are `__MDUNPACKH` and `__MBTOHE`, the GCC forms of which pass 128-bit values by pointer rather than by value.

Most of the functions are named after specific FR-V instructions. Such functions are said to be "directly mapped" and are summarized here in tabular form.

6.60.12.1 Argument Types

The arguments to the built-in functions can be divided into three groups: register numbers, compile-time constants and run-time values. In order to make this classification clear at a glance, the arguments and return values are given the following pseudo types:

Pseudo type	Real C type	Constant?	Description
`uh`	`unsigned short`	No	an unsigned halfword
`uw1`	`unsigned int`	No	an unsigned word
`sw1`	`int`	No	a signed word
`uw2`	`unsigned long long`	No	an unsigned doubleword
`sw2`	`long long`	No	a signed doubleword
`const`	`int`	Yes	an integer constant
`acc`	`int`	Yes	an ACC register number
`iacc`	`int`	Yes	an IACC register number

These pseudo types are not defined by GCC, they are simply a notational convenience used in this manual.

Arguments of type `uh`, `uw1`, `sw1`, `uw2` and `sw2` are evaluated at run time. They correspond to register operands in the underlying FR-V instructions.

`const` arguments represent immediate operands in the underlying FR-V instructions. They must be compile-time constants.

`acc` arguments are evaluated at compile time and specify the number of an accumulator register. For example, an `acc` argument of 2 selects the ACC2 register.

`iacc` arguments are similar to `acc` arguments but specify the number of an IACC register. See see Section 6.60.12.5 [Other Built-in Functions], page 674 for more details.

6.60.12.2 Directly-Mapped Integer Functions

The functions listed below map directly to FR-V I-type instructions.

Function prototype	Example usage	Assembly output
`sw1 __ADDSS (sw1, sw1)`	`c = __ADDSS (a, b)`	`ADDSS a,b,c`

sw1 __SCAN (sw1, sw1)	c = __SCAN (a, b)	SCAN a,b,c
sw1 __SCUTSS (sw1)	b = __SCUTSS (a)	SCUTSS a,b
sw1 __SLASS (sw1, sw1)	c = __SLASS (a, b)	SLASS a,b,c
void __SMASS (sw1, sw1)	__SMASS (a, b)	SMASS a,b
void __SMSSS (sw1, sw1)	__SMSSS (a, b)	SMSSS a,b
void __SMU (sw1, sw1)	__SMU (a, b)	SMU a,b
sw2 __SMUL (sw1, sw1)	c = __SMUL (a, b)	SMUL a,b,c
sw1 __SUBSS (sw1, sw1)	c = __SUBSS (a, b)	SUBSS a,b,c
uw2 __UMUL (uw1, uw1)	c = __UMUL (a, b)	UMUL a,b,c

6.60.12.3 Directly-Mapped Media Functions

The functions listed below map directly to FR-V M-type instructions.

Function prototype	Example usage	Assembly output
uw1 __MABSHS (sw1)	b = __MABSHS (a)	MABSHS a,b
void __MADDACCS (acc, acc)	__MADDACCS (b, a)	MADDACCS a,b
sw1 __MADDHSS (sw1, sw1)	c = __MADDHSS (a, b)	MADDHSS a,b,c
uw1 __MADDHUS (uw1, uw1)	c = __MADDHUS (a, b)	MADDHUS a,b,c
uw1 __MAND (uw1, uw1)	c = __MAND (a, b)	MAND a,b,c
void __MASACCS (acc, acc)	__MASACCS (b, a)	MASACCS a,b
uw1 __MAVEH (uw1, uw1)	c = __MAVEH (a, b)	MAVEH a,b,c
uw2 __MBTOH (uw1)	b = __MBTOH (a)	MBTOH a,b
void __MBTOHE (uw1 *, uw1)	__MBTOHE (&b, a)	MBTOHE a,b
void __MCLRACC (acc)	__MCLRACC (a)	MCLRACC a
void __MCLRACCA (void)	__MCLRACCA ()	MCLRACCA
uw1 __Mcop1 (uw1, uw1)	c = __Mcop1 (a, b)	Mcop1 a,b,c
uw1 __Mcop2 (uw1, uw1)	c = __Mcop2 (a, b)	Mcop2 a,b,c
uw1 __MCPLHI (uw2, const)	c = __MCPLHI (a, b)	MCPLHI a,#b,c
uw1 __MCPLI (uw2, const)	c = __MCPLI (a, b)	MCPLI a,#b,c
void __MCPXIS (acc, sw1, sw1)	__MCPXIS (c, a, b)	MCPXIS a,b,c
void __MCPXIU (acc, uw1, uw1)	__MCPXIU (c, a, b)	MCPXIU a,b,c
void __MCPXRS (acc, sw1, sw1)	__MCPXRS (c, a, b)	MCPXRS a,b,c
void __MCPXRU (acc, uw1, uw1)	__MCPXRU (c, a, b)	MCPXRU a,b,c
uw1 __MCUT (acc, uw1)	c = __MCUT (a, b)	MCUT a,b,c
uw1 __MCUTSS (acc, sw1)	c = __MCUTSS (a, b)	MCUTSS a,b,c
void __MDADDACCS (acc, acc)	__MDADDACCS (b, a)	MDADDACCS a,b
void __MDASACCS (acc, acc)	__MDASACCS (b, a)	MDASACCS a,b
uw2 __MDCUTSSI (acc, const)	c = __MDCUTSSI (a, b)	MDCUTSSI a,#b,c
uw2 __MDPACKH (uw2, uw2)	c = __MDPACKH (a, b)	MDPACKH a,b,c
uw2 __MDROTLI (uw2, const)	c = __MDROTLI (a, b)	MDROTLI a,#b,c
void __MDSUBACCS (acc, acc)	__MDSUBACCS (b, a)	MDSUBACCS a,b
void __MDUNPACKH (uw1 *, uw2)	__MDUNPACKH (&b, a)	MDUNPACKH a,b
uw2 __MEXPDHD (uw1, const)	c = __MEXPDHD (a, b)	MEXPDHD a,#b,c
uw1 __MEXPDHW (uw1, const)	c = __MEXPDHW (a, b)	MEXPDHW a,#b,c
uw1 __MHDSETH (uw1, const)	c = __MHDSETH (a, b)	MHDSETH a,#b,c
sw1 __MHDSETS (const)	b = __MHDSETS (a)	MHDSETS #a,b
uw1 __MHSETHIH (uw1, const)	b = __MHSETHIH (b, a)	MHSETHIH #a,b

sw1 __MHSETHIS (sw1, const)	*b* = __MHSETHIS (*b*, *a*)	MHSETHIS #*a*,*b*
uw1 __MHSETLOH (uw1, const)	*b* = __MHSETLOH (*b*, *a*)	MHSETLOH #*a*,*b*
sw1 __MHSETLOS (sw1, const)	*b* = __MHSETLOS (*b*, *a*)	MHSETLOS #*a*,*b*
uw1 __MHTOB (uw2)	*b* = __MHTOB (*a*)	MHTOB *a*,*b*
void __MMACHS (acc, sw1, sw1)	__MMACHS (*c*, *a*, *b*)	MMACHS *a*,*b*,*c*
void __MMACHU (acc, uw1, uw1)	__MMACHU (*c*, *a*, *b*)	MMACHU *a*,*b*,*c*
void __MMRDHS (acc, sw1, sw1)	__MMRDHS (*c*, *a*, *b*)	MMRDHS *a*,*b*,*c*
void __MMRDHU (acc, uw1, uw1)	__MMRDHU (*c*, *a*, *b*)	MMRDHU *a*,*b*,*c*
void __MMULHS (acc, sw1, sw1)	__MMULHS (*c*, *a*, *b*)	MMULHS *a*,*b*,*c*
void __MMULHU (acc, uw1, uw1)	__MMULHU (*c*, *a*, *b*)	MMULHU *a*,*b*,*c*
void __MMULXHS (acc, sw1, sw1)	__MMULXHS (*c*, *a*, *b*)	MMULXHS *a*,*b*,*c*
void __MMULXHU (acc, uw1, uw1)	__MMULXHU (*c*, *a*, *b*)	MMULXHU *a*,*b*,*c*
uw1 __MNOT (uw1)	*b* = __MNOT (*a*)	MNOT *a*,*b*
uw1 __MOR (uw1, uw1)	*c* = __MOR (*a*, *b*)	MOR *a*,*b*,*c*
uw1 __MPACKH (uh, uh)	*c* = __MPACKH (*a*, *b*)	MPACKH *a*,*b*,*c*
sw2 __MQADDHSS (sw2, sw2)	*c* = __MQADDHSS (*a*, *b*)	MQADDHSS *a*,*b*,*c*
uw2 __MQADDHUS (uw2, uw2)	*c* = __MQADDHUS (*a*, *b*)	MQADDHUS *a*,*b*,*c*
void __MQCPXIS (acc, sw2, sw2)	__MQCPXIS (*c*, *a*, *b*)	MQCPXIS *a*,*b*,*c*
void __MQCPXIU (acc, uw2, uw2)	__MQCPXIU (*c*, *a*, *b*)	MQCPXIU *a*,*b*,*c*
void __MQCPXRS (acc, sw2, sw2)	__MQCPXRS (*c*, *a*, *b*)	MQCPXRS *a*,*b*,*c*
void __MQCPXRU (acc, uw2, uw2)	__MQCPXRU (*c*, *a*, *b*)	MQCPXRU *a*,*b*,*c*
sw2 __MQLCLRHS (sw2, sw2)	*c* = __MQLCLRHS (*a*, *b*)	MQLCLRHS *a*,*b*,*c*
sw2 __MQLMTHS (sw2, sw2)	*c* = __MQLMTHS (*a*, *b*)	MQLMTHS *a*,*b*,*c*
void __MQMACHS (acc, sw2, sw2)	__MQMACHS (*c*, *a*, *b*)	MQMACHS *a*,*b*,*c*
void __MQMACHU (acc, uw2, uw2)	__MQMACHU (*c*, *a*, *b*)	MQMACHU *a*,*b*,*c*
void __MQMACXHS (acc, sw2, sw2)	__MQMACXHS (*c*, *a*, *b*)	MQMACXHS *a*,*b*,*c*
void __MQMULHS (acc, sw2, sw2)	__MQMULHS (*c*, *a*, *b*)	MQMULHS *a*,*b*,*c*
void __MQMULHU (acc, uw2, uw2)	__MQMULHU (*c*, *a*, *b*)	MQMULHU *a*,*b*,*c*
void __MQMULXHS (acc, sw2, sw2)	__MQMULXHS (*c*, *a*, *b*)	MQMULXHS *a*,*b*,*c*
void __MQMULXHU (acc, uw2, uw2)	__MQMULXHU (*c*, *a*, *b*)	MQMULXHU *a*,*b*,*c*
sw2 __MQSATHS (sw2, sw2)	*c* = __MQSATHS (*a*, *b*)	MQSATHS *a*,*b*,*c*
uw2 __MQSLLHI (uw2, int)	*c* = __MQSLLHI (*a*, *b*)	MQSLLHI *a*,*b*,*c*
sw2 __MQSRAHI (sw2, int)	*c* = __MQSRAHI (*a*, *b*)	MQSRAHI *a*,*b*,*c*
sw2 __MQSUBHSS (sw2, sw2)	*c* = __MQSUBHSS (*a*, *b*)	MQSUBHSS *a*,*b*,*c*
uw2 __MQSUBHUS (uw2, uw2)	*c* = __MQSUBHUS (*a*, *b*)	MQSUBHUS *a*,*b*,*c*
void __MQXMACHS (acc, sw2, sw2)	__MQXMACHS (*c*, *a*, *b*)	MQXMACHS *a*,*b*,*c*
void __MQXMACXHS (acc, sw2, sw2)	__MQXMACXHS (*c*, *a*, *b*)	MQXMACXHS *a*,*b*,*c*
uw1 __MRDACC (acc)	*b* = __MRDACC (*a*)	MRDACC *a*,*b*
uw1 __MRDACCG (acc)	*b* = __MRDACCG (*a*)	MRDACCG *a*,*b*
uw1 __MROTLI (uw1, const)	*c* = __MROTLI (*a*, *b*)	MROTLI *a*,#*b*,*c*
uw1 __MROTRI (uw1, const)	*c* = __MROTRI (*a*, *b*)	MROTRI *a*,#*b*,*c*
sw1 __MSATHS (sw1, sw1)	*c* = __MSATHS (*a*, *b*)	MSATHS *a*,*b*,*c*
uw1 __MSATHU (uw1, uw1)	*c* = __MSATHU (*a*, *b*)	MSATHU *a*,*b*,*c*
uw1 __MSLLHI (uw1, const)	*c* = __MSLLHI (*a*, *b*)	MSLLHI *a*,#*b*,*c*
sw1 __MSRAHI (sw1, const)	*c* = __MSRAHI (*a*, *b*)	MSRAHI *a*,#*b*,*c*
uw1 __MSRLHI (uw1, const)	*c* = __MSRLHI (*a*, *b*)	MSRLHI *a*,#*b*,*c*
void __MSUBACCS (acc, acc)	__MSUBACCS (*b*, *a*)	MSUBACCS *a*,*b*

`sw1 __MSUBHSS (sw1, sw1)`	`c = __MSUBHSS (a, b)`	`MSUBHSS a,b,c`
`uw1 __MSUBHUS (uw1, uw1)`	`c = __MSUBHUS (a, b)`	`MSUBHUS a,b,c`
`void __MTRAP (void)`	`__MTRAP ()`	`MTRAP`
`uw2 __MUNPACKH (uw1)`	`b = __MUNPACKH (a)`	`MUNPACKH a,b`
`uw1 __MWCUT (uw2, uw1)`	`c = __MWCUT (a, b)`	`MWCUT a,b,c`
`void __MWTACC (acc, uw1)`	`__MWTACC (b, a)`	`MWTACC a,b`
`void __MWTACCG (acc, uw1)`	`__MWTACCG (b, a)`	`MWTACCG a,b`
`uw1 __MXOR (uw1, uw1)`	`c = __MXOR (a, b)`	`MXOR a,b,c`

6.60.12.4 Raw Read/Write Functions

This sections describes built-in functions related to read and write instructions to access memory. These functions generate `membar` instructions to flush the I/O load and stores where appropriate, as described in Fujitsu's manual described above.

`unsigned char __builtin_read8 (void *data)`
`unsigned short __builtin_read16 (void *data)`
`unsigned long __builtin_read32 (void *data)`
`unsigned long long __builtin_read64 (void *data)`
`void __builtin_write8 (void *data, unsigned char datum)`
`void __builtin_write16 (void *data, unsigned short datum)`
`void __builtin_write32 (void *data, unsigned long datum)`
`void __builtin_write64 (void *data, unsigned long long datum)`

6.60.12.5 Other Built-in Functions

This section describes built-in functions that are not named after a specific FR-V instruction.

`sw2 __IACCreadll (iacc reg)`
Return the full 64-bit value of IACC0. The *reg* argument is reserved for future expansion and must be 0.

`sw1 __IACCreadl (iacc reg)`
Return the value of IACC0H if *reg* is 0 and IACC0L if *reg* is 1. Other values of *reg* are rejected as invalid.

`void __IACCsetll (iacc reg, sw2 x)`
Set the full 64-bit value of IACC0 to *x*. The *reg* argument is reserved for future expansion and must be 0.

`void __IACCsetl (iacc reg, sw1 x)`
Set IACC0H to *x* if *reg* is 0 and IACC0L to *x* if *reg* is 1. Other values of *reg* are rejected as invalid.

`void __data_prefetch0 (const void *x)`
Use the `dcpl` instruction to load the contents of address *x* into the data cache.

`void __data_prefetch (const void *x)`
Use the `nldub` instruction to load the contents of address *x* into the data cache. The instruction is issued in slot I1.

6.60.13 MIPS DSP Built-in Functions

The MIPS DSP Application-Specific Extension (ASE) includes new instructions that are designed to improve the performance of DSP and media applications. It provides instructions that operate on packed 8-bit/16-bit integer data, Q7, Q15 and Q31 fractional data.

GCC supports MIPS DSP operations using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 626) and a collection of MIPS-specific built-in functions. Both kinds of support are enabled by the '`-mdsp`' command-line option.

Revision 2 of the ASE was introduced in the second half of 2006. This revision adds extra instructions to the original ASE, but is otherwise backwards-compatible with it. You can select revision 2 using the command-line option '`-mdspr2`'; this option implies '`-mdsp`'.

The SCOUNT and POS bits of the DSP control register are global. The WRDSP, EXTPDP, EXTPDPV and MTHLIP instructions modify the SCOUNT and POS bits. During optimization, the compiler does not delete these instructions and it does not delete calls to functions containing these instructions.

At present, GCC only provides support for operations on 32-bit vectors. The vector type associated with 8-bit integer data is usually called `v4i8`, the vector type associated with Q7 is usually called `v4q7`, the vector type associated with 16-bit integer data is usually called `v2i16`, and the vector type associated with Q15 is usually called `v2q15`. They can be defined in C as follows:

```
typedef signed char v4i8 __attribute__ ((vector_size(4)));
typedef signed char v4q7 __attribute__ ((vector_size(4)));
typedef short v2i16 __attribute__ ((vector_size(4)));
typedef short v2q15 __attribute__ ((vector_size(4)));
```

`v4i8`, `v4q7`, `v2i16` and `v2q15` values are initialized in the same way as aggregates. For example:

```
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};

v2q15 c = {0x0fcb, 0x3a75};
v2q15 d;
d = (v2q15) {0.1234 * 0x1.0p15, 0.4567 * 0x1.0p15};
```

Note: The CPU's endianness determines the order in which values are packed. On little-endian targets, the first value is the least significant and the last value is the most significant. The opposite order applies to big-endian targets. For example, the code above sets the lowest byte of `a` to `1` on little-endian targets and `4` on big-endian targets.

Note: Q7, Q15 and Q31 values must be initialized with their integer representation. As shown in this example, the integer representation of a Q7 value can be obtained by multiplying the fractional value by `0x1.0p7`. The equivalent for Q15 values is to multiply by `0x1.0p15`. The equivalent for Q31 values is to multiply by `0x1.0p31`.

The table below lists the `v4i8` and `v2q15` operations for which hardware support exists. `a` and `b` are `v4i8` values, and `c` and `d` are `v2q15` values.

C code	MIPS instruction
`a + b`	`addu.qb`
`c + d`	`addq.ph`
`a - b`	`subu.qb`

`c - d`	`subq.ph`

The table below lists the `v2i16` operation for which hardware support exists for the DSP ASE REV 2. `e` and `f` are `v2i16` values.

C code	MIPS instruction
`e * f`	`mul.ph`

It is easier to describe the DSP built-in functions if we first define the following types:

```
typedef int q31;
typedef int i32;
typedef unsigned int ui32;
typedef long long a64;
```

`q31` and `i32` are actually the same as `int`, but we use `q31` to indicate a Q31 fractional value and `i32` to indicate a 32-bit integer value. Similarly, `a64` is the same as `long long`, but we use `a64` to indicate values that are placed in one of the four DSP accumulators (`$ac0`, `$ac1`, `$ac2` or `$ac3`).

Also, some built-in functions prefer or require immediate numbers as parameters, because the corresponding DSP instructions accept both immediate numbers and register operands, or accept immediate numbers only. The immediate parameters are listed as follows.

```
imm0_3: 0 to 3.
imm0_7: 0 to 7.
imm0_15: 0 to 15.
imm0_31: 0 to 31.
imm0_63: 0 to 63.
imm0_255: 0 to 255.
imm_n32_31: -32 to 31.
imm_n512_511: -512 to 511.
```

The following built-in functions map directly to a particular MIPS DSP instruction. Please refer to the architecture specification for details on what each instruction does.

```
v2q15 __builtin_mips_addq_ph (v2q15, v2q15)
v2q15 __builtin_mips_addq_s_ph (v2q15, v2q15)
q31 __builtin_mips_addq_s_w (q31, q31)
v4i8 __builtin_mips_addu_qb (v4i8, v4i8)
v4i8 __builtin_mips_addu_s_qb (v4i8, v4i8)
v2q15 __builtin_mips_subq_ph (v2q15, v2q15)
v2q15 __builtin_mips_subq_s_ph (v2q15, v2q15)
q31 __builtin_mips_subq_s_w (q31, q31)
v4i8 __builtin_mips_subu_qb (v4i8, v4i8)
v4i8 __builtin_mips_subu_s_qb (v4i8, v4i8)
i32 __builtin_mips_addsc (i32, i32)
i32 __builtin_mips_addwc (i32, i32)
i32 __builtin_mips_modsub (i32, i32)
i32 __builtin_mips_raddu_w_qb (v4i8)
v2q15 __builtin_mips_absq_s_ph (v2q15)
q31 __builtin_mips_absq_s_w (q31)
v4i8 __builtin_mips_precrq_qb_ph (v2q15, v2q15)
v2q15 __builtin_mips_precrq_ph_w (q31, q31)
v2q15 __builtin_mips_precrq_rs_ph_w (q31, q31)
v4i8 __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15)
q31 __builtin_mips_preceq_w_phl (v2q15)
q31 __builtin_mips_preceq_w_phr (v2q15)
v2q15 __builtin_mips_precequ_ph_qbl (v4i8)
v2q15 __builtin_mips_precequ_ph_qbr (v4i8)
v2q15 __builtin_mips_precequ_ph_qbla (v4i8)
v2q15 __builtin_mips_precequ_ph_qbra (v4i8)
```

```
v2q15 __builtin_mips_preceu_ph_qbl (v4i8)
v2q15 __builtin_mips_preceu_ph_qbr (v4i8)
v2q15 __builtin_mips_preceu_ph_qbla (v4i8)
v2q15 __builtin_mips_preceu_ph_qbra (v4i8)
v4i8 __builtin_mips_shll_qb (v4i8, imm0_7)
v4i8 __builtin_mips_shll_qb (v4i8, i32)
v2q15 __builtin_mips_shll_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shll_ph (v2q15, i32)
v2q15 __builtin_mips_shll_s_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shll_s_ph (v2q15, i32)
q31 __builtin_mips_shll_s_w (q31, imm0_31)
q31 __builtin_mips_shll_s_w (q31, i32)
v4i8 __builtin_mips_shrl_qb (v4i8, imm0_7)
v4i8 __builtin_mips_shrl_qb (v4i8, i32)
v2q15 __builtin_mips_shra_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shra_ph (v2q15, i32)
v2q15 __builtin_mips_shra_r_ph (v2q15, imm0_15)
v2q15 __builtin_mips_shra_r_ph (v2q15, i32)
q31 __builtin_mips_shra_r_w (q31, imm0_31)
q31 __builtin_mips_shra_r_w (q31, i32)
v2q15 __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15)
v2q15 __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15)
v2q15 __builtin_mips_mulq_rs_ph (v2q15, v2q15)
q31 __builtin_mips_muleq_s_w_phl (v2q15, v2q15)
q31 __builtin_mips_muleq_s_w_phr (v2q15, v2q15)
a64 __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8)
a64 __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8)
a64 __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8)
a64 __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8)
a64 __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_dpaq_sa_l_w (a64, q31, q31)
a64 __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_dpsq_sa_l_w (a64, q31, q31)
a64 __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15)
a64 __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15)
a64 __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15)
a64 __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15)
a64 __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15)
i32 __builtin_mips_bitrev (i32)
i32 __builtin_mips_insv (i32, i32)
v4i8 __builtin_mips_repl_qb (imm0_255)
v4i8 __builtin_mips_repl_qb (i32)
v2q15 __builtin_mips_repl_ph (imm_n512_511)
v2q15 __builtin_mips_repl_ph (i32)
void __builtin_mips_cmpu_eq_qb (v4i8, v4i8)
void __builtin_mips_cmpu_lt_qb (v4i8, v4i8)
void __builtin_mips_cmpu_le_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_eq_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_lt_qb (v4i8, v4i8)
i32 __builtin_mips_cmpgu_le_qb (v4i8, v4i8)
void __builtin_mips_cmp_eq_ph (v2q15, v2q15)
void __builtin_mips_cmp_lt_ph (v2q15, v2q15)
void __builtin_mips_cmp_le_ph (v2q15, v2q15)
v4i8 __builtin_mips_pick_qb (v4i8, v4i8)
v2q15 __builtin_mips_pick_ph (v2q15, v2q15)
v2q15 __builtin_mips_packrl_ph (v2q15, v2q15)
i32 __builtin_mips_extr_w (a64, imm0_31)
i32 __builtin_mips_extr_w (a64, i32)
```

```
i32 __builtin_mips_extr_r_w (a64, imm0_31)
i32 __builtin_mips_extr_s_h (a64, i32)
i32 __builtin_mips_extr_rs_w (a64, imm0_31)
i32 __builtin_mips_extr_rs_w (a64, i32)
i32 __builtin_mips_extr_s_h (a64, imm0_31)
i32 __builtin_mips_extr_r_w (a64, i32)
i32 __builtin_mips_extp (a64, imm0_31)
i32 __builtin_mips_extp (a64, i32)
i32 __builtin_mips_extpdp (a64, imm0_31)
i32 __builtin_mips_extpdp (a64, i32)
a64 __builtin_mips_shilo (a64, imm_n32_31)
a64 __builtin_mips_shilo (a64, i32)
a64 __builtin_mips_mthlip (a64, i32)
void __builtin_mips_wrdsp (i32, imm0_63)
i32 __builtin_mips_rddsp (imm0_63)
i32 __builtin_mips_lbux (void *, i32)
i32 __builtin_mips_lhx (void *, i32)
i32 __builtin_mips_lwx (void *, i32)
a64 __builtin_mips_ldx (void *, i32) [MIPS64 only]
i32 __builtin_mips_bposge32 (void)
a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);
```

The following built-in functions map directly to a particular MIPS DSP REV 2 instruction. Please refer to the architecture specification for details on what each instruction does.

```
v4q7 __builtin_mips_absq_s_qb (v4q7);
v2i16 __builtin_mips_addu_ph (v2i16, v2i16);
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);
v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);
i32 __builtin_mips_append (i32, i32, imm0_31);
i32 __builtin_mips_balign (i32, i32, imm0_3);
i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);
a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);
v2i16 __builtin_mips_mul_ph (v2i16, v2i16);
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);
q31 __builtin_mips_mulq_rs_w (q31, q31);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);
q31 __builtin_mips_mulq_s_w (q31, q31);
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);
v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);
i32 __builtin_mips_prepend (i32, i32, imm0_31);
v4i8 __builtin_mips_shra_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_r_qb (v4i8, imm0_7);
v4i8 __builtin_mips_shra_qb (v4i8, i32);
v4i8 __builtin_mips_shra_r_qb (v4i8, i32);
v2i16 __builtin_mips_shrl_ph (v2i16, imm0_15);
v2i16 __builtin_mips_shrl_ph (v2i16, i32);
v2i16 __builtin_mips_subu_ph (v2i16, v2i16);
```

```
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);
v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);
v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);
v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);
```

6.60.14 MIPS Paired-Single Support

The MIPS64 architecture includes a number of instructions that operate on pairs of single-precision floating-point values. Each pair is packed into a 64-bit floating-point register, with one element being designated the "upper half" and the other being designated the "lower half".

GCC supports paired-single operations using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 626) and a collection of MIPS-specific built-in functions. Both kinds of support are enabled by the '`-mpaired-single`' command-line option.

The vector type associated with paired-single values is usually called `v2sf`. It can be defined in C as follows:

```
typedef float v2sf __attribute__ ((vector_size (8)));
```

`v2sf` values are initialized in the same way as aggregates. For example:

```
v2sf a = {1.5, 9.1};
v2sf b;
float e, f;
b = (v2sf) {e, f};
```

Note: The CPU's endianness determines which value is stored in the upper half of a register and which value is stored in the lower half. On little-endian targets, the first value is the lower one and the second value is the upper one. The opposite order applies to big-endian targets. For example, the code above sets the lower half of `a` to `1.5` on little-endian targets and `9.1` on big-endian targets.

6.60.15 MIPS Loongson Built-in Functions

GCC provides intrinsics to access the SIMD instructions provided by the ST Microelectronics Loongson-2E and -2F processors. These intrinsics, available after inclusion of the `loongson.h` header file, operate on the following 64-bit vector types:

- `uint8x8_t`, a vector of eight unsigned 8-bit integers;
- `uint16x4_t`, a vector of four unsigned 16-bit integers;
- `uint32x2_t`, a vector of two unsigned 32-bit integers;
- `int8x8_t`, a vector of eight signed 8-bit integers;

- `int16x4_t`, a vector of four signed 16-bit integers;
- `int32x2_t`, a vector of two signed 32-bit integers.

The intrinsics provided are listed below; each is named after the machine instruction to which it corresponds, with suffixes added as appropriate to distinguish intrinsics that expand to the same machine instruction yet have different argument types. Refer to the architecture documentation for a description of the functionality of each instruction.

```
int16x4_t packsswh (int32x2_t s, int32x2_t t);
int8x8_t packsshb (int16x4_t s, int16x4_t t);
uint8x8_t packushb (uint16x4_t s, uint16x4_t t);
uint32x2_t paddw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t paddh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t paddb_u (uint8x8_t s, uint8x8_t t);
int32x2_t paddw_s (int32x2_t s, int32x2_t t);
int16x4_t paddh_s (int16x4_t s, int16x4_t t);
int8x8_t paddb_s (int8x8_t s, int8x8_t t);
uint64_t paddd_u (uint64_t s, uint64_t t);
int64_t paddd_s (int64_t s, int64_t t);
int16x4_t paddsh (int16x4_t s, int16x4_t t);
int8x8_t paddsb (int8x8_t s, int8x8_t t);
uint16x4_t paddush (uint16x4_t s, uint16x4_t t);
uint8x8_t paddusb (uint8x8_t s, uint8x8_t t);
uint64_t pandn_ud (uint64_t s, uint64_t t);
uint32x2_t pandn_uw (uint32x2_t s, uint32x2_t t);
uint16x4_t pandn_uh (uint16x4_t s, uint16x4_t t);
uint8x8_t pandn_ub (uint8x8_t s, uint8x8_t t);
int64_t pandn_sd (int64_t s, int64_t t);
int32x2_t pandn_sw (int32x2_t s, int32x2_t t);
int16x4_t pandn_sh (int16x4_t s, int16x4_t t);
int8x8_t pandn_sb (int8x8_t s, int8x8_t t);
uint16x4_t pavgh (uint16x4_t s, uint16x4_t t);
uint8x8_t pavgb (uint8x8_t s, uint8x8_t t);
uint32x2_t pcmpeqw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t pcmpeqh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t pcmpeqb_u (uint8x8_t s, uint8x8_t t);
int32x2_t pcmpeqw_s (int32x2_t s, int32x2_t t);
int16x4_t pcmpeqh_s (int16x4_t s, int16x4_t t);
int8x8_t pcmpeqb_s (int8x8_t s, int8x8_t t);
uint32x2_t pcmpgtw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t pcmpgth_u (uint16x4_t s, uint16x4_t t);
uint8x8_t pcmpgtb_u (uint8x8_t s, uint8x8_t t);
int32x2_t pcmpgtw_s (int32x2_t s, int32x2_t t);
int16x4_t pcmpgth_s (int16x4_t s, int16x4_t t);
int8x8_t pcmpgtb_s (int8x8_t s, int8x8_t t);
uint16x4_t pextrh_u (uint16x4_t s, int field);
int16x4_t pextrh_s (int16x4_t s, int field);
uint16x4_t pinsrh_0_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_1_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_2_u (uint16x4_t s, uint16x4_t t);
uint16x4_t pinsrh_3_u (uint16x4_t s, uint16x4_t t);
int16x4_t pinsrh_0_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_1_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_2_s (int16x4_t s, int16x4_t t);
int16x4_t pinsrh_3_s (int16x4_t s, int16x4_t t);
int32x2_t pmaddhw (int16x4_t s, int16x4_t t);
int16x4_t pmaxsh (int16x4_t s, int16x4_t t);
uint8x8_t pmaxub (uint8x8_t s, uint8x8_t t);
```

```
int16x4_t pminsh (int16x4_t s, int16x4_t t);
uint8x8_t pminub (uint8x8_t s, uint8x8_t t);
uint8x8_t pmovmskb_u (uint8x8_t s);
int8x8_t pmovmskb_s (int8x8_t s);
uint16x4_t pmulhuh (uint16x4_t s, uint16x4_t t);
int16x4_t pmulhh (int16x4_t s, int16x4_t t);
int16x4_t pmullh (int16x4_t s, int16x4_t t);
int64_t pmuluw (uint32x2_t s, uint32x2_t t);
uint8x8_t pasubub (uint8x8_t s, uint8x8_t t);
uint16x4_t biadd (uint8x8_t s);
uint16x4_t psadbh (uint8x8_t s, uint8x8_t t);
uint16x4_t pshufh_u (uint16x4_t dest, uint16x4_t s, uint8_t order);
int16x4_t pshufh_s (int16x4_t dest, int16x4_t s, uint8_t order);
uint16x4_t psllh_u (uint16x4_t s, uint8_t amount);
int16x4_t psllh_s (int16x4_t s, uint8_t amount);
uint32x2_t psllw_u (uint32x2_t s, uint8_t amount);
int32x2_t psllw_s (int32x2_t s, uint8_t amount);
uint16x4_t psrlh_u (uint16x4_t s, uint8_t amount);
int16x4_t psrlh_s (int16x4_t s, uint8_t amount);
uint32x2_t psrlw_u (uint32x2_t s, uint8_t amount);
int32x2_t psrlw_s (int32x2_t s, uint8_t amount);
uint16x4_t psrah_u (uint16x4_t s, uint8_t amount);
int16x4_t psrah_s (int16x4_t s, uint8_t amount);
uint32x2_t psraw_u (uint32x2_t s, uint8_t amount);
int32x2_t psraw_s (int32x2_t s, uint8_t amount);
uint32x2_t psubw_u (uint32x2_t s, uint32x2_t t);
uint16x4_t psubh_u (uint16x4_t s, uint16x4_t t);
uint8x8_t psubb_u (uint8x8_t s, uint8x8_t t);
int32x2_t psubw_s (int32x2_t s, int32x2_t t);
int16x4_t psubh_s (int16x4_t s, int16x4_t t);
int8x8_t psubb_s (int8x8_t s, int8x8_t t);
uint64_t psubd_u (uint64_t s, uint64_t t);
int64_t psubd_s (int64_t s, int64_t t);
int16x4_t psubsh (int16x4_t s, int16x4_t t);
int8x8_t psubsb (int8x8_t s, int8x8_t t);
uint16x4_t psubush (uint16x4_t s, uint16x4_t t);
uint8x8_t psubusb (uint8x8_t s, uint8x8_t t);
uint32x2_t punpckhwd_u (uint32x2_t s, uint32x2_t t);
uint16x4_t punpckhhw_u (uint16x4_t s, uint16x4_t t);
uint8x8_t punpckhbh_u (uint8x8_t s, uint8x8_t t);
int32x2_t punpckhwd_s (int32x2_t s, int32x2_t t);
int16x4_t punpckhhw_s (int16x4_t s, int16x4_t t);
int8x8_t punpckhbh_s (int8x8_t s, int8x8_t t);
uint32x2_t punpcklwd_u (uint32x2_t s, uint32x2_t t);
uint16x4_t punpcklhw_u (uint16x4_t s, uint16x4_t t);
uint8x8_t punpcklbh_u (uint8x8_t s, uint8x8_t t);
int32x2_t punpcklwd_s (int32x2_t s, int32x2_t t);
int16x4_t punpcklhw_s (int16x4_t s, int16x4_t t);
int8x8_t punpcklbh_s (int8x8_t s, int8x8_t t);
```

6.60.15.1 Paired-Single Arithmetic

The table below lists the `v2sf` operations for which hardware support exists. `a`, `b` and `c` are `v2sf` values and `x` is an integral value.

C code	MIPS instruction
`a + b`	`add.ps`
`a - b`	`sub.ps`

`-a`	`neg.ps`
`a * b`	`mul.ps`
`a * b + c`	`madd.ps`
`a * b - c`	`msub.ps`
`-(a * b + c)`	`nmadd.ps`
`-(a * b - c)`	`nmsub.ps`
`x ? a : b`	`movn.ps/movz.ps`

Note that the multiply-accumulate instructions can be disabled using the command-line option `-mno-fused-madd`.

6.60.15.2 Paired-Single Built-in Functions

The following paired-single functions map directly to a particular MIPS instruction. Please refer to the architecture specification for details on what each instruction does.

`v2sf __builtin_mips_pll_ps (v2sf, v2sf)`
: Pair lower lower (`pll.ps`).

`v2sf __builtin_mips_pul_ps (v2sf, v2sf)`
: Pair upper lower (`pul.ps`).

`v2sf __builtin_mips_plu_ps (v2sf, v2sf)`
: Pair lower upper (`plu.ps`).

`v2sf __builtin_mips_puu_ps (v2sf, v2sf)`
: Pair upper upper (`puu.ps`).

`v2sf __builtin_mips_cvt_ps_s (float, float)`
: Convert pair to paired single (`cvt.ps.s`).

`float __builtin_mips_cvt_s_pl (v2sf)`
: Convert pair lower to single (`cvt.s.pl`).

`float __builtin_mips_cvt_s_pu (v2sf)`
: Convert pair upper to single (`cvt.s.pu`).

`v2sf __builtin_mips_abs_ps (v2sf)`
: Absolute value (`abs.ps`).

`v2sf __builtin_mips_alnv_ps (v2sf, v2sf, int)`
: Align variable (`alnv.ps`).

  Note: The value of the third parameter must be 0 or 4 modulo 8, otherwise the result is unpredictable. Please read the instruction description for details.

The following multi-instruction functions are also available. In each case, *cond* can be any of the 16 floating-point conditions: `f`, `un`, `eq`, `ueq`, `olt`, `ult`, `ole`, `ule`, `sf`, `ngle`, `seq`, `ngl`, `lt`, `nge`, `le` or `ngt`.

`v2sf __builtin_mips_movt_c_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
`v2sf __builtin_mips_movf_c_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
: Conditional move based on floating-point comparison (`c.`*`cond`*`.ps`, `movt.ps/movf.ps`).

  The `movt` functions return the value x computed by:

```
c.cond.ps cc,a,b
mov.ps x,c
movt.ps x,d,cc
```

The `movf` functions are similar but use `movf.ps` instead of `movt.ps`.

`int __builtin_mips_upper_c_cond_ps (v2sf a, v2sf b)`
`int __builtin_mips_lower_c_cond_ps (v2sf a, v2sf b)`

Comparison of two paired-single values (`c.cond.ps`, `bc1t`/`bc1f`).

These functions compare *a* and *b* using `c.cond.ps` and return either the upper or lower half of the result. For example:

```
v2sf a, b;
if (__builtin_mips_upper_c_eq_ps (a, b))
  upper_halves_are_equal ();
else
  upper_halves_are_unequal ();

if (__builtin_mips_lower_c_eq_ps (a, b))
  lower_halves_are_equal ();
else
  lower_halves_are_unequal ();
```

6.60.15.3 MIPS-3D Built-in Functions

The MIPS-3D Application-Specific Extension (ASE) includes additional paired-single instructions that are designed to improve the performance of 3D graphics operations. Support for these instructions is controlled by the '`-mips3d`' command-line option.

The functions listed below map directly to a particular MIPS-3D instruction. Please refer to the architecture specification for more details on what each instruction does.

`v2sf __builtin_mips_addr_ps (v2sf, v2sf)`

Reduction add (`addr.ps`).

`v2sf __builtin_mips_mulr_ps (v2sf, v2sf)`

Reduction multiply (`mulr.ps`).

`v2sf __builtin_mips_cvt_pw_ps (v2sf)`

Convert paired single to paired word (`cvt.pw.ps`).

`v2sf __builtin_mips_cvt_ps_pw (v2sf)`

Convert paired word to paired single (`cvt.ps.pw`).

`float __builtin_mips_recip1_s (float)`
`double __builtin_mips_recip1_d (double)`
`v2sf __builtin_mips_recip1_ps (v2sf)`

Reduced-precision reciprocal (sequence step 1) (`recip1.fmt`).

`float __builtin_mips_recip2_s (float, float)`
`double __builtin_mips_recip2_d (double, double)`
`v2sf __builtin_mips_recip2_ps (v2sf, v2sf)`

Reduced-precision reciprocal (sequence step 2) (`recip2.fmt`).

`float __builtin_mips_rsqrt1_s (float)`
`double __builtin_mips_rsqrt1_d (double)`
`v2sf __builtin_mips_rsqrt1_ps (v2sf)`

Reduced-precision reciprocal square root (sequence step 1) (`rsqrt1.fmt`).

`float __builtin_mips_rsqrt2_s (float, float)`
`double __builtin_mips_rsqrt2_d (double, double)`
`v2sf __builtin_mips_rsqrt2_ps (v2sf, v2sf)`

Reduced-precision reciprocal square root (sequence step 2) (`rsqrt2.`*`fmt`*).

The following multi-instruction functions are also available. In each case, *cond* can be any of the 16 floating-point conditions: `f`, `un`, `eq`, `ueq`, `olt`, `ult`, `ole`, `ule`, `sf`, `ngle`, `seq`, `ngl`, `lt`, `nge`, `le` or `ngt`.

`int __builtin_mips_cabs_`*`cond`*`_s (float a, float b)`
`int __builtin_mips_cabs_`*`cond`*`_d (double a, double b)`

Absolute comparison of two scalar values (`cabs.`*`cond.fmt`*, `bc1t`/`bc1f`).

These functions compare *a* and *b* using `cabs.`*`cond`*`.s` or `cabs.`*`cond`*`.d` and return the result as a boolean value. For example:

```
float a, b;
if (__builtin_mips_cabs_eq_s (a, b))
  true ();
else
  false ();
```

`int __builtin_mips_upper_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_lower_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`

Absolute comparison of two paired-single values (`cabs.`*`cond`*`.ps`, `bc1t`/`bc1f`).

These functions compare *a* and *b* using `cabs.`*`cond`*`.ps` and return either the upper or lower half of the result. For example:

```
v2sf a, b;
if (__builtin_mips_upper_cabs_eq_ps (a, b))
  upper_halves_are_equal ();
else
  upper_halves_are_unequal ();

if (__builtin_mips_lower_cabs_eq_ps (a, b))
  lower_halves_are_equal ();
else
  lower_halves_are_unequal ();
```

`v2sf __builtin_mips_movt_cabs_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`
`v2sf __builtin_mips_movf_cabs_`*`cond`*`_ps (v2sf a, v2sf b, v2sf c, v2sf d)`

Conditional move based on absolute comparison (`cabs.`*`cond`*`.ps`, `movt.ps`/`movf.ps`).

The `movt` functions return the value *x* computed by:

```
cabs.cond.ps cc,a,b
mov.ps x,c
movt.ps x,d,cc
```

The `movf` functions are similar but use `movf.ps` instead of `movt.ps`.

`int __builtin_mips_any_c_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_all_c_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_any_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`
`int __builtin_mips_all_cabs_`*`cond`*`_ps (v2sf a, v2sf b)`

Comparison of two paired-single values (`c.`*`cond`*`.ps`/`cabs.`*`cond`*`.ps`, `bc1any2t`/`bc1any2f`).

These functions compare *a* and *b* using `c.cond.ps` or `cabs.cond.ps`. The `any` forms return `true` if either result is `true` and the `all` forms return `true` if both results are `true`. For example:

```
v2sf a, b;
if (__builtin_mips_any_c_eq_ps (a, b))
  one_is_true ();
else
  both_are_false ();

if (__builtin_mips_all_c_eq_ps (a, b))
  both_are_true ();
else
  one_is_false ();
```

`int __builtin_mips_any_c_cond_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_all_c_cond_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_any_cabs_cond_4s (v2sf a, v2sf b, v2sf c, v2sf d)`
`int __builtin_mips_all_cabs_cond_4s (v2sf a, v2sf b, v2sf c, v2sf d)`

Comparison of four paired-single values (`c.cond.ps`/`cabs.cond.ps`, `bc1any4t`/`bc1any4f`).

These functions use `c.cond.ps` or `cabs.cond.ps` to compare *a* with *b* and to compare *c* with *d*. The `any` forms return `true` if any of the four results are `true` and the `all` forms return `true` if all four results are `true`. For example:

```
v2sf a, b, c, d;
if (__builtin_mips_any_c_eq_4s (a, b, c, d))
  some_are_true ();
else
  all_are_false ();

if (__builtin_mips_all_c_eq_4s (a, b, c, d))
  all_are_true ();
else
  some_are_false ();
```

6.60.16 MIPS SIMD Architecture (MSA) Support

GCC provides intrinsics to access the SIMD instructions provided by the MSA MIPS SIMD Architecture. The interface is made available by including `<msa.h>` and using '`-mmsa -mhard-float -mfp64 -mnan=2008`'. For each `__builtin_msa_*`, there is a shortened name of the intrinsic, `__msa_*`.

MSA implements 128-bit wide vector registers, operating on 8-, 16-, 32- and 64-bit integer, 16- and 32-bit fixed-point, or 32- and 64-bit floating point data elements. The following vectors typedefs are included in `msa.h`:

- `v16i8`, a vector of sixteen signed 8-bit integers;
- `v16u8`, a vector of sixteen unsigned 8-bit integers;
- `v8i16`, a vector of eight signed 16-bit integers;
- `v8u16`, a vector of eight unsigned 16-bit integers;
- `v4i32`, a vector of four signed 32-bit integers;
- `v4u32`, a vector of four unsigned 32-bit integers;
- `v2i64`, a vector of two signed 64-bit integers;

- `v2u64`, a vector of two unsigned 64-bit integers;
- `v4f32`, a vector of four 32-bit floats;
- `v2f64`, a vector of two 64-bit doubles.

Instructions and corresponding built-ins may have additional restrictions and/or input/output values manipulated:

- `imm0_1`, an integer literal in range 0 to 1;
- `imm0_3`, an integer literal in range 0 to 3;
- `imm0_7`, an integer literal in range 0 to 7;
- `imm0_15`, an integer literal in range 0 to 15;
- `imm0_31`, an integer literal in range 0 to 31;
- `imm0_63`, an integer literal in range 0 to 63;
- `imm0_255`, an integer literal in range 0 to 255;
- `imm_n16_15`, an integer literal in range -16 to 15;
- `imm_n512_511`, an integer literal in range -512 to 511;
- `imm_n1024_1022`, an integer literal in range -512 to 511 left shifted by 1 bit, i.e., -1024, -1022, . . . , 1020, 1022;
- `imm_n2048_2044`, an integer literal in range -512 to 511 left shifted by 2 bits, i.e., -2048, -2044, . . . , 2040, 2044;
- `imm_n4096_4088`, an integer literal in range -512 to 511 left shifted by 3 bits, i.e., -4096, -4088, . . . , 4080, 4088;
- `imm1_4`, an integer literal in range 1 to 4;
- `i32, i64, u32, u64, f32, f64`, defined as follows:

```
{
typedef int i32;
#if __LONG_MAX__ == __LONG_LONG_MAX__
typedef long i64;
#else
typedef long long i64;
#endif

typedef unsigned int u32;
#if __LONG_MAX__ == __LONG_LONG_MAX__
typedef unsigned long u64;
#else
typedef unsigned long long u64;
#endif

typedef double f64;
typedef float f32;
}
```

6.60.16.1 MIPS SIMD Architecture Built-in Functions

The intrinsics provided are listed below; each is named after the machine instruction.

```
v16i8 __builtin_msa_add_a_b (v16i8, v16i8);
v8i16 __builtin_msa_add_a_h (v8i16, v8i16);
v4i32 __builtin_msa_add_a_w (v4i32, v4i32);
v2i64 __builtin_msa_add_a_d (v2i64, v2i64);
```

```
v16i8 __builtin_msa_adds_a_b (v16i8, v16i8);
v8i16 __builtin_msa_adds_a_h (v8i16, v8i16);
v4i32 __builtin_msa_adds_a_w (v4i32, v4i32);
v2i64 __builtin_msa_adds_a_d (v2i64, v2i64);

v16i8 __builtin_msa_adds_s_b (v16i8, v16i8);
v8i16 __builtin_msa_adds_s_h (v8i16, v8i16);
v4i32 __builtin_msa_adds_s_w (v4i32, v4i32);
v2i64 __builtin_msa_adds_s_d (v2i64, v2i64);

v16u8 __builtin_msa_adds_u_b (v16u8, v16u8);
v8u16 __builtin_msa_adds_u_h (v8u16, v8u16);
v4u32 __builtin_msa_adds_u_w (v4u32, v4u32);
v2u64 __builtin_msa_adds_u_d (v2u64, v2u64);

v16i8 __builtin_msa_addv_b (v16i8, v16i8);
v8i16 __builtin_msa_addv_h (v8i16, v8i16);
v4i32 __builtin_msa_addv_w (v4i32, v4i32);
v2i64 __builtin_msa_addv_d (v2i64, v2i64);

v16i8 __builtin_msa_addvi_b (v16i8, imm0_31);
v8i16 __builtin_msa_addvi_h (v8i16, imm0_31);
v4i32 __builtin_msa_addvi_w (v4i32, imm0_31);
v2i64 __builtin_msa_addvi_d (v2i64, imm0_31);

v16u8 __builtin_msa_and_v (v16u8, v16u8);

v16u8 __builtin_msa_andi_b (v16u8, imm0_255);

v16i8 __builtin_msa_asub_s_b (v16i8, v16i8);
v8i16 __builtin_msa_asub_s_h (v8i16, v8i16);
v4i32 __builtin_msa_asub_s_w (v4i32, v4i32);
v2i64 __builtin_msa_asub_s_d (v2i64, v2i64);

v16u8 __builtin_msa_asub_u_b (v16u8, v16u8);
v8u16 __builtin_msa_asub_u_h (v8u16, v8u16);
v4u32 __builtin_msa_asub_u_w (v4u32, v4u32);
v2u64 __builtin_msa_asub_u_d (v2u64, v2u64);

v16i8 __builtin_msa_ave_s_b (v16i8, v16i8);
v8i16 __builtin_msa_ave_s_h (v8i16, v8i16);
v4i32 __builtin_msa_ave_s_w (v4i32, v4i32);
v2i64 __builtin_msa_ave_s_d (v2i64, v2i64);

v16u8 __builtin_msa_ave_u_b (v16u8, v16u8);
v8u16 __builtin_msa_ave_u_h (v8u16, v8u16);
v4u32 __builtin_msa_ave_u_w (v4u32, v4u32);
v2u64 __builtin_msa_ave_u_d (v2u64, v2u64);

v16i8 __builtin_msa_aver_s_b (v16i8, v16i8);
v8i16 __builtin_msa_aver_s_h (v8i16, v8i16);
v4i32 __builtin_msa_aver_s_w (v4i32, v4i32);
v2i64 __builtin_msa_aver_s_d (v2i64, v2i64);

v16u8 __builtin_msa_aver_u_b (v16u8, v16u8);
v8u16 __builtin_msa_aver_u_h (v8u16, v8u16);
v4u32 __builtin_msa_aver_u_w (v4u32, v4u32);
```

```
v2u64 __builtin_msa_aver_u_d (v2u64, v2u64);

v16u8 __builtin_msa_bclr_b (v16u8, v16u8);
v8u16 __builtin_msa_bclr_h (v8u16, v8u16);
v4u32 __builtin_msa_bclr_w (v4u32, v4u32);
v2u64 __builtin_msa_bclr_d (v2u64, v2u64);

v16u8 __builtin_msa_bclri_b (v16u8, imm0_7);
v8u16 __builtin_msa_bclri_h (v8u16, imm0_15);
v4u32 __builtin_msa_bclri_w (v4u32, imm0_31);
v2u64 __builtin_msa_bclri_d (v2u64, imm0_63);

v16u8 __builtin_msa_binsl_b (v16u8, v16u8, v16u8);
v8u16 __builtin_msa_binsl_h (v8u16, v8u16, v8u16);
v4u32 __builtin_msa_binsl_w (v4u32, v4u32, v4u32);
v2u64 __builtin_msa_binsl_d (v2u64, v2u64, v2u64);

v16u8 __builtin_msa_binsli_b (v16u8, v16u8, imm0_7);
v8u16 __builtin_msa_binsli_h (v8u16, v8u16, imm0_15);
v4u32 __builtin_msa_binsli_w (v4u32, v4u32, imm0_31);
v2u64 __builtin_msa_binsli_d (v2u64, v2u64, imm0_63);

v16u8 __builtin_msa_binsr_b (v16u8, v16u8, v16u8);
v8u16 __builtin_msa_binsr_h (v8u16, v8u16, v8u16);
v4u32 __builtin_msa_binsr_w (v4u32, v4u32, v4u32);
v2u64 __builtin_msa_binsr_d (v2u64, v2u64, v2u64);

v16u8 __builtin_msa_binsri_b (v16u8, v16u8, imm0_7);
v8u16 __builtin_msa_binsri_h (v8u16, v8u16, imm0_15);
v4u32 __builtin_msa_binsri_w (v4u32, v4u32, imm0_31);
v2u64 __builtin_msa_binsri_d (v2u64, v2u64, imm0_63);

v16u8 __builtin_msa_bmnz_v (v16u8, v16u8, v16u8);

v16u8 __builtin_msa_bmnzi_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bmz_v (v16u8, v16u8, v16u8);

v16u8 __builtin_msa_bmzi_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bneg_b (v16u8, v16u8);
v8u16 __builtin_msa_bneg_h (v8u16, v8u16);
v4u32 __builtin_msa_bneg_w (v4u32, v4u32);
v2u64 __builtin_msa_bneg_d (v2u64, v2u64);

v16u8 __builtin_msa_bnegi_b (v16u8, imm0_7);
v8u16 __builtin_msa_bnegi_h (v8u16, imm0_15);
v4u32 __builtin_msa_bnegi_w (v4u32, imm0_31);
v2u64 __builtin_msa_bnegi_d (v2u64, imm0_63);

i32 __builtin_msa_bnz_b (v16u8);
i32 __builtin_msa_bnz_h (v8u16);
i32 __builtin_msa_bnz_w (v4u32);
i32 __builtin_msa_bnz_d (v2u64);

i32 __builtin_msa_bnz_v (v16u8);

v16u8 __builtin_msa_bsel_v (v16u8, v16u8, v16u8);
```

```
v16u8 __builtin_msa_bseli_b (v16u8, v16u8, imm0_255);

v16u8 __builtin_msa_bset_b (v16u8, v16u8);
v8u16 __builtin_msa_bset_h (v8u16, v8u16);
v4u32 __builtin_msa_bset_w (v4u32, v4u32);
v2u64 __builtin_msa_bset_d (v2u64, v2u64);

v16u8 __builtin_msa_bseti_b (v16u8, imm0_7);
v8u16 __builtin_msa_bseti_h (v8u16, imm0_15);
v4u32 __builtin_msa_bseti_w (v4u32, imm0_31);
v2u64 __builtin_msa_bseti_d (v2u64, imm0_63);

i32 __builtin_msa_bz_b (v16u8);
i32 __builtin_msa_bz_h (v8u16);
i32 __builtin_msa_bz_w (v4u32);
i32 __builtin_msa_bz_d (v2u64);

i32 __builtin_msa_bz_v (v16u8);

v16i8 __builtin_msa_ceq_b (v16i8, v16i8);
v8i16 __builtin_msa_ceq_h (v8i16, v8i16);
v4i32 __builtin_msa_ceq_w (v4i32, v4i32);
v2i64 __builtin_msa_ceq_d (v2i64, v2i64);

v16i8 __builtin_msa_ceqi_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_ceqi_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_ceqi_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_ceqi_d (v2i64, imm_n16_15);

i32 __builtin_msa_cfcmsa (imm0_31);

v16i8 __builtin_msa_cle_s_b (v16i8, v16i8);
v8i16 __builtin_msa_cle_s_h (v8i16, v8i16);
v4i32 __builtin_msa_cle_s_w (v4i32, v4i32);
v2i64 __builtin_msa_cle_s_d (v2i64, v2i64);

v16i8 __builtin_msa_cle_u_b (v16u8, v16u8);
v8i16 __builtin_msa_cle_u_h (v8u16, v8u16);
v4i32 __builtin_msa_cle_u_w (v4u32, v4u32);
v2i64 __builtin_msa_cle_u_d (v2u64, v2u64);

v16i8 __builtin_msa_clei_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_clei_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_clei_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_clei_s_d (v2i64, imm_n16_15);

v16i8 __builtin_msa_clei_u_b (v16u8, imm0_31);
v8i16 __builtin_msa_clei_u_h (v8u16, imm0_31);
v4i32 __builtin_msa_clei_u_w (v4u32, imm0_31);
v2i64 __builtin_msa_clei_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_clt_s_b (v16i8, v16i8);
v8i16 __builtin_msa_clt_s_h (v8i16, v8i16);
v4i32 __builtin_msa_clt_s_w (v4i32, v4i32);
v2i64 __builtin_msa_clt_s_d (v2i64, v2i64);

v16i8 __builtin_msa_clt_u_b (v16u8, v16u8);
```

```
v8i16 __builtin_msa_clt_u_h (v8u16, v8u16);
v4i32 __builtin_msa_clt_u_w (v4u32, v4u32);
v2i64 __builtin_msa_clt_u_d (v2u64, v2u64);

v16i8 __builtin_msa_clti_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_clti_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_clti_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_clti_s_d (v2i64, imm_n16_15);

v16i8 __builtin_msa_clti_u_b (v16u8, imm0_31);
v8i16 __builtin_msa_clti_u_h (v8u16, imm0_31);
v4i32 __builtin_msa_clti_u_w (v4u32, imm0_31);
v2i64 __builtin_msa_clti_u_d (v2u64, imm0_31);

i32 __builtin_msa_copy_s_b (v16i8, imm0_15);
i32 __builtin_msa_copy_s_h (v8i16, imm0_7);
i32 __builtin_msa_copy_s_w (v4i32, imm0_3);
i64 __builtin_msa_copy_s_d (v2i64, imm0_1);

u32 __builtin_msa_copy_u_b (v16i8, imm0_15);
u32 __builtin_msa_copy_u_h (v8i16, imm0_7);
u32 __builtin_msa_copy_u_w (v4i32, imm0_3);
u64 __builtin_msa_copy_u_d (v2i64, imm0_1);

void __builtin_msa_ctcmsa (imm0_31, i32);

v16i8 __builtin_msa_div_s_b (v16i8, v16i8);
v8i16 __builtin_msa_div_s_h (v8i16, v8i16);
v4i32 __builtin_msa_div_s_w (v4i32, v4i32);
v2i64 __builtin_msa_div_s_d (v2i64, v2i64);

v16u8 __builtin_msa_div_u_b (v16u8, v16u8);
v8u16 __builtin_msa_div_u_h (v8u16, v8u16);
v4u32 __builtin_msa_div_u_w (v4u32, v4u32);
v2u64 __builtin_msa_div_u_d (v2u64, v2u64);

v8i16 __builtin_msa_dotp_s_h (v16i8, v16i8);
v4i32 __builtin_msa_dotp_s_w (v8i16, v8i16);
v2i64 __builtin_msa_dotp_s_d (v4i32, v4i32);

v8u16 __builtin_msa_dotp_u_h (v16u8, v16u8);
v4u32 __builtin_msa_dotp_u_w (v8u16, v8u16);
v2u64 __builtin_msa_dotp_u_d (v4u32, v4u32);

v8i16 __builtin_msa_dpadd_s_h (v8i16, v16i8, v16i8);
v4i32 __builtin_msa_dpadd_s_w (v4i32, v8i16, v8i16);
v2i64 __builtin_msa_dpadd_s_d (v2i64, v4i32, v4i32);

v8u16 __builtin_msa_dpadd_u_h (v8u16, v16u8, v16u8);
v4u32 __builtin_msa_dpadd_u_w (v4u32, v8u16, v8u16);
v2u64 __builtin_msa_dpadd_u_d (v2u64, v4u32, v4u32);

v8i16 __builtin_msa_dpsub_s_h (v8i16, v16i8, v16i8);
v4i32 __builtin_msa_dpsub_s_w (v4i32, v8i16, v8i16);
v2i64 __builtin_msa_dpsub_s_d (v2i64, v4i32, v4i32);

v8i16 __builtin_msa_dpsub_u_h (v8i16, v16u8, v16u8);
v4i32 __builtin_msa_dpsub_u_w (v4i32, v8u16, v8u16);
```

```
v2i64 __builtin_msa_dpsub_u_d (v2i64, v4u32, v4u32);

v4f32 __builtin_msa_fadd_w (v4f32, v4f32);
v2f64 __builtin_msa_fadd_d (v2f64, v2f64);

v4i32 __builtin_msa_fcaf_w (v4f32, v4f32);
v2i64 __builtin_msa_fcaf_d (v2f64, v2f64);

v4i32 __builtin_msa_fceq_w (v4f32, v4f32);
v2i64 __builtin_msa_fceq_d (v2f64, v2f64);

v4i32 __builtin_msa_fclass_w (v4f32);
v2i64 __builtin_msa_fclass_d (v2f64);

v4i32 __builtin_msa_fcle_w (v4f32, v4f32);
v2i64 __builtin_msa_fcle_d (v2f64, v2f64);

v4i32 __builtin_msa_fclt_w (v4f32, v4f32);
v2i64 __builtin_msa_fclt_d (v2f64, v2f64);

v4i32 __builtin_msa_fcne_w (v4f32, v4f32);
v2i64 __builtin_msa_fcne_d (v2f64, v2f64);

v4i32 __builtin_msa_fcor_w (v4f32, v4f32);
v2i64 __builtin_msa_fcor_d (v2f64, v2f64);

v4i32 __builtin_msa_fcueq_w (v4f32, v4f32);
v2i64 __builtin_msa_fcueq_d (v2f64, v2f64);

v4i32 __builtin_msa_fcule_w (v4f32, v4f32);
v2i64 __builtin_msa_fcule_d (v2f64, v2f64);

v4i32 __builtin_msa_fcult_w (v4f32, v4f32);
v2i64 __builtin_msa_fcult_d (v2f64, v2f64);

v4i32 __builtin_msa_fcun_w (v4f32, v4f32);
v2i64 __builtin_msa_fcun_d (v2f64, v2f64);

v4i32 __builtin_msa_fcune_w (v4f32, v4f32);
v2i64 __builtin_msa_fcune_d (v2f64, v2f64);

v4f32 __builtin_msa_fdiv_w (v4f32, v4f32);
v2f64 __builtin_msa_fdiv_d (v2f64, v2f64);

v8i16 __builtin_msa_fexdo_h (v4f32, v4f32);
v4f32 __builtin_msa_fexdo_w (v2f64, v2f64);

v4f32 __builtin_msa_fexp2_w (v4f32, v4i32);
v2f64 __builtin_msa_fexp2_d (v2f64, v2i64);

v4f32 __builtin_msa_fexupl_w (v8i16);
v2f64 __builtin_msa_fexupl_d (v4f32);

v4f32 __builtin_msa_fexupr_w (v8i16);
v2f64 __builtin_msa_fexupr_d (v4f32);

v4f32 __builtin_msa_ffint_s_w (v4i32);
v2f64 __builtin_msa_ffint_s_d (v2i64);
```

```
v4f32 __builtin_msa_ffint_u_w (v4u32);
v2f64 __builtin_msa_ffint_u_d (v2u64);

v4f32 __builtin_msa_ffql_w (v8i16);
v2f64 __builtin_msa_ffql_d (v4i32);

v4f32 __builtin_msa_ffqr_w (v8i16);
v2f64 __builtin_msa_ffqr_d (v4i32);

v16i8 __builtin_msa_fill_b (i32);
v8i16 __builtin_msa_fill_h (i32);
v4i32 __builtin_msa_fill_w (i32);
v2i64 __builtin_msa_fill_d (i64);

v4f32 __builtin_msa_flog2_w (v4f32);
v2f64 __builtin_msa_flog2_d (v2f64);

v4f32 __builtin_msa_fmadd_w (v4f32, v4f32, v4f32);
v2f64 __builtin_msa_fmadd_d (v2f64, v2f64, v2f64);

v4f32 __builtin_msa_fmax_w (v4f32, v4f32);
v2f64 __builtin_msa_fmax_d (v2f64, v2f64);

v4f32 __builtin_msa_fmax_a_w (v4f32, v4f32);
v2f64 __builtin_msa_fmax_a_d (v2f64, v2f64);

v4f32 __builtin_msa_fmin_w (v4f32, v4f32);
v2f64 __builtin_msa_fmin_d (v2f64, v2f64);

v4f32 __builtin_msa_fmin_a_w (v4f32, v4f32);
v2f64 __builtin_msa_fmin_a_d (v2f64, v2f64);

v4f32 __builtin_msa_fmsub_w (v4f32, v4f32, v4f32);
v2f64 __builtin_msa_fmsub_d (v2f64, v2f64, v2f64);

v4f32 __builtin_msa_fmul_w (v4f32, v4f32);
v2f64 __builtin_msa_fmul_d (v2f64, v2f64);

v4f32 __builtin_msa_frint_w (v4f32);
v2f64 __builtin_msa_frint_d (v2f64);

v4f32 __builtin_msa_frcp_w (v4f32);
v2f64 __builtin_msa_frcp_d (v2f64);

v4f32 __builtin_msa_frsqrt_w (v4f32);
v2f64 __builtin_msa_frsqrt_d (v2f64);

v4i32 __builtin_msa_fsaf_w (v4f32, v4f32);
v2i64 __builtin_msa_fsaf_d (v2f64, v2f64);

v4i32 __builtin_msa_fseq_w (v4f32, v4f32);
v2i64 __builtin_msa_fseq_d (v2f64, v2f64);

v4i32 __builtin_msa_fsle_w (v4f32, v4f32);
v2i64 __builtin_msa_fsle_d (v2f64, v2f64);

v4i32 __builtin_msa_fslt_w (v4f32, v4f32);
```

```
v2i64 __builtin_msa_fslt_d (v2f64, v2f64);

v4i32 __builtin_msa_fsne_w (v4f32, v4f32);
v2i64 __builtin_msa_fsne_d (v2f64, v2f64);

v4i32 __builtin_msa_fsor_w (v4f32, v4f32);
v2i64 __builtin_msa_fsor_d (v2f64, v2f64);

v4f32 __builtin_msa_fsqrt_w (v4f32);
v2f64 __builtin_msa_fsqrt_d (v2f64);

v4f32 __builtin_msa_fsub_w (v4f32, v4f32);
v2f64 __builtin_msa_fsub_d (v2f64, v2f64);

v4i32 __builtin_msa_fsueq_w (v4f32, v4f32);
v2i64 __builtin_msa_fsueq_d (v2f64, v2f64);

v4i32 __builtin_msa_fsule_w (v4f32, v4f32);
v2i64 __builtin_msa_fsule_d (v2f64, v2f64);

v4i32 __builtin_msa_fsult_w (v4f32, v4f32);
v2i64 __builtin_msa_fsult_d (v2f64, v2f64);

v4i32 __builtin_msa_fsun_w (v4f32, v4f32);
v2i64 __builtin_msa_fsun_d (v2f64, v2f64);

v4i32 __builtin_msa_fsune_w (v4f32, v4f32);
v2i64 __builtin_msa_fsune_d (v2f64, v2f64);

v4i32 __builtin_msa_ftint_s_w (v4f32);
v2i64 __builtin_msa_ftint_s_d (v2f64);

v4u32 __builtin_msa_ftint_u_w (v4f32);
v2u64 __builtin_msa_ftint_u_d (v2f64);

v8i16 __builtin_msa_ftq_h (v4f32, v4f32);
v4i32 __builtin_msa_ftq_w (v2f64, v2f64);

v4i32 __builtin_msa_ftrunc_s_w (v4f32);
v2i64 __builtin_msa_ftrunc_s_d (v2f64);

v4u32 __builtin_msa_ftrunc_u_w (v4f32);
v2u64 __builtin_msa_ftrunc_u_d (v2f64);

v8i16 __builtin_msa_hadd_s_h (v16i8, v16i8);
v4i32 __builtin_msa_hadd_s_w (v8i16, v8i16);
v2i64 __builtin_msa_hadd_s_d (v4i32, v4i32);

v8u16 __builtin_msa_hadd_u_h (v16u8, v16u8);
v4u32 __builtin_msa_hadd_u_w (v8u16, v8u16);
v2u64 __builtin_msa_hadd_u_d (v4u32, v4u32);

v8i16 __builtin_msa_hsub_s_h (v16i8, v16i8);
v4i32 __builtin_msa_hsub_s_w (v8i16, v8i16);
v2i64 __builtin_msa_hsub_s_d (v4i32, v4i32);

v8i16 __builtin_msa_hsub_u_h (v16u8, v16u8);
v4i32 __builtin_msa_hsub_u_w (v8u16, v8u16);
```

```
v2i64 __builtin_msa_hsub_u_d (v4u32, v4u32);

v16i8 __builtin_msa_ilvev_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvev_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvev_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvev_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvl_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvl_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvl_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvl_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvod_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvod_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvod_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvod_d (v2i64, v2i64);

v16i8 __builtin_msa_ilvr_b (v16i8, v16i8);
v8i16 __builtin_msa_ilvr_h (v8i16, v8i16);
v4i32 __builtin_msa_ilvr_w (v4i32, v4i32);
v2i64 __builtin_msa_ilvr_d (v2i64, v2i64);

v16i8 __builtin_msa_insert_b (v16i8, imm0_15, i32);
v8i16 __builtin_msa_insert_h (v8i16, imm0_7, i32);
v4i32 __builtin_msa_insert_w (v4i32, imm0_3, i32);
v2i64 __builtin_msa_insert_d (v2i64, imm0_1, i64);

v16i8 __builtin_msa_insve_b (v16i8, imm0_15, v16i8);
v8i16 __builtin_msa_insve_h (v8i16, imm0_7, v8i16);
v4i32 __builtin_msa_insve_w (v4i32, imm0_3, v4i32);
v2i64 __builtin_msa_insve_d (v2i64, imm0_1, v2i64);

v16i8 __builtin_msa_ld_b (void *, imm_n512_511);
v8i16 __builtin_msa_ld_h (void *, imm_n1024_1022);
v4i32 __builtin_msa_ld_w (void *, imm_n2048_2044);
v2i64 __builtin_msa_ld_d (void *, imm_n4096_4088);

v16i8 __builtin_msa_ldi_b (imm_n512_511);
v8i16 __builtin_msa_ldi_h (imm_n512_511);
v4i32 __builtin_msa_ldi_w (imm_n512_511);
v2i64 __builtin_msa_ldi_d (imm_n512_511);

v8i16 __builtin_msa_madd_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_madd_q_w (v4i32, v4i32, v4i32);

v8i16 __builtin_msa_maddr_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_maddr_q_w (v4i32, v4i32, v4i32);

v16i8 __builtin_msa_maddv_b (v16i8, v16i8, v16i8);
v8i16 __builtin_msa_maddv_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_maddv_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_maddv_d (v2i64, v2i64, v2i64);

v16i8 __builtin_msa_max_a_b (v16i8, v16i8);
v8i16 __builtin_msa_max_a_h (v8i16, v8i16);
v4i32 __builtin_msa_max_a_w (v4i32, v4i32);
v2i64 __builtin_msa_max_a_d (v2i64, v2i64);
```

```
v16i8 __builtin_msa_max_s_b (v16i8, v16i8);
v8i16 __builtin_msa_max_s_h (v8i16, v8i16);
v4i32 __builtin_msa_max_s_w (v4i32, v4i32);
v2i64 __builtin_msa_max_s_d (v2i64, v2i64);

v16u8 __builtin_msa_max_u_b (v16u8, v16u8);
v8u16 __builtin_msa_max_u_h (v8u16, v8u16);
v4u32 __builtin_msa_max_u_w (v4u32, v4u32);
v2u64 __builtin_msa_max_u_d (v2u64, v2u64);

v16i8 __builtin_msa_maxi_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_maxi_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_maxi_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_maxi_s_d (v2i64, imm_n16_15);

v16u8 __builtin_msa_maxi_u_b (v16u8, imm0_31);
v8u16 __builtin_msa_maxi_u_h (v8u16, imm0_31);
v4u32 __builtin_msa_maxi_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_maxi_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_min_a_b (v16i8, v16i8);
v8i16 __builtin_msa_min_a_h (v8i16, v8i16);
v4i32 __builtin_msa_min_a_w (v4i32, v4i32);
v2i64 __builtin_msa_min_a_d (v2i64, v2i64);

v16i8 __builtin_msa_min_s_b (v16i8, v16i8);
v8i16 __builtin_msa_min_s_h (v8i16, v8i16);
v4i32 __builtin_msa_min_s_w (v4i32, v4i32);
v2i64 __builtin_msa_min_s_d (v2i64, v2i64);

v16u8 __builtin_msa_min_u_b (v16u8, v16u8);
v8u16 __builtin_msa_min_u_h (v8u16, v8u16);
v4u32 __builtin_msa_min_u_w (v4u32, v4u32);
v2u64 __builtin_msa_min_u_d (v2u64, v2u64);

v16i8 __builtin_msa_mini_s_b (v16i8, imm_n16_15);
v8i16 __builtin_msa_mini_s_h (v8i16, imm_n16_15);
v4i32 __builtin_msa_mini_s_w (v4i32, imm_n16_15);
v2i64 __builtin_msa_mini_s_d (v2i64, imm_n16_15);

v16u8 __builtin_msa_mini_u_b (v16u8, imm0_31);
v8u16 __builtin_msa_mini_u_h (v8u16, imm0_31);
v4u32 __builtin_msa_mini_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_mini_u_d (v2u64, imm0_31);

v16i8 __builtin_msa_mod_s_b (v16i8, v16i8);
v8i16 __builtin_msa_mod_s_h (v8i16, v8i16);
v4i32 __builtin_msa_mod_s_w (v4i32, v4i32);
v2i64 __builtin_msa_mod_s_d (v2i64, v2i64);

v16u8 __builtin_msa_mod_u_b (v16u8, v16u8);
v8u16 __builtin_msa_mod_u_h (v8u16, v8u16);
v4u32 __builtin_msa_mod_u_w (v4u32, v4u32);
v2u64 __builtin_msa_mod_u_d (v2u64, v2u64);

v16i8 __builtin_msa_move_v (v16i8);

v8i16 __builtin_msa_msub_q_h (v8i16, v8i16, v8i16);
```

```
v4i32 __builtin_msa_msub_q_w (v4i32, v4i32, v4i32);

v8i16 __builtin_msa_msubr_q_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_msubr_q_w (v4i32, v4i32, v4i32);

v16i8 __builtin_msa_msubv_b (v16i8, v16i8, v16i8);
v8i16 __builtin_msa_msubv_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_msubv_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_msubv_d (v2i64, v2i64, v2i64);

v8i16 __builtin_msa_mul_q_h (v8i16, v8i16);
v4i32 __builtin_msa_mul_q_w (v4i32, v4i32);

v8i16 __builtin_msa_mulr_q_h (v8i16, v8i16);
v4i32 __builtin_msa_mulr_q_w (v4i32, v4i32);

v16i8 __builtin_msa_mulv_b (v16i8, v16i8);
v8i16 __builtin_msa_mulv_h (v8i16, v8i16);
v4i32 __builtin_msa_mulv_w (v4i32, v4i32);
v2i64 __builtin_msa_mulv_d (v2i64, v2i64);

v16i8 __builtin_msa_nloc_b (v16i8);
v8i16 __builtin_msa_nloc_h (v8i16);
v4i32 __builtin_msa_nloc_w (v4i32);
v2i64 __builtin_msa_nloc_d (v2i64);

v16i8 __builtin_msa_nlzc_b (v16i8);
v8i16 __builtin_msa_nlzc_h (v8i16);
v4i32 __builtin_msa_nlzc_w (v4i32);
v2i64 __builtin_msa_nlzc_d (v2i64);

v16u8 __builtin_msa_nor_v (v16u8, v16u8);

v16u8 __builtin_msa_nori_b (v16u8, imm0_255);

v16u8 __builtin_msa_or_v (v16u8, v16u8);

v16u8 __builtin_msa_ori_b (v16u8, imm0_255);

v16i8 __builtin_msa_pckev_b (v16i8, v16i8);
v8i16 __builtin_msa_pckev_h (v8i16, v8i16);
v4i32 __builtin_msa_pckev_w (v4i32, v4i32);
v2i64 __builtin_msa_pckev_d (v2i64, v2i64);

v16i8 __builtin_msa_pckod_b (v16i8, v16i8);
v8i16 __builtin_msa_pckod_h (v8i16, v8i16);
v4i32 __builtin_msa_pckod_w (v4i32, v4i32);
v2i64 __builtin_msa_pckod_d (v2i64, v2i64);

v16i8 __builtin_msa_pcnt_b (v16i8);
v8i16 __builtin_msa_pcnt_h (v8i16);
v4i32 __builtin_msa_pcnt_w (v4i32);
v2i64 __builtin_msa_pcnt_d (v2i64);

v16i8 __builtin_msa_sat_s_b (v16i8, imm0_7);
v8i16 __builtin_msa_sat_s_h (v8i16, imm0_15);
v4i32 __builtin_msa_sat_s_w (v4i32, imm0_31);
v2i64 __builtin_msa_sat_s_d (v2i64, imm0_63);
```

```
v16u8 __builtin_msa_sat_u_b (v16u8, imm0_7);
v8u16 __builtin_msa_sat_u_h (v8u16, imm0_15);
v4u32 __builtin_msa_sat_u_w (v4u32, imm0_31);
v2u64 __builtin_msa_sat_u_d (v2u64, imm0_63);

v16i8 __builtin_msa_shf_b (v16i8, imm0_255);
v8i16 __builtin_msa_shf_h (v8i16, imm0_255);
v4i32 __builtin_msa_shf_w (v4i32, imm0_255);

v16i8 __builtin_msa_sld_b (v16i8, v16i8, i32);
v8i16 __builtin_msa_sld_h (v8i16, v8i16, i32);
v4i32 __builtin_msa_sld_w (v4i32, v4i32, i32);
v2i64 __builtin_msa_sld_d (v2i64, v2i64, i32);

v16i8 __builtin_msa_sldi_b (v16i8, v16i8, imm0_15);
v8i16 __builtin_msa_sldi_h (v8i16, v8i16, imm0_7);
v4i32 __builtin_msa_sldi_w (v4i32, v4i32, imm0_3);
v2i64 __builtin_msa_sldi_d (v2i64, v2i64, imm0_1);

v16i8 __builtin_msa_sll_b (v16i8, v16i8);
v8i16 __builtin_msa_sll_h (v8i16, v8i16);
v4i32 __builtin_msa_sll_w (v4i32, v4i32);
v2i64 __builtin_msa_sll_d (v2i64, v2i64);

v16i8 __builtin_msa_slli_b (v16i8, imm0_7);
v8i16 __builtin_msa_slli_h (v8i16, imm0_15);
v4i32 __builtin_msa_slli_w (v4i32, imm0_31);
v2i64 __builtin_msa_slli_d (v2i64, imm0_63);

v16i8 __builtin_msa_splat_b (v16i8, i32);
v8i16 __builtin_msa_splat_h (v8i16, i32);
v4i32 __builtin_msa_splat_w (v4i32, i32);
v2i64 __builtin_msa_splat_d (v2i64, i32);

v16i8 __builtin_msa_splati_b (v16i8, imm0_15);
v8i16 __builtin_msa_splati_h (v8i16, imm0_7);
v4i32 __builtin_msa_splati_w (v4i32, imm0_3);
v2i64 __builtin_msa_splati_d (v2i64, imm0_1);

v16i8 __builtin_msa_sra_b (v16i8, v16i8);
v8i16 __builtin_msa_sra_h (v8i16, v8i16);
v4i32 __builtin_msa_sra_w (v4i32, v4i32);
v2i64 __builtin_msa_sra_d (v2i64, v2i64);

v16i8 __builtin_msa_srai_b (v16i8, imm0_7);
v8i16 __builtin_msa_srai_h (v8i16, imm0_15);
v4i32 __builtin_msa_srai_w (v4i32, imm0_31);
v2i64 __builtin_msa_srai_d (v2i64, imm0_63);

v16i8 __builtin_msa_srar_b (v16i8, v16i8);
v8i16 __builtin_msa_srar_h (v8i16, v8i16);
v4i32 __builtin_msa_srar_w (v4i32, v4i32);
v2i64 __builtin_msa_srar_d (v2i64, v2i64);

v16i8 __builtin_msa_srari_b (v16i8, imm0_7);
v8i16 __builtin_msa_srari_h (v8i16, imm0_15);
v4i32 __builtin_msa_srari_w (v4i32, imm0_31);
```

```
v2i64 __builtin_msa_srari_d (v2i64, imm0_63);

v16i8 __builtin_msa_srl_b (v16i8, v16i8);
v8i16 __builtin_msa_srl_h (v8i16, v8i16);
v4i32 __builtin_msa_srl_w (v4i32, v4i32);
v2i64 __builtin_msa_srl_d (v2i64, v2i64);

v16i8 __builtin_msa_srli_b (v16i8, imm0_7);
v8i16 __builtin_msa_srli_h (v8i16, imm0_15);
v4i32 __builtin_msa_srli_w (v4i32, imm0_31);
v2i64 __builtin_msa_srli_d (v2i64, imm0_63);

v16i8 __builtin_msa_srlr_b (v16i8, v16i8);
v8i16 __builtin_msa_srlr_h (v8i16, v8i16);
v4i32 __builtin_msa_srlr_w (v4i32, v4i32);
v2i64 __builtin_msa_srlr_d (v2i64, v2i64);

v16i8 __builtin_msa_srlri_b (v16i8, imm0_7);
v8i16 __builtin_msa_srlri_h (v8i16, imm0_15);
v4i32 __builtin_msa_srlri_w (v4i32, imm0_31);
v2i64 __builtin_msa_srlri_d (v2i64, imm0_63);

void __builtin_msa_st_b (v16i8, void *, imm_n512_511);
void __builtin_msa_st_h (v8i16, void *, imm_n1024_1022);
void __builtin_msa_st_w (v4i32, void *, imm_n2048_2044);
void __builtin_msa_st_d (v2i64, void *, imm_n4096_4088);

v16i8 __builtin_msa_subs_s_b (v16i8, v16i8);
v8i16 __builtin_msa_subs_s_h (v8i16, v8i16);
v4i32 __builtin_msa_subs_s_w (v4i32, v4i32);
v2i64 __builtin_msa_subs_s_d (v2i64, v2i64);

v16u8 __builtin_msa_subs_u_b (v16u8, v16u8);
v8u16 __builtin_msa_subs_u_h (v8u16, v8u16);
v4u32 __builtin_msa_subs_u_w (v4u32, v4u32);
v2u64 __builtin_msa_subs_u_d (v2u64, v2u64);

v16u8 __builtin_msa_subsus_u_b (v16u8, v16i8);
v8u16 __builtin_msa_subsus_u_h (v8u16, v8i16);
v4u32 __builtin_msa_subsus_u_w (v4u32, v4i32);
v2u64 __builtin_msa_subsus_u_d (v2u64, v2i64);

v16i8 __builtin_msa_subsuu_s_b (v16u8, v16u8);
v8i16 __builtin_msa_subsuu_s_h (v8u16, v8u16);
v4i32 __builtin_msa_subsuu_s_w (v4u32, v4u32);
v2i64 __builtin_msa_subsuu_s_d (v2u64, v2u64);

v16i8 __builtin_msa_subv_b (v16i8, v16i8);
v8i16 __builtin_msa_subv_h (v8i16, v8i16);
v4i32 __builtin_msa_subv_w (v4i32, v4i32);
v2i64 __builtin_msa_subv_d (v2i64, v2i64);

v16i8 __builtin_msa_subvi_b (v16i8, imm0_31);
v8i16 __builtin_msa_subvi_h (v8i16, imm0_31);
v4i32 __builtin_msa_subvi_w (v4i32, imm0_31);
v2i64 __builtin_msa_subvi_d (v2i64, imm0_31);

v16i8 __builtin_msa_vshf_b (v16i8, v16i8, v16i8);
```

```
v8i16 __builtin_msa_vshf_h (v8i16, v8i16, v8i16);
v4i32 __builtin_msa_vshf_w (v4i32, v4i32, v4i32);
v2i64 __builtin_msa_vshf_d (v2i64, v2i64, v2i64);

v16u8 __builtin_msa_xor_v (v16u8, v16u8);

v16u8 __builtin_msa_xori_b (v16u8, imm0_255);
```

6.60.17 Other MIPS Built-in Functions

GCC provides other MIPS-specific built-in functions:

`void __builtin_mips_cache (int op, const volatile void *addr)`
: Insert a '`cache`' instruction with operands *op* and *addr*. GCC defines the preprocessor macro `___GCC_HAVE_BUILTIN_MIPS_CACHE` when this function is available.

`unsigned int __builtin_mips_get_fcsr (void)`
`void __builtin_mips_set_fcsr (unsigned int value)`
: Get and set the contents of the floating-point control and status register (FPU control register 31). These functions are only available in hard-float code but can be called in both MIPS16 and non-MIPS16 contexts.

  `__builtin_mips_set_fcsr` can be used to change any bit of the register except the condition codes, which GCC assumes are preserved.

6.60.18 MSP430 Built-in Functions

GCC provides a couple of special builtin functions to aid in the writing of interrupt handlers in C.

`__bic_SR_register_on_exit (int mask)`
: This clears the indicated bits in the saved copy of the status register currently residing on the stack. This only works inside interrupt handlers and the changes to the status register will only take affect once the handler returns.

`__bis_SR_register_on_exit (int mask)`
: This sets the indicated bits in the saved copy of the status register currently residing on the stack. This only works inside interrupt handlers and the changes to the status register will only take affect once the handler returns.

`__delay_cycles (long long cycles)`
: This inserts an instruction sequence that takes exactly *cycles* cycles (between 0 and about 17E9) to complete. The inserted sequence may use jumps, loops, or no-ops, and does not interfere with any other instructions. Note that *cycles* must be a compile-time constant integer - that is, you must pass a number, not a variable that may be optimized to a constant later. The number of cycles delayed by this builtin is exact.

6.60.19 NDS32 Built-in Functions

These built-in functions are available for the NDS32 target:

`void __builtin_nds32_isync (int *addr)` [Built-in Function]
: Insert an ISYNC instruction into the instruction stream where *addr* is an instruction address for serialization.

`void __builtin_nds32_isb (void)` [Built-in Function]
Insert an ISB instruction into the instruction stream.

`int __builtin_nds32_mfsr (int sr)` [Built-in Function]
Return the content of a system register which is mapped by *sr*.

`int __builtin_nds32_mfusr (int usr)` [Built-in Function]
Return the content of a user space register which is mapped by *usr*.

`void __builtin_nds32_mtsr (int value, int sr)` [Built-in Function]
Move the *value* to a system register which is mapped by *sr*.

`void __builtin_nds32_mtusr (int value, int usr)` [Built-in Function]
Move the *value* to a user space register which is mapped by *usr*.

`void __builtin_nds32_setgie_en (void)` [Built-in Function]
Enable global interrupt.

`void __builtin_nds32_setgie_dis (void)` [Built-in Function]
Disable global interrupt.

6.60.20 picoChip Built-in Functions

GCC provides an interface to selected machine instructions from the picoChip instruction set.

`int __builtin_sbc (int value)`
Sign bit count. Return the number of consecutive bits in *value* that have the same value as the sign bit. The result is the number of leading sign bits minus one, giving the number of redundant sign bits in *value*.

`int __builtin_byteswap (int value)`
Byte swap. Return the result of swapping the upper and lower bytes of *value*.

`int __builtin_brev (int value)`
Bit reversal. Return the result of reversing the bits in *value*. Bit 15 is swapped with bit 0, bit 14 is swapped with bit 1, and so on.

`int __builtin_adds (int x, int y)`
Saturating addition. Return the result of adding x and y, storing the value 32767 if the result overflows.

`int __builtin_subs (int x, int y)`
Saturating subtraction. Return the result of subtracting y from x, storing the value -32768 if the result overflows.

`void __builtin_halt (void)`
Halt. The processor stops execution. This built-in is useful for implementing assertions.

6.60.21 Basic PowerPC Built-in Functions

This section describes PowerPC built-in functions that do not require the inclusion of any special header files to declare prototypes or provide macro definitions. The sections that follow describe additional PowerPC built-in functions.

6.60.21.1 Basic PowerPC Built-in Functions Available on all Configurations

`void __builtin_cpu_init` (*void*) [Built-in Function]

This function is a `nop` on the PowerPC platform and is included solely to maintain API compatibility with the x86 builtins.

`int __builtin_cpu_is` (*const char* ***cpuname***) [Built-in Function]

This function returns a value of `1` if the run-time CPU is of type *cpuname* and returns `0` otherwise

The `__builtin_cpu_is` function requires GLIBC 2.23 or newer which exports the hardware capability bits. GCC defines the macro `__BUILTIN_CPU_SUPPORTS__` if the `__builtin_cpu_supports` built-in function is fully supported.

If GCC was configured to use a GLIBC before 2.23, the built-in function `__builtin_cpu_is` always returns a 0 and the compiler issues a warning.

The following CPU names can be detected:

'`power9`' IBM POWER9 Server CPU.

'`power8`' IBM POWER8 Server CPU.

'`power7`' IBM POWER7 Server CPU.

'`power6x`' IBM POWER6 Server CPU (RAW mode).

'`power6`' IBM POWER6 Server CPU (Architected mode).

'`power5+`' IBM POWER5+ Server CPU.

'`power5`' IBM POWER5 Server CPU.

'`ppc970`' IBM 970 Server CPU (ie, Apple G5).

'`power4`' IBM POWER4 Server CPU.

'`ppca2`' IBM A2 64-bit Embedded CPU

'`ppc476`' IBM PowerPC 476FP 32-bit Embedded CPU.

'`ppc464`' IBM PowerPC 464 32-bit Embedded CPU.

'`ppc440`' PowerPC 440 32-bit Embedded CPU.

'`ppc405`' PowerPC 405 32-bit Embedded CPU.

'`ppc-cell-be`'
IBM PowerPC Cell Broadband Engine Architecture CPU.

Here is an example:

```
#ifdef __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_is ("power8"))
    {
       do_power8 (); // POWER8 specific implementation.
    }
  else
#endif
    {
       do_generic (); // Generic implementation.
    }
```

`int __builtin_cpu_supports (const char *feature)` [Built-in Function]

This function returns a value of `1` if the run-time CPU supports the HWCAP feature *feature* and returns `0` otherwise.

The `__builtin_cpu_supports` function requires GLIBC 2.23 or newer which exports the hardware capability bits. GCC defines the macro `__BUILTIN_CPU_SUPPORTS__` if the `__builtin_cpu_supports` built-in function is fully supported.

If GCC was configured to use a GLIBC before 2.23, the built-in function `__builtin_cpu_suports` always returns a 0 and the compiler issues a warning.

The following features can be detected:

'`4xxmac`' 4xx CPU has a Multiply Accumulator.

'`altivec`' CPU has a SIMD/Vector Unit.

'`arch_2_05`'
CPU supports ISA 2.05 (eg, POWER6)

'`arch_2_06`'
CPU supports ISA 2.06 (eg, POWER7)

'`arch_2_07`'
CPU supports ISA 2.07 (eg, POWER8)

'`arch_3_00`'
CPU supports ISA 3.0 (eg, POWER9)

'`archpmu`' CPU supports the set of compatible performance monitoring events.

'`booke`' CPU supports the Embedded ISA category.

'`cellbe`' CPU has a CELL broadband engine.

'`darn`' CPU supports the `darn` (deliver a random number) instruction.

'`dfp`' CPU has a decimal floating point unit.

'`dscr`' CPU supports the data stream control register.

'`ebb`' CPU supports event base branching.

'`efpdouble`'
CPU has a SPE double precision floating point unit.

'`efpsingle`'
CPU has a SPE single precision floating point unit.

'`fpu`' CPU has a floating point unit.

'`htm`' CPU has hardware transaction memory instructions.

'`htm-nosc`'
Kernel aborts hardware transactions when a syscall is made.

'`htm-no-suspend`'
CPU supports hardware transaction memory but does not support the `tsuspend.` instruction.

'ic_snoop'
: CPU supports icache snooping capabilities.

'ieee128' CPU supports 128-bit IEEE binary floating point instructions.

'isel' CPU supports the integer select instruction.

'mmu' CPU has a memory management unit.

'notb' CPU does not have a timebase (eg, 601 and 403gx).

'pa6t' CPU supports the PA Semi 6T CORE ISA.

'power4' CPU supports ISA 2.00 (eg, POWER4)

'power5' CPU supports ISA 2.02 (eg, POWER5)

'power5+' CPU supports ISA 2.03 (eg, POWER5+)

'power6x' CPU supports ISA 2.05 (eg, POWER6) extended opcodes mffgpr and mftgpr.

'ppc32' CPU supports 32-bit mode execution.

'ppc601' CPU supports the old POWER ISA (eg, 601)

'ppc64' CPU supports 64-bit mode execution.

'ppcle' CPU supports a little-endian mode that uses address swizzling.

'scv' Kernel supports system call vectored.

'smt' CPU support simultaneous multi-threading.

'spe' CPU has a signal processing extension unit.

'tar' CPU supports the target address register.

'true_le' CPU supports true little-endian mode.

'ucache' CPU has unified I/D cache.

'vcrypto' CPU supports the vector cryptography instructions.

'vsx' CPU supports the vector-scalar extension.

Here is an example:

```
#ifdef __BUILTIN_CPU_SUPPORTS__
  if (__builtin_cpu_supports ("fpu"))
    {
       asm("fadd %0,%1,%2" : "=d"(dst) : "d"(src1), "d"(src2));
    }
  else
#endif
    {
       dst = __fadd (src1, src2); // Software FP addition function.
    }
```

The following built-in functions are also available on all PowerPC processors:

```
uint64_t __builtin_ppc_get_timebase ();
unsigned long __builtin_ppc_mftb ();
double __builtin_unpack_ibm128 (__ibm128, int);
__ibm128 __builtin_pack_ibm128 (double, double);
double __builtin_mffs (void);
void __builtin_mtfsb0 (const int);
void __builtin_mtfsb1 (const int);
void __builtin_set_fpscr_rn (int);
```

The `__builtin_ppc_get_timebase` and `__builtin_ppc_mftb` functions generate instructions to read the Time Base Register. The `__builtin_ppc_get_timebase` function may generate multiple instructions and always returns the 64 bits of the Time Base Register. The `__builtin_ppc_mftb` function always generates one instruction and returns the Time Base Register value as an unsigned long, throwing away the most significant word on 32-bit environments. The `__builtin_mffs` return the value of the FPSCR register. Note, ISA 3.0 supports the `__builtin_mffsl()` which permits software to read the control and non-sticky status bits in the FSPCR without the higher latency associated with accessing the sticky status bits. The `__builtin_mtfsb0` and `__builtin_mtfsb1` take the bit to change as an argument. The valid bit range is between 0 and 31. The builtins map to the `mtfsb0` and `mtfsb1` instructions which take the argument and add 32. Hence these instructions only modify the FPSCR[32:63] bits by changing the specified bit to a zero or one respectively. The `__builtin_set_fpscr_rn` builtin allows changing both of the floating point rounding mode bits. The argument is a 2-bit value. The argument can either be a `const int` or stored in a variable. The builtin uses the ISA 3.0 instruction `mffscrn` if available, otherwise it reads the FPSCR, masks the current rounding mode bits out and OR's in the new value.

6.60.21.2 Basic PowerPC Built-in Functions Available on ISA 2.05

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.05 or later. Unless specific options are explicitly disabled on the command line, specifying option '`-mcpu=power6`' has the effect of enabling the '`-mpowerpc64`', '`-mpowerpc-gpopt`', '`-mpowerpc-gfxopt`', '`-mmfcrf`', '`-mpopcntb`', '`-mfprnd`', '`-mcmpb`', '`-mhard-dfp`', and '`-mrecip-precision`' options. Specify the '`-maltivec`' and '`-mfpgpr`' options explicitly in combination with the above options if they are desired.

The following functions require option '`-mcmpb`'.

```
unsigned long long __builtin_cmpb (unsigned long long int, unsigned long long int);
unsigned int __builtin_cmpb (unsigned int, unsigned int);
```

The `__builtin_cmpb` function performs a byte-wise compare on the contents of its two arguments, returning the result of the byte-wise comparison as the returned value. For each byte comparison, the corresponding byte of the return value holds 0xff if the input bytes are equal and 0 if the input bytes are not equal. If either of the arguments to this built-in function is wider than 32 bits, the function call expands into the form that expects `unsigned long long int` arguments which is only available on 64-bit targets.

The following built-in functions are available when hardware decimal floating point ('`-mhard-dfp`') is available:

```
void __builtin_set_fpscr_drn(int);
_Decimal64 __builtin_ddedpd (int, _Decimal64);
_Decimal128 __builtin_ddedpdq (int, _Decimal128);
```

```
_Decimal64 __builtin_denbcd (int, _Decimal64);
_Decimal128 __builtin_denbcdq (int, _Decimal128);
_Decimal64 __builtin_diex (long long, _Decimal64);
_Decimal128 _builtin_diexq (long long, _Decimal128);
_Decimal64 __builtin_dscli (_Decimal64, int);
_Decimal128 __builtin_dscliq (_Decimal128, int);
_Decimal64 __builtin_dscri (_Decimal64, int);
_Decimal128 __builtin_dscriq (_Decimal128, int);
long long __builtin_dxex (_Decimal64);
long long __builtin_dxexq (_Decimal128);
_Decimal128 __builtin_pack_dec128 (unsigned long long, unsigned long long);
unsigned long long __builtin_unpack_dec128 (_Decimal128, int);

The __builtin_set_fpscr_drn builtin allows changing the three decimal
floating point rounding mode bits.  The argument is a 3-bit value.  The
argument can either be a const int or the value can be stored in
a variable.
The builtin uses the ISA 3.0 instruction mffscdrn if available.
Otherwise the builtin reads the FPSCR, masks the current decimal rounding
mode bits out and OR's in the new value.
```

The following functions require '`-mhard-float`', '`-mpowerpc-gfxopt`', and '`-mpopcntb`' options.

```
double __builtin_recipdiv (double, double);
float __builtin_recipdivf (float, float);
double __builtin_rsqrt (double);
float __builtin_rsqrtf (float);
```

The `vec_rsqrt`, `__builtin_rsqrt`, and `__builtin_rsqrtf` functions generate multiple instructions to implement the reciprocal sqrt functionality using reciprocal sqrt estimate instructions.

The `__builtin_recipdiv`, and `__builtin_recipdivf` functions generate multiple instructions to implement division using the reciprocal estimate instructions.

The following functions require '`-mhard-float`' and '`-mmultiple`' options.

The `__builtin_unpack_longdouble` function takes a `long double` argument and a compile time constant of 0 or 1. If the constant is 0, the first `double` within the `long double` is returned, otherwise the second `double` is returned. The `__builtin_unpack_longdouble` function is only available if `long double` uses the IBM extended double representation.

The `__builtin_pack_longdouble` function takes two `double` arguments and returns a `long double` value that combines the two arguments. The `__builtin_pack_longdouble` function is only available if `long double` uses the IBM extended double representation.

The `__builtin_unpack_ibm128` function takes a `__ibm128` argument and a compile time constant of 0 or 1. If the constant is 0, the first `double` within the `__ibm128` is returned, otherwise the second `double` is returned.

The `__builtin_pack_ibm128` function takes two `double` arguments and returns a `__ibm128` value that combines the two arguments.

Additional built-in functions are available for the 64-bit PowerPC family of processors, for efficient use of 128-bit floating point (`__float128`) values.

6.60.21.3 Basic PowerPC Built-in Functions Available on ISA 2.06

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.05 or later. Unless specific options are explicitly disabled on the command line, specifying option '-mcpu=power7' has the effect of enabling all the same options as for '-mcpu=power6' in addition to the '-maltivec', '-mpopcntd', and '-mvsx' options.

The following basic built-in functions require '-mpopcntd':

```
unsigned int __builtin_addg6s (unsigned int, unsigned int);
long long __builtin_bpermd (long long, long long);
unsigned int __builtin_cbcdtd (unsigned int);
unsigned int __builtin_cdtbcd (unsigned int);
long long __builtin_divde (long long, long long);
unsigned long long __builtin_divdeu (unsigned long long, unsigned long long);
int __builtin_divwe (int, int);
unsigned int __builtin_divweu (unsigned int, unsigned int);
vector __int128 __builtin_pack_vector_int128 (long long, long long);
void __builtin_rs6000_speculation_barrier (void);
long long __builtin_unpack_vector_int128 (vector __int128, signed char);
```

Of these, the `__builtin_divde` and `__builtin_divdeu` functions require a 64-bit environment.

The following basic built-in functions, which are also supported on x86 targets, require '-mfloat128'.

```
__float128 __builtin_fabsq (__float128);
__float128 __builtin_copysignq (__float128, __float128);
__float128 __builtin_infq (void);
__float128 __builtin_huge_valq (void);
__float128 __builtin_nanq (void);
__float128 __builtin_nansq (void);

__float128 __builtin_sqrtf128 (__float128);
__float128 __builtin_fmaf128 (__float128, __float128, __float128);
```

6.60.21.4 Basic PowerPC Built-in Functions Available on ISA 2.07

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.07 or later. Unless specific options are explicitly disabled on the command line, specifying option '-mcpu=power8' has the effect of enabling all the same options as for '-mcpu=power7' in addition to the '-mpower8-fusion', '-mpower8-vector', '-mcrypto', '-mhtm', '-mquad-memory', and '-mquad-memory-atomic' options.

This section intentionally empty.

6.60.21.5 Basic PowerPC Built-in Functions Available on ISA 3.0

The basic built-in functions described in this section are available on the PowerPC family of processors starting with ISA 3.0 or later. Unless specific options are explicitly disabled on the command line, specifying option '-mcpu=power9' has the effect of enabling all the same options as for '-mcpu=power8' in addition to the '-misel' option.

The following built-in functions are available on Linux 64-bit systems that use the ISA 3.0 instruction set ('-mcpu=power9'):

`__float128 __builtin_addf128_round_to_odd (__float128, __float128)`
Perform a 128-bit IEEE floating point add using round to odd as the rounding mode.

`__float128 __builtin_subf128_round_to_odd (__float128, __float128)`
Perform a 128-bit IEEE floating point subtract using round to odd as the rounding mode.

`__float128 __builtin_mulf128_round_to_odd (__float128, __float128)`
Perform a 128-bit IEEE floating point multiply using round to odd as the rounding mode.

`__float128 __builtin_divf128_round_to_odd (__float128, __float128)`
Perform a 128-bit IEEE floating point divide using round to odd as the rounding mode.

`__float128 __builtin_sqrtf128_round_to_odd (__float128)`
Perform a 128-bit IEEE floating point square root using round to odd as the rounding mode.

`__float128 __builtin_fmaf128_round_to_odd (__float128, __float128, __float128)`
Perform a 128-bit IEEE floating point fused multiply and add operation using round to odd as the rounding mode.

`double __builtin_truncf128_round_to_odd (__float128)`
Convert a 128-bit IEEE floating point value to `double` using round to odd as the rounding mode.

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later:

```
long long __builtin_darn (void);
long long __builtin_darn_raw (void);
int __builtin_darn_32 (void);
```

The `__builtin_darn` and `__builtin_darn_raw` functions require a 64-bit environment supporting ISA 3.0 or later. The `__builtin_darn` function provides a 64-bit conditioned random number. The `__builtin_darn_raw` function provides a 64-bit raw random number. The `__builtin_darn_32` function provides a 32-bit conditioned random number.

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later:

```
int __builtin_byte_in_set (unsigned char u, unsigned long long set);
int __builtin_byte_in_range (unsigned char u, unsigned int range);
int __builtin_byte_in_either_range (unsigned char u, unsigned int ranges);

int __builtin_dfp_dtstsfi_lt (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_lt (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_lt_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_lt_td (unsigned int comparison, _Decimal128 value);

int __builtin_dfp_dtstsfi_gt (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_gt (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_gt_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_gt_td (unsigned int comparison, _Decimal128 value);
```

```
int __builtin_dfp_dtstsfi_eq (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_eq (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_eq_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_eq_td (unsigned int comparison, _Decimal128 value);

int __builtin_dfp_dtstsfi_ov (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_ov (unsigned int comparison, _Decimal128 value);
int __builtin_dfp_dtstsfi_ov_dd (unsigned int comparison, _Decimal64 value);
int __builtin_dfp_dtstsfi_ov_td (unsigned int comparison, _Decimal128 value);

double __builtin_mffsl(void);
```

The `__builtin_byte_in_set` function requires a 64-bit environment supporting ISA 3.0 or later. This function returns a non-zero value if and only if its `u` argument exactly equals one of the eight bytes contained within its 64-bit `set` argument.

The `__builtin_byte_in_range` and `__builtin_byte_in_either_range` require an environment supporting ISA 3.0 or later. For these two functions, the `range` argument is encoded as 4 bytes, organized as `hi_1:lo_1:hi_2:lo_2`. The `__builtin_byte_in_range` function returns a non-zero value if and only if its `u` argument is within the range bounded between `lo_2` and `hi_2` inclusive. The `__builtin_byte_in_either_range` function returns non-zero if and only if its `u` argument is within either the range bounded between `lo_1` and `hi_1` inclusive or the range bounded between `lo_2` and `hi_2` inclusive.

The `__builtin_dfp_dtstsfi_lt` function returns a non-zero value if and only if the number of signficant digits of its `value` argument is less than its `comparison` argument. The `__builtin_dfp_dtstsfi_lt_dd` and `__builtin_dfp_dtstsfi_lt_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_gt` function returns a non-zero value if and only if the number of signficant digits of its `value` argument is greater than its `comparison` argument. The `__builtin_dfp_dtstsfi_gt_dd` and `__builtin_dfp_dtstsfi_gt_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_eq` function returns a non-zero value if and only if the number of signficant digits of its `value` argument equals its `comparison` argument. The `__builtin_dfp_dtstsfi_eq_dd` and `__builtin_dfp_dtstsfi_eq_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_dfp_dtstsfi_ov` function returns a non-zero value if and only if its `value` argument has an undefined number of significant digits, such as when `value` is an encoding of `NaN`. The `__builtin_dfp_dtstsfi_ov_dd` and `__builtin_dfp_dtstsfi_ov_td` functions behave similarly, but require that the type of the `value` argument be `__Decimal64` and `__Decimal128` respectively.

The `__builtin_mffsl` uses the ISA 3.0 `mffsl` instruction to read the FPSCR. The instruction is a lower latency version of the `mffs` instruction. If the `mffsl` instruction is not available, then the builtin uses the older `mffs` instruction to read the FPSCR.

6.60.22 PowerPC AltiVec/VSX Built-in Functions

GCC provides an interface for the PowerPC family of processors to access the AltiVec operations described in Motorola's AltiVec Programming Interface Manual. The interface is made available by including `<altivec.h>` and using '`-maltivec`' and '`-mabi=altivec`'. The interface supports the following vector types.

```
vector unsigned char
vector signed char
vector bool char

vector unsigned short
vector signed short
vector bool short
vector pixel

vector unsigned int
vector signed int
vector bool int
vector float
```

GCC's implementation of the high-level language interface available from C and C++ code differs from Motorola's documentation in several ways.

- A vector constant is a list of constant expressions within curly braces.
- A vector initializer requires no cast if the vector constant is of the same type as the variable it is initializing.
- If `signed` or `unsigned` is omitted, the signedness of the vector type is the default signedness of the base type. The default varies depending on the operating system, so a portable program should always specify the signedness.
- Compiling with '`-maltivec`' adds keywords `__vector`, `vector`, `__pixel`, `pixel`, `__bool` and `bool`. When compiling ISO C, the context-sensitive substitution of the keywords `vector`, `pixel` and `bool` is disabled. To use them, you must include `<altivec.h>` instead.
- GCC allows using a `typedef` name as the type specifier for a vector type, but only under the following circumstances:
  - When using `__vector` instead of `vector`; for example,

    ```
    typedef signed short int16;
    __vector int16 data;
    ```

  - When using `vector` in keyword-and-predefine mode; for example,

    ```
    typedef signed short int16;
    vector int16 data;
    ```

    Note that keyword-and-predefine mode is enabled by disabling GNU extensions (e.g., by using `-std=c11`) and including `<altivec.h>`.
- For C, overloaded functions are implemented with macros so the following does not work:

  ```
  vec_add ((vector signed int){1, 2, 3, 4}, foo);
  ```

  Since `vec_add` is a macro, the vector constant in the example is treated as four separate arguments. Wrap the entire argument in parentheses for this to work.

Note: Only the `<altivec.h>` interface is supported. Internally, GCC uses built-in functions to achieve the functionality in the aforementioned header file, but they are not supported and are subject to change without notice.

GCC complies with the OpenPOWER 64-Bit ELF V2 ABI Specification, which may be found at `http://openpowerfoundation.org/wp-content/uploads/resources/leabi-prd/content/index.html`. Appendix A of this document lists the vector API interfaces that must be provided by compliant compilers. Programmers should preferentially use the interfaces described therein. However, historically GCC has provided additional interfaces for access to vector instructions. These are briefly described below.

6.60.22.1 PowerPC AltiVec Built-in Functions on ISA 2.05

The following interfaces are supported for the generic and specific AltiVec operations and the AltiVec predicates. In cases where there is a direct mapping between generic and specific operations, only the generic names are shown here, although the specific operations can also be used.

Arguments that are documented as `const int` require literal integral values within the range required for that operation.

```
vector signed char vec_abs (vector signed char);
vector signed short vec_abs (vector signed short);
vector signed int vec_abs (vector signed int);
vector float vec_abs (vector float);

vector signed char vec_abss (vector signed char);
vector signed short vec_abss (vector signed short);
vector signed int vec_abss (vector signed int);

vector signed char vec_add (vector bool char, vector signed char);
vector signed char vec_add (vector signed char, vector bool char);
vector signed char vec_add (vector signed char, vector signed char);
vector unsigned char vec_add (vector bool char, vector unsigned char);
vector unsigned char vec_add (vector unsigned char, vector bool char);
vector unsigned char vec_add (vector unsigned char, vector unsigned char);
vector signed short vec_add (vector bool short, vector signed short);
vector signed short vec_add (vector signed short, vector bool short);
vector signed short vec_add (vector signed short, vector signed short);
vector unsigned short vec_add (vector bool short, vector unsigned short);
vector unsigned short vec_add (vector unsigned short, vector bool short);
vector unsigned short vec_add (vector unsigned short, vector unsigned short);
vector signed int vec_add (vector bool int, vector signed int);
vector signed int vec_add (vector signed int, vector bool int);
vector signed int vec_add (vector signed int, vector signed int);
vector unsigned int vec_add (vector bool int, vector unsigned int);
vector unsigned int vec_add (vector unsigned int, vector bool int);
vector unsigned int vec_add (vector unsigned int, vector unsigned int);
vector float vec_add (vector float, vector float);

vector unsigned int vec_addc (vector unsigned int, vector unsigned int);

vector unsigned char vec_adds (vector bool char, vector unsigned char);
vector unsigned char vec_adds (vector unsigned char, vector bool char);
vector unsigned char vec_adds (vector unsigned char, vector unsigned char);
vector signed char vec_adds (vector bool char, vector signed char);
vector signed char vec_adds (vector signed char, vector bool char);
vector signed char vec_adds (vector signed char, vector signed char);
```

```
vector unsigned short vec_adds (vector bool short, vector unsigned short);
vector unsigned short vec_adds (vector unsigned short, vector bool short);
vector unsigned short vec_adds (vector unsigned short, vector unsigned short);
vector signed short vec_adds (vector bool short, vector signed short);
vector signed short vec_adds (vector signed short, vector bool short);
vector signed short vec_adds (vector signed short, vector signed short);
vector unsigned int vec_adds (vector bool int, vector unsigned int);
vector unsigned int vec_adds (vector unsigned int, vector bool int);
vector unsigned int vec_adds (vector unsigned int, vector unsigned int);
vector signed int vec_adds (vector bool int, vector signed int);
vector signed int vec_adds (vector signed int, vector bool int);
vector signed int vec_adds (vector signed int, vector signed int);

int vec_all_eq (vector signed char, vector bool char);
int vec_all_eq (vector signed char, vector signed char);
int vec_all_eq (vector unsigned char, vector bool char);
int vec_all_eq (vector unsigned char, vector unsigned char);
int vec_all_eq (vector bool char, vector bool char);
int vec_all_eq (vector bool char, vector unsigned char);
int vec_all_eq (vector bool char, vector signed char);
int vec_all_eq (vector signed short, vector bool short);
int vec_all_eq (vector signed short, vector signed short);
int vec_all_eq (vector unsigned short, vector bool short);
int vec_all_eq (vector unsigned short, vector unsigned short);
int vec_all_eq (vector bool short, vector bool short);
int vec_all_eq (vector bool short, vector unsigned short);
int vec_all_eq (vector bool short, vector signed short);
int vec_all_eq (vector pixel, vector pixel);
int vec_all_eq (vector signed int, vector bool int);
int vec_all_eq (vector signed int, vector signed int);
int vec_all_eq (vector unsigned int, vector bool int);
int vec_all_eq (vector unsigned int, vector unsigned int);
int vec_all_eq (vector bool int, vector bool int);
int vec_all_eq (vector bool int, vector unsigned int);
int vec_all_eq (vector bool int, vector signed int);
int vec_all_eq (vector float, vector float);

int vec_all_ge (vector bool char, vector unsigned char);
int vec_all_ge (vector unsigned char, vector bool char);
int vec_all_ge (vector unsigned char, vector unsigned char);
int vec_all_ge (vector bool char, vector signed char);
int vec_all_ge (vector signed char, vector bool char);
int vec_all_ge (vector signed char, vector signed char);
int vec_all_ge (vector bool short, vector unsigned short);
int vec_all_ge (vector unsigned short, vector bool short);
int vec_all_ge (vector unsigned short, vector unsigned short);
int vec_all_ge (vector signed short, vector signed short);
int vec_all_ge (vector bool short, vector signed short);
int vec_all_ge (vector signed short, vector bool short);
int vec_all_ge (vector bool int, vector unsigned int);
int vec_all_ge (vector unsigned int, vector bool int);
int vec_all_ge (vector unsigned int, vector unsigned int);
int vec_all_ge (vector bool int, vector signed int);
int vec_all_ge (vector signed int, vector bool int);
int vec_all_ge (vector signed int, vector signed int);
int vec_all_ge (vector float, vector float);

int vec_all_gt (vector bool char, vector unsigned char);
```

```
int vec_all_gt (vector unsigned char, vector bool char);
int vec_all_gt (vector unsigned char, vector unsigned char);
int vec_all_gt (vector bool char, vector signed char);
int vec_all_gt (vector signed char, vector bool char);
int vec_all_gt (vector signed char, vector signed char);
int vec_all_gt (vector bool short, vector unsigned short);
int vec_all_gt (vector unsigned short, vector bool short);
int vec_all_gt (vector unsigned short, vector unsigned short);
int vec_all_gt (vector bool short, vector signed short);
int vec_all_gt (vector signed short, vector bool short);
int vec_all_gt (vector signed short, vector signed short);
int vec_all_gt (vector bool int, vector unsigned int);
int vec_all_gt (vector unsigned int, vector bool int);
int vec_all_gt (vector unsigned int, vector unsigned int);
int vec_all_gt (vector bool int, vector signed int);
int vec_all_gt (vector signed int, vector bool int);
int vec_all_gt (vector signed int, vector signed int);
int vec_all_gt (vector float, vector float);

int vec_all_in (vector float, vector float);

int vec_all_le (vector bool char, vector unsigned char);
int vec_all_le (vector unsigned char, vector bool char);
int vec_all_le (vector unsigned char, vector unsigned char);
int vec_all_le (vector bool char, vector signed char);
int vec_all_le (vector signed char, vector bool char);
int vec_all_le (vector signed char, vector signed char);
int vec_all_le (vector bool short, vector unsigned short);
int vec_all_le (vector unsigned short, vector bool short);
int vec_all_le (vector unsigned short, vector unsigned short);
int vec_all_le (vector bool short, vector signed short);
int vec_all_le (vector signed short, vector bool short);
int vec_all_le (vector signed short, vector signed short);
int vec_all_le (vector bool int, vector unsigned int);
int vec_all_le (vector unsigned int, vector bool int);
int vec_all_le (vector unsigned int, vector unsigned int);
int vec_all_le (vector bool int, vector signed int);
int vec_all_le (vector signed int, vector bool int);
int vec_all_le (vector signed int, vector signed int);
int vec_all_le (vector float, vector float);

int vec_all_lt (vector bool char, vector unsigned char);
int vec_all_lt (vector unsigned char, vector bool char);
int vec_all_lt (vector unsigned char, vector unsigned char);
int vec_all_lt (vector bool char, vector signed char);
int vec_all_lt (vector signed char, vector bool char);
int vec_all_lt (vector signed char, vector signed char);
int vec_all_lt (vector bool short, vector unsigned short);
int vec_all_lt (vector unsigned short, vector bool short);
int vec_all_lt (vector unsigned short, vector unsigned short);
int vec_all_lt (vector bool short, vector signed short);
int vec_all_lt (vector signed short, vector bool short);
int vec_all_lt (vector signed short, vector signed short);
int vec_all_lt (vector bool int, vector unsigned int);
int vec_all_lt (vector unsigned int, vector bool int);
int vec_all_lt (vector unsigned int, vector unsigned int);
int vec_all_lt (vector bool int, vector signed int);
int vec_all_lt (vector signed int, vector bool int);
```

```
int vec_all_lt (vector signed int, vector signed int);
int vec_all_lt (vector float, vector float);

int vec_all_nan (vector float);

int vec_all_ne (vector signed char, vector bool char);
int vec_all_ne (vector signed char, vector signed char);
int vec_all_ne (vector unsigned char, vector bool char);
int vec_all_ne (vector unsigned char, vector unsigned char);
int vec_all_ne (vector bool char, vector bool char);
int vec_all_ne (vector bool char, vector unsigned char);
int vec_all_ne (vector bool char, vector signed char);
int vec_all_ne (vector signed short, vector bool short);
int vec_all_ne (vector signed short, vector signed short);
int vec_all_ne (vector unsigned short, vector bool short);
int vec_all_ne (vector unsigned short, vector unsigned short);
int vec_all_ne (vector bool short, vector bool short);
int vec_all_ne (vector bool short, vector unsigned short);
int vec_all_ne (vector bool short, vector signed short);
int vec_all_ne (vector pixel, vector pixel);
int vec_all_ne (vector signed int, vector bool int);
int vec_all_ne (vector signed int, vector signed int);
int vec_all_ne (vector unsigned int, vector bool int);
int vec_all_ne (vector unsigned int, vector unsigned int);
int vec_all_ne (vector bool int, vector bool int);
int vec_all_ne (vector bool int, vector unsigned int);
int vec_all_ne (vector bool int, vector signed int);
int vec_all_ne (vector float, vector float);

int vec_all_nge (vector float, vector float);

int vec_all_ngt (vector float, vector float);

int vec_all_nle (vector float, vector float);

int vec_all_nlt (vector float, vector float);

int vec_all_numeric (vector float);

vector float vec_and (vector float, vector float);
vector float vec_and (vector float, vector bool int);
vector float vec_and (vector bool int, vector float);
vector bool int vec_and (vector bool int, vector bool int);
vector signed int vec_and (vector bool int, vector signed int);
vector signed int vec_and (vector signed int, vector bool int);
vector signed int vec_and (vector signed int, vector signed int);
vector unsigned int vec_and (vector bool int, vector unsigned int);
vector unsigned int vec_and (vector unsigned int, vector bool int);
vector unsigned int vec_and (vector unsigned int, vector unsigned int);
vector bool short vec_and (vector bool short, vector bool short);
vector signed short vec_and (vector bool short, vector signed short);
vector signed short vec_and (vector signed short, vector bool short);
vector signed short vec_and (vector signed short, vector signed short);
vector unsigned short vec_and (vector bool short, vector unsigned short);
vector unsigned short vec_and (vector unsigned short, vector bool short);
vector unsigned short vec_and (vector unsigned short, vector unsigned short);
vector signed char vec_and (vector bool char, vector signed char);
vector bool char vec_and (vector bool char, vector bool char);
```

```
vector signed char vec_and (vector signed char, vector bool char);
vector signed char vec_and (vector signed char, vector signed char);
vector unsigned char vec_and (vector bool char, vector unsigned char);
vector unsigned char vec_and (vector unsigned char, vector bool char);
vector unsigned char vec_and (vector unsigned char, vector unsigned char);

vector float vec_andc (vector float, vector float);
vector float vec_andc (vector float, vector bool int);
vector float vec_andc (vector bool int, vector float);
vector bool int vec_andc (vector bool int, vector bool int);
vector signed int vec_andc (vector bool int, vector signed int);
vector signed int vec_andc (vector signed int, vector bool int);
vector signed int vec_andc (vector signed int, vector signed int);
vector unsigned int vec_andc (vector bool int, vector unsigned int);
vector unsigned int vec_andc (vector unsigned int, vector bool int);
vector unsigned int vec_andc (vector unsigned int, vector unsigned int);
vector bool short vec_andc (vector bool short, vector bool short);
vector signed short vec_andc (vector bool short, vector signed short);
vector signed short vec_andc (vector signed short, vector bool short);
vector signed short vec_andc (vector signed short, vector signed short);
vector unsigned short vec_andc (vector bool short, vector unsigned short);
vector unsigned short vec_andc (vector unsigned short, vector bool short);
vector unsigned short vec_andc (vector unsigned short, vector unsigned short);
vector signed char vec_andc (vector bool char, vector signed char);
vector bool char vec_andc (vector bool char, vector bool char);
vector signed char vec_andc (vector signed char, vector bool char);
vector signed char vec_andc (vector signed char, vector signed char);
vector unsigned char vec_andc (vector bool char, vector unsigned char);
vector unsigned char vec_andc (vector unsigned char, vector bool char);
vector unsigned char vec_andc (vector unsigned char, vector unsigned char);

int vec_any_eq (vector signed char, vector bool char);
int vec_any_eq (vector signed char, vector signed char);
int vec_any_eq (vector unsigned char, vector bool char);
int vec_any_eq (vector unsigned char, vector unsigned char);
int vec_any_eq (vector bool char, vector bool char);
int vec_any_eq (vector bool char, vector unsigned char);
int vec_any_eq (vector bool char, vector signed char);
int vec_any_eq (vector signed short, vector bool short);
int vec_any_eq (vector signed short, vector signed short);
int vec_any_eq (vector unsigned short, vector bool short);
int vec_any_eq (vector unsigned short, vector unsigned short);
int vec_any_eq (vector bool short, vector bool short);
int vec_any_eq (vector bool short, vector unsigned short);
int vec_any_eq (vector bool short, vector signed short);
int vec_any_eq (vector pixel, vector pixel);
int vec_any_eq (vector signed int, vector bool int);
int vec_any_eq (vector signed int, vector signed int);
int vec_any_eq (vector unsigned int, vector bool int);
int vec_any_eq (vector unsigned int, vector unsigned int);
int vec_any_eq (vector bool int, vector bool int);
int vec_any_eq (vector bool int, vector unsigned int);
int vec_any_eq (vector bool int, vector signed int);
int vec_any_eq (vector float, vector float);

int vec_any_ge (vector signed char, vector bool char);
int vec_any_ge (vector unsigned char, vector bool char);
int vec_any_ge (vector unsigned char, vector unsigned char);
```

```
int vec_any_ge (vector signed char, vector signed char);
int vec_any_ge (vector bool char, vector unsigned char);
int vec_any_ge (vector bool char, vector signed char);
int vec_any_ge (vector unsigned short, vector bool short);
int vec_any_ge (vector unsigned short, vector unsigned short);
int vec_any_ge (vector signed short, vector signed short);
int vec_any_ge (vector signed short, vector bool short);
int vec_any_ge (vector bool short, vector unsigned short);
int vec_any_ge (vector bool short, vector signed short);
int vec_any_ge (vector signed int, vector bool int);
int vec_any_ge (vector unsigned int, vector bool int);
int vec_any_ge (vector unsigned int, vector unsigned int);
int vec_any_ge (vector signed int, vector signed int);
int vec_any_ge (vector bool int, vector unsigned int);
int vec_any_ge (vector bool int, vector signed int);
int vec_any_ge (vector float, vector float);

int vec_any_gt (vector bool char, vector unsigned char);
int vec_any_gt (vector unsigned char, vector bool char);
int vec_any_gt (vector unsigned char, vector unsigned char);
int vec_any_gt (vector bool char, vector signed char);
int vec_any_gt (vector signed char, vector bool char);
int vec_any_gt (vector signed char, vector signed char);
int vec_any_gt (vector bool short, vector unsigned short);
int vec_any_gt (vector unsigned short, vector bool short);
int vec_any_gt (vector unsigned short, vector unsigned short);
int vec_any_gt (vector bool short, vector signed short);
int vec_any_gt (vector signed short, vector bool short);
int vec_any_gt (vector signed short, vector signed short);
int vec_any_gt (vector bool int, vector unsigned int);
int vec_any_gt (vector unsigned int, vector bool int);
int vec_any_gt (vector unsigned int, vector unsigned int);
int vec_any_gt (vector bool int, vector signed int);
int vec_any_gt (vector signed int, vector bool int);
int vec_any_gt (vector signed int, vector signed int);
int vec_any_gt (vector float, vector float);

int vec_any_le (vector bool char, vector unsigned char);
int vec_any_le (vector unsigned char, vector bool char);
int vec_any_le (vector unsigned char, vector unsigned char);
int vec_any_le (vector bool char, vector signed char);
int vec_any_le (vector signed char, vector bool char);
int vec_any_le (vector signed char, vector signed char);
int vec_any_le (vector bool short, vector unsigned short);
int vec_any_le (vector unsigned short, vector bool short);
int vec_any_le (vector unsigned short, vector unsigned short);
int vec_any_le (vector bool short, vector signed short);
int vec_any_le (vector signed short, vector bool short);
int vec_any_le (vector signed short, vector signed short);
int vec_any_le (vector bool int, vector unsigned int);
int vec_any_le (vector unsigned int, vector bool int);
int vec_any_le (vector unsigned int, vector unsigned int);
int vec_any_le (vector bool int, vector signed int);
int vec_any_le (vector signed int, vector bool int);
int vec_any_le (vector signed int, vector signed int);
int vec_any_le (vector float, vector float);

int vec_any_lt (vector bool char, vector unsigned char);
```

```
int vec_any_lt (vector unsigned char, vector bool char);
int vec_any_lt (vector unsigned char, vector unsigned char);
int vec_any_lt (vector bool char, vector signed char);
int vec_any_lt (vector signed char, vector bool char);
int vec_any_lt (vector signed char, vector signed char);
int vec_any_lt (vector bool short, vector unsigned short);
int vec_any_lt (vector unsigned short, vector bool short);
int vec_any_lt (vector unsigned short, vector unsigned short);
int vec_any_lt (vector bool short, vector signed short);
int vec_any_lt (vector signed short, vector bool short);
int vec_any_lt (vector signed short, vector signed short);
int vec_any_lt (vector bool int, vector unsigned int);
int vec_any_lt (vector unsigned int, vector bool int);
int vec_any_lt (vector unsigned int, vector unsigned int);
int vec_any_lt (vector bool int, vector signed int);
int vec_any_lt (vector signed int, vector bool int);
int vec_any_lt (vector signed int, vector signed int);
int vec_any_lt (vector float, vector float);

int vec_any_nan (vector float);

int vec_any_ne (vector signed char, vector bool char);
int vec_any_ne (vector signed char, vector signed char);
int vec_any_ne (vector unsigned char, vector bool char);
int vec_any_ne (vector unsigned char, vector unsigned char);
int vec_any_ne (vector bool char, vector bool char);
int vec_any_ne (vector bool char, vector unsigned char);
int vec_any_ne (vector bool char, vector signed char);
int vec_any_ne (vector signed short, vector bool short);
int vec_any_ne (vector signed short, vector signed short);
int vec_any_ne (vector unsigned short, vector bool short);
int vec_any_ne (vector unsigned short, vector unsigned short);
int vec_any_ne (vector bool short, vector bool short);
int vec_any_ne (vector bool short, vector unsigned short);
int vec_any_ne (vector bool short, vector signed short);
int vec_any_ne (vector pixel, vector pixel);
int vec_any_ne (vector signed int, vector bool int);
int vec_any_ne (vector signed int, vector signed int);
int vec_any_ne (vector unsigned int, vector bool int);
int vec_any_ne (vector unsigned int, vector unsigned int);
int vec_any_ne (vector bool int, vector bool int);
int vec_any_ne (vector bool int, vector unsigned int);
int vec_any_ne (vector bool int, vector signed int);
int vec_any_ne (vector float, vector float);

int vec_any_nge (vector float, vector float);

int vec_any_ngt (vector float, vector float);

int vec_any_nle (vector float, vector float);

int vec_any_nlt (vector float, vector float);

int vec_any_numeric (vector float);

int vec_any_out (vector float, vector float);

vector unsigned char vec_avg (vector unsigned char, vector unsigned char);
```

```
vector signed char vec_avg (vector signed char, vector signed char);
vector unsigned short vec_avg (vector unsigned short, vector unsigned short);
vector signed short vec_avg (vector signed short, vector signed short);
vector unsigned int vec_avg (vector unsigned int, vector unsigned int);
vector signed int vec_avg (vector signed int, vector signed int);

vector float vec_ceil (vector float);

vector signed int vec_cmpb (vector float, vector float);

vector bool char vec_cmpeq (vector bool char, vector bool char);
vector bool short vec_cmpeq (vector bool short, vector bool short);
vector bool int vec_cmpeq (vector bool int, vector bool int);
vector bool char vec_cmpeq (vector signed char, vector signed char);
vector bool char vec_cmpeq (vector unsigned char, vector unsigned char);
vector bool short vec_cmpeq (vector signed short, vector signed short);
vector bool short vec_cmpeq (vector unsigned short, vector unsigned short);
vector bool int vec_cmpeq (vector signed int, vector signed int);
vector bool int vec_cmpeq (vector unsigned int, vector unsigned int);
vector bool int vec_cmpeq (vector float, vector float);

vector bool int vec_cmpge (vector float, vector float);

vector bool char vec_cmpgt (vector unsigned char, vector unsigned char);
vector bool char vec_cmpgt (vector signed char, vector signed char);
vector bool short vec_cmpgt (vector unsigned short, vector unsigned short);
vector bool short vec_cmpgt (vector signed short, vector signed short);
vector bool int vec_cmpgt (vector unsigned int, vector unsigned int);
vector bool int vec_cmpgt (vector signed int, vector signed int);
vector bool int vec_cmpgt (vector float, vector float);

vector bool int vec_cmple (vector float, vector float);

vector bool char vec_cmplt (vector unsigned char, vector unsigned char);
vector bool char vec_cmplt (vector signed char, vector signed char);
vector bool short vec_cmplt (vector unsigned short, vector unsigned short);
vector bool short vec_cmplt (vector signed short, vector signed short);
vector bool int vec_cmplt (vector unsigned int, vector unsigned int);
vector bool int vec_cmplt (vector signed int, vector signed int);
vector bool int vec_cmplt (vector float, vector float);

vector float vec_cpsgn (vector float, vector float);

vector float vec_ctf (vector unsigned int, const int);
vector float vec_ctf (vector signed int, const int);

vector signed int vec_cts (vector float, const int);

vector unsigned int vec_ctu (vector float, const int);

void vec_dss (const int);

void vec_dssall (void);

void vec_dst (const vector unsigned char *, int, const int);
void vec_dst (const vector signed char *, int, const int);
void vec_dst (const vector bool char *, int, const int);
void vec_dst (const vector unsigned short *, int, const int);
```

```
void vec_dst (const vector signed short *, int, const int);
void vec_dst (const vector bool short *, int, const int);
void vec_dst (const vector pixel *, int, const int);
void vec_dst (const vector unsigned int *, int, const int);
void vec_dst (const vector signed int *, int, const int);
void vec_dst (const vector bool int *, int, const int);
void vec_dst (const vector float *, int, const int);
void vec_dst (const unsigned char *, int, const int);
void vec_dst (const signed char *, int, const int);
void vec_dst (const unsigned short *, int, const int);
void vec_dst (const short *, int, const int);
void vec_dst (const unsigned int *, int, const int);
void vec_dst (const int *, int, const int);
void vec_dst (const float *, int, const int);

void vec_dstst (const vector unsigned char *, int, const int);
void vec_dstst (const vector signed char *, int, const int);
void vec_dstst (const vector bool char *, int, const int);
void vec_dstst (const vector unsigned short *, int, const int);
void vec_dstst (const vector signed short *, int, const int);
void vec_dstst (const vector bool short *, int, const int);
void vec_dstst (const vector pixel *, int, const int);
void vec_dstst (const vector unsigned int *, int, const int);
void vec_dstst (const vector signed int *, int, const int);
void vec_dstst (const vector bool int *, int, const int);
void vec_dstst (const vector float *, int, const int);
void vec_dstst (const unsigned char *, int, const int);
void vec_dstst (const signed char *, int, const int);
void vec_dstst (const unsigned short *, int, const int);
void vec_dstst (const short *, int, const int);
void vec_dstst (const unsigned int *, int, const int);
void vec_dstst (const int *, int, const int);
void vec_dstst (const unsigned long *, int, const int);
void vec_dstst (const long *, int, const int);
void vec_dstst (const float *, int, const int);

void vec_dststt (const vector unsigned char *, int, const int);
void vec_dststt (const vector signed char *, int, const int);
void vec_dststt (const vector bool char *, int, const int);
void vec_dststt (const vector unsigned short *, int, const int);
void vec_dststt (const vector signed short *, int, const int);
void vec_dststt (const vector bool short *, int, const int);
void vec_dststt (const vector pixel *, int, const int);
void vec_dststt (const vector unsigned int *, int, const int);
void vec_dststt (const vector signed int *, int, const int);
void vec_dststt (const vector bool int *, int, const int);
void vec_dststt (const vector float *, int, const int);
void vec_dststt (const unsigned char *, int, const int);
void vec_dststt (const signed char *, int, const int);
void vec_dststt (const unsigned short *, int, const int);
void vec_dststt (const short *, int, const int);
void vec_dststt (const unsigned int *, int, const int);
void vec_dststt (const int *, int, const int);
void vec_dststt (const float *, int, const int);

void vec_dstt (const vector unsigned char *, int, const int);
void vec_dstt (const vector signed char *, int, const int);
void vec_dstt (const vector bool char *, int, const int);
```

```
void vec_dstt (const vector unsigned short *, int, const int);
void vec_dstt (const vector signed short *, int, const int);
void vec_dstt (const vector bool short *, int, const int);
void vec_dstt (const vector pixel *, int, const int);
void vec_dstt (const vector unsigned int *, int, const int);
void vec_dstt (const vector signed int *, int, const int);
void vec_dstt (const vector bool int *, int, const int);
void vec_dstt (const vector float *, int, const int);
void vec_dstt (const unsigned char *, int, const int);
void vec_dstt (const signed char *, int, const int);
void vec_dstt (const unsigned short *, int, const int);
void vec_dstt (const short *, int, const int);
void vec_dstt (const unsigned int *, int, const int);
void vec_dstt (const int *, int, const int);
void vec_dstt (const float *, int, const int);

vector float vec_expte (vector float);

vector float vec_floor (vector float);

vector float vec_ld (int, const vector float *);
vector float vec_ld (int, const float *);
vector bool int vec_ld (int, const vector bool int *);
vector signed int vec_ld (int, const vector signed int *);
vector signed int vec_ld (int, const int *);
vector unsigned int vec_ld (int, const vector unsigned int *);
vector unsigned int vec_ld (int, const unsigned int *);
vector bool short vec_ld (int, const vector bool short *);
vector pixel vec_ld (int, const vector pixel *);
vector signed short vec_ld (int, const vector signed short *);
vector signed short vec_ld (int, const short *);
vector unsigned short vec_ld (int, const vector unsigned short *);
vector unsigned short vec_ld (int, const unsigned short *);
vector bool char vec_ld (int, const vector bool char *);
vector signed char vec_ld (int, const vector signed char *);
vector signed char vec_ld (int, const signed char *);
vector unsigned char vec_ld (int, const vector unsigned char *);
vector unsigned char vec_ld (int, const unsigned char *);

vector signed char vec_lde (int, const signed char *);
vector unsigned char vec_lde (int, const unsigned char *);
vector signed short vec_lde (int, const short *);
vector unsigned short vec_lde (int, const unsigned short *);
vector float vec_lde (int, const float *);
vector signed int vec_lde (int, const int *);
vector unsigned int vec_lde (int, const unsigned int *);

vector float vec_ldl (int, const vector float *);
vector float vec_ldl (int, const float *);
vector bool int vec_ldl (int, const vector bool int *);
vector signed int vec_ldl (int, const vector signed int *);
vector signed int vec_ldl (int, const int *);
vector unsigned int vec_ldl (int, const vector unsigned int *);
vector unsigned int vec_ldl (int, const unsigned int *);
vector bool short vec_ldl (int, const vector bool short *);
vector pixel vec_ldl (int, const vector pixel *);
vector signed short vec_ldl (int, const vector signed short *);
vector signed short vec_ldl (int, const short *);
```

```
vector unsigned short vec_ldl (int, const vector unsigned short *);
vector unsigned short vec_ldl (int, const unsigned short *);
vector bool char vec_ldl (int, const vector bool char *);
vector signed char vec_ldl (int, const vector signed char *);
vector signed char vec_ldl (int, const signed char *);
vector unsigned char vec_ldl (int, const vector unsigned char *);
vector unsigned char vec_ldl (int, const unsigned char *);

vector float vec_loge (vector float);

vector signed char vec_lvebx (int, char *);
vector unsigned char vec_lvebx (int, unsigned char *);

vector signed short vec_lvehx (int, short *);
vector unsigned short vec_lvehx (int, unsigned short *);

vector float vec_lvewx (int, float *);
vector signed int vec_lvewx (int, int *);
vector unsigned int vec_lvewx (int, unsigned int *);

vector unsigned char vec_lvsl (int, const unsigned char *);
vector unsigned char vec_lvsl (int, const signed char *);
vector unsigned char vec_lvsl (int, const unsigned short *);
vector unsigned char vec_lvsl (int, const short *);
vector unsigned char vec_lvsl (int, const unsigned int *);
vector unsigned char vec_lvsl (int, const int *);
vector unsigned char vec_lvsl (int, const float *);

vector unsigned char vec_lvsr (int, const unsigned char *);
vector unsigned char vec_lvsr (int, const signed char *);
vector unsigned char vec_lvsr (int, const unsigned short *);
vector unsigned char vec_lvsr (int, const short *);
vector unsigned char vec_lvsr (int, const unsigned int *);
vector unsigned char vec_lvsr (int, const int *);
vector unsigned char vec_lvsr (int, const float *);

vector float vec_madd (vector float, vector float, vector float);

vector signed short vec_madds (vector signed short, vector signed short,
                               vector signed short);

vector unsigned char vec_max (vector bool char, vector unsigned char);
vector unsigned char vec_max (vector unsigned char, vector bool char);
vector unsigned char vec_max (vector unsigned char, vector unsigned char);
vector signed char vec_max (vector bool char, vector signed char);
vector signed char vec_max (vector signed char, vector bool char);
vector signed char vec_max (vector signed char, vector signed char);
vector unsigned short vec_max (vector bool short, vector unsigned short);
vector unsigned short vec_max (vector unsigned short, vector bool short);
vector unsigned short vec_max (vector unsigned short, vector unsigned short);
vector signed short vec_max (vector bool short, vector signed short);
vector signed short vec_max (vector signed short, vector bool short);
vector signed short vec_max (vector signed short, vector signed short);
vector unsigned int vec_max (vector bool int, vector unsigned int);
vector unsigned int vec_max (vector unsigned int, vector bool int);
vector unsigned int vec_max (vector unsigned int, vector unsigned int);
vector signed int vec_max (vector bool int, vector signed int);
vector signed int vec_max (vector signed int, vector bool int);
```

```
vector signed int vec_max (vector signed int, vector signed int);
vector float vec_max (vector float, vector float);

vector bool char vec_mergeh (vector bool char, vector bool char);
vector signed char vec_mergeh (vector signed char, vector signed char);
vector unsigned char vec_mergeh (vector unsigned char, vector unsigned char);
vector bool short vec_mergeh (vector bool short, vector bool short);
vector pixel vec_mergeh (vector pixel, vector pixel);
vector signed short vec_mergeh (vector signed short, vector signed short);
vector unsigned short vec_mergeh (vector unsigned short, vector unsigned short);
vector float vec_mergeh (vector float, vector float);
vector bool int vec_mergeh (vector bool int, vector bool int);
vector signed int vec_mergeh (vector signed int, vector signed int);
vector unsigned int vec_mergeh (vector unsigned int, vector unsigned int);

vector bool char vec_mergel (vector bool char, vector bool char);
vector signed char vec_mergel (vector signed char, vector signed char);
vector unsigned char vec_mergel (vector unsigned char, vector unsigned char);
vector bool short vec_mergel (vector bool short, vector bool short);
vector pixel vec_mergel (vector pixel, vector pixel);
vector signed short vec_mergel (vector signed short, vector signed short);
vector unsigned short vec_mergel (vector unsigned short, vector unsigned short);
vector float vec_mergel (vector float, vector float);
vector bool int vec_mergel (vector bool int, vector bool int);
vector signed int vec_mergel (vector signed int, vector signed int);
vector unsigned int vec_mergel (vector unsigned int, vector unsigned int);

vector unsigned short vec_mfvscr (void);

vector unsigned char vec_min (vector bool char, vector unsigned char);
vector unsigned char vec_min (vector unsigned char, vector bool char);
vector unsigned char vec_min (vector unsigned char, vector unsigned char);
vector signed char vec_min (vector bool char, vector signed char);
vector signed char vec_min (vector signed char, vector bool char);
vector signed char vec_min (vector signed char, vector signed char);
vector unsigned short vec_min (vector bool short, vector unsigned short);
vector unsigned short vec_min (vector unsigned short, vector bool short);
vector unsigned short vec_min (vector unsigned short, vector unsigned short);
vector signed short vec_min (vector bool short, vector signed short);
vector signed short vec_min (vector signed short, vector bool short);
vector signed short vec_min (vector signed short, vector signed short);
vector unsigned int vec_min (vector bool int, vector unsigned int);
vector unsigned int vec_min (vector unsigned int, vector bool int);
vector unsigned int vec_min (vector unsigned int, vector unsigned int);
vector signed int vec_min (vector bool int, vector signed int);
vector signed int vec_min (vector signed int, vector bool int);
vector signed int vec_min (vector signed int, vector signed int);
vector float vec_min (vector float, vector float);

vector signed short vec_mladd (vector signed short, vector signed short,
                               vector signed short);
vector signed short vec_mladd (vector signed short, vector unsigned short,
                               vector unsigned short);
vector signed short vec_mladd (vector unsigned short, vector signed short,
                               vector signed short);
vector unsigned short vec_mladd (vector unsigned short, vector unsigned short,
                                 vector unsigned short);
```

```
vector signed short vec_mradds (vector signed short, vector signed short,
                                vector signed short);

vector unsigned int vec_msum (vector unsigned char, vector unsigned char,
                              vector unsigned int);
vector signed int vec_msum (vector signed char, vector unsigned char,
                            vector signed int);
vector unsigned int vec_msum (vector unsigned short, vector unsigned short,
                              vector unsigned int);
vector signed int vec_msum (vector signed short, vector signed short,
                            vector signed int);

vector unsigned int vec_msums (vector unsigned short, vector unsigned short,
                               vector unsigned int);
vector signed int vec_msums (vector signed short, vector signed short,
                             vector signed int);

void vec_mtvscr (vector signed int);
void vec_mtvscr (vector unsigned int);
void vec_mtvscr (vector bool int);
void vec_mtvscr (vector signed short);
void vec_mtvscr (vector unsigned short);
void vec_mtvscr (vector bool short);
void vec_mtvscr (vector pixel);
void vec_mtvscr (vector signed char);
void vec_mtvscr (vector unsigned char);
void vec_mtvscr (vector bool char);

vector float vec_mul (vector float, vector float);

vector unsigned short vec_mule (vector unsigned char, vector unsigned char);
vector signed short vec_mule (vector signed char, vector signed char);
vector unsigned int vec_mule (vector unsigned short, vector unsigned short);
vector signed int vec_mule (vector signed short, vector signed short);

vector unsigned short vec_mulo (vector unsigned char, vector unsigned char);
vector signed short vec_mulo (vector signed char, vector signed char);
vector unsigned int vec_mulo (vector unsigned short, vector unsigned short);
vector signed int vec_mulo (vector signed short, vector signed short);

vector signed char vec_nabs (vector signed char);
vector signed short vec_nabs (vector signed short);
vector signed int vec_nabs (vector signed int);
vector float vec_nabs (vector float);

vector float vec_nmsub (vector float, vector float, vector float);

vector float vec_nor (vector float, vector float);
vector signed int vec_nor (vector signed int, vector signed int);
vector unsigned int vec_nor (vector unsigned int, vector unsigned int);
vector bool int vec_nor (vector bool int, vector bool int);
vector signed short vec_nor (vector signed short, vector signed short);
vector unsigned short vec_nor (vector unsigned short, vector unsigned short);
vector bool short vec_nor (vector bool short, vector bool short);
vector signed char vec_nor (vector signed char, vector signed char);
vector unsigned char vec_nor (vector unsigned char, vector unsigned char);
vector bool char vec_nor (vector bool char, vector bool char);
```

```
vector float vec_or (vector float, vector float);
vector float vec_or (vector float, vector bool int);
vector float vec_or (vector bool int, vector float);
vector bool int vec_or (vector bool int, vector bool int);
vector signed int vec_or (vector bool int, vector signed int);
vector signed int vec_or (vector signed int, vector bool int);
vector signed int vec_or (vector signed int, vector signed int);
vector unsigned int vec_or (vector bool int, vector unsigned int);
vector unsigned int vec_or (vector unsigned int, vector bool int);
vector unsigned int vec_or (vector unsigned int, vector unsigned int);
vector bool short vec_or (vector bool short, vector bool short);
vector signed short vec_or (vector bool short, vector signed short);
vector signed short vec_or (vector signed short, vector bool short);
vector signed short vec_or (vector signed short, vector signed short);
vector unsigned short vec_or (vector bool short, vector unsigned short);
vector unsigned short vec_or (vector unsigned short, vector bool short);
vector unsigned short vec_or (vector unsigned short, vector unsigned short);
vector signed char vec_or (vector bool char, vector signed char);
vector bool char vec_or (vector bool char, vector bool char);
vector signed char vec_or (vector signed char, vector bool char);
vector signed char vec_or (vector signed char, vector signed char);
vector unsigned char vec_or (vector bool char, vector unsigned char);
vector unsigned char vec_or (vector unsigned char, vector bool char);
vector unsigned char vec_or (vector unsigned char, vector unsigned char);

vector signed char vec_pack (vector signed short, vector signed short);
vector unsigned char vec_pack (vector unsigned short, vector unsigned short);
vector bool char vec_pack (vector bool short, vector bool short);
vector signed short vec_pack (vector signed int, vector signed int);
vector unsigned short vec_pack (vector unsigned int, vector unsigned int);
vector bool short vec_pack (vector bool int, vector bool int);

vector pixel vec_packpx (vector unsigned int, vector unsigned int);

vector unsigned char vec_packs (vector unsigned short, vector unsigned short);
vector signed char vec_packs (vector signed short, vector signed short);
vector unsigned short vec_packs (vector unsigned int, vector unsigned int);
vector signed short vec_packs (vector signed int, vector signed int);

vector unsigned char vec_packsu (vector unsigned short, vector unsigned short);
vector unsigned char vec_packsu (vector signed short, vector signed short);
vector unsigned short vec_packsu (vector unsigned int, vector unsigned int);
vector unsigned short vec_packsu (vector signed int, vector signed int);

vector float vec_perm (vector float, vector float, vector unsigned char);
vector signed int vec_perm (vector signed int, vector signed int, vector unsigned char);
vector unsigned int vec_perm (vector unsigned int, vector unsigned int,
                              vector unsigned char);
vector bool int vec_perm (vector bool int, vector bool int, vector unsigned char);
vector signed short vec_perm (vector signed short, vector signed short,
                              vector unsigned char);
vector unsigned short vec_perm (vector unsigned short, vector unsigned short,
                                vector unsigned char);
vector bool short vec_perm (vector bool short, vector bool short, vector unsigned char);
vector pixel vec_perm (vector pixel, vector pixel, vector unsigned char);
vector signed char vec_perm (vector signed char, vector signed char,
                             vector unsigned char);
vector unsigned char vec_perm (vector unsigned char, vector unsigned char,
```

```
                                vector unsigned char);
vector bool char vec_perm (vector bool char, vector bool char, vector unsigned char);

vector float vec_re (vector float);

vector bool char vec_reve (vector bool char);
vector signed char vec_reve (vector signed char);
vector unsigned char vec_reve (vector unsigned char);
vector bool int vec_reve (vector bool int);
vector signed int vec_reve (vector signed int);
vector unsigned int vec_reve (vector unsigned int);
vector bool short vec_reve (vector bool short);
vector signed short vec_reve (vector signed short);
vector unsigned short vec_reve (vector unsigned short);

vector signed char vec_rl (vector signed char, vector unsigned char);
vector unsigned char vec_rl (vector unsigned char, vector unsigned char);
vector signed short vec_rl (vector signed short, vector unsigned short);
vector unsigned short vec_rl (vector unsigned short, vector unsigned short);
vector signed int vec_rl (vector signed int, vector unsigned int);
vector unsigned int vec_rl (vector unsigned int, vector unsigned int);

vector float vec_round (vector float);

vector float vec_rsqrt (vector float);

vector float vec_rsqrte (vector float);

vector float vec_sel (vector float, vector float, vector bool int);
vector float vec_sel (vector float, vector float, vector unsigned int);
vector signed int vec_sel (vector signed int, vector signed int, vector bool int);
vector signed int vec_sel (vector signed int, vector signed int, vector unsigned int);
vector unsigned int vec_sel (vector unsigned int, vector unsigned int, vector bool int);
vector unsigned int vec_sel (vector unsigned int, vector unsigned int,
                             vector unsigned int);
vector bool int vec_sel (vector bool int, vector bool int, vector bool int);
vector bool int vec_sel (vector bool int, vector bool int, vector unsigned int);
vector signed short vec_sel (vector signed short, vector signed short,
                             vector bool short);
vector signed short vec_sel (vector signed short, vector signed short,
                             vector unsigned short);
vector unsigned short vec_sel (vector unsigned short, vector unsigned short,
                               vector bool short);
vector unsigned short vec_sel (vector unsigned short, vector unsigned short,
                               vector unsigned short);
vector bool short vec_sel (vector bool short, vector bool short, vector bool short);
vector bool short vec_sel (vector bool short, vector bool short, vector unsigned short);
vector signed char vec_sel (vector signed char, vector signed char, vector bool char);
vector signed char vec_sel (vector signed char, vector signed char,
                            vector unsigned char);
vector unsigned char vec_sel (vector unsigned char, vector unsigned char,
                              vector bool char);
vector unsigned char vec_sel (vector unsigned char, vector unsigned char,
                              vector unsigned char);
vector bool char vec_sel (vector bool char, vector bool char, vector bool char);
vector bool char vec_sel (vector bool char, vector bool char, vector unsigned char);

vector signed char vec_sl (vector signed char, vector unsigned char);
```

```
vector unsigned char vec_sl (vector unsigned char, vector unsigned char);
vector signed short vec_sl (vector signed short, vector unsigned short);
vector unsigned short vec_sl (vector unsigned short, vector unsigned short);
vector signed int vec_sl (vector signed int, vector unsigned int);
vector unsigned int vec_sl (vector unsigned int, vector unsigned int);

vector float vec_sld (vector float, vector float, const int);
vector signed int vec_sld (vector signed int, vector signed int, const int);
vector unsigned int vec_sld (vector unsigned int, vector unsigned int, const int);
vector bool int vec_sld (vector bool int, vector bool int, const int);
vector signed short vec_sld (vector signed short, vector signed short, const int);
vector unsigned short vec_sld (vector unsigned short, vector unsigned short, const int);
vector bool short vec_sld (vector bool short, vector bool short, const int);
vector pixel vec_sld (vector pixel, vector pixel, const int);
vector signed char vec_sld (vector signed char, vector signed char, const int);
vector unsigned char vec_sld (vector unsigned char, vector unsigned char, const int);
vector bool char vec_sld (vector bool char, vector bool char, const int);

vector signed int vec_sll (vector signed int, vector unsigned int);
vector signed int vec_sll (vector signed int, vector unsigned short);
vector signed int vec_sll (vector signed int, vector unsigned char);
vector unsigned int vec_sll (vector unsigned int, vector unsigned int);
vector unsigned int vec_sll (vector unsigned int, vector unsigned short);
vector unsigned int vec_sll (vector unsigned int, vector unsigned char);
vector bool int vec_sll (vector bool int, vector unsigned int);
vector bool int vec_sll (vector bool int, vector unsigned short);
vector bool int vec_sll (vector bool int, vector unsigned char);
vector signed short vec_sll (vector signed short, vector unsigned int);
vector signed short vec_sll (vector signed short, vector unsigned short);
vector signed short vec_sll (vector signed short, vector unsigned char);
vector unsigned short vec_sll (vector unsigned short, vector unsigned int);
vector unsigned short vec_sll (vector unsigned short, vector unsigned short);
vector unsigned short vec_sll (vector unsigned short, vector unsigned char);
vector bool short vec_sll (vector bool short, vector unsigned int);
vector bool short vec_sll (vector bool short, vector unsigned short);
vector bool short vec_sll (vector bool short, vector unsigned char);
vector pixel vec_sll (vector pixel, vector unsigned int);
vector pixel vec_sll (vector pixel, vector unsigned short);
vector pixel vec_sll (vector pixel, vector unsigned char);
vector signed char vec_sll (vector signed char, vector unsigned int);
vector signed char vec_sll (vector signed char, vector unsigned short);
vector signed char vec_sll (vector signed char, vector unsigned char);
vector unsigned char vec_sll (vector unsigned char, vector unsigned int);
vector unsigned char vec_sll (vector unsigned char, vector unsigned short);
vector unsigned char vec_sll (vector unsigned char, vector unsigned char);
vector bool char vec_sll (vector bool char, vector unsigned int);
vector bool char vec_sll (vector bool char, vector unsigned short);
vector bool char vec_sll (vector bool char, vector unsigned char);

vector float vec_slo (vector float, vector signed char);
vector float vec_slo (vector float, vector unsigned char);
vector signed int vec_slo (vector signed int, vector signed char);
vector signed int vec_slo (vector signed int, vector unsigned char);
vector unsigned int vec_slo (vector unsigned int, vector signed char);
vector unsigned int vec_slo (vector unsigned int, vector unsigned char);
vector signed short vec_slo (vector signed short, vector signed char);
vector signed short vec_slo (vector signed short, vector unsigned char);
vector unsigned short vec_slo (vector unsigned short, vector signed char);
```

```
vector unsigned short vec_slo (vector unsigned short, vector unsigned char);
vector pixel vec_slo (vector pixel, vector signed char);
vector pixel vec_slo (vector pixel, vector unsigned char);
vector signed char vec_slo (vector signed char, vector signed char);
vector signed char vec_slo (vector signed char, vector unsigned char);
vector unsigned char vec_slo (vector unsigned char, vector signed char);
vector unsigned char vec_slo (vector unsigned char, vector unsigned char);

vector signed char vec_splat (vector signed char, const int);
vector unsigned char vec_splat (vector unsigned char, const int);
vector bool char vec_splat (vector bool char, const int);
vector signed short vec_splat (vector signed short, const int);
vector unsigned short vec_splat (vector unsigned short, const int);
vector bool short vec_splat (vector bool short, const int);
vector pixel vec_splat (vector pixel, const int);
vector float vec_splat (vector float, const int);
vector signed int vec_splat (vector signed int, const int);
vector unsigned int vec_splat (vector unsigned int, const int);
vector bool int vec_splat (vector bool int, const int);

vector signed short vec_splat_s16 (const int);

vector signed int vec_splat_s32 (const int);

vector signed char vec_splat_s8 (const int);

vector unsigned short vec_splat_u16 (const int);

vector unsigned int vec_splat_u32 (const int);

vector unsigned char vec_splat_u8 (const int);

vector signed char vec_splats (signed char);
vector unsigned char vec_splats (unsigned char);
vector signed short vec_splats (signed short);
vector unsigned short vec_splats (unsigned short);
vector signed int vec_splats (signed int);
vector unsigned int vec_splats (unsigned int);
vector float vec_splats (float);

vector signed char vec_sr (vector signed char, vector unsigned char);
vector unsigned char vec_sr (vector unsigned char, vector unsigned char);
vector signed short vec_sr (vector signed short, vector unsigned short);
vector unsigned short vec_sr (vector unsigned short, vector unsigned short);
vector signed int vec_sr (vector signed int, vector unsigned int);
vector unsigned int vec_sr (vector unsigned int, vector unsigned int);

vector signed char vec_sra (vector signed char, vector unsigned char);
vector unsigned char vec_sra (vector unsigned char, vector unsigned char);
vector signed short vec_sra (vector signed short, vector unsigned short);
vector unsigned short vec_sra (vector unsigned short, vector unsigned short);
vector signed int vec_sra (vector signed int, vector unsigned int);
vector unsigned int vec_sra (vector unsigned int, vector unsigned int);

vector signed int vec_srl (vector signed int, vector unsigned int);
vector signed int vec_srl (vector signed int, vector unsigned short);
vector signed int vec_srl (vector signed int, vector unsigned char);
vector unsigned int vec_srl (vector unsigned int, vector unsigned int);
```

```
vector unsigned int vec_srl (vector unsigned int, vector unsigned short);
vector unsigned int vec_srl (vector unsigned int, vector unsigned char);
vector bool int vec_srl (vector bool int, vector unsigned int);
vector bool int vec_srl (vector bool int, vector unsigned short);
vector bool int vec_srl (vector bool int, vector unsigned char);
vector signed short vec_srl (vector signed short, vector unsigned int);
vector signed short vec_srl (vector signed short, vector unsigned short);
vector signed short vec_srl (vector signed short, vector unsigned char);
vector unsigned short vec_srl (vector unsigned short, vector unsigned int);
vector unsigned short vec_srl (vector unsigned short, vector unsigned short);
vector unsigned short vec_srl (vector unsigned short, vector unsigned char);
vector bool short vec_srl (vector bool short, vector unsigned int);
vector bool short vec_srl (vector bool short, vector unsigned short);
vector bool short vec_srl (vector bool short, vector unsigned char);
vector pixel vec_srl (vector pixel, vector unsigned int);
vector pixel vec_srl (vector pixel, vector unsigned short);
vector pixel vec_srl (vector pixel, vector unsigned char);
vector signed char vec_srl (vector signed char, vector unsigned int);
vector signed char vec_srl (vector signed char, vector unsigned short);
vector signed char vec_srl (vector signed char, vector unsigned char);
vector unsigned char vec_srl (vector unsigned char, vector unsigned int);
vector unsigned char vec_srl (vector unsigned char, vector unsigned short);
vector unsigned char vec_srl (vector unsigned char, vector unsigned char);
vector bool char vec_srl (vector bool char, vector unsigned int);
vector bool char vec_srl (vector bool char, vector unsigned short);
vector bool char vec_srl (vector bool char, vector unsigned char);

vector float vec_sro (vector float, vector signed char);
vector float vec_sro (vector float, vector unsigned char);
vector signed int vec_sro (vector signed int, vector signed char);
vector signed int vec_sro (vector signed int, vector unsigned char);
vector unsigned int vec_sro (vector unsigned int, vector signed char);
vector unsigned int vec_sro (vector unsigned int, vector unsigned char);
vector signed short vec_sro (vector signed short, vector signed char);
vector signed short vec_sro (vector signed short, vector unsigned char);
vector unsigned short vec_sro (vector unsigned short, vector signed char);
vector unsigned short vec_sro (vector unsigned short, vector unsigned char);
vector pixel vec_sro (vector pixel, vector signed char);
vector pixel vec_sro (vector pixel, vector unsigned char);
vector signed char vec_sro (vector signed char, vector signed char);
vector signed char vec_sro (vector signed char, vector unsigned char);
vector unsigned char vec_sro (vector unsigned char, vector signed char);
vector unsigned char vec_sro (vector unsigned char, vector unsigned char);

void vec_st (vector float, int, vector float *);
void vec_st (vector float, int, float *);
void vec_st (vector signed int, int, vector signed int *);
void vec_st (vector signed int, int, int *);
void vec_st (vector unsigned int, int, vector unsigned int *);
void vec_st (vector unsigned int, int, unsigned int *);
void vec_st (vector bool int, int, vector bool int *);
void vec_st (vector bool int, int, unsigned int *);
void vec_st (vector bool int, int, int *);
void vec_st (vector signed short, int, vector signed short *);
void vec_st (vector signed short, int, short *);
void vec_st (vector unsigned short, int, vector unsigned short *);
void vec_st (vector unsigned short, int, unsigned short *);
void vec_st (vector bool short, int, vector bool short *);
```

```
void vec_st (vector bool short, int, unsigned short *);
void vec_st (vector pixel, int, vector pixel *);
void vec_st (vector bool short, int, short *);
void vec_st (vector signed char, int, vector signed char *);
void vec_st (vector signed char, int, signed char *);
void vec_st (vector unsigned char, int, vector unsigned char *);
void vec_st (vector unsigned char, int, unsigned char *);
void vec_st (vector bool char, int, vector bool char *);
void vec_st (vector bool char, int, unsigned char *);
void vec_st (vector bool char, int, signed char *);

void vec_ste (vector signed char, int, signed char *);
void vec_ste (vector unsigned char, int, unsigned char *);
void vec_ste (vector bool char, int, signed char *);
void vec_ste (vector bool char, int, unsigned char *);
void vec_ste (vector signed short, int, short *);
void vec_ste (vector unsigned short, int, unsigned short *);
void vec_ste (vector bool short, int, short *);
void vec_ste (vector bool short, int, unsigned short *);
void vec_ste (vector pixel, int, short *);
void vec_ste (vector pixel, int, unsigned short *);
void vec_ste (vector float, int, float *);
void vec_ste (vector signed int, int, int *);
void vec_ste (vector unsigned int, int, unsigned int *);
void vec_ste (vector bool int, int, int *);
void vec_ste (vector bool int, int, unsigned int *);

void vec_stl (vector float, int, vector float *);
void vec_stl (vector float, int, float *);
void vec_stl (vector signed int, int, vector signed int *);
void vec_stl (vector signed int, int, int *);
void vec_stl (vector unsigned int, int, vector unsigned int *);
void vec_stl (vector unsigned int, int, unsigned int *);
void vec_stl (vector bool int, int, vector bool int *);
void vec_stl (vector bool int, int, unsigned int *);
void vec_stl (vector bool int, int, int *);
void vec_stl (vector signed short, int, vector signed short *);
void vec_stl (vector signed short, int, short *);
void vec_stl (vector unsigned short, int, vector unsigned short *);
void vec_stl (vector unsigned short, int, unsigned short *);
void vec_stl (vector bool short, int, vector bool short *);
void vec_stl (vector bool short, int, unsigned short *);
void vec_stl (vector bool short, int, short *);
void vec_stl (vector pixel, int, vector pixel *);
void vec_stl (vector signed char, int, vector signed char *);
void vec_stl (vector signed char, int, signed char *);
void vec_stl (vector unsigned char, int, vector unsigned char *);
void vec_stl (vector unsigned char, int, unsigned char *);
void vec_stl (vector bool char, int, vector bool char *);
void vec_stl (vector bool char, int, unsigned char *);
void vec_stl (vector bool char, int, signed char *);

void vec_stvebx (vector signed char, int, signed char *);
void vec_stvebx (vector unsigned char, int, unsigned char *);
void vec_stvebx (vector bool char, int, signed char *);
void vec_stvebx (vector bool char, int, unsigned char *);

void vec_stvehx (vector signed short, int, short *);
```

```
void vec_stvehx (vector unsigned short, int, unsigned short *);
void vec_stvehx (vector bool short, int, short *);
void vec_stvehx (vector bool short, int, unsigned short *);

void vec_stvewx (vector float, int, float *);
void vec_stvewx (vector signed int, int, int *);
void vec_stvewx (vector unsigned int, int, unsigned int *);
void vec_stvewx (vector bool int, int, int *);
void vec_stvewx (vector bool int, int, unsigned int *);

vector signed char vec_sub (vector bool char, vector signed char);
vector signed char vec_sub (vector signed char, vector bool char);
vector signed char vec_sub (vector signed char, vector signed char);
vector unsigned char vec_sub (vector bool char, vector unsigned char);
vector unsigned char vec_sub (vector unsigned char, vector bool char);
vector unsigned char vec_sub (vector unsigned char, vector unsigned char);
vector signed short vec_sub (vector bool short, vector signed short);
vector signed short vec_sub (vector signed short, vector bool short);
vector signed short vec_sub (vector signed short, vector signed short);
vector unsigned short vec_sub (vector bool short, vector unsigned short);
vector unsigned short vec_sub (vector unsigned short, vector bool short);
vector unsigned short vec_sub (vector unsigned short, vector unsigned short);
vector signed int vec_sub (vector bool int, vector signed int);
vector signed int vec_sub (vector signed int, vector bool int);
vector signed int vec_sub (vector signed int, vector signed int);
vector unsigned int vec_sub (vector bool int, vector unsigned int);
vector unsigned int vec_sub (vector unsigned int, vector bool int);
vector unsigned int vec_sub (vector unsigned int, vector unsigned int);
vector float vec_sub (vector float, vector float);

vector signed int vec_subc (vector signed int, vector signed int);
vector unsigned int vec_subc (vector unsigned int, vector unsigned int);

vector signed int vec_sube (vector signed int, vector signed int,
                            vector signed int);
vector unsigned int vec_sube (vector unsigned int, vector unsigned int,
                              vector unsigned int);

vector signed int vec_subec (vector signed int, vector signed int,
                             vector signed int);
vector unsigned int vec_subec (vector unsigned int, vector unsigned int,
                               vector unsigned int);

vector unsigned char vec_subs (vector bool char, vector unsigned char);
vector unsigned char vec_subs (vector unsigned char, vector bool char);
vector unsigned char vec_subs (vector unsigned char, vector unsigned char);
vector signed char vec_subs (vector bool char, vector signed char);
vector signed char vec_subs (vector signed char, vector bool char);
vector signed char vec_subs (vector signed char, vector signed char);
vector unsigned short vec_subs (vector bool short, vector unsigned short);
vector unsigned short vec_subs (vector unsigned short, vector bool short);
vector unsigned short vec_subs (vector unsigned short, vector unsigned short);
vector signed short vec_subs (vector bool short, vector signed short);
vector signed short vec_subs (vector signed short, vector bool short);
vector signed short vec_subs (vector signed short, vector signed short);
vector unsigned int vec_subs (vector bool int, vector unsigned int);
vector unsigned int vec_subs (vector unsigned int, vector bool int);
vector unsigned int vec_subs (vector unsigned int, vector unsigned int);
```

```
vector signed int vec_subs (vector bool int, vector signed int);
vector signed int vec_subs (vector signed int, vector bool int);
vector signed int vec_subs (vector signed int, vector signed int);

vector signed int vec_sum2s (vector signed int, vector signed int);

vector unsigned int vec_sum4s (vector unsigned char, vector unsigned int);
vector signed int vec_sum4s (vector signed char, vector signed int);
vector signed int vec_sum4s (vector signed short, vector signed int);

vector signed int vec_sums (vector signed int, vector signed int);

vector float vec_trunc (vector float);

vector signed short vec_unpackh (vector signed char);
vector bool short vec_unpackh (vector bool char);
vector signed int vec_unpackh (vector signed short);
vector bool int vec_unpackh (vector bool short);
vector unsigned int vec_unpackh (vector pixel);

vector signed short vec_unpackl (vector signed char);
vector bool short vec_unpackl (vector bool char);
vector unsigned int vec_unpackl (vector pixel);
vector signed int vec_unpackl (vector signed short);
vector bool int vec_unpackl (vector bool short);

vector float vec_vaddfp (vector float, vector float);

vector signed char vec_vaddsbs (vector bool char, vector signed char);
vector signed char vec_vaddsbs (vector signed char, vector bool char);
vector signed char vec_vaddsbs (vector signed char, vector signed char);

vector signed short vec_vaddshs (vector bool short, vector signed short);
vector signed short vec_vaddshs (vector signed short, vector bool short);
vector signed short vec_vaddshs (vector signed short, vector signed short);

vector signed int vec_vaddsws (vector bool int, vector signed int);
vector signed int vec_vaddsws (vector signed int, vector bool int);
vector signed int vec_vaddsws (vector signed int, vector signed int);

vector signed char vec_vaddubm (vector bool char, vector signed char);
vector signed char vec_vaddubm (vector signed char, vector bool char);
vector signed char vec_vaddubm (vector signed char, vector signed char);
vector unsigned char vec_vaddubm (vector bool char, vector unsigned char);
vector unsigned char vec_vaddubm (vector unsigned char, vector bool char);
vector unsigned char vec_vaddubm (vector unsigned char, vector unsigned char);

vector unsigned char vec_vaddubs (vector bool char, vector unsigned char);
vector unsigned char vec_vaddubs (vector unsigned char, vector bool char);
vector unsigned char vec_vaddubs (vector unsigned char, vector unsigned char);

vector signed short vec_vadduhm (vector bool short, vector signed short);
vector signed short vec_vadduhm (vector signed short, vector bool short);
vector signed short vec_vadduhm (vector signed short, vector signed short);
vector unsigned short vec_vadduhm (vector bool short, vector unsigned short);
vector unsigned short vec_vadduhm (vector unsigned short, vector bool short);
vector unsigned short vec_vadduhm (vector unsigned short, vector unsigned short);
```

```
vector unsigned short vec_vadduhs (vector bool short, vector unsigned short);
vector unsigned short vec_vadduhs (vector unsigned short, vector bool short);
vector unsigned short vec_vadduhs (vector unsigned short, vector unsigned short);

vector signed int vec_vadduwm (vector bool int, vector signed int);
vector signed int vec_vadduwm (vector signed int, vector bool int);
vector signed int vec_vadduwm (vector signed int, vector signed int);
vector unsigned int vec_vadduwm (vector bool int, vector unsigned int);
vector unsigned int vec_vadduwm (vector unsigned int, vector bool int);
vector unsigned int vec_vadduwm (vector unsigned int, vector unsigned int);

vector unsigned int vec_vadduws (vector bool int, vector unsigned int);
vector unsigned int vec_vadduws (vector unsigned int, vector bool int);
vector unsigned int vec_vadduws (vector unsigned int, vector unsigned int);

vector signed char vec_vavgsb (vector signed char, vector signed char);

vector signed short vec_vavgsh (vector signed short, vector signed short);

vector signed int vec_vavgsw (vector signed int, vector signed int);

vector unsigned char vec_vavgub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vavguh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vavguw (vector unsigned int, vector unsigned int);

vector float vec_vcfsx (vector signed int, const int);

vector float vec_vcfux (vector unsigned int, const int);

vector bool int vec_vcmpeqfp (vector float, vector float);

vector bool char vec_vcmpequb (vector signed char, vector signed char);
vector bool char vec_vcmpequb (vector unsigned char, vector unsigned char);

vector bool short vec_vcmpequh (vector signed short, vector signed short);
vector bool short vec_vcmpequh (vector unsigned short, vector unsigned short);

vector bool int vec_vcmpequw (vector signed int, vector signed int);
vector bool int vec_vcmpequw (vector unsigned int, vector unsigned int);

vector bool int vec_vcmpgtfp (vector float, vector float);

vector bool char vec_vcmpgtsb (vector signed char, vector signed char);

vector bool short vec_vcmpgtsh (vector signed short, vector signed short);

vector bool int vec_vcmpgtsw (vector signed int, vector signed int);

vector bool char vec_vcmpgtub (vector unsigned char, vector unsigned char);

vector bool short vec_vcmpgtuh (vector unsigned short, vector unsigned short);

vector bool int vec_vcmpgtuw (vector unsigned int, vector unsigned int);

vector float vec_vmaxfp (vector float, vector float);
```

```
vector signed char vec_vmaxsb (vector bool char, vector signed char);
vector signed char vec_vmaxsb (vector signed char, vector bool char);
vector signed char vec_vmaxsb (vector signed char, vector signed char);

vector signed short vec_vmaxsh (vector bool short, vector signed short);
vector signed short vec_vmaxsh (vector signed short, vector bool short);
vector signed short vec_vmaxsh (vector signed short, vector signed short);

vector signed int vec_vmaxsw (vector bool int, vector signed int);
vector signed int vec_vmaxsw (vector signed int, vector bool int);
vector signed int vec_vmaxsw (vector signed int, vector signed int);

vector unsigned char vec_vmaxub (vector bool char, vector unsigned char);
vector unsigned char vec_vmaxub (vector unsigned char, vector bool char);
vector unsigned char vec_vmaxub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vmaxuh (vector bool short, vector unsigned short);
vector unsigned short vec_vmaxuh (vector unsigned short, vector bool short);
vector unsigned short vec_vmaxuh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vmaxuw (vector bool int, vector unsigned int);
vector unsigned int vec_vmaxuw (vector unsigned int, vector bool int);
vector unsigned int vec_vmaxuw (vector unsigned int, vector unsigned int);

vector float vec_vminfp (vector float, vector float);

vector signed char vec_vminsb (vector bool char, vector signed char);
vector signed char vec_vminsb (vector signed char, vector bool char);
vector signed char vec_vminsb (vector signed char, vector signed char);

vector signed short vec_vminsh (vector bool short, vector signed short);
vector signed short vec_vminsh (vector signed short, vector bool short);
vector signed short vec_vminsh (vector signed short, vector signed short);

vector signed int vec_vminsw (vector bool int, vector signed int);
vector signed int vec_vminsw (vector signed int, vector bool int);
vector signed int vec_vminsw (vector signed int, vector signed int);

vector unsigned char vec_vminub (vector bool char, vector unsigned char);
vector unsigned char vec_vminub (vector unsigned char, vector bool char);
vector unsigned char vec_vminub (vector unsigned char, vector unsigned char);

vector unsigned short vec_vminuh (vector bool short, vector unsigned short);
vector unsigned short vec_vminuh (vector unsigned short, vector bool short);
vector unsigned short vec_vminuh (vector unsigned short, vector unsigned short);

vector unsigned int vec_vminuw (vector bool int, vector unsigned int);
vector unsigned int vec_vminuw (vector unsigned int, vector bool int);
vector unsigned int vec_vminuw (vector unsigned int, vector unsigned int);

vector bool char vec_vmrghb (vector bool char, vector bool char);
vector signed char vec_vmrghb (vector signed char, vector signed char);
vector unsigned char vec_vmrghb (vector unsigned char, vector unsigned char);

vector bool short vec_vmrghh (vector bool short, vector bool short);
vector signed short vec_vmrghh (vector signed short, vector signed short);
vector unsigned short vec_vmrghh (vector unsigned short, vector unsigned short);
vector pixel vec_vmrghh (vector pixel, vector pixel);
```

```
vector float vec_vmrghw (vector float, vector float);
vector bool int vec_vmrghw (vector bool int, vector bool int);
vector signed int vec_vmrghw (vector signed int, vector signed int);
vector unsigned int vec_vmrghw (vector unsigned int, vector unsigned int);

vector bool char vec_vmrglb (vector bool char, vector bool char);
vector signed char vec_vmrglb (vector signed char, vector signed char);
vector unsigned char vec_vmrglb (vector unsigned char, vector unsigned char);

vector bool short vec_vmrglh (vector bool short, vector bool short);
vector signed short vec_vmrglh (vector signed short, vector signed short);
vector unsigned short vec_vmrglh (vector unsigned short, vector unsigned short);
vector pixel vec_vmrglh (vector pixel, vector pixel);

vector float vec_vmrglw (vector float, vector float);
vector signed int vec_vmrglw (vector signed int, vector signed int);
vector unsigned int vec_vmrglw (vector unsigned int, vector unsigned int);
vector bool int vec_vmrglw (vector bool int, vector bool int);

vector signed int vec_vmsummbm (vector signed char, vector unsigned char,
                                vector signed int);

vector signed int vec_vmsumshm (vector signed short, vector signed short,
                                vector signed int);

vector signed int vec_vmsumshs (vector signed short, vector signed short,
                                vector signed int);

vector unsigned int vec_vmsumubm (vector unsigned char, vector unsigned char,
                                  vector unsigned int);

vector unsigned int vec_vmsumuhm (vector unsigned short, vector unsigned short,
                                  vector unsigned int);

vector unsigned int vec_vmsumuhs (vector unsigned short, vector unsigned short,
                                  vector unsigned int);

vector signed short vec_vmulesb (vector signed char, vector signed char);

vector signed int vec_vmulesh (vector signed short, vector signed short);

vector unsigned short vec_vmuleub (vector unsigned char, vector unsigned char);

vector unsigned int vec_vmuleuh (vector unsigned short, vector unsigned short);

vector signed short vec_vmulosb (vector signed char, vector signed char);

vector signed int vec_vmulosh (vector signed short, vector signed short);

vector unsigned short vec_vmuloub (vector unsigned char, vector unsigned char);

vector unsigned int vec_vmulouh (vector unsigned short, vector unsigned short);

vector signed char vec_vpkshss (vector signed short, vector signed short);

vector unsigned char vec_vpkshus (vector signed short, vector signed short);
```

```
vector signed short vec_vpkswss (vector signed int, vector signed int);

vector unsigned short vec_vpkswus (vector signed int, vector signed int);

vector bool char vec_vpkuhum (vector bool short, vector bool short);
vector signed char vec_vpkuhum (vector signed short, vector signed short);
vector unsigned char vec_vpkuhum (vector unsigned short, vector unsigned short);

vector unsigned char vec_vpkuhus (vector unsigned short, vector unsigned short);

vector bool short vec_vpkuwum (vector bool int, vector bool int);
vector signed short vec_vpkuwum (vector signed int, vector signed int);
vector unsigned short vec_vpkuwum (vector unsigned int, vector unsigned int);

vector unsigned short vec_vpkuwus (vector unsigned int, vector unsigned int);

vector signed char vec_vrlb (vector signed char, vector unsigned char);
vector unsigned char vec_vrlb (vector unsigned char, vector unsigned char);

vector signed short vec_vrlh (vector signed short, vector unsigned short);
vector unsigned short vec_vrlh (vector unsigned short, vector unsigned short);

vector signed int vec_vrlw (vector signed int, vector unsigned int);
vector unsigned int vec_vrlw (vector unsigned int, vector unsigned int);

vector signed char vec_vslb (vector signed char, vector unsigned char);
vector unsigned char vec_vslb (vector unsigned char, vector unsigned char);

vector signed short vec_vslh (vector signed short, vector unsigned short);
vector unsigned short vec_vslh (vector unsigned short, vector unsigned short);

vector signed int vec_vslw (vector signed int, vector unsigned int);
vector unsigned int vec_vslw (vector unsigned int, vector unsigned int);

vector signed char vec_vspltb (vector signed char, const int);
vector unsigned char vec_vspltb (vector unsigned char, const int);
vector bool char vec_vspltb (vector bool char, const int);

vector bool short vec_vsplth (vector bool short, const int);
vector signed short vec_vsplth (vector signed short, const int);
vector unsigned short vec_vsplth (vector unsigned short, const int);
vector pixel vec_vsplth (vector pixel, const int);

vector float vec_vspltw (vector float, const int);
vector signed int vec_vspltw (vector signed int, const int);
vector unsigned int vec_vspltw (vector unsigned int, const int);
vector bool int vec_vspltw (vector bool int, const int);

vector signed char vec_vsrab (vector signed char, vector unsigned char);
vector unsigned char vec_vsrab (vector unsigned char, vector unsigned char);

vector signed short vec_vsrah (vector signed short, vector unsigned short);
vector unsigned short vec_vsrah (vector unsigned short, vector unsigned short);

vector signed int vec_vsraw (vector signed int, vector unsigned int);
vector unsigned int vec_vsraw (vector unsigned int, vector unsigned int);

vector signed char vec_vsrb (vector signed char, vector unsigned char);
```

```
vector unsigned char vec_vsrb (vector unsigned char, vector unsigned char);

vector signed short vec_vsrh (vector signed short, vector unsigned short);
vector unsigned short vec_vsrh (vector unsigned short, vector unsigned short);

vector signed int vec_vsrw (vector signed int, vector unsigned int);
vector unsigned int vec_vsrw (vector unsigned int, vector unsigned int);

vector float vec_vsubfp (vector float, vector float);

vector signed char vec_vsubsbs (vector bool char, vector signed char);
vector signed char vec_vsubsbs (vector signed char, vector bool char);
vector signed char vec_vsubsbs (vector signed char, vector signed char);

vector signed short vec_vsubshs (vector bool short, vector signed short);
vector signed short vec_vsubshs (vector signed short, vector bool short);
vector signed short vec_vsubshs (vector signed short, vector signed short);

vector signed int vec_vsubsws (vector bool int, vector signed int);
vector signed int vec_vsubsws (vector signed int, vector bool int);
vector signed int vec_vsubsws (vector signed int, vector signed int);

vector signed char vec_vsububm (vector bool char, vector signed char);
vector signed char vec_vsububm (vector signed char, vector bool char);
vector signed char vec_vsububm (vector signed char, vector signed char);
vector unsigned char vec_vsububm (vector bool char, vector unsigned char);
vector unsigned char vec_vsububm (vector unsigned char, vector bool char);
vector unsigned char vec_vsububm (vector unsigned char, vector unsigned char);

vector unsigned char vec_vsububs (vector bool char, vector unsigned char);
vector unsigned char vec_vsububs (vector unsigned char, vector bool char);
vector unsigned char vec_vsububs (vector unsigned char, vector unsigned char);

vector signed short vec_vsubuhm (vector bool short, vector signed short);
vector signed short vec_vsubuhm (vector signed short, vector bool short);
vector signed short vec_vsubuhm (vector signed short, vector signed short);
vector unsigned short vec_vsubuhm (vector bool short, vector unsigned short);
vector unsigned short vec_vsubuhm (vector unsigned short, vector bool short);
vector unsigned short vec_vsubuhm (vector unsigned short, vector unsigned short);

vector unsigned short vec_vsubuhs (vector bool short, vector unsigned short);
vector unsigned short vec_vsubuhs (vector unsigned short, vector bool short);
vector unsigned short vec_vsubuhs (vector unsigned short, vector unsigned short);

vector signed int vec_vsubuwm (vector bool int, vector signed int);
vector signed int vec_vsubuwm (vector signed int, vector bool int);
vector signed int vec_vsubuwm (vector signed int, vector signed int);
vector unsigned int vec_vsubuwm (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuwm (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuwm (vector unsigned int, vector unsigned int);

vector unsigned int vec_vsubuws (vector bool int, vector unsigned int);
vector unsigned int vec_vsubuws (vector unsigned int, vector bool int);
vector unsigned int vec_vsubuws (vector unsigned int, vector unsigned int);

vector signed int vec_vsum4sbs (vector signed char, vector signed int);

vector signed int vec_vsum4shs (vector signed short, vector signed int);
```

```
vector unsigned int vec_vsum4ubs (vector unsigned char, vector unsigned int);

vector unsigned int vec_vupkhpx (vector pixel);

vector bool short vec_vupkhsb (vector bool char);
vector signed short vec_vupkhsb (vector signed char);

vector bool int vec_vupkhsh (vector bool short);
vector signed int vec_vupkhsh (vector signed short);

vector unsigned int vec_vupklpx (vector pixel);

vector bool short vec_vupklsb (vector bool char);
vector signed short vec_vupklsb (vector signed char);

vector bool int vec_vupklsh (vector bool short);
vector signed int vec_vupklsh (vector signed short);

vector float vec_xor (vector float, vector float);
vector float vec_xor (vector float, vector bool int);
vector float vec_xor (vector bool int, vector float);
vector bool int vec_xor (vector bool int, vector bool int);
vector signed int vec_xor (vector bool int, vector signed int);
vector signed int vec_xor (vector signed int, vector bool int);
vector signed int vec_xor (vector signed int, vector signed int);
vector unsigned int vec_xor (vector bool int, vector unsigned int);
vector unsigned int vec_xor (vector unsigned int, vector bool int);
vector unsigned int vec_xor (vector unsigned int, vector unsigned int);
vector bool short vec_xor (vector bool short, vector bool short);
vector signed short vec_xor (vector bool short, vector signed short);
vector signed short vec_xor (vector signed short, vector bool short);
vector signed short vec_xor (vector signed short, vector signed short);
vector unsigned short vec_xor (vector bool short, vector unsigned short);
vector unsigned short vec_xor (vector unsigned short, vector bool short);
vector unsigned short vec_xor (vector unsigned short, vector unsigned short);
vector signed char vec_xor (vector bool char, vector signed char);
vector bool char vec_xor (vector bool char, vector bool char);
vector signed char vec_xor (vector signed char, vector bool char);
vector signed char vec_xor (vector signed char, vector signed char);
vector unsigned char vec_xor (vector bool char, vector unsigned char);
vector unsigned char vec_xor (vector unsigned char, vector bool char);
vector unsigned char vec_xor (vector unsigned char, vector unsigned char);
```

6.60.22.2 PowerPC AltiVec Built-in Functions Available on ISA 2.06

The AltiVec built-in functions described in this section are available on the PowerPC family of processors starting with ISA 2.06 or later. These are normally enabled by adding '`-mvsx`' to the command line.

When '`-mvsx`' is used, the following additional vector types are implemented.

```
vector unsigned __int128
vector signed __int128
vector unsigned long long int
vector signed long long int
vector double
```

The long long types are only implemented for 64-bit code generation.

```
vector bool long long vec_and (vector bool long long int, vector bool long long);

vector double vec_ctf (vector unsigned long, const int);
vector double vec_ctf (vector signed long, const int);

vector signed long vec_cts (vector double, const int);

vector unsigned long vec_ctu (vector double, const int);

void vec_dst (const unsigned long *, int, const int);
void vec_dst (const long *, int, const int);

void vec_dststt (const unsigned long *, int, const int);
void vec_dststt (const long *, int, const int);

void vec_dstt (const unsigned long *, int, const int);
void vec_dstt (const long *, int, const int);

vector unsigned char vec_lvsl (int, const unsigned long *);
vector unsigned char vec_lvsl (int, const long *);

vector unsigned char vec_lvsr (int, const unsigned long *);
vector unsigned char vec_lvsr (int, const long *);

vector double vec_mul (vector double, vector double);
vector long vec_mul (vector long, vector long);
vector unsigned long vec_mul (vector unsigned long, vector unsigned long);

vector unsigned long long vec_mule (vector unsigned int, vector unsigned int);
vector signed long long vec_mule (vector signed int, vector signed int);

vector unsigned long long vec_mulo (vector unsigned int, vector unsigned int);
vector signed long long vec_mulo (vector signed int, vector signed int);

vector double vec_nabs (vector double);

vector bool long long vec_reve (vector bool long long);
vector signed long long vec_reve (vector signed long long);
vector unsigned long long vec_reve (vector unsigned long long);
vector double vec_sld (vector double, vector double, const int);

vector bool long long int vec_sld (vector bool long long int,
                                   vector bool long long int, const int);
vector long long int vec_sld (vector long long int, vector  long long int, const int);
vector unsigned long long int vec_sld (vector unsigned long long int,
                                       vector unsigned long long int, const int);

vector long long int vec_sll (vector long long int, vector unsigned char);
vector unsigned long long int vec_sll (vector unsigned long long int,
                                       vector unsigned char);

vector signed long long vec_slo (vector signed long long, vector signed char);
vector signed long long vec_slo (vector signed long long, vector unsigned char);
vector unsigned long long vec_slo (vector unsigned long long, vector signed char);
vector unsigned long long vec_slo (vector unsigned long long, vector unsigned char);

vector signed long vec_splat (vector signed long, const int);
```

```
vector unsigned long vec_splat (vector unsigned long, const int);

vector long long int vec_srl (vector long long int, vector unsigned char);
vector unsigned long long int vec_srl (vector unsigned long long int,
                                       vector unsigned char);

vector long long int vec_sro (vector long long int, vector char);
vector long long int vec_sro (vector long long int, vector unsigned char);
vector unsigned long long int vec_sro (vector unsigned long long int, vector char);
vector unsigned long long int vec_sro (vector unsigned long long int,
                                       vector unsigned char);

vector signed __int128 vec_subc (vector signed __int128, vector signed __int128);
vector unsigned __int128 vec_subc (vector unsigned __int128, vector unsigned __int128);

vector signed __int128 vec_sube (vector signed __int128, vector signed __int128,
                                 vector signed __int128);
vector unsigned __int128 vec_sube (vector unsigned __int128, vector unsigned __int128,
                                   vector unsigned __int128);

vector signed __int128 vec_subec (vector signed __int128, vector signed __int128,
                                  vector signed __int128);
vector unsigned __int128 vec_subec (vector unsigned __int128, vector unsigned __int128,
                                    vector unsigned __int128);

vector double vec_unpackh (vector float);

vector double vec_unpackl (vector float);

vector double vec_doublee (vector float);
vector double vec_doublee (vector signed int);
vector double vec_doublee (vector unsigned int);

vector double vec_doubleo (vector float);
vector double vec_doubleo (vector signed int);
vector double vec_doubleo (vector unsigned int);

vector double vec_doubleh (vector float);
vector double vec_doubleh (vector signed int);
vector double vec_doubleh (vector unsigned int);

vector double vec_doublel (vector float);
vector double vec_doublel (vector signed int);
vector double vec_doublel (vector unsigned int);

vector float vec_float (vector signed int);
vector float vec_float (vector unsigned int);

vector float vec_float2 (vector signed long long, vector signed long long);
vector float vec_float2 (vector unsigned long long, vector signed long long);

vector float vec_floate (vector double);
vector float vec_floate (vector signed long long);
vector float vec_floate (vector unsigned long long);

vector float vec_floato (vector double);
vector float vec_floato (vector signed long long);
vector float vec_floato (vector unsigned long long);
```

```
vector signed long long vec_signed (vector double);
vector signed int vec_signed (vector float);

vector signed int vec_signede (vector double);

vector signed int vec_signedo (vector double);

vector signed char vec_sldw (vector signed char, vector signed char, const int);
vector unsigned char vec_sldw (vector unsigned char, vector unsigned char, const int);
vector signed short vec_sldw (vector signed short, vector signed short, const int);
vector unsigned short vec_sldw (vector unsigned short,
                                vector unsigned short, const int);
vector signed int vec_sldw (vector signed int, vector signed int, const int);
vector unsigned int vec_sldw (vector unsigned int, vector unsigned int, const int);
vector signed long long vec_sldw (vector signed long long,
                                  vector signed long long, const int);
vector unsigned long long vec_sldw (vector unsigned long long,
                                    vector unsigned long long, const int);

vector signed long long vec_unsigned (vector double);
vector signed int vec_unsigned (vector float);

vector signed int vec_unsignede (vector double);

vector signed int vec_unsignedo (vector double);

vector double vec_abs (vector double);
vector double vec_add (vector double, vector double);
vector double vec_and (vector double, vector double);
vector double vec_and (vector double, vector bool long);
vector double vec_and (vector bool long, vector double);
vector long vec_and (vector long, vector long);
vector long vec_and (vector long, vector bool long);
vector long vec_and (vector bool long, vector long);
vector unsigned long vec_and (vector unsigned long, vector unsigned long);
vector unsigned long vec_and (vector unsigned long, vector bool long);
vector unsigned long vec_and (vector bool long, vector unsigned long);
vector double vec_andc (vector double, vector double);
vector double vec_andc (vector double, vector bool long);
vector double vec_andc (vector bool long, vector double);
vector long vec_andc (vector long, vector long);
vector long vec_andc (vector long, vector bool long);
vector long vec_andc (vector bool long, vector long);
vector unsigned long vec_andc (vector unsigned long, vector unsigned long);
vector unsigned long vec_andc (vector unsigned long, vector bool long);
vector unsigned long vec_andc (vector bool long, vector unsigned long);
vector double vec_ceil (vector double);
vector bool long vec_cmpeq (vector double, vector double);
vector bool long vec_cmpge (vector double, vector double);
vector bool long vec_cmpgt (vector double, vector double);
vector bool long vec_cmple (vector double, vector double);
vector bool long vec_cmplt (vector double, vector double);
vector double vec_cpsgn (vector double, vector double);
vector float vec_div (vector float, vector float);
vector double vec_div (vector double, vector double);
vector long vec_div (vector long, vector long);
vector unsigned long vec_div (vector unsigned long, vector unsigned long);
```

```
vector double vec_floor (vector double);
vector signed long long vec_ld (int, const vector signed long long *);
vector signed long long vec_ld (int, const signed long long *);
vector unsigned long long vec_ld (int, const vector unsigned long long *);
vector unsigned long long vec_ld (int, const unsigned long long *);
vector __int128 vec_ld (int, const vector __int128 *);
vector unsigned __int128 vec_ld (int, const vector unsigned __int128 *);
vector __int128 vec_ld (int, const __int128 *);
vector unsigned __int128 vec_ld (int, const unsigned __int128 *);
vector double vec_ld (int, const vector double *);
vector double vec_ld (int, const double *);
vector double vec_ldl (int, const vector double *);
vector double vec_ldl (int, const double *);
vector unsigned char vec_lvsl (int, const double *);
vector unsigned char vec_lvsr (int, const double *);
vector double vec_madd (vector double, vector double, vector double);
vector double vec_max (vector double, vector double);
vector signed long vec_mergeh (vector signed long, vector signed long);
vector signed long vec_mergeh (vector signed long, vector bool long);
vector signed long vec_mergeh (vector bool long, vector signed long);
vector unsigned long vec_mergeh (vector unsigned long, vector unsigned long);
vector unsigned long vec_mergeh (vector unsigned long, vector bool long);
vector unsigned long vec_mergeh (vector bool long, vector unsigned long);
vector signed long vec_mergel (vector signed long, vector signed long);
vector signed long vec_mergel (vector signed long, vector bool long);
vector signed long vec_mergel (vector bool long, vector signed long);
vector unsigned long vec_mergel (vector unsigned long, vector unsigned long);
vector unsigned long vec_mergel (vector unsigned long, vector bool long);
vector unsigned long vec_mergel (vector bool long, vector unsigned long);
vector double vec_min (vector double, vector double);
vector float vec_msub (vector float, vector float, vector float);
vector double vec_msub (vector double, vector double, vector double);
vector float vec_nearbyint (vector float);
vector double vec_nearbyint (vector double);
vector float vec_nmadd (vector float, vector float, vector float);
vector double vec_nmadd (vector double, vector double, vector double);
vector double vec_nmsub (vector double, vector double, vector double);
vector double vec_nor (vector double, vector double);
vector long vec_nor (vector long, vector long);
vector long vec_nor (vector long, vector bool long);
vector long vec_nor (vector bool long, vector long);
vector unsigned long vec_nor (vector unsigned long, vector unsigned long);
vector unsigned long vec_nor (vector unsigned long, vector bool long);
vector unsigned long vec_nor (vector bool long, vector unsigned long);
vector double vec_or (vector double, vector double);
vector double vec_or (vector double, vector bool long);
vector double vec_or (vector bool long, vector double);
vector long vec_or (vector long, vector long);
vector long vec_or (vector long, vector bool long);
vector long vec_or (vector bool long, vector long);
vector unsigned long vec_or (vector unsigned long, vector unsigned long);
vector unsigned long vec_or (vector unsigned long, vector bool long);
vector unsigned long vec_or (vector bool long, vector unsigned long);
vector double vec_perm (vector double, vector double, vector unsigned char);
vector long vec_perm (vector long, vector long, vector unsigned char);
vector unsigned long vec_perm (vector unsigned long, vector unsigned long,
                               vector unsigned char);
vector bool char vec_permxor (vector bool char, vector bool char,
```

```
                                vector bool char);
vector unsigned char vec_permxor (vector signed char, vector signed char,
                                  vector signed char);
vector unsigned char vec_permxor (vector unsigned char, vector unsigned char,
                                  vector unsigned char);
vector double vec_rint (vector double);
vector double vec_recip (vector double, vector double);
vector double vec_rsqrt (vector double);
vector double vec_rsqrte (vector double);
vector double vec_sel (vector double, vector double, vector bool long);
vector double vec_sel (vector double, vector double, vector unsigned long);
vector long vec_sel (vector long, vector long, vector long);
vector long vec_sel (vector long, vector long, vector unsigned long);
vector long vec_sel (vector long, vector long, vector bool long);
vector unsigned long vec_sel (vector unsigned long, vector unsigned long,
                              vector long);
vector unsigned long vec_sel (vector unsigned long, vector unsigned long,
                              vector unsigned long);
vector unsigned long vec_sel (vector unsigned long, vector unsigned long,
                              vector bool long);
vector double vec_splats (double);
vector signed long vec_splats (signed long);
vector unsigned long vec_splats (unsigned long);
vector float vec_sqrt (vector float);
vector double vec_sqrt (vector double);
void vec_st (vector signed long long, int, vector signed long long *);
void vec_st (vector signed long long, int, signed long long *);
void vec_st (vector unsigned long long, int, vector unsigned long long *);
void vec_st (vector unsigned long long, int, unsigned long long *);
void vec_st (vector bool long long, int, vector bool long long *);
void vec_st (vector bool long long, int, signed long long *);
void vec_st (vector bool long long, int, unsigned long long *);
void vec_st (vector double, int, vector double *);
void vec_st (vector double, int, double *);
vector double vec_sub (vector double, vector double);
vector double vec_trunc (vector double);
vector double vec_xl (int, vector double *);
vector double vec_xl (int, double *);
vector long long vec_xl (int, vector long long *);
vector long long vec_xl (int, long long *);
vector unsigned long long vec_xl (int, vector unsigned long long *);
vector unsigned long long vec_xl (int, unsigned long long *);
vector float vec_xl (int, vector float *);
vector float vec_xl (int, float *);
vector int vec_xl (int, vector int *);
vector int vec_xl (int, int *);
vector unsigned int vec_xl (int, vector unsigned int *);
vector unsigned int vec_xl (int, unsigned int *);
vector double vec_xor (vector double, vector double);
vector double vec_xor (vector double, vector bool long);
vector double vec_xor (vector bool long, vector double);
vector long vec_xor (vector long, vector long);
vector long vec_xor (vector long, vector bool long);
vector long vec_xor (vector bool long, vector long);
vector unsigned long vec_xor (vector unsigned long, vector unsigned long);
vector unsigned long vec_xor (vector unsigned long, vector bool long);
vector unsigned long vec_xor (vector bool long, vector unsigned long);
void vec_xst (vector double, int, vector double *);
```

```
void vec_xst (vector double, int, double *);
void vec_xst (vector long long, int, vector long long *);
void vec_xst (vector long long, int, long long *);
void vec_xst (vector unsigned long long, int, vector unsigned long long *);
void vec_xst (vector unsigned long long, int, unsigned long long *);
void vec_xst (vector float, int, vector float *);
void vec_xst (vector float, int, float *);
void vec_xst (vector int, int, vector int *);
void vec_xst (vector int, int, int *);
void vec_xst (vector unsigned int, int, vector unsigned int *);
void vec_xst (vector unsigned int, int, unsigned int *);
int vec_all_eq (vector double, vector double);
int vec_all_ge (vector double, vector double);
int vec_all_gt (vector double, vector double);
int vec_all_le (vector double, vector double);
int vec_all_lt (vector double, vector double);
int vec_all_nan (vector double);
int vec_all_ne (vector double, vector double);
int vec_all_nge (vector double, vector double);
int vec_all_ngt (vector double, vector double);
int vec_all_nle (vector double, vector double);
int vec_all_nlt (vector double, vector double);
int vec_all_numeric (vector double);
int vec_any_eq (vector double, vector double);
int vec_any_ge (vector double, vector double);
int vec_any_gt (vector double, vector double);
int vec_any_le (vector double, vector double);
int vec_any_lt (vector double, vector double);
int vec_any_nan (vector double);
int vec_any_ne (vector double, vector double);
int vec_any_nge (vector double, vector double);
int vec_any_ngt (vector double, vector double);
int vec_any_nle (vector double, vector double);
int vec_any_nlt (vector double, vector double);
int vec_any_numeric (vector double);

vector double vec_vsx_ld (int, const vector double *);
vector double vec_vsx_ld (int, const double *);
vector float vec_vsx_ld (int, const vector float *);
vector float vec_vsx_ld (int, const float *);
vector bool int vec_vsx_ld (int, const vector bool int *);
vector signed int vec_vsx_ld (int, const vector signed int *);
vector signed int vec_vsx_ld (int, const int *);
vector signed int vec_vsx_ld (int, const long *);
vector unsigned int vec_vsx_ld (int, const vector unsigned int *);
vector unsigned int vec_vsx_ld (int, const unsigned int *);
vector unsigned int vec_vsx_ld (int, const unsigned long *);
vector bool short vec_vsx_ld (int, const vector bool short *);
vector pixel vec_vsx_ld (int, const vector pixel *);
vector signed short vec_vsx_ld (int, const vector signed short *);
vector signed short vec_vsx_ld (int, const short *);
vector unsigned short vec_vsx_ld (int, const vector unsigned short *);
vector unsigned short vec_vsx_ld (int, const unsigned short *);
vector bool char vec_vsx_ld (int, const vector bool char *);
vector signed char vec_vsx_ld (int, const vector signed char *);
vector signed char vec_vsx_ld (int, const signed char *);
vector unsigned char vec_vsx_ld (int, const vector unsigned char *);
vector unsigned char vec_vsx_ld (int, const unsigned char *);
```

```
void vec_vsx_st (vector double, int, vector double *);
void vec_vsx_st (vector double, int, double *);
void vec_vsx_st (vector float, int, vector float *);
void vec_vsx_st (vector float, int, float *);
void vec_vsx_st (vector signed int, int, vector signed int *);
void vec_vsx_st (vector signed int, int, int *);
void vec_vsx_st (vector unsigned int, int, vector unsigned int *);
void vec_vsx_st (vector unsigned int, int, unsigned int *);
void vec_vsx_st (vector bool int, int, vector bool int *);
void vec_vsx_st (vector bool int, int, unsigned int *);
void vec_vsx_st (vector bool int, int, int *);
void vec_vsx_st (vector signed short, int, vector signed short *);
void vec_vsx_st (vector signed short, int, short *);
void vec_vsx_st (vector unsigned short, int, vector unsigned short *);
void vec_vsx_st (vector unsigned short, int, unsigned short *);
void vec_vsx_st (vector bool short, int, vector bool short *);
void vec_vsx_st (vector bool short, int, unsigned short *);
void vec_vsx_st (vector pixel, int, vector pixel *);
void vec_vsx_st (vector pixel, int, unsigned short *);
void vec_vsx_st (vector pixel, int, short *);
void vec_vsx_st (vector bool short, int, short *);
void vec_vsx_st (vector signed char, int, vector signed char *);
void vec_vsx_st (vector signed char, int, signed char *);
void vec_vsx_st (vector unsigned char, int, vector unsigned char *);
void vec_vsx_st (vector unsigned char, int, unsigned char *);
void vec_vsx_st (vector bool char, int, vector bool char *);
void vec_vsx_st (vector bool char, int, unsigned char *);
void vec_vsx_st (vector bool char, int, signed char *);

vector double vec_xxpermdi (vector double, vector double, const int);
vector float vec_xxpermdi (vector float, vector float, const int);
vector long long vec_xxpermdi (vector long long, vector long long, const int);
vector unsigned long long vec_xxpermdi (vector unsigned long long,
                                         vector unsigned long long, const int);
vector int vec_xxpermdi (vector int, vector int, const int);
vector unsigned int vec_xxpermdi (vector unsigned int,
                                  vector unsigned int, const int);
vector short vec_xxpermdi (vector short, vector short, const int);
vector unsigned short vec_xxpermdi (vector unsigned short,
                                    vector unsigned short, const int);
vector signed char vec_xxpermdi (vector signed char, vector signed char,
                                 const int);
vector unsigned char vec_xxpermdi (vector unsigned char,
                                   vector unsigned char, const int);

vector double vec_xxsldi (vector double, vector double, int);
vector float vec_xxsldi (vector float, vector float, int);
vector long long vec_xxsldi (vector long long, vector long long, int);
vector unsigned long long vec_xxsldi (vector unsigned long long,
                                      vector unsigned long long, int);
vector int vec_xxsldi (vector int, vector int, int);
vector unsigned int vec_xxsldi (vector unsigned int, vector unsigned int, int);
vector short vec_xxsldi (vector short, vector short, int);
vector unsigned short vec_xxsldi (vector unsigned short,
                                  vector unsigned short, int);
vector signed char vec_xxsldi (vector signed char, vector signed char, int);
vector unsigned char vec_xxsldi (vector unsigned char,
```

```
                                    vector unsigned char, int);
```

Note that the 'vec_ld' and 'vec_st' built-in functions always generate the AltiVec 'LVX' and 'STVX' instructions even if the VSX instruction set is available. The 'vec_vsx_ld' and 'vec_vsx_st' built-in functions always generate the VSX 'LXVD2X', 'LXVW4X', 'STXVD2X', and 'STXVW4X' instructions.

6.60.22.3 PowerPC AltiVec Built-in Functions Available on ISA 2.07

If the ISA 2.07 additions to the vector/scalar (power8-vector) instruction set are available, the following additional functions are available for both 32-bit and 64-bit targets. For 64-bit targets, you can use *vector long* instead of *vector long long*, *vector bool long* instead of *vector bool long long*, and *vector unsigned long* instead of *vector unsigned long long*.

```
vector signed char vec_neg (vector signed char);
vector signed short vec_neg (vector signed short);
vector signed int vec_neg (vector signed int);
vector signed long long vec_neg (vector signed long long);
vector float  char vec_neg (vector float);
vector double vec_neg (vector double);

vector signed int vec_signed2 (vector double, vector double);

vector signed int vec_unsigned2 (vector double, vector double);

vector long long vec_abs (vector long long);

vector long long vec_add (vector long long, vector long long);
vector unsigned long long vec_add (vector unsigned long long,
                                   vector unsigned long long);

int vec_all_eq (vector long long, vector long long);
int vec_all_eq (vector unsigned long long, vector unsigned long long);
int vec_all_ge (vector long long, vector long long);
int vec_all_ge (vector unsigned long long, vector unsigned long long);
int vec_all_gt (vector long long, vector long long);
int vec_all_gt (vector unsigned long long, vector unsigned long long);
int vec_all_le (vector long long, vector long long);
int vec_all_le (vector unsigned long long, vector unsigned long long);
int vec_all_lt (vector long long, vector long long);
int vec_all_lt (vector unsigned long long, vector unsigned long long);
int vec_all_ne (vector long long, vector long long);
int vec_all_ne (vector unsigned long long, vector unsigned long long);

int vec_any_eq (vector long long, vector long long);
int vec_any_eq (vector unsigned long long, vector unsigned long long);
int vec_any_ge (vector long long, vector long long);
int vec_any_ge (vector unsigned long long, vector unsigned long long);
int vec_any_gt (vector long long, vector long long);
int vec_any_gt (vector unsigned long long, vector unsigned long long);
int vec_any_le (vector long long, vector long long);
int vec_any_le (vector unsigned long long, vector unsigned long long);
int vec_any_lt (vector long long, vector long long);
int vec_any_lt (vector unsigned long long, vector unsigned long long);
int vec_any_ne (vector long long, vector long long);
int vec_any_ne (vector unsigned long long, vector unsigned long long);
```

```
vector bool long long vec_cmpeq (vector bool long long, vector bool long long);

vector long long vec_eqv (vector long long, vector long long);
vector long long vec_eqv (vector bool long long, vector long long);
vector long long vec_eqv (vector long long, vector bool long long);
vector unsigned long long vec_eqv (vector unsigned long long, vector unsigned long long);
vector unsigned long long vec_eqv (vector bool long long, vector unsigned long long);
vector unsigned long long vec_eqv (vector unsigned long long,
                                   vector bool long long);
vector int vec_eqv (vector int, vector int);
vector int vec_eqv (vector bool int, vector int);
vector int vec_eqv (vector int, vector bool int);
vector unsigned int vec_eqv (vector unsigned int, vector unsigned int);
vector unsigned int vec_eqv (vector bool unsigned int, vector unsigned int);
vector unsigned int vec_eqv (vector unsigned int, vector bool unsigned int);
vector short vec_eqv (vector short, vector short);
vector short vec_eqv (vector bool short, vector short);
vector short vec_eqv (vector short, vector bool short);
vector unsigned short vec_eqv (vector unsigned short, vector unsigned short);
vector unsigned short vec_eqv (vector bool unsigned short, vector unsigned short);
vector unsigned short vec_eqv (vector unsigned short, vector bool unsigned short);
vector signed char vec_eqv (vector signed char, vector signed char);
vector signed char vec_eqv (vector bool signed char, vector signed char);
vector signed char vec_eqv (vector signed char, vector bool signed char);
vector unsigned char vec_eqv (vector unsigned char, vector unsigned char);
vector unsigned char vec_eqv (vector bool unsigned char, vector unsigned char);
vector unsigned char vec_eqv (vector unsigned char, vector bool unsigned char);

vector long long vec_max (vector long long, vector long long);
vector unsigned long long vec_max (vector unsigned long long,
                                   vector unsigned long long);

vector signed int vec_mergee (vector signed int, vector signed int);
vector unsigned int vec_mergee (vector unsigned int, vector unsigned int);
vector bool int vec_mergee (vector bool int, vector bool int);

vector signed int vec_mergeo (vector signed int, vector signed int);
vector unsigned int vec_mergeo (vector unsigned int, vector unsigned int);
vector bool int vec_mergeo (vector bool int, vector bool int);

vector long long vec_min (vector long long, vector long long);
vector unsigned long long vec_min (vector unsigned long long,
                                   vector unsigned long long);

vector signed long long vec_nabs (vector signed long long);

vector long long vec_nand (vector long long, vector long long);
vector long long vec_nand (vector bool long long, vector long long);
vector long long vec_nand (vector long long, vector bool long long);
vector unsigned long long vec_nand (vector unsigned long long,
                                    vector unsigned long long);
vector unsigned long long vec_nand (vector bool long long, vector unsigned long long);
vector unsigned long long vec_nand (vector unsigned long long, vector bool long long);
vector int vec_nand (vector int, vector int);
vector int vec_nand (vector bool int, vector int);
vector int vec_nand (vector int, vector bool int);
vector unsigned int vec_nand (vector unsigned int, vector unsigned int);
vector unsigned int vec_nand (vector bool unsigned int, vector unsigned int);
```

```
vector unsigned int vec_nand (vector unsigned int, vector bool unsigned int);
vector short vec_nand (vector short, vector short);
vector short vec_nand (vector bool short, vector short);
vector short vec_nand (vector short, vector bool short);
vector unsigned short vec_nand (vector unsigned short, vector unsigned short);
vector unsigned short vec_nand (vector bool unsigned short, vector unsigned short);
vector unsigned short vec_nand (vector unsigned short, vector bool unsigned short);
vector signed char vec_nand (vector signed char, vector signed char);
vector signed char vec_nand (vector bool signed char, vector signed char);
vector signed char vec_nand (vector signed char, vector bool signed char);
vector unsigned char vec_nand (vector unsigned char, vector unsigned char);
vector unsigned char vec_nand (vector bool unsigned char, vector unsigned char);
vector unsigned char vec_nand (vector unsigned char, vector bool unsigned char);

vector long long vec_orc (vector long long, vector long long);
vector long long vec_orc (vector bool long long, vector long long);
vector long long vec_orc (vector long long, vector bool long long);
vector unsigned long long vec_orc (vector unsigned long long,
                                   vector unsigned long long);
vector unsigned long long vec_orc (vector bool long long, vector unsigned long long);
vector unsigned long long vec_orc (vector unsigned long long, vector bool long long);
vector int vec_orc (vector int, vector int);
vector int vec_orc (vector bool int, vector int);
vector int vec_orc (vector int, vector bool int);
vector unsigned int vec_orc (vector unsigned int, vector unsigned int);
vector unsigned int vec_orc (vector bool unsigned int, vector unsigned int);
vector unsigned int vec_orc (vector unsigned int, vector bool unsigned int);
vector short vec_orc (vector short, vector short);
vector short vec_orc (vector bool short, vector short);
vector short vec_orc (vector short, vector bool short);
vector unsigned short vec_orc (vector unsigned short, vector unsigned short);
vector unsigned short vec_orc (vector bool unsigned short, vector unsigned short);
vector unsigned short vec_orc (vector unsigned short, vector bool unsigned short);
vector signed char vec_orc (vector signed char, vector signed char);
vector signed char vec_orc (vector bool signed char, vector signed char);
vector signed char vec_orc (vector signed char, vector bool signed char);
vector unsigned char vec_orc (vector unsigned char, vector unsigned char);
vector unsigned char vec_orc (vector bool unsigned char, vector unsigned char);
vector unsigned char vec_orc (vector unsigned char, vector bool unsigned char);

vector int vec_pack (vector long long, vector long long);
vector unsigned int vec_pack (vector unsigned long long, vector unsigned long long);
vector bool int vec_pack (vector bool long long, vector bool long long);
vector float vec_pack (vector double, vector double);

vector int vec_packs (vector long long, vector long long);
vector unsigned int vec_packs (vector unsigned long long, vector unsigned long long);

vector unsigned char vec_packsu (vector signed short, vector signed short)
vector unsigned char vec_packsu (vector unsigned short, vector unsigned short)
vector unsigned short int vec_packsu (vector signed int, vector signed int);
vector unsigned short int vec_packsu (vector unsigned int, vector unsigned int);
vector unsigned int vec_packsu (vector long long, vector long long);
vector unsigned int vec_packsu (vector unsigned long long, vector unsigned long long);
vector unsigned int vec_packsu (vector signed long long, vector signed long long);

vector unsigned char vec_popcnt (vector signed char);
vector unsigned char vec_popcnt (vector unsigned char);
```

```
vector unsigned short vec_popcnt (vector signed short);
vector unsigned short vec_popcnt (vector unsigned short);
vector unsigned int vec_popcnt (vector signed int);
vector unsigned int vec_popcnt (vector unsigned int);
vector unsigned long long vec_popcnt (vector signed long long);
vector unsigned long long vec_popcnt (vector unsigned long long);

vector long long vec_rl (vector long long, vector unsigned long long);
vector long long vec_rl (vector unsigned long long, vector unsigned long long);

vector long long vec_sl (vector long long, vector unsigned long long);
vector long long vec_sl (vector unsigned long long, vector unsigned long long);

vector long long vec_sr (vector long long, vector unsigned long long);
vector unsigned long long char vec_sr (vector unsigned long long,
                                       vector unsigned long long);

vector long long vec_sra (vector long long, vector unsigned long long);
vector unsigned long long vec_sra (vector unsigned long long,
                                   vector unsigned long long);

vector long long vec_sub (vector long long, vector long long);
vector unsigned long long vec_sub (vector unsigned long long,
                                   vector unsigned long long);

vector long long vec_unpackh (vector int);
vector unsigned long long vec_unpackh (vector unsigned int);

vector long long vec_unpackl (vector int);
vector unsigned long long vec_unpackl (vector unsigned int);

vector long long vec_vaddudm (vector long long, vector long long);
vector long long vec_vaddudm (vector bool long long, vector long long);
vector long long vec_vaddudm (vector long long, vector bool long long);
vector unsigned long long vec_vaddudm (vector unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vaddudm (vector bool unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vaddudm (vector unsigned long long,
                                       vector bool unsigned long long);

vector long long vec_vbpermq (vector signed char, vector signed char);
vector long long vec_vbpermq (vector unsigned char, vector unsigned char);

vector unsigned char vec_bperm (vector unsigned char, vector unsigned char);
vector unsigned char vec_bperm (vector unsigned long long, vector unsigned char);
vector unsigned long long vec_bperm (vector unsigned __int128, vector unsigned char);

vector long long vec_cntlz (vector long long);
vector unsigned long long vec_cntlz (vector unsigned long long);
vector int vec_cntlz (vector int);
vector unsigned int vec_cntlz (vector int);
vector short vec_cntlz (vector short);
vector unsigned short vec_cntlz (vector unsigned short);
vector signed char vec_cntlz (vector signed char);
vector unsigned char vec_cntlz (vector unsigned char);

vector long long vec_vclz (vector long long);
```

```
vector unsigned long long vec_vclz (vector unsigned long long);
vector int vec_vclz (vector int);
vector unsigned int vec_vclz (vector int);
vector short vec_vclz (vector short);
vector unsigned short vec_vclz (vector unsigned short);
vector signed char vec_vclz (vector signed char);
vector unsigned char vec_vclz (vector unsigned char);

vector signed char vec_vclzb (vector signed char);
vector unsigned char vec_vclzb (vector unsigned char);

vector long long vec_vclzd (vector long long);
vector unsigned long long vec_vclzd (vector unsigned long long);

vector short vec_vclzh (vector short);
vector unsigned short vec_vclzh (vector unsigned short);

vector int vec_vclzw (vector int);
vector unsigned int vec_vclzw (vector int);

vector signed char vec_vgbbd (vector signed char);
vector unsigned char vec_vgbbd (vector unsigned char);

vector long long vec_vmaxsd (vector long long, vector long long);

vector unsigned long long vec_vmaxud (vector unsigned long long,
                                      unsigned vector long long);

vector long long vec_vminsd (vector long long, vector long long);

vector unsigned long long vec_vminud (vector long long, vector long long);

vector int vec_vpksdss (vector long long, vector long long);
vector unsigned int vec_vpksdss (vector long long, vector long long);

vector unsigned int vec_vpkudus (vector unsigned long long,
                                 vector unsigned long long);

vector int vec_vpkudum (vector long long, vector long long);
vector unsigned int vec_vpkudum (vector unsigned long long,
                                 vector unsigned long long);
vector bool int vec_vpkudum (vector bool long long, vector bool long long);

vector long long vec_vpopcnt (vector long long);
vector unsigned long long vec_vpopcnt (vector unsigned long long);
vector int vec_vpopcnt (vector int);
vector unsigned int vec_vpopcnt (vector int);
vector short vec_vpopcnt (vector short);
vector unsigned short vec_vpopcnt (vector unsigned short);
vector signed char vec_vpopcnt (vector signed char);
vector unsigned char vec_vpopcnt (vector unsigned char);

vector signed char vec_vpopcntb (vector signed char);
vector unsigned char vec_vpopcntb (vector unsigned char);

vector long long vec_vpopcntd (vector long long);
vector unsigned long long vec_vpopcntd (vector unsigned long long);
```

```
vector short vec_vpopcnth (vector short);
vector unsigned short vec_vpopcnth (vector unsigned short);

vector int vec_vpopcntw (vector int);
vector unsigned int vec_vpopcntw (vector int);

vector long long vec_vrld (vector long long, vector unsigned long long);
vector unsigned long long vec_vrld (vector unsigned long long,
                                    vector unsigned long long);

vector long long vec_vsld (vector long long, vector unsigned long long);
vector long long vec_vsld (vector unsigned long long,
                           vector unsigned long long);

vector long long vec_vsrad (vector long long, vector unsigned long long);
vector unsigned long long vec_vsrad (vector unsigned long long,
                                     vector unsigned long long);

vector long long vec_vsrd (vector long long, vector unsigned long long);
vector unsigned long long char vec_vsrd (vector unsigned long long,
                                         vector unsigned long long);

vector long long vec_vsubudm (vector long long, vector long long);
vector long long vec_vsubudm (vector bool long long, vector long long);
vector long long vec_vsubudm (vector long long, vector bool long long);
vector unsigned long long vec_vsubudm (vector unsigned long long,
                                       vector unsigned long long);
vector unsigned long long vec_vsubudm (vector bool long long,
                                       vector unsigned long long);
vector unsigned long long vec_vsubudm (vector unsigned long long,
                                       vector bool long long);

vector long long vec_vupkhsw (vector int);
vector unsigned long long vec_vupkhsw (vector unsigned int);

vector long long vec_vupklsw (vector int);
vector unsigned long long vec_vupklsw (vector int);
```

If the ISA 2.07 additions to the vector/scalar (power8-vector) instruction set are available, the following additional functions are available for 64-bit targets. New vector types (*vector __int128* and *vector __uint128*) are available to hold the *__int128* and *__uint128* types to use these builtins.

The normal vector extract, and set operations work on *vector __int128* and *vector __uint128* types, but the index value must be 0.

```
vector __int128 vec_vaddcuq (vector __int128, vector __int128);
vector __uint128 vec_vaddcuq (vector __uint128, vector __uint128);

vector __int128 vec_vadduqm (vector __int128, vector __int128);
vector __uint128 vec_vadduqm (vector __uint128, vector __uint128);

vector __int128 vec_vaddecuq (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vaddecuq (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vaddeuqm (vector __int128, vector __int128,
                              vector __int128);
```

```
vector __uint128 vec_vaddeuqm (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubecuq (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vsubecuq (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubeuqm (vector __int128, vector __int128,
                              vector __int128);
vector __uint128 vec_vsubeuqm (vector __uint128, vector __uint128,
                               vector __uint128);

vector __int128 vec_vsubcuq (vector __int128, vector __int128);
vector __uint128 vec_vsubcuq (vector __uint128, vector __uint128);

__int128 vec_vsubuqm (__int128, __int128);
__uint128 vec_vsubuqm (__uint128, __uint128);

vector __int128 __builtin_bcdadd (vector __int128, vector __int128, const int);
int __builtin_bcdadd_lt (vector __int128, vector __int128, const int);
int __builtin_bcdadd_eq (vector __int128, vector __int128, const int);
int __builtin_bcdadd_gt (vector __int128, vector __int128, const int);
int __builtin_bcdadd_ov (vector __int128, vector __int128, const int);
vector __int128 __builtin_bcdsub (vector __int128, vector __int128, const int);
int __builtin_bcdsub_lt (vector __int128, vector __int128, const int);
int __builtin_bcdsub_eq (vector __int128, vector __int128, const int);
int __builtin_bcdsub_gt (vector __int128, vector __int128, const int);
int __builtin_bcdsub_ov (vector __int128, vector __int128, const int);
```

6.60.22.4 PowerPC AltiVec Built-in Functions Available on ISA 3.0

The following additional built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 ('`-mcpu=power9`') or later:

```
unsigned int scalar_extract_exp (double source);
unsigned long long int scalar_extract_exp (__ieee128 source);

unsigned long long int scalar_extract_sig (double source);
unsigned __int128 scalar_extract_sig (__ieee128 source);

double scalar_insert_exp (unsigned long long int significand,
                          unsigned long long int exponent);
double scalar_insert_exp (double significand, unsigned long long int exponent);

ieee_128 scalar_insert_exp (unsigned __int128 significand,
                            unsigned long long int exponent);
ieee_128 scalar_insert_exp (ieee_128 significand, unsigned long long int exponent);

int scalar_cmp_exp_gt (double arg1, double arg2);
int scalar_cmp_exp_lt (double arg1, double arg2);
int scalar_cmp_exp_eq (double arg1, double arg2);
int scalar_cmp_exp_unordered (double arg1, double arg2);

bool scalar_test_data_class (float source, const int condition);
bool scalar_test_data_class (double source, const int condition);
bool scalar_test_data_class (__ieee128 source, const int condition);
```

```
bool scalar_test_neg (float source);
bool scalar_test_neg (double source);
bool scalar_test_neg (__ieee128 source);
```

The `scalar_extract_exp` and `scalar_extract_sig` functions require a 64-bit environment supporting ISA 3.0 or later. The `scalar_extract_exp` and `scalar_extract_sig` built-in functions return the significand and the biased exponent value respectively of their `source` arguments. When supplied with a 64-bit `source` argument, the result returned by `scalar_extract_sig` has the `0x0010000000000000` bit set if the function's `source` argument is in normalized form. Otherwise, this bit is set to 0. When supplied with a 128-bit `source` argument, the `0x00010000000000000000000000000000` bit of the result is treated similarly. Note that the sign of the significand is not represented in the result returned from the `scalar_extract_sig` function. Use the `scalar_test_neg` function to test the sign of its `double` argument.

The `scalar_insert_exp` functions require a 64-bit environment supporting ISA 3.0 or later. When supplied with a 64-bit first argument, the `scalar_insert_exp` built-in function returns a double-precision floating point value that is constructed by assembling the values of its `significand` and `exponent` arguments. The sign of the result is copied from the most significant bit of the `significand` argument. The significand and exponent components of the result are composed of the least significant 11 bits of the `exponent` argument and the least significant 52 bits of the `significand` argument respectively.

When supplied with a 128-bit first argument, the `scalar_insert_exp` built-in function returns a quad-precision ieee floating point value. The sign bit of the result is copied from the most significant bit of the `significand` argument. The significand and exponent components of the result are composed of the least significant 15 bits of the `exponent` argument and the least significant 112 bits of the `significand` argument respectively.

The `scalar_cmp_exp_gt`, `scalar_cmp_exp_lt`, `scalar_cmp_exp_eq`, and `scalar_cmp_exp_unordered` built-in functions return a non-zero value if `arg1` is greater than, less than, equal to, or not comparable to `arg2` respectively. The arguments are not comparable if one or the other equals NaN (not a number).

The `scalar_test_data_class` built-in function returns 1 if any of the condition tests enabled by the value of the `condition` variable are true, and 0 otherwise. The `condition` argument must be a compile-time constant integer with value not exceeding 127. The `condition` argument is encoded as a bitmask with each bit enabling the testing of a different condition, as characterized by the following:

```
0x40    Test for NaN
0x20    Test for +Infinity
0x10    Test for -Infinity
0x08    Test for +Zero
0x04    Test for -Zero
0x02    Test for +Denormal
0x01    Test for -Denormal
```

The `scalar_test_neg` built-in function returns 1 if its `source` argument holds a negative value, 0 otherwise.

The following built-in functions are also available for the PowerPC family of processors, starting with ISA 3.0 or later ('`-mcpu=power9`'). These string functions are described separately in order to group the descriptions closer to the function prototypes:

```
int vec_all_nez (vector signed char, vector signed char);
```

```
int vec_all_nez (vector unsigned char, vector unsigned char);
int vec_all_nez (vector signed short, vector signed short);
int vec_all_nez (vector unsigned short, vector unsigned short);
int vec_all_nez (vector signed int, vector signed int);
int vec_all_nez (vector unsigned int, vector unsigned int);

int vec_any_eqz (vector signed char, vector signed char);
int vec_any_eqz (vector unsigned char, vector unsigned char);
int vec_any_eqz (vector signed short, vector signed short);
int vec_any_eqz (vector unsigned short, vector unsigned short);
int vec_any_eqz (vector signed int, vector signed int);
int vec_any_eqz (vector unsigned int, vector unsigned int);

vector bool char vec_cmpnez (vector signed char arg1, vector signed char arg2);
vector bool char vec_cmpnez (vector unsigned char arg1, vector unsigned char arg2);
vector bool short vec_cmpnez (vector signed short arg1, vector signed short arg2);
vector bool short vec_cmpnez (vector unsigned short arg1, vector unsigned short arg2);
vector bool int vec_cmpnez (vector signed int arg1, vector signed int arg2);
vector bool int vec_cmpnez (vector unsigned int, vector unsigned int);

vector signed char vec_cnttz (vector signed char);
vector unsigned char vec_cnttz (vector unsigned char);
vector signed short vec_cnttz (vector signed short);
vector unsigned short vec_cnttz (vector unsigned short);
vector signed int vec_cnttz (vector signed int);
vector unsigned int vec_cnttz (vector unsigned int);
vector signed long long vec_cnttz (vector signed long long);
vector unsigned long long vec_cnttz (vector unsigned long long);

signed int vec_cntlz_lsbb (vector signed char);
signed int vec_cntlz_lsbb (vector unsigned char);

signed int vec_cnttz_lsbb (vector signed char);
signed int vec_cnttz_lsbb (vector unsigned char);

unsigned int vec_first_match_index (vector signed char, vector signed char);
unsigned int vec_first_match_index (vector unsigned char, vector unsigned char);
unsigned int vec_first_match_index (vector signed int, vector signed int);
unsigned int vec_first_match_index (vector unsigned int, vector unsigned int);
unsigned int vec_first_match_index (vector signed short, vector signed short);
unsigned int vec_first_match_index (vector unsigned short, vector unsigned short);
unsigned int vec_first_match_or_eos_index (vector signed char, vector signed char);
unsigned int vec_first_match_or_eos_index (vector unsigned char, vector unsigned char);
unsigned int vec_first_match_or_eos_index (vector signed int, vector signed int);
unsigned int vec_first_match_or_eos_index (vector unsigned int, vector unsigned int);
unsigned int vec_first_match_or_eos_index (vector signed short, vector signed short);
unsigned int vec_first_match_or_eos_index (vector unsigned short,
                                           vector unsigned short);
unsigned int vec_first_mismatch_index (vector signed char, vector signed char);
unsigned int vec_first_mismatch_index (vector unsigned char, vector unsigned char);
unsigned int vec_first_mismatch_index (vector signed int, vector signed int);
unsigned int vec_first_mismatch_index (vector unsigned int, vector unsigned int);
unsigned int vec_first_mismatch_index (vector signed short, vector signed short);
unsigned int vec_first_mismatch_index (vector unsigned short, vector unsigned short);
unsigned int vec_first_mismatch_or_eos_index (vector signed char, vector signed char);
unsigned int vec_first_mismatch_or_eos_index (vector unsigned char,
                                              vector unsigned char);
unsigned int vec_first_mismatch_or_eos_index (vector signed int, vector signed int);
```

```
unsigned int vec_first_mismatch_or_eos_index (vector unsigned int, vector unsigned int);
unsigned int vec_first_mismatch_or_eos_index (vector signed short, vector signed short);
unsigned int vec_first_mismatch_or_eos_index (vector unsigned short,
                                              vector unsigned short);

vector unsigned short vec_pack_to_short_fp32 (vector float, vector float);

vector signed char vec_xl_be (signed long long, signed char *);
vector unsigned char vec_xl_be (signed long long, unsigned char *);
vector signed int vec_xl_be (signed long long, signed int *);
vector unsigned int vec_xl_be (signed long long, unsigned int *);
vector signed __int128 vec_xl_be (signed long long, signed __int128 *);
vector unsigned __int128 vec_xl_be (signed long long, unsigned __int128 *);
vector signed long long vec_xl_be (signed long long, signed long long *);
vector unsigned long long vec_xl_be (signed long long, unsigned long long *);
vector signed short vec_xl_be (signed long long, signed short *);
vector unsigned short vec_xl_be (signed long long, unsigned short *);
vector double vec_xl_be (signed long long, double *);
vector float vec_xl_be (signed long long, float *);

vector signed char vec_xl_len (signed char *addr, size_t len);
vector unsigned char vec_xl_len (unsigned char *addr, size_t len);
vector signed int vec_xl_len (signed int *addr, size_t len);
vector unsigned int vec_xl_len (unsigned int *addr, size_t len);
vector signed __int128 vec_xl_len (signed __int128 *addr, size_t len);
vector unsigned __int128 vec_xl_len (unsigned __int128 *addr, size_t len);
vector signed long long vec_xl_len (signed long long *addr, size_t len);
vector unsigned long long vec_xl_len (unsigned long long *addr, size_t len);
vector signed short vec_xl_len (signed short *addr, size_t len);
vector unsigned short vec_xl_len (unsigned short *addr, size_t len);
vector double vec_xl_len (double *addr, size_t len);
vector float vec_xl_len (float *addr, size_t len);

vector unsigned char vec_xl_len_r (unsigned char *addr, size_t len);

void vec_xst_len (vector signed char data, signed char *addr, size_t len);
void vec_xst_len (vector unsigned char data, unsigned char *addr, size_t len);
void vec_xst_len (vector signed int data, signed int *addr, size_t len);
void vec_xst_len (vector unsigned int data, unsigned int *addr, size_t len);
void vec_xst_len (vector unsigned __int128 data, unsigned __int128 *addr, size_t len);
void vec_xst_len (vector signed long long data, signed long long *addr, size_t len);
void vec_xst_len (vector unsigned long long data, unsigned long long *addr, size_t len);
void vec_xst_len (vector signed short data, signed short *addr, size_t len);
void vec_xst_len (vector unsigned short data, unsigned short *addr, size_t len);
void vec_xst_len (vector signed __int128 data, signed __int128 *addr, size_t len);
void vec_xst_len (vector double data, double *addr, size_t len);
void vec_xst_len (vector float data, float *addr, size_t len);

void vec_xst_len_r (vector unsigned char data, unsigned char *addr, size_t len);

signed char vec_xlx (unsigned int index, vector signed char data);
unsigned char vec_xlx (unsigned int index, vector unsigned char data);
signed short vec_xlx (unsigned int index, vector signed short data);
unsigned short vec_xlx (unsigned int index, vector unsigned short data);
signed int vec_xlx (unsigned int index, vector signed int data);
unsigned int vec_xlx (unsigned int index, vector unsigned int data);
float vec_xlx (unsigned int index, vector float data);
```

```
signed char vec_xrx (unsigned int index, vector signed char data);
unsigned char vec_xrx (unsigned int index, vector unsigned char data);
signed short vec_xrx (unsigned int index, vector signed short data);
unsigned short vec_xrx (unsigned int index, vector unsigned short data);
signed int vec_xrx (unsigned int index, vector signed int data);
unsigned int vec_xrx (unsigned int index, vector unsigned int data);
float vec_xrx (unsigned int index, vector float data);
```

The `vec_all_nez`, `vec_any_eqz`, and `vec_cmpnez` perform pairwise comparisons between the elements at the same positions within their two vector arguments. The `vec_all_nez` function returns a non-zero value if and only if all pairwise comparisons are not equal and no element of either vector argument contains a zero. The `vec_any_eqz` function returns a non-zero value if and only if at least one pairwise comparison is equal or if at least one element of either vector argument contains a zero. The `vec_cmpnez` function returns a vector of the same type as its two arguments, within which each element consists of all ones to denote that either the corresponding elements of the incoming arguments are not equal or that at least one of the corresponding elements contains zero. Otherwise, the element of the returned vector contains all zeros.

The `vec_cntlz_lsbb` function returns the count of the number of consecutive leading byte elements (starting from position 0 within the supplied vector argument) for which the least-significant bit equals zero. The `vec_cnttz_lsbb` function returns the count of the number of consecutive trailing byte elements (starting from position 15 and counting backwards within the supplied vector argument) for which the least-significant bit equals zero.

The `vec_xl_len` and `vec_xst_len` functions require a 64-bit environment supporting ISA 3.0 or later. The `vec_xl_len` function loads a variable length vector from memory. The `vec_xst_len` function stores a variable length vector to memory. With both the `vec_xl_len` and `vec_xst_len` functions, the `addr` argument represents the memory address to or from which data will be transferred, and the `len` argument represents the number of bytes to be transferred, as computed by the C expression `min((len & 0xff), 16)`. If this expression's value is not a multiple of the vector element's size, the behavior of this function is undefined. In the case that the underlying computer is configured to run in big-endian mode, the data transfer moves bytes 0 to `(len - 1)` of the corresponding vector. In little-endian mode, the data transfer moves bytes `(16 - len)` to `15` of the corresponding vector. For the load function, any bytes of the result vector that are not loaded from memory are set to zero. The value of the `addr` argument need not be aligned on a multiple of the vector's element size.

The `vec_xlx` and `vec_xrx` functions extract the single element selected by the `index` argument from the vector represented by the `data` argument. The `index` argument always specifies a byte offset, regardless of the size of the vector element. With `vec_xlx`, `index` is the offset of the first byte of the element to be extracted. With `vec_xrx`, `index` represents the last byte of the element to be extracted, measured from the right end of the vector. In other words, the last byte of the element to be extracted is found at position `(15 - index)`. There is no requirement that `index` be a multiple of the vector element size. However, if the size of the vector element added to `index` is greater than 15, the content of the returned value is undefined.

If the ISA 3.0 instruction set additions ('`-mcpu=power9`') are available:

```
vector unsigned long long vec_bperm (vector unsigned long long, vector unsigned char);
```

```
vector bool char vec_cmpne (vector bool char, vector bool char);
vector bool char vec_cmpne (vector signed char, vector signed char);
vector bool char vec_cmpne (vector unsigned char, vector unsigned char);
vector bool int vec_cmpne (vector bool int, vector bool int);
vector bool int vec_cmpne (vector signed int, vector signed int);
vector bool int vec_cmpne (vector unsigned int, vector unsigned int);
vector bool long long vec_cmpne (vector bool long long, vector bool long long);
vector bool long long vec_cmpne (vector signed long long, vector signed long long);
vector bool long long vec_cmpne (vector unsigned long long, vector unsigned long long);
vector bool short vec_cmpne (vector bool short, vector bool short);
vector bool short vec_cmpne (vector signed short, vector signed short);
vector bool short vec_cmpne (vector unsigned short, vector unsigned short);
vector bool long long vec_cmpne (vector double, vector double);
vector bool int vec_cmpne (vector float, vector float);

vector float vec_extract_fp32_from_shorth (vector unsigned short);
vector float vec_extract_fp32_from_shortl (vector unsigned short);

vector long long vec_vctz (vector long long);
vector unsigned long long vec_vctz (vector unsigned long long);
vector int vec_vctz (vector int);
vector unsigned int vec_vctz (vector int);
vector short vec_vctz (vector short);
vector unsigned short vec_vctz (vector unsigned short);
vector signed char vec_vctz (vector signed char);
vector unsigned char vec_vctz (vector unsigned char);

vector signed char vec_vctzb (vector signed char);
vector unsigned char vec_vctzb (vector unsigned char);

vector long long vec_vctzd (vector long long);
vector unsigned long long vec_vctzd (vector unsigned long long);

vector short vec_vctzh (vector short);
vector unsigned short vec_vctzh (vector unsigned short);

vector int vec_vctzw (vector int);
vector unsigned int vec_vctzw (vector int);

vector unsigned long long vec_extract4b (vector unsigned char, const int);

vector unsigned char vec_insert4b (vector signed int, vector unsigned char,
                                   const int);
vector unsigned char vec_insert4b (vector unsigned int, vector unsigned char,
                                   const int);

vector unsigned int vec_parity_lsbb (vector signed int);
vector unsigned int vec_parity_lsbb (vector unsigned int);
vector unsigned __int128 vec_parity_lsbb (vector signed __int128);
vector unsigned __int128 vec_parity_lsbb (vector unsigned __int128);
vector unsigned long long vec_parity_lsbb (vector signed long long);
vector unsigned long long vec_parity_lsbb (vector unsigned long long);

vector int vec_vprtyb (vector int);
vector unsigned int vec_vprtyb (vector unsigned int);
vector long long vec_vprtyb (vector long long);
vector unsigned long long vec_vprtyb (vector unsigned long long);
```

```
vector int vec_vprtybw (vector int);
vector unsigned int vec_vprtybw (vector unsigned int);

vector long long vec_vprtybd (vector long long);
vector unsigned long long vec_vprtybd (vector unsigned long long);
```

On 64-bit targets, if the ISA 3.0 additions ('`-mcpu=power9`') are available:

```
vector long vec_vprtyb (vector long);
vector unsigned long vec_vprtyb (vector unsigned long);
vector __int128 vec_vprtyb (vector __int128);
vector __uint128 vec_vprtyb (vector __uint128);

vector long vec_vprtybd (vector long);
vector unsigned long vec_vprtybd (vector unsigned long);

vector __int128 vec_vprtybq (vector __int128);
vector __uint128 vec_vprtybd (vector __uint128);
```

The following built-in vector functions are available for the PowerPC family of processors, starting with ISA 3.0 or later ('`-mcpu=power9`'):

```
__vector unsigned char
vec_slv (__vector unsigned char src, __vector unsigned char shift_distance);
__vector unsigned char
vec_srv (__vector unsigned char src, __vector unsigned char shift_distance);
```

The `vec_slv` and `vec_srv` functions operate on all of the bytes of their `src` and `shift_distance` arguments in parallel. The behavior of the `vec_slv` is as if there existed a temporary array of 17 unsigned characters `slv_array` within which elements 0 through 15 are the same as the entries in the `src` array and element 16 equals 0. The result returned from the `vec_slv` function is a `__vector` of 16 unsigned characters within which element `i` is computed using the C expression `0xff & (*((unsigned short *)(slv_array + i)) << (0x07 & shift_distance[i]))`, with this resulting value coerced to the `unsigned char` type. The behavior of the `vec_srv` is as if there existed a temporary array of 17 unsigned characters `srv_array` within which element 0 equals zero and elements 1 through 16 equal the elements 0 through 15 of the `src` array. The result returned from the `vec_srv` function is a `__vector` of 16 unsigned characters within which element `i` is computed using the C expression `0xff & (*((unsigned short *)(srv_array + i)) >> (0x07 & shift_distance[i]))`, with this resulting value coerced to the `unsigned char` type.

The following built-in functions are available for the PowerPC family of processors, starting with ISA 3.0 or later ('`-mcpu=power9`'):

```
__vector unsigned char
vec_absd (__vector unsigned char arg1, __vector unsigned char arg2);
__vector unsigned short
vec_absd (__vector unsigned short arg1, __vector unsigned short arg2);
__vector unsigned int
vec_absd (__vector unsigned int arg1, __vector unsigned int arg2);

__vector unsigned char
vec_absdb (__vector unsigned char arg1, __vector unsigned char arg2);
__vector unsigned short
vec_absdh (__vector unsigned short arg1, __vector unsigned short arg2);
__vector unsigned int
vec_absdw (__vector unsigned int arg1, __vector unsigned int arg2);
```

The `vec_absd`, `vec_absdb`, `vec_absdh`, and `vec_absdw` built-in functions each computes the absolute differences of the pairs of vector elements supplied in its two vector arguments, placing the absolute differences into the corresponding elements of the vector result.

The following built-in functions are available for the PowerPC family of processors, starting with ISA 3.0 or later ('`-mcpu=power9`'):

```
__vector unsigned int vec_extract_exp (__vector float source);
__vector unsigned long long int vec_extract_exp (__vector double source);

__vector unsigned int vec_extract_sig (__vector float source);
__vector unsigned long long int vec_extract_sig (__vector double source);

__vector float vec_insert_exp (__vector unsigned int significands,
                               __vector unsigned int exponents);
__vector float vec_insert_exp (__vector unsigned float significands,
                               __vector unsigned int exponents);
__vector double vec_insert_exp (__vector unsigned long long int significands,
                                __vector unsigned long long int exponents);
__vector double vec_insert_exp (__vector unsigned double significands,
                                __vector unsigned long long int exponents);

__vector bool int vec_test_data_class (__vector float source, const int condition);
__vector bool long long int vec_test_data_class (__vector double source,
                                                 const int condition);
```

The `vec_extract_sig` and `vec_extract_exp` built-in functions return vectors representing the significands and biased exponent values of their `source` arguments respectively. Within the result vector returned by `vec_extract_sig`, the `0x800000` bit of each vector element returned when the function's `source` argument is of type `float` is set to 1 if the corresponding floating point value is in normalized form. Otherwise, this bit is set to 0. When the `source` argument is of type `double`, the `0x10000000000000` bit within each of the result vector's elements is set according to the same rules. Note that the sign of the significand is not represented in the result returned from the `vec_extract_sig` function. To extract the sign bits, use the `vec_cpsgn` function, which returns a new vector within which all of the sign bits of its second argument vector are overwritten with the sign bits copied from the coresponding elements of its first argument vector, and all other (non-sign) bits of the second argument vector are copied unchanged into the result vector.

The `vec_insert_exp` built-in functions return a vector of single- or double-precision floating point values constructed by assembling the values of their `significands` and `exponents` arguments into the corresponding elements of the returned vector. The sign of each element of the result is copied from the most significant bit of the corresponding entry within the `significands` argument. Note that the relevant bits of the `significands` argument are the same, for both integer and floating point types. The significand and exponent components of each element of the result are composed of the least significant bits of the corresponding `significands` element and the least significant bits of the corresponding `exponents` element.

The `vec_test_data_class` built-in function returns a vector representing the results of testing the `source` vector for the condition selected by the `condition` argument. The `condition` argument must be a compile-time constant integer with value not exceeding 127. The `condition` argument is encoded as a bitmask with each bit enabling the testing of a different condition, as characterized by the following:

```
0x40    Test for NaN
0x20    Test for +Infinity
0x10    Test for -Infinity
0x08    Test for +Zero
0x04    Test for -Zero
0x02    Test for +Denormal
0x01    Test for -Denormal
```

If any of the enabled test conditions is true, the corresponding entry in the result vector is -1. Otherwise (all of the enabled test conditions are false), the corresponding entry of the result vector is 0.

The following built-in functions are available for the PowerPC family of processors, starting with ISA 3.0 or later ('`-mcpu=power9`'):

```
vector unsigned int vec_rlmi (vector unsigned int, vector unsigned int,
                              vector unsigned int);
vector unsigned long long vec_rlmi (vector unsigned long long,
                                    vector unsigned long long,
                                    vector unsigned long long);
vector unsigned int vec_rlnm (vector unsigned int, vector unsigned int,
                              vector unsigned int);
vector unsigned long long vec_rlnm (vector unsigned long long,
                                    vector unsigned long long,
                                    vector unsigned long long);
vector unsigned int vec_vrlnm (vector unsigned int, vector unsigned int);
vector unsigned long long vec_vrlnm (vector unsigned long long,
                                     vector unsigned long long);
```

The result of `vec_rlmi` is obtained by rotating each element of the first argument vector left and inserting it under mask into the second argument vector. The third argument vector contains the mask beginning in bits 11:15, the mask end in bits 19:23, and the shift count in bits 27:31, of each element.

The result of `vec_rlnm` is obtained by rotating each element of the first argument vector left and ANDing it with a mask specified by the second and third argument vectors. The second argument vector contains the shift count for each element in the low-order byte. The third argument vector contains the mask end for each element in the low-order byte, with the mask begin in the next higher byte.

The result of `vec_vrlnm` is obtained by rotating each element of the first argument vector left and ANDing it with a mask. The second argument vector contains the mask beginning in bits 11:15, the mask end in bits 19:23, and the shift count in bits 27:31, of each element.

If the ISA 3.0 instruction set additions ('`-mcpu=power9`') are available:

```
vector signed bool char vec_revb (vector signed char);
vector signed char vec_revb (vector signed char);
vector unsigned char vec_revb (vector unsigned char);
vector bool short vec_revb (vector bool short);
vector short vec_revb (vector short);
vector unsigned short vec_revb (vector unsigned short);
vector bool int vec_revb (vector bool int);
vector int vec_revb (vector int);
vector unsigned int vec_revb (vector unsigned int);
vector float vec_revb (vector float);
vector bool long long vec_revb (vector bool long long);
vector long long vec_revb (vector long long);
vector unsigned long long vec_revb (vector unsigned long long);
vector double vec_revb (vector double);
```

On 64-bit targets, if the ISA 3.0 additions ('-mcpu=power9') are available:

```
vector long vec_revb (vector long);
vector unsigned long vec_revb (vector unsigned long);
vector __int128 vec_revb (vector __int128);
vector __uint128 vec_revb (vector __uint128);
```

The vec_revb built-in function reverses the bytes on an element by element basis. A vector of vector unsigned char or vector signed char reverses the bytes in the whole word.

If the cryptographic instructions are enabled ('-mcrypto' or '-mcpu=power8'), the following builtins are enabled.

```
vector unsigned long long __builtin_crypto_vsbox (vector unsigned long long);

vector unsigned char vec_sbox_be (vector unsigned char);

vector unsigned long long __builtin_crypto_vcipher (vector unsigned long long,
                                                    vector unsigned long long);

vector unsigned char vec_cipher_be (vector unsigned char, vector unsigned char);

vector unsigned long long __builtin_crypto_vcipherlast
                                    (vector unsigned long long,
                                     vector unsigned long long);

vector unsigned char vec_cipherlast_be (vector unsigned char,
                                        vector unsigned char);

vector unsigned long long __builtin_crypto_vncipher (vector unsigned long long,
                                                     vector unsigned long long);

vector unsigned char vec_ncipher_be (vector unsigned char,
                                     vector unsigned char);

vector unsigned long long __builtin_crypto_vncipherlast (vector unsigned long long,
                                                         vector unsigned long long);

vector unsigned char vec_ncipherlast_be (vector unsigned char,
                                         vector unsigned char);

vector unsigned char __builtin_crypto_vpermxor (vector unsigned char,
                                                vector unsigned char,
                                                vector unsigned char);

vector unsigned short __builtin_crypto_vpermxor (vector unsigned short,
                                                 vector unsigned short,
                                                 vector unsigned short);

vector unsigned int __builtin_crypto_vpermxor (vector unsigned int,
                                               vector unsigned int,
                                               vector unsigned int);

vector unsigned long long __builtin_crypto_vpermxor (vector unsigned long long,
                                                     vector unsigned long long,
                                                     vector unsigned long long);

vector unsigned char __builtin_crypto_vpmsumb (vector unsigned char,
                                               vector unsigned char);
```

```
vector unsigned short __builtin_crypto_vpmsumb (vector unsigned short,
                                                vector unsigned short);

vector unsigned int __builtin_crypto_vpmsumb (vector unsigned int,
                                              vector unsigned int);

vector unsigned long long __builtin_crypto_vpmsumb (vector unsigned long long,
                                                    vector unsigned long long);

vector unsigned long long __builtin_crypto_vshasigmad (vector unsigned long long,
                                                       int, int);

vector unsigned int __builtin_crypto_vshasigmaw (vector unsigned int, int, int);
```

The second argument to *__builtin_crypto_vshasigmad* and *__builtin_crypto_vshasigmaw* must be a constant integer that is 0 or 1. The third argument to these built-in functions must be a constant integer in the range of 0 to 15.

If the ISA 3.0 instruction set additions are enabled ('`-mcpu=power9`'), the following additional functions are available for both 32-bit and 64-bit targets.

```
vector short vec_xl (int, vector short *);
vector short vec_xl (int, short *);
vector unsigned short vec_xl (int, vector unsigned short *);
vector unsigned short vec_xl (int, unsigned short *);
vector char vec_xl (int, vector char *);
vector char vec_xl (int, char *);
vector unsigned char vec_xl (int, vector unsigned char *);
vector unsigned char vec_xl (int, unsigned char *);

void vec_xst (vector short, int, vector short *);
void vec_xst (vector short, int, short *);
void vec_xst (vector unsigned short, int, vector unsigned short *);
void vec_xst (vector unsigned short, int, unsigned short *);
void vec_xst (vector char, int, vector char *);
void vec_xst (vector char, int, char *);
void vec_xst (vector unsigned char, int, vector unsigned char *);
void vec_xst (vector unsigned char, int, unsigned char *);
```

6.60.23 PowerPC Hardware Transactional Memory Built-in Functions

GCC provides two interfaces for accessing the Hardware Transactional Memory (HTM) instructions available on some of the PowerPC family of processors (eg, POWER8). The two interfaces come in a low level interface, consisting of built-in functions specific to PowerPC and a higher level interface consisting of inline functions that are common between PowerPC and S/390.

6.60.23.1 PowerPC HTM Low Level Built-in Functions

The following low level built-in functions are available with '`-mhtm`' or '`-mcpu=CPU`' where CPU is 'power8' or later. They all generate the machine instruction that is part of the name.

The HTM builtins (with the exception of `__builtin_tbegin`) return the full 4-bit condition register value set by their associated hardware instruction. The header file `htmintrin.h` defines some macros that can be used to decipher the return value. The `__builtin_tbegin`

builtin returns a simple `true` or `false` value depending on whether a transaction was successfully started or not. The arguments of the builtins match exactly the type and order of the associated hardware instruction's operands, except for the `__builtin_tcheck` builtin, which does not take any input arguments. Refer to the ISA manual for a description of each instruction's operands.

```
unsigned int __builtin_tbegin (unsigned int)
unsigned int __builtin_tend (unsigned int)

unsigned int __builtin_tabort (unsigned int)
unsigned int __builtin_tabortdc (unsigned int, unsigned int, unsigned int)
unsigned int __builtin_tabortdci (unsigned int, unsigned int, int)
unsigned int __builtin_tabortwc (unsigned int, unsigned int, unsigned int)
unsigned int __builtin_tabortwci (unsigned int, unsigned int, int)

unsigned int __builtin_tcheck (void)
unsigned int __builtin_treclaim (unsigned int)
unsigned int __builtin_trechkpt (void)
unsigned int __builtin_tsr (unsigned int)
```

In addition to the above HTM built-ins, we have added built-ins for some common extended mnemonics of the HTM instructions:

```
unsigned int __builtin_tendall (void)
unsigned int __builtin_tresume (void)
unsigned int __builtin_tsuspend (void)
```

Note that the semantics of the above HTM builtins are required to mimic the locking semantics used for critical sections. Builtins that are used to create a new transaction or restart a suspended transaction must have lock acquisition like semantics while those builtins that end or suspend a transaction must have lock release like semantics. Specifically, this must mimic lock semantics as specified by C++11, for example: Lock acquisition is as-if an execution of __atomic_exchange_n(&globallock,1,__ATOMIC_ACQUIRE) that returns 0, and lock release is as-if an execution of __atomic_store(&globallock,0,__ATOMIC_RELEASE), with globallock being an implicit implementation-defined lock used for all transactions. The HTM instructions associated with with the builtins inherently provide the correct acquisition and release hardware barriers required. However, the compiler must also be prohibited from moving loads and stores across the builtins in a way that would violate their semantics. This has been accomplished by adding memory barriers to the associated HTM instructions (which is a conservative approach to provide acquire and release semantics). Earlier versions of the compiler did not treat the HTM instructions as memory barriers. A `__TM_FENCE__` macro has been added, which can be used to determine whether the current compiler treats HTM instructions as memory barriers or not. This allows the user to explicitly add memory barriers to their code when using an older version of the compiler.

The following set of built-in functions are available to gain access to the HTM specific special purpose registers.

```
unsigned long __builtin_get_texasr (void)
unsigned long __builtin_get_texasru (void)
unsigned long __builtin_get_tfhar (void)
unsigned long __builtin_get_tfiar (void)

void __builtin_set_texasr (unsigned long);
```

```
void __builtin_set_texasru (unsigned long);
void __builtin_set_tfhar (unsigned long);
void __builtin_set_tfiar (unsigned long);
```

Example usage of these low level built-in functions may look like:

```
#include <htmintrin.h>

int num_retries = 10;

while (1)
  {
    if (__builtin_tbegin (0))
      {
        /* Transaction State Initiated.  */
        if (is_locked (lock))
          __builtin_tabort (0);
        ... transaction code...
        __builtin_tend (0);
        break;
      }
    else
      {
        /* Transaction State Failed.  Use locks if the transaction
           failure is "persistent" or we've tried too many times.  */
        if (num_retries-- <= 0
            || _TEXASRU_FAILURE_PERSISTENT (__builtin_get_texasru ()))
          {
            acquire_lock (lock);
            ... non transactional fallback path...
            release_lock (lock);
            break;
          }
      }
  }
```

One final built-in function has been added that returns the value of the 2-bit Transaction State field of the Machine Status Register (MSR) as stored in `CR0`.

```
unsigned long __builtin_ttest (void)
```

This built-in can be used to determine the current transaction state using the following code example:

```
#include <htmintrin.h>

unsigned char tx_state = _HTM_STATE (__builtin_ttest ());

if (tx_state == _HTM_TRANSACTIONAL)
  {
    /* Code to use in transactional state.  */
  }
else if (tx_state == _HTM_NONTRANSACTIONAL)
  {
    /* Code to use in non-transactional state.  */
  }
else if (tx_state == _HTM_SUSPENDED)
  {
    /* Code to use in transaction suspended state.  */
  }
```

6.60.23.2 PowerPC HTM High Level Inline Functions

The following high level HTM interface is made available by including `<htmxlintrin.h>` and using '`-mhtm`' or '`-mcpu=CPU`' where CPU is 'power8' or later. This interface is common between PowerPC and S/390, allowing users to write one HTM source implementation that can be compiled and executed on either system.

```
long __TM_simple_begin (void)
long __TM_begin (void* const TM_buff)
long __TM_end (void)
void __TM_abort (void)
void __TM_named_abort (unsigned char const code)
void __TM_resume (void)
void __TM_suspend (void)

long __TM_is_user_abort (void* const TM_buff)
long __TM_is_named_user_abort (void* const TM_buff, unsigned char *code)
long __TM_is_illegal (void* const TM_buff)
long __TM_is_footprint_exceeded (void* const TM_buff)
long __TM_nesting_depth (void* const TM_buff)
long __TM_is_nested_too_deep(void* const TM_buff)
long __TM_is_conflict(void* const TM_buff)
long __TM_is_failure_persistent(void* const TM_buff)
long __TM_failure_address(void* const TM_buff)
long long __TM_failure_code(void* const TM_buff)
```

Using these common set of HTM inline functions, we can create a more portable version of the HTM example in the previous section that will work on either PowerPC or S/390:

```
#include <htmxlintrin.h>

int num_retries = 10;
TM_buff_type TM_buff;

while (1)
  {
    if (__TM_begin (TM_buff) == _HTM_TBEGIN_STARTED)
      {
        /* Transaction State Initiated.  */
        if (is_locked (lock))
          __TM_abort ();
        ... transaction code...
        __TM_end ();
        break;
      }
    else
      {
        /* Transaction State Failed.  Use locks if the transaction
           failure is "persistent" or we've tried too many times.  */
        if (num_retries-- <= 0
            || __TM_is_failure_persistent (TM_buff))
          {
            acquire_lock (lock);
            ... non transactional fallback path...
            release_lock (lock);
            break;
          }
      }
  }
```

6.60.24 PowerPC Atomic Memory Operation Functions

ISA 3.0 of the PowerPC added new atomic memory operation (amo) instructions. GCC provides support for these instructions in 64-bit environments. All of the functions are declared in the include file `amo.h`.

The functions supported are:

```
#include <amo.h>

uint32_t amo_lwat_add (uint32_t *, uint32_t);
uint32_t amo_lwat_xor (uint32_t *, uint32_t);
uint32_t amo_lwat_ior (uint32_t *, uint32_t);
uint32_t amo_lwat_and (uint32_t *, uint32_t);
uint32_t amo_lwat_umax (uint32_t *, uint32_t);
uint32_t amo_lwat_umin (uint32_t *, uint32_t);
uint32_t amo_lwat_swap (uint32_t *, uint32_t);

int32_t amo_lwat_sadd (int32_t *, int32_t);
int32_t amo_lwat_smax (int32_t *, int32_t);
int32_t amo_lwat_smin (int32_t *, int32_t);
int32_t amo_lwat_sswap (int32_t *, int32_t);

uint64_t amo_ldat_add (uint64_t *, uint64_t);
uint64_t amo_ldat_xor (uint64_t *, uint64_t);
uint64_t amo_ldat_ior (uint64_t *, uint64_t);
uint64_t amo_ldat_and (uint64_t *, uint64_t);
uint64_t amo_ldat_umax (uint64_t *, uint64_t);
uint64_t amo_ldat_umin (uint64_t *, uint64_t);
uint64_t amo_ldat_swap (uint64_t *, uint64_t);

int64_t amo_ldat_sadd (int64_t *, int64_t);
int64_t amo_ldat_smax (int64_t *, int64_t);
int64_t amo_ldat_smin (int64_t *, int64_t);
int64_t amo_ldat_sswap (int64_t *, int64_t);

void amo_stwat_add (uint32_t *, uint32_t);
void amo_stwat_xor (uint32_t *, uint32_t);
void amo_stwat_ior (uint32_t *, uint32_t);
void amo_stwat_and (uint32_t *, uint32_t);
void amo_stwat_umax (uint32_t *, uint32_t);
void amo_stwat_umin (uint32_t *, uint32_t);

void amo_stwat_sadd (int32_t *, int32_t);
void amo_stwat_smax (int32_t *, int32_t);
void amo_stwat_smin (int32_t *, int32_t);

void amo_stdat_add (uint64_t *, uint64_t);
void amo_stdat_xor (uint64_t *, uint64_t);
void amo_stdat_ior (uint64_t *, uint64_t);
void amo_stdat_and (uint64_t *, uint64_t);
void amo_stdat_umax (uint64_t *, uint64_t);
void amo_stdat_umin (uint64_t *, uint64_t);

void amo_stdat_sadd (int64_t *, int64_t);
void amo_stdat_smax (int64_t *, int64_t);
void amo_stdat_smin (int64_t *, int64_t);
```

6.60.25 RX Built-in Functions

GCC supports some of the RX instructions which cannot be expressed in the C programming language via the use of built-in functions. The following functions are supported:

`void __builtin_rx_brk` (*void*) [Built-in Function]
Generates the `brk` machine instruction.

`void __builtin_rx_clrpsw` (*int*) [Built-in Function]
Generates the `clrpsw` machine instruction to clear the specified bit in the processor status word.

`void __builtin_rx_int` (*int*) [Built-in Function]
Generates the `int` machine instruction to generate an interrupt with the specified value.

`void __builtin_rx_machi` (*int, int*) [Built-in Function]
Generates the `machi` machine instruction to add the result of multiplying the top 16 bits of the two arguments into the accumulator.

`void __builtin_rx_maclo` (*int, int*) [Built-in Function]
Generates the `maclo` machine instruction to add the result of multiplying the bottom 16 bits of the two arguments into the accumulator.

`void __builtin_rx_mulhi` (*int, int*) [Built-in Function]
Generates the `mulhi` machine instruction to place the result of multiplying the top 16 bits of the two arguments into the accumulator.

`void __builtin_rx_mullo` (*int, int*) [Built-in Function]
Generates the `mullo` machine instruction to place the result of multiplying the bottom 16 bits of the two arguments into the accumulator.

`int __builtin_rx_mvfachi` (*void*) [Built-in Function]
Generates the `mvfachi` machine instruction to read the top 32 bits of the accumulator.

`int __builtin_rx_mvfacmi` (*void*) [Built-in Function]
Generates the `mvfacmi` machine instruction to read the middle 32 bits of the accumulator.

`int __builtin_rx_mvfc` (*int*) [Built-in Function]
Generates the `mvfc` machine instruction which reads the control register specified in its argument and returns its value.

`void __builtin_rx_mvtachi` (*int*) [Built-in Function]
Generates the `mvtachi` machine instruction to set the top 32 bits of the accumulator.

`void __builtin_rx_mvtaclo` (*int*) [Built-in Function]
Generates the `mvtaclo` machine instruction to set the bottom 32 bits of the accumulator.

`void __builtin_rx_mvtc` (*int reg, int val*) [Built-in Function]
Generates the `mvtc` machine instruction which sets control register number `reg` to `val`.

`void __builtin_rx_mvtipl (int)` [Built-in Function]
Generates the `mvtipl` machine instruction set the interrupt priority level.

`void __builtin_rx_racw (int)` [Built-in Function]
Generates the `racw` machine instruction to round the accumulator according to the specified mode.

`int __builtin_rx_revw (int)` [Built-in Function]
Generates the `revw` machine instruction which swaps the bytes in the argument so that bits 0–7 now occupy bits 8–15 and vice versa, and also bits 16–23 occupy bits 24–31 and vice versa.

`void __builtin_rx_rmpa (void)` [Built-in Function]
Generates the `rmpa` machine instruction which initiates a repeated multiply and accumulate sequence.

`void __builtin_rx_round (float)` [Built-in Function]
Generates the `round` machine instruction which returns the floating-point argument rounded according to the current rounding mode set in the floating-point status word register.

`int __builtin_rx_sat (int)` [Built-in Function]
Generates the `sat` machine instruction which returns the saturated value of the argument.

`void __builtin_rx_setpsw (int)` [Built-in Function]
Generates the `setpsw` machine instruction to set the specified bit in the processor status word.

`void __builtin_rx_wait (void)` [Built-in Function]
Generates the `wait` machine instruction.

6.60.26 S/390 System z Built-in Functions

`int __builtin_tbegin (void*)` [Built-in Function]
Generates the `tbegin` machine instruction starting a non-constrained hardware transaction. If the parameter is non-NULL the memory area is used to store the transaction diagnostic buffer and will be passed as first operand to `tbegin`. This buffer can be defined using the `struct __htm_tdb` C struct defined in `htmintrin.h` and must reside on a double-word boundary. The second tbegin operand is set to `0xff0c`. This enables save/restore of all GPRs and disables aborts for FPR and AR manipulations inside the transaction body. The condition code set by the tbegin instruction is returned as integer value. The tbegin instruction by definition overwrites the content of all FPRs. The compiler will generate code which saves and restores the FPRs. For soft-float code it is recommended to used the `*_nofloat` variant. In order to prevent a TDB from being written it is required to pass a constant zero value as parameter. Passing a zero value through a variable is not sufficient. Although modifications of access registers inside the transaction will not trigger an transaction abort it is not supported to actually modify them. Access registers do not get saved when entering a transaction. They will have undefined state when reaching the abort code.

Macros for the possible return codes of tbegin are defined in the `htmintrin.h` header file:

`_HTM_TBEGIN_STARTED`
: `tbegin` has been executed as part of normal processing. The transaction body is supposed to be executed.

`_HTM_TBEGIN_INDETERMINATE`
: The transaction was aborted due to an indeterminate condition which might be persistent.

`_HTM_TBEGIN_TRANSIENT`
: The transaction aborted due to a transient failure. The transaction should be re-executed in that case.

`_HTM_TBEGIN_PERSISTENT`
: The transaction aborted due to a persistent failure. Re-execution under same circumstances will not be productive.

`_HTM_FIRST_USER_ABORT_CODE` [Macro]
The `_HTM_FIRST_USER_ABORT_CODE` defined in `htmintrin.h` specifies the first abort code which can be used for `__builtin_tabort`. Values below this threshold are reserved for machine use.

`struct __htm_tdb` [Data type]
The `struct __htm_tdb` defined in `htmintrin.h` describes the structure of the transaction diagnostic block as specified in the Principles of Operation manual chapter 5-91.

`int __builtin_tbegin_nofloat` (*void**) [Built-in Function]
Same as `__builtin_tbegin` but without FPR saves and restores. Using this variant in code making use of FPRs will leave the FPRs in undefined state when entering the transaction abort handler code.

`int __builtin_tbegin_retry` (*void*, int*) [Built-in Function]
In addition to `__builtin_tbegin` a loop for transient failures is generated. If tbegin returns a condition code of 2 the transaction will be retried as often as specified in the second argument. The perform processor assist instruction is used to tell the CPU about the number of fails so far.

`int __builtin_tbegin_retry_nofloat` (*void*, int*) [Built-in Function]
Same as `__builtin_tbegin_retry` but without FPR saves and restores. Using this variant in code making use of FPRs will leave the FPRs in undefined state when entering the transaction abort handler code.

`void __builtin_tbeginc` (*void*) [Built-in Function]
Generates the `tbeginc` machine instruction starting a constrained hardware transaction. The second operand is set to `0xff08`.

`int __builtin_tend` (*void*) [Built-in Function]
Generates the `tend` machine instruction finishing a transaction and making the changes visible to other threads. The condition code generated by tend is returned as integer value.

`void __builtin_tabort (int)` [Built-in Function]
Generates the `tabort` machine instruction with the specified abort code. Abort codes from 0 through 255 are reserved and will result in an error message.

`void __builtin_tx_assist (int)` [Built-in Function]
Generates the `ppa rX,rY,1` machine instruction. Where the integer parameter is loaded into rX and a value of zero is loaded into rY. The integer parameter specifies the number of times the transaction repeatedly aborted.

`int __builtin_tx_nesting_depth (void)` [Built-in Function]
Generates the `etnd` machine instruction. The current nesting depth is returned as integer value. For a nesting depth of 0 the code is not executed as part of an transaction.

`void __builtin_non_tx_store (uint64_t *, uint64_t)` [Built-in Function]
Generates the `ntstg` machine instruction. The second argument is written to the first arguments location. The store operation will not be rolled-back in case of an transaction abort.

6.60.27 SH Built-in Functions

The following built-in functions are supported on the SH1, SH2, SH3 and SH4 families of processors:

`void __builtin_set_thread_pointer (void *ptr)` [Built-in Function]
Sets the '`GBR`' register to the specified value *ptr*. This is usually used by system code that manages threads and execution contexts. The compiler normally does not generate code that modifies the contents of '`GBR`' and thus the value is preserved across function calls. Changing the '`GBR`' value in user code must be done with caution, since the compiler might use '`GBR`' in order to access thread local variables.

`void * __builtin_thread_pointer (void)` [Built-in Function]
Returns the value that is currently set in the '`GBR`' register. Memory loads and stores that use the thread pointer as a base address are turned into '`GBR`' based displacement loads and stores, if possible. For example:

```
struct my_tcb
{
   int a, b, c, d, e;
};

int get_tcb_value (void)
{
  // Generate 'mov.l @(8,gbr),r0' instruction
  return ((my_tcb*)__builtin_thread_pointer ())->c;
}
```

`unsigned int __builtin_sh_get_fpscr (void)` [Built-in Function]
Returns the value that is currently set in the '`FPSCR`' register.

`void __builtin_sh_set_fpscr (unsigned int val)` [Built-in Function]
Sets the '`FPSCR`' register to the specified value *val*, while preserving the current values of the FR, SZ and PR bits.

6.60.28 SPARC VIS Built-in Functions

GCC supports SIMD operations on the SPARC using both the generic vector extensions (see Section 6.52 [Vector Extensions], page 626) as well as built-in functions for the SPARC Visual Instruction Set (VIS). When you use the '-mvis' switch, the VIS extension is exposed as the following built-in functions:

```
typedef int v1si __attribute__ ((vector_size (4)));
typedef int v2si __attribute__ ((vector_size (8)));
typedef short v4hi __attribute__ ((vector_size (8)));
typedef short v2hi __attribute__ ((vector_size (4)));
typedef unsigned char v8qi __attribute__ ((vector_size (8)));
typedef unsigned char v4qi __attribute__ ((vector_size (4)));

void __builtin_vis_write_gsr (int64_t);
int64_t __builtin_vis_read_gsr (void);

void * __builtin_vis_alignaddr (void *, long);
void * __builtin_vis_alignaddrl (void *, long);
int64_t __builtin_vis_falignadatadi (int64_t, int64_t);
v2si __builtin_vis_falignadatav2si (v2si, v2si);
v4hi __builtin_vis_falignadatav4hi (v4si, v4si);
v8qi __builtin_vis_falignadatav8qi (v8qi, v8qi);

v4hi __builtin_vis_fexpand (v4qi);

v4hi __builtin_vis_fmul8x16 (v4qi, v4hi);
v4hi __builtin_vis_fmul8x16au (v4qi, v2hi);
v4hi __builtin_vis_fmul8x16al (v4qi, v2hi);
v4hi __builtin_vis_fmul8sux16 (v8qi, v4hi);
v4hi __builtin_vis_fmul8ulx16 (v8qi, v4hi);
v2si __builtin_vis_fmuld8sux16 (v4qi, v2hi);
v2si __builtin_vis_fmuld8ulx16 (v4qi, v2hi);

v4qi __builtin_vis_fpack16 (v4hi);
v8qi __builtin_vis_fpack32 (v2si, v8qi);
v2hi __builtin_vis_fpackfix (v2si);
v8qi __builtin_vis_fpmerge (v4qi, v4qi);

int64_t __builtin_vis_pdist (v8qi, v8qi, int64_t);

long __builtin_vis_edge8 (void *, void *);
long __builtin_vis_edge8l (void *, void *);
long __builtin_vis_edge16 (void *, void *);
long __builtin_vis_edge16l (void *, void *);
long __builtin_vis_edge32 (void *, void *);
long __builtin_vis_edge32l (void *, void *);

long __builtin_vis_fcmple16 (v4hi, v4hi);
long __builtin_vis_fcmple32 (v2si, v2si);
long __builtin_vis_fcmpne16 (v4hi, v4hi);
long __builtin_vis_fcmpne32 (v2si, v2si);
long __builtin_vis_fcmpgt16 (v4hi, v4hi);
long __builtin_vis_fcmpgt32 (v2si, v2si);
long __builtin_vis_fcmpeq16 (v4hi, v4hi);
long __builtin_vis_fcmpeq32 (v2si, v2si);

v4hi __builtin_vis_fpadd16 (v4hi, v4hi);
v2hi __builtin_vis_fpadd16s (v2hi, v2hi);
```

```
v2si __builtin_vis_fpadd32 (v2si, v2si);
v1si __builtin_vis_fpadd32s (v1si, v1si);
v4hi __builtin_vis_fpsub16 (v4hi, v4hi);
v2hi __builtin_vis_fpsub16s (v2hi, v2hi);
v2si __builtin_vis_fpsub32 (v2si, v2si);
v1si __builtin_vis_fpsub32s (v1si, v1si);

long __builtin_vis_array8 (long, long);
long __builtin_vis_array16 (long, long);
long __builtin_vis_array32 (long, long);
```

When you use the '-mvis2' switch, the VIS version 2.0 built-in functions also become available:

```
long __builtin_vis_bmask (long, long);
int64_t __builtin_vis_bshuffledi (int64_t, int64_t);
v2si __builtin_vis_bshufflev2si (v2si, v2si);
v4hi __builtin_vis_bshufflev2si (v4hi, v4hi);
v8qi __builtin_vis_bshufflev2si (v8qi, v8qi);

long __builtin_vis_edge8n (void *, void *);
long __builtin_vis_edge8ln (void *, void *);
long __builtin_vis_edge16n (void *, void *);
long __builtin_vis_edge16ln (void *, void *);
long __builtin_vis_edge32n (void *, void *);
long __builtin_vis_edge32ln (void *, void *);
```

When you use the '-mvis3' switch, the VIS version 3.0 built-in functions also become available:

```
void __builtin_vis_cmask8 (long);
void __builtin_vis_cmask16 (long);
void __builtin_vis_cmask32 (long);

v4hi __builtin_vis_fchksm16 (v4hi, v4hi);

v4hi __builtin_vis_fsll16 (v4hi, v4hi);
v4hi __builtin_vis_fslas16 (v4hi, v4hi);
v4hi __builtin_vis_fsrl16 (v4hi, v4hi);
v4hi __builtin_vis_fsra16 (v4hi, v4hi);
v2si __builtin_vis_fsll16 (v2si, v2si);
v2si __builtin_vis_fslas16 (v2si, v2si);
v2si __builtin_vis_fsrl16 (v2si, v2si);
v2si __builtin_vis_fsra16 (v2si, v2si);

long __builtin_vis_pdistn (v8qi, v8qi);

v4hi __builtin_vis_fmean16 (v4hi, v4hi);

int64_t __builtin_vis_fpadd64 (int64_t, int64_t);
int64_t __builtin_vis_fpsub64 (int64_t, int64_t);

v4hi __builtin_vis_fpadds16 (v4hi, v4hi);
v2hi __builtin_vis_fpadds16s (v2hi, v2hi);
v4hi __builtin_vis_fpsubs16 (v4hi, v4hi);
v2hi __builtin_vis_fpsubs16s (v2hi, v2hi);
v2si __builtin_vis_fpadds32 (v2si, v2si);
v1si __builtin_vis_fpadds32s (v1si, v1si);
v2si __builtin_vis_fpsubs32 (v2si, v2si);
v1si __builtin_vis_fpsubs32s (v1si, v1si);
```

```
long __builtin_vis_fucmple8 (v8qi, v8qi);
long __builtin_vis_fucmpne8 (v8qi, v8qi);
long __builtin_vis_fucmpgt8 (v8qi, v8qi);
long __builtin_vis_fucmpeq8 (v8qi, v8qi);

float __builtin_vis_fhadds (float, float);
double __builtin_vis_fhaddd (double, double);
float __builtin_vis_fhsubs (float, float);
double __builtin_vis_fhsubd (double, double);
float __builtin_vis_fnhadds (float, float);
double __builtin_vis_fnhaddd (double, double);

int64_t __builtin_vis_umulxhi (int64_t, int64_t);
int64_t __builtin_vis_xmulx (int64_t, int64_t);
int64_t __builtin_vis_xmulxhi (int64_t, int64_t);
```

When you use the '`-mvis4`' switch, the VIS version 4.0 built-in functions also become available:

```
v8qi __builtin_vis_fpadd8 (v8qi, v8qi);
v8qi __builtin_vis_fpadds8 (v8qi, v8qi);
v8qi __builtin_vis_fpaddus8 (v8qi, v8qi);
v4hi __builtin_vis_fpaddus16 (v4hi, v4hi);

v8qi __builtin_vis_fpsub8 (v8qi, v8qi);
v8qi __builtin_vis_fpsubs8 (v8qi, v8qi);
v8qi __builtin_vis_fpsubus8 (v8qi, v8qi);
v4hi __builtin_vis_fpsubus16 (v4hi, v4hi);

long __builtin_vis_fpcmple8 (v8qi, v8qi);
long __builtin_vis_fpcmpgt8 (v8qi, v8qi);
long __builtin_vis_fpcmpule16 (v4hi, v4hi);
long __builtin_vis_fpcmpugt16 (v4hi, v4hi);
long __builtin_vis_fpcmpule32 (v2si, v2si);
long __builtin_vis_fpcmpugt32 (v2si, v2si);

v8qi __builtin_vis_fpmax8 (v8qi, v8qi);
v4hi __builtin_vis_fpmax16 (v4hi, v4hi);
v2si __builtin_vis_fpmax32 (v2si, v2si);

v8qi __builtin_vis_fpmaxu8 (v8qi, v8qi);
v4hi __builtin_vis_fpmaxu16 (v4hi, v4hi);
v2si __builtin_vis_fpmaxu32 (v2si, v2si);


v8qi __builtin_vis_fpmin8 (v8qi, v8qi);
v4hi __builtin_vis_fpmin16 (v4hi, v4hi);
v2si __builtin_vis_fpmin32 (v2si, v2si);

v8qi __builtin_vis_fpminu8 (v8qi, v8qi);
v4hi __builtin_vis_fpminu16 (v4hi, v4hi);
v2si __builtin_vis_fpminu32 (v2si, v2si);
```

When you use the '`-mvis4b`' switch, the VIS version 4.0B built-in functions also become available:

```
v8qi __builtin_vis_dictunpack8 (double, int);
v4hi __builtin_vis_dictunpack16 (double, int);
v2si __builtin_vis_dictunpack32 (double, int);
```

```
long __builtin_vis_fpcmple8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpgt8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpeq8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpne8shl (v8qi, v8qi, int);

long __builtin_vis_fpcmple16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpgt16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpeq16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpne16shl (v4hi, v4hi, int);

long __builtin_vis_fpcmple32shl (v2si, v2si, int);
long __builtin_vis_fpcmpgt32shl (v2si, v2si, int);
long __builtin_vis_fpcmpeq32shl (v2si, v2si, int);
long __builtin_vis_fpcmpne32shl (v2si, v2si, int);

long __builtin_vis_fpcmpule8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpugt8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpule16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpugt16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpule32shl (v2si, v2si, int);
long __builtin_vis_fpcmpugt32shl (v2si, v2si, int);

long __builtin_vis_fpcmpde8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpde16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpde32shl (v2si, v2si, int);

long __builtin_vis_fpcmpur8shl (v8qi, v8qi, int);
long __builtin_vis_fpcmpur16shl (v4hi, v4hi, int);
long __builtin_vis_fpcmpur32shl (v2si, v2si, int);
```

6.60.29 SPU Built-in Functions

GCC provides extensions for the SPU processor as described in the Sony/Toshiba/IBM SPU Language Extensions Specification. GCC's implementation differs in several ways.

- The optional extension of specifying vector constants in parentheses is not supported.
- A vector initializer requires no cast if the vector constant is of the same type as the variable it is initializing.
- If `signed` or `unsigned` is omitted, the signedness of the vector type is the default signedness of the base type. The default varies depending on the operating system, so a portable program should always specify the signedness.
- By default, the keyword `__vector` is added. The macro `vector` is defined in `<spu_intrinsics.h>` and can be undefined.
- GCC allows using a `typedef` name as the type specifier for a vector type.
- For C, overloaded functions are implemented with macros so the following does not work:

  ```
  spu_add ((vector signed int){1, 2, 3, 4}, foo);
  ```

  Since `spu_add` is a macro, the vector constant in the example is treated as four separate arguments. Wrap the entire argument in parentheses for this to work.
- The extended version of `__builtin_expect` is not supported.

Note: Only the interface described in the aforementioned specification is supported. Internally, GCC uses built-in functions to implement the required functionality, but these are not supported and are subject to change without notice.

6.60.30 TI C6X Built-in Functions

GCC provides intrinsics to access certain instructions of the TI C6X processors. These intrinsics, listed below, are available after inclusion of the `c6x_intrinsics.h` header file. They map directly to C6X instructions.

```
int _sadd (int, int)
int _ssub (int, int)
int _sadd2 (int, int)
int _ssub2 (int, int)
long long _mpy2 (int, int)
long long _smpy2 (int, int)
int _add4 (int, int)
int _sub4 (int, int)
int _saddu4 (int, int)

int _smpy (int, int)
int _smpyh (int, int)
int _smpyhl (int, int)
int _smpylh (int, int)

int _sshl (int, int)
int _subc (int, int)

int _avg2 (int, int)
int _avgu4 (int, int)

int _clrr (int, int)
int _extr (int, int)
int _extru (int, int)
int _abs (int)
int _abs2 (int)
```

6.60.31 TILE-Gx Built-in Functions

GCC provides intrinsics to access every instruction of the TILE-Gx processor. The intrinsics are of the form:

```
unsigned long long __insn_op (...)
```

Where *op* is the name of the instruction. Refer to the ISA manual for the complete list of instructions.

GCC also provides intrinsics to directly access the network registers. The intrinsics are:

```
unsigned long long __tile_idn0_receive (void)
unsigned long long __tile_idn1_receive (void)
unsigned long long __tile_udn0_receive (void)
unsigned long long __tile_udn1_receive (void)
unsigned long long __tile_udn2_receive (void)
unsigned long long __tile_udn3_receive (void)
```

```
void __tile_idn_send (unsigned long long)
void __tile_udn_send (unsigned long long)
```

The intrinsic `void __tile_network_barrier (void)` is used to guarantee that no network operations before it are reordered with those after it.

6.60.32 TILEPro Built-in Functions

GCC provides intrinsics to access every instruction of the TILEPro processor. The intrinsics are of the form:

```
unsigned __insn_op (...)
```

where *op* is the name of the instruction. Refer to the ISA manual for the complete list of instructions.

GCC also provides intrinsics to directly access the network registers. The intrinsics are:

```
unsigned __tile_idn0_receive (void)
unsigned __tile_idn1_receive (void)
unsigned __tile_sn_receive (void)
unsigned __tile_udn0_receive (void)
unsigned __tile_udn1_receive (void)
unsigned __tile_udn2_receive (void)
unsigned __tile_udn3_receive (void)
void __tile_idn_send (unsigned)
void __tile_sn_send (unsigned)
void __tile_udn_send (unsigned)
```

The intrinsic `void __tile_network_barrier (void)` is used to guarantee that no network operations before it are reordered with those after it.

6.60.33 x86 Built-in Functions

These built-in functions are available for the x86-32 and x86-64 family of computers, depending on the command-line switches used.

If you specify command-line switches such as '`-msse`', the compiler could use the extended instruction sets even if the built-ins are not used explicitly in the program. For this reason, applications that perform run-time CPU detection must compile separate files for each supported architecture, using the appropriate flags. In particular, the file containing the CPU detection code should be compiled without these options.

The following machine modes are available for use with MMX built-in functions (see Section 6.52 [Vector Extensions], page 626): `V2SI` for a vector of two 32-bit integers, `V4HI` for a vector of four 16-bit integers, and `V8QI` for a vector of eight 8-bit integers. Some of the built-in functions operate on MMX registers as a whole 64-bit entity, these use `V1DI` as their mode.

If 3DNow! extensions are enabled, `V2SF` is used as a mode for a vector of two 32-bit floating-point values.

If SSE extensions are enabled, `V4SF` is used for a vector of four 32-bit floating-point values. Some instructions use a vector of four 32-bit integers, these use `V4SI`. Finally, some

instructions operate on an entire vector register, interpreting it as a 128-bit integer, these use mode `TI`.

The x86-32 and x86-64 family of processors use additional built-in functions for efficient use of `TF` (`__float128`) 128-bit floating point and `TC` 128-bit complex floating-point values.

The following floating-point built-in functions are always available. All of them implement the function that is part of the name.

```
__float128 __builtin_fabsq (__float128)
__float128 __builtin_copysignq (__float128, __float128)
```

The following built-in functions are always available.

`__float128 __builtin_infq (void)`
: Similar to `__builtin_inf`, except the return type is `__float128`.

`__float128 __builtin_huge_valq (void)`
: Similar to `__builtin_huge_val`, except the return type is `__float128`.

`__float128 __builtin_nanq (void)`
: Similar to `__builtin_nan`, except the return type is `__float128`.

`__float128 __builtin_nansq (void)`
: Similar to `__builtin_nans`, except the return type is `__float128`.

The following built-in function is always available.

`void __builtin_ia32_pause (void)`
: Generates the `pause` machine instruction with a compiler memory barrier.

The following built-in functions are always available and can be used to check the target platform type.

`void __builtin_cpu_init` (*void*) [Built-in Function]

This function runs the CPU detection code to check the type of CPU and the features supported. This built-in function needs to be invoked along with the built-in functions to check CPU type and features, `__builtin_cpu_is` and `__builtin_cpu_supports`, only when used in a function that is executed before any constructors are called. The CPU detection code is automatically executed in a very high priority constructor.

For example, this function has to be used in `ifunc` resolvers that check for CPU type using the built-in functions `__builtin_cpu_is` and `__builtin_cpu_supports`, or in constructors on targets that don't support constructor priority.

```
static void (*resolve_memcpy (void)) (void)
{
  // ifunc resolvers fire before constructors, explicitly call the init
  // function.
  __builtin_cpu_init ();
  if (__builtin_cpu_supports ("ssse3"))
    return ssse3_memcpy; // super fast memcpy with ssse3 instructions.
  else
    return default_memcpy;
}

void *memcpy (void *, const void *, size_t)
     __attribute__ ((ifunc ("resolve_memcpy")));
```

`int __builtin_cpu_is (const char *cpuname)` [Built-in Function]

This function returns a positive integer if the run-time CPU is of type *cpuname* and returns `0` otherwise. The following CPU names can be detected:

- ‘`amd`’ AMD CPU.
- ‘`intel`’ Intel CPU.
- ‘`atom`’ Intel Atom CPU.
- ‘`slm`’ Intel Silvermont CPU.
- ‘`core2`’ Intel Core 2 CPU.
- ‘`corei7`’ Intel Core i7 CPU.
- ‘`nehalem`’ Intel Core i7 Nehalem CPU.
- ‘`westmere`’ Intel Core i7 Westmere CPU.
- ‘`sandybridge`’ Intel Core i7 Sandy Bridge CPU.
- ‘`ivybridge`’ Intel Core i7 Ivy Bridge CPU.
- ‘`haswell`’ Intel Core i7 Haswell CPU.
- ‘`broadwell`’ Intel Core i7 Broadwell CPU.
- ‘`skylake`’ Intel Core i7 Skylake CPU.
- ‘`skylake-avx512`’ Intel Core i7 Skylake AVX512 CPU.
- ‘`cannonlake`’ Intel Core i7 Cannon Lake CPU.
- ‘`icelake-client`’ Intel Core i7 Ice Lake Client CPU.
- ‘`icelake-server`’ Intel Core i7 Ice Lake Server CPU.
- ‘`cascadelake`’ Intel Core i7 Cascadelake CPU.
- ‘`bonnell`’ Intel Atom Bonnell CPU.
- ‘`silvermont`’ Intel Atom Silvermont CPU.
- ‘`goldmont`’ Intel Atom Goldmont CPU.
- ‘`goldmont-plus`’ Intel Atom Goldmont Plus CPU.

'tremont' Intel Atom Tremont CPU.

'knl' Intel Knights Landing CPU.

'knm' Intel Knights Mill CPU.

'amdfam10h'
AMD Family 10h CPU.

'barcelona'
AMD Family 10h Barcelona CPU.

'shanghai'
AMD Family 10h Shanghai CPU.

'istanbul'
AMD Family 10h Istanbul CPU.

'btver1' AMD Family 14h CPU.

'amdfam15h'
AMD Family 15h CPU.

'bdver1' AMD Family 15h Bulldozer version 1.

'bdver2' AMD Family 15h Bulldozer version 2.

'bdver3' AMD Family 15h Bulldozer version 3.

'bdver4' AMD Family 15h Bulldozer version 4.

'btver2' AMD Family 16h CPU.

'amdfam17h'
AMD Family 17h CPU.

'znver1' AMD Family 17h Zen version 1.

'znver2' AMD Family 17h Zen version 2.

Here is an example:

```
if (__builtin_cpu_is ("corei7"))
  {
     do_corei7 (); // Core i7 specific implementation.
  }
else
  {
     do_generic (); // Generic implementation.
  }
```

`int __builtin_cpu_supports (const char *feature)` [Built-in Function]

This function returns a positive integer if the run-time CPU supports *feature* and returns `0` otherwise. The following features can be detected:

'cmov' CMOV instruction.

'mmx' MMX instructions.

'popcnt' POPCNT instruction.

'sse' SSE instructions.

‘sse2’ SSE2 instructions.

‘sse3’ SSE3 instructions.

‘ssse3’ SSSE3 instructions.

‘sse4.1’ SSE4.1 instructions.

‘sse4.2’ SSE4.2 instructions.

‘avx’ AVX instructions.

‘avx2’ AVX2 instructions.

‘sse4a’ SSE4A instructions.

‘fma4’ FMA4 instructions.

‘xop’ XOP instructions.

‘fma’ FMA instructions.

‘avx512f’ AVX512F instructions.

‘bmi’ BMI instructions.

‘bmi2’ BMI2 instructions.

‘aes’ AES instructions.

‘pclmul’ PCLMUL instructions.

‘avx512vl’
AVX512VL instructions.

‘avx512bw’
AVX512BW instructions.

‘avx512dq’
AVX512DQ instructions.

‘avx512cd’
AVX512CD instructions.

‘avx512er’
AVX512ER instructions.

‘avx512pf’
AVX512PF instructions.

‘avx512vbmi’
AVX512VBMI instructions.

‘avx512ifma’
AVX512IFMA instructions.

‘avx5124vnniw’
AVX5124VNNIW instructions.

‘avx5124fmaps’
AVX5124FMAPS instructions.

'avx512vpopcntdq'
AVX512VPOPCNTDQ instructions.

'avx512vbmi2'
AVX512VBMI2 instructions.

'gfni' GFNI instructions.

'vpclmulqdq'
VPCLMULQDQ instructions.

'avx512vnni'
AVX512VNNI instructions.

'avx512bitalg'
AVX512BITALG instructions.

Here is an example:

```
if (__builtin_cpu_supports ("popcnt"))
  {
     asm("popcnt %1,%0" : "=r"(count) : "rm"(n) : "cc");
  }
else
  {
     count = generic_countbits (n); //generic implementation.
  }
```

The following built-in functions are made available by '-mmmx'. All of them generate the machine instruction that is part of the name.

```
v8qi __builtin_ia32_paddb (v8qi, v8qi)
v4hi __builtin_ia32_paddw (v4hi, v4hi)
v2si __builtin_ia32_paddd (v2si, v2si)
v8qi __builtin_ia32_psubb (v8qi, v8qi)
v4hi __builtin_ia32_psubw (v4hi, v4hi)
v2si __builtin_ia32_psubd (v2si, v2si)
v8qi __builtin_ia32_paddsb (v8qi, v8qi)
v4hi __builtin_ia32_paddsw (v4hi, v4hi)
v8qi __builtin_ia32_psubsb (v8qi, v8qi)
v4hi __builtin_ia32_psubsw (v4hi, v4hi)
v8qi __builtin_ia32_paddusb (v8qi, v8qi)
v4hi __builtin_ia32_paddusw (v4hi, v4hi)
v8qi __builtin_ia32_psubusb (v8qi, v8qi)
v4hi __builtin_ia32_psubusw (v4hi, v4hi)
v4hi __builtin_ia32_pmullw (v4hi, v4hi)
v4hi __builtin_ia32_pmulhw (v4hi, v4hi)
di __builtin_ia32_pand (di, di)
di __builtin_ia32_pandn (di,di)
di __builtin_ia32_por (di, di)
di __builtin_ia32_pxor (di, di)
v8qi __builtin_ia32_pcmpeqb (v8qi, v8qi)
v4hi __builtin_ia32_pcmpeqw (v4hi, v4hi)
v2si __builtin_ia32_pcmpeqd (v2si, v2si)
v8qi __builtin_ia32_pcmpgtb (v8qi, v8qi)
v4hi __builtin_ia32_pcmpgtw (v4hi, v4hi)
v2si __builtin_ia32_pcmpgtd (v2si, v2si)
v8qi __builtin_ia32_punpckhbw (v8qi, v8qi)
v4hi __builtin_ia32_punpckhwd (v4hi, v4hi)
v2si __builtin_ia32_punpckhdq (v2si, v2si)
```

```
v8qi __builtin_ia32_punpcklbw (v8qi, v8qi)
v4hi __builtin_ia32_punpcklwd (v4hi, v4hi)
v2si __builtin_ia32_punpckldq (v2si, v2si)
v8qi __builtin_ia32_packsswb (v4hi, v4hi)
v4hi __builtin_ia32_packssdw (v2si, v2si)
v8qi __builtin_ia32_packuswb (v4hi, v4hi)

v4hi __builtin_ia32_psllw (v4hi, v4hi)
v2si __builtin_ia32_pslld (v2si, v2si)
v1di __builtin_ia32_psllq (v1di, v1di)
v4hi __builtin_ia32_psrlw (v4hi, v4hi)
v2si __builtin_ia32_psrld (v2si, v2si)
v1di __builtin_ia32_psrlq (v1di, v1di)
v4hi __builtin_ia32_psraw (v4hi, v4hi)
v2si __builtin_ia32_psrad (v2si, v2si)
v4hi __builtin_ia32_psllwi (v4hi, int)
v2si __builtin_ia32_pslldi (v2si, int)
v1di __builtin_ia32_psllqi (v1di, int)
v4hi __builtin_ia32_psrlwi (v4hi, int)
v2si __builtin_ia32_psrldi (v2si, int)
v1di __builtin_ia32_psrlqi (v1di, int)
v4hi __builtin_ia32_psrawi (v4hi, int)
v2si __builtin_ia32_psradi (v2si, int)
```

The following built-in functions are made available either with '`-msse`', or with '`-m3dnowa`'. All of them generate the machine instruction that is part of the name.

```
v4hi __builtin_ia32_pmulhuw (v4hi, v4hi)
v8qi __builtin_ia32_pavgb (v8qi, v8qi)
v4hi __builtin_ia32_pavgw (v4hi, v4hi)
v1di __builtin_ia32_psadbw (v8qi, v8qi)
v8qi __builtin_ia32_pmaxub (v8qi, v8qi)
v4hi __builtin_ia32_pmaxsw (v4hi, v4hi)
v8qi __builtin_ia32_pminub (v8qi, v8qi)
v4hi __builtin_ia32_pminsw (v4hi, v4hi)
int __builtin_ia32_pmovmskb (v8qi)
void __builtin_ia32_maskmovq (v8qi, v8qi, char *)
void __builtin_ia32_movntq (di *, di)
void __builtin_ia32_sfence (void)
```

The following built-in functions are available when '`-msse`' is used. All of them generate the machine instruction that is part of the name.

```
int __builtin_ia32_comieq (v4sf, v4sf)
int __builtin_ia32_comineq (v4sf, v4sf)
int __builtin_ia32_comilt (v4sf, v4sf)
int __builtin_ia32_comile (v4sf, v4sf)
int __builtin_ia32_comigt (v4sf, v4sf)
int __builtin_ia32_comige (v4sf, v4sf)
int __builtin_ia32_ucomieq (v4sf, v4sf)
int __builtin_ia32_ucomineq (v4sf, v4sf)
int __builtin_ia32_ucomilt (v4sf, v4sf)
int __builtin_ia32_ucomile (v4sf, v4sf)
int __builtin_ia32_ucomigt (v4sf, v4sf)
int __builtin_ia32_ucomige (v4sf, v4sf)
v4sf __builtin_ia32_addps (v4sf, v4sf)
v4sf __builtin_ia32_subps (v4sf, v4sf)
v4sf __builtin_ia32_mulps (v4sf, v4sf)
v4sf __builtin_ia32_divps (v4sf, v4sf)
```

```
v4sf __builtin_ia32_addss (v4sf, v4sf)
v4sf __builtin_ia32_subss (v4sf, v4sf)
v4sf __builtin_ia32_mulss (v4sf, v4sf)
v4sf __builtin_ia32_divss (v4sf, v4sf)
v4sf __builtin_ia32_cmpeqps (v4sf, v4sf)
v4sf __builtin_ia32_cmpltps (v4sf, v4sf)
v4sf __builtin_ia32_cmpleps (v4sf, v4sf)
v4sf __builtin_ia32_cmpgtps (v4sf, v4sf)
v4sf __builtin_ia32_cmpgeps (v4sf, v4sf)
v4sf __builtin_ia32_cmpunordps (v4sf, v4sf)
v4sf __builtin_ia32_cmpneqps (v4sf, v4sf)
v4sf __builtin_ia32_cmpnltps (v4sf, v4sf)
v4sf __builtin_ia32_cmpnleps (v4sf, v4sf)
v4sf __builtin_ia32_cmpngtps (v4sf, v4sf)
v4sf __builtin_ia32_cmpngeps (v4sf, v4sf)
v4sf __builtin_ia32_cmpordps (v4sf, v4sf)
v4sf __builtin_ia32_cmpeqss (v4sf, v4sf)
v4sf __builtin_ia32_cmpltss (v4sf, v4sf)
v4sf __builtin_ia32_cmpless (v4sf, v4sf)
v4sf __builtin_ia32_cmpunordss (v4sf, v4sf)
v4sf __builtin_ia32_cmpneqss (v4sf, v4sf)
v4sf __builtin_ia32_cmpnltss (v4sf, v4sf)
v4sf __builtin_ia32_cmpnless (v4sf, v4sf)
v4sf __builtin_ia32_cmpordss (v4sf, v4sf)
v4sf __builtin_ia32_maxps (v4sf, v4sf)
v4sf __builtin_ia32_maxss (v4sf, v4sf)
v4sf __builtin_ia32_minps (v4sf, v4sf)
v4sf __builtin_ia32_minss (v4sf, v4sf)
v4sf __builtin_ia32_andps (v4sf, v4sf)
v4sf __builtin_ia32_andnps (v4sf, v4sf)
v4sf __builtin_ia32_orps (v4sf, v4sf)
v4sf __builtin_ia32_xorps (v4sf, v4sf)
v4sf __builtin_ia32_movss (v4sf, v4sf)
v4sf __builtin_ia32_movhlps (v4sf, v4sf)
v4sf __builtin_ia32_movlhps (v4sf, v4sf)
v4sf __builtin_ia32_unpckhps (v4sf, v4sf)
v4sf __builtin_ia32_unpcklps (v4sf, v4sf)
v4sf __builtin_ia32_cvtpi2ps (v4sf, v2si)
v4sf __builtin_ia32_cvtsi2ss (v4sf, int)
v2si __builtin_ia32_cvtps2pi (v4sf)
int __builtin_ia32_cvtss2si (v4sf)
v2si __builtin_ia32_cvttps2pi (v4sf)
int __builtin_ia32_cvttss2si (v4sf)
v4sf __builtin_ia32_rcpps (v4sf)
v4sf __builtin_ia32_rsqrtps (v4sf)
v4sf __builtin_ia32_sqrtps (v4sf)
v4sf __builtin_ia32_rcpss (v4sf)
v4sf __builtin_ia32_rsqrtss (v4sf)
v4sf __builtin_ia32_sqrtss (v4sf)
v4sf __builtin_ia32_shufps (v4sf, v4sf, int)
void __builtin_ia32_movntps (float *, v4sf)
int __builtin_ia32_movmskps (v4sf)
```

The following built-in functions are available when '`-msse`' is used.

`v4sf __builtin_ia32_loadups (float *)`

Generates the `movups` machine instruction as a load from memory.

`void __builtin_ia32_storeups (float *, v4sf)`
Generates the `movups` machine instruction as a store to memory.

`v4sf __builtin_ia32_loadss (float *)`
Generates the `movss` machine instruction as a load from memory.

`v4sf __builtin_ia32_loadhps (v4sf, const v2sf *)`
Generates the `movhps` machine instruction as a load from memory.

`v4sf __builtin_ia32_loadlps (v4sf, const v2sf *)`
Generates the `movlps` machine instruction as a load from memory

`void __builtin_ia32_storehps (v2sf *, v4sf)`
Generates the `movhps` machine instruction as a store to memory.

`void __builtin_ia32_storelps (v2sf *, v4sf)`
Generates the `movlps` machine instruction as a store to memory.

The following built-in functions are available when '`-msse2`' is used. All of them generate the machine instruction that is part of the name.

```
int __builtin_ia32_comisdeq (v2df, v2df)
int __builtin_ia32_comisdlt (v2df, v2df)
int __builtin_ia32_comisdle (v2df, v2df)
int __builtin_ia32_comisdgt (v2df, v2df)
int __builtin_ia32_comisdge (v2df, v2df)
int __builtin_ia32_comisdneq (v2df, v2df)
int __builtin_ia32_ucomisdeq (v2df, v2df)
int __builtin_ia32_ucomisdlt (v2df, v2df)
int __builtin_ia32_ucomisdle (v2df, v2df)
int __builtin_ia32_ucomisdgt (v2df, v2df)
int __builtin_ia32_ucomisdge (v2df, v2df)
int __builtin_ia32_ucomisdneq (v2df, v2df)
v2df __builtin_ia32_cmpeqpd (v2df, v2df)
v2df __builtin_ia32_cmpltpd (v2df, v2df)
v2df __builtin_ia32_cmplepd (v2df, v2df)
v2df __builtin_ia32_cmpgtpd (v2df, v2df)
v2df __builtin_ia32_cmpgepd (v2df, v2df)
v2df __builtin_ia32_cmpunordpd (v2df, v2df)
v2df __builtin_ia32_cmpneqpd (v2df, v2df)
v2df __builtin_ia32_cmpnltpd (v2df, v2df)
v2df __builtin_ia32_cmpnlepd (v2df, v2df)
v2df __builtin_ia32_cmpngtpd (v2df, v2df)
v2df __builtin_ia32_cmpngepd (v2df, v2df)
v2df __builtin_ia32_cmpordpd (v2df, v2df)
v2df __builtin_ia32_cmpeqsd (v2df, v2df)
v2df __builtin_ia32_cmpltsd (v2df, v2df)
v2df __builtin_ia32_cmplesd (v2df, v2df)
v2df __builtin_ia32_cmpunordsd (v2df, v2df)
v2df __builtin_ia32_cmpneqsd (v2df, v2df)
v2df __builtin_ia32_cmpnltsd (v2df, v2df)
v2df __builtin_ia32_cmpnlesd (v2df, v2df)
v2df __builtin_ia32_cmpordsd (v2df, v2df)
v2di __builtin_ia32_paddq (v2di, v2di)
v2di __builtin_ia32_psubq (v2di, v2di)
v2df __builtin_ia32_addpd (v2df, v2df)
v2df __builtin_ia32_subpd (v2df, v2df)
v2df __builtin_ia32_mulpd (v2df, v2df)
v2df __builtin_ia32_divpd (v2df, v2df)
```

```
v2df __builtin_ia32_addsd (v2df, v2df)
v2df __builtin_ia32_subsd (v2df, v2df)
v2df __builtin_ia32_mulsd (v2df, v2df)
v2df __builtin_ia32_divsd (v2df, v2df)
v2df __builtin_ia32_minpd (v2df, v2df)
v2df __builtin_ia32_maxpd (v2df, v2df)
v2df __builtin_ia32_minsd (v2df, v2df)
v2df __builtin_ia32_maxsd (v2df, v2df)
v2df __builtin_ia32_andpd (v2df, v2df)
v2df __builtin_ia32_andnpd (v2df, v2df)
v2df __builtin_ia32_orpd (v2df, v2df)
v2df __builtin_ia32_xorpd (v2df, v2df)
v2df __builtin_ia32_movsd (v2df, v2df)
v2df __builtin_ia32_unpckhpd (v2df, v2df)
v2df __builtin_ia32_unpcklpd (v2df, v2df)
v16qi __builtin_ia32_paddb128 (v16qi, v16qi)
v8hi __builtin_ia32_paddw128 (v8hi, v8hi)
v4si __builtin_ia32_paddd128 (v4si, v4si)
v2di __builtin_ia32_paddq128 (v2di, v2di)
v16qi __builtin_ia32_psubb128 (v16qi, v16qi)
v8hi __builtin_ia32_psubw128 (v8hi, v8hi)
v4si __builtin_ia32_psubd128 (v4si, v4si)
v2di __builtin_ia32_psubq128 (v2di, v2di)
v8hi __builtin_ia32_pmullw128 (v8hi, v8hi)
v8hi __builtin_ia32_pmulhw128 (v8hi, v8hi)
v2di __builtin_ia32_pand128 (v2di, v2di)
v2di __builtin_ia32_pandn128 (v2di, v2di)
v2di __builtin_ia32_por128 (v2di, v2di)
v2di __builtin_ia32_pxor128 (v2di, v2di)
v16qi __builtin_ia32_pavgb128 (v16qi, v16qi)
v8hi __builtin_ia32_pavgw128 (v8hi, v8hi)
v16qi __builtin_ia32_pcmpeqb128 (v16qi, v16qi)
v8hi __builtin_ia32_pcmpeqw128 (v8hi, v8hi)
v4si __builtin_ia32_pcmpeqd128 (v4si, v4si)
v16qi __builtin_ia32_pcmpgtb128 (v16qi, v16qi)
v8hi __builtin_ia32_pcmpgtw128 (v8hi, v8hi)
v4si __builtin_ia32_pcmpgtd128 (v4si, v4si)
v16qi __builtin_ia32_pmaxub128 (v16qi, v16qi)
v8hi __builtin_ia32_pmaxsw128 (v8hi, v8hi)
v16qi __builtin_ia32_pminub128 (v16qi, v16qi)
v8hi __builtin_ia32_pminsw128 (v8hi, v8hi)
v16qi __builtin_ia32_punpckhbw128 (v16qi, v16qi)
v8hi __builtin_ia32_punpckhwd128 (v8hi, v8hi)
v4si __builtin_ia32_punpckhdq128 (v4si, v4si)
v2di __builtin_ia32_punpckhqdq128 (v2di, v2di)
v16qi __builtin_ia32_punpcklbw128 (v16qi, v16qi)
v8hi __builtin_ia32_punpcklwd128 (v8hi, v8hi)
v4si __builtin_ia32_punpckldq128 (v4si, v4si)
v2di __builtin_ia32_punpcklqdq128 (v2di, v2di)
v16qi __builtin_ia32_packsswb128 (v8hi, v8hi)
v8hi __builtin_ia32_packssdw128 (v4si, v4si)
v16qi __builtin_ia32_packuswb128 (v8hi, v8hi)
v8hi __builtin_ia32_pmulhuw128 (v8hi, v8hi)
void __builtin_ia32_maskmovdqu (v16qi, v16qi)
v2df __builtin_ia32_loadupd (double *)
void __builtin_ia32_storeupd (double *, v2df)
v2df __builtin_ia32_loadhpd (v2df, double const *)
v2df __builtin_ia32_loadlpd (v2df, double const *)
```

```
int __builtin_ia32_movmskpd (v2df)
int __builtin_ia32_pmovmskb128 (v16qi)
void __builtin_ia32_movnti (int *, int)
void __builtin_ia32_movnti64 (long long int *, long long int)
void __builtin_ia32_movntpd (double *, v2df)
void __builtin_ia32_movntdq (v2df *, v2df)
v4si __builtin_ia32_pshufd (v4si, int)
v8hi __builtin_ia32_pshuflw (v8hi, int)
v8hi __builtin_ia32_pshufhw (v8hi, int)
v2di __builtin_ia32_psadbw128 (v16qi, v16qi)
v2df __builtin_ia32_sqrtpd (v2df)
v2df __builtin_ia32_sqrtsd (v2df)
v2df __builtin_ia32_shufpd (v2df, v2df, int)
v2df __builtin_ia32_cvtdq2pd (v4si)
v4sf __builtin_ia32_cvtdq2ps (v4si)
v4si __builtin_ia32_cvtpd2dq (v2df)
v2si __builtin_ia32_cvtpd2pi (v2df)
v4sf __builtin_ia32_cvtpd2ps (v2df)
v4si __builtin_ia32_cvttpd2dq (v2df)
v2si __builtin_ia32_cvttpd2pi (v2df)
v2df __builtin_ia32_cvtpi2pd (v2si)
int __builtin_ia32_cvtsd2si (v2df)
int __builtin_ia32_cvttsd2si (v2df)
long long __builtin_ia32_cvtsd2si64 (v2df)
long long __builtin_ia32_cvttsd2si64 (v2df)
v4si __builtin_ia32_cvtps2dq (v4sf)
v2df __builtin_ia32_cvtps2pd (v4sf)
v4si __builtin_ia32_cvttps2dq (v4sf)
v2df __builtin_ia32_cvtsi2sd (v2df, int)
v2df __builtin_ia32_cvtsi642sd (v2df, long long)
v4sf __builtin_ia32_cvtsd2ss (v4sf, v2df)
v2df __builtin_ia32_cvtss2sd (v2df, v4sf)
void __builtin_ia32_clflush (const void *)
void __builtin_ia32_lfence (void)
void __builtin_ia32_mfence (void)
v16qi __builtin_ia32_loaddqu (const char *)
void __builtin_ia32_storedqu (char *, v16qi)
v1di __builtin_ia32_pmuludq (v2si, v2si)
v2di __builtin_ia32_pmuludq128 (v4si, v4si)
v8hi __builtin_ia32_psllw128 (v8hi, v8hi)
v4si __builtin_ia32_pslld128 (v4si, v4si)
v2di __builtin_ia32_psllq128 (v2di, v2di)
v8hi __builtin_ia32_psrlw128 (v8hi, v8hi)
v4si __builtin_ia32_psrld128 (v4si, v4si)
v2di __builtin_ia32_psrlq128 (v2di, v2di)
v8hi __builtin_ia32_psraw128 (v8hi, v8hi)
v4si __builtin_ia32_psrad128 (v4si, v4si)
v2di __builtin_ia32_pslldqi128 (v2di, int)
v8hi __builtin_ia32_psllwi128 (v8hi, int)
v4si __builtin_ia32_pslldi128 (v4si, int)
v2di __builtin_ia32_psllqi128 (v2di, int)
v2di __builtin_ia32_psrldqi128 (v2di, int)
v8hi __builtin_ia32_psrlwi128 (v8hi, int)
v4si __builtin_ia32_psrldi128 (v4si, int)
v2di __builtin_ia32_psrlqi128 (v2di, int)
v8hi __builtin_ia32_psrawi128 (v8hi, int)
v4si __builtin_ia32_psradi128 (v4si, int)
v4si __builtin_ia32_pmaddwd128 (v8hi, v8hi)
```

```
v2di __builtin_ia32_movq128 (v2di)
```

The following built-in functions are available when '-msse3' is used. All of them generate the machine instruction that is part of the name.

```
v2df __builtin_ia32_addsubpd (v2df, v2df)
v4sf __builtin_ia32_addsubps (v4sf, v4sf)
v2df __builtin_ia32_haddpd (v2df, v2df)
v4sf __builtin_ia32_haddps (v4sf, v4sf)
v2df __builtin_ia32_hsubpd (v2df, v2df)
v4sf __builtin_ia32_hsubps (v4sf, v4sf)
v16qi __builtin_ia32_lddqu (char const *)
void __builtin_ia32_monitor (void *, unsigned int, unsigned int)
v4sf __builtin_ia32_movshdup (v4sf)
v4sf __builtin_ia32_movsldup (v4sf)
void __builtin_ia32_mwait (unsigned int, unsigned int)
```

The following built-in functions are available when '-mssse3' is used. All of them generate the machine instruction that is part of the name.

```
v2si __builtin_ia32_phaddd (v2si, v2si)
v4hi __builtin_ia32_phaddw (v4hi, v4hi)
v4hi __builtin_ia32_phaddsw (v4hi, v4hi)
v2si __builtin_ia32_phsubd (v2si, v2si)
v4hi __builtin_ia32_phsubw (v4hi, v4hi)
v4hi __builtin_ia32_phsubsw (v4hi, v4hi)
v4hi __builtin_ia32_pmaddubsw (v8qi, v8qi)
v4hi __builtin_ia32_pmulhrsw (v4hi, v4hi)
v8qi __builtin_ia32_pshufb (v8qi, v8qi)
v8qi __builtin_ia32_psignb (v8qi, v8qi)
v2si __builtin_ia32_psignd (v2si, v2si)
v4hi __builtin_ia32_psignw (v4hi, v4hi)
v1di __builtin_ia32_palignr (v1di, v1di, int)
v8qi __builtin_ia32_pabsb (v8qi)
v2si __builtin_ia32_pabsd (v2si)
v4hi __builtin_ia32_pabsw (v4hi)
```

The following built-in functions are available when '-mssse3' is used. All of them generate the machine instruction that is part of the name.

```
v4si __builtin_ia32_phaddd128 (v4si, v4si)
v8hi __builtin_ia32_phaddw128 (v8hi, v8hi)
v8hi __builtin_ia32_phaddsw128 (v8hi, v8hi)
v4si __builtin_ia32_phsubd128 (v4si, v4si)
v8hi __builtin_ia32_phsubw128 (v8hi, v8hi)
v8hi __builtin_ia32_phsubsw128 (v8hi, v8hi)
v8hi __builtin_ia32_pmaddubsw128 (v16qi, v16qi)
v8hi __builtin_ia32_pmulhrsw128 (v8hi, v8hi)
v16qi __builtin_ia32_pshufb128 (v16qi, v16qi)
v16qi __builtin_ia32_psignb128 (v16qi, v16qi)
v4si __builtin_ia32_psignd128 (v4si, v4si)
v8hi __builtin_ia32_psignw128 (v8hi, v8hi)
v2di __builtin_ia32_palignr128 (v2di, v2di, int)
v16qi __builtin_ia32_pabsb128 (v16qi)
v4si __builtin_ia32_pabsd128 (v4si)
v8hi __builtin_ia32_pabsw128 (v8hi)
```

The following built-in functions are available when '-msse4.1' is used. All of them generate the machine instruction that is part of the name.

```
v2df __builtin_ia32_blendpd (v2df, v2df, const int)
v4sf __builtin_ia32_blendps (v4sf, v4sf, const int)
```

```
v2df __builtin_ia32_blendvpd (v2df, v2df, v2df)
v4sf __builtin_ia32_blendvps (v4sf, v4sf, v4sf)
v2df __builtin_ia32_dppd (v2df, v2df, const int)
v4sf __builtin_ia32_dpps (v4sf, v4sf, const int)
v4sf __builtin_ia32_insertps128 (v4sf, v4sf, const int)
v2di __builtin_ia32_movntdqa (v2di *);
v16qi __builtin_ia32_mpsadbw128 (v16qi, v16qi, const int)
v8hi __builtin_ia32_packusdw128 (v4si, v4si)
v16qi __builtin_ia32_pblendvb128 (v16qi, v16qi, v16qi)
v8hi __builtin_ia32_pblendw128 (v8hi, v8hi, const int)
v2di __builtin_ia32_pcmpeqq (v2di, v2di)
v8hi __builtin_ia32_phminposuw128 (v8hi)
v16qi __builtin_ia32_pmaxsb128 (v16qi, v16qi)
v4si __builtin_ia32_pmaxsd128 (v4si, v4si)
v4si __builtin_ia32_pmaxud128 (v4si, v4si)
v8hi __builtin_ia32_pmaxuw128 (v8hi, v8hi)
v16qi __builtin_ia32_pminsb128 (v16qi, v16qi)
v4si __builtin_ia32_pminsd128 (v4si, v4si)
v4si __builtin_ia32_pminud128 (v4si, v4si)
v8hi __builtin_ia32_pminuw128 (v8hi, v8hi)
v4si __builtin_ia32_pmovsxbd128 (v16qi)
v2di __builtin_ia32_pmovsxbq128 (v16qi)
v8hi __builtin_ia32_pmovsxbw128 (v16qi)
v2di __builtin_ia32_pmovsxdq128 (v4si)
v4si __builtin_ia32_pmovsxwd128 (v8hi)
v2di __builtin_ia32_pmovsxwq128 (v8hi)
v4si __builtin_ia32_pmovzxbd128 (v16qi)
v2di __builtin_ia32_pmovzxbq128 (v16qi)
v8hi __builtin_ia32_pmovzxbw128 (v16qi)
v2di __builtin_ia32_pmovzxdq128 (v4si)
v4si __builtin_ia32_pmovzxwd128 (v8hi)
v2di __builtin_ia32_pmovzxwq128 (v8hi)
v2di __builtin_ia32_pmuldq128 (v4si, v4si)
v4si __builtin_ia32_pmulld128 (v4si, v4si)
int __builtin_ia32_ptestc128 (v2di, v2di)
int __builtin_ia32_ptestnzc128 (v2di, v2di)
int __builtin_ia32_ptestz128 (v2di, v2di)
v2df __builtin_ia32_roundpd (v2df, const int)
v4sf __builtin_ia32_roundps (v4sf, const int)
v2df __builtin_ia32_roundsd (v2df, v2df, const int)
v4sf __builtin_ia32_roundss (v4sf, v4sf, const int)
```

The following built-in functions are available when '`-msse4.1`' is used.

`v4sf __builtin_ia32_vec_set_v4sf (v4sf, float, const int)`
: Generates the `insertps` machine instruction.

`int __builtin_ia32_vec_ext_v16qi (v16qi, const int)`
: Generates the `pextrb` machine instruction.

`v16qi __builtin_ia32_vec_set_v16qi (v16qi, int, const int)`
: Generates the `pinsrb` machine instruction.

`v4si __builtin_ia32_vec_set_v4si (v4si, int, const int)`
: Generates the `pinsrd` machine instruction.

`v2di __builtin_ia32_vec_set_v2di (v2di, long long, const int)`
: Generates the `pinsrq` machine instruction in 64bit mode.

The following built-in functions are changed to generate new SSE4.1 instructions when '`-msse4.1`' is used.

`float __builtin_ia32_vec_ext_v4sf (v4sf, const int)`
Generates the `extractps` machine instruction.

`int __builtin_ia32_vec_ext_v4si (v4si, const int)`
Generates the `pextrd` machine instruction.

`long long __builtin_ia32_vec_ext_v2di (v2di, const int)`
Generates the `pextrq` machine instruction in 64bit mode.

The following built-in functions are available when '`-msse4.2`' is used. All of them generate the machine instruction that is part of the name.

```
v16qi __builtin_ia32_pcmpestrm128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestri128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestria128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestric128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestrio128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestris128 (v16qi, int, v16qi, int, const int)
int __builtin_ia32_pcmpestriz128 (v16qi, int, v16qi, int, const int)
v16qi __builtin_ia32_pcmpistrm128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistri128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistria128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistric128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistrio128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistris128 (v16qi, v16qi, const int)
int __builtin_ia32_pcmpistriz128 (v16qi, v16qi, const int)
v2di __builtin_ia32_pcmpgtq (v2di, v2di)
```

The following built-in functions are available when '`-msse4.2`' is used.

`unsigned int __builtin_ia32_crc32qi (unsigned int, unsigned char)`
Generates the `crc32b` machine instruction.

`unsigned int __builtin_ia32_crc32hi (unsigned int, unsigned short)`
Generates the `crc32w` machine instruction.

`unsigned int __builtin_ia32_crc32si (unsigned int, unsigned int)`
Generates the `crc32l` machine instruction.

`unsigned long long __builtin_ia32_crc32di (unsigned long long, unsigned long long)`
Generates the `crc32q` machine instruction.

The following built-in functions are changed to generate new SSE4.2 instructions when '`-msse4.2`' is used.

`int __builtin_popcount (unsigned int)`
Generates the `popcntl` machine instruction.

`int __builtin_popcountl (unsigned long)`
Generates the `popcntl` or `popcntq` machine instruction, depending on the size of `unsigned long`.

`int __builtin_popcountll (unsigned long long)`
Generates the `popcntq` machine instruction.

The following built-in functions are available when '-mavx' is used. All of them generate the machine instruction that is part of the name.

```
v4df __builtin_ia32_addpd256 (v4df,v4df)
v8sf __builtin_ia32_addps256 (v8sf,v8sf)
v4df __builtin_ia32_addsubpd256 (v4df,v4df)
v8sf __builtin_ia32_addsubps256 (v8sf,v8sf)
v4df __builtin_ia32_andnpd256 (v4df,v4df)
v8sf __builtin_ia32_andnps256 (v8sf,v8sf)
v4df __builtin_ia32_andpd256 (v4df,v4df)
v8sf __builtin_ia32_andps256 (v8sf,v8sf)
v4df __builtin_ia32_blendpd256 (v4df,v4df,int)
v8sf __builtin_ia32_blendps256 (v8sf,v8sf,int)
v4df __builtin_ia32_blendvpd256 (v4df,v4df,v4df)
v8sf __builtin_ia32_blendvps256 (v8sf,v8sf,v8sf)
v2df __builtin_ia32_cmppd (v2df,v2df,int)
v4df __builtin_ia32_cmppd256 (v4df,v4df,int)
v4sf __builtin_ia32_cmpps (v4sf,v4sf,int)
v8sf __builtin_ia32_cmpps256 (v8sf,v8sf,int)
v2df __builtin_ia32_cmpsd (v2df,v2df,int)
v4sf __builtin_ia32_cmpss (v4sf,v4sf,int)
v4df __builtin_ia32_cvtdq2pd256 (v4si)
v8sf __builtin_ia32_cvtdq2ps256 (v8si)
v4si __builtin_ia32_cvtpd2dq256 (v4df)
v4sf __builtin_ia32_cvtpd2ps256 (v4df)
v8si __builtin_ia32_cvtps2dq256 (v8sf)
v4df __builtin_ia32_cvtps2pd256 (v4sf)
v4si __builtin_ia32_cvttpd2dq256 (v4df)
v8si __builtin_ia32_cvttps2dq256 (v8sf)
v4df __builtin_ia32_divpd256 (v4df,v4df)
v8sf __builtin_ia32_divps256 (v8sf,v8sf)
v8sf __builtin_ia32_dpps256 (v8sf,v8sf,int)
v4df __builtin_ia32_haddpd256 (v4df,v4df)
v8sf __builtin_ia32_haddps256 (v8sf,v8sf)
v4df __builtin_ia32_hsubpd256 (v4df,v4df)
v8sf __builtin_ia32_hsubps256 (v8sf,v8sf)
v32qi __builtin_ia32_lddqu256 (pcchar)
v32qi __builtin_ia32_loaddqu256 (pcchar)
v4df __builtin_ia32_loadupd256 (pcdouble)
v8sf __builtin_ia32_loadups256 (pcfloat)
v2df __builtin_ia32_maskloadpd (pcv2df,v2df)
v4df __builtin_ia32_maskloadpd256 (pcv4df,v4df)
v4sf __builtin_ia32_maskloadps (pcv4sf,v4sf)
v8sf __builtin_ia32_maskloadps256 (pcv8sf,v8sf)
void __builtin_ia32_maskstorepd (pv2df,v2df,v2df)
void __builtin_ia32_maskstorepd256 (pv4df,v4df,v4df)
void __builtin_ia32_maskstoreps (pv4sf,v4sf,v4sf)
void __builtin_ia32_maskstoreps256 (pv8sf,v8sf,v8sf)
v4df __builtin_ia32_maxpd256 (v4df,v4df)
v8sf __builtin_ia32_maxps256 (v8sf,v8sf)
v4df __builtin_ia32_minpd256 (v4df,v4df)
v8sf __builtin_ia32_minps256 (v8sf,v8sf)
v4df __builtin_ia32_movddup256 (v4df)
int __builtin_ia32_movmskpd256 (v4df)
int __builtin_ia32_movmskps256 (v8sf)
v8sf __builtin_ia32_movshdup256 (v8sf)
v8sf __builtin_ia32_movsldup256 (v8sf)
v4df __builtin_ia32_mulpd256 (v4df,v4df)
v8sf __builtin_ia32_mulps256 (v8sf,v8sf)
```

```
v4df __builtin_ia32_orpd256 (v4df,v4df)
v8sf __builtin_ia32_orps256 (v8sf,v8sf)
v2df __builtin_ia32_pd_pd256 (v4df)
v4df __builtin_ia32_pd256_pd (v2df)
v4sf __builtin_ia32_ps_ps256 (v8sf)
v8sf __builtin_ia32_ps256_ps (v4sf)
int __builtin_ia32_ptestc256 (v4di,v4di,ptest)
int __builtin_ia32_ptestnzc256 (v4di,v4di,ptest)
int __builtin_ia32_ptestz256 (v4di,v4di,ptest)
v8sf __builtin_ia32_rcpps256 (v8sf)
v4df __builtin_ia32_roundpd256 (v4df,int)
v8sf __builtin_ia32_roundps256 (v8sf,int)
v8sf __builtin_ia32_rsqrtps_nr256 (v8sf)
v8sf __builtin_ia32_rsqrtps256 (v8sf)
v4df __builtin_ia32_shufpd256 (v4df,v4df,int)
v8sf __builtin_ia32_shufps256 (v8sf,v8sf,int)
v4si __builtin_ia32_si_si256 (v8si)
v8si __builtin_ia32_si256_si (v4si)
v4df __builtin_ia32_sqrtpd256 (v4df)
v8sf __builtin_ia32_sqrtps_nr256 (v8sf)
v8sf __builtin_ia32_sqrtps256 (v8sf)
void __builtin_ia32_storedqu256 (pchar,v32qi)
void __builtin_ia32_storeupd256 (pdouble,v4df)
void __builtin_ia32_storeups256 (pfloat,v8sf)
v4df __builtin_ia32_subpd256 (v4df,v4df)
v8sf __builtin_ia32_subps256 (v8sf,v8sf)
v4df __builtin_ia32_unpckhpd256 (v4df,v4df)
v8sf __builtin_ia32_unpckhps256 (v8sf,v8sf)
v4df __builtin_ia32_unpcklpd256 (v4df,v4df)
v8sf __builtin_ia32_unpcklps256 (v8sf,v8sf)
v4df __builtin_ia32_vbroadcastf128_pd256 (pcv2df)
v8sf __builtin_ia32_vbroadcastf128_ps256 (pcv4sf)
v4df __builtin_ia32_vbroadcastsd256 (pcdouble)
v4sf __builtin_ia32_vbroadcastss (pcfloat)
v8sf __builtin_ia32_vbroadcastss256 (pcfloat)
v2df __builtin_ia32_vextractf128_pd256 (v4df,int)
v4sf __builtin_ia32_vextractf128_ps256 (v8sf,int)
v4si __builtin_ia32_vextractf128_si256 (v8si,int)
v4df __builtin_ia32_vinsertf128_pd256 (v4df,v2df,int)
v8sf __builtin_ia32_vinsertf128_ps256 (v8sf,v4sf,int)
v8si __builtin_ia32_vinsertf128_si256 (v8si,v4si,int)
v4df __builtin_ia32_vperm2f128_pd256 (v4df,v4df,int)
v8sf __builtin_ia32_vperm2f128_ps256 (v8sf,v8sf,int)
v8si __builtin_ia32_vperm2f128_si256 (v8si,v8si,int)
v2df __builtin_ia32_vpermil2pd (v2df,v2df,v2di,int)
v4df __builtin_ia32_vpermil2pd256 (v4df,v4df,v4di,int)
v4sf __builtin_ia32_vpermil2ps (v4sf,v4sf,v4si,int)
v8sf __builtin_ia32_vpermil2ps256 (v8sf,v8sf,v8si,int)
v2df __builtin_ia32_vpermilpd (v2df,int)
v4df __builtin_ia32_vpermilpd256 (v4df,int)
v4sf __builtin_ia32_vpermilps (v4sf,int)
v8sf __builtin_ia32_vpermilps256 (v8sf,int)
v2df __builtin_ia32_vpermilvarpd (v2df,v2di)
v4df __builtin_ia32_vpermilvarpd256 (v4df,v4di)
v4sf __builtin_ia32_vpermilvarps (v4sf,v4si)
v8sf __builtin_ia32_vpermilvarps256 (v8sf,v8si)
int __builtin_ia32_vtestcpd (v2df,v2df,ptest)
int __builtin_ia32_vtestcpd256 (v4df,v4df,ptest)
```

```
int __builtin_ia32_vtestcps (v4sf,v4sf,ptest)
int __builtin_ia32_vtestcps256 (v8sf,v8sf,ptest)
int __builtin_ia32_vtestnzcpd (v2df,v2df,ptest)
int __builtin_ia32_vtestnzcpd256 (v4df,v4df,ptest)
int __builtin_ia32_vtestnzcps (v4sf,v4sf,ptest)
int __builtin_ia32_vtestnzcps256 (v8sf,v8sf,ptest)
int __builtin_ia32_vtestzpd (v2df,v2df,ptest)
int __builtin_ia32_vtestzpd256 (v4df,v4df,ptest)
int __builtin_ia32_vtestzps (v4sf,v4sf,ptest)
int __builtin_ia32_vtestzps256 (v8sf,v8sf,ptest)
void __builtin_ia32_vzeroall (void)
void __builtin_ia32_vzeroupper (void)
v4df __builtin_ia32_xorpd256 (v4df,v4df)
v8sf __builtin_ia32_xorps256 (v8sf,v8sf)
```

The following built-in functions are available when '`-mavx2`' is used. All of them generate the machine instruction that is part of the name.

```
v32qi __builtin_ia32_mpsadbw256 (v32qi,v32qi,int)
v32qi __builtin_ia32_pabsb256 (v32qi)
v16hi __builtin_ia32_pabsw256 (v16hi)
v8si __builtin_ia32_pabsd256 (v8si)
v16hi __builtin_ia32_packssdw256 (v8si,v8si)
v32qi __builtin_ia32_packsswb256 (v16hi,v16hi)
v16hi __builtin_ia32_packusdw256 (v8si,v8si)
v32qi __builtin_ia32_packuswb256 (v16hi,v16hi)
v32qi __builtin_ia32_paddb256 (v32qi,v32qi)
v16hi __builtin_ia32_paddw256 (v16hi,v16hi)
v8si __builtin_ia32_paddd256 (v8si,v8si)
v4di __builtin_ia32_paddq256 (v4di,v4di)
v32qi __builtin_ia32_paddsb256 (v32qi,v32qi)
v16hi __builtin_ia32_paddsw256 (v16hi,v16hi)
v32qi __builtin_ia32_paddusb256 (v32qi,v32qi)
v16hi __builtin_ia32_paddusw256 (v16hi,v16hi)
v4di __builtin_ia32_palignr256 (v4di,v4di,int)
v4di __builtin_ia32_andsi256 (v4di,v4di)
v4di __builtin_ia32_andnotsi256 (v4di,v4di)
v32qi __builtin_ia32_pavgb256 (v32qi,v32qi)
v16hi __builtin_ia32_pavgw256 (v16hi,v16hi)
v32qi __builtin_ia32_pblendvb256 (v32qi,v32qi,v32qi)
v16hi __builtin_ia32_pblendw256 (v16hi,v16hi,int)
v32qi __builtin_ia32_pcmpeqb256 (v32qi,v32qi)
v16hi __builtin_ia32_pcmpeqw256 (v16hi,v16hi)
v8si __builtin_ia32_pcmpeqd256 (c8si,v8si)
v4di __builtin_ia32_pcmpeqq256 (v4di,v4di)
v32qi __builtin_ia32_pcmpgtb256 (v32qi,v32qi)
v16hi __builtin_ia32_pcmpgtw256 (16hi,v16hi)
v8si __builtin_ia32_pcmpgtd256 (v8si,v8si)
v4di __builtin_ia32_pcmpgtq256 (v4di,v4di)
v16hi __builtin_ia32_phaddw256 (v16hi,v16hi)
v8si __builtin_ia32_phaddd256 (v8si,v8si)
v16hi __builtin_ia32_phaddsw256 (v16hi,v16hi)
v16hi __builtin_ia32_phsubw256 (v16hi,v16hi)
v8si __builtin_ia32_phsubd256 (v8si,v8si)
v16hi __builtin_ia32_phsubsw256 (v16hi,v16hi)
v32qi __builtin_ia32_pmaddubsw256 (v32qi,v32qi)
v16hi __builtin_ia32_pmaddwd256 (v16hi,v16hi)
v32qi __builtin_ia32_pmaxsb256 (v32qi,v32qi)
v16hi __builtin_ia32_pmaxsw256 (v16hi,v16hi)
```

```
v8si __builtin_ia32_pmaxsd256 (v8si,v8si)
v32qi __builtin_ia32_pmaxub256 (v32qi,v32qi)
v16hi __builtin_ia32_pmaxuw256 (v16hi,v16hi)
v8si __builtin_ia32_pmaxud256 (v8si,v8si)
v32qi __builtin_ia32_pminsb256 (v32qi,v32qi)
v16hi __builtin_ia32_pminsw256 (v16hi,v16hi)
v8si __builtin_ia32_pminsd256 (v8si,v8si)
v32qi __builtin_ia32_pminub256 (v32qi,v32qi)
v16hi __builtin_ia32_pminuw256 (v16hi,v16hi)
v8si __builtin_ia32_pminud256 (v8si,v8si)
int __builtin_ia32_pmovmskb256 (v32qi)
v16hi __builtin_ia32_pmovsxbw256 (v16qi)
v8si __builtin_ia32_pmovsxbd256 (v16qi)
v4di __builtin_ia32_pmovsxbq256 (v16qi)
v8si __builtin_ia32_pmovsxwd256 (v8hi)
v4di __builtin_ia32_pmovsxwq256 (v8hi)
v4di __builtin_ia32_pmovsxdq256 (v4si)
v16hi __builtin_ia32_pmovzxbw256 (v16qi)
v8si __builtin_ia32_pmovzxbd256 (v16qi)
v4di __builtin_ia32_pmovzxbq256 (v16qi)
v8si __builtin_ia32_pmovzxwd256 (v8hi)
v4di __builtin_ia32_pmovzxwq256 (v8hi)
v4di __builtin_ia32_pmovzxdq256 (v4si)
v4di __builtin_ia32_pmuldq256 (v8si,v8si)
v16hi __builtin_ia32_pmulhrsw256 (v16hi, v16hi)
v16hi __builtin_ia32_pmulhuw256 (v16hi,v16hi)
v16hi __builtin_ia32_pmulhw256 (v16hi,v16hi)
v16hi __builtin_ia32_pmullw256 (v16hi,v16hi)
v8si __builtin_ia32_pmulld256 (v8si,v8si)
v4di __builtin_ia32_pmuludq256 (v8si,v8si)
v4di __builtin_ia32_por256 (v4di,v4di)
v16hi __builtin_ia32_psadbw256 (v32qi,v32qi)
v32qi __builtin_ia32_pshufb256 (v32qi,v32qi)
v8si __builtin_ia32_pshufd256 (v8si,int)
v16hi __builtin_ia32_pshufhw256 (v16hi,int)
v16hi __builtin_ia32_pshuflw256 (v16hi,int)
v32qi __builtin_ia32_psignb256 (v32qi,v32qi)
v16hi __builtin_ia32_psignw256 (v16hi,v16hi)
v8si __builtin_ia32_psignd256 (v8si,v8si)
v4di __builtin_ia32_pslldqi256 (v4di,int)
v16hi __builtin_ia32_psllwi256 (16hi,int)
v16hi __builtin_ia32_psllw256(v16hi,v8hi)
v8si __builtin_ia32_pslldi256 (v8si,int)
v8si __builtin_ia32_pslld256(v8si,v4si)
v4di __builtin_ia32_psllqi256 (v4di,int)
v4di __builtin_ia32_psllq256(v4di,v2di)
v16hi __builtin_ia32_psrawi256 (v16hi,int)
v16hi __builtin_ia32_psraw256 (v16hi,v8hi)
v8si __builtin_ia32_psradi256 (v8si,int)
v8si __builtin_ia32_psrad256 (v8si,v4si)
v4di __builtin_ia32_psrldqi256 (v4di, int)
v16hi __builtin_ia32_psrlwi256 (v16hi,int)
v16hi __builtin_ia32_psrlw256 (v16hi,v8hi)
v8si __builtin_ia32_psrldi256 (v8si,int)
v8si __builtin_ia32_psrld256 (v8si,v4si)
v4di __builtin_ia32_psrlqi256 (v4di,int)
v4di __builtin_ia32_psrlq256(v4di,v2di)
v32qi __builtin_ia32_psubb256 (v32qi,v32qi)
```

```
v32hi __builtin_ia32_psubw256 (v16hi,v16hi)
v8si __builtin_ia32_psubd256 (v8si,v8si)
v4di __builtin_ia32_psubq256 (v4di,v4di)
v32qi __builtin_ia32_psubsb256 (v32qi,v32qi)
v16hi __builtin_ia32_psubsw256 (v16hi,v16hi)
v32qi __builtin_ia32_psubusb256 (v32qi,v32qi)
v16hi __builtin_ia32_psubusw256 (v16hi,v16hi)
v32qi __builtin_ia32_punpckhbw256 (v32qi,v32qi)
v16hi __builtin_ia32_punpckhwd256 (v16hi,v16hi)
v8si __builtin_ia32_punpckhdq256 (v8si,v8si)
v4di __builtin_ia32_punpckhqdq256 (v4di,v4di)
v32qi __builtin_ia32_punpcklbw256 (v32qi,v32qi)
v16hi __builtin_ia32_punpcklwd256 (v16hi,v16hi)
v8si __builtin_ia32_punpckldq256 (v8si,v8si)
v4di __builtin_ia32_punpcklqdq256 (v4di,v4di)
v4di __builtin_ia32_pxor256 (v4di,v4di)
v4di __builtin_ia32_movntdqa256 (pv4di)
v4sf __builtin_ia32_vbroadcastss_ps (v4sf)
v8sf __builtin_ia32_vbroadcastss_ps256 (v4sf)
v4df __builtin_ia32_vbroadcastsd_pd256 (v2df)
v4di __builtin_ia32_vbroadcastsi256 (v2di)
v4si __builtin_ia32_pblendd128 (v4si,v4si)
v8si __builtin_ia32_pblendd256 (v8si,v8si)
v32qi __builtin_ia32_pbroadcastb256 (v16qi)
v16hi __builtin_ia32_pbroadcastw256 (v8hi)
v8si __builtin_ia32_pbroadcastd256 (v4si)
v4di __builtin_ia32_pbroadcastq256 (v2di)
v16qi __builtin_ia32_pbroadcastb128 (v16qi)
v8hi __builtin_ia32_pbroadcastw128 (v8hi)
v4si __builtin_ia32_pbroadcastd128 (v4si)
v2di __builtin_ia32_pbroadcastq128 (v2di)
v8si __builtin_ia32_permvarsi256 (v8si,v8si)
v4df __builtin_ia32_permdf256 (v4df,int)
v8sf __builtin_ia32_permvarsf256 (v8sf,v8sf)
v4di __builtin_ia32_permdi256 (v4di,int)
v4di __builtin_ia32_permti256 (v4di,v4di,int)
v4di __builtin_ia32_extract128i256 (v4di,int)
v4di __builtin_ia32_insert128i256 (v4di,v2di,int)
v8si __builtin_ia32_maskloadd256 (pcv8si,v8si)
v4di __builtin_ia32_maskloadq256 (pcv4di,v4di)
v4si __builtin_ia32_maskloadd (pcv4si,v4si)
v2di __builtin_ia32_maskloadq (pcv2di,v2di)
void __builtin_ia32_maskstored256 (pv8si,v8si,v8si)
void __builtin_ia32_maskstoreq256 (pv4di,v4di,v4di)
void __builtin_ia32_maskstored (pv4si,v4si,v4si)
void __builtin_ia32_maskstoreq (pv2di,v2di,v2di)
v8si __builtin_ia32_psllv8si (v8si,v8si)
v4si __builtin_ia32_psllv4si (v4si,v4si)
v4di __builtin_ia32_psllv4di (v4di,v4di)
v2di __builtin_ia32_psllv2di (v2di,v2di)
v8si __builtin_ia32_psrav8si (v8si,v8si)
v4si __builtin_ia32_psrav4si (v4si,v4si)
v8si __builtin_ia32_psrlv8si (v8si,v8si)
v4si __builtin_ia32_psrlv4si (v4si,v4si)
v4di __builtin_ia32_psrlv4di (v4di,v4di)
v2di __builtin_ia32_psrlv2di (v2di,v2di)
v2df __builtin_ia32_gathersiv2df (v2df, pcdouble,v4si,v2df,int)
v4df __builtin_ia32_gathersiv4df (v4df, pcdouble,v4si,v4df,int)
```

```
v2df __builtin_ia32_gatherdiv2df (v2df, pcdouble,v2di,v2df,int)
v4df __builtin_ia32_gatherdiv4df (v4df, pcdouble,v4di,v4df,int)
v4sf __builtin_ia32_gathersiv4sf (v4sf, pcfloat,v4si,v4sf,int)
v8sf __builtin_ia32_gathersiv8sf (v8sf, pcfloat,v8si,v8sf,int)
v4sf __builtin_ia32_gatherdiv4sf (v4sf, pcfloat,v2di,v4sf,int)
v4sf __builtin_ia32_gatherdiv4sf256 (v4sf, pcfloat,v4di,v4sf,int)
v2di __builtin_ia32_gathersiv2di (v2di, pcint64,v4si,v2di,int)
v4di __builtin_ia32_gathersiv4di (v4di, pcint64,v4si,v4di,int)
v2di __builtin_ia32_gatherdiv2di (v2di, pcint64,v2di,v2di,int)
v4di __builtin_ia32_gatherdiv4di (v4di, pcint64,v4di,v4di,int)
v4si __builtin_ia32_gathersiv4si (v4si, pcint,v4si,v4si,int)
v8si __builtin_ia32_gathersiv8si (v8si, pcint,v8si,v8si,int)
v4si __builtin_ia32_gatherdiv4si (v4si, pcint,v2di,v4si,int)
v4si __builtin_ia32_gatherdiv4si256 (v4si, pcint,v4di,v4si,int)
```

The following built-in functions are available when '`-maes`' is used. All of them generate the machine instruction that is part of the name.

```
v2di __builtin_ia32_aesenc128 (v2di, v2di)
v2di __builtin_ia32_aesenclast128 (v2di, v2di)
v2di __builtin_ia32_aesdec128 (v2di, v2di)
v2di __builtin_ia32_aesdeclast128 (v2di, v2di)
v2di __builtin_ia32_aeskeygenassist128 (v2di, const int)
v2di __builtin_ia32_aesimc128 (v2di)
```

The following built-in function is available when '`-mpclmul`' is used.

`v2di __builtin_ia32_pclmulqdq128 (v2di, v2di, const int)`
: Generates the `pclmulqdq` machine instruction.

The following built-in function is available when '`-mfsgsbase`' is used. All of them generate the machine instruction that is part of the name.

```
unsigned int __builtin_ia32_rdfsbase32 (void)
unsigned long long __builtin_ia32_rdfsbase64 (void)
unsigned int __builtin_ia32_rdgsbase32 (void)
unsigned long long __builtin_ia32_rdgsbase64 (void)
void _writefsbase_u32 (unsigned int)
void _writefsbase_u64 (unsigned long long)
void _writegsbase_u32 (unsigned int)
void _writegsbase_u64 (unsigned long long)
```

The following built-in function is available when '`-mrdrnd`' is used. All of them generate the machine instruction that is part of the name.

```
unsigned int __builtin_ia32_rdrand16_step (unsigned short *)
unsigned int __builtin_ia32_rdrand32_step (unsigned int *)
unsigned int __builtin_ia32_rdrand64_step (unsigned long long *)
```

The following built-in function is available when '`-mptwrite`' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_ptwrite32 (unsigned)
void __builtin_ia32_ptwrite64 (unsigned long long)
```

The following built-in functions are available when '`-msse4a`' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_movntsd (double *, v2df)
void __builtin_ia32_movntss (float *, v4sf)
v2di __builtin_ia32_extrq  (v2di, v16qi)
v2di __builtin_ia32_extrqi (v2di, const unsigned int, const unsigned int)
v2di __builtin_ia32_insertq (v2di, v2di)
```

```
v2di __builtin_ia32_insertqi (v2di, v2di, const unsigned int, const unsigned int)
```

The following built-in functions are available when '-mxop' is used.

```
v2df __builtin_ia32_vfrczpd (v2df)
v4sf __builtin_ia32_vfrczps (v4sf)
v2df __builtin_ia32_vfrczsd (v2df)
v4sf __builtin_ia32_vfrczss (v4sf)
v4df __builtin_ia32_vfrczpd256 (v4df)
v8sf __builtin_ia32_vfrczps256 (v8sf)
v2di __builtin_ia32_vpcmov (v2di, v2di, v2di)
v2di __builtin_ia32_vpcmov_v2di (v2di, v2di, v2di)
v4si __builtin_ia32_vpcmov_v4si (v4si, v4si, v4si)
v8hi __builtin_ia32_vpcmov_v8hi (v8hi, v8hi, v8hi)
v16qi __builtin_ia32_vpcmov_v16qi (v16qi, v16qi, v16qi)
v2df __builtin_ia32_vpcmov_v2df (v2df, v2df, v2df)
v4sf __builtin_ia32_vpcmov_v4sf (v4sf, v4sf, v4sf)
v4di __builtin_ia32_vpcmov_v4di256 (v4di, v4di, v4di)
v8si __builtin_ia32_vpcmov_v8si256 (v8si, v8si, v8si)
v16hi __builtin_ia32_vpcmov_v16hi256 (v16hi, v16hi, v16hi)
v32qi __builtin_ia32_vpcmov_v32qi256 (v32qi, v32qi, v32qi)
v4df __builtin_ia32_vpcmov_v4df256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vpcmov_v8sf256 (v8sf, v8sf, v8sf)
v16qi __builtin_ia32_vpcomeqb (v16qi, v16qi)
v8hi __builtin_ia32_vpcomeqw (v8hi, v8hi)
v4si __builtin_ia32_vpcomeqd (v4si, v4si)
v2di __builtin_ia32_vpcomeqq (v2di, v2di)
v16qi __builtin_ia32_vpcomequb (v16qi, v16qi)
v4si __builtin_ia32_vpcomequd (v4si, v4si)
v2di __builtin_ia32_vpcomequq (v2di, v2di)
v8hi __builtin_ia32_vpcomequw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomeqw (v8hi, v8hi)
v16qi __builtin_ia32_vpcomfalseb (v16qi, v16qi)
v4si __builtin_ia32_vpcomfalsed (v4si, v4si)
v2di __builtin_ia32_vpcomfalseq (v2di, v2di)
v16qi __builtin_ia32_vpcomfalseub (v16qi, v16qi)
v4si __builtin_ia32_vpcomfalseud (v4si, v4si)
v2di __builtin_ia32_vpcomfalseuq (v2di, v2di)
v8hi __builtin_ia32_vpcomfalseuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomfalsew (v8hi, v8hi)
v16qi __builtin_ia32_vpcomgeb (v16qi, v16qi)
v4si __builtin_ia32_vpcomged (v4si, v4si)
v2di __builtin_ia32_vpcomgeq (v2di, v2di)
v16qi __builtin_ia32_vpcomgeub (v16qi, v16qi)
v4si __builtin_ia32_vpcomgeud (v4si, v4si)
v2di __builtin_ia32_vpcomgeuq (v2di, v2di)
v8hi __builtin_ia32_vpcomgeuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomgew (v8hi, v8hi)
v16qi __builtin_ia32_vpcomgtb (v16qi, v16qi)
v4si __builtin_ia32_vpcomgtd (v4si, v4si)
v2di __builtin_ia32_vpcomgtq (v2di, v2di)
v16qi __builtin_ia32_vpcomgtub (v16qi, v16qi)
v4si __builtin_ia32_vpcomgtud (v4si, v4si)
v2di __builtin_ia32_vpcomgtuq (v2di, v2di)
v8hi __builtin_ia32_vpcomgtuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomgtw (v8hi, v8hi)
v16qi __builtin_ia32_vpcomleb (v16qi, v16qi)
v4si __builtin_ia32_vpcomled (v4si, v4si)
v2di __builtin_ia32_vpcomleq (v2di, v2di)
v16qi __builtin_ia32_vpcomleub (v16qi, v16qi)
```

```
v4si __builtin_ia32_vpcomleud (v4si, v4si)
v2di __builtin_ia32_vpcomleuq (v2di, v2di)
v8hi __builtin_ia32_vpcomleuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomlew (v8hi, v8hi)
v16qi __builtin_ia32_vpcomltb (v16qi, v16qi)
v4si __builtin_ia32_vpcomltd (v4si, v4si)
v2di __builtin_ia32_vpcomltq (v2di, v2di)
v16qi __builtin_ia32_vpcomltub (v16qi, v16qi)
v4si __builtin_ia32_vpcomltud (v4si, v4si)
v2di __builtin_ia32_vpcomltuq (v2di, v2di)
v8hi __builtin_ia32_vpcomltuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomltw (v8hi, v8hi)
v16qi __builtin_ia32_vpcomneb (v16qi, v16qi)
v4si __builtin_ia32_vpcomned (v4si, v4si)
v2di __builtin_ia32_vpcomneq (v2di, v2di)
v16qi __builtin_ia32_vpcomneub (v16qi, v16qi)
v4si __builtin_ia32_vpcomneud (v4si, v4si)
v2di __builtin_ia32_vpcomneuq (v2di, v2di)
v8hi __builtin_ia32_vpcomneuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomnew (v8hi, v8hi)
v16qi __builtin_ia32_vpcomtrueb (v16qi, v16qi)
v4si __builtin_ia32_vpcomtrued (v4si, v4si)
v2di __builtin_ia32_vpcomtrueq (v2di, v2di)
v16qi __builtin_ia32_vpcomtrueub (v16qi, v16qi)
v4si __builtin_ia32_vpcomtrueud (v4si, v4si)
v2di __builtin_ia32_vpcomtrueuq (v2di, v2di)
v8hi __builtin_ia32_vpcomtrueuw (v8hi, v8hi)
v8hi __builtin_ia32_vpcomtruew (v8hi, v8hi)
v4si __builtin_ia32_vphaddbd (v16qi)
v2di __builtin_ia32_vphaddbq (v16qi)
v8hi __builtin_ia32_vphaddbw (v16qi)
v2di __builtin_ia32_vphadddq (v4si)
v4si __builtin_ia32_vphaddubd (v16qi)
v2di __builtin_ia32_vphaddubq (v16qi)
v8hi __builtin_ia32_vphaddubw (v16qi)
v2di __builtin_ia32_vphaddudq (v4si)
v4si __builtin_ia32_vphadduwd (v8hi)
v2di __builtin_ia32_vphadduwq (v8hi)
v4si __builtin_ia32_vphaddwd (v8hi)
v2di __builtin_ia32_vphaddwq (v8hi)
v8hi __builtin_ia32_vphsubbw (v16qi)
v2di __builtin_ia32_vphsubdq (v4si)
v4si __builtin_ia32_vphsubwd (v8hi)
v4si __builtin_ia32_vpmacsdd (v4si, v4si, v4si)
v2di __builtin_ia32_vpmacsdqh (v4si, v4si, v2di)
v2di __builtin_ia32_vpmacsdql (v4si, v4si, v2di)
v4si __builtin_ia32_vpmacssdd (v4si, v4si, v4si)
v2di __builtin_ia32_vpmacssdqh (v4si, v4si, v2di)
v2di __builtin_ia32_vpmacssdql (v4si, v4si, v2di)
v4si __builtin_ia32_vpmacsswd (v8hi, v8hi, v4si)
v8hi __builtin_ia32_vpmacssww (v8hi, v8hi, v8hi)
v4si __builtin_ia32_vpmacswd (v8hi, v8hi, v4si)
v8hi __builtin_ia32_vpmacsww (v8hi, v8hi, v8hi)
v4si __builtin_ia32_vpmadcsswd (v8hi, v8hi, v4si)
v4si __builtin_ia32_vpmadcswd (v8hi, v8hi, v4si)
v16qi __builtin_ia32_vpperm (v16qi, v16qi, v16qi)
v16qi __builtin_ia32_vprotb (v16qi, v16qi)
v4si __builtin_ia32_vprotd (v4si, v4si)
```

```
v2di __builtin_ia32_vprotq (v2di, v2di)
v8hi __builtin_ia32_vprotw (v8hi, v8hi)
v16qi __builtin_ia32_vpshab (v16qi, v16qi)
v4si __builtin_ia32_vpshad (v4si, v4si)
v2di __builtin_ia32_vpshaq (v2di, v2di)
v8hi __builtin_ia32_vpshaw (v8hi, v8hi)
v16qi __builtin_ia32_vpshlb (v16qi, v16qi)
v4si __builtin_ia32_vpshld (v4si, v4si)
v2di __builtin_ia32_vpshlq (v2di, v2di)
v8hi __builtin_ia32_vpshlw (v8hi, v8hi)
```

The following built-in functions are available when '`-mfma4`' is used. All of them generate the machine instruction that is part of the name.

```
v2df __builtin_ia32_vfmaddpd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmaddps (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfmaddsd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmaddss (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfmsubpd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmsubps (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfmsubsd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmsubss (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfnmaddpd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfnmaddps (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfnmaddsd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfnmaddss (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfnmsubpd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfnmsubps (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfnmsubsd (v2df, v2df, v2df)
v4sf __builtin_ia32_vfnmsubss (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfmaddsubpd  (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmaddsubps  (v4sf, v4sf, v4sf)
v2df __builtin_ia32_vfmsubaddpd  (v2df, v2df, v2df)
v4sf __builtin_ia32_vfmsubaddps  (v4sf, v4sf, v4sf)
v4df __builtin_ia32_vfmaddpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfmaddps256 (v8sf, v8sf, v8sf)
v4df __builtin_ia32_vfmsubpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfmsubps256 (v8sf, v8sf, v8sf)
v4df __builtin_ia32_vfnmaddpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfnmaddps256 (v8sf, v8sf, v8sf)
v4df __builtin_ia32_vfnmsubpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfnmsubps256 (v8sf, v8sf, v8sf)
v4df __builtin_ia32_vfmaddsubpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfmaddsubps256 (v8sf, v8sf, v8sf)
v4df __builtin_ia32_vfmsubaddpd256 (v4df, v4df, v4df)
v8sf __builtin_ia32_vfmsubaddps256 (v8sf, v8sf, v8sf)
```

The following built-in functions are available when '`-mlwp`' is used.

```
void __builtin_ia32_llwpcb16 (void *);
void __builtin_ia32_llwpcb32 (void *);
void __builtin_ia32_llwpcb64 (void *);
void * __builtin_ia32_llwpcb16 (void);
void * __builtin_ia32_llwpcb32 (void);
void * __builtin_ia32_llwpcb64 (void);
void __builtin_ia32_lwpval16 (unsigned short, unsigned int, unsigned short)
void __builtin_ia32_lwpval32 (unsigned int, unsigned int, unsigned int)
void __builtin_ia32_lwpval64 (unsigned __int64, unsigned int, unsigned int)
unsigned char __builtin_ia32_lwpins16 (unsigned short, unsigned int, unsigned short)
unsigned char __builtin_ia32_lwpins32 (unsigned int, unsigned int, unsigned int)
```

```
unsigned char __builtin_ia32_lwpins64 (unsigned __int64, unsigned int, unsigned int)
```

The following built-in functions are available when '-mbmi' is used. All of them generate the machine instruction that is part of the name.

```
unsigned int __builtin_ia32_bextr_u32(unsigned int, unsigned int);
unsigned long long __builtin_ia32_bextr_u64 (unsigned long long, unsigned long long);
```

The following built-in functions are available when '-mbmi2' is used. All of them generate the machine instruction that is part of the name.

```
unsigned int _bzhi_u32 (unsigned int, unsigned int)
unsigned int _pdep_u32 (unsigned int, unsigned int)
unsigned int _pext_u32 (unsigned int, unsigned int)
unsigned long long _bzhi_u64 (unsigned long long, unsigned long long)
unsigned long long _pdep_u64 (unsigned long long, unsigned long long)
unsigned long long _pext_u64 (unsigned long long, unsigned long long)
```

The following built-in functions are available when '-mlzcnt' is used. All of them generate the machine instruction that is part of the name.

```
unsigned short __builtin_ia32_lzcnt_u16(unsigned short);
unsigned int __builtin_ia32_lzcnt_u32(unsigned int);
unsigned long long __builtin_ia32_lzcnt_u64 (unsigned long long);
```

The following built-in functions are available when '-mfxsr' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_fxsave (void *)
void __builtin_ia32_fxrstor (void *)
void __builtin_ia32_fxsave64 (void *)
void __builtin_ia32_fxrstor64 (void *)
```

The following built-in functions are available when '-mxsave' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_xsave (void *, long long)
void __builtin_ia32_xrstor (void *, long long)
void __builtin_ia32_xsave64 (void *, long long)
void __builtin_ia32_xrstor64 (void *, long long)
```

The following built-in functions are available when '-mxsaveopt' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_xsaveopt (void *, long long)
void __builtin_ia32_xsaveopt64 (void *, long long)
```

The following built-in functions are available when '-mtbm' is used. Both of them generate the immediate form of the bextr machine instruction.

```
unsigned int __builtin_ia32_bextri_u32 (unsigned int,
                                        const unsigned int);
unsigned long long __builtin_ia32_bextri_u64 (unsigned long long,
                                              const unsigned long long);
```

The following built-in functions are available when '-m3dnow' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_femms (void)
v8qi __builtin_ia32_pavgusb (v8qi, v8qi)
v2si __builtin_ia32_pf2id (v2sf)
v2sf __builtin_ia32_pfacc (v2sf, v2sf)
v2sf __builtin_ia32_pfadd (v2sf, v2sf)
v2si __builtin_ia32_pfcmpeq (v2sf, v2sf)
v2si __builtin_ia32_pfcmpge (v2sf, v2sf)
v2si __builtin_ia32_pfcmpgt (v2sf, v2sf)
```

```
v2sf __builtin_ia32_pfmax (v2sf, v2sf)
v2sf __builtin_ia32_pfmin (v2sf, v2sf)
v2sf __builtin_ia32_pfmul (v2sf, v2sf)
v2sf __builtin_ia32_pfrcp (v2sf)
v2sf __builtin_ia32_pfrcpit1 (v2sf, v2sf)
v2sf __builtin_ia32_pfrcpit2 (v2sf, v2sf)
v2sf __builtin_ia32_pfrsqrt (v2sf)
v2sf __builtin_ia32_pfsub (v2sf, v2sf)
v2sf __builtin_ia32_pfsubr (v2sf, v2sf)
v2sf __builtin_ia32_pi2fd (v2si)
v4hi __builtin_ia32_pmulhrw (v4hi, v4hi)
```

The following built-in functions are available when '`-m3dnowa`' is used. All of them generate the machine instruction that is part of the name.

```
v2si __builtin_ia32_pf2iw (v2sf)
v2sf __builtin_ia32_pfnacc (v2sf, v2sf)
v2sf __builtin_ia32_pfpnacc (v2sf, v2sf)
v2sf __builtin_ia32_pi2fw (v2si)
v2sf __builtin_ia32_pswapdsf (v2sf)
v2si __builtin_ia32_pswapdsi (v2si)
```

The following built-in functions are available when '`-mrtm`' is used They are used for restricted transactional memory. These are the internal low level functions. Normally the functions in Section 6.60.34 [x86 transactional memory intrinsics], page 799 should be used instead.

```
int __builtin_ia32_xbegin ()
void __builtin_ia32_xend ()
void __builtin_ia32_xabort (status)
int __builtin_ia32_xtest ()
```

The following built-in functions are available when '`-mmwaitx`' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_ia32_monitorx (void *, unsigned int, unsigned int)
void __builtin_ia32_mwaitx (unsigned int, unsigned int, unsigned int)
```

The following built-in functions are available when '`-mclzero`' is used. All of them generate the machine instruction that is part of the name.

```
void __builtin_i32_clzero (void *)
```

The following built-in functions are available when '`-mpku`' is used. They generate reads and writes to PKRU.

```
void __builtin_ia32_wrpkru (unsigned int)
unsigned int __builtin_ia32_rdpkru ()
```

The following built-in functions are available when '`-mcet`' or '`-mshstk`' option is used. They support shadow stack machine instructions from Intel Control-flow Enforcement Technology (CET). Each built-in function generates the machine instruction that is part of the function's name. These are the internal low-level functions. Normally the functions in Section 6.60.35 [x86 control-flow protection intrinsics], page 800 should be used instead.

```
unsigned int __builtin_ia32_rdsspd (void)
unsigned long long __builtin_ia32_rdsspq (void)
void __builtin_ia32_incsspd (unsigned int)
void __builtin_ia32_incsspq (unsigned long long)
void __builtin_ia32_saveprevssp(void);
void __builtin_ia32_rstorssp(void *);
void __builtin_ia32_wrssd(unsigned int, void *);
void __builtin_ia32_wrssq(unsigned long long, void *);
```

```
void __builtin_ia32_wrussd(unsigned int, void *);
void __builtin_ia32_wrussq(unsigned long long, void *);
void __builtin_ia32_setssbsy(void);
void __builtin_ia32_clrssbsy(void *);
```

6.60.34 x86 Transactional Memory Intrinsics

These hardware transactional memory intrinsics for x86 allow you to use memory transactions with RTM (Restricted Transactional Memory). This support is enabled with the '`-mrtm`' option. For using HLE (Hardware Lock Elision) see Section 6.57 [x86 specific memory model extensions for transactional memory], page 638 instead.

A memory transaction commits all changes to memory in an atomic way, as visible to other threads. If the transaction fails it is rolled back and all side effects discarded.

Generally there is no guarantee that a memory transaction ever succeeds and suitable fallback code always needs to be supplied.

`unsigned _xbegin ()` [RTM Function]

Start a RTM (Restricted Transactional Memory) transaction. Returns `_XBEGIN_STARTED` when the transaction started successfully (note this is not 0, so the constant has to be explicitly tested).

If the transaction aborts, all side effects are undone and an abort code encoded as a bit mask is returned. The following macros are defined:

`_XABORT_EXPLICIT`
: Transaction was explicitly aborted with `_xabort`. The parameter passed to `_xabort` is available with `_XABORT_CODE(status)`.

`_XABORT_RETRY`
: Transaction retry is possible.

`_XABORT_CONFLICT`
: Transaction abort due to a memory conflict with another thread.

`_XABORT_CAPACITY`
: Transaction abort due to the transaction using too much memory.

`_XABORT_DEBUG`
: Transaction abort due to a debug trap.

`_XABORT_NESTED`
: Transaction abort in an inner nested transaction.

There is no guarantee any transaction ever succeeds, so there always needs to be a valid fallback path.

`void _xend ()` [RTM Function]

Commit the current transaction. When no transaction is active this faults. All memory side effects of the transaction become visible to other threads in an atomic manner.

`int _xtest ()` [RTM Function]

Return a nonzero value if a transaction is currently active, otherwise 0.

`void _xabort` (*status*) [RTM Function]

Abort the current transaction. When no transaction is active this is a no-op. The *status* is an 8-bit constant; its value is encoded in the return value from `_xbegin`.

Here is an example showing handling for `_XABORT_RETRY` and a fallback path for other failures:

```
#include <immintrin.h>

int n_tries, max_tries;
unsigned status = _XABORT_EXPLICIT;
...

for (n_tries = 0; n_tries < max_tries; n_tries++)
  {
    status = _xbegin ();
    if (status == _XBEGIN_STARTED || !(status & _XABORT_RETRY))
      break;
  }
if (status == _XBEGIN_STARTED)
  {
    ... transaction code...
    _xend ();
  }
else
  {
    ... non-transactional fallback path...
  }
```

Note that, in most cases, the transactional and non-transactional code must synchronize together to ensure consistency.

6.60.35 x86 Control-Flow Protection Intrinsics

`ret_type _get_ssp` (*void*) [CET Function]

Get the current value of shadow stack pointer if shadow stack support from Intel CET is enabled in the hardware or `0` otherwise. The `ret_type` is `unsigned long long` for 64-bit targets and `unsigned int` for 32-bit targets.

`void _inc_ssp` (*unsigned int*) [CET Function]

Increment the current shadow stack pointer by the size specified by the function argument. The argument is masked to a byte value for security reasons, so to increment by more than 255 bytes you must call the function multiple times.

The shadow stack unwind code looks like:

```
#include <immintrin.h>

/* Unwind the shadow stack for EH.  */
#define _Unwind_Frames_Extra(x)       \
  do                                  \
    {                                 \
      _Unwind_Word ssp = _get_ssp (); \
      if (ssp != 0)                   \
        {                             \
          _Unwind_Word tmp = (x);     \
          while (tmp > 255)           \
```

```
            {                          \
              _inc_ssp (tmp);           \
              tmp -= 255;               \
            }                          \
          _inc_ssp (tmp);              \
        }                              \
      }                                \
      while (0)
```

This code runs unconditionally on all 64-bit processors. For 32-bit processors the code runs on those that support multi-byte NOP instructions.

6.61 Format Checks Specific to Particular Target Machines

For some target machines, GCC supports additional options to the format attribute (see Section 6.33 [Declaring Attributes of Functions], page 477).

6.61.1 Solaris Format Checks

Solaris targets support the `cmn_err` (or `__cmn_err__`) format check. `cmn_err` accepts a subset of the standard `printf` conversions, and the two-argument `%b` conversion for displaying bit-fields. See the Solaris man page for `cmn_err` for more information.

6.61.2 Darwin Format Checks

In addition to the full set of format archetypes (attribute format style arguments such as `printf`, `scanf`, `strftime`, and `strfmon`), Darwin targets also support the `CFString` (or `__CFString__`) archetype in the `format` attribute. Declarations with this archetype are parsed for correct syntax and argument types. However, parsing of the format string itself and validating arguments against it in calls to such functions is currently not performed.

Additionally, `CFStringRefs` (defined by the `CoreFoundation` headers) may also be used as format arguments. Note that the relevant headers are only likely to be available on Darwin (OSX) installations. On such installations, the XCode and system documentation provide descriptions of `CFString`, `CFStringRefs` and associated functions.

6.62 Pragmas Accepted by GCC

GCC supports several types of pragmas, primarily in order to compile code originally written for other compilers. Note that in general we do not recommend the use of pragmas; See Section 6.33 [Function Attributes], page 477, for further explanation.

The GNU C preprocessor recognizes several pragmas in addition to the compiler pragmas documented here. Refer to the CPP manual for more information.

6.62.1 AArch64 Pragmas

The pragmas defined by the AArch64 target correspond to the AArch64 target function attributes. They can be specified as below:

```
#pragma GCC target("string")
```

where *`string`* can be any string accepted as an AArch64 target attribute. See Section 6.33.2 [AArch64 Function Attributes], page 497, for more details on the permissible values of `string`.

6.62.2 ARM Pragmas

The ARM target defines pragmas for controlling the default addition of `long_call` and `short_call` attributes to functions. See Section 6.33 [Function Attributes], page 477, for information about the effects of these attributes.

`long_calls`
: Set all subsequent functions to have the `long_call` attribute.

`no_long_calls`
: Set all subsequent functions to have the `short_call` attribute.

`long_calls_off`
: Do not affect the `long_call` or `short_call` attributes of subsequent functions.

6.62.3 M32C Pragmas

`GCC memregs` *`number`*
: Overrides the command-line option `-memregs=` for the current file. Use with care! This pragma must be before any function in the file, and mixing different memregs values in different objects may make them incompatible. This pragma is useful when a performance-critical function uses a memreg for temporary values, as it may allow you to reduce the number of memregs used.

`ADDRESS` *`name address`*
: For any declared symbols matching *name*, this does three things to that symbol: it forces the symbol to be located at the given address (a number), it forces the symbol to be volatile, and it changes the symbol's scope to be static. This pragma exists for compatibility with other compilers, but note that the common `1234H` numeric syntax is not supported (use `0x1234` instead). Example:

```
#pragma ADDRESS port3 0x103
char port3;
```

6.62.4 MeP Pragmas

`custom io_volatile (on|off)`
: Overrides the command-line option `-mio-volatile` for the current file. Note that for compatibility with future GCC releases, this option should only be used once before any `io` variables in each file.

`GCC coprocessor available` *`registers`*
: Specifies which coprocessor registers are available to the register allocator. *registers* may be a single register, register range separated by ellipses, or comma-separated list of those. Example:

```
#pragma GCC coprocessor available $c0...$c10, $c28
```

`GCC coprocessor call_saved` *`registers`*
: Specifies which coprocessor registers are to be saved and restored by any function using them. *registers* may be a single register, register range separated by ellipses, or comma-separated list of those. Example:

```
#pragma GCC coprocessor call_saved $c4...$c6, $c31
```

`GCC coprocessor subclass '(A|B|C|D)' = registers`

Creates and defines a register class. These register classes can be used by inline `asm` constructs. *registers* may be a single register, register range separated by ellipses, or comma-separated list of those. Example:

```
#pragma GCC coprocessor subclass 'B' = $c2, $c4, $c6

asm ("cpfoo %0" : "=B" (x));
```

`GCC disinterrupt name , name ...`

For the named functions, the compiler adds code to disable interrupts for the duration of those functions. If any functions so named are not encountered in the source, a warning is emitted that the pragma is not used. Examples:

```
#pragma disinterrupt foo
#pragma disinterrupt bar, grill
int foo () { ... }
```

`GCC call name , name ...`

For the named functions, the compiler always uses a register-indirect call model when calling the named functions. Examples:

```
extern int foo ();
#pragma call foo
```

6.62.5 RS/6000 and PowerPC Pragmas

The RS/6000 and PowerPC targets define one pragma for controlling whether or not the `longcall` attribute is added to function declarations by default. This pragma overrides the '`-mlongcall`' option, but not the `longcall` and `shortcall` attributes. See Section 3.18.42 [RS/6000 and PowerPC Options], page 358, for more information about when long calls are and are not necessary.

`longcall (1)`

Apply the `longcall` attribute to all subsequent function declarations.

`longcall (0)`

Do not apply the `longcall` attribute to subsequent function declarations.

6.62.6 S/390 Pragmas

The pragmas defined by the S/390 target correspond to the S/390 target function attributes and some the additional options:

'`zvector`'
'`no-zvector`'

Note that options of the pragma, unlike options of the target attribute, do change the value of preprocessor macros like `__VEC__`. They can be specified as below:

```
#pragma GCC target("string[,string]...")
#pragma GCC target("string"["string"]...)
```

6.62.7 Darwin Pragmas

The following pragmas are available for all architectures running the Darwin operating system. These are useful for compatibility with other Mac OS compilers.

`mark tokens...`
This pragma is accepted, but has no effect.

`options align=alignment`
This pragma sets the alignment of fields in structures. The values of *alignment* may be `mac68k`, to emulate m68k alignment, or `power`, to emulate PowerPC alignment. Uses of this pragma nest properly; to restore the previous setting, use `reset` for the *alignment*.

`segment tokens...`
This pragma is accepted, but has no effect.

`unused (var [, var]...)`
This pragma declares variables to be possibly unused. GCC does not produce warnings for the listed variables. The effect is similar to that of the `unused` attribute, except that this pragma may appear anywhere within the variables' scopes.

6.62.8 Solaris Pragmas

The Solaris target supports `#pragma redefine_extname` (see Section 6.62.9 [Symbol-Renaming Pragmas], page 804). It also supports additional `#pragma` directives for compatibility with the system compiler.

`align alignment (variable [, variable]...)`
Increase the minimum alignment of each *variable* to *alignment*. This is the same as GCC's `aligned` attribute see Section 6.34 [Variable Attributes], page 536). Macro expansion occurs on the arguments to this pragma when compiling C and Objective-C. It does not currently occur when compiling C++, but this is a bug which may be fixed in a future release.

`fini (function [, function]...)`
This pragma causes each listed *function* to be called after main, or during shared module unloading, by adding a call to the `.fini` section.

`init (function [, function]...)`
This pragma causes each listed *function* to be called during initialization (before `main`) or during shared module loading, by adding a call to the `.init` section.

6.62.9 Symbol-Renaming Pragmas

GCC supports a `#pragma` directive that changes the name used in assembly for a given declaration. While this pragma is supported on all platforms, it is intended primarily to provide compatibility with the Solaris system headers. This effect can also be achieved using the asm labels extension (see Section 6.47.4 [Asm Labels], page 620).

`redefine_extname oldname newname`
This pragma gives the C function *oldname* the assembly symbol *newname*. The preprocessor macro `__PRAGMA_REDEFINE_EXTNAME` is defined if this pragma is available (currently on all platforms).

This pragma and the `asm` labels extension interact in a complicated manner. Here are some corner cases you may want to be aware of:

1. This pragma silently applies only to declarations with external linkage. The `asm` label feature does not have this restriction.
2. In C++, this pragma silently applies only to declarations with "C" linkage. Again, `asm` labels do not have this restriction.
3. If either of the ways of changing the assembly name of a declaration are applied to a declaration whose assembly name has already been determined (either by a previous use of one of these features, or because the compiler needed the assembly name in order to generate code), and the new name is different, a warning issues and the name does not change.
4. The *oldname* used by `#pragma redefine_extname` is always the C-language name.

6.62.10 Structure-Layout Pragmas

For compatibility with Microsoft Windows compilers, GCC supports a set of `#pragma` directives that change the maximum alignment of members of structures (other than zero-width bit-fields), unions, and classes subsequently defined. The *n* value below always is required to be a small power of two and specifies the new alignment in bytes.

1. `#pragma pack(n)` simply sets the new alignment.
2. `#pragma pack()` sets the alignment to the one that was in effect when compilation started (see also command-line option '`-fpack-struct[=n]`' see Section 3.16 [Code Gen Options], page 218).
3. `#pragma pack(push[,n])` pushes the current alignment setting on an internal stack and then optionally sets the new alignment.
4. `#pragma pack(pop)` restores the alignment setting to the one saved at the top of the internal stack (and removes that stack entry). Note that `#pragma pack([n])` does not influence this internal stack; thus it is possible to have `#pragma pack(push)` followed by multiple `#pragma pack(n)` instances and finalized by a single `#pragma pack(pop)`.

Some targets, e.g. x86 and PowerPC, support the `#pragma ms_struct` directive which lays out structures and unions subsequently defined as the documented `__attribute__ ((ms_struct))`.

1. `#pragma ms_struct on` turns on the Microsoft layout.
2. `#pragma ms_struct off` turns off the Microsoft layout.
3. `#pragma ms_struct reset` goes back to the default layout.

Most targets also support the `#pragma scalar_storage_order` directive which lays out structures and unions subsequently defined as the documented `__attribute__ ((scalar_storage_order))`.

1. `#pragma scalar_storage_order big-endian` sets the storage order of the scalar fields to big-endian.
2. `#pragma scalar_storage_order little-endian` sets the storage order of the scalar fields to little-endian.
3. `#pragma scalar_storage_order default` goes back to the endianness that was in effect when compilation started (see also command-line option '`-fsso-struct=endianness`' see Section 3.4 [C Dialect Options], page 37).

6.62.11 Weak Pragmas

For compatibility with SVR4, GCC supports a set of `#pragma` directives for declaring symbols to be weak, and defining weak aliases.

`#pragma weak` *`symbol`*

This pragma declares *symbol* to be weak, as if the declaration had the attribute of the same name. The pragma may appear before or after the declaration of *symbol*. It is not an error for *symbol* to never be defined at all.

`#pragma weak` *`symbol1`* `=` *`symbol2`*

This pragma declares *symbol1* to be a weak alias of *symbol2*. It is an error if *symbol2* is not defined in the current translation unit.

6.62.12 Diagnostic Pragmas

GCC allows the user to selectively enable or disable certain types of diagnostics, and change the kind of the diagnostic. For example, a project's policy might require that all sources compile with '`-Werror`' but certain files might have exceptions allowing specific types of warnings. Or, a project might selectively enable diagnostics and treat them as errors depending on which preprocessor macros are defined.

`#pragma GCC diagnostic` *`kind option`*

Modifies the disposition of a diagnostic. Note that not all diagnostics are modifiable; at the moment only warnings (normally controlled by '`-W...`') can be controlled, and not all of them. Use '`-fdiagnostics-show-option`' to determine which diagnostics are controllable and which option controls them.

kind is '`error`' to treat this diagnostic as an error, '`warning`' to treat it like a warning (even if '`-Werror`' is in effect), or '`ignored`' if the diagnostic is to be ignored. *option* is a double quoted string that matches the command-line option.

```
#pragma GCC diagnostic warning "-Wformat"
#pragma GCC diagnostic error "-Wformat"
#pragma GCC diagnostic ignored "-Wformat"
```

Note that these pragmas override any command-line options. GCC keeps track of the location of each pragma, and issues diagnostics according to the state as of that point in the source file. Thus, pragmas occurring after a line do not affect diagnostics caused by that line.

`#pragma GCC diagnostic push`
`#pragma GCC diagnostic pop`

Causes GCC to remember the state of the diagnostics as of each `push`, and restore to that point at each `pop`. If a `pop` has no matching `push`, the command-line options are restored.

```
#pragma GCC diagnostic error "-Wuninitialized"
  foo(a);                       /* error is given for this one */
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wuninitialized"
  foo(b);                       /* no diagnostic for this one */
#pragma GCC diagnostic pop
  foo(c);                       /* error is given for this one */
#pragma GCC diagnostic pop
  foo(d);                       /* depends on command-line options */
```

GCC also offers a simple mechanism for printing messages during compilation.

`#pragma message` *`string`*

Prints *string* as a compiler message on compilation. The message is informational only, and is neither a compilation warning nor an error. Newlines can be included in the string by using the '`\n`' escape sequence.

```
#pragma message "Compiling " __FILE__ "..."
```

string may be parenthesized, and is printed with location information. For example,

```
#define DO_PRAGMA(x) _Pragma (#x)
#define TODO(x) DO_PRAGMA(message ("TODO - " #x))

TODO(Remember to fix this)
```

prints '`/tmp/file.c:4: note: #pragma message: TODO - Remember to fix this`'.

`#pragma GCC error` *`message`*

Generates an error message. This pragma *is* considered to indicate an error in the compilation, and it will be treated as such.

Newlines can be included in the string by using the '`\n`' escape sequence. They will be displayed as newlines even if the '`-fmessage-length`' option is set to zero.

The error is only generated if the pragma is present in the code after preprocessing has been completed. It does not matter however if the code containing the pragma is unreachable:

```
#if 0
#pragma GCC error "this error is not seen"
#endif
void foo (void)
{
  return;
#pragma GCC error "this error is seen"
}
```

`#pragma GCC warning` *`message`*

This is just like '`pragma GCC error`' except that a warning message is issued instead of an error message. Unless '`-Werror`' is in effect, in which case this pragma will generate an error as well.

6.62.13 Visibility Pragmas

`#pragma GCC visibility push(`*`visibility`*`)`
`#pragma GCC visibility pop`

This pragma allows the user to set the visibility for multiple declarations without having to give each a visibility attribute (see Section 6.33 [Function Attributes], page 477).

In C++, '`#pragma GCC visibility`' affects only namespace-scope declarations. Class members and template specializations are not affected; if you want to override the visibility for a particular member or instantiation, you must use an attribute.

6.62.14 Push/Pop Macro Pragmas

For compatibility with Microsoft Windows compilers, GCC supports '`#pragma push_macro("macro_name")`' and '`#pragma pop_macro("macro_name")`'.

`#pragma push_macro("macro_name")`
This pragma saves the value of the macro named as *macro_name* to the top of the stack for this macro.

`#pragma pop_macro("macro_name")`
This pragma sets the value of the macro named as *macro_name* to the value on top of the stack for this macro. If the stack for *macro_name* is empty, the value of the macro remains unchanged.

For example:

```
#define X  1
#pragma push_macro("X")
#undef X
#define X -1
#pragma pop_macro("X")
int x [X];
```

In this example, the definition of X as 1 is saved by `#pragma push_macro` and restored by `#pragma pop_macro`.

6.62.15 Function Specific Option Pragmas

`#pragma GCC target (string, ...)`
This pragma allows you to set target-specific options for functions defined later in the source file. One or more strings can be specified. Each function that is defined after this point is treated as if it had been declared with one `target(string)` attribute for each *string* argument. The parentheses around the strings in the pragma are optional. See Section 6.33 [Function Attributes], page 477, for more information about the `target` attribute and the attribute syntax.

The `#pragma GCC target` pragma is presently implemented for x86, ARM, AArch64, PowerPC, S/390, and Nios II targets only.

`#pragma GCC optimize (string, ...)`
This pragma allows you to set global optimization options for functions defined later in the source file. One or more strings can be specified. Each function that is defined after this point is treated as if it had been declared with one `optimize(string)` attribute for each *string* argument. The parentheses around the strings in the pragma are optional. See Section 6.33 [Function Attributes], page 477, for more information about the `optimize` attribute and the attribute syntax.

`#pragma GCC push_options`
`#pragma GCC pop_options`
These pragmas maintain a stack of the current target and optimization options. It is intended for include files where you temporarily want to switch to using a different '`#pragma GCC target`' or '`#pragma GCC optimize`' and then to pop back to the previous options.

`#pragma GCC reset_options`

This pragma clears the current `#pragma GCC target` and `#pragma GCC optimize` to use the default switches as specified on the command line.

6.62.16 Loop-Specific Pragmas

`#pragma GCC ivdep`

With this pragma, the programmer asserts that there are no loop-carried dependencies which would prevent consecutive iterations of the following loop from executing concurrently with SIMD (single instruction multiple data) instructions.

For example, the compiler can only unconditionally vectorize the following loop with the pragma:

```
void foo (int n, int *a, int *b, int *c)
{
  int i, j;
#pragma GCC ivdep
  for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
}
```

In this example, using the `restrict` qualifier had the same effect. In the following example, that would not be possible. Assume $k < -m$ or $k >= m$. Only with the pragma, the compiler knows that it can unconditionally vectorize the following loop:

```
void ignore_vec_dep (int *a, int k, int c, int m)
{
#pragma GCC ivdep
  for (int i = 0; i < m; i++)
    a[i] = a[i + k] * c;
}
```

`#pragma GCC unroll n`

You can use this pragma to control how many times a loop should be unrolled. It must be placed immediately before a `for`, `while` or `do` loop or a `#pragma GCC ivdep`, and applies only to the loop that follows. *n* is an integer constant expression specifying the unrolling factor. The values of 0 and 1 block any unrolling of the loop.

6.63 Unnamed Structure and Union Fields

As permitted by ISO C11 and for compatibility with other compilers, GCC allows you to define a structure or union that contains, as fields, structures and unions without names. For example:

```
struct {
  int a;
  union {
    int b;
    float c;
  };
  int d;
} foo;
```

In this example, you are able to access members of the unnamed union with code like '`foo.b`'. Note that only unnamed structs and unions are allowed, you may not have, for example, an unnamed `int`.

You must never create such structures that cause ambiguous field definitions. For example, in this structure:

```
struct {
  int a;
  struct {
    int a;
  };
} foo;
```

it is ambiguous which `a` is being referred to with '`foo.a`'. The compiler gives errors for such constructs.

Unless '`-fms-extensions`' is used, the unnamed field must be a structure or union definition without a tag (for example, '`struct { int a; };`'). If '`-fms-extensions`' is used, the field may also be a definition with a tag such as '`struct foo { int a; };`', a reference to a previously defined structure or union such as '`struct foo;`', or a reference to a `typedef` name for a previously defined structure or union type.

The option '`-fplan9-extensions`' enables '`-fms-extensions`' as well as two other extensions. First, a pointer to a structure is automatically converted to a pointer to an anonymous field for assignments and function calls. For example:

```
struct s1 { int a; };
struct s2 { struct s1; };
extern void f1 (struct s1 *);
void f2 (struct s2 *p) { f1 (p); }
```

In the call to `f1` inside `f2`, the pointer `p` is converted into a pointer to the anonymous field.

Second, when the type of an anonymous field is a `typedef` for a `struct` or `union`, code may refer to the field using the name of the `typedef`.

```
typedef struct { int a; } s1;
struct s2 { s1; };
s1 f1 (struct s2 *p) { return p->s1; }
```

These usages are only permitted when they are not ambiguous.

6.64 Thread-Local Storage

Thread-local storage (TLS) is a mechanism by which variables are allocated such that there is one instance of the variable per extant thread. The runtime model GCC uses to implement this originates in the IA-64 processor-specific ABI, but has since been migrated to other processors as well. It requires significant support from the linker (`ld`), dynamic linker (`ld.so`), and system libraries ('`libc.so`' and '`libpthread.so`'), so it is not available everywhere.

At the user level, the extension is visible with a new storage class keyword: `__thread`. For example:

```
__thread int i;
extern __thread struct state s;
static __thread char *p;
```

The `__thread` specifier may be used alone, with the `extern` or `static` specifiers, but with no other storage class specifier. When used with `extern` or `static`, `__thread` must appear immediately after the other storage class specifier.

The `__thread` specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data member.

When the address-of operator is applied to a thread-local variable, it is evaluated at run time and returns the address of the current thread's instance of that variable. An address so obtained may be used by any thread. When a thread terminates, any pointers to thread-local variables in that thread become invalid.

No static initialization may refer to the address of a thread-local variable.

In C++, if an initializer is present for a thread-local variable, it must be a *constant-expression*, as defined in 5.19.2 of the ANSI/ISO C++ standard.

See ELF Handling For Thread-Local Storage (`https://www.akkadia.org/drepper/tls.pdf`) for a detailed explanation of the four thread-local storage addressing models, and how the runtime is expected to function.

6.64.1 ISO/IEC 9899:1999 Edits for Thread-Local Storage

The following are a set of changes to ISO/IEC 9899:1999 (aka C99) that document the exact semantics of the language extension.

- *5.1.2 Execution environments*

  Add new text after paragraph 1

  > Within either execution environment, a *thread* is a flow of control within a program. It is implementation defined whether or not there may be more than one thread associated with a program. It is implementation defined how threads beyond the first are created, the name and type of the function called at thread startup, and how threads may be terminated. However, objects with thread storage duration shall be initialized before thread startup.

- *6.2.4 Storage durations of objects*

  Add new text before paragraph 3

  > An object whose identifier is declared with the storage-class specifier `__thread` has *thread storage duration*. Its lifetime is the entire execution of the thread, and its stored value is initialized only once, prior to thread startup.

- *6.4.1 Keywords*

  Add `__thread`.

- *6.7.1 Storage-class specifiers*

  Add `__thread` to the list of storage class specifiers in paragraph 1.

  Change paragraph 2 to

  > With the exception of `__thread`, at most one storage-class specifier may be given [. . .]. The `__thread` specifier may be used alone, or immediately following `extern` or `static`.

Add new text after paragraph 6

> The declaration of an identifier for a variable that has block scope that specifies `__thread` shall also specify either `extern` or `static`.
>
> The `__thread` specifier shall be used only with variables.

6.64.2 ISO/IEC 14882:1998 Edits for Thread-Local Storage

The following are a set of changes to ISO/IEC 14882:1998 (aka C++98) that document the exact semantics of the language extension.

- **[intro.execution]**

  New text after paragraph 4

  > A *thread* is a flow of control within the abstract machine. It is implementation defined whether or not there may be more than one thread.

  New text after paragraph 7

  > It is unspecified whether additional action must be taken to ensure when and whether side effects are visible to other threads.

- **[lex.key]**

  Add `__thread`.

- **[basic.start.main]**

  Add after paragraph 5

  > The thread that begins execution at the `main` function is called the *main thread*. It is implementation defined how functions beginning threads other than the main thread are designated or typed. A function so designated, as well as the `main` function, is called a *thread startup function*. It is implementation defined what happens if a thread startup function returns. It is implementation defined what happens to other threads when any thread calls `exit`.

- **[basic.start.init]**

  Add after paragraph 4

  > The storage for an object of thread storage duration shall be statically initialized before the first statement of the thread startup function. An object of thread storage duration shall not require dynamic initialization.

- **[basic.start.term]**

  Add after paragraph 3

  > The type of an object with thread storage duration shall not have a non-trivial destructor, nor shall it be an array type whose elements (directly or indirectly) have non-trivial destructors.

- **[basic.stc]**

  Add "thread storage duration" to the list in paragraph 1.

  Change paragraph 2

  > Thread, static, and automatic storage durations are associated with objects introduced by declarations [. . .].

  Add `__thread` to the list of specifiers in paragraph 3.

- **[basic.stc.thread]**

  New section before **[basic.stc.static]**

  > The keyword `__thread` applied to a non-local object gives the object thread storage duration.
  >
  > A local variable or class data member declared both `static` and `__thread` gives the variable or member thread storage duration.

- **[basic.stc.static]**

  Change paragraph 1

  > All objects that have neither thread storage duration, dynamic storage duration nor are local [. . .].

- **[dcl.stc]**

  Add `__thread` to the list in paragraph 1.

  Change paragraph 1

  > With the exception of `__thread`, at most one *storage-class-specifier* shall appear in a given *decl-specifier-seq*. The `__thread` specifier may be used alone, or immediately following the `extern` or `static` specifiers. [. . .]

  Add after paragraph 5

  > The `__thread` specifier can be applied only to the names of objects and to anonymous unions.

- **[class.mem]**

  Add after paragraph 6

  > Non-`static` members shall not be `__thread`.

6.65 Binary Constants using the '`0b`' Prefix

Integer constants can be written as binary constants, consisting of a sequence of '`0`' and '`1`' digits, prefixed by '`0b`' or '`0B`'. This is particularly useful in environments that operate a lot on the bit level (like microcontrollers).

The following statements are identical:

```
i =       42;
i =     0x2a;
i =      052;
i = 0b101010;
```

The type of these constants follows the same rules as for octal or hexadecimal integer constants, so suffixes like '`L`' or '`UL`' can be applied.

7 Extensions to the C++ Language

The GNU compiler provides these extensions to the C++ language (and you can also use most of the C language extensions in your C++ programs). If you want to write code that checks whether these features are available, you can test for the GNU compiler the same way as for C programs: check for a predefined macro `__GNUC__`. You can also use `__GNUG__` to test specifically for GNU C++ (see Section "Predefined Macros" in *The GNU C Preprocessor*).

7.1 When is a Volatile C++ Object Accessed?

The C++ standard differs from the C standard in its treatment of volatile objects. It fails to specify what constitutes a volatile access, except to say that C++ should behave in a similar manner to C with respect to volatiles, where possible. However, the different lvalueness of expressions between C and C++ complicate the behavior. G++ behaves the same as GCC for volatile access, See Chapter 6 [Volatiles], page 451, for a description of GCC's behavior.

The C and C++ language specifications differ when an object is accessed in a void context:

```
volatile int *src = somevalue;
*src;
```

The C++ standard specifies that such expressions do not undergo lvalue to rvalue conversion, and that the type of the dereferenced object may be incomplete. The C++ standard does not specify explicitly that it is lvalue to rvalue conversion that is responsible for causing an access. There is reason to believe that it is, because otherwise certain simple expressions become undefined. However, because it would surprise most programmers, G++ treats dereferencing a pointer to volatile object of complete type as GCC would do for an equivalent type in C. When the object has incomplete type, G++ issues a warning; if you wish to force an error, you must force a conversion to rvalue with, for instance, a static cast.

When using a reference to volatile, G++ does not treat equivalent expressions as accesses to volatiles, but instead issues a warning that no volatile is accessed. The rationale for this is that otherwise it becomes difficult to determine where volatile access occur, and not possible to ignore the return value from functions returning volatile references. Again, if you wish to force a read, cast the reference to an rvalue.

G++ implements the same behavior as GCC does when assigning to a volatile object—there is no reread of the assigned-to object, the assigned rvalue is reused. Note that in C++ assignment expressions are lvalues, and if used as an lvalue, the volatile object is referred to. For instance, *vref* refers to *vobj*, as expected, in the following example:

```
volatile int vobj;
volatile int &vref = vobj = something;
```

7.2 Restricting Pointer Aliasing

As with the C front end, G++ understands the C99 feature of restricted pointers, specified with the `__restrict__`, or `__restrict` type qualifier. Because you cannot compile C++ by specifying the '`-std=c99`' language flag, `restrict` is not a keyword in C++.

In addition to allowing restricted pointers, you can specify restricted references, which indicate that the reference is not aliased in the local context.

```
void fn (int *__restrict__ rptr, int &__restrict__ rref)
{
  /* ... */
}
```

In the body of `fn`, *rptr* points to an unaliased integer and *rref* refers to a (different) unaliased integer.

You may also specify whether a member function's *this* pointer is unaliased by using `__restrict__` as a member function qualifier.

```
void T::fn () __restrict__
{
  /* ... */
}
```

Within the body of `T::fn`, *this* has the effective definition `T *__restrict__ const this`. Notice that the interpretation of a `__restrict__` member function qualifier is different to that of `const` or `volatile` qualifier, in that it is applied to the pointer rather than the object. This is consistent with other compilers that implement restricted pointers.

As with all outermost parameter qualifiers, `__restrict__` is ignored in function definition matching. This means you only need to specify `__restrict__` in a function definition, rather than in a function prototype as well.

7.3 Vague Linkage

There are several constructs in C++ that require space in the object file but are not clearly tied to a single translation unit. We say that these constructs have "vague linkage". Typically such constructs are emitted wherever they are needed, though sometimes we can be more clever.

Inline Functions

Inline functions are typically defined in a header file which can be included in many different compilations. Hopefully they can usually be inlined, but sometimes an out-of-line copy is necessary, if the address of the function is taken or if inlining fails. In general, we emit an out-of-line copy in all translation units where one is needed. As an exception, we only emit inline virtual functions with the vtable, since it always requires a copy.

Local static variables and string constants used in an inline function are also considered to have vague linkage, since they must be shared between all inlined and out-of-line instances of the function.

VTables

C++ virtual functions are implemented in most compilers using a lookup table, known as a vtable. The vtable contains pointers to the virtual functions provided by a class, and each object of the class contains a pointer to its vtable (or vtables, in some multiple-inheritance situations). If the class declares any non-inline, non-pure virtual functions, the first one is chosen as the "key method" for the class, and the vtable is only emitted in the translation unit where the key method is defined.

Note: If the chosen key method is later defined as inline, the vtable is still emitted in every translation unit that defines it. Make sure that any inline virtuals are declared inline in the class body, even if they are not defined there.

`type_info` objects
: C++ requires information about types to be written out in order to implement '`dynamic_cast`', '`typeid`' and exception handling. For polymorphic classes (classes with virtual functions), the '`type_info`' object is written out along with the vtable so that '`dynamic_cast`' can determine the dynamic type of a class object at run time. For all other types, we write out the '`type_info`' object when it is used: when applying '`typeid`' to an expression, throwing an object, or referring to a type in a catch clause or exception specification.

Template Instantiations
: Most everything in this section also applies to template instantiations, but there are other options as well. See Section 7.5 [Where's the Template?], page 818.

When used with GNU ld version 2.8 or later on an ELF system such as GNU/Linux or Solaris 2, or on Microsoft Windows, duplicate copies of these constructs will be discarded at link time. This is known as COMDAT support.

On targets that don't support COMDAT, but do support weak symbols, GCC uses them. This way one copy overrides all the others, but the unused copies still take up space in the executable.

For targets that do not support either COMDAT or weak symbols, most entities with vague linkage are emitted as local symbols to avoid duplicate definition errors from the linker. This does not happen for local statics in inlines, however, as having multiple copies almost certainly breaks things.

See Section 7.4 [Declarations and Definitions in One Header], page 817, for another way to control placement of these constructs.

7.4 C++ Interface and Implementation Pragmas

`#pragma interface` and `#pragma implementation` provide the user with a way of explicitly directing the compiler to emit entities with vague linkage (and debugging information) in a particular translation unit.

Note: These `#pragma`s have been superceded as of GCC 2.7.2 by COMDAT support and the "key method" heuristic mentioned in Section 7.3 [Vague Linkage], page 816. Using them can actually cause your program to grow due to unnecessary out-of-line copies of inline functions.

`#pragma interface`
`#pragma interface "`*`subdir/objects`*`.h"`
: Use this directive in *header files* that define object classes, to save space in most of the object files that use those classes. Normally, local copies of certain information (backup copies of inline member functions, debugging information, and the internal tables that implement virtual functions) must be kept in each object file that includes class definitions. You can use this pragma to avoid such duplication. When a header file containing '`#pragma interface`' is included in a compilation, this auxiliary information is not generated (unless the main input source file itself uses '`#pragma implementation`'). Instead, the object files contain references to be resolved at link time.

The second form of this directive is useful for the case where you have multiple headers with the same name in different directories. If you use this form, you must specify the same string to '`#pragma implementation`'.

`#pragma implementation`
`#pragma implementation "`*`objects`*`.h"`

Use this pragma in a *main input file*, when you want full output from included header files to be generated (and made globally visible). The included header file, in turn, should use '`#pragma interface`'. Backup copies of inline member functions, debugging information, and the internal tables used to implement virtual functions are all generated in implementation files.

If you use '`#pragma implementation`' with no argument, it applies to an include file with the same basename[1] as your source file. For example, in '`allclass.cc`', giving just '`#pragma implementation`' by itself is equivalent to '`#pragma implementation "allclass.h"`'.

Use the string argument if you want a single implementation file to include code from multiple header files. (You must also use '`#include`' to include the header file; '`#pragma implementation`' only specifies how to use the file—it doesn't actually include it.)

There is no way to split up the contents of a single header file into multiple implementation files.

'`#pragma implementation`' and '`#pragma interface`' also have an effect on function inlining.

If you define a class in a header file marked with '`#pragma interface`', the effect on an inline function defined in that class is similar to an explicit `extern` declaration—the compiler emits no code at all to define an independent version of the function. Its definition is used only for inlining with its callers.

Conversely, when you include the same header file in a main source file that declares it as '`#pragma implementation`', the compiler emits code for the function itself; this defines a version of the function that can be found via pointers (or by callers compiled without inlining). If all calls to the function can be inlined, you can avoid emitting the function by compiling with '`-fno-implement-inlines`'. If any calls are not inlined, you will get linker errors.

7.5 Where's the Template?

C++ templates were the first language feature to require more intelligence from the environment than was traditionally found on a UNIX system. Somehow the compiler and linker have to make sure that each template instance occurs exactly once in the executable if it is needed, and not at all otherwise. There are two basic approaches to this problem, which are referred to as the Borland model and the Cfront model.

Borland model

Borland C++ solved the template instantiation problem by adding the code equivalent of common blocks to their linker; the compiler emits template in-

[1] A file's *basename* is the name stripped of all leading path information and of trailing suffixes, such as '`.h`' or '`.C`' or '`.cc`'.

stances in each translation unit that uses them, and the linker collapses them together. The advantage of this model is that the linker only has to consider the object files themselves; there is no external complexity to worry about. The disadvantage is that compilation time is increased because the template code is being compiled repeatedly. Code written for this model tends to include definitions of all templates in the header file, since they must be seen to be instantiated.

Cfront model
: The AT&T C++ translator, Cfront, solved the template instantiation problem by creating the notion of a template repository, an automatically maintained place where template instances are stored. A more modern version of the repository works as follows: As individual object files are built, the compiler places any template definitions and instantiations encountered in the repository. At link time, the link wrapper adds in the objects in the repository and compiles any needed instances that were not previously emitted. The advantages of this model are more optimal compilation speed and the ability to use the system linker; to implement the Borland model a compiler vendor also needs to replace the linker. The disadvantages are vastly increased complexity, and thus potential for error; for some code this can be just as transparent, but in practice it can been very difficult to build multiple programs in one directory and one program in multiple directories. Code written for this model tends to separate definitions of non-inline member templates into a separate file, which should be compiled separately.

G++ implements the Borland model on targets where the linker supports it, including ELF targets (such as GNU/Linux), Mac OS X and Microsoft Windows. Otherwise G++ implements neither automatic model.

You have the following options for dealing with template instantiations:

1. Do nothing. Code written for the Borland model works fine, but each translation unit contains instances of each of the templates it uses. The duplicate instances will be discarded by the linker, but in a large program, this can lead to an unacceptable amount of code duplication in object files or shared libraries.

   Duplicate instances of a template can be avoided by defining an explicit instantiation in one object file, and preventing the compiler from doing implicit instantiations in any other object files by using an explicit instantiation declaration, using the `extern template` syntax:

   ```
   extern template int max (int, int);
   ```

   This syntax is defined in the C++ 2011 standard, but has been supported by G++ and other compilers since well before 2011.

   Explicit instantiations can be used for the largest or most frequently duplicated instances, without having to know exactly which other instances are used in the rest of the program. You can scatter the explicit instantiations throughout your program, perhaps putting them in the translation units where the instances are used or the translation units that define the templates themselves; you can put all of the explicit instantiations you need into one big file; or you can create small files like

   ```
   #include "Foo.h"
   ```

```
#include "Foo.cc"

template class Foo<int>;
template ostream& operator <<
                (ostream&, const Foo<int>&);
```

for each of the instances you need, and create a template instantiation library from those.

This is the simplest option, but also offers flexibility and fine-grained control when necessary. It is also the most portable alternative and programs using this approach will work with most modern compilers.

2. Compile your template-using code with '`-frepo`'. The compiler generates files with the extension '`.rpo`' listing all of the template instantiations used in the corresponding object files that could be instantiated there; the link wrapper, '`collect2`', then updates the '`.rpo`' files to tell the compiler where to place those instantiations and rebuild any affected object files. The link-time overhead is negligible after the first pass, as the compiler continues to place the instantiations in the same files.

   This can be a suitable option for application code written for the Borland model, as it usually just works. Code written for the Cfront model needs to be modified so that the template definitions are available at one or more points of instantiation; usually this is as simple as adding `#include <tmethods.cc>` to the end of each template header.

   For library code, if you want the library to provide all of the template instantiations it needs, just try to link all of its object files together; the link will fail, but cause the instantiations to be generated as a side effect. Be warned, however, that this may cause conflicts if multiple libraries try to provide the same instantiations. For greater control, use explicit instantiation as described in the next option.

3. Compile your code with '`-fno-implicit-templates`' to disable the implicit generation of template instances, and explicitly instantiate all the ones you use. This approach requires more knowledge of exactly which instances you need than do the others, but it's less mysterious and allows greater control if you want to ensure that only the intended instances are used.

   If you are using Cfront-model code, you can probably get away with not using '`-fno-implicit-templates`' when compiling files that don't '`#include`' the member template definitions.

   If you use one big file to do the instantiations, you may want to compile it without '`-fno-implicit-templates`' so you get all of the instances required by your explicit instantiations (but not by any other files) without having to specify them as well.

   In addition to forward declaration of explicit instantiations (with `extern`), G++ has extended the template instantiation syntax to support instantiation of the compiler support data for a template class (i.e. the vtable) without instantiating any of its members (with `inline`), and instantiation of only the static data members of a template class, without the support data or member functions (with `static`):

```
inline template class Foo<int>;
static template class Foo<int>;
```

7.6 Extracting the Function Pointer from a Bound Pointer to Member Function

In C++, pointer to member functions (PMFs) are implemented using a wide pointer of sorts to handle all the possible call mechanisms; the PMF needs to store information about how to adjust the '`this`' pointer, and if the function pointed to is virtual, where to find the vtable, and where in the vtable to look for the member function. If you are using PMFs in an inner loop, you should really reconsider that decision. If that is not an option, you can extract the pointer to the function that would be called for a given object/PMF pair and call it directly inside the inner loop, to save a bit of time.

Note that you still pay the penalty for the call through a function pointer; on most modern architectures, such a call defeats the branch prediction features of the CPU. This is also true of normal virtual function calls.

The syntax for this extension is

```
extern A a;
extern int (A::*fp)();
typedef int (*fptr)(A *);

fptr p = (fptr)(a.*fp);
```

For PMF constants (i.e. expressions of the form '`&Klasse::Member`'), no object is needed to obtain the address of the function. They can be converted to function pointers directly:

```
fptr p1 = (fptr)(&A::foo);
```

You must specify '`-Wno-pmf-conversions`' to use this extension.

7.7 C++-Specific Variable, Function, and Type Attributes

Some attributes only make sense for C++ programs.

`abi_tag ("tag", ...)`

The `abi_tag` attribute can be applied to a function, variable, or class declaration. It modifies the mangled name of the entity to incorporate the tag name, in order to distinguish the function or class from an earlier version with a different ABI; perhaps the class has changed size, or the function has a different return type that is not encoded in the mangled name.

The attribute can also be applied to an inline namespace, but does not affect the mangled name of the namespace; in this case it is only used for '`-Wabi-tag`' warnings and automatic tagging of functions and variables. Tagging inline namespaces is generally preferable to tagging individual declarations, but the latter is sometimes necessary, such as when only certain members of a class need to be tagged.

The argument can be a list of strings of arbitrary length. The strings are sorted on output, so the order of the list is unimportant.

A redeclaration of an entity must not add new ABI tags, since doing so would change the mangled name.

The ABI tags apply to a name, so all instantiations and specializations of a template have the same tags. The attribute will be ignored if applied to an explicit specialization or instantiation.

The '`-Wabi-tag`' flag enables a warning about a class which does not have all the ABI tags used by its subobjects and virtual functions; for users with code

that needs to coexist with an earlier ABI, using this option can help to find all affected types that need to be tagged.

When a type involving an ABI tag is used as the type of a variable or return type of a function where that tag is not already present in the signature of the function, the tag is automatically applied to the variable or function. '`-Wabi-tag`' also warns about this situation; this warning can be avoided by explicitly tagging the variable or function or moving it into a tagged inline namespace.

`init_priority (priority)`

In Standard C++, objects defined at namespace scope are guaranteed to be initialized in an order in strict accordance with that of their definitions *in a given translation unit*. No guarantee is made for initializations across translation units. However, GNU C++ allows users to control the order of initialization of objects defined at namespace scope with the `init_priority` attribute by specifying a relative *priority*, a constant integral expression currently bounded between 101 and 65535 inclusive. Lower numbers indicate a higher priority.

In the following example, `A` would normally be created before `B`, but the `init_priority` attribute reverses that order:

```
Some_Class  A  __attribute__ ((init_priority (2000)));
Some_Class  B  __attribute__ ((init_priority (543)));
```

Note that the particular values of *priority* do not matter; only their relative ordering.

`warn_unused`

For C++ types with non-trivial constructors and/or destructors it is impossible for the compiler to determine whether a variable of this type is truly unused if it is not referenced. This type attribute informs the compiler that variables of this type should be warned about if they appear to be unused, just like variables of fundamental types.

This attribute is appropriate for types which just represent a value, such as `std::string`; it is not appropriate for types which control a resource, such as `std::lock_guard`.

This attribute is also accepted in C, but it is unnecessary because C does not have constructors or destructors.

7.8 Function Multiversioning

With the GNU C++ front end, for x86 targets, you may specify multiple versions of a function, where each function is specialized for a specific target feature. At runtime, the appropriate version of the function is automatically executed depending on the characteristics of the execution platform. Here is an example.

```
__attribute__ ((target ("default")))
int foo ()
{
  // The default version of foo.
  return 0;
}
```

```
__attribute__ ((target ("sse4.2")))
int foo ()
{
  // foo version for SSE4.2
  return 1;
}

__attribute__ ((target ("arch=atom")))
int foo ()
{
  // foo version for the Intel ATOM processor
  return 2;
}

__attribute__ ((target ("arch=amdfam10")))
int foo ()
{
  // foo version for the AMD Family 0x10 processors.
  return 3;
}

int main ()
{
  int (*p)() = &foo;
  assert ((*p) () == foo ());
  return 0;
}
```

In the above example, four versions of function foo are created. The first version of foo with the target attribute "default" is the default version. This version gets executed when no other target specific version qualifies for execution on a particular platform. A new version of foo is created by using the same function signature but with a different target string. Function foo is called or a pointer to it is taken just like a regular function. GCC takes care of doing the dispatching to call the right version at runtime. Refer to the GCC wiki on Function Multiversioning (`http://gcc.gnu.org/wiki/FunctionMultiVersioning`) for more details.

7.9 Type Traits

The C++ front end implements syntactic extensions that allow compile-time determination of various characteristics of a type (or of a pair of types).

`__has_nothrow_assign (type)`
: If `type` is `const`-qualified or is a reference type then the trait is `false`. Otherwise if `__has_trivial_assign (type)` is `true` then the trait is `true`, else if `type` is a cv-qualified class or union type with copy assignment operators that are known not to throw an exception then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_nothrow_copy (type)`
: If `__has_trivial_copy (type)` is `true` then the trait is `true`, else if `type` is a cv-qualified class or union type with copy constructors that are known not to throw an exception then the trait is `true`, else it is `false`. Requires: `type`

shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_nothrow_constructor (type)`
: If `__has_trivial_constructor (type)` is `true` then the trait is `true`, else if `type` is a cv class or union type (or array thereof) with a default constructor that is known not to throw an exception then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_trivial_assign (type)`
: If `type` is `const`- qualified or is a reference type then the trait is `false`. Otherwise if `__is_pod (type)` is `true` then the trait is `true`, else if `type` is a cv-qualified class or union type with a trivial copy assignment ([class.copy]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_trivial_copy (type)`
: If `__is_pod (type)` is `true` or `type` is a reference type then the trait is `true`, else if `type` is a cv class or union type with a trivial copy constructor ([class.copy]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_trivial_constructor (type)`
: If `__is_pod (type)` is `true` then the trait is `true`, else if `type` is a cv-qualified class or union type (or array thereof) with a trivial default constructor ([class.ctor]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_trivial_destructor (type)`
: If `__is_pod (type)` is `true` or `type` is a reference type then the trait is `true`, else if `type` is a cv class or union type (or array thereof) with a trivial destructor ([class.dtor]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__has_virtual_destructor (type)`
: If `type` is a class type with a virtual destructor ([class.dtor]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_abstract (type)`
: If `type` is an abstract class ([class.abstract]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_base_of (base_type, derived_type)`
: If `base_type` is a base class of `derived_type` ([class.derived]) then the trait is `true`, otherwise it is `false`. Top-level cv-qualifications of `base_type` and `derived_type` are ignored. For the purposes of this trait, a class type is considered is own base. Requires: if `__is_class (base_type)` and `__is_class (derived_type)` are `true` and `base_type` and `derived_type` are not the same

type (disregarding cv-qualifiers), `derived_type` shall be a complete type. A diagnostic is produced if this requirement is not met.

`__is_class (type)`
: If `type` is a cv-qualified class type, and not a union type ([basic.compound]) the trait is `true`, else it is `false`.

`__is_empty (type)`
: If `__is_class (type)` is `false` then the trait is `false`. Otherwise `type` is considered empty if and only if: `type` has no non-static data members, or all non-static data members, if any, are bit-fields of length 0, and `type` has no virtual members, and `type` has no virtual base classes, and `type` has no base classes `base_type` for which `__is_empty (base_type)` is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_enum (type)`
: If `type` is a cv enumeration type ([basic.compound]) the trait is `true`, else it is `false`.

`__is_literal_type (type)`
: If `type` is a literal type ([basic.types]) the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_pod (type)`
: If `type` is a cv POD type ([basic.types]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_polymorphic (type)`
: If `type` is a polymorphic class ([class.virtual]) then the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_standard_layout (type)`
: If `type` is a standard-layout type ([basic.types]) the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_trivial (type)`
: If `type` is a trivial type ([basic.types]) the trait is `true`, else it is `false`. Requires: `type` shall be a complete type, (possibly cv-qualified) `void`, or an array of unknown bound.

`__is_union (type)`
: If `type` is a cv union type ([basic.compound]) the trait is `true`, else it is `false`.

`__underlying_type (type)`
: The underlying type of `type`. Requires: `type` shall be an enumeration type ([dcl.enum]).

`__integer_pack (length)`
: When used as the pattern of a pack expansion within a template definition, expands to a template argument pack containing integers from `0` to `length-1`. This is provided for efficient implementation of `std::make_integer_sequence`.

7.10 C++ Concepts

C++ concepts provide much-improved support for generic programming. In particular, they allow the specification of constraints on template arguments. The constraints are used to extend the usual overloading and partial specialization capabilities of the language, allowing generic data structures and algorithms to be "refined" based on their properties rather than their type names.

The following keywords are reserved for concepts.

`assumes`
: States an expression as an assumption, and if possible, verifies that the assumption is valid. For example, `assume(n > 0)`.

`axiom`
: Introduces an axiom definition. Axioms introduce requirements on values.

`forall`
: Introduces a universally quantified object in an axiom. For example, `forall (int n) n + 0 == n`).

`concept`
: Introduces a concept definition. Concepts are sets of syntactic and semantic requirements on types and their values.

`requires`
: Introduces constraints on template arguments or requirements for a member function of a class template.

The front end also exposes a number of internal mechanism that can be used to simplify the writing of type traits. Note that some of these traits are likely to be removed in the future.

`__is_same (type1, type2)`
: A binary type trait: `true` whenever the type arguments are the same.

7.11 Deprecated Features

In the past, the GNU C++ compiler was extended to experiment with new features, at a time when the C++ language was still evolving. Now that the C++ standard is complete, some of those features are superseded by superior alternatives. Using the old features might cause a warning in some cases that the feature will be dropped in the future. In other cases, the feature might be gone already.

G++ allows a virtual function returning '`void *`' to be overridden by one returning a different pointer type. This extension to the covariant return type rules is now deprecated and will be removed from a future version.

The use of default arguments in function pointers, function typedefs and other places where they are not permitted by the standard is deprecated and will be removed from a future version of G++.

G++ allows floating-point literals to appear in integral constant expressions, e.g. ' `enum E { e = int(2.2 * 3.7) }` ' This extension is deprecated and will be removed from a future version.

G++ allows static data members of const floating-point type to be declared with an initializer in a class definition. The standard only allows initializers for static members of const integral types and const enumeration types so this extension has been deprecated and will be removed from a future version.

G++ allows attributes to follow a parenthesized direct initializer, e.g. ‘ `int f (0) __attribute__ ((something));` ’ This extension has been ignored since G++ 3.3 and is deprecated.

G++ allows anonymous structs and unions to have members that are not public non-static data members (i.e. fields). These extensions are deprecated.

7.12 Backwards Compatibility

Now that there is a definitive ISO standard C++, G++ has a specification to adhere to. The C++ language evolved over time, and features that used to be acceptable in previous drafts of the standard, such as the ARM [Annotated C++ Reference Manual], are no longer accepted. In order to allow compilation of C++ written to such drafts, G++ contains some backwards compatibilities. *All such backwards compatibility features are liable to disappear in future versions of G++.* They should be considered deprecated. See Section 7.11 [Deprecated Features], page 826.

`Implicit C language`

Old C system header files did not contain an `extern "C" {...}` scope to set the language. On such systems, all system header files are implicitly scoped inside a C language scope. Such headers must correctly prototype function argument types, there is no leeway for `()` to indicate an unspecified set of arguments.

8 GNU Objective-C Features

This document is meant to describe some of the GNU Objective-C features. It is not intended to teach you Objective-C. There are several resources on the Internet that present the language.

8.1 GNU Objective-C Runtime API

This section is specific for the GNU Objective-C runtime. If you are using a different runtime, you can skip it.

The GNU Objective-C runtime provides an API that allows you to interact with the Objective-C runtime system, querying the live runtime structures and even manipulating them. This allows you for example to inspect and navigate classes, methods and protocols; to define new classes or new methods, and even to modify existing classes or protocols.

If you are using a "Foundation" library such as GNUstep-Base, this library will provide you with a rich set of functionality to do most of the inspection tasks, and you probably will only need direct access to the GNU Objective-C runtime API to define new classes or methods.

8.1.1 Modern GNU Objective-C Runtime API

The GNU Objective-C runtime provides an API which is similar to the one provided by the "Objective-C 2.0" Apple/NeXT Objective-C runtime. The API is documented in the public header files of the GNU Objective-C runtime:

- '`objc/objc.h`': this is the basic Objective-C header file, defining the basic Objective-C types such as `id`, `Class` and `BOOL`. You have to include this header to do almost anything with Objective-C.
- '`objc/runtime.h`': this header declares most of the public runtime API functions allowing you to inspect and manipulate the Objective-C runtime data structures. These functions are fairly standardized across Objective-C runtimes and are almost identical to the Apple/NeXT Objective-C runtime ones. It does not declare functions in some specialized areas (constructing and forwarding message invocations, threading) which are in the other headers below. You have to include '`objc/objc.h`' and '`objc/runtime.h`' to use any of the functions, such as `class_getName()`, declared in '`objc/runtime.h`'.
- '`objc/message.h`': this header declares public functions used to construct, deconstruct and forward message invocations. Because messaging is done in quite a different way on different runtimes, functions in this header are specific to the GNU Objective-C runtime implementation.
- '`objc/objc-exception.h`': this header declares some public functions related to Objective-C exceptions. For example functions in this header allow you to throw an Objective-C exception from plain C/C++ code.
- '`objc/objc-sync.h`': this header declares some public functions related to the Objective-C `@synchronized()` syntax, allowing you to emulate an Objective-C `@synchronized()` block in plain C/C++ code.

- 'objc/thr.h': this header declares a public runtime API threading layer that is only provided by the GNU Objective-C runtime. It declares functions such as objc_mutex_lock(), which provide a platform-independent set of threading functions.

The header files contain detailed documentation for each function in the GNU Objective-C runtime API.

8.1.2 Traditional GNU Objective-C Runtime API

The GNU Objective-C runtime used to provide a different API, which we call the "traditional" GNU Objective-C runtime API. Functions belonging to this API are easy to recognize because they use a different naming convention, such as class_get_super_class() (traditional API) instead of class_getSuperclass() (modern API). Software using this API includes the file 'objc/objc-api.h' where it is declared.

Starting with GCC 4.7.0, the traditional GNU runtime API is no longer available.

8.2 +load: Executing Code before main

This section is specific for the GNU Objective-C runtime. If you are using a different runtime, you can skip it.

The GNU Objective-C runtime provides a way that allows you to execute code before the execution of the program enters the main function. The code is executed on a per-class and a per-category basis, through a special class method +load.

This facility is very useful if you want to initialize global variables which can be accessed by the program directly, without sending a message to the class first. The usual way to initialize global variables, in the +initialize method, might not be useful because +initialize is only called when the first message is sent to a class object, which in some cases could be too late.

Suppose for example you have a FileStream class that declares Stdin, Stdout and Stderr as global variables, like below:

```
FileStream *Stdin = nil;
FileStream *Stdout = nil;
FileStream *Stderr = nil;

@implementation FileStream

+ (void)initialize
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

In this example, the initialization of Stdin, Stdout and Stderr in +initialize occurs too late. The programmer can send a message to one of these objects before the variables are actually initialized, thus sending messages to the nil object. The +initialize method which actually initializes the global variables is not invoked until the first message is sent

to the class object. The solution would require these variables to be initialized just before entering `main`.

The correct solution of the above problem is to use the `+load` method instead of `+initialize`:

```
@implementation FileStream

+ (void)load
{
    Stdin = [[FileStream new] initWithFd:0];
    Stdout = [[FileStream new] initWithFd:1];
    Stderr = [[FileStream new] initWithFd:2];
}

/* Other methods here */
@end
```

The `+load` is a method that is not overridden by categories. If a class and a category of it both implement `+load`, both methods are invoked. This allows some additional initializations to be performed in a category.

This mechanism is not intended to be a replacement for `+initialize`. You should be aware of its limitations when you decide to use it instead of `+initialize`.

8.2.1 What You Can and Cannot Do in `+load`

`+load` is to be used only as a last resort. Because it is executed very early, most of the Objective-C runtime machinery will not be ready when `+load` is executed; hence `+load` works best for executing C code that is independent on the Objective-C runtime.

The `+load` implementation in the GNU runtime guarantees you the following things:

- you can write whatever C code you like;
- you can allocate and send messages to objects whose class is implemented in the same file;
- the `+load` implementation of all super classes of a class are executed before the `+load` of that class is executed;
- the `+load` implementation of a class is executed before the `+load` implementation of any category.

In particular, the following things, even if they can work in a particular case, are not guaranteed:

- allocation of or sending messages to arbitrary objects;
- allocation of or sending messages to objects whose classes have a category implemented in the same file;
- sending messages to Objective-C constant strings (`@"this is a constant string"`);

You should make no assumptions about receiving `+load` in sibling classes when you write `+load` of a class. The order in which sibling classes receive `+load` is not guaranteed.

The order in which `+load` and `+initialize` are called could be problematic if this matters. If you don't allocate objects inside `+load`, it is guaranteed that `+load` is called before `+initialize`. If you create an object inside `+load` the `+initialize` method of object's

class is invoked even if `+load` was not invoked. Note if you explicitly call `+load` on a class, `+initialize` will be called first. To avoid possible problems try to implement only one of these methods.

The `+load` method is also invoked when a bundle is dynamically loaded into your running program. This happens automatically without any intervening operation from you. When you write bundles and you need to write `+load` you can safely create and send messages to objects whose classes already exist in the running program. The same restrictions as above apply to classes defined in bundle.

8.3 Type Encoding

This is an advanced section. Type encodings are used extensively by the compiler and by the runtime, but you generally do not need to know about them to use Objective-C.

The Objective-C compiler generates type encodings for all the types. These type encodings are used at runtime to find out information about selectors and methods and about objects and classes.

The types are encoded in the following way:

`_Bool`	`B`
`char`	`c`
`unsigned char`	`C`
`short`	`s`
`unsigned short`	`S`
`int`	`i`
`unsigned int`	`I`
`long`	`l`
`unsigned long`	`L`
`long long`	`q`
`unsigned long long`	`Q`
`float`	`f`
`double`	`d`
`long double`	`D`
`void`	`v`
`id`	`@`
`Class`	`#`
`SEL`	`:`
`char*`	`*`
`enum`	an `enum` is encoded exactly as the integer type that the compiler uses for it, which depends on the enumeration values. Often the compiler users `unsigned int`, which is then encoded as `I`.
unknown type	`?`
Complex types	`j` followed by the inner type. For example `_Complex double` is encoded as `"jd"`.
bit-fields	`b` followed by the starting position of the bit-field, the type of the bit-field and the size of the bit-field (the bit-fields encoding was changed from the NeXT's compiler encoding, see below)

The encoding of bit-fields has changed to allow bit-fields to be properly handled by the runtime functions that compute sizes and alignments of types that contain bit-fields. The previous encoding contained only the size of the bit-field. Using only this information it is not possible to reliably compute the size occupied by the bit-field. This is very important in the presence of the Boehm's garbage collector because the objects are allocated using the typed memory facility available in this collector. The typed memory allocation requires information about where the pointers are located inside the object.

The position in the bit-field is the position, counting in bits, of the bit closest to the beginning of the structure.

The non-atomic types are encoded as follows:

pointers	'`^`' followed by the pointed type.
arrays	'`[`' followed by the number of elements in the array followed by the type of the elements followed by '`]`'
structures	'`{`' followed by the name of the structure (or '`?`' if the structure is unnamed), the '`=`' sign, the type of the members and by '`}`'
unions	'`(`' followed by the name of the structure (or '`?`' if the union is unnamed), the '`=`' sign, the type of the members followed by '`)`'
vectors	'`![`' followed by the vector_size (the number of bytes composing the vector) followed by a comma, followed by the alignment (in bytes) of the vector, followed by the type of the elements followed by '`]`'

Here are some types and their encodings, as they are generated by the compiler on an i386 machine:

Objective-C type	Compiler encoding
`int a[10];`	`[10i]`
`struct { int i; float f[3]; int a:3; int b:2; char c; }`	`{?=i[3f]b128i3b131i2c}`
`int a __attribute__ ((vector_size (16)));`	`![16,16i]` (alignment depends on the machine)

In addition to the types the compiler also encodes the type specifiers. The table below describes the encoding of the current Objective-C type specifiers:

Specifier	Encoding
`const`	`r`
`in`	`n`
`inout`	`N`
`out`	`o`
`bycopy`	`O`

`byref`	`R`
`oneway`	`V`

The type specifiers are encoded just before the type. Unlike types however, the type specifiers are only encoded when they appear in method argument types.

Note how `const` interacts with pointers:

Objective-C type	Compiler encoding
`const int`	`ri`
`const int*`	`^ri`
`int *const`	`r^i`

`const int*` is a pointer to a `const int`, and so is encoded as `^ri`. `int* const`, instead, is a `const` pointer to an `int`, and so is encoded as `r^i`.

Finally, there is a complication when encoding `const char *` versus `char * const`. Because `char *` is encoded as `*` and not as `^c`, there is no way to express the fact that `r` applies to the pointer or to the pointee.

Hence, it is assumed as a convention that `r*` means `const char *` (since it is what is most often meant), and there is no way to encode `char *const`. `char *const` would simply be encoded as `*`, and the `const` is lost.

8.3.1 Legacy Type Encoding

Unfortunately, historically GCC used to have a number of bugs in its encoding code. The NeXT runtime expects GCC to emit type encodings in this historical format (compatible with GCC-3.3), so when using the NeXT runtime, GCC will introduce on purpose a number of incorrect encodings:

- the read-only qualifier of the pointee gets emitted before the '^'. The read-only qualifier of the pointer itself gets ignored, unless it is a typedef. Also, the 'r' is only emitted for the outermost type.
- 32-bit longs are encoded as 'l' or 'L', but not always. For typedefs, the compiler uses 'i' or 'I' instead if encoding a struct field or a pointer.
- `enum`s are always encoded as 'i' (int) even if they are actually unsigned or long.

In addition to that, the NeXT runtime uses a different encoding for bitfields. It encodes them as `b` followed by the size, without a bit offset or the underlying field type.

8.3.2 `@encode`

GNU Objective-C supports the `@encode` syntax that allows you to create a type encoding from a C/Objective-C type. For example, `@encode(int)` is compiled by the compiler into `"i"`.

`@encode` does not support type qualifiers other than `const`. For example, `@encode(const char*)` is valid and is compiled into `"r*"`, while `@encode(bycopy char *)` is invalid and will cause a compilation error.

8.3.3 Method Signatures

This section documents the encoding of method types, which is rarely needed to use Objective-C. You should skip it at a first reading; the runtime provides functions that will work on methods and can walk through the list of parameters and interpret them for you. These functions are part of the public "API" and are the preferred way to interact with method signatures from user code.

But if you need to debug a problem with method signatures and need to know how they are implemented (i.e., the "ABI"), read on.

Methods have their "signature" encoded and made available to the runtime. The "signature" encodes all the information required to dynamically build invocations of the method at runtime: return type and arguments.

The "signature" is a null-terminated string, composed of the following:

- The return type, including type qualifiers. For example, a method returning `int` would have `i` here.
- The total size (in bytes) required to pass all the parameters. This includes the two hidden parameters (the object `self` and the method selector `_cmd`).
- Each argument, with the type encoding, followed by the offset (in bytes) of the argument in the list of parameters.

For example, a method with no arguments and returning `int` would have the signature `i8@0:4` if the size of a pointer is 4. The signature is interpreted as follows: the `i` is the return type (an `int`), the `8` is the total size of the parameters in bytes (two pointers each of size 4), the `@0` is the first parameter (an object at byte offset `0`) and `:4` is the second parameter (a `SEL` at byte offset `4`).

You can easily find more examples by running the "strings" program on an Objective-C object file compiled by GCC. You'll see a lot of strings that look very much like `i8@0:4`. They are signatures of Objective-C methods.

8.4 Garbage Collection

This section is specific for the GNU Objective-C runtime. If you are using a different runtime, you can skip it.

Support for garbage collection with the GNU runtime has been added by using a powerful conservative garbage collector, known as the Boehm-Demers-Weiser conservative garbage collector.

To enable the support for it you have to configure the compiler using an additional argument, '`--enable-objc-gc`'. This will build the boehm-gc library, and build an additional runtime library which has several enhancements to support the garbage collector. The new library has a new name, '`libobjc_gc.a`' to not conflict with the non-garbage-collected library.

When the garbage collector is used, the objects are allocated using the so-called typed memory allocation mechanism available in the Boehm-Demers-Weiser collector. This mode requires precise information on where pointers are located inside objects. This information is computed once per class, immediately after the class has been initialized.

There is a new runtime function `class_ivar_set_gcinvisible()` which can be used to declare a so-called *weak pointer* reference. Such a pointer is basically hidden for the garbage collector; this can be useful in certain situations, especially when you want to keep track of the allocated objects, yet allow them to be collected. This kind of pointers can only be members of objects, you cannot declare a global pointer as a weak reference. Every type which is a pointer type can be declared a weak pointer, including `id`, `Class` and `SEL`.

Here is an example of how to use this feature. Suppose you want to implement a class whose instances hold a weak pointer reference; the following class does this:

```
@interface WeakPointer : Object
{
    const void* weakPointer;
}

- initWithPointer:(const void*)p;
- (const void*)weakPointer;
@end


@implementation WeakPointer

+ (void)initialize
{
  if (self == objc_lookUpClass ("WeakPointer"))
    class_ivar_set_gcinvisible (self, "weakPointer", YES);
}

- initWithPointer:(const void*)p
{
  weakPointer = p;
  return self;
}

- (const void*)weakPointer
{
  return weakPointer;
}

@end
```

Weak pointers are supported through a new type character specifier represented by the '`!`' character. The `class_ivar_set_gcinvisible()` function adds or removes this specifier to the string type description of the instance variable named as argument.

8.5 Constant String Objects

GNU Objective-C provides constant string objects that are generated directly by the compiler. You declare a constant string object by prefixing a C constant string with the character '`@`':

```
id myString = @"this is a constant string object";
```

The constant string objects are by default instances of the `NXConstantString` class which is provided by the GNU Objective-C runtime. To get the definition of this class you must include the '`objc/NXConstStr.h`' header file.

User defined libraries may want to implement their own constant string class. To be able to support them, the GNU Objective-C compiler provides a new command line options '`-fconstant-string-class=`*class-name*'. The provided class should adhere to a strict structure, the same as `NXConstantString`'s structure:

```
@interface MyConstantStringClass
{
  Class isa;
  char *c_string;
  unsigned int len;
}
@end
```

`NXConstantString` inherits from `Object`; user class libraries may choose to inherit the customized constant string class from a different class than `Object`. There is no requirement in the methods the constant string class has to implement, but the final ivar layout of the class must be the compatible with the given structure.

When the compiler creates the statically allocated constant string object, the `c_string` field will be filled by the compiler with the string; the `length` field will be filled by the compiler with the string length; the `isa` pointer will be filled with `NULL` by the compiler, and it will later be fixed up automatically at runtime by the GNU Objective-C runtime library to point to the class which was set by the '`-fconstant-string-class`' option when the object file is loaded (if you wonder how it works behind the scenes, the name of the class to use, and the list of static objects to fixup, are stored by the compiler in the object file in a place where the GNU runtime library will find them at runtime).

As a result, when a file is compiled with the '`-fconstant-string-class`' option, all the constant string objects will be instances of the class specified as argument to this option. It is possible to have multiple compilation units referring to different constant string classes, neither the compiler nor the linker impose any restrictions in doing this.

8.6 compatibility_alias

The keyword `@compatibility_alias` allows you to define a class name as equivalent to another class name. For example:

```
@compatibility_alias WOApplication GSWApplication;
```

tells the compiler that each time it encounters `WOApplication` as a class name, it should replace it with `GSWApplication` (that is, `WOApplication` is just an alias for `GSWApplication`).

There are some constraints on how this can be used—

- `WOApplication` (the alias) must not be an existing class;
- `GSWApplication` (the real class) must be an existing class.

8.7 Exceptions

GNU Objective-C provides exception support built into the language, as in the following example:

```
@try {
  ...
```

```
    @throw expr;
  ...
}
@catch (AnObjCClass *exc) {
  ...
    @throw expr;
  ...
    @throw;
  ...
}
@catch (AnotherClass *exc) {
  ...
}
@catch (id allOthers) {
  ...
}
@finally {
  ...
    @throw expr;
  ...
}
```

The `@throw` statement may appear anywhere in an Objective-C or Objective-C++ program; when used inside of a `@catch` block, the `@throw` may appear without an argument (as shown above), in which case the object caught by the `@catch` will be rethrown.

Note that only (pointers to) Objective-C objects may be thrown and caught using this scheme. When an object is thrown, it will be caught by the nearest `@catch` clause capable of handling objects of that type, analogously to how `catch` blocks work in C++ and Java. A `@catch(id ...)` clause (as shown above) may also be provided to catch any and all Objective-C exceptions not caught by previous `@catch` clauses (if any).

The `@finally` clause, if present, will be executed upon exit from the immediately preceding `@try ... @catch` section. This will happen regardless of whether any exceptions are thrown, caught or rethrown inside the `@try ... @catch` section, analogously to the behavior of the `finally` clause in Java.

There are several caveats to using the new exception mechanism:

- The '`-fobjc-exceptions`' command line option must be used when compiling Objective-C files that use exceptions.
- With the GNU runtime, exceptions are always implemented as "native" exceptions and it is recommended that the '`-fexceptions`' and '`-shared-libgcc`' options are used when linking.
- With the NeXT runtime, although currently designed to be binary compatible with `NS_HANDLER`-style idioms provided by the `NSException` class, the new exceptions can only be used on Mac OS X 10.3 (Panther) and later systems, due to additional functionality needed in the NeXT Objective-C runtime.
- As mentioned above, the new exceptions do not support handling types other than Objective-C objects. Furthermore, when used from Objective-C++, the Objective-C exception model does not interoperate with C++ exceptions at this time. This means you cannot `@throw` an exception from Objective-C and `catch` it in C++, or vice versa (i.e., `throw ... @catch`).

8.8 Synchronization

GNU Objective-C provides support for synchronized blocks:

```
@synchronized (ObjCClass *guard) {
  ...
}
```

Upon entering the `@synchronized` block, a thread of execution shall first check whether a lock has been placed on the corresponding `guard` object by another thread. If it has, the current thread shall wait until the other thread relinquishes its lock. Once `guard` becomes available, the current thread will place its own lock on it, execute the code contained in the `@synchronized` block, and finally relinquish the lock (thereby making `guard` available to other threads).

Unlike Java, Objective-C does not allow for entire methods to be marked `@synchronized`. Note that throwing exceptions out of `@synchronized` blocks is allowed, and will cause the guarding object to be unlocked properly.

Because of the interactions between synchronization and exception handling, you can only use `@synchronized` when compiling with exceptions enabled, that is with the command line option '`-fobjc-exceptions`'.

8.9 Fast Enumeration

8.9.1 Using Fast Enumeration

GNU Objective-C provides support for the fast enumeration syntax:

```
id array = ...;
id object;

for (object in array)
{
  /* Do something with 'object' */
}
```

`array` needs to be an Objective-C object (usually a collection object, for example an array, a dictionary or a set) which implements the "Fast Enumeration Protocol" (see below). If you are using a Foundation library such as GNUstep Base or Apple Cocoa Foundation, all collection objects in the library implement this protocol and can be used in this way.

The code above would iterate over all objects in `array`. For each of them, it assigns it to `object`, then executes the `Do something with 'object'` statements.

Here is a fully worked-out example using a Foundation library (which provides the implementation of `NSArray`, `NSString` and `NSLog`):

```
NSArray *array = [NSArray arrayWithObjects: @"1", @"2", @"3", nil];
NSString *object;

for (object in array)
  NSLog (@"Iterating over %@", object);
```

8.9.2 C99-Like Fast Enumeration Syntax

A c99-like declaration syntax is also allowed:

```
id array = ...;
```

```
for (id object in array)
{
  /* Do something with 'object'  */
}
```

this is completely equivalent to:

```
id array = ...;

{
  id object;
  for (object in array)
  {
    /* Do something with 'object'  */
  }
}
```

but can save some typing.

Note that the option '`-std=c99`' is not required to allow this syntax in Objective-C.

8.9.3 Fast Enumeration Details

Here is a more technical description with the gory details. Consider the code

```
for (object expression in collection expression)
{
  statements
}
```

here is what happens when you run it:

- *`collection expression`* is evaluated exactly once and the result is used as the collection object to iterate over. This means it is safe to write code such as `for (object in [NSDictionary keyEnumerator]) ...`.
- the iteration is implemented by the compiler by repeatedly getting batches of objects from the collection object using the fast enumeration protocol (see below), then iterating over all objects in the batch. This is faster than a normal enumeration where objects are retrieved one by one (hence the name "fast enumeration").
- if there are no objects in the collection, then *`object expression`* is set to `nil` and the loop immediately terminates.
- if there are objects in the collection, then for each object in the collection (in the order they are returned) *`object expression`* is set to the object, then *`statements`* are executed.
- *`statements`* can contain `break` and `continue` commands, which will abort the iteration or skip to the next loop iteration as expected.
- when the iteration ends because there are no more objects to iterate over, *`object expression`* is set to `nil`. This allows you to determine whether the iteration finished because a `break` command was used (in which case *`object expression`* will remain set to the last object that was iterated over) or because it iterated over all the objects (in which case *`object expression`* will be set to `nil`).
- *`statements`* must not make any changes to the collection object; if they do, it is a hard error and the fast enumeration terminates by invoking `objc_enumerationMutation`, a runtime function that normally aborts the program but which can be customized by Foundation libraries via `objc_set_mutation_handler` to do something different, such as raising an exception.

8.9.4 Fast Enumeration Protocol

If you want your own collection object to be usable with fast enumeration, you need to have it implement the method

```
- (unsigned long) countByEnumeratingWithState: (NSFastEnumerationState *)state
                                       objects: (id *)objects
                                         count: (unsigned long)len;
```

where `NSFastEnumerationState` must be defined in your code as follows:

```
typedef struct
{
  unsigned long state;
  id            *itemsPtr;
  unsigned long *mutationsPtr;
  unsigned long extra[5];
} NSFastEnumerationState;
```

If no `NSFastEnumerationState` is defined in your code, the compiler will automatically replace `NSFastEnumerationState *` with `struct __objcFastEnumerationState *`, where that type is silently defined by the compiler in an identical way. This can be confusing and we recommend that you define `NSFastEnumerationState` (as shown above) instead.

The method is called repeatedly during a fast enumeration to retrieve batches of objects. Each invocation of the method should retrieve the next batch of objects.

The return value of the method is the number of objects in the current batch; this should not exceed `len`, which is the maximum size of a batch as requested by the caller. The batch itself is returned in the `itemsPtr` field of the `NSFastEnumerationState` struct.

To help with returning the objects, the `objects` array is a C array preallocated by the caller (on the stack) of size `len`. In many cases you can put the objects you want to return in that `objects` array, then do `itemsPtr = objects`. But you don't have to; if your collection already has the objects to return in some form of C array, it could return them from there instead.

The `state` and `extra` fields of the `NSFastEnumerationState` structure allows your collection object to keep track of the state of the enumeration. In a simple array implementation, `state` may keep track of the index of the last object that was returned, and `extra` may be unused.

The `mutationsPtr` field of the `NSFastEnumerationState` is used to keep track of mutations. It should point to a number; before working on each object, the fast enumeration loop will check that this number has not changed. If it has, a mutation has happened and the fast enumeration will abort. So, `mutationsPtr` could be set to point to some sort of version number of your collection, which is increased by one every time there is a change (for example when an object is added or removed). Or, if you are content with less strict mutation checks, it could point to the number of objects in your collection or some other value that can be checked to perform an approximate check that the collection has not been mutated.

Finally, note how we declared the `len` argument and the return value to be of type `unsigned long`. They could also be declared to be of type `unsigned int` and everything would still work.

8.10 Messaging with the GNU Objective-C Runtime

This section is specific for the GNU Objective-C runtime. If you are using a different runtime, you can skip it.

The implementation of messaging in the GNU Objective-C runtime is designed to be portable, and so is based on standard C.

Sending a message in the GNU Objective-C runtime is composed of two separate steps. First, there is a call to the lookup function, `objc_msg_lookup ()` (or, in the case of messages to super, `objc_msg_lookup_super ()`). This runtime function takes as argument the receiver and the selector of the method to be called; it returns the `IMP`, that is a pointer to the function implementing the method. The second step of method invocation consists of casting this pointer function to the appropriate function pointer type, and calling the function pointed to it with the right arguments.

For example, when the compiler encounters a method invocation such as `[object init]`, it compiles it into a call to `objc_msg_lookup (object, @selector(init))` followed by a cast of the returned value to the appropriate function pointer type, and then it calls it.

8.10.1 Dynamically Registering Methods

If `objc_msg_lookup()` does not find a suitable method implementation, because the receiver does not implement the required method, it tries to see if the class can dynamically register the method.

To do so, the runtime checks if the class of the receiver implements the method

```
+ (BOOL) resolveInstanceMethod: (SEL)selector;
```

in the case of an instance method, or

```
+ (BOOL) resolveClassMethod: (SEL)selector;
```

in the case of a class method. If the class implements it, the runtime invokes it, passing as argument the selector of the original method, and if it returns `YES`, the runtime tries the lookup again, which could now succeed if a matching method was added dynamically by `+resolveInstanceMethod:` or `+resolveClassMethod:`.

This allows classes to dynamically register methods (by adding them to the class using `class_addMethod`) when they are first called. To do so, a class should implement `+resolveInstanceMethod:` (or, depending on the case, `+resolveClassMethod:`) and have it recognize the selectors of methods that can be registered dynamically at runtime, register them, and return `YES`. It should return `NO` for methods that it does not dynamically registered at runtime.

If `+resolveInstanceMethod:` (or `+resolveClassMethod:`) is not implemented or returns `NO`, the runtime then tries the forwarding hook.

Support for `+resolveInstanceMethod:` and `resolveClassMethod:` was added to the GNU Objective-C runtime in GCC version 4.6.

8.10.2 Forwarding Hook

The GNU Objective-C runtime provides a hook, called `__objc_msg_forward2`, which is called by `objc_msg_lookup()` when it cannot find a method implementation in the runtime tables and after calling `+resolveInstanceMethod:` and `+resolveClassMethod:` has been attempted and did not succeed in dynamically registering the method.

To configure the hook, you set the global variable `__objc_msg_forward2` to a function with the same argument and return types of `objc_msg_lookup()`. When `objc_msg_lookup()` cannot find a method implementation, it invokes the hook function you provided to get a method implementation to return. So, in practice `__objc_msg_forward2` allows you to extend `objc_msg_lookup()` by adding some custom code that is called to do a further lookup when no standard method implementation can be found using the normal lookup.

This hook is generally reserved for "Foundation" libraries such as GNUstep Base, which use it to implement their high-level method forwarding API, typically based around the `forwardInvocation:` method. So, unless you are implementing your own "Foundation" library, you should not set this hook.

In a typical forwarding implementation, the `__objc_msg_forward2` hook function determines the argument and return type of the method that is being looked up, and then creates a function that takes these arguments and has that return type, and returns it to the caller. Creating this function is non-trivial and is typically performed using a dedicated library such as `libffi`.

The forwarding method implementation thus created is returned by `objc_msg_lookup()` and is executed as if it was a normal method implementation. When the forwarding method implementation is called, it is usually expected to pack all arguments into some sort of object (typically, an `NSInvocation` in a "Foundation" library), and hand it over to the programmer (`forwardInvocation:`) who is then allowed to manipulate the method invocation using a high-level API provided by the "Foundation" library. For example, the programmer may want to examine the method invocation arguments and name and potentially change them before forwarding the method invocation to one or more local objects (`performInvocation:`) or even to remote objects (by using Distributed Objects or some other mechanism). When all this completes, the return value is passed back and must be returned correctly to the original caller.

Note that the GNU Objective-C runtime currently provides no support for method forwarding or method invocations other than the `__objc_msg_forward2` hook.

If the forwarding hook does not exist or returns `NULL`, the runtime currently attempts forwarding using an older, deprecated API, and if that fails, it aborts the program. In future versions of the GNU Objective-C runtime, the runtime will immediately abort.

9 Binary Compatibility

Binary compatibility encompasses several related concepts:

application binary interface (ABI)
: The set of runtime conventions followed by all of the tools that deal with binary representations of a program, including compilers, assemblers, linkers, and language runtime support. Some ABIs are formal with a written specification, possibly designed by multiple interested parties. Others are simply the way things are actually done by a particular set of tools.

ABI conformance
: A compiler conforms to an ABI if it generates code that follows all of the specifications enumerated by that ABI. A library conforms to an ABI if it is implemented according to that ABI. An application conforms to an ABI if it is built using tools that conform to that ABI and does not contain source code that specifically changes behavior specified by the ABI.

calling conventions
: Calling conventions are a subset of an ABI that specify of how arguments are passed and function results are returned.

interoperability
: Different sets of tools are interoperable if they generate files that can be used in the same program. The set of tools includes compilers, assemblers, linkers, libraries, header files, startup files, and debuggers. Binaries produced by different sets of tools are not interoperable unless they implement the same ABI. This applies to different versions of the same tools as well as tools from different vendors.

intercallability
: Whether a function in a binary built by one set of tools can call a function in a binary built by a different set of tools is a subset of interoperability.

implementation-defined features
: Language standards include lists of implementation-defined features whose behavior can vary from one implementation to another. Some of these features are normally covered by a platform's ABI and others are not. The features that are not covered by an ABI generally affect how a program behaves, but not intercallability.

compatibility
: Conformance to the same ABI and the same behavior of implementation-defined features are both relevant for compatibility.

The application binary interface implemented by a C or C++ compiler affects code generation and runtime support for:

- size and alignment of data types
- layout of structured types
- calling conventions

- register usage conventions
- interfaces for runtime arithmetic support
- object file formats

In addition, the application binary interface implemented by a C++ compiler affects code generation and runtime support for:

- name mangling
- exception handling
- invoking constructors and destructors
- layout, alignment, and padding of classes
- layout and alignment of virtual tables

Some GCC compilation options cause the compiler to generate code that does not conform to the platform's default ABI. Other options cause different program behavior for implementation-defined features that are not covered by an ABI. These options are provided for consistency with other compilers that do not follow the platform's default ABI or the usual behavior of implementation-defined features for the platform. Be very careful about using such options.

Most platforms have a well-defined ABI that covers C code, but ABIs that cover C++ functionality are not yet common.

Starting with GCC 3.2, GCC binary conventions for C++ are based on a written, vendor-neutral C++ ABI that was designed to be specific to 64-bit Itanium but also includes generic specifications that apply to any platform. This C++ ABI is also implemented by other compiler vendors on some platforms, notably GNU/Linux and BSD systems. We have tried hard to provide a stable ABI that will be compatible with future GCC releases, but it is possible that we will encounter problems that make this difficult. Such problems could include different interpretations of the C++ ABI by different vendors, bugs in the ABI, or bugs in the implementation of the ABI in different compilers. GCC's '`-Wabi`' switch warns when G++ generates code that is probably not compatible with the C++ ABI.

The C++ library used with a C++ compiler includes the Standard C++ Library, with functionality defined in the C++ Standard, plus language runtime support. The runtime support is included in a C++ ABI, but there is no formal ABI for the Standard C++ Library. Two implementations of that library are interoperable if one follows the de-facto ABI of the other and if they are both built with the same compiler, or with compilers that conform to the same ABI for C++ compiler and runtime support.

When G++ and another C++ compiler conform to the same C++ ABI, but the implementations of the Standard C++ Library that they normally use do not follow the same ABI for the Standard C++ Library, object files built with those compilers can be used in the same program only if they use the same C++ library. This requires specifying the location of the C++ library header files when invoking the compiler whose usual library is not being used. The location of GCC's C++ header files depends on how the GCC build was configured, but can be seen by using the G++ '`-v`' option. With default configuration options for G++ 3.3 the compile line for a different C++ compiler needs to include

```
-Igcc_install_directory/include/c++/3.3
```

Similarly, compiling code with G++ that must use a C++ library other than the GNU C++ library requires specifying the location of the header files for that other library.

The most straightforward way to link a program to use a particular C++ library is to use a C++ driver that specifies that C++ library by default. The `g++` driver, for example, tells the linker where to find GCC's C++ library ('`libstdc++`') plus the other libraries and startup files it needs, in the proper order.

If a program must use a different C++ library and it's not possible to do the final link using a C++ driver that uses that library by default, it is necessary to tell `g++` the location and name of that library. It might also be necessary to specify different startup files and other runtime support libraries, and to suppress the use of GCC's support libraries with one or more of the options '`-nostdlib`', '`-nostartfiles`', and '`-nodefaultlibs`'.

10 gcov—a Test Coverage Program

gcov is a tool you can use in conjunction with GCC to test code coverage in your programs.

10.1 Introduction to gcov

gcov is a test coverage program. Use it in concert with GCC to analyze your programs to help create more efficient, faster running code and to discover untested parts of your program. You can use gcov as a profiling tool to help discover where your optimization efforts will best affect your code. You can also use gcov along with the other profiling tool, gprof, to assess which parts of your code use the greatest amount of computing time.

Profiling tools help you analyze your code's performance. Using a profiler such as gcov or gprof, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. gcov helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use gcov because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for 'hot spots' where the code is using a great deal of computer time. Likewise, because gcov accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful—they only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

gcov creates a logfile called '*sourcefile*.gcov' which indicates how many times each line of a source file '*sourcefile*.c' has executed. You can use these logfiles along with gprof to aid in fine-tuning the performance of your programs. gprof gives timing information you can use along with the information you get from gcov.

gcov works only on code compiled with GCC. It is not compatible with any other profiling or test coverage mechanism.

10.2 Invoking gcov

gcov [*options*] *files*

gcov accepts the following options:

`-a`
`--all-blocks`
: Write individual execution counts for every basic block. Normally gcov outputs execution counts only for the main blocks of a line. With this option you can determine if blocks within a single line are not being executed.

`-b`
`--branch-probabilities`
: Write branch frequencies to the output file, and write branch summary info to the standard output. This option allows you to see how often each branch in your program was taken. Unconditional branches will not be shown, unless the '`-u`' option is given.

`-c`
`--branch-counts`
: Write branch frequencies as the number of branches taken, rather than the percentage of branches taken.

`-d`
`--display-progress`
: Display the progress on the standard output.

`-f`
`--function-summaries`
: Output summaries for each function in addition to the file level summary.

`-h`
`--help`
: Display help about using `gcov` (on the standard output), and exit without doing any further processing.

`-i`
`--json-format`
: Output gcov file in an easy-to-parse JSON intermediate format which does not require source code for generation. The JSON file is compressed with gzip compression algorithm and the files have '`.gcov.json.gz`' extension.

  Structure of the JSON is following:

```
{
  "current_working_directory": current_working_directory,
  "data_file": data_file,
  "format_version": format_version,
  "gcc_version": gcc_version
  "files": [file]
}
```

  Fields of the root element have following semantics:

  - *current_working_directory*: working directory where a compilation unit was compiled
  - *data_file*: name of the data file (GCDA)
  - *format_version*: semantic version of the format
  - *gcc_version*: version of the GCC compiler

  Each *file* has the following form:

```
{
  "file": file_name,
  "functions": [function],
  "lines": [line]
}
```

Fields of the *file* element have following semantics:

- *file_name*: name of the source file

Each *function* has the following form:

```
{
  "blocks": blocks,
  "blocks_executed": blocks_executed,
  "demangled_name": "demangled_name,
  "end_column": end_column,
  "end_line": end_line,
  "execution_count": execution_count,
  "name": name,
  "start_column": start_column
  "start_line": start_line
}
```

Fields of the *function* element have following semantics:

- *blocks*: number of blocks that are in the function
- *blocks_executed*: number of executed blocks of the function
- *demangled_name*: demangled name of the function
- *end_column*: column in the source file where the function ends
- *end_line*: line in the source file where the function ends
- *execution_count*: number of executions of the function
- *name*: name of the function
- *start_column*: column in the source file where the function begins
- *start_line*: line in the source file where the function begins

Note that line numbers and column numbers number from 1. In the current implementation, *start_line* and *start_column* do not include any template parameters and the leading return type but that this is likely to be fixed in the future.

Each *line* has the following form:

```
{
  "branches": [branch],
  "count": count,
  "line_number": line_number,
  "unexecuted_block": unexecuted_block
  "function_name": function_name,
}
```

Branches are present only with *-b* option. Fields of the *line* element have following semantics:

- *count*: number of executions of the line
- *line_number*: line number
- *unexecuted_block*: flag whether the line contains an unexecuted block (not all statements on the line are executed)

- *function_name*: a name of a function this *line* belongs to (for a line with an inlined statements can be not set)

Each *branch* has the following form:

```
{
  "count": count,
  "fallthrough": fallthrough,
  "throw": throw
}
```

Fields of the *branch* element have following semantics:

- *count*: number of executions of the branch
- *fallthrough*: true when the branch is a fall through branch
- *throw*: true when the branch is an exceptional branch

`-j`
`--human-readable`
: Write counts in human readable format (like 24.6k).

`-k`
`--use-colors`
: Use colors for lines of code that have zero coverage. We use red color for non-exceptional lines and cyan for exceptional. Same colors are used for basic blocks with '`-a`' option.

`-l`
`--long-file-names`
: Create long file names for included source files. For example, if the header file '`x.h`' contains code, and was included in the file '`a.c`', then running `gcov` on the file '`a.c`' will produce an output file called '`a.c##x.h.gcov`' instead of '`x.h.gcov`'. This can be useful if '`x.h`' is included in multiple source files and you want to see the individual contributions. If you use the '`-p`' option, both the including and included file names will be complete path names.

`-m`
`--demangled-names`
: Display demangled function names in output. The default is to show mangled function names.

`-n`
`--no-output`
: Do not create the `gcov` output file.

`-o` *`directory|file`*
`--object-directory` *`directory`*
`--object-file` *`file`*
: Specify either the directory containing the gcov data files, or the object path name. The '`.gcno`', and '`.gcda`' data files are searched for using this option. If a directory is specified, the data files are in that directory and named after the input file name, without its extension. If a file is specified here, the data files are named after that file, without its extension.

`-p`
`--preserve-paths`
: Preserve complete path information in the names of generated '`.gcov`' files. Without this option, just the filename component is used. With this option, all directories are used, with '`/`' characters translated to '`#`' characters, '`.`' directory components removed and unremoveable '`..`' components renamed to '`^`'. This is useful if sourcefiles are in several different directories.

`-q`
`--use-hotness-colors`
: Emit perf-like colored output for hot lines. Legend of the color scale is printed at the very beginning of the output file.

`-r`
`--relative-only`
: Only output information about source files with a relative pathname (after source prefix elision). Absolute paths are usually system header files and coverage of any inline functions therein is normally uninteresting.

`-s` *`directory`*
`--source-prefix` *`directory`*
: A prefix for source file names to remove when generating the output coverage files. This option is useful when building in a separate directory, and the pathname to the source directory is not wanted when determining the output file names. Note that this prefix detection is applied before determining whether the source file is absolute.

`-t`
`--stdout` Output to standard output instead of output files.

`-u`
`--unconditional-branches`
: When branch probabilities are given, include those of unconditional branches. Unconditional branches are normally not interesting.

`-v`
`--version`
: Display the `gcov` version number (on the standard output), and exit without doing any further processing.

`-w`
`--verbose`
: Print verbose informations related to basic blocks and arcs.

`-x`
`--hash-filenames`
: When using *–preserve-paths*, gcov uses the full pathname of the source files to create an output filename. This can lead to long filenames that can overflow filesystem limits. This option creates names of the form '*`source-file##md5`*`.gcov`', where the *source-file* component is the final filename part and the *md5* component is calculated from the full mangled

name that would have been used otherwise. The option is an alternative to the *–preserve-paths* on systems which have a filesystem limit.

`gcov` should be run with the current directory the same as that when you invoked the compiler. Otherwise it will not be able to locate the source files. `gcov` produces files called '*mangledname*`.gcov`' in the current directory. These contain the coverage information of the source file they correspond to. One '`.gcov`' file is produced for each source (or header) file containing code, which was compiled to produce the data files. The *mangledname* part of the output file name is usually simply the source file name, but can be something more complicated if the '`-l`' or '`-p`' options are given. Refer to those options for details.

If you invoke `gcov` with multiple input files, the contributions from each input file are summed. Typically you would invoke it with the same list of files as the final link of your executable.

The '`.gcov`' files contain the '`:`' separated fields along with program source code. The format is

```
execution_count:line_number:source line text
```

Additional block information may succeed each line, when requested by command line option. The *execution_count* is '`-`' for lines containing no code. Unexecuted lines are marked '`#####`' or '`=====`', depending on whether they are reachable by non-exceptional paths or only exceptional paths such as C++ exception handlers, respectively. Given the '`-a`' option, unexecuted blocks are marked '`$$$$$`' or '`%%%%%`', depending on whether a basic block is reachable via non-exceptional or exceptional paths. Executed basic blocks having a statement with zero *execution_count* end with '`*`' character and are colored with magenta color with the '`-k`' option. This functionality is not supported in Ada.

Note that GCC can completely remove the bodies of functions that are not needed – for instance if they are inlined everywhere. Such functions are marked with '`-`', which can be confusing. Use the '`-fkeep-inline-functions`' and '`-fkeep-static-functions`' options to retain these functions and allow gcov to properly show their *execution_count*.

Some lines of information at the start have *line_number* of zero. These preamble lines are of the form

```
-:0:tag:value
```

The ordering and number of these preamble lines will be augmented as `gcov` development progresses — do not rely on them remaining unchanged. Use *tag* to locate a particular preamble line.

The additional block information is of the form

```
tag information
```

The *information* is human readable, but designed to be simple enough for machine parsing too.

When printing percentages, 0% and 100% are only printed when the values are *exactly* 0% and 100% respectively. Other values which would conventionally be rounded to 0% or 100% are instead printed as the nearest non-boundary value.

When using `gcov`, you must first compile your program with a special GCC option '`--coverage`'. This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files

for generating the extra profiling information needed by gcov. These additional files are placed in the directory where the object file is located.

Running the program will cause profile output to be generated. For each source file compiled with '-fprofile-arcs', an accompanying '.gcda' file will be placed in the object file directory.

Running gcov with your program's source file names as arguments will now produce a listing of the code along with frequency of execution for each line. For example, if your program is called 'tmp.cpp', this is what you see when you use the basic gcov facility:

```
$ g++ --coverage tmp.cpp
$ a.out
$ gcov tmp.cpp -m
File 'tmp.cpp'
Lines executed:92.86% of 14
Creating 'tmp.cpp.gcov'
```

The file 'tmp.cpp.gcov' contains output from gcov. Here is a sample:

```
            -:    0:Source:tmp.cpp
            -:    0:Working directory:/home/gcc/testcase
            -:    0:Graph:tmp.gcno
            -:    0:Data:tmp.gcda
            -:    0:Runs:1
            -:    0:Programs:1
            -:    1:#include <stdio.h>
            -:    2:
            -:    3:template<class T>
            -:    4:class Foo
            -:    5:{
            -:    6:  public:
           1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
            1:    7:  Foo(): b (1000) {}
------------------
           2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
            2:    8:  void inc () { b++; }
------------------
            -:    9:
            -:   10:  private:
            -:   11:  int b;
            -:   12:};
            -:   13:
            -:   14:template class Foo<int>;
            -:   15:template class Foo<char>;
            -:   16:
            -:   17:int
            1:   18:main (void)
            -:   19:{
            -:   20:  int i, total;
```

```
        1:   21:  Foo<int> counter;
        -:   22:
        1:   23:  counter.inc();
        1:   24:  counter.inc();
        1:   25:  total = 0;
        -:   26:
       11:   27:  for (i = 0; i < 10; i++)
       10:   28:    total += i;
        -:   29:
       1*:   30:  int v = total > 100 ? 1 : 2;
        -:   31:
        1:   32:  if (total != 45)
    #####:   33:    printf ("Failure\n");
        -:   34:  else
        1:   35:    printf ("Success\n");
        1:   36:  return 0;
        -:   37:}
```

Note that line 7 is shown in the report multiple times. First occurrence presents total number of execution of the line and the next two belong to instances of class Foo constructors. As you can also see, line 30 contains some unexecuted basic blocks and thus execution count has asterisk symbol.

When you use the '`-a`' option, you will get individual block counts, and the output looks like this:

```
        -:    0:Source:tmp.cpp
        -:    0:Working directory:/home/gcc/testcase
        -:    0:Graph:tmp.gcno
        -:    0:Data:tmp.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:
        -:    3:template<class T>
        -:    4:class Foo
        -:    5:{
        -:    6:  public:
       1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
        1:    7:  Foo(): b (1000) {}
------------------
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
        2:    8:  void inc () { b++; }
------------------
        -:    9:
        -:   10:  private:
        -:   11:  int b;
        -:   12:};
        -:   13:
```

```
    -:   14:template class Foo<int>;
    -:   15:template class Foo<char>;
    -:   16:
    -:   17:int
    1:   18:main (void)
    -:   19:{
    -:   20:  int i, total;
    1:   21:  Foo<int> counter;
    1:   21-block  0
    -:   22:
    1:   23:  counter.inc();
    1:   23-block  0
    1:   24:  counter.inc();
    1:   24-block  0
    1:   25:  total = 0;
    -:   26:
   11:   27:  for (i = 0; i < 10; i++)
    1:   27-block  0
   11:   27-block  1
   10:   28:    total += i;
   10:   28-block  0
    -:   29:
   1*:   30:  int v = total > 100 ? 1 : 2;
    1:   30-block  0
%%%%%:   30-block  1
    1:   30-block  2
    -:   31:
    1:   32:  if (total != 45)
    1:   32-block  0
#####:   33:    printf ("Failure\n");
%%%%%:   33-block  0
    -:   34:  else
    1:   35:    printf ("Success\n");
    1:   35-block  0
    1:   36:  return 0;
    1:   36-block  0
    -:   37:}
```

In this mode, each basic block is only shown on one line – the last line of the block. A multi-line block will only contribute to the execution count of that last line, and other lines will not be shown to contain code, unless previous blocks end on those lines. The total execution count of a line is shown and subsequent lines show the execution counts for individual blocks that end on that line. After each block, the branch and call counts of the block will be shown, if the '`-b`' option is given.

Because of the way GCC instruments calls, a call count can be shown after a line with no individual blocks. As you can see, line 33 contains a basic block that was not executed.

When you use the '`-b`' option, your output looks like this:

```
    -:    0:Source:tmp.cpp
    -:    0:Working directory:/home/gcc/testcase
    -:    0:Graph:tmp.gcno
    -:    0:Data:tmp.gcda
    -:    0:Runs:1
    -:    0:Programs:1
    -:    1:#include <stdio.h>
    -:    2:
    -:    3:template<class T>
```

```
        -:    4:class Foo
        -:    5:{
        -:    6:  public:
       1*:    7:  Foo(): b (1000) {}
------------------
Foo<char>::Foo():
function Foo<char>::Foo() called 0 returned 0% blocks executed 0%
    #####:    7:  Foo(): b (1000) {}
------------------
Foo<int>::Foo():
function Foo<int>::Foo() called 1 returned 100% blocks executed 100%
        1:    7:  Foo(): b (1000) {}
------------------
       2*:    8:  void inc () { b++; }
------------------
Foo<char>::inc():
function Foo<char>::inc() called 0 returned 0% blocks executed 0%
    #####:    8:  void inc () { b++; }
------------------
Foo<int>::inc():
function Foo<int>::inc() called 2 returned 100% blocks executed 100%
        2:    8:  void inc () { b++; }
------------------
        -:    9:
        -:   10:  private:
        -:   11:  int b;
        -:   12:};
        -:   13:
        -:   14:template class Foo<int>;
        -:   15:template class Foo<char>;
        -:   16:
        -:   17:int
function main called 1 returned 100% blocks executed 81%
        1:   18:main (void)
        -:   19:{
        -:   20:  int i, total;
        1:   21:  Foo<int> counter;
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        -:   22:
        1:   23:  counter.inc();
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   24:  counter.inc();
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   25:  total = 0;
        -:   26:
       11:   27:  for (i = 0; i < 10; i++)
branch  0 taken 91% (fallthrough)
branch  1 taken 9%
       10:   28:    total += i;
        -:   29:
       1*:   30:  int v = total > 100 ? 1 : 2;
branch  0 taken 0% (fallthrough)
```

```
branch  1 taken 100%
        -:   31:
        1:   32:  if (total != 45)
branch  0 taken 0% (fallthrough)
branch  1 taken 100%
    #####:   33:    printf ("Failure\n");
call    0 never executed
branch  1 never executed
branch  2 never executed
        -:   34:  else
        1:   35:    printf ("Success\n");
call    0 returned 100%
branch  1 taken 100% (fallthrough)
branch  2 taken 0% (throw)
        1:   36:  return 0;
        -:   37:}
```

For each function, a line is printed showing how many times the function is called, how many times it returns and what percentage of the function's blocks were executed.

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions that call `exit` or `longjmp`, and thus may not return every time they are called.

The execution counts are cumulative. If the example program were executed again without removing the '`.gcda`' file, the count for the number of times each line in the source was executed would be added to the results of the previous run(s). This is potentially useful in several ways. For example, it could be used to accumulate data over a number of program runs as part of a test verification suite, or to provide more accurate long-term information over a large number of program runs.

The data in the '`.gcda`' files is saved immediately before the program exits. For each source file compiled with '`-fprofile-arcs`', the profiling code first attempts to read in an existing '`.gcda`' file; if the file doesn't match the executable (differing number of basic block counts) it will ignore the contents of the file. It then adds in the new execution counts and finally writes the data to the file.

10.3 Using gcov with GCC Optimization

If you plan to use `gcov` to help optimize your code, you must first compile your program with a special GCC option '`--coverage`'. Aside from that, you can use any other GCC options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer

can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for `gcov` to calculate separate execution counts for each line because there isn't separate code for each line. Hence the `gcov` output looks like this if you compiled the program with optimization:

```
100:   12:if (a != b)
100:   13:  c = 1;
100:   14:else
100:   15:  c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

Inlineable functions can create unexpected line counts. Line counts are shown for the source code of the inlineable function, but what is shown depends on where the function is inlined, or if it is not inlined at all.

If the function is not inlined, the compiler must emit an out of line copy of the function, in any object file that needs it. If '`fileA.o`' and '`fileB.o`' both contain out of line bodies of a particular inlineable function, they will also both contain coverage counts for that function. When '`fileA.o`' and '`fileB.o`' are linked together, the linker will, on many systems, select one of those out of line bodies for all calls to that function, and remove or ignore the other. Unfortunately, it will not remove the coverage counters for the unused function body. Hence when instrumented, all but one use of that function will show zero counts.

If the function is inlined in several places, the block structure in each location might not be the same. For instance, a condition might now be calculable at compile time in some instances. Because the coverage of all the uses of the inline function will be shown for the same source lines, the line counts themselves might seem inconsistent.

Long-running applications can use the `__gcov_reset` and `__gcov_dump` facilities to restrict profile collection to the program region of interest. Calling `__gcov_reset(void)` will clear all profile counters to zero, and calling `__gcov_dump(void)` will cause the profile information collected at that point to be dumped to '`.gcda`' output files. Instrumented applications use a static destructor with priority 99 to invoke the `__gcov_dump` function. Thus `__gcov_dump` is executed after all user defined static destructors, as well as handlers registered with `atexit`. If an executable loads a dynamic shared object via dlopen functionality, '`-Wl,--dynamic-list-data`' is needed to dump all profile data.

Profiling run-time library reports various errors related to profile manipulation and profile saving. Errors are printed into standard error output or '`GCOV_ERROR_FILE`' file, if environment variable is used. In order to terminate immediately after an errors occurs set '`GCOV_EXIT_AT_ERROR`' environment variable. That can help users to find profile clashing which leads to a misleading profile.

10.4 Brief Description of gcov Data Files

gcov uses two files for profiling. The names of these files are derived from the original *object* file by substituting the file suffix with either '.gcno', or '.gcda'. The files contain coverage and profile data stored in a platform-independent format. The '.gcno' files are placed in the same directory as the object file. By default, the '.gcda' files are also stored in the same directory as the object file, but the GCC '-fprofile-dir' option may be used to store the '.gcda' files in a separate directory.

The '.gcno' notes file is generated when the source file is compiled with the GCC '-ftest-coverage' option. It contains information to reconstruct the basic block graphs and assign source line numbers to blocks.

The '.gcda' count data file is generated when a program containing object files built with the GCC '-fprofile-arcs' option is executed. A separate '.gcda' file is created for each object file compiled with this option. It contains arc transition counts, value profile counts, and some summary information.

It is not recommended to access the coverage files directly. Consumers should use the intermediate format that is provided by gcov tool via '--json-format' option.

10.5 Data File Relocation to Support Cross-Profiling

Running the program will cause profile output to be generated. For each source file compiled with '-fprofile-arcs', an accompanying '.gcda' file will be placed in the object file directory. That implicitly requires running the program on the same system as it was built or having the same absolute directory structure on the target system. The program will try to create the needed directory structure, if it is not already present.

To support cross-profiling, a program compiled with '-fprofile-arcs' can relocate the data files based on two environment variables:

- GCOV_PREFIX contains the prefix to add to the absolute paths in the object file. Prefix can be absolute, or relative. The default is no prefix.
- GCOV_PREFIX_STRIP indicates the how many initial directory names to strip off the hardwired absolute paths. Default value is 0.

  Note: If GCOV_PREFIX_STRIP is set without GCOV_PREFIX is undefined, then a relative path is made out of the hardwired absolute paths.

For example, if the object file '/user/build/foo.o' was built with '-fprofile-arcs', the final executable will try to create the data file '/user/build/foo.gcda' when running on the target system. This will fail if the corresponding directory does not exist and it is unable to create it. This can be overcome by, for example, setting the environment as 'GCOV_PREFIX=/target/run' and 'GCOV_PREFIX_STRIP=1'. Such a setting will name the data file '/target/run/build/foo.gcda'.

You must move the data files to the expected directory tree in order to use them for profile directed optimizations ('-fprofile-use'), or to use the gcov tool.

11 gcov-tool—an Offline Gcda Profile Processing Tool

gcov-tool is a tool you can use in conjunction with GCC to manipulate or process gcda profile files offline.

11.1 Introduction to gcov-tool

gcov-tool is an offline tool to process gcc's gcda profile files.

Current gcov-tool supports the following functionalities:

- merge two sets of profiles with weights.
- read one set of profile and rewrite profile contents. One can scale or normalize the count values.

Examples of the use cases for this tool are:

- Collect the profiles for different set of inputs, and use this tool to merge them. One can specify the weight to factor in the relative importance of each input.
- Rewrite the profile after removing a subset of the gcda files, while maintaining the consistency of the summary and the histogram.
- It can also be used to debug or libgcov code as the tools shares the majority code as the runtime library.

Note that for the merging operation, this profile generated offline may contain slight different values from the online merged profile. Here are a list of typical differences:

- histogram difference: This offline tool recomputes the histogram after merging the counters. The resulting histogram, therefore, is precise. The online merging does not have this capability – the histogram is merged from two histograms and the result is an approximation.
- summary checksum difference: Summary checksum uses a CRC32 operation. The value depends on the link list order of gcov-info objects. This order is different in gcov-tool from that in the online merge. It's expected to have different summary checksums. It does not really matter as the compiler does not use this checksum anywhere.
- value profile counter values difference: Some counter values for value profile are runtime dependent, like heap addresses. It's normal to see some difference in these kind of counters.

11.2 Invoking gcov-tool

gcov-tool [*global-options*] SUB_COMMAND [*sub_command-options*] *profile_dir*

gcov-tool accepts the following options:

-h
--help
: Display help about using gcov-tool (on the standard output), and exit without doing any further processing.

-v
--version
: Display the gcov-tool version number (on the standard output), and exit without doing any further processing.

`merge` Merge two profile directories.

`-o` *`directory`*
`--output` *`directory`*
: Set the output profile directory. Default output directory name is *merged_profile*.

`-v`
`--verbose`
: Set the verbose mode.

`-w` *`w1,w2`*
`--weight` *`w1,w2`*
: Set the merge weights of the *directory1* and *directory2*, respectively. The default weights are 1 for both.

`rewrite` Read the specified profile directory and rewrite to a new directory.

`-n` *`long_long_value`*
`--normalize <long_long_value>`
: Normalize the profile. The specified value is the max counter value in the new profile.

`-o` *`directory`*
`--output` *`directory`*
: Set the output profile directory. Default output name is *rewrite_profile*.

`-s` *`float_or_simple-frac_value`*
`--scale` *`float_or_simple-frac_value`*
: Scale the profile counters. The specified value can be in floating point value, or simple fraction value form, such 1, 2, 2/3, and 5/3.

`-v`
`--verbose`
: Set the verbose mode.

`overlap` Compute the overlap score between the two specified profile directories. The overlap score is computed based on the arc profiles. It is defined as the sum of min (p1_counter[i] / p1_sum_all, p2_counter[i] / p2_sum_all), for all arc counter i, where p1_counter[i] and p2_counter[i] are two matched counters and p1_sum_all and p2_sum_all are the sum of counter values in profile 1 and profile 2, respectively.

`-f`
`--function`
: Print function level overlap score.

`-F`
`--fullname`
: Print full gcda filename.

`-h`
`--hotonly`
: Only print info for hot objects/functions.

`-o`
`--object` Print object level overlap score.

`-t` *`float`*
`--hot_threshold <float>`
Set the threshold for hot counter value.

`-v`
`--verbose`
Set the verbose mode.

12 gcov-dump—an Offline Gcda and Gcno Profile Dump Tool

12.1 Introduction to gcov-dump

gcov-dump is a tool you can use in conjunction with GCC to dump content of gcda and gcno profile files offline.

12.2 Invoking gcov-dump

Usage: gcov-dump [*OPTION*] ... *gcovfiles*

gcov-dump accepts the following options:

-h
--help
: Display help about using gcov-dump (on the standard output), and exit without doing any further processing.

-l
--long
: Dump content of records.

-p
--positions
: Dump positions of records.

-v
--version
: Display the gcov-dump version number (on the standard output), and exit without doing any further processing.

13 Known Causes of Trouble with GCC

This section describes known problems that affect users of GCC. Most of these are not GCC bugs per se—if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

13.1 Actual Bugs We Haven't Fixed Yet

- The `fixincludes` script interacts badly with automounters; if the directory of system header files is automounted, it tends to be unmounted while `fixincludes` is running. This would seem to be a bug in the automounter. We don't know any good way to work around it.

13.2 Interoperation

This section lists various difficulties encountered in using GCC together with other compilers or with the assemblers, linkers, libraries and debuggers on certain systems.

- On many platforms, GCC supports a different ABI for C++ than do other compilers, so the object files compiled by GCC cannot be used with object files generated by another C++ compiler.

  An area where the difference is most apparent is name mangling. The use of different name mangling is intentional, to protect you from more subtle problems. Compilers differ as to many internal details of C++ implementation, including: how class instances are laid out, how multiple inheritance is implemented, and how virtual function calls are handled. If the name encoding were made the same, your programs would link against libraries provided from other compilers—but the programs would then crash when run. Incompatible libraries are then detected at link time, rather than at run time.

- On some BSD systems, including some versions of Ultrix, use of profiling causes static variable destructors (currently used only in C++) not to be run.

- On a SPARC, GCC aligns all values of type `double` on an 8-byte boundary, and it expects every `double` to be so aligned. The Sun compiler usually gives `double` values 8-byte alignment, with one exception: function arguments of type `double` may not be aligned.

  As a result, if a function compiled with Sun CC takes the address of an argument of type `double` and passes this pointer of type `double *` to a function compiled with GCC, dereferencing the pointer may cause a fatal signal.

  One way to solve this problem is to compile your entire program with GCC. Another solution is to modify the function that is compiled with Sun CC to copy the argument into a local variable; local variables are always properly aligned. A third solution is to modify the function that uses the pointer to dereference it via the following function `access_double` instead of directly with '`*`':

```
inline double
access_double (double *unaligned_ptr)
{
  union d2i { double d; int i[2]; };

  union d2i *p = (union d2i *) unaligned_ptr;
  union d2i u;

  u.i[0] = p->i[0];
  u.i[1] = p->i[1];

  return u.d;
}
```

Storing into the pointer can be done likewise with the same union.

- On Solaris, the `malloc` function in the '`libmalloc.a`' library may allocate memory that is only 4 byte aligned. Since GCC on the SPARC assumes that doubles are 8 byte aligned, this may result in a fatal signal if doubles are stored in memory allocated by the '`libmalloc.a`' library.

  The solution is to not use the '`libmalloc.a`' library. Use instead `malloc` and related functions from '`libc.a`'; they do not have this problem.
- On the HP PA machine, ADB sometimes fails to work on functions compiled with GCC. Specifically, it fails to work on functions that use `alloca` or variable-size arrays. This is because GCC doesn't generate HP-UX unwind descriptors for such functions. It may even be impossible to generate them.
- Debugging ('`-g`') is not supported on the HP PA machine, unless you use the preliminary GNU tools.
- Taking the address of a label may generate errors from the HP-UX PA assembler. GAS for the PA does not have this problem.
- Using floating point parameters for indirect calls to static functions will not work when using the HP assembler. There simply is no way for GCC to specify what registers hold arguments for static functions when using the HP assembler. GAS for the PA does not have this problem.
- In extremely rare cases involving some very large functions you may receive errors from the HP linker complaining about an out of bounds unconditional branch offset. This used to occur more often in previous versions of GCC, but is now exceptionally rare. If you should run into it, you can work around by making your function smaller.
- GCC compiled code sometimes emits warnings from the HP-UX assembler of the form:

  ```
  (warning) Use of GR3 when
    frame >= 8192 may cause conflict.
  ```

  These warnings are harmless and can be safely ignored.
- In extremely rare cases involving some very large functions you may receive errors from the AIX Assembler complaining about a displacement that is too large. If you should run into it, you can work around by making your function smaller.
- The '`libstdc++.a`' library in GCC relies on the SVR4 dynamic linker semantics which merges global symbols between libraries and applications, especially necessary for C++ streams functionality. This is not the default behavior of AIX shared libraries and dynamic linking. '`libstdc++.a`' is built on AIX with "runtime-linking" enabled so

that symbol merging can occur. To utilize this feature, the application linked with '`libstdc++.a`' must include the '`-Wl,-brtl`' flag on the link line. G++ cannot impose this because this option may interfere with the semantics of the user program and users may not always use '`g++`' to link his or her application. Applications are not required to use the '`-Wl,-brtl`' flag on the link line—the rest of the '`libstdc++.a`' library which is not dependent on the symbol merging semantics will continue to function correctly.

- An application can interpose its own definition of functions for functions invoked by '`libstdc++.a`' with "runtime-linking" enabled on AIX. To accomplish this the application must be linked with "runtime-linking" option and the functions explicitly must be exported by the application ('`-Wl,-brtl,-bE:exportfile`').
- AIX on the RS/6000 provides support (NLS) for environments outside of the United States. Compilers and assemblers use NLS to support locale-specific representations of various objects including floating-point numbers ('`.`' vs '`,`' for separating decimal fractions). There have been problems reported where the library linked with GCC does not produce the same floating-point formats that the assembler accepts. If you have this problem, set the `LANG` environment variable to '`C`' or '`En_US`'.
- Even if you specify '`-fdollars-in-identifiers`', you cannot successfully use '`$`' in identifiers on the RS/6000 due to a restriction in the IBM assembler. GAS supports these identifiers.

13.3 Incompatibilities of GCC

There are several noteworthy incompatibilities between GNU C and K&R (non-ISO) versions of C.

- GCC normally makes string constants read-only. If several identical-looking string constants are used, GCC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some very old systems when passed a string constant as its format control string or input. This is because `sscanf` incorrectly tries to write into the string constant. Likewise `fscanf` and `scanf`.

  The solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants.
- `-2147483648` is positive.

  This is because 2147483648 cannot fit in the type `int`, so (following the ISO C rules) its data type is `unsigned long int`. Negating this value yields 2147483648 again.
- GCC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GCC

  ```
  #define foo(a) "a"
  ```

  will produce output `"a"` regardless of what the argument *a* is.
- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;
  ```

```
foo ()
{
  int a, b;

  a = fun1 ();
  if (setjmp (j))
    return a;

  a = fun2 ();
  /* longjmp (j) may occur in fun3. */
  return a + fun3 ();
}
```

Here a may or may not be restored to its first value when the longjmp occurs. If a is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

If you use the '-W' option with the '-O' option, you will get a warning when GCC thinks such a problem might be possible.

- Programs that use preprocessing directives in the middle of macro arguments do not work with GCC. For example, a program like this will not work:

```
foobar (
#define luser
        hack)
```

  ISO C does not permit such a construct.

- K&R compilers allow comments to cross over an inclusion boundary (i.e. started in an include file and ended in the including file).
- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, an extern declaration affects all the rest of the file even if it happens within a block.

- In traditional C, you can combine long, etc., with a typedef name, as shown here:

```
typedef int foo;
typedef long foo bar;
```

  In ISO C, this is not allowed: long and other type modifiers require an explicit int.

- PCC allows typedef names to be used as function parameters.
- Traditional C allows the following erroneous pair of declarations to appear together in a given scope:

```
typedef int foo;
typedef foo foo;
```

- GCC treats all characters of identifiers as significant. According to K&R-1 (2.2), "No more than the first eight characters are significant, although more may be used.". Also according to K&R-1 (2.2), "An identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter.", but GCC also allows dollar signs in identifiers.
- PCC allows whitespace in the middle of compound assignment operators such as '+='. GCC, following the ISO standard, does not allow this.

- GCC complains about unterminated character constants inside of preprocessing conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GCC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '`/*...*/`'.

- Many user programs contain the declaration '`long time ();`'. In the past, the system header files on many systems did not actually declare `time`, so it did not matter what type your program declared it to return. But in systems with ISO C headers, `time` is declared to return `time_t`, and if that is not the same as `long`, then '`long time ();`' is erroneous.

  The solution is to change your program to use appropriate system headers (`<time.h>` on systems with ISO C headers) and not to declare `time` if the system header files declare it, or failing that to use `time_t` as the return type of `time`.

- When compiling functions that return `float`, PCC converts it to a double. GCC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.

- When compiling functions that return structures or unions, GCC output code normally uses a method different from that used on most versions of Unix. As a result, code compiled with GCC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GCC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller (usually in a special, fixed register, but on some machines it is passed on the stack). The target hook `TARGET_STRUCT_VALUE_RTX` tells GCC where to pass this address.

  By contrast, PCC on most target machines returns structures and unions of any size by copying the data into an area of static storage, and then returning the address of that storage as if it were a pointer value. The caller must copy the data from that memory area to the place where the value is wanted. GCC does not use this method because it is slower and nonreentrant.

  On some newer machines, PCC uses a reentrant convention for all structure and union returning. GCC on most of these machines uses a compatible convention when returning structures and unions in memory, but still returns small structures and unions in registers.

  You can tell GCC to use a compatible convention for all structure and union returning with the option '`-fpcc-struct-return`'.

- GCC complains about program fragments such as '`0x74ae-0x4000`' which appear to be two hexadecimal constants separated by the minus operator. Actually, this string is a single *preprocessing token*. Each such token must correspond to one token in C. Since this does not, GCC prints an error message. Although it may appear obvious that

what is meant is an operator and two values, the ISO C standard specifically requires that this be treated as erroneous.

A *preprocessing token* is a *preprocessing number* if it begins with a digit and is followed by letters, underscores, digits, periods and '`e+`', '`e-`', '`E+`', '`E-`', '`p+`', '`p-`', '`P+`', or '`P-`' character sequences. (In strict C90 mode, the sequences '`p+`', '`p-`', '`P+`' and '`P-`' cannot appear in preprocessing numbers.)

To make the above program fragment valid, place whitespace in front of the minus sign. This whitespace will end the preprocessing number.

13.4 Fixed Header Files

GCC needs to install corrected versions of some system header files. This is because most target systems have some header files that won't work with GCC unless they are changed. Some have bugs, some are incompatible with ISO C, and some depend on special features of other compilers.

Installing GCC automatically creates and installs the fixed header files, by running a program called `fixincludes`. Normally, you don't need to pay attention to this. But there are cases where it doesn't do the right thing automatically.

- If you update the system's header files, such as by installing a new system version, the fixed header files of GCC are not automatically updated. They can be updated using the `mkheaders` script installed in '`libexecdir/gcc/target/version/install-tools/`'.
- On some systems, header file directories contain machine-specific symbolic links in certain places. This makes it possible to share most of the header files among hosts running the same version of the system on different machine models.

  The programs that fix the header files do not understand this special way of using symbolic links; therefore, the directory of fixed header files is good only for the machine model used to build it.

  It is possible to make separate sets of fixed header files for the different machine models, and arrange a structure of symbolic links so as to use the proper set, but you'll have to do this by hand.

13.5 Standard Libraries

GCC by itself attempts to be a conforming freestanding implementation. See Chapter 2 [Language Standards Supported by GCC], page 5, for details of what this means. Beyond the library facilities required of such an implementation, the rest of the C library is supplied by the vendor of the operating system. If that C library doesn't conform to the C standards, then your programs might get warnings (especially when using '`-Wall`') that you don't expect.

For example, the `sprintf` function on SunOS 4.1.3 returns `char *` while the C standard says that `sprintf` returns an `int`. The `fixincludes` program could make the prototype for this function match the Standard, but that would be wrong, since the function will still return `char *`.

If you need a Standard compliant library, then you need to find one, as GCC does not provide one. The GNU C library (called `glibc`) provides ISO C, POSIX, BSD, SystemV and

X/Open compatibility for GNU/Linux and HURD-based GNU systems; no recent version of it supports other systems, though some very old versions did. Version 2.2 of the GNU C library includes nearly complete C99 support. You could also ask your operating system vendor if newer libraries are available.

13.6 Disappointments and Misunderstandings

These problems are perhaps regrettable, but we don't know any practical way around them.

- Certain local variables aren't recognized by debuggers when you compile with optimization.

  This occurs because sometimes GCC optimizes the variable out of existence. There is no way to tell the debugger how to compute the value such a variable "would have had", and it is not clear that would be desirable anyway. So GCC simply does not mention the eliminated variable when it writes debugging information.

  You have to expect a certain amount of disagreement between the executable and your source code, when you use optimization.

- Users often think it is a bug when GCC reports an error for code like this:

  ```
  int foo (struct mumble *);

  struct mumble { ... };

  int foo (struct mumble *x)
  { ... }
  ```

  This code really is erroneous, because the scope of `struct mumble` in the prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

  This behavior may seem silly, but it's what the ISO standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype. It's not worth being incompatible with ISO C just to avoid an error for the example shown above.

- Accesses to bit-fields even in volatile objects works by accessing larger objects, such as a byte or a word. You cannot rely on what size of object is accessed in order to read or write the bit-field; it may even vary for a given bit-field according to the precise usage.

  If you care about controlling the amount of memory that is accessed, use volatile but do not use bit-fields.

- GCC comes with shell scripts to fix certain known problems in system header files. They install corrected copies of various header files in a special directory where only GCC will normally look for them. The scripts adapt to various systems by searching all the system header files for the problem cases that we know about.

  If new system header files are installed, nothing automatically arranges to update the corrected header files. They can be updated using the `mkheaders` script installed in '*libexecdir*/gcc/*target*/*version*/install-tools/'.

- On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. For example, you can find that a floating point value which is not a NaN is not equal to itself. This results from the fact that the floating point registers hold a few more bits of precision than fit in a `double` in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

  You can partially avoid this problem by using the '`-ffloat-store`' option (see Section 3.10 [Optimize Options], page 125).

- On AIX and other platforms without weak symbol support, templates need to be instantiated explicitly and symbols for static members of templates will not be generated.

- On AIX, GCC scans object files and library archives for static constructors and destructors when linking an application before the linker prunes unreferenced symbols. This is necessary to prevent the AIX linker from mistakenly assuming that static constructor or destructor are unused and removing them before the scanning can occur. All static constructors and destructors found will be referenced even though the modules in which they occur may not be used by the program. This may lead to both increased executable size and unexpected symbol references.

13.7 Common Misunderstandings with GNU C++

C++ is a complex language and an evolving one, and its standard definition (the ISO C++ standard) was only recently completed. As a result, your C++ compiler may occasionally surprise you, even when its behavior is correct. This section discusses some areas that frequently give rise to questions of this sort.

13.7.1 Declare *and* Define Static Members

When a class has static data members, it is not enough to *declare* the static member; you must also *define* it. For example:

```
class Foo
{
  ...
  void method();
  static int bar;
};
```

This declaration only establishes that the class `Foo` has an `int` named `Foo::bar`, and a member function named `Foo::method`. But you still need to define *both* `method` and `bar` elsewhere. According to the ISO standard, you must supply an initializer in one (and only one) source file, such as:

```
int Foo::bar = 0;
```

Other C++ compilers may not correctly implement the standard behavior. As a result, when you switch to `g++` from one of these compilers, you may discover that a program that appeared to work correctly in fact does not conform to the standard: `g++` reports as undefined symbols any static data members that lack definitions.

13.7.2 Name Lookup, Templates, and Accessing Members of Base Classes

The C++ standard prescribes that all names that are not dependent on template parameters are bound to their present definitions when parsing a template function or class.[1] Only names that are dependent are looked up at the point of instantiation. For example, consider

```
void foo(double);

struct A {
  template <typename T>
  void f () {
    foo (1);        // 1
    int i = N;      // 2
    T t;
    t.bar();        // 3
    foo (t);        // 4
  }

  static const int N;
};
```

Here, the names `foo` and `N` appear in a context that does not depend on the type of `T`. The compiler will thus require that they are defined in the context of use in the template, not only before the point of instantiation, and will here use `::foo(double)` and `A::N`, respectively. In particular, it will convert the integer value to a `double` when passing it to `::foo(double)`.

Conversely, `bar` and the call to `foo` in the fourth marked line are used in contexts that do depend on the type of `T`, so they are only looked up at the point of instantiation, and you can provide declarations for them after declaring the template, but before instantiating it. In particular, if you instantiate `A::f<int>`, the last line will call an overloaded `::foo(int)` if one was provided, even if after the declaration of `struct A`.

This distinction between lookup of dependent and non-dependent names is called two-stage (or dependent) name lookup. G++ implements it since version 3.4.

Two-stage name lookup sometimes leads to situations with behavior different from non-template codes. The most common is probably this:

```
template <typename T> struct Base {
  int i;
};

template <typename T> struct Derived : public Base<T> {
  int get_i() { return i; }
};
```

In `get_i()`, `i` is not used in a dependent context, so the compiler will look for a name declared at the enclosing namespace scope (which is the global scope here). It will not look into the base class, since that is dependent and you may declare specializations of `Base` even after declaring `Derived`, so the compiler cannot really know what `i` would refer to. If there is no global variable `i`, then you will get an error message.

In order to make it clear that you want the member of the base class, you need to defer lookup until instantiation time, at which the base class is known. For this, you need to

[1] The C++ standard just uses the term "dependent" for names that depend on the type or value of template parameters. This shorter term will also be used in the rest of this section.

access `i` in a dependent context, by either using `this->i` (remember that `this` is of type `Derived<T>*`, so is obviously dependent), or using `Base<T>::i`. Alternatively, `Base<T>::i` might be brought into scope by a `using`-declaration.

Another, similar example involves calling member functions of a base class:

```
template <typename T> struct Base {
    int f();
};

template <typename T> struct Derived : Base<T> {
    int g() { return f(); };
};
```

Again, the call to `f()` is not dependent on template arguments (there are no arguments that depend on the type `T`, and it is also not otherwise specified that the call should be in a dependent context). Thus a global declaration of such a function must be available, since the one in the base class is not visible until instantiation time. The compiler will consequently produce the following error message:

```
x.cc: In member function ‘int Derived<T>::g()’:
x.cc:6: error: there are no arguments to ‘f’ that depend on a template
   parameter, so a declaration of ‘f’ must be available
x.cc:6: error: (if you use ‘-fpermissive’, G++ will accept your code, but
   allowing the use of an undeclared name is deprecated)
```

To make the code valid either use `this->f()`, or `Base<T>::f()`. Using the '`-fpermissive`' flag will also let the compiler accept the code, by marking all function calls for which no declaration is visible at the time of definition of the template for later lookup at instantiation time, as if it were a dependent call. We do not recommend using '`-fpermissive`' to work around invalid code, and it will also only catch cases where functions in base classes are called, not where variables in base classes are used (as in the example above).

Note that some compilers (including G++ versions prior to 3.4) get these examples wrong and accept above code without an error. Those compilers do not implement two-stage name lookup correctly.

13.7.3 Temporaries May Vanish Before You Expect

It is dangerous to use pointers or references to *portions* of a temporary object. The compiler may very well delete the object before you expect it to, leaving a pointer to garbage. The most common place where this problem crops up is in classes like string classes, especially ones that define a conversion function to type `char *` or `const char *`—which is one reason why the standard `string` class requires you to call the `c_str` member function. However, any class that returns a pointer to some internal structure is potentially subject to this problem.

For example, a program may use a function `strfunc` that returns `string` objects, and another function `charfunc` that operates on pointers to `char`:

```
string strfunc ();
void charfunc (const char *);

void
f ()
{
  const char *p = strfunc().c_str();
```

```
  ...
  charfunc (p);
  ...
  charfunc (p);
}
```

In this situation, it may seem reasonable to save a pointer to the C string returned by the `c_str` member function and use that rather than call `c_str` repeatedly. However, the temporary string created by the call to `strfunc` is destroyed after `p` is initialized, at which point `p` is left pointing to freed memory.

Code like this may run successfully under some other compilers, particularly obsolete cfront-based compilers that delete temporaries along with normal local variables. However, the GNU C++ behavior is standard-conforming, so if your program depends on late destruction of temporaries it is not portable.

The safe way to write such code is to give the temporary a name, which forces it to remain until the end of the scope of the name. For example:

```
const string& tmp = strfunc ();
charfunc (tmp.c_str ());
```

13.7.4 Implicit Copy-Assignment for Virtual Bases

When a base class is virtual, only one subobject of the base class belongs to each full object. Also, the constructors and destructors are invoked only once, and called from the most-derived class. However, such objects behave unspecified when being assigned. For example:

```
struct Base{
  char *name;
  Base(char *n) : name(strdup(n)){}
  Base& operator= (const Base& other){
   free (name);
   name = strdup (other.name);
  }
};

struct A:virtual Base{
  int val;
  A():Base("A"){}
};

struct B:virtual Base{
  int bval;
  B():Base("B"){}
};

struct Derived:public A, public B{
  Derived():Base("Derived"){}
};

void func(Derived &d1, Derived &d2)
{
  d1 = d2;
}
```

The C++ standard specifies that '`Base::Base`' is only called once when constructing or copy-constructing a Derived object. It is unspecified whether '`Base::operator=`' is called

more than once when the implicit copy-assignment for Derived objects is invoked (as it is inside '`func`' in the example).

G++ implements the "intuitive" algorithm for copy-assignment: assign all direct bases, then assign all members. In that algorithm, the virtual base subobject can be encountered more than once. In the example, copying proceeds in the following order: '`val`', '`name`' (via `strdup`), '`bval`', and '`name`' again.

If application code relies on copy-assignment, a user-defined copy-assignment operator removes any uncertainties. With such an operator, the application can define whether and how the virtual base subobject is assigned.

13.8 Certain Changes We Don't Want to Make

This section lists changes that people frequently request, but which we do not make because we think GCC is better without them.

- Checking the number and type of arguments to a function which has an old-fashioned definition and no prototype.

  Such a feature would work only occasionally—only for calls that appear in the same file as the called function, following the definition. The only way to check all calls reliably is to add a prototype for the function. But adding a prototype eliminates the motivation for this feature. So the feature is not worthwhile.

- Warning about using an expression whose type is signed as a shift count.

  Shift count operands are probably signed more often than unsigned. Warning about this would cause far more annoyance than good.

- Warning about assigning a signed value to an unsigned variable.

  Such assignments must be very common; warning about them would cause more annoyance than good.

- Warning when a non-void function value is ignored.

  C contains many standard functions that return a value that most programs choose to ignore. One obvious example is `printf`. Warning about this practice only leads the defensive programmer to clutter programs with dozens of casts to `void`. Such casts are required so frequently that they become visual noise. Writing those casts becomes so automatic that they no longer convey useful information about the intentions of the programmer. For functions where the return value should never be ignored, use the `warn_unused_result` function attribute (see Section 6.33 [Function Attributes], page 477).

- Making '`-fshort-enums`' the default.

  This would cause storage layout to be incompatible with most other C compilers. And it doesn't seem very important, given that you can get the same result in other ways. The case where it matters most is when the enumeration-valued object is inside a structure, and in that case you can specify a field width explicitly.

- Making bit-fields unsigned by default on particular machines where "the ABI standard" says to do so.

  The ISO C standard leaves it up to the implementation whether a bit-field declared plain `int` is signed or not. This in effect creates two alternative dialects of C.

The GNU C compiler supports both dialects; you can specify the signed dialect with '`-fsigned-bitfields`' and the unsigned dialect with '`-funsigned-bitfields`'. However, this leaves open the question of which dialect to use by default.

Currently, the preferred dialect makes plain bit-fields signed, because this is simplest. Since `int` is the same as `signed int` in every other context, it is cleanest for them to be the same in bit-fields as well.

Some computer manufacturers have published Application Binary Interface standards which specify that plain bit-fields should be unsigned. It is a mistake, however, to say anything about this issue in an ABI. This is because the handling of plain bit-fields distinguishes two dialects of C. Both dialects are meaningful on every type of machine. Whether a particular object file was compiled using signed bit-fields or unsigned is of no concern to other object files, even if they access the same bit-fields in the same data structures.

A given program is written in one or the other of these two dialects. The program stands a chance to work on most any machine if it is compiled with the proper dialect. It is unlikely to work at all if compiled with the wrong dialect.

Many users appreciate the GNU C compiler because it provides an environment that is uniform across machines. These users would be inconvenienced if the compiler treated plain bit-fields differently on certain machines.

Occasionally users write programs intended only for a particular machine type. On these occasions, the users would benefit if the GNU C compiler were to support by default the same dialect as the other compilers on that machine. But such applications are rare. And users writing a program to run on more than one type of machine cannot possibly benefit from this kind of compatibility.

This is why GCC does and will treat plain bit-fields in the same fashion on all types of machines (by default).

There are some arguments for making bit-fields unsigned by default on all machines. If, for example, this becomes a universal de facto standard, it would make sense for GCC to go along with it. This is something to be considered in the future.

(Of course, users strongly concerned about portability should indicate explicitly in each bit-field whether it is signed or not. In this way, they write programs which have the same meaning in both C dialects.)

- Undefining `__STDC__` when '`-ansi`' is not used.

  Currently, GCC defines `__STDC__` unconditionally. This provides good results in practice.

  Programmers normally use conditionals on `__STDC__` to ask whether it is safe to use certain features of ISO C, such as function prototypes or ISO token concatenation. Since plain `gcc` supports all the features of ISO C, the correct answer to these questions is "yes".

  Some users try to use `__STDC__` to check for the availability of certain library facilities. This is actually incorrect usage in an ISO C program, because the ISO C standard says that a conforming freestanding implementation should define `__STDC__` even though it does not have the library facilities. '`gcc -ansi -pedantic`' is a conforming freestanding implementation, and it is therefore required to define `__STDC__`, even though it does not come with an ISO C library.

Sometimes people say that defining `__STDC__` in a compiler that does not completely conform to the ISO C standard somehow violates the standard. This is illogical. The standard is a standard for compilers that claim to support ISO C, such as '`gcc -ansi`'—not for other compilers such as plain `gcc`. Whatever the ISO C standard says is relevant to the design of plain `gcc` without '`-ansi`' only for pragmatic reasons, not as a requirement.

GCC normally defines `__STDC__` to be 1, and in addition defines `__STRICT_ANSI__` if you specify the '`-ansi`' option, or a '`-std`' option for strict conformance to some version of ISO C. On some hosts, system include files use a different convention, where `__STDC__` is normally 0, but is 1 if the user specifies strict conformance to the C Standard. GCC follows the host convention when processing system include files, but when processing user files it follows the usual GNU C convention.

- Undefining `__STDC__` in C++.

  Programs written to compile with C++-to-C translators get the value of `__STDC__` that goes with the C compiler that is subsequently used. These programs must test `__STDC__` to determine what kind of C preprocessor that compiler uses: whether they should concatenate tokens in the ISO C fashion or in the traditional fashion.

  These programs work properly with GNU C++ if `__STDC__` is defined. They would not work otherwise.

  In addition, many header files are written to provide prototypes in ISO C but not in traditional C. Many of these header files can work without change in C++ provided `__STDC__` is defined. If `__STDC__` is not defined, they will all fail, and will all need to be changed to test explicitly for C++ as well.

- Deleting "empty" loops.

  Historically, GCC has not deleted "empty" loops under the assumption that the most likely reason you would put one in a program is to have a delay, so deleting them will not make real programs run any faster.

  However, the rationale here is that optimization of a nonempty loop cannot produce an empty one. This held for carefully written C compiled with less powerful optimizers but is not always the case for carefully written C++ or with more powerful optimizers. Thus GCC will remove operations from loops whenever it can determine those operations are not externally visible (apart from the time taken to execute them, of course). In case the loop can be proved to be finite, GCC will also remove the loop itself.

  Be aware of this when performing timing tests, for instance the following loop can be completely removed, provided `some_expression` can provably not change any global state.

```
{
   int sum = 0;
   int ix;

   for (ix = 0; ix != 10000; ix++)
      sum += some_expression;
}
```

  Even though `sum` is accumulated in the loop, no use is made of that summation, so the accumulation can be removed.

- Making side effects happen in the same order as in some other compiler.

It is never safe to depend on the order of evaluation of side effects. For example, a function call like this may very well behave differently from one compiler to another:

```
void func (int, int);

int i = 2;
func (i++, i++);
```

There is no guarantee (in either the C or the C++ standard language definitions) that the increments will be evaluated in any particular order. Either increment might happen first. `func` might get the arguments '`2, 3`', or it might get '`3, 2`', or even '`2, 2`'.

- Making certain warnings into errors by default.

  Some ISO C testsuites report failure when the compiler does not produce an error message for a certain program.

  ISO C requires a "diagnostic" message for certain kinds of invalid programs, but a warning is defined by GCC to count as a diagnostic. If GCC produces a warning but not an error, that is correct ISO C support. If testsuites call this "failure", they should be run with the GCC option '`-pedantic-errors`', which will turn these warnings into errors.

13.9 Warning Messages and Error Messages

The GNU compiler can produce two kinds of diagnostics: errors and warnings. Each kind has a different purpose:

Errors report problems that make it impossible to compile your program. GCC reports errors with the source file name and line number where the problem is apparent.

Warnings report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text '`warning:`' to distinguish them from error messages.

Warnings may indicate danger points where you should check to make sure that your program really does what you intend; or the use of obsolete features; or the use of nonstandard features of GNU C or C++. Many warnings are issued only if you ask for them, with one of the '`-W`' options (for instance, '`-Wall`' requests a variety of useful warnings).

GCC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The '`-pedantic`' option tells GCC to issue warnings in such cases; '`-pedantic-errors`' says to make them errors instead. This does not mean that *all* non-ISO constructs get warnings or errors.

See Section 3.8 [Options to Request or Suppress Warnings], page 70, for more detail on these and related command-line options.

14 Reporting Bugs

Your bug reports play an essential role in making GCC reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Chapter 13 [Trouble], page 869. If it isn't known, then you should report the problem.

14.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C or C++ compiler.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results. But the value of the function is undefined if `return` is omitted; it is not a bug when GCC produces different results.

  Problems often result from expressions with two increment operators, as in `f (*p++, *p++)`. Your previous compiler might have interpreted that expression the way you intended; GCC might interpret it another way. Neither compiler is wrong. The bug is in your code.

  After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.
- If the compiler produces an error message for valid input, that is a compiler bug.
- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be someone else's idea of "an extension" or "support for traditional practice".
- If you are an experienced user of one of the languages GCC supports, your suggestions for improvement of GCC are welcome in any case.

14.2 How and Where to Report Bugs

Bugs should be reported to the bug database at `http://gcc.gnu.org/bugs/`.

15 How To Get Help with GCC

If you need help installing, using or changing GCC, there are two ways to find it:

- Send a message to a suitable network mailing list. First try `gcc-help@gcc.gnu.org` (for help installing or using GCC), and if that brings no response, try `gcc@gcc.gnu.org`. For help changing GCC, ask `gcc@gcc.gnu.org`. If you think you have found a bug in GCC, please report it following the instructions at see Section 14.2 [Bug Reporting], page 885.
- Look in the service directory for someone who might help you for a fee. The service directory is found at `https://www.fsf.org/resources/service`.

For further information, see `http://gcc.gnu.org/faq.html#support`.

16 Contributing to GCC Development

If you would like to help pretest GCC releases to assure they work well, current development sources are available by SVN (see `http://gcc.gnu.org/svn.html`). Source and binary snapshots are also available for FTP; see `http://gcc.gnu.org/snapshots.html`.

If you would like to work on improvements to GCC, please read the advice at these URLs:

```
http://gcc.gnu.org/contribute.html
http://gcc.gnu.org/contributewhy.html
```

for information on how to make useful contributions and avoid duplication of effort. Suggested projects are listed at `http://gcc.gnu.org/projects/`.

Funding Free Software

If you want to have more free software a few years from now, it makes sense for you to help encourage people to contribute funds for its development. The most effective approach known is to encourage commercial redistributors to donate.

Users of free software systems can boost the pace of development by encouraging for-a-fee distributors to donate part of their selling price to free software developers—the Free Software Foundation, and others.

The way to convince distributors to do this is to demand it and expect it from them. So when you compare distributors, judge them partly by how much they give to free software development. Show distributors they must compete to be the one who gives the most.

To make this approach work, you must insist on numbers that you can compare, such as, "We will donate ten dollars to the Frobnitz project for each disk sold." Don't be satisfied with a vague promise, such as "A portion of the profits are donated," since it doesn't give a basis for comparison.

Even a precise fraction "of the profits from this disk" is not very meaningful, since creative accounting and unrelated business decisions can greatly alter what fraction of the sales price counts as profit. If the price you pay is $50, ten percent of the profit is probably less than a dollar; it might be a few cents, or nothing at all.

Some redistributors do development work themselves. This is useful too; but to keep everyone honest, you need to inquire how much they do, and what kind. Some kinds of development make much more long-term difference than others. For example, maintaining a separate version of a program contributes very little; maintaining the standard version of a program for the whole community contributes much. Easy new ports contribute little, since someone else would surely do them; difficult ports such as adding a new CPU to the GNU Compiler Collection contribute more; major new features or packages contribute the most.

By establishing the idea that supporting further development is "the proper thing to do" when distributing free software for a fee, we can assure a steady flow of resources into making more free software.

Copyright © 1994 Free Software Foundation, Inc.
Verbatim copying and redistribution of this section is permitted
without royalty; alteration is not permitted.

The GNU Project and GNU/Linux

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. (GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced "guh-NEW".) Variants of the GNU operating system, which use the kernel Linux, are now widely used; though these systems are often referred to as "Linux", they are more accurately called GNU/Linux systems.

For more information, see:

`http://www.gnu.org/`
`http://www.gnu.org/gnu/linux-and-gnu.html`

GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Contributors to GCC

The GCC project would like to thank its many contributors. Without them the project would not have been nearly as successful as it has been. Any omissions in this list are accidental. Feel free to contact `law@redhat.com` or `gerald@pfeifer.com` if you have been left out or some of your contributions are not listed. Please keep this list in alphabetical order.

- Analog Devices helped implement the support for complex data types and iterators.
- John David Anglin for threading-related fixes and improvements to libstdc++-v3, and the HP-UX port.
- James van Artsdalen wrote the code that makes efficient use of the Intel 80387 register stack.
- Abramo and Roberto Bagnara for the SysV68 Motorola 3300 Delta Series port.
- Alasdair Baird for various bug fixes.
- Giovanni Bajo for analyzing lots of complicated C++ problem reports.
- Peter Barada for his work to improve code generation for new ColdFire cores.
- Gerald Baumgartner added the signature extension to the C++ front end.
- Godmar Back for his Java improvements and encouragement.
- Scott Bambrough for help porting the Java compiler.
- Wolfgang Bangerth for processing tons of bug reports.
- Jon Beniston for his Microsoft Windows port of Java and port to Lattice Mico32.
- Daniel Berlin for better DWARF 2 support, faster/better optimizations, improved alias analysis, plus migrating GCC to Bugzilla.
- Geoff Berry for his Java object serialization work and various patches.
- David Binderman tests weekly snapshots of GCC trunk against Fedora Rawhide for several architectures.
- Laurynas Biveinis for memory management work and DJGPP port fixes.
- Uros Bizjak for the implementation of x87 math built-in functions and for various middle end and i386 back end improvements and bug fixes.
- Eric Blake for helping to make GCJ and libgcj conform to the specifications.
- Janne Blomqvist for contributions to GNU Fortran.
- Hans-J. Boehm for his garbage collector, IA-64 libffi port, and other Java work.
- Segher Boessenkool for helping maintain the PowerPC port and the instruction combiner plus various contributions to the middle end.
- Neil Booth for work on cpplib, lang hooks, debug hooks and other miscellaneous clean-ups.
- Steven Bosscher for integrating the GNU Fortran front end into GCC and for contributing to the tree-ssa branch.
- Eric Botcazou for fixing middle- and backend bugs left and right.
- Per Bothner for his direction via the steering committee and various improvements to the infrastructure for supporting new languages. Chill front end implementation.

Initial implementations of cpplib, fix-header, config.guess, libio, and past C++ library (libg++) maintainer. Dreaming up, designing and implementing much of GCJ.

- Devon Bowen helped port GCC to the Tahoe.
- Don Bowman for mips-vxworks contributions.
- James Bowman for the FT32 port.
- Dave Brolley for work on cpplib and Chill.
- Paul Brook for work on the ARM architecture and maintaining GNU Fortran.
- Robert Brown implemented the support for Encore 32000 systems.
- Christian Bruel for improvements to local store elimination.
- Herman A.J. ten Brugge for various fixes.
- Joerg Brunsmann for Java compiler hacking and help with the GCJ FAQ.
- Joe Buck for his direction via the steering committee from its creation to 2013.
- Iain Buclaw for the D frontend.
- Craig Burley for leadership of the G77 Fortran effort.
- Tobias Burnus for contributions to GNU Fortran.
- Stephan Buys for contributing Doxygen notes for libstdc++.
- Paolo Carlini for libstdc++ work: lots of efficiency improvements to the C++ strings, streambufs and formatted I/O, hard detective work on the frustrating localization issues, and keeping up with the problem reports.
- John Carr for his alias work, SPARC hacking, infrastructure improvements, previous contributions to the steering committee, loop optimizations, etc.
- Stephane Carrez for 68HC11 and 68HC12 ports.
- Steve Chamberlain for support for the Renesas SH and H8 processors and the PicoJava processor, and for GCJ config fixes.
- Glenn Chambers for help with the GCJ FAQ.
- John-Marc Chandonia for various libgcj patches.
- Denis Chertykov for contributing and maintaining the AVR port, the first GCC port for an 8-bit architecture.
- Kito Cheng for his work on the RISC-V port, including bringing up the test suite and maintenance.
- Scott Christley for his Objective-C contributions.
- Eric Christopher for his Java porting help and clean-ups.
- Branko Cibej for more warning contributions.
- The GNU Classpath project for all of their merged runtime code.
- Nick Clifton for arm, mcore, fr30, v850, m32r, msp430 rx work, '`--help`', and other random hacking.
- Michael Cook for libstdc++ cleanup patches to reduce warnings.
- R. Kelley Cook for making GCC buildable from a read-only directory as well as other miscellaneous build process and documentation clean-ups.
- Ralf Corsepius for SH testing and minor bug fixing.

- François-Xavier Coudert for contributions to GNU Fortran.
- Stan Cox for care and feeding of the x86 port and lots of behind the scenes hacking.
- Alex Crain provided changes for the 3b1.
- Ian Dall for major improvements to the NS32k port.
- Paul Dale for his work to add uClinux platform support to the m68k backend.
- Palmer Dabbelt for his work maintaining the RISC-V port.
- Dario Dariol contributed the four varieties of sample programs that print a copy of their source.
- Russell Davidson for fstream and stringstream fixes in libstdc++.
- Bud Davis for work on the G77 and GNU Fortran compilers.
- Mo DeJong for GCJ and libgcj bug fixes.
- Jerry DeLisle for contributions to GNU Fortran.
- DJ Delorie for the DJGPP port, build and libiberty maintenance, various bug fixes, and the M32C, MeP, MSP430, and RL78 ports.
- Arnaud Desitter for helping to debug GNU Fortran.
- Gabriel Dos Reis for contributions to G++, contributions and maintenance of GCC diagnostics infrastructure, libstdc++-v3, including `valarray<>`, `complex<>`, maintaining the numerics library (including that pesky `<limits>` :-) and keeping up-to-date anything to do with numbers.
- Ulrich Drepper for his work on glibc, testing of GCC using glibc, ISO C99 support, CFG dumping support, etc., plus support of the C++ runtime libraries including for all kinds of C interface issues, contributing and maintaining `complex<>`, sanity checking and disbursement, configuration architecture, libio maintenance, and early math work.
- François Dumont for his work on libstdc++-v3, especially maintaining and improving `debug-mode` and associative and unordered containers.
- Zdenek Dvorak for a new loop unroller and various fixes.
- Michael Eager for his work on the Xilinx MicroBlaze port.
- Richard Earnshaw for his ongoing work with the ARM.
- David Edelsohn for his direction via the steering committee, ongoing work with the RS6000/PowerPC port, help cleaning up Haifa loop changes, doing the entire AIX port of libstdc++ with his bare hands, and for ensuring GCC properly keeps working on AIX.
- Kevin Ediger for the floating point formatting of num_put::do_put in libstdc++.
- Phil Edwards for libstdc++ work including configuration hackery, documentation maintainer, chief breaker of the web pages, the occasional iostream bug fix, and work on shared library symbol versioning.
- Paul Eggert for random hacking all over GCC.
- Mark Elbrecht for various DJGPP improvements, and for libstdc++ configuration support for locales and fstream-related fixes.
- Vadim Egorov for libstdc++ fixes in strings, streambufs, and iostreams.
- Christian Ehrhardt for dealing with bug reports.

- Ben Elliston for his work to move the Objective-C runtime into its own subdirectory and for his work on autoconf.
- Revital Eres for work on the PowerPC 750CL port.
- Marc Espie for OpenBSD support.
- Doug Evans for much of the global optimization framework, arc, m32r, and SPARC work.
- Christopher Faylor for his work on the Cygwin port and for caring and feeding the gcc.gnu.org box and saving its users tons of spam.
- Fred Fish for BeOS support and Ada fixes.
- Ivan Fontes Garcia for the Portuguese translation of the GCJ FAQ.
- Peter Gerwinski for various bug fixes and the Pascal front end.
- Kaveh R. Ghazi for his direction via the steering committee, amazing work to make '`-W -Wall -W* -Werror`' useful, and testing GCC on a plethora of platforms. Kaveh extends his gratitude to the CAIP Center at Rutgers University for providing him with computing resources to work on Free Software from the late 1980s to 2010.
- John Gilmore for a donation to the FSF earmarked improving GNU Java.
- Judy Goldberg for `c++` contributions.
- Torbjorn Granlund for various fixes and the c-torture testsuite, multiply- and divide-by-constant optimization, improved long long support, improved leaf function register allocation, and his direction via the steering committee.
- Jonny Grant for improvements to `collect2's` '`--help`' documentation.
- Anthony Green for his '`-Os`' contributions, the moxie port, and Java front end work.
- Stu Grossman for gdb hacking, allowing GCJ developers to debug Java code.
- Michael K. Gschwind contributed the port to the PDP-11.
- Richard Biener for his ongoing middle-end contributions and bug fixes and for release management.
- Ron Guilmette implemented the `protoize` and `unprotoize` tools, the support for DWARF 1 symbolic debugging information, and much of the support for System V Release 4. He has also worked heavily on the Intel 386 and 860 support.
- Sumanth Gundapaneni for contributing the CR16 port.
- Mostafa Hagog for Swing Modulo Scheduling (SMS) and post reload GCSE.
- Bruno Haible for improvements in the runtime overhead for EH, new warnings and assorted bug fixes.
- Andrew Haley for his amazing Java compiler and library efforts.
- Chris Hanson assisted in making GCC work on HP-UX for the 9000 series 300.
- Michael Hayes for various thankless work he's done trying to get the c30/c40 ports functional. Lots of loop and unroll improvements and fixes.
- Dara Hazeghi for wading through myriads of target-specific bug reports.
- Kate Hedstrom for staking the G77 folks with an initial testsuite.
- Richard Henderson for his ongoing SPARC, alpha, ia32, and ia64 work, loop opts, and generally fixing lots of old problems we've ignored for years, flow rewrite and lots of further stuff, including reviewing tons of patches.

- Aldy Hernandez for working on the PowerPC port, SIMD support, and various fixes.
- Nobuyuki Hikichi of Software Research Associates, Tokyo, contributed the support for the Sony NEWS machine.
- Kazu Hirata for caring and feeding the Renesas H8/300 port and various fixes.
- Katherine Holcomb for work on GNU Fortran.
- Manfred Hollstein for his ongoing work to keep the m88k alive, lots of testing and bug fixing, particularly of GCC configury code.
- Steve Holmgren for MachTen patches.
- Mat Hostetter for work on the TILE-Gx and TILEPro ports.
- Jan Hubicka for his x86 port improvements.
- Falk Hueffner for working on C and optimization bug reports.
- Bernardo Innocenti for his m68k work, including merging of ColdFire improvements and uClinux support.
- Christian Iseli for various bug fixes.
- Kamil Iskra for general m68k hacking.
- Lee Iverson for random fixes and MIPS testing.
- Balaji V. Iyer for Cilk+ development and merging.
- Andreas Jaeger for testing and benchmarking of GCC and various bug fixes.
- Martin Jambor for his work on inter-procedural optimizations, the switch conversion pass, and scalar replacement of aggregates.
- Jakub Jelinek for his SPARC work and sibling call optimizations as well as lots of bug fixes and test cases, and for improving the Java build system.
- Janis Johnson for ia64 testing and fixes, her quality improvement sidetracks, and web page maintenance.
- Kean Johnston for SCO OpenServer support and various fixes.
- Tim Josling for the sample language treelang based originally on Richard Kenner's "toy" language.
- Nicolai Josuttis for additional libstdc++ documentation.
- Klaus Kaempf for his ongoing work to make alpha-vms a viable target.
- Steven G. Kargl for work on GNU Fortran.
- David Kashtan of SRI adapted GCC to VMS.
- Ryszard Kabatek for many, many libstdc++ bug fixes and optimizations of strings, especially member functions, and for auto_ptr fixes.
- Geoffrey Keating for his ongoing work to make the PPC work for GNU/Linux and his automatic regression tester.
- Brendan Kehoe for his ongoing work with G++ and for a lot of early work in just about every part of libstdc++.
- Oliver M. Kellogg of Deutsche Aerospace contributed the port to the MIL-STD-1750A.
- Richard Kenner of the New York University Ultracomputer Research Laboratory wrote the machine descriptions for the AMD 29000, the DEC Alpha, the IBM RT PC, and the IBM RS/6000 as well as the support for instruction attributes. He also made

changes to better support RISC processors including changes to common subexpression elimination, strength reduction, function calling sequence handling, and condition code support, in addition to generalizing the code for frame pointer elimination and delay slot scheduling. Richard Kenner was also the head maintainer of GCC for several years.

- Mumit Khan for various contributions to the Cygwin and Mingw32 ports and maintaining binary releases for Microsoft Windows hosts, and for massive libstdc++ porting work to Cygwin/Mingw32.
- Robin Kirkham for cpu32 support.
- Mark Klein for PA improvements.
- Thomas Koenig for various bug fixes.
- Bruce Korb for the new and improved fixincludes code.
- Benjamin Kosnik for his G++ work and for leading the libstdc++-v3 effort.
- Maxim Kuvyrkov for contributions to the instruction scheduler, the Android and m68k/Coldfire ports, and optimizations.
- Charles LaBrec contributed the support for the Integrated Solutions 68020 system.
- Asher Langton and Mike Kumbera for contributing Cray pointer support to GNU Fortran, and for other GNU Fortran improvements.
- Jeff Law for his direction via the steering committee, coordinating the entire egcs project and GCC 2.95, rolling out snapshots and releases, handling merges from GCC2, reviewing tons of patches that might have fallen through the cracks else, and random but extensive hacking.
- Walter Lee for work on the TILE-Gx and TILEPro ports.
- Marc Lehmann for his direction via the steering committee and helping with analysis and improvements of x86 performance.
- Victor Leikehman for work on GNU Fortran.
- Ted Lemon wrote parts of the RTL reader and printer.
- Kriang Lerdsuwanakij for C++ improvements including template as template parameter support, and many C++ fixes.
- Warren Levy for tremendous work on libgcj (Java Runtime Library) and random work on the Java front end.
- Alain Lichnewsky ported GCC to the MIPS CPU.
- Oskar Liljeblad for hacking on AWT and his many Java bug reports and patches.
- Robert Lipe for OpenServer support, new testsuites, testing, etc.
- Chen Liqin for various S+core related fixes/improvement, and for maintaining the S+core port.
- Martin Liska for his work on identical code folding, the sanitizers, HSA, general bug fixing and for running automated regression testing of GCC and reporting numerous bugs.
- Weiwen Liu for testing and various bug fixes.
- Manuel López-Ibáñez for improving '`-Wconversion`' and many other diagnostics fixes and improvements.
- Dave Love for his ongoing work with the Fortran front end and runtime libraries.

- Martin von Löwis for internal consistency checking infrastructure, various C++ improvements including namespace support, and tons of assistance with libstdc++/compiler merges.
- H.J. Lu for his previous contributions to the steering committee, many x86 bug reports, prototype patches, and keeping the GNU/Linux ports working.
- Greg McGary for random fixes and (someday) bounded pointers.
- Andrew MacLeod for his ongoing work in building a real EH system, various code generation improvements, work on the global optimizer, etc.
- Vladimir Makarov for hacking some ugly i960 problems, PowerPC hacking improvements to compile-time performance, overall knowledge and direction in the area of instruction scheduling, design and implementation of the automaton based instruction scheduler and design and implementation of the integrated and local register allocators.
- David Malcolm for his work on improving GCC diagnostics, JIT, self-tests and unit testing.
- Bob Manson for his behind the scenes work on dejagnu.
- John Marino for contributing the DragonFly BSD port.
- Philip Martin for lots of libstdc++ string and vector iterator fixes and improvements, and string clean up and testsuites.
- Michael Matz for his work on dominance tree discovery, the x86-64 port, link-time optimization framework and general optimization improvements.
- All of the Mauve project contributors for Java test code.
- Bryce McKinlay for numerous GCJ and libgcj fixes and improvements.
- Adam Megacz for his work on the Microsoft Windows port of GCJ.
- Michael Meissner for LRS framework, ia32, m32r, v850, m88k, MIPS, powerpc, haifa, ECOFF debug support, and other assorted hacking.
- Jason Merrill for his direction via the steering committee and leading the G++ effort.
- Martin Michlmayr for testing GCC on several architectures using the entire Debian archive.
- David Miller for his direction via the steering committee, lots of SPARC work, improvements in jump.c and interfacing with the Linux kernel developers.
- Gary Miller ported GCC to Charles River Data Systems machines.
- Alfred Minarik for libstdc++ string and ios bug fixes, and turning the entire libstdc++ testsuite namespace-compatible.
- Mark Mitchell for his direction via the steering committee, mountains of C++ work, load/store hoisting out of loops, alias analysis improvements, ISO C `restrict` support, and serving as release manager from 2000 to 2011.
- Alan Modra for various GNU/Linux bits and testing.
- Toon Moene for his direction via the steering committee, Fortran maintenance, and his ongoing work to make us make Fortran run fast.
- Jason Molenda for major help in the care and feeding of all the services on the gcc.gnu.org (formerly egcs.cygnus.com) machine—mail, web services, ftp services, etc etc. Doing all this work on scrap paper and the backs of envelopes would have been... difficult.

- Catherine Moore for fixing various ugly problems we have sent her way, including the haifa bug which was killing the Alpha & PowerPC Linux kernels.
- Mike Moreton for his various Java patches.
- David Mosberger-Tang for various Alpha improvements, and for the initial IA-64 port.
- Stephen Moshier contributed the floating point emulator that assists in cross-compilation and permits support for floating point numbers wider than 64 bits and for ISO C99 support.
- Bill Moyer for his behind the scenes work on various issues.
- Philippe De Muyter for his work on the m68k port.
- Joseph S. Myers for his work on the PDP-11 port, format checking and ISO C99 support, and continuous emphasis on (and contributions to) documentation.
- Nathan Myers for his work on libstdc++-v3: architecture and authorship through the first three snapshots, including implementation of locale infrastructure, string, shadow C headers, and the initial project documentation (DESIGN, CHECKLIST, and so forth). Later, more work on MT-safe string and shadow headers.
- Felix Natter for documentation on porting libstdc++.
- Nathanael Nerode for cleaning up the configuration/build process.
- NeXT, Inc. donated the front end that supports the Objective-C language.
- Hans-Peter Nilsson for the CRIS and MMIX ports, improvements to the search engine setup, various documentation fixes and other small fixes.
- Geoff Noer for his work on getting cygwin native builds working.
- Vegard Nossum for running automated regression testing of GCC and reporting numerous bugs.
- Diego Novillo for his work on Tree SSA, OpenMP, SPEC performance tracking web pages, GIMPLE tuples, and assorted fixes.
- David O'Brien for the FreeBSD/alpha, FreeBSD/AMD x86-64, FreeBSD/ARM, FreeBSD/PowerPC, and FreeBSD/SPARC64 ports and related infrastructure improvements.
- Alexandre Oliva for various build infrastructure improvements, scripts and amazing testing work, including keeping libtool issues sane and happy.
- Stefan Olsson for work on mt_alloc.
- Melissa O'Neill for various NeXT fixes.
- Rainer Orth for random MIPS work, including improvements to GCC's o32 ABI support, improvements to dejagnu's MIPS support, Java configuration clean-ups and porting work, and maintaining the IRIX, Solaris 2, and Tru64 UNIX ports.
- Steven Pemberton for his contribution of '`enquire`' which allowed GCC to determine various properties of the floating point unit and generate '`float.h`' in older versions of GCC.
- Hartmut Penner for work on the s390 port.
- Paul Petersen wrote the machine description for the Alliant FX/8.
- Alexandre Petit-Bianco for implementing much of the Java compiler and continued Java maintainership.

- Matthias Pfaller for major improvements to the NS32k port.
- Gerald Pfeifer for his direction via the steering committee, pointing out lots of problems we need to solve, maintenance of the web pages, and taking care of documentation maintenance in general.
- Marek Polacek for his work on the C front end, the sanitizers and general bug fixing.
- Andrew Pinski for processing bug reports by the dozen.
- Ovidiu Predescu for his work on the Objective-C front end and runtime libraries.
- Jerry Quinn for major performance improvements in C++ formatted I/O.
- Ken Raeburn for various improvements to checker, MIPS ports and various cleanups in the compiler.
- Rolf W. Rasmussen for hacking on AWT.
- David Reese of Sun Microsystems contributed to the Solaris on PowerPC port.
- John Regehr for running automated regression testing of GCC and reporting numerous bugs.
- Volker Reichelt for running automated regression testing of GCC and reporting numerous bugs and for keeping up with the problem reports.
- Joern Rennecke for maintaining the sh port, loop, regmove & reload hacking and developing and maintaining the Epiphany port.
- Loren J. Rittle for improvements to libstdc++-v3 including the FreeBSD port, threading fixes, thread-related configury changes, critical threading documentation, and solutions to really tricky I/O problems, as well as keeping GCC properly working on FreeBSD and continuous testing.
- Craig Rodrigues for processing tons of bug reports.
- Ola Rönnerup for work on mt_alloc.
- Gavin Romig-Koch for lots of behind the scenes MIPS work.
- David Ronis inspired and encouraged Craig to rewrite the G77 documentation in texinfo format by contributing a first pass at a translation of the old '`g77-0.5.16/f/DOC`' file.
- Ken Rose for fixes to GCC's delay slot filling code.
- Ira Rosen for her contributions to the auto-vectorizer.
- Paul Rubin wrote most of the preprocessor.
- Pétur Runólfsson for major performance improvements in C++ formatted I/O and large file support in C++ filebuf.
- Chip Salzenberg for libstdc++ patches and improvements to locales, traits, Makefiles, libio, libtool hackery, and "long long" support.
- Juha Sarlin for improvements to the H8 code generator.
- Greg Satz assisted in making GCC work on HP-UX for the 9000 series 300.
- Roger Sayle for improvements to constant folding and GCC's RTL optimizers as well as for fixing numerous bugs.
- Bradley Schatz for his work on the GCJ FAQ.
- Peter Schauer wrote the code to allow debugging to work on the Alpha.
- William Schelter did most of the work on the Intel 80386 support.

- Tobias Schlüter for work on GNU Fortran.
- Bernd Schmidt for various code generation improvements and major work in the reload pass, serving as release manager for GCC 2.95.3, and work on the Blackfin and C6X ports.
- Peter Schmid for constant testing of libstdc++—especially application testing, going above and beyond what was requested for the release criteria—and libstdc++ header file tweaks.
- Jason Schroeder for jcf-dump patches.
- Andreas Schwab for his work on the m68k port.
- Lars Segerlund for work on GNU Fortran.
- Dodji Seketeli for numerous C++ bug fixes and debug info improvements.
- Tim Shen for major work on `<regex>`.
- Joel Sherrill for his direction via the steering committee, RTEMS contributions and RTEMS testing.
- Nathan Sidwell for many C++ fixes/improvements.
- Jeffrey Siegal for helping RMS with the original design of GCC, some code which handles the parse tree and RTL data structures, constant folding and help with the original VAX & m68k ports.
- Kenny Simpson for prompting libstdc++ fixes due to defect reports from the LWG (thereby keeping GCC in line with updates from the ISO).
- Franz Sirl for his ongoing work with making the PPC port stable for GNU/Linux.
- Andrey Slepuhin for assorted AIX hacking.
- Trevor Smigiel for contributing the SPU port.
- Christopher Smith did the port for Convex machines.
- Danny Smith for his major efforts on the Mingw (and Cygwin) ports. Retired from GCC maintainership August 2010, having mentored two new maintainers into the role.
- Randy Smith finished the Sun FPA support.
- Ed Smith-Rowland for his continuous work on libstdc++-v3, special functions, `<random>`, and various improvements to C++11 features.
- Scott Snyder for queue, iterator, istream, and string fixes and libstdc++ testsuite entries. Also for providing the patch to G77 to add rudimentary support for `INTEGER*1`, `INTEGER*2`, and `LOGICAL*1`.
- Zdenek Sojka for running automated regression testing of GCC and reporting numerous bugs.
- Arseny Solokha for running automated regression testing of GCC and reporting numerous bugs.
- Jayant Sonar for contributing the CR16 port.
- Brad Spencer for contributions to the GLIBCPP_FORCE_NEW technique.
- Richard Stallman, for writing the original GCC and launching the GNU project.
- Jan Stein of the Chalmers Computer Society provided support for Genix, as well as part of the 32000 machine description.

- Gerhard Steinmetz for running automated regression testing of GCC and reporting numerous bugs.
- Nigel Stephens for various mips16 related fixes/improvements.
- Jonathan Stone wrote the machine description for the Pyramid computer.
- Graham Stott for various infrastructure improvements.
- John Stracke for his Java HTTP protocol fixes.
- Mike Stump for his Elxsi port, G++ contributions over the years and more recently his vxworks contributions
- Jeff Sturm for Java porting help, bug fixes, and encouragement.
- Zhendong Su for running automated regression testing of GCC and reporting numerous bugs.
- Chengnian Sun for running automated regression testing of GCC and reporting numerous bugs.
- Shigeya Suzuki for this fixes for the bsdi platforms.
- Ian Lance Taylor for the Go frontend, the initial mips16 and mips64 support, general configury hacking, fixincludes, etc.
- Holger Teutsch provided the support for the Clipper CPU.
- Gary Thomas for his ongoing work to make the PPC work for GNU/Linux.
- Paul Thomas for contributions to GNU Fortran.
- Philipp Thomas for random bug fixes throughout the compiler
- Jason Thorpe for thread support in libstdc++ on NetBSD.
- Kresten Krab Thorup wrote the run time support for the Objective-C language and the fantastic Java bytecode interpreter.
- Michael Tiemann for random bug fixes, the first instruction scheduler, initial C++ support, function integration, NS32k, SPARC and M88k machine description work, delay slot scheduling.
- Andreas Tobler for his work porting libgcj to Darwin.
- Teemu Torma for thread safe exception handling support.
- Leonard Tower wrote parts of the parser, RTL generator, and RTL definitions, and of the VAX machine description.
- Daniel Towner and Hariharan Sandanagobalane contributed and maintain the picoChip port.
- Tom Tromey for internationalization support and for his many Java contributions and libgcj maintainership.
- Lassi Tuura for improvements to config.guess to determine HP processor types.
- Petter Urkedal for libstdc++ CXXFLAGS, math, and algorithms fixes.
- Andy Vaught for the design and initial implementation of the GNU Fortran front end.
- Brent Verner for work with the libstdc++ cshadow files and their associated configure steps.
- Todd Vierling for contributions for NetBSD ports.
- Andrew Waterman for contributing the RISC-V port, as well as maintaining it.

- Jonathan Wakely for contributing libstdc++ Doxygen notes and XHTML guidance and maintaining libstdc++.
- Dean Wakerley for converting the install documentation from HTML to texinfo in time for GCC 3.0.
- Krister Walfridsson for random bug fixes.
- Feng Wang for contributions to GNU Fortran.
- Stephen M. Webb for time and effort on making libstdc++ shadow files work with the tricky Solaris 8+ headers, and for pushing the build-time header tree. Also, for starting and driving the `<regex>` effort.
- John Wehle for various improvements for the x86 code generator, related infrastructure improvements to help x86 code generation, value range propagation and other work, WE32k port.
- Ulrich Weigand for work on the s390 port.
- Janus Weil for contributions to GNU Fortran.
- Zack Weinberg for major work on cpplib and various other bug fixes.
- Matt Welsh for help with Linux Threads support in GCJ.
- Urban Widmark for help fixing java.io.
- Mark Wielaard for new Java library code and his work integrating with Classpath.
- Dale Wiles helped port GCC to the Tahoe.
- Bob Wilson from Tensilica, Inc. for the Xtensa port.
- Jim Wilson for his direction via the steering committee, tackling hard problems in various places that nobody else wanted to work on, strength reduction and other loop optimizations.
- Paul Woegerer and Tal Agmon for the CRX port.
- Carlo Wood for various fixes.
- Tom Wood for work on the m88k port.
- Chung-Ju Wu for his work on the Andes NDS32 port.
- Canqun Yang for work on GNU Fortran.
- Masanobu Yuhara of Fujitsu Laboratories implemented the machine description for the Tron architecture (specifically, the Gmicro).
- Kevin Zachmann helped port GCC to the Tahoe.
- Ayal Zaks for Swing Modulo Scheduling (SMS).
- Qirun Zhang for running automated regression testing of GCC and reporting numerous bugs.
- Xiaoqiang Zhang for work on GNU Fortran.
- Gilles Zunino for help porting Java to Irix.

The following people are recognized for their contributions to GNAT, the Ada front end of GCC:

- Bernard Banner
- Romain Berrendonner

- Geert Bosch
- Emmanuel Briot
- Joel Brobecker
- Ben Brosgol
- Vincent Celier
- Arnaud Charlet
- Chien Chieng
- Cyrille Comar
- Cyrille Crozes
- Robert Dewar
- Gary Dismukes
- Robert Duff
- Ed Falis
- Ramon Fernandez
- Sam Figueroa
- Vasiliy Fofanov
- Michael Friess
- Franco Gasperoni
- Ted Giering
- Matthew Gingell
- Laurent Guerby
- Jerome Guitton
- Olivier Hainque
- Jerome Hugues
- Hristian Kirtchev
- Jerome Lambourg
- Bruno Leclerc
- Albert Lee
- Sean McNeil
- Javier Miranda
- Laurent Nana
- Pascal Obry
- Dong-Ik Oh
- Laurent Pautet
- Brett Porter
- Thomas Quinot
- Nicolas Roche
- Pat Rogers
- Jose Ruiz

- Douglas Rupp
- Sergey Rybin
- Gail Schenker
- Ed Schonberg
- Nicolas Setton
- Samuel Tardieu

The following people are recognized for their contributions of new features, bug reports, testing and integration of classpath/libgcj for GCC version 4.1:

- Lillian Angel for `JTree` implementation and lots Free Swing additions and bug fixes.
- Wolfgang Baer for `GapContent` bug fixes.
- Anthony Balkissoon for `JList`, Free Swing 1.5 updates and mouse event fixes, lots of Free Swing work including `JTable` editing.
- Stuart Ballard for RMI constant fixes.
- Goffredo Baroncelli for `HTTPURLConnection` fixes.
- Gary Benson for `MessageFormat` fixes.
- Daniel Bonniot for `Serialization` fixes.
- Chris Burdess for lots of gnu.xml and http protocol fixes, `StAX` and `DOM` `xml:id` support.
- Ka-Hing Cheung for `TreePath` and `TreeSelection` fixes.
- Archie Cobbs for build fixes, VM interface updates, `URLClassLoader` updates.
- Kelley Cook for build fixes.
- Martin Cordova for Suggestions for better `SocketTimeoutException`.
- David Daney for `BitSet` bug fixes, `HttpURLConnection` rewrite and improvements.
- Thomas Fitzsimmons for lots of upgrades to the gtk+ AWT and Cairo 2D support. Lots of imageio framework additions, lots of AWT and Free Swing bug fixes.
- Jeroen Frijters for `ClassLoader` and nio cleanups, serialization fixes, better `Proxy` support, bug fixes and IKVM integration.
- Santiago Gala for `AccessControlContext` fixes.
- Nicolas Geoffray for `VMClassLoader` and `AccessController` improvements.
- David Gilbert for `basic` and `metal` icon and plaf support and lots of documenting, Lots of Free Swing and metal theme additions. `MetalIconFactory` implementation.
- Anthony Green for `MIDI` framework, `ALSA` and `DSSI` providers.
- Andrew Haley for `Serialization` and `URLClassLoader` fixes, gcj build speedups.
- Kim Ho for `JFileChooser` implementation.
- Andrew John Hughes for `Locale` and net fixes, URI RFC2986 updates, `Serialization` fixes, `Properties` XML support and generic branch work, VMIntegration guide update.
- Bastiaan Huisman for `TimeZone` bug fixing.
- Andreas Jaeger for mprec updates.
- Paul Jenner for better '`-Werror`' support.
- Ito Kazumitsu for `NetworkInterface` implementation and updates.

- Roman Kennke for `BoxLayout`, `GrayFilter` and `SplitPane`, plus bug fixes all over. Lots of Free Swing work including styled text.
- Simon Kitching for `String` cleanups and optimization suggestions.
- Michael Koch for configuration fixes, `Locale` updates, bug and build fixes.
- Guilhem Lavaux for configuration, thread and channel fixes and Kaffe integration. JCL native `Pointer` updates. Logger bug fixes.
- David Lichteblau for JCL support library global/local reference cleanups.
- Aaron Luchko for JDWP updates and documentation fixes.
- Ziga Mahkovec for `Graphics2D` upgraded to Cairo 0.5 and new regex features.
- Sven de Marothy for BMP imageio support, CSS and `TextLayout` fixes. `GtkImage` rewrite, 2D, awt, free swing and date/time fixes and implementing the Qt4 peers.
- Casey Marshall for crypto algorithm fixes, `FileChannel` lock, `SystemLogger` and `FileHandler` rotate implementations, NIO `FileChannel.map` support, security and policy updates.
- Bryce McKinlay for RMI work.
- Audrius Meskauskas for lots of Free Corba, RMI and HTML work plus testing and documenting.
- Kalle Olavi Niemitalo for build fixes.
- Rainer Orth for build fixes.
- Andrew Overholt for `File` locking fixes.
- Ingo Proetel for `Image`, `Logger` and `URLClassLoader` updates.
- Olga Rodimina for `MenuSelectionManager` implementation.
- Jan Roehrich for `BasicTreeUI` and `JTree` fixes.
- Julian Scheid for documentation updates and gjdoc support.
- Christian Schlichtherle for zip fixes and cleanups.
- Robert Schuster for documentation updates and beans fixes, `TreeNode` enumerations and `ActionCommand` and various fixes, XML and URL, AWT and Free Swing bug fixes.
- Keith Seitz for lots of JDWP work.
- Christian Thalinger for 64-bit cleanups, Configuration and VM interface fixes and `CACAO` integration, `fdlibm` updates.
- Gael Thomas for `VMClassLoader` boot packages support suggestions.
- Andreas Tobler for Darwin and Solaris testing and fixing, `Qt4` support for Darwin/OS X, `Graphics2D` support, `gtk+` updates.
- Dalibor Topic for better `DEBUG` support, build cleanups and Kaffe integration. `Qt4` build infrastructure, `SHA1PRNG` and `GdkPixbugDecoder` updates.
- Tom Tromey for Eclipse integration, generics work, lots of bug fixes and gcj integration including coordinating The Big Merge.
- Mark Wielaard for bug fixes, packaging and release management, `Clipboard` implementation, system call interrupts and network timeouts and `GdkPixpufDecoder` fixes.

In addition to the above, all of which also contributed time and energy in testing GCC, we would like to thank the following for their contributions to testing:

- Michael Abd-El-Malek
- Thomas Arend
- Bonzo Armstrong
- Steven Ashe
- Chris Baldwin
- David Billinghurst
- Jim Blandy
- Stephane Bortzmeyer
- Horst von Brand
- Frank Braun
- Rodney Brown
- Sidney Cadot
- Bradford Castalia
- Robert Clark
- Jonathan Corbet
- Ralph Doncaster
- Richard Emberson
- Levente Farkas
- Graham Fawcett
- Mark Fernyhough
- Robert A. French
- Jörgen Freyh
- Mark K. Gardner
- Charles-Antoine Gauthier
- Yung Shing Gene
- David Gilbert
- Simon Gornall
- Fred Gray
- John Griffin
- Patrik Hagglund
- Phil Hargett
- Amancio Hasty
- Takafumi Hayashi
- Bryan W. Headley
- Kevin B. Hendricks
- Joep Jansen
- Christian Joensson
- Michel Kern
- David Kidd

- Tobias Kuipers
- Anand Krishnaswamy
- A. O. V. Le Blanc
- llewelly
- Damon Love
- Brad Lucier
- Matthias Klose
- Martin Knoblauch
- Rick Lutowski
- Jesse Macnish
- Stefan Morrell
- Anon A. Mous
- Matthias Mueller
- Pekka Nikander
- Rick Niles
- Jon Olson
- Magnus Persson
- Chris Pollard
- Richard Polton
- Derk Reefman
- David Rees
- Paul Reilly
- Tom Reilly
- Torsten Rueger
- Danny Sadinoff
- Marc Schifer
- Erik Schnetter
- Wayne K. Schroll
- David Schuler
- Vin Shelton
- Tim Souder
- Adam Sulmicki
- Bill Thorson
- George Talbot
- Pedro A. M. Vazquez
- Gregory Warnes
- Ian Watson
- David E. Young
- And many others

And finally we'd like to thank everyone who uses the compiler, provides feedback and generally reminds us why we're doing this work in the first place.

Option Index

GCC's command line options are indexed here without any initial '-' or '--'. Where an option has both positive and negative forms (such as '-*foption*' and '-fno-*option*'), relevant entries in the manual are indexed under the most appropriate form; it may sometimes be useful to look up both forms.

#

... 34

-

-Wattribute-alias ... 97
-Wno-attribute-alias ... 97

8

80387 ... 409

A

all_load ... 296
allowable_client ... 298
ansi ... 5, 37, 641, 881
arch_errors_fatal ... 296
aux-info ... 41
A ... 207

B

Bdynamic ... 401
bind_at_load ... 296
B ... 217
Bstatic ... 401
bundle ... 296
bundle_loader ... 296

C

c ... 33, 210
C ... 207
CC ... 207
client_name ... 298
compatibility_version ... 298
coverage ... 190
current_version ... 298

D

d ... 208, 229
da ... 232
dA ... 232
dD ... 208, 232
dead_strip ... 298
dependency-file ... 298
dH ... 232
dI ... 208
dM ... 208
dN ... 208
dp ... 232
dP ... 232
dumpfullversion ... 245
dumpmachine ... 244
dumpspecs ... 245
dumpversion ... 244
dU ... 208
dx ... 232
dylib_file ... 298
dylinker_install_name ... 298
dynamic ... 298
dynamiclib ... 296
D ... 202

E

e ... 212
E ... 33, 210
EB ... 262, 292, 326
EL ... 263, 292, 326
entry ... 212
exported_symbols_list ... 298

F

fabi-compat-version ... 45
fabi-version ... 44
faccess-control ... 46
fada-spec-parent ... 36
faggressive-loop-optimizations ... 133
falign-functions ... 152
falign-jumps ... 153
falign-labels ... 152
falign-loops ... 153
faligned-new ... 46
fallow-parameterless-variadic-functions ... 41
fasan-shadow-offset ... 196
fasm ... 41
fassociative-math ... 161
fasynchronous-unwind-tables ... 220
fauto-inc-dec ... 134
fauto-profile ... 159
fbranch-count-reg ... 131
fbranch-probabilities ... 163
fbranch-target-load-optimize ... 166
fbranch-target-load-optimize2 ... 166

fbtr-bb-exclusive 166
fbuiltin 41
fcall-saved 225
fcall-used 225
fcaller-saves 140
fcf-protection 197
fchar8_t 46
fcheck-new 46
fchecking 239
fcode-hoisting 140
fcombine-stack-adjustments 140
fcommon 222, 538
fcompare-debug 241
fcompare-debug-second 241
fcompare-elim 158
fconcepts 47
fcond-mismatch 43
fconserve-stack 140
fconstant-string-class 60
fconstexpr-depth 47
fconstexpr-loop-limit 47
fconstexpr-ops-limit 47
fcprop-registers 158
fcrossjumping 134
fcse-follow-jumps 132
fcse-skip-blocks 132
fcx-fortran-rules 163
fcx-limited-range 163
fdata-sections 165
fdbg-cnt 243
fdbg-cnt-list 243
fdce 134
fdebug-cpp 208
fdebug-prefix-map 120
fdebug-types-section 121
fdeclone-ctor-dtor 134
fdeduce-init-list 47
fdefer-pop 128
fdelayed-branch 136
fdelete-dead-exceptions 220
fdelete-null-pointer-checks 134
fdevirtualize 135
fdevirtualize-at-ltrans 135
fdevirtualize-speculatively 135
fdiagnostics-color 64
fdiagnostics-format 67
fdiagnostics-generate-patch 66
fdiagnostics-minimum-margin-width 66
fdiagnostics-parseable-fixits 66
fdiagnostics-show-caret 65
fdiagnostics-show-labels 66
fdiagnostics-show-line-numbers 66
fdiagnostics-show-location 64
fdiagnostics-show-option 65
fdiagnostics-show-template-tree 67
fdirectives-only 205
fdisable- 238
fdollars-in-identifiers 205, 871
fdpic 387
fdse 134
fdump-ada-spec 36
fdump-debug 232
fdump-earlydebug 232
fdump-final-insns 241
fdump-go-spec 36
fdump-ipa 233
fdump-lang 233
fdump-lang-all 233
fdump-noaddr 232
fdump-passes 234
fdump-rtl-alignments 229
fdump-rtl-all 232
fdump-rtl-asmcons 229
fdump-rtl-auto_inc_dec 229
fdump-rtl-barriers 229
fdump-rtl-bbpart 229
fdump-rtl-bbro 229
fdump-rtl-btl2 229
fdump-rtl-bypass 229
fdump-rtl-ce1 229
fdump-rtl-ce2 229
fdump-rtl-ce3 229
fdump-rtl-combine 229
fdump-rtl-compgotos 229
fdump-rtl-cprop_hardreg 229
fdump-rtl-csa 230
fdump-rtl-cse1 230
fdump-rtl-cse2 230
fdump-rtl-dbr 230
fdump-rtl-dce 230
fdump-rtl-dce1 230
fdump-rtl-dce2 230
fdump-rtl-dfinish 232
fdump-rtl-dfinit 232
fdump-rtl-eh 230
fdump-rtl-eh_ranges 230
fdump-rtl-expand 230
fdump-rtl-fwprop1 230
fdump-rtl-fwprop2 230
fdump-rtl-gcse1 230
fdump-rtl-gcse2 230
fdump-rtl-init-regs 230
fdump-rtl-initvals 230
fdump-rtl-into_cfglayout 230
fdump-rtl-ira 230
fdump-rtl-jump 230
fdump-rtl-loop2 230
fdump-rtl-mach 231
fdump-rtl-mode_sw 231
fdump-rtl-outof_cfglayout 231
fdump-rtl-*pass* 229
fdump-rtl-peephole2 231
fdump-rtl-postreload 231
fdump-rtl-pro_and_epilogue 231
fdump-rtl-ree 231
fdump-rtl-regclass 232

fdump-rtl-rnreg 231
fdump-rtl-sched1 231
fdump-rtl-sched2 231
fdump-rtl-seqabstr 231
fdump-rtl-shorten 231
fdump-rtl-sibling 231
fdump-rtl-sms 231
fdump-rtl-split1 231
fdump-rtl-split2 231
fdump-rtl-split3 231
fdump-rtl-split4 231
fdump-rtl-split5 231
fdump-rtl-stack 231
fdump-rtl-subreg1 232
fdump-rtl-subreg2 232
fdump-rtl-subregs_of_mode_finish 232
fdump-rtl-subregs_of_mode_init 232
fdump-rtl-unshare 232
fdump-rtl-vartrack 232
fdump-rtl-vregs 232
fdump-rtl-web 232
fdump-statistics 234
fdump-tree 234
fdump-tree-all 234
fdump-unnumbered 233
fdump-unnumbered-links 233
fdwarf2-cfi-asm 125
fearly-inlining 130
felide-constructors 47
felide-type 67
feliminate-unused-debug-symbols 120
feliminate-unused-debug-types 125
femit-class-debug-always 120
femit-struct-debug-baseonly 124
femit-struct-debug-detailed 124
femit-struct-debug-reduced 124
fenable- 238
fenforce-eh-specs 48
fexceptions 220
fexcess-precision 160
fexec-charset 206
fexpensive-optimizations 135
fext-numeric-literals 58
fextended-identifiers 205
fextern-tls-init 48
ffast-math 160
ffat-lto-objects 158
ffile-prefix-map 36
ffinite-math-only 161
ffix-and-continue 296
ffixed 225
ffloat-store 159, 876
fforward-propagate 128
ffp-contract 128
ffp-int-builtin-inexact 162
ffreestanding 6, 42, 74, 483
ffunction-cse 132
ffunction-sections 165
fgcse 133
fgcse-after-reload 133
fgcse-las 133
fgcse-lm 133
fgcse-sm 133
fgimple 42
fgnu-keywords 48
fgnu-runtime 60
fgnu-tm 43
fgnu-unique 220
fgnu89-inline 40
fgraphite-identity 145
fguess-branch-probability 150
fhoist-adjacent-loads 141
fhosted 42
fident 222
fif-conversion 134
fif-conversion2 134
fiji 253
filelist 298
fimplement-inlines 49
fimplicit-inline-templates 48
fimplicit-templates 48
findirect-data 296
findirect-inlining 129
finhibit-size-directive 222
finline 129
finline-functions 129
finline-functions-called-once 130
finline-limit 130
finline-small-functions 129
finput-charset 206
finstrument-functions 200, 488
finstrument-functions-exclude-file-list 201
finstrument-functions-exclude-function-list 201
fipa-bit-cp 142
fipa-cp 141
fipa-cp-clone 142
fipa-icf 142
fipa-profile 141
fipa-pta 141
fipa-pure-const 141
fipa-ra 140
fipa-reference 141
fipa-reference-addressable 141
fipa-sra 130
fipa-stack-alignment 141
fipa-vrp 142
fira-algorithm 136
fira-hoist-pressure 136
fira-loop-pressure 136
fira-region 136
fira-share-save-slots 136
fira-share-spill-slots 136
fira-verbose 242
fisolate-erroneous-paths-attribute 143

fisolate-erroneous-paths-dereference..... 143
fivar-visibility..... 62
fivopts..... 147
fjump-tables..... 224
fkeep-inline-dllexport..... 130
fkeep-inline-functions..... 131, 565
fkeep-static-consts..... 131
fkeep-static-functions..... 131
flat_namespace..... 298
flax-vector-conversions..... 43
fleading-underscore..... 225
flifetime-dse..... 135
flinker-output..... 210
flive-patching..... 142
flive-range-shrinkage..... 135
flocal-ivars..... 62
floop-block..... 145
floop-interchange..... 146
floop-nest-optimize..... 145
floop-parallelize-all..... 145
floop-strip-mine..... 145
floop-unroll-and-jam..... 146
flra-remat..... 136
flto..... 154
flto-compression-level..... 157
flto-odr-type-merging..... 157
flto-partition..... 157
flto-report..... 242
flto-report-wpa..... 242
fmacro-prefix-map..... 205
fmath-errno..... 160
fmax-errors..... 70
fmem-report..... 242
fmem-report-wpa..... 242
fmerge-all-constants..... 131
fmerge-constants..... 131
fmerge-debug-strings..... 120
fmessage-length..... 64
fmodulo-sched..... 131
fmodulo-sched-allow-regmoves..... 131
fmove-loop-invariants..... 165
fms-extensions..... 43, 49, 810
fnew-inheriting-ctors..... 49
fnew-ttp-matching..... 49
fnext-runtime..... 60
fnil-receivers..... 60
fno-access-control..... 46
fno-asm..... 41
fno-branch-count-reg..... 131
fno-builtin..... 41, 74, 483, 641
fno-canonical-system-headers..... 205
fno-char8_t..... 46
fno-checking..... 239
fno-common..... 222, 538
fno-compare-debug..... 241
fno-debug-types-section..... 121
fno-default-inline..... 565
fno-defer-pop..... 128
fno-diagnostics-show-caret..... 65
fno-diagnostics-show-labels..... 66
fno-diagnostics-show-line-numbers..... 66
fno-diagnostics-show-option..... 65
fno-dwarf2-cfi-asm..... 125
fno-elide-constructors..... 47
fno-elide-type..... 67
fno-eliminate-unused-debug-types..... 125
fno-enforce-eh-specs..... 48
fno-ext-numeric-literals..... 58
fno-extern-tls-init..... 48
fno-fp-int-builtin-inexact..... 162
fno-function-cse..... 132
fno-gnu-keywords..... 48
fno-gnu-unique..... 220
fno-guess-branch-probability..... 150
fno-ident..... 222
fno-implement-inlines..... 49, 818
fno-implicit-inline-templates..... 48
fno-implicit-templates..... 48, 820
fno-inline..... 129
fno-ira-share-save-slots..... 136
fno-ira-share-spill-slots..... 136
fno-jump-tables..... 224
fno-keep-inline-dllexport..... 130
fno-lifetime-dse..... 135
fno-local-ivars..... 62
fno-math-errno..... 160
fno-merge-debug-strings..... 120
fno-nil-receivers..... 60
fno-nonansi-builtins..... 49
fno-operator-names..... 49
fno-optional-diags..... 49
fno-peephole..... 150
fno-peephole2..... 150
fno-plt..... 224
fno-pretty-templates..... 49
fno-printf-return-value..... 149
fno-rtti..... 50
fno-sanitize-recover..... 196
fno-sanitize=all..... 195
fno-sched-interblock..... 137
fno-sched-spec..... 137
fno-set-stack-executable..... 426
fno-show-column..... 67
fno-signed-bitfields..... 44
fno-signed-zeros..... 162
fno-stack-limit..... 199
fno-threadsafe-statics..... 51
fno-toplevel-reorder..... 153
fno-trapping-math..... 162
fno-unsigned-bitfields..... 44
fno-use-cxa-get-exception-ptr..... 51
fno-var-tracking-assignments..... 121
fno-var-tracking-assignments-toggle..... 242
fno-weak..... 52
fno-working-directory..... 206
fno-writable-relocated-rdata..... 426

fno-zero-initialized-in-bss 132
fnon-call-exceptions 220
fnonansi-builtins 49
fnothrow-opt 49
fobjc-abi-version 60
fobjc-call-cxx-cdtors 61
fobjc-direct-dispatch 61
fobjc-exceptions 61
fobjc-gc 61
fobjc-nilcheck 61
fobjc-std 62
fomit-frame-pointer 129
fopenacc 42
fopenacc-dim 42
fopenmp 42
fopenmp-simd 43
foperator-names 49
fopt-info 236
foptimize-sibling-calls 129
foptimize-strlen 129
foptional-diags 49
force_cpusubtype_ALL 296
force_flat_namespace 298
fpack-struct 225
fpartial-inlining 149
fpatchable-function-entry 201
fpcc-struct-return 221, 873
fpch-deps 206
fpch-preprocess 206
fpeel-loops 165
fpeephole 150
fpeephole2 150
fpermissive 49
fpermitted-flt-eval-methods 40
fpermitted-flt-eval-methods=c11 40
fpermitted-flt-eval-methods=ts-18661-3 ... 40
fpic 224
fPIC 224
fpie 224
fPIE 224
fplan9-extensions 43, 810
fplt 224
fplugin 36
fplugin-arg 36
fpost-ipa-mem-report 242
fpre-ipa-mem-report 242
fpredictive-commoning 149
fprefetch-loop-arrays 149
fpreprocessed 204
fpretty-templates 49
fprintf-return-value 149
fprofile-abs-path 191
fprofile-arcs 189, 649
fprofile-correction 158
fprofile-dir 191
fprofile-exclude-files 192
fprofile-filter-files 192
fprofile-generate 191
fprofile-reorder-functions 164
fprofile-report 242
fprofile-update 191
fprofile-use 158
fprofile-values 164
fpu 374
frandom-seed 239
freciprocal-math 161
frecord-gcc-switches 223
free 135
freg-struct-return 221
frename-registers 164
freorder-blocks 150
freorder-blocks-algorithm 150
freorder-blocks-and-partition 150
freorder-functions 151
freplace-objc-classes 62
frepo 50, 820
freport-bug 233
frerun-cse-after-loop 133
freschedule-modulo-scheduled-loops 139
frounding-math 162
frtti 50
fsanitize-address-use-after-scope 196
fsanitize-coverage=trace-cmp 197
fsanitize-coverage=trace-pc 196
fsanitize-recover 196
fsanitize-sections 196
fsanitize-undefined-trap-on-error 196
fsanitize=address 192
fsanitize=alignment 194
fsanitize=bool 195
fsanitize=bounds 194
fsanitize=bounds-strict 194
fsanitize=builtin 195
fsanitize=enum 195
fsanitize=float-cast-overflow 195
fsanitize=float-divide-by-zero 194
fsanitize=integer-divide-by-zero 193
fsanitize=kernel-address 192
fsanitize=leak 193
fsanitize=nonnull-attribute 195
fsanitize=null 194
fsanitize=object-size 194
fsanitize=pointer-compare 192
fsanitize=pointer-overflow 195
fsanitize=pointer-subtract 192
fsanitize=return 194
fsanitize=returns-nonnull-attribute 195
fsanitize=shift 193
fsanitize=shift-base 193
fsanitize=shift-exponent 193
fsanitize=signed-integer-overflow 194
fsanitize=thread 193
fsanitize=undefined 193
fsanitize=unreachable 193
fsanitize=vla-bound 194
fsanitize=vptr 195

fsave-optimization-record 237
fsched-critical-path-heuristic 138
fsched-dep-count-heuristic 139
fsched-group-heuristic 138
fsched-interblock 137
fsched-last-insn-heuristic 138
fsched-pressure 137
fsched-rank-heuristic 138
fsched-spec 137
fsched-spec-insn-heuristic 138
fsched-spec-load 137
fsched-spec-load-dangerous 137
fsched-stalled-insns 137
fsched-stalled-insns-dep 138
fsched-verbose 238
fsched2-use-superblocks 138
fschedule-fusion 164
fschedule-insns 137
fschedule-insns2 137
fsection-anchors 166
fsel-sched-pipelining 139
fsel-sched-pipelining-outer-loops 139
fselective-scheduling 139
fselective-scheduling2 139
fsemantic-interposition 139
fset-stack-executable 426
fshort-enums 221, 446, 553, 880
fshort-wchar 221
fshow-column 67
fshrink-wrap 139
fshrink-wrap-separate 139
fsignaling-nans 162
fsigned-bitfields 44, 880
fsigned-char 44, 442
fsigned-zeros 162
fsimd-cost-model 148
fsingle-precision-constant 163
fsized-deallocation 50
fsplit-ivs-in-unroller 148
fsplit-loops 165
fsplit-paths 148
fsplit-stack 199, 489
fsplit-wide-types 132
fssa-backprop 144
fssa-phiopt 144
fsso-struct 44
fstack-check 198
fstack-clash-protection 198
fstack-limit-register 199
fstack-limit-symbol 199
fstack-protector 197
fstack-protector-all 197
fstack-protector-explicit 198
fstack-protector-strong 197
fstack-usage 242
fstack_reuse 218
fstats 243
fstdarg-opt 166
fstore-merging 147
fstrict-aliasing 151
fstrict-enums 50
fstrict-overflow 220
fstrict-volatile-bitfields 227
fstrong-eval-order 50
fsync-libcalls 228
fsyntax-only 70
ftabstop 205
ftemplate-backtrace-limit 50
ftemplate-depth 51
ftest-coverage 191
fthread-jumps 132
fthreadsafe-statics 51
ftime-report 242
ftime-report-details 242
ftls-model 226
ftoplevel-reorder 153
ftracer 164
ftrack-macro-expansion 205
ftrampolines 226
ftrapping-math 162
ftrapv 219
ftree-bit-ccp 143
ftree-builtin-call-dce 144
ftree-ccp 144
ftree-ch 144
ftree-coalesce-vars 145
ftree-copy-prop 141
ftree-dce 144
ftree-dominator-opts 144
ftree-dse 144
ftree-forwprop 140
ftree-fre 141
ftree-loop-distribute-patterns 146
ftree-loop-distribution 145
ftree-loop-if-convert 145
ftree-loop-im 147
ftree-loop-ivcanon 147
ftree-loop-linear 145
ftree-loop-optimize 145
ftree-loop-vectorize 148
ftree-parallelize-loops 147
ftree-partial-pre 140
ftree-phiprop 141
ftree-pre 140
ftree-pta 147
ftree-reassoc 140
ftree-scev-cprop 147
ftree-sink 143
ftree-slp-vectorize 148
ftree-slsr 148
ftree-sra 147
ftree-switch-conversion 144
ftree-tail-merge 144
ftree-ter 147
ftree-vectorize 148
ftree-vrp 148

funconstrained-commons 133
funit-at-a-time 153
funroll-all-loops 165
funroll-loops 164
funsafe-math-optimizations 161
funsigned-bitfields 44, 445, 880
funsigned-char 43, 442
funswitch-loops 165
funwind-tables 220
fuse-cxa-atexit 51
fuse-cxa-get-exception-ptr 51
fuse-ld=bfd 210
fuse-ld=gold 210
fuse-ld=lld 211
fuse-linker-plugin 157
fvar-tracking 121
fvar-tracking-assignments 121
fvar-tracking-assignments-toggle 242
fvariable-expansion-in-unroller 149
fvect-cost-model 148
fverbose-asm 222
fversion-loops-for-strides 165
fvisibility 226
fvisibility-inlines-hidden 51
fvisibility-ms-compat 51
fvpt 164
fvtable-verify 199
fvtv-counts 200
fvtv-debug 200
fweak 52
fweb 154
fwhole-program 154
fwide-exec-charset 206
fworking-directory 206
fwrapv 219
fwrapv-pointer 220
fwritable-relocated-rdata 426
fzero-initialized-in-bss 132
fzero-link 62
F 295

G

g 119
gas-loc-support 122
gas-locview-support 122
gcolumn-info 122
gdescribe-dies 121
gdwarf 119
gen-decls 62
gfull 295
ggdb 119
ggnu-pubnames 121
ginline-points 123
ginternal-reset-location-views 123
gno-as-loc-support 122
gno-column-info 122
gno-inline-points 123
gno-internal-reset-location-views 123
gno-record-gcc-switches 121
gno-statement-frontiers 122
gno-strict-dwarf 122
gno-variable-location-views 122
gpubnames 121
grecord-gcc-switches 121
gsplit-dwarf 121
gstabs 119
gstabs+ 119
gstatement-frontiers 122
gstrict-dwarf 122
gtoggle 241
gused 295
gvariable-location-views 122
gvariable-location-views=incompat5 122
gvms 120
gxcoff 119
gxcoff+ 119
gz 123
G 260, 316, 333, 346, 370, 395

H

headerpad_max_install_names 298
help 34
H 208

I

I- 216
idirafter 215
iframework 295
imacros 202
image_base 298
imultilib 216
include 202
init 298
install_name 298
iplugindir= 217
iprefix 216
iquote 215
isysroot 216
isystem 215
iwithprefix 216
iwithprefixbefore 216
I 215

K

keep_private_externs 298

L

l 211
lobjc 211
L 217

M

m ... 367
m1 ... 381
m10 ... 353
m128bit-long-double ... 410
m16 ... 424
m16-bit ... 291, 346
m1reg- ... 253
m2 ... 381
m210 ... 323
m2a ... 381
m2a-nofpu ... 381
m2a-single ... 381
m2a-single-only ... 381
m3 ... 381
m31 ... 378
m32 ... 351, 362, 392, 396, 424
m32-bit ... 291
m32bit-doubles ... 358, 373
m32r ... 315
m32r2 ... 315
m32rx ... 315
m340 ... 323
m3dnow ... 414
m3dnowa ... 414
m3e ... 381
m4 ... 381
m4-100 ... 381
m4-100-nofpu ... 381
m4-100-single ... 381
m4-100-single-only ... 381
m4-200 ... 381
m4-200-nofpu ... 381
m4-200-single ... 382
m4-200-single-only ... 382
m4-300 ... 382
m4-300-nofpu ... 382
m4-300-single ... 382
m4-300-single-only ... 382
m4-340 ... 382
m4-500 ... 382
m4-nofpu ... 381
m4-single ... 381
m4-single-only ... 381
m40 ... 353
m45 ... 353
m4a ... 382
m4a-nofpu ... 382
m4a-single ... 382
m4a-single-only ... 382
m4al ... 382
m4byte-functions ... 322
m5200 ... 319
m5206e ... 319
m528x ... 319
m5307 ... 319
m5407 ... 319
m64 ... 351, 362, 378, 392, 396, 424
m64bit-doubles ... 358, 373
m68000 ... 318
m68010 ... 318
m68020 ... 318
m68020-40 ... 319
m68020-60 ... 319
m68030 ... 318
m68040 ... 318
m68060 ... 319
m68881 ... 320
m8-bit ... 291
m8bit-idiv ... 423
m8byte-align ... 399
m96bit-long-double ... 410
mA6 ... 254
mA7 ... 254
mabi ... 245, 263, 354, 367, 418
mabi=32 ... 328
mabi=64 ... 328
mabi=eabi ... 328
mabi=elfv1 ... 368
mabi=elfv2 ... 368
mabi=gnu ... 341
mabi=ibmlongdouble ... 367
mabi=ieeelongdouble ... 367
mabi=mmixware ... 341
mabi=n32 ... 328
mabi=o64 ... 328
mabicalls ... 329
mabm ... 414
mabort-on-noreturn ... 274
mabs=2008 ... 331
mabs=legacy ... 331
mabsdata ... 280
mabsdiff ... 323
mac0 ... 353
macc-4 ... 305
macc-8 ... 305
maccumulate-args ... 280
maccumulate-outgoing-args ... 385, 419
maddress-mode=long ... 425
maddress-mode=short ... 425
maddress-space-conversion ... 394
mads ... 368
madx ... 414
maes ... 414
maix-struct-return ... 367
maix32 ... 363
maix64 ... 363
malign-300 ... 308
malign-call ... 260
malign-data ... 411
malign-double ... 410
malign-int ... 321
malign-labels ... 305
malign-loops ... 316
malign-natural ... 363
malign-power ... 363

malign-stringops 421
mall-opts 323
malloc-cc 303
mallow-string-insns 376
mallregs 358
maltivec 360
mam33 342
mam33-2 342
mam34 342
manchor 293
mandroid 307
mannotate-align 259
mapcs 263
mapcs-frame 263
mapp-regs 387, 399
mARC600 254
mARC601 254
mARC700 254
march .. 247, 253, 264, 289, 290, 308, 310, 317, 326, 346, 348, 351, 355, 379, 401
march= 292
marclinux 259
marclinux_prof 259
margonaut 262
marm 275
mas100-syntax 375
masm-hex 343
masm-syntax-unified 277
masm=*dialect* 409
matomic 256
matomic-model=*model* 383
matomic-updates 395
mauto-litpools 427
mauto-modify-reg 260
mauto-pic 312
maverage 323
mavoid-indexed-addresses 364
mavx 413
mavx2 413
mavx256-split-unaligned-load 423
mavx256-split-unaligned-store 423
mavx5124fmaps 415
mavx5124vnniw 415
mavx512bitalg 415
mavx512bw 414
mavx512cd 414
mavx512dq 414
mavx512er 414
mavx512f 414
mavx512ifma 414
mavx512pf 414
mavx512vbmi 414
mavx512vbmi2 414
mavx512vl 414
mavx512vnni 415
mavx512vpopcntdq 415
max-vect-align 253
mb 382
mbackchain 377
mbarrel-shift-enabled 314
mbarrel-shifter 254
mbarrel_shifter 263
mbase-addresses 342
mbased= 323
mbbit-peephole 260
mbe8 264
mbig 365
mbig-endian 245, 262, 264, 289, 292, 311, 322, 325, 345, 365, 396
mbig-endian-data 374
mbig-switch 399
mbigtable 382
mbionic 307
mbit-align 365
mbit-ops 292
mbitfield 320
mbitops 323, 383
mblock-compare-inline-limit 369
mblock-compare-inline-loop-limit 370
mblock-move-inline-limit 369
mbmi 414
mbmi2 414
mboard 352
mbranch-cost 251, 280, 339, 354
mbranch-cost= 294
mbranch-cost=*num* 386
mbranch-cost=*number* 316
mbranch-hints 394
mbranch-index 258
mbranch-likely 339
mbranch-predict 342
mbranch-protection 249
mbss-plt 361
mbuild-constants 300
mbwx 300
mbypass-cache 348
mc= 323
mc68000 318
mc68020 318
mcache 293
mcache-block-size 346
mcache-size 394
mcache-volatile 348
mcall-eabi 366
mcall-freebsd 367
mcall-linux 367
mcall-ms2sysv-xlogues 418
mcall-netbsd 367
mcall-prologues 280
mcall-sysv 366
mcall-sysv-eabi 366
mcall-sysv-noeabi 367
mcallee-super-interworking 275
mcaller-copies 308
mcaller-super-interworking 276
mcallgraph-data 322

mcase-vector-pcrel 260
mcbcond 391
mcbranch-force-delay-slot 386
mcc-init 290
mccrt 294
mcfv4e 319
mcheck-zero-division 335
mcix 300
mcld 416
mcldemote 415
mclear-hwcap 387
mclflushopt 414
mclip 323
mclwb 414
mclzero 414
mcmodel 346, 392
mcmodel=kernel 425
mcmodel=large 246, 360, 396, 425
mcmodel=medany 356
mcmodel=medium 360, 425
mcmodel=medlow 356
mcmodel=small 245, 360, 396, 425
mcmodel=tiny 245
mcmov 345, 352
mcmove 251
mcmpb 358
mcmse 277
mcode-density 256
mcode-density-frame 261
mcode-readable 334
mcode-region 345
mcompact-branches=always 339
mcompact-branches=never 339
mcompact-branches=optimal 339
mcompact-casesi 260
mcompat-align-parm 373
mcompress 303
mcond-exec 306
mcond-move 305
mconfig= 323
mconsole 425
mconst-align 291
mconst16 426
mconstant-gp 311
mconstpool 294
mcop 323
mcop32 323
mcop64 323
mcorea 289
mcoreb 289
mcp 293
mcpu... 248, 254, 272, 290, 301, 307, 317, 354, 357, 359, 374, 389, 396, 400, 408
mcpu= 286, 292, 315, 325, 344
mcpu32 319
mcr16c 291
mcr16cplus 291
mcrc 333
mcrc32 417
mcrypto 361
mcsync-anomaly 287
mctor-dtor 346
mcustom-fpu-cfg 350
mcustom-*insn* 348
mcx16 416
mdalign 382
mdata-align 291
mdata-model 292
mdata-region 345
mdc 324
mdebug 316, 379, 399
mdebug-main=*prefix* 400
mdec-asm 353
mdisable-callt 397
mdisable-fpregs 308
mdisable-indexing 308
mdiv 293, 320, 322, 324, 355
mdiv-rem 256
mdiv=*strategy* 385
mdivide-breaks 335
mdivide-enabled 315
mdivide-traps 335
mdivsi3_libfunc=*name* 386
mdll 425
mdlmzb 364
mdmx 332
mdouble 304
mdouble-float 292, 330
mdpfp 255
mdpfp-compact 255
mdpfp-fast 255
mdpfp_compact 263
mdpfp_fast 263
mdsp 293, 332
mdsp-packa 259
mdsp_packa 263
mdspr2 332
mdual-nops 395
mdump-tune-features 415
mdvbf 259
mdwarf2-asm 312
mdword 303, 304
mdynamic-no-pic 365
MD 204
mea 255
mea32 394
mea64 394
meabi 368
mearly-cbranchsi 261
mearly-stop-bits 312
mEA 263
meb 324, 343, 348, 380
medsp 293
mel 324, 343, 348, 380
melf 291, 342
melrw 293

memb ... 368
membedded-data ... 334
memregs= ... 315
mep ... 396
mepsilon ... 341
merror-reloc ... 393
mesa ... 378
metrax100 ... 290
metrax4 ... 290
meva ... 332
mexpand-adddi ... 261
mexplicit-relocs ... 301, 335
mexr ... 307
mext-perf ... 345
mext-perf2 ... 345
mext-string ... 345
mextern-sdata ... 334
mf16c ... 414
mfancy-math-387 ... 410
mfast-fp ... 288
mfast-indirect-calls ... 308
mfast-sw-div ... 348
mfaster-structs ... 389
mfdiv ... 355
mfdivdu ... 292
mfdpic ... 304
mfentry ... 422
mfentry-name ... 422
mfentry-section ... 422
mfix ... 300
mfix-24k ... 336
mfix-and-continue ... 296
mfix-at697f ... 392
mfix-cortex-a53-835769 ... 247
mfix-cortex-a53-843419 ... 247
mfix-cortex-m3-ldrd ... 276
mfix-gr712rc ... 392
mfix-r10000 ... 337
mfix-r4000 ... 336
mfix-r4400 ... 337
mfix-r5900 ... 337
mfix-rm7000 ... 337
mfix-sb1 ... 338
mfix-ut699 ... 392
mfix-ut700 ... 392
mfix-vr4120 ... 337
mfix-vr4130 ... 337
mfixed-cc ... 303
mfixed-range ... 309, 312, 386, 394
mflat ... 387
mflip-mips16 ... 328
mflip-thumb ... 275
mfloat-abi ... 264
mfloat-ieee ... 301
mfloat-vax ... 301
mfloat128 ... 362
mfloat128-hardware ... 362
mflush-func ... 339
mflush-func=*name* ... 317
mflush-trap=*number* ... 316
mfma ... 414
mfma4 ... 414
mfmaf ... 392
mfmovd ... 383
mforce-indirect-call ... 418
mforce-no-pic ... 427
mfp-exceptions ... 339
mfp-mode ... 252
mfp-reg ... 298
mfp-ret-in-387 ... 410
mfp-rounding-mode ... 299
mfp-trap-mode ... 299
mfp16-format ... 273
mfp32 ... 330
mfp64 ... 330
mfpmath ... 160, 408
mfpr-32 ... 303
mfpr-64 ... 303
mfprnd ... 358
mfpu ... 257, 273, 353, 388, 399
mfpu= ... 293
mfpxx ... 330
mfract-convert-truncate ... 282
mframe-header-opt ... 341
mfriz ... 372
mfsca ... 386
mfsgsbase ... 414
mfsmuld ... 392
mfsrra ... 386
mft32b ... 303
mfull-regs ... 345
mfull-toc ... 362
mfunction-return ... 424
mfused-madd ... 312, 336, 364, 379, 386, 426
mfxsr ... 414
MF ... 203
mg ... 399
mg10 ... 357
mg13 ... 357
mg14 ... 357
mgas ... 309
mgas-isr-prologues ... 280
mgcc-abi ... 398
mgeneral-regs-only ... 245, 264, 423
mgfni ... 414
mghs ... 398
mginv ... 333
mglibc ... 307
mgnu ... 399
mgnu-as ... 311
mgnu-asm ... 353
mgnu-attribute ... 368
mgnu-ld ... 309, 311
mgomp ... 352
mgotplt ... 291
mgp32 ... 330

mgp64 ... 330
mgpopt ... 334, 346
mgpr-32 ... 303
mgpr-64 ... 303
mgprel-ro ... 304
mgprel-sec ... 347
MG ... 203
mh ... 307
mhal ... 351
mhalf-reg-file ... 251
mhard-dfp ... 358, 377
mhard-div ... 352
mhard-float ... 292, 303, 320, 324, 330, 364, 376, 388, 398, 399, 409
mhard-mul ... 352
mhard-quad-float ... 388
mhardlit ... 322
mhigh-registers ... 293
mhint-max-distance ... 395
mhint-max-nops ... 395
mhle ... 414
mhotpatch ... 380
mhp-ld ... 309
mhtm ... 361, 378
mhw-div ... 348
mhw-mul ... 348
mhw-mulx ... 348
mhwmult= ... 344
miamcu ... 424
micplb ... 289
mid-shared-library ... 288
mieee ... 299, 383
mieee-conformant ... 300
mieee-fp ... 409
mieee-with-inexact ... 299
milp32 ... 313
mimadd ... 336
mimpure-text ... 387
mincoming-stack-boundary ... 413
mindexed-loads ... 261
mindirect-branch ... 423
mindirect-branch-register ... 424
minline-all-stringops ... 421
minline-float-divide-max-throughput ... 312
minline-float-divide-min-latency ... 312
minline-ic_invalidate ... 383
minline-int-divide ... 312
minline-int-divide-max-throughput ... 312
minline-int-divide-min-latency ... 312
minline-plt ... 289, 304
minline-sqrt-max-throughput ... 312
minline-sqrt-min-latency ... 312
minline-stringops-dynamically ... 421
minrt ... 344
minsert-sched-nops ... 366
minstrument-return ... 422
mint-register ... 375
mint16 ... 353
mint32 ... 292, 308, 353
mint8 ... 280
minterlink-compressed ... 328
minterlink-mips16 ... 328
mio-volatile ... 324
mips1 ... 327
mips16 ... 328
mips2 ... 327
mips3 ... 327
mips32 ... 327
mips32r3 ... 327
mips32r5 ... 327
mips32r6 ... 327
mips3d ... 332
mips4 ... 327
mips64 ... 328
mips64r2 ... 328
mips64r3 ... 328
mips64r5 ... 328
mips64r6 ... 328
mirq-ctrl-saved ... 258
misel ... 361
misize ... 259, 383
misr-vector-size ... 346
missue-rate=*number* ... 316
mistack ... 293
mivc2 ... 324
mjli-alawys ... 254
mjsr ... 376
mjump-in-delay ... 308
mkernel ... 295
mknuthdiv ... 341
ml ... 324, 382
mlarge ... 344
mlarge-data ... 301
mlarge-data-threshold ... 411
mlarge-mem ... 394
mlarge-text ... 301
mleadz ... 324
mleaf-id-shared-library ... 288
mlibfuncs ... 341
mlibrary-pic ... 305
mlinked-fp ... 305
mlinker-opt ... 309
mlinux ... 291
mlittle ... 365
mlittle-endian ... 245, 263, 264, 289, 292, 311, 322, 325, 345, 365, 396
mlittle-endian-data ... 374
mliw ... 343
mll64 ... 256
mllsc ... 331
mload-store-pairs ... 336
mlocal-sdata ... 333
mlock ... 259
mlong-calls ... 252, 259, 274, 288, 305, 310, 336, 396
mlong-double-128 ... 377, 411

mlong-double-64 ... 377, 411
mlong-double-80 ... 411
mlong-jump-table-offsets ... 322
mlong-jumps ... 398
mlong-load-store ... 309
mlong32 ... 333
mlong64 ... 333
mlongcall ... 370
mlongcalls ... 428
mloongson-ext ... 333
mloongson-ext2 ... 333
mloongson-mmi ... 333
mloop ... 398
mlow-precision-div ... 247
mlow-precision-recip-sqrt ... 247
mlow-precision-sqrt ... 247
mlow64k ... 287
mlp64 ... 313
mlpc-width ... 258
mlra ... 261, 303, 353, 389
mlra-priority-compact ... 261
mlra-priority-noncompact ... 261
mlra-priority-none ... 261
mlwp ... 414
mlxc1-sxc1 ... 341
mlzcnt ... 414
mm ... 324
mmac ... 291, 380
mmac-24 ... 259
mmac-d16 ... 259
mmac_24 ... 263
mmac_d16 ... 263
mmad ... 336
mmadd4 ... 341
mmain-is-OS_task ... 281
mmainkernel ... 352
mmalloc64 ... 400
mmanual-endbr ... 418
mmax ... 300
mmax-constant-size ... 375
mmax-stack-frame ... 290
mmcount-ra-address ... 340
mmcu ... 277, 332
mmcu= ... 343
MMD ... 204
mmedia ... 304
mmedium-calls ... 260
mmemcpy ... 325, 336
mmemcpy-strategy=*strategy* ... 421
mmemory-latency ... 302
mmemory-model ... 393
mmemset-strategy=*strategy* ... 421
mmfcrf ... 358
mmfpgpr ... 358
mmicromips ... 332
mmillicode ... 261
mminimal-toc ... 362
mminmax ... 324
mmixed-code ... 261
mmmx ... 413
mmodel=large ... 316
mmodel=medium ... 315
mmodel=small ... 315
mmovbe ... 417
mmovdir64b ... 415
mmovdiri ... 415
mmp ... 293
mmpy ... 255
mmpy-option ... 256
mms-bitfields ... 419
mmt ... 332
mmul ... 357
mmul-bug-workaround ... 290
mmul.x ... 343
mmul32x16 ... 255
mmul64 ... 255
mmuladd ... 304
mmulhw ... 364
mmult ... 324
mmult-bug ... 342
mmultcost ... 262
mmulti-cond-exec ... 306
mmulticore ... 289
mmultiple ... 364
mmultiple-stld ... 294
mmusl ... 307
mmvcle ... 379
mmvme ... 368
mmwaitx ... 414
MM ... 203
mn ... 307
mn-flash ... 281
mnan=2008 ... 331
mnan=legacy ... 331
mneon-for-64bits ... 276
mnested-cond-exec ... 306
mnewlib ... 352
mnhwloop ... 380
mno-16-bit ... 346
mno-4byte-functions ... 322
mno-8byte-align ... 399
mno-abicalls ... 329
mno-ac0 ... 353
mno-address-space-conversion ... 394
mno-align-double ... 410
mno-align-int ... 321
mno-align-loops ... 316
mno-align-stringops ... 421
mno-allow-string-insns ... 376
mno-altivec ... 360
mno-am33 ... 342
mno-app-regs ... 387, 399
mno-as100-syntax ... 375
mno-atomic-updates ... 395
mno-auto-litpools ... 427
mno-avoid-indexed-addresses ... 364

mno-backchain 377
mno-base-addresses 342
mno-bit-align 365
mno-bitfield 320
mno-branch-likely 339
mno-branch-predict 342
mno-brcc 260
mno-bwx 300
mno-bypass-cache 348
mno-cache-volatile 348
mno-call-ms2sysv-xlogues 418
mno-callgraph-data 322
mno-cbcond 391
mno-check-zero-division 335
mno-cix 300
mno-clearbss 325
mno-cmov 345
mno-cmpb 358
mno-cond-exec 260, 306
mno-cond-move 305
mno-const-align 291
mno-const16 426
mno-crc 333
mno-crt0 343
mno-crypto 361
mno-csync-anomaly 287
mno-custom-*insn* 348
mno-data-align 291
mno-debug 379
mno-default 416
mno-disable-callt 397
mno-div 320, 322
mno-dlmzb 364
mno-double 304
mno-dpfp-lrsr 255
mno-dsp 332
mno-dspr2 332
mno-dwarf2-asm 312
mno-dword 304
mno-eabi 368
mno-early-stop-bits 312
mno-eflags 305
mno-embedded-data 334
mno-ep 396
mno-epsilon 341
mno-eva 332
mno-explicit-relocs 301, 335
mno-exr 307
mno-ext-perf 345
mno-ext-perf2 345
mno-ext-string 346
mno-extern-sdata 334
mno-fancy-math-387 410
mno-fast-sw-div 348
mno-faster-structs 389
mno-fix 300
mno-fix-24k 336
mno-fix-cortex-a53-835769 247
mno-fix-cortex-a53-843419 247
mno-fix-r10000 337
mno-fix-r4000 336
mno-fix-r4400 337
mno-flat 387
mno-float 330
mno-float128 362
mno-float128-hardware 362
mno-flush-func 317
mno-flush-trap 316
mno-fmaf 392
mno-fp-in-toc 362
mno-fp-regs 298
mno-fp-ret-in-387 410
mno-fprnd 358
mno-fpu 388, 399
mno-fsca 386
mno-fsmuld 392
mno-fsrra 386
mno-fused-madd 312, 336, 364, 379, 386, 426
mno-ginv 333
mno-gnu-as 311
mno-gnu-attribute 368
mno-gnu-ld 311
mno-gotplt 291
mno-gpopt 334, 346
mno-hard-dfp 358, 377
mno-hardlit 322
mno-htm 361, 378
mno-hw-div 348
mno-hw-mul 348
mno-hw-mulx 348
mno-id-shared-library 288
mno-ieee 383
mno-ieee-fp 409
mno-imadd 336
mno-inline-float-divide 312
mno-inline-int-divide 312
mno-inline-sqrt 312
mno-int16 353
mno-int32 353
mno-interlink-compressed 328
mno-interlink-mips16 328
mno-interrupts 281
mno-isel 361
mno-jsr 376
mno-knuthdiv 341
mno-leaf-id-shared-library 288
mno-libfuncs 341
mno-liw 343
mno-llsc 331
mno-load-store-pairs 336
mno-local-sdata 333
mno-long-calls 274, 288, 310, 336, 396
mno-long-jumps 398
mno-longcall 370
mno-longcalls 428
mno-loongson-ext 333

mno-loongson-ext2 333
mno-loongson-mmi 333
mno-low-precision-div 247
mno-low-precision-recip-sqrt 247
mno-low-precision-sqrt 247
mno-low64k 287
mno-lra 389
mno-lsim 303, 323
mno-mad 336
mno-max 300
mno-mcount-ra-address 340
mno-mcu 332
mno-mdmx 332
mno-media 304
mno-memcpy 336
mno-mfcrf 358
mno-mfpgpr 358
mno-mips16 328
mno-mips3d 332
mno-mmicromips 332
mno-mpy 255
mno-ms-bitfields 419
mno-mt 332
mno-mul-bug-workaround 290
mno-muladd 304
mno-mulhw 364
mno-mult-bug 342
mno-multi-cond-exec 306
mno-multiple 364
mno-mvcle 379
mno-nested-cond-exec 306
mno-odd-spreg 331
mno-omit-leaf-frame-pointer 246
mno-optimize-membar 306
mno-opts 324
mno-pack 305
mno-packed-stack 377
mno-paired-single 332
mno-pc-relative-literal-loads 249
mno-pic 311
mno-pid 375
mno-plt 329
mno-pltseq 371
mno-popc 392
mno-popcntb 358
mno-popcntd 358
mno-postinc 253
mno-postmodify 253
mno-power8-fusion 361
mno-power8-vector 361
mno-powerpc-gfxopt 358
mno-powerpc-gpopt 358
mno-powerpc64 358
mno-prolog-function 396
mno-prologue-epilogue 291
mno-prototype 368
mno-push-args 418
mno-quad-memory 361
mno-quad-memory-atomic 361
mno-readonly-in-sdata 369
mno-red-zone 424
mno-register-names 311
mno-regnames 370
mno-relax 398
mno-relax-immediate 322
mno-relocatable 365
mno-relocatable-lib 365
mno-renesas 383
mno-round-nearest 252
mno-save-mduc-in-interrupts 358
mno-scc 305
mno-sched-ar-data-spec 313
mno-sched-ar-in-data-spec 313
mno-sched-br-data-spec 313
mno-sched-br-in-data-spec 313
mno-sched-control-spec 313
mno-sched-count-spec-in-critical-path 314
mno-sched-in-control-spec 313
mno-sched-prefer-non-control-spec-insns 314
mno-sched-prefer-non-data-spec-insns 314
mno-sched-prolog 264
mno-sdata 260, 311, 369
mno-sep-data 288
mno-serialize-volatile 427
mno-setlb 343
mno-short 320
mno-side-effects 290
mno-sim 374
mno-single-exit 342
mno-slow-bytes 322
mno-small-exec 378
mno-smartmips 332
mno-soft-cmpsf 251
mno-soft-float 298
mno-space-regs 308
mno-specld-anomaly 287
mno-split-addresses 335
mno-split-lohi 253
mno-stack-align 291
mno-stack-bias 393
mno-std-struct-return 389
mno-strict-align 246, 321, 365
mno-subxc 392
mno-sum-in-toc 362
mno-sym32 333
mno-target-align 427
mno-text-section-literals 427
mno-tls-markers 371
mno-toc 365
mno-toplevel-symbols 342
mno-tpf-trace 379
mno-unaligned-access 276
mno-unaligned-doubles 388
mno-uninit-const-in-rodata 334
mno-update 364

mno-user-mode ... 389
mno-usermode ... 385
mno-v3push ... 346
mno-v8plus ... 391
mno-vect-double ... 253
mno-virt ... 332
mno-vis ... 391
mno-vis2 ... 391
mno-vis3 ... 391
mno-vis4 ... 391
mno-vis4b ... 391
mno-vliw-branch ... 306
mno-volatile-asm-stop ... 311
mno-volatile-cache ... 260
mno-vrsave ... 360
mno-vsx ... 361
mno-vx ... 378
mno-warn-mcu ... 344
mno-warn-multiple-fast-interrupts ... 376
mno-wide-bitfields ... 322
mno-xgot ... 321, 330
mno-xl-compat ... 363
mno-xpa ... 332
mno-zdcbranch ... 386
mno-zero-extend ... 341
mno-zvector ... 378
mnobitfield ... 320
mnodiv ... 303
mnomacsave ... 383
mnop-fun-dllimport ... 425
mnop-mcount ... 422
mnopm ... 303
mnops ... 251
mnorm ... 255
modd-spreg ... 331
momit-leaf-frame-pointer ... 246, 287, 421
mone-byte-bool ... 296
moptimize ... 352
moptimize-membar ... 306
moverride ... 249
mpa-risc-1-0 ... 308
mpa-risc-1-1 ... 308
mpa-risc-2-0 ... 308
mpack ... 305
mpacked-stack ... 377
mpadstruct ... 383
mpaired-single ... 332
mpc-relative-literal-loads ... 249
mpc32 ... 412
mpc64 ... 412
mpc80 ... 412
mpclmul ... 414
mpconfig ... 414
mpcrel ... 321
mpdebug ... 290
mpe ... 363
mpe-aligned-commons ... 426
mpic-data-is-text-relative ... 274, 326
mpic-register ... 274
mpid ... 375
mpku ... 414
mplt ... 329
mpltseq ... 371
mpointer-size=*size* ... 400
mpointers-to-nested-functions ... 372
mpoke-function-name ... 275
mpopc ... 392
mpopcnt ... 414
mpopcntb ... 358
mpopcntd ... 358
mportable-runtime ... 309
mpostinc ... 253
mpostmodify ... 253
mpower8-fusion ... 361
mpower8-vector ... 361
mpowerpc-gfxopt ... 358
mpowerpc-gpopt ... 358
mpowerpc64 ... 358
mprefer-avx128 ... 416
mprefer-short-insn-regs ... 251
mprefer-vector-width ... 416
mprefergot ... 385
mpreferred-stack-boundary ... 355, 413
mprefetchwt1 ... 414
mpretend-cmove ... 387
mprfchw ... 414
mprint-tune-info ... 277
mprioritize-restricted-insns ... 366
mprolog-function ... 396
mprologue-epilogue ... 291
mprototype ... 368
mptwrite ... 414
mpure-code ... 277
mpush-args ... 418
mpushpop ... 293
MP ... 204
mq-class ... 261
mquad-memory ... 361
mquad-memory-atomic ... 361
MQ ... 204
mr0rel-sec ... 347
mr10k-cache-barrier ... 338
mRcq ... 261
mRcw ... 261
mrdpid ... 414
mrdrnd ... 414
mrdseed ... 414
mreadonly-in-sdata ... 369
mrecip ... 371, 417
mrecip-precision ... 372
mrecip=opt ... 371, 417
mrecord-mcount ... 422
mrecord-return ... 422
mred-zone ... 424
mreduced-regs ... 345
mregister-names ... 311

mregnames ... 370
mregparm ... 412
mrelax ... 281, 307, 343, 344, 346, 375, 382, 398
mrelax-immediate ... 322
mrelax-pic-calls ... 340
mrelocatable ... 365
mrelocatable-lib ... 365
mrenesas ... 383
mrepeat ... 324
mrestrict-it ... 277
mreturn-pointer-on-d0 ... 343
mrf16 ... 258
mrgf-banked-regs ... 258
mrh850-abi ... 398
mrl78 ... 357
mrmw ... 281
mror ... 353
mround-nearest ... 252
mrtd ... 320, 411, 525
mrtm ... 414
mrtp ... 400
mrtsc ... 259
ms ... 307, 324
ms2600 ... 307
msafe-dma ... 394
msafe-hints ... 395
msahf ... 416
msatur ... 324
msave-acc-in-interrupts ... 375
msave-mduc-in-interrupts ... 358
msave-restore ... 356
msave-toc-indirect ... 373
mscc ... 305
msched-ar-data-spec ... 313
msched-ar-in-data-spec ... 313
msched-br-data-spec ... 313
msched-br-in-data-spec ... 313
msched-control-spec ... 313
msched-costly-dep ... 366
msched-count-spec-in-critical-path ... 314
msched-fp-mem-deps-zero-cost ... 314
msched-in-control-spec ... 313
msched-max-memory-insns ... 314
msched-max-memory-insns-hard-limit ... 314
msched-prefer-non-control-spec-insns ... 314
msched-prefer-non-data-spec-insns ... 314
msched-prolog ... 264, 294
msched-spec-ldc ... 314
msched-stop-bits-after-every-cycle ... 314
mschedule ... 309
mscore5 ... 380
mscore5u ... 380
mscore7 ... 380
mscore7d ... 380
msda ... 397
msdata ... 260, 311, 369
msdata=all ... 290
msdata=data ... 369
msdata=default ... 289, 369
msdata=eabi ... 369
msdata=none ... 290, 316, 369
msdata=sdata ... 316
msdata=sysv ... 369
msdata=use ... 316
msdram ... 289, 324
msecure-plt ... 361
msecurity ... 293
msel-sched-dont-check-control-spec ... 314
msep-data ... 288
mserialize-volatile ... 427
msetlb ... 343
msext ... 353
msfimm ... 353
msgx ... 414
msha ... 414
mshared-library-id ... 288
mshftimm ... 353
mshort ... 320
mshort-calls ... 281
mshstk ... 417
mside-effects ... 290
msign-extend-enabled ... 315
msign-return-address ... 249
msilicon-errata ... 345
msilicon-errata-warn ... 345
msim ... 287, 289, 292, 303, 315, 324, 344, 356, 368, 374, 399, 426
msimd ... 255
msimnovec ... 324
msingle-exit ... 342
msingle-float ... 330
msingle-pic-base ... 274, 366
msio ... 309
msize-level ... 262
mskip-rax-setup ... 423
mslow-bytes ... 322
mslow-flash-data ... 277
msmall ... 344
msmall-data ... 301
msmall-data-limit ... 356, 374
msmall-divides ... 325
msmall-exec ... 378
msmall-mem ... 394
msmall-model ... 302
msmall-text ... 301
msmall16 ... 252
msmallc ... 351
msmart ... 293
msmartmips ... 332
msoft-cmpsf ... 251
msoft-div ... 353
msoft-float ... 255, 292, 298, 303, 309, 320, 324, 330, 353, 364, 376, 388, 398, 399, 409
msoft-mul ... 353
msoft-quad-float ... 388
msoft-stack ... 352

msp8 ... 281
mspace ... 397
mspace-regs ... 308
mspecld-anomaly ... 287
mspfp ... 255
mspfp-compact ... 255
mspfp-fast ... 255
mspfp_compact ... 263
mspfp_fast ... 263
msplit ... 353
msplit-addresses ... 335
msplit-lohi ... 253
msplit-vecmove-early ... 253
msse ... 413
msse2 ... 413
msse2avx ... 422
msse3 ... 413
msse4 ... 413
msse4.1 ... 413
msse4.2 ... 413
msse4a ... 413
msseregparm ... 412
mssse3 ... 413
mstack-align ... 291
mstack-bias ... 393
mstack-check-l1 ... 288
mstack-guard ... 380
mstack-increment ... 323
mstack-offset ... 252
mstack-protector-guard ... 246, 373, 423
mstack-protector-guard-offset ... 246, 373, 423
mstack-protector-guard-reg ... 246, 373, 423
mstack-protector-guard-symbol ... 373
mstack-size ... 253, 294, 380
mstackrealign ... 412
mstd-struct-return ... 389
mstdmain ... 394
mstrict-align ... 246, 321, 356, 365
mstrict-X ... 281
mstring-compare-inline-limit ... 370
mstringop-strategy=*alg* ... 421
mstructure-size-boundary ... 273
msubxc ... 392
msv-mode ... 400
msve-vector-bits ... 249
msvr4-struct-return ... 367
mswap ... 256
mswape ... 259
msym32 ... 333
msynci ... 340
msys-crt0 ... 351
msys-lib ... 351
mtarget-align ... 427
mtas ... 384
mtbm ... 414
mtda ... 397
mtelephony ... 259
mtext-section-literals ... 427
mtf ... 324
mthread ... 425
mthreads ... 419
mthumb ... 275
mthumb-interwork ... 263
mtiny-stack ... 282
mtiny= ... 324
mtls ... 304
mtls-dialect ... 276, 418
mtls-dialect=desc ... 246
mtls-dialect=traditional ... 246
mtls-direct-seg-refs ... 422
mtls-markers ... 371
mtls-size ... 246, 313
mTLS ... 304
mtoc ... 365
mtomcat-stats ... 306
mtoplevel-symbols ... 342
mtp ... 276
mtp-regno ... 256
mtpcs-frame ... 275
mtpcs-leaf-frame ... 275
mtpf-trace ... 379
mtraceback ... 367
mtrap-precision ... 300
mtrust ... 293
mtune .. 248, 253, 262, 263, 271, 290, 302, 313, 318, 327, 342, 355, 360, 379, 390, 400, 407
mtune-ctrl=*feature-list* ... 415
MT ... 204
muclibc ... 307
muls ... 380
multcost ... 263
multcost=*number* ... 385
multi_module ... 298
multilib-library-pic ... 304
multiply-enabled ... 315
multiply_defined ... 298
multiply_defined_unused ... 298
munalign-prob-threshold ... 262
munaligned-access ... 276
munaligned-doubles ... 388
municode ... 425
muniform-simt ... 352
muninit-const-in-rodata ... 334
munix ... 399
munix-asm ... 353
munsafe-dma ... 394
mupdate ... 364
muser-enabled ... 315
muser-mode ... 389, 400
musermode ... 385
mv3push ... 346
mv850 ... 397
mv850e ... 397
mv850e1 ... 397
mv850e2 ... 397
mv850e2v3 ... 397

mv850e2v4 ... 397
mv850e3v5 ... 397
mv850es ... 397
mv8plus ... 391
mvaes ... 414
mvdsp ... 293
mveclibabi ... 372, 417
mvect-double ... 253
mvect8-ret-in-mem ... 412
mverbose-cost-dump ... 249, 277
mvirt ... 332
mvis ... 391
mvis2 ... 391
mvis3 ... 391
mvis4 ... 391
mvis4b ... 391
mvliw-branch ... 306
mvms-return-codes ... 400
mvolatile-asm-stop ... 311
mvolatile-cache ... 260
mvpclmulqdq ... 415
mvr4130-align ... 340
mvrsave ... 360
mvsx ... 361
mvx ... 378
mvxworks ... 368
mvzeroupper ... 416
mwaitpkg ... 415
mwarn-dynamicstack ... 379
mwarn-framesize ... 379
mwarn-mcu ... 344
mwarn-multiple-fast-interrupts ... 376
mwarn-reloc ... 393
mwbnoinvd ... 414
mwide-bitfields ... 322
mwin32 ... 426
mwindows ... 426
mword-relocations ... 276
mx32 ... 424
mxgot ... 321, 330
mxl-barrel-shift ... 325
mxl-compat ... 363
mxl-float-convert ... 325
mxl-float-sqrt ... 325
mxl-gp-opt ... 325
mxl-multiply-high ... 325
mxl-pattern-compare ... 325
mxl-reorder ... 325
mxl-soft-div ... 325
mxl-soft-mul ... 325
mxl-stack-check ... 325
mxop ... 414
mxpa ... 332
mxsave ... 414
mxsavec ... 414
mxsaveopt ... 414
mxsaves ... 414
mxy ... 259
myellowknife ... 368
mzarch ... 378
mzda ... 397
mzdcbranch ... 386
mzero-extend ... 341
mzvector ... 378
M ... 203

N

no-80387 ... 409
no-canonical-prefixes ... 217
no-integrated-cpp ... 209
no-pie ... 212
no-sysroot-suffix ... 218
no_dead_strip_inits_and_terms ... 298
noall_load ... 298
nocpp ... 336
nodefaultlibs ... 211
nodevicelib ... 282
nofixprebinding ... 298
nofpu ... 374
nolibc ... 211
nolibdld ... 311
nomultidefs ... 298
non-static ... 401
noprebind ... 298
noseglinkedit ... 298
nostartfiles ... 211
nostdinc ... 216
nostdinc++ ... 52, 217
nostdlib ... 212

O

o ... 33
O ... 125
O0 ... 127
O1 ... 125
O2 ... 126
O3 ... 127
Ofast ... 128
Og ... 128
Os ... 127

P

p ... 189, 488
pagezero_size ... 298
param ... 167
pass-exit-codes ... 35
pedantic ... 5, 71, 451, 624, 883
pedantic-errors ... 5, 72, 883
pg ... 189, 488
pie ... 212
pipe ... 36
plt ... 354
prebind ... 298

prebind_all_twolevel_modules 298
print-file-name 243
print-libgcc-file-name 244
print-multi-directory 243
print-multi-lib 243
print-multi-os-directory 243
print-multiarch 244
print-objc-runtime-info 63
print-prog-name 244
print-search-dirs 244
print-sysroot 244
print-sysroot-headers-suffix 244
private_bundle 298
pthread 203, 212
pthreads 387
P 207

Q

Q 242
Qn 395
Qy 395

R

r 212
rdynamic 212
read_only_relocs 298
remap 208

S

s 213
save-temps 240
save-temps=obj 240
sectalign 298
sectcreate 298
sectobjectsymbols 298
sectorder 298
seg_addr_table 298
seg_addr_table_filename 298
seg1addr 298
segaddr 298
seglinkedit 298
segprot 298
segs_read_only_addr 298
segs_read_write_addr 298
shared 213
shared-libgcc 213
short-calls 252
sim 291
sim2 291
single_module 298
specs 36
static 213, 298, 311
static-libasan 213
static-libgcc 213
static-liblsan 214
static-libstdc++ 214
static-libtsan 214
static-libubsan 214
static-pie 212
std 5, 38, 641, 881
sub_library 298
sub_umbrella 298
symbolic 214
sysroot 217
S 33, 210

T

target-help 34
threads 311
time 240
tno-android-cc 307
tno-android-ld 307
traditional 207, 871
traditional-cpp 207
trigraphs 207
twolevel_namespace 298
T 214

U

u 215
umbrella 298
undef 203
undefined 298
unexported_symbols_list 298
U 202

V

v 34
version 35

W

w 70
W 73, 112, 114, 872
Wa 209
Wabi 52
Wabi-tag 54
Wabsolute-value 104
Waddr-space-convert 282
Waddress 110
Waddress-of-packed-member 110
Waggregate-return 111
Waggressive-loop-optimizations 111
Waligned-new 103
Wall 72, 874
Walloc-size-larger-than= 95
Walloc-zero 95
Walloca 95
Walloca-larger-than= 95
Warray-bounds 96

Wassign-intercept 62
Wattribute-warning 113
Wattributes 111
Wbad-function-cast 105
Wbool-compare 97
Wbool-operation 97
Wbuiltin-declaration-mismatch 111
Wbuiltin-macro-redefined 111
Wc++-compat 105
Wc++11-compat 106
Wc++14-compat 106
Wc++17-compat 106
Wc90-c99-compat 105
Wc99-c11-compat 105
Wcast-align 106
Wcast-align=strict 106
Wcast-function-type 106
Wcast-qual 106
Wcatch-value 107
Wchar-subscripts 74
Wclass-conversion 60
Wclass-memaccess 56
Wclobbered 107
Wcomment 104
Wcomments 104
Wconditionally-supported 107
Wconversion 107
Wconversion-null 107
Wcoverage-mismatch 74
Wctor-dtor-privacy 54
Wdangling-else 108
Wdate-time 108
Wdeclaration-after-statement 101
Wdelete-incomplete 108
Wdelete-non-virtual-dtor 54
Wdeprecated 114
Wdeprecated-copy 54
Wdeprecated-declarations 114
Wdisabled-optimization 117
Wdiscarded-array-qualifiers 98
Wdiscarded-qualifiers 98
Wdiv-by-zero 98
Wdouble-promotion 74
Wduplicate-decl-specifier 74
Wduplicated-branches 98
Wduplicated-cond 98
weak_reference_mismatches 298
Weffc++ 58
Wempty-body 108
Wendif-labels 105
Wenum-compare 108
Werror 70
Werror= 70
Wexpansion-to-defined 105
Wextra 73, 112, 114
Wextra-semi 108
Wfatal-errors 71
Wfloat-conversion 109
Wfloat-equal 99
Wformat 74, 75, 94, 483
Wformat-contains-nul 75
Wformat-extra-args 75
Wformat-nonliteral 77, 484
Wformat-overflow 75, 76
Wformat-security 77
Wformat-signedness 77
Wformat-truncation 78
Wformat-y2k 78
Wformat-zero-length 77
Wformat= 74
Wformat=1 75
Wformat=2 77
Wframe-address 98
Wframe-larger-than= 102
Wfree-nonheap-object 102
whatsloaded 298
whyload 298
Wif-not-aligned 81
Wignored-attributes 81
Wignored-qualifiers 81
Wimplicit 79
Wimplicit-fallthrough 79
Wimplicit-fallthrough= 79
Wimplicit-function-declaration 79
Wimplicit-int 79
Wincompatible-pointer-types 98
Winherited-variadic-ctor 115
Winit-list-lifetime 54
Winit-self 79
Winline 116, 565
Wint-conversion 98
Wint-in-bool-context 116
Wint-to-pointer-cast 116
Winvalid-memory-model 89
Winvalid-offsetof 116
Winvalid-pch 116
Wjump-misses-init 109
Wl 215
Wlarger-than-*byte-size* 102
Wlarger-than= 102
Wliteral-suffix 55
Wlogical-not-parentheses 111
Wlogical-op 110
Wlong-long 116
Wlto-type-mismatch 55
Wmain 81
Wmaybe-uninitialized 89
Wmemset-elt-size 110
Wmemset-transposed-args 110
Wmisleading-indentation 81
Wmissing-attributes 82
Wmissing-braces 83
Wmissing-declarations 112
Wmissing-field-initializers 112
Wmissing-format-attribute 94
Wmissing-include-dirs 83

Wmissing-noreturn 94
Wmissing-parameter-type 112
Wmissing-profile 83
Wmissing-prototypes 112
Wmisspelled-isr 282
Wmultichar 113
Wmultiple-inheritance 59
Wmultistatement-macros 83
Wnamespaces 59
Wnarrowing 55
Wnested-externs 115
Wno-abi 52
Wno-absolute-value 104
Wno-addr-space-convert 282
Wno-address 110
Wno-address-of-packed-member 110
Wno-aggregate-return 111
Wno-aggressive-loop-optimizations 111
Wno-aligned-new 103
Wno-all 72
Wno-alloc-size-larger-than 95
Wno-alloc-zero 95
Wno-alloca 95
Wno-alloca-larger-than 95, 96
Wno-array-bounds 96
Wno-assign-intercept 62
Wno-attribute-warning 113
Wno-attributes 111
Wno-bad-function-cast 105
Wno-bool-compare 97
Wno-bool-operation 97
Wno-builtin-declaration-mismatch 111
Wno-builtin-macro-redefined 111
Wno-c++-compat 105
Wno-c++11-compat 106
Wno-c++14-compat 106
Wno-c++17-compat 106
Wno-c90-c99-compat 105
Wno-c99-c11-compat 105
Wno-cast-align 106
Wno-cast-function-type 106
Wno-cast-qual 106
Wno-catch-value 107
Wno-char-subscripts 74
Wno-class-conversion 60
Wno-class-memaccess 56
Wno-clobbered 107
Wno-conditionally-supported 107
Wno-conversion 107
Wno-conversion-null 107
Wno-coverage-mismatch 74
Wno-ctor-dtor-privacy 54
Wno-dangling-else 108
Wno-date-time 108
Wno-declaration-after-statement 101
Wno-delete-incomplete 108
Wno-delete-non-virtual-dtor 54
Wno-deprecated 114
Wno-deprecated-copy 54
Wno-deprecated-declarations 114
Wno-disabled-optimization 117
Wno-discarded-array-qualifiers 98
Wno-discarded-qualifiers 98
Wno-div-by-zero 98
Wno-double-promotion 74
Wno-duplicate-decl-specifier 74
Wno-duplicated-branches 98
Wno-duplicated-cond 98
Wno-effc++ 58
Wno-empty-body 108
Wno-endif-labels 105
Wno-enum-compare 108
Wno-error 70
Wno-error= 70
Wno-extra 73, 112, 114
Wno-extra-semi 108
Wno-fatal-errors 71
Wno-float-conversion 109
Wno-float-equal 99
Wno-format 74, 94
Wno-format-contains-nul 75
Wno-format-extra-args 75
Wno-format-nonliteral 77
Wno-format-overflow 75, 76
Wno-format-security 77
Wno-format-signedness 77
Wno-format-truncation 78
Wno-format-y2k 78
Wno-format-zero-length 77
Wno-frame-address 98
Wno-frame-larger-than 102
Wno-free-nonheap-object 102
Wno-if-not-aligned 81
Wno-ignored-attributes 81
Wno-ignored-qualifiers 81
Wno-implicit 79
Wno-implicit-fallthrough 79
Wno-implicit-function-declaration 79
Wno-implicit-int 79
Wno-incompatible-pointer-types 98
Wno-inherited-variadic-ctor 115
Wno-init-list-lifetime 54
Wno-init-self 79
Wno-inline 116
Wno-int-conversion 98
Wno-int-in-bool-context 116
Wno-int-to-pointer-cast 116
Wno-invalid-memory-model 89
Wno-invalid-offsetof 116
Wno-invalid-pch 116
Wno-jump-misses-init 109
Wno-larger-than 102
Wno-literal-suffix 55
Wno-logical-not-parentheses 111
Wno-logical-op 110
Wno-long-long 116

Wno-lto-type-mismatch 55
Wno-main 81
Wno-maybe-uninitialized 89
Wno-memset-elt-size 110
Wno-memset-transposed-args 110
Wno-misleading-indentation 81
Wno-missing-attributes 82
Wno-missing-braces 83
Wno-missing-declarations 112
Wno-missing-field-initializers 112
Wno-missing-format-attribute 94
Wno-missing-include-dirs 83
Wno-missing-noreturn 94
Wno-missing-parameter-type 112
Wno-missing-profile 83
Wno-missing-prototypes 112
Wno-misspelled-isr 282
Wno-multichar 113
Wno-multiple-inheritance 59
Wno-multistatement-macros 83
Wno-namespaces 59
Wno-narrowing 55
Wno-nested-externs 115
Wno-noexcept 55
Wno-noexcept-type 56
Wno-non-template-friend 58
Wno-non-virtual-dtor 56
Wno-nonnull 78
Wno-nonnull-compare 78
Wno-normalized 113
Wno-null-dereference 78
Wno-odr 114
Wno-old-style-cast 58
Wno-old-style-declaration 112
Wno-old-style-definition 112
Wno-openmp-simd 114
Wno-overflow 114
Wno-overlength-strings 118
Wno-overloaded-virtual 59
Wno-override-init 114
Wno-override-init-side-effects 114
Wno-packed 114
Wno-packed-bitfield-compat 114
Wno-packed-not-aligned 115
Wno-padded 115
Wno-parentheses 83
Wno-pedantic 71
Wno-pedantic-ms-format 103
Wno-pessimizing-move 57
Wno-placement-new 103
Wno-pmf-conversions 59, 821
Wno-pointer-arith 104
Wno-pointer-compare 104
Wno-pointer-sign 118
Wno-pointer-to-int-cast 116
Wno-pragmas 90
Wno-prio-ctor-dtor 90
Wno-protocol 63
Wno-redundant-decls 115
Wno-redundant-move 57
Wno-register 56
Wno-reorder 56
Wno-restrict 115
Wno-return-local-addr 85
Wno-return-type 85
Wno-scalar-storage-order 109
Wno-selector 63
Wno-sequence-point 84
Wno-shadow 101
Wno-shadow-ivar 101
Wno-shift-count-negative 85
Wno-shift-count-overflow 85
Wno-shift-negative-value 85
Wno-shift-overflow 85
Wno-sign-compare 109
Wno-sign-conversion 109
Wno-sign-promo 59
Wno-sized-deallocation 109
Wno-sizeof-array-argument 110
Wno-sizeof-pointer-div 109
Wno-sizeof-pointer-memaccess 109
Wno-stack-protector 118
Wno-stack-usage 102, 103
Wno-strict-aliasing 90
Wno-strict-null-sentinel 58
Wno-strict-overflow 91
Wno-strict-prototypes 112
Wno-strict-selector-match 63
Wno-stringop-overflow 91, 92
Wno-stringop-truncation 93
Wno-subobject-linkage 107
Wno-suggest-attribute= 94
Wno-suggest-attribute=cold 94
Wno-suggest-attribute=const 94
Wno-suggest-attribute=format 94
Wno-suggest-attribute=malloc 94
Wno-suggest-attribute=noreturn 94
Wno-suggest-attribute=pure 94
Wno-suggest-final-methods 95
Wno-suggest-final-types 95
Wno-switch 86
Wno-switch-bool 86
Wno-switch-default 86
Wno-switch-enum 86
Wno-switch-unreachable 86
Wno-sync-nand 87
Wno-system-headers 99
Wno-tautological-compare 99
Wno-templates 59
Wno-terminate 59
Wno-traditional 99
Wno-traditional-conversion 101
Wno-trampolines 99
Wno-type-limits 104
Wno-undeclared-selector 63
Wno-undef 104

Wno-uninitialized 88
Wno-unknown-pragmas 90
Wno-unsafe-loop-optimizations 103
Wno-unsuffixed-float-constants 118
Wno-unused 88
Wno-unused-but-set-parameter 87
Wno-unused-but-set-variable 87
Wno-unused-const-variable 88
Wno-unused-function 87
Wno-unused-label 87
Wno-unused-local-typedefs 87
Wno-unused-parameter 87
Wno-unused-result 87
Wno-unused-value 88
Wno-unused-variable 87
Wno-useless-cast 108
Wno-varargs 117
Wno-variadic-macros 116
Wno-vector-operation-performance 117
Wno-virtual-inheritance 59
Wno-virtual-move-assign 117
Wno-vla 117
Wno-vla-larger-than 117
Wno-volatile-register-var 117
Wno-write-strings 106
Wno-zero-as-null-pointer-constant 107
Wnoexcept 55
Wnoexcept-type 56
Wnon-template-friend 58
Wnon-virtual-dtor 56
Wnonnull 78
Wnonnull-compare 78
Wnormalized 113
Wnormalized= 113
Wnull-dereference 78
Wodr 114
Wold-style-cast 58
Wold-style-declaration 112
Wold-style-definition 112
Wopenmp-simd 114
Woverflow 114
Woverlength-strings 118
Woverloaded-virtual 59
Woverride-init 114
Woverride-init-side-effects 114
Wp 209
Wpacked 114
Wpacked-bitfield-compat 114
Wpacked-not-aligned 115
Wpadded 115
Wparentheses 83
Wpedantic 71
Wpedantic-ms-format 103
Wpessimizing-move 57
Wplacement-new 103
Wpmf-conversions 59
Wpointer-arith 104, 472
Wpointer-compare 104
Wpointer-sign 118
Wpointer-to-int-cast 116
Wpragmas 90
Wprio-ctor-dtor 90
Wprotocol 63
wrapper 36
Wredundant-decls 115
Wredundant-move 57
Wregister 56
Wreorder 56
Wrestrict 115
Wreturn-local-addr 85
Wreturn-type 85
Wscalar-storage-order 109
Wselector 63
Wsequence-point 84
Wshadow 101
Wshadow-ivar 101
Wshadow=compatible-local 101
Wshadow=local 101
Wshift-count-negative 85
Wshift-count-overflow 85
Wshift-negative-value 85
Wshift-overflow 85
Wsign-compare 109
Wsign-conversion 109
Wsign-promo 59
Wsized-deallocation 109
Wsizeof-array-argument 110
Wsizeof-pointer-div 109
Wsizeof-pointer-memaccess 109
Wstack-protector 118
Wstack-usage 102
Wstrict-aliasing 90
Wstrict-aliasing=n 90
Wstrict-null-sentinel 58
Wstrict-overflow 91
Wstrict-prototypes 112
Wstrict-selector-match 63
Wstringop-overflow 91, 92
Wstringop-truncation 93
Wsubobject-linkage 107
Wsuggest-attribute= 94
Wsuggest-attribute=cold 94
Wsuggest-attribute=const 94
Wsuggest-attribute=format 94
Wsuggest-attribute=malloc 94
Wsuggest-attribute=noreturn 94
Wsuggest-attribute=pure 94
Wsuggest-final-methods 95
Wsuggest-final-types 95
Wswitch 86
Wswitch-bool 86
Wswitch-default 86
Wswitch-enum 86
Wswitch-unreachable 86
Wsync-nand 87
Wsystem-headers 99

Wtautological-compare 99
Wtemplates 59
Wterminate 59
Wtraditional 99
Wtraditional-conversion 101
Wtrampolines 99
Wtrigraphs 104
Wtype-limits 104
Wundeclared-selector 63
Wundef 104
Wuninitialized 88
Wunknown-pragmas 90
Wunsafe-loop-optimizations 103
Wunsuffixed-float-constants 118
Wunused 88
Wunused-but-set-parameter 87
Wunused-but-set-variable 87
Wunused-const-variable 88
Wunused-function 87
Wunused-label 87
Wunused-local-typedefs 87
Wunused-macros 105
Wunused-parameter 87
Wunused-result 87
Wunused-value 88
Wunused-variable 87
Wuseless-cast 108
Wvarargs 117
Wvariadic-macros 116
Wvector-operation-performance 117
Wvirtual-inheritance 59
Wvirtual-move-assign 117
Wvla 117
Wvla-larger-than= 117
Wvolatile-register-var 117
Wwrite-strings 106
Wzero-as-null-pointer-constant 107

X

x 33
Xassembler 209
Xbind-lazy 401
Xbind-now 401
Xlinker 214
Xpreprocessor 209

Y

Ym 395
YP 395

Z

z 215

Keyword Index

#

#pragma 801
#pragma implementation 818
#pragma implementation, implied 818
#pragma interface 817

$

$ 564

%

'%' in constraint 589
%include 428
%include_noerr 428
%rename 429

&

'&' in constraint 589

'

' 872

(

(const 653

*

*__builtin_alloca 643
*__builtin_alloca_with_align 643
*__builtin_alloca_with_align_and_max 644

+

'+' in constraint 589

-

'-lgcc', use with '-nodefaultlibs' 212
'-lgcc', use with '-nostdlib' 212
'-march' feature modifiers 250
'-mcpu' feature modifiers 250
'-nodefaultlibs' and unresolved references 212
'-nostdlib' and unresolved references 212

.

.sdata/.sdata2 references (PowerPC) 370

/

// 564

<

'<' in constraint 586

=

'=' in constraint 589

>

'>' in constraint 587

?

?: extensions 460
?: side effect 460

_

'_' in variables in macros 459
__atomic_add_fetch 635
__atomic_always_lock_free 636
__atomic_and_fetch 635
__atomic_clear 636
__atomic_compare_exchange 635
__atomic_compare_exchange_n 634
__atomic_exchange 634
__atomic_exchange_n 634
__atomic_fetch_add 635
__atomic_fetch_and 635
__atomic_fetch_nand 635
__atomic_fetch_or 635
__atomic_fetch_sub 635
__atomic_fetch_xor 635
__atomic_is_lock_free 636
__atomic_load 634
__atomic_load_n 634
__atomic_nand_fetch 635
__atomic_or_fetch 635
__atomic_signal_fence 636
__atomic_store 634
__atomic_store_n 634
__atomic_sub_fetch 635
__atomic_test_and_set 635
__atomic_thread_fence 636
__atomic_xor_fetch 635
__builtin___clear_cache 652
__builtin___fprintf_chk 639
__builtin___memcpy_chk 639
__builtin___memmove_chk 639
__builtin___mempcpy_chk 639

__builtin___memset_chk 639
__builtin___printf_chk 639
__builtin___snprintf_chk 639
__builtin___sprintf_chk 639
__builtin___stpcpy_chk 639
__builtin___strcat_chk 639
__builtin___strcpy_chk 639
__builtin___strncat_chk 639
__builtin___strncpy_chk 639
__builtin___vfprintf_chk 639
__builtin___vprintf_chk 639
__builtin___vsnprintf_chk 639
__builtin___vsprintf_chk 639
__builtin_add_overflow 636
__builtin_add_overflow_p 638
__builtin_addf128_round_to_odd 707
__builtin_alloca 641
__builtin_alloca_with_align 641
__builtin_alloca_with_align_and_max 641
__builtin_apply 458
__builtin_apply_args 457
__builtin_arc_aligned 660
__builtin_arc_brk 660
__builtin_arc_core_read 660
__builtin_arc_core_write 660
__builtin_arc_divaw 660
__builtin_arc_flag 661
__builtin_arc_lr 661
__builtin_arc_mul64 661
__builtin_arc_mulu64 661
__builtin_arc_nop 661
__builtin_arc_norm 661
__builtin_arc_normw 661
__builtin_arc_rtie 661
__builtin_arc_sleep 661
__builtin_arc_sr 661
__builtin_arc_swap 662
__builtin_arc_swi 662
__builtin_arc_sync 662
__builtin_arc_trap_s 662
__builtin_arc_unimp_s 662
__builtin_assume_aligned 651
__builtin_bswap16 656
__builtin_bswap32 656
__builtin_bswap64 656
__builtin_call_with_static_chain 641, 647
__builtin_choose_expr 647
__builtin_clrsb 655
__builtin_clrsbl 655
__builtin_clrsbll 656
__builtin_clz 655
__builtin_clzl 655
__builtin_clzll 655
__builtin_complex 648
__builtin_constant_p 648
__builtin_convertvector 629
__builtin_cpu_init 701, 775
__builtin_cpu_is 701, 776
__builtin_cpu_supports 702, 777
__builtin_ctz 655
__builtin_ctzl 655
__builtin_ctzll 656
__builtin_divf128_round_to_odd 707
__builtin_expect 649
__builtin_expect_with_probability 650
__builtin_extend_pointer 641, 656
__builtin_extract_return_addr 626
__builtin_ffs 655
__builtin_ffsl 655
__builtin_ffsll 655
__builtin_FILE 651
__builtin_fmaf128_round_to_odd 707
__builtin_fpclassify 641, 653
__builtin_frame_address 626
__builtin_frob_return_address 626
__builtin_FUNCTION 651
__builtin_goacc_parlevel_id 656
__builtin_goacc_parlevel_size 656
__builtin_has_attribute 641, 644
__builtin_huge_val 653
__builtin_huge_valf 653
__builtin_huge_valf*n* 653
__builtin_huge_valf*n*x 653
__builtin_huge_vall 653
__builtin_huge_valq 775
__builtin_inf 653
__builtin_infd128 653
__builtin_infd32 653
__builtin_infd64 653
__builtin_inff 653
__builtin_inff*n* 653
__builtin_inff*n*x 654
__builtin_infl 653
__builtin_infq 775
__builtin_is_constant_evaluated 649
__builtin_isfinite 641
__builtin_isgreater 641
__builtin_isgreaterequal 641
__builtin_isinf_sign 641, 654
__builtin_isless 641
__builtin_islessequal 641
__builtin_islessgreater 641
__builtin_isnormal 641
__builtin_isunordered 641
__builtin_LINE 651
__builtin_longjmp 457
__builtin_mul_overflow 637
__builtin_mul_overflow_p 638
__builtin_mulf128_round_to_odd 707
__builtin_nan 654
__builtin_nand128 654
__builtin_nand32 654
__builtin_nand64 654
__builtin_nanf 654
__builtin_nanf*n* 654
__builtin_nanf*n*x 654

__builtin_nanl 654
__builtin_nanq 775
__builtin_nans 654
__builtin_nansf 654
__builtin_nansf*n* 655
__builtin_nansf*n*x 655
__builtin_nansl 654
__builtin_nansq 775
__builtin_nds32_isb 700
__builtin_nds32_isync 699
__builtin_nds32_mfsr 700
__builtin_nds32_mfusr 700
__builtin_nds32_mtsr 700
__builtin_nds32_mtusr 700
__builtin_nds32_setgie_dis 700
__builtin_nds32_setgie_en 700
__builtin_non_tx_store 768
__builtin_object_size 639, 641
__builtin_offsetof 630
__builtin_parity 655
__builtin_parityl 655
__builtin_parityll 656
__builtin_popcount 655
__builtin_popcountl 655
__builtin_popcountll 656
__builtin_powi 641, 656
__builtin_powif 641, 656
__builtin_powil 641, 656
__builtin_prefetch 652
__builtin_return 458
__builtin_return_address 625
__builtin_rx_brk 765
__builtin_rx_clrpsw 765
__builtin_rx_int 765
__builtin_rx_machi 765
__builtin_rx_maclo 765
__builtin_rx_mulhi 765
__builtin_rx_mullo 765
__builtin_rx_mvfachi 765
__builtin_rx_mvfacmi 765
__builtin_rx_mvfc 765
__builtin_rx_mvtachi 765
__builtin_rx_mvtaclo 765
__builtin_rx_mvtc 765
__builtin_rx_mvtipl 766
__builtin_rx_racw 766
__builtin_rx_revw 766
__builtin_rx_rmpa 766
__builtin_rx_round 766
__builtin_rx_sat 766
__builtin_rx_setpsw 766
__builtin_rx_wait 766
__builtin_sadd_overflow 636
__builtin_saddl_overflow 636
__builtin_saddll_overflow 636
__builtin_set_thread_pointer 768
__builtin_setjmp 457
__builtin_sh_get_fpscr 768
__builtin_sh_set_fpscr 768
__builtin_shuffle 628
__builtin_smul_overflow 637
__builtin_smull_overflow 637
__builtin_smulll_overflow 637
__builtin_speculation_safe_value 641, 645
__builtin_sqrtf128_round_to_odd 707
__builtin_ssub_overflow 637
__builtin_ssubl_overflow 637
__builtin_ssubll_overflow 637
__builtin_sub_overflow 637
__builtin_sub_overflow_p 638
__builtin_subf128_round_to_odd 707
__builtin_tabort 768
__builtin_tbegin 766
__builtin_tbegin_nofloat 767
__builtin_tbegin_retry 767
__builtin_tbegin_retry_nofloat 767
__builtin_tbeginc 767
__builtin_tend 767
__builtin_tgmath 647
__builtin_thread_pointer 768
__builtin_trap 650
__builtin_truncf128_round_to_odd 707
__builtin_tx_assist 768
__builtin_tx_nesting_depth 768
__builtin_types_compatible_p 646
__builtin_uadd_overflow 636
__builtin_uaddl_overflow 637
__builtin_uaddll_overflow 637
__builtin_umul_overflow 637
__builtin_umull_overflow 637
__builtin_umulll_overflow 637
__builtin_unreachable 650
__builtin_usub_overflow 637
__builtin_usubl_overflow 637
__builtin_usubll_overflow 637
__builtin_va_arg_pack 458
__builtin_va_arg_pack_len 458
__complex__ keyword 461
__declspec(dllexport) 511
__declspec(dllimport) 512
__ea SPU Named Address Spaces 468
__extension__ 624
__far M32C Named Address Spaces 468
__far RL78 Named Address Spaces 468
__flash AVR Named Address Spaces 466
__flash1 AVR Named Address Spaces 466
__flash2 AVR Named Address Spaces 466
__flash3 AVR Named Address Spaces 466
__flash4 AVR Named Address Spaces 466
__flash5 AVR Named Address Spaces 466
__float128 data type 462
__float80 data type 462
__fp16 data type 463
__func__ identifier 624
__FUNCTION__ identifier 624
__ibm128 data type 462

__imag__ keyword ... 461
__int128 data types ... 461
__memx AVR Named Address Spaces ... 466
__PRETTY_FUNCTION__ identifier ... 624
__real__ keyword ... 461
__seg_fs x86 named address space ... 468
__seg_gs x86 named address space ... 468
__STDC_HOSTED__ ... 5
__sync_add_and_fetch ... 631
__sync_and_and_fetch ... 631
__sync_bool_compare_and_swap ... 631
__sync_fetch_and_add ... 631
__sync_fetch_and_and ... 631
__sync_fetch_and_nand ... 631
__sync_fetch_and_or ... 631
__sync_fetch_and_sub ... 631
__sync_fetch_and_xor ... 631
__sync_lock_release ... 632
__sync_lock_test_and_set ... 632
__sync_nand_and_fetch ... 631
__sync_or_and_fetch ... 631
__sync_sub_and_fetch ... 631
__sync_synchronize ... 632
__sync_val_compare_and_swap ... 631
__sync_xor_and_fetch ... 631
__thread ... 810
_Accum data type ... 464
_Complex keyword ... 461
_Decimal128 data type ... 464
_Decimal32 data type ... 464
_Decimal64 data type ... 464
_Exit ... 641
_exit ... 641
_Floatn data types ... 462
_Floatnx data types ... 462
_Fract data type ... 464
_get_ssp ... 800
_HTM_FIRST_USER_ABORT_CODE ... 767
_inc_ssp ... 800
_Sat data type ... 464
_xabort ... 800
_xbegin ... 799
_xend ... 799
_xtest ... 799

0

'0' in constraint ... 588

A

AArch64 Options ... 245
abi_tag function attribute ... 821
abi_tag type attribute ... 821
abi_tag variable attribute ... 821
ABI ... 845
abort ... 641
abs ... 641
absdata variable attribute, AVR ... 544
accessing volatiles ... 566, 815
acos ... 641
acosf ... 641
acosh ... 641
acoshf ... 641
acoshl ... 641
acosl ... 641
Ada ... 3
additional floating types ... 462
address constraints ... 588
address of a label ... 454
address variable attribute, AVR ... 544
address_operand ... 588
alias function attribute ... 478
aligned function attribute ... 478
aligned type attribute ... 549
aligned variable attribute ... 536
alignment ... 564
alloc_align function attribute ... 479
alloc_size function attribute ... 479
alloc_size type attribute ... 551
alloc_size variable attribute ... 538
alloca ... 641
alloca vs variable-length arrays ... 470
Allow nesting in an interrupt handler on the Blackfin processor ... 506
Altera Nios II options ... 346
alternate keywords ... 624
altivec type attribute, PowerPC ... 557
altivec variable attribute, PowerPC ... 547
always_inline function attribute ... 479
AMD GCN Options ... 253
AMD1 ... 5
amdgpu_hsa_kernel function attribute, AMD GCN ... 500
ANSI C ... 5
ANSI C standard ... 5
ANSI C89 ... 5
ANSI support ... 37
ANSI X3.159-1989 ... 5
apostrophes ... 872
application binary interface ... 845
ARC options ... 254
arch= function attribute, AArch64 ... 498
arch= function attribute, ARM ... 503
ARM [Annotated C++ Reference Manual] ... 827
ARM options ... 263
arrays of length zero ... 468
arrays of variable length ... 470
arrays, non-lvalue ... 472
artificial function attribute ... 480
asin ... 641
asinf ... 641
asinh ... 641
asinhf ... 641
asinhl ... 641
asinl ... 641

asm assembler template ... 573
asm clobbers ... 579
asm constraints ... 586
asm expressions ... 578
asm flag output operands ... 577
asm goto labels ... 582
asm inline ... 624
asm input operands ... 578
asm keyword ... 567
asm output operands ... 574
asm scratch registers ... 579
asm volatile ... 571
assembler names for identifiers ... 620
assembly code, invalid ... 885
assembly language in C ... 567
assembly language in C, basic ... 567
assembly language in C, extended ... 569
assume_aligned function attribute ... 480
atan ... 641
atan2 ... 641
atan2f ... 641
atan2l ... 641
atanf ... 641
atanh ... 641
atanhf ... 641
atanhl ... 641
atanl ... 641
attribute of types ... 548
attribute of variables ... 536
attribute syntax ... 559
autoincrement/decrement addressing ... 586
automatic inline for C++ member fns ... 565
aux variable attribute, ARC ... 542
AVR Options ... 277

B

Backwards Compatibility ... 827
bank_switch function attribute, M32C ... 508
base class members ... 877
based type attribute, MeP ... 557
based variable attribute, MeP ... 545
basic asm ... 567
bcmp ... 641
below100 variable attribute, Xstormy16 ... 548
binary compatibility ... 845
Binary constants using the '0b' prefix ... 813
Blackfin Options ... 286
bound pointer to member function ... 821
branch-protection function attribute, AArch64 ... 498
break handler functions ... 511
break_handler function attribute, MicroBlaze ... 511
brk_interrupt function attribute, RL78 ... 521
bug criteria ... 885
bugs ... 885
bugs, known ... 869
built-in functions ... 41, 641
bzero ... 641

C

c++ ... 37
C compilation options ... 11
C intermediate output, nonexistent ... 3
C language extensions ... 451
C language, traditional ... 207
C standard ... 5
C standards ... 5
C++ ... 3
C++ comments ... 564
C++ interface and implementation headers ... 817
C++ language extensions ... 815
C++ member fns, automatically inline ... 565
C++ misunderstandings ... 876
C++ options, command-line ... 44
C++ pragmas, effect on inlining ... 818
C++ source file suffixes ... 37
C++ static data, declaring and defining ... 876
C-SKY Options ... 292
C_INCLUDE_PATH ... 437
C11 ... 5
C17 ... 5
C1X ... 5
C2X ... 5
C6X Options ... 289
C89 ... 5
C90 ... 5
C94 ... 5
C95 ... 5
C99 ... 5
C9X ... 5
cabs ... 641
cabsf ... 641
cabsl ... 641
cacos ... 641
cacosf ... 641
cacosh ... 641
cacoshf ... 641
cacoshl ... 641
cacosl ... 641
callee_pop_aggregate_return function attribute, x86 ... 526
calling functions through the function vector on SH2A ... 523
calloc ... 641
carg ... 641
cargf ... 641
cargl ... 641
case labels in initializers ... 474
case ranges ... 476
casin ... 641
casinf ... 641
casinh ... 641
casinhf ... 641

casinhl ... 641
casinl ... 641
cast to a union ... 476
catan ... 641
catanf ... 641
catanh ... 641
catanhf ... 641
catanhl ... 641
catanl ... 641
cb variable attribute, MeP ... 546
cbrt ... 641
cbrtf ... 641
cbrtl ... 641
ccos ... 641
ccosf ... 641
ccosh ... 641
ccoshf ... 641
ccoshl ... 641
ccosl ... 641
cdecl function attribute, x86-32 ... 525
ceil ... 641
ceilf ... 641
ceill ... 641
cexp ... 641
cexpf ... 641
cexpl ... 641
cf_check function attribute, x86 ... 535
character set, execution ... 206
character set, input ... 206
character set, input normalization ... 113
character set, wide execution ... 206
cimag ... 641
cimagf ... 641
cimagl ... 641
cleanup variable attribute ... 538
clog ... 641
clog10 ... 641
clog10f ... 641
clog10l ... 641
clogf ... 641
clogl ... 641
cmodel= function attribute, AArch64 ... 498
COBOL ... 3
code generation conventions ... 218
code, mixed with declarations ... 477
cold function attribute ... 480
cold label attribute ... 558
command options ... 11
comments, C++ style ... 564
common variable attribute ... 538
comparison of signed and unsigned values, warning ... 109
compilation statistics ... 228
compiler bugs, reporting ... 885
compiler compared to C++ preprocessor ... 3
compiler options, C++ ... 44
compiler options, Objective-C and Objective-C++ ... 60
compiler version, specifying ... 11
COMPILER_PATH ... 437
complex conjugation ... 462
complex numbers ... 461
compound literals ... 473
computed gotos ... 454
conditional expressions, extensions ... 460
conflicting types ... 875
conj ... 641
conjf ... 641
conjl ... 641
const applied to function ... 477
const function attribute ... 480
const qualifier ... 473
constants in constraints ... 587
constraint modifier characters ... 589
constraint, matching ... 588
constraints, asm ... 586
constraints, machine specific ... 590
constructing calls ... 457
constructor expressions ... 473
constructor function attribute ... 481
contributors ... 915
copy function attribute ... 481
copy type attribute ... 551
copy variable attribute ... 539
copysign ... 641
copysignf ... 641
copysignl ... 641
core dump ... 885
cos ... 641
cosf ... 641
cosh ... 641
coshf ... 641
coshl ... 641
cosl ... 641
CPATH ... 437
CPLUS_INCLUDE_PATH ... 437
cpow ... 641
cpowf ... 641
cpowl ... 641
cproj ... 641
cprojf ... 641
cprojl ... 641
cpu= function attribute, AArch64 ... 498
CR16 Options ... 291
creal ... 641
crealf ... 641
creall ... 641
CRIS Options ... 290
critical function attribute, MSP430 ... 514
cross compiling ... 11
csin ... 641
csinf ... 641
csinh ... 641
csinhf ... 641
csinhl ... 641
csinl ... 641

csqrt ... 641
csqrtf ... 641
csqrtl ... 641
ctan ... 641
ctanf ... 641
ctanh ... 641
ctanhf ... 641
ctanhl ... 641
ctanl ... 641

D

Darwin options ... 294
dcgettext ... 641
dd integer suffix ... 464
DD integer suffix ... 464
deallocating variable length arrays ... 470
debug dump options ... 228
debugging GCC ... 228
debugging information options ... 118
decimal floating types ... 464
declaration scope ... 872
declarations inside expressions ... 451
declarations, mixed with code ... 477
declaring attributes of functions ... 477
declaring static data in C++ ... 876
defining static data in C++ ... 876
dependencies for make as output ... 438
dependencies, make ... 203
DEPENDENCIES_OUTPUT ... 438
dependent name lookup ... 877
deprecated enumerator attribute ... 559
deprecated function attribute ... 482
deprecated type attribute ... 552
deprecated variable attribute ... 539
designated initializers ... 474
designated_init type attribute ... 552
designator lists ... 476
designators ... 475
destructor function attribute ... 481
developer options ... 228
df integer suffix ... 464
DF integer suffix ... 464
dgettext ... 641
diagnostic messages ... 63
dialect options ... 37
diff-delete GCC_COLORS capability ... 65
diff-filename GCC_COLORS capability ... 65
diff-hunk GCC_COLORS capability ... 65
diff-insert GCC_COLORS capability ... 65
digits in constraint ... 588
directory options ... 215
disinterrupt function attribute, Epiphany ... 507
disinterrupt function attribute, MeP ... 510
dl integer suffix ... 464
DL integer suffix ... 464
dllexport function attribute ... 511
dllexport variable attribute ... 546
dllimport function attribute ... 512
dllimport variable attribute ... 546
dollar signs in identifier names ... 564
double-word arithmetic ... 461
downward funargs ... 455
drem ... 641
dremf ... 641
dreml ... 641
dump options ... 228
D ... 3

E

'E' in constraint ... 587
earlyclobber operand ... 589
eight-bit data on the H8/300, H8/300H, and H8S ... 544
eightbit_data variable attribute, H8/300 ... 544
EIND ... 282
either function attribute, MSP430 ... 515
either variable attribute, MSP430 ... 547
empty structures ... 470
Enumerator Attributes ... 558
environment variables ... 436
erf ... 641
erfc ... 641
erfcf ... 641
erfcl ... 641
erff ... 641
erfl ... 641
error function attribute ... 482
error GCC_COLORS capability ... 65
error messages ... 883
escaped newlines ... 472
exception function attribute ... 516
exception handler functions, Blackfin ... 505
exception handler functions, NDS32 ... 516
exception_handler function attribute ... 505
exit ... 641
exp ... 641
exp10 ... 641
exp10f ... 641
exp10l ... 641
exp2 ... 641
exp2f ... 641
exp2l ... 641
expf ... 641
expl ... 641
explicit register variables ... 621
expm1 ... 641
expm1f ... 641
expm1l ... 641
expressions containing statements ... 451
expressions, constructor ... 473
extended asm ... 569
extensible constraints ... 588
extensions, ?: ... 460
extensions, C language ... 451

extensions, C++ language 815
external declaration scope 872
externally_visible function attribute 483
extra NOP instructions at the function entry point 491

F

'F' in constraint 587
fabs 641
fabsf 641
fabsl 641
fallthrough statement attribute 559
far function attribute, MeP 510
far function attribute, MIPS 513
far type attribute, MeP 557
far variable attribute, MeP 545
fast_interrupt function attribute, M32C 508
fast_interrupt function attribute, MicroBlaze 511
fast_interrupt function attribute, RX 521
fastcall function attribute, x86-32 525
fatal signal 885
fdim 641
fdimf 641
fdiml 641
FDL, GNU Free Documentation License 907
fentry_name function attribute, x86 535
fentry_section function attribute, x86 536
ffs 641
file name suffix 31
file names 209
fix-cortex-a53-835769 function attribute, AArch64 497
fixed-point types 464
fixit-delete GCC_COLORS capability 65
fixit-insert GCC_COLORS capability 65
flatten function attribute 483
flexible array members 468
float as function value type 873
floating point precision 876
floating-point precision 159
floor 641
floorf 641
floorl 641
fma 641
fmaf 641
fmal 641
fmax 641
fmaxf 641
fmaxl 641
fmin 641
fminf 641
fminl 641
fmod 641
fmodf 641
fmodl 641
force_align_arg_pointer function attribute, x86 526
format function attribute 483
format_arg function attribute 484
Fortran 3
forwarder_section function attribute, Epiphany 507
forwarding calls 457
fprintf 641
fprintf_unlocked 641
fputs 641
fputs_unlocked 641
FR30 Options 302
freestanding environment 5
freestanding implementation 5
frexp 641
frexpf 641
frexpl 641
FRV Options 303
fscanf 641
fscanf, and constant strings 871
FT32 Options 303
function addressability on the M32R/D 509
function attributes 477
function pointers, arithmetic 472
function prototype declarations 563
function versions 822
function, size of pointer to 472
function_return function attribute, x86 534
function_vector function attribute, H8/300 .. 508
function_vector function attribute, M16C/M32C 508
function_vector function attribute, SH 523
functions in arbitrary sections 492
functions that are dynamically resolved 485
functions that are passed arguments in registers on x86-32 526
functions that behave like malloc 487
functions that have no side effects 480, 491
functions that never return 490
functions that pop the argument stack on x86-32 ..... 525, 527
functions that return more than once 492
functions with non-null pointer arguments 489
functions with printf, scanf, strftime or strfmon style arguments 483

G

'g' in constraint 587
g++ 37
'G' in constraint 587
G++ 3
gamma 641
gamma_r 641
gammaf 641
gammaf_r 641
gammal 641

gammal_r ... 641
gcc_struct type attribute, PowerPC ... 557
gcc_struct type attribute, x86 ... 557
gcc_struct variable attribute, PowerPC ... 547
gcc_struct variable attribute, x86 ... 548
GCC ... 3
GCC command options ... 11
GCC_COLORS environment variable ... 64
GCC_COMPARE_DEBUG ... 436
GCC_EXEC_PREFIX ... 436
gcov ... 190
general-regs-only function attribute, AArch64 ... 497
general-regs-only function attribute, ARM ... 502
gettext ... 641
global offset table ... 224
global register after longjmp ... 622
global register variables ... 621
GNAT ... 3
gnu_inline function attribute ... 485
GNU C Compiler ... 3
GNU Compiler Collection ... 3
Go ... 3
goto with computed label ... 454
gprof ... 189
grouping options ... 11

H

'H' in constraint ... 587
half-precision floating point ... 463
hardware models and configurations, specifying ... 245
hex floats ... 464
highlight, color ... 64
hk fixed-suffix ... 464
HK fixed-suffix ... 464
hosted environment ... 5, 42
hosted implementation ... 5
hot function attribute ... 485
hot label attribute ... 558
hotpatch function attribute, S/390 ... 522
HPPA Options ... 308
hr fixed-suffix ... 464
HR fixed-suffix ... 464
hypot ... 641
hypotf ... 641
hypotl ... 641

I

'i' in constraint ... 587
'I' in constraint ... 587
IA-64 Options ... 311
IBM RS/6000 and PowerPC Options ... 358
identifier names, dollar signs in ... 564
identifiers, names in assembler code ... 620
ifunc function attribute ... 485
ilogb ... 641
ilogbf ... 641
ilogbl ... 641
imaxabs ... 641
implementation-defined behavior, C language ... 441
implementation-defined behavior, C++ language ... 449
implied #pragma implementation ... 818
incompatibilities of GCC ... 871
increment operators ... 885
index ... 641
indirect calls, ARC ... 501
indirect calls, ARM ... 502
indirect calls, Blackfin ... 506
indirect calls, Epiphany ... 507
indirect calls, MIPS ... 513
indirect calls, PowerPC ... 517
indirect functions ... 485
indirect_branch function attribute, x86 ... 534
indirect_return function attribute, x86 ... 535
init_priority variable attribute ... 822
initializations in expressions ... 473
initializers with labeled elements ... 474
initializers, non-constant ... 473
inline assembly language ... 567
inline automatic for C++ member fns ... 565
inline functions ... 564
inline functions, omission of ... 565
inlining and C++ pragmas ... 818
installation trouble ... 869
instrumentation options ... 189
integrating function code ... 564
interface and implementation headers, C++ ... 817
intermediate C version, nonexistent ... 3
interrupt function attribute, ARC ... 501
interrupt function attribute, ARM ... 502
interrupt function attribute, AVR ... 504
interrupt function attribute, C-SKY ... 506
interrupt function attribute, CR16 ... 506
interrupt function attribute, Epiphany ... 507
interrupt function attribute, m68k ... 510
interrupt function attribute, M32C ... 509
interrupt function attribute, M32R/D ... 509
interrupt function attribute, MeP ... 510
interrupt function attribute, MIPS ... 512
interrupt function attribute, MSP430 ... 515
interrupt function attribute, NDS32 ... 516
interrupt function attribute, RISC-V ... 521
interrupt function attribute, RL78 ... 521
interrupt function attribute, RX ... 521
interrupt function attribute, V850 ... 525
interrupt function attribute, Visium ... 525
interrupt function attribute, x86 ... 527
interrupt function attribute, Xstormy16 ... 536
interrupt_handler function attribute, Blackfin ... 505

interrupt_handler function attribute, H8/300 ... 508
interrupt_handler function attribute, m68k ... 510
interrupt_handler function attribute, MicroBlaze ... 511
interrupt_handler function attribute, SH ... 524
interrupt_handler function attribute, V850 ... 525
interrupt_thread function attribute, fido ... 510
introduction ... 1
invalid assembly code ... 885
invalid input ... 885
invoking g++ ... 37
io variable attribute, AVR ... 543
io variable attribute, MeP ... 546
io_low variable attribute, AVR ... 543
isalnum ... 641
isalpha ... 641
isascii ... 641
isblank ... 641
iscntrl ... 641
isdigit ... 641
isgraph ... 641
islower ... 641
ISO 9899 ... 5
ISO C ... 5
ISO C standard ... 5
ISO C11 ... 5
ISO C17 ... 5
ISO C1X ... 5
ISO C2X ... 5
ISO C90 ... 5
ISO C94 ... 5
ISO C95 ... 5
ISO C99 ... 5
ISO C9X ... 5
ISO support ... 37
ISO/IEC 9899 ... 5
isprint ... 641
ispunct ... 641
isr function attribute, ARM ... 502
isr function attribute, C-SKY ... 506
isspace ... 641
isupper ... 641
iswalnum ... 641
iswalpha ... 641
iswblank ... 641
iswcntrl ... 641
iswdigit ... 641
iswgraph ... 641
iswlower ... 641
iswprint ... 641
iswpunct ... 641
iswspace ... 641
iswupper ... 641
iswxdigit ... 641
isxdigit ... 641

J

j0 ... 641
j0f ... 641
j0l ... 641
j1 ... 641
j1f ... 641
j1l ... 641
jli_always function attribute, ARC ... 502
jli_fixed function attribute, ARC ... 502
jn ... 641
jnf ... 641
jnl ... 641

K

k fixed-suffix ... 464
K fixed-suffix ... 464
keep_interrupts_masked function attribute, MIPS ... 513
kernel attribute, Nvidia PTX ... 517
keywords, alternate ... 624
known causes of trouble ... 869
kspisusp function attribute, Blackfin ... 505

L

l1_data variable attribute, Blackfin ... 544
l1_data_A variable attribute, Blackfin ... 544
l1_data_B variable attribute, Blackfin ... 544
l1_text function attribute, Blackfin ... 505
l2 function attribute, Blackfin ... 506
l2 variable attribute, Blackfin ... 544
Label Attributes ... 557
labeled elements in initializers ... 474
labels as values ... 454
labs ... 641
language dialect options ... 37
LANG ... 436, 437
LC_ALL ... 436
LC_CTYPE ... 436
LC_MESSAGES ... 436
ldexp ... 641
ldexpf ... 641
ldexpl ... 641
leaf function attribute ... 487
length-zero arrays ... 468
lgamma ... 641
lgamma_r ... 641
lgammaf ... 641
lgammaf_r ... 641
lgammal ... 641
lgammal_r ... 641
Libraries ... 211
LIBRARY_PATH ... 437
link options ... 209
linker script ... 214
lk fixed-suffix ... 464
LK fixed-suffix ... 464

LL integer suffix ... 461
llabs ... 641
llk fixed-suffix ... 464
LLK fixed-suffix ... 464
llr fixed-suffix ... 464
LLR fixed-suffix ... 464
llrint ... 641
llrintf ... 641
llrintl ... 641
llround ... 641
llroundf ... 641
llroundl ... 641
LM32 options ... 314
load address instruction ... 588
local labels ... 453
local variables in macros ... 459
local variables, specifying registers ... 622
locale ... 436
locale definition ... 437
locus GCC_COLORS capability ... 65
log ... 641
log10 ... 641
log10f ... 641
log10l ... 641
log1p ... 641
log1pf ... 641
log1pl ... 641
log2 ... 641
log2f ... 641
log2l ... 641
logb ... 641
logbf ... 641
logbl ... 641
logf ... 641
logl ... 641
long long data types ... 461
long_call function attribute, ARC ... 501
long_call function attribute, ARM ... 502
long_call function attribute, Epiphany ... 507
long_call function attribute, MIPS ... 513
longcall function attribute, Blackfin ... 506
longcall function attribute, PowerPC ... 517
longjmp ... 622
longjmp incompatibilities ... 871
longjmp warnings ... 89
lower function attribute, MSP430 ... 515
lower variable attribute, MSP430 ... 547
lr fixed-suffix ... 464
LR fixed-suffix ... 464
lrint ... 641
lrintf ... 641
lrintl ... 641
lround ... 641
lroundf ... 641
lroundl ... 641

M

'm' in constraint ... 586
M32C options ... 315
M32R/D options ... 315
M680x0 options ... 317
machine specific constraints ... 590
machine-dependent options ... 245
macro with variable arguments ... 471
macros, inline alternative ... 564
macros, local labels ... 453
macros, local variables in ... 459
macros, statements in expressions ... 451
macros, types of arguments ... 459
make ... 203
malloc ... 641
malloc function attribute ... 487
matching constraint ... 588
may_alias type attribute ... 552
MCore options ... 322
medium_call function attribute, ARC ... 501
member fns, automatically inline ... 565
memchr ... 641
memcmp ... 641
memcpy ... 641
memory references in constraints ... 586
mempcpy ... 641
memset ... 641
MeP options ... 323
Mercury ... 3
message formatting ... 63
messages, warning ... 70
messages, warning and error ... 883
MicroBlaze Options ... 324
micromips function attribute ... 514
middle-operands, omitted ... 460
mips16 function attribute, MIPS ... 514
MIPS options ... 326
misunderstandings in C++ ... 876
mixed declarations and code ... 477
mixing assembly language and C ... 567
mktemp, and constant strings ... 871
MMIX Options ... 341
MN10300 options ... 342
mode type attribute ... 553
mode variable attribute ... 539
model function attribute, M32R/D ... 509
model variable attribute, IA-64 ... 545
model-name variable attribute, M32R/D ... 545
modf ... 641
modff ... 641
modfl ... 641
modifiers in constraints ... 589
Moxie Options ... 343
ms_abi function attribute, x86 ... 525
ms_hook_prologue function attribute, x86 ... 526
ms_struct type attribute, PowerPC ... 557
ms_struct type attribute, x86 ... 557
ms_struct variable attribute, PowerPC ... 547

ms_struct variable attribute, x86 548
MSP430 Options 343
multiple alternative constraints 588
multiprecision arithmetic 461

N

'n' in constraint 587
naked function attribute, ARM 502
naked function attribute, AVR 504
naked function attribute, C-SKY 506
naked function attribute, MCORE 510
naked function attribute, MSP430 515
naked function attribute, NDS32 516
naked function attribute, RISC-V 520
naked function attribute, RL78 521
naked function attribute, RX 522
naked function attribute, SPU 524
naked function attribute, x86 526
Named Address Spaces 466
names used in assembler code 620
naming convention, implementation headers ... 818
NDS32 Options 345
near function attribute, MeP 510
near function attribute, MIPS 513
near type attribute, MeP 557
near variable attribute, MeP 545
nearbyint 641
nearbyintf 641
nearbyintl 641
nested function attribute, NDS32 516
nested functions 455
nested_ready function attribute, NDS32 516
nesting function attribute, Blackfin 506
newlines (escaped) 472
nextafter 641
nextafterf 641
nextafterl 641
nexttoward 641
nexttowardf 641
nexttowardl 641
NFC 113
NFKC 113
Nios II options 346
nmi function attribute, NDS32 516
nmi_handler function attribute, Blackfin 506
NMI handler functions on the Blackfin processor 506
no_caller_saved_registers function attribute, x86 527
no_gccisr function attribute, AVR 504
no_icf function attribute 488
no_instrument_function function attribute .. 488
no_profile_instrument_function function attribute 488
no_reorder function attribute 488
no_sanitize function attribute 488
no_sanitize_address function attribute 488
no_sanitize_thread function attribute 488
no_sanitize_undefined function attribute ... 488
no_split_stack function attribute 489
no_stack_limit function attribute 489
nocf_check function attribute 534
noclone function attribute 489
nocommon variable attribute 538
nocompression function attribute, MIPS 514
noinit variable attribute, MSP430 547
noinline function attribute 489
noipa function attribute 489
nomicromips function attribute 514
nomips16 function attribute, MIPS 514
non-constant initializers 473
non-static inline function 566
nonlocal gotos 456
nonnull function attribute 489
nonstring variable attribute 539
noplt function attribute 490
noreturn function attribute 490
nosave_low_regs function attribute, SH 524
not_nested function attribute, NDS32 516
note GCC_COLORS capability 65
nothrow function attribute 491
notshared type attribute, ARM 556
Nvidia PTX options 351
nvptx options 351

O

'o' in constraint 586
OBJC_INCLUDE_PATH 437
Objective-C 3, 7
Objective-C and Objective-C++ options, command-line 60
Objective-C++ 3, 7
offsettable address 586
old-style function definitions 563
omit-leaf-frame-pointer function attribute, AArch64 498
omitted middle-operands 460
open coding 564
OpenACC accelerator programming 42
OpenMP parallel 42
OpenMP SIMD 43
OpenRISC Options 352
operand constraints, asm 586
optimize function attribute 491
optimize options 125
options to control diagnostics formatting 63
options to control warnings 70
options, C++ 44
options, code generation 218
options, debugging 118
options, dialect 37
options, directory search 215
options, GCC command 11
options, grouping 11

options, linking ... 209
options, Objective-C and Objective-C++ ... 60
options, optimization ... 125
options, order ... 11
options, preprocessor ... 202
options, profiling ... 189
options, program instrumentation ... 189
options, run-time error checking ... 189
order of evaluation, side effects ... 882
order of options ... 11
`OS_main` function attribute, AVR ... 505
`OS_task` function attribute, AVR ... 505
other register constraints ... 588
output file option ... 33
overloaded virtual function, warning ... 59

P

'p' in constraint ... 588
`packed` type attribute ... 553
`packed` variable attribute ... 540
parameter forward declaration ... 471
`partial_save` function attribute, NDS32 ... 516
`patchable_function_entry` function attribute ... 491
`pcs` function attribute, ARM ... 503
PDP-11 Options ... 353
`persistent` variable attribute, MSP430 ... 547
picoChip options ... 354
PIC ... 224
pmf ... 821
pointer arguments ... 481
pointer arguments in variadic functions ... 473
pointer to member function ... 821
pointers to arrays ... 473
portions of temporary objects, pointers to ... 878
`pow` ... 641
`pow10` ... 641
`pow10f` ... 641
`pow10l` ... 641
PowerPC options ... 354
`powf` ... 641
`powl` ... 641
pragma GCC ivdep ... 809
pragma GCC optimize ... 808
pragma GCC pop_options ... 808
pragma GCC push_options ... 808
pragma GCC reset_options ... 809
pragma GCC target ... 808
pragma GCC unroll *n* ... 809
pragma, address ... 802
pragma, align ... 804
pragma, call ... 803
pragma, coprocessor available ... 802
pragma, coprocessor call_saved ... 802
pragma, coprocessor subclass ... 803
pragma, custom io_volatile ... 802
pragma, diagnostic ... 806, 807
pragma, disinterrupt ... 803
pragma, fini ... 804
pragma, init ... 804
pragma, long_calls ... 802
pragma, long_calls_off ... 802
pragma, longcall ... 803
pragma, mark ... 804
pragma, memregs ... 802
pragma, no_long_calls ... 802
pragma, options align ... 804
pragma, pop_macro ... 808
pragma, push_macro ... 808
pragma, redefine_extname ... 804
pragma, segment ... 804
pragma, unused ... 804
pragma, visibility ... 807
pragma, weak ... 806
pragmas ... 801
pragmas in C++, effect on inlining ... 818
pragmas, interface and implementation ... 817
pragmas, warning of unknown ... 90
precompiled headers ... 438
preprocessing numbers ... 873
preprocessing tokens ... 873
preprocessor options ... 202
`printf` ... 641
`printf_unlocked` ... 641
`prof` ... 189
profiling options ... 189
`progmem` variable attribute, AVR ... 542
program instrumentation options ... 189
promotion of formal parameters ... 563
`pure` function attribute ... 491
push address instruction ... 588
`putchar` ... 641
`puts` ... 641

Q

`q` floating point suffix ... 462
`Q` floating point suffix ... 462
`qsort`, and global register variables ... 622
`quote GCC_COLORS` capability ... 65

R

`r` fixed-suffix ... 464
'r' in constraint ... 587
`R` fixed-suffix ... 464
`RAMPD` ... 284
`RAMPX` ... 284
`RAMPY` ... 284
`RAMPZ` ... 284
`range1 GCC_COLORS` capability ... 65
`range2 GCC_COLORS` capability ... 65
ranges in case statements ... 476
read-only strings ... 871
`reentrant` function attribute, MSP430 ... 515

register variable after `longjmp` ... 622
registers for local variables ... 622
registers in constraints ... 587
registers, global allocation ... 621
registers, global variables in ... 621
`regparm` function attribute, x86 ... 526
relocation truncated to fit (ColdFire) ... 322
relocation truncated to fit (MIPS) ... 330
`remainder` ... 641
`remainderf` ... 641
`remainderl` ... 641
`remquo` ... 641
`remquof` ... 641
`remquol` ... 641
`renesas` function attribute, SH ... 524
reordering, warning ... 56
reporting bugs ... 885
`resbank` function attribute, SH ... 524
`reset` function attribute, NDS32 ... 516
reset handler functions ... 516
rest argument (in macro) ... 471
restricted pointers ... 815
restricted references ... 815
restricted this pointer ... 815
`returns_nonnull` function attribute ... 492
`returns_twice` function attribute ... 492
`rindex` ... 641
`rint` ... 641
`rintf` ... 641
`rintl` ... 641
RISC-V Options ... 354
RL78 Options ... 356
`round` ... 641
`roundf` ... 641
`roundl` ... 641
RS/6000 and PowerPC Options ... 358
RTTI ... 817
run-time error checking options ... 189
run-time options ... 218
RX Options ... 373

S

'`s`' in constraint ... 587
S/390 and zSeries Options ... 376
`saddr` variable attribute, RL78 ... 547
save all registers on the Blackfin ... 506
save all registers on the H8/300, H8/300H, and H8S ... 508
`save_all` function attribute, NDS32 ... 516
`save_volatiles` function attribute, MicroBlaze ... 511
`saveall` function attribute, Blackfin ... 506
`saveall` function attribute, H8/300 ... 508
`scalar_storage_order` type attribute ... 554
`scalb` ... 641
`scalbf` ... 641
`scalbl` ... 641
`scalbln` ... 641
`scalblnf` ... 641
`scalbn` ... 641
`scalbnf` ... 641
`scanf`, and constant strings ... 871
`scanfnl` ... 641
scope of a variable length array ... 470
scope of declaration ... 875
scope of external declarations ... 872
Score Options ... 380
`sda` variable attribute, V850 ... 548
search path ... 215
`section` function attribute ... 492
`section` variable attribute ... 540
`secure_call` function attribute, ARC ... 502
`selectany` variable attribute ... 546
`sentinel` function attribute ... 493
`setjmp` ... 622
`setjmp` incompatibilities ... 871
`shared` attribute, Nvidia PTX ... 547
shared strings ... 871
`shared` variable attribute ... 546
`short_call` function attribute, ARC ... 501
`short_call` function attribute, ARM ... 502
`short_call` function attribute, Epiphany ... 507
`short_call` function attribute, MIPS ... 513
`shortcall` function attribute, Blackfin ... 506
`shortcall` function attribute, PowerPC ... 517
side effect in `?:` ... 460
side effects, macro argument ... 451
side effects, order of evaluation ... 882
`sign-return-address` function attribute, AArch64 ... 498
`signal` function attribute, AVR ... 505
`signbit` ... 641
`signbitd128` ... 641
`signbitd32` ... 641
`signbitd64` ... 641
`signbitf` ... 641
`signbitl` ... 641
signed and unsigned values, comparison warning ... 109
`significand` ... 641
`significandf` ... 641
`significandl` ... 641
`simd` function attribute ... 493
SIMD ... 43
simple constraints ... 586
`sin` ... 641
`sincos` ... 641
`sincosf` ... 641
`sincosl` ... 641
`sinf` ... 641
`sinh` ... 641
`sinhf` ... 641
`sinhl` ... 641
`sinl` ... 641
`sizeof` ... 459

smaller data references 316, 346
smaller data references (PowerPC) 370
snprintf 641
Solaris 2 options 387
SOURCE_DATE_EPOCH 438
sp_switch function attribute, SH 524
SPARC options 387
Spec Files 428
specified registers 621
specifying compiler version and target machine 11
specifying hardware config 245
specifying machine version 11
specifying registers for local variables 622
speed of compilation 438
sprintf 641
spu_vector type attribute, SPU 557
spu_vector variable attribute, SPU 547
SPU options 393
sqrt 641
sqrtf 641
sqrtl 641
sscanf 641
sscanf, and constant strings 871
sseregparm function attribute, x86 526
stack_protect function attribute 493
Statement Attributes 559
statements inside expressions 451
static data in C++, declaring and defining 876
stdcall function attribute, x86-32 527
stpcpy 641
stpncpy 641
strcasecmp 641
strcat 641
strchr 641
strcmp 641
strcpy 641
strcspn 641
strdup 641
strfmon 641
strftime 641
strict-align function attribute, AArch64 498
string constants 871
strlen 641
strncasecmp 641
strncat 641
strncmp 641
strncpy 641
strndup 641
strnlen 641
strpbrk 641
strrchr 641
strspn 641
strstr 641
struct 809
struct __htm_tdb 767
structures 873
structures, constructor expression 473
submodel options 245
subscripting 472
subscripting and function values 472
suffixes for C++ source 37
SUNPRO_DEPENDENCIES 438
suppressing warnings 70
surprises in C++ 876
syntax checking 70
syscall_linkage function attribute, IA-64 508
system headers, warnings from 99
sysv_abi function attribute, x86 525

T

tan 641
tanf 641
tanh 641
tanhf 641
tanhl 641
tanl 641
target function attribute 494, 503, 516, 517, 522, 528
target machine, specifying 11
target("3dnow") function attribute, x86 528
target("3dnowa") function attribute, x86 528
target("abm") function attribute, x86 528
target("adx") function attribute, x86 528
target("aes") function attribute, x86 528
target("align-stringops") function attribute, x86 533
target("altivec") function attribute, PowerPC 517
target("arch=*ARCH*") function attribute, x86 534
target("arm") function attribute, ARM 503
target("avoid-indexed-addresses") function attribute, PowerPC 520
target("avx") function attribute, x86 528
target("avx2") function attribute, x86 528
target("avx5124fmaps") function attribute, x86 528
target("avx5124vnniw") function attribute, x86 528
target("avx512bitalg") function attribute, x86 528
target("avx512bw") function attribute, x86 529
target("avx512cd") function attribute, x86 529
target("avx512dq") function attribute, x86 529
target("avx512er") function attribute, x86 529
target("avx512f") function attribute, x86 529
target("avx512ifma") function attribute, x86 529
target("avx512pf") function attribute, x86 529
target("avx512vbmi") function attribute, x86 529
target("avx512vbmi2") function attribute, x86 529
target("avx512vl") function attribute, x86 529

target("avx512vnni") function attribute, x86 529
target("avx512vpopcntdq") function attribute, x86 529
target("bmi") function attribute, x86 529
target("bmi2") function attribute, x86 529
target("cld") function attribute, x86 533
target("cldemote") function attribute, x86 .. 530
target("clflushopt") function attribute, x86 530
target("clwb") function attribute, x86 530
target("clzero") function attribute, x86 530
target("cmpb") function attribute, PowerPC 518
target("cpu=*CPU*") function attribute, PowerPC 520
target("crc32") function attribute, x86 530
target("custom-fpu-cfg=*name*") function attribute, Nios II 517
target("custom-*insn=N*") function attribute, Nios II 517
target("cx16") function attribute, x86 530
target("default") function attribute, x86 ... 530
target("dlmzb") function attribute, PowerPC 518
target("f16c") function attribute, x86 530
target("fancy-math-387") function attribute, x86 533
target("fma") function attribute, x86 530
target("fma4") function attribute, x86 530
target("fpmath=*FPMATH*") function attribute, x86 534
target("fprnd") function attribute, PowerPC 518
target("fpu=") function attribute, ARM 503
target("friz") function attribute, PowerPC 520
target("fsgsbase") function attribute, x86 .. 530
target("fxsr") function attribute, x86 530
target("gfni") function attribute, x86 530
target("hard-dfp") function attribute, PowerPC 518
target("hle") function attribute, x86 530
target("ieee-fp") function attribute, x86 ... 533
target("inline-all-stringops") function attribute, x86 533
target("inline-stringops-dynamically") function attribute, x86 533
target("isel") function attribute, PowerPC 518
target("longcall") function attribute, PowerPC 520
target("lwp") function attribute, x86 530
target("lzcnt") function attribute, x86 530
target("mfcrf") function attribute, PowerPC 518
target("mfpgpr") function attribute, PowerPC 518
target("mmx") function attribute, x86 530
target("movbe") function attribute, x86 531
target("movdir64b") function attribute, x86 531
target("movdiri") function attribute, x86 ... 531
target("mulhw") function attribute, PowerPC 518
target("multiple") function attribute, PowerPC 518
target("mwaitx") function attribute, x86 531
target("no-custom-*insn*") function attribute, Nios II 517
target("paired") function attribute, PowerPC 520
target("pclmul") function attribute, x86 531
target("pconfig") function attribute, x86 ... 531
target("pku") function attribute, x86 531
target("popcnt") function attribute, x86 531
target("popcntb") function attribute, PowerPC 519
target("popcntd") function attribute, PowerPC 519
target("powerpc-gfxopt") function attribute, PowerPC 519
target("powerpc-gpopt") function attribute, PowerPC 519
target("prefetchwt1") function attribute, x86 531
target("prfchw") function attribute, x86 531
target("ptwrite") function attribute, x86 ... 531
target("rdpid") function attribute, x86 531
target("rdrnd") function attribute, x86 531
target("rdseed") function attribute, x86 531
target("recip") function attribute, x86 534
target("recip-precision") function attribute, PowerPC 519
target("rtm") function attribute, x86 532
target("sahf") function attribute, x86 532
target("sgx") function attribute, x86 532
target("sha") function attribute, x86 532
target("shstk") function attribute, x86 532
target("sse") function attribute, x86 532
target("sse2") function attribute, x86 532
target("sse3") function attribute, x86 532
target("sse4") function attribute, x86 532
target("sse4.1") function attribute, x86 532
target("sse4.2") function attribute, x86 532
target("sse4a") function attribute, x86 532
target("ssse3") function attribute, x86 532
target("string") function attribute, PowerPC 519
target("tbm") function attribute, x86 532
target("thumb") function attribute, ARM 503
target("tune=*TUNE*") function attribute, PowerPC 520
target("tune=*TUNE*") function attribute, x86 534

target("update") function attribute, PowerPC ... 519
target("vaes") function attribute, x86 ... 532
target("vpclmulqdq") function attribute, x86 ... 532
target("vsx") function attribute, PowerPC ... 519
target("waitpkg") function attribute, x86 ... 533
target("wbnoinvd") function attribute, x86 ... 533
target("xop") function attribute, x86 ... 533
target("xsave") function attribute, x86 ... 533
target("xsavec") function attribute, x86 ... 533
target("xsaveopt") function attribute, x86 ... 533
target("xsaves") function attribute, x86 ... 533
target-dependent options ... 245
target_clones function attribute ... 494
TC1 ... 5
TC2 ... 5
TC3 ... 5
tda variable attribute, V850 ... 548
Technical Corrigenda ... 5
Technical Corrigendum 1 ... 5
Technical Corrigendum 2 ... 5
Technical Corrigendum 3 ... 5
template instantiation ... 818
temporaries, lifetime of ... 878
tentative definitions ... 222
tgamma ... 641
tgammaf ... 641
tgammal ... 641
thiscall function attribute, x86-32 ... 525
Thread-Local Storage ... 810
thunks ... 455
TILE-Gx options ... 396
TILEPro options ... 396
tiny data section on the H8/300H and H8S ... 544
tiny type attribute, MeP ... 557
tiny variable attribute, MeP ... 545
tiny_data variable attribute, H8/300 ... 544
tls-dialect= function attribute, AArch64 ... 498
tls_model variable attribute ... 541
TLS ... 810
TMPDIR ... 436
toascii ... 641
tolower ... 641
toupper ... 641
towlower ... 641
towupper ... 641
traditional C language ... 207
transparent_union type attribute ... 554
trap_exit function attribute, SH ... 524
trapa_handler function attribute, SH ... 524
trunc ... 641
truncf ... 641
truncl ... 641
tune= function attribute, AArch64 ... 498
two-stage name lookup ... 877
type alignment ... 564
type attributes ... 548
type-diff GCC_COLORS capability ... 65
type_info ... 817
typedef names as function parameters ... 872
typeof ... 459

U

uhk fixed-suffix ... 464
UHK fixed-suffix ... 464
uhr fixed-suffix ... 464
UHR fixed-suffix ... 464
uk fixed-suffix ... 464
UK fixed-suffix ... 464
ulk fixed-suffix ... 464
ULK fixed-suffix ... 464
ULL integer suffix ... 461
ullk fixed-suffix ... 464
ULLK fixed-suffix ... 464
ullr fixed-suffix ... 464
ULLR fixed-suffix ... 464
ulr fixed-suffix ... 464
ULR fixed-suffix ... 464
uncached type attribute, ARC ... 556
undefined behavior ... 885
undefined function value ... 885
underscores in variables in macros ... 459
union ... 809
union, casting to a ... 476
unions ... 873
unknown pragmas, warning ... 90
unresolved references and '-nodefaultlibs' ... 212
unresolved references and '-nostdlib' ... 212
unused function attribute ... 495
unused label attribute ... 558
unused type attribute ... 555
unused variable attribute ... 541
upper function attribute, MSP430 ... 515
upper variable attribute, MSP430 ... 547
ur fixed-suffix ... 464
UR fixed-suffix ... 464
use_debug_exception_return function attribute, MIPS ... 513
use_shadow_register_set function attribute, MIPS ... 513
used function attribute ... 495
used variable attribute ... 541
User stack pointer in interrupts on the Blackfin ... 505

V

'V' in constraint ... 586
V850 Options ... 396
vague linkage ... 816
value after longjmp ... 622
variable addressability on the M32R/D ... 545
variable alignment ... 564
variable attributes ... 536

variable number of arguments 471
variable-length array in a structure 470
variable-length array scope 470
variable-length arrays 470
variables in specified registers 621
variables, local, in macros 459
variadic functions, pointer arguments 473
variadic macros 471
VAX options 399
`vector` function attribute, RX 522
vector types, using with x86 intrinsics 629
`vector_size` type attribute 555
`vector_size` variable attribute 541
`version_id` function attribute, IA-64 508
`vfprintf` 641
`vfscanf` 641
`visibility` function attribute 495
`visibility` type attribute 556
`visibility` variable attribute 542
Visium options 399
VLAs 470
`vliw` function attribute, MeP 510
void pointers, arithmetic 472
void, size of pointer to 472
volatile access 566, 815
`volatile` applied to function 477
volatile `asm` 571
volatile read 566, 815
volatile write 566, 815
`vprintf` 641
`vscanf` 641
`vsnprintf` 641
`vsprintf` 641
`vsscanf` 641
vtable 816
VxWorks Options 400

W

`w` floating point suffix 462
`W` floating point suffix 462
`wakeup` function attribute, MSP430 515
`warm` function attribute, NDS32 516
`warn_if_not_aligned` type attribute 550
`warn_if_not_aligned` variable attribute 537
`warn_unused` type attribute 822
`warn_unused_result` function attribute 496
warning for comparison of signed and unsigned values 109
warning for overloaded virtual function 59
warning for reordering of member initializers ... 56
warning for unknown pragmas 90
`warning` function attribute 482
`warning GCC_COLORS` capability 65
warning messages 70
warnings from system headers 99
warnings vs errors 883
`weak` function attribute 497
`weak` variable attribute 542
`weakref` function attribute 497
whitespace 872
Windows Options for x86 425

X

x86 named address spaces 468
x86 Options 401
x86 Windows Options 425
'`X`' in constraint 588
X3.159-1989 5
Xstormy16 Options 426
Xtensa Options 426

Y

`y0` 641
`y0f` 641
`y0l` 641
`y1` 641
`y1f` 641
`y1l` 641
`yn` 641
`ynf` 641
`ynl` 641

Z

`zda` variable attribute, V850 548
zero-length arrays 468
zero-size structures 470
zSeries options 428