

# Tuple Representation for GIMPLE

Richard Henderson   Aldy Hernandez   Andrew Macleod   Diego Novillo

April 10, 2007

## 1 Introduction

One of the early implementation strategies used for the GIMPLE representation was to use the same internal data structures used by Front Ends to represent parse trees. This simplified implementation because we could leverage existing functionality and interfaces.

However, GIMPLE is a much more restrictive representation than abstract syntax trees (AST). Therefore, it does not require the full structural complexity provided by the main `tree` data structure. In this document, we propose a new set of internal data structures that more closely reflect the tuple-like characteristics of GIMPLE.

The major design goal of this project is to reduce the memory footprint by using compact data structures. A secondary design goal is to simplify the streaming of these data structures by making them as flat as possible (minimizing the use of pointers and indirections).

The changes required in the middle end will be pervasive, so we expect this to be a multi-month effort. The implementation will be carried on the SVN branch `gimple-tuples-branch`. The following is a list of the major work items that will need to be done:

- Define data structures to represent GIMPLE statements (this document). GIMPLE statements will use the new `gimple` type. Both High and Low GIMPLE will use the same basic type.
- Re-implement the gimplifier to use an expansion approach instead of the current tree rewriting mechanism.
- Re-implement every SSA optimizer, operand scanner and alias analysis to use `gimple`.
- Extract functionality from helpers like `fold()` to operate on GIMPLE.
- Rewrite out-of-SSA and RTL expansion to operate on GIMPLE directly. This may be a good opportunity to remove the expression reconstruction done in the SSA $\rightarrow$ normal conversion. However, the initial implementation will very likely build new `trees` out of GIMPLE and feed that into RTL expansion.

In this first phase, we have purposely left out the specification of a type system for GIMPLE. The operands of every GIMPLE instruction (memory references, constants, symbols and SSA names) are going to be **trees**. This is only done to simplify the implementation effort. The conversion to tuples will be complex enough, and the type system can always be defined later.

## 2 Tuple representation

GIMPLE instructions are tuples of variable size divided in two groups: a header describing the instruction and its locations, and a variable length body with all the operands.

<code>code</code>	16 bits	header (6 words)
<code>subcode</code>	16 bits	
<code>next</code>	word	
<code>prev</code>	word	
<code>bb</code>	word	
<code>locus</code>	word	
<code>block</code>	word	body (variable)
<code>op<sub>0</sub></code>	word	
<code>...</code>	<code>...</code>	
<code>op<sub>N</sub></code>	word	

`code`

Main identifier for a GIMPLE instruction.

`subcode`

Used to distinguish different variants of the same basic instruction.

`next`

Pointer to the next instruction.

`prev`

Pointer to the previous instruction.

`bb`

Basic block holding the instruction.

`locus`

Source file/line information.

`block`

Parent lexical block.

`op0 ... opN`

Operands.

It is possible to further reduce the size of the header by replacing `locus` and `block` with the instruction locator scheme used by RTL. This should be straightforward to implement once the main conversion is complete.

We distinguish two main forms of GIMPLE, namely High and Low. The main difference between the two is that High GIMPLE allows lexical scopes and high-level control structures like `try/catch` and `try/finally`. In Low GIMPLE these constructs are converted into the corresponding conditional jump representation.

The `subcode` field will have different uses depending on the `code` of the instruction. In principle, the sub-code is used to distinguish instructions of the same family. However, some instructions are used so frequently that it is convenient to give them a different code instead of using the `code:subcode` combination.

Promoting subcodes to first level codes allows for quicker recognition of commonly used instructions. The following statistics on GIMPLE instructions is based on a set of C and C++ applications (GCC, libstdc++, SPEC 2000, MICO, TRAMP3D, DLV and MICO), totaling around 11M lines of pre-processed C and 14M lines of pre-processed C++.

GIMPLE statement	Frequency
GIMPLE_MODIFY_STMT	51.13%
LABEL_EXPR	19.75%
PHI_NODE	12.56%
COND_EXPR	10.92%
CALL_EXPR	4.64%
RETURN_EXPR	0.81%
SWITCH_EXPR	0.13%
NOP_EXPR	0.01%
RESX_EXPR	0.01%

Only the instructions with a non-zero frequency were included in the previous table. Instructions were counted inside the SSA verification routines, so the raw numbers do not reflect the actual number of instructions. However, every instruction was counted the same number of times, so the frequencies are consistent.

The relative frequencies are used as a popularity index. Instructions used often (most notably, `GIMPLE_MODIFY_STMT`) are encoded to minimize the effort required to recognize and handle them.

### 3 GIMPLE statements

Figure 1 briefly describes the proposed GIMPLE instruction set, the level of each instruction and the section documenting the instruction.

Instruction names reflect the existing `tree` code names and are prefixed with `GS_` to prevent confusion with existing `tree` codes.

Instruction	High GIMPLE	Low GIMPLE	Section
GS_ASM	✓	✓	3.2
GS_ASSIGN	✓	✓	3.3
GS_ASSIGN_COPY	✓	✓	3.3
GS_BIND	✓		3.4
GS_CALL	✓	✓	3.5
GS_CATCH	✓		3.6
GS_COND	✓	✓	3.7
GS_COND_NE	✓	✓	3.7
GS_COND_EQ	✓	✓	3.7
GS_EH_FILTER	✓		3.8
GS_GOTO	✓	✓	3.9
GS_LABEL	✓	✓	3.10
GS_NOP	✓	✓	3.11
GS_OMP_CONTINUE	✓	✓	4.1
GS_OMP_CRITICAL	✓	✓	4.2
GS_OMP_FOR	✓	✓	4.3
GS_OMP_MASTER	✓	✓	4.4
GS_OMP_ORDERED	✓	✓	4.5
GS_OMP_PARALLEL	✓	✓	4.6
GS_OMP_RETURN	✓	✓	4.7
GS_OMP_SECTIONS	✓	✓	4.8
GS_OMP_SECTION	✓	✓	4.9
GS_OMP_SINGLE	✓	✓	4.10
GS_PHI	✓	✓	3.12
GS_RESX		✓	3.13
GS_RETURN	✓	✓	3.14
GS_SWITCH	✓	✓	3.15
GS_TRY	✓		3.16

Table 1: GIMPLE instruction set

### 3.1 Data structure hierarchy

Data structures used to represent the different instruction tuples are organized into a simple hierarchy as depicted in Figure 1. The root of the hierarchy (`struct gimple_statement_base`) contains the fields that make up the tuple header. The rest of the hierarchy is divided in three main classes:

1. Tuples with no operands are instructions that either contain no operands, or the operands are not under control of the operand scanner. The latter is a historical consequence of the way that PHI nodes are handled. The operand scanner uses additional arrays to represent operands in every statement, except inside PHI nodes. This will be addressed during implementation, but for now this hierarchy considers `GS_PHI` a “no operand” instruction.
2. Tuples with operands require additional storage to maintain immediate uses information. These are in turn divided into tuples with only real (or register) operands (`gimple_statement_with_ops`) and tuples that are allowed to have real and memory operands (`gimple_statement_with_memory_ops`).  
This structural division between register and memory operands is important because there is additional storage required to represent memory operands. The statistics gathered over the programs mentioned in the previous section show that about 43% of all statements contain 0 or more register operands.
3. Tuples for OpenMP representation (`gimple_statement_omp`).

All the different structures are collected inside union `gimple_statement_d`.

```
struct gimple_statement_base
{
  unsigned int code : 16;
  unsigned int subcode : 16;
  struct gimple_statement_base *next;
  struct gimple_statement_base *prev;
  struct basic_block_def *bb;
  source_locus locus;
  tree block;
};
```

```
struct gimple_statement_with_ops
{
  struct gimple_statement_base base;
  unsigned modified : 1;
  bitmap addresses_taken;
```

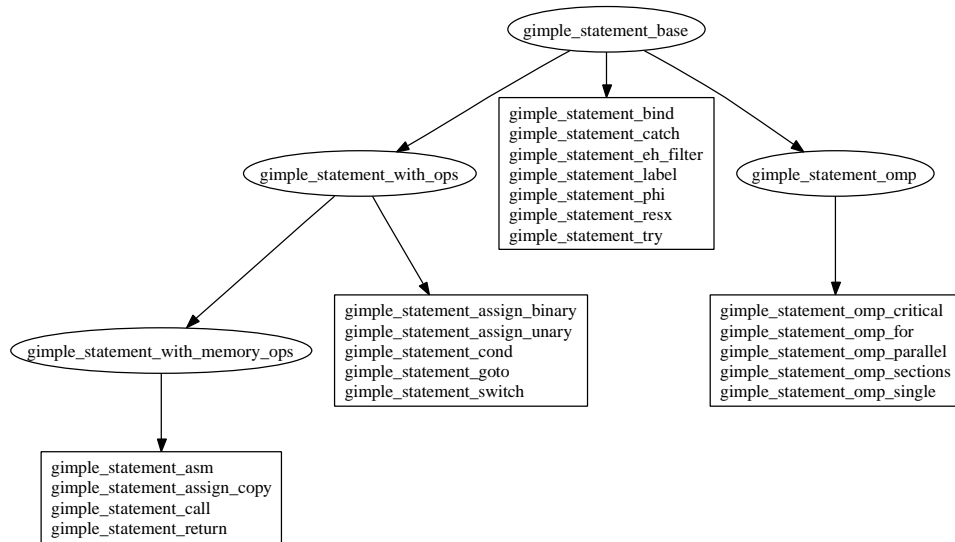


Figure 1: Tuple hierarchy

```

struct def_optype_d *def_ops;
struct use_optype_d *use_ops;
};

struct gimple_statement_with_memory_ops
{
    struct gimple_statement_with_ops base;
    unsigned has_volatile_ops : 1;
    struct voptype_d *vdef_ops;
    struct voptype_d *vuse_ops;
    bitmap stores;
    bitmap loads;
};

struct gimple_statement_omp
{
    struct gimple_statement_base base;
    gimple body;
};

union gimple_statement_d
{
    struct gimple_statement_base base;

```

```

struct gimple_statement_with_ops with_ops;
struct gimple_statement_with_memory_ops with_mem_ops;
struct gimple_statement_omp omp;
struct gimple_statement_bind bind;
struct gimple_statement_catch catch;
struct gimple_statement_eh_filter eh_filter;
struct gimple_statement_label label;
struct gimple_statement_phi phi;
struct gimple_statement_resx resx;
struct gimple_statement_try try;
struct gimple_statement_assign_binary assign_binary;
struct gimple_statement_assign_binary assign_unary;
struct gimple_statement_cond cond;
struct gimple_statement_goto goto;
struct gimple_statement_switch switch;
struct gimple_statement_asm asm;
struct gimple_statement_assign_copy assign_copy;
struct gimple_statement_call call;
struct gimple_statement_return return;
struct gimple_statement_omp_critical critical;
struct gimple_statement_omp_for for;
struct gimple_statement_omp_parallel parallel;
struct gimple_statement_omp_sections sections;
struct gimple_statement_omp_single single;
};

typedef union gimple_statement_d *gimple;

```

The rest of the document describes the data structure used for every statement in Table 1. Access or macros and/or inlined functions will be used to manipulate and query all the fields in the structures.

### 3.2 GS\_ASM

code	GS_ASM
subcode	$\emptyset$
op <sub>0</sub>	ni: number of inputs
op <sub>1</sub>	no: number of outputs
op <sub>2</sub>	nc: number of clobbers
op <sub>3</sub> ... op <sub>ni+2</sub>	inputs
op <sub>ni+3</sub> ... op <sub>ni+no+2</sub>	outputs
op <sub>ni+no+3</sub> ... op <sub>ni+no+nc+2</sub>	clobbers

```

struct gimple_statement_asm
{
  struct gimple_statement_with_memory_ops base;

```

```

const char *string;
unsigned ni;
unsigned no;
unsigned nc;
tree GTY ((length ("(unsigned) %h.ni"))) op[1];
};

```

### 3.3 GS\_ASSIGN

```

code      GS_ASSIGN
subcode   unary or binary expression code. For simplicity, the same codes
          used by tree *_EXPR nodes will be used.
op0    LHS of assignment
op1    RHS of assignment

```

Since assignments are the most frequently used instructions, some of the sub-codes of assignments may be worth encoding as first-level codes. The following are the relative frequencies of the various types of assignments.

Type of assignment	Frequency
copies	52.55%
C = A + B	10.47%
C = NOP_EXPR	9.61%
All others	27.37%

Copy assignments are so common, that they are represented using the first-level code `GS_ASSIGN_COPY`. It may be worth exploring if other sub-codes (e.g., `PLUS_EXPR`, `NOP_EXPR`) are worth moving up. For now, only copies are the obvious candidates.

```

struct gimple_statement_assign_expr
{
  struct gimple_statement_with_ops base;
  tree op[3];
};

struct gimple_statement_assign_unary
{
  struct gimple_statement_with_ops base;
  tree op[2];
};

struct gimple_statement_assign_copy
{
  struct gimple_statement_with_memory_ops base;
  tree op[2];
};

```

1. All the assignments that have a binary expression on the RHS are of type `struct gimple_statement_assign_expr`.
2. Assignments with unary expressions on the RHS are of type `struct gimple_statement_assign_unary`.
3. Copy assignments are of type `struct gimple_statement_assign_copy`.

### 3.4 GS\_BIND

Local variable declaration.

```
code      GS_BIND
subcode   0
op0       variables
op1       body
```

```
struct gimple_statement_bind
{
  struct gimple_statement_base base;
  tree vars;
  gimple body;
};
```

### 3.5 GS\_CALL

```
code      GS_CALL
subcode   bitmask: CALL_EXPR_* attributes
op0       LHS
op1       function
op2       static chain
op3       na: number of arguments
op4... opna+3 arguments
```

```
struct gimple_statement_call
{
  struct gimple_statement_with_memory_ops base;
  tree lhs;
  tree fn;
  tree chain;
  long nargs;
  tree GTY ((length ("%h.nargs"))) args[1];
};
```

### 3.6 GS\_CATCH

```
code      GS_CATCH
subcode   0
op0       catch types
op1       handler
```

```

struct gimple_statement_catch
{
  struct gimple_statement_base base;
  tree types;
  gimple handler;
};

```

### 3.7 GS\_COND

Since conditionals are fairly frequent, the most frequent subcodes of `GS_COND` are encoded as separate first-level codes.

```

code      GS_COND
subcode   NE, EQ, LE, LT, GE, GT
op0    LHS of comparison
op1    RHS of comparison
op2    true branch target
op3    false branch target

```

The relative frequencies for the different types of conditionals are:

Conditional	Frequency
GS_COND : NE	44.75%
GS_COND : EQ	32.74%
GS_COND : GT	7.50%
GS_COND : LE	4.81%
GS_COND : SINGLE	3.76%
GS_COND : GE	3.35%
GS_COND : LT	3.10%

Given the high frequency of subcodes `NE` and `EQ`, they are moved to first-level codes as `GS_COND_NE` and `GS_COND_EQ`.

To avoid special casing unary conditionals (`SINGLE` in the previous table) of the form `if (SSA_NAME)` or `if (*DECL)`, they will be canonicalized into the corresponding binary `NE` conditional.

```

struct gimple_statement_cond
{
  struct gimple_statement_with_ops base;
  tree op[2];
  struct gimple_statement_label *true;
  struct gimple_statement_label *false;
};

```

### 3.8 GS\_EH\_FILTER

```
code      GS_EH_FILTER
subcode   bitmask: { EH_FILTER_MUST_NOT_THROW }
op0       filter types
op1       failure actions
```

```
struct gimple_statement_eh_filter
{
  struct gimple_statement_base base;
  tree types;
  gimple failure;
};
```

### 3.9 GS\_GOTO

```
code      GS_GOTO
subcode   {}
op0       destination
```

```
struct gimple_statement_goto
{
  struct gimple_statement_with_ops base;
  tree dest;
};
```

### 3.10 GS\_LABEL

```
code      GS_LABEL
subcode   {}
op0       label
```

```
struct gimple_statement_label
{
  struct gimple_statement_base base;
  tree label;
};
```

### 3.11 GS\_NOP

```
code      GS_NOP
subcode   {}
```

```
struct gimple_statement_nop
{
  struct gimple_statement_base base;
};
```

### 3.12 GS\_PHI

```
code      GS_PHI
subcode   ∅
op0     result
op1     na: number of arguments
op1... opna arguments
```

```
struct gimple_statement_phi
{
  struct gimple_statement_base base;
  unsigned capacity;
  unsigned nargs;
  tree result;
  struct phi_arg_d GTY ((length ("(unsigned) %h.nargs"))) args[1];
};
```

### 3.13 GS\_RESX

```
code      GS_RESX
subcode   ∅
op0     exception region number
```

```
struct gimple_statement_resx
{
  struct gimple_statement_base base;
  int region;
};
```

### 3.14 GS\_RETURN

Note that this instructions is not really needed. The return value can be encoded as an assignment into `RESULT_DECL`. It may be worth removing in the future.

```
code      GS_RETURN
subcode   non-zero if using RESULT_DECL
op0     return value
```

```
struct gimple_statement_return
{
  struct gimple_statement_memory_ops base;
  tree retval;
};
```

### 3.15 GS\_SWITCH

```
code      GS_SWITCH
subcode   n1: number of labels
op0     switch index
op1     default label
op2... opn1 labels
```

```
struct gimple_statement_switch
{
  struct gimple_statement_with_ops base;
  tree index;
  tree default_label;
  tree GTY ((length ("(unsigned) %h.base.subcode + 1"))) labels[1];
};
```

### 3.16 GS\_TRY

```
code      GS_TRY
subcode   CATCH, FINALLY
op0     expression to evaluate
op1     cleanup expression
```

```
struct gimple_statement_try
{
  struct gimple_statement_base base;
  gimple eval;
  gimple cleanup;
};
```

## 4 OpenMP

Note that OpenMP clauses are probably easiest to keep as trees. Since they get lowered fairly early, converting them to gimple structures may not be worth the effort.

### 4.1 GS\_OMP\_CONTINUE

```
code      GS_OMP_CONTINUE
subcode   ∅
  Uses struct gimple_statement_base.
```

### 4.2 GS\_OMP\_CRITICAL

```
code      GS_OMP_CRITICAL
subcode   ∅
op0     body
op1     critical section name
```

---

```
struct gimple_statement_omp_critical
{
  struct gimple_statement_omp base;
  tree name;
};
```

### 4.3 GS\_OMP\_FOR

```
code      GS_OMP_FOR
subcode   predicate: { LT, GT, LE, GE }
op1     clauses
op2     index variable
op3     n1: initial value
op4     n2: final value
op5     incr: increment
op6     pre-body evaluated before the loop body begins
```

```
struct gimple_statement_omp_for
{
  struct gimple_statement_omp base;
  tree clauses;
  tree index;
  tree inital;
  tree final;
  tree incr;
  gimple pre_body;
};
```

### 4.4 GS\_OMP\_MASTER

```
code      GS_OMP_MASTER
subcode   ∅
op0     body
  Uses struct gimple_statement_omp.
```

### 4.5 GS\_OMP\_ORDERED

```
code      GS_OMP_ORDERED
subcode   ∅
op0     body
  Uses struct gimple_statement_omp.
```

#### 4.6 GS\_OMP\_PARALLEL

```
code      GS_OMP_PARALLEL
subcode   0
op1       clauses
op2       child function
op3       shared data argument
```

```
struct gimple_statement_omp_parallel
{
  struct gimple_statement_omp base;
  tree clauses;
  tree child_fn;
  tree data_arg;
};
```

#### 4.7 GS\_OMP\_RETURN

```
code      GS_OMP_CRITICAL
subcode   bitmask: { OMP_RETURN_NOWAIT }
          Uses struct gimple_statement_base.
```

#### 4.8 GS\_OMP\_SECTIONS

```
code      GS_OMP_SECTIONS
subcode   0
op0       body
op1       clauses
```

```
struct gimple_statement_omp_sections
{
  struct gimple_statement_omp base;
  tree clauses;
};
```

#### 4.9 GS\_OMP\_SECTION

```
code      GS_OMP_SECTION
subcode   bitmask: { OMP_SECTION_LAST }
op0       body
          Uses struct gimple_statement_omp.
```

#### 4.10 GS\_OMP\_SINGLE

```
code      GS_OMP_SINGLE
subcode   0
op0       body
op1       clauses
```

```
struct gimple_statement_omp_single
{
  struct gimple_statement_omp base;
  tree clauses;
};
```