Porting GCC Compiler
Part I : How GCC works?

Choi, Hyung- Kyu
hectoct@altair.snu.ac.kr

July 17, 2003

Microprocessor Architecture and System Software Laboratory
About this presentation material

- Basically this document is based on GCC 2.95.2
  - Some examples are from Calm32 (Samsung) port of GCC 2.95.2
- This document has been updated to follow up GCC 3.4.
Main Goal of GCC

- Make a good, fast compiler
- for machines on which the GNU system aims to run including GNU/Linux variants
  - *int* is at least a 32-bit type
  - have several general registers
  - with a flat (non-segmented) byte addressed data address space (the code address space can be separate).
  - Target *ABI* (application binary interfaces) may have 8, 16, 32 or 64-bit int type. char can be wider than 8 bits
- Elegance, theoretical power and simplicity are only secondary
GCC Compilation System

- Compilation system includes the phases
  - Preprocessor
  - Compiler
  - Optimizer
  - Assembler
  - Linker

- Compiler Driver coordinates these phases.
GCC Execution

gcc

Input file
output file

cpp
cc1
g++
gas (assembler)
Id (linker)
The Structure of GCC Compiler

- Parsing
- TREE
- RTL
- Global Optimizations
  - Jump Optimization
  - Common Subexpr. Elimination
  - Loop Optimization
  - Data Flow Analysis
- Instruction Combining
- Instruction Scheduling
- Register Class Preferencing
- Register Allocation
- Peephole Optimizations
- Assembly

Languages supported:
- C
- C++
- ObjC
- Fortran

Machine Description
Macro Definition

Microprocessor Architecture and System Software Laboratory
GCC Compiler flow

For each function:
- Parser
- Tree Optimization
- RTL Generation
- RTL Optimization
- Assembler Code Generation

IR: Tree Representation
IR: Register Transfer Language (RTL)

Assembler Code with or w/o debugging information
Intermediate Language (1/2)

- RTL (Register Transfer Language)
- Used to describe *insns* (instructions)
- Written in LISP-like form

```
(set (mem:SI (reg: SI 54))
  (reg:SI 53))
```

Above RTL statement means “set memory pointed by register 54 with value in register 53”

i.e. `st [r54], r53`

destination    source
Example of RTL

**(plus:SI (reg:SI 55) (const_int -1))**

- Adds two 4-byte integer (SI mode) operands.
- First operand is register
  - Register is also 4-byte integer.
  - Register number is 55.
- Second operand is constant integer.
  - Value is “-1”.
  - Mode is VOIDmode (not given).
Intermediate Language:
machine mode

- **BI mode**
  - a single bit, for predicate registers

- **[QI/HI/SI/DI/TI/OI] mode**
  - Quarter-Integer(1bytes)
  - Half-Integer(2bytes)
  - Single Integer(4bytes)
  - Double Integer(8bytes)
  - Tetra Integer(16bytes)
  - Octa Integer(32bytes)

- **PSI mode**
  - Partial single integer mode which occupies for bytes but doesn’t really use all four. e.g. pointers on some machines

- And many other machine mode such as float-point mode, complex mode and etc.
Three Main Conversions in the compiler

- The front end reads the source code and builds a `parse tree`.
- The `parse tree` is used to generate an RTL `insn` list based on `named instruction patterns`.
- The `insn` list is matched against the RTL templates to produce `assembler code`.
Name, RTL pattern and Assembler code

- Tree has pre-defined standard names.

Parse Tree -> Register Transfer Language (RTL) -> Assembler code
Optimizations

- Tree optimization
  - tree based inlining
  - constant folding
  - some arithmetic simplification

- RTL optimization
  - performs many well known optimizations
  - e.g. jump optimization, common subexpression elimination (CSE), loop optimization, if-conversion, instruction scheduling, register allocation and etc

*Microprocessor Architecture and System Software Laboratory*
How to port?

For each function:
- Parser
- Tree Optimization
- RTL Generation
- RTL Optimization
- Assembler Code Generation

Should we modify these all?

NO!
Then how? (1/2)

- Just write below 3 files in `<gcc_root>/gcc/config/machine/
  - `machine.md`: machine description
  - `machine.h`: target description macros
  - `machine.c`: user-defined functions used in `machine.md` and `machine.h`
- e.g. SPARC
  - in `<gcc_root>/gcc/config/sparc/`
  - `sparc.md`, `sparc.h`, `sparc.c`
Then how? (2/2)

- Then let Makefile does below two jobs
  - Generate some .c and .h files from machine description(*machine*.md) file
  - Then actually compile .c and .h files including generated files.
Build process (1/2)

- insn-flag.h
- insn-codes.h
- insn-emit.c
- insn-recog.c
- insn-extract.c
- insn-attr.h
- insn-atrtrtab.c
- insn-output.c

- genflags
- gencodes
- genemit
- genemit
- genoutput
- genconfig
- genrecog
- genextract
- genattr
- genattrtab

machine.md

machine.h
machine.c

Microprocessor Architecture and System Software Laboratory
Build process (2/2)

- RTL generation
- RTL Optimization
- Assembler Code Generation
- Debugging information Generation

- insn-recog.c
- insn-extract.c
- insn-attr.h
- insn-attrtab.h
- insn-emit.c
- insn-output.c
- insn-flags.h
- insn-codes.h

- machine.h
- machine.md

DBX, SDB, DWARF, DWARF2, VMS are supported

Microprocessor Architecture and System Software Laboratory
Machine Description

- *machine*.md contains machine description

- Machine Description
  - CPU description
    - Functional Units, Latency and etc
  - RTL Patterns
    - Used when convert Tree into RTL
    - All kind of RTL Patterns which can be generated
  - Assembler mnemonic
  - etc.
Target Description Macro

- `machine.h` contains target description macros
- **Target Description Macro**
  - Storage layout
    - alignment, endian, structure padding and etc
  - ABI (Application Binary Interface)
    - calling convention, stack layout and etc
  - Register usage
    - allocation strategy, how value fit in registers and etc
  - Defining output assembler language
  - Controlling Debugging Information Format
  - Library supports
  - etc.
Porting GCC Compiler
Part II: In details

Choi, Hyung-Kyu
hectoct@altair.snu.ac.kr

July 17, 2003
Microprocessor Architecture and System Software Laboratory
RTL Generation

- Convert parse tree into RTL *insn* list based on *named instruction patterns*

**How?**

- Tree has pre-defined standard pattern names
  - e.g. “addsi3”, “movsi” and etc.
  - Example of standard pattern name
    - “addsi3” : which means “add op2 and op1, and storing result in op0, where all operands have SI mode”

- For each standard pattern name, generate RTL *insn* list defined in *machine.md*
RTL Generation Example 1

Machine description: define_insn

```
(defineInsn "adde"
  [(set (match_operand:SI 0 "arith_reg_operand" ":=r")
    (plus:SI (plus:SI (match_operand:SI 1 "arith_reg_operand" ":0")
      (match_operand:SI 2 "arith_reg_operand" ":r"))
    (reg:SI 18)))
  (set (reg:SI 18))
  (ltu:SI (plus:SI (match_dup 1) (match_dup 2))
    (match_dup 1))])

"ADC\t%0,%2"

[(set_attr "type" "arith")]
```
RTL Generation Example 1a

Standard name in Tree

In addsi3.c

```c
int i;
int
main()
{
    i = i + 1;
}
```

movsi ; r54 ← #i
movsi ; r55 ← #i
movsi ; r56 ← mem(r55)
addsi3 ; r57 ← r56 + 1
movsi ; mem(r54) ← r57
RTL Generation Example 1b

Find RTL pattern by name defined in .md

In `machine.md`

```c
(define_insn "addsi3"
 [([set (match_operand:SI 0
 "arith_reg_operand" ";r,r")
   (plus:SI (match_operand:SI 1
 "arith_operand" ";%0,0")
           (match_operand:SI 2
 "arith_operand" ";r,I")))
   (clobber (reg:SI 18))]
 ""
 ... )
```

In `insn-emit.c`

```c
rtx
gen_addsi3 (operand0, operand1, operand2)

rtx operand0; rtx operand1; rtx operand2;
{
  return gen_rtx_PARALLEL (VOIDmode,
                          gen_rtvec (2,
                                     gen_rtx_SET (VOIDmode, operand0,
                                                  gen_rtx_PLUS (SImode, operand1, operand2)),
                                     gen_rtx_CLOBBER (VOIDmode,
                                                      gen_rtx_REG (SImode, 18))));
}
```

Microprocessor Architecture and System Software Laboratory
Generate RTL from Tree

movsi ; r54 ← #i
movsi ; r55 ← #i
movsi ; r56 ← mem(r55)
addsi3 ; r57 ← r56 + 1
movsi ; mem(r54) ← r57

In addsi3.c.rtl

(insn 14 13 16 (parallel[
  (set (reg:SI 57)
    (plus:SI (reg:SI 56)
      (const_int 1 [0x1])))
    (clobber (reg:SI 18 t))
  ] ) -1 (nil)
(nil))
RTL Generation Example 2

Machine description: define_expand

```
(define_expand "zero_extendhisi2"

([(set (match_operand:SI 0 "register_operand" "")
    (and:SI (subreg:SI
              (match_operand:HI 1 "register_operand" "")
              0)
              (match_dup 2)))]
   ""
   "operands[2]=force_reg(SImode,GEN_INT(65535));")
```

Name

RTL Template

Condition (optional)

Preparation statement

Microprocessor Architecture and System Software Laboratory
We can generate RTL sequences from standard name while generating RTL patterns.
- by using "define_expand" instead of "define_insn"
RTL Generation Example 3

Machine description: define_split

```latex
(define_split
[(set(match_operand:DI 0 "arith_reg_operand" "=r")
  (plus:DI(match_operand:DI 1 "arith_reg_operand" "%0")
   (match_operand:DI 2 "arith_reg_operand" "r" )))
 (clobber (reg:SI 18))]
"reload_completed"
[. . .]
{ . .
  DONE;}
"
```
### RTL Generation Example 3a

- We can also split generated insn pattern into new insn patterns.
  - by using "define_split" instead of "define_insn"

```c
long long i;

int main()
{
    i = i+1;
}
```

```c
In adddi3.c

movdi ; r54 ← #i
movdi ; r55 ← #I
movdi ; r56 ← mem(r55)
movdi ; r57 ← const 1
adddi3 ; let's split
movdi ; mem(r54) ← r58
```

```c
While RTL generation

movdi ; r54 ← #i
movdi ; r55 ← #I
movdi ; r56 ← mem(r55)
movdi ; r57 ← const 1
adddi3 ;
addc ;
movdi ; mem(r54) ← r58
```

### After splitting standard name

```c
movdi ; r54 ← #i
movdi ; r55 ← #I
movdi ; r56 ← mem(r55)
movdi ; r57 ← const 1
adddi3 ;
addc ;
movdi ; mem(r54) ← r58
```
There can be only one name (standard or not) for one unique RTL pattern. But one name can have multiple RTL patterns.

- **Name 1**
  - RTL Pattern 1
  - RTL Pattern 2

- **Name 2**
  - RTL Pattern 3

- **Name 3**
  - RTL Pattern 4
  - RTL Pattern 5

---

**Assembler code**

**Optimizations**

**Register Transfer Language (RTL)**
GCC Compiler flow

For each function → Parser → Tree Optimization → RTL Generation → RTL Optimization → Assembler Code Generation

IR: Tree Representation
IR: Register Transfer Language (RTL)

Microprocessor Architecture and System Software Laboratory
Assembly code Generation

Example 1a

Let’s think previous example

In adddi3.c

```c
long long i;

int main()
{
  i = i+1;
}
```

After splitting standard name
( and just before Assembly code generation)

```asm
movdi ;
movdi ;
movdi ;
movdi ;
addsi3 ; r2 ← r2 + r4
addc ; r1 ← r1 + r3
movdi ;
```
Assembly code Generation

Example 1b

Find asm output by RTL pattern matching

In `machine.md`

```c
(define_insn "addc"

[(set (match_operand:SI 0 "arith_reg_operand" ":=r")
   (plus:SI (plus:SI (match_operand:SI 1 "arith_reg_operand" ":0")
         (match_operand:SI 2 "arith_reg_operand" ":r"))
   (reg:SI 18)))
  (set (reg:SI 18))
  (ltu:SI (plus:SI (match_dup 1) (match_dup 2))
        (match_dup 1)))]

"

"ADC\t%0,%2"

[(set_attr "type" "arith")])
```

In `insn-output.c`

```c
const char * const insn_template[] = {
   . . .
   "ADC\t%0,%2",
   . . .
};
```
Assembly code Generation

Example 1c

Find asm output by RTL pattern matching

After splitting standard name

( and just before Assembler code generation)

```
... addsi3 ; r2 ← r2 + r4
addc ; r1 ← r1 + r3
...```

In adddi3.s

```
... ADD r2, r4
ADC r1, r3
...```
Microprocessor Architecture and System Software Laboratory

GCC Compiler flow

For each function:
- Parser
- Tree Optimization
- RTL Generation
- RTL Optimization

IR: Register Transfer Language (RTL)

Assembler Code Generation

With or w/o debugging information
Optimization and RTL

Example 1a

Let’s think about two different RTL for one standard name

(define_insn "addsi3"
  [(set (match_operand:SI 0 "arith_reg_operand" "=r,r")
      (plus:SI (match_operand:SI 1 "arith_operand" "%0,0")
               (match_operand:SI 2 "arith_operand" "r,I")))
   (clobber (reg:SI 18))]
  ""
  ""
  )
Optimization and RTL
Example 1b

For same example as before

Before optimization

.(.
addsi3
addc
movesi
movesi
.(.

In RTL form

[(set (match_operand:SI 0 "arith_reg_operand" ",r,r")
  (plus:SI (match_operand:SI 1 "arith_operand" ",%0,0")
    (match_operand:SI 2 "arith_operand" ",r,I")))

(clobber (reg:SI 18)])  \(\leftarrow\) Define register 18

[(set (match_operand:SI 0 "arith_reg_operand" ",r")
  (plus:SI (plus:SI (match_operand:SI 1 "arith_reg_operand" ",0")
                (match_operand:SI 2 "arith_reg_operand" ",r"))
    (reg:SI 18)))  \(\leftarrow\) Use register 18

(set (reg:SI 18)
  (ltu:SI (plus:SI (match_dup 1) (match_dup 2)) (match_dup 1))))

Microprocessor Architecture and SystemSoftware Laboratory
Optimization and RTL

Example 1b

- For same example as before

After instruction scheduling
Without (clobber 18 t)

...  
addc  
movsi  
addsi3  
movsi  
...  

Sequence changed!

Wrong !!

After instruction scheduling
With (clobber 18 t)

...  
addsi3  
addc  
movsi  
movsi  
...  

Microprocessor Architecture and System Software Laboratory
You can introduce attributes by using "\texttt{define\_attr}\n\texttt{"
You can assign multiple attributes for each RTL pattern

In \texttt{machine.md}

\begin{verbatim}
(define_attr "\texttt{needs\_delay\_slot}" "yes,no" (const_string "no"))

(define_attr "\texttt{in\_delay\_slot}" "yes,no"
  (cond [(eq_attr "\texttt{type}" "arith") (const_string "no")]
        (const_string "no")))

(define_insn "\texttt{return}"
  [\dots]
  "\ldots"
  "JMPD\tR14%#"
  [(set_attr "\texttt{type}" "return")
   (set_attr "\texttt{needs\_delay\_slot}" "yes")])

(define_insn "\texttt{addc}"
  [\dots]
  ""
  "ADC\t0,%2"
  [(set_attr "\texttt{type}" "arith")])
\end{verbatim}
Set "in_delay_slot" attribute of "addc" to be "no"

In `machine.md`

```lisp
(define_attr "needs_delay_slot" "yes,no" (const_string "no"))

(define_attr "in_delay_slot" "yes,no"
  (cond [(eq_attr "type" "arith") (const_string "no")]
        (const_string "no")))

(define_insn "addc"
  [(set_attr "type" "arith")
   (set_attr "needs_delay_slot" "yes")]
  
  "ADC\t0,0"
)
```

```
(define_insn "return"
  [(set_attr "type" "return")
   (set_attr "needs_delay_slot" "yes")]
  
  "JMPD\tR14%"
)
```
Finally you can specify delay slot scheduling policy by “define_delay”

- e.g. “addc” can not be in delay slot of “return”

In machine.md

```lisp
(define_delay
  (eq_attr "needs_delay_slot" "yes")
  [(eq_attr "in_delay_slot" "yes") (nil) (nil)])

(define_delay
  (eq_attr "type" "return")
  . . )

(define_insn "return"
  . .
  [(set_attr "type" "return")
   (set_attr "needs_delay_slot" "yes")])
```

```
(define_insn "addc"
  . .
  [(set_attr "type" "arith"))]}
```
Porting GCC Compiler
Part III : Other things

Choi, Hyung- Kyu
hectoct@altair.snu.ac.kr

July 17, 2003
Microprocessor Architecture and System Software Laboratory
When generate RTL from Tree

- Find RTL template by name?
  - No, also check machine mode and predicate for operand

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "arith_reg_operand" ";=r,r")
    (plus:SI (match_operand:SI 1 "arith_operand" ";%0,0")
      (match_operand:SI 2 "arith_operand" ";r,I")))]
  ...
)
```

- How and where should we define predicate?
Predicate is C function with 2 arguments defined in `machine.c`

```c
/* Returns 1 if OP is a valid source operand for an arithmetic insn. */

int arith_operand (rtx op, enum machine_mode mode)
{
  if (arith_reg_operand (op, mode))
    return 1;
  if (GET_CODE (op) == CONST_INT && CONST_OK_FOR_I (INTVAL (op)))
    return 1;
  return 0;
}
```
What happens if there is no matching RTL template?
- Automatically convert operand’s machine mode by generating \texttt{movm} pattern to generate RTL
- If fails, just abort!
- e.g. If “addsi3” accepts only register and immediate operands

```c
int i;
int main()
{
    i = i + 1;
}
```

Generated automatically by GCC

```c
movsi ; r54 ← #i
movsi ; r55 ← #i
movsi ; r56 ← mem(r55)
addsi3 ; r57 ← r56 + 1
movsi ; mem(r54) ← r57
```
In this presentation, only flow of GCC Compiler is explained.

No implementation detail

- RTL syntax
- Target Macros in `machine.h`
- Machine descriptions in `machine.md`
- etc
Limitation of GCC

- When new architecture feature is introduced, we can’t porting by this method explained before.
  - You should modify core part of GCC Compiler.
  - e.g. There was no support for “register window”, “delay slot” in old version.
  - e.g. There was no support for 1bit- register before, such as predicate register in Itanium
- You don’t have to consider optimization every time. But sometimes you should consider!
Summary

RTL generation

RTL Optimization

Assembler Code Generation

Debugging information Generation

 insn-recog.c, insn-extract.c, insn-attr.h, insn-attrtab.c

 insn-flags.h
 insn-codes.h
 insn-emit.c

 machine.md

 insn-output.c

 machine.h

 machine.c

 DBX, SDB, DWARF, DWARF2, VMS are supported

Microprocessor Architecture and System Software Laboratory
References

- GCC Internals Manual
  - http://gcc.gnu.org/onlinedocs/
    - Especially Ch.7 ~ Ch.11
- GCC home page
  - http://gcc.gnu.org
- crossgcc (mailing list)
  - archives : http://sources.redhat.com/ml/crossgcc/