GCC Internals: A Conceptual View – Part II

Abhijat Vichare

CFDVS,
Indian Institute of Technology, Bombay

January 2008
Plan

PART I

• GCC: Conceptual Structure
• C Program through GCC
• Building GCC

PART II

• Gimple
• The MD-RTL and IR-RTL Languages in GCC
• GCC Machine Descriptions
Part I

Gimple
The Goals of GIMPLE are

- Lower control flow
  Program = sequenced statements + unrestricted jump
- Simplify expressions, introduce temporary variables as needed
  Typically: two operand assignments!
- Simplify scope
  move local scope to block begin – including temporaries

Notice
Lowered control flow → nearer to register machines + Easier SSA!
Tree manipulation passes (tree-optimize.c)

- **Gimplifier** case analyzes GENERIC nodes, calls corresponding gimplifier.
  \[ \{\text{Gimple}\} = \{\text{AST/Generic}\} - \{\text{Control flow nodes}\} \]
- Node type specific gimplifiers
- Optimization passes on tree representation, and
- Translate to next IR, i.e. RTL
  - Depth first traverse the “input” Gimple representation
  - Generate a **linear list** RTL representation

**But we have a problem ...**
**Problem:** Gimple (m/c indep.) → RTL (m/c specific)!

**To Do:** Implement m/c indep. to m/c dep. translation at \( t_{dev} \)

**Given:** the actual target will be known only at \( t_{build} \)

---

**Diagram:**

- **Choose HLL**
  - HLL Specific Code, per HLL
  - Selected
  - Copied
- **Language and Machine Independent Generic Code**
- **Machine dependent Generator Code**
- **Set of Machine Descriptions**

- **Choose Target MD**
- **Parser**
- **Genericizer**
- **Gimplifier**
- **Tree SSA Optimizer**
- **RTL Generator**
- **Optimizer**
- **Code Generator**

- **Source Program**
- **TT 1**
- **Gimple → RTL**

- **Assembly Program**
- **TT 3**

---

**Time Phases:**

- \( t_{dev} \)
- \( t_{build} \)
- \( t_{op} \)
**Problem:** Gimple (m/c indep.) → RTL (m/c specific)!

**To Do:** Implement m/c indep. to m/c dep. translation at $t_{dev}$

**Given:** the actual target will be known only at $t_{build}$
Target indep. rep. to target dep. rep. in GCC (3:1:9)

MODIFY_EXPR

(set (<dest>) (<src>))
Target indep. rep. to target dep. rep. in GCC

MODIFY_EXPR  "movsi"  (set (<dest>) (<src>))

Standard Pattern Name
Target indep. rep. to target dep. rep. in GCC

- MODIFY_EXPR
- "movsi"
- (set (<dest>) (<src>))

Standard Pattern Name

Separate to generic code and MD

- MODIFY_EXPR
- "movsi"
- "movsi"
- (set (<dest>) (<src>))
Target indep. rep. to target dep. rep. in GCC

(3:1:9)

MODIFY_EXPR  "movsi"  (set (<dest>) (<src>))

Standard Pattern Name

Separate to generic code and MD

MODIFY_EXPR  "movsi"  "movsi"  (set (<dest>) (<src>))

Implement at $t_{dev}$

MODIFY_EXPR  "movsi"  "movsi"  (set (<dest>) (<src>))

Unnecessary in CGF; hard code

Implement in MD at $t_{dev}$
Retargetability ⇒ Multiple MD vs. One CGF!

MODIFY_EXPR

"movsi"

CGF

MD 1
"movsi", (set (<dest>) (<src>))

... 

MD n
"movsi", (set (<dest>) (<src>))
Retargetability ⇒ Multiple MD vs. One CGF!

\[ \text{MD 1} \]

\[ "\text{movsi},(\text{set (<dest>) (<src>))} \]

\[ \vdots \]

\[ \text{MD n} \]

\[ "\text{movsi},(\text{set (<dest>) (<src>))} \]

How at \( t_{\text{build}} \)?
Retargetability ⇒ Multiple MD vs. One CGF!

Basic Approach: Tabulate

GIMPLE – RTL

struct optab [ ]
struct insn_data [ ]

Convert SPNs to indices

How at $t_{build}$?

MODIFY_EXPR

"movsi", (set (<dest>) (<src>))

CGF

MD 1

MD n
Part II

The MD-RTL and IR-RTL Languages in GCC
### RTL Goals and Use

**Goal 1:** Specify target instruction semantics at $t_{dev}$
- **Capture** target instruction semantics as RTXs
  - Use MD constructs and operators → The MD-RTL Language

**Goal 2:** Represent input program at $t_{op}$
  - **Lower** data
  - “Express” the “captured” target semantics in IR
  - **Goal:** Every RTX of last RTL pass = unique ASM string.
    - Use IR constructs and operators → The IR-RTL Language

**Notice**
- Lowered data and procedures → nearer to typical hardware
Goal 1 Example: Specifying target inst. semantics

```
(define_insn
  "movsi"
  (set
    (match_operand 0 "register_operand" "r")
    (match_operand 1 "const_int_operand" "k")
  )

"" /* C boolean expression, if required */
"mov %0, %1"
)
```
Goal 1 Example: Specifying target inst. semantics

Define new inst. pattern

(define_insn
  "movsi"
  (set
    (match_operand 0 "register_operand" "r")
    (match_operand 1 "const_int_operand" "k")
  )
  "mov %0, %1"
"
/* C boolean expression, if required */

(Standard) Pattern Name

RTX: Capture semantics of target inst.

target asm inst. = Concrete syntax for RTX
(define_insn
  "movsi"
  (set
    (match_operand 0 "register_operand" "r")
    (match_operand 1 "const_int_operand" "k")
  )
  "mov %0, %1"
)
From Goal 1 to Goal 2: RTL at $t_{\text{build}}$

**Alert: At $t_{\text{build}}$**

Convert MD-RTL at $t_{\text{develop}}$ to RTL data structures, and compile.

### The Data Structure for RTL Objects (in rtl.h)

```c
struct rtx_def { /* RTL codes (e.g. SET) enum-med from $\text{GCCHOME}/gcc/rtl.def */
    ENUM_BITFIELD(rtx_code) code : 16;
    ENUM_BITFIELD(machine_mode) mode : 8;
    unsigned int jump : 1;

    /* ... a few such flags */
    union u {
        rtunion fld[1];
        HOST_WIDE_INT hwint[1];
    };
};
```

A.Vichare

GCC Internals
(define_insn "movsi"
  (set (op0) (op1))
  ""
  "mov %0, %1")
(define_insn
  "movsi"
  ( set (op0) (op1))
  ""
  "mov %0, %1")

rtx
gen_movsi (rtx operand0, rtx operand1)
{
  ...
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
  ...
}
Goal 1 to Goal 2: Gimple → RTL Translation Table
Conversion of RTL from Textual to Internal Form

(define_insn "movsi"
  (set (op0) (op1))
  ""
  "mov %0, %1")

rtx

rtx gen_movsi (rtx operand0, rtx operand1)
{
  ...
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
  ...
}

A.Vichare
GCC Internals
Goal 1 to Goal 2: Gimple → RTL Translation Table
Conversion of RTL from Textual to Internal Form

(define_insn "movsi"
  (set (op0) (op1))
  ""
  "mov %0, %1")

rtx
gen_movsi (rtx operand0, rtx operand1)
{
  ...
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
  ...
}
Goal 1 to Goal 2: Gimple → RTL Translation Table
Conversion of RTL from Textual to Internal Form

```
(define_insn
  "movsi"
  ( ( set (op0) (op1))
    ""
    "mov %0, %1")

rtx
gen_movsi (rtx operand0, rtx operand1)
{
  ...
  emit_insn (gen_rtx_SET (VOIDmode, op0, op1));
  ...
}
```

IR object to create
Create IR object
Goal 1 to Goal 2: Gimple → RTL Translation Table
Conversion of RTL from Textual to Internal Form

\[ (\text{define_insn} \ "\text{movsi}\" \ (\text{set} \ (\text{op0}) \ (\text{op1})) \ "" \ "\text{mov \ %0, \ %1}\") \]

\[
\text{rtx}
\text{gen_movsi} \ (\text{rtx} \ \text{operand0}, \ \text{rtx} \ \text{operand1})
\begin{cases}
\text{...}
\text{emit_insn} \ (\text{gen_rtx_SET} \ (\text{VOIDmode}, \ \text{op0}, \ \text{op1}));
\text{...}
\end{cases}
\]
RTL at $t_{op}$: Example of RTL in use as IR

\begin{itemize}
  \item (insn 24 22 25 1)
  \begin{itemize}
    \item (set)
      \begin{itemize}
        \item (reg:SI 58 [D.1283])
        \item (const_int 0 [0x0])
      \end{itemize}
    \item -1
    \item (nil)
    \item (nil)
  \end{itemize}
\end{itemize}

See: RTL dump in “C Program through GCC”
RTL at $t_{op}$: Example of RTL in use as IR

\begin{verbatim}
(insn 24 22 25 1
  (set
    (reg:SI 58 [D.1283])
    (const_int 0 [0x0])
  )
  -1
  (nil)
  (nil)
)
\end{verbatim}
RTL at $t_{op}$: Example of RTL in use as IR

Instruction position in sequence

```
(insn 24 22 25 1
  (set
    (reg:SI 58 [D.1283])
    (const_int 0 [0x0])
  )
  -1
  (nil)
  (nil)
)``

Instantiation of RTX specified in MD.

Register #58 matches the corresponding match_operand spec.

Similarly, for the next operand.
RTL at $t_{op}$: Example of RTL in use as IR

\[
{\text{(insn 24 22 25 1)}}
\]
\[
{\text{(set)}}
\]
\[
{\text{(reg:SI 58 [D.1283])}}
\]
\[
{\text{(const_int 0 [0x0])}}
\]
\[
{-1}
\]
\[
{\text{(nil)}}
\]
\[
{\text{(nil)}}
\]

\text{Note:}

RTX is incomplete.

Hard register for pseudoregister #58 unknown (yet).

\text{Allocate: } #58 = eax

RTX is now complete.

\text{Instantiated RTX is target dependent} because the RTX specified in MD captures target instruction semantics.
Part III

GCC Machine Descriptions
Why MD at $t_{dev}$?

MDs are written in MD-RTL

Main Purpose: Gimple → IR-RTL Translation at $t_{run}$

<table>
<thead>
<tr>
<th>Gimple</th>
<th>IR-RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>m/c indep.</td>
<td>m/c specific.</td>
</tr>
<tr>
<td>Known at $t_{dev}$</td>
<td>Unknown at $t_{dev}$, known at $t_{build}$</td>
</tr>
<tr>
<td>Single “instance”</td>
<td>Per target “instance”</td>
</tr>
</tbody>
</table>

Other Purposes

- IR-RTL Manipulations
  - Gimple → IR-RTL
  - IR-RTL → IR-RTL (combine or split)
  - IR-RTL → Target ASM

- Support
  - Programmer support through convenience constructs
  - Additional heuristics for Instruction selection
The Information in MD

- Processor instructions useful to GCC
- Processor characteristics useful to GCC
- Target ASM syntax
- IR-RTL → IR-RTL transformations (GCC code performs the transformation computations, MD supplies their *target patterns*)
- Target Specific Optimizations
Syntactic Entities in GCC MD

Two kinds:

- **Necessary Specifications**
  - Processor instructions useful to GCC
    - One Gimple $\rightarrow$ One IR-RTL $\quad$ \texttt{define_insn}
    - One Gimple $\rightarrow$ More than one IR-RTL $\quad$ \texttt{define_expand}
  - Processor characteristics useful to GCC $\quad$ \texttt{define_cpu_unit}
    - Target ASM syntax
    - IR-RTL $\rightarrow$ IR-RTL transformations
    - Target Specific Optimizations $\quad$ \texttt{define_split}$\quad$ \texttt{define_peephole2}

- **Programming Conveniences**
  - \texttt{define_insn_and_split}
  - \texttt{define_constants}
  - \texttt{define_cond_exec}
  - \texttt{define_automaton}

**Tip**

See: \$GCCHOME/gcc/rtl.def.
Focus on the RTX!

- Target semantics are in the RTX!
- Target ASM syntax is one argument of `define_insn`
- The set of `define_insns` must be “complete”.
- All IR-RTL manipulations must have a `define_insn`.

Example

```
(define_insn "trap" ; "trap" pattern in i386.md
  [(trap_if (const_int 1) (const_int 5))]
  "" "int $5")
```

```
(define_insn "trap" ; "trap" pattern in mips.md
  [(trap_if (const_int 1) (const_int 0))]
  "" "break 0") ; some target asm details ignored
```