The Retargetability Model of GCC

Uday Khedker
(www.cse.iitb.ac.in/~udy)

GCC Resource Center,
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

13 June 2014
Outline

- A Recap
- Generating the code generators
- Using the generator code generators
Part 1

A Recap
Retargetability Mechanism of GCC

![Diagram showing the retargetability mechanism of GCC]

- **Input Language**
  - Language Specific Code
  - Language and Machine Independent Generic Code

- **Compiler Generation Framework**
  - Machine Dependent Generator Code
  - Machine Descriptions

- **Target Name**

- **Development Time**
  - Parser
  - Gimplifier
  - Tree SSA Optimizer

- **Build Time**
  - Expander
  - Optimizer
  - Recognizer

- **Use Time**

- **Generated Compiler**

Uday Khedker
GRC, IIT Bombay
Retargetability Mechanism of GCC

Input Language → Compiler Generation Framework → Target Name

- Language Specific Code
- Language and Machine Independent Generic Code
- Machine Dependent Generator Code
- Machine Descriptions

Development Time:
- GIMPLE → IR-RTL

Build Time:
- IR-RTL → ASM

Use Time:
- Use

Parser → Gimplifier → Tree SSA Optimizer → Expander → Optimizer → Recognizer → Generated Compiler
Retargetability Mechanism of GCC

Input Language

- Language Specific Code
- Language and Machine Independent Generic Code
- Machine Dependent Generator Code
- Machine Descriptions

Target Name

- GIMPLE → PN
- PN → IR-RTL
- IR-RTL → ASM

Parser
- Gimplifier
- Tree SSA Optimizer
- Expander
- Optimizer
- Recognizer

Development Time

Use Time

Build Time

Selected

Copied

Generated

Generated Compiler
Retargetability Mechanism of GCC

Input Language

Parser
Gimplifier
Tree SSA Optimizer
Expander
Optimizer
Recognizer

Generated Compiler

Language Specific Code
Language and Machine Independent Generic Code
Machine Dependent Generator Code
Machine Descriptions

GIMPLE → PN
PN → IR-RTL
IR-RTL → ASM

Development Time
Build Time
Use Time

GRC, IIT Bombay
Retargetability Mechanism of GCC

Input Language

Compiler Generation Framework

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Target Name

Selected

Copied

Generated

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Generated Compiler

GIMPLE → PN

PN → IR-RTL

IR-RTL → ASM

GIMPLE → IR-RTL

IR-RTL → ASM

Development Time

Build Time

Use Time
Plugin Structure in cc1

Double arrow represents control flow whereas single arrow represents pointer or index.

For simplicity, we have included all passes in a single list. Actually passes are organized into five lists and are invoked as five different sequences.
Plugin Structure in \texttt{cc1}

- \texttt{toplev}
- \texttt{main}
- \texttt{frontend}
- \texttt{pass manager}
- \texttt{pass 1}
- \texttt{code for pass 1}
- \texttt{pass 2}
- \texttt{code for pass 2}
- \texttt{pass expand}
- \texttt{code for language 1}
- \texttt{code for language 2}
- \texttt{code for language n}
- \texttt{insn_data}
- \texttt{optab_table}
- \texttt{recognizer code}
- \texttt{language 1}
- \texttt{language 2}
- \texttt{language n}
- \texttt{MD 1}
- \texttt{MD 2}
- \texttt{MD n}
Plugin Structure in cc1

toplev
main

front
end

pass
manager

pass 1

code for
pass 1

pass 2

code for
pass 2

pass expand

expander
code

optab_table

pass n

recognizer
code

langhook

... code for
language 1

code for
language 2

code for
language n

insn_data
generated
code for
machine 2

MD n

MD 2

MD 1
Plugin Structure in cc1

- toplevel
- main
- frontend
- pass manager
  - pass 1
  - pass 2
  - pass expand
  - pass n
- code for language 1
- code for language 2
- code for language n
- langhook
- code for language 1

- code for pass 1
- code for pass 2
- code for language 2
- code for language n
- insn_data
- generated code
- for machine n
- optab_table
- recognizer
- code

MD 1
MD 2
MD n
What is “Generated”?

- Info about instructions supported by chosen target, e.g.
  - **Listing** data structures (e.g. instruction pattern lists)
  - **Indexing** data structures, since different targets give different lists.

- C functions that **generate** RTL internal representation

- Any useful “attributes”, e.g.
  - Semantic groupings: arithmetic, logical, I/O etc.
  - Processor unit usage groups for pipeline utilisation
Information Supplied by Machine Descriptions

- The target instructions – as ASM strings
- A description of the semantics of each
- A description of the features of each like:
  - Data size limits
  - One of the operands must be a register
  - Implicit operands
  - Register restrictions

<table>
<thead>
<tr>
<th>Information supplied</th>
<th>in <code>define_insn</code> as</th>
</tr>
</thead>
<tbody>
<tr>
<td>The target instruction</td>
<td>ASM string</td>
</tr>
<tr>
<td>A description of it’s semantics</td>
<td>RTL Template</td>
</tr>
<tr>
<td>Operand data size limits</td>
<td>predicates</td>
</tr>
<tr>
<td>Register restrictions</td>
<td>constraints</td>
</tr>
</tbody>
</table>
Part 2

Generating the Code Generators
Using Target Specific RTL as IR

GIMPLE_Assign

(set (<dest>) (<src>))
Using Target Specific RTL as IR

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

Standard Pattern Name
Using Target Specific RTL as IR

Standard Pattern Name

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

Separate CGF code and MD

GIMPLE_ASSIGN "movsi" "movsi" (set (<dest>) (<src>))
Using Target Specific RTL as IR

- **GIMPLE_ASSIGN**
  - "movsi"
  - (set (<dest>) (<src>))

Standard Pattern Name

Separate CGF code and MD

- **GIMPLE_ASSIGN**
  - "movsi"
  - "movsi"
  - (set (<dest>) (<src>))

Implement

- **GIMPLE_ASSIGN**
  - "movsi"
  - "movsi"
  - (set (<dest>) (<src>))

Unnecessary in CGF; hard code

Implement in MD
Retargetability ⇒ Multiple MD vs. One CGF!

CGF

GIMPLE_ASSIGN
"movsi"

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

CGF needs:
An interface immune to MD authoring variations
Retargetability ⇒ Multiple MD vs. One CGF!

CGF needs:
An interface \textit{immune} to MD authoring variations
Retargetability ⇒ Multiple MD vs. One CGF!

How?

Basic Approach: Tabulate

GIMPLE – RTL

struct optab_table []

struct insn_data []

CGF needs:
An interface immune to MD authoring variations
MD Information Data Structures

Two principal data structures

- `struct optab` – Interface to CGF
- `struct insn_data` – All information about a pattern
  - Array of each pattern read
  - Some patterns are SPNs
  - Each pattern is accessed using the generated index

Supporting data structures

- `enum insn_code`: Index of patterns available in the given MD

Note

Data structures are named in the CGF, but populated at build time. Generating target specific code = populating these data structures.
Operation Table

- One optab for every standard pattern name

```c
struct optab_d
{
    enum rtx_code code;
    char libcall_suffix;
    const char *libcall_basename;
    void (*libcall_gen)(struct optab_d *, const char *name, char suffix, enum machine_mode);
    struct optab_handlers handlers[NUM_MACHINE_MODES];
};
typedef struct optab_d * optab;
```
Instruction Data

- One entry for every pattern defined in .md file

- struct insn_data_d
  - Name
  - Information about assembly code generation
    - Single string
    - Multiple string
    - Function returning the required string
    - No assembly code
  - A gen function (as generated in insn-emit.c)
  - Information about operand data
    (pointer to struct insn_operand_data_d)
  - Output format (1=single, 2=multi, 3=function, 0=none).
Assume `movsi` is supported but `movsf` is not supported...

```bash
$(SOURCE_D)/gcc/optabs.h
$(SOURCE_D)/gcc/optabs.c
```

<table>
<thead>
<tr>
<th>optab_table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mov_optab</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OTI_mov</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Assume `movsi` is supported but `movsf` is not supported...

```
$(SOURCE_D)/gcc/optabs.h
$(SOURCE_D)/gcc/optabs.c
```

```
<table>
<thead>
<tr>
<th>optab_table</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>....</td>
</tr>
<tr>
<td></td>
<td>mov_optab</td>
</tr>
<tr>
<td></td>
<td>handler</td>
</tr>
</tbody>
</table>
```

Uday Khedker
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...
Assume **movsi** is supported but **movsf** is not supported...

$$\text{optab_table}$$

<table>
<thead>
<tr>
<th>OTI_mov</th>
<th>mov_optab</th>
<th>handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td></td>
<td>insn_code</td>
</tr>
<tr>
<td>SF</td>
<td></td>
<td>insn_code</td>
</tr>
</tbody>
</table>

$$\text{insn_data}$$

| 1280 | "movsi" |
| ...  | ...     |
| gen_movsi | ... |
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...

\begin{itemize}
  \item \texttt{$(SOURCE_D)/gcc/optabs.h}$
  \item \texttt{$(SOURCE_D)/gcc/optabs.c}$
\end{itemize}

\begin{itemize}
  \item \texttt{$(BUILD)/gcc/insn-output.c}$
\end{itemize}

\begin{itemize}
  \item \texttt{$(BUILD)/gcc/insn-codes.h}$
\end{itemize}

- CODE\_FOR\_movsi=1280
- CODE\_FOR\_movsf=CODE\_FOR\_nothing
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...

```plaintext
$(SOURCE_D)/gcc/optabs.h
$(SOURCE_D)/gcc/optabs.c

\begin{verbatim}
<table>
<thead>
<tr>
<th>optab_table</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>mov_optab</td>
</tr>
<tr>
<td>handler</td>
</tr>
<tr>
<td>SI</td>
</tr>
<tr>
<td>insn_code</td>
</tr>
<tr>
<td>SF</td>
</tr>
<tr>
<td>insn_code</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>insn_data</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;movsi&quot;</td>
</tr>
<tr>
<td>1280</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>gen_movsi</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
$BUILD/gcc/insn-codes.h
CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

$BUILD/gcc/insn-opinit.c
... 
\end{verbatim}
```
Assume **movsi** is supported but **movsf** is not supported...

```plaintext
$(SOURCE_D)/gcc/optabs.h
$(SOURCE_D)/gcc/optabs.c
```

![Diagram showing the code generation process](image)

```plaintext
 insn_data

<table>
<thead>
<tr>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

| 1280 | "movsi" |
|      | ...    |
|      | gen_movsi |

<table>
<thead>
<tr>
<th>$BUILD/gcc/insn-opinit.c</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>$BUILD/gcc/insn-codes.h</th>
</tr>
</thead>
</table>

| CODE_FOR_movsi=1280 |
| CODE_FOR_movsf=CODE_FOR_nothing |

<table>
<thead>
<tr>
<th>$BUILD/gcc/insn-opinit.c</th>
</tr>
</thead>
</table>

...
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...

$(\text{SOURCE} \_D)/\text{gcc/optabs.h}$

$(\text{SOURCE} \_D)/\text{gcc/optabs.c}$

$(\text{BUILD})/\text{gcc/insn-output.c}$

$\text{optab_table}$

- \texttt{movsi}
- \texttt{movsf}

$\text{insn_data}$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1280</td>
<td>&quot;movsi&quot;</td>
</tr>
<tr>
<td>...</td>
<td>gen_movsi</td>
</tr>
</tbody>
</table>

$\text{insn_code}$

- \texttt{CODE\_FOR\_movsi}
- \texttt{CODE\_FOR\_nothing}

$\text{gen\_movsi}$

$\text{Runtime initialization of data structure in cc1 through function init\_all\_optabs}$

$\text{OTI\_mov}$

$\text{SI}$

- \texttt{insn\_code}
- \texttt{CODE\_FOR\_movsi}

$\text{SF}$

- \texttt{insn\_code}
- \texttt{CODE\_FOR\_nothing}$

$\text{CODE\_FOR\_movsi}=1280$

$\text{CODE\_FOR\_movsf}=\text{CODE\_FOR\_nothing}$

$\text{Uday Khedker}$

GRC, IIT Bombay
Assume `movsi` is supported but `movsf` is not supported...

Runtime initialization of data structure in `cc1` through function `init_all_optabs`.

- $(SOURCE_D)/gcc/optabs.h
- $(SOURCE_D)/gcc/optabs.c
- $(BUILD)/gcc/insn-output.c
- insn_data
  - "movsi"
  - gen_movsi
  - ...
  - ...
  - ...
- insn_code
  - CODE_FOR_movsi=1280
  - CODE_FOR_movsf=CODE_FOR_nothing
- ...
<table>
<thead>
<tr>
<th>Generator</th>
<th>Generated from MD</th>
<th>Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genopinit</td>
<td>insn-opinit.c</td>
<td>void init_all_optabs (void);</td>
<td>Operations Table Initialiser</td>
</tr>
<tr>
<td>gencodes</td>
<td>insn-codes.h</td>
<td>enum insn_code = { ... CODE_FOR_movsi = 1280, ... }</td>
<td>Index of patterns</td>
</tr>
<tr>
<td>genoutput</td>
<td>insn-output.c</td>
<td>struct insn_data [CODE].genfun = /* fn ptr */</td>
<td>All insn data e.g. gen function</td>
</tr>
<tr>
<td>genemit</td>
<td>insn-emit.c</td>
<td>rtx gen_rtx_movsi (/* args <em>/) /</em> body */</td>
<td>RTL emission functions</td>
</tr>
</tbody>
</table>
Explicit Calls to \texttt{gen<SPN>} functions

- In some cases, an entry is not made in \texttt{insn_data} table for some SPNs.
- \texttt{gen} functions for such SPNs are explicitly called.
- These are mostly related to
  - Function calls
  - Setting up of activation records
  - Non-local jumps
  - etc. (i.e. deeper study is required on this aspect)
Handling C Code in define_expand

```
(define_expand "movsi"
  [(set (op0) (op1))]
  ""
  "{ /* C CODE OF DEFINE EXPAND */ }"
)

rtx

gen_movsi (rtx operand0, rtx operand1)
{
  ...
  {
    /* C CODE OF DEFINE EXPAND */
  }
  emit_insn (gen_rtx_SET (VOIDmode, operand0, operand1)
  ...
}
```
Part 3

Using the Code Generators
The Process of Expansion

GIMPLE

\texttt{gimple\_expand\_cfg}

RTL
The Process of Expansion

GIMPLE

\[\text{gimple\_expand\_cfg}\]

RTL
The Process of Expansion

GIMPLE

\texttt{gimple\_expand\_cfg}

\texttt{RTL}

Match SPN

Search optab_table[] and extract CODE\_FOR\_<SPN>
The Process of Expansion

GIMPLE ➔ gimple_expand_cfg ➔ RTL

Match SPN

Search optab_table[] and extract CODE_FOR_<SPN>

Search insn_data[] and extract gen_<SPN>
The Process of Expansion

GIMPLE → gimple_expand_cfg → RTL

Match SPN

Search optab_table[] and extract CODE_FOR_<SPN>

Search insn_data[] and extract gen_<SPN>

invoke gen_<SPN>
The Process of Expansion

GIMPLE → gimple_expand_cfg → RTL

Match SPN

Search optab_table[] and extract CODE_FOR_<SPN>

Search insn_data[] and extract gen_<SPN>

Generated

invoke gen_<SPN>
Control Flow: GIMPLE to RTL Expansion (pass expands)

gimple_expand_cfg
    expand_gimple_basic_block(bb)
    expand_gimple_cond(stmt)
    expand_gimple_stmt(stmt)
    expand_gimple_stmt_1(stmt)
    expand_expr_real_2
    expand_expr  /* Operands */
        expand_expr_real
    optab_for_tree_code
    expand_binop /* Now we have rtx for operands */
        expand_binop_directly
        /* The plugin for a machine */
        code=optab_handler(binoptab,mode)
    GEN_FCN
    emit_insn
expand_binop_directly

... /* Various cases of expansion */

/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
expand_binop_directly

... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
RTL Generation

```
expand_binop_directly
    ... /* Various cases of expansion */
    /* One case: integer mode move */
    icode = mov_optab->handler[SImode].insn_code
    if (icode != CODE_FOR_nothing) {
        ... /* preparatory code */
        emit_insn (GEN_FCN(icode)(dest,src));
    }
```

```
insn-codes.h enum insn_code
    = {...
    CODE_FOR_movsi = 1280,
    ...}
```
expand_binop_directly

... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}
expand_binop_directly

... /* Various cases of expansion */

/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}

Use icode (= 1280)

#define GEN_FCN(code) insn_data[code].genfun
expand_binop_directly
... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
... /* preparatory code */
emit_insn (GEN_FCN(icode)(dest,src))
}

#define GEN_FCN(code) insn_data[code].genfun
# define insn-output.c
insn_data[1280].genfun
= gen_movsi
RTL Generation

expand_binop_directly
    ... /* Various cases of expansion */
/* One case: integer mode move */
icode = mov_optab->handler[SImode].insn_code
if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(dest,src));
}

#define GEN_FCN(code) insn_data[code].genfun

Execute: gen_movsi (dest,src)
RTL to ASM Conversion

- Simple pattern matching of IR RTLs and the patterns present in all named, un-named, standard, non-standard patterns defined using `define_insn`.
- A DFA (deterministic finite automaton) is constructed and the first match is used.
Part 4

Conclusions
• Retargetability in Davidson Fraser Model
  ▶ Manually rewriting expander and recognizer
  ▶ Simple enough for machines of 1984 era

• Retargetability in GCC
  Automatic construction possible by separating machine specific details in carefully designed data structures
  ▶ List instructions as they appear in the chosen MD
  ▶ Index them
  ▶ Supply index to the CGF