Workshop on Essential Abstractions in GCC

The Retargetability Model of GCC

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

2 July 2012
Outline

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

A Recap
Retargetability Model: A Recap

Retargetability Mechanism of GCC

Input Language

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Compiler Generation Framework

Selected

Copied

Copied

Copied

Generated

Generated

Generated

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Target Name

Development Time

Build Time

Use Time

Generated Compiler

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Retargetability Model: A Recap

Retargetability Mechanism of GCC

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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

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

Development Time

Use Time

Build Time

Selected

- Copied
- Copied

Generated

- Generated
- Generated

Parser

- Gimplifier
- Tree SSA Optimizer

Expander

- Optimizer
- Recognizer

Generated Compiler

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Retargetability Mechanism of GCC

Input Language

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Target Name

Compiler Generation Framework

Development Time

Build Time

Use Time

GIMPLE → PN

PN → IR-RTL

IR-RTL → ASM

GIMPLE → IR-RTL

IR-RTL → ASM

Parser

Gimiplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Generated Compiler

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Retargetability Mechanism of GCC

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

Compiler Generation Framework

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

Generated Compiler

- Development Time
  - GIMPLE → PN
  - PN → IR-RTL
  - IR-RTL → ASM

- Build Time
  - GIMPLE → IR-RTL
  - IR-RTL → ASM

- Use Time
Plugin Structure in \texttt{cc1}

- toplev (main)
- frontend
- pass manager

- langhook
- code for language 1

- pass 1
- code for pass 1

- pass 2
- code for pass 2
- pass expand
- optab\_table

- pass n
- recognizer code

Double arrow represents control flow whereas single arrow represents pointer or index.
Plugin Structure in \texttt{cc1}

- **toplevel**: main
- **frontend**
- **pass manager**
  - pass 1
  - pass 2
  - ... 
  - pass n
- **langhook**
  - code for language 1
  - code for language 2
  - code for language n
- **code for pass 1**
- **code for pass 2**
- **insn_data**
  - generated code for machine 1
- **optab_table**
- **recognizer code**

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Plugin Structure in `cc1`

- `toplev` main
- `frontend`
- `pass manager`
- `langhook`...
- `code for language 1`
- `code for language 2`
- `code for language n`
- `pass 1`
  - `code for pass 1`
- `pass 2`
  - `code for pass 2`
- `pass expand`
  - `expander`
  - `code`
  - `optab_table`
- `insn_data` generated code for machine 2
- `recognizer` code
- `MD n`
- `MD 2`
- `MD 1`
Plugin Structure in \textit{cc1}

- toplev
  - main
- langhook
  - code for language 1
  - code for language 2
  - code for language n
- front end
- pass manager
  - code for pass 1
  - code for pass 2
  - ... (pass n)
    - recognizer
    - code
    - generated code for machine n
    - insn\_data
      - optab\_table

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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 define_insn 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

Separate CGF code and MD

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

Standard Pattern Name

Separate CGF code and MD

Implement

Unnecessary in CGF; hard code

Implement in MD
Retargetability $\Rightarrow$ Multiple MD vs. One CGF!

CGF needs:

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

CGF needs:
An interface **immune** to MD authoring variations
Retargetability \Rightarrow \text{Multiple MD vs. One CGF!}

Basic Approach: Tabulate

GIMPLE - RTL

\begin{align*}
\text{struct optab_table} & \quad \text{struct insn_data} \\
\end{align*}

CGF needs:
An interface immune to MD authoring variations

GIMPLE_ASSIGN

"movsi"

How?

MD 1
"movsi", (set \(<\text{dest}>\)) \(<\text{src}>\))

\[ \cdots \]

MD n
"movsi", (set \(<\text{dest}>\)) \(<\text{src}>\))
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 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)
  - Output format (1=single, 2=multi, 3=function, 0=none).
Assume `movsi` is supported but `movsf` is not supported...

```plaintext
$(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>
</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>
</tbody>
</table>
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...

```
$(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>mov_optab</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>handler</td>
</tr>
</tbody>
</table>

\texttt{OTI\_mov}

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Assume `movsi` is supported but `movsf` is not supported...

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

---

**OTI_mov**

```
OTI_mov
```

---

**mov_optab**

```
mov_optab
```

---

**handler**

```
handler
```

---

**insn_code**

```
insn_code
```

---

**SI**

```
SI
```

---

**SF**

```
SF
```

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Assume `movsi` is supported but `movsf` is not supported...

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

```
$(BUILD)/gcc/insn-output.c
```

![Diagram](image)
Assume `movsi` is supported but `movsf` is not supported...

\[
\begin{align*}
\text{OTI}_\text{mov} & \quad \text{optab_table} \\
\text{mov_optab} & \quad \text{insn_data} \\
\text{SI} & \quad \text{insn_code} \\
\text{SF} & \quad \text{insn_code} \\
\end{align*}
\]

\[
\begin{align*}
\text{insn_data} & \quad \text{insn_data} \\
\text{movsi} & \quad \text{insn_data} \\
1280 & \quad \text{insn_data} \\
\text{gen_movsi} & \quad \text{insn_data} \\
\end{align*}
\]

\[
\begin{align*}
\text{CODE_FOR_movsi} & = 1280 \\
\text{CODE_FOR_movsf} & = \text{CODE_FOR_nothing}
\end{align*}
\]
Assume **movsi** is supported but **movsf** is not supported...

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

optab_table

---

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

insn_data

---

$BUILD/gcc/insn-codes.h
CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

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

\begin{verbatim}
$(SOURCE_D)/gcc/optabs.h
$(SOURCE_D)/gcc/optabs.c
\end{verbatim}

\begin{verbatim}
insn_data

... ...

"movsi"

1280

... gen_movsi

... ...
\end{verbatim}

\begin{verbatim}
$BUILD/gcc/insn-codes.h
$BUILD/gcc/insn-opinit.c
\end{verbatim}

\begin{verbatim}
CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing
\end{verbatim}
Assume `movsi` is supported but `movsf` is not supported...

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

**optab_table**

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

**Optab table**: Runtime initialization of data structure using function `set_optab_handler`

```plaintext
$(BUILD)/gcc/insn-output.c
```

**insn_data**

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

```plaintext
$BUILD/gcc/insn-codes.h
CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing
```

```plaintext
$BUILD/gcc/insn-opinit.c
```

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Assume \texttt{movsi} is supported but \texttt{movsf} is not supported...

\begin{itemize}
\item \texttt{optab} table
\item \texttt{insn} data
\end{itemize}

\begin{itemize}
\item \texttt{optab} table
\item \texttt{insn} data
\end{itemize}
### GCC Generation Phase – Revisited

<table>
<thead>
<tr>
<th>Generator</th>
<th>Generated Information</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genopinit</td>
<td>void init_all_optabs (void);</td>
<td>Operations Table Initialiser</td>
</tr>
<tr>
<td>gencodes</td>
<td>enum insn_code = {... CODE_FOR_movsi = 1280, ...}</td>
<td>Index of patterns</td>
</tr>
<tr>
<td>genooutput</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>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`

```c
(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
cc1 Control Flow: GIMPLE to RTL Expansion (pass_expand)

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 */
ical = 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));
}
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,
  ...
}
```

Got index into insn_data

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
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));
}

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

insn-output.c
insn_data[1280].genfun
= gen_movsi

#define GEN_FCN(code) insn_data[code].genfun
 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
A Comparison with Davidson Fraser Model

- 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 insns as they appear in the chosen MD
  - Index them
  - Supply index to the CGF