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

1 July 2013
Outline

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

A Recap
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
- Build Time
- Use Time

**Use of Essential Abstractions in GCC**

- Input Language
- Target Name
- Compiler Generation Framework
- Development Time
- Build Time
- Use Time

**Essential Abstractions in GCC**
- Parser
- Gimplifier
- Tree SSA Optimizer
- Expander
- Optimizer
- Recognizer

**Generated Compiler**
Retargetability Mechanism of GCC

Essential Abstractions in GCC

GCC Resource Center, 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

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Selected

Copied

Copied

Copied

Generated

Generated

Generated

Development Time

Build Time

Use Time

GIMPLE → PN

PN → IR-RTL

IR-RTL → ASM

GIMPLE → IR-RTL

IR-RTL → ASM

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

Development Time

PN → IR-RTL

Build Time

IR-RTL → ASM

Use Time

GIMPLE → IR-RTL

GIMPLE → PN

PN → IR-RTL

IR-RTL → ASM

Essential Abstractions in GCC

Generated Compiler

Compiler Generation Framework

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer
Retargetability Mechanism of GCC

Input Language

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Target Name

Selected

Copied

Copied

Generated

Generated

Selected

Generated Compiler

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Compiler Generation Framework

GIMPLE → PN +

PN → IR-RTL +

IR-RTL → ASM

GIMPLE → IR-RTL +

IR-RTL → ASM
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.

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Plugin Structure in cc1

1. toplev
   main

2. frontend

3. pass manager
   pass 1
   code for pass 1

4. pass 2
   code for pass 2

5. pass expand
   expander code
   optab_table

6. pass n
   recognizer code

7. langhook
   code for language 1
   ... code for language n

8. MD 1
   insn_data
   generated code for machine 1

9. MD 2
   ... MD n
Plugin Structure in cc1

- toplev
  - main
- frontend
- pass manager
  - pass 1
    - code for pass 1
  - pass 2
    - code for pass 2
  - pass expand
    - expander code
      - optab_table
  - pass n
    - recognizer code

- 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

Essential Abstractions in GCC

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

- **toplev**
  - main

- **frontend**

- **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

- **MD n**
- **MD 2**
- **MD 1**

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 diff. targets give diff. 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_ASSUME  (set (<dest>) (<src>))
Using Target Specific RTL as IR

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

Standard Pattern Name
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>))
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 needs:
An interface immune to MD authoring variations
Retargetability $\Rightarrow$ Multiple MD vs. One CGF!

CGF needs:
An interface **immune** to MD authoring variations
Retargetability \( \Rightarrow \) Multiple MD vs. One CGF!

**CGF needs:**
An interface **immune** to MD authoring variations

Basic Approach: Tabulate

GIMPLE - RTL

**struct optab_table []**

**struct insn_data []**
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...

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

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
```
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>
</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>

<table>
<thead>
<tr>
<th>OTI_mov</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov_optab</td>
</tr>
<tr>
<td>handler</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SI</td>
</tr>
<tr>
<td>insn_code</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SF</td>
</tr>
<tr>
<td>insn_code</td>
</tr>
</tbody>
</table>
Assume `movsi` is supported but `movsf` is not supported...

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

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

#### insn_data

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

#### optab_table

<p>| | |</p>
<table>
<thead>
<tr>
<th></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></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### OTI_mov

- `handler`
- `mov_optab`
- `insn_code`
- `insn_code`
Assume `movsi` is supported but `movsf` is not supported...

![Diagram showing the optab_table structure]

**`optab_table`**

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

**`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>... gen_movsi</td>
</tr>
</tbody>
</table>

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

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

$BUILD/gcc/insn-codes.h
```

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing
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}

\textbf{optab\_table}

\begin{verbatim}
... ... 
... ... 
\end{verbatim}

\textbf{OTI\_mov}

\begin{verbatim}
handler

\begin{tabular}{|c|c|}
\hline
SI & insn\_code \\
\hline
SF & insn\_code \\
\hline
\end{tabular}
\end{verbatim}

\textbf{insn\_data}

\begin{verbatim}
... ... 
1280  "movsi" 
... gen\_movsi 
... 
\end{verbatim}

\textbf{$(BUILD)/gcc/insn-codes.h}

\begin{verbatim}
CODE\_FOR\_movsi=1280
CODE\_FOR\_movsf=CODE\_FOR\_nothing
\end{verbatim}

\textbf{$(BUILD)/gcc/insn-opinit.c}

...
Assume `movsi` is supported but `movsf` is not supported...

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

```
optab_table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
OTI_mov

<table>
<thead>
<tr>
<th>handler</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>mov_optab</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
insn_data

<table>
<thead>
<tr>
<th>insn_code</th>
<th>insn_code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```
$BUILD/gcc/insn-codes.h

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

```

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

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

```

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
```

```
insn_data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| ... | ...
| 1280 | "movsi"
|       | ...
```

```
$BUILD/gcc/insn-codes.h

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing
```

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

... 
```
Assume `movsi` is supported but `movsf` is not supported...

1. Assume `movsi` is supported but `movsf` is not supported.

2. 
   - $(SOURCE_D)/gcc/optabs.h
   - $(SOURCE_D)/gcc/optabs.c

3. 
   - $(BUILD)/gcc/insn-output.c

4. 
   - insn_data
     - ...
     - ...
     - "movsi"
     - 1280
       - ...
       - gen_movsi
     - ...

5. 
   - $BUILD/gcc/insn-codes.h
     - CODE_FOR_movsi=1280
     - CODE_FOR_movsf=CODE_FOR_nothing

6. 
   - $BUILD/gcc/insn-opinit.c
     - ...

---

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
### GCC Generation Phase – Revisited

<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</td>
<td>Operations Table Initialiser</td>
</tr>
<tr>
<td></td>
<td></td>
<td>init_all_optabs (void);</td>
<td></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>genooutput</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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rtx</td>
<td></td>
<td>rtx</td>
<td></td>
</tr>
<tr>
<td>genemit</td>
<td>insn-emit.c</td>
<td>gen_rtx_movsi /* args */</td>
<td>RTL emission functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ /* body */}</td>
<td></td>
</tr>
</tbody>
</table>
Explicit Calls to `gen<SPN>` functions

- In some cases, an entry is not made in `insn_data` table for some SPNs.
- `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
**cc1 Control Flow: GIMPLE to RTL Expansion (pass_expand)**

```c
void gimple_expand_cfg {
    void expand_gimple_basic_block (gimple_bb)
    void expand_gimple_cond (gimple_stmt)
    void expand_gimple_stmt (gimple_stmt)
        void expand_gimple_stmt_1 (gimple_stmt)
    void expand_expr_real_2
        void expand_expr  /* Operands */
            void expand_expr_real
        void optab_for_tree_code
    void expand_binop /* Now we have rtx for operands */
        void expand_binop_directly
            /* The plugin for a machine */
            void code=optab_handler (binoptab,mode)
        GEN_FCN
    void 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));
}
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));
}
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

Use icode (= 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));
}

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