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

Notes
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

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

Development Time
Build Time
Use Time
Plugin Structure in \texttt{cc1}

- \texttt{toplevel main}
- \texttt{frontend}
- \texttt{pass manager}
- \texttt{pass 1}
- \texttt{pass 2}
- \texttt{pass \ldots}
- \texttt{pass n}
- \texttt{langhook}
- \texttt{code for language 1}
- \texttt{code for language 2}
- \texttt{code for language n}
- \texttt{pass expand}
- \texttt{expander code}
- \texttt{optab_table}
- \texttt{recognizer code}

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

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>
Using Target Specific RTL as IR

GIMPLE_ASSIGN

"movsi"

(set (<dest>) (<src>))

Standard Pattern Name

Separate CGF code and MD

Implement

GIMPLE_ASSIGN

"movsi"

(set (<dest>) (<src>))

Unnecessary in CGF; hard code

Implement in MD

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Retargetability \(\Rightarrow\) Multiple MD vs. One CGF!

CGF needs:
An interface immune to MD authoring variations

Basic Approach: Tabulate

- GIMPLE\_ASSIGN
- "movsi"

How?

Structures:
- `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

```c
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_DIR})/gcc/insn-data.c
$({SOURCE_DIR})/gcc/insn-output.c

insn_table

.....

mov_optab

OTI_move

handler

SI
insn_code

SF
insn_code

1280
"movsi"

... gen_movi...

$BUILD/gcc/insn-opinit.c

$BUILD/gcc/insn-codes.h

CODE_FOR_movsi=1280
CODE_FOR_movsf=CODE_FOR_nothing

$BUILD/gcc/insn-codes.h
```
### 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 = [...]</td>
<td>Index of patterns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CODE_FOR_movsi = 1280,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></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>rtx</td>
<td></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>
GCC Generation Phase – Revisited

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`

```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)
  ...
}
```
Handling C Code in define_expand

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

Notes
**RTL Generation**

```c
expand_binop_directly
  ... /* Various cases of expansion */
  /* One case: integer mode move */
  icode = mov_optab->handler[Si_mode].insn_code
  if (icode != CODE_FOR_nothing) {
    ... /* preparatory code */
    emit_insn (GEN_FCN(icode)(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.
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
A Comparison with Davidson Fraser Model