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

Outline

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

Retargetability Mechanism of GCC

Part 1

A Recap
2 July 2012 Retargetability Model: A Recap 3/18

Plugin Structure in \texttt{cc1}

- toplevel main → frontend → pass manager → pass 1 → code for pass 1 → langhook → code for language 1
- pass 2 → code for language 2
- pass expand → optab_table
- pass n → recognizer code
- double arrow represents control flow whereas single arrow represents pointer or index

Essential Abstractions in GCC GCC Resource Center, IIT Bombay

2 July 2012 Retargetability Model: A Recap 4/18

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

Essential Abstractions in GCC GCC Resource Center, IIT Bombay

2 July 2012 Retargetability Model: A Recap 5/18

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

Information supplied in define_insn as

<table>
<thead>
<tr>
<th>Information supplied</th>
<th>define_insn as</th>
</tr>
</thead>
<tbody>
<tr>
<td>The target instruction</td>
<td>ASM string</td>
</tr>
<tr>
<td>A description of its 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>

Essential Abstractions in GCC GCC Resource Center, IIT Bombay
Part 2

Generating the Code Generators

Retargetability ⇒ Multiple MD vs. One CGF!

Using Target Specific RTL as IR

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

Standard Pattern Name

Separate CGF code and MD

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

Implement

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

Implement in MD

Unnecessary in CGF; hard code

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 {
    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...

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

- Runtime initialization of data structure using function `set_optab_handler`
## 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>gencodes</td>
<td>insn-codes.h</td>
<td>enum insn_code = [...]</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 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 */ }
```

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

### Using the Code Generators

---

Part 3
Control Flow: GIMPLE to RTL Expansion (pass\_expand)

```c
cc1

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
```

RTL Generation

```c
expand\_binop\_directly
    ... /* Various cases of expansion */
    /* One case: integer mode move */
    icode = mov\_optab->handler[\texttt{SIMode}]\_.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.

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