Outline

• A Recap
• Generating the code generators
• Using the generator code generators
Retargetability Mechanism of GCC
Plugin Structure in `cc1`

- `toplevel main` → `frontend` → `pass manager` → `pass 1` → `code for pass 1`
- `langhook` → `pass 2` → `code for pass 2`
- `code for language 1` → `pass n` → `code for pass n`
- `pass expand` → `expander code` → `optab_table`

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 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 the MD

- 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

How GCC uses target specific RTL as IR

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

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"

How?

Basic Approach: Tabulate

GIMPLE – RTL

Struct optab []

Struct insn_data []

MD 1

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

MD n

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

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

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

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

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

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

Notes

Notes

Notes

Notes
### 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 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>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>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`

```
(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)
  ...
}
```

Notes
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

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));
   }
• 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.
Retargetability in Davidson Fraser Model
- Manually rewriting Expander and patter matcher
- Expected to be simple 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