Outline

- Some details of MD constructs
  - On names of patterns in .md files
  - On the role of \texttt{define}\_\texttt{expand}
  - On the role of predicates and constraints
  - Mode and code iterators
  - Defining attributes
  - Other constructs

- Improving machine descriptions and instruction selection
  - New constructs to factor out redundancy
  - Cost based tree tiling for instruction selection

Part 1

More Features

Pattern Names in .md File

- All Patterns
  - Named Patterns
    - With $^\star$
      - No \texttt{gen}\_function
    - Without $^\star$
      - No \texttt{gen}\_function
  - Unnamed Patterns
    - Standard
      - \texttt{gen}\_name function
        - Called implicitly
        - Can be called explicitly
    - Non-Standard
      - \texttt{gen}\_name function
        - Not called implicitly
        - Can be called explicitly
Role of **define_expand**

**Uses of define_expand**

- Generate RTL
- Do not generate RTL
- Setting operands
- Setting global variables

Using **define_expand** for Generating RTL statements

**Calling gen_pattern function**

- Implicit call
- Explicit call
- Non-standard pattern
- Standard pattern
- During expansion
- During expansion
- Some other pass
- Preparatory statement of define_expand
- Some function in a .c file

Use of Predicates

```
(define_insn "<name>"
 [(set (match_operand:SI 0 "general_operand" "=r")
   (plus:SI (match_dup 0)
     (match_operand:SI 1 "general_operand" "r")))]
 "" 
 "...")
```

Predicates are used for matching operands

- For constructing an insn during expansion
- `<name>` must be a standard pattern name
- For recognizing an instruction (in subsequent RTL passes including pattern matching)

Understanding Constraints

```
(define_insn "<name>"
 [(set (match_operand:SI 0 "general_operand" "=r")
   (plus:SI (match_dup 0)
     (match_operand:SI 1 "general_operand" "r")))]
 "" 
 "...")
```

Constraints

- Reloading operands in the most suitable register class
- Fine-tuning within the set of operands allowed by the predicate
- If omitted, operands will depend only on the predicates
Role of Constraints

Consider the following two instruction patterns:

- (define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
    (plus:SI (match_dup 0)
      (match_operand:SI 1 "general_operand" "r")))]
  ")"

  During expansion, the destination and left operands must match the same predicate.
  During recognition, the destination and left operands must be identical.

- (define_insn ""
  [(set (match_operand:SI 0 "general_operand" "=r")
    (plus:SI (match_operand:SI 1 "general_operand" "z")
      (match_operand:SI 2 "general_operand" "r")))]
  ")"

  Predicates of the first pattern do not match (because they require identical operands during recognition).
  Constraints do not match for operand 1 of the second pattern.
  Reload pass generates additional insn to that the first pattern can be used.

Handling Mode Differences

(define_insn "subsdi3"
  [(set (match_operand:DI 0 "register_operand" "=d")
    (minus:DI (match_operand:DI 1 "register_operand" "d")
      (match_operand:DI 2 "register_operand" "d")))]
  "sub\%0,\%1,\%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "DI")])

(define_insn "subsi3"
  [(set (match_operand:SI 0 "register_operand" "=d")
    (minus:SI (match_operand:SI 1 "register_operand" "d")
      (match_operand:SI 2 "register_operand" "d")))]
  ")"
  "sub\%0,\%1,\%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])
Mode Iterators: Abstracting Out Mode Differences

(define_mode_iterator GPR [SI (DI “TARGET_64BIT”)])
(define_mode_attr d [(SI “”) (DI “d”)])
(define_insn “sub<MODE>”
  [(set (match_operand:GPR 0 “register_operand” “=d”))
   (minus:GPR (match_operand:GPR 1 “register_operand” “d”))
   (match_operand:GPR 2 “register_operand” “d”)])
“”
“<d>subu%0,%1,%2”
[(set_attr “mode” “<MODE>”)]

Handling Code Differences

(define_expand “bunordered”
  [(set (pc) (if_then_else (unordered:CC (cc0) (const_int 0))
                        (label_ref (match_operand 0 “”))
                        (pc))])
“”
{ mips_expand_conditional_branch (operands, UNORDERED); DONE; }
)

(define_expand “bordered”
  [(set (pc) (if_then_else (ordered:CC (cc0) (const_int 0))
                        (label_ref (match_operand 0 “”))
                        (pc))])
“”
{ mips_expand_conditional_branch (operands, ORDERED); DONE; }
)

Code Iterators: Abstracting Out Code Differences

(define_code_iterator any_cond [unordered ordered])
(define_expand “b<code>”
  [(set (pc)
       (if_then_else (any_cond:CC (cc0)
                        (const_int 0))
                     (label_ref (match_operand 0 “”))
                     (pc))])
“”
{ mips_expand_conditional_branch (operands, <CODE>); DONE; }
)

Part 3

Miscellaneous Features
Defining Attributes

- Classifications are need based
- Useful to GCC phases – e.g. pipelining

Property: Pipelining
Need: To classify target instructions
Construct: define_attr

; ; Instruction type.
(define_attr "type"
  "other, multi, alu, alul, negnot, ... str, cld, ..."
  (const_string "other"))

Fields: Attribute name, all possible values, one of the possible values, default.

Specifying Instruction Attributes

- Optional field of a define_insn
- For an i386, we choose to mark string instructions with the attribute value str

(define_insn "*strmovdi_rex_1"
  [(set (mem:DI (match_operand:DI 2 ...)))
   "TARGET_64BIT && (TARGET_SINGLE_ ...)"
   "movsq"
  [ (set_attr "type" "str")
    ...
    (set_attr "memory" "both")])

NOTE
An instruction may have more than one attribute!

Using Attributes

(define_insn_reservation "pent_str" 12
  (and (eq_attr "cpu" "pentium")
       (eq_attr "type" "str")
  )
  "pentium-np*12")

Pipeline specification requires the CPU type to be "pentium" and the instruction type to be "str"

Some Other RTL Constructs

- define_split: Split complex insn into simpler ones e.g. for better use of delay slots
- define_insn and split: A combination of define_insn and define_split
  Used when the split pattern matches and insn exactly.
- define_peephole2: Peephole optimization over insns that substitutes insns. Run after register allocation, and before scheduling.
- define_constants: Use literal constants in rest of the MD.
The Need for Improving Machine Descriptions

The Problems:

- The specification mechanism for Machine descriptions is quite adhoc
  - Only syntax borrowed from LISP, neither semantics not spirit!
  - Non-composable rules
  - Mode and code iterator mechanisms are insufficient
- Adhoc design decisions
  - Honouring operand constraints delayed to global register allocation
  - During GIMPLE to RTL translation, a lot of C code is required
  - Choice of insertion of NOPs

Handing Constraints

- `define_insn` patterns have operand predicates and constraints
- While generating an RTL insn from GIMPLE, only the predicates are checked. The constraints are completely ignored
- An insn which is generated in the expander is modified in the reload pass to satisfy the constraints
- It may be possible to generate this final form of RTL during expansion by honouring constraints
  - Honouring contraints earlier than the current place
    ⇒ May get rid of some C code in `define_expand`

Design Flaws in Machine Descriptions

- Repetition of almost similar RTL expressions across multiple `define_insn` an `define_expand` patterns
  - Some Modes, Predicates, Constraints, Boolean Condition, or RTL Expression may differ everything else may be identical
  - One RTL expression may appears as a sub-expression of some other RTL expression
- Repetition of C code along with RTL expressions in these patterns.
Redundancy in MIPS Machine Descriptions: Example 1

\[
\text{(set (match_operand: } m \text{ 0 "register_operand" ) (plus: } m \text{ (match_operand: } m \text{ 1 "register_operand" ) (match_operand: } m \text{ 2 "\#" ) )})
\]

Redundancy in MIPS Machine Descriptions: Example 2

\[
\text{(set (match_operand: } m \text{ 0 "register_operand" ) (mult: } m \text{ (match_operand: } m \text{ 1 "register_operand" ) (match_operand: } m \text{ 2 "\#" ) )})
\]

Redundancy in MIPS Machine Descriptions: Example 3

\[
\text{(set (match_operand: } m \text{ 0 "register_operand" ) (plus: } m \text{ (match_operand: } m \text{ 1 "register_operand" ) (match_operand: } m \text{ 2 "\#" ) )})
\]

Insufficient Iterator Mechanism

- Iterators cannot be used across `define_insn`, `define_expand`, `define_peephole2` and other patterns
- Defining iterator attribute for each varying parameter becomes tedious
- For same set of modes and rtx codes, change in other fields of pattern makes use of iterators impossible
- Mode and code attributes cannot be defined for operator or operand number, name of the pattern etc.
- Patterns with different RTL template share attribute value vector for which iterators can not be used
Many Similar Patterns Cannot be Combined

(define expand "iordi3"
 [(set (match operand:DI 0 "nonimmediate_operand" "")
   (ior:DI (match_operand:DI 1 "nonimmediate_operand" "")
     (match_operand:DI 2 "x86_64_general_operand" "")))
  (clobber (reg:CC FLAGS REG))]
 "TARGET_64BIT"
 "ix86_binary_operator (IOR, DImode, operands); DONE;"
)

(defineInsn "iordi_1_rex64"
 [(set (match_operand:DI 0 "nonimmediate_operand" "rm,r")
   (ior:DI (match_operand:DI 1 "nonimmediate_operand" "%0,0")
     (match_operand:DI 2 "x86_64_general_operand" "re,rme")))
  (clobber (reg:CC FLAGS REG))]
 "TARGET_64BIT"
 REX & ix86_binary_operator OP (IOR, DImode, operands)
 "or{q,t}{%2, %0,%0, %2}
 [(set_attr "type" "alu"
   (set_attr "mode" "DI"))]

---

### Measuring Redundancy in RTL Templates

<table>
<thead>
<tr>
<th>MD File</th>
<th>Total number of patterns</th>
<th>Number of primitive trees</th>
<th>Number of times primitive trees are used to create composite trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386.md</td>
<td>1303</td>
<td>349</td>
<td>4308</td>
</tr>
<tr>
<td>arm.md</td>
<td>534</td>
<td>232</td>
<td>1369</td>
</tr>
<tr>
<td>mips.md</td>
<td>337</td>
<td>147</td>
<td>921</td>
</tr>
</tbody>
</table>

---

### specRTL: Key Observations

- Davidson Fraser insight
  
  *Register transfers are target specific but their form is target independent*

- GCC’s approach
  - Use Target independent RTL for machine specification
  - Generate expander and recognizer by reading machine descriptions

Main problems with GCC’s Approach

*Although the shapes of RTL statements are target independent, they have to be provided in RTL templates*

- Our key idea:
  
  *Separate shapes of RTL statements from the target specific details*

---

### Specification Goals of specRTL

Support all of the following

- Separation of shapes from target specific details
- Creation of new shapes by composing shapes
- Associating concrete details with shapes
- Overriding concrete details
Software Engineering Goals of specRTL

- Allow non-disruptive migration for existing machine descriptions
  - Incremental changes
  - No need to change GCC source until we are sure of the new specification

GCC must remain usable after each small change made in the machine descriptions

Meeting the Specification Goals: Key Idea

- Separation of shapes from target specific details:
  - Shape ≡ tree structure of RTL templates
  - Details ≡ attributes of tree nodes (e.g., modes, predicates, constraints etc.)

- Abstract patterns and Concrete patterns
  - Abstract patterns are shapes with “holes” in them that represent missing information
  - Concrete patterns are shapes in which all holes are plugged in using target specific information

- Abstract patterns capture shapes which can be concretized by providing details

Meeting the Specification Goals: Operations

- Creating new shapes by composing shapes: extends
- Associating concrete details with shapes: instantiates
- Overriding concrete details: overrides

Properties of Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Base pattern</th>
<th>Derived pattern</th>
<th>Nodes influenced</th>
<th>Can change</th>
</tr>
</thead>
<tbody>
<tr>
<td>extends</td>
<td>Abstract</td>
<td>Abstract</td>
<td>Leaf nodes</td>
<td>Structure</td>
</tr>
<tr>
<td>instantiates</td>
<td>Abstract</td>
<td>Concrete</td>
<td>All nodes</td>
<td>Attributes</td>
</tr>
<tr>
<td>overrides</td>
<td>Abstract</td>
<td>Concrete</td>
<td>Internal nodes</td>
<td>Attributes</td>
</tr>
<tr>
<td></td>
<td>Concrete</td>
<td>Concrete</td>
<td>All nodes</td>
<td>Attributes</td>
</tr>
</tbody>
</table>
Creating Abstract Patterns

abstract set_plus extends set
{
  root.2 = plus;
}

abstract set_macc extends set_plus
{
  root.2.2 = mult;
}

Creating Concrete Patterns

concrete add<mode>3.insn instantiates set_plus
{
  set_plus(register_operand:ANYF:"=f",
           register_operand:ANYF:"f",
           register_operand:ANYF:"f");
  root.2.mode = ANYF;
}

concrete add<mode>3.expand instantiates set_plus
{
  set_plus(register_operand:GPR:"",
           register_operand:GPR:"",
           arith_operand:GPR:"");
  root.2.mode = GPR;
}

Generating Conventional Machine Descriptions

concrete add<mode>3.insn instantiates set_plus
{
  set_plus(register_operand:ANYF:"=f",
           register_operand:ANYF:"f",
           register_operand:ANYF:"f");
  root.2.mode = ANYF;
}

Resulting MD Specification

(define_insn "add<mode>3"
  [(set (match_operand:ANYF 0 "register_operand" "="f")
          (plus:ANYF (match_operand:ANYF 1 "register_operand" "f")
                     (match_operand:ANYF 2 "register_operand" "f")))]
  "/* Conventional Machine Description Fragments */
  ""
)

Overriding Details

concrete add<mode>3.expand instantiates set_plus
{
  set_plus(register_operand:GPR:"",
           register_operand:GPR:"",
           arith_operand:GPR:"");
  root.2.mode = GPR;
}

concrete *add<mode>3.insn overrides add<mode>3.expand
{
  allconstraints = ("=d,d", "d,d", "d,Q");
}
Some More Examples

Omitting conventional MD fragments

```c
concrete *mul<mode>3.insn instantiates set_mult
{
  set_mult(register_operand:SCALARF:"=f",
           register_operand:SCALARF:"f",
           register_operand:SCALARF:"f"),
  root.2.mode = SCALARF;
}
```

Current Status and Plans for Future Work

- specRTL compiler is ready
- Many of the i386 instructions and all spim instructions have been rewritten
- We invite more people to try out specRTL in writing other descriptions

Conclusions

- Separating shapes from concrete details is very helpful
- It may be possible to identify a large number of common shapes
- Machine descriptions may become much smaller
  Only the concrete details need to be specified
- Non-disruptive and incremental migration to new machine descriptions
- GCC source need not change until these machine descriptions have been found useful