Outline

- Some details of MD constructs
  - On names of patterns in .md files
  - On the role of \texttt{define} and \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

Pattern Names in .md File

All Patterns

Named Patterns
- With *
  - No \texttt{gen\_} function
- Standard
  - gen\_name function
  - Called implicitly
  - Can be called explicitly
- Non-Standard
  - gen\_name function
  - Not called implicitly
  - Can be called explicitly

Unnamed Patterns
- Without *
  - No \texttt{gen\_} function

Part 1

More Features
**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
- some other pass
- preparatory statement of define_expand
- some function in a .c file

**Use of Predicates**

```assembly
(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 using 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**

```assembly
(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")))]
  """

Handling Mode Differences

(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")))]
  "subu\%t %0,%1,%2"
[(set_attr "type" "arith")
 (set_attr "mode" "SI")])

(define_insn "subdi3"
  [(set (match_operand:DI 0 "register_operand" "=d")
    (minus:DI (match_operand:DI 1 "register_operand" "d")
      (match_operand:DI 2 "register_operand" "d")))]
  "dsubu\%t %0,%1,%2"
[(set_attr "type" "arith")
 (set_attr "mode" "DI")])
Mode Iterators: Abstracing Out Mode Differences

(define_mode_iterator GPR [SI (DI 'TARGET_64BIT')])
(define_mode_attr d [(SI "") (DI "d")])
(define_insn "sub<mode>3"
  [(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>sub	0,%1,%2"
  [(set_attr "type" "arith"
             (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: Abstracing 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; }
)
Defining Attributes

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

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

```c
;; Instruction type.
(define_attr "type"
  "other, multi, alu, alu1, 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`

```c
(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

```c
(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.
Part 4

Improving Machine Descriptions

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_expansion`

Design Flaws in Machine Descriptions

Multiple patterns with same structure

* Repetition of almost similar RTL expressions across multiple `define_insn` an `define_expansion` 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

```lisp
[(set (match_operand: m 0 "register_operand" "c0")
    (plus: m (match_operand: m 1 "register_operand" "c1")
      (match_operand: m 2 "p" "c2")))]
```

**RTL Template**

![RTL Template Diagram]

**Details**

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>m</th>
<th>p</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>define_insn</td>
<td>ANYF</td>
<td>register_operand</td>
<td>=f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>add&lt;mode&gt;3</td>
<td>GPR</td>
<td>arith_operand</td>
<td>=d,d</td>
<td>d,d</td>
<td>d,d</td>
</tr>
<tr>
<td>define_insn +add&lt;mode&gt;3</td>
<td>GPR</td>
<td>arith_operand</td>
<td>=d,d</td>
<td>d,d</td>
<td>d,Q</td>
</tr>
</tbody>
</table>

### Redundancy in MIPS Machine Descriptions: Example 2

```lisp
[(set (match_operand: m 0 "register_operand" "c0")
    (mult: m (match_operand: m 1 "register_operand" "c1")
      (match_operand: m 2 "register_operand" "c2")))]
```

**RTL Template**

![RTL Template Diagram]

**Details**

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>m</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>define_insn *mul&lt;mode&gt;3</td>
<td>SCALARF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>define_insn *mul&lt;mode&gt;3</td>
<td>4300</td>
<td>=f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>define_insn mulv2sf3</td>
<td>V2SF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>define_expand_mul&lt;mode&gt;3</td>
<td>GPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>define_insn mul&lt;mode&gt;3</td>
<td>GPR</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>define_insn mul&lt;mode&gt;3</td>
<td>GPR</td>
<td>d,1</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

### Redundancy in MIPS Machine Descriptions: Example 3

```lisp
[(set (match_operand: m 0 "register_operand" "c0")
    (plus: m
t    (mult: m (match_operand: m 1 "register_operand" "c1")
      (match_operand: m 2 "register_operand" "c2"))))]
```

**RTL Template**

![RTL Template Diagram]

**Details**

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>m</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
</tr>
</thead>
<tbody>
<tr>
<td>*mul&lt;acc&gt;si</td>
<td>SI</td>
<td>l*?*?,d?</td>
<td>d,d</td>
<td>d,d</td>
<td>0,d</td>
</tr>
<tr>
<td>*mul&lt;acc&gt;si_r3900</td>
<td>SI</td>
<td>l*?*?,d?</td>
<td>d,d</td>
<td>d,d</td>
<td>0,1,d</td>
</tr>
<tr>
<td>*macc</td>
<td>SI</td>
<td>l,1</td>
<td>d,d</td>
<td>d,d</td>
<td>0,1,1</td>
</tr>
<tr>
<td>*madd4&lt;mode&gt;</td>
<td>ANYF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>*madd3&lt;mode&gt;</td>
<td>ANYF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
<td>0</td>
</tr>
</tbody>
</table>

### 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 général_operand")]
(clobber (reg:CC FLAGS_REG))
"TARGET64BIT"
"ix86_expand_binary_operator (IOR, DImode, operands); DONE;"

##define insn "iordi1-rex64"
[(set (match_operand:DI 0 "nonimmediate_operand" ")=rm,r")
  (ior:DI (match_operand:DI 1 "nonimmediate_operand" ");0")
  (match_operand:DI 2 "x86_64 general_operand" "re,rme")]
(clobber (reg:CC FLAGS_REG))
"TARGET64BIT"
&i ix86_binary_operator ok (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 $\equiv$ tree structure of RTL templates
  - Details $\equiv$ 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

Creating Abstract Patterns

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

abstract set_macc extends set_plus {
  root.2.2 = mult;
}
```
Creating Concrete Patterns

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

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:""); 
  root.2.mode = GPR; 
}

Generating Conventional Machine Descriptions

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

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:""); 
  root.2.mode = GPR; 
}

Resulting MD Specification

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

Overriding Details

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

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

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

Currently Status and Plans for Future Work

- specRTL parser has been augmented with semantic checks
  Emitting conventional machine descriptions is pending
- i386 move instructions and spim add instructions have been rewritten
  Other instructions are being rewritten
- Suggestions have been received to improve the syntax
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