Outline

- Some details of MD constructs
  - On names of patterns in .md files
  - On the role of define, 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 *
  - No gen function

- Without *
  - No gen function

Unnamed Patterns

- No gen_function

Standard

- gen_name function
  - Called implicitly
  - Can be called explicitly

Non-Standard

- 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

### Notes
- Using define_expand for Generating RTL statements
- Calling gen_pattern function
  - implicit call
  - explicit call
    - non-standard pattern
      - during expansion
    - standard pattern
      - during expansion
    - some other pass
      - preparatory statement of define_expand
      - some function in a .c file
Use of Predicates

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)

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:

- \( \text{(define_insn "" } \)
  
  \([ (\text{set (match_operand:SI 0 "general_operand" "]=r"]}} \)
  
  \(\text{(plus:SI (match_dup 0) \)
  \(\text{\text{(match_operand:SI 1 "general_operand" "]r"]}}))]) \)
  
  \(\text{"..."}\)

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

- \( \text{(define_insn "" } \)
  
  \([ (\text{set (match_operand:SI 0 "general_operand" "]=r"]}} \)
  
  \(\text{(plus:SI (match_operand:SI 1 "general_operand" "]z"]}} \)
  
  \(\text{\text{(match_operand:SI 2 "general_operand" "]r"]}}))]) \)
  
  \(\text{"..."}\)

Notes

- Consider an insn for recognition
  
  \(\text{(insn n prev next} \)
  \(\text{(set (reg:SI 3) \)
  \(\text{(plus:SI (reg:SI 6) (reg:SI 109))) \)
  \(\text{...}))}\)

- 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

- \(\text{(insn n2 prev n} \)
  \(\text{(set (reg:SI 3) (reg:SI 6)) \)
  \(\text{...})\)

- \(\text{(insn n2 next} \)
  \(\text{(set (reg:SI 3) \)
  \(\text{(plus:SI (reg:SI 3) (reg:SI 109))) \)
  \(\text{...})\)
Part 2

Factoring Out Common Information

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")))]
  "sub\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")))]
  "dsub\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "DI")])
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>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")))]
  ""
  "{subu\t%0,%1,%2"
  [(set_attr "type" "arith")
   (set_attr "mode" "<MODE>")])

Notes

Handling Code Differences

(define_expand "unordered"
  [(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;
})
(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

;; 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

(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

\begin{verbatim}
(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"
\end{verbatim}

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 nor 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_insns 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 constraints earlier than the current place
    ⇒ May get rid of some C code in define expands

Design Flaws in Machine Descriptions

Multiple patterns with same structure
- 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

\[
\left(\text{set (match_operand: } m \text{ 0 "register_operand" } \text{"c0"})
\left(\text{plus: } m \text{ (match_operand: } m \text{ 1 "register_operand" } \text{"c1"})
\text{(match_operand: } m \text{ 2 "p" } \text{"c2"))}\right)\]

**RTL Template**

```
= +
```

**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 add&lt;mode&gt;3</td>
<td>ANYF</td>
<td>register_operand</td>
<td>=f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>define_expand add&lt;mode&gt;3</td>
<td>GPR</td>
<td>arith_operand</td>
<td></td>
<td></td>
<td></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

\[
\left(\text{set (match_operand: } m \text{ 0 "register_operand" } \text{"c0"})
\left(\text{mult: } m \text{ (match_operand: } m \text{ 1 "register_operand" } \text{"c1"})
\text{(match_operand: } m \text{ 2 "register_operand" } \text{"c2"))}\right)\]

**RTL Template**

```
= *
```

**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>SCALARF</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,d</td>
<td>d,d</td>
</tr>
</tbody>
</table>
Redundancy in MIPS Machine Descriptions: Example 3

\[
\begin{array}{c}
\text{RTL Template} \\
\text{Details}
\end{array}
\]

<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_acc_si SI</td>
<td>SI</td>
<td>=1,*d?,d?</td>
<td>d,d</td>
<td>d</td>
<td>0,d</td>
</tr>
<tr>
<td>*mul_acc_si_r3900 SI</td>
<td>SI</td>
<td>=1,*d?,d?</td>
<td>d,d</td>
<td>d,d</td>
<td>d,d</td>
</tr>
<tr>
<td>*macc SI</td>
<td>SI</td>
<td>=1,d</td>
<td>d,d</td>
<td>d</td>
<td>0,1</td>
</tr>
<tr>
<td>*madd4&lt;mode&gt; ANYF</td>
<td>ANYF</td>
<td>=f</td>
<td>f</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>*madd3&lt;mode&gt; ANYF</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 cannot 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_expand_binary_operator (IOR, DImode, operands); DONE;"

(define_insn "*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,rm0")))
  (clobber (reg:CC FLAGS_REG))]
"TARGET_64BIT"
&k 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 ≡ tree structure of RTL templates
  - Details ≡ attributes of tree nodes
    (eg. 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;
}
```

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

Essential Abstractions in GCC
GCC Resource Center, IIT Bombay
Creating Concrete Patterns

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

cornerstone 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;
}
cornerstone 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

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

cornerstone 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;
}
{: /* Conventional Machine Description Fragments */ :}

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

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

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");
}

Notes

Current 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