Workshop on Essential Abstractions in GCC

Incremental Machine Descriptions for Spim: Levels 2, 3, and 4

GCC Resource Center
(www.cse.iitb.ac.in/grc)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

2 July 2011
Outline

• Constructs supported in level 2
• Constructs supported in level 3
• Constructs supported in level 4
Part 1

Constructs Supported in Level 2
## Arithmetic Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 - \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j - R_k )</td>
<td>sub ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow -\text{Src} )</td>
<td>( R_i \leftarrow -R_j )</td>
<td>neg ri, rj</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 / \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j / R_k )</td>
<td>div rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 % \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j % R_k )</td>
<td>rem ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td>( \text{Dest} \leftarrow \text{Src}_1 \times \text{Src}_2 )</td>
<td>( R_i \leftarrow R_j \times R_k )</td>
<td>mul ri, rj, rk</td>
<td></td>
</tr>
</tbody>
</table>
## Arithmetic Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Dest ← Src_1 − Src_2</code></td>
<td>( R_i ← R_j − R_k )</td>
<td>sub ( ri, rj, rk )</td>
<td></td>
</tr>
<tr>
<td><code>Dest ← −Src</code></td>
<td>( R_i ← −R_j )</td>
<td>neg ( ri, rj )</td>
<td></td>
</tr>
<tr>
<td><code>Dest ← Src_1/Src_2</code></td>
<td>( R_i ← R_j/R_k )</td>
<td>( \text{div} rj, rk )</td>
<td>level 2</td>
</tr>
<tr>
<td>( \text{mflo} ri )</td>
<td></td>
<td>( \text{rem} ri, rj, rk )</td>
<td>( \text{mul} ri, rj, rk )</td>
</tr>
</tbody>
</table>
## Bitwise Operations Required in Level 2

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dest ← $S_{rc1} \ll S_{rc2}$</td>
<td>$R_i \leftarrow R_j \ll R_k$</td>
<td>sllv ri, rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td></td>
<td>$R_i \leftarrow R_j \ll C_5$</td>
<td>sll ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← $S_{rc1} \gg S_{rc2}$</td>
<td>$R_i \leftarrow R_j \gg R_k$</td>
<td>srav ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R_i \leftarrow R_j \gg C_5$</td>
<td>sra ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← $S_{rc1} &amp; S_{rc2}$</td>
<td>$R_i \leftarrow R_j &amp; R_k$</td>
<td>and ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R_i \leftarrow R_j &amp; C$</td>
<td>andi ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← $S_{rc1}</td>
<td>S_{rc2}$</td>
<td>$R_i \leftarrow R_j \mid R_k$</td>
<td>or ri, rj, rk</td>
</tr>
<tr>
<td></td>
<td>$R_i \leftarrow R_j \mid C$</td>
<td>ori ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← $S_{rc1} ^ S_{rc2}$</td>
<td>$R_i \leftarrow R_j ^ R_k$</td>
<td>xor ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$R_i \leftarrow R_j ^ C$</td>
<td>xori ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← $\sim S_{rc}$</td>
<td>$R_i \leftarrow \sim R_j$</td>
<td>not ri, rj</td>
<td></td>
</tr>
</tbody>
</table>
Divide Operation in `spim2.md` using `define_insn`

- For division, the spim architecture imposes use of multiple `asm` instructions for single operation.

```
(define_insn "divsi3"
 [(set (match_operand:SI 0 "register_operand" ";r")
       (div:SI (match_operand:SI 1 "register_operand" ;"r")
               (match_operand:SI 2 "register_operand" ;"r")))]
 ""
 "div\t%1, %2\n\tmflo\t%0"
)
```
Divide Operation in `spim2.md` using `define_insn`

- For division, the spim architecture imposes use of multiple asm instructions for single operation.
- Two ASM instructions are emitted using single RTL pattern

```
(define_insn "divsi3"
 [(set (match_operand:SI 0 "register_operand" ";=r")
   (div:SI (match_operand:SI 1 "register_operand" ";r")
     (match_operand:SI 2 "register_operand" ";r")))]
 ""
 "\tdiv\t%1, %2\n\ttmflo\t%0"
)
```
Advantages/Disadvantages of using define_insn

- Very simple to add the pattern
- Primitive target feature represented as single insn pattern in .md
- Unnecessary atomic grouping of instructions
- May hamper optimizations in general, and instruction scheduling, in particular
Divide Operation in spim2.md using define_expand

- The RTL pattern can be expanded into two different RTLS.

```lisp
(define_expand "divsi3"
  [(parallel
    [(set (match_operand:SI 0 "register_operand" ")")
      (div:SI (match_operand:SI 1 "register_operand" "")
        (match_operand:SI 2 "register_operand" ""))]
    (clobber (reg:SI 26))
    (clobber (reg:SI 27)))]"

{"emit_insn(gen_IITB_divide(gen_rtx_REG(SImode,26),
  operands[1], operands[2]));
emit_insn(gen_IITB_move_from_lo(operands[0],
  gen_rtx_REG(SImode,26)));
DONE;"}
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Divide Operation in *spim2.md* using *define_expand*

- Divide pattern equivalent to div instruction in architecture.

```
(define_insn "IITB_divide"
  [(parallel
    [(set (match_operand:SI 0 "LO_register_operand" "=q")
      (div:SI (match_operand:SI 1 "register_operand" "r")
        (match_operand:SI 2 "register_operand" "r")))
     (clobber (reg:SI 27))])
  ""
  "div t%1, %2"
  )
```
Divide Operation in spim2.md using define_expan

- Divide pattern equivalent to div instruction in architecture.

```lisp
(define_insn "IITB_divide"
 [(parallel [(set (match_operand:SI 0 "LO_register_operand" "=q")
                (div:SI (match_operand:SI 1 "register_operand" "r")
                        (match_operand:SI 2 "register_operand" "r"))
                (clobber (reg:SI 27)))]
   ""
   "div t%1, %2"
)
```
Divide Operation in `spim2.md` using `define_expant` 

- Moving contents of special purpose register LO to/from general purpose register

```lisp
(define_insn "IITB_move_from_lo"
  [(set (match_operand:SI 0 "register_operand" ";=r")
       (match_operand:SI 1 "LO_register_operand" ";q"))]
  ""
  "mflo \t%0"
)

(define_insn "IITB_move_to_lo"
  [(set (match_operand:SI 0 "LO_register_operand" ";q")
       (match_operand:SI 1 "register_operand" ";r"))]
  ""
  "mtlo \t%1"
)
```
Divide Operation in `spim2.md` using `define_expand`

- Divide pattern equivalent to div instruction in architecture.

```scheme
(define_insn "modsi3"
[(parallel
  [(set (match_operand:SI 0 "register_operand" ";=r")
    (mod:SI (match_operand:SI 1 "register_operand" ";r")
      (match_operand:SI 2 "register_operand" ";r")))
   (clobber (reg:SI 26))
   (clobber (reg:SI 27))])"
  "rem \t%0, %1, %2"
)
```
Divide Operation in spim2.md using define_expand

- Divide pattern equivalent to div instruction in architecture.

(define_insn "modsi3"
  [(parallel[(set (match_operand:SI 0 "register_operand" ";=r")
     (mod:SI (match_operand:SI 1 "register_operand" ";r")
     (match_operand:SI 2 "register_operand" ";r")))
     (clobber (reg:SI 26))
     (clobber (reg:SI 27)))]
  ""
  "rem \t%0, %1, %2"
)
Advantages/Disadvantages of Using `define_expand_for` Division

- Two instructions are separated out at GIMPLE to RTL conversion phase
- Both instructions can undergo all RTL optimizations independently
- C interface is needed in `md`
- Compilation becomes slower and requires more space
(define_split
  [(parallel [(set (match_operand:SI 0 "register_operand" "")
    (div:SI (match_operand:SI 1 "register_operand" "")
      (match_operand:SI 2 "register_operand" ""))
    )
    (clobber (reg:SI 26))
    (clobber (reg:SI 27)))]"

[(parallel [(set (match_dup 3)
    (div:SI (match_dup 1)
      (match_dup 2))
    (clobber (reg:SI 27))]
    (set (match_dup 0)
      (match_dup 3))
  ]
    "operands[3]=gen_rtx_REG(SImode,26); "
)
Divide Operation in `spim2.md` using `define_split`

```
(define_split
 [(parallel [(set (match_operand:SI 0 "register_operand" "")
                   (div:SI (match_operand:SI 1 "register_operand" "")
                   (match_operand:SI 2 "register_operand" ""))
                 )
               (clobber (reg:SI 26))
               (clobber (reg:SI 27)))]

""

[(parallel [(set (match_dup 3)
              (div:SI (match_dup 1)
              (match_dup 2)))
            (clobber (reg:SI 27)))]

(set (match_dup 0)
     (match_dup 3))
]

"operands[3]=gen_rtx_REG(SImode,26); "
```

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
Divide Operation in spim2.md using define_split

(define_split
  [(parallel [(set (match_operand:SI 0 "register_operand" ""))
      (div:SI (match_operand:SI 1 "register_operand" ""))
      (match_operand:SI 2 "register_operand" ""))])
  (clobber (reg:SI 26))
  (clobber (reg:SI 27)))]
  ""

[(parallel [(set (match_dup 3)
      (div:SI (match_dup 1)
      (match_dup 2)))
    (clobber (reg:SI 27))])
  (set (match_dup 0)
    (match_dup 3))]

"operands[3]=gen_rtx_REG(SImode,26); "
)
Part 2

Constructs Supported in Level 3
## Operations Required in Level 3

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Dest} \leftarrow \text{fun}(P_1, \ldots, P_n) )</td>
<td>( \text{call } L_{\text{fun}}, n )</td>
<td>\begin{align*} \text{lwr}_i, &amp; \ [\text{SP}+\text{c1}] \ \text{swr}_i, &amp; \ [\text{SP}] \ \text{lwr}_i, &amp; \ [\text{SP}+\text{c2}] \ \text{swr}_i, &amp; \ [\text{SP}-%4] \end{align*}</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{jal } L )</td>
<td>New</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{Dest} \leftarrow $v0 )</td>
<td>level 1</td>
</tr>
<tr>
<td>( \text{fun}(P_1, P_2, \ldots, P_n) )</td>
<td>( \text{call } L_{\text{fun}}, n )</td>
<td>\begin{align*} \text{lwr}_i, &amp; \ [\text{SP}+\text{c1}] \ \text{swr}_i, &amp; \ [\text{SP}] \ \text{lwr}_i, &amp; \ [\text{SP}+\text{c2}] \ \text{swr}_i, &amp; \ [\text{SP}-%4] \end{align*}</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \text{jal } L )</td>
<td>New</td>
</tr>
</tbody>
</table>
(define_insn "call"
  [(call (match_operand:SI 0 "memory_operand" "m")
          (match_operand:SI 1 "immediate_operand" "i"))
   (clobber (reg:SI 31))
  ]
  ""
  "*
   return emit_asm_call(operands,0);
  "")
Call Operation in `spim3.md`

```lisp
(defun insn "call_value"
  [(set (match_operand:SI 0 "register_operand" ":=r")
       (call (match_operand:SI 1 "memory_operand" "m")
             (match_operand:SI 2 "immediate_operand" "i")))
   (clobber (reg:SI 31))
  ]
"
"*

  return emit_asm_call(operands,1);
"
)
```
Activation Record Generation during Call

- Operations performed by caller

- Operations performed by callee
Activation Record Generation during Call

- Operations performed by caller
- Operations performed by callee

Caller’s Activation Record
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

### Caller’s Activation Record

- Parameter $n$
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
\textbf{Activation Record Generation during Call}

- Operations performed by caller
  - Push parameters on stack.

- Operations performed by callee

\begin{center}
\begin{tabular}{|l|}
\hline
Caller’s Activation Record \\
\hline
Parameter \( n \) \\
\hline
Parameter \( n - 1 \) \\
\hline
\ldots \\
\hline
Parameter 1 \\
\hline
\end{tabular}
\end{center}
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee

<table>
<thead>
<tr>
<th>Caller's Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.
- Operations performed by callee
  - Push Return address stored by caller on stack.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.

---

### Caller’s Activation Record

<table>
<thead>
<tr>
<th>Parameter $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
</tbody>
</table>
Activation Record Generation during Call

• Operations performed by caller
  ▶ Push parameters on stack.
  ▶ Load return address in return address register.
  ▶ Transfer control to Callee.

• Operations performed by callee
  ▶ Push Return address stored by caller on stack.
  ▶ Push caller’s Frame Pointer Register.
  ▶ Push caller’s Stack Pointer.
  ▶ Save callee saved registers, if used by callee.
  ▶ Create local variables frame.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
</tbody>
</table>
### Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

---

**Caller’s Activation Record**

- Parameter $n$
- Parameter $n-1$
- ...
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
- Local Variable 1
- Local Variable 2
Activation Record Generation during Call

- **Operations performed by caller**
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- **Operations performed by callee**
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

---

**Caller’s Activation Record**

- Parameter $n$
- Parameter $n - 1$
- ...
- Parameter 1
- Return Address
- Caller’s FPR (Control Link)
- Caller’s SPR
- Callee Saved Registers
  - Local Variable 1
  - Local Variable 2
  - ...

---

**Essential Abstractions in GCC**

GCC Resource Center, IIT Bombay
Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.
- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n - 1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
<tr>
<td>Local Variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local Variable $n$</td>
</tr>
</tbody>
</table>
 Activation Record Generation during Call

- Operations performed by caller
  - Push parameters on stack.
  - Load return address in return address register.
  - Transfer control to Callee.

- Operations performed by callee
  - Push Return address stored by caller on stack.
  - Push caller’s Frame Pointer Register.
  - Push caller’s Stack Pointer.
  - Save callee saved registers, if used by callee.
  - Create local variables frame.
  - Start callee body execution.

<table>
<thead>
<tr>
<th>Caller’s Activation Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter $n$</td>
</tr>
<tr>
<td>Parameter $n-1$</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Parameter 1</td>
</tr>
<tr>
<td>Return Address</td>
</tr>
<tr>
<td>Caller’s FPR (Control Link)</td>
</tr>
<tr>
<td>Caller’s SPR</td>
</tr>
<tr>
<td>Callee Saved Registers</td>
</tr>
<tr>
<td>Local Variable 1</td>
</tr>
<tr>
<td>Local Variable 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Local Variable $n$</td>
</tr>
</tbody>
</table>
(define_expand "prologue"
  [(clobber (const_int 0))]
  ""
  {
    spim_prologue();
    DONE;
  })
(define_expand "prologue"
  [(clobber (const_int 0))]
  ""
  {
    spim_prologue();
    DONE;
  })

(set (mem:SI (reg:SI $sp))
  (reg:SI 31 $ra))

(set (mem:SI (plus:SI (reg:SI $sp)
    (const_int -4 )))
  (reg:SI $sp))

(set (mem:SI (plus:SI (reg:SI $sp)
    (const_int -8 )))
  (reg:SI $fp))

(set (reg:SI $fp)
  (reg:SI $sp))

(set (reg:SI $sp)
  (plus:SI (reg:SI $fp)
    (const_int -36)))
Epilogue in spim3.md

(define Expand "epilogue"
  [(clobber (const-int 0))]"

   spim_epilogue();
   DONE;
)

(set (reg:SI $sp)
     (reg:SI $fp))

(set (reg:SI $fp)
     (mem:SI (plus:SI (reg:SI $sp)
                        (const-int -8 ))))

(set (reg:SI $ra)
     (mem:SI (reg:SI $sp)))

(parallel [
            (return)
            (use (reg:SI $ra))])
Part 3

Constructs Supported in Level 4
## Operations Required in Level 4

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
</table>
| $\text{Src}_1 < \text{Src}_2$ ? goto L : PC | $CC \leftarrow R_i < R_j$  
$CC < 0$ ? goto L : PC | blt $r_i, r_j, L$ | |
| $\text{Src}_1 > \text{Src}_2$ ? goto L : PC | $CC \leftarrow R_i > R_j$  
$CC > 0$ ? goto L : PC | bgt $r_i, r_j, L$ | |
| $\text{Src}_1 \leq \text{Src}_2$ ? goto L : PC | $CC \leftarrow R_i \leq R_j$  
$CC \leq 0$ ? goto L : PC | ble $r_i, r_j, L$ | |
| $\text{Src}_1 \geq \text{Src}_2$ ? goto L : PC | $CC \leftarrow R_i \geq R_j$  
$CC \geq 0$ ? goto L : PC | bge $r_i, r_j, L$ | |
# Operations Required in Level 4

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
</table>
| $Src_1 == Src_2$ ? goto L : PC | $CC \leftarrow R_i == R_j$  
$CC == 0$ ? goto L : PC | beq $r_i, r_j, L$ |
| $Src_1 \neq Src_2$ ? goto L : PC | $CC \leftarrow R_i \neq R_j$  
$CC \neq 0$ ? goto L : PC | bne $r_i, r_j, L$ |
(define_insn "cbranchsi4"
  [(set (pc)
    (if_then_else
     (match_operator:SI 0 "comparison_operator"
      [(match_operand:SI 1 "register_operand" "")
       (match_operand:SI 2 "register_operand" "")])
     (label_ref (match_operand 3 "" ""))
     (pc)))]

  ""
  "*
   return conditional_insn(GET_CODE(operands[0]),operands);
 "
)

Conditional Branch Instruction in spim4.md
Support for Branch pattern in `spim4.c`

```c
char *
conditional_insn (enum rtx_code code, rtx operands[])
{
  switch(code)
  {
    case EQ:return "beq %1, %2, %l3";
    case NE:return "bne %1, %2, %l3";
    case GE:return "bge %1, %2, %l3";
    case GT:return "bgt %1, %2, %l3";
    case LT:return "blt %1, %2, %l3";
    case LE:return "ble %1, %2, %l3";
    case GEU:return "bgeu %1, %2, %l3";
    case GTU:return "bgtu %1, %2, %l3";
    case LTU:return "bltu %1, %2, %l3";
    case LEU:return "bleu %1, %2, %l3";
    default: /* Error. Issue ICE */
  }
}
```
(define_code_iterator cond_code
    [lt ltu eq ge geu gt gtu le leu ne])

(define_expand "cmpsi"
    [(set (cc0) (compare
        (match_operand:SI 0 "register_operand" "")
        (match_operand:SI 1 "nonmemory_operand" "")))]
    ""
    {
        compare_op0=operands[0];
        compare_op1=operands[1];
        DONE;
    }
)
Alternative for Branch: Branch pattern in `spim4.md`

```lisp
(define_expand "b<code>"
  [(set (pc) (if_then_else (cond_code:SI (match_dup 1)
                                      (match_dup 2))
         (label_ref (match_operand 0 "" ""))
         (pc)))]

"
{
  operands[1]=compare_op0;
  operands[2]=compare_op1;
  if(immediate_operand(operands[2],SImode))
  {
  }
}
)
Alternative for Branch: Branch pattern in spim4.md

(define_insn "*insn_b<code>"
 [(set (pc)
   (if_then_else
    (cond_code:SI
     (match_operand:SI 1 "register_operand" "r")
     (match_operand:SI 2 "register_operand" "r"))
     (label_ref (match_operand 0 "" ""))(pc))))]
 ""
 "*

 return conditional_insn(<CODE>,operands);
"
)