Outline

- Constructs supported in level 2
- Constructs supported in level 3
- Constructs supported in level 4
# 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><code>Dest ← Src₁ - Src₂</code></td>
<td><code>Rᵢ ← Rⱼ - Rₖ</code></td>
<td>sub ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td><code>Dest ← -Src</code></td>
<td><code>Rᵢ ← -Rⱼ</code></td>
<td>neg ri, rj</td>
<td></td>
</tr>
<tr>
<td><code>Dest ← Src₁/Src₂</code></td>
<td><code>Rᵢ ← Rⱼ/Rₖ</code></td>
<td>div rj, rk</td>
<td>level 2</td>
</tr>
<tr>
<td><code>Dest ← Src₁%Src₂</code></td>
<td><code>Rᵢ ← Rᵢ%Rₖ</code></td>
<td>rem ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td><code>Dest ← Src₁ * Src₂</code></td>
<td><code>Rᵢ ← Rⱼ * Rₖ</code></td>
<td>mul ri, rj, rk</td>
<td></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 ← Src₁ &lt;&lt; Src₂</td>
<td>Rᵢ ← Rⱼ &lt;&lt; Rₖ</td>
<td>sllv ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rᵢ ← Rⱼ &lt;&lt; C₅</td>
<td>sll ri, rj, c</td>
</tr>
<tr>
<td>Dest ← Src₁ &gt;&gt; Src₂</td>
<td>Rᵢ ← Rⱼ &gt;&gt; Rₖ</td>
<td>srav ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rᵢ ← Rⱼ &gt;&gt; C₅</td>
<td>sra ri, rj, c</td>
</tr>
<tr>
<td>Dest ← Src₁ &amp; Src₂</td>
<td>Rᵢ ← Rⱼ &amp; Rₖ</td>
<td>and ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rᵢ ← Rⱼ &amp; C</td>
<td>andi ri, rj, c</td>
</tr>
<tr>
<td>Dest ← Src₁</td>
<td>Rᵢ ← Rⱼ</td>
<td>or ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rᵢ ← Rⱼ</td>
<td>ori ri, rj, c</td>
</tr>
<tr>
<td>Dest ← Src₁ ^ Src₂</td>
<td>Rᵢ ← Rⱼ ^ Rₖ</td>
<td>xor ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rᵢ ← Rⱼ ^ C</td>
<td>xori ri, rj, c</td>
</tr>
<tr>
<td>Dest ← ~ Src</td>
<td>Rᵢ ← ~ Rⱼ</td>
<td>not ri, rj</td>
<td></td>
</tr>
</tbody>
</table>

---

### Notes

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

Two ASM instructions are emitted using single RTL pattern

```python
(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"
)
```
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

Notes

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay

Divide Operation in `spim2.md` using `define_expand`

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

```c
(define_expand "divs13"
  [(parallel [\(\text{set} \text{match\_operand:SI 0 "register\_operand" ""}\\
              \(\text{div:SI} \text{match\_operand:SI 1 "register\_operand" ""}\\
                \(\text{match\_operand:SI 2 "register\_operand" ""})\\
          ]\\
          \clobber\(\text{reg:SI 26})\\
          \clobber\(\text{reg:SI 27})\\
        ]"
    ""

    \{\n      \text{emit\_insn}gen\_IITB\_divide(gen\_rtx\text{REG(SImode,26)},
                    operands[1], operands[2]));
      \text{emit\_insn}gen\_IITB\_move\_from\_lo(operands[0],
                    gen\_rtx\text{REG(SImode,26)});
      \text{DONE};\n    \}\n  )
```

Notes

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.

```c
(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"
)
```

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

```c
(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"
)
```

```c
(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_expande`

- 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"
 )
```

Advantages/Disadvantages of Using `define_expande` 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
Divide Operation in `spim2.md` using `define_split`

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

**Notes**

Essential Abstractions in GCC

GCC Resource Center, IIT Bombay
### 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>Dest ← fun(P₁,...,Pₙ)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>call Lₙₙₙ</td>
<td>lw r₁, [SP+c₁]</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw r₁, [SP]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw r₁, [SP+c₂]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw r₁, [SP-n*4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>jal L</td>
<td>New</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest ← $v₀</td>
<td>level 1</td>
</tr>
<tr>
<td>fun(P₁,P₂,...,Pₙ)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>call Lₙₙₙ</td>
<td>lw r₁, [SP+c₁]</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw r₁, [SP]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw r₁, [SP+c₂]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw r₁, [SP-n*4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>jal L</td>
<td>New</td>
</tr>
</tbody>
</table>

### Call Operation in `spim3.md`

```c
(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);
  ""
)```
(define_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);"
  )

- 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.
Prologue in `spim3.md`

```lisp
(define expand "prologue" [(clobber (const int 0))] ""
{
  spim_prologue();
  DONE;
}
)
```

Epilogue in `spim3.md`

```lisp
(define expand "epilogue" [(clobber (const int 0))] ""
{
  spim_epilogue();
  DONE;
}
```

Notes
### Operations Required in Level 4

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Src_1 == Src_2 ? goto L : PC}</td>
<td>\texttt{CC ← R_i == R_j} \texttt{\quad CC == 0 ? goto L : PC}</td>
<td>\texttt{beq r_i, r_j, L}</td>
<td></td>
</tr>
<tr>
<td>\texttt{Src_1 ≠ Src_2 ? goto L : PC}</td>
<td>\texttt{CC ← R_i ≠ R_j} \texttt{\quad CC ≠ 0 ? goto L : PC}</td>
<td>\texttt{bne r_i, r_j, L}</td>
<td></td>
</tr>
</tbody>
</table>

Notes

```
(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" ")])
        (match_operand 3 "" ")))
      (pc))]

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

Alternative for Branch: Conditional compare in spim4.md

```c
(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;
    })
```
```
(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)))]
  ""
  
  
  operands[1]=compare_op0;
  operands[2]=compare_op1;
  if(immediate_operand(operands[2],SImode))
  
  
  return conditional_insn(<CODE>,operands);
  "*

```

```
(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))
  
  
)
```