Part 1

**Constructs Supported in Level 2**

### Essential Abstractions in GCC

**Constructs Supported in Level 2**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Primitive Variants</th>
<th>Implementation</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Dest ← Src_1 − Src_2$</td>
<td>$R_i ← R_j − R_k$</td>
<td>sub $ri, rj, rk$</td>
<td></td>
</tr>
<tr>
<td>$Dest ← −Src$</td>
<td>$R_i ← −R_j$</td>
<td>neg $ri, rj$</td>
<td></td>
</tr>
<tr>
<td>$Dest ← Src_1/Src_2$</td>
<td>$R_i ← R_j/R_k$</td>
<td>div $rj, rk$</td>
<td></td>
</tr>
<tr>
<td>$Dest ← Src_1%Src_2$</td>
<td>$R_i ← R_j%R_k$</td>
<td>rem $ri, rj, rk$</td>
<td></td>
</tr>
<tr>
<td>$Dest ← Src_1 * Src_2$</td>
<td>$R_i ← R_j * R_k$</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 ≪ Rk</td>
<td>sllv ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rl ← Rj ≪ C5</td>
<td>sll ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src1 &gt;&gt; Src2</td>
<td>Ri ← Rj &gt;&gt; Rk</td>
<td>sra ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rl ← Rj &gt;&gt; C5</td>
<td>sra ri, rj, c</td>
<td></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>Rl ← Rj &amp; C</td>
<td>andi ri, rj, c</td>
<td></td>
</tr>
<tr>
<td>Dest ← Src1</td>
<td>Src2</td>
<td>Rl ← Rj</td>
<td>or ri, rj, rk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rl ← Rj &amp; C</td>
<td>ori ri, rj, c</td>
</tr>
<tr>
<td>Dest ← Src1 ^ Src2</td>
<td>Rl ← Rj ^ Rk</td>
<td>xor ri, rj, rk</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rl ← Rj ^ C</td>
<td>xori ri, rj, c</td>
</tr>
<tr>
<td>Dest ← ~ Src</td>
<td>Rl ← ~ Rj</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.
- 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"))
)
"
"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

Divide Operation in spim2.md using define_expand

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

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

```c
{ 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;
}
```
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, \t%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
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_regx(REG(SImode,26);"
)

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>call L₁fun,n</td>
<td>lw rᵢ, [SP+c1] sw rᵢ, [SP]</td>
<td>Level 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1w rᵢ, [SP+c2] sw rᵢ, [SP-n+4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>jal L</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dest ← $v₀</td>
<td>New</td>
</tr>
</tbody>
</table>

Call Operation in spim3.md

(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`

```
(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);"
)
```

Activiation 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.

### Prologue in `spim3.md`
```
(define_expand "prologue"
  [(clobber (const int 0))]
  ""
  {spim_prologue();
   DONE;
  }
```

### Epilogue in `spim3.md`
```
(define_expand "epilogue"
  [(clobber (const int 0))]
  ""
  {spim_epilogue();
   DONE;
  }
```

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
  - ...
  - Local Variable n
### 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 = \text{Src}_2 ) ( \text{goto} \ L : \text{PC} )</td>
<td>( CC \leftarrow R_i = R_j ) ( CC = 0 \ ? \ 	ext{goto} \ L : \text{PC} )</td>
<td>beq ( r_i, r_j, L )</td>
<td></td>
</tr>
</tbody>
</table>
| \( \text{Src}_1 \neq \text{Src}_2 \) \( \text{goto} \ L : \text{PC} \) | \( CC \leftarrow R_i \neq R_j \) \( CC 
eq 0 \ ? \ 	ext{goto} \ L : \text{PC} \) | bne \( r_i, r_j, L \) | |

### Conditional Branch Instruction in `spim4.md`

```c
(define_insn "cbbranchis4"
 [(set (pc)
   (if_then Else
   (match_operator:SI 0 "comparison_operator"
   [(match_operand:SI 1 "register_operand" "")
    (match_operand:SI 2 "register_operand" "")]
   (match_operator:SI 0 "comparison_operator"
   [(match_operand:SI 1 "register_operand" "")
    (match_operand:SI 2 "register_operand" "")]
   (match_operand:SI 3 "label_operand" ""))]
   (label_ref (match_operand 3 "" "")
   (pc))))
 "*
 return conditional_insn(GET_CODE(operands[0]),operands);
"
```

---

**Essential Abstractions in GCC**  
**GCC Resource Center, IIT Bombay**
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`

```markdown
(define_code_iterator cond_code

[lt ltu eq ge geu gt gtu le leu ne])
```

```c
(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`

```markdown
(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);
```

```c
```

Alternative for Branch: Branch pattern in `spim4.md`

```markdown
```