### Compiling a switch statement

```java
switch (k) {
    case 0:  f = i + j; break;
    case 1:  f = g + h; break;
    case 2:  f = g - h; break;
    case 3:  f = i - j; break;
}
```

Assume, $s0$-$s5$ contain $f, g, h, i, j, k,$ and $t2$ stores 4. The following codes check if $k < 0$ or $k > 3$, and skips to "Exit". Otherwise it jumps to the right case

```assembly
slt $t3, $s5, $zero  # if k < 0 then $t3 = 1 else $t3=0
bne $t3, $zero, Exit  # if k < 0 then Exit
slt $t3, $s5, $t2    # if k<4 then $t3 = 1 else $t3=0
beq $t3, $zero, Exit  # if k ≥ 4 the Exit
```
jumptable register $t4

32-bit address L0
32-bit address L1
32-bit address L2
32-bit address L3

f = i + j
J Exit

f = g+h
j Exit

Base address of the jumptable

register $t4

MEMORY
Here is the remainder of the program:

```
add $t1, $s5, $s5        # t1 = 2*k
add $t1, $t1, $t1        # t1 = 4*k
add $t1, $t1, $t4        # t1 = base address + 4*k
lw  $t0, 0($t1)          # load the addr pointed by t1 into t0
jr   $t0                # jump to addr pointed by t0
L0:  add $s0, $s3, $s4   # f = i + j
     J Exit
L1:  add $s0, $s1, $s2   # f = g+h
     J Exit
L2:  sub $s0, $s1, $s2   # f = g-h
     J Exit
L3:  sub $s0, $s3, $s4   # f = i-j
Exit: <next instruction> Done
```
The instruction format for jump

\[ J \quad 10000 \] is represented as

\[
\begin{array}{c|c}
2 & 2500 \\
\hline
6\text{-bits} & 26\text{ bits}
\end{array}
\]

This is the \textit{J-type format} of MIPS instructions.

Conditional branch is represented using I-type format:

\[ \text{bne } \$s0, \$s1, \text{Label} \]

is represented as

\[
\begin{array}{c|c|c|c}
5 & 16 & 17 & \\
\hline
6 & 5 & 5 & \text{16-bit offset}
\end{array}
\]

Current \textit{PC} + (4 \times \text{offset}) determines the branch target \textit{Label}.

This is called \textit{PC-relative addressing}.
Revisiting machine language of MIPS

# starts from 80000

Loop:
- add $t1, $s3, $s3
- add $t1, $t1, $t1
- add $t1, $t1, $s6
- lw $t0, 0($t1)
- bne $t0, $s5, Exit
- add $s3, $s3, $s4
- j Loop

Exit:

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>5</th>
<th>6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>80008</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>80012</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80016</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
<td>(why?)</td>
<td>I-type</td>
</tr>
<tr>
<td>80020</td>
<td>0</td>
<td>19</td>
<td>20</td>
<td>19</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>80024</td>
<td>2</td>
<td>20000</td>
<td>(why?)</td>
<td></td>
<td></td>
<td>J-type</td>
</tr>
<tr>
<td>80028</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Addressing Modes**

*What are the different ways to access an operand?*

- **Register addressing**
  
  Operand is in register
  
  \[
  \text{add } s1, s2, s3 \text{ means } s1 \leftarrow s2 + s3
  \]

- **Base addressing**

  Operand is in memory.

  The address is the sum of a register and a constant.

  \[
  \text{lw } s1, 32(s3) \text{ means } s1 \leftarrow M[s3 + 32]
  \]

  As special cases, you can implement

  - **Direct addressing**
    \[
    s1 \leftarrow M[32]
    \]

  - **Indirect addressing**
    \[
    s1 \leftarrow M[s3]
    \]

  Which helps implement pointers.
• **Immediate addressing**
  
  The operand is a constant.
  
  How can you execute $s1 ≡ 7$?
  
  addi $s1$, $zero$, 7 means $s1 ≡ 0 + 7$
  
  *(add immediate, uses the I-type format)*

• **PC-relative addressing**
  
  The operand address = PC + an offset
  
  Implements position-independent codes. A small offset is adequate for short loops.

• **Pseudo-direct addressing**
  
  Used in the J format. The target address is the concatenation of the 4 MSB’s of the PC with the 28-bit offset. This is a minor variation of the PC-relative addressing format.
Revisiting pseudoinstructions

These are simple assembly language instructions that do not have a direct machine language equivalent. During assembly, the assembler translates each pseudo-instruction into one or more machine language instructions.

Example

\textbf{move} \$t0, \$t1 \quad \# \$t0 \rightarrow \$t1

\begin{center}
\textit{Implemented as} \textbf{add} \$t0, \$zer0, \$t1
\end{center}

How to implement \textbf{li} \$v0, 4 \?

\textbf{blt} \$s0, \$s1, \textit{label} \quad \# \text{if} \$s0 < \$s1 \text{then goto label}

\begin{center}
\textit{Implemented as}
\end{center}

\begin{tabular}{ll}
\textbf{slt} \$t0, \$s0, \$s1 & \# \text{if} \$s0 < \$s1 \text{then} \$t0 = 1 \text{else} \$t0 = 0 \\
\textbf{bne} \$t0, \$zero, \textit{label} & \# \text{if} \$t0 \neq 0 \text{then goto label}
\end{tabular}

Pseudo-instructions give MIPS a richer set of assembly language instructions.
Procedure Call

Main

procedure

Typically uses a stack.

Question. Can we implement procedure calls using jump instructions (like `j xxxx`)?