Pseudo-instructions

These are easy-to-use 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. Pseudo-instructions enrich the instruction set, and make programming easier.

Example

```
move $t0, $t1    # $t0 ← $t1 (pseudo-instruction)
```

The assembler will translate it to
```
add $t0, $zer0, $t1
```

Consider the new instruction
```
slt $s1, $s2, $s3  (set less than) if $s2 < $s3 then set $s1 to 1
```

Now, there is a pseudo-instruction
```
blt $s0, $s1, label
```

The assembler translates this to
```
slt $t0, $s0, $s1    # if $s0 < $s1 then $t0 = 1 else $t0 = 0
bne $t0, $zero, label  # if $t0 ≠ 0 then goto label
```
Loading a 32-bit constant into a register

Quite often, we would like to load a constant value into a register (or a memory location)

\[
\text{lui } \$s0, 42 \quad \# \text{ load upper-half immediate}
\]
\[
\text{ori } \$s0, \$s0, 18 \quad \# \text{(one can also use andi)}
\]

What is the end result?
Compiling a switch statement

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. Let $t2$ contain 4.

    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

What next? Jump to the right case!
jumptable

Base address of the 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

MEMORY

L0

L1

Exit
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 address pointed by t1 into register 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
    J Exit
Exit: <next instruction>
```
The instruction formats for jump and branch

\[
\text{J 10000 is represented as}
\]

\[
\begin{array}{c|c}
2 & 2500 \\
\end{array}
\]

6-bits 26 bits

This is the J-type format of MIPS instructions.

Conditional branch is represented using I-type format:

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

\[
\begin{array}{c|c|c}
5 & 16 & 17 \\
\end{array}
\]

6 5 5 16-bit offset

Current PC + (4 * offset) determines the branch target Label

This is called 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>
</tr>
</thead>
<tbody>
<tr>
<td>80000</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80004</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80008</td>
<td>0</td>
<td>9</td>
<td>22</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>80012</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>80016</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2 (why?)</td>
<td></td>
</tr>
<tr>
<td>80020</td>
<td>0</td>
<td>19</td>
<td>20</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>80024</td>
<td>2</td>
<td></td>
<td>20000 (why?)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80028</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
  
  add $s1, $s2, $s3 means $s1 ← $s2 + $s3

- **Base addressing**
  
  Operand is in memory.
  
  The address is the sum of a register and a constant.
  
  lw $s1, 32($s3) means $s1 ← M[s3 + 32]

  As special cases, you can implement

- **Direct addressing**  
  $s1 ← M[32]

- **Indirect addressing**  
  $s1 ← M[s3]

  Which helps implement pointers
• **Immediate addressing**

The operand is a constant.

How can you execute $s1 \leftarrow 7$?

`addi $s1, $zero, 7` means $s1 \leftarrow 0 + 7$

*(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.