MIPS Instruction formats

R-type format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shift amt</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Used by add, sub etc.

I-type format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

Used by lw (load word), sw (store word) etc

There is one more format: the J-type format. Each MIPS instruction must belong to one of these formats.
The instruction format for jump

\[ J \ 10000 \] is represented as

\begin{center}
\begin{array}{c|c}
2 & 10000 \\
\end{array}
\end{center}

6-bits \hspace{1cm} 26 bits

This is the J-type format of MIPS instructions.

Conditional branch is represented using I-type format:

\[ \text{bne } s0, s1, 1234 \] is represented as

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

\begin{center}
\begin{array}{c|c|c}
6 & 5 & 5 \\
\end{array}
\end{center}

16-bit offset

PC + offset determines the branch target.

This is called **PC-relative addressing**.
Revisiting machine language of MIPS

(check out pp 101-105)

Loop:

```
add $t1, $s3, $s3  # starts from 80000
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></th>
<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>
<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></td>
<td>2</td>
<td></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></td>
<td></td>
<td>20000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>80028</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MIPS 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 \( s1 \leftarrow 7 \)?

  \[
  \text{addi} \ s1, \ $zero, \ 7 \text{ means } s1 \leftarrow 0 + 7
  \]

  \( \text{add immediate, uses the I-type format} \)

• **PC-relative addressing**

  The operand address = PC + an offset

  Implements *position-independent codes*. A small offset is adequate for short loops.
Procedure Call

Uses a stack. What is a stack?
The stack

Occupies a part of the main memory. In MIPS, it grows from high address to low address as you push data on the stack. Consequently, the content of the stack pointer ($sp) decreases.
Use of the stack in procedure call

Before the subroutine executes, save registers.
Jump to the subroutine using jump-and-link (jal address)
(jal address means ra ← PC + 4; PC ← address)

After the subroutine executes, restore the registers.
Return from the subroutine using jr (jump register)
(jr ra means PC ← (ra))

Example
int leaf (int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}

The arguments g, h, i, j are put in $a0-$a3.
The result f is put into $s0, and returned to $v0.
The structure of the procedure

Leaf: 

```
subi $sp, $sp, 12  # $sp = $sp-12, make room
sw $t1, 8($sp)     # save $t1 on stack
sw $t0, 4($sp)     # save $t0 on stack
sw $s0, 0($sp)     # save $s0 on stack
```

Now we can use the registers $t1, $t0, $s0 in the body of the procedure.

```
add $t0, $a1, $a2  # $t0 = g + h
add $t1, $a2, $a3  # $t1 = i + j
sub $s0, $t0, $t1  # $s0 = (g + h) - (i + j)
```

Return the result into the register $v0.

```
add $v0, $s0, $zero  # returns f = (g+h)-(i+j) to $v0
```
Now restore the old values of the registers by popping the stack.

lw $s0, 0($sp)    # restore $s0
lw $t0, 4($sp)    # restore $t0
lw $t1, 8($sp)    # restore $t1
addi $sp, $sp, 12 # adjust $sp

Finally, return to the main program.

jr $ra           # return to caller.
A recursive procedure

Example. Compute factorial (n)

```c
int fact (int n)
{
    if (n < 1) return (1);
    else return (n * fact(n-1))
}
```

(Plan) Put n in $a0. Result should be available in $v0.

```
fact:    sub  $sp, $sp, 8
        sw   $ra, 4($sp)
        sw   $a0, 0($sp)
```
calling program

... 
... 
a0 = n (3)
jal fact (4000)
read fact(n) from v0

procedure fact

push ra
push a0
if n<1 then {v0=1
  Return to ra}
a0=n-1
jal fact (4000)
v0=old a0* fact(n-1)
return to old ra

ra = 1004
a0 = 3
ra = 4024
a0 = 2
ra= 4024
a0 = 1

a0
3

v0
result
Now test if $n < 1$ (i.e. $n = 0$). In that case return 0 to $v0$

```
slti $t0, $a0, 1              # if $n \geq 1$ then goto L1
beq $t0, $zero, L1
addi $v0, $zero, 1           # return 1 to $v0
addi $sp, $sp, 8             # pop 2 items from stack
jr $ra                        # return
L1: subi $a0, $a0, 1          # decrement $n$
jal fact                     # call fact with ($n - 1$)
```

Now, we need to compute $n \times \text{fact}(n-1)$

```
lw $a0, 0($sp)               # restore argument $n$
lw $ra, 4($sp)               # restore return address
addi $sp, $sp, 8             # pop 2 items
mult $v0, $a0, $v0           # return $n \times \text{fact}(n-1)$
jr $ra                       # return to caller
```