Various branch instructions

beq $6, $8, there  (branch if equal)
bn $6, $8, here  (branch if not equal)
j label  {unconditional branch to label}
jr $6  {branch to the address stored in $6}

Which format do these instruction use?

Instructions for comparison

slt $1, $2, $3  (set less than)

If r2 < r3 then r1:=1 else $r1:=0

There is a pseudo-instruction blt $s0, $s1, label The assembler translates this to the following:

slt $t0, $s0, $s1  # if $s0 < $s1 then $t0 =1 else $t0 = 0
bne $t0, $zero, label  # if $t0 \neq 0 then goto label
Compiling a switch statement

switch (k) {
    case 0:  f = i + j; break;  s0: f
    case 1:  f = g + h; break;  s1: g
    case 2:  f = g - h; break;  s2: h
    case 3:  f = i - j; break;  s3: i
}

s4: j  s5: k

Assume, $s0-s5$ contain f, g, h, i, j, k. Let $t2$ contain 4.

{Check if k is within the range 0-3}
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

Exit:

What next? Jump to the right case!
register $t4
Base address of the jumptable

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

f = i + j
J Exit

L1
f = g+h
j Exit

Exit

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 address pointed to
                           # 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
    Exit: <next instruction>
```

The instruction formats for jump and branch

J 10000 is represented as

<table>
<thead>
<tr>
<th>2</th>
<th>2500</th>
</tr>
</thead>
</table>

6-bits 26 bits

This is the **J-type format** of MIPS instructions.

[Actually, the target address is the concatenation of the 4 MSB's of the PC with the 28-bit offset.]

Conditional branch is represented using I-type format:

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

is represented as

<table>
<thead>
<tr>
<th>5</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
</table>

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>
</tbody>
</table>

What does this program do?

Machine language version
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 $s1 ← 7$?

  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.
Procedure Call

Typically procedure call uses a stack. What is a stack?

**Question.** Can’t we use a jump instruction to implement a procedure call?
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 (why?).
Jump to the subroutine using jump-and-link (jal address)
(jal address means ra ← PC+4; PC ← address) For MIPS, (ra=r31)

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

Example of a function call

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 will be put into $s0, and returned to $v0.
The structure of the procedure

Leaf:

```
addi $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
```

The contents of $t1, $t0, $s0 in the main program have been saved and can be restores later. Now we can use these registers in the body of the function.

```
add $t0, $a0, $a1    # $t0 = g + h
add $t1, $a2, $a3   # $t1 = i + j
sub $s0, $t0, $t1    # $s0 = (g + h) – (i + j)
```

Main Procedure

Pass g,h,i,j into $a1-$a3

Return result f into $v0
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.
```
Nested subroutine call

\[ f(x, y) = \sqrt{x.y} \]
Handling recursive procedure calls

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.

{Structure of the fact procedure}

```
fact:    subi $sp, $sp, 8
        sw $ra, 4($sp) {why?}
        sw $a0, 0($sp)
```

OLD

$sp (current top of the stack)

NEW

$sp
The growth of the stack as the recursion unfolds
Now test if \( n < 1 \) (i.e. \( n = 0 \)). In that case return 1 to $v0

\[
\begin{align*}
  &\text{slti } \$t0, \$a0, 1 \quad \# \text{ if } n \geq 1 \text{ then goto } L1 \\
  &\text{beq } \$t0, \$zero, L1 \\
  &\text{addi } \$v0, \$zero, 1 \quad \# \text{ return 1 to } \$v0 \\
  &\text{addi } \$sp, \$sp, 8 \quad \# \text{ pop 2 items from stack} \\
  &\text{jr } \$ra \quad \# \text{ return} \\
\end{align*}
\]

L1:
\[
\begin{align*}
  &\text{addi } \$a0, \$a0, -1 \quad \# \text{ decrement } n \\
  &\text{jal } \text{fact} \quad \# \text{ call fact with } (n - 1)
\end{align*}
\]

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

\[
\begin{align*}
  &\text{lw } \$a0, 0(\$sp) \quad \# \text{ restore argument } n \\
  &\text{lw } \$ra, 4(\$sp) \quad \# \text{ restore return address} \\
  &\text{addi } \$sp, \$sp, 8 \quad \# \text{ pop 2 items} \\
  &\text{mult } \$v0, \$a0, \$v0 \quad \# \text{return } n \times \text{fact}(n-1) \\
  &\text{jr } \$ra \quad \# \text{return to caller}
\end{align*}
\]
Run time environment of a MIPS program

Stack pointer

Low address

Growth of stack

High address

Temporary local variables

Return address

Saved argument registers beyond a0-a3

Stack pointer
A translation hierarchy

HLL program

COMPILER

Assembly language program

ASSEMBLER

Machine language module

LINKER

Library routine

Executable machine language program

LOADER

Memory
What are Assembler directives?

Instructions that are not executed, but they tell the assembler about how to interpret something. Here are some examples:

. text

{Program instructions here}

. data

{Data begins here}

. byte 84, 104, 101

. asciiz “The quick brown fox”

. float f1,. . . , fn

. word w1,. . . . wn

. space n {reserve n bytes of space}
How does an assembler work?

In a two-pass assembler

PASS 1: Symbol table generation
PASS 2: Code generation

Follow the example in the class.