MIPS control flow instructions:
Jumps, Branches, and Loops

Pick up handout and bring back Friday
Today’s lecture

- Control Flow
  - Programmatically updating the program counter (PC)

- Jumps
  - Unconditional control flow
  - How is it implemented?

- Branches
  - Loops
  - If/then/else
  - How implemented?
Sequential lines of code are executed by “incrementing” the Program Counter.

<table>
<thead>
<tr>
<th>Im address</th>
<th>Instruction</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400004</td>
<td>mul</td>
<td>$14, $13, $20</td>
</tr>
<tr>
<td>0x00400008</td>
<td>addi</td>
<td>$14, $14, 4</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>$15, $14, $15</td>
</tr>
<tr>
<td></td>
<td>xor</td>
<td>$12, $15, $8</td>
</tr>
</tbody>
</table>

Where is instruction XOR located?

a) 0x00400007 b) 0x00400008
c) 0x00400010 d) 0x00400016
We use **control** flow in high level languages to implement conditionals and loops.

### Loops

```c
for (int i = 0 ; i < N ; i ++) {
    sum += i;
}
```

### Conditionals

```c
if (x < 0) {
    x = -x;
}
```

How do we implement these in MIPS assembly?
An unconditional jumps always transfer control (like a goto statement in C)

- Use a “label” to tell where in the code to jump to: main:

```c
j target_label
```

- Example:

```
Loop: j Loop
```

- What does this code do?
Jumps are encoded with J-type instructions

- Where do the other 6 bits come from?
  - Last two bits are always 00, because PC is word aligned
  - 4 most significant bits come from existing PC value.

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>26 bits</td>
</tr>
</tbody>
</table>

```
0x00000000 0x00000001 0x00000002 0x00000003 0x00000004 0x00000005 ...
```

32 bits

Data
Example encoding: The infinite loop

```
0x00400024: j 0x00400024
```

```
0x00400024: 0000 0100 0000 0000 0000 0000 0010 0100
```

```
0x00400024: 0000 0100 0000 0000 0000 0000 0010 0100
```

```
0x00400024: 0000 0100 0000 0000 0000 0000 0010 0100
```
Jump instructions can only stay within 1 of 16 regions

- A 26-bit address field lets you jump to any address from 0 to $2^{28}$.
  - your Lab solutions had better be smaller than 256MB
Implement Jump

What should \( \text{wr\_enable} \) be?

a) 0  
b) 1  
c) don’t care
Branches provide conditional control flow

```
beq rs, rt, target_label
```

- **Branch if EQual (BEQ):**
  - If \((R[rs] == R[rt])\), then branch to \(target\_label\)
  - Otherwise execute next instruction (\(PC+4\))

```
bne rs, rt, target_label
```

- **Branch if Not Equal (BNE):**
  - Branch when \((R[rs] != R[rt])\)
Implement the C code in MIPS assembly

```c
int sum = 0;
int i = 0;
do {
    sum += i;
    i ++;
} while (i != 10)
```

`A)` `eq`  
`B)` `ne`

Assembly:
```
add $4, $0, 10 # puts 10 in $4
add $3, $0, 0 # i = 0
do:
    add $2, $2, $3 # sum += i
    add $3, $3, 1 # i++
    b $3, $4, do # i != 10
```
Implement the C code in MIPS assembly

```c
int sum = 0;
for (int i = 0 ; i != x ; i ++) {
    sum += i;
}
```

Assembly:
```
add $2, $0, 0  # sum = 0
add $3, $0, 0  # c = 0
loop:
    b $3, $4, end_loop
add $2, $2, $3 # i += c
add $3, $3, 1  # i++
end_loop:
```
Implement the C code in MIPS assembly

```
if (x == 0) {
    x = 1;
}
```

**Hint:** Sometimes it's easier to invert the original condition.

Change "continue if x < 0" to "skip if x >= 0".

**Assembly:**

```
b     $2, $0, skip-if
add   $2, $0, 1  # x=1
```

A) eq
B) ne
The address in branch is an offset from PC+4 to the target address

- What value should be stored in the address of the beq instruction?

```
<table>
<thead>
<tr>
<th>OP</th>
<th>RS</th>
<th>RT</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>100100</td>
<td>0000001</td>
<td>0000000</td>
<td>01100</td>
</tr>
</tbody>
</table>
```

- a) 1
- b) 2
- c) 3
- d) 4
Architecture Design: Make the common Case fast

- Most branches go to targets less than 32,767 instructions away

- Slowly simulate branches that are farther than 32,767 (i.e., Far) instructions away

```
beq $s0, $s1, Far
...
```

```
bne $s0, $s1, Next
j Far
Next:....
```
Implement Branches (w/o jumps)

What should we set alu_op to for beq and bne?

a) ADD  
b) SUB  
c) AND  
d) OR  
e) NOR
Use Jump Register (JR) to jump beyond 256MB

\[ \text{jr } rs \]

\[ \text{PC} = R[rs] \]

- rs acts as a pointer to a pointer

Which rs could be used correctly in JR?
A) $1  B) $2  C) $3  D) $4  E) Any
Jump register is R-type but only needs 1 register specifier

\[
\text{jr } \$rs
\]

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

Example:

\[
\text{jr } \$3
\]

<p>| 603300 | 00011 | — | — | — | 001000 |</p>
<table>
<thead>
<tr>
<th></th>
<th>A[31:0]</th>
<th>B[31:0]</th>
<th>out[31:0]</th>
<th>overflow</th>
<th>zero</th>
<th>negative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A_addr</th>
<th>B_addr</th>
<th>W_addr</th>
<th>A_data</th>
<th>B_data</th>
<th>W_en</th>
<th>W_data</th>
<th>reset</th>
<th>enable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>opcode[5:0]</th>
<th>funct[5:0]</th>
<th>except</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>PC[31:0]</th>
<th>nextPC[31:0]</th>
<th>PC[31:2]</th>
<th>except</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Implementing Jump Register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iClicker</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ADD</th>
<th>4</th>
<th>32</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Instruction Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>addr[29:0]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Register File</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D[31:0]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Sign Extender</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>in[15:0]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump Register
Control Implemented

Which type of branch is taken when control_type = 10
a) No branch taken
b) Taken branch
c) j
d) jr
Architecture Design: Make the common Case fast

- To use JR we need to set all 32 bits in a register, but we do not have an instruction to do this directly.

- Most of the time, 16-bit constants are enough.

- It’s still possible to load 32-bit constants, but at the cost of multiple instructions and temporary registers.
Use two instructions `ori` and `lui` to construct 32-bit addresses

- **ori** can set the lower 16 bits
  
  \[
  \text{ori } \$12, \$0, 0x\text{beef} \quad \# \$12 = 0x0000\text{beef}
  \]

- **Load Upper Immediate (lui)** can set the upper 16 bits
  - **lui** loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  
  \[
  \text{lui } \$12, 0x\text{dead} \quad \# \$12 = 0x\text{dead}0000
  \]
lui is an I-type instruction

\[
\begin{array}{cccc}
06 & 1111 & - & 01100 \\
\text{op} & \text{rs} & \text{rt} & \text{imm} \\
\end{array}
\]

- \(R[rt] = \{\text{imm, 16'b0}\}\)

\[
lui \ $12, \ 0x\text{dead} \quad \# \ $12 = 0x\text{dead}0000
\]
These two code snippets will store the same value in Register 12.
A) True
B) False
lui Implemented

Value for *alu_src2?*

rd_src?

a) 0
b) 1
c) x
Implement the C code in MIPS assembly

if (x < 0) {
    x = -x;
}

$2 \downarrow \text{ invert condition to } x \geq 0

Assembly:

s/r $1, $2, $0
beq $1, $0, skip-if
nor $2, $2, $2
add $2, $2, 1

Skip-if:

A) eq
B) ne
Set if Less Than (slt) sets a register to a Boolean (1 or 0) based on a comparison.

```
slt rd, rs, rt  # R[rd] = (R[rs]<R[rt]) ? 1 : 0
```

```
| op | rs | rt | rd | shamt | func |
```

```
slti rt, rs, imm  # R[rt] = (R[rs]<imm) ? 1 : 0
```

```
| op | rs | rt | imm |
```
slt and slti

 Implemented

\[ RC_s - RC_t < 0 \]
Full Machine Datapath (so far)