MIPS control flow instructions: Jumps, Branches, and Loops
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

0x00400004  mul  $14, $13, $20
            addi $14, $14, 4
            sub  $15, $14, $15
            xor  $12, $15, $8
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 jump always transfer control (like a goto statement in C)

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

  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.
Example encoding: The infinite loop

0x00400024: j 0x00400024

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0000 0000 0100 0000 0000 0000 0000 0010 0100</td>
</tr>
</tbody>
</table>
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

PC[31] → jump_target[31]
PC[29] → jump_target[29]
PC[28] → jump_target[28]
inst[25] → jump_target[27]
inst[24] → jump_target[26]
inst[23] → jump_target[25]
inst[22] → jump_target[24]
inst[21] → jump_target[23]
inst[20] → jump_target[22]
inst[18] → jump_target[20]
inst[17] → jump_target[19]
inst[16] → jump_target[18]
inst[14] → jump_target[16]
inst[12] → jump_target[14]
inst[0] → jump_target[2]
0 → jump_target[1]
0 → jump_target[0]
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

int sum = 0;
int i = 0;

do {
    sum += i;
    i ++;
} while (i != 10)
Implement the C code in MIPS assembly

```c
int sum = 0;

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

Assembly:
Implement the C code in MIPS assembly

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

**Assembly:**

```
```

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

*Change “continue if x < 0” to “skip if x >= 0”.*
The address in branch is an *offset* from PC+4 to the target address.

```
beq $1, $0, L
add $1, $3, $0
add $2, $3, $3
j Somewhere
L:
    add $2, $3, $3
```
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

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

bne $s0, $s1, Next
j Far
Next: ...
```
Implement Branches (w/o jumps)
Use Jump Register (JR) to jump beyond 256MB

jr rs

PC=R[rs]

- rs acts as a pointer to a pointer
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:**

\[ \text{jr } \$3 \]
Implement Jump Register
Control Implemented
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

  ori $12, $0, 0xbeef  # $12 = 0x0000beef

- 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.

  lui $12, 0xdead  # $12 = 0xdead0000
lui is an I-type instruction

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

\[
\text{lui } \$12, \text{ 0xdead} \quad \# \quad \$12 = \text{ 0xdead0000}
\]
lui Implemented
Implement the C code in MIPS assembly

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

Assembly:
Set if Less Than (slt) sets a register to a Boolean (1 or 0) based on a comparison.

\[
\text{slt \ rd, rs, rt} \quad \# \quad R[rd] = (R[rs] < R[rt]) \ ? \ 1 \ : \ 0
\]

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{func} \\
\hline
\end{array}
\]

\[
\text{slti \ rt, rs, imm} \quad \# \quad R[rt] = (R[rs] < \text{imm}) \ ? \ 1 \ : \ 0
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{op} & \text{rs} & \text{rt} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{imm} \\
\hline
\end{array}
\]
slt and slti Implemented
Full Machine Datapath (so far)