MIPS control flow instructions: Jumps, Branches, and Loops

Pick up handout, bring back on Friday
Quick Policy Reminder

Penalty for involvement in Plagiarism

- All parties involved will receive a 0 on that assignment or exam and their final course grade reduced by one letter grade (e.g., A->B, B->C, etc.). A second offense will result in a failing grade for the class.
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

- 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

**Repetition via Loops**

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

**Selective execution via Conditionals**

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

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

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

  \[ \text{j target\_label} \]

- Example:

  \[ \text{Loop: j Loop} \]

- What does this code do? \(\text{Infinite}\)
Jumps use the J-type encoding

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.

\[ \text{jump } PC[31:28] = PC[31:28] \]
Example encoding: The infinite loop

Loop:   j Loop

PC = 0x00400024:  j 0x00400024

address

op

address

0x00400024 0000 0000 0100 0000 0000 0000 0000 0010 0100

always goes
Jump instructions can only move 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

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>0xFFFFF...F</td>
<td>...</td>
</tr>
<tr>
<td>0x10000000</td>
<td></td>
</tr>
<tr>
<td>0x20000000</td>
<td></td>
</tr>
<tr>
<td>0x30000000</td>
<td></td>
</tr>
<tr>
<td>0xF0000000</td>
<td></td>
</tr>
<tr>
<td>0xFF ...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xF0000000</td>
<td></td>
</tr>
<tr>
<td>0xFF ...</td>
<td></td>
</tr>
</tbody>
</table>
```
Implement Jump

What should wr_enable be?

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

\[
\text{beq } rs, \ rt, \ target\_label
\]

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

\[
\text{bne\_rs, rt, target\_label}
\]

- 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)
```

Assembly:

```
add $10, $0, 10
add $2, $0, $0 # sum:
add $3, $0, 0 # i = 0

do:
    add $2, $2, $3
    add $3, $3, 1 # i++
bne $3, $10, do
add i $7, $0, Ox dead,beef
```

A) eq  
B) ne
Implement the C code in MIPS assembly

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

A) eq
B) ne

Assembly:
```
add $2, $0, $0 # A
add $3, $0, $0 # B
loop: b eq $3, $4, done # C
    add $2, $2, $3 # D
    add $3, $3, 1 # E
    j loop # F
done:
```
Implement the C code in MIPS assembly

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

Assembly:
```
bne $2,$0, skip
add $2,$0, 1 # x = 1
```

**Hint:** Sometimes it’s easier to invert the original condition.
Change “continue if x < 0” to “skip if x >= 0”.

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>000100</td>
<td>000001</td>
<td>000000</td>
<td>0000 0000 0000 0011</td>
</tr>
</tbody>
</table>
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

\[
\text{beq} \quad \text{\$s0, \$s1, Far} \\
... \\
\text{bne} \quad \text{\$s0, \$s1, Next} \\
\text{j} \quad \text{Far} \\
\text{Next: ...}
\]
Implement Branches (w/o jumps)

Which alu_op 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

Example:

```
jr $3
```

```
00 0000 0001 0000 0000 001000
```
Implementing 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 \texttt{lui}, \texttt{ori} sequence to construct 32-bit addresses

- \texttt{ori} can set the lower 16 bits

  \begin{verbatim}
  ori $12, $0, 0xbeef  # $12 = 0x0000beef
  \end{verbatim}

- Load Upper Immediate (\texttt{lui}) can set the upper 16 bits
  \begin{itemize}
  \item \texttt{lui} loads the highest 16 bits of a register with a constant, and clears the lowest 16 bits to 0s.
  \end{itemize}

  \begin{verbatim}
  lui $12, 0xdead  # $12 = 0xdead0000
  \end{verbatim}
lui is an I-type instruction

\[
\begin{array}{cccccccc}
00 & 111 & 00000 & 01100 & 1101 & 1110 & 1010 & 1101 \\
\end{array}
\]

- **op**
- **rs**
- **rt**
- **imm**

- \( R[rt] = \{imm, 16\text{'b}0} \)

```
lui $12, 0xdead   # $12 = 0xdead0000
```
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

```
x = -2       $2
if (x < 0) {
  x = -x;
}
```

Assembly:
```
slt  $3, $2, $0  # if x < 0
beq  $3, $0, skip
sub  $2, $0, $2
```

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

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

\[
\text{slti \ rt, rs, imm \ # \ R[rt] = (R[rs] < \text{imm}) \ ? \ 1 : 0}
\]
slt and slti
Implemented
Full Machine Datapath (so far)