MIPS control flow instructions: Jumps, Branches, and Loops

Sign up for exam 2

Pick up handout

Bring back FRIDAY
#1 tip for exam 2

- Work one state at a time
- Analyze ALL POSSIBLE input combinations for each state before analyzing the next one
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

PC→0x00400004 mul $14, $13, $20
     8 addi $14, $14, 4
     < sub $15, $14, $15
     10 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

**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:
  
  ```
  jump
  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
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[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[16] ——— jump_target[18]
inst[14] ——— jump_target[16]
inst[0] ——— jump_target[2]
0 ——— jump_target[1]
0 ——— jump_target[0]

What should wr_enable be?
a) 0  
b) 1  
c) don’t care
Branches provide conditional control flow

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

- 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:

```
addi $4, $0, 10  # temp = 10+0
addi $2, $0, 0   # sum=0
addi $3, $0, 0   # i = 0
do:
    add $2, $2, $3  # sum += i
    addi $3, $3, 1  # i++
    add $3, $4, do  # while (i != 10)
```

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

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

Assembly:
```
addi $2, $0, 0  # a
addi $3, $0, 0  # b

loop:  
    add $2, $2, $3  # c
    addi $3, $3, 1  # d
    bnez $3, $4, end_loop  # e

end_loop:  # f
```

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

\[
\begin{align*}
\text{if } (x \neq 0) \{ \\
\quad b \ x = 1; \\
\}
\end{align*}
\]

\[
\text{Assembly: } \quad b \ c \quad \text{skip if: } \quad \text{addi } \quad \text{skip if: } \quad \text{skip if: } \quad \text{skip if: } \\
\]

\text{Hint: Sometimes it's easier to invert the original condition.}
\text{Change “continue if } x < 0 \text{” to “skip if } x \geq 0 \text{”}.

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?

- a) 1
- b) 2
- c) 3
- d) 4

Branch Address: 12
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

jr rs

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:

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

| 000000 | 000 11 | X | X | X | 00 1000 |
Implementing Jump Register
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{ori} and \texttt{lui} to construct 32-bit addresses

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

\[
\texttt{ori} \ $12, \ $0, \ 0x\textbf{beef} \quad \# \ $12 = 0x\textbf{0000beef}
\]

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

\[
\texttt{lui} \ $12, \ 0x\textbf{dead} \quad \# \ $12 = 0x\textbf{dead0000}
\]
**lui is an I-type instruction**

<table>
<thead>
<tr>
<th>00 1111</th>
<th>X</th>
<th>01100</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
</tr>
</tbody>
</table>

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

\[
\text{lui} \; \$12, \; 0x\text{dead} \; \# \; \$12 = 0x\text{dead}0000
\]
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

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

Assembly:

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

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>imm</th>
</tr>
</thead>
</table>
slt and slti Implemented
Full Machine Datapath (so far)