MIPS control flow instructions: Jumps, Branches, and Loops
Today’s lecture

- **Control Flow**
  - Programatically updating the program counter (PC)

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

- **Branches**
  - Loops
  - If/then/else
  - How implemented?
Control Flow

- So far, only considered sequences of arithmetic instructions

  mul $14, $13, $20
  addi $14, $14, 4
  sub $15, $14, $15

- These are executed one after another
  - Stored sequentially in memory
  - Program counter is incremented by 4 each cycle.

  a) 0x400010   b) 0x400012   c) 0x40000b   d) 0x40000c
Control Flow in high-level languages

- In high-level languages, we can:
  - Repeat statements with loops
    ```java
    for (int i = 0; i < N; i++) {
      sum += i;
    }
    ```
  - Selectively execute statements with if/then/else
    ```java
    if (x < 0) {
      x = -x;
    }
    ```
- Need ways to control which instruction is executed next.
Unconditional Jumps

- The simplest control flow instruction is jump:
  - Unconditional control flow transfer
    - always taken, much like a goto statement in C

```
j target_label
```

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

- Example:

```
Loop:     j Loop
```

- What does this code do?
Encoding Jumps

- To encode jumps we use the J-type instruction format:

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

- This format provides a very long immediate
  - But, not quite long enough to specify a whole 32-bit PC
  - Where do the other 6 bits come from?
    - Last two bits are always 00, because PC value is always word aligned
    - 4 most significant bits come from existing PC value.
Example encoding

- The infinite loop:

  Loop: j Loop

- After assigning instructions to memory addresses

  0x400024: j 0x400024

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x400024</td>
<td>0000 0000 0100 0000 0000 0000 0010 0100</td>
</tr>
</tbody>
</table>
Limitations

- Top 4 bits coming from current PC means:
  - Memory is cut into 16 regions
  - Can only jump within current region with j instruction.

- 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
Implementing Jumps
Conditional Branches

- For our loops to exit, we need conditional control flow.
  
  \[ \text{beq } rs, rt, \text{target\_label} \]

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

- Also, Branch if Not Equal (BNE):
  
  - Same, but branch when \( (R[rs] != R[rt]) \)
Using beq/bne to implement loops:

- How could we use branches to implement the following?

```c
int sum = 0, i = 0;
do {
    sum += i;
    i++
} while (i != 10)
```
Using `beq/bne` and `j` to implement loops:

- Let’s implement the for version of the loop?

```c
int sum = 0;
for (int i = 0 ; i != x ; i ++) {
    sum += i;
}
```
Using beq/bne to implement if/then:

- How could we use branches to implement the following?

```java
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”.

```java
if (x <= 0) {
    x = 1;
}
```
Encoding Branches

- For branch instructions, the constant field is not an address, but an offset from the current program counter (PC) to the target address.

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

- Since the target L is 3 instructions past the beq, the address field would contain 3. The whole beq instruction would be stored as:

```
SPIM's encoding of branch offsets is off by one, so its code would contain an address of 4. (But it has a compensating error when it executes branches.)
```
Larger branch constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away
  - branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short.
- If you do need to branch further, you can use a jump with a branch. For example, if “Far” is far away, then the effect of:

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

can be simulated with the following actual code.

```
bne $s0, $s1, Next
    j   Far
Next:     ...
```

- The MIPS designers have taken care of the common case first.
Implementing Branches
Jump Register

- j instructions allow you to jump within a 256MB range
  - What if you want to go outside that range

  \[ \text{jr } $3 \]

- Jump Register (JR)
  - Copy the 32-bit contents of a register into the PC.
    \[ \text{PC}=R[rs] \]
  - That value better be word aligned (i.e., divisible by 4)

- We’ll see how this is used later.
Encoding Jump Register

- Jump register only needs 1 register specifier
- Use R-type encoding, because it is cheapest opcode-wise.

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

<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:

\[ \text{jr } \$3 \]
Implementing Jump Register
How do we put a 32-bit value into a register?

- I-type instructions can do a 16-bit immediate:
  - Many instructions are useful for setting the low 16b, e.g.,

  \[
  \text{addi } \$12, \$0, 0\text{beef} \quad \# \$12 = 0\text{0000beef}
  \]

- Would be useful to be able to set the top 16b.

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

  \[
  \text{lui } \$12, 0\text{dead} \quad \# \$12 = 0\text{dead0000}
  \]
Load Upper Immediate

This illustrates the principle of making the common case fast.

- Most of the time, 16-bit constants are enough.
- It’s still possible to load 32-bit constants, but at the cost of two instructions and one temporary register.

- `lui $12, 0x3d`
- `ori $12, $12, 0x900`

- a) `0x0000093d`
- b) `0x0003d900`
- c) `0x003d0900`
- d) `0x0009003d`
- e) `0x0900003d`
LUI is an I-type instruction

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

\[
lui \; $12, \; 0xdead \quad \# \; $12 = 0xdead0000
\]
lui Implemented
Using beq/bne to implement if/then:

- How could we use branches to implement the following?

```java
if (x < 0) {
    x = -x;
}
```
Set if Less Than (slt)

- Set a register to a Boolean (1 or 0) based on a comparison.

\begin{align*}
\text{slt} & \quad \text{rd}, \quad \text{rs}, \quad \text{rt} \quad \# \quad R[rd] = (R[rs]<R[rt]) \quad ? \quad 1 \quad : \quad 0 \\
\text{slti} & \quad \text{rt}, \quad \text{rs}, \quad \text{imm} \quad \# \quad R[rt] = (R[rs]<\text{imm}) \quad ? \quad 1 \quad : \quad 0
\end{align*}

- Use R-type and I-type encodings, respectively
  - No big surprises in encoding.