Lecture 9 (10/13/2008)

- Lab #1 Simulation - Due Mon Oct 13 - TODAY
- Lab #1 Hardware - Due Fri Oct 17
- HW #2 - MIPS programming, due Wed Oct 22
- Midterm - Fri Oct 24

Now we talk about how to control this datapath.
Multicycle control unit

- The control unit is responsible for producing all of the control signals.
- Each instruction requires a sequence of control signals, generated over multiple clock cycles.
  - This implies that we need a state machine.
  - The datapath control signals will be outputs of the state machine.
- Different instructions require different sequences of steps.
  - This implies the instruction word is an input to the state machine.
  - The next state depends upon the exact instruction being executed.
- After we finish executing one instruction, we’ll have to repeat the entire process again to execute the next instruction.

Finite-state machine for the control unit

- Each bubble is a state
  - Holds the control signals for a single cycle
  - Note: All instructions do the same things during the first two cycles
Stage 1: Instruction Fetch

- Stage 1 includes two actions which use two separate functional units: the memory and the ALU.
  - Fetch the instruction from memory and store it in IR.
    \[ IR = \text{Mem}[PC] \]
  - Use the ALU to increment the PC by 4.
    \[ PC = PC + 4 \]

Stage 1: Instruction fetch and PC increment

\[
\begin{align*}
\text{IR} &= \text{Mem}[PC] \\
\text{PC} &= \text{PC} + 4
\end{align*}
\]
Stage 1 control signals

- Instruction fetch: \( \text{IR} = \text{Mem}[\text{PC}] \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>1</td>
<td>Read from memory</td>
</tr>
<tr>
<td>IorD</td>
<td>0</td>
<td>Use PC as the memory read address</td>
</tr>
<tr>
<td>IRWrite</td>
<td>1</td>
<td>Save memory contents to instruction register</td>
</tr>
</tbody>
</table>

- Increment the PC: \( \text{PC} = \text{PC} + 4 \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>0</td>
<td>Use PC as the first ALU operand</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>01</td>
<td>Use constant 4 as the second ALU operand</td>
</tr>
<tr>
<td>ALUOp</td>
<td>ADD</td>
<td>Perform addition</td>
</tr>
<tr>
<td>PCWrite</td>
<td>1</td>
<td>Change PC</td>
</tr>
<tr>
<td>PCSource</td>
<td>0</td>
<td>Update PC from the ALU output</td>
</tr>
</tbody>
</table>

- We'll assume that all control signals not listed are implicitly set to 0.

Stage 2: Read registers

- Stage 2 is much simpler.
  - Read the contents of source registers \( \text{rs} \) and \( \text{rt} \), and store them in the intermediate registers \( A \) and \( B \). (Remember the \( \text{rs} \) and \( \text{rt} \) fields come from the instruction register \( \text{IR} \).)

\[
A = \text{Reg}[\text{IR}[25-21]] \\
B = \text{Reg}[\text{IR}[20-16]]
\]
Stage 2: Register File Read

Stage 2 control signals

- No control signals need to be set for the register reading operations $A = \text{Reg}[\text{IR}[25-21]]$ and $B = \text{Reg}[\text{IR}[20-16]]$.
  - \text{IR}[25-21] and \text{IR}[20-16] are already applied to the register file.
  - Registers $A$ and $B$ are already written on every clock cycle.
Executing Arithmetic Instructions: Stages 3 & 4

- We'll start with R-type instructions like `add $t1, $t1, $t2`.
- **Stage 3** for an arithmetic instruction is simply ALU computation.
  \[ \text{ALUOut} = A \text{ op } B \]
  - A and B are the intermediate registers holding the source operands.
  - The ALU operation is determined by the instruction’s “func” field and could be one of add, sub, and, or, slt.

- **Stage 4**, the final R-type stage, is to store the ALU result generated in the previous cycle into the destination register rd.
  \[ \text{Reg}[\text{IR}[15-11]] = \text{ALUOut} \]
### Stage 4 (R-type): write back

...and store it to register “rd”

Take the ALU result from the last cycle...

### Stages 3-4 (R-type) control signals

- **Stage 3** (execution): \( \text{ALUOut} = \text{A op B} \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>1</td>
<td>Use A as the first ALU operand</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>00</td>
<td>Use B as the second ALU operand</td>
</tr>
<tr>
<td>ALUOp</td>
<td>func</td>
<td>Do the operation specified in the “func” field</td>
</tr>
</tbody>
</table>

- **Stage 4** (writeback): \( \text{Reg[IR[15-11]]} = \text{ALUOut} \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>Write to the register file</td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>Use field rd as the destination register</td>
</tr>
<tr>
<td>MemToReg</td>
<td>0</td>
<td>ALUOut contains the data to write</td>
</tr>
</tbody>
</table>
Executing a beq instruction

- We can execute a branch instruction in three stages or clock cycles.
  - But it requires a little cleverness...

  - **Stage 1** involves instruction fetch and PC increment.
    \[
    IR = \text{Mem}[PC] \\
    PC = PC + 4
    \]

  - **Stage 2** is register fetch and branch target computation.
    \[
    A = \text{Reg}[\text{IR}[25-21]] \\
    B = \text{Reg}[\text{IR}[20-16]]
    \]

  - **Stage 3** is the final cycle needed for executing a branch instruction.
    - Assuming we have the branch target available
      \[
      \text{if } (A == B) \text{ then} \\
      PC = \text{branch\_target}
      \]

When should we compute the branch target?

- We need the ALU to do the computation.
  - When is the ALU not busy?

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Optimistic execution

- But, we don’t know whether or not the branch is taken in cycle 2!!
- That’s okay…. we can still go ahead and compute the branch target first. The book calls this optimistic execution.
  - The ALU is otherwise free during this clock cycle.
  - Nothing is harmed by doing the computation early. If the branch is not taken, we can just ignore the ALU result.
- This idea is also used in more advanced CPU design techniques.
  - Modern CPUs perform branch prediction, which we’ll discuss in a few weeks in the context of pipelining.

Stage 2 Revisited: Compute the branch target

- To Stage 2, we’ll add the computation of the branch target.
  - Compute the branch target address by adding the new PC (the original PC + 4) to the sign-extended, shifted constant from IR.

\[
ALUOut = PC + (\text{sign-extend}(IR[15-0]) \ll 2)
\]

We save the target address in ALUOut for now, since we don’t know yet if the branch should be taken.

- What about R-type instructions that always go to PC+4?
Stage 2: Register fetch & branch target computation

Branch completion

- **Stage 3** is the final cycle needed for executing a branch instruction.

  \[
  \text{if (} A \text{ == } B \text{) then} \\
  \text{PC} = \text{ALUOut}
  \]

- Remember that A and B are compared by subtracting and testing for a result of 0, so we must use the ALU again in this stage.
Stage 3 (beq): Branch completion

- **Comparison:** if \((A = B)\) ...

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>1</td>
<td>Use (A) as the first ALU operand</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>00</td>
<td>Use (B) as the second ALU operand</td>
</tr>
<tr>
<td>ALUOp</td>
<td>SUB</td>
<td>Subtract, so Zero will be set if (A = B)</td>
</tr>
</tbody>
</table>

- **Branch:** …then \(PC = ALUOut\)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCWrite</td>
<td>Zero</td>
<td>Change PC only if Zero is true (i.e., (A = B))</td>
</tr>
<tr>
<td>PCSource</td>
<td>1</td>
<td>Update PC from the (ALUOut) register</td>
</tr>
</tbody>
</table>

- **ALUOut** contains the ALU result from the *previous* cycle, which would be the branch target. We can write that to the PC, even though the ALU is doing something different (comparing \(A\) and \(B\)) during the *current* cycle.
Executing a sw instruction

- A store instruction, like `sw $a0, 16($sp)`, also shares the same first two stages as the other instructions.
  - **Stage 1**: instruction fetch and PC increment.
  - **Stage 2**: register fetch and branch target computation.

- **Stage 3** computes the effective memory address using the ALU.

  \[ \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) \]

  A contains the base register (like $sp), and \text{IR}[15-0] is the 16-bit constant offset from the instruction word, which is not shifted.

- **Stage 4** saves the register contents (here, $a0) into memory.

  \[ \text{Mem[ALUOut]} = B \]

  Remember that the second source register \text{rt} was already read in Stage 2 (and again in Stage 3), and its contents are in intermediate register B.

---

**Stage 3 (sw): effective address computation**

Compute an effective address and store it in ALUOut.
**Stage 4 (sw): memory write**

...into memory.

Use the effective address from stage 3...

...to store data from one of the registers...

**Stages 3-4 (sw) control signals**

- **Stage 3** (address computation): \( \text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0]) \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSrcA</td>
<td>1</td>
<td>Use A as the first ALU operand</td>
</tr>
<tr>
<td>ALUSrcB</td>
<td>10</td>
<td>Use sign-extend((\text{IR}[15-0])) as the second operand</td>
</tr>
<tr>
<td>ALUOp</td>
<td>010</td>
<td>Add and store the resulting address in (\text{ALUOut})</td>
</tr>
</tbody>
</table>

- **Stage 4** (memory write): \( \text{Mem}[\text{ALUOut}] = B \)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemWrite</td>
<td>1</td>
<td>Write to the memory</td>
</tr>
<tr>
<td>IorD</td>
<td>1</td>
<td>Use (\text{ALUOut}) as the memory address</td>
</tr>
</tbody>
</table>

The memory’s “Write data” input always comes from the B intermediate register, so no selection is needed.
Executing a lw instruction

- Finally, lw is the most complex instruction, requiring five stages.
  - The first two are like all the other instructions.
    - **Stage 1**: instruction fetch and PC increment.
    - **Stage 2**: register fetch and branch target computation.
  - The third stage is the same as for sw, since we have to compute an effective memory address in both cases.
    - **Stage 3**: compute the effective memory address.

Stages 4-5 (lw): memory read and register write

- **Stage 4** is to read from the effective memory address, and to store the value in the intermediate register MDR (memory data register).
  \[
  \text{MDR} = \text{Mem[ALUOut]}
  \]

- **Stage 5** stores the contents of MDR into the destination register.
  \[
  \text{Reg}[\text{IR}[20-16]] = \text{MDR}
  \]

Remember that the destination register for lw is field rt (bits 20-16) and *not* field rd (bits 15-11).
Stage 4 (lw): memory read

...to read data from memory...

Use the effective address from stage 3...

...into MDR.

Stage 5 (lw): register write

...and store it in register rt.

Take MDR...
Stages 4-5 (lw) control signals

- **Stage 4 (memory read):** $MDR = Mem[ALUOut]$

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemRead</td>
<td>1</td>
<td>Read from memory</td>
</tr>
<tr>
<td>IorD</td>
<td>1</td>
<td>Use ALUOut as the memory address</td>
</tr>
</tbody>
</table>

  The memory contents will be automatically written to $MDR$.

- **Stage 5 (writeback):** $Reg[IR[20-16]] = MDR$

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>Store new data in the register file</td>
</tr>
<tr>
<td>RegDst</td>
<td>0</td>
<td>Use field rt as the destination register</td>
</tr>
<tr>
<td>MemToReg</td>
<td>1</td>
<td>Write data from $MDR$ (from memory)</td>
</tr>
</tbody>
</table>

Finite-state machine for the control unit
Implementing the FSM

- This can be translated into a state table; here are the first two states.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Input (Op)</th>
<th>Next State</th>
<th>Output (Control signals)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg Fetch</td>
<td>X</td>
<td>Reg Fetch</td>
<td>PC Write</td>
</tr>
<tr>
<td>Reg Fetch</td>
<td>BNE Branch</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>Reg Fetch</td>
<td>R-type</td>
<td>R-type</td>
<td>X</td>
</tr>
<tr>
<td>Reg Fetch</td>
<td>LW/S W</td>
<td>Compute</td>
<td>0</td>
</tr>
</tbody>
</table>

- You can implement this the hard way.
  - Represent the current state using flip-flops or a register.
  - Find equations for the next state and (control signal) outputs in terms of the current state and input (instruction word).

- Or you can use the easy way.
  - Stick the whole state table into a memory, like a ROM.
  - This would be much easier, since you don’t have to derive equations.

Summary

- Now you know how to build a multicycle controller!
  - Each instruction takes several cycles to execute.
  - Different instructions require different control signals and a different number of cycles.
  - We have to provide the control signals in the right sequence.