Notes and Updates

- New HW
- Chapter 5 - Multicycle Procs
- Recommended, print out slides, drawing
- Office hours: Tues 3:30-5, Wed 12-1:30
- QUESTIONS!

It is time NOW to prepare for the midterm

Feb 3 (a week from Thurs)

What will you do to study?
(remember the homework has been easy!)

WED Discussion section: QUESTIONS
THURS class: QUESTIONS
Summary of execution steps

<table>
<thead>
<tr>
<th>Step</th>
<th>R-type</th>
<th>Memory</th>
<th>Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>IR = Mem[PC]</td>
<td>PC = PC + 4</td>
<td></td>
</tr>
<tr>
<td>Instruction Decode/</td>
<td>A = Reg[IR[25-21]]</td>
<td>B = Reg[IR[20-16]]</td>
<td></td>
</tr>
<tr>
<td>register fetch</td>
<td>ALUout = PC + (sign-extend(IR[15-0]) &lt;&lt; 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution, address</td>
<td>ALUout = A op B</td>
<td>ALUout = A + sign-extend(IR[15-0])</td>
<td></td>
</tr>
<tr>
<td>computation, branch</td>
<td></td>
<td>if (A==B) then PC=ALUout</td>
<td></td>
</tr>
<tr>
<td>completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory access or R-type</td>
<td>Reg[IR[15-11]] = ALUout</td>
<td>memory-data = Mem[ALUout] or Mem[ALUout] = B</td>
<td></td>
</tr>
<tr>
<td>type completion</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-back</td>
<td>Reg[IR[20-16]] = memory-data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is Register Transfer Language (RTL)
"High level" description of changes to state elements
We'll go through these in exacting detail
And translate them to "low level" control signal settings
Modern design tools do this automatically

Control Lines (page 324):

- 1-bit
  - RegDst, RegWrite, MemRead, MemWrite
  - ALUSrcA: top ALU operand from reg or PC?
  - MemtoReg: data to write register from ALUout or MDR(memory) ?
  - IorD: Runs to Memory to indicate reading/writing Instruction or Data
  - IRWrite: Buffer register for Instr should be written
  - PCWrite: PC should written absolutely
  - PCWriteCond: PC should be written, maybe
Control Lines (page 324):

- **2-bit**
  - ALU Op: 00 add, 01 sub, 10 funct based
  - ALUSrcB:
    | What selected      | What type of op uses? |
    |--------------------|-----------------------|
    | 00 B register      | R-type                |
    | 01 4               | All (PC+4)            |
    | 10 Sign ext immed  | Lw/sw/ I-types        |
    | 10 Sign ext immed * 4 | branch               |
  - PCSource

Cycle 1: Instruction Fetch

Datapath RTL:
Control: IorD MemRead MemWr IRwrite ALUsrcA ALUsrcB
Control for IF cycle

- We'll design a Finite State Machine to detail our control logic.
  - Basically a sequential logic function OR
  - A program

  MemRead
  ALUsrcA = 0
  IorD = 0
  IRwrite
  ALUsrcB = 01
  ALUop = 00

Detour: Finite State Machines (Programs?)

- Set of inputs and outputs
- Next-state function that maps current state and (possibly) inputs to a next state and (possibly) outputs

<table>
<thead>
<tr>
<th></th>
<th>NEITHER</th>
<th>FRONT</th>
<th>REAR</th>
<th>BOTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>C</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>CLOSED</td>
<td>C</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Problem: Automatic door sensor
Outline of FSM for Multicycle CPU

Cycle 1

- MemRead
  - ALUsrcA = 0
  - IorD = 0
- IRwrite
- ALUsrcB = 01
- ALUop = 00
- PCwrite
- PCsource = 00

Cycle 2
- Any inst

Any inst

Cycle 3

Cycle 2: Instruction Decode (and register fetch)

\[
\begin{align*}
A &= \text{Reg}[\text{IR}[25-21]] \\
B &= \text{Reg}[\text{IR}[20-16]] \\
\text{ALUout} &= \text{PC} + (\text{sign-extend} (\text{IR}[15-0]) << 2)
\end{align*}
\]

We compute target address even though we don’t know if it will be used
  - Operation may not be branch
  - Even if it is, branch may not be taken

Why NOW?

Why for ALL?
A = Register[IR[25-21]]
B = Register[IR[20-16]]
ALUout = PC + (sign-extend (IR[15-0]) << 2)

Control for first two cycles

After cycle 2, we can treat different instructions separately
Cycle 3 for beq: Execute

- In cycle 1, PC was incremented by 4
- In cycle 2, ALUout was set to branch target
- This cycle, we conditionally reset PC: \( \text{if} \ (A==B) \ \text{PC=ALUout} \)

FSM state for cycle 3 of beq
Break! KQS Card report!

- Few respondents - THANKS to those who did
  - You can still turn them in anytime!
- Pace of class OK (3)
- You like the breaking things up
- You like doing exercises
- One person would prefer that I write on the board
  - I stink at that.

R-type instruction review

- Cycle 1 (Fetch)
  \[ PC = PC + 4 \]
  \[ IR = MEM[PC] \]
- Cycle 2 (Decode)
  \[ ALUout = PC + (\text{sign-extend } (IR[15-0]) \ll 2) \]
  \[ A = \text{Reg}[IR[25-21]] \]
  \[ B = \text{Reg}[IR[20-16]] \]
- Cycle 3 (Execute)
  \[ ALUout = A \text{ op } B \]
- Cycle 4 (WriteBack)
  \[ \text{Reg}[IR[15-11]] = ALUout \]

Then R-type instruction is finished
**R-type EXecution Then WriteBack**

*Cycle 3: ALUout = A op B*

*Cycle 4: Reg[IR[15-11]] = ALUout*

**FSM states for R-type Instructions**

- From state 1
  - ALUsrcA = 1
  - ALUsrcB = 00
  - ALUop = 10

- Execution

- WriteBack

- To state 0
Our Control FSM so far...

Quick Prediction:

- How many states will a load / store instruction use?
  - What are the names of those states and what do they “do”? (HINT: Use RTL to most easily define what is “done” on a datapath diagram)
Load and Store

- EXecute (cycle 3): compute address
  \[ ALU_{\text{out}} = A + \text{sign-extend}(IR[15-0]) \]
- Mem (cycle 4): access memory

- WB (cycle 5, only for ): write register
  \[ \text{Reg}[IR[20-16]] = MDR \]
Cycle 4 for Store: Memory Access

Memory[ALUout] = B

Cycle 4 for Load: Memory Access

Memory Data Register = Memory[ALUout]
Cycle 5 for load: WriteBack

Reg[IR[20-16]] = memory-data

Memory Instruction States

from state 1

MemRead IorD = 1
Memory Access

MemWrite IorD = 1

Address Computation

write-back

5/5!!!!
Cycle 3 for Jump

Cycle 3 JMP FSM state

from state 1

PCWrite
PCSource=10
RECAP: Control of Multicycle Implementation

- We've been building up a Finite State Machine (FSM)
- FSM shows a succession of states, transitions between states (based on inputs), and outputs for each state.
- First two states are the same for every instruction, later states depend on opcode

The Complete FSM

page 339
The Problem with FSMs as control sequencers

- They get unmanageable quickly as they grow.
  - hard to specify
  - hard to manipulate
    - Hundreds of states
    - thousands of arcs
  - error prone
  - hard to visualize

- Just like PROGRAMS
Microprogramming

- Specifying the “behavior” of the CPU via a “program” of microinstructions
  - Recreate FSM with “program” of microinsts
- A microinstruction must express 2 things:
  - Set of control signals to be asserted (from a single FSM state)
  - Sequencing information indicating what “state” or microinstruction must be executed next
- == “designing the control as a program that implements the machine instructions in terms of simpler microinstructions”

Why Microprogramming?

- If a microprogram is fundamentally the same as the FSM, what’s the big deal?
  - Easier to specify (program), visualize, and manipulate.
  - allows us to think about the control symbolically
Implementing a control FSM with a microprogram

Each line in the ROM is now a microprogram instruction, corresponding to a FSM state, with an operation (control signals) and branch destination (next state info).

Microprogram Implementation
Microinstructions versus MIPS instructions

- **Multiple "formats"**
  - Each with some type of fields
  - Each performs a basic “component” of computation

- **One format**
  - Each inst has 7 fields
  - Each performs a basic STEP (think RTL-sized step) of control for our multicycle datapath diagram

Microinstruction Fields

- **Different format than regular insts:**
  - Fields: ALU Control, SRC1, SRC2, Register Control, Memory, PCWrite control, Sequencing

- **Fields are combination of control signals SUCH THAT**
  - Signals can "share field" if never asserted simultaneously
  - Memory: ReadPC, ReadALU, WriteALU
  - Combines: IorD, MemRead, MemWrite, IRWrite
Real MIPS: Microinstruction fields

<table>
<thead>
<tr>
<th>Field name</th>
<th>Value</th>
<th>Signals active</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU control</td>
<td>Add</td>
<td>ALUOp = 00</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td></td>
<td>Sub</td>
<td>ALUOp = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td></td>
<td>Func</td>
<td>ALUOp = 10</td>
<td>Use the instruction's function code to determine ALU control.</td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>ALSRCRA = 0</td>
<td>Use the PC as the first ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUSRCA = 1</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>ALUSRIB = 0</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Extend</td>
<td>ALUSRIB = 0</td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUSRIB = 1</td>
<td>Use output of the shift-by-two unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td></td>
<td>Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rs field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write a register using the rt field of the IR as the register number and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td></td>
<td>Write MDR</td>
<td>RegWrite, RegDst = 0, MemtoReg = 1</td>
<td>Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.</td>
</tr>
<tr>
<td></td>
<td>Read PC</td>
<td>MemRead, ioD = 0</td>
<td>Read memory using the PC as address; write result into IR (and MDR).</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>MemWrite, ioD = 0</td>
<td>Write memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write memory using the ALUOut as address; contents of B as the data.</td>
</tr>
<tr>
<td></td>
<td>PC write control</td>
<td>PCCSource = 00, PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALUWriteCond = 01</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jumpladdress = PCCSource = 01, PCWriteCond = 10, PCW</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>AddCC = 11</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td></td>
<td>Patch</td>
<td>AddCC = 05</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 1</td>
<td>AddCC = 01</td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>AddCC = 10</td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>

So a sample microinst might be

<table>
<thead>
<tr>
<th>ALU Control</th>
<th>Src1</th>
<th>Src2</th>
<th>Register control</th>
<th>Memory</th>
<th>PCWrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
</tbody>
</table>

- So what RTL is this?
- What “stage” does it belong to?
Microinstruction “Branching”

- Controlled by “Sequence” field
  - Sequence: Go to next (+1) instruction (next state)
  - Fetch: Go to “next real instruction” -- goto Fetch state (some instruction == probably inst 0)
  - Dispatch: Deals with complex multi-arc states
    - Uses separate “dispatch tables” to “look up” net state/micro inst to go to
    - Need one dispatch table for each “state” that has multiple arcs leaving it

A Microprogram: Can you fill in the labels?

- Add labels indicating start of “real” instructions and their type (lw, etc)

<table>
<thead>
<tr>
<th>Label</th>
<th>ALU Control</th>
<th>Src1</th>
<th>Src2</th>
<th>Memory</th>
<th>PCWrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Add</td>
<td>PC</td>
<td>1</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
</tr>
<tr>
<td>Add PC</td>
<td>PC</td>
<td>Fetch</td>
<td>Read</td>
<td>Dispatch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MemEx: Add</td>
<td>A</td>
<td>Ext</td>
<td>ReadALU</td>
<td>Dispatch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WriteMDR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WriteALU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Func code : Add</td>
<td>B</td>
<td>WriteALU</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Func code : Sub</td>
<td>A</td>
<td>b</td>
<td>ALUcond</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Jump addr</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
You should be able to take each of these “stages” and be able to write a microinst for it