Notes and Updates

- HW: Second problem from 5.7 off of CD
  - We’ll cover it today in class
  - Exceptions not on midterm
- Whatever we cover today will be on midterm
- Recommended, print out slides, drawing
- Office hours: Tues 3:30-5, Wed 12-1:30
  - Extra office hours Tues until 6pm!
- QUESTIONS!

It is time NOW to prepare for the midterm

Feb 3 (a week)

(remember the homework has been easy!)

Webboard: Saturday study session proposed!
(Thanks Igor!)
Clics 12-3pm

Some review today
More on Tues: Send questions by midnight Monday!
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

Show active lines for each cycle of each type of instruction

**Beq Cycle 3:** if \((A == B)\) \(PC = ALUout\)
Our Control FSM so far…

Instruction Fetch, state 0

- MemRead
- ALUsrcA = 0
- IorD = 0
- IRWrite
- ALUsrcB = 01
- ALUOp = 00
- PCWrite
- PCSource = 00

Instruction Decode, state 1

- ALUsrcA = 0
- ALU src B = 11
- ALU Op = 00

- Opcode = LW or SW
- Opcode = R-Type
- Opcode = BEQ
- Opcode = JMP

This picture
Not finished yet

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
  \[ \text{ALUout} = A + \text{sign-extend}(\text{IR}[15-0]) \]
- Mem (cycle 4): access memory
  \[ \text{Reg}[\text{IR}[20-16]] = \text{MDR} \]

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

load

store

write-back

Address Computation

Memory Access

5/5!!!!
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

RegDst = 0
MemtoReg = 1
BREAK: What is the most confusing thing about Multicycle design?

Simplifying Control Design Through Microprogramming: Chapter 2.7 on CD
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
Microprogramming

- 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
**Microprogram Implementation**

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

---

**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><strong>ALU control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALUOp</td>
<td>Add</td>
<td>ALUDp = 00</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td></td>
<td>Sub</td>
<td>ALUDp = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUOp = 10</td>
<td>Use the instruction’s function code to determine ALU control.</td>
<td></td>
</tr>
<tr>
<td>Src1</td>
<td>ALUSrcA = 0</td>
<td>Use the PC as the second ALU input.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>ALUSrcA = 1</td>
<td>Register A is the first ALU input.</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>ALUSrcB = 0</td>
<td>Register B is the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Ext</td>
<td>ALUSrcB = 01</td>
<td>Use 4 as the second ALU input.</td>
</tr>
<tr>
<td><strong>Register control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read ALU</td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rt field of the IR as the register number and putting the data into registers A and B.</td>
<td></td>
</tr>
<tr>
<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>
<td></td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Head PC</td>
<td>MemHead, iorD = 0</td>
<td>Head memory using the PC as address; write result into IR (and the MDR).</td>
<td></td>
</tr>
<tr>
<td>Head ALU</td>
<td>MemHead, iorD = 1</td>
<td>Head memory using the ALUOut as address; write result into MDR.</td>
<td></td>
</tr>
<tr>
<td>Write ALU</td>
<td>MemWrite, iorD = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
<td></td>
</tr>
<tr>
<td><strong>PC write control</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>PCsource = 00, PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
<td></td>
</tr>
<tr>
<td>ALUOut-cond</td>
<td>PCsource = 01, PCWriteCond</td>
<td>If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.</td>
<td></td>
</tr>
<tr>
<td>Jump address</td>
<td>PCsource = 010, PCWrite</td>
<td>Write the PC with the jump address from the instruction.</td>
<td></td>
</tr>
<tr>
<td><strong>Sequencing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seq</td>
<td>AddCtrl = 11</td>
<td>Choose the next microinstruction sequentially.</td>
<td></td>
</tr>
<tr>
<td>Fetch</td>
<td>AddCtrl = 00</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
<td></td>
</tr>
<tr>
<td>Dispatch 1</td>
<td>AddCtrl = 01</td>
<td>Dispatch using the ROM 1.</td>
<td></td>
</tr>
<tr>
<td>Dispatch 2</td>
<td>AddCtrl = 10</td>
<td>Dispatch using the ROM 2.</td>
<td></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>WCWrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</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>Register control</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>4</td>
<td>Read PC</td>
<td>Read PC</td>
<td>ALU Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td>MemInt:</td>
<td>Add A Ext</td>
<td>PC Ext</td>
<td>Read</td>
<td>Read ALU</td>
<td>ReadALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WriteALU</td>
<td>WriteALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALUoutcond</td>
<td>ALUoutcond</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

You should be able to take each of these "stages" and be able to write a microinst for it.
Quiz Review

- Why do we have 2 memories in single cycle design?
  - And why can we get away with only one in multicycle?
- Why do we have 3 adders/ALUs in single cycle design
  - And why can we get away with only one in multicycle?

Single Cycle CPU: bne, jal, jr?
**Single Cycle CPU:**
*Iw and sw can't have an offset*

---

**Performance Comparisons:**
*SINGLE vs Multicycle*

- For a program P that executes: 20% loads, 10% stores, 50% Rtypes, and 20% branches on the processors designed in class
- Assuming a 8ns cycle time for a single cycle processor, and a 2ns cycle time for the multicycle processor, which design will be faster? Give the times speedup over the slower machine.

- If we could redesign our processor so that an RType could be done in 3 cycles (multicycle) and only in 6ns (single cycle) how would your answer change?
Exceptions

• Prediction time:

What can possibly go wrong in executing an instruction?  What would you want to do about it?

Exceptions

• There are two sources of non-sequential control flow in a processor
  - explicit branch and jump instructions
  - exceptions

• Branches are synchronous and deterministic
• Exceptions are typically asynchronous and non-deterministic
  - Sometimes also called interrupts
• Guess which is more difficult to handle?
Exceptions and Interrupts

- the terminology is not consistent, but we'll refer to
  - exceptions as any unexpected change in control flow
  - interrupts as any externally-caused exception

- So then, what is:

<table>
<thead>
<tr>
<th>Exc</th>
<th>Int</th>
</tr>
</thead>
<tbody>
<tr>
<td>arithmetic overflow</td>
<td></td>
</tr>
<tr>
<td>divide by zero</td>
<td></td>
</tr>
<tr>
<td>I/O device signals completion to CPU</td>
<td></td>
</tr>
<tr>
<td>user program invokes the OS</td>
<td></td>
</tr>
<tr>
<td>memory parity error</td>
<td></td>
</tr>
<tr>
<td>illegal instruction</td>
<td></td>
</tr>
<tr>
<td>timer signal</td>
<td></td>
</tr>
</tbody>
</table>

For now...

- The machine we’ve been designing in class can generate two types of exceptions.
  - arithmetic overflow
  - illegal instruction

- On an exception, we need to

  - transfer control to OS
    - Let OS do something
      - Pop a window, close app, reboot, wipe hard drive...
Handling exceptions

- PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
- 2 ways of handling
  - A status register, and a single exception handler may be used to record the exception and transfer control, or
  - A vectored interrupt transfers control to a different location for each possible type of interrupt/exception

Supporting exceptions

- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user's PC
  - Cause: A register to record the cause of the exception
    - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.
Supporting exceptions in our DataPath