Notes and Updates

- Exceptions (finishing 5), then review
  - Starting with microinsts
- Office hours: Tues 3:30-5, Wed 12-1:30
  - Extra office hours today until 6pm!
- NON-MIDTERM QUESTIONS!

Microprogramming

- Specifying the "behavior" of the CPU via a "program" of microinstructions
  - Recreate FSM with "program" of microinsts

| Java/C | Assembly/ISA | Microinstruction |
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

<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>ALUDop = 00</td>
<td>Cause the ALU to add.</td>
</tr>
<tr>
<td></td>
<td>Sub</td>
<td>ALUDop = 01</td>
<td>Cause the ALU to subtract; this implements the compare for branches.</td>
</tr>
<tr>
<td>Func code</td>
<td>ALUCodes = 10</td>
<td>Use the instruction's function code to determine ALU control.</td>
<td></td>
</tr>
<tr>
<td>Src1</td>
<td>PC</td>
<td>Use the PC as the first 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 = 11</td>
<td>Use output of the shift unit as the second ALU input.</td>
</tr>
<tr>
<td></td>
<td>Ext</td>
<td>ALUSrcB = 10</td>
<td>Use output of the sign extension unit as the second ALU input.</td>
</tr>
<tr>
<td>Register control</td>
<td>Head ALU</td>
<td>RegWrite, RegDst = 1, MemtoReg = 0</td>
<td>Write a register using the rd field of the IR as the register number and putting the data into registers A and B.</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 and the contents of the ALUOut as the data.</td>
</tr>
<tr>
<td>Memory</td>
<td>Head PC</td>
<td>MemHead, lorD = 0</td>
<td>Head memory using the PC as address; write result into IR (and the MDR).</td>
</tr>
<tr>
<td></td>
<td>Head ALU</td>
<td>MemHead, lorD = 1</td>
<td>Head memory using the ALUOut as address; write result into MDR.</td>
</tr>
<tr>
<td></td>
<td>Write ALU</td>
<td>MemWrite, lorD = 1</td>
<td>Write memory using the ALUOut as address, contents of B as the data.</td>
</tr>
<tr>
<td>PC write control</td>
<td>ALU</td>
<td>PCSource = 00, PCWrite</td>
<td>Write the output of the ALU into the PC.</td>
</tr>
<tr>
<td></td>
<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>
</tr>
<tr>
<td></td>
<td>Jump addr</td>
<td>PCSource = 10, PCWrite</td>
<td>Write the PC with the jump address from the instruction.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Seq</td>
<td>AddCtl = 11</td>
<td>Choose the next microinstruction sequentially.</td>
</tr>
<tr>
<td></td>
<td>Fetch</td>
<td>AddCtl = 00</td>
<td>Go to the first microinstruction to begin a new instruction.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 1</td>
<td>AddCtl = 01</td>
<td>Dispatch using the ROM 1.</td>
</tr>
<tr>
<td></td>
<td>Dispatch 2</td>
<td>AddCtl = 00</td>
<td>Dispatch using the ROM 2.</td>
</tr>
</tbody>
</table>
So a sample microinst might be

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

<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>Add</td>
<td>PC</td>
<td>4</td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
</tbody>
</table>

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>FCWrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch:</td>
<td>Add</td>
<td>PC 4</td>
<td></td>
<td>Read PC</td>
<td>ALU</td>
<td>Seq</td>
<td></td>
</tr>
<tr>
<td>MemIns:</td>
<td>Add</td>
<td>PC 1</td>
<td></td>
<td>Read</td>
<td>ALU</td>
<td>Seq</td>
<td>Dispatch</td>
</tr>
<tr>
<td>MemOut:</td>
<td>Add</td>
<td>A</td>
<td></td>
<td>WriteMDR</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ReadALU</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WriteALU</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ALUOutCond</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jump addr</td>
<td>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
</tbody>
</table>

Quiz Review

- 1) A) Single Cycle CPU time?

<table>
<thead>
<tr>
<th></th>
<th>Instruction Cache</th>
<th>Decode and Register Read</th>
<th>ALU</th>
<th>Data Cache</th>
<th>Register Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type</td>
<td>3</td>
<td>2.8</td>
<td>0.9</td>
<td>--</td>
<td>2.8</td>
</tr>
<tr>
<td>Load</td>
<td>3</td>
<td>2.8</td>
<td>0.9</td>
<td>1</td>
<td>2.8</td>
</tr>
<tr>
<td>Store</td>
<td>3</td>
<td>2.8</td>
<td>0.9</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Branch</td>
<td>3</td>
<td>2.8</td>
<td>0.9</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

- 1) B) Multicycle CPU time?  Better choice?
Quiz Review

• Why are there "registers" in the MULTICYCLE design?

  - Can you name them and tell me what they are used for?

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?
### Field name | Value | Signals active | Comment
--- | --- | --- | ---
**ALU control** | Add | ALUOp = 00 | Cause the ALU to add.
| Sub | ALUOp = 01 | Cause the ALU to subtract, this implements the compare for branches.
| Func code | ALUOp = 10 | Use the instruction's function code to determine ALU control.
**SRC1** | PC | ALUSrc = 0 | Use the PC as the first ALU input.
| A | ALUSrc = 1 | Register A is the first ALU input.
| B | ALUSrc = 00 | Register B is the second ALU input.
| ExtALU | ALUSrc = 10 | Use the output of the sign extension unit as the second ALU input.
**SRC2** | ExtALU | ALUSrc = 11 | Use the output of the shift-by-two unit as the second ALU input.
**Register control** | RegWrite, RegDst = 1, MemtoReg = 0 | Write a register using the rt field of the IR as the register number and the contents of the ALUOut as the data.
**Write ALU** | RegWrite, RegDst = 0, MemtoReg = 1 | Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
**Write MDR** | MDRWrite, MIMread, MIFread = 1 | Head memory using the ALUOut as address, contents of B as the data.
**Memory** | MIMread, MIFread = 0 | Head memory using the PC as address, write result into IR (and the MDR).
**Head ALU** | MIMwrite, IorD = 1 | Head memory using the ALUOut as address, write result into MDR.
**Write ALU** | MIMwrite, IorD = 1 | Write memory using the ALUOut as address, contents of B as the data.
**PC write control** | PCWrite, PCSource = 00 | Write the output of the ALU into the PC.
| PCWriteCond | PCSource = 01 | If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
| Jump address | PCSource = 10, PCWrite | Write the PC with the jump address from the instruction.
**Sequence** | AddCtl = 11 | Choose the next microinstruction sequentially.
**Fetch** | AddCtl = 00 | Go to the first microinstruction to begin a new instruction.
| Dispatch 1 | AddCtl = 01 | Dispatch using the ROM 1.
| Dispatch 2 | AddCtl = 10 | Dispatch using the ROM 2.
ADDI: Microprogram Insts for new FSM states

<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>ALU</td>
<td>Seq</td>
<td>Fetch</td>
</tr>
<tr>
<td>MemEx</td>
<td>Add</td>
<td>Ext1</td>
<td>Ext1</td>
<td>Read</td>
<td>ALU</td>
<td>Ext1</td>
<td>Seq</td>
</tr>
<tr>
<td>Func code</td>
<td>A</td>
<td>B</td>
<td></td>
<td>WriteALU</td>
<td>Seq</td>
<td>Fetch</td>
<td>Fetch</td>
</tr>
<tr>
<td>Sub</td>
<td>A</td>
<td>B</td>
<td></td>
<td>WriteALU</td>
<td>Seq</td>
<td>Fetch</td>
<td>Jump addr</td>
</tr>
</tbody>
</table>

- Modify Dispatch 1 (after ID) so that it uses the opcode to go to our new state "3"

Single Cycle CPU
Branches

- How are branch targets calculated?

- How are jump targets calculated?
  - What if our opcode was 8 bits instead of 6.
  - How would that change how would that change our calculation of our jump address?
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:
  - arithmetic overflow
  - divide by zero
  - I/O device signals completion to CPU
  - user program invokes the OS
  - memory parity error
  - illegal instruction
  - timer signal

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

<table>
<thead>
<tr>
<th>user code</th>
<th>exception handler: read status register</th>
</tr>
</thead>
<tbody>
<tr>
<td>user code</td>
<td>...</td>
</tr>
<tr>
<td>user code</td>
<td>...</td>
</tr>
<tr>
<td>user code</td>
<td>...</td>
</tr>
<tr>
<td>user code</td>
<td>status register</td>
</tr>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
<tr>
<td>user code</td>
<td>user code</td>
</tr>
</tbody>
</table>

overflow handler: ...

<table>
<thead>
<tr>
<th>user code</th>
<th>illegal inst handler: ....</th>
</tr>
</thead>
<tbody>
<tr>
<td>user code</td>
<td>...</td>
</tr>
<tr>
<td>user code</td>
<td>...</td>
</tr>
<tr>
<td>user code</td>
<td>I/O interrupt handler: ...</td>
</tr>
<tr>
<td>user code</td>
<td>...</td>
</tr>
</tbody>
</table>

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

Supporting exceptions in our FSM

Instruction Fetch, state 0
Start
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00

MemInst FSM
R-type Inst FSM
Branch Inst FSM

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

Instruction Decode/ Register Fetch, state 1

ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00

To state 10ten

Opcode = anything else
Supporting exceptions in our FSM

from ID

R-type instructions

ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

RegDst = 1
RegWrite
MemtoReg=0

To state 11

To state 0
(IF)

Supporting exceptions in our FSM

illegal instruction

arithmetic overflow

intCaused=0
CauseWrite

state 10

intCaused=1
CauseWrite

state 11

state 12

PCWrite
EPCWrite

Interrupt Handler Address

IntSource

IntCause

CauseWrite

sub
4

fetch

37

38
Key Points

- microprogramming can simplify (conceptually) CPU control generation
- a microprogram is a small program inside the CPU that executes the individual instructions of the “real” program.
- Exception-handling is difficult in the CPU, because the interactions between the executing instructions and the interrupt are complex and unpredictable.