Recap: Microprogramming

- Microprogramming is a convenient method for implementing structured control state diagrams:
  - Random logic replaced by microPC sequencer and ROM
  - Each line of ROM called a microinstruction: contains sequencer control + values for control points
  - Limited state transitions: branch to zero, next sequential, branch to microinstruction address from dispatch ROM

- Design of a Microprogramming language
  1. Start with list of control signals
  2. Group signals together that make sense (vs. random): called “fields”
  3. Place fields in some logical order (e.g., ALU operation & ALU operands first and microinstruction sequencing last)
  4. To minimize the width, encode operations that will never be used at the same time
  5. Create a symbolic legend for the microinstruction format, showing name of field values and how they set the control signals

Recap: Multicycle datapath (book)
Recap: Start with List of control signals

<table>
<thead>
<tr>
<th>Signal name</th>
<th>Effect when deasserted</th>
<th>Effect when asserted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUSelA</td>
<td>1st ALU operand = PC</td>
<td>1st ALU operand = Reg[rs]</td>
</tr>
<tr>
<td>RegWrite</td>
<td>None</td>
<td>Reg. is written</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>Reg. write data input = ALU</td>
<td>Reg. write data input = memory</td>
</tr>
<tr>
<td>RegDst</td>
<td>Reg. dest. no. = rt</td>
<td>Reg. dest. no. = rd</td>
</tr>
<tr>
<td>MemRead</td>
<td>None</td>
<td>Memory at address is read, MDR &lt;= Mem[addr]</td>
</tr>
<tr>
<td>MemWrite</td>
<td>None</td>
<td>Memory at address is written</td>
</tr>
<tr>
<td>IorD</td>
<td>Memory address = PC</td>
<td>Memory address = S</td>
</tr>
<tr>
<td>IRWrite</td>
<td>None</td>
<td>IR &lt;= Memory</td>
</tr>
<tr>
<td>PCWrite</td>
<td>None</td>
<td>PC &lt;= PCSource</td>
</tr>
<tr>
<td>PCWriteCond</td>
<td>None</td>
<td>IF ALUzero then PC &lt;= PCSource</td>
</tr>
<tr>
<td>PCSource</td>
<td>PCSource = ALU</td>
<td>PCSource = ALUout</td>
</tr>
<tr>
<td>ExtOp</td>
<td>Zero</td>
<td>Extended</td>
</tr>
</tbody>
</table>

Recap: Group together related signals

Signal name | Value | Effect
---|---|---
ALUOp | 00 | ALU adds
10 | ALU does function code
11 | ALU does logical OR
ALUSelB | 00 | 2nd ALU input = 4
01 | 2nd ALU input = Reg[rt]
10 | 2nd ALU input = extended, shift left 2
11 | 2nd ALU input = extended

Recap: Group into Fields, Order and Assign Names

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Values for Field</th>
<th>Function of Field with Specific Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Add</td>
<td>ALU adds</td>
</tr>
<tr>
<td>Subt.</td>
<td>ALU subtracts</td>
<td></td>
</tr>
<tr>
<td>Func</td>
<td>ALU does function code</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>ALU does logical OR</td>
<td></td>
</tr>
<tr>
<td>SRC1</td>
<td>PC</td>
<td>1st ALU input &lt;= PC</td>
</tr>
<tr>
<td>rs</td>
<td>1st ALU input &lt;= Reg[rs]</td>
<td></td>
</tr>
<tr>
<td>SRC2</td>
<td>4</td>
<td>2nd ALU input &lt;= 4</td>
</tr>
<tr>
<td>Extnd</td>
<td>2nd ALU input &lt;= sign ext. IR[15-0]</td>
<td></td>
</tr>
<tr>
<td>Extnd0</td>
<td>2nd ALU input &lt;= zero ext. IR[15-0]</td>
<td></td>
</tr>
<tr>
<td>Extshift</td>
<td>2nd ALU input &lt;= sign ext. st IR[15-0]</td>
<td></td>
</tr>
<tr>
<td>rt</td>
<td>2nd ALU input &lt;= Reg[rt]</td>
<td></td>
</tr>
<tr>
<td>dest(ination)</td>
<td>rd ALU</td>
<td>Reg[rd] &lt;= ALUout</td>
</tr>
<tr>
<td>Mem(ory)</td>
<td>Read PC</td>
<td>Read memory using PC</td>
</tr>
<tr>
<td>Memreg</td>
<td>IR</td>
<td>IR &lt;= Mem</td>
</tr>
<tr>
<td>PCWrite</td>
<td>PCWr</td>
<td>PC &lt;= PCSource</td>
</tr>
<tr>
<td>PCSrc</td>
<td>IF Zero then PCSrc &lt;= ALUout else ALU</td>
<td></td>
</tr>
<tr>
<td>PCWrCond</td>
<td>IF Zero then PC &lt;= PCSrc</td>
<td></td>
</tr>
<tr>
<td>Seq(uencing)</td>
<td>Seq</td>
<td>Go to next sequential instruction</td>
</tr>
<tr>
<td>Fetch</td>
<td>Go to the first microinstruction</td>
<td></td>
</tr>
<tr>
<td>Dispatch</td>
<td>Dispatch using ROM.</td>
<td></td>
</tr>
</tbody>
</table>

Recap: Quick check: what do these fieldnames mean?

Destination:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>RegWrite</th>
<th>MemToReg</th>
<th>RegDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>---</td>
<td>0</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>01</td>
<td>rd ALU</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>rt ALU</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>rt MEM</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

SRC2:

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>ALUSelB</th>
<th>ExtOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>---</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>001</td>
<td>4</td>
<td>00</td>
<td>X</td>
</tr>
<tr>
<td>010</td>
<td>rt</td>
<td>01</td>
<td>X</td>
</tr>
<tr>
<td>011</td>
<td>ExtShift</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>Extend</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>Extend0</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Recap: Single Bit Control

Recap: Multiple Bit Control

Recap: Ideal Memory

Recap: 32-bit Control

Recap: Instruction Register File

Recap: PC Write

Recap: PC Read

Recap: Control

Recap: ALU Out Ext

Recap: ALUSelB

Recap: Mem Write

Recap: Mem Read

Recap: Reg Write
Recap: Finite State Machine (FSM) Spec

- IR <= MEM[PC]
  - “instruction fetch”
- PC <= PC + 4
- R-type:
  - ALUout <= A fun B
  - R[rd] <= ALUout
  - ORi
  - ALUout <= A or ZX
  - R[rt] <= ALUout
  - ORi
  - ALUout <= A + SX
  - R[rt] <= M
  - MEM <= M <= MEM[ALUout]
  - LW: ALUout <= A + SX
    - MEM[ALUout] <= B
  - SW:
    - 0000: Add PC 4
    - 0001: Add PC Extshft
    - 0010: Subt. rs rt
    - 0011: Or rs Extend0
    - 0100: Func rs rt
    - 0101: rd ALU
    - 0110: Or rs Extend0
    - 0111: rt ALU
    - 1000: Add rs Extend
    - 1001: Read ALU
    - 1010: rt MEM
    - 1011: Add rs Extend
    - 1100: Write ALU
    - 1101: Add rs Extend
    - 1110: Write ALU

Recap: Microprogram it yourself!

- Addr ALU SRC1 SRC2 Dest. Memory Mem. Reg. PC Write Sequencing
  - Fetch:
    - 0000: Add PC 4 Read PC IR ALU Seq Dispatch
    - 0001: Add PC Extshft
  - BEQ:
    - 0010: Subt. rs rt ALUoutCond. Fetch
    - 0011: Or rs Extend0
    - 0100: Func rs rt
    - 0101: rd ALU Fetch
  - Rtype:
    - 0110: Or rs Extend0
    - 0111: rt ALU Fetch
  - LW:
    - 1000: Add rs Extend
    - 1001: Read ALU Seq
    - 1010: rt MEM Fetch
  - SW:
    - 1011: Add rs Extend
    - 1100: Write ALU Fetch

Recap: Specific Sequencer from last lecture

- Sequencer-based control unit from last lecture
  - Called “microPC” or “μPC” vs. state register

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>fetch</td>
<td>Next µaddress = 0</td>
</tr>
<tr>
<td>01</td>
<td>dispatch</td>
<td>Next µaddress = dispatch ROM</td>
</tr>
<tr>
<td>10</td>
<td>seq</td>
<td>Next µaddress = µaddress + 1</td>
</tr>
</tbody>
</table>

- Opcode: Dispatch state
  - 000000: Rtype (0100)
  - 000100: BEQ (0010)
  - 001100: ORI (0100)
  - 100011: LW (0000)
  - 101011: SW (0011)

Exceptions

- Exception = unprogrammed control transfer
  - system takes action to handle the exception
    - must record the address of the offending instruction
    - record any other information necessary to return afterwards
  - returns control to user
    - must save & restore user state

- Allows construction of a “user virtual machine”
Two Types of Exceptions: Interrupts and Traps

- **Interrupts**
  - caused by external events:
    - Network, Keyboard, Disk I/O, Timer
  - asynchronous to program execution
    - Most interrupts can be disabled for brief periods of time
    - Some (like “Power Failing”) are non-maskable (NMI)
  - may be handled between instructions
  - simply suspend and resume user program

- **Traps**
  - caused by internal events
    - exceptional conditions (overflow)
    - errors (parity)
    - faults (non-resident page)
  - synchronous to program execution
  - condition must be remedied by the handler
  - instruction may be retried or simulated and program continued
  - or program may be aborted

Precise Exceptions

- **Precise** ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started
  - Same system code will work on different implementations
  - Position clearly established by IBM
  - Difficult in the presence of pipelining, out-of-order execution, ...
  - MIPS takes this position

- **Imprecise** ⇒ system software has to figure out what is where and put it all back together

- Performance goals often lead designers to forsake precise interrupts
  - system software developers, user, markets etc. usually wish they had not done this

- Modern techniques for out-of-order execution and branch prediction help implement precise interrupts

Big Picture: user / system modes

- Two modes of execution (user/system):
  - operating system runs in privileged mode and has access to all of the resources of the computer
  - presents “virtual resources” to each user that are more convenient than the physical resources
    - files vs. disk sectors
    - virtual memory vs physical memory
  - protects each user program from others
  - protects system from malicious users.
  - OS is assumed to “know best”; and is trusted code, so enter system mode on exception

- Exceptions allow the system to taken action in response to events that occur while user program is executing:
  - Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
  - “Unimplemented instruction” used to emulate instructions that were not included in hardware (i.e. MicroVax)

Administrivia

- **Midterm I this Wednesday**
  - 5:30 - 8:30 in 306 Soda Hall
  - Bring a Calculator!
  - One 8 1/2 by 11 page (both sides) of notes
  - Make up exam: Tuesday 5:30 – 8:30 in 606 Soda Hall
  - Meet at LaVal’s afterwards for Pizza
  - North-side Lavals!
  - I’ll Buy!

- **Materials through Chapter 5, Appendix A, B & C**
  - Should understand single-cycle processor
  - Multicycle processor, including exceptions

- **Lab 4 breakdown due midnight tomorrow**
  - EMail to your TA
  - This is a complicated lab – may need to give updates as we get the boards installed
### Addressing the Exception Handler

**Traditional Approach: Interrupt Vector**
- $PC \leftarrow MEM[IV\_base + cause || 00]$
- 370, 68000, Vax, 80x86, ...

**RISC Handler Table**
- $PC \leftarrow IT\_base + cause || 0000$
- saves state and jumps
- Sparc, PA, M88K, ...

**MIPS Approach: fixed entry**
- $PC \leftarrow EXC\_addr$
- Actually very small table
  - RESET entry
  - TLB
  - other

### Saving State

**Push it onto the stack**
- Vax, 68k, 80x86

**Shadow Registers**
- M88k
  - Save state in a shadow of the internal pipeline registers

**Save it in special registers**
- MIPS EPC, BadVaddr, Status, Cause

### Additions to MIPS ISA to support Exceptions?

- Exception state is kept in “coprocessor 0”.
  - Use mfc0 read contents of these registers
  - Every register is 32 bits, but may be only partially defined

**BadVAddr (register 8)**
- register contained memory address at which memory reference occurred

**Status (register 12)**
- interrupt mask and enable bits

**Cause (register 13)**
- the cause of the exception
  - Bits 5 to 2 of this register encodes the exception type (e.g. undefined instruction=10 and arithmetic overflow=12)
  - EPC (register 14)
  - address of the affected instruction (register 14 of coprocessor 0).

**Control signals to write BadVAddr, Status, Cause, and EPC**
- Be able to write exception address into PC (8000 0080hex)
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor): PC = PC - 4

### Details of Status register

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>8</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Mask</td>
<td>old</td>
<td>prev</td>
<td>current</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Mask** = 1 bit for each of 5 hardware and 3 software interrupt levels
  - 1 => enables interrupts
  - 0 => disables interrupts

- **k** = kernel/user
  - 0 => was in the kernel when interrupt occurred
  - 1 => was running user mode

- **e** = interrupt enable
  - 0 => interrupts were disabled
  - 1 => interrupts were enabled

- When interrupt occurs, 6 LSB shifted left 2 bits, setting 2 LSB to 0
  - run in kernel mode with interrupts disabled
### Details of Cause register

<table>
<thead>
<tr>
<th>Status</th>
<th>Pending</th>
<th>Code</th>
</tr>
</thead>
</table>

° **Pending interrupt** 5 hardware levels: bit set if interrupt occurs but not yet serviced
- handles cases when more than one interrupt occurs at same time, or while records interrupt requests when interrupts disabled

° **Exception Code** encodes reasons for interrupt
- 0 (INT) => external interrupt
- 4 (ADDRL) => address error exception (load or instr fetch)
- 5 (ADDRS) => address error exception (store)
- 6 (IBUS) => bus error on instruction fetch
- 7 (DBUS) => bus error on data fetch
- 8 (Syscall) => Syscall exception
- 9 (BKPT) => Breakpoint exception
- 10 (RI) => Reserved Instruction exception
- 12 (OVF) => Arithmetic overflow exception

### Part of the handler in trap_handler.s

```
entry: # Exceptions/interrupts come here
.set noat
.move $k1 $at # Save $at
.set at
.sw \$v0 \$s1 # Not re-entrant and we can't trust $sp
sf0 $k0 $s13 # Cause << Grab the cause register
li \$v0 4 # syscall 4 (print_str)
lw \$s0 __m1_syscall
syscall
li \$v0 1 # syscall 1 (print_int)
srl \$a0 \$k0 2 # shift Cause reg
syscall

.set: 
.lw \$v0 \$s1
lw \$a0 \$s2
mfc0 \$k0 \$13 # Cause << Get the return address (EPC)
.set noat
.move \$at \$k1 # Restore $at
.set at
.rfe # Return from exception handler
.addiu \$s0 \$k0 4 # Return to next instruction
jr \$k0
```

### Example: How Control Handles Traps in our FSD

° **Undefined Instruction**–detected when no next state is defined from state 1 for the op value.
- We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
- Shown symbolically using “other” to indicate that the op field does not match any of the opcodes that label arcs out of state 1.

° **Arithmetic overflow**–detected on ALU ops such as signed add
- Used to save PC and enter exception handler

° **External Interrupt** – flagged by asserted interrupt line
- Again, must save PC and enter exception handler

° **Note:** Challenge in designing control of a real machine is to handle different interactions between instructions and other exception-causing events such that control logic remains small and fast.
- Complex interactions makes the control unit the most challenging aspect of hardware design

### How add traps and interrupts to state diagram?

```
```

**Interruptions are precise** because user-visible state committed after exceptions flagged!
But: What has to change in our μ-sequencer?

- Need concept of branch at micro-code level

\[ R \text{-type} \]

\[ S := A \text{ fun } B \]

\[ 0100 \]

\[ \text{overflow} \]

\[ \text{EPC} := \text{PC} - 4 \]

\[ \text{PC} := \exp \text{addr cause} = 12 \text{ (Ovf)} \]

Example: Can easily use with for non-ideal memory

- Instruction fetch

\[ \text{IR} := \text{MEM}[\text{PC}] \]

- Decode / operand fetch

\[ A := \text{R}[rs] \]

\[ B := \text{R}[rt] \]

- Execute

\[ S := A \text{ or } ZX \]

\[ R[rt] := S \]

\[ \text{PC} := \text{PC} + 4 \]

\[ \text{MEM}[S] := B \]

- Write-back

\[ \text{MEM}[S] := B \]

Recall: Performance Evaluation

- What is the average CPI?

<table>
<thead>
<tr>
<th>Type</th>
<th>CPI, for type</th>
<th>Frequency</th>
<th>CPI × freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arith/Logic</td>
<td>4</td>
<td>40%</td>
<td>1.6</td>
</tr>
<tr>
<td>Load</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
</tr>
<tr>
<td>Store</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
</tr>
<tr>
<td>branch</td>
<td>3</td>
<td>20%</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Average CPI: 4.1
**Question #2: Can we get CPI < 4.1?**

Seems to be lots of “idle” hardware
- Why not overlap instructions???

**The Big Picture: Where are We Now?**

- The Five Classic Components of a Computer

**Pipelining is Natural!**

- Laundry Example
  - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

**Sequential Laundry**

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?
Pipelined Laundry: Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads

3/10/03 ©UCB Spring 2003
CS152 / Kubiatowicz
Lec12.33

Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced speedup = Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

3/10/03 ©UCB Spring 2003
CS152 / Kubiatowicz
Lec12.34

The Five Stages of Load

- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

3/10/03 ©UCB Spring 2003
CS152 / Kubiatowicz
Lec12.35

Note: These 5 stages were there all along!

3/10/03 ©UCB Spring 2003
CS152 / Kubiatowicz
Lec12.36
Pipelining

- Improve performance by increasing throughput

Ideal speedup is number of stages in the pipeline. Do we achieve this?

Graphically Representing Pipelines

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

Basic Idea

- What do we need to add to split the datapath into stages?

Conventional Pipelined Execution Representation

- Time
- Program Flow

Can help with answering questions like:
- how many cycles does it take to execute this code?
- what is the ALU doing during cycle 4?
- use this representation to help understand datapaths
### Single Cycle, Multiple Cycle, vs. Pipeline

**Single Cycle Implementation:**
- Load
- Store
- Waste

**Multiple Cycle Implementation:**
- Cycle 1
- Cycle 2
- Cycle 3
- Cycle 4
- Cycle 5
- Cycle 6
- Cycle 7
- Cycle 8
- Cycle 9
- Cycle 10

**Pipeline Implementation:**
- Ifetch
- Reg
- Exec
- Mem
- Wr

---

### Why Pipeline?

- **Suppose we execute 100 instructions**
  - Single Cycle Machine: 45 ns/cycle × 1 CPI × 100 inst = 4500 ns
  - Multicycle Machine: 10 ns/cycle × 4.1 CPI (due to inst mix) × 100 inst = 4100 ns
  - Ideal pipelined machine: 10 ns/cycle × (1 CPI × 100 inst + 4 cycle drain) = 1040 ns

### Can pipelining get us into trouble?

- **Yes:** Pipeline Hazards
  - **Structural hazards:** attempt to use the same resource two different ways at the same time
    - E.g., combined washer/dryer would be a structural hazard
  - **Control hazards:** attempt to make a decision before condition is evaluated
    - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
  - **Data hazards:** attempt to use item before it is ready
    - E.g., one sock of pair in dryer and one in washer; can’t fold until get sock from washer through dryer
    - Instruction depends on result of prior instruction still in the pipeline

- **Can always resolve hazards by waiting**
  - Pipeline control must detect the hazard
  - Take action (or delay action) to resolve hazards
Single Memory is a Structural Hazard

- Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance

- Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI ≥ 1.3
  - otherwise resource is more than 100% utilized

Control Hazard Solution #1: Stall

- Stall: wait until decision is clear
- Impact: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
- Move decision to end of decode
  - save 1 cycle per branch

Control Hazard Solution #2: Predict

- Predict: guess one direction then back up if wrong
- Impact: 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)
  - Need to “Squash” and restart following instruction if wrong
  - Produce CPI on branch of (1 * .5 + 2 * .5) = 1.5
  - Total CPI might then be: 1.5 * .2 + 1 * .8 = 1.1 (20% branch)
- More dynamic scheme: history of 1 branch (- 90%)
Control Hazard Solution #3: Delayed Branch

- **Delayed Branch**: Redefine branch behavior (takes place after next instruction)
- **Impact**: 0 clock cycles per branch instruction if can find instruction to put in “slot” (~50% of time)
- **As launch more instruction per clock cycle, less useful**

Data Hazard on r1:

- Dependencies backwards in time are hazards
- **“Forward”** result from one stage to another

- add r1, r2, r3
- sub r4, r1, r3
- and r6, r1, r7
- or r8, r1, r9
- xor r10, r1, r11

Data Hazard Solution:

- “or” OK if define read/write properly
Dependencies backwards in time are hazards

Can’t solve with forwarding:
Must delay/stall instruction dependent on loads

Designing a Pipelined Processor

Go back and examine your datapath and control diagram
associated resources with states
ensure that flows do not conflict, or figure out how to resolve
assert control in appropriate stage

Control and Datapath: Split state diag into 5 pieces
Summary: Exceptions

- Microprogramming is a fundamental concept
  - Implement an instruction set by building a very simple processor and interpreting the instructions
  - Essential for very complex instructions and when few register transfers are possible
  - Control design reduces to Microprogramming

- Exceptions are the hard part of control
  - Need to find convenient place to detect exceptions and to branch to state or microinstruction that saves PC and invokes the operating system
  - Providing clean interrupt model gets hard with pipelining!

- Precise Exception ⇒ state of the machine is preserved as if program executed up to the offending instruction
  - All previous instructions completed
  - Offending instruction and all following instructions act as if they have not even started

Summary: Pipelining

- Reduce CPI by overlapping many instructions
  - Average throughput of approximately 1 CPI with fast clock

- Utilize capabilities of the Datapath
  - Start next instruction while working on the current one
  - Limited by length of longest stage (plus fill/flush)
  - Detect and resolve hazards

- What makes it easy
  - All instructions are the same length
  - Just a few instruction formats
  - Memory operands appear only in loads and stores

- What makes it hard?
  - Structural hazards: suppose we had only one memory
  - Control hazards: need to worry about branch instructions
  - Data hazards: an instruction depends on a previous instruction

Summary: Where this class is going

- We’ll build a simple pipeline and look at these issues
  - Lab 5 ⇒ Pipelined Processor
  - Lab 6 ⇒ With caches

- We’ll talk about modern processors and what’s really hard:
  - Branches (control hazards) are really hard!
  - Exception handling
  - Trying to improve performance with out-of-order execution, etc.
  - Trying to get CPI < 1 (Superscalar execution)