CS152
Computer Architecture and Engineering
Lecture 10

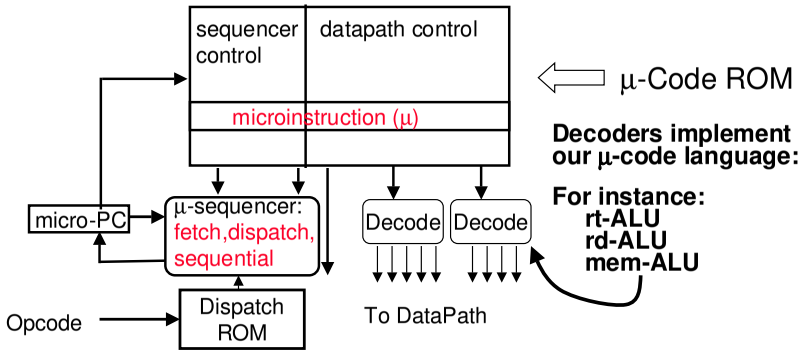
Exceptions (continued)
Introduction to Pipelining

March 1, 2004

John Kubiatowicz (www.cs.berkeley.edu/~kubitron)

lecture slides: http://inst.eecs.berkeley.edu/~cs152/

Recap: Microprogramming



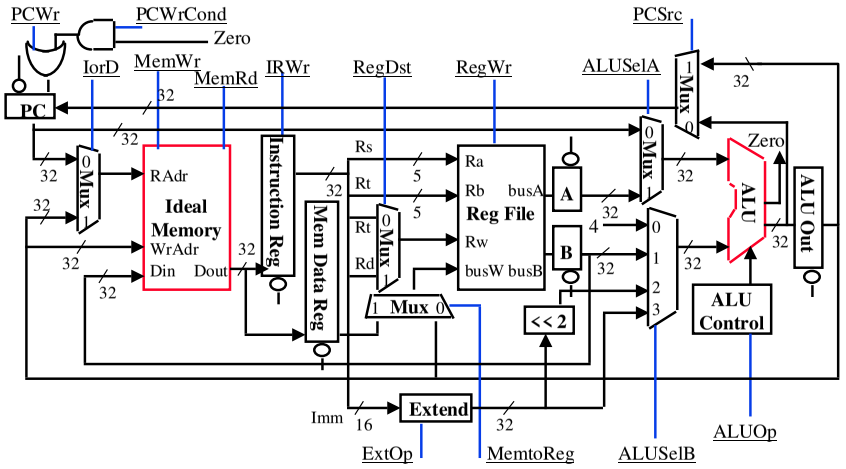
° **Microprogramming is a fundamental concept**
- implement an instruction set by building a very simple processor and interpreting the instructions
- essential for very complex instructions and when few register transfers are possible
- overkill when ISA matches datapath 1-1

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 μinstruction: contains sequencer control + values for control points
- limited state transitions: branch to zero, next sequential, branch to μinstruction address from displatch 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

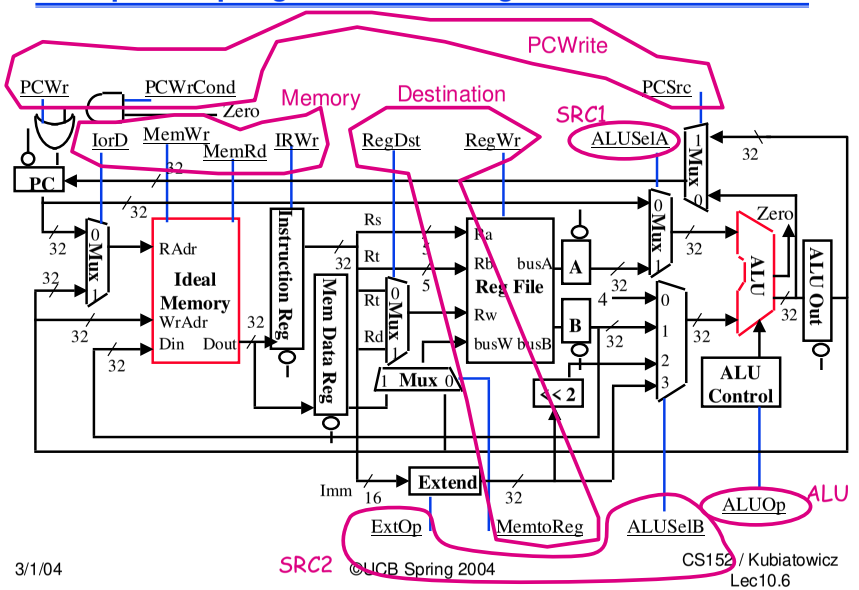
Single Bit Control

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

Multiple Bit Control

Signal name	Value	Effect
ALUOp	00	ALU adds
	01	ALU subtracts
	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 together related signals



Recap: Group into Fields, Order and Assign Names

ALU	SRC 1	SRC 2	Dest	Mem	Memreg	PCwrite	Seq

Field Name	Values for Field	Function of Field with Specific Value
ALU	Add	ALU adds
	Subt.	ALU subtracts
	Func	ALU does function code
	Or	ALU does logical OR
SRC1	PC	1st ALU input <= PC
	rs	1st ALU input <= Reg[rs]
SRC2	4	2nd ALU input <= 4
	Extend	2nd ALU input <= sign ext. IR[15-0]
	Extend0	2nd ALU input <= zero ext. IR[15-0]
	Extshft	2nd ALU input <= sign ex., sl IR[15-0]
	rt	2nd ALU input <= Reg[rt]
dest(ination)	rd ALU	Reg[rd] <= ALUout
	rt ALU	Reg[rt] <= ALUout
	rt Mem	Reg[rt] <= Mem
Mem(ory)	Read PC	Read memory using PC
	Read ALU	Read memory using ALUout for addr
	Write ALU	Write memory using ALUout for addr
Memreg	IR	IR <= Mem
PCwrite	PCwr	PC <= PCSource
	PCSrc	IF Zero then PCSource <= ALUout else ALU
	PCWrCond	IF Zero then PC <= PCSource
Seq(uencing)	Seq	Go to next sequential µinstruction
	Fetch	Go to the first microinstruction
	Dispatch	Dispatch using ROM.

Recap: Quick check: what do these fieldnames mean?

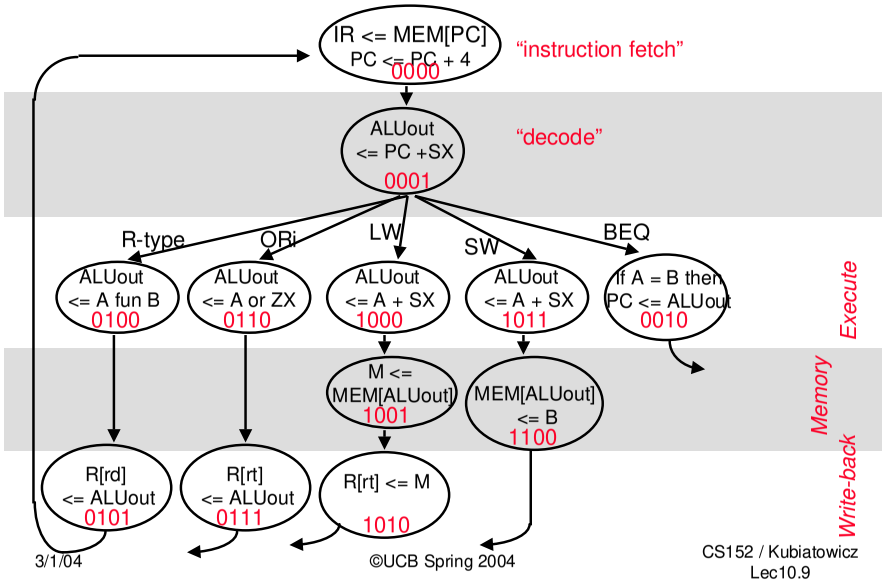
Destination:

Code	Name	RegWrite	MemToReg	RegDest
00	---	0	X	X
01	rd ALU	1	0	1
10	rt ALU	1	0	0
11	rt MEM	1	1	0

SRC2:

Code	Name	ALUSelB	ExtOp
000	---	X	X
001	4	00	X
010	rt	01	X
011	ExtShft	10	1
100	Extend	11	1
111	Extend0	11	0

Recap: Finite State Machine (FSM) Spec



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
0001:	Add	PC	Extshft					Dispatch
BEQ:								
0010:	Subt.	rs	rt				ALUoutCond.	Fetch
Rtype:								
0100:	Func	rs	rt					Seq
0101:				rd ALU				Fetch
ORI:								
0110:	Or	rs	Extend0					Seq
0111:				rt ALU				Fetch
LW:								
1000:	Add	rs	Extend					Seq
1001:					Read ALU			Seq
1010:				rt MEM				Fetch
SW:								
1011:	Add	rs	Extend					Seq
1100:					Write ALU			Fetch

Recap: Specific Sequencer from last lecture

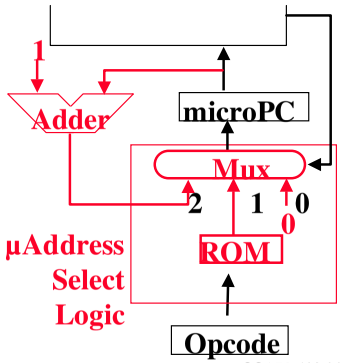
Sequencer-based control unit from last lecture

- Called "microPC" or "μPC" vs. state register

Code	Name	Effect
00	fetch	Next μaddress = 0
01	dispatch	Next μaddress = dispatch ROM
10	seq	Next μaddress = μaddress + 1

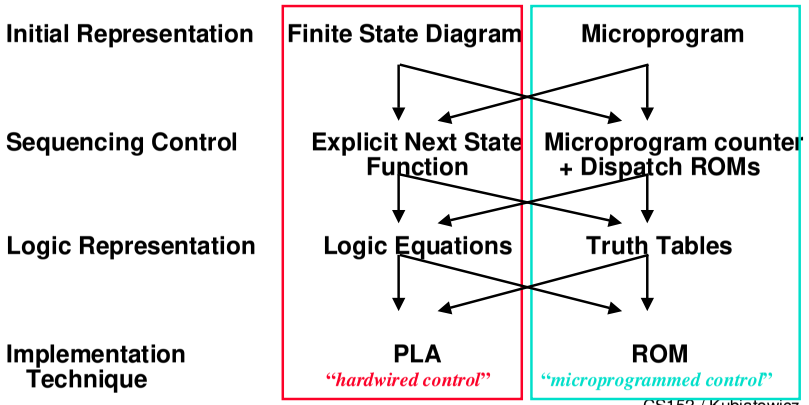
ROM:

Opcode: Dispatch state	
000000: Rtype	(0100)
⋮	
000100: BEQ	(0010)
⋮	
001101: ORI	(0110)
⋮	
100011: LW	(1000)
⋮	
101011: SW	(1011)
⋮	

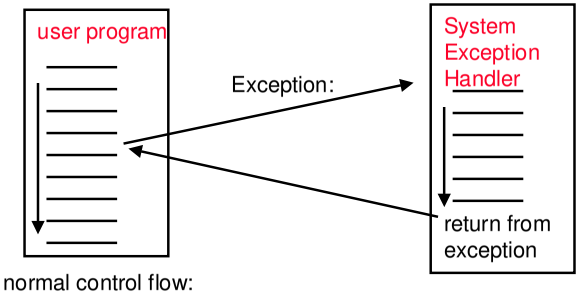


Review: Overview of Control

° Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Exceptions



normal control flow:
sequential, jumps, branches, calls, returns

° 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 constuction 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-ot-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 that 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

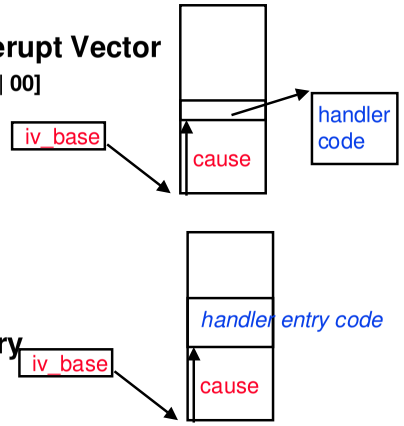
° Exceptions allow the system to taken action in response to events that occur while user program is executing:
- Might provide supplemental behavior (dealing with denormal floating-point numbers for instance).
- "Unimplemented instruction" used to emulate instructions that were not included in hardware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

Administrivia

- Midterm I on Wednesday 3/10
  - 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, some of Chapter 6, Appendix A, B & C
  - Should understand single-cycle processor
  - multicycle processor, including exceptions
- Get moving on Lab 3!
  - This is a hard lab, since there are so many new things

Addressing the Exception Handler

- Traditional Approach: Interupt Vector
  - PC <- MEM[IV_base + cause || 00]
  - 370, 68000, Vax, 80x86, . . .
- RISC Handler Table
  - PC <– IT_base + cause || 0000
  - saves state and jumps
  - Sparc, PA, M88K, . . .
- MIPS Approach: fixed entry
  - PC <– 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

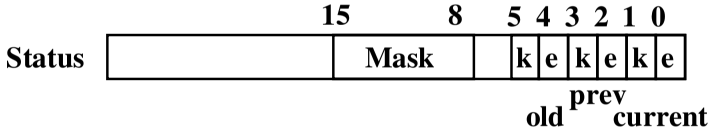
  BadVAddr (register 8)
  - register contained memory address at which memory reference occurred

  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)

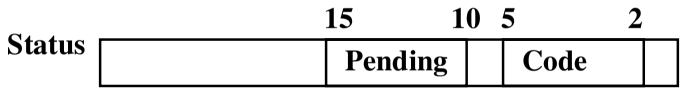
  EPC (register 14)
  - address of the affected instruction (register 14 of coprocessor 0).
- Control signals to write BadVAddr, Status, Cause, and EPC
- Be able to write exception address into PC ($8000\ 0080_{hex}$)
- May have to undo PC = PC + 4, since want EPC to point to offending instruction (not its successor): PC = PC - 4

Details of Status register



- 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



- 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

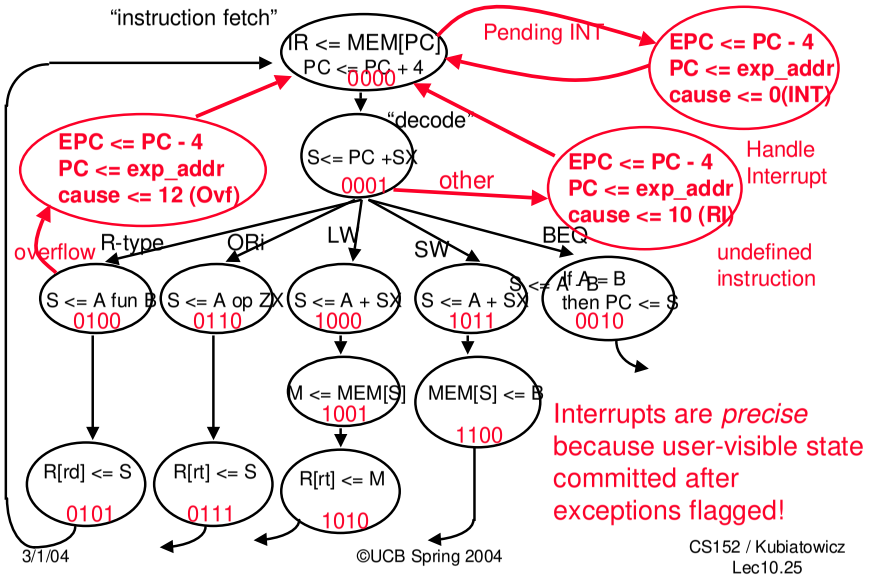
```
  .ktext 0x80000080
entry:                                  <= Exceptions/interrupts come here
  .set noat
      move  $k1 $at       # Save $at
  .set at
      sw    $v0 s1        # Not re-entrent and we can't trust $sp
      sw    $a0 s2
      mfc0  $k0 $13       # Cause         <= Grab the cause register
      li    $v0 4         # syscall 4 (print_str)
      la    $a0 __m1_
      syscall
      li    $v0 1         # syscall 1 (print_int)
      srl    $a0 $k0 2    # shift Cause reg
      syscall


ret:  lw    $v0 s1
      lw    $a0 s2
      mfc0  $k0 $14       # EPC           <= Get the return address (EPC)
  .set noat
      move  $at $k1       # Restore $at
  .set at
      rfe                 # Return from exception handler
      addiu $k0 $k0 4     # Return to next instruction
      jr    $k0
```


Example: How Control Handles Traps in our FSD

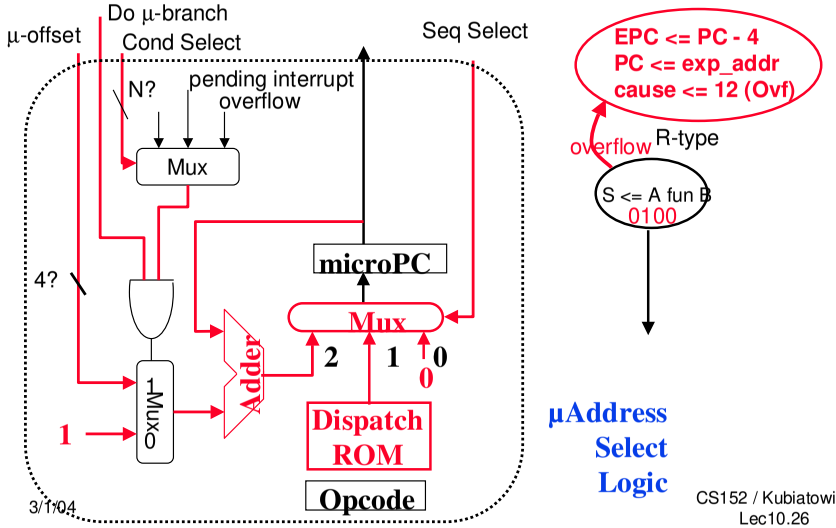
- Undefined Instruction–detected when no next state is defined from state 1 for the op value.
  - We handle this exception by defining the next state value for all op values other than lw, sw, 0 (R-type), jmp, beq, and ori as new state 12.
  - Shown symbolically using "other" to indicate that the op 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?

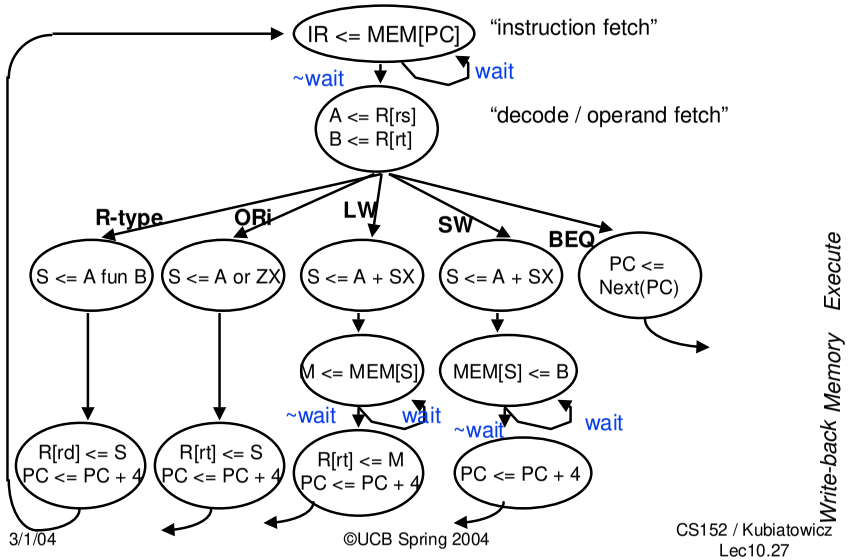


But: What has to change in our μ-sequencer?

° Need concept of *branch* at micro-code level



Example: Can easily use with for non-ideal memory



Question #1: Why do microcoding?

° If simple instruction could execute at very high clock rate…

° If you could even write compilers to produce microinstructions…

° If most programs use simple instructions and addressing modes…

° If microcode is kept in RAM instead of ROM so as to fix bugs …

° If same memory used for control memory could be used instead as cache for "macroinstructions"…

° Then why not skip instruction interpretation by a microprogram and simply compile directly into lowest language of machine? (microprogramming is overkill when ISA matches datapath 1-1)

Legacy Software and Microprogramming

- IBM bet company on 360 Instruction Set Architecture (ISA): single instruction set for many classes of machines
  - (8-bit to 64-bit)
- Stewart Tucker stuck with job of what to do about software compatibility
  - If microprogramming could easily do same instruction set on many different microarchitectures, then why couldn't multiple microprograms do multiple instruction sets on the same microarchitecture?
  - Coined term "emulation": instruction set interpreter in microcode for non-native instruction set
  - Very successful: in early years of IBM 360 it was hard to know whether old instruction set or new instruction set was more frequently used

Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

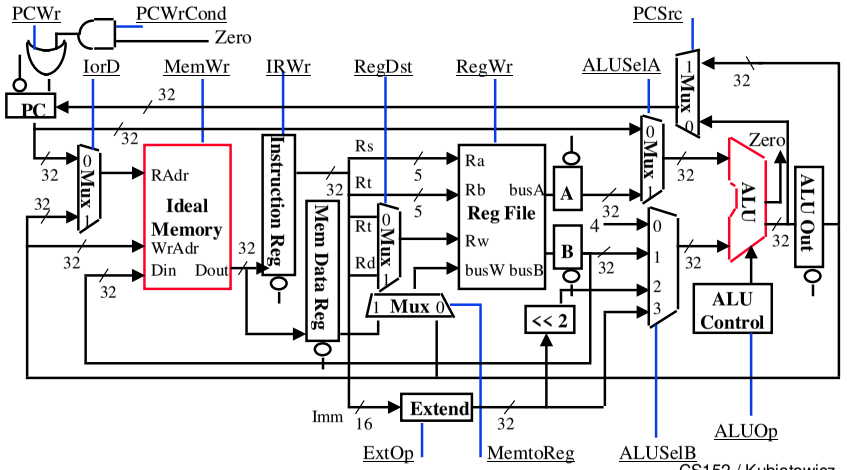
Recall: Performance Evaluation

- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

Type	CPI_i for type	Frequency	CPI_i x $freq_i$
Arith/Logic	4	40%	1.6
Load	5	30%	1.5
Store	4	10%	0.4
branch	3	20%	0.6
		Average CPI:4.1	

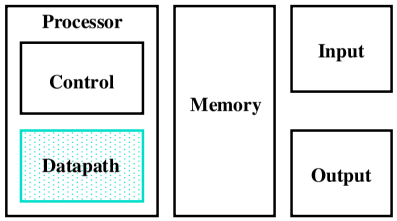
Question #2: Can we get CPI < 4.1?

- 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



° Next Topics:
- Pipelining by Analogy
- Pipeline hazards

Pipelining is Natural!

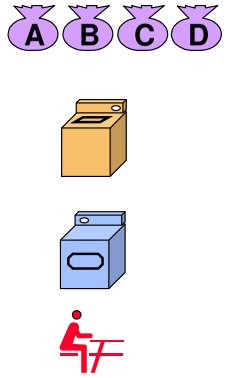
° Laundry Example

° Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold

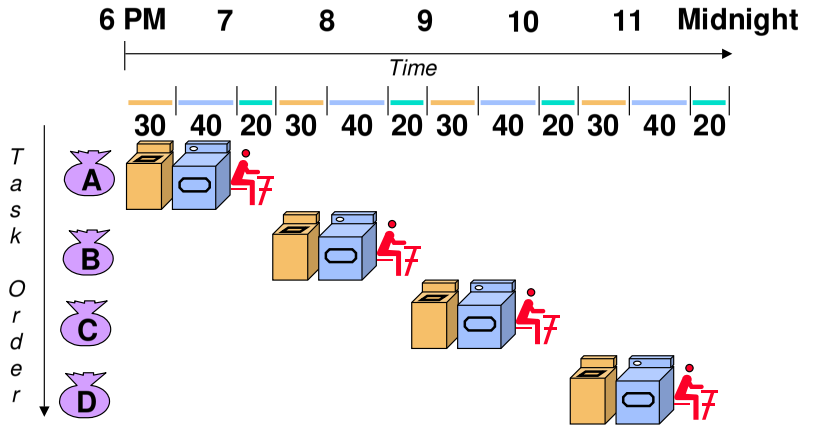
° Washer takes 30 minutes

° Dryer takes 40 minutes

° "Folder" takes 20 minutes



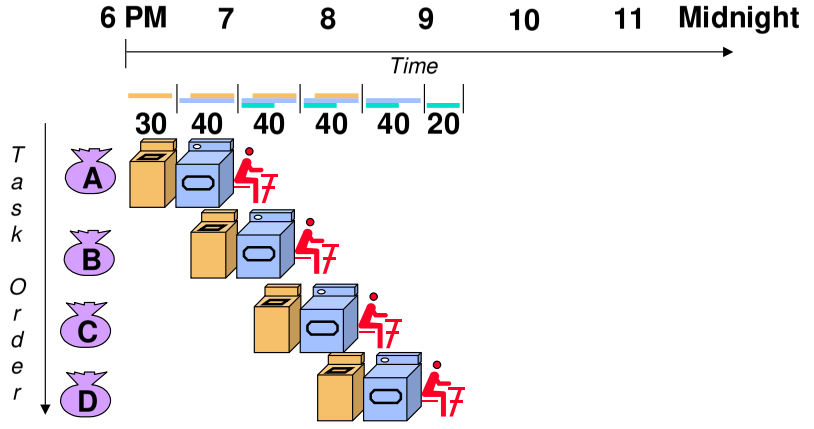
Sequential Laundry



° Sequential laundry takes 6 hours for 4 loads

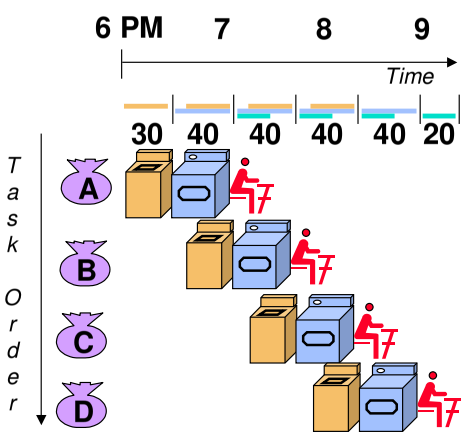
° If they learned pipelining, how long would laundry take?

Pipelined Laundry: Start work ASAP



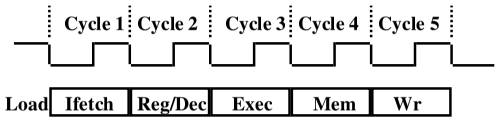
° Pipelined laundry takes 3.5 hours for 4 loads

Pipelining Lessons



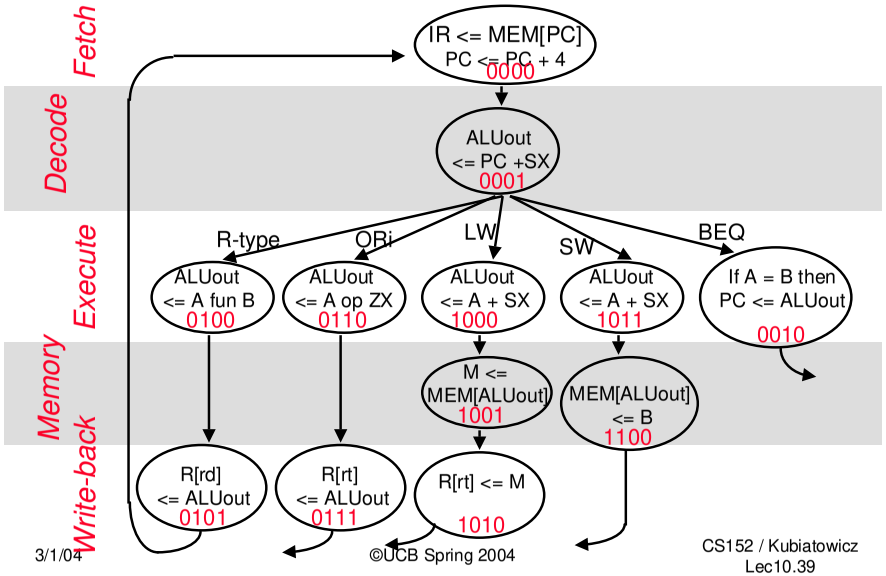
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

The Five Stages of Load



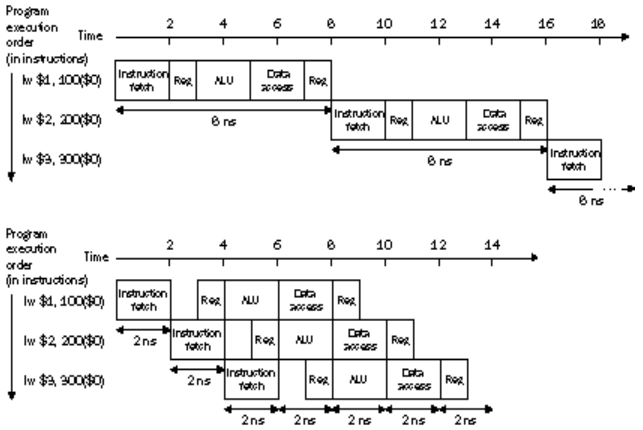
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Read the data from the Data Memory
- Wr: Write the data back to the register file

Note: These 5 stages were there all along!



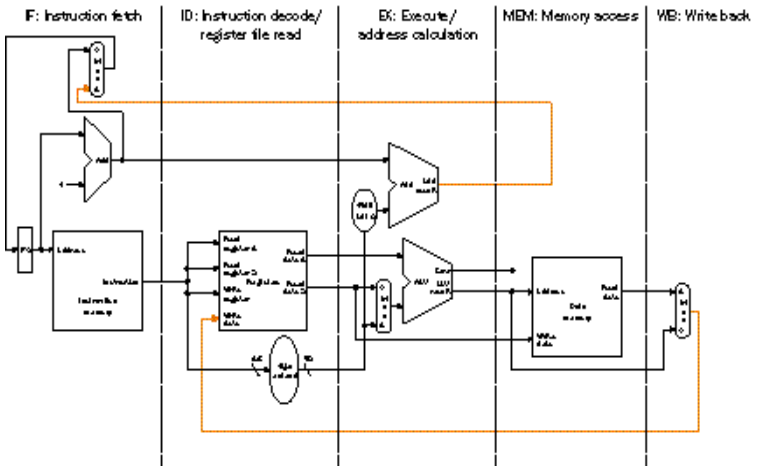
Pipelining

- Improve performance by increasing throughput



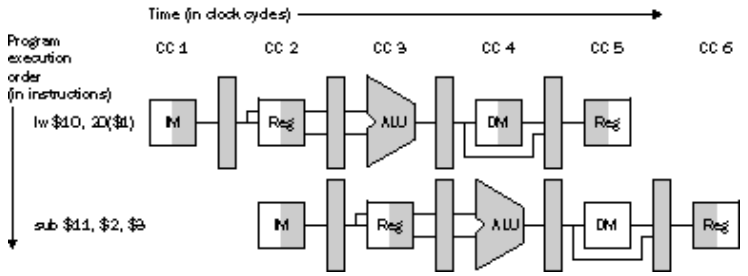
Ideal speedup is number of stages in the pipeline. Do we achieve this?

Basic Idea



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

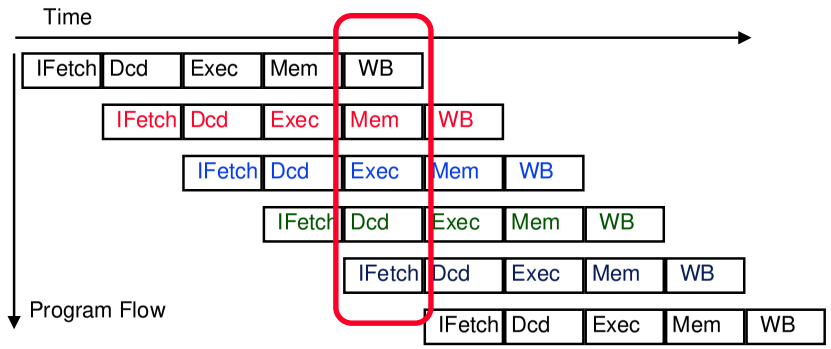
Graphically Representing Pipelines



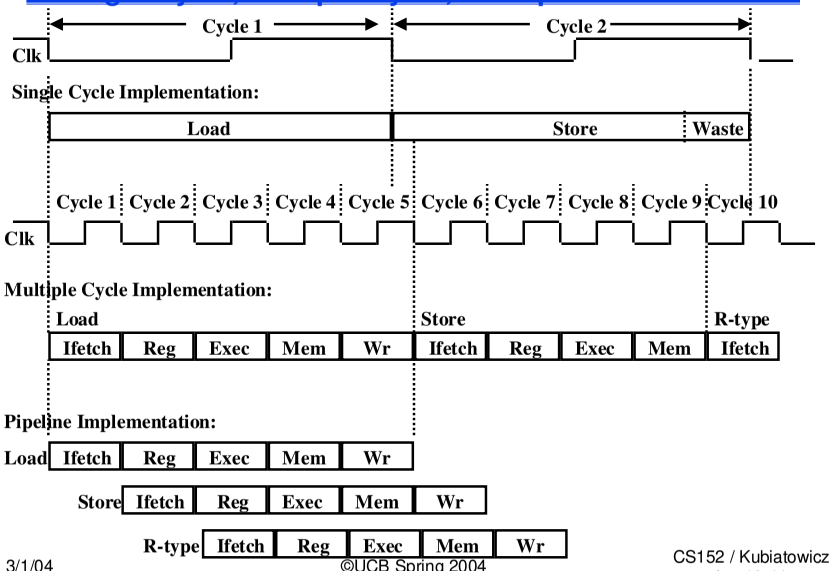
° **Can help with answering questions like:**

- **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**

Conventional Pipelined Execution Representation



Single Cycle, Multiple Cycle, vs. Pipeline



Why Pipeline?

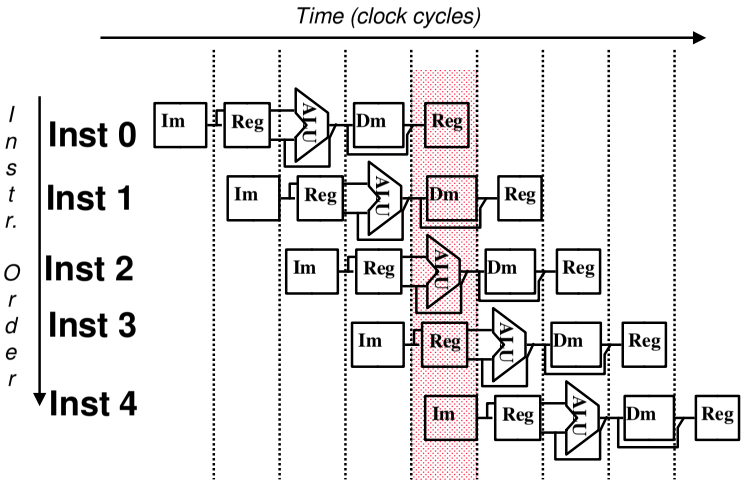
° Suppose we execute 100 instructions

° Single Cycle Machine
- 45 ns/cycle x 1 CPI x 100 inst = 4500 ns

° Multicycle Machine
- 10 ns/cycle x 4.1 CPI (due to inst mix) x 100 inst = 4100 ns

° Ideal pipelined machine
- 10 ns/cycle x (1 CPI x 100 inst + 4 cycle drain) = 1040 ns

Why Pipeline? Because we can!

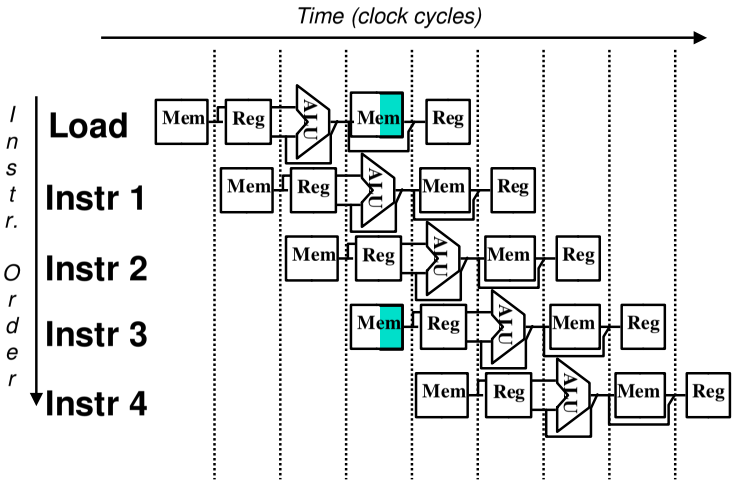


Can pipelining get us into trouble?

° 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 or folder busy doing something else (watching TV)
- 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
  - branch instructions
- 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

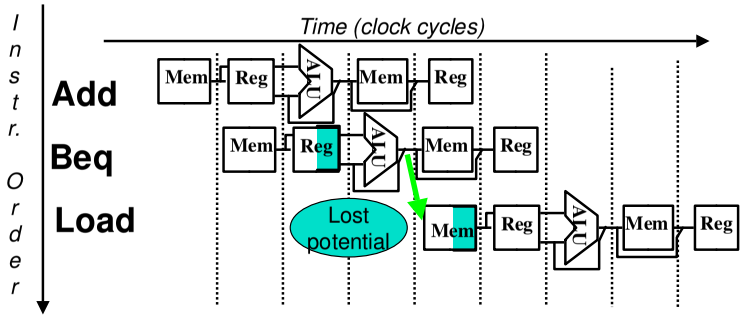


Detection is easy in this case! (right half highlight means read, left half write)

Structural Hazards limit performance

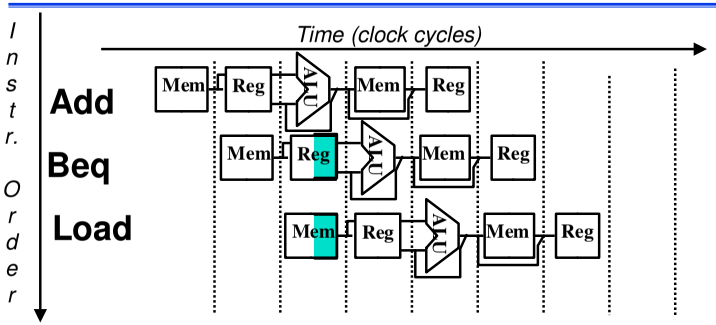
° Example: if 1.3 memory accesses per instruction and only one memory access per cycle then
  - average CPI ≥ 1.3
  - otherwise resource is more than 100% utilized

Control Hazard Solution #1: Stall



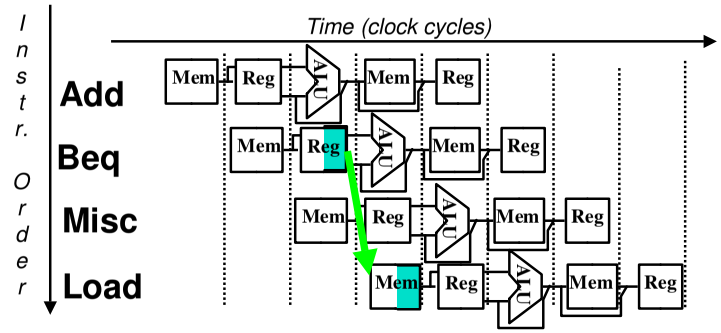
° Stall: wait until decision is clear
° Impact: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow
° Move decision to end of decode
  - save 1 cycle per branch

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

add r1,r2,r3

sub r4,r1,r3

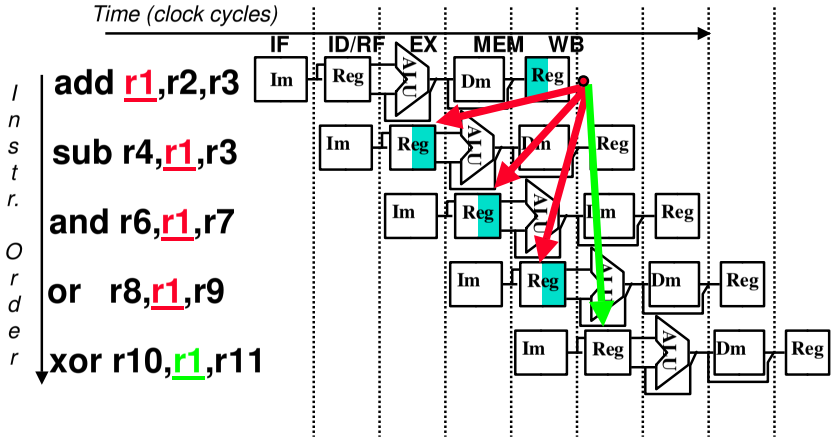
and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11

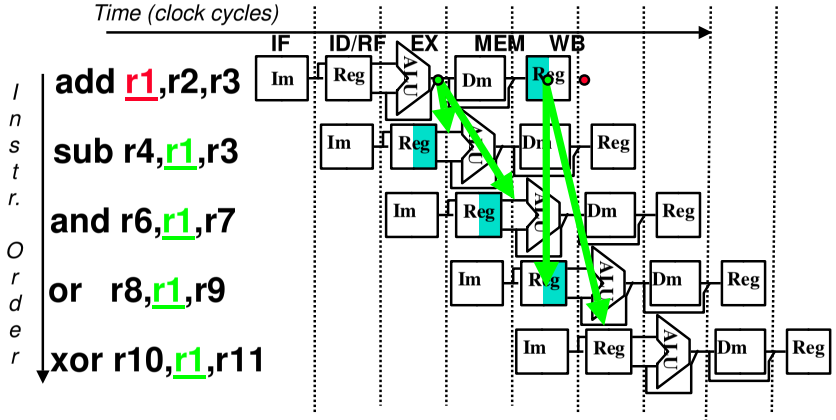
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

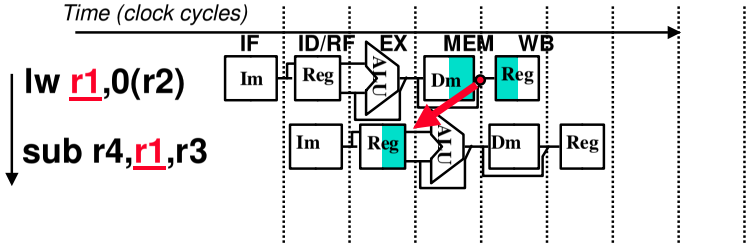
- "Forward" result from one stage to another



- "or" OK if define read/write properly

Forwarding (or Bypassing): What about Loads?

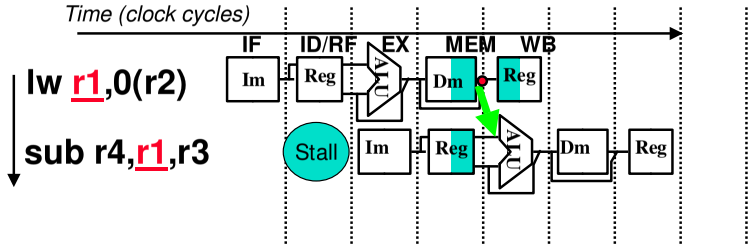
- Dependencies backwards in time are hazards



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

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

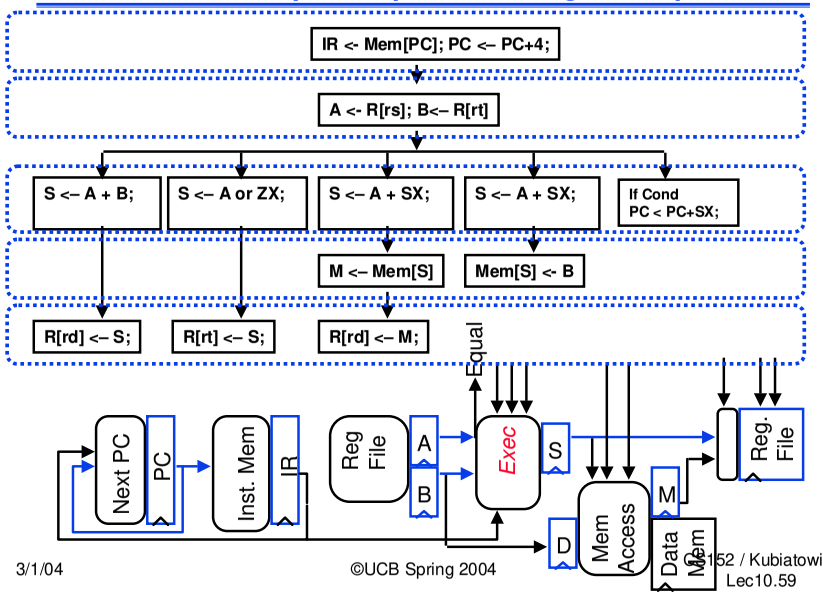


- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads

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)