CS152
Computer Architecture and Engineering
Lecture 12

Introduction to Pipelining

Mar 10, 1999

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

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

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

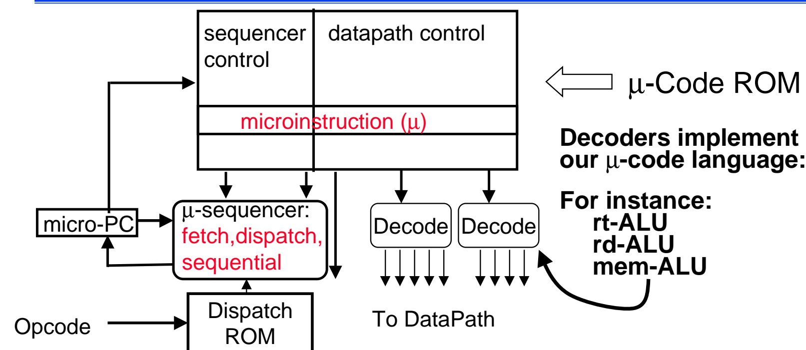
° Horizontal µCode: one control bit in µInstruction for every control line in datapath

° Vertical µCode: groups of control-lines coded together in µInstruction (e.g. possible ALU dest)

° Control design reduces to Microprogramming

- Part of the design process is to develop a "language" that describes control and is easy for humans to understand

Recap: Microprogramming



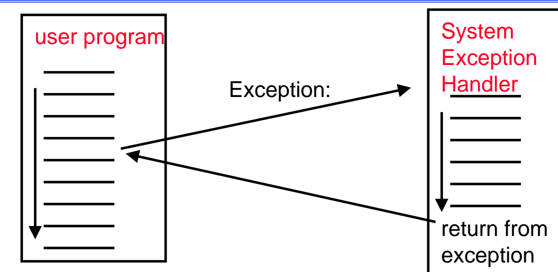
Decoders implement our µ-code language:

For instance:
rt-ALU
rd-ALU
mem-ALU

° Microprogramming is a fundamental concept

- implement an instruction set by building a very simple processor and <u>interpreting</u> the instructions
- essential for very complex instructions and when few register transfers are possible
- overkill when ISA matches datapath 1-1

Recap: 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"

Recap: 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
  - MIPS takes this position
- **Imprecise ⇒ system software has to figure out what is where and put it all back together**
  - Precise exceptions particularly important for virtual memory, since we may need to quickly schedule another user
  - Precise exceptions also important for frequent interrupts, where need low-overhead restart
- **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**

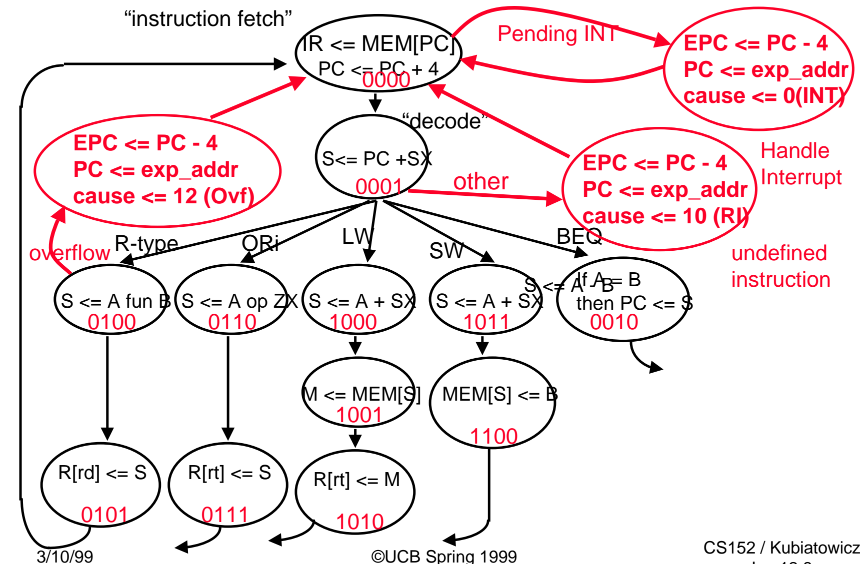
Recap: Two Types of Exceptions

- **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)
- **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

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

μ-offset
Do μ-branch
Cond Select
Seq Select
N?
pending interrupt
overflow
Mux
4?
1
Mux
0
1
Adder
microPC
Mux
2
1
0
0
Dispatch ROM
Opcode

EPC <= PC - 4
PC <= exp_addr
cause <= 12 (Ovf)

overflow
R-type
S <= A fun B
0100

μAddress Select Logic

Example: Can easily use with for non-ideal memory

IR <= MEM[PC] "instruction fetch"
~wait
wait
A <= R[rs]
B <= R[rt] "decode / operand fetch"

R-type
S <= A fun B
R[rd] <= S
PC <= PC + 4

ORi
S <= A or ZX
R[rt] <= S
PC <= PC + 4

LW
S <= A + SX
M <= MEM[S]
~wait
wait
R[rt] <= M
PC <= PC + 4

SW
S <= A + SX
MEM[S] <= B
~wait
wait
PC <= PC + 4

BEQ
PC <= Next(PC)

Execute
Memory
Write-back

Summary: Microprogramming one inspiration for RISC

° 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)

The Big Picture: Where are We Now?

° The Five Classic Components of a Computer

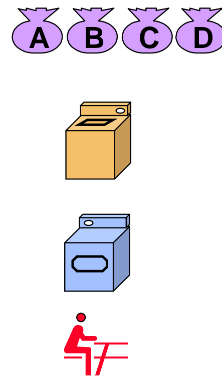
Processor
Control
Datapath
Memory
Input
Output

° Next Topics:

- Pipelining by Analogy
- Administrivia; Course road map

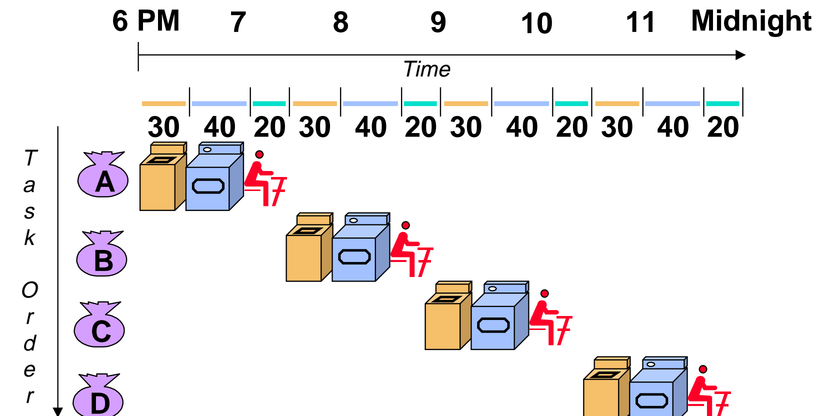
Pipelining is Natural!

- 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



©UCB Spring 1999

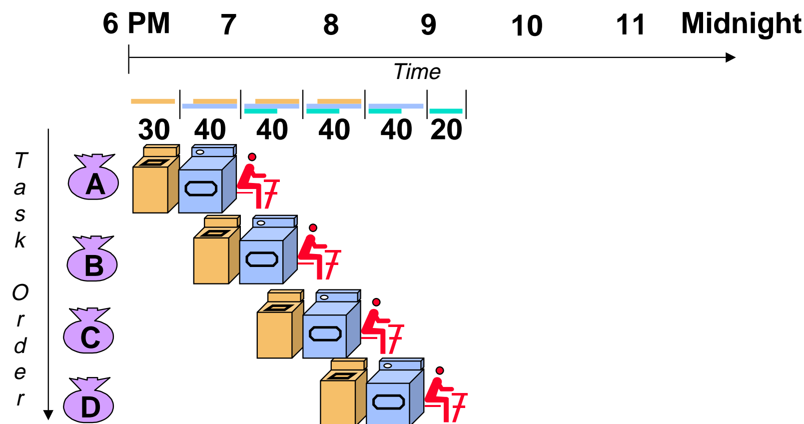
Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

©UCB Spring 1999

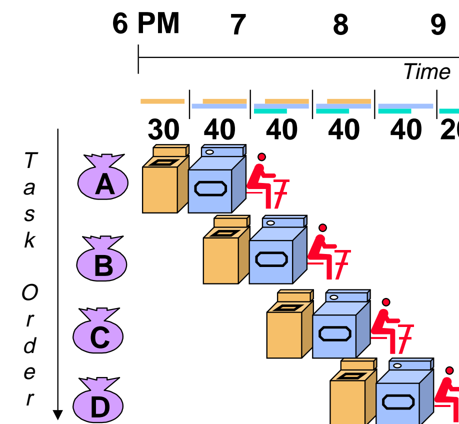
Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

©UCB Spring 1999

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

©UCB Spring 1999

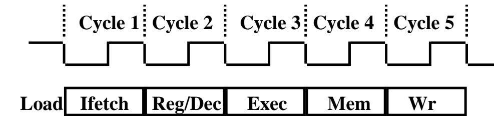
Administrative Issues

° **Computers in the news:**
- **Intel decided to settle out of court**
- **FTC had much narrower case with Intel than with Microsoft**
- **Intel choose quieter solution still stinging from bad publicity due to Divide bug.**

° **Moving on to Chapter 6**

° **Next week⇒sections in Cory Lab**

The Five Stages of Load



° **Ifetch: Instruction Fetch**
- **Fetch the instruction from the Instruction Memory**

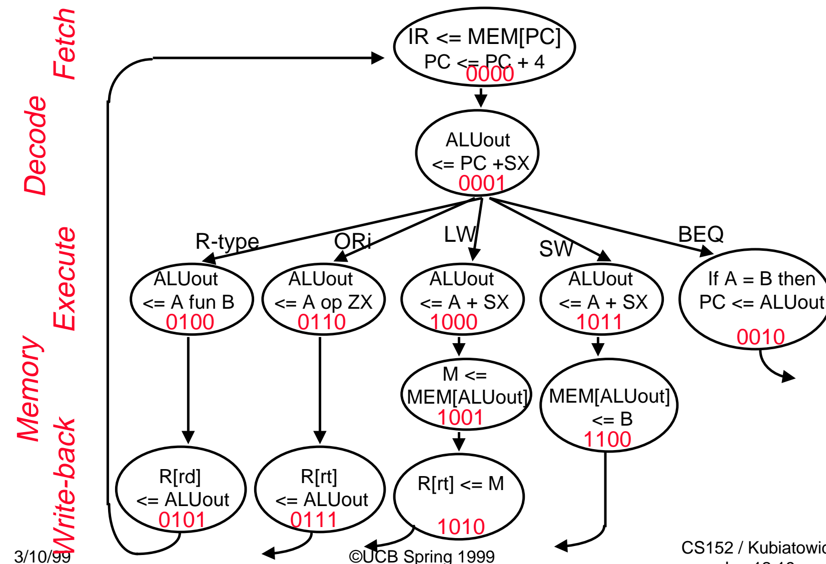
° **Reg/Dec: Registers Fetch and Instruction Decode**

° **Exec: Calculate the memory address**

° **Mem: Read the data from the Data Memory**

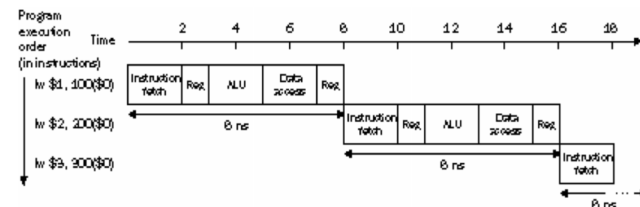
° **Wr: Write the data back to the register file**

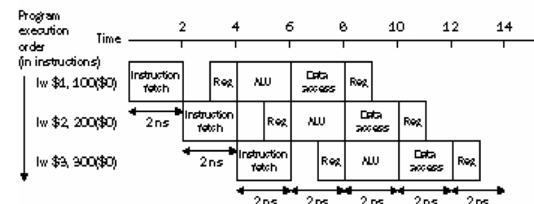
Note: These 5 stages were there all along!



Pipelining

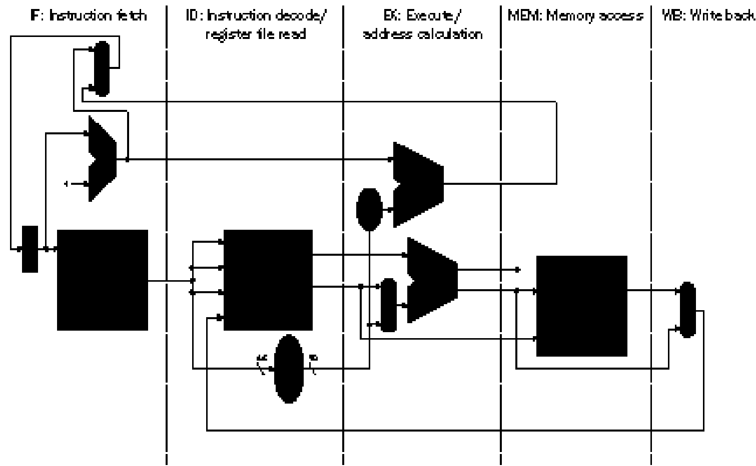
° **Improve performance by increasing throughput**





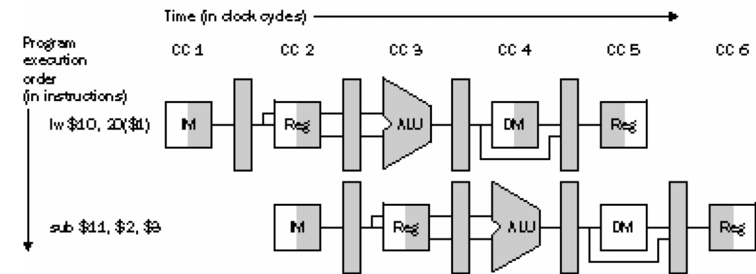
Ideal speedup is number of stages in the pipeline. Do we achieve this?

Basic Idea



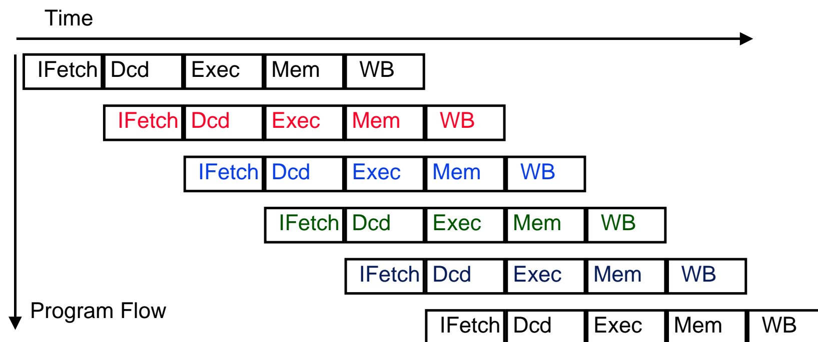
° *What do we need to add to split the datapath into stages?*

Graphically Representing Pipelines

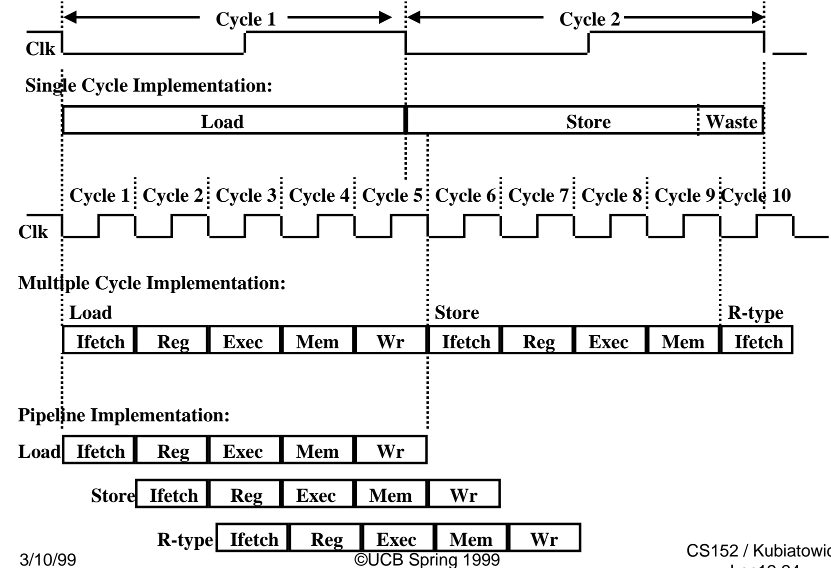


° **Can help with answering questions like:**
- **how many cycles does it take to execute this code?**
- **what is the ALU doing during cycle 4?**
- **use this representation to help understand datapaths**

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

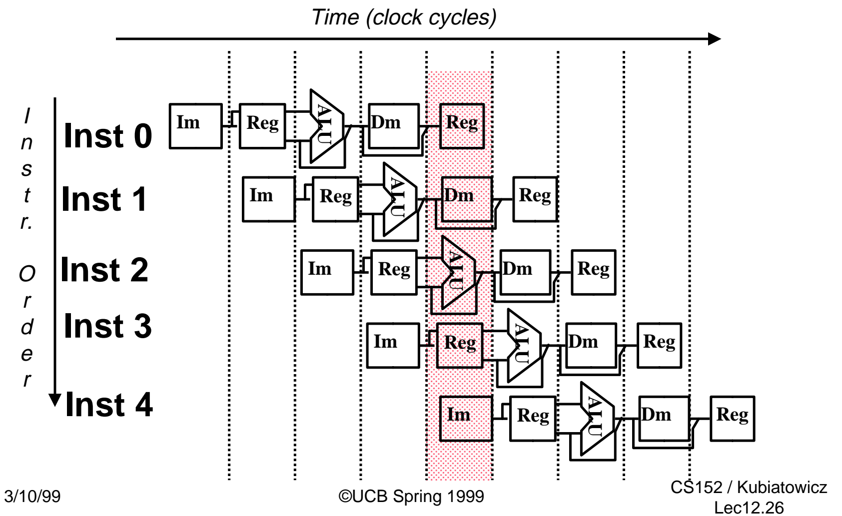
° Multicycle Machine

- 10 ns/cycle x 4.6 CPI (due to inst mix) x 100 inst = 4600 ns

° Ideal pipelined machine

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

Why Pipeline? Because the resources are there!



Can pipelining get us into trouble?

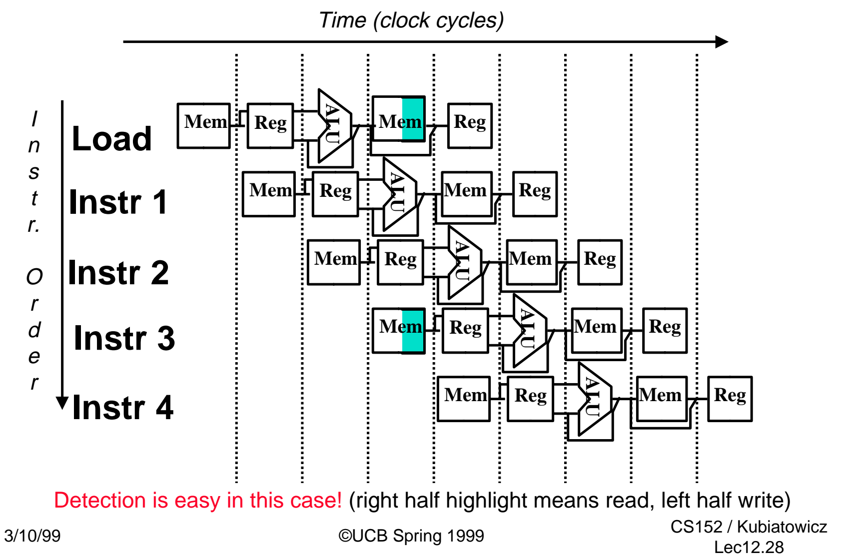
° Yes: Pipeline Hazards

- 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)
- 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
- control hazards: attempt to make a decision before condition is evaulated
  - E.g., washing football uniforms and need to get proper detergent level; need to see after dryer before next load in
  - branch instructions

° 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)

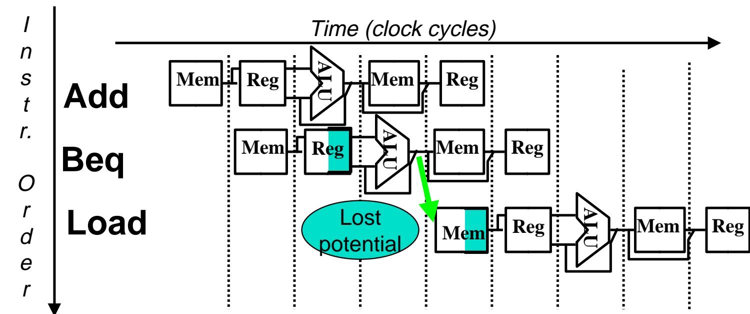
Structural Hazards limit performance

° 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 Solutions

° Stall: wait until decision is clear



° Impact: 2 lost cycles (i.e. 3 clock cycles per branch instruction) => slow

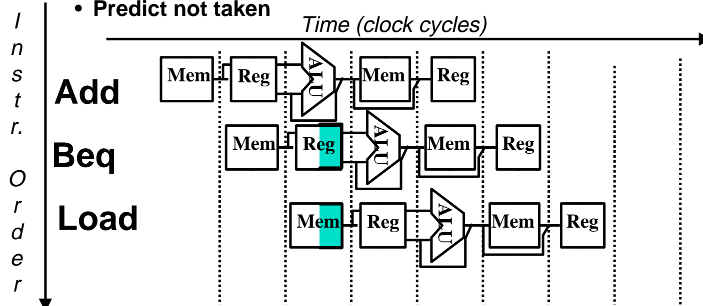
° Move decision to end of decode

- save 1 cycle per branch

Control Hazard Solutions

° Predict: guess one direction then back up if wrong

- Predict not taken



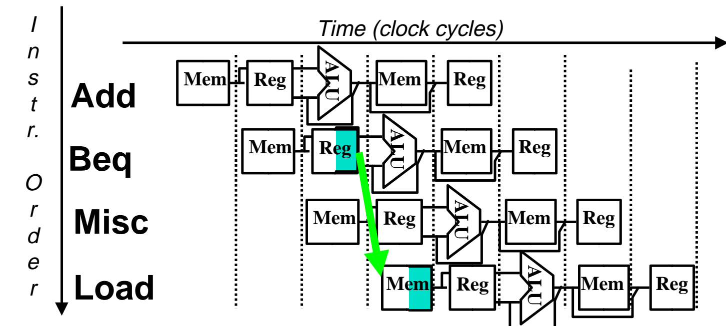
° Impact: 0 lost cycles per branch instruction if right, 1 if wrong (right - 50% of time)

- 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 Solutions

° Redefine branch behavior (takes place after next instruction) "delayed branch"



° 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

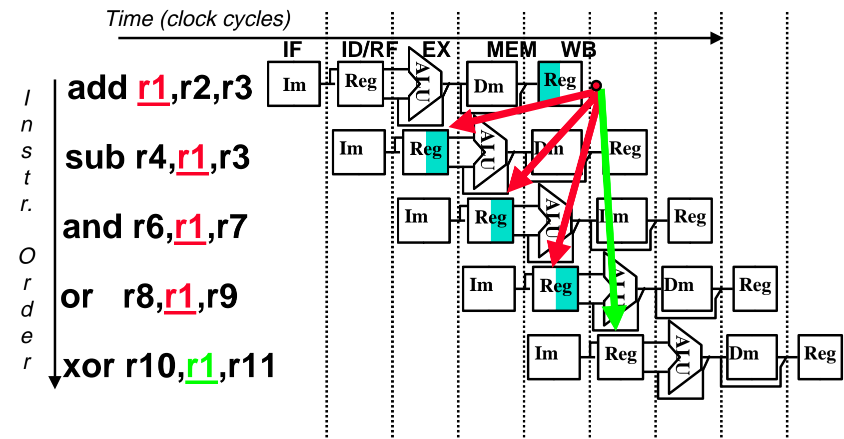
and r6, r1 ,r7

or r8, r1 ,r9

xor r10, r1 ,r11

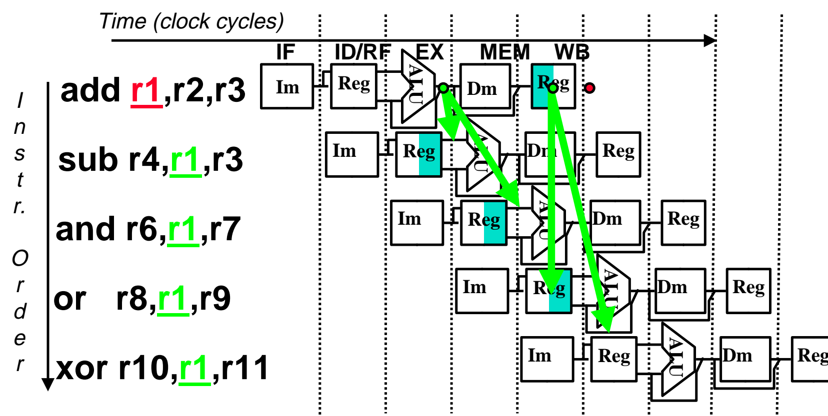
Data Hazard on r1:

- Dependencies backwards in time are hazards



Data Hazard Solution:

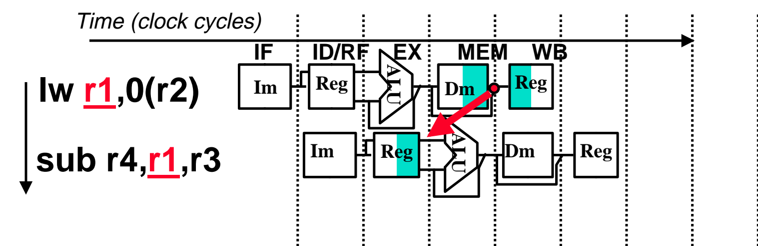
- "Forward" result from one stage to another



- "or" OK if define read/write properly

Forwarding (or Bypassing): What about Loads

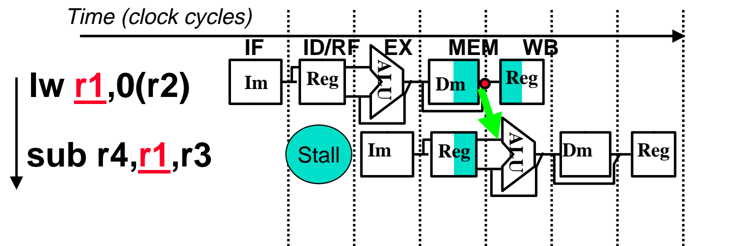
- Dependencies backwards in time are hazards



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

Forwarding (or Bypassing): What about Loads

- Dependencies backwards in time are hazards

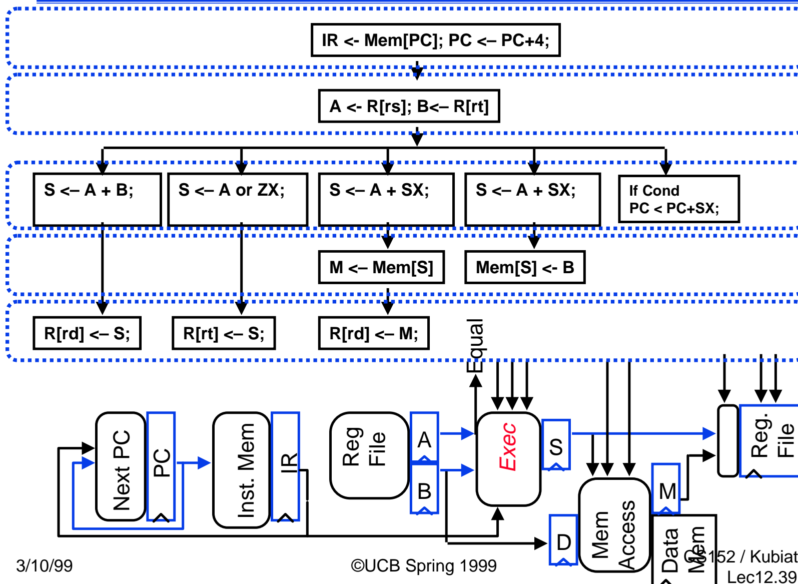


- Can't solve with forwarding:
- Must delay/stall instruction dependent on loads

Designing a Pipelined Processor

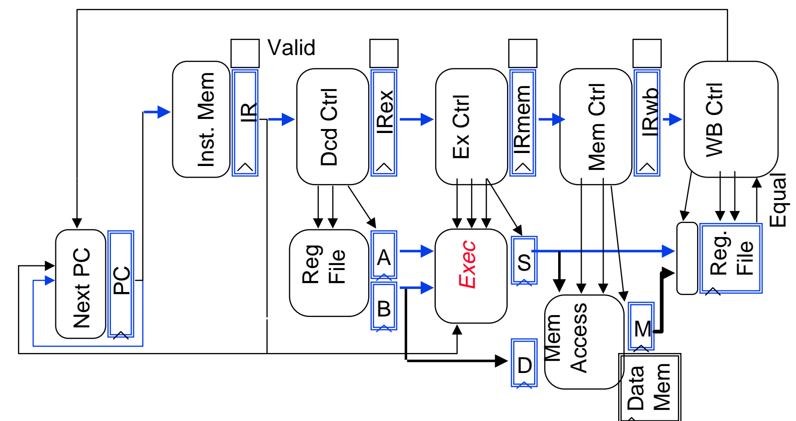
- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

Control and Datapath: Split state diag into 5 pieces

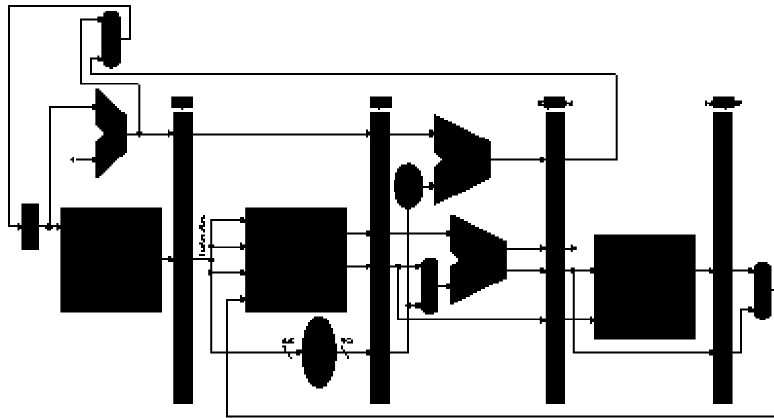


Pipelined Processor (almost) for slides

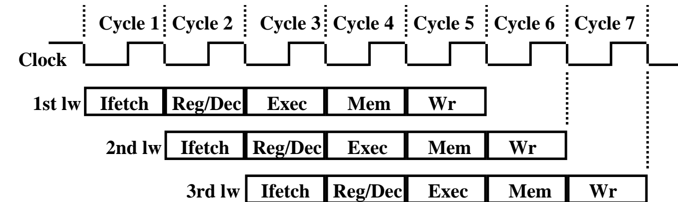
- What happens if we start a new instruction every cycle?



Pipelined Datapath (as in book); hard to read

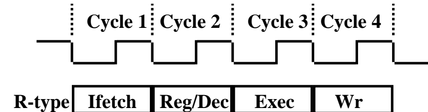


Pipelining the Load Instruction



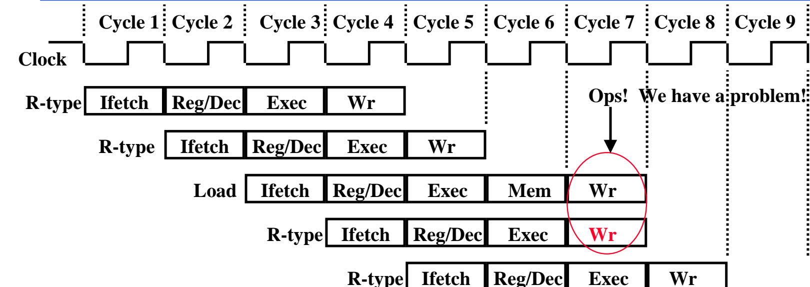
- The five independent functional units in the pipeline datapath are:
  - Instruction Memory for the Ifetch stage
  - Register File's Read ports (bus A and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the Mem stage
  - Register File's Write port (bus W) for the Wr stage

The Four Stages of R-type



- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec:
  - ALU operates on the two register operands
  - Update PC
- Wr: Write the ALU output back to the register file

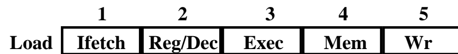
Pipelining the R-type and Load Instruction



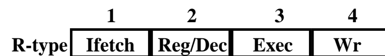
- We have pipeline conflict or structural hazard:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Important Observation

- Each functional unit can only be used **once** per instruction
- Each functional unit must be used at the **same** stage for all instructions:
  - Load uses Register File's Write Port during its 5th stage

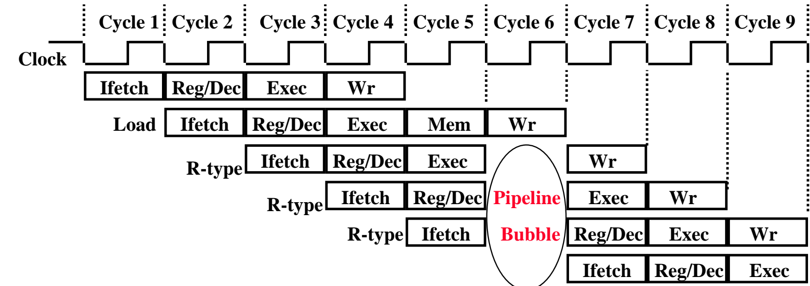


  - R-type uses Register File's Write Port during its 4th stage



- 2 ways to solve this pipeline hazard.

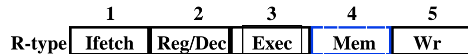
Solution 1: Insert "Bubble" into the Pipeline

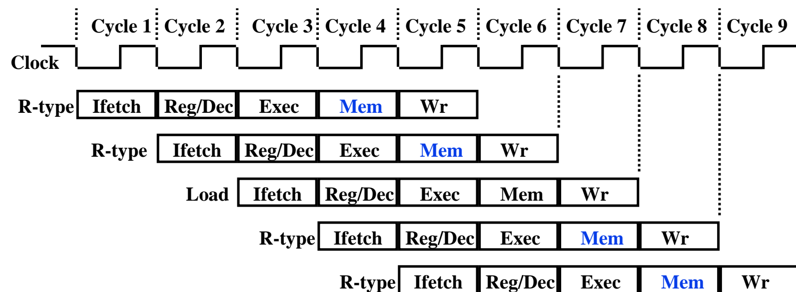


- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

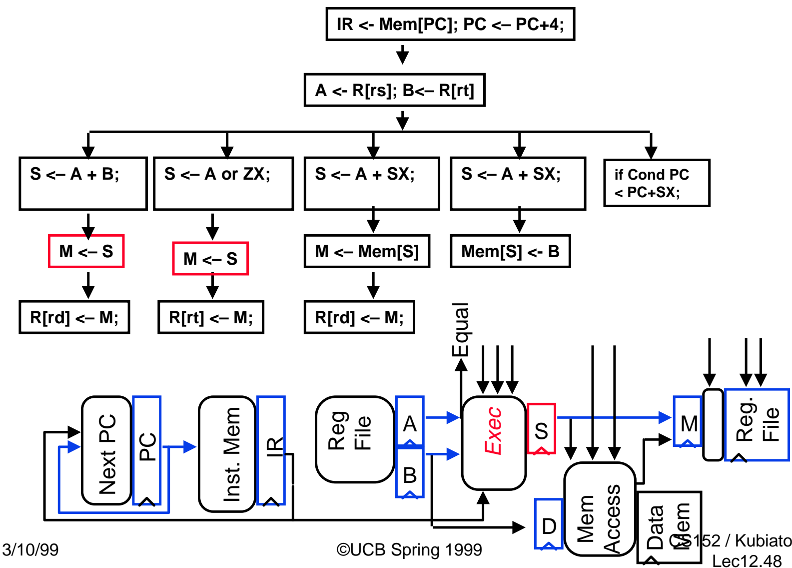
Solution 2: Delay R-type's Write by One Cycle

- Delay R-type's register write by one cycle:
  - Now R-type instructions also use Reg File's write port at Stage 5
  - Mem stage is a NOOP stage: nothing is being done.

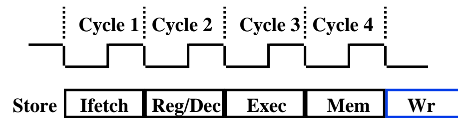




Modified Control & Datapath

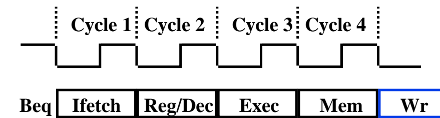


The Four Stages of Store



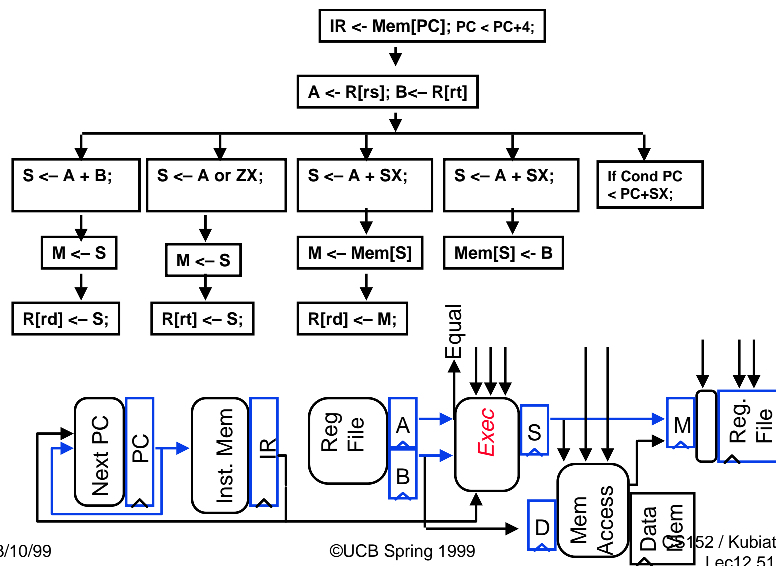
- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec: Registers Fetch and Instruction Decode
- Exec: Calculate the memory address
- Mem: Write the data into the Data Memory

The Three Stages of Beq

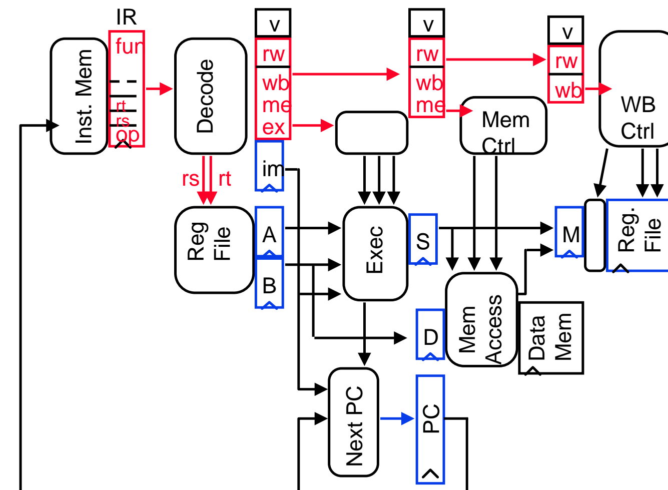


- Ifetch: Instruction Fetch
  - Fetch the instruction from the Instruction Memory
- Reg/Dec:
  - Registers Fetch and Instruction Decode
- Exec:
  - compares the two register operand,
  - select correct branch target address
  - latch into PC

Control Diagram



Datapath + Data Stationary Control

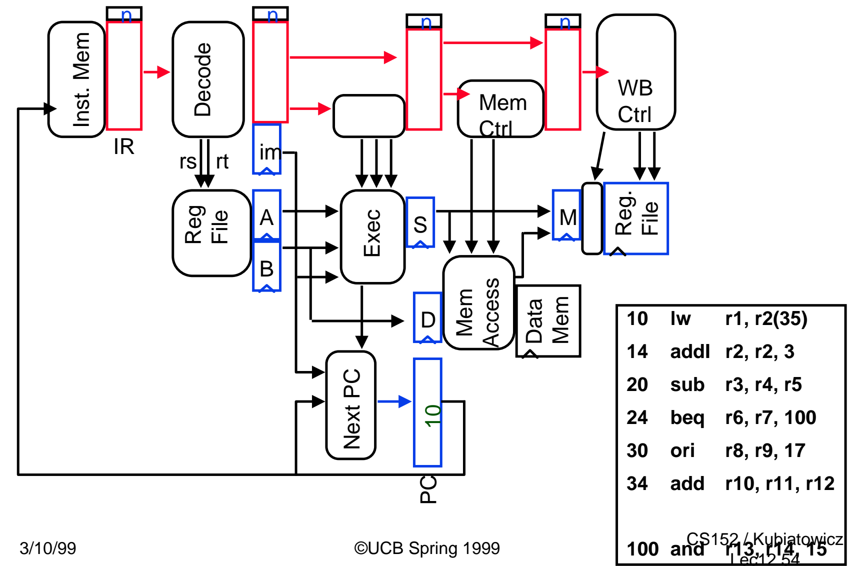


Let's Try it Out

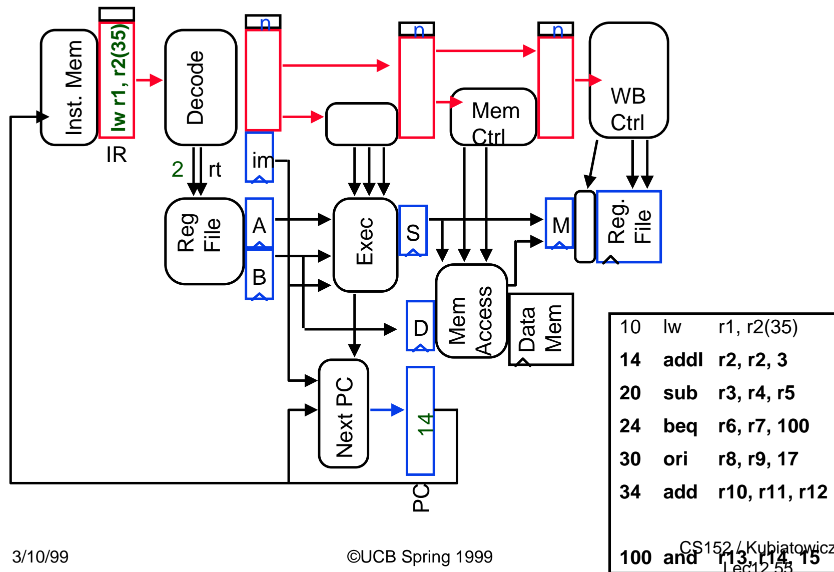
10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

these addresses are octal

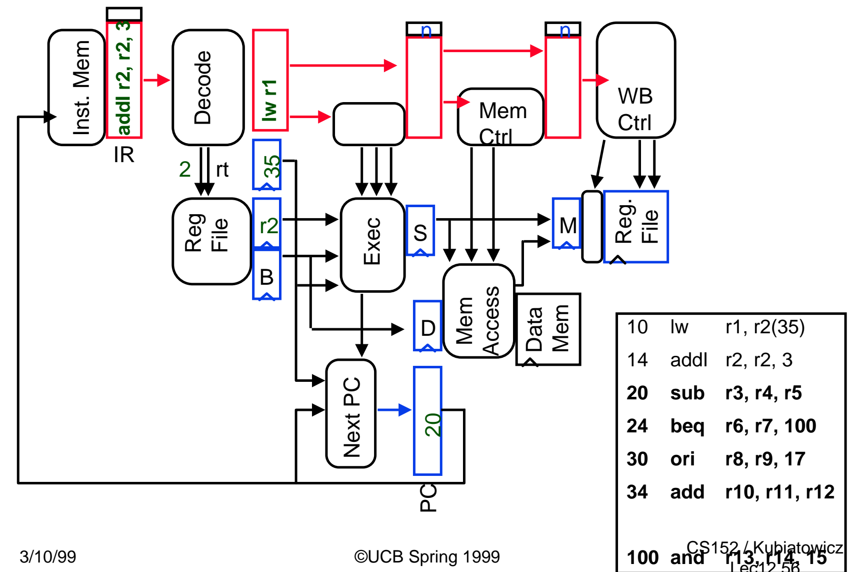
Start: Fetch 10



Fetch 14, Decode 10



Fetch 20, Decode 14, Exec 10



Fetch 24, Decode 20, Exec 14, Mem 10

Inst. Mem | IR: sub r3, r4, r5 | Decode: 4, 5 | addl r2, r2, 3 | Reg File: r2, B, 3 | Exec | lw r1 | r2+35 | Mem Ctrl | Mem Access | D | Data Mem | M | WB Ctrl | Reg. File | Next PC | PC: 24

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	**beq**	**r6, r7, 100**
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 30, Dcd 24, Ex 20, Mem 14, WB 10

Inst. Mem | IR: beq r6, r7 100 | Decode: 6, 7 | sub r3 | Reg File: r4, r5 | Exec | addI r2 | r2+3 | Mem Ctrl | Mem Access | D | Data Mem | lw r1 | M[r2+35] | WB Ctrl | Reg. File | Next PC | PC: 30

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	**ori**	**r8, r9, 17**
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 34, Dcd 30, Ex 24, Mem 20, WB 14

Inst. Mem | IR: ori r8, r9 17 | Decode: 9, xx | beq | 100 | Reg File: r6, r7 | Exec | sub r3 | r4-r5 | Mem Ctrl | Mem Access | D | Data Mem | addI r2 | r2+3 | WB Ctrl | Reg. File | Next PC | PC: 34

r1=M[r2+35]

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	**add**	**r10, r11, r12**
100	**and**	**r13, r14, 15**

Fetch 100, Dcd 34, Ex 30, Mem 24, WB 20

Inst. Mem | IR: add r10, r11, r12 | Decode: 11, 12 | ori r8 | 17 | Reg File: r9, x | Exec | beq | xxx | Mem Ctrl | Mem Access | D | Data Mem | sub r3 | r4-r5 | WB Ctrl | Reg. File | Next PC | PC: 100

r1=M[r2+35]
r2 = r2+3

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	**and**	**r13, r14, 15**

ooops, we should have only one delayed instruction

Fetch 104, Dcd 100, Ex 34, Mem 30, WB 24

r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Squash the extra instruction in the branch shadow!

Fetch 108, Dcd 104, Ex 100, Mem 34, WB 30

r1=M[r2+35]
r2 = r2+3
r3 = r4-r5

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Squash the extra instruction in the branch shadow!

Fetch 114, Dcd 110, Ex 104, Mem 100, WB 34

NO WB
NO Ovflow

r1=M[r2+35]
r2 = r2+3
r3 = r4-r5
r8 = r9 | 17

10	lw	r1, r2(35)
14	addI	r2, r2, 3
20	sub	r3, r4, r5
24	beq	r6, r7, 100
30	ori	r8, r9, 17
34	add	r10, r11, r12
100	and	r13, r14, 15

Squash the extra instruction in the branch shadow!

Summary: Pipelining

- **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**
- **We'll build a simple pipeline and look at these issues**
- **We'll talk about modern processors and what really makes it hard:**
  - **exception handling**
  - **trying to improve performance with out-of-order execution, etc.**

Summary

° **Pipelining is a fundamental concept**
  • multiple steps using distinct resources

° **Utilize capabilities of the Datapath by pipelined instruction processing**
  • start next instruction while working on the current one
  • limited by length of longest stage (plus fill/flush)
  • detect and resolve hazards