

**CS152**  
**Computer Architecture and Engineering**  
**Lecture 13**

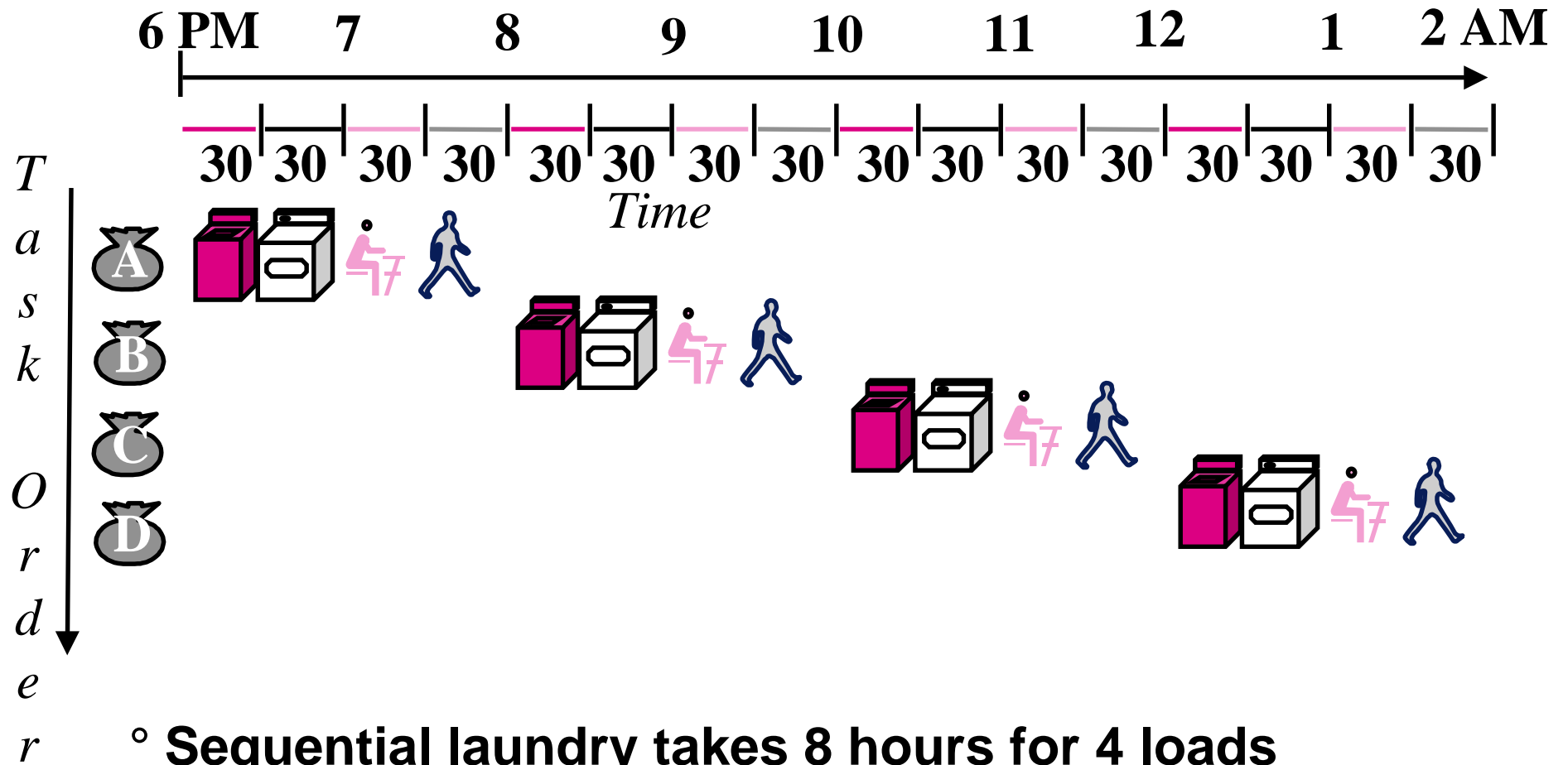
**Introduction to Pipelining II:**  
**Control**

**March 15, 1999**

**John Kubiatoicz (<http://cs.berkeley.edu/~kubitron>)**

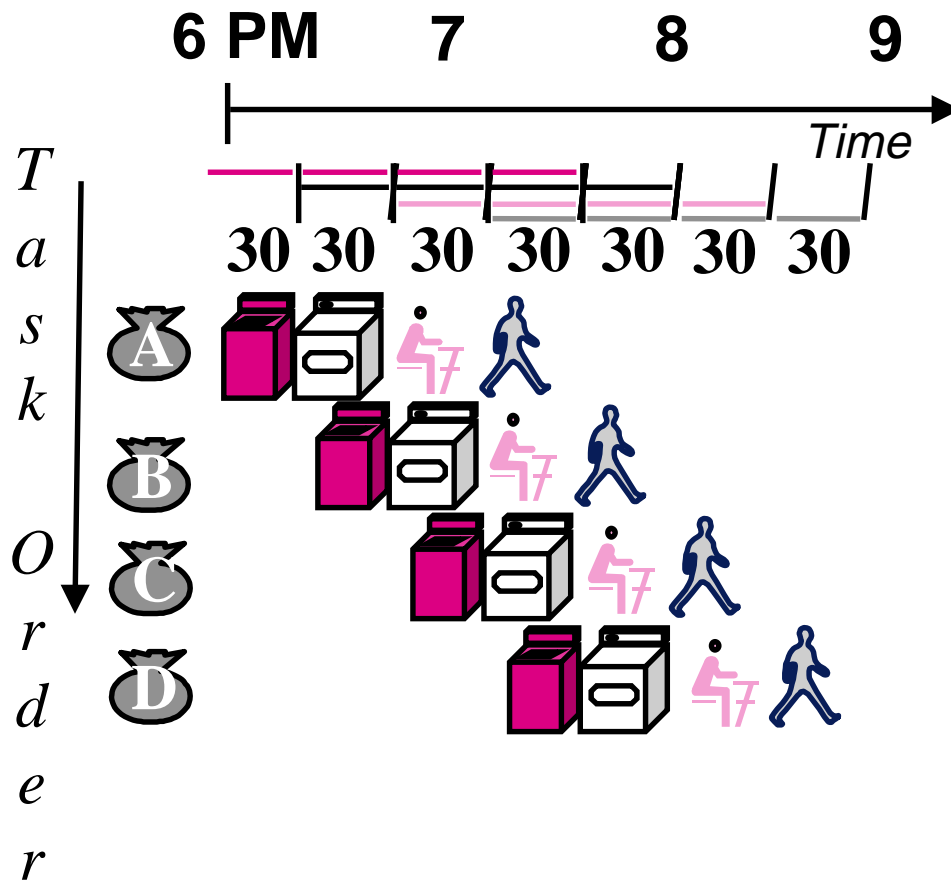
**lecture slides: <http://www-inst.eecs.berkeley.edu/~cs152/>**

# Recap: Sequential Laundry



- Sequential laundry takes 8 hours for 4 loads
- If they learned pipelining, how long would laundry take?

## Recap: Pipelining Lessons (its intuitive!)

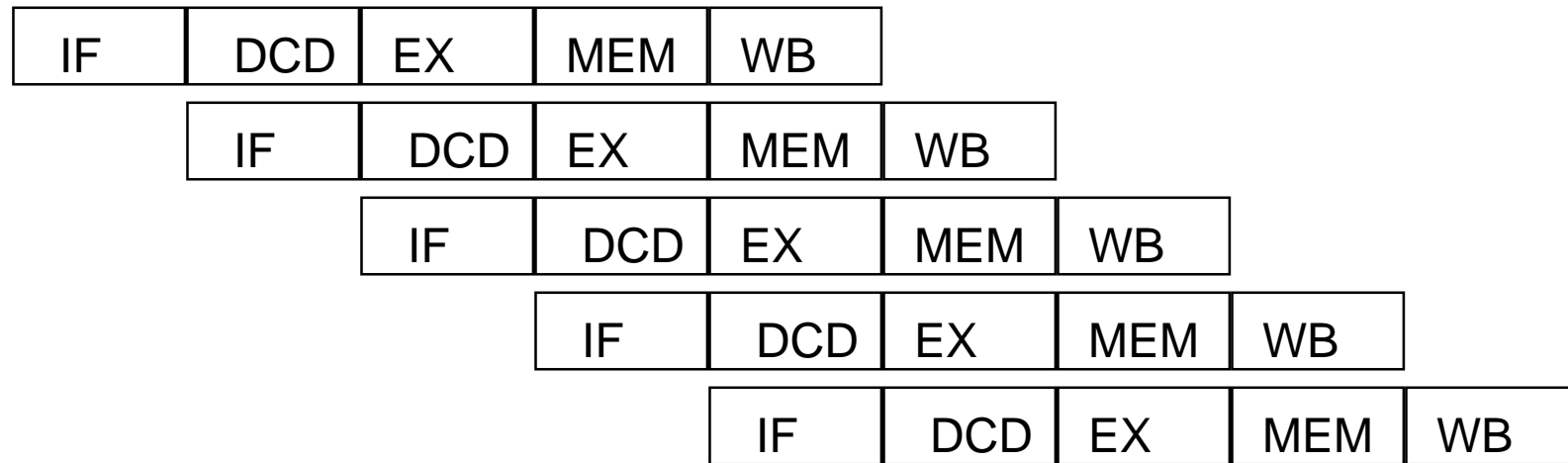


- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages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

# Recap: Ideal Pipelining

---

**Assume instructions are completely independent!**



**Maximum Speedup  $\leq$  Number of stages**

**Speedup  $\leq \frac{\text{Time for unpipelined operation}}{\text{Time for longest stage}}$**

Example: 40ns data path, 5 stages, Longest stage is 10 ns, Speedup  $\leq 4$

## Recap: Can pipelining get us into trouble?

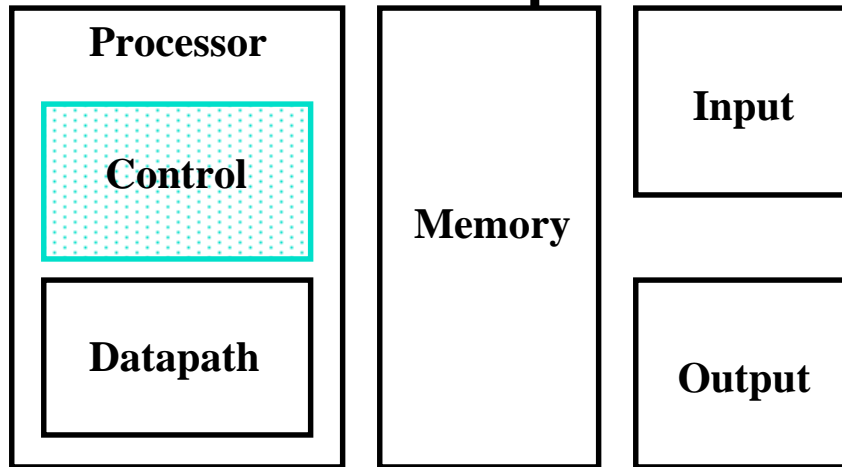
### Yes: Pipeline Hazards

- **structural hazards**: attempt to use the same resource two different ways at the same time
  - e.g., multiple memory accesses, multiple register writes
  - solutions: multiple memories, stretch pipeline
- **control hazards**: attempt to make a decision before condition is evaluated
  - e.g., any conditional branch
  - solutions: prediction, delayed branch
- **data hazards**: attempt to use item before it is ready
  - e.g., add r1,r2,r3; sub r4, r1 ,r5; lw r6, 0(r7); or r8, r6 ,r9
  - solutions: forwarding/bypassing, stall/bubble

# The Big Picture: Where are We Now?

---

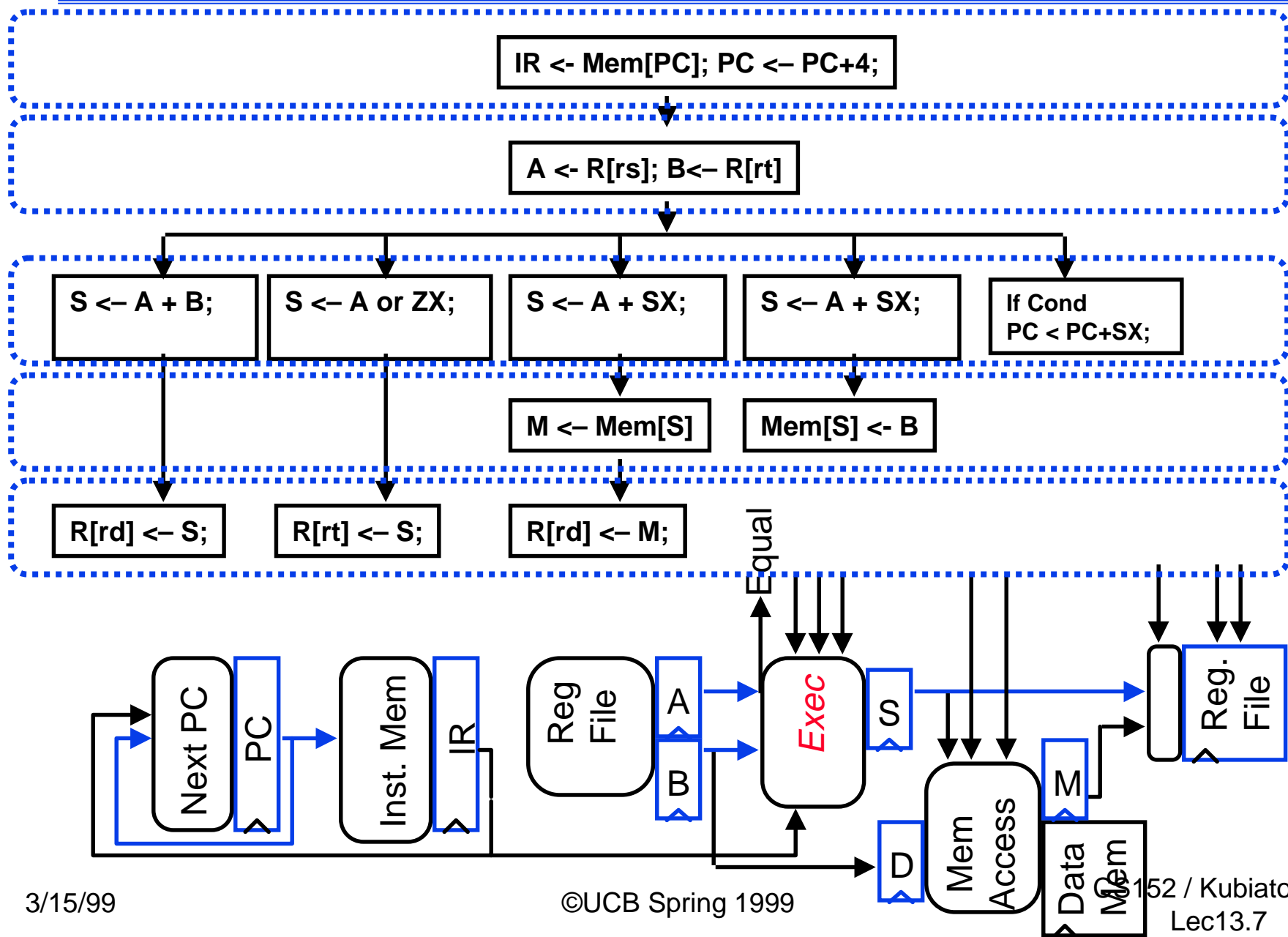
## ◦ The Five Classic Components of a Computer



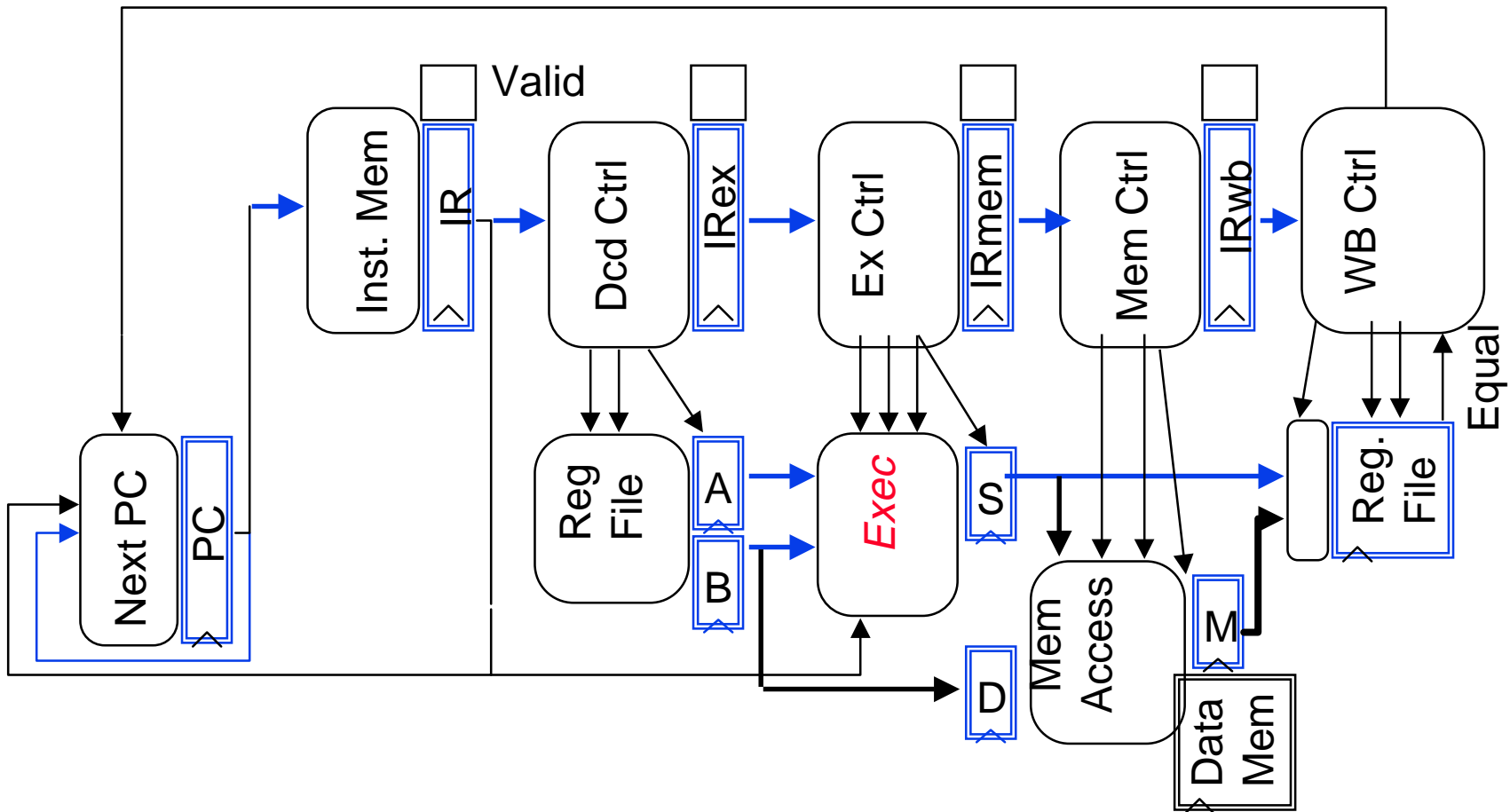
## ◦ Today's Topics:

- Recap last lecture
- Pipelined Control/ Do it yourself Pipelined Control
- Administrivia
- Hazards/Forwarding
- Exceptions
- Review MIPS R3000 pipeline
- Advanced Pipelining?

# Control and Datapath: Split state diag into 5 pieces

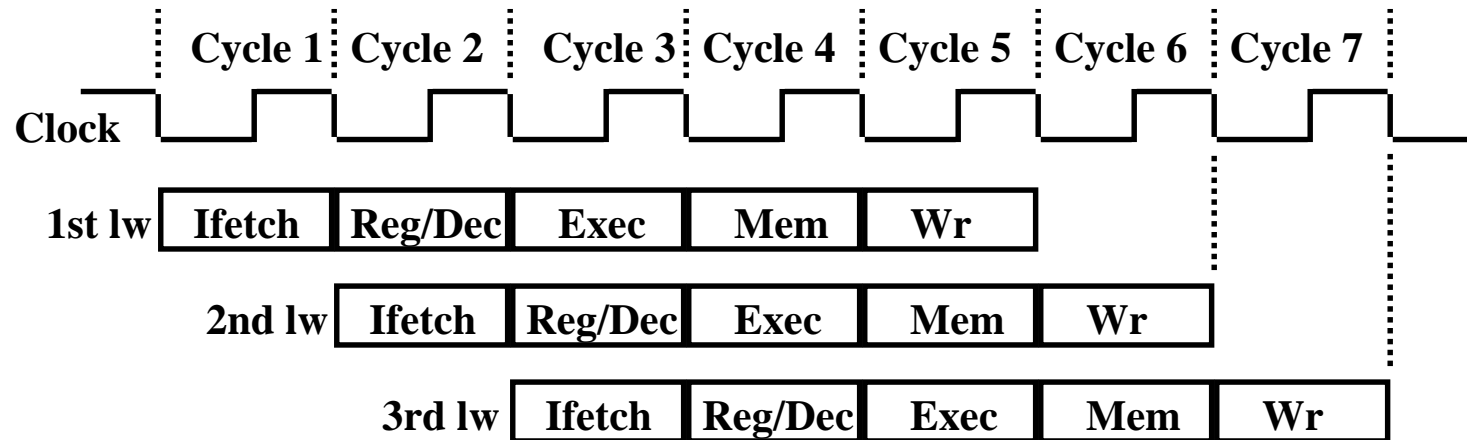


# Pipelined Processor (almost) for slides



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

# Pipelining the Load Instruction

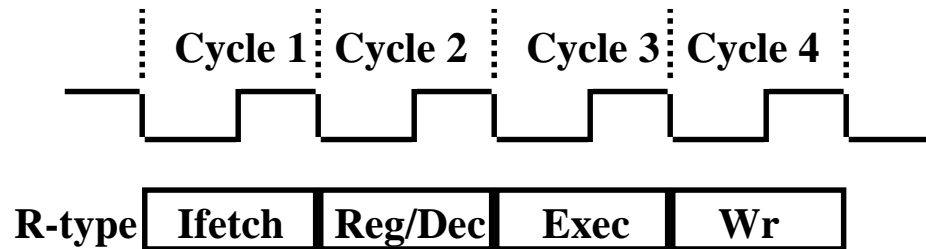


◦ The five independent functional units in the pipeline datapath are:

- 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

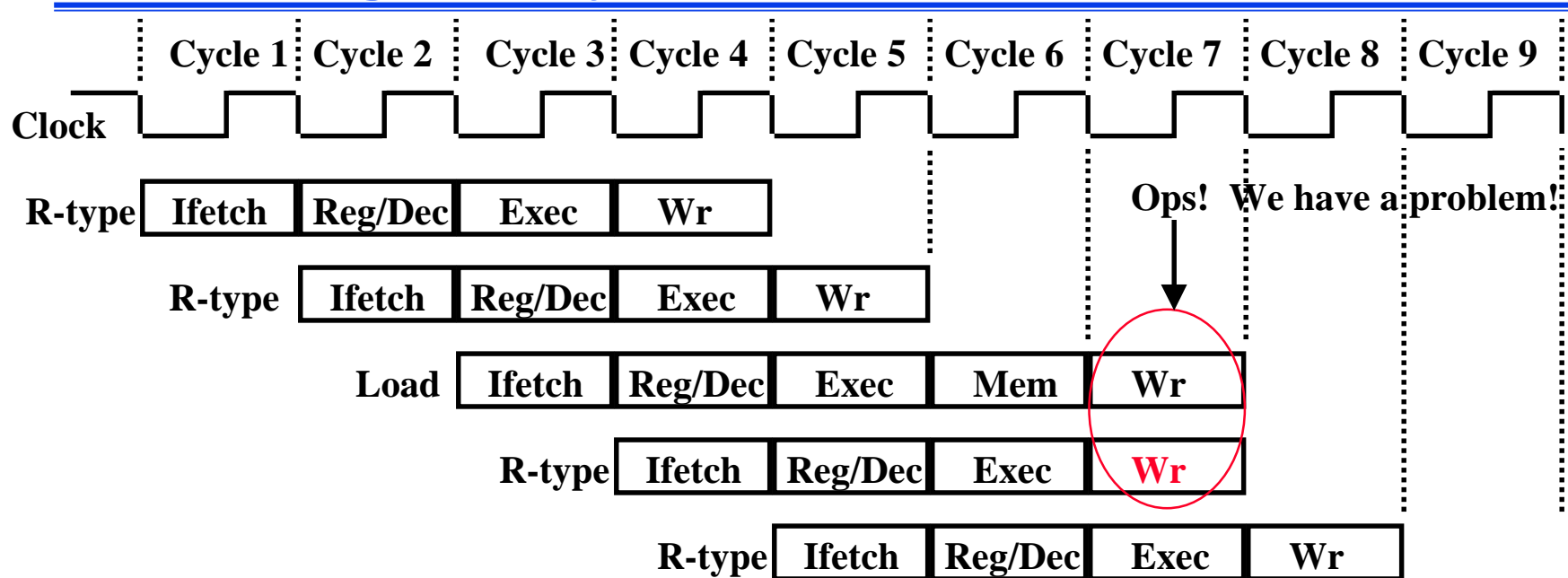
# The Four Stages of R-type

---



- **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

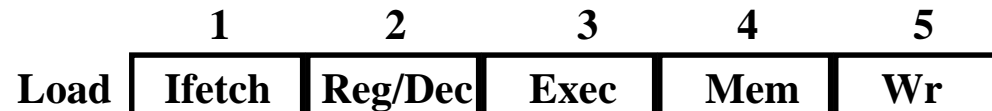


- **We have pipeline conflict or structural hazard:**
  - Two instructions try to write to the register file at the same time!
  - Only one write port

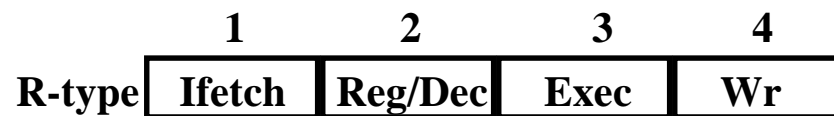
## Important Observation

---

- 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

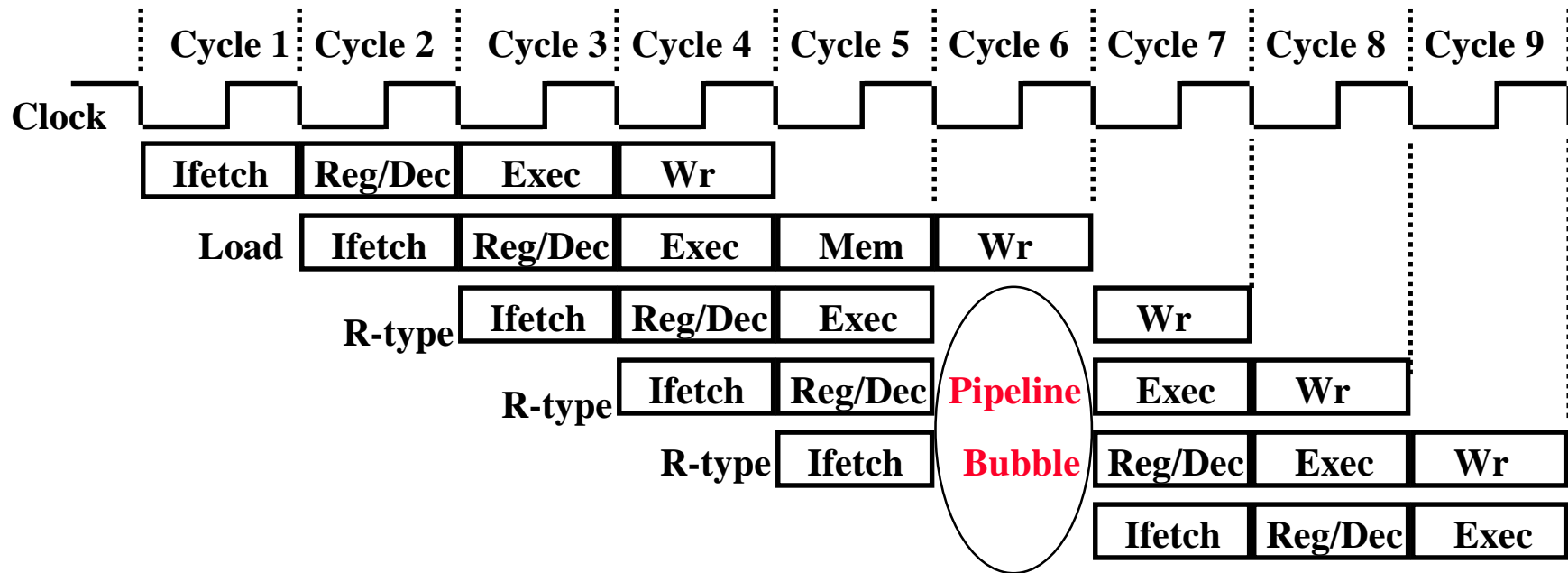


- R-type uses Register File's Write Port during its **4th** stage



- 2 ways to solve this pipeline hazard.

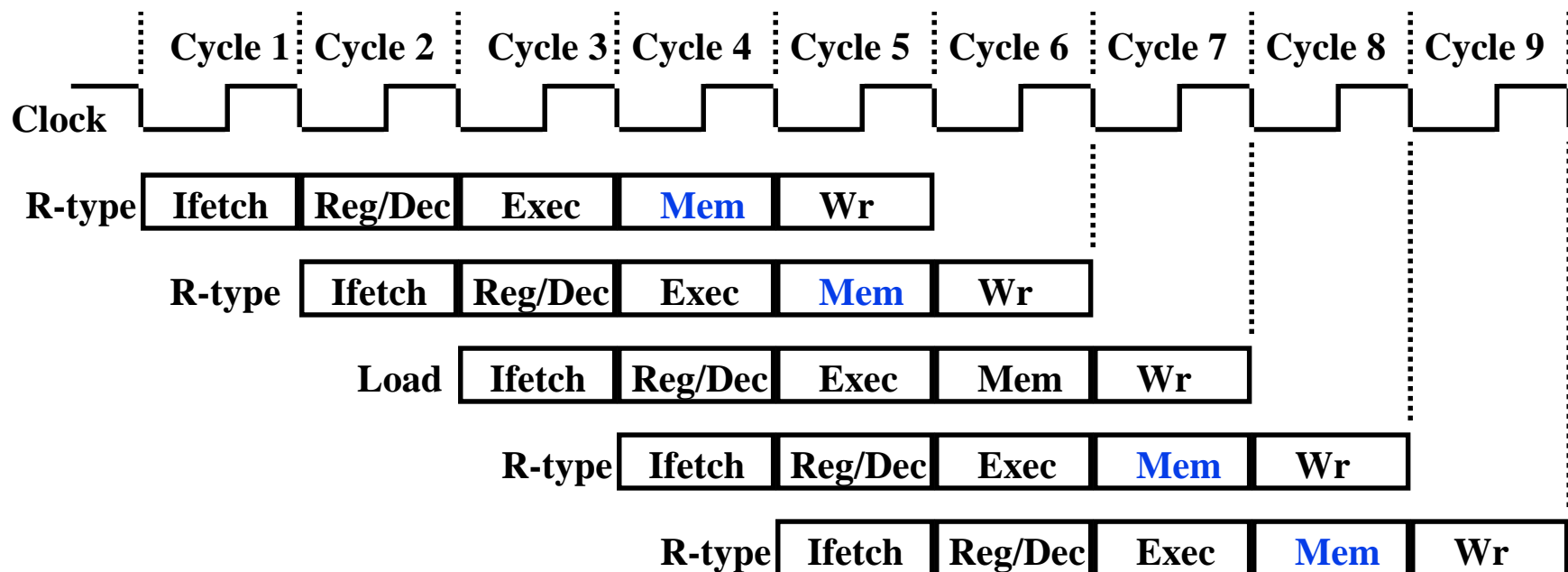
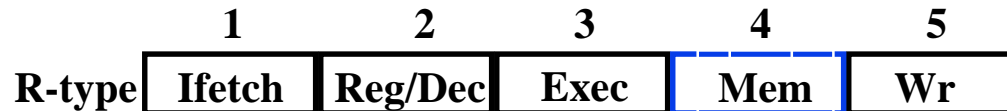
## Solution 1: Insert “Bubble” into the Pipeline



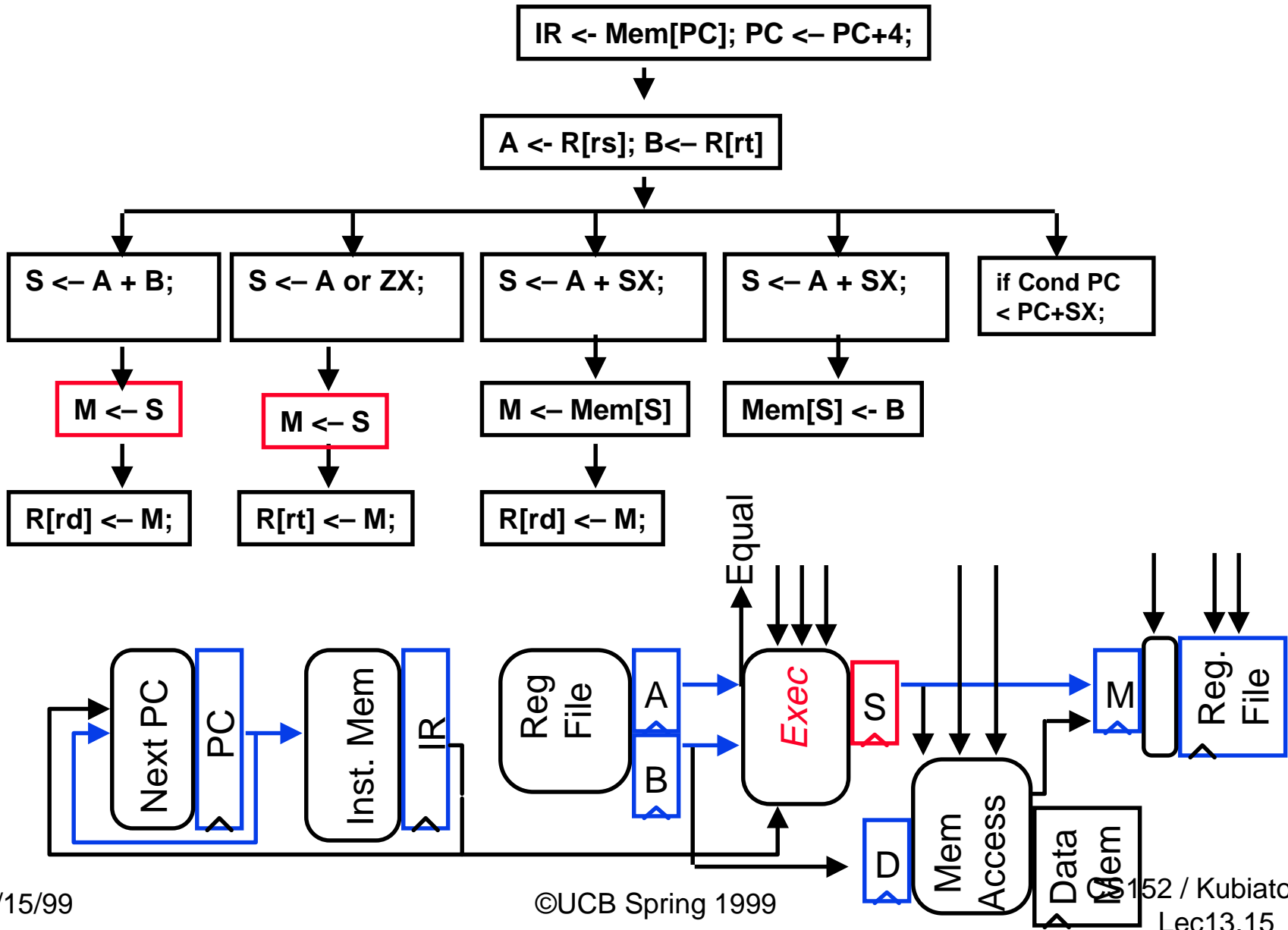
- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

## 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.

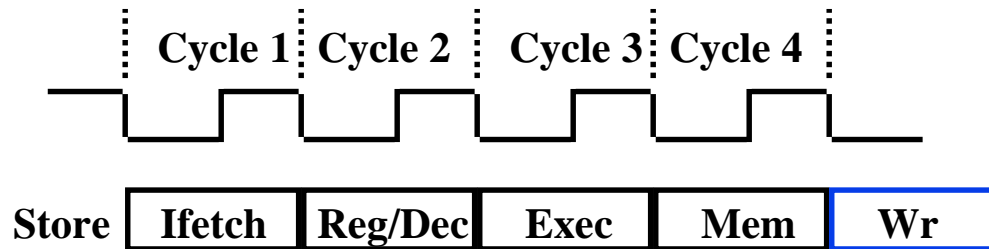


# Modified Control & Datapath



# The Four Stages of Store

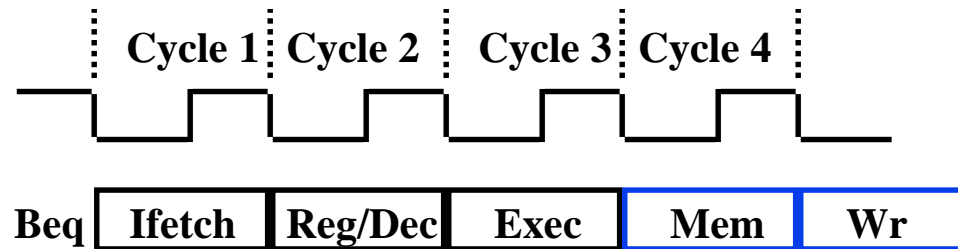
---



- **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

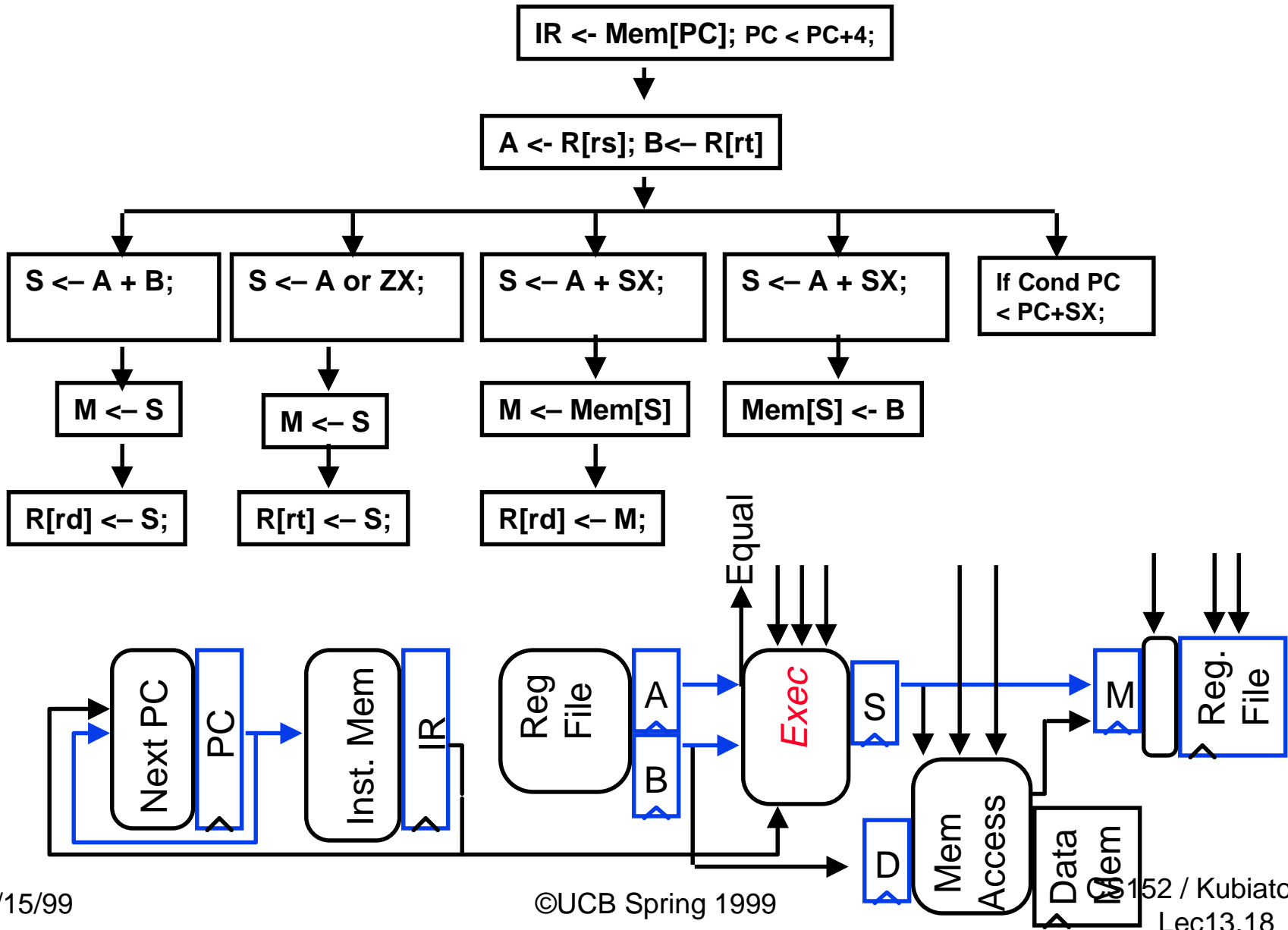
## ◦ **Reg/Dec:**

- Registers Fetch and Instruction Decode

## ◦ **Exec:**

- compares the two register operand,
- select correct branch target address
- latch into PC

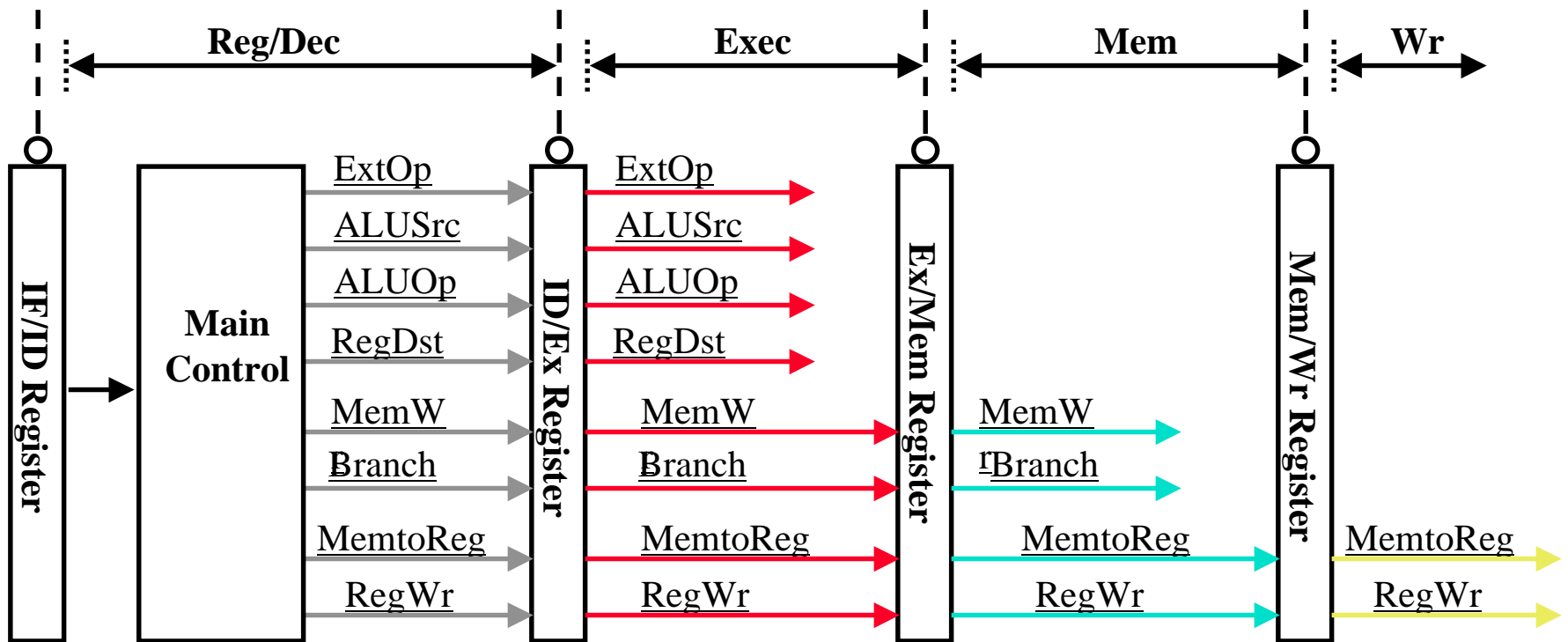
# Control Diagram



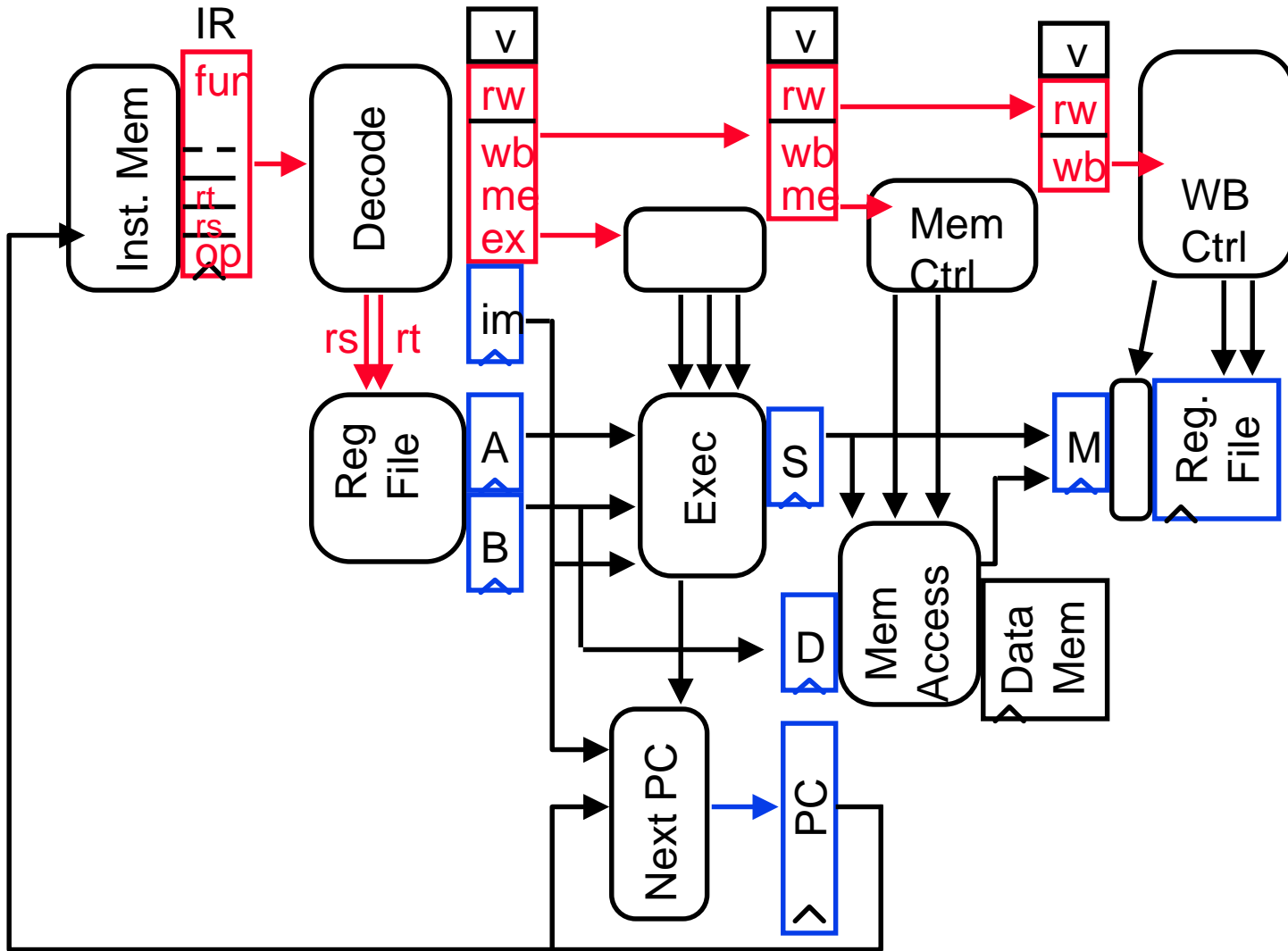
# Data Stationary Control

## ◦ The Main Control generates the control signals during Reg/Dec

- Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later
- Control signals for Mem (MemWr Branch) are used 2 cycles later
- Control signals for Wr (MemtoReg MemWr) are used 3 cycles later



# Datapath + Data Stationary Control



## Administrivia

---

- **Get started on LAB 5!**
  - Problem 0 due tonight at 12 Midnight via email: evaluate your teammates.
  - Organization on Lab due by Wednesday.
- **Starting tomorrow: Sections in Cory lab.  
Tomorrow: run “mystery program” on Lab 4.**
- **Generally positive feedback about course.**

## Let's Try it Out

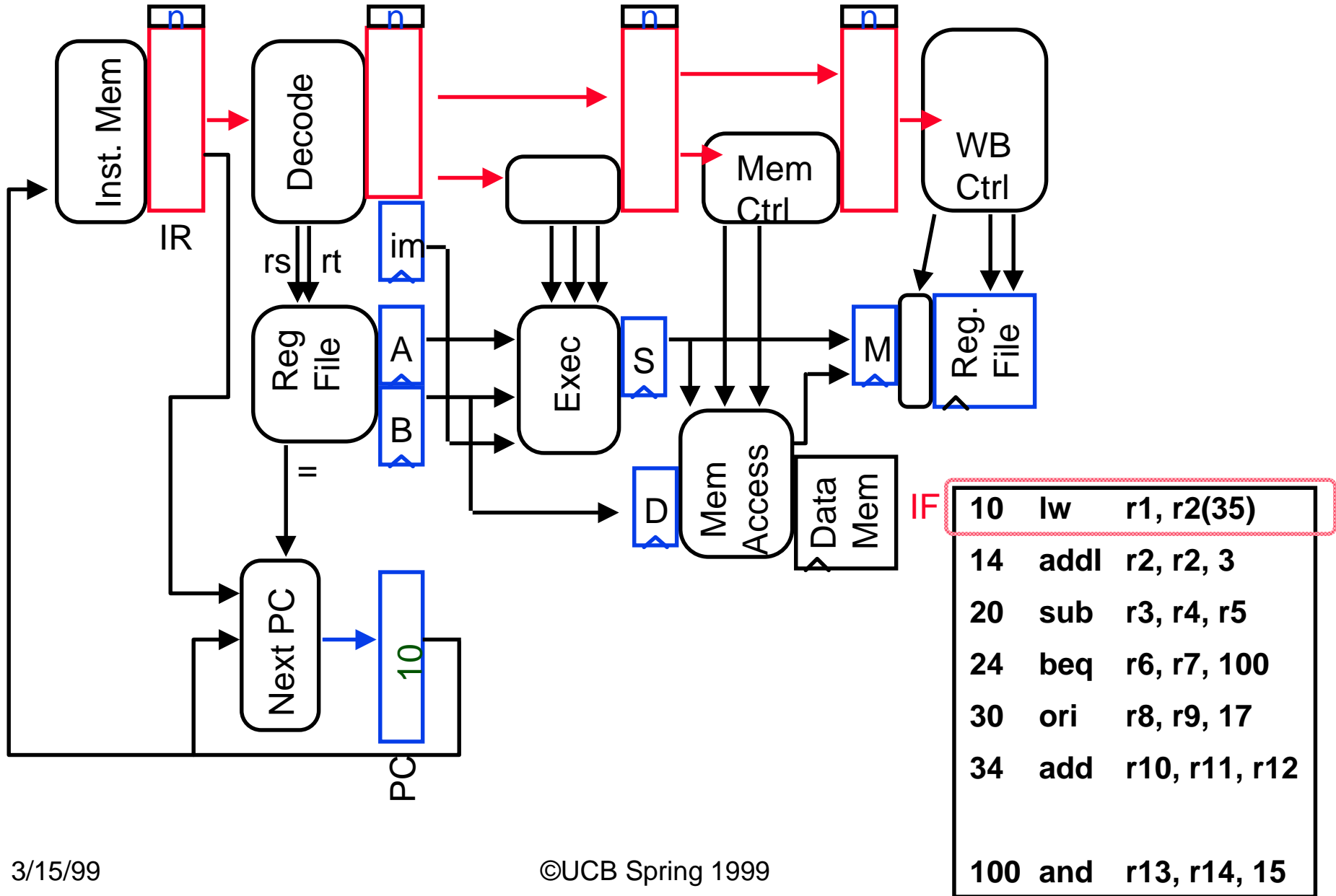
---

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

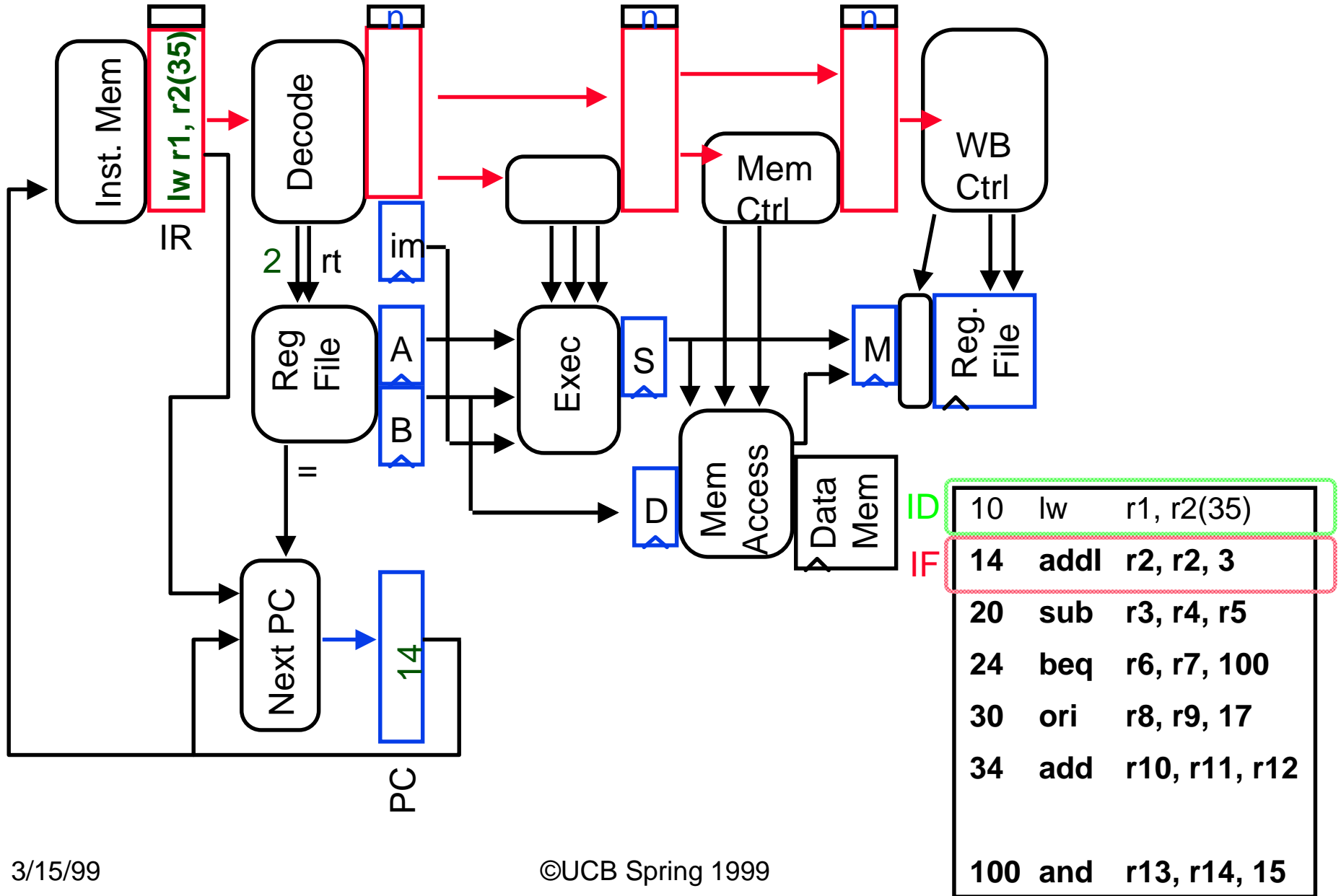
100   and   r13, r14, 15
```

these addresses are octal

# Start: Fetch 10

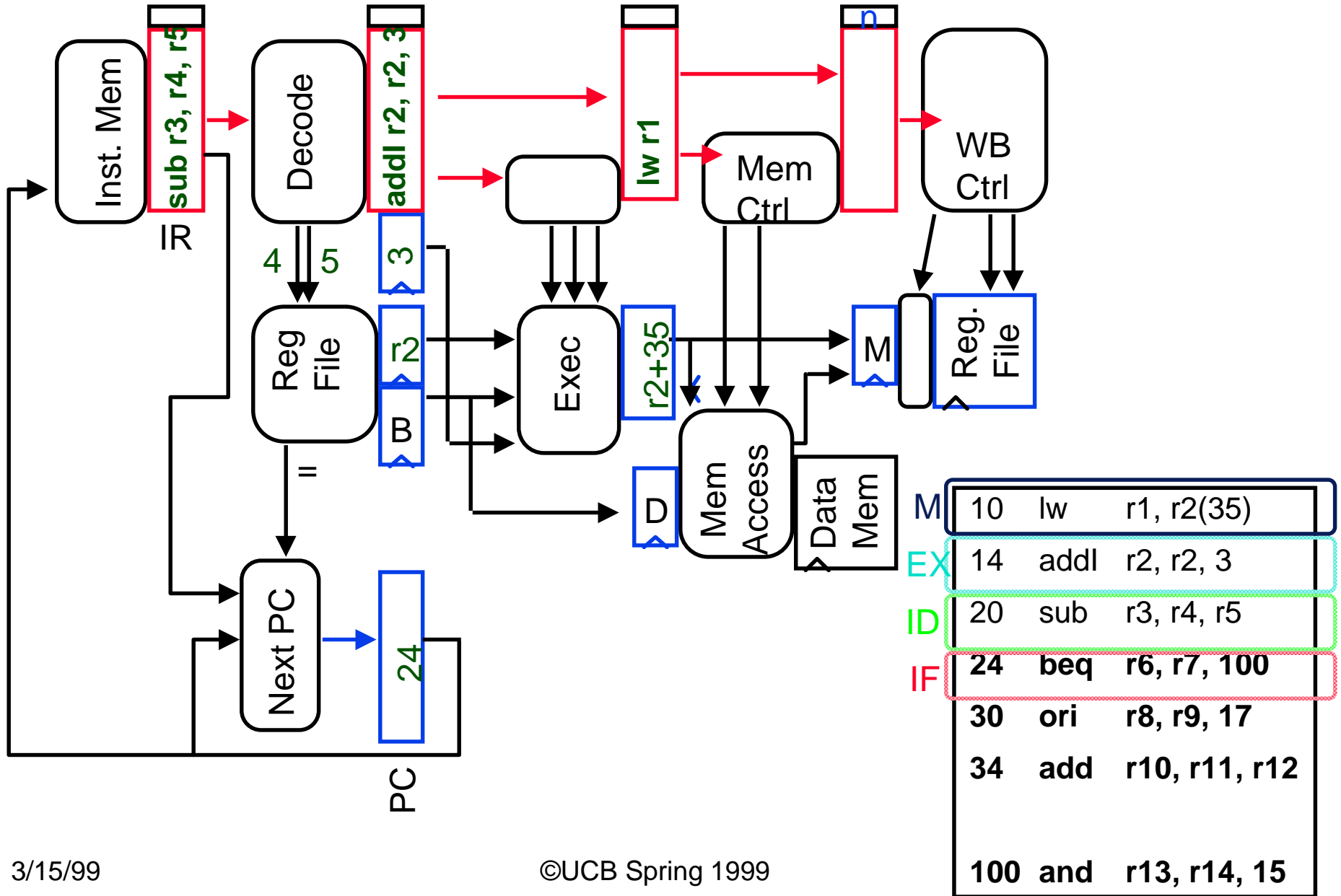


# Fetch 14, Decode 10

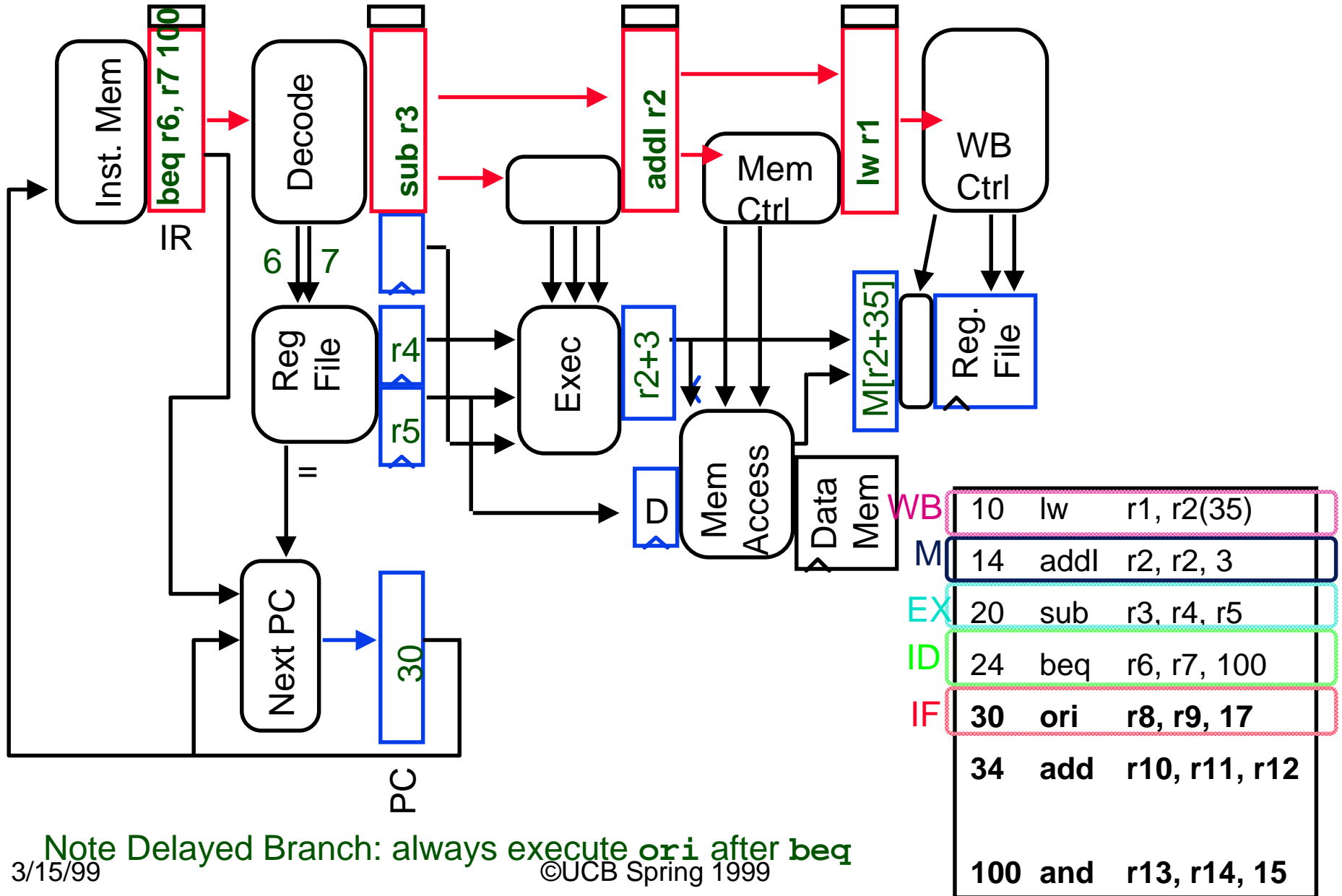




# Fetch 24, Decode 20, Exec 14, Mem 10



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

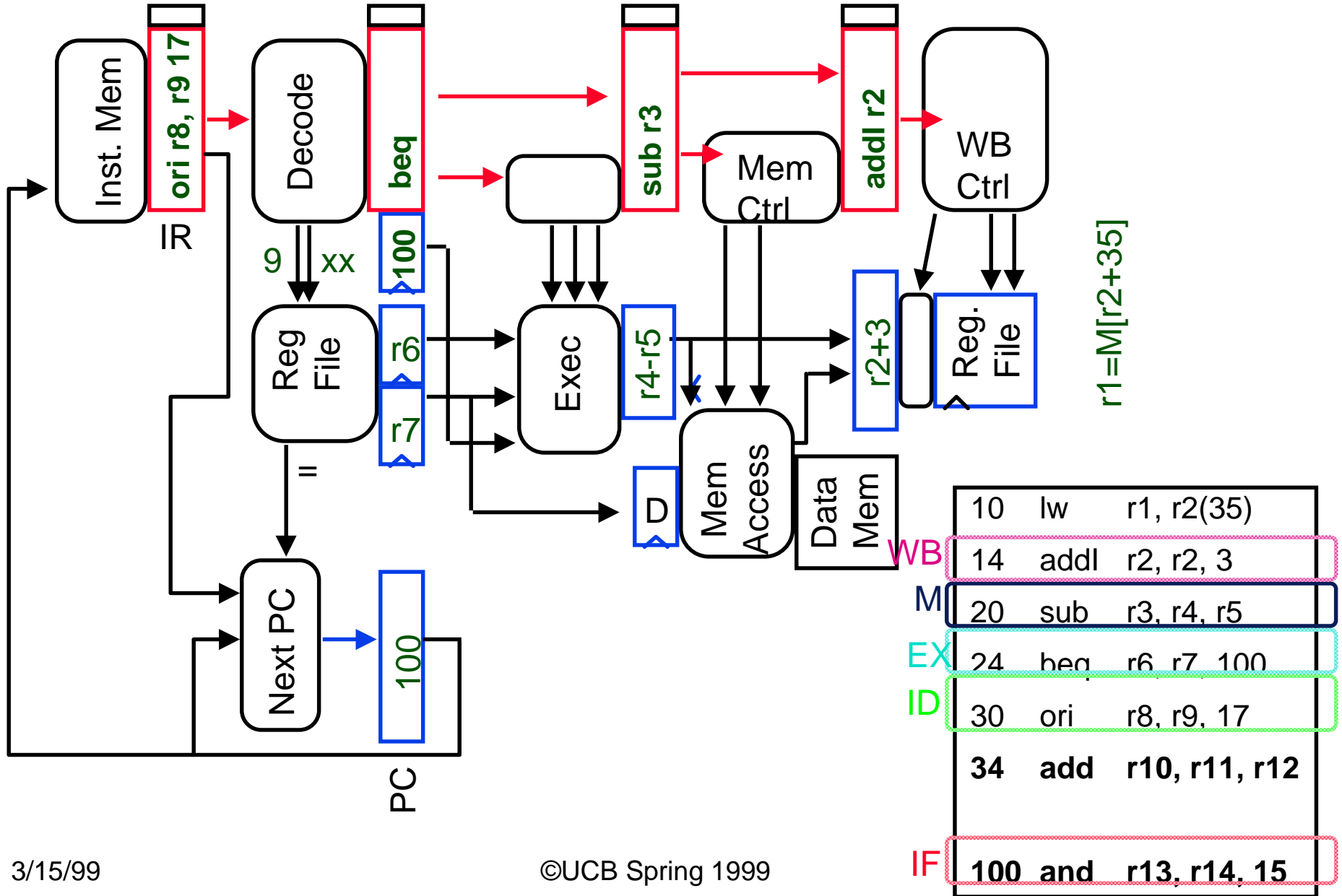


Note Delayed Branch: always execute `ori` after `beq`

3/15/99

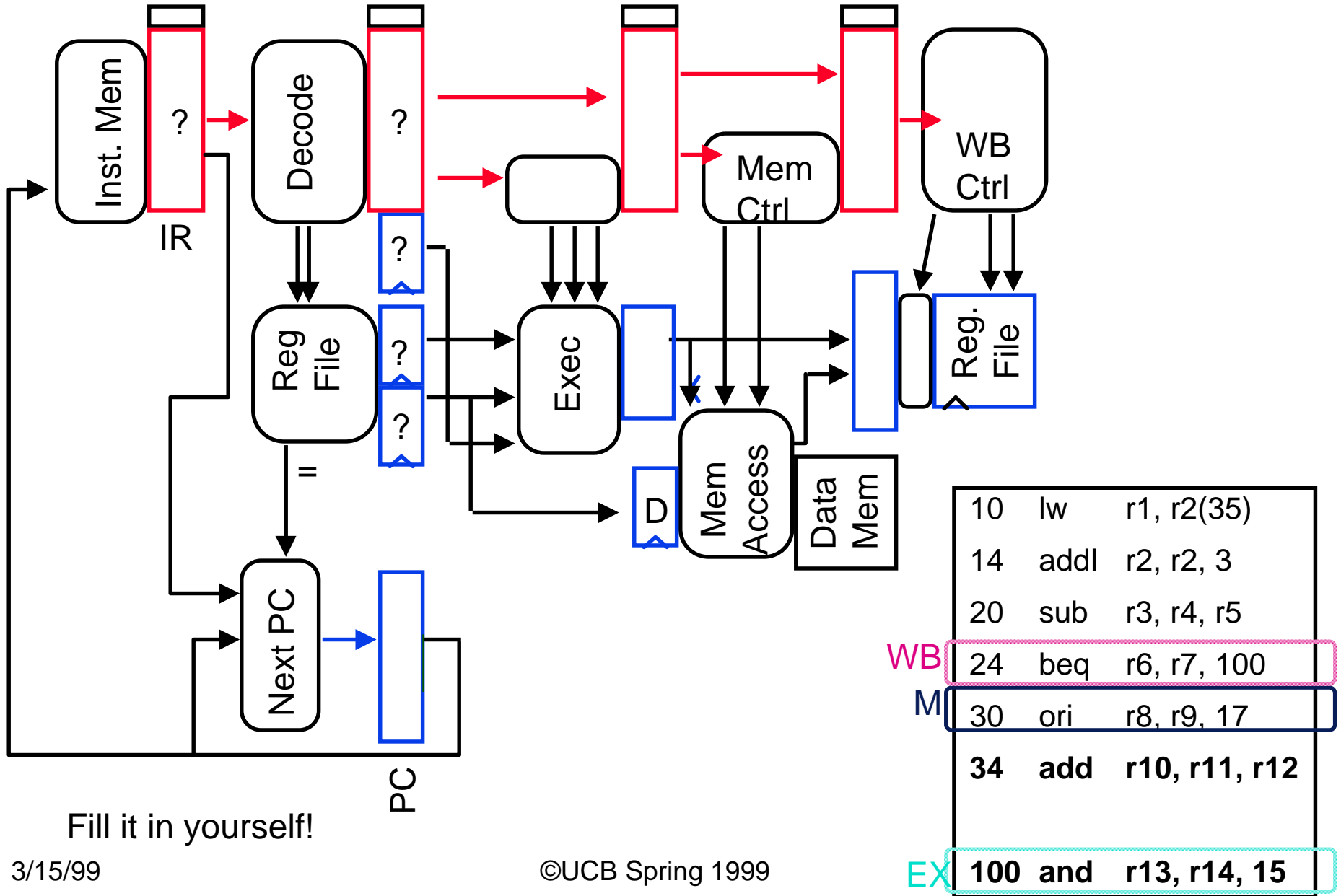
©UCB Spring 1999

# Fetch 100, Dcd 30, Ex 24, Mem 20, WB 14

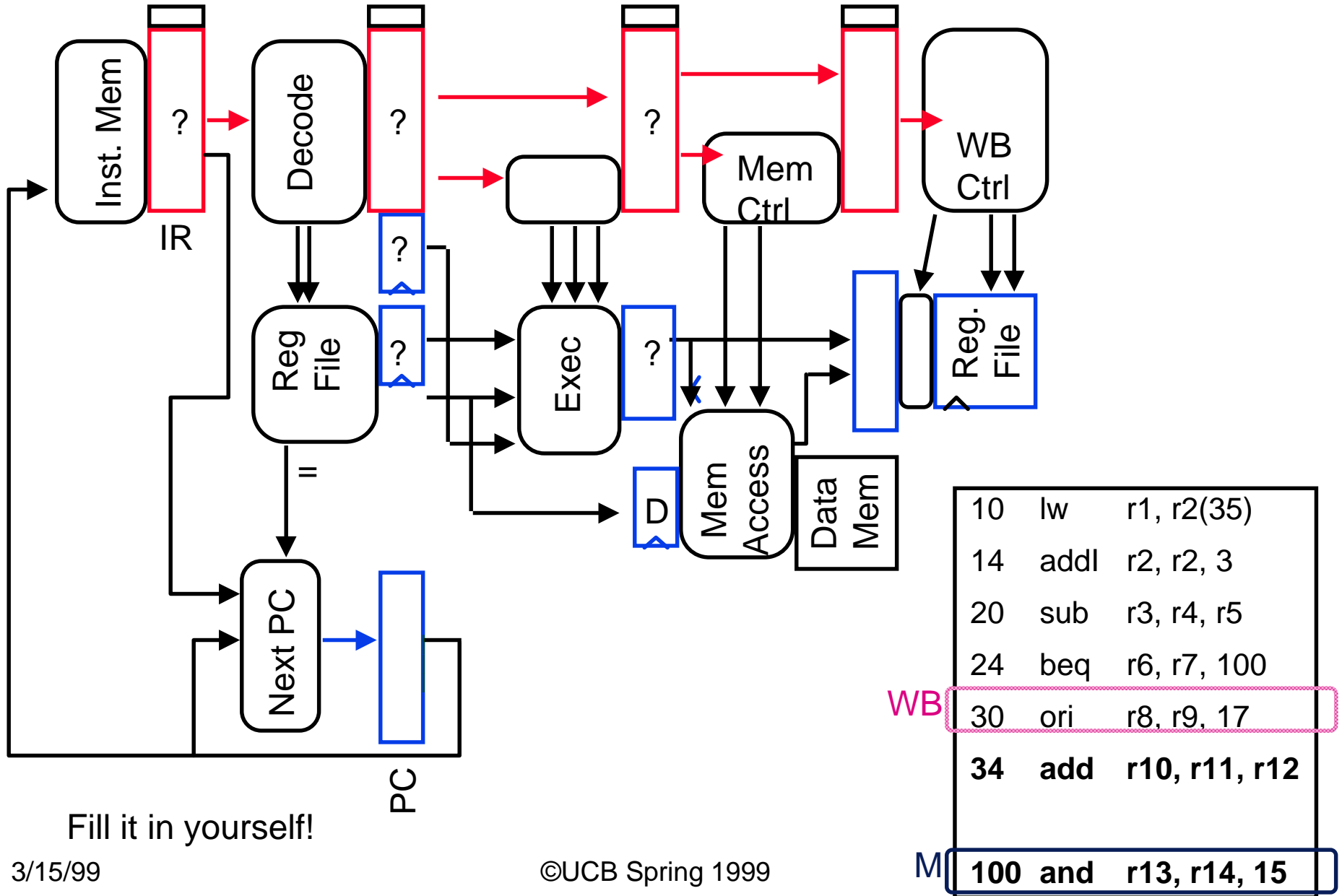




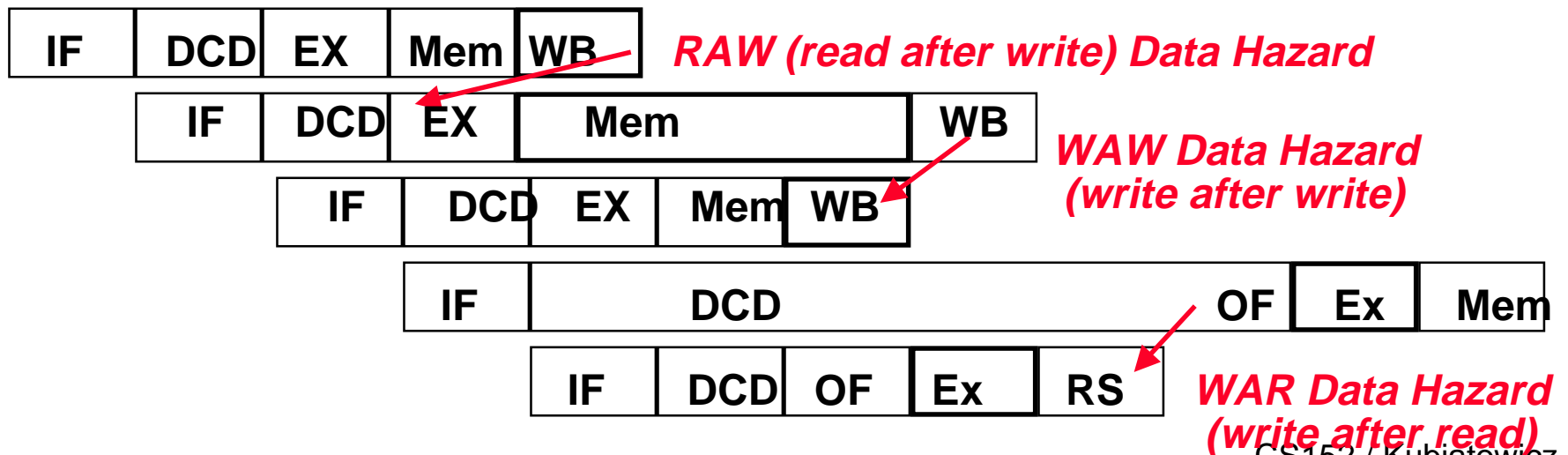
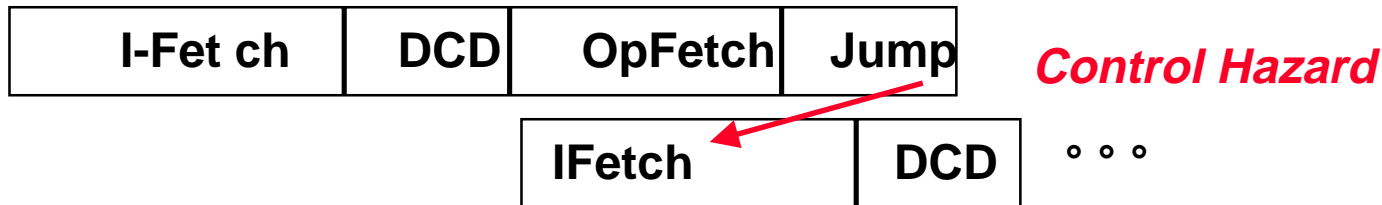
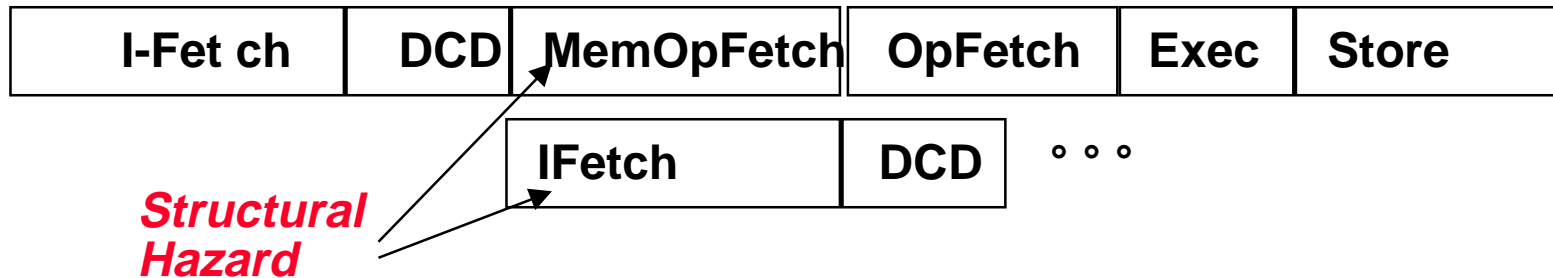
# Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24



# Fetch 114, Dcd 110, Ex 104, Mem 100, WB 30

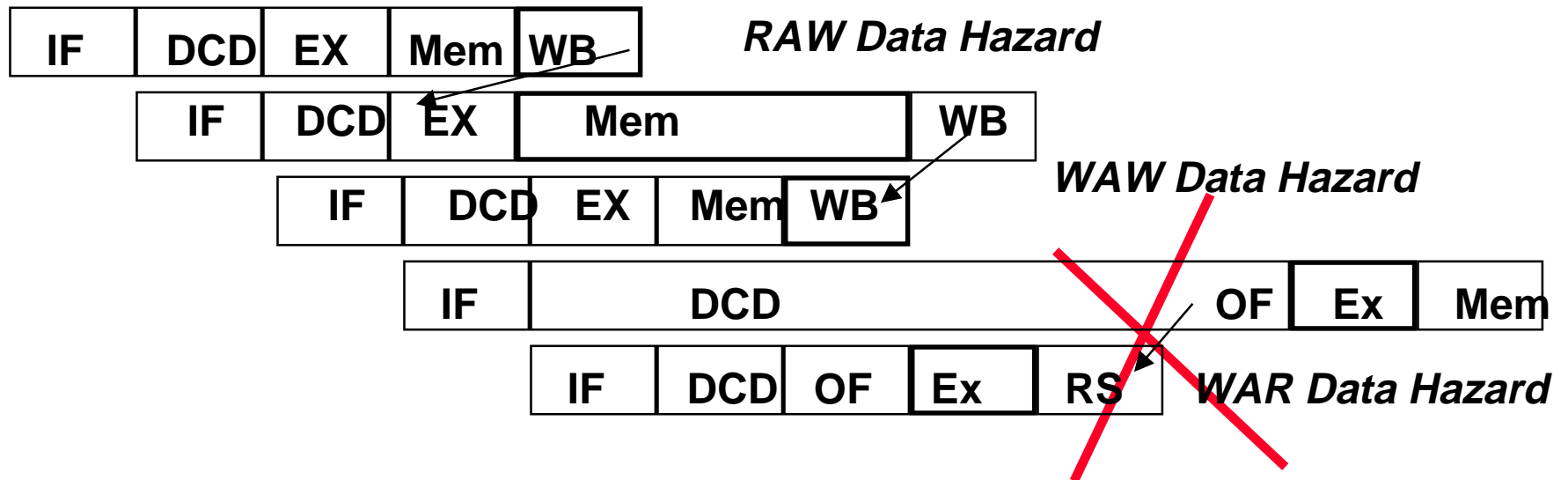


# Pipeline Hazards Again



# Data Hazards

- Avoid some “by design”
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eliminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)



# Hazard Detection

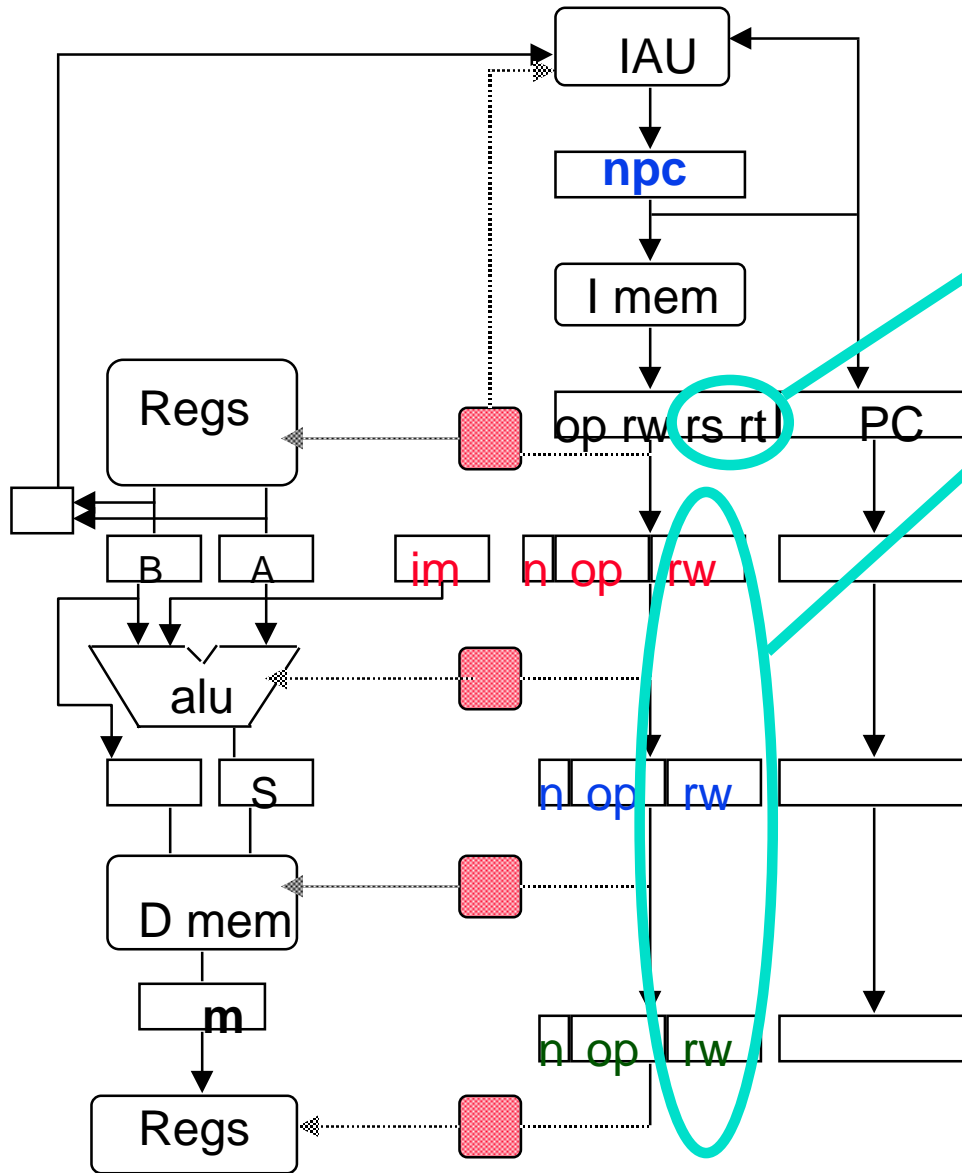
---

- Suppose instruction  $i$  is about to be issued and a predecessor instruction  $j$  is in the instruction pipeline.
- A RAW hazard exists on register  $\rho$  if  $\rho \in \text{Rregs}(i) \cap \text{Wregs}(j)$ 
  - Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.
  - When instruction issues, reserve its result register.
  - When an operation completes, remove its write reservation.



- A WAW hazard exists on register  $\rho$  if  $\rho \in \text{Wregs}(i) \cap \text{Wregs}(j)$
- A WAR hazard exists on register  $\rho$  if  $\rho \in \text{Wregs}(i) \cap \text{Rregs}(j)$

# Record of Pending Writes



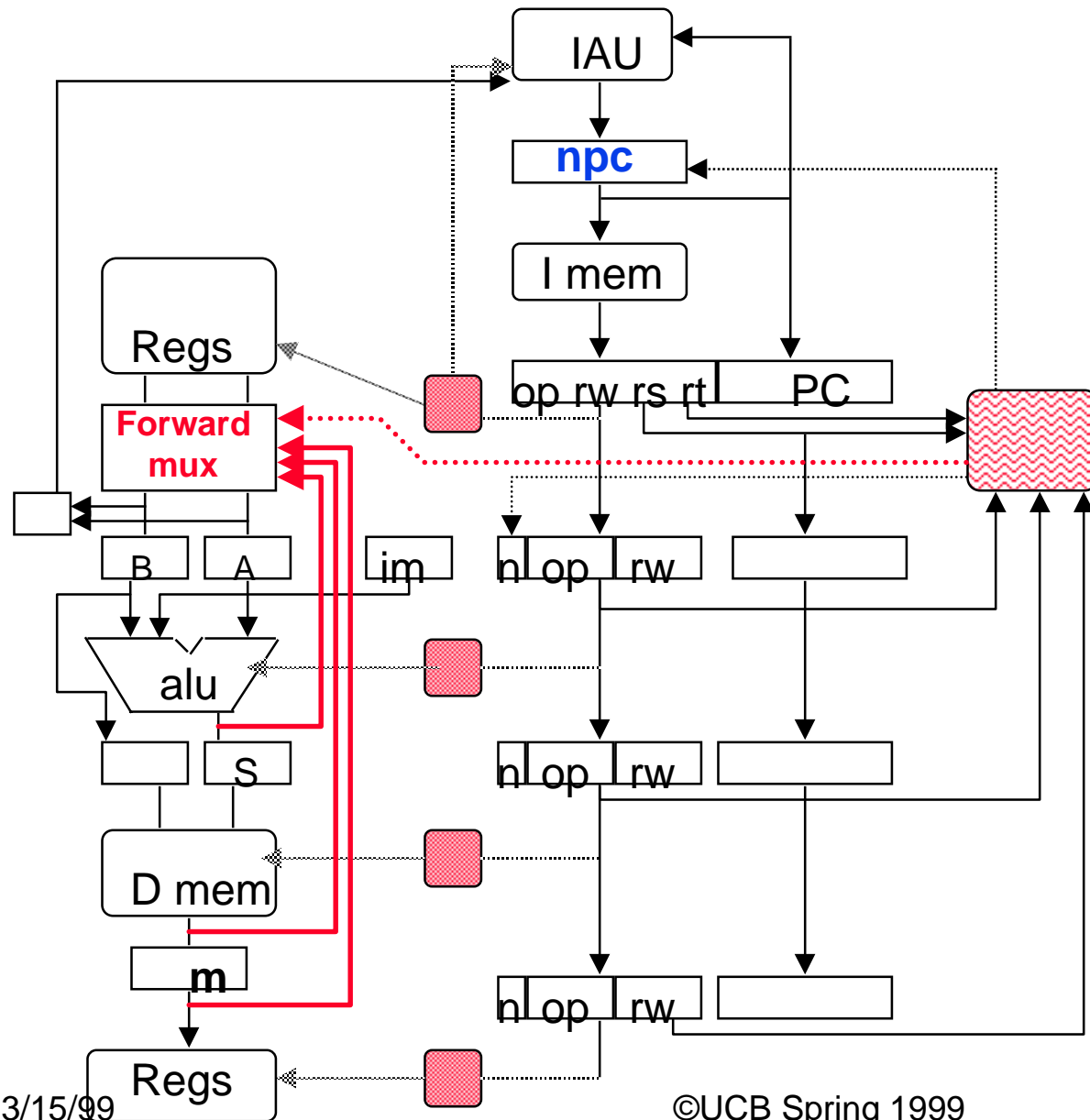
Current operand registers

Pending writes

hazard  $\leq$

- $((rs == rw_{ex}) \ \& \ regW_{ex}) \ OR$
- $((rs == rw_{mem}) \ \& \ regW_{me}) \ OR$
- $((rs == rw_{wb}) \ \& \ regW_{wb}) \ OR$
- $((rt == rw_{ex}) \ \& \ regW_{ex}) \ OR$
- $((rt == rw_{mem}) \ \& \ regW_{me}) \ OR$
- $((rt == rw_{wb}) \ \& \ regW_{wb})$

# Resolve RAW by forwarding

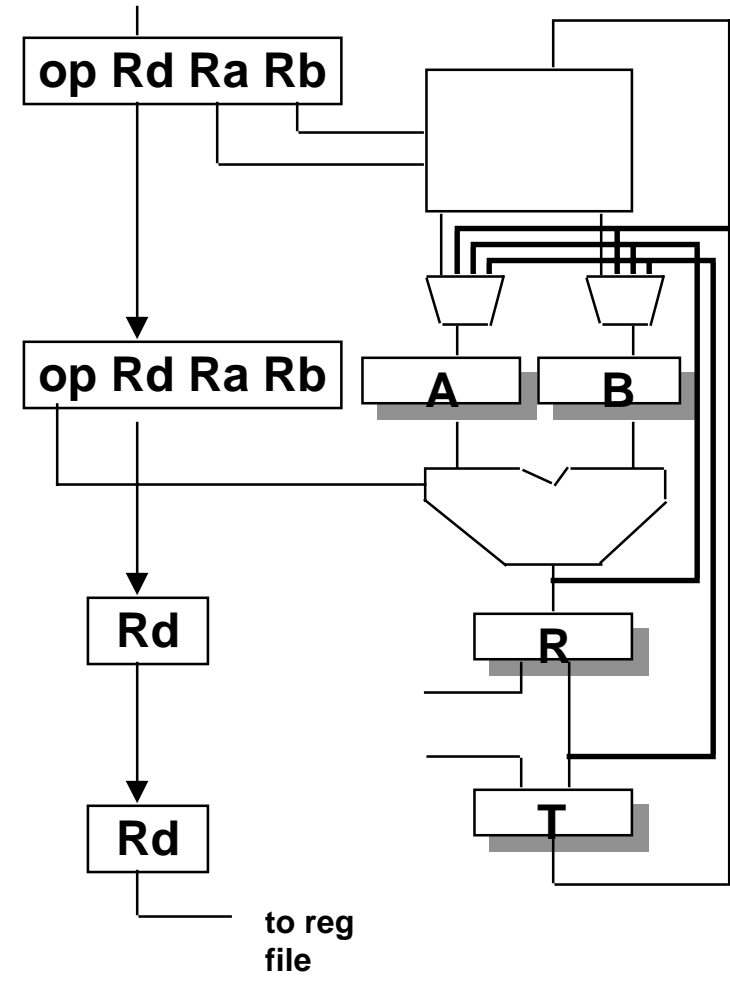


- Detect nearest **valid** write op operand register and **forward** into op latches, **bypassing** remainder of the pipe
- Increase muxes to add paths from pipeline registers
- **Data Forwarding = Data Bypassing**

# What about memory operations?

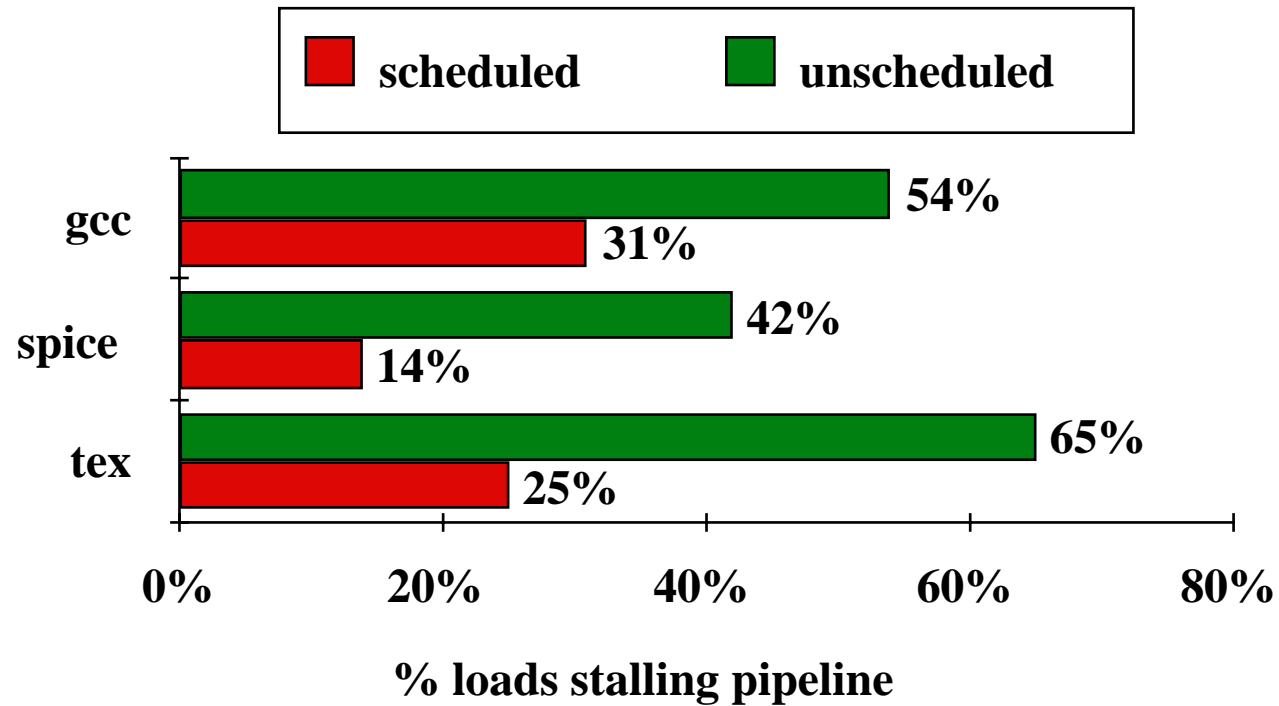
- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
  - cycles ?
  - hardware ?
- What about data dependence on loads?
  - R1 ← R4 + R5
  - R2 ← Mem[ R2 + I ]
  - R3 ← R2 + R1
 => "Delayed Loads"

Tricky situation:  
 R1 ← Mem[ R2 + I ]  
 Mem[R3+34] ← R1



# Compiler Avoiding Load Stalls:

---

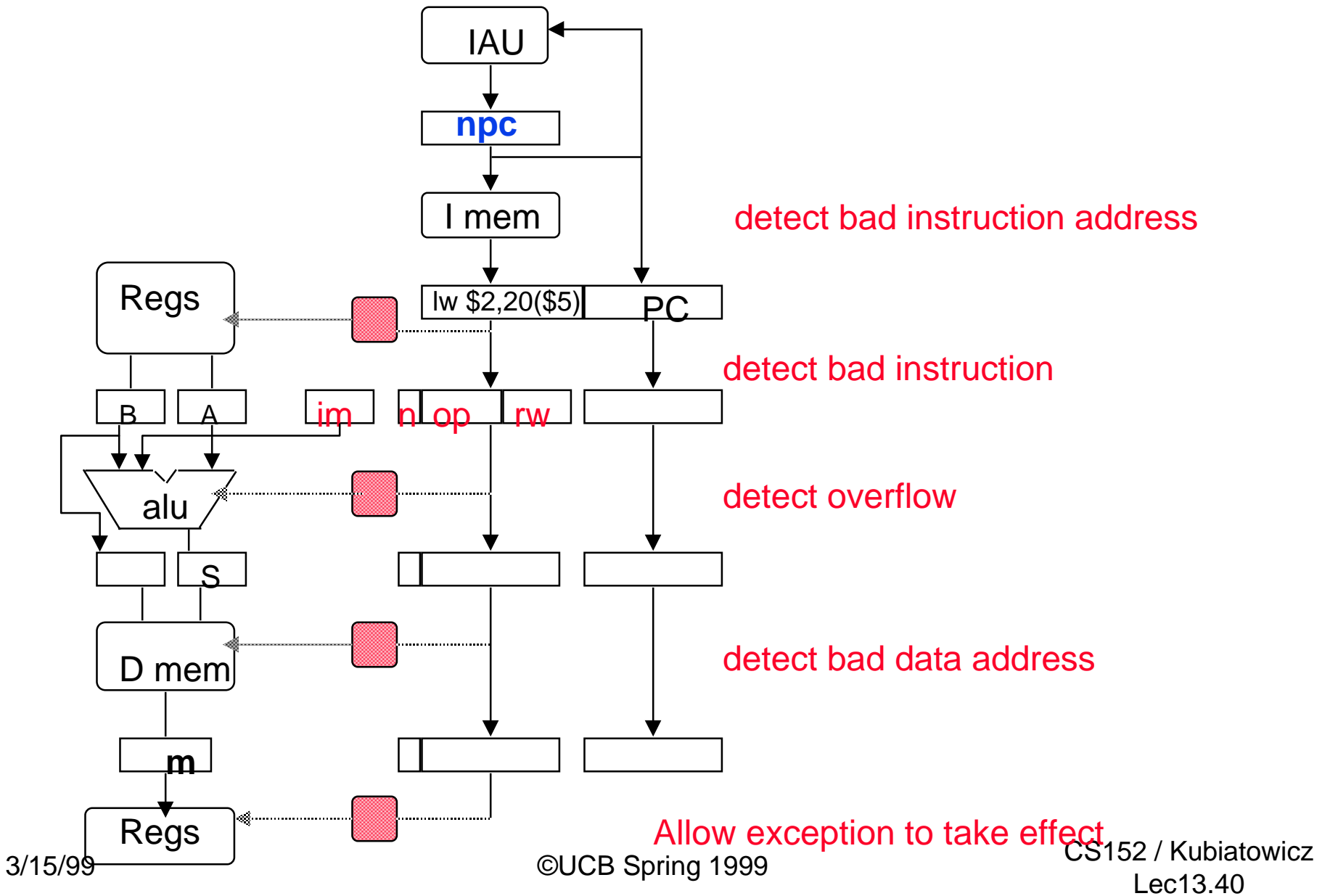


# What about Interrupts, Traps, Faults?

---

- **External Interrupts:**
  - Allow pipeline to drain,
  - Load PC with interrupt address
- **Faults (within instruction, restartable)**
  - Force trap instruction into IF
  - disable writes till trap hits WB
  - must save multiple PCs or PC + state

# Exception Handling



# Exception Problem

---

- **Exceptions/Interrupts:** 5 instructions executing in 5 stage pipeline
  - How to stop the pipeline?
  - Restart?
  - Who caused the interrupt?

**Stage**      *Problem interrupts occurring*

**IF**          Page fault on instruction fetch; misaligned memory access; memory-protection violation

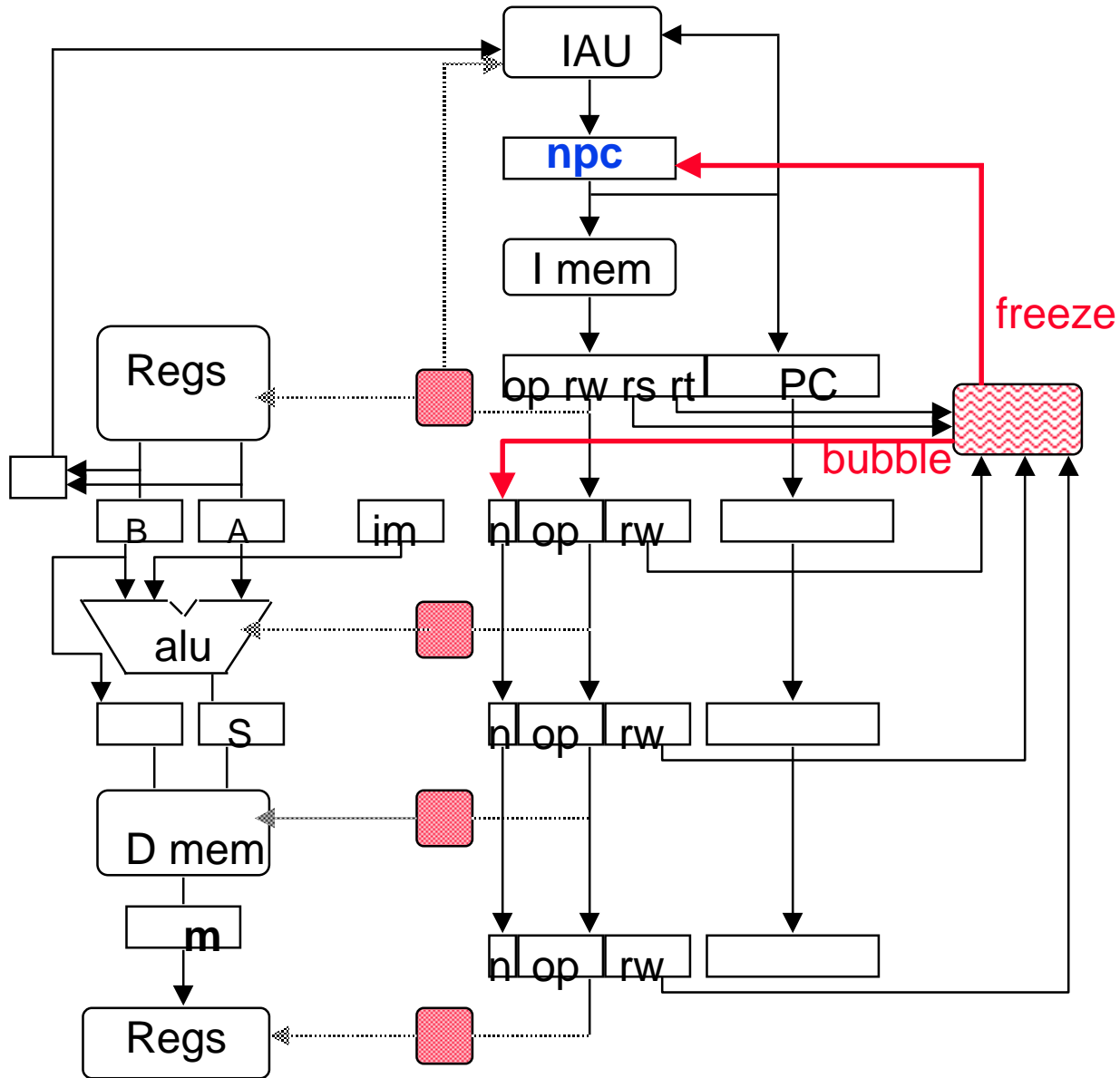
**ID**          Undefined or illegal opcode

**EX**          Arithmetic exception

**MEM**        Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

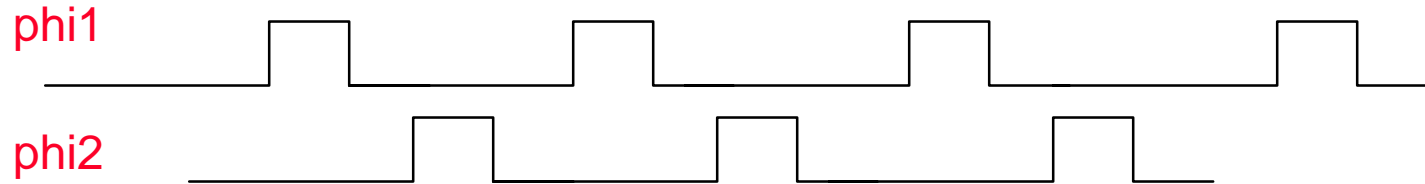
- Load with data page fault, Add with instruction page fault?
- **Solution 1:** interrupt vector/instruction 2: interrupt ASAP, restart everything incomplete

# Resolution: Freeze above & Bubble Below

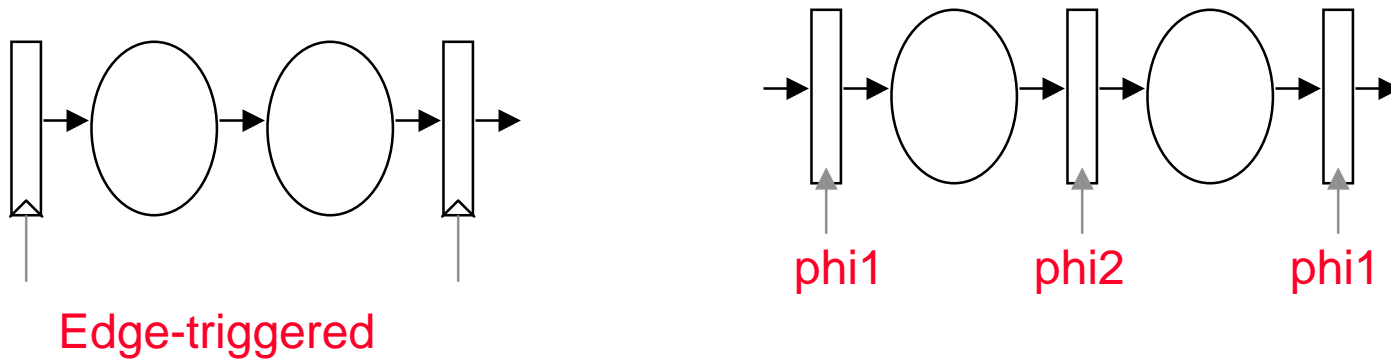


# FYI: MIPS R3000 clocking discipline

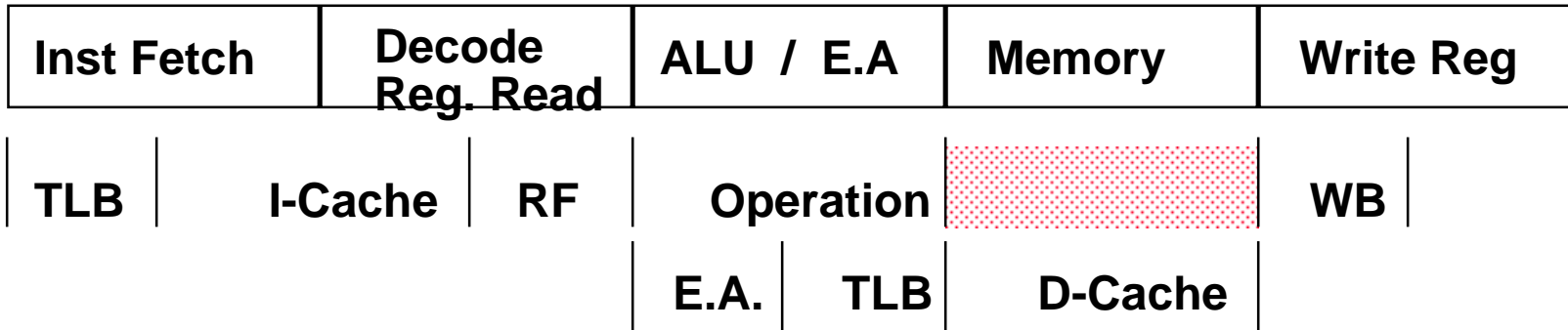
---



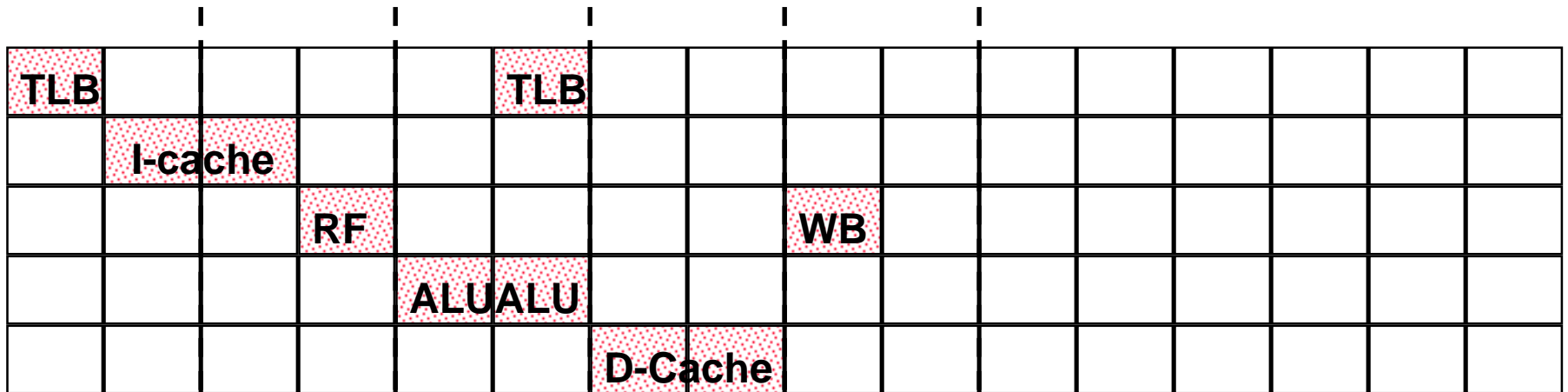
- **2-phase non-overlapping clocks**
- **Pipeline stage is two (level sensitive) latches**



# MIPS R3000 Instruction Pipeline

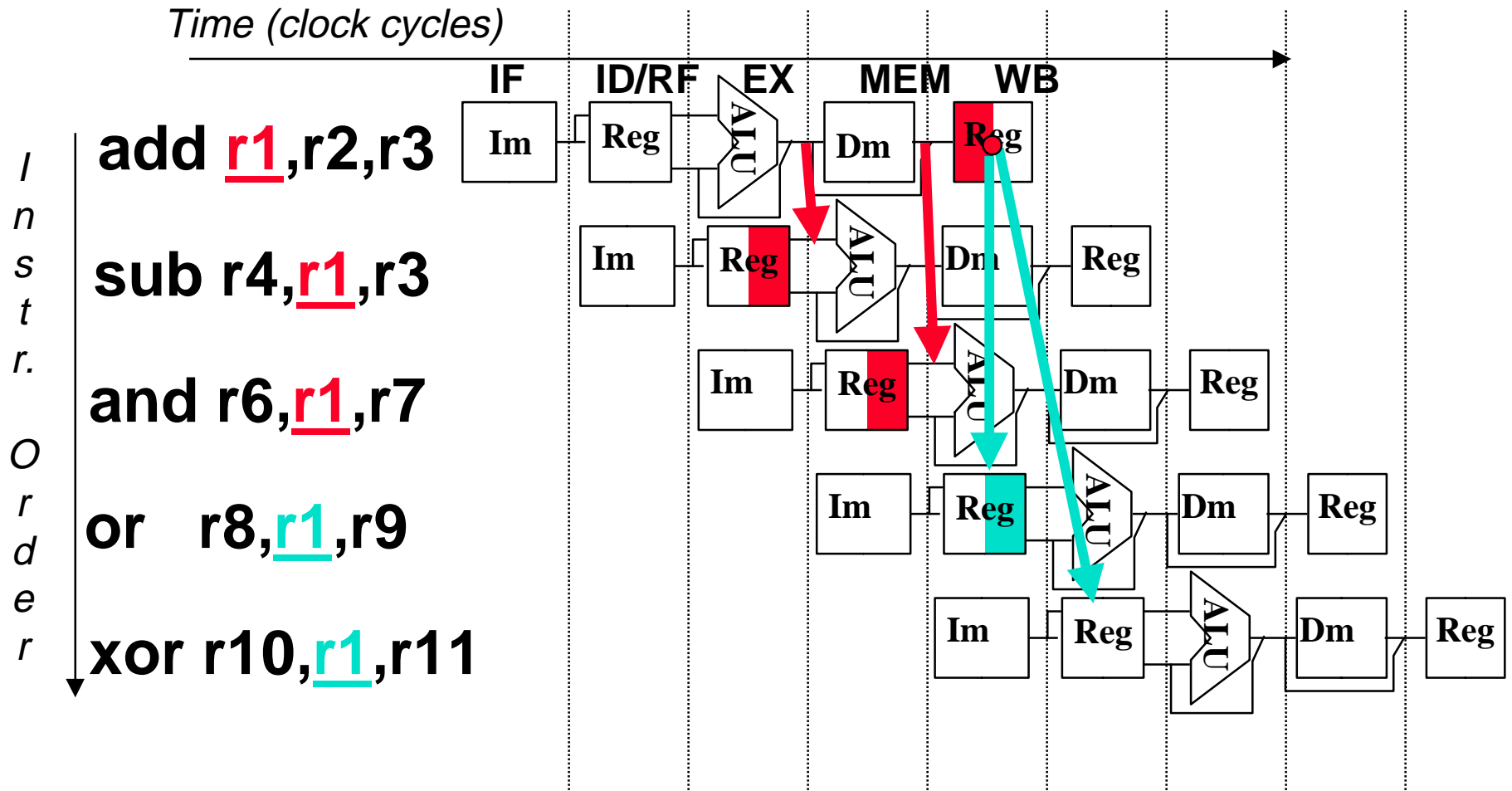


## Resource Usage



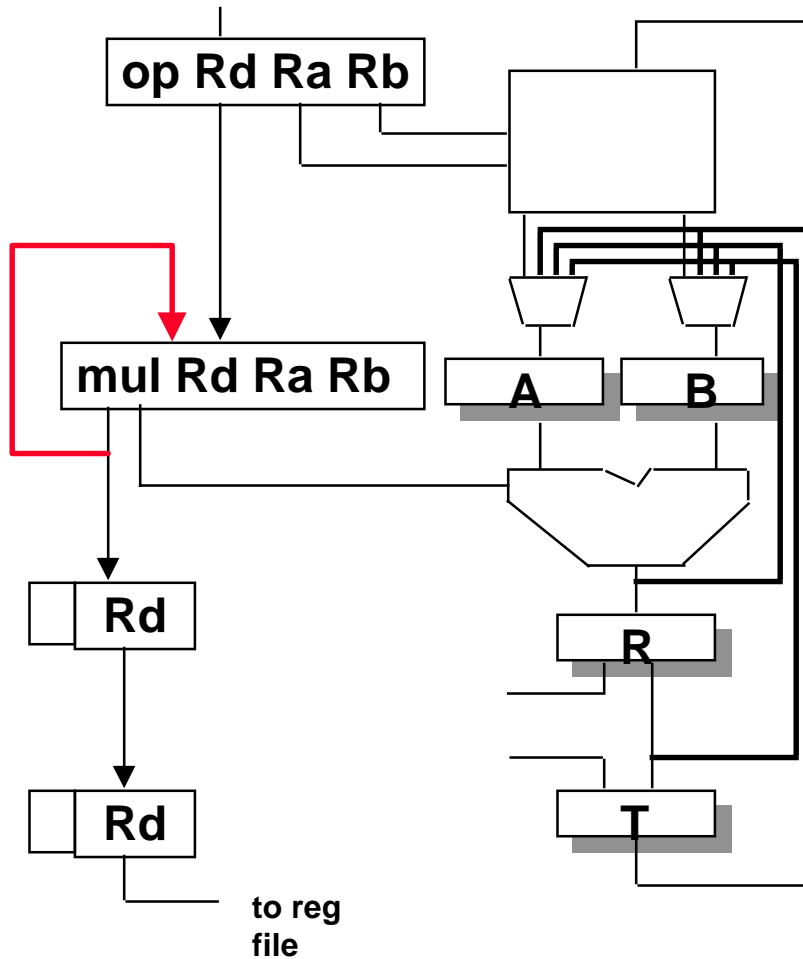
Write in phase 1, read in phase 2 => eliminates bypass from WB

# Recall: Data Hazard on r1



With MIPS R3000 pipeline, no need to forward from WB stage

# MIPS R3000 Multicycle Operations



**Ex: Multiply, Divide, Cache Miss**

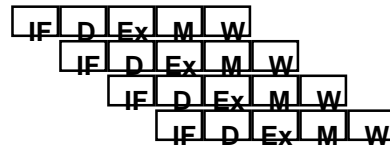
**Stall all stages above multicycle operation in the pipeline**

**Drain (bubble) stages below it**

**Use control word of local stage state to step through multicycle operation**

# Issues in Pipelined design

◦ Pipelining

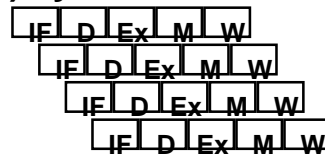


Limitation

Issue rate, FU stalls, FU depth

◦ Super-pipeline

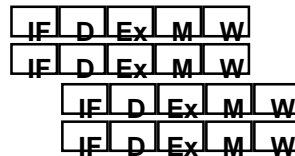
- Issue one instruction per (fast) cycle
- ALU takes multiple cycles



Clock skew, FU stalls, FU depth

◦ Super-scalar

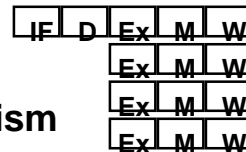
- Issue multiple scalar instructions per cycle



Hazard resolution

◦ VLIW ("EPIC")

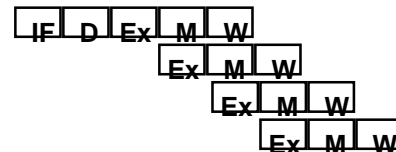
- Each instruction specifies multiple scalar operations
- Compiler determines parallelism



Packing

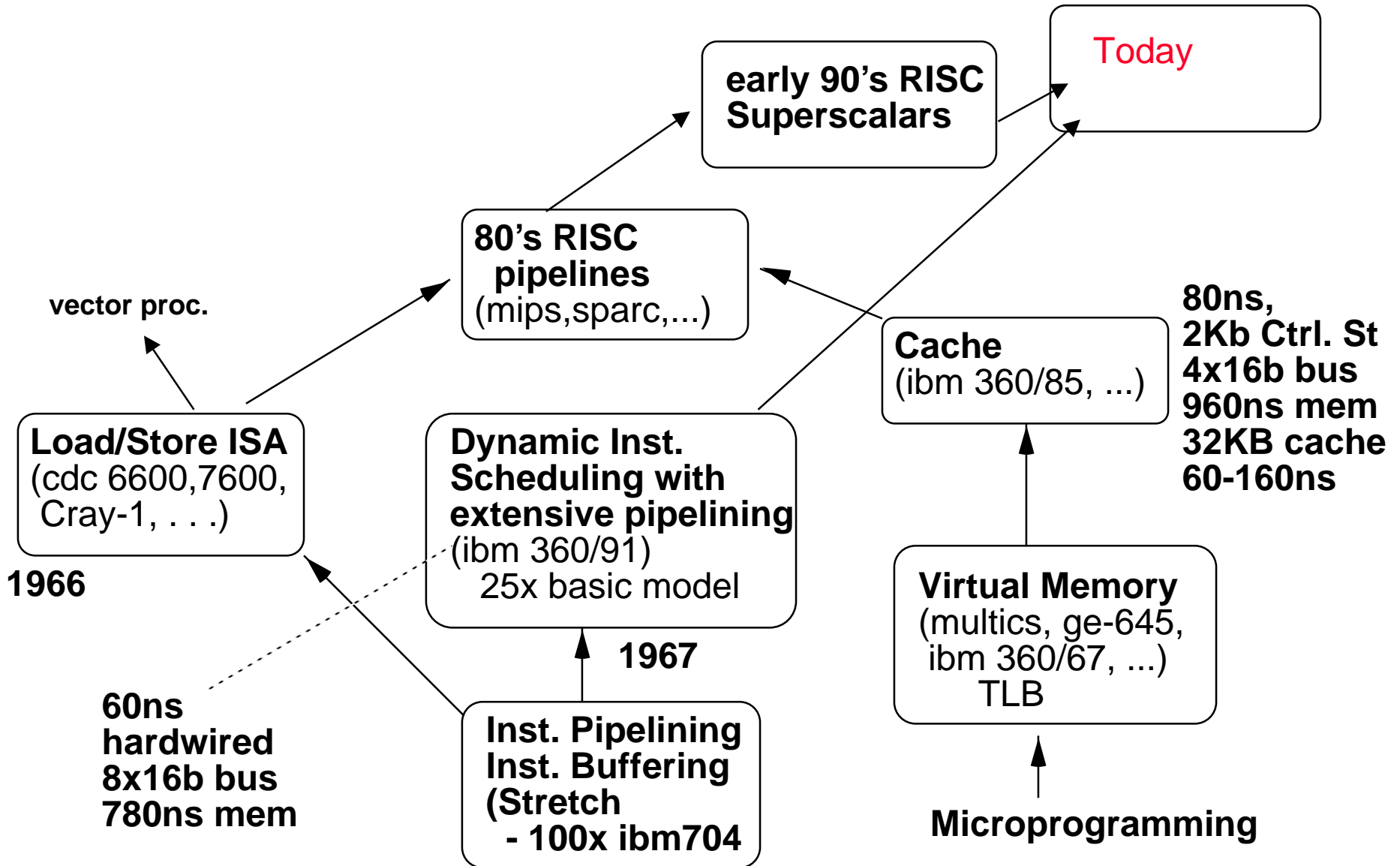
◦ Vector operations

- Each instruction specifies series of identical operations

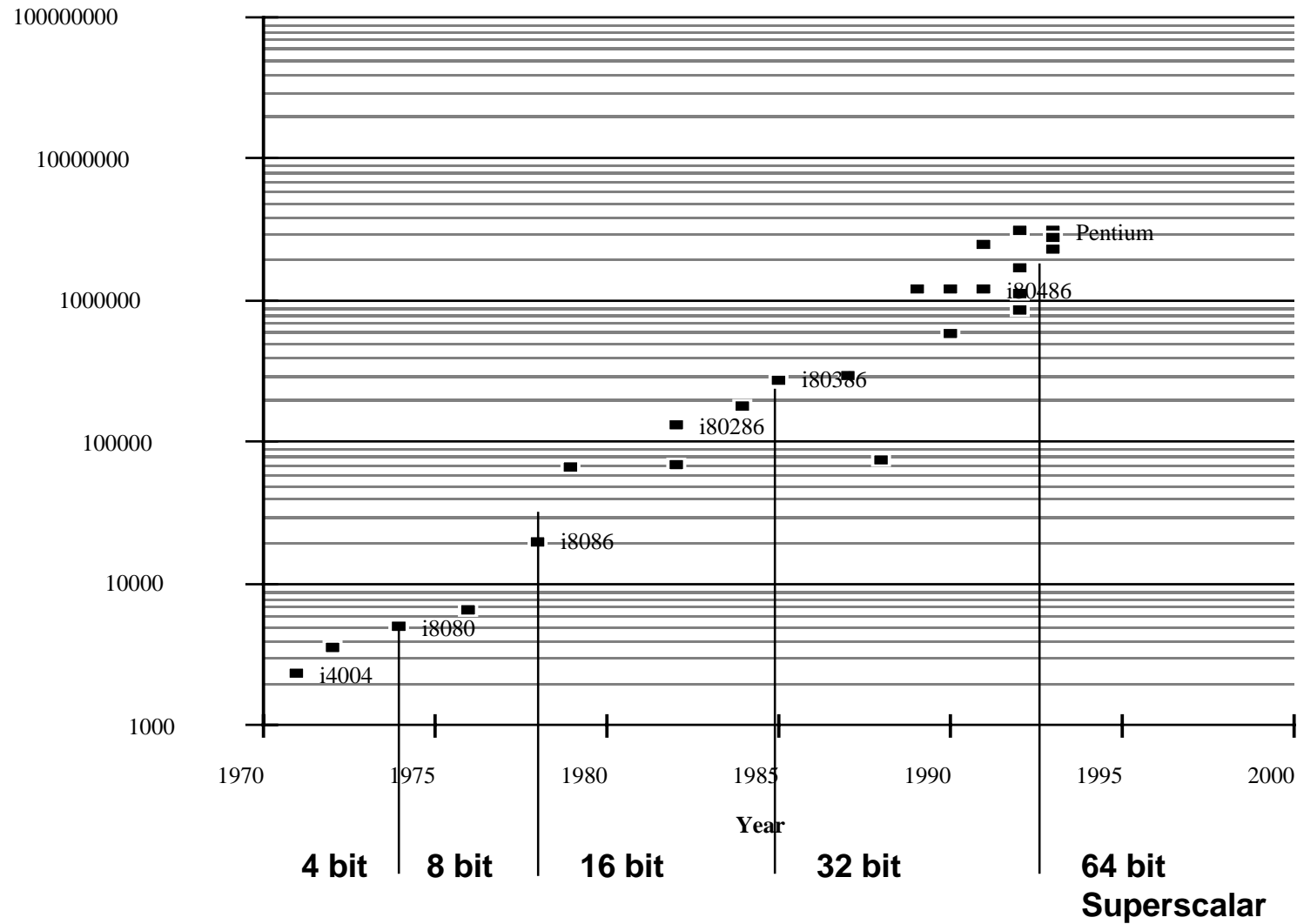


Applicability

# Historical Perspective

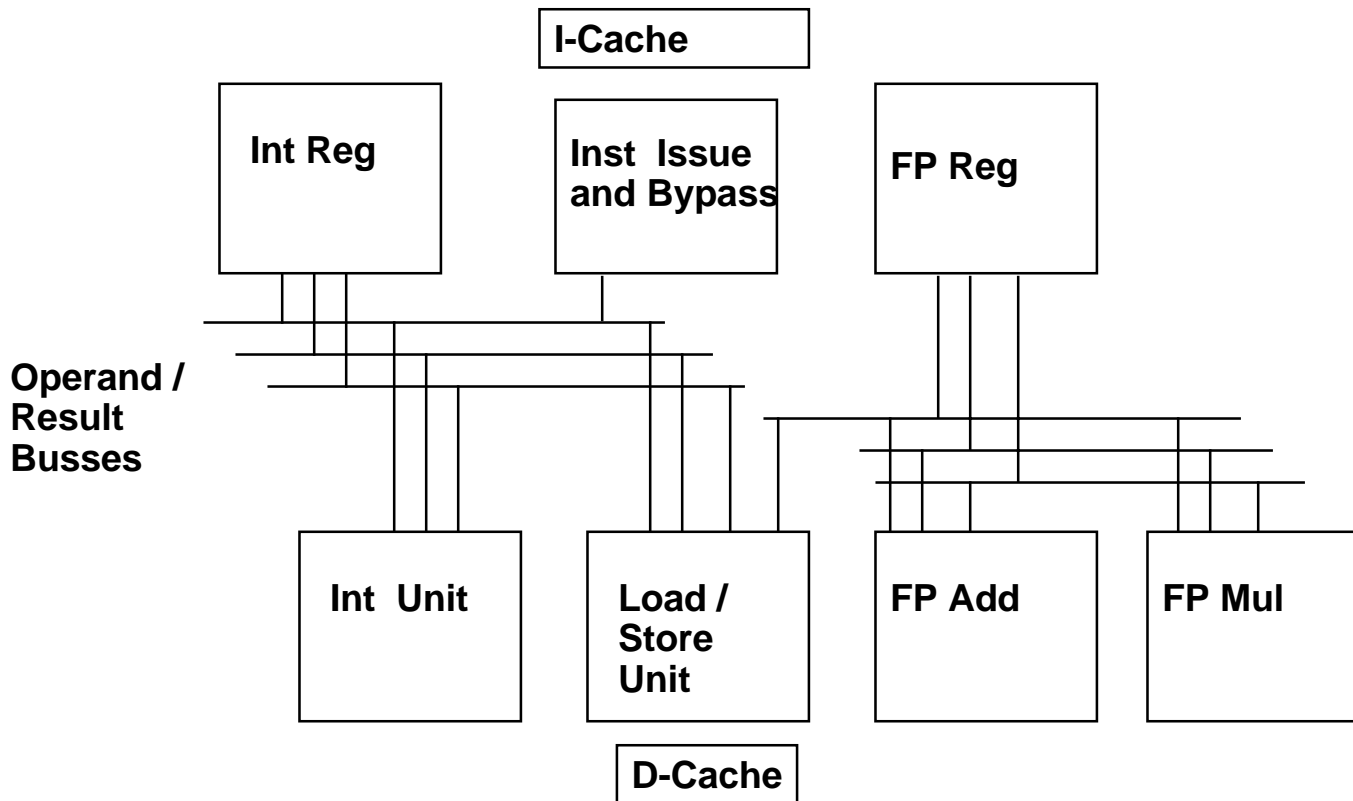


# Technology Perspective



# Partitioned Instruction Issue (simple Superscalar)

independent int and FP issue to separate pipelines



Single Issue Total Time = Int Time + FP Time

Max Speedup:  $\frac{\text{Total Time}}{\text{MAX(Int Time, FP Time)}}$

# Example: DAXPY

---

Basic Loop:  $\leftarrow Rm + Ry$

Total Single Issue Cycles: 19 ( 7 integer, 12 floating point)

Minimum with Dual Issue: 12

Potential Speedup: 1.6 !!!

Actual Cycles: 18

# Unrolling

## Basic Loop:

```
load  a <- Ai
load  y <- Yi
mult  m <- a*s
add   r <- m+y
store Ai <- r
inc   Ai
inc   Yi
dec   i
branch
```

about 9 inst. per 2 FP ops

## Unrolled Loop:

```
[ load,load,
  mult, add,
  store
```

```
[ load,load
  mult, add,
  store
```

```
[ load,load
  mult,
  add,store
```

```
[ load,load,
  mult, add,
  store
```

```
inc,inc, dec,
branch
```

about 6 inst. per 2 FP ops  
dependencies between  
instructions remain.

## Reordered Unrolled Loop:

```
load, load,
load, ...
```

```
mult, mult,
mult, mult,
```

```
add, add, add,
add,
```

```
store, store,
store, store
```

```
inc, inc, dec,
```

```
branch
```

schedule 24 inst basic  
block relative to pipeline  
- delay slots  
- function unit stalls  
- multiple function units  
- pipeline depth

# Software Pipelining

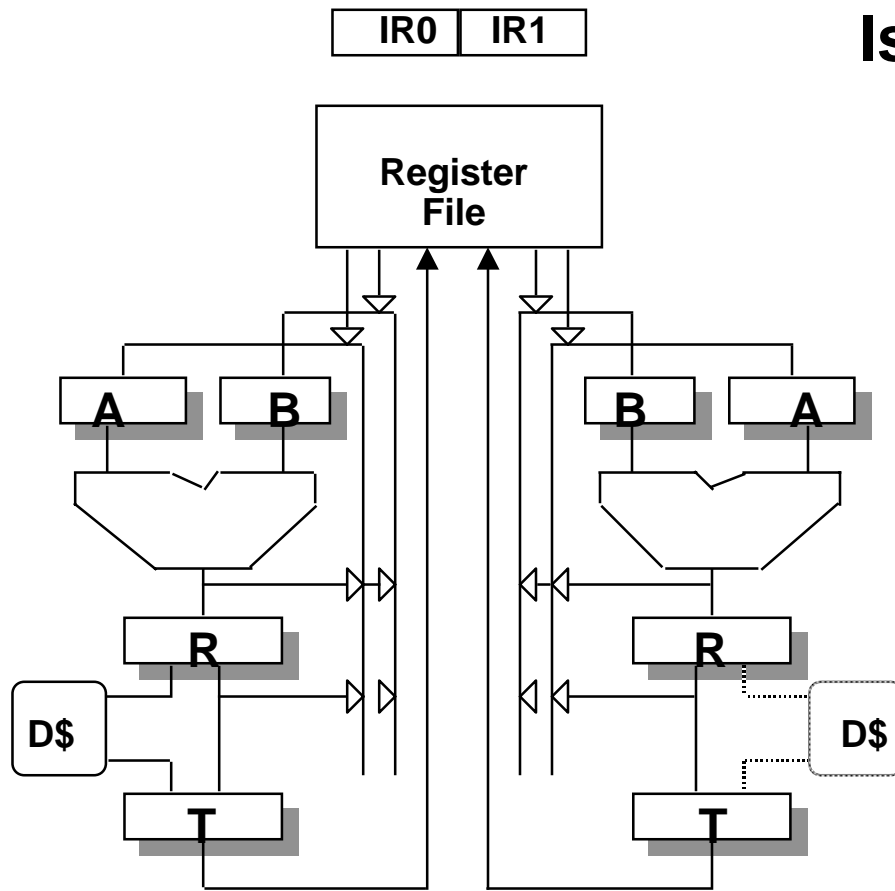
load a <- A1			
load y <- Y1 mult m <- a*s	load a' <- A2		
add r <- m+y inc, dec	load y' <- Y2 mult m' <- a'*s	load a'' <- A3	
store Ai <- r branch	add r' <- m'+y' inc, dec	load y'' <- Yi+2 mult m'' <- a''*s	load A''' <- Ai+3
	store Ai+1 <- r'	add r'' <- m''+y'' inc	

**Pipelined Loop:**

```

load a''' <- Ai+3
load y'' <- Yi+2
mult m'' <- a''*s
add r' <- m'+y'
store Ai <- r
inc Ai+3
inc Yi
dec i
a'' <- a'''; Y' <- y''; m' <- m''; r <- r'
branch
    
```

# Multiple Pipes/ Harder Superscalar



## Issues:

**Reg. File ports**

**Detecting Data  
Dependences**

**Bypassing**

**RAW Hazard**

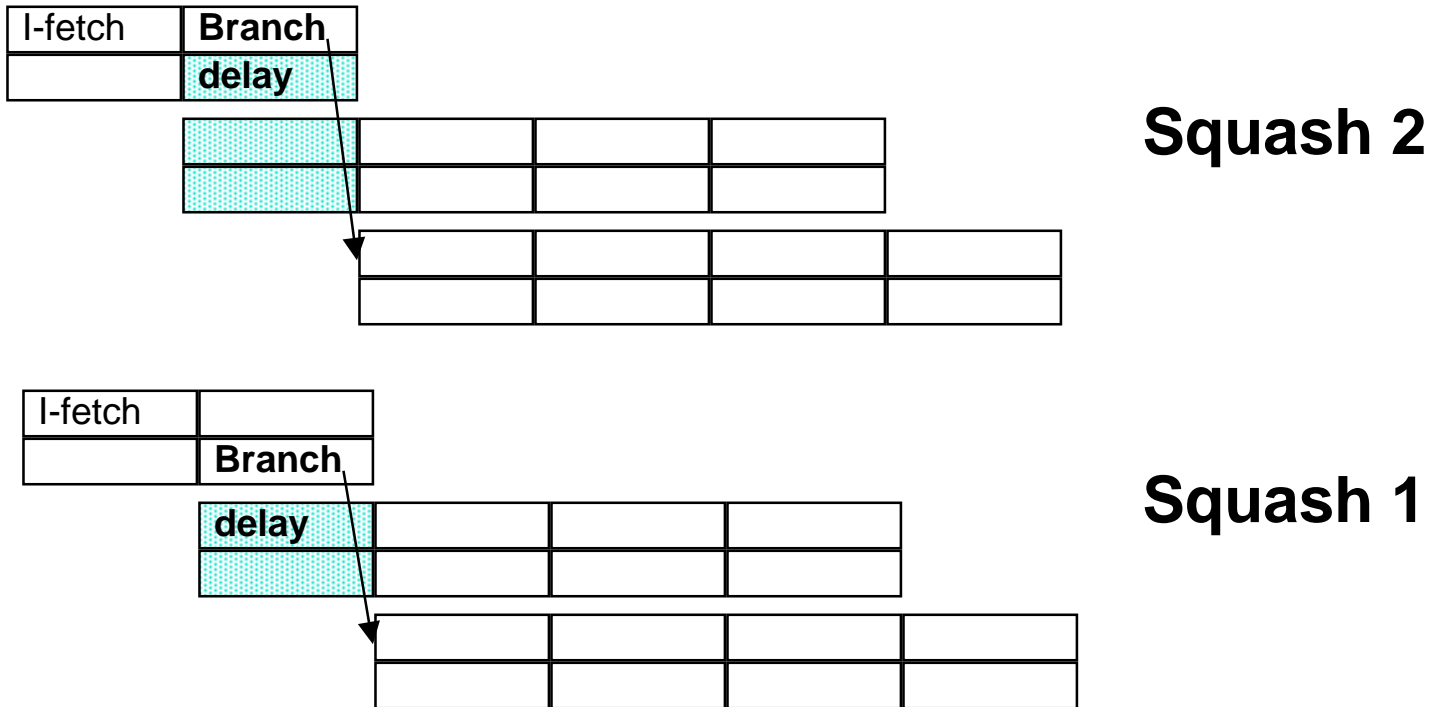
**WAR Hazard**

**Multiple load/store ops?**

**Branches**

# Branch penalties in superscalar

**Example: resolved in op-fetch stage,  
single exposed delay (ala MIPS, Sparc)**



## 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
- **Pipelines pass control information down the pipe just as data moves down pipe**
- **Forwarding/Stalls handled by local control**
- **Exceptions stop the pipeline**

## Summary

---

- **Pipelines pass control information down the pipe just as data moves down pipe**
- **Forwarding/Stalls handled by local control**
- **Exceptions stop the pipeline**
- **MIPS I instruction set architecture made pipeline visible (delayed branch, delayed load)**
- **More performance from deeper pipelines, parallelism**