

---

# **CS152: Computer Architecture and Engineering**

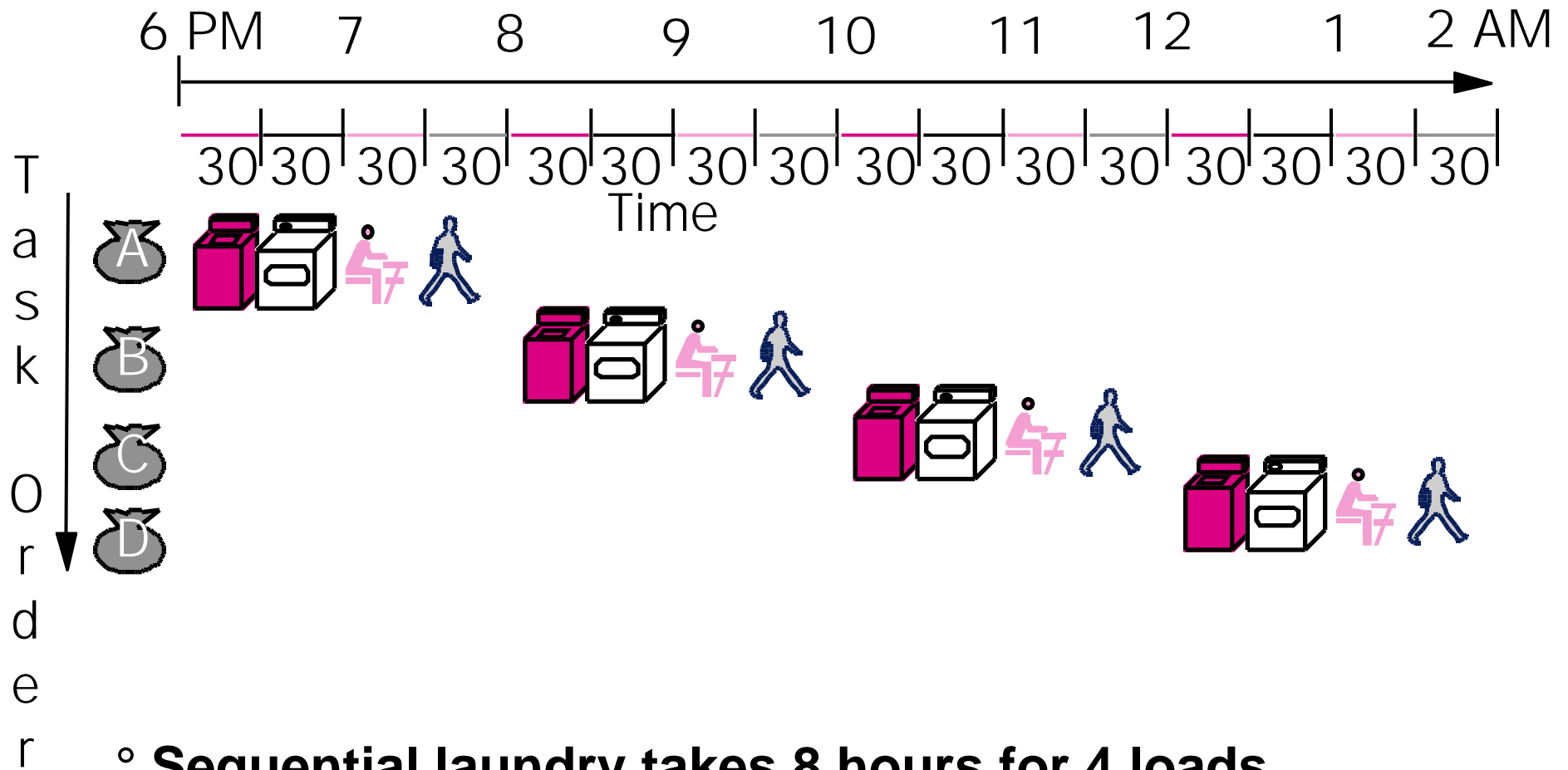
## **Introduction to Pipelining**

**October 22, 1997**

**Dave Patterson (<http://cs.berkeley.edu/~patterson>)**

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

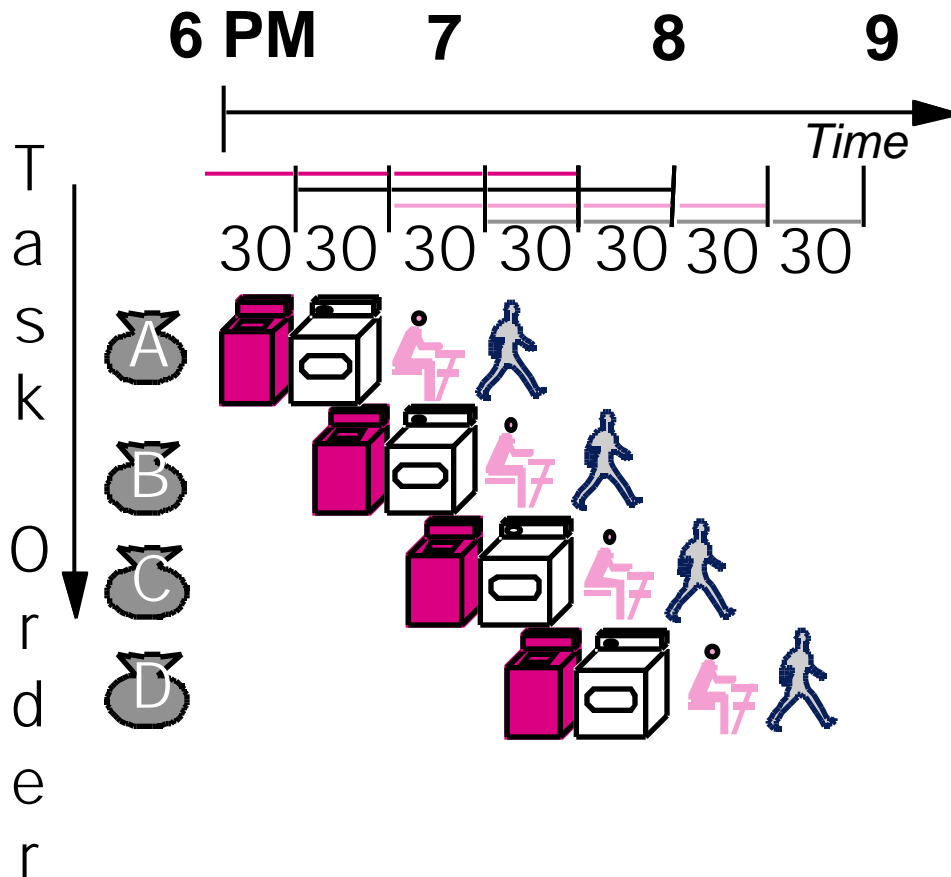
# Recap: Sequential Laundry



◦ Sequential laundry takes 8 hours for 4 loads

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

# Recap: Pipelining Lessons (its intuitive!)

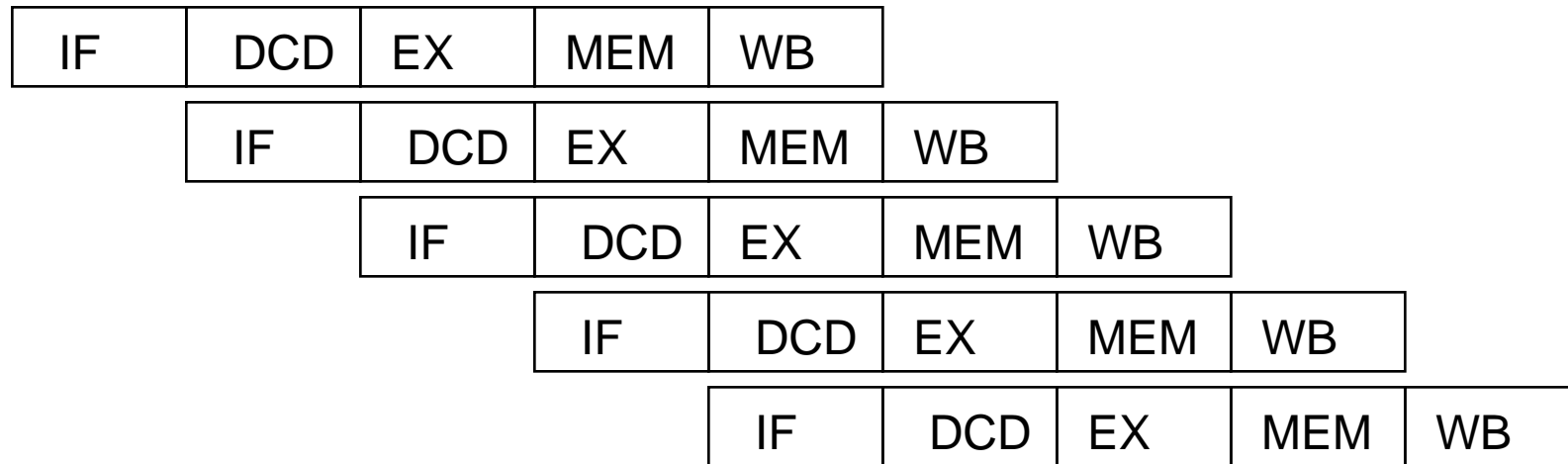


- Pipelining doesn't help **latency** of single task, it helps **throughput** of entire workload
- **Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Pipeline rate limited by **slowest** pipeline stage
- Unbalanced lengths of pipe stages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

# Recap: Ideal Pipelining

---

Assume instructions are completely independent!

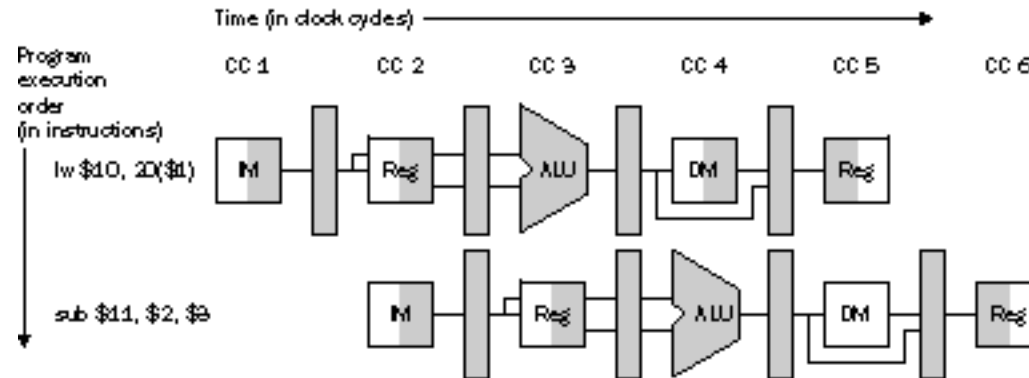


**Maximum Speedup  $\leq$  Number of stages**

**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: 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

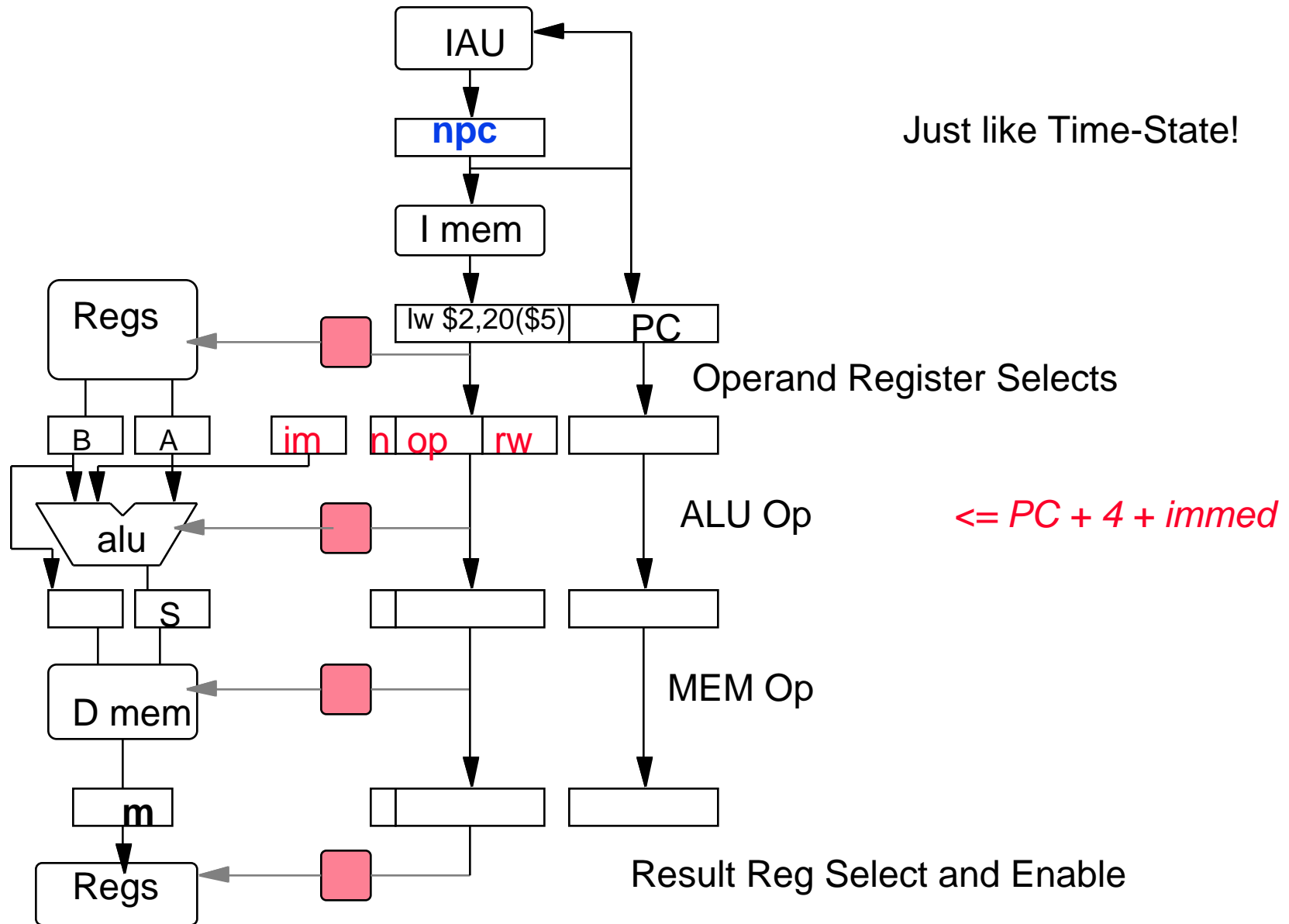
# Recap: Can pipelining get us into trouble?

---

## ◦ 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

# Recap: Pipelined Datapath with Data Stationary Control



# Recap

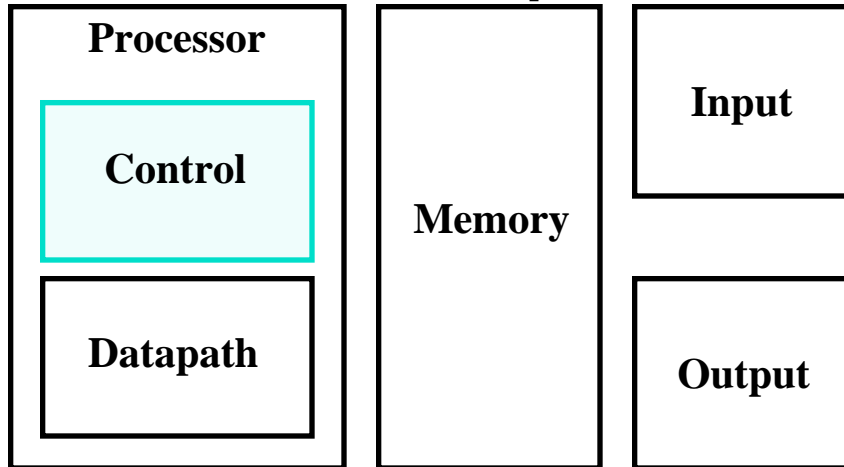
---

- **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
- **What makes it easy**
  - all instructions are the same length
  - just a few instruction formats
  - memory operands appear only in loads and stores
- **Hazards make it hard**
- **We'll build a simple pipeline and look at these issues**

# The Big Picture: Where are We Now?

---

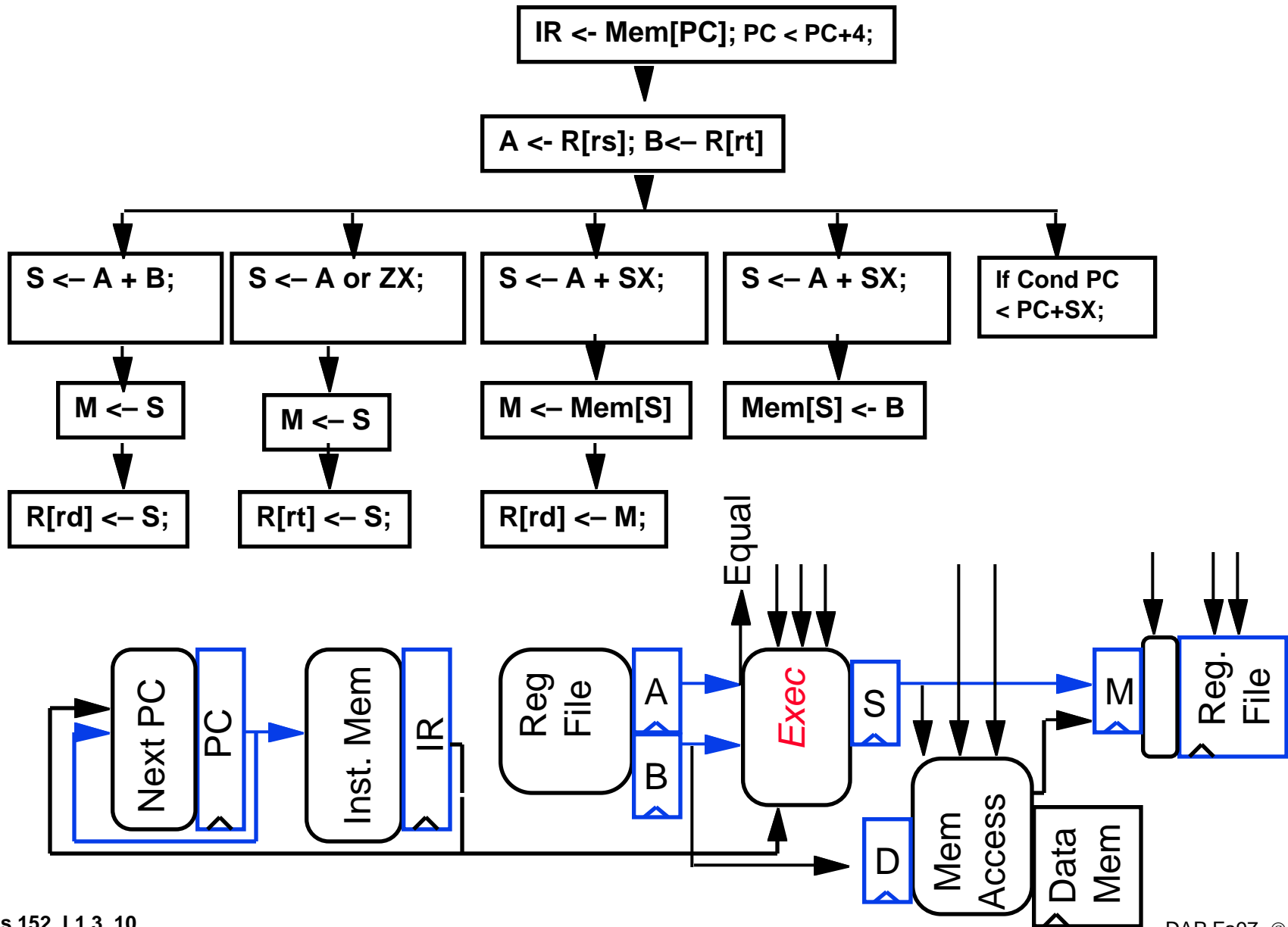
## ◦ The Five Classic Components of a Computer



## ◦ Today's Topics:

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

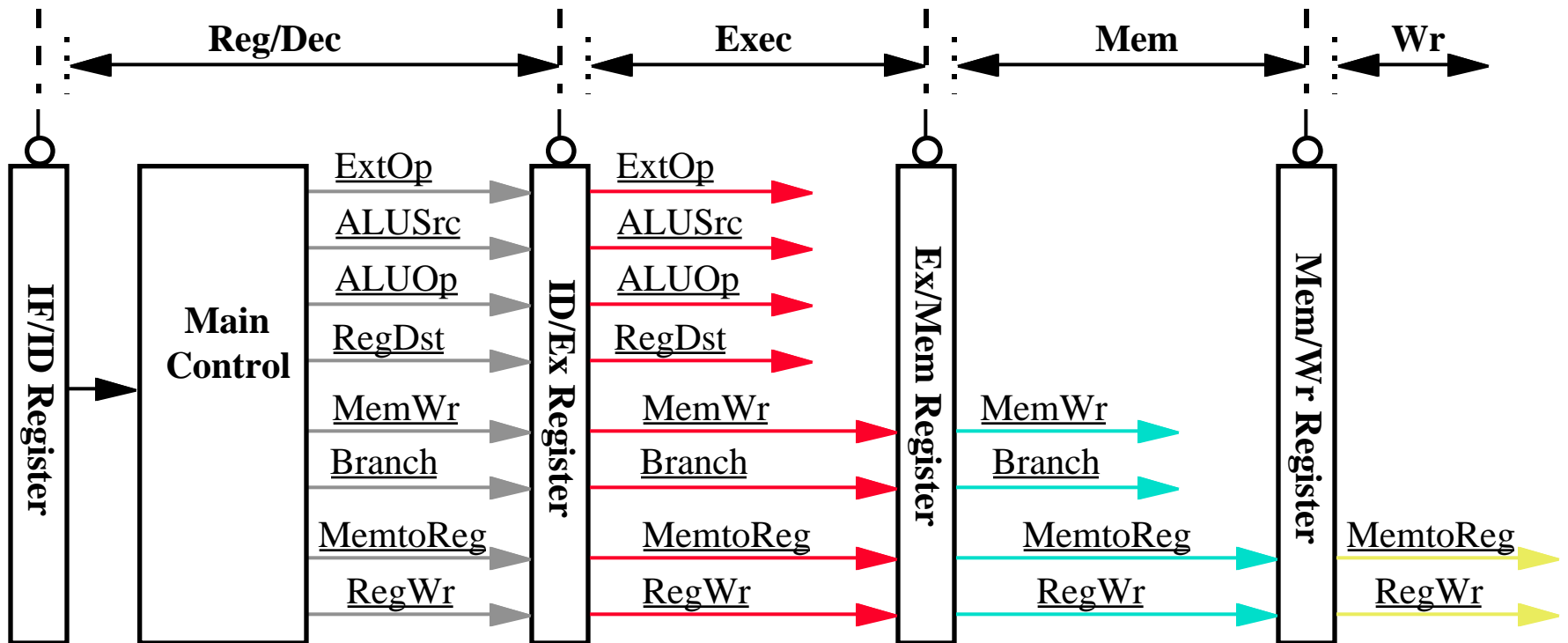
# Recap: Control Diagram



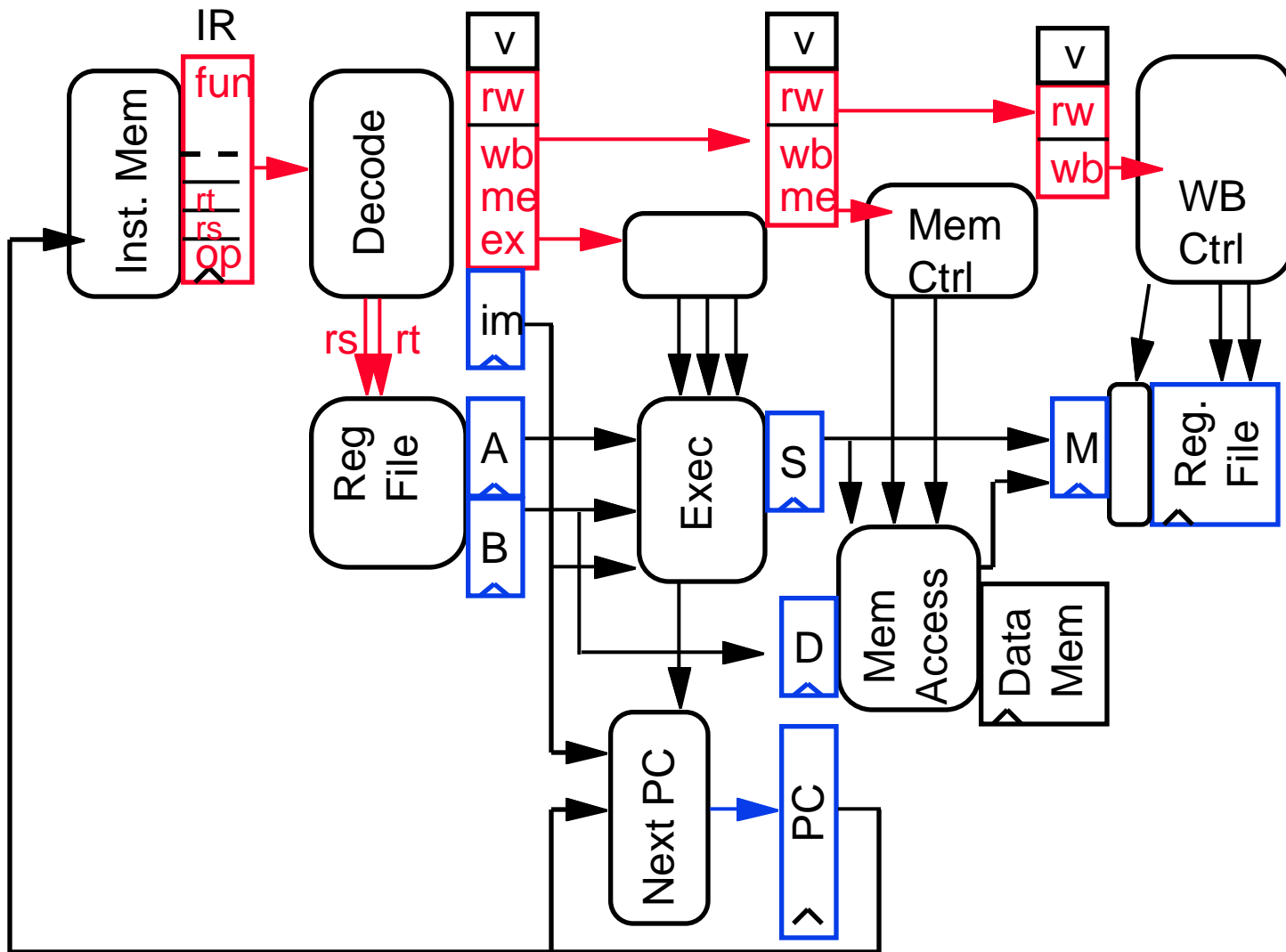
## But recall use of “Data Stationary Control”

### ◦ 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



## Let's Try it Out

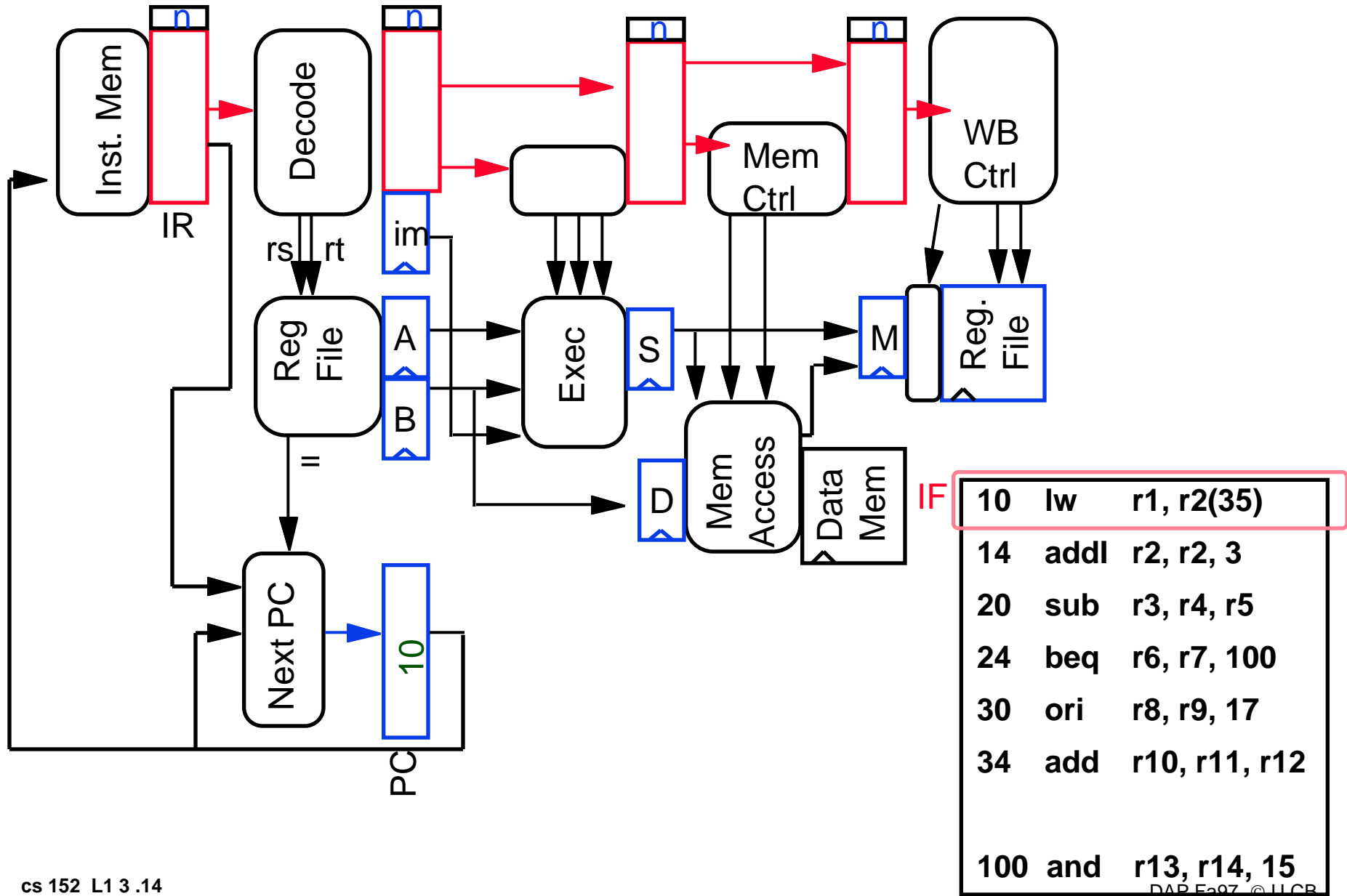
---

```
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

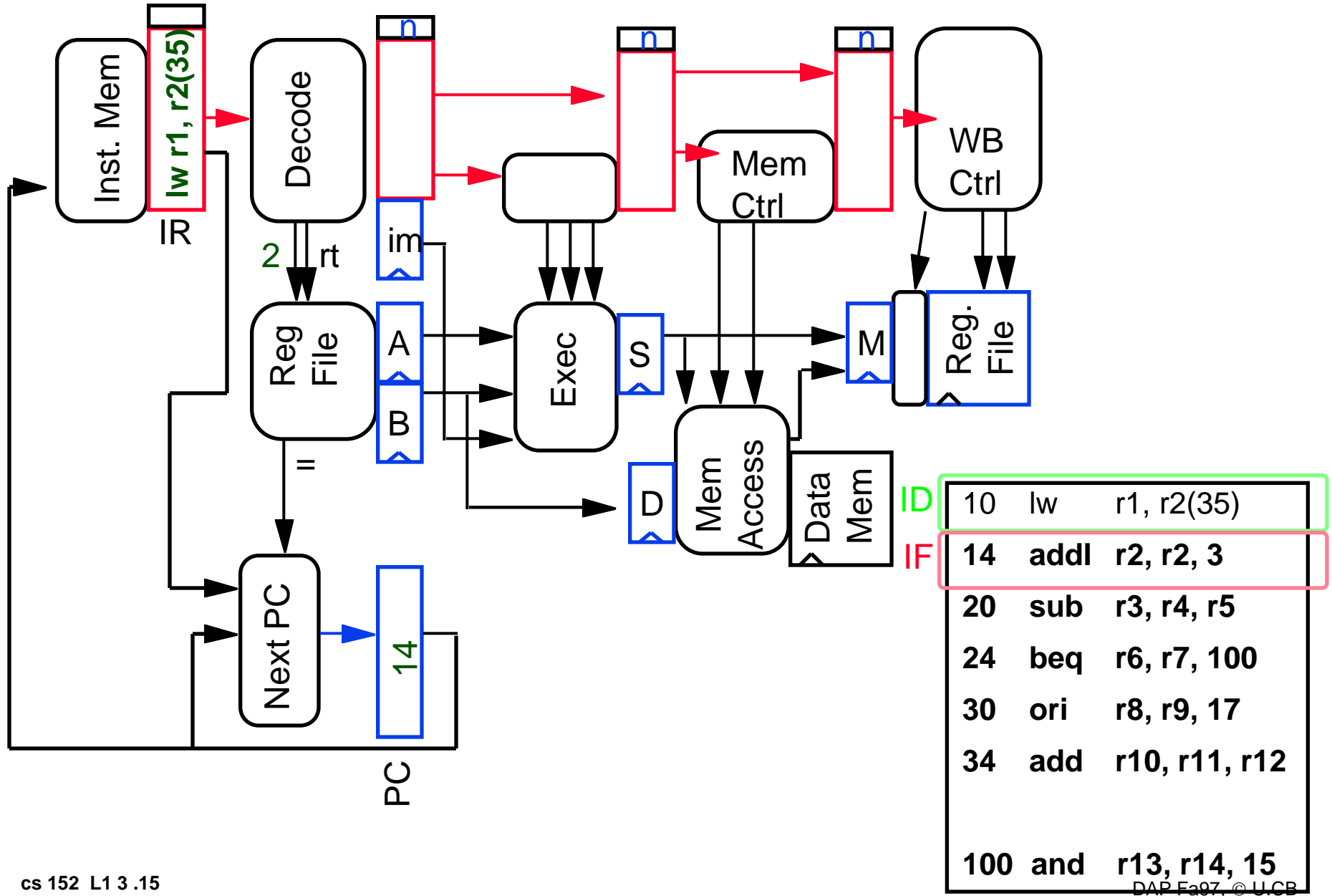
100   and   r13, r14, 15
```

these addresses are octal

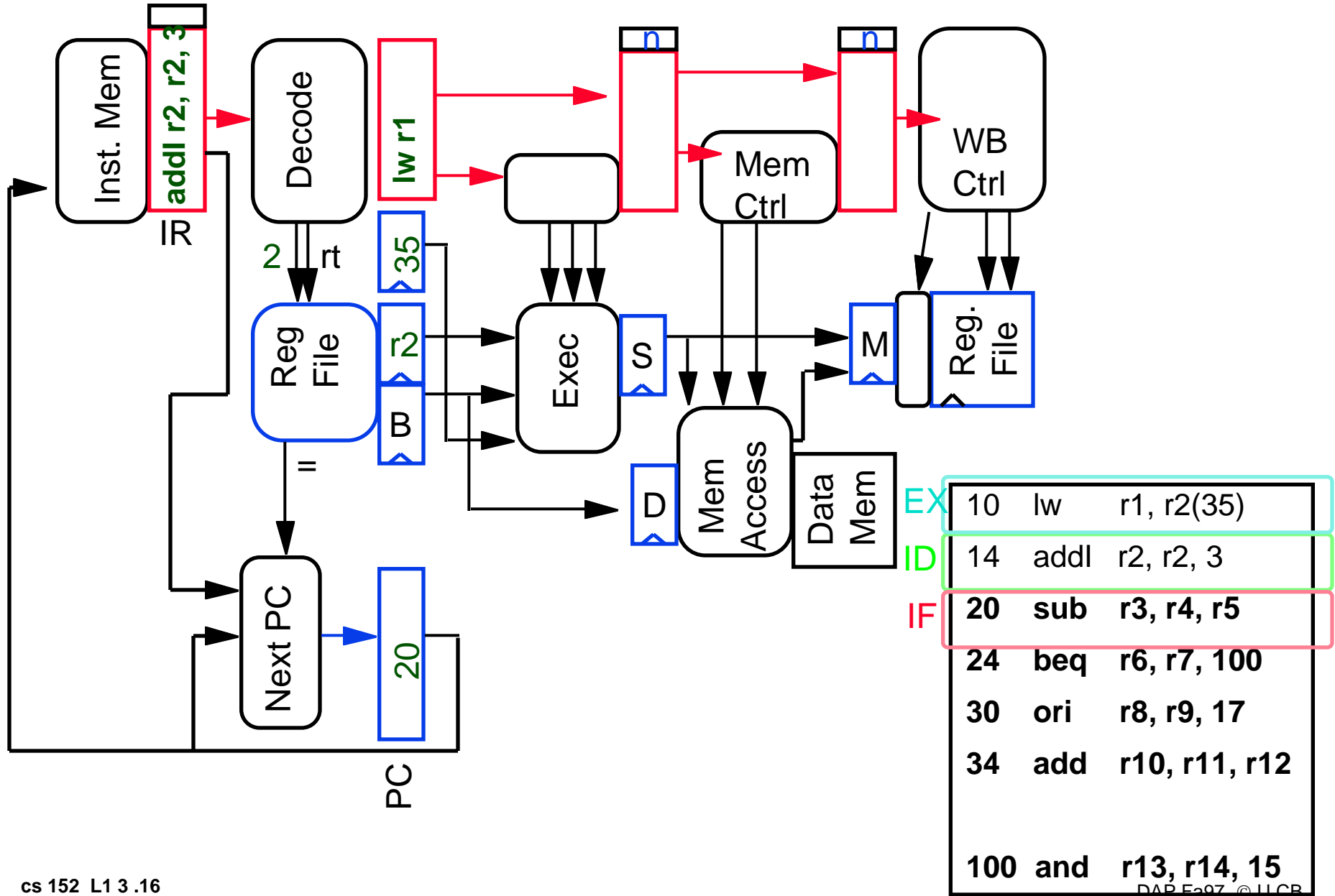
# Start: Fetch 10



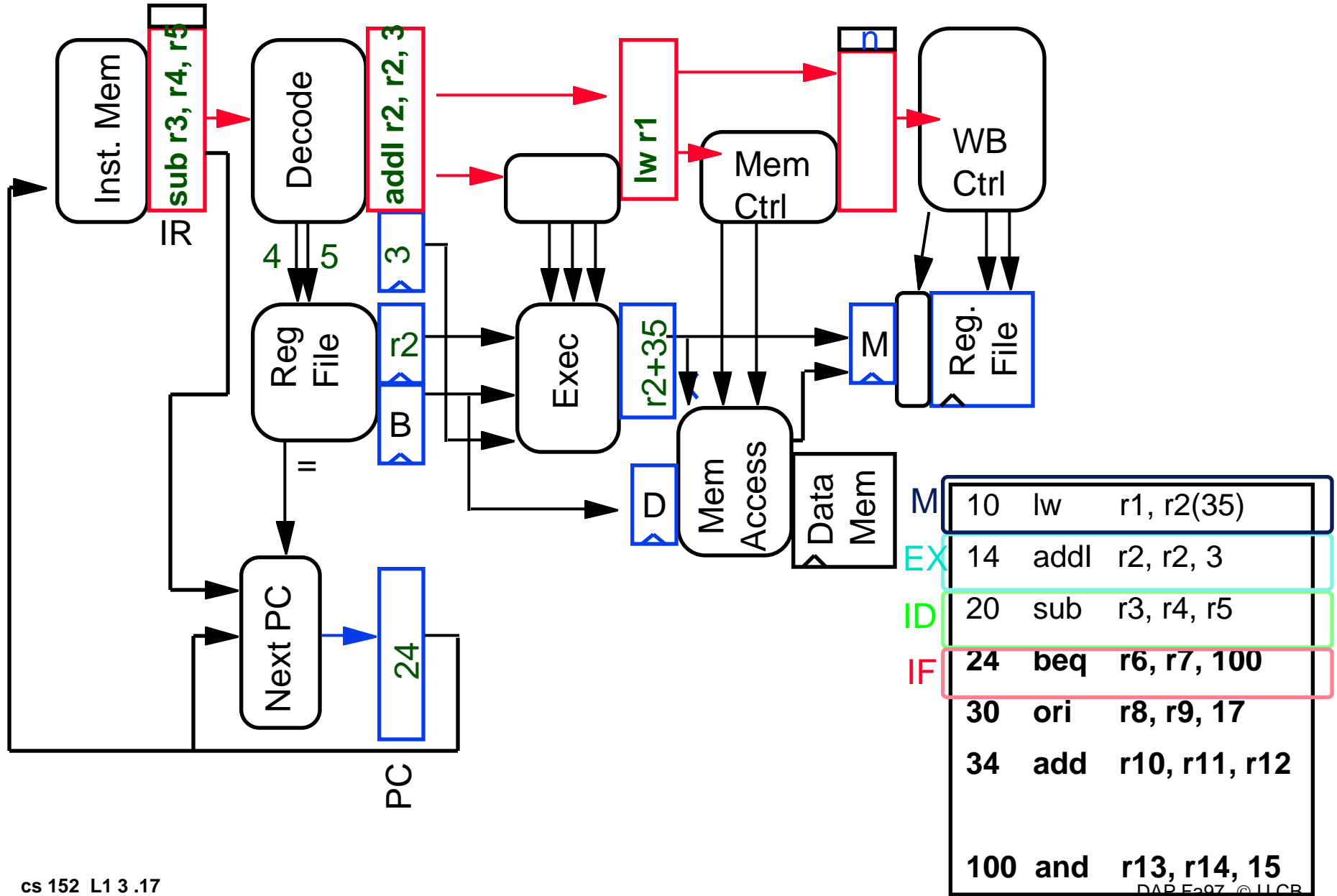
# Fetch 14, Decode 10



# Fetch 20, Decode 14, Exec 10



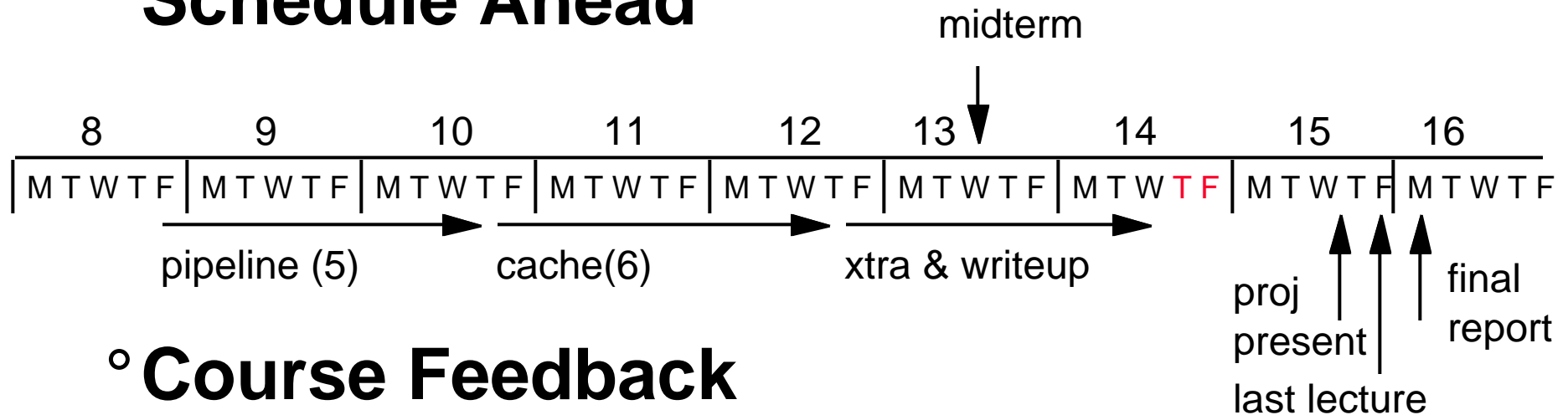
# Fetch 24, Decode 20, Exec 14, Mem 10



# Administrative Issues

---

## ◦ Schedule Ahead

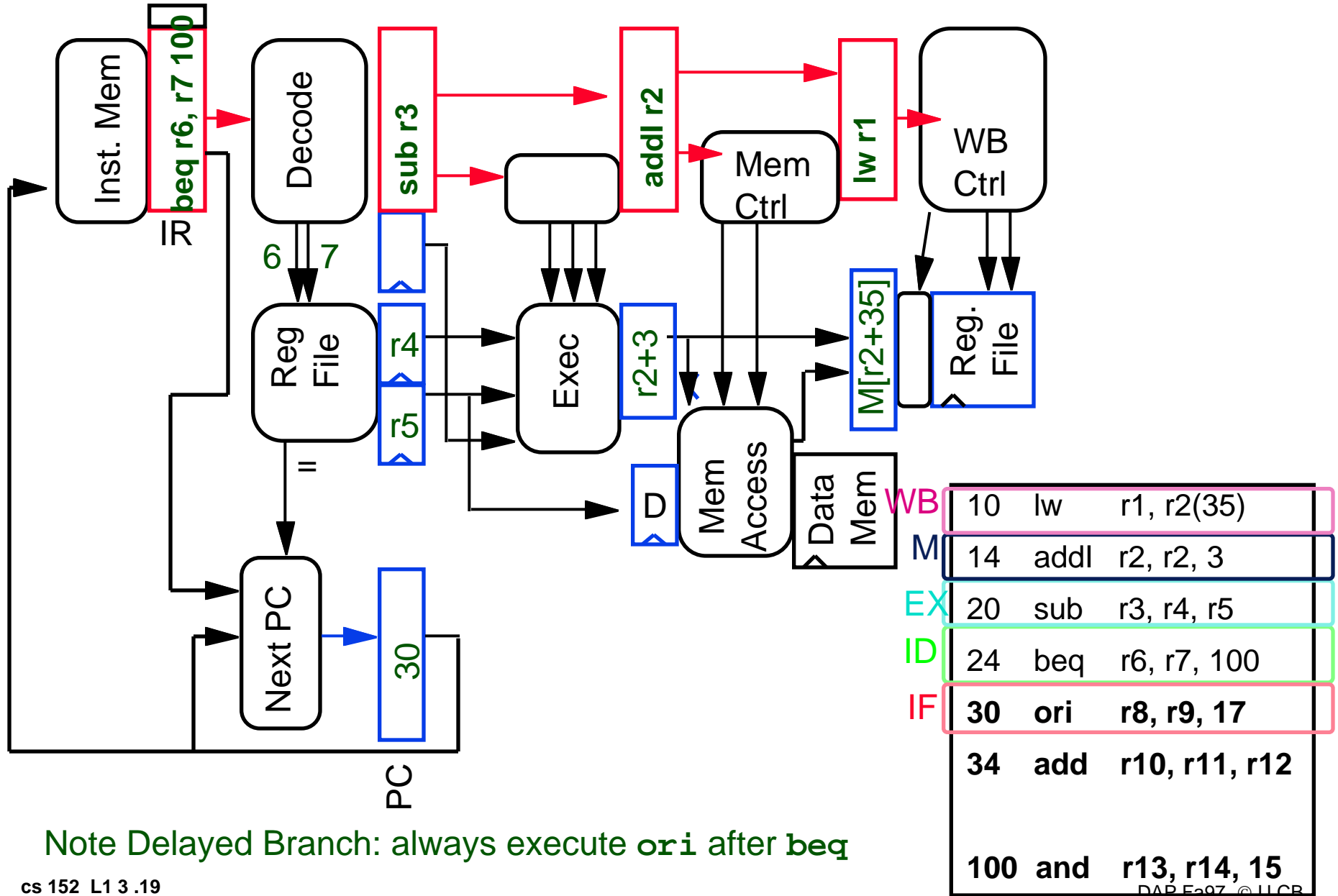


## ◦ Course Feedback

- Like on-line lecture notes!! pace of class!!
- Like Computers in the news!!
- Prerequisite Quiz? 39 great, 2 so-so, 1 bad idea
- Online Submission?
- Spread TA office hours?
- Slow lectures last 20 minutes?

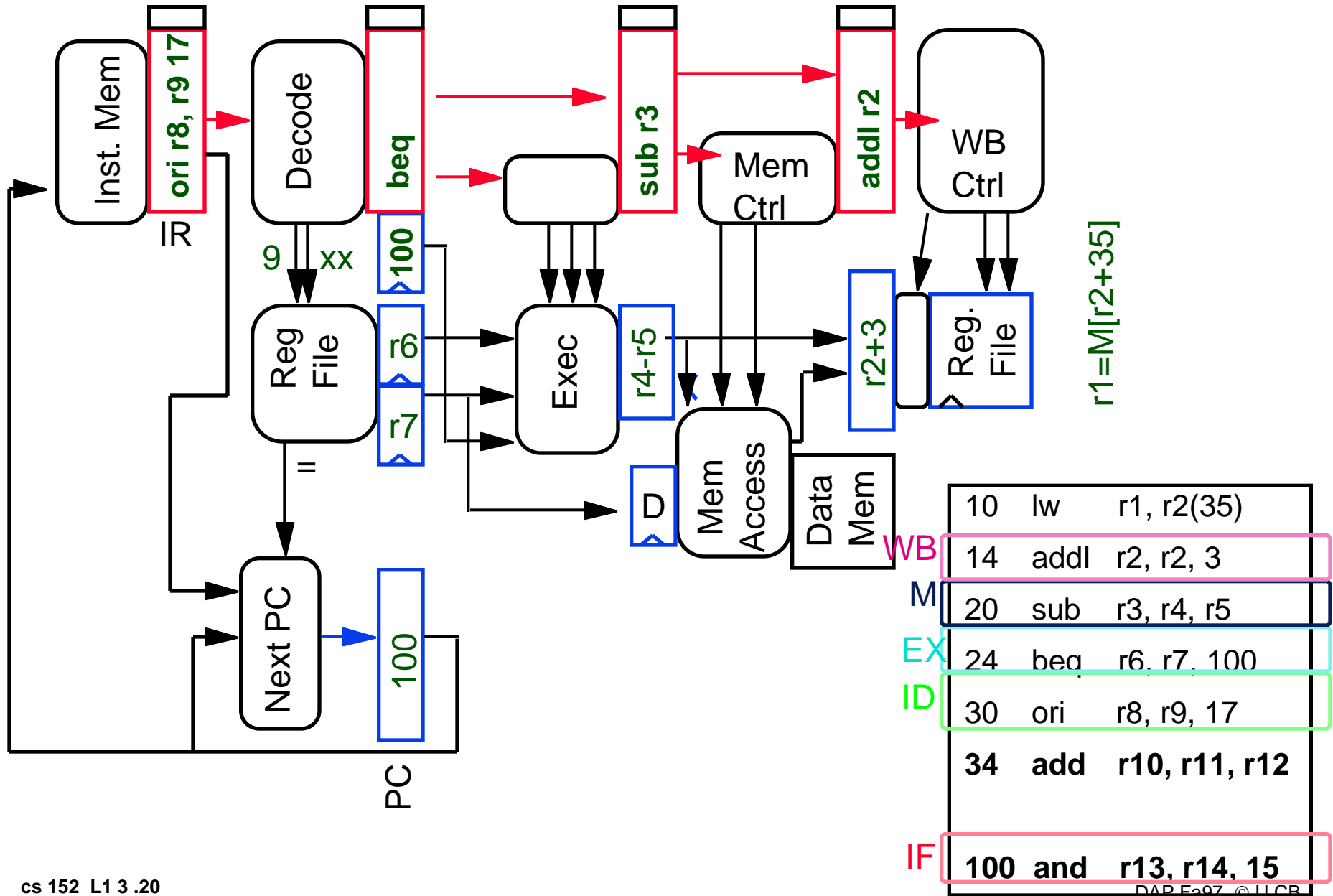
## ◦ Computers in the news:

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

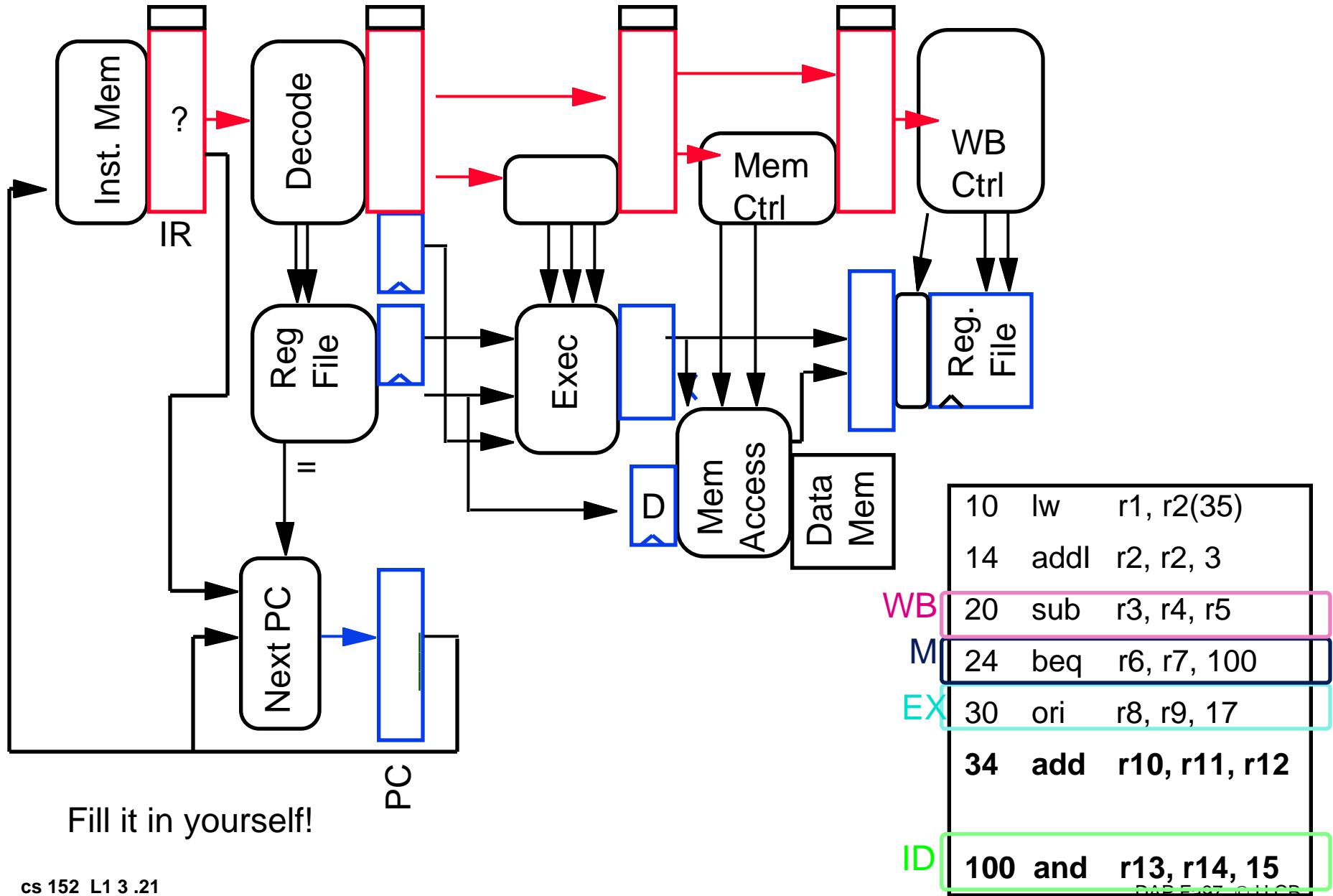


Note Delayed Branch: always execute `ori` after `beq`

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

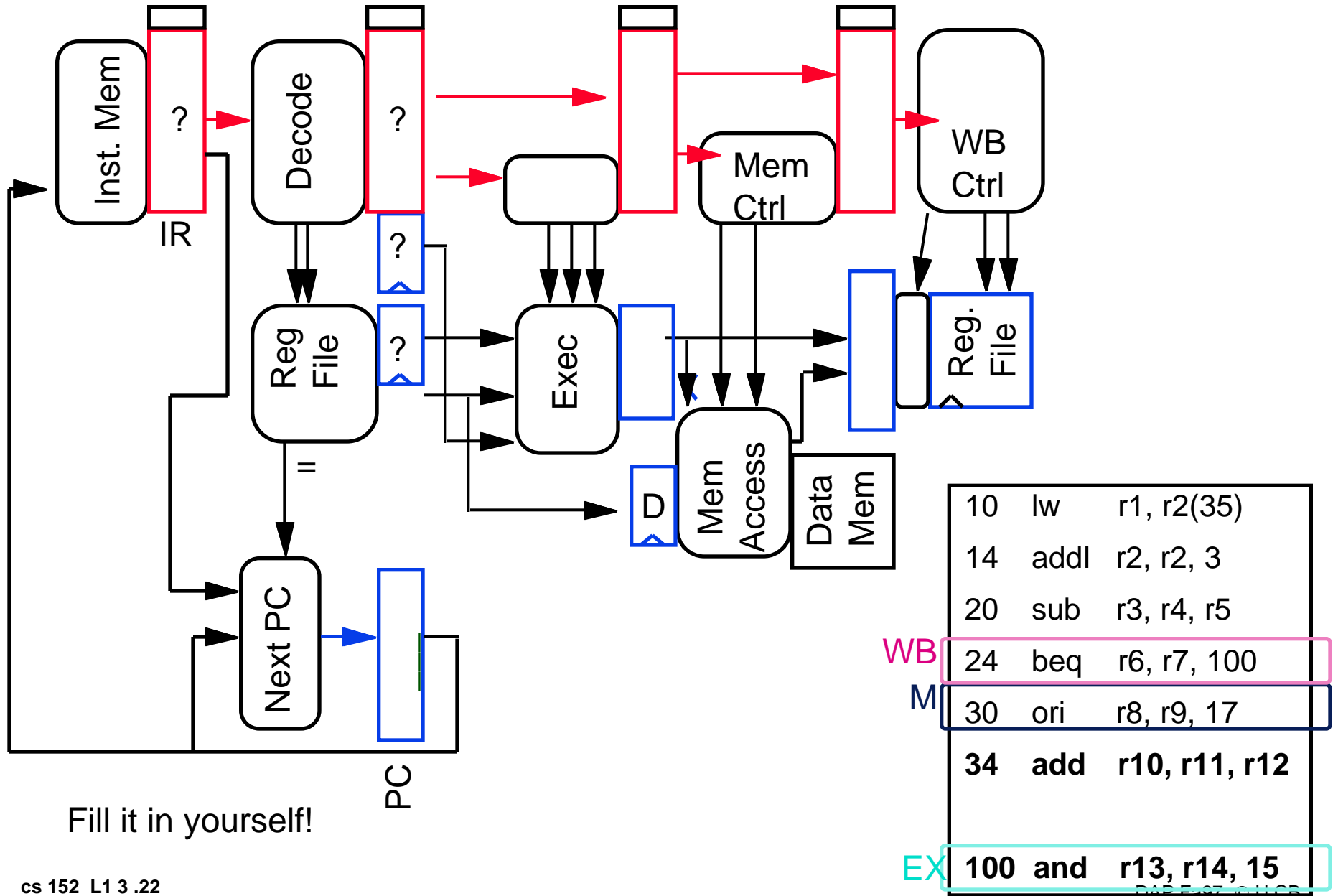


# Fetch 104, Dcd 100, Ex 30, Mem 24, WB 20

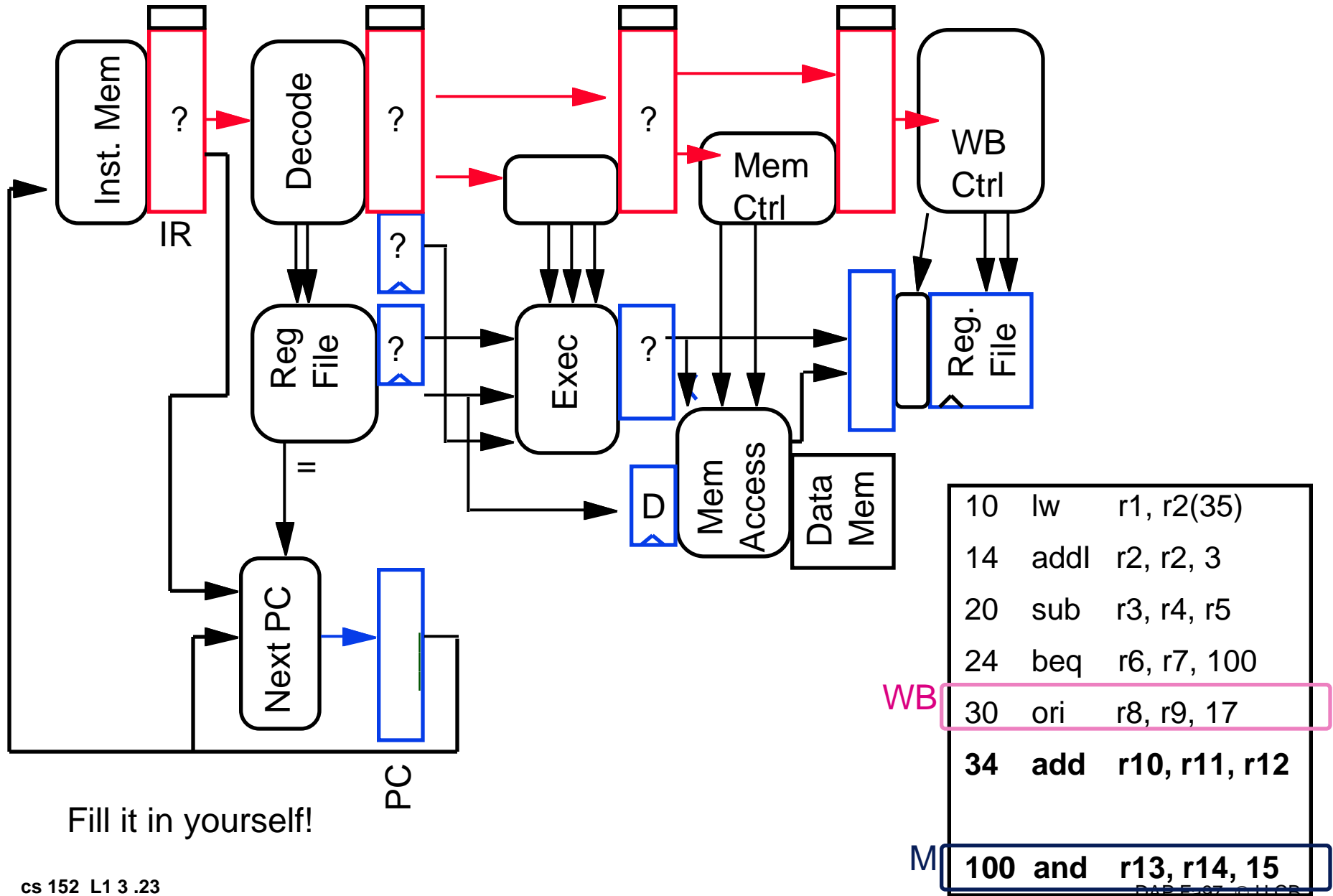


Fill it in yourself!

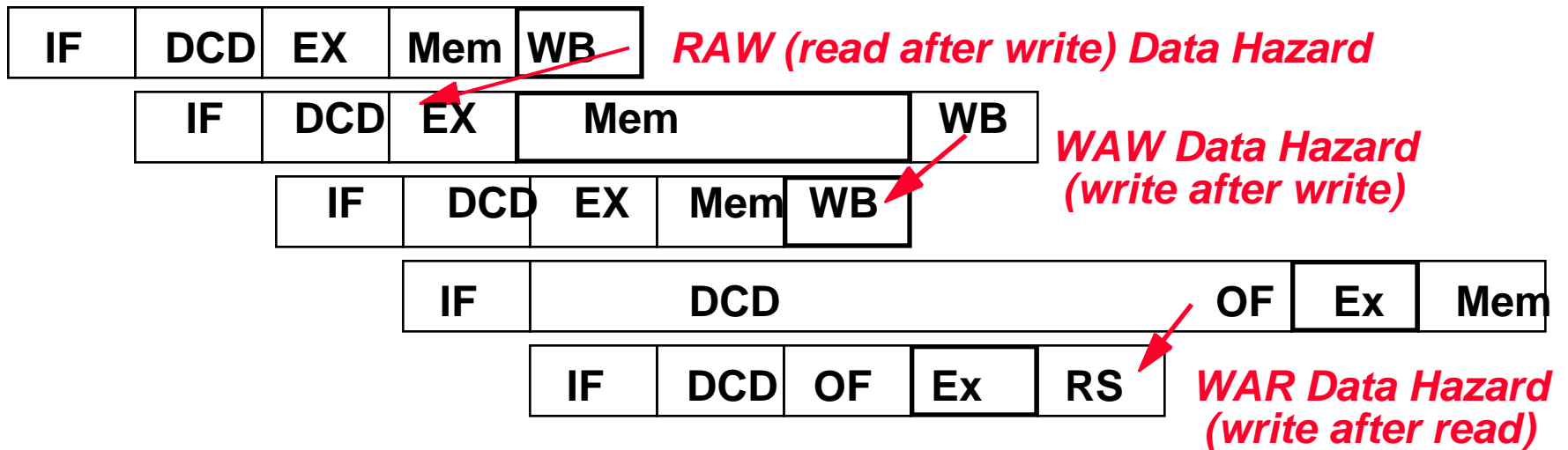
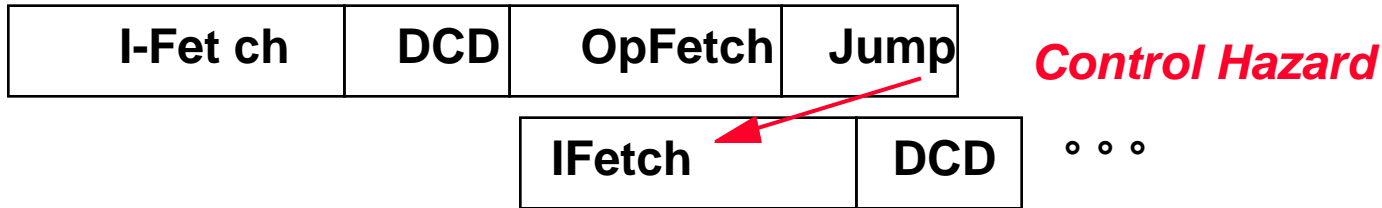
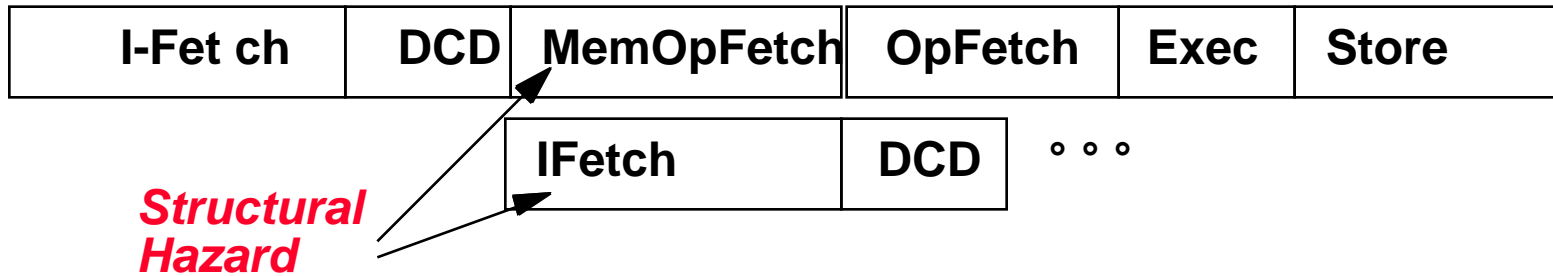
# Fetch 110, Dcd 104, Ex 100, Mem 30, WB 24



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



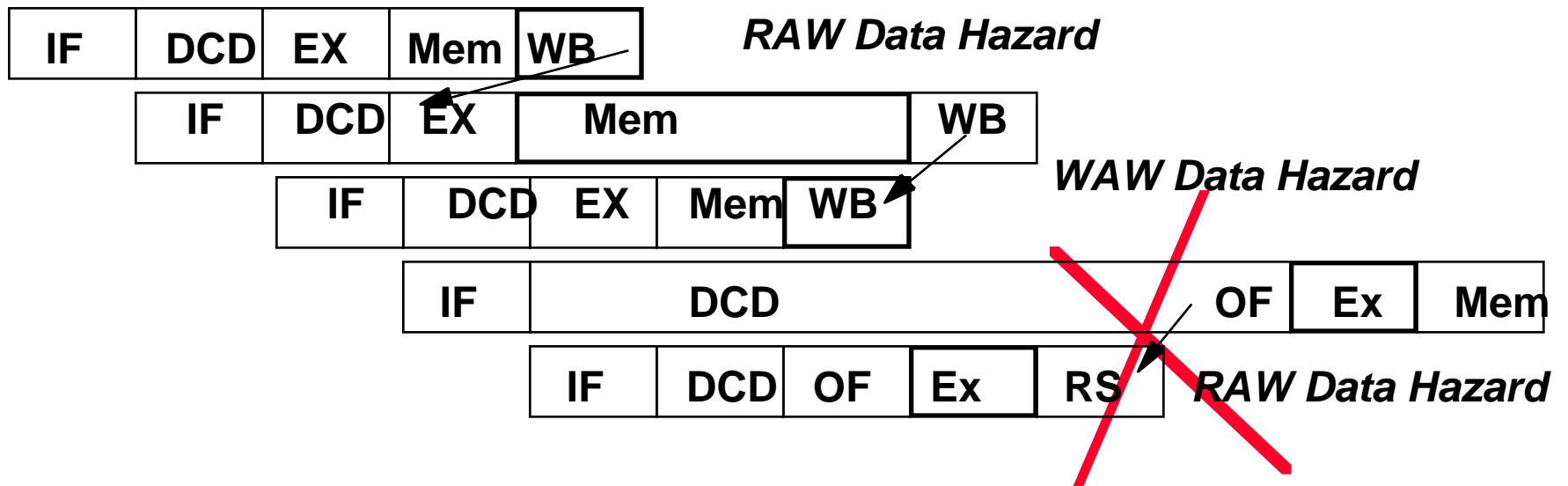
# Pipeline Hazards Again



# Data Hazards

---

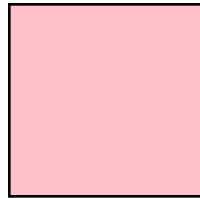
- Avoid some “by design”
  - eliminate WAR by always fetching operands early (DCD) in pipe
  - eliminate WAW by doing all WBs in order (last stage, static)
- Detect and resolve remaining ones
  - stall or forward (if possible)



# Hazard Detection

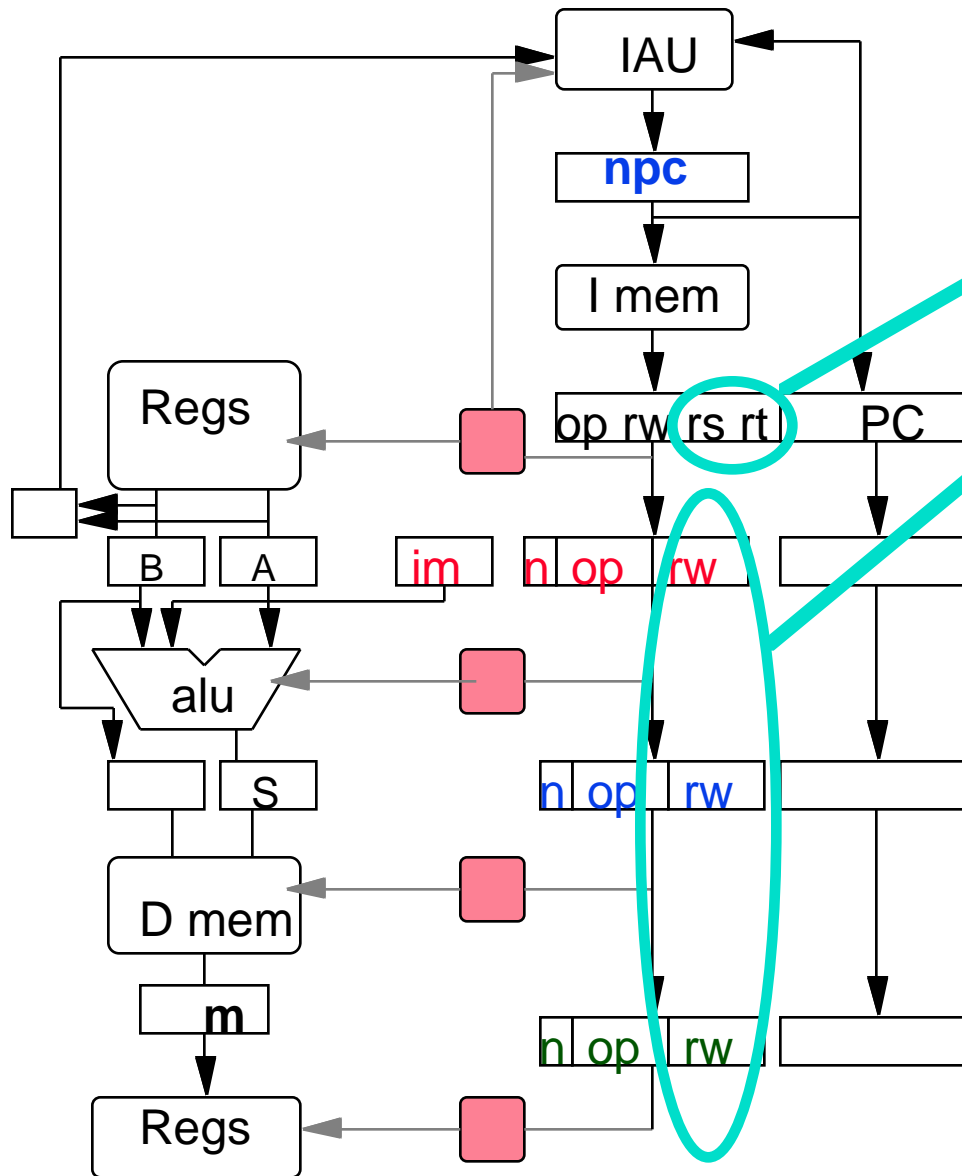
---

- 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 on 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



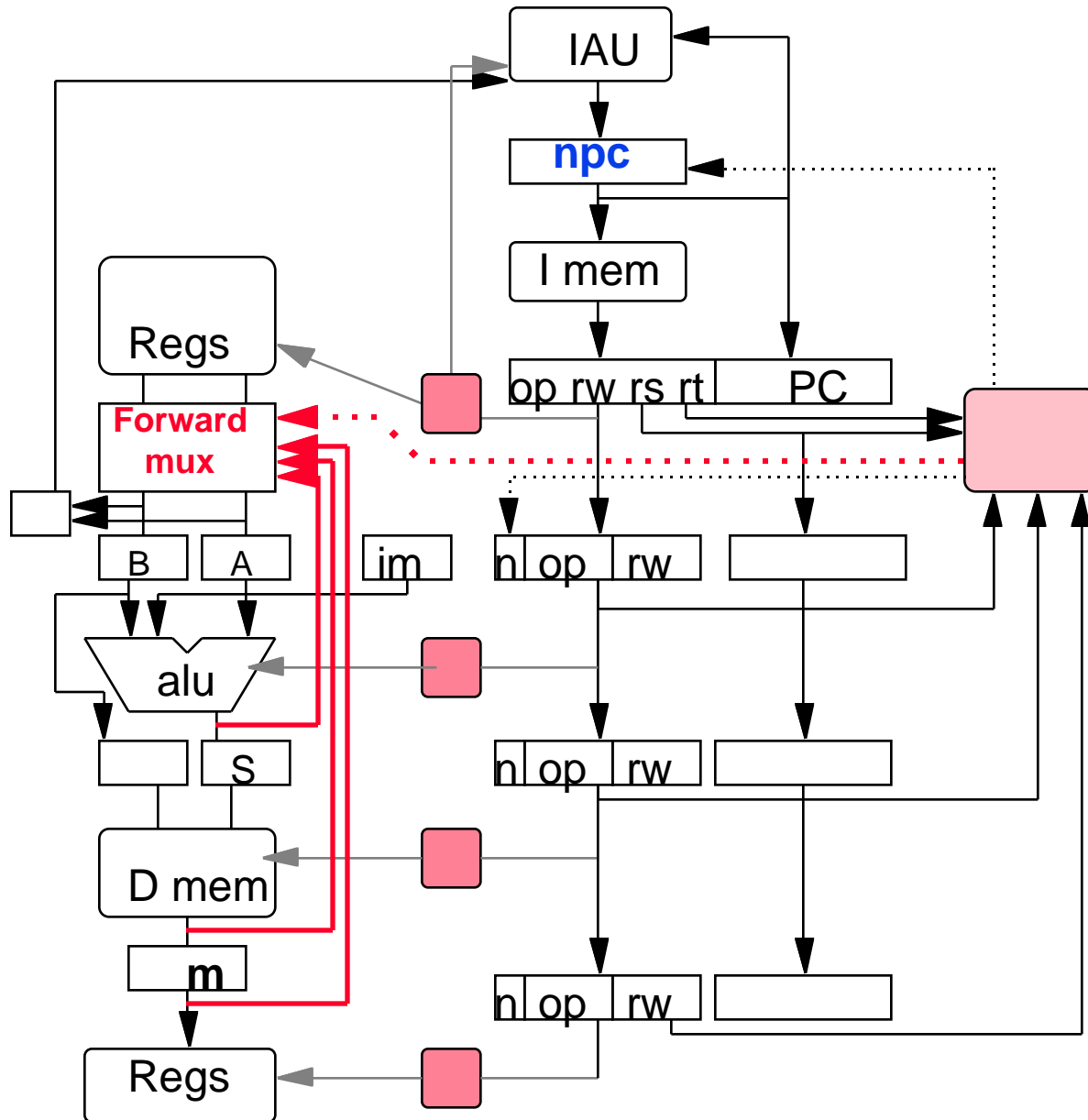
◦ Current operand registers

◦ Pending writes

◦ hazard  $\Leftarrow$

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

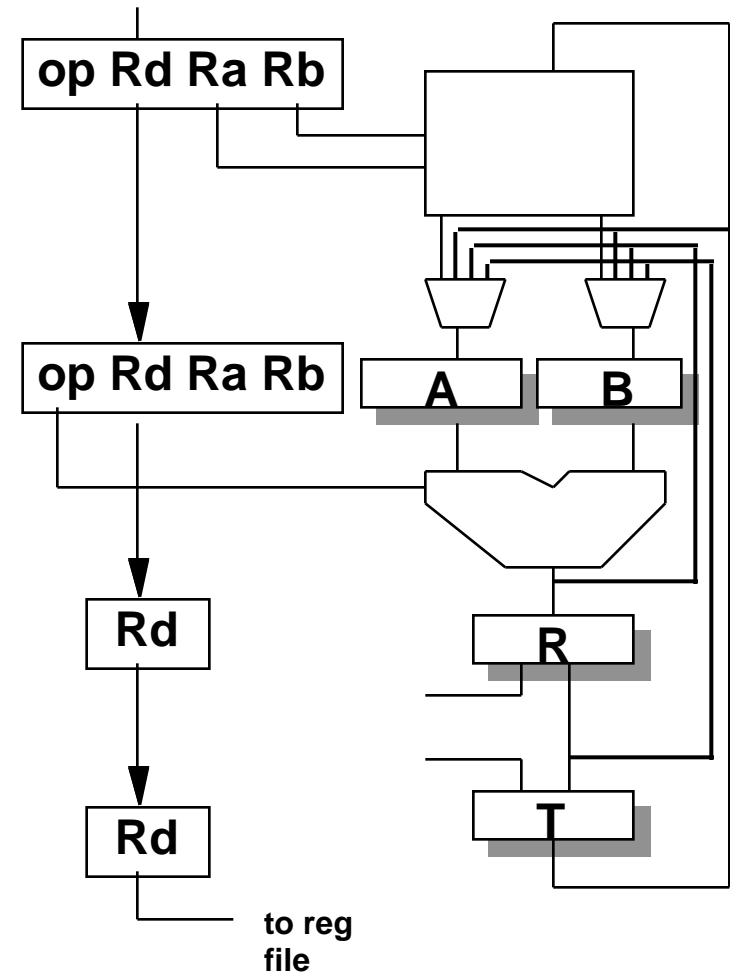
# Resolve RAW by forwarding



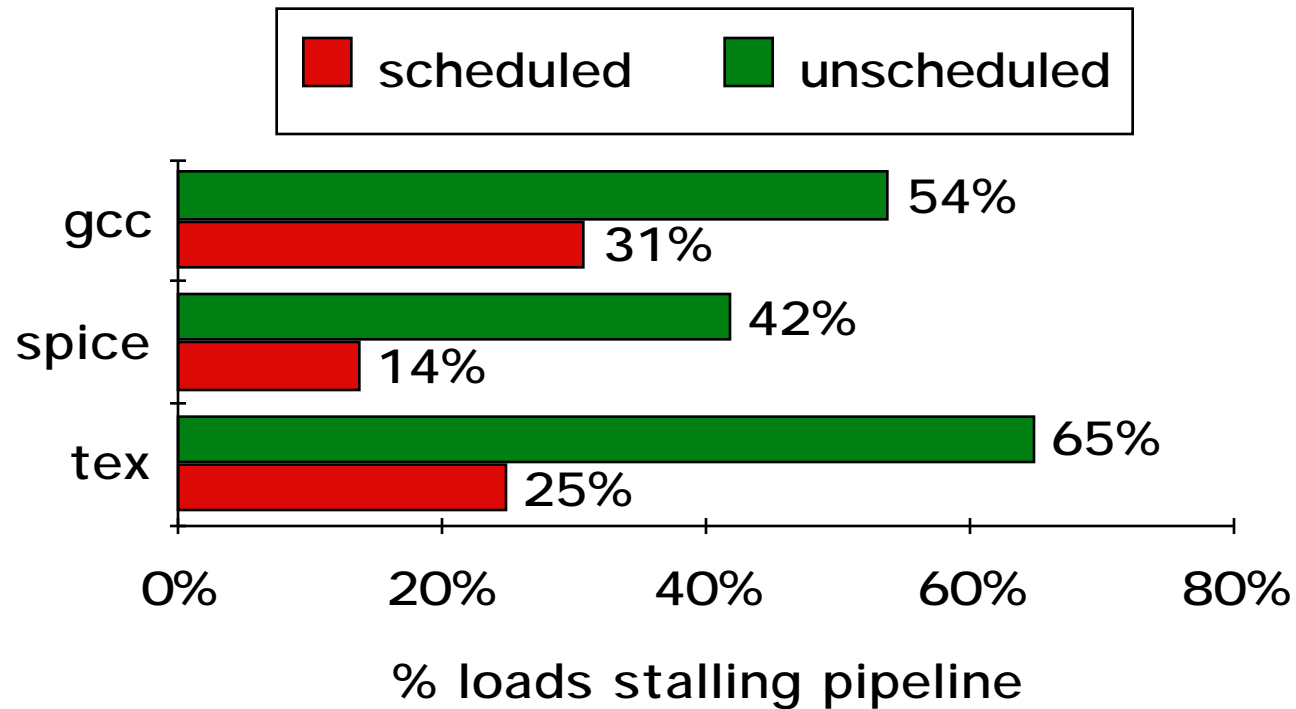
- Detect nearest **valid** write op operand register and **forward** into op latches, **bypassing** remainder of the pipe
- Increase muxes to add paths from pipeline registers
- **Data Forwarding = Data Bypassing**

# 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?
    - $R1 \leftarrow R4 + R5$
    - $R2 \leftarrow \text{Mem}[ R2 + I ]$
    - $R3 \leftarrow R2 + R1$
- => "Delayed Loads"



## Compiler Avoiding Load Stalls:



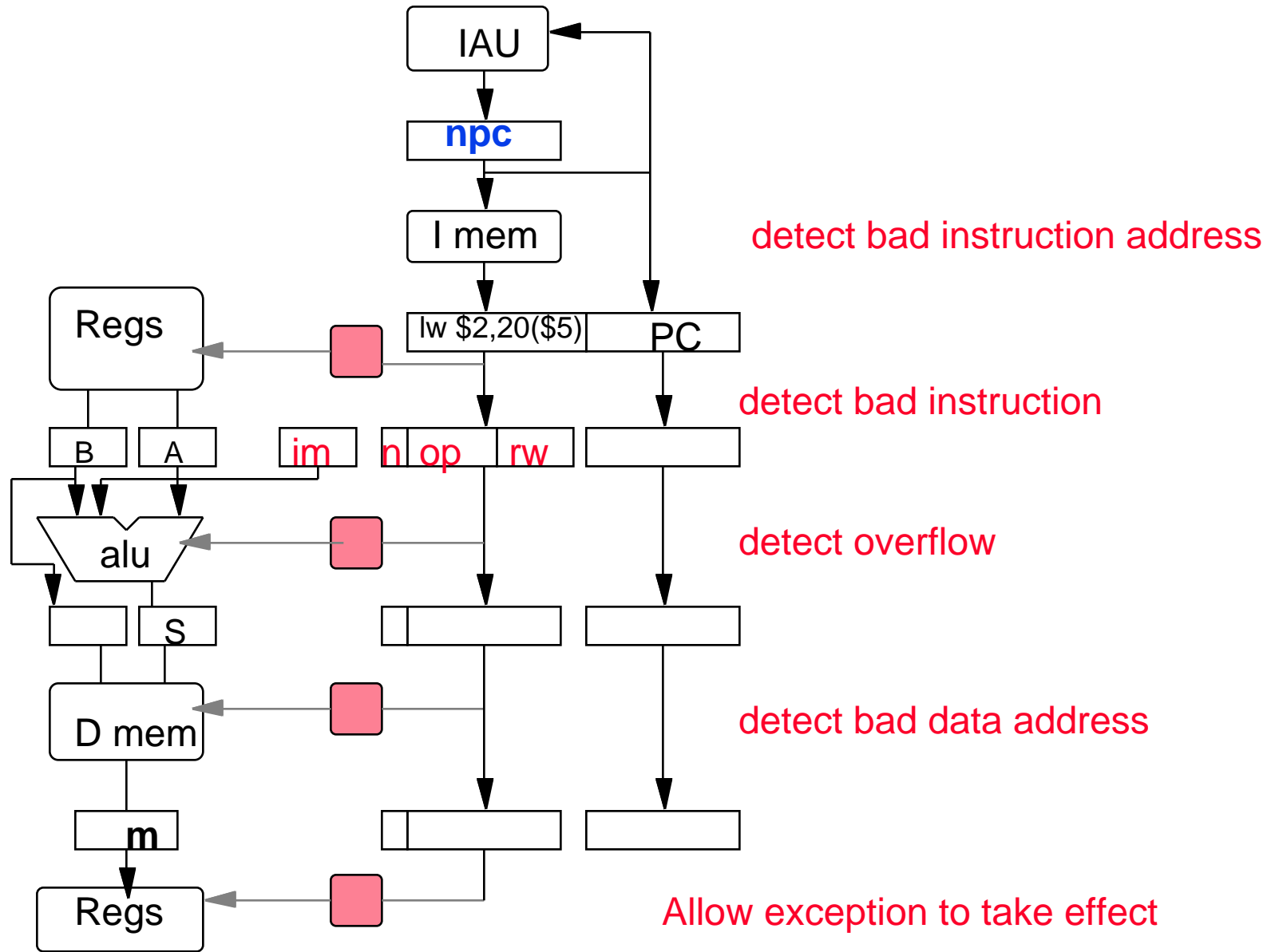
# What about Interrupts, Traps, Faults?

---

- **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

Refer to MIPS solution

# Exception Handling



# Exception Problem

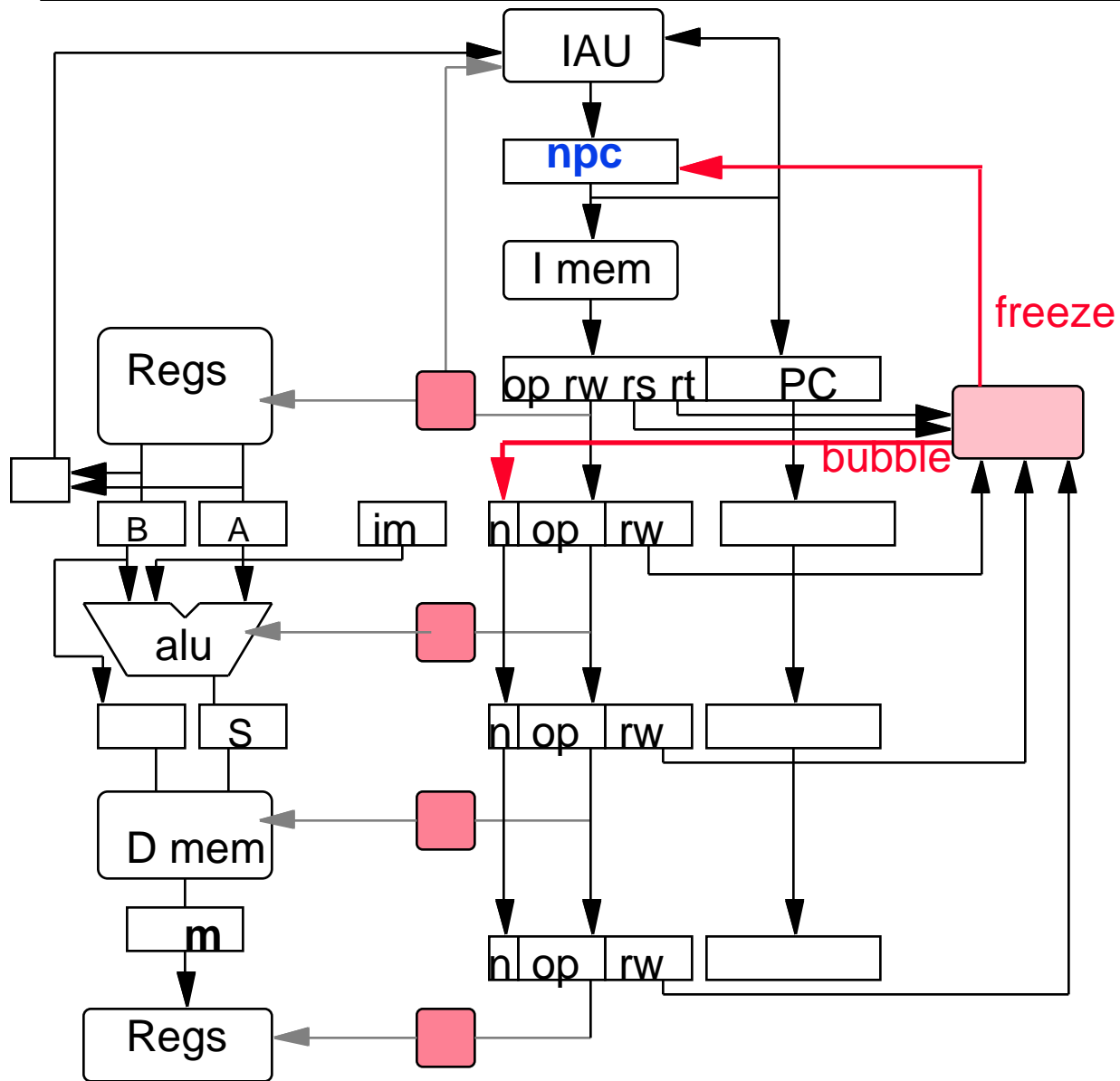
---

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

<i>Stage</i>	<i>Problem interrupts occurring</i>
<b>IF</b>	Page fault on instruction fetch; misaligned memory access; memory-protection violation
<b>ID</b>	Undefined or illegal opcode
<b>EX</b>	Arithmetic exception
<b>MEM</b>	Page fault on data fetch; misaligned memory access; memory-protection violation; memory error

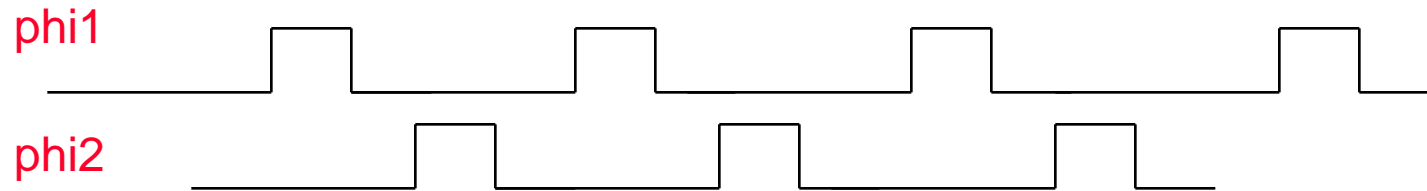
- Load with data page fault, Add with instruction page fault?
- Solution 1: interrupt vector/instruction, check last stage
- Solution 2: interrupt ASAP, restart everything incomplete

# Resolution: Freeze above & Bubble Below

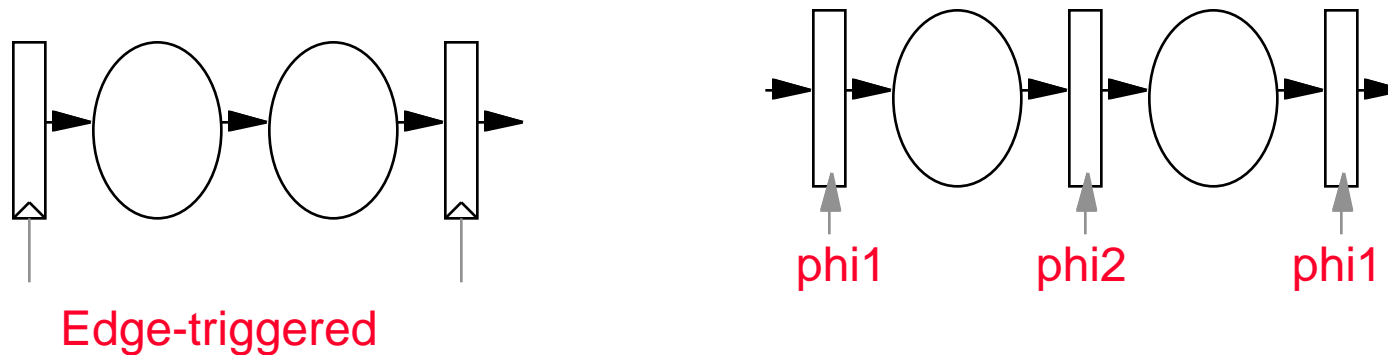


# FYI: MIPS R3000 clocking discipline

---



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



# MIPS R3000 Instruction Pipeline

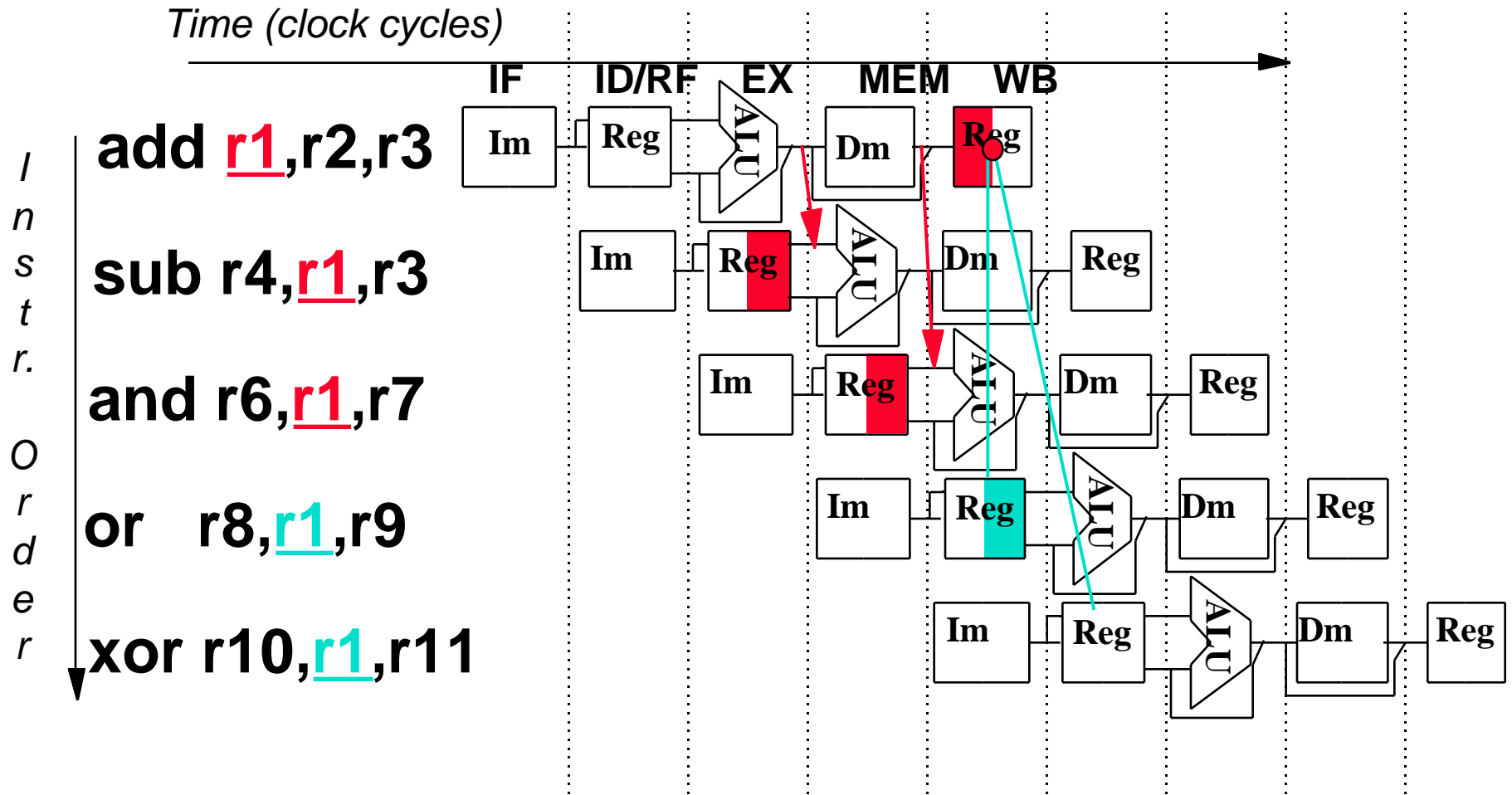
<b>Inst Fetch</b>	<b>Decode Reg. Read</b>	<b>ALU / E.A</b>	<b>Memory</b>	<b>Write Reg</b>
<b>TLB</b>	<b>I-Cache</b>	<b>RF</b>	<b>Operation</b>	<b>WB</b>
		<b>E.A.</b>	<b>TLB</b>	<b>D-Cache</b>

## Resource Usage

<b>TLB</b>				<b>TLB</b>										
	<b>I-cache</b>													
		<b>RF</b>					<b>WB</b>							
			<b>ALU</b>	<b>ALU</b>										
					<b>D-Cache</b>									

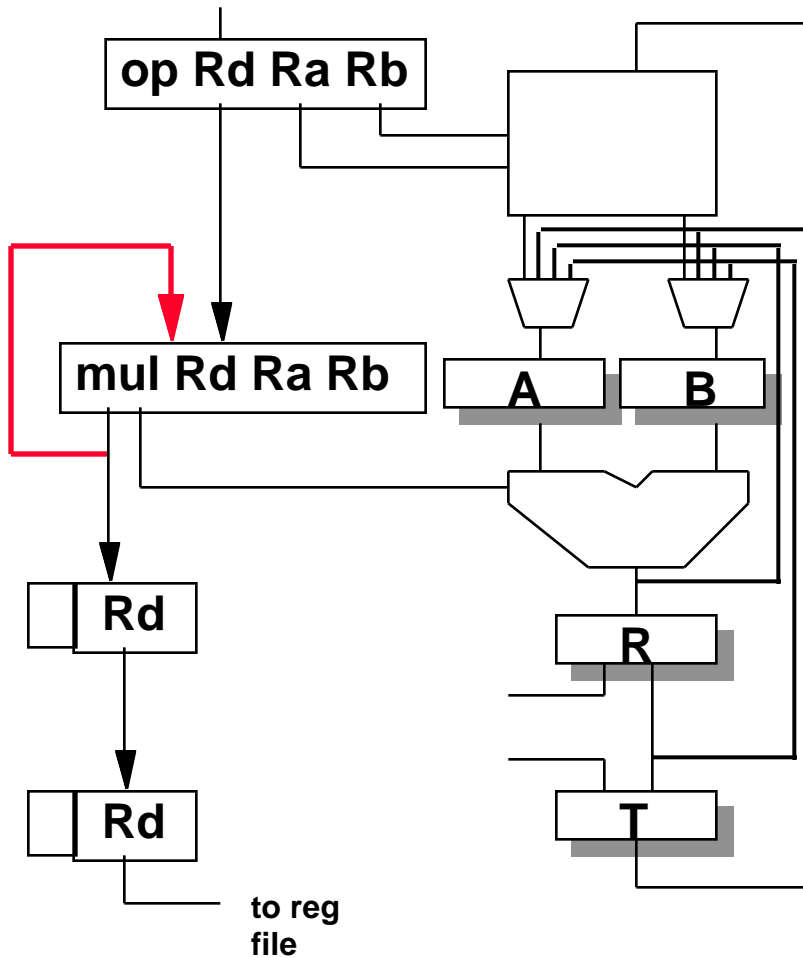
Write in phase 1, read in phase 2 => eliminates bypass from WB

# 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**

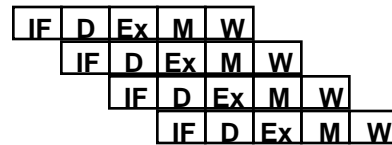
**Drain (bubble) stages below it**

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

# Issues in Pipelined design

---

◦ **Pipelining**

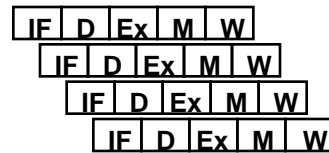


**Limitation**

Issue rate, FU stalls, FU depth

◦ **Super-pipeline**

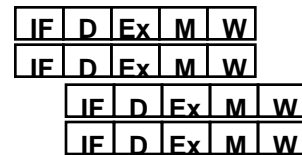
- Issue one instruction per (fast) cycle
- ALU takes multiple cycles



Clock skew, FU stalls, FU depth

◦ **Super-scalar**

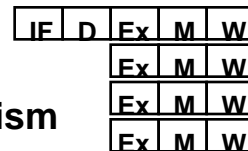
- Issue multiple scalar instructions per cycle



Hazard resolution

◦ **VLIW ("EPIC")**

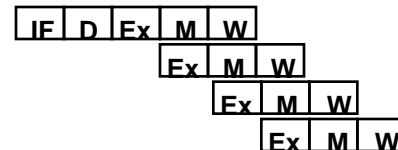
- Each instruction specifies multiple scalar operations
- Compiler determines parallelism



Packing

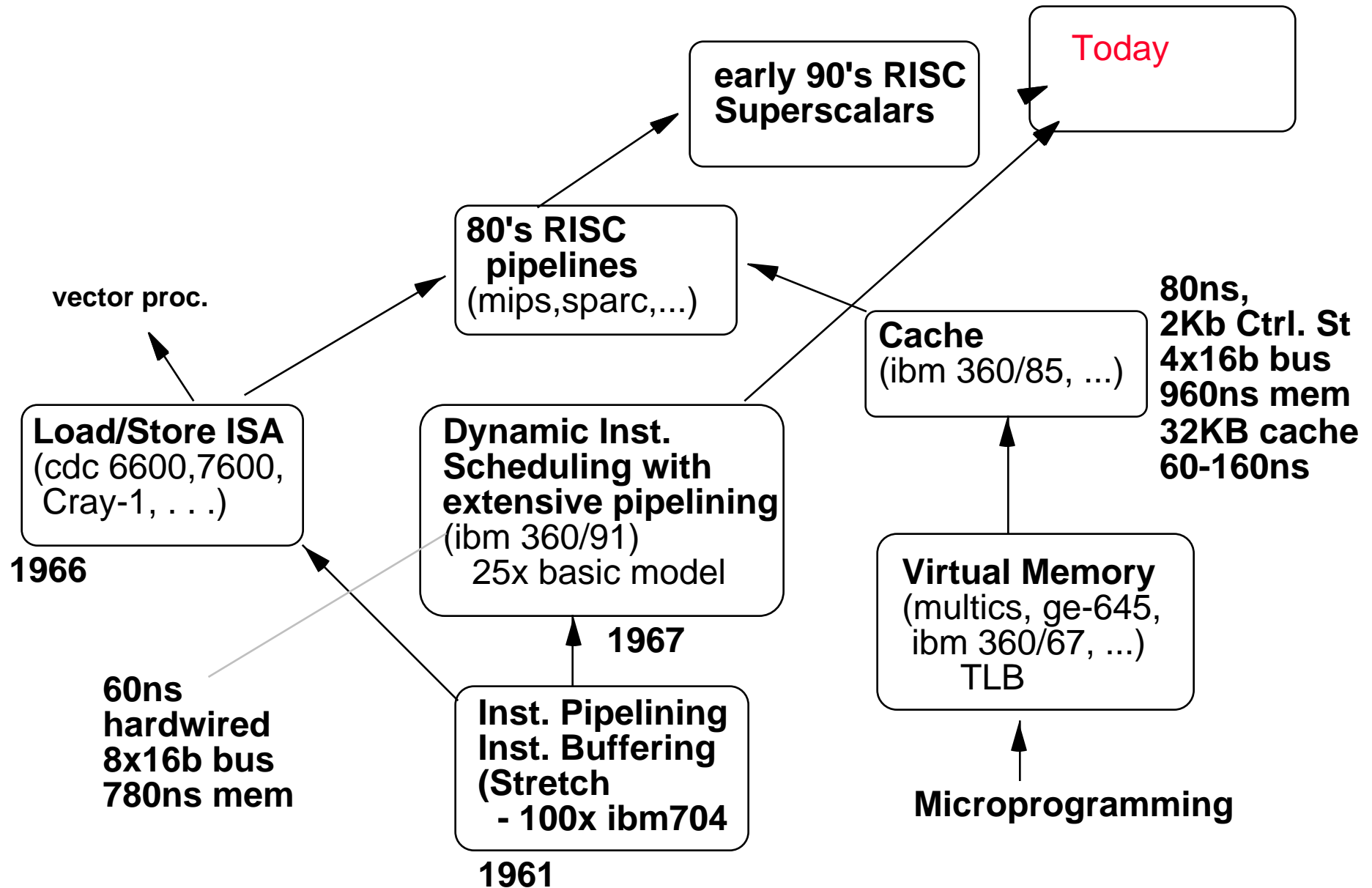
◦ **Vector operations**

- Each instruction specifies series of identical operations

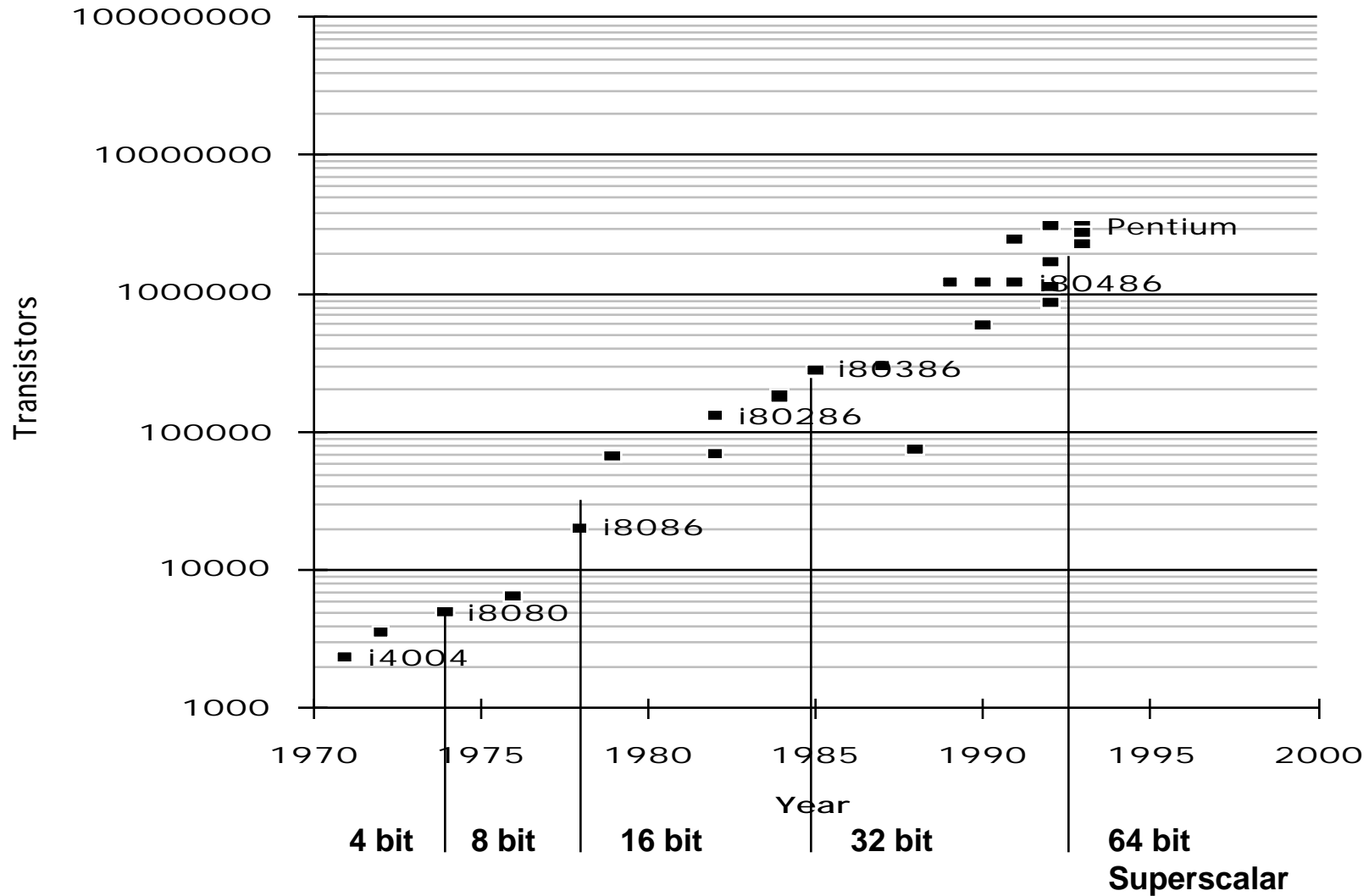


Applicability

# Historical Perspective

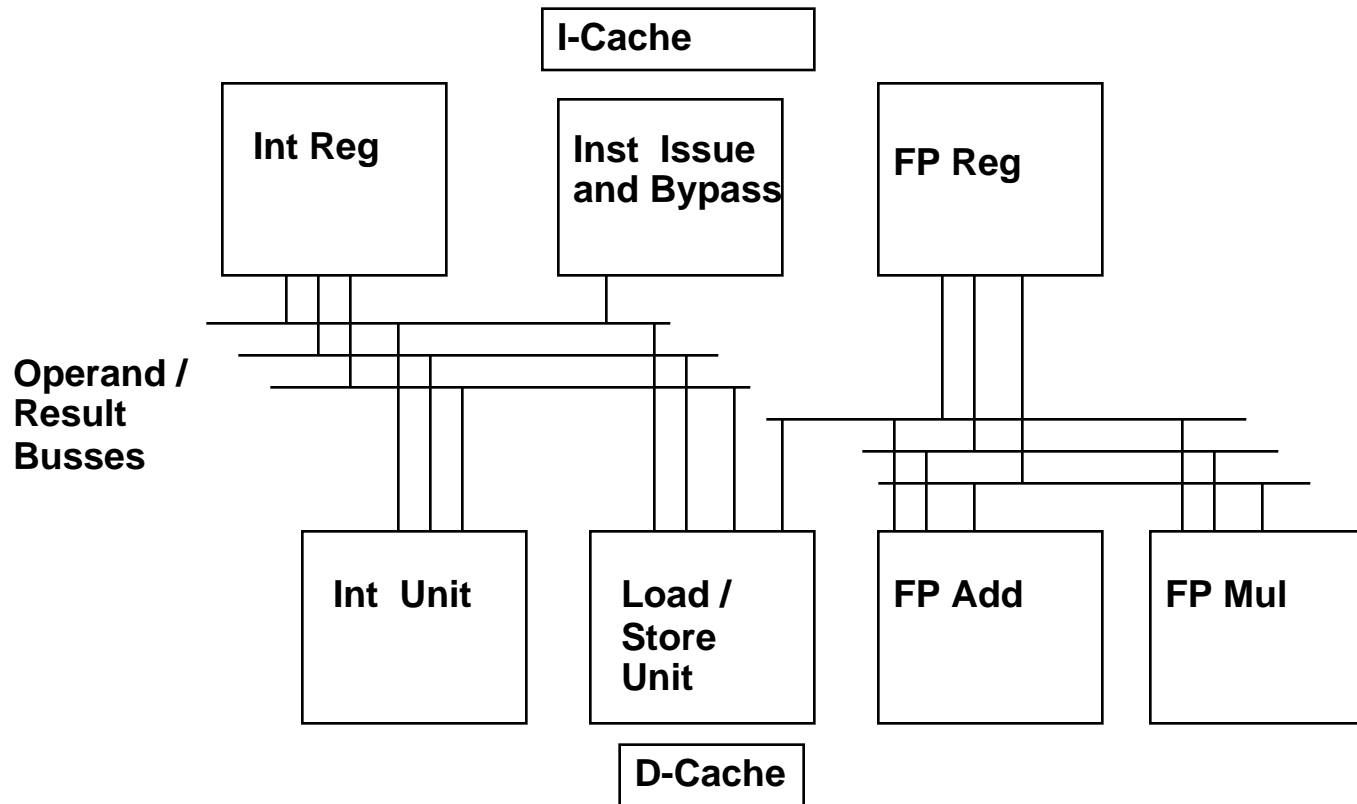


# Technology Perspective



# 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:	Cycles	Assumptions
load Ra <- Ai	1	
load Ry <- Yi	1	
fmult Rm <- Ra*Rx	1+ 6	6 cycle mult, 3 stage
fadd Rs <- Rm+Ry	1 + 4	4 cycle add, 2 stage
store Ai <- Rs	1	
inc Yi	1	
dec i	1	
inc Ai	1	
branch	1	

**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		
addr <- 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	

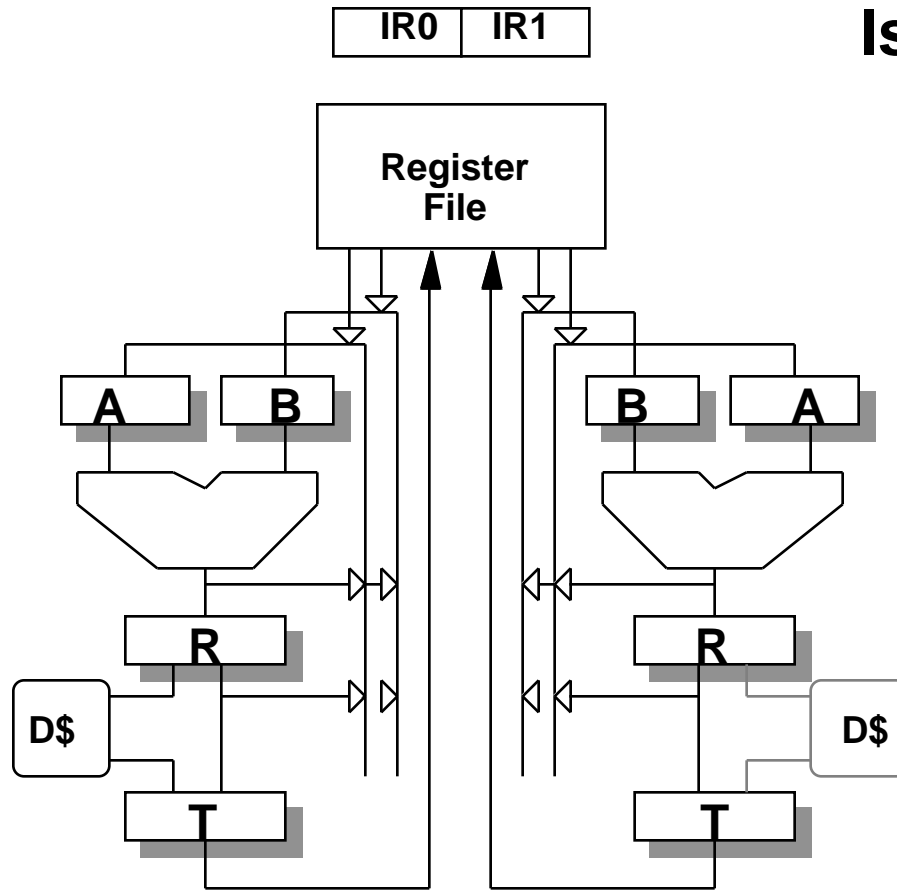
## Pipelined Loop:

```

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**

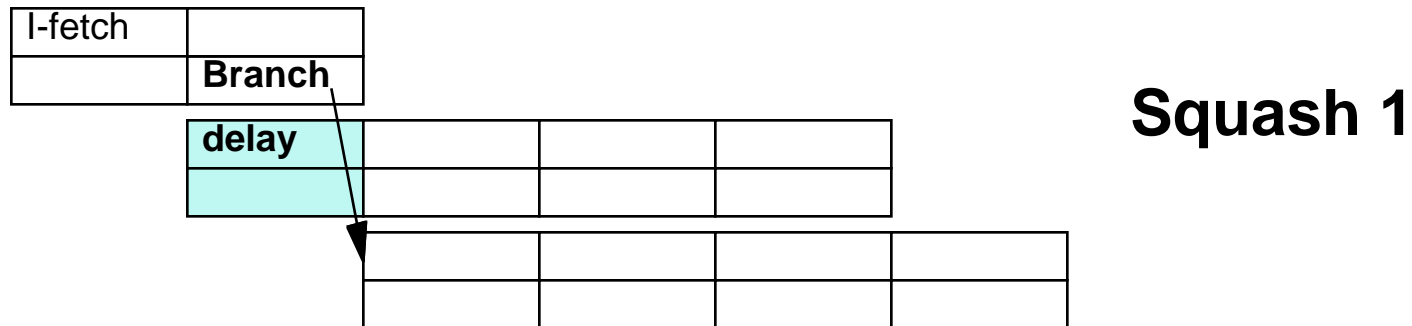
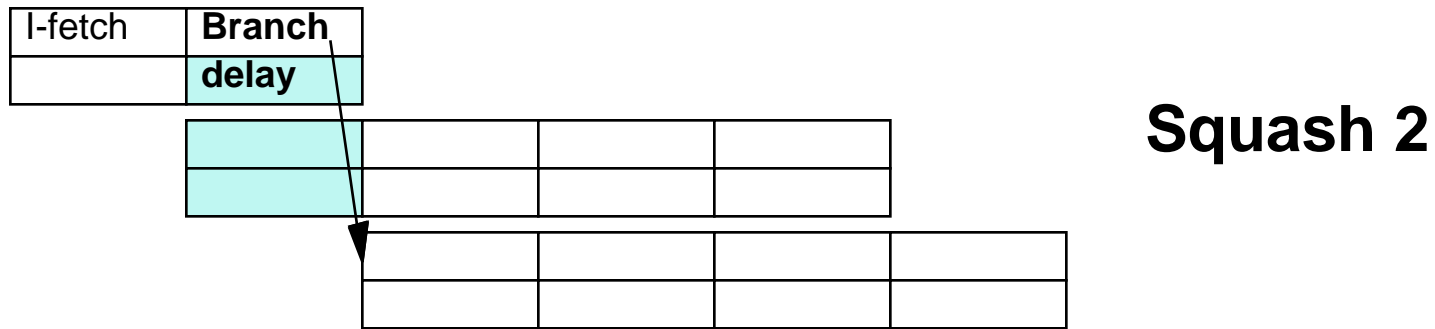
**Multiple load/store ops?**

**Branches**

# Branch penalties in superscalar

---

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



# Summary

---

- **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**