

# **ECE 252 / CPS 220**

## **Advanced Computer Architecture I**

### **Lecture 6**

### **Pipelining – Part 1**

Benjamin Lee  
Electrical and Computer Engineering  
Duke University

[www.duke.edu/~bcl15](http://www.duke.edu/~bcl15)  
[www.duke.edu/~bcl15/class/class\\_ece252fall11.html](http://www.duke.edu/~bcl15/class/class_ece252fall11.html)



# ECE252 Administrivia

## 29 September – Homework #2 Due

- Use blackboard forum for questions
- Attend office hours with questions
- Email for separate meetings

## 4 October – Class Discussion

Roughly one reading per class. Do not wait until the day before!

1. Srinivasan et al. "Optimizing pipelines for power and performance"
2. Mahlke et al. "A comparison of full and partial predicated execution support for ILP processors"
3. Palacharla et al. "Complexity-effective superscalar processors"
4. Yeh et al. "Two-level adaptive training branch prediction"



# Pipelining

$$\text{Latency} = (\text{Instructions} / \text{Program}) \times (\text{Cycles} / \text{Instruction}) \times (\text{Seconds} / \text{Cycle})$$

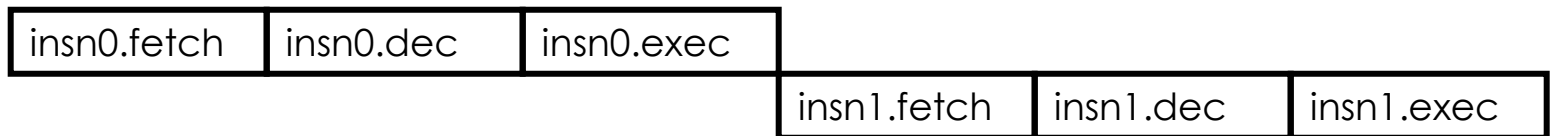
## Performance Enhancement

- Increases number of cycles per instruction
- Reduces number of seconds per cycle

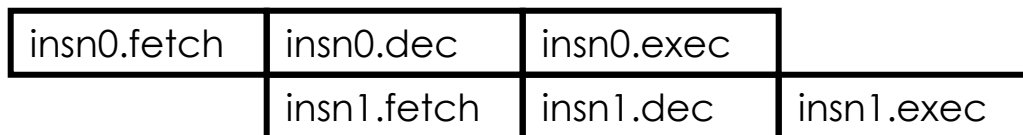
## Instruction-Level Parallelism

- Begin with multi-cycle design
- When one instruction advances from stage-1 to stage=2, allow next instruction to enter stage-1.
- Individual instructions require the same number of stages
- Multiple instructions in-flight, entering and leaving at faster rate

**Multi-cycle**

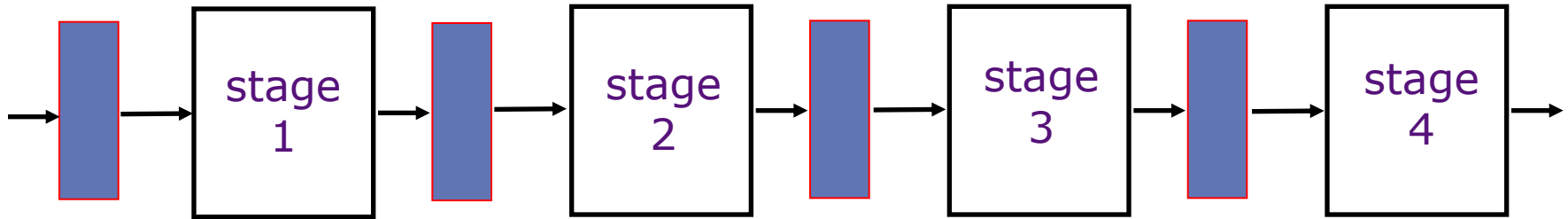


**Pipelined**





# Ideal Pipelining



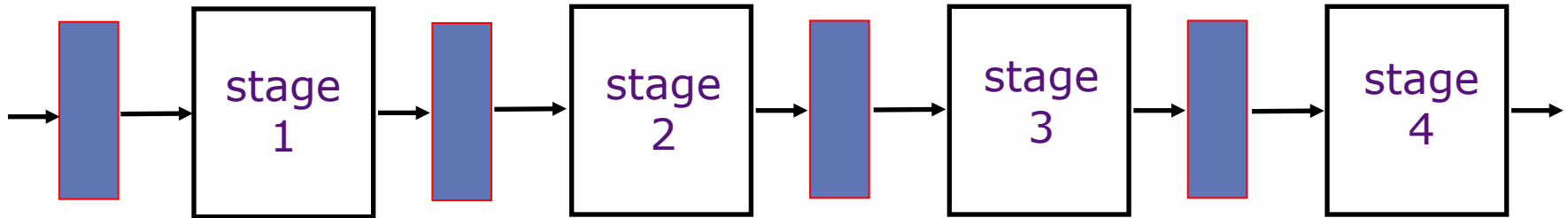
- All objects go through the same stages
- No resources shared between any two stages
- Equal propagation delay through all pipeline stages
- An object entering the pipeline is not affected by objects in other stages
- These conditions generally hold for industrial assembly lines
- But can an instruction pipeline satisfy the last condition?

## Technology Assumptions

- Small, very fast memory (caches) backed by large, slower memory
- Multi-ported register file, which is slower than a single-ported one
- Consider 5-stage pipelined Harvard architecture



# Practical Pipelining



## Pipeline Overheads

- Each stage requires registers, which hold state/data communicated from one stage to next, incurring hardware and delay overheads
- Each stage requires partitioning logic into “equal” lengths
- Introduces diminishing marginal returns from deeper pipelines

## Pipeline Hazards

- Instructions do not execute independently
- Instructions entering the pipeline depend on in-flight instructions or contend for shared hardware resources



# Pipelining MIPS

First, build MIPS without pipelining

- Single-cycle MIPS datapath

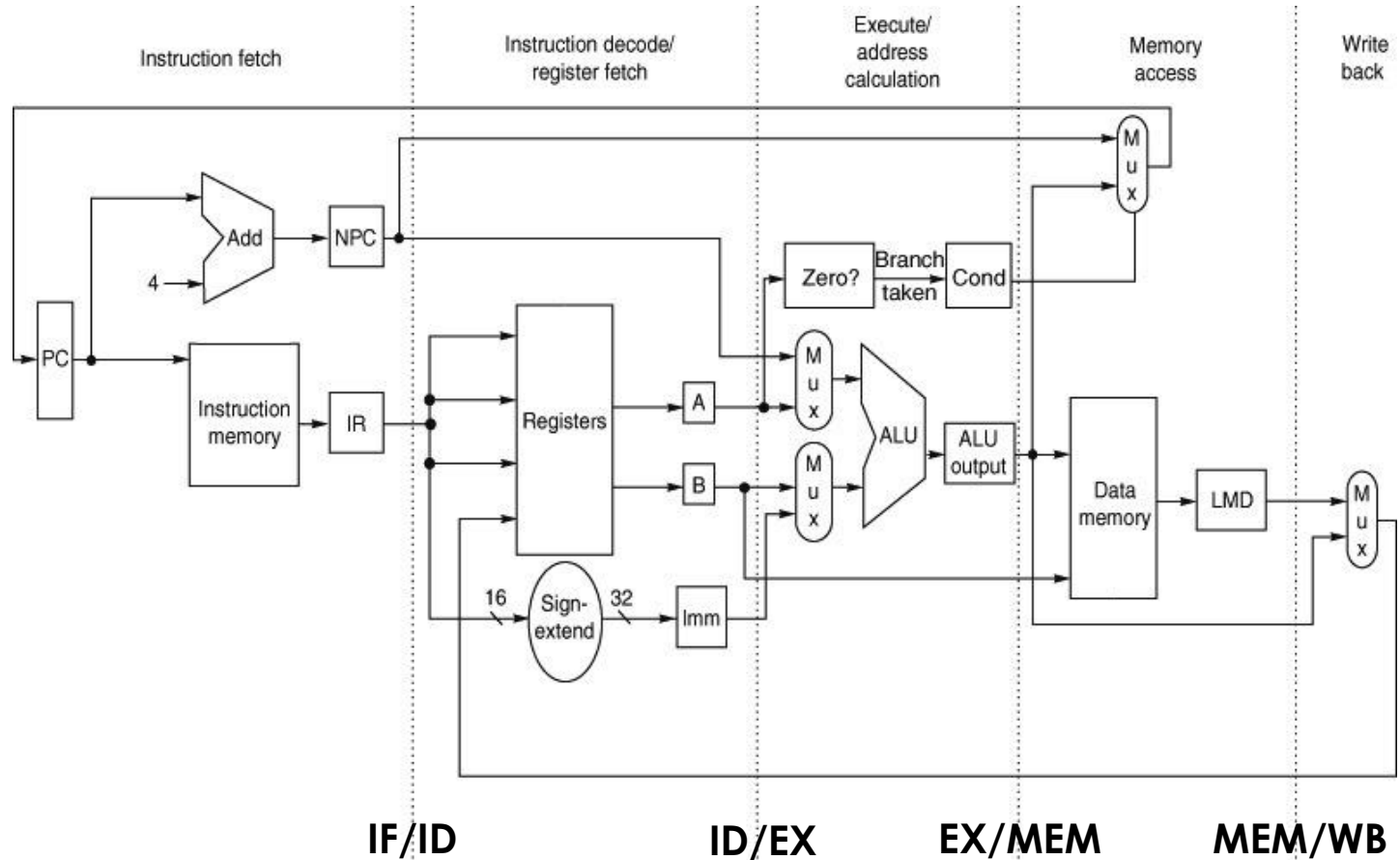
Then, pipeline into multiple stages

- Multi-cycle MIPS datapath
- Add pipeline registers to separate logic into stages
- MIPS partitions into 5 stages
- 1: Instruction Fetch (IF)
- 2: Instruction Decode (ID)
- 3: Execute (EX)
- 4: Memory (MEM )
- 5: Write Back (WB)



# 5-Stage Pipelined Datapath (MIPS)

Figure A.17, Page A-29



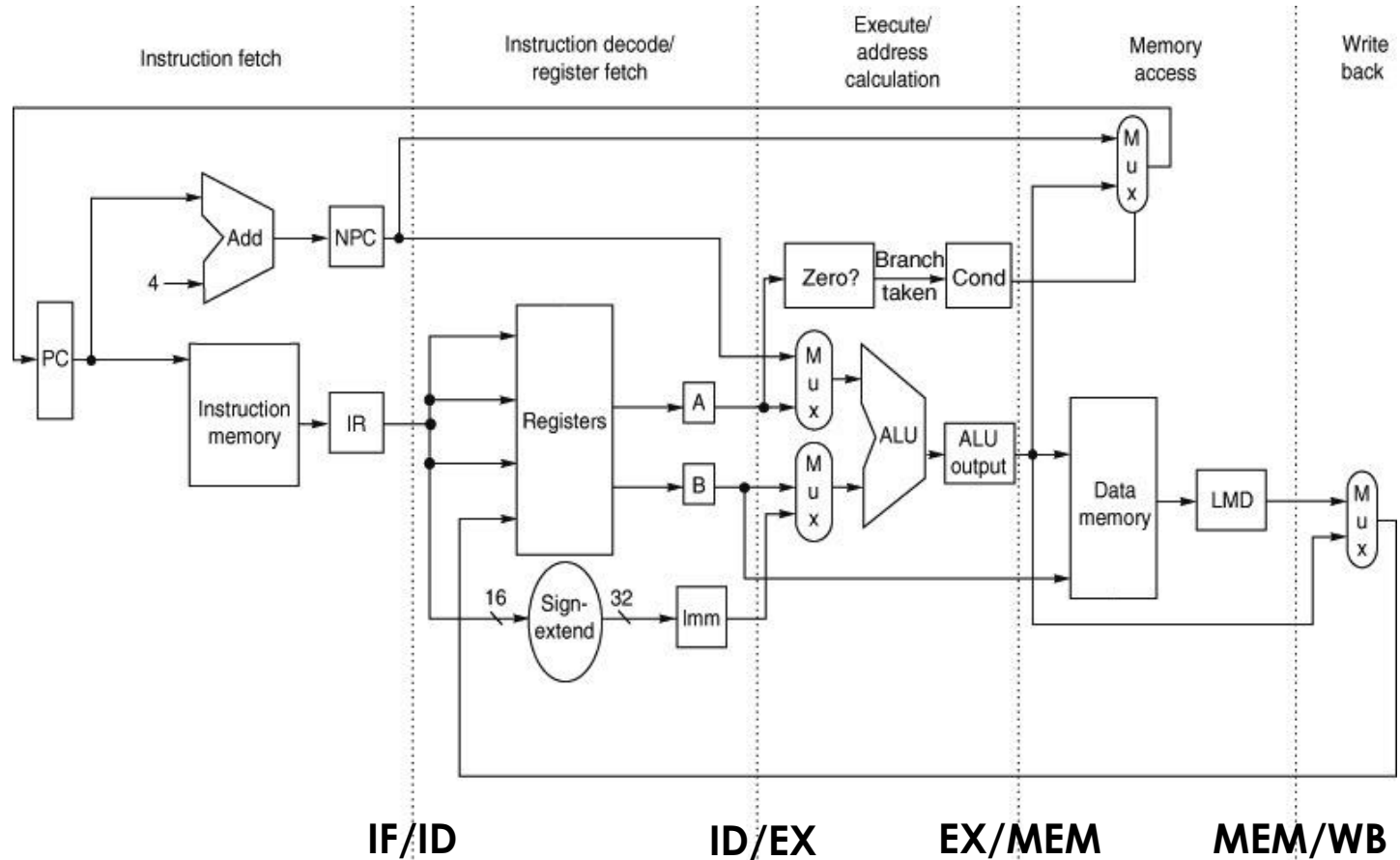
© 2007 Elsevier, Inc. All rights reserved.

$$IR \leftarrow \text{mem}[PC]; PC \leftarrow PC + 4; \text{Reg}[IR_{rd}] \leftarrow \text{Reg}[IR_{rs}] \text{ op}_{IR_{op}} \text{Reg}[IR_{rt}]$$



# 5-Stage Pipelined Datapath (MIPS)

Figure A.17, Page A-29



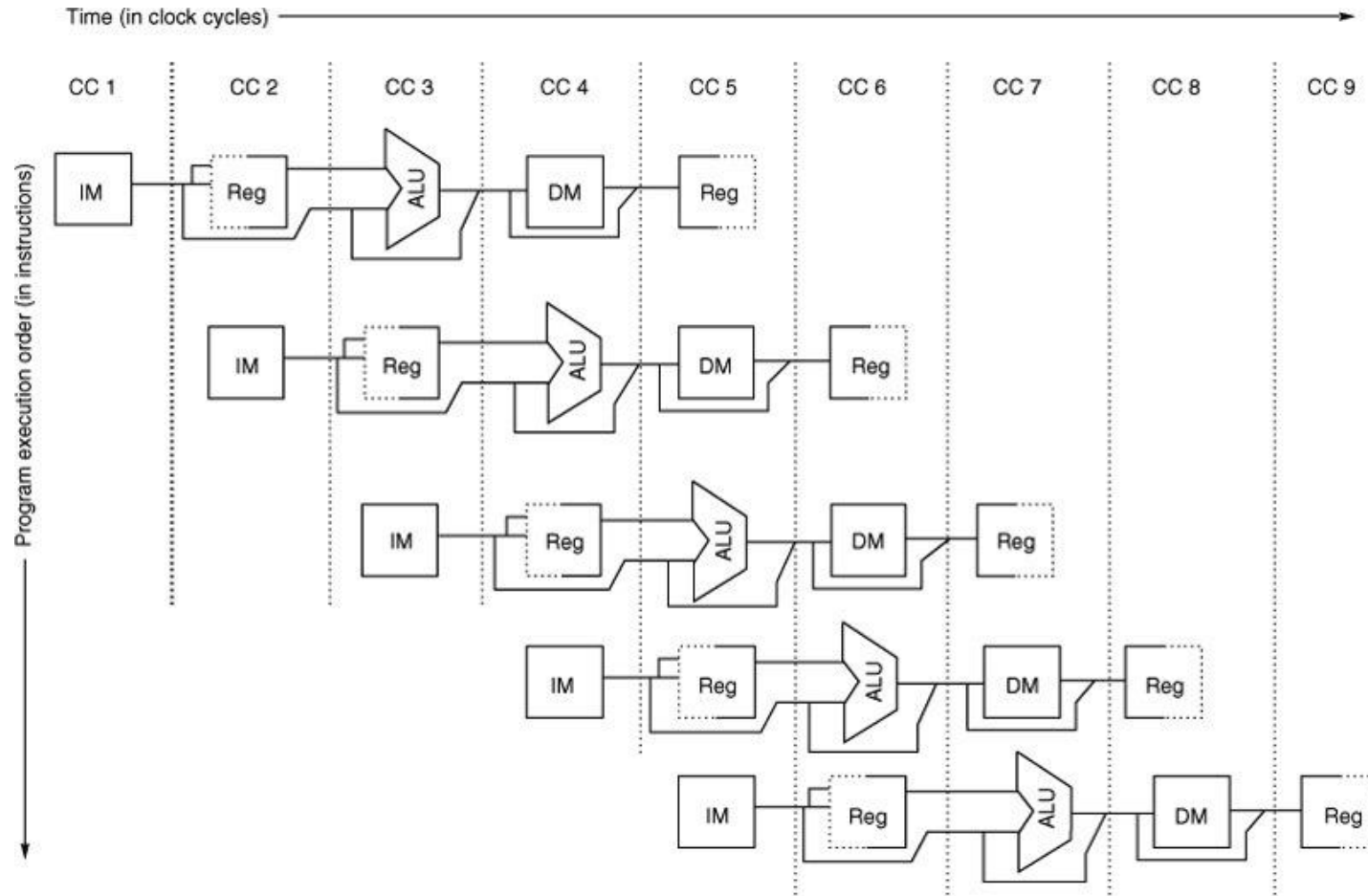
© 2007 Elsevier, Inc. All rights reserved.

$A \leftarrow \text{Reg}[\text{IR}_{rs}]; B \leftarrow \text{Reg}[\text{IR}_{rt}]; \text{Result} \leftarrow A \text{ op}_{\text{IR}_{\text{rop}}} B;$   
 $\text{WB} \leftarrow \text{Result}; \text{Reg}[\text{IR}_{rd}] \leftarrow \text{WB}$



# Visualizing the Pipeline

Figure A.2, Page A-8



© 2007 Elsevier, Inc. All rights reserved.



# Hazards and Limits to Pipelining

Hazards prevent next instruction from executing during its designated clock cycle

## Structural Hazards

- Hardware cannot support this combination of instructions.
- Example: Limited resources required by multiple instructions (e.g. FPU)

## Data Hazards

- Instruction depends on result of prior instruction still in pipeline
- Example: An integer operation is waiting for value loaded from memory

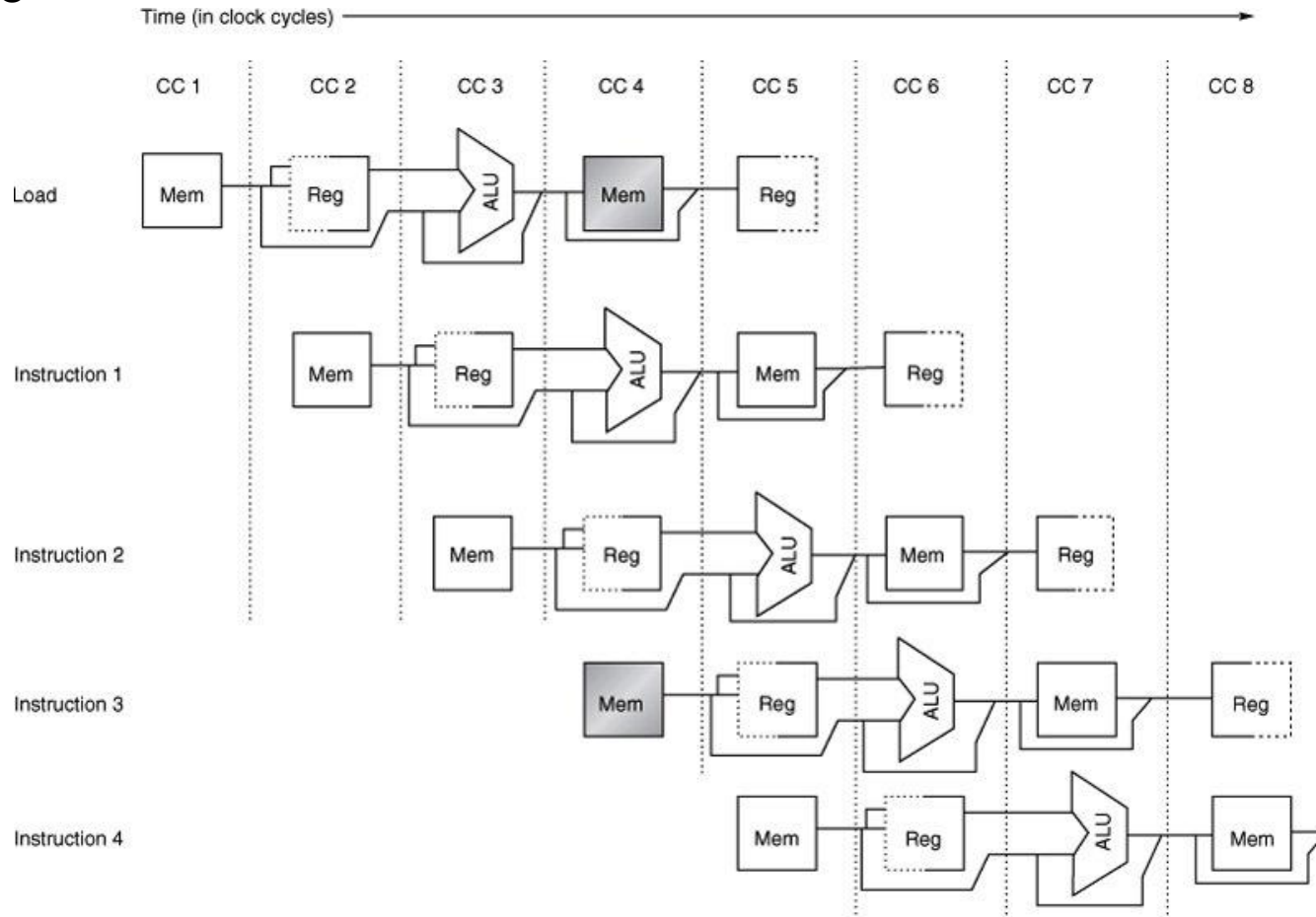
## Control Hazards

- Instruction fetch depends on decision about control flow
- Example: Branches and jumps change PC



# Structural Hazards

Figure A.4, A-14



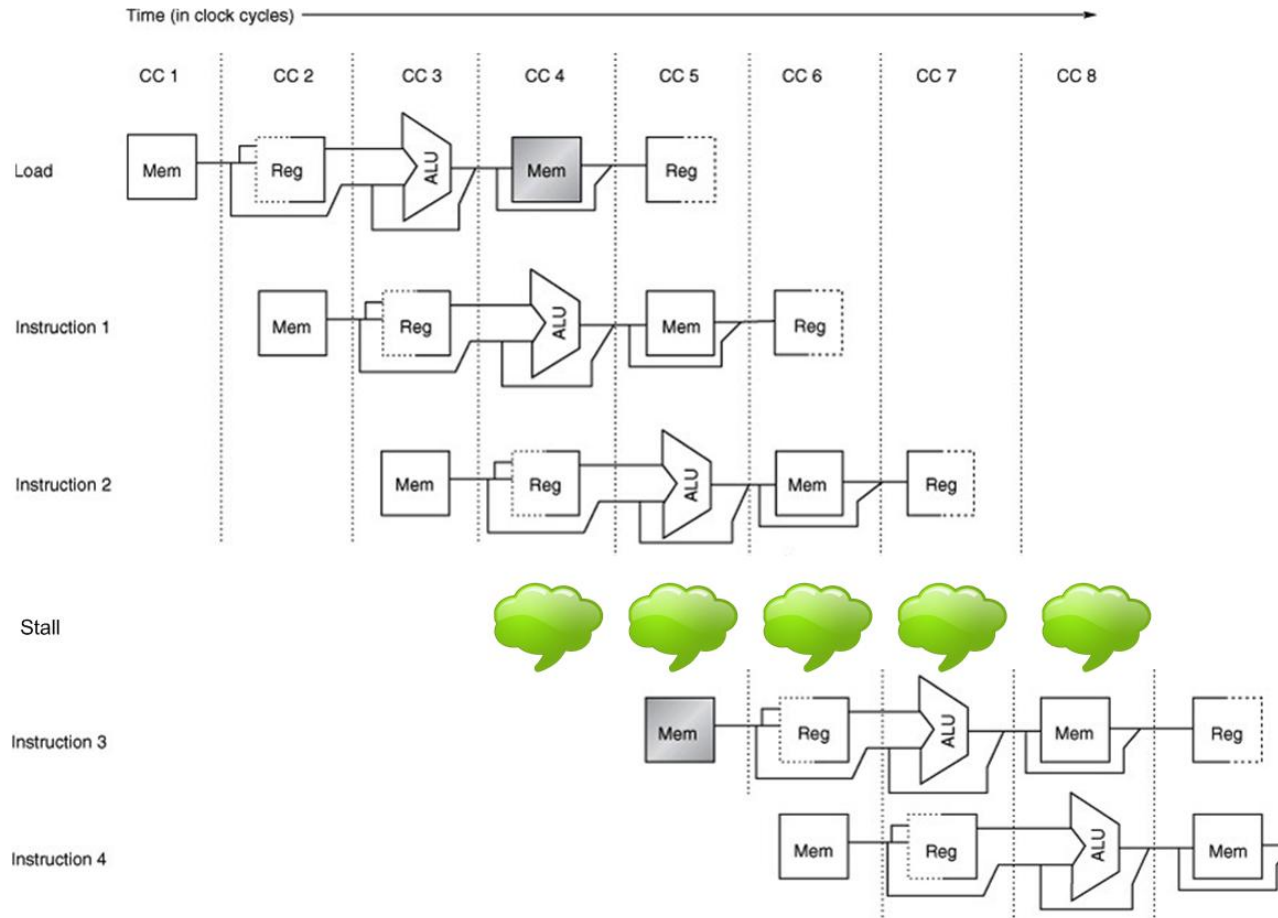
© 2007 Elsevier, Inc. All rights reserved.

A single memory port causes structural hazard during data load, instr fetch



# Structural Hazards

Figure A.4, A-14

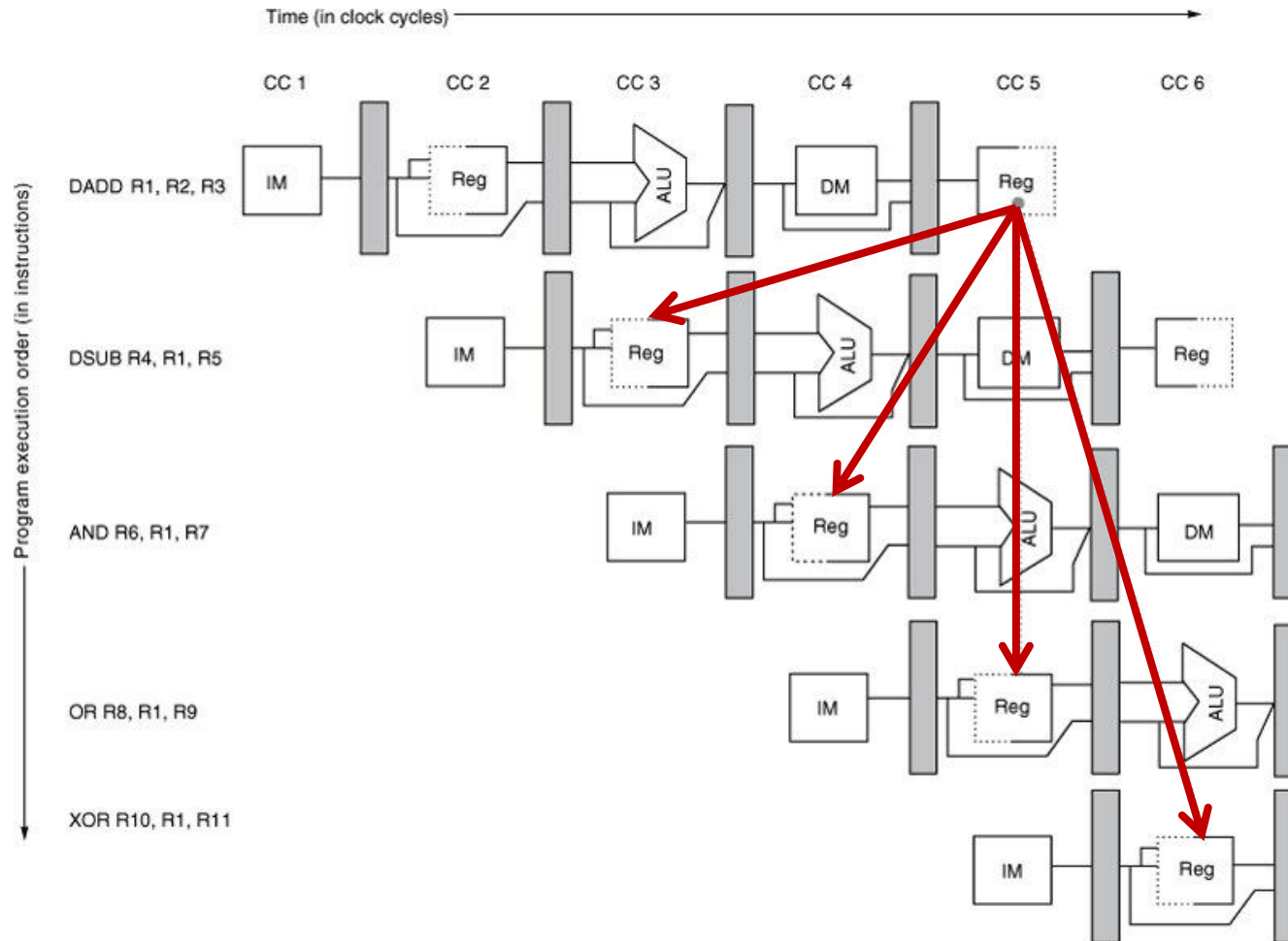


Stall the pipeline, creating bubbles, by freezing earlier stages → interlocks  
Use Harvard Architecture (separate instruction, data memories)



# Data Hazards

Figure A.6, A-16



© 2007 Elsevier, Inc. All rights reserved.

Instruction depends on result of prior instruction still in pipeline



# Data Hazards

## Read After Write (RAW)

- Caused by a dependence, need for communication
- Instr-j tries to read operand before Instr-I writes it

i: add r1, r2, r3

j: sub r4, r1, 43

## Write After Read (WAR)

- Caused by an anti-dependence and the re-use of the name "r1"
- Instr-j writes operand (r1) before Instr-I reads it

i: add r4, r1, r3

j: add r1, r2, r3

k: mul r6, r1, r7

## Write After Write (WAW)

- Caused by an output dependence and the re-use of the name "r1"
- Instr-j writes operand (r1) before Instr-I writes it

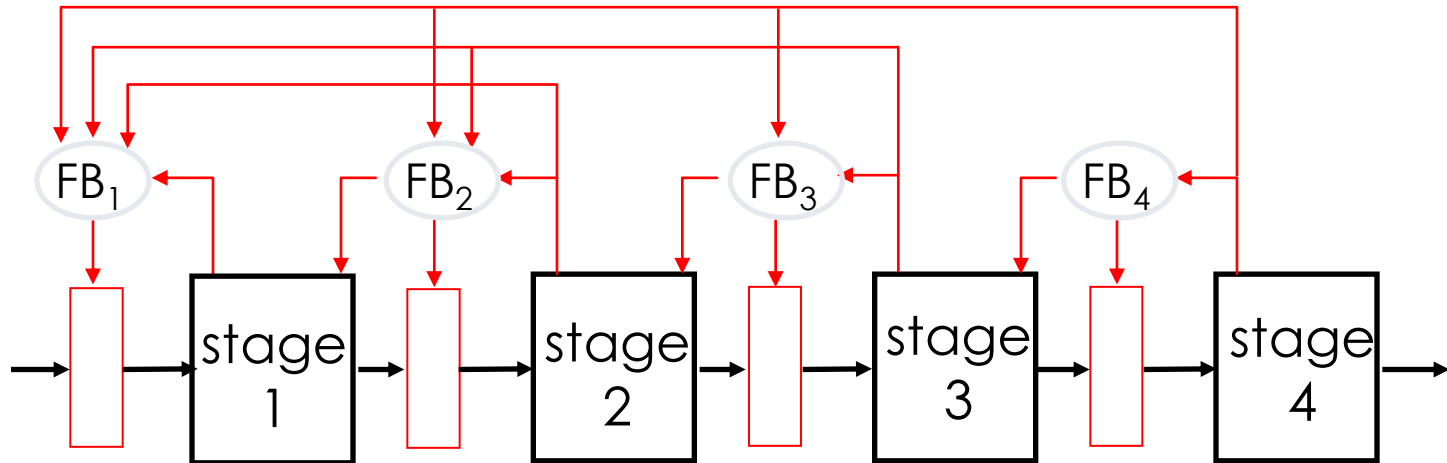
i: sub r1, r4, r3

j: add r1, r2, r3

k: mul r6, r1, r7



# Resolving Data Hazards

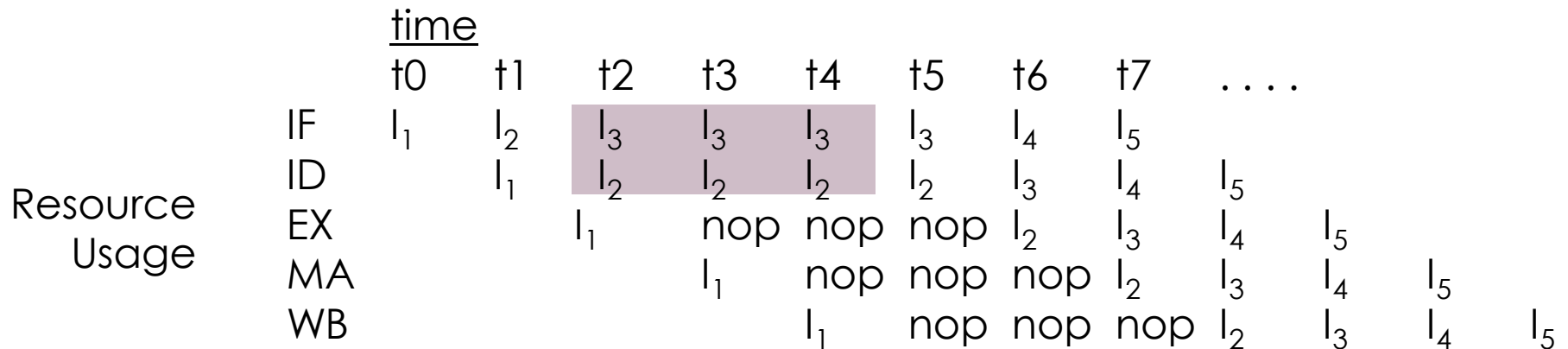
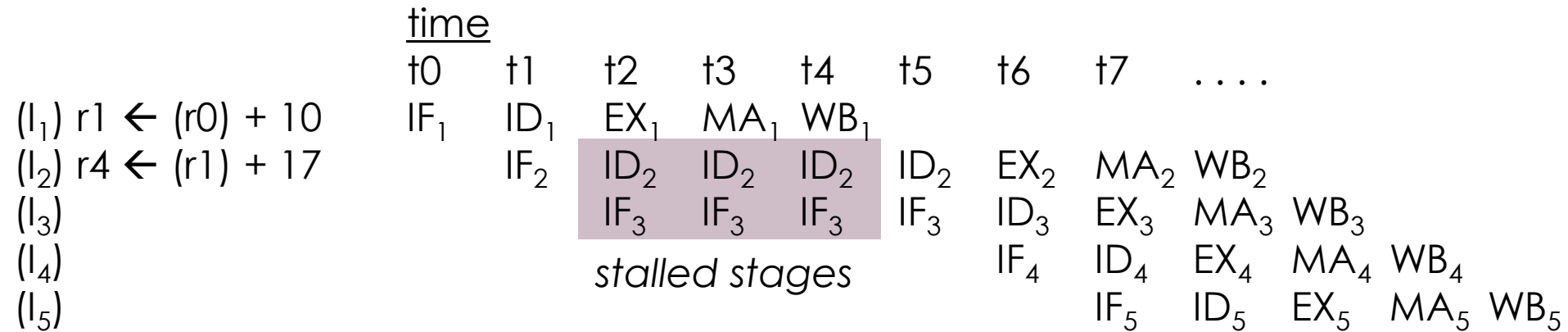


## Strategy 1 – Interlocks and Pipeline Stalls

- Later stages provide dependence information to earlier stages, which can stall or kill instructions
- Works as long as instruction at stage  $i+1$  can complete without any interference from instructions in stages 1 through  $i$  (otherwise, deadlocks may occur)



# Interlocks & Pipeline Stalls





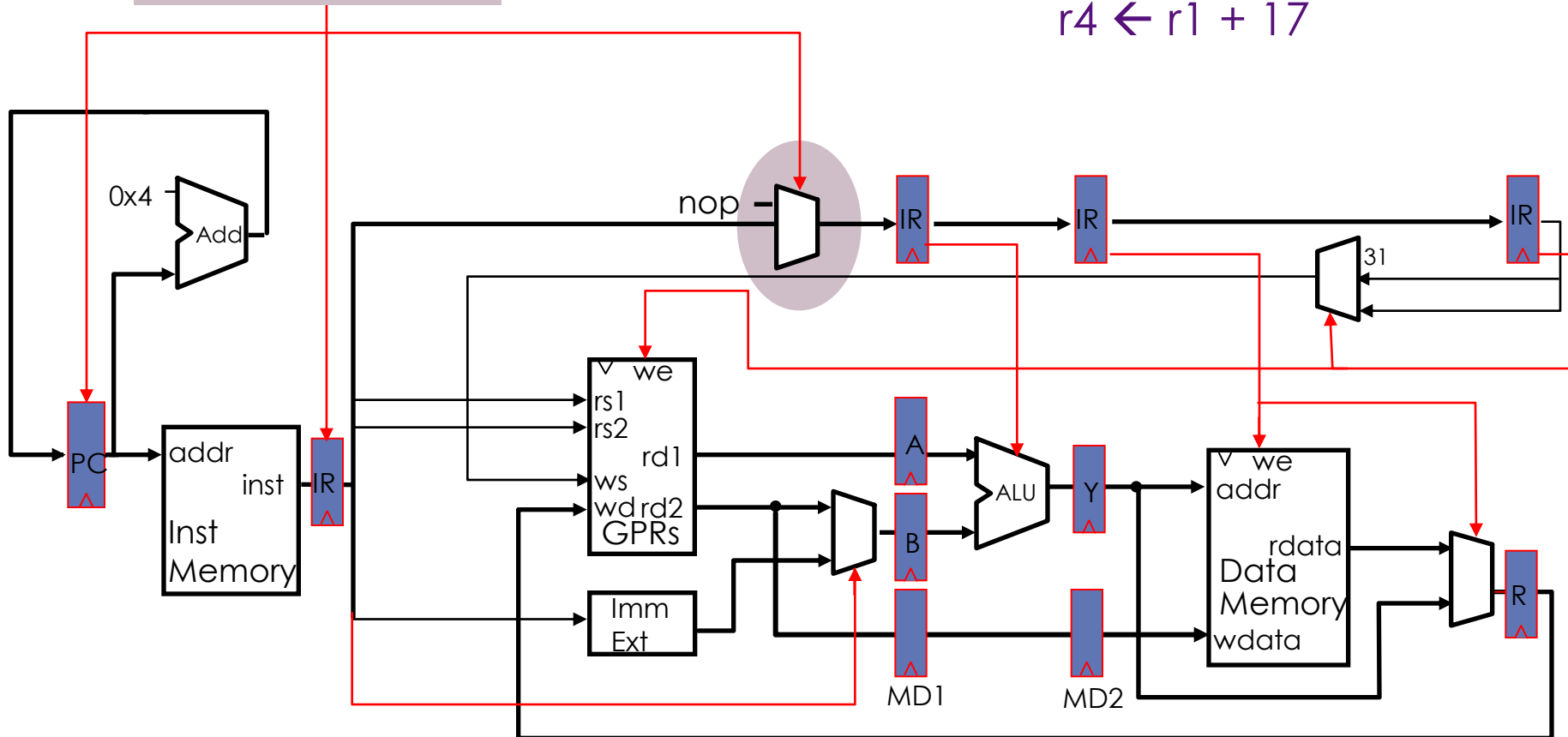
# Interlocks & Pipeline Stalls

Example Dependence

$r1 \leftarrow r0 + 10$

$r4 \leftarrow r1 + 17$

*Stall Condition*





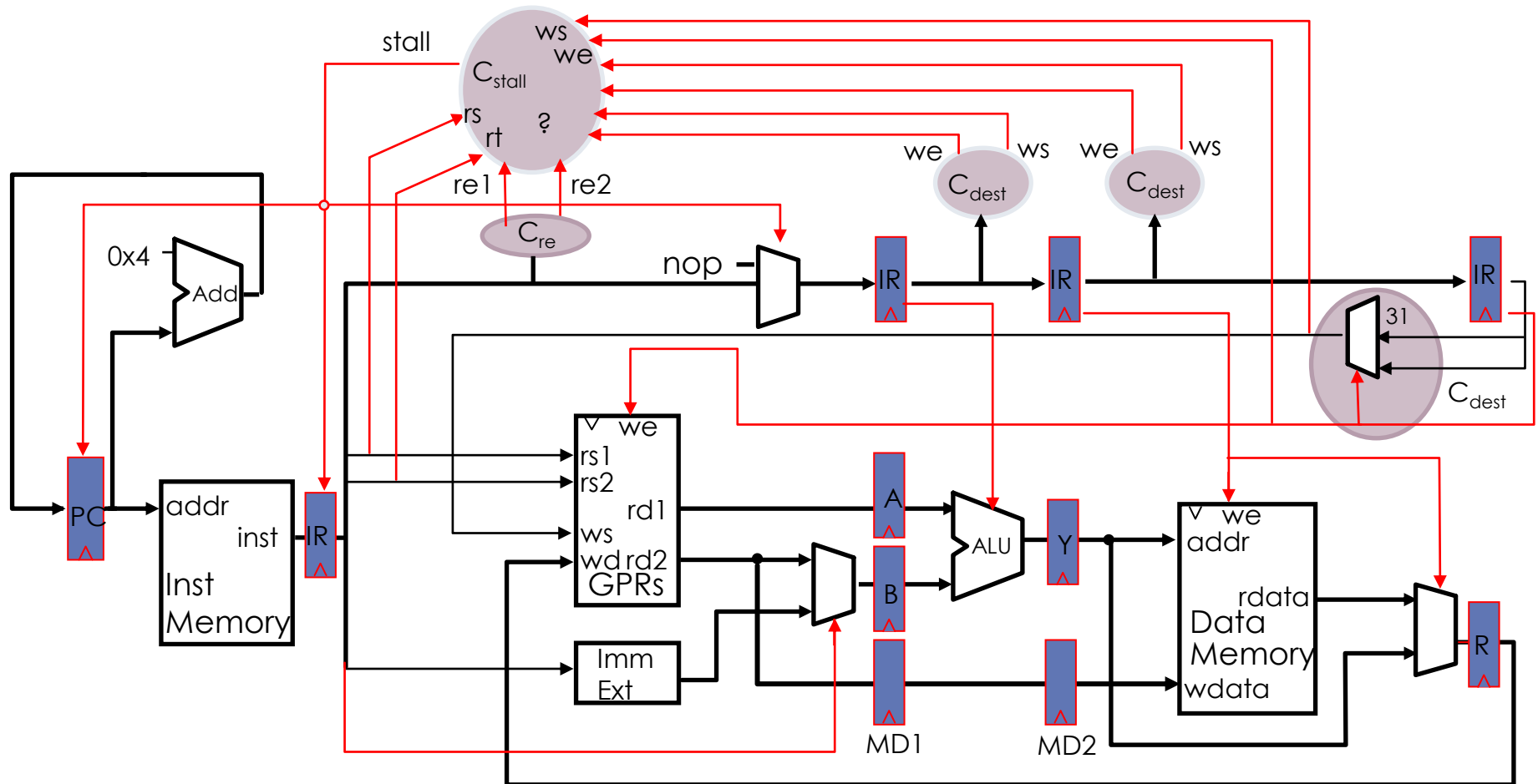
# Interlock Control Logic

- Compare the source registers of instruction in decode stage with the destination registers of uncommitted instructions
- Stall if a source register in decode matches some destination register?
- No, not every instruction writes to a register
- No, not every instruction reads from a register
- Derive stall signal from conditions in the pipeline





# Interlock Control Logic



Should we always stall if RS/RT matches some RD? No, because not every instruction writes/reads a register. Introduce write/read enable signals (we/re)



# Source and Destination Registers

R-type:                    

op	rs	rt	rd		func
----	----	----	----	--	------

I-type:                    

op	rs	rt	immediate16
----	----	----	-------------

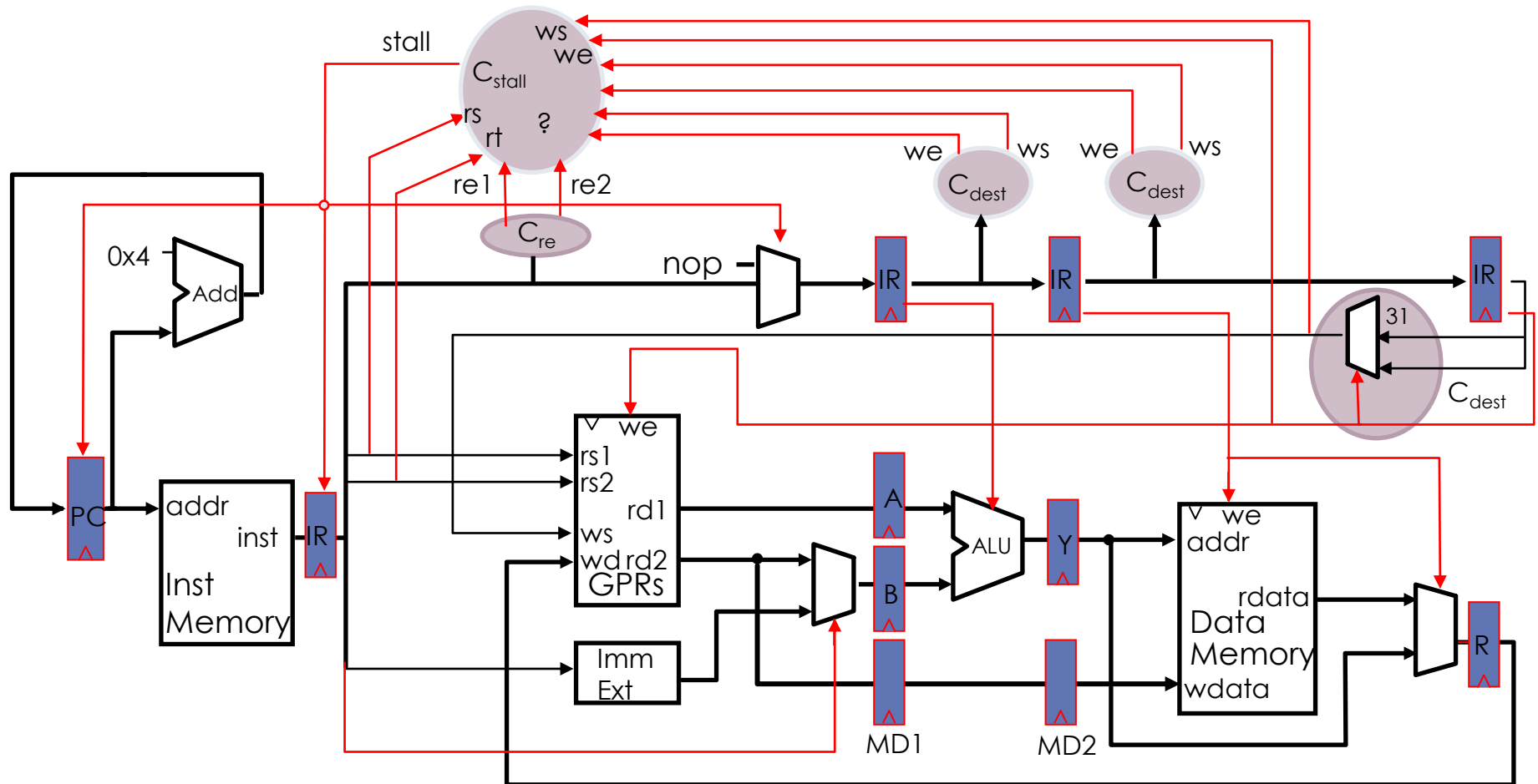
J-type:                    

op	immediate26
----	-------------

instruction		source(s)	destination
ALU	$rd \leftarrow (rs) \text{ func } (rt)$	rs, rt	rd
ALUi	$rt \leftarrow (rs) \text{ op } \text{imm}$	rs	rt
LW	$rt \leftarrow M[(rs) + \text{imm}]$	rs	rt
SW	$M[(rs) + \text{imm}] \leftarrow (rt)$	rs, rt	
BZ	cond (rs)		
	true: $PC \leftarrow (PC) + \text{imm}$	rs	
	false: $PC \leftarrow (PC) + 4$	rs	
J	$PC \leftarrow (PC) + \text{imm}$		
JAL	$r31 \leftarrow (PC), PC \leftarrow (PC) + \text{imm}$		R31
JR	$PC \leftarrow (rs)$	rs	
JALR	$r31 \leftarrow (PC), PC \leftarrow (rs)$	rs	R31



# Interlock Control Logic



Should we always stall if RS/RT matches some RD? No, because not every instruction writes/reads a register. Introduce write/read enable signals (we/re)



# Deriving the Stall Signal

Cdest

ws	Case(opcode)
	ALU: $ws \leftarrow rd$
	ALUi: $ws \leftarrow rt$
	JAL, JALR: $ws \leftarrow R31$

we	Case(opcode)
	ALU, ALUi, LW $we \leftarrow (ws \neq 0)$
	JAL, JALR $we \leftarrow 1$
	otherwise $we \leftarrow 0$

Cre

re1	Case(opcode)
	ALU, ALUi $re1 \leftarrow 1$
	LW, SW, BZ $re1 \leftarrow 1$
	JR, JALR $re1 \leftarrow 1$
	J, JAL $re1 \leftarrow 0$

re2	Case(opcode)
	<< same as re1 but for register rt >>



Xrt denote register rt for instruction in pipeline stage X

Cstall

```
stall-2 ← ( (Drt == Ews) & Ewe |
             (Drt == Mws) & Mwe |
             (Drt == Wws) & Wwe
           ) & Dre2
```

$$\text{stall} \leftarrow \text{stall-1} \mid \text{stall-2}$$



# Load/Store Data Hazards

$M[(r1)+7] \leftarrow (r2)$

$r4 \leftarrow M[(r3)+5]$

What is the problem here?

What if  $(r1)+7 == (r3)+5$ ?

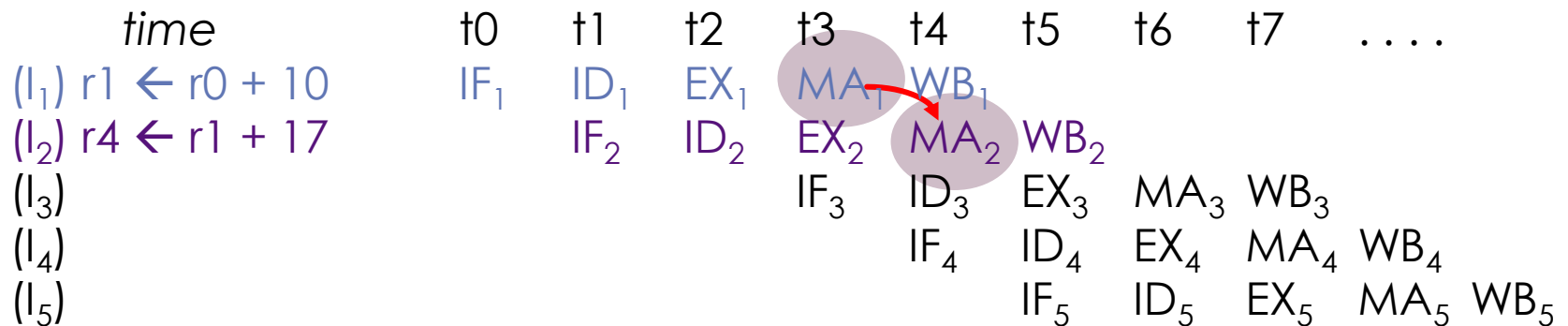
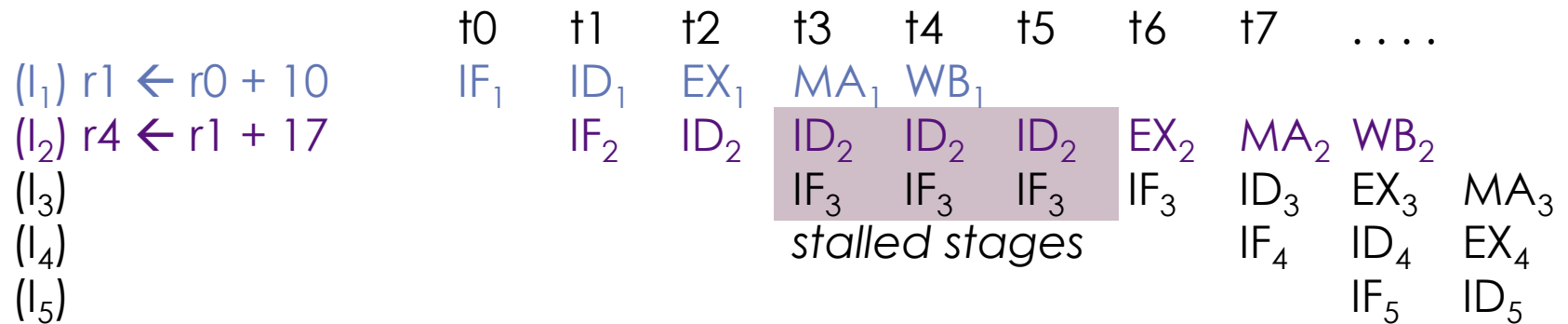
Load/Store hazards may be resolved in the pipeline or may be resolved in the memory system. More later.



# Resolving Data Hazards

## Strategy 2 – Forwarding (aka Bypasses)

- Route data as soon as possible to earlier stages in the pipeline
- Example: forward ALU output to its input







# Deriving Forwarding Signals

This forwarding path only applies to the ALU operations...

Eforward	Case(Eopcode)	
	ALU, ALUi	Eforward $\leftarrow$ (ws $\neq$ 0)
	otherwise	Eforward $\leftarrow$ 0

...and all other operations will need to stall as before

Estall	Case(Eopcode)	
	LW	Estall $\leftarrow$ (ws $\neq$ 0)
	JAL, JALR	Estall $\leftarrow$ 1
	otherwise	Estall $\leftarrow$ 0

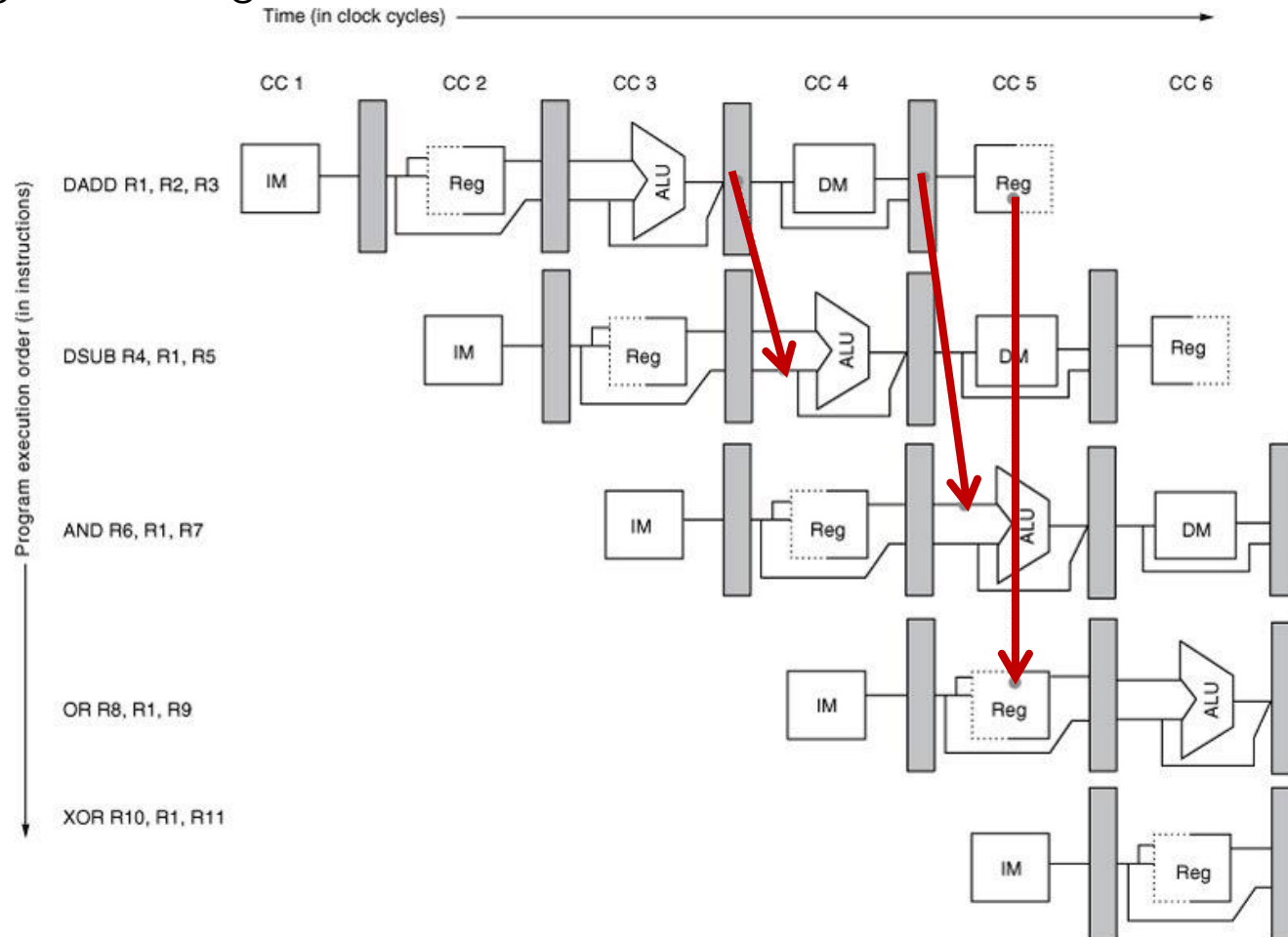
$$\text{Asrc} \leftarrow (\text{Drs} == \text{Ews}) \ \& \ \text{Dre1} \ \& \ \text{Eforward}$$

Remember to update stall signal, removing case covered by this forwarding path



# Multiple Forwarding Paths

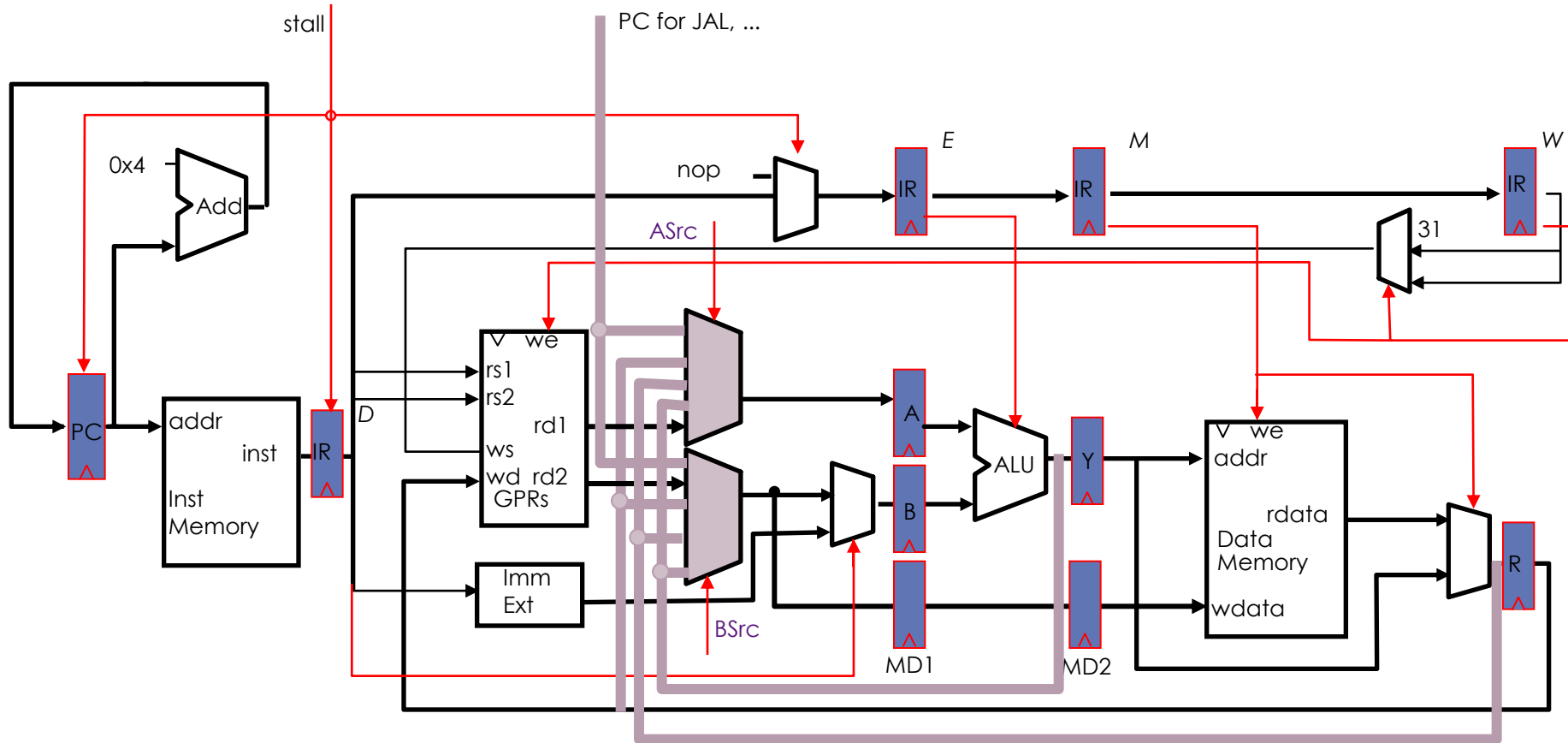
Figure A.7, Page A-18



© 2007 Elsevier, Inc. All rights reserved.



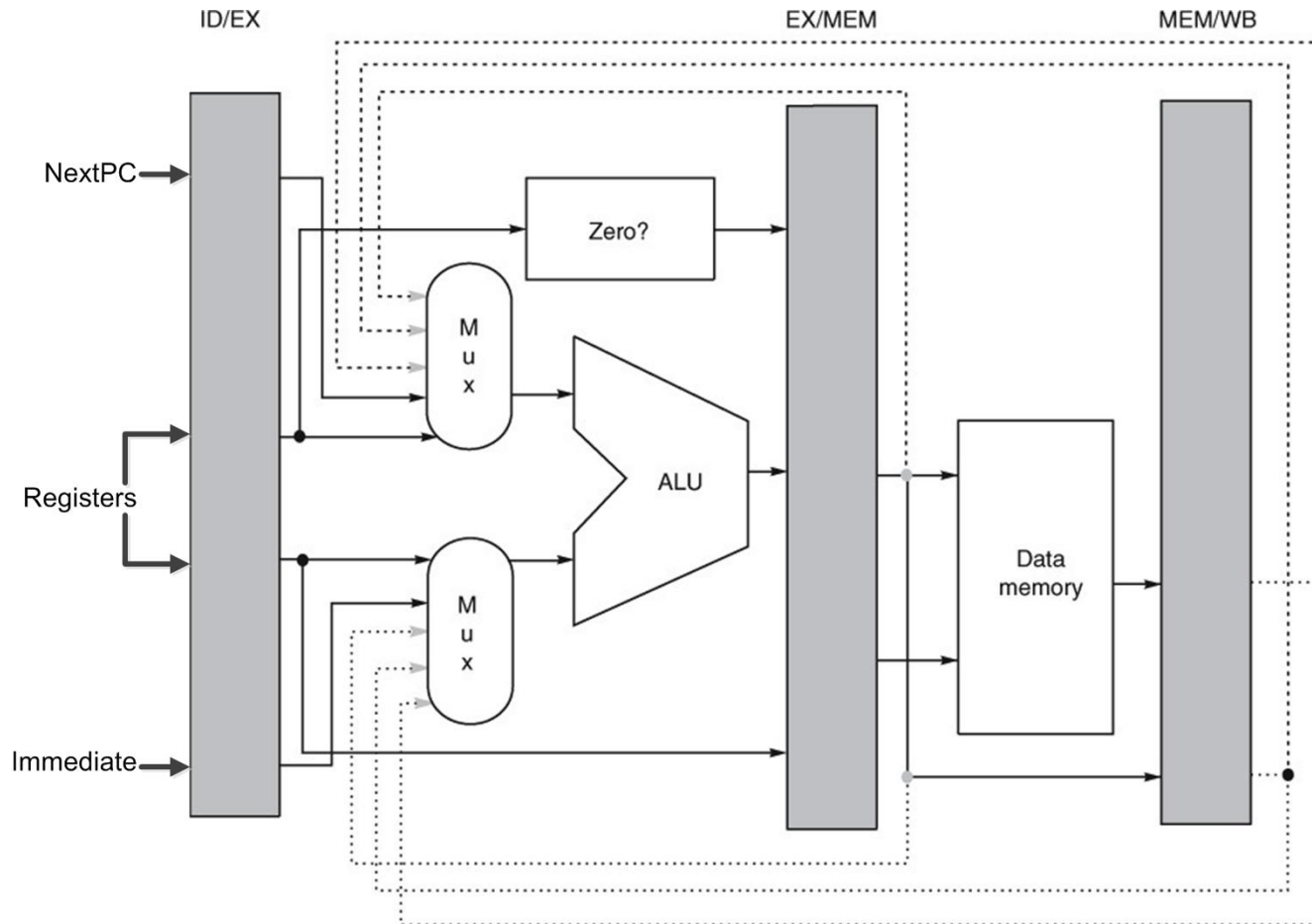
# Multiple Forwarding Paths





# Forwarding Hardware

Figure A.23, Page A-37

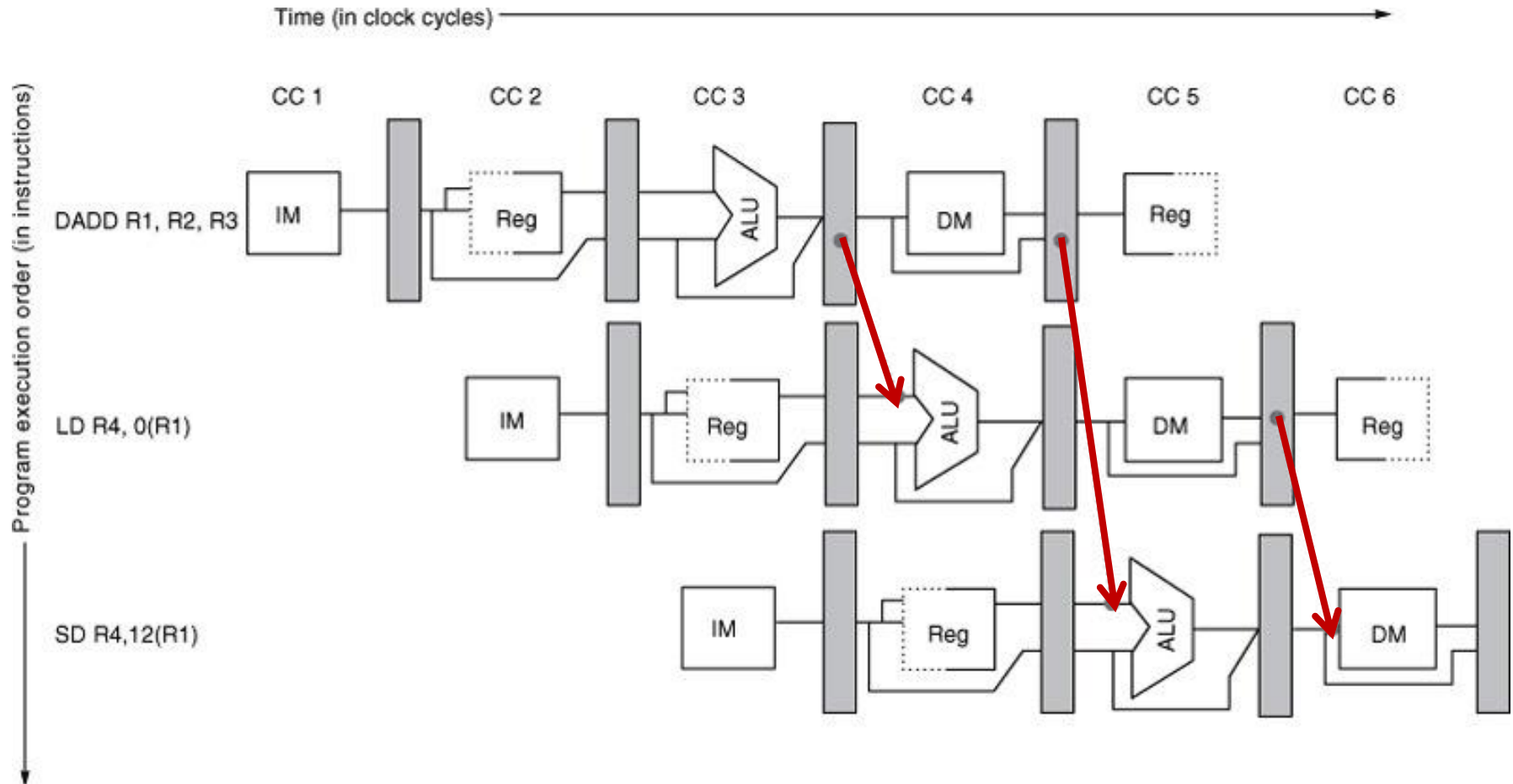


© 2007 Elsevier, Inc. All rights reserved.



# Forwarding Loads/Stores

Figure A.8, Page A-19

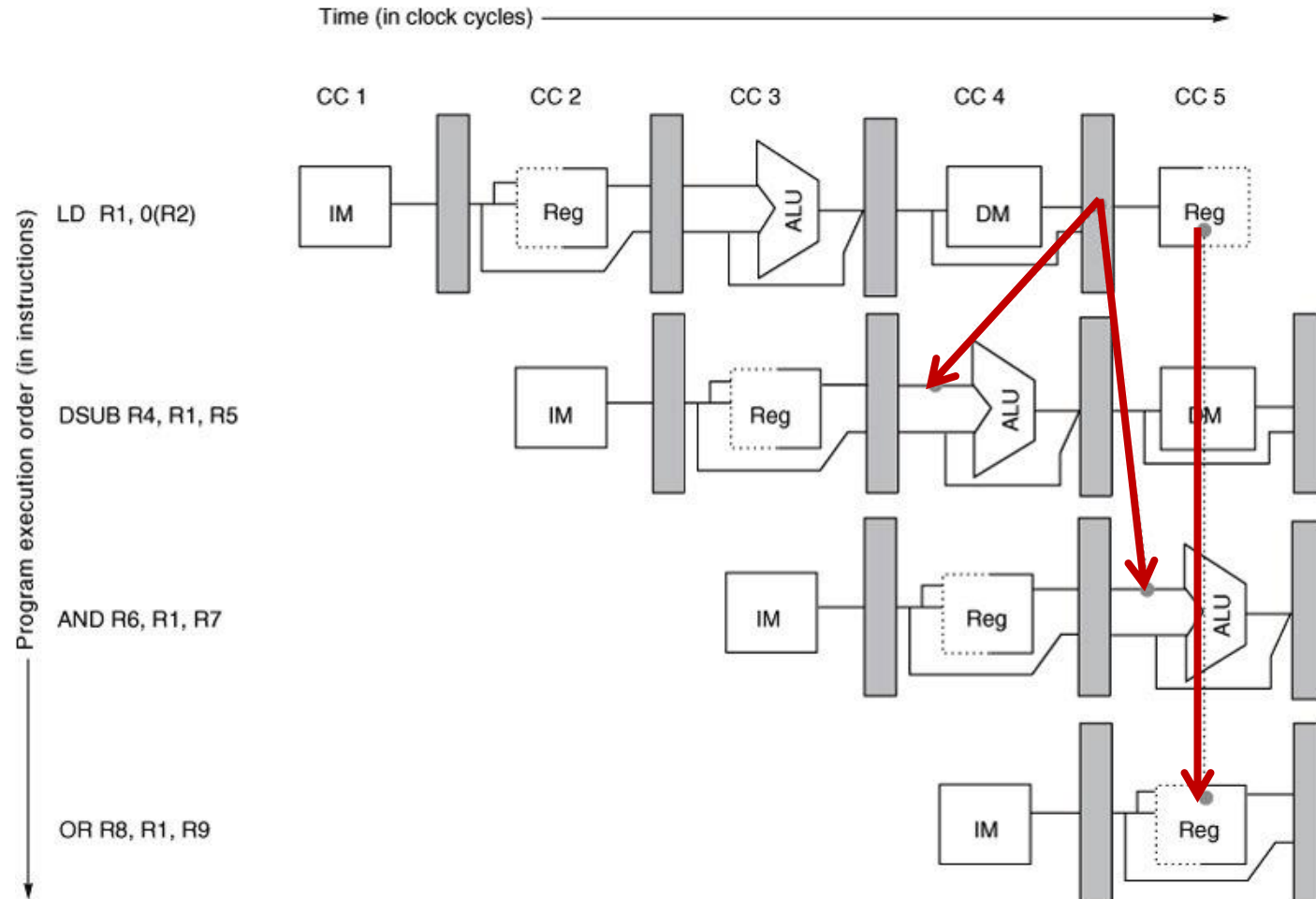


© 2007 Elsevier, Inc. All rights reserved.



# Data Hazard Despite Forwarding

Figure A.9, Page A-20



© 2007 Elsevier, Inc. All rights reserved.

LD cannot forward (backwards in time) to DSUB. What is the solution?



# Data Hazards and Scheduling

## Try producing faster code for

- $A = B + C; D = E - F;$
- Assume A, B, C, D, E, and F are in memory
- Assume pipelined processor

### Slow Code

LW	Rb, b
LW	Rc, c
ADD	Ra, Rb, Rc
SW	a, Ra
LW	Re, e
LW	Rf, f
SUB	Rd, Re, Rf
SW	d, Rd

### Fast Code

LW	Rb, b
LW	Rc, c
LW	Re, e
ADD	Ra, Rb, Rc
LW	Rf, f
SW	a, Ra
SUB	Rd, Re, Rf
SW	d, Rd



# Acknowledgements

These slides contain material developed and copyright by

- Arvind (MIT)
- Krste Asanovic (MIT/UCB)
- Joel Emer (Intel/MIT)
- James Hoe (CMU)
- John Kubiatawicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)