ECE 552 / CPS 550
Advanced Computer Architecture I

Lecture 6
Pipelining – Part 1

Benjamin Lee
Electrical and Computer Engineering
Duke University

www.duke.edu/~bcl15
www.duke.edu/~bcl15/class/class_ece252fall12.html

ECE552 Administrivia

27 September – Homework #2 Due

Assignment on web page. Teams of 2-3.

Submit soft copies to Sakai.

Use Piazza for questions.

2 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

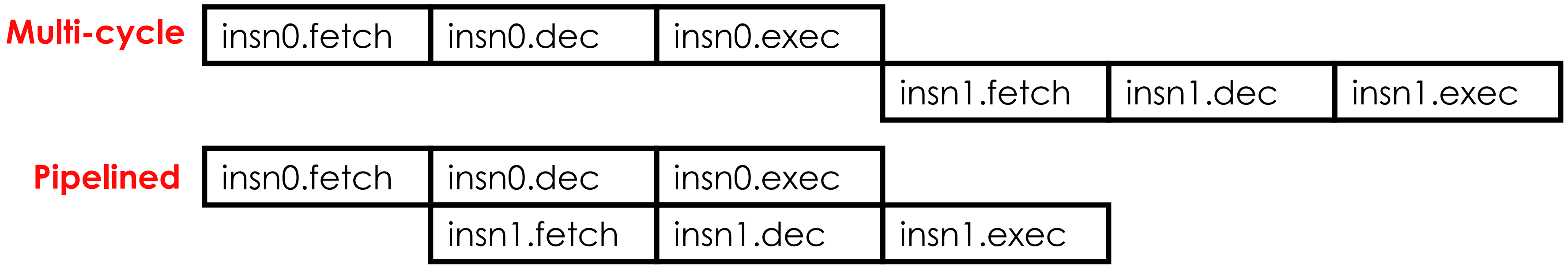
Latency = (Instructions / Program) x (Cycles / Instruction) x (Seconds / Cycle)

Performance Enhancement

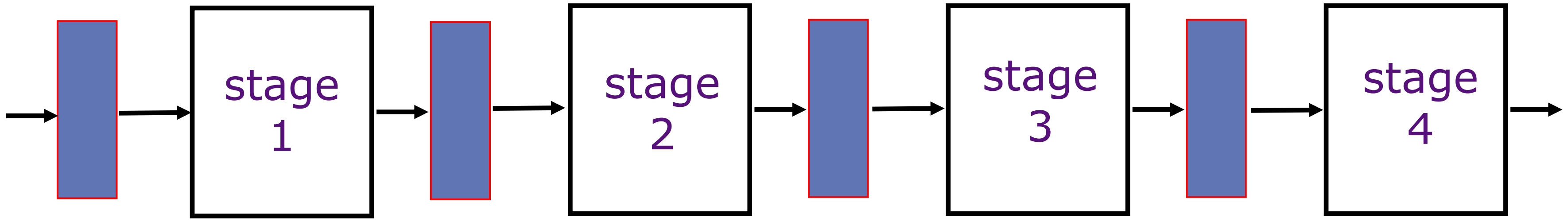
- Increases number of cycles per instruction
- Reduces number of seconds per cycle

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



Ideal Pipelining



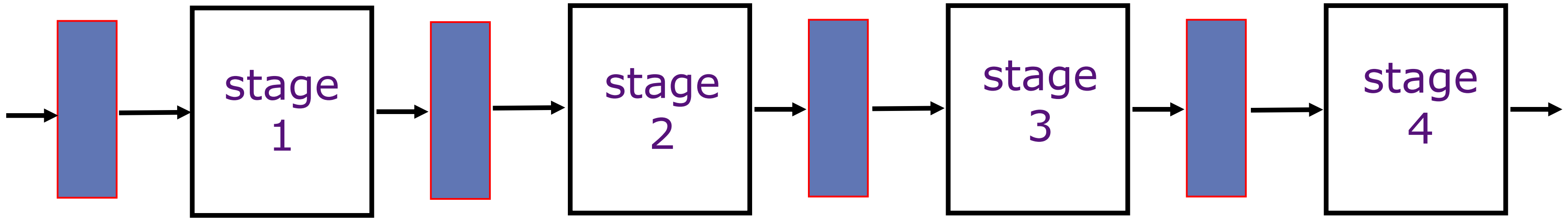
- All objects go through the same stages
- No resources shared between any two stages
- Equal propagation delay through all pipeline stages
- An object entering the pipeline is not affected by objects in other stages

- 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

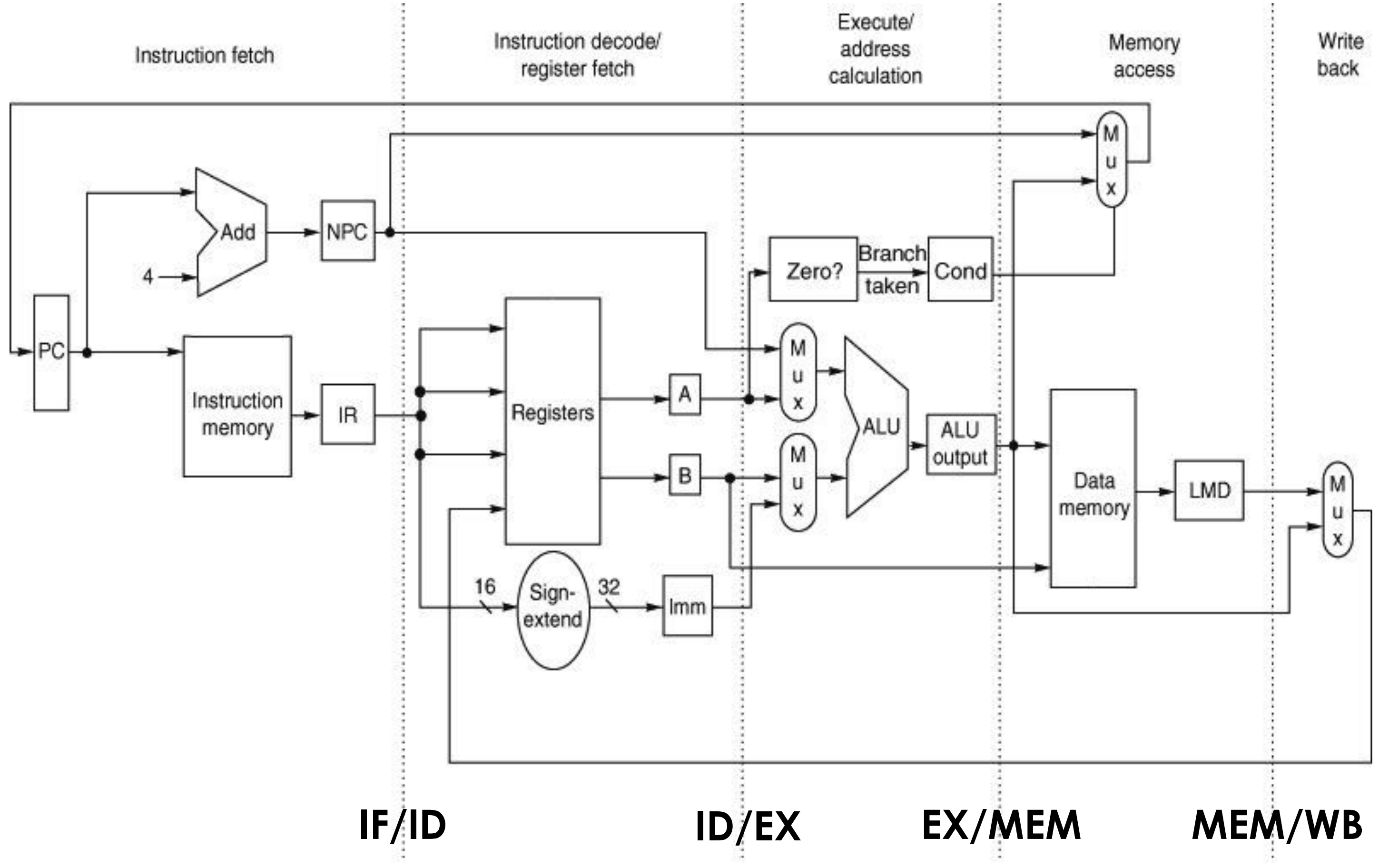
- Single-cycle MIPS datapath

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)

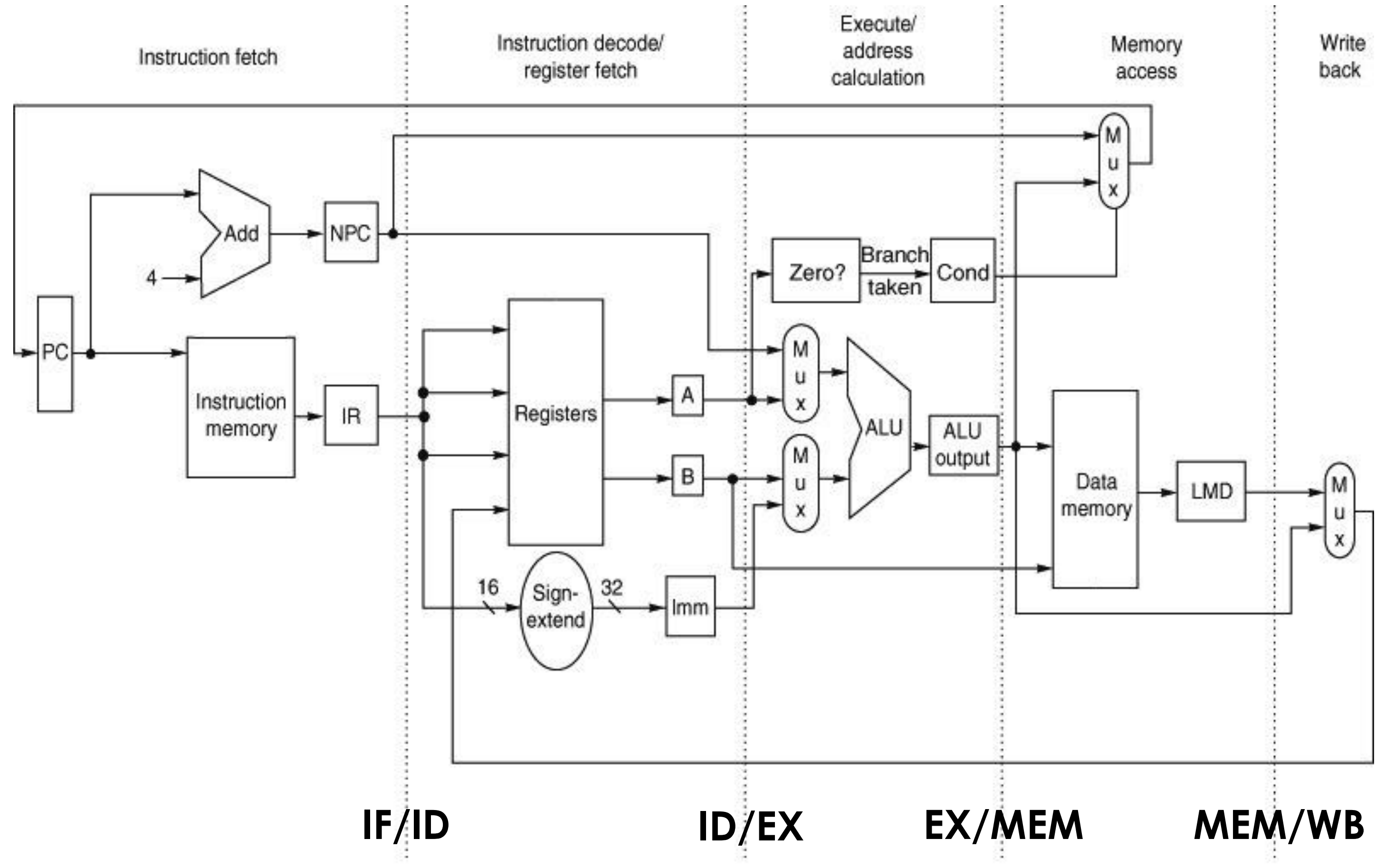
5-Stage Pipelined Datapath (MIPS)



IF: IR ← mem[PC]; PC ← PC + 4;

ID: A ← Reg[IR_{rs}]; B ← Reg[IR_{rt}];

5-Stage Pipelined Datapath (MIPS)



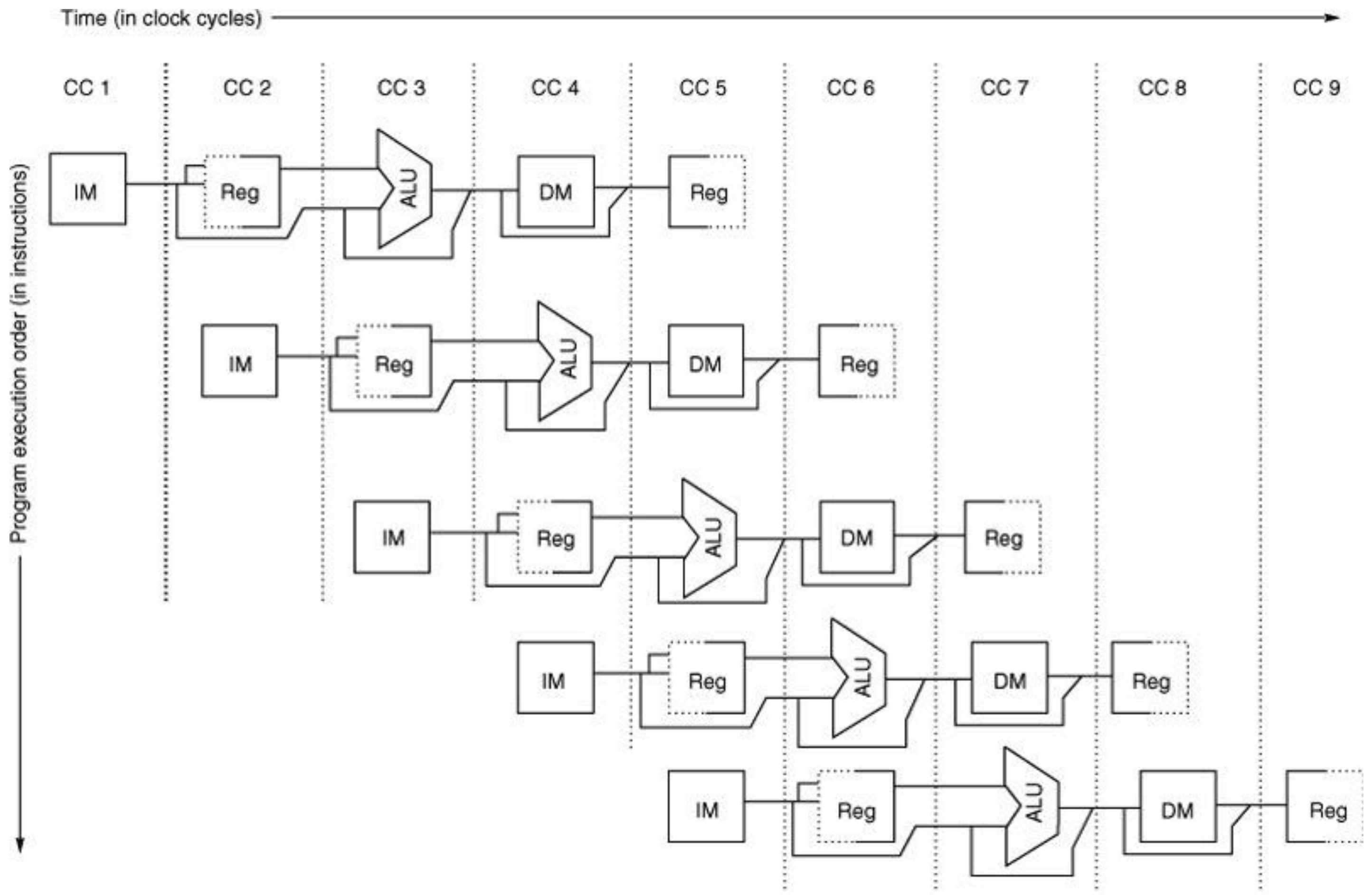
© 2007 Elsevier, Inc. All rights reserved.

EX: Result ← A op_{IRop} B;

MEM: WB ← Result;

WB: $Reg[IR_{rd}]$ ← WB

Visualizing the Pipeline



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)

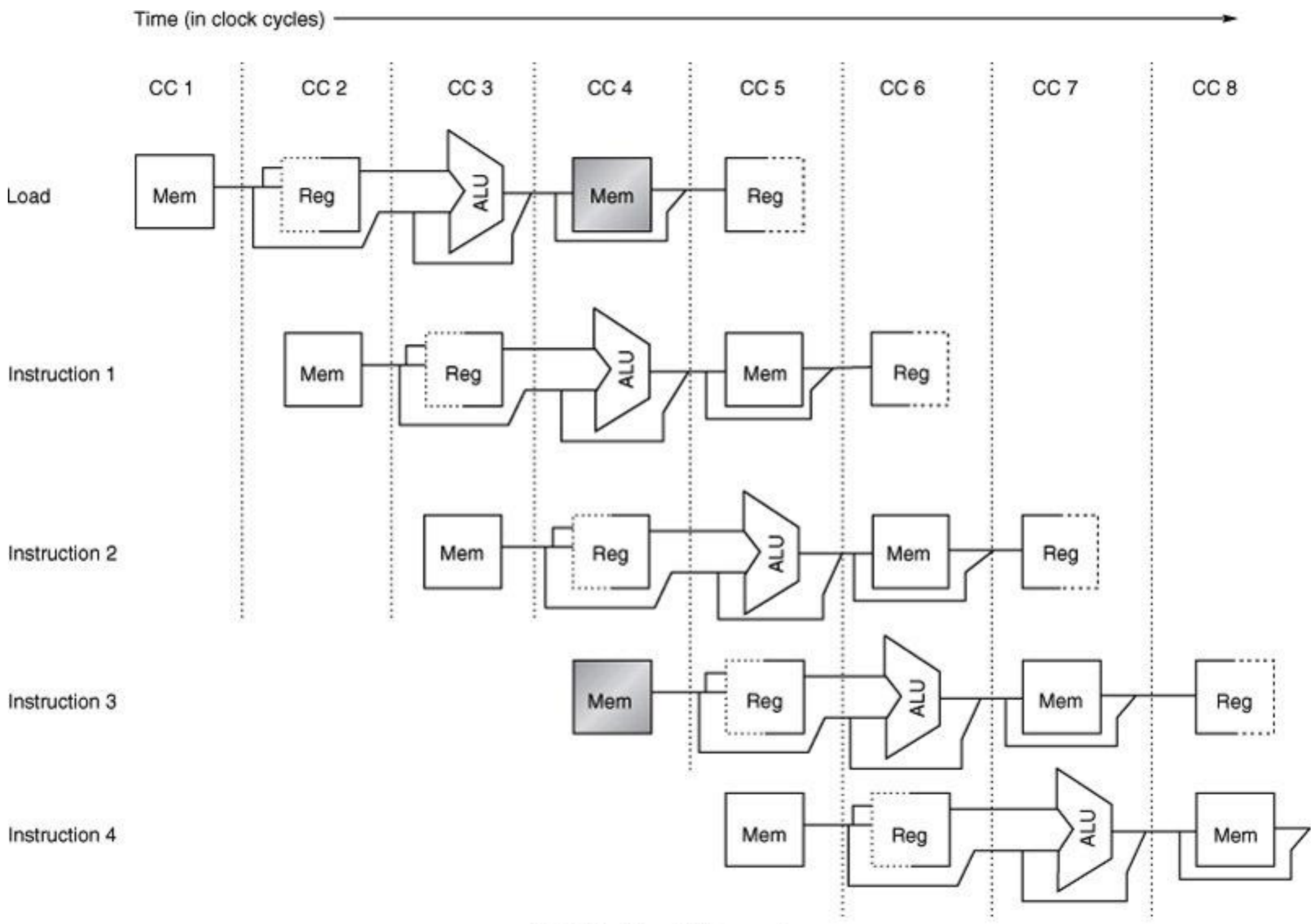
Data Hazards

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

Control Hazards

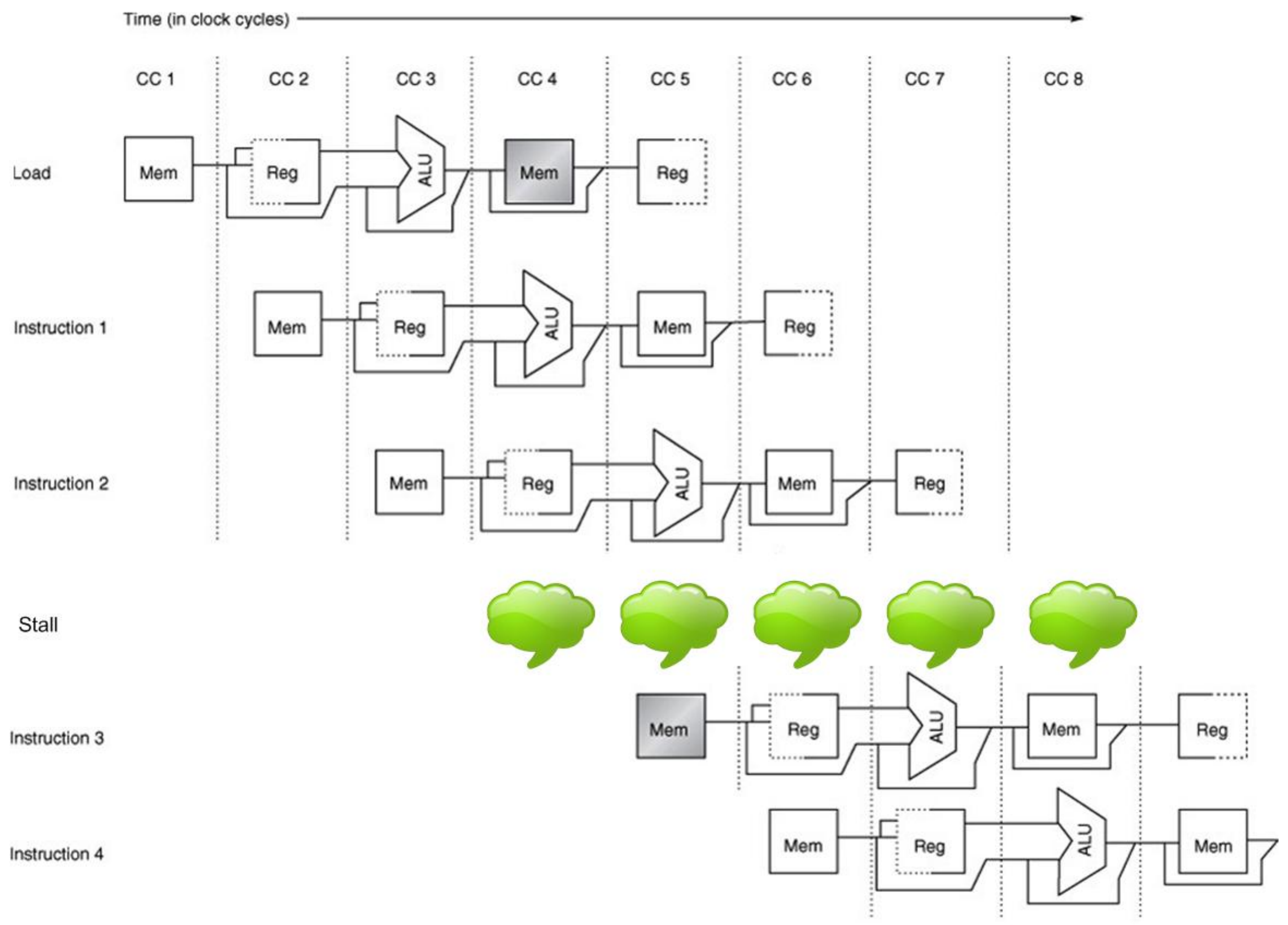
- Instruction fetch depends on decision about control flow
- Example: Branches and jumps change PC

Structural Hazards



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

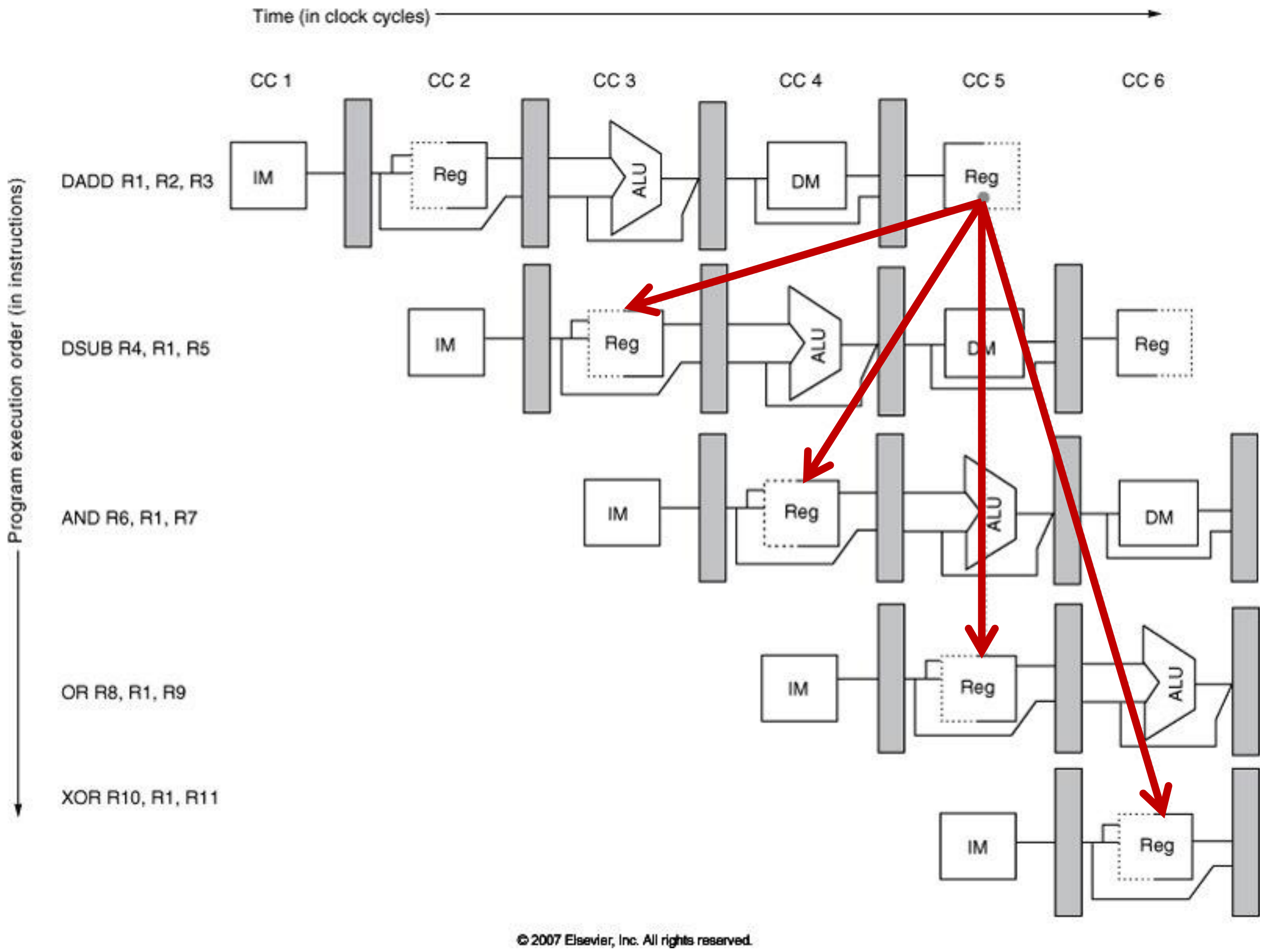
Structural Hazards



Stall the pipeline, creating bubbles, by freezing earlier stages → interlocks

Use Harvard Architecture (separate instruction, data memories)

Data Hazards



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 tries to write operand (r1) before Instr-I reads it

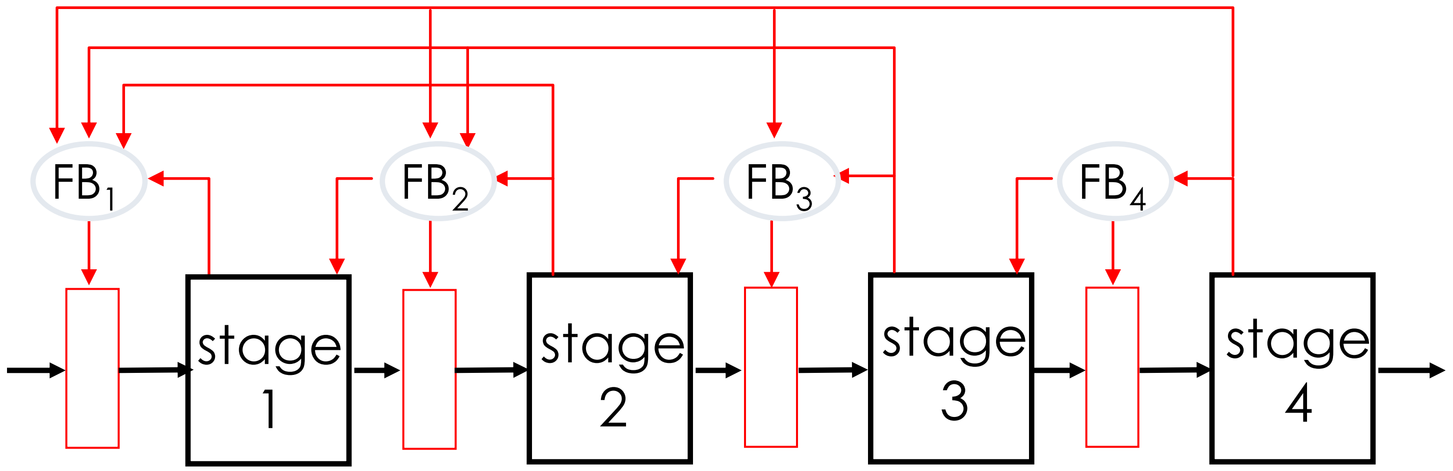
```
i: add r4, r1, r3
j: add r1, r2, r3
k: mul r6, r1, r7
```

Write After Write (WAW)

- Caused by an output dependence and the re-use of the name "r1"
- Instr-j tries to write 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

time

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
(I_1) r1 ← (r0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1							
(I_2) r4 ← (r1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2			
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)							IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)								IF_5	ID_5	EX_5	MA_5	WB_5

stalled stages

time

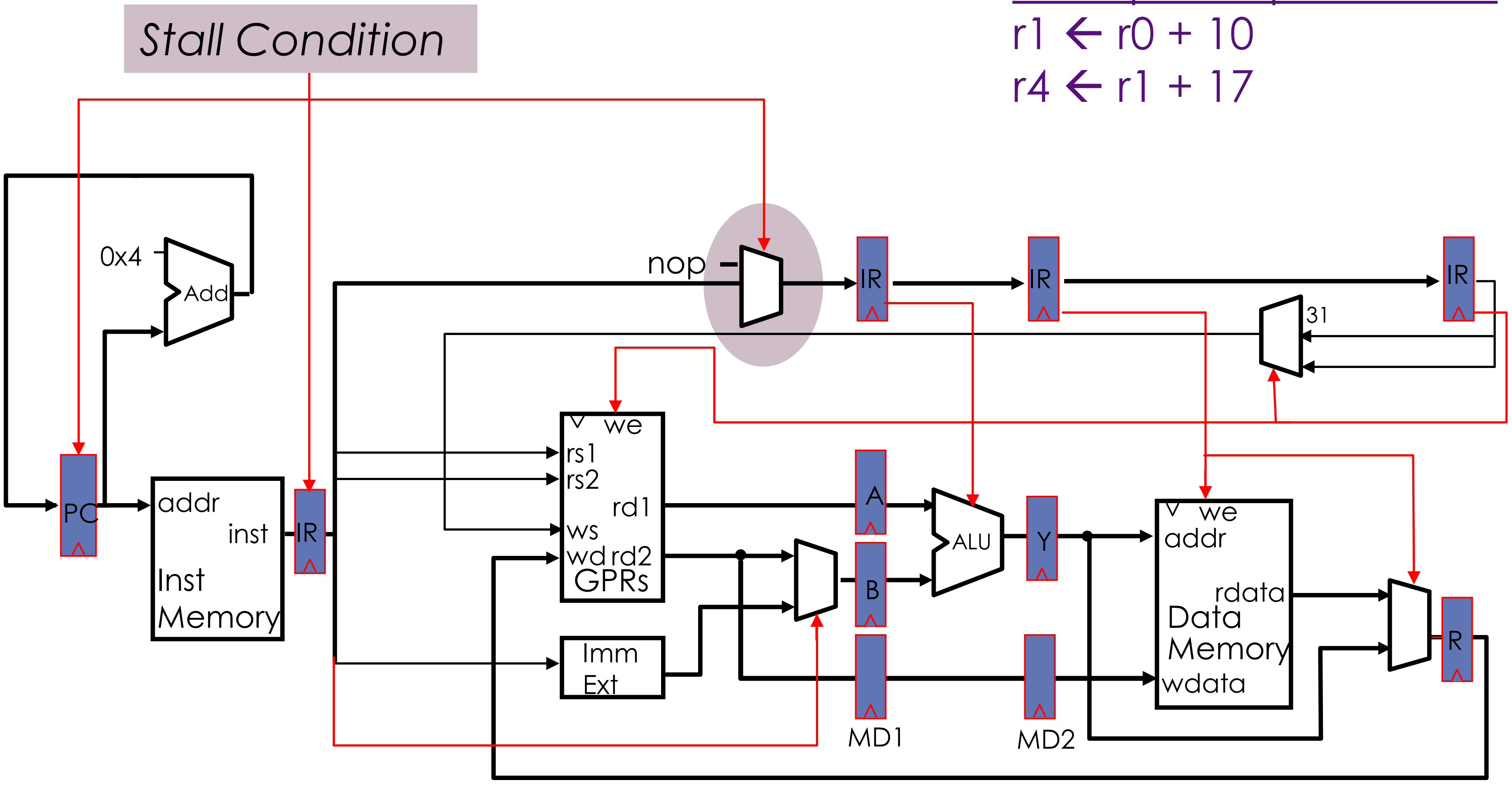
Resource Usage		t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
	IF	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5				
	ID		I_1	I_2	I_2	I_2	I_2	I_3	I_4	I_5			
	EX			I_1	nop	nop	nop	I_2	I_3	I_4	I_5		
	MA				I_1	nop	nop	nop	I_2	I_3	I_4	I_5	
	WB					I_1	nop	nop	nop	I_2	I_3	I_4	I_5

Interlocks & Pipeline Stalls

Example Dependence

r1 ← r0 + 10

r4 ← r1 + 17



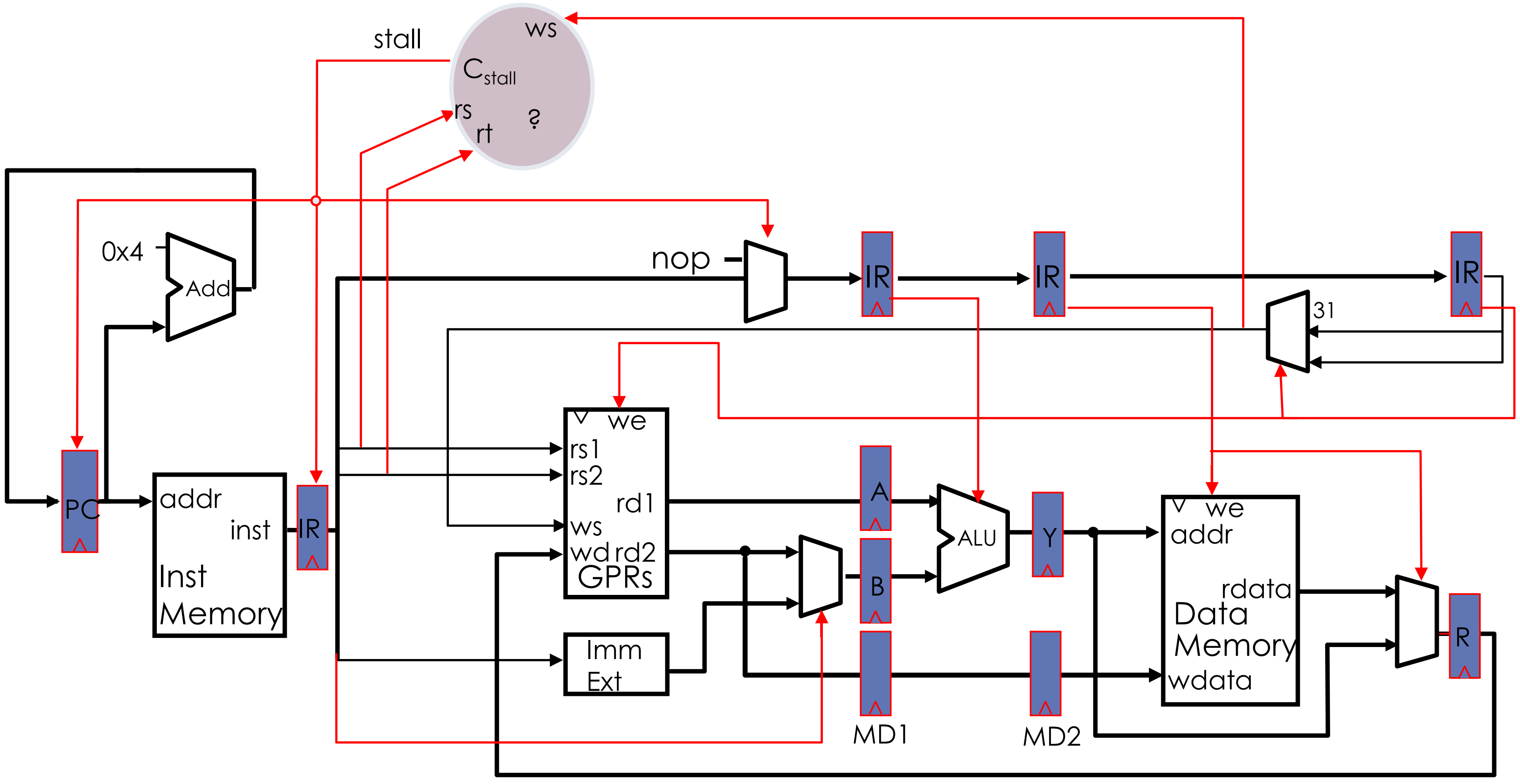
Interlock Control Logic

- Compare the <u>source registers</u> of instruction in decode stage with the <u>destination registers</u> 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

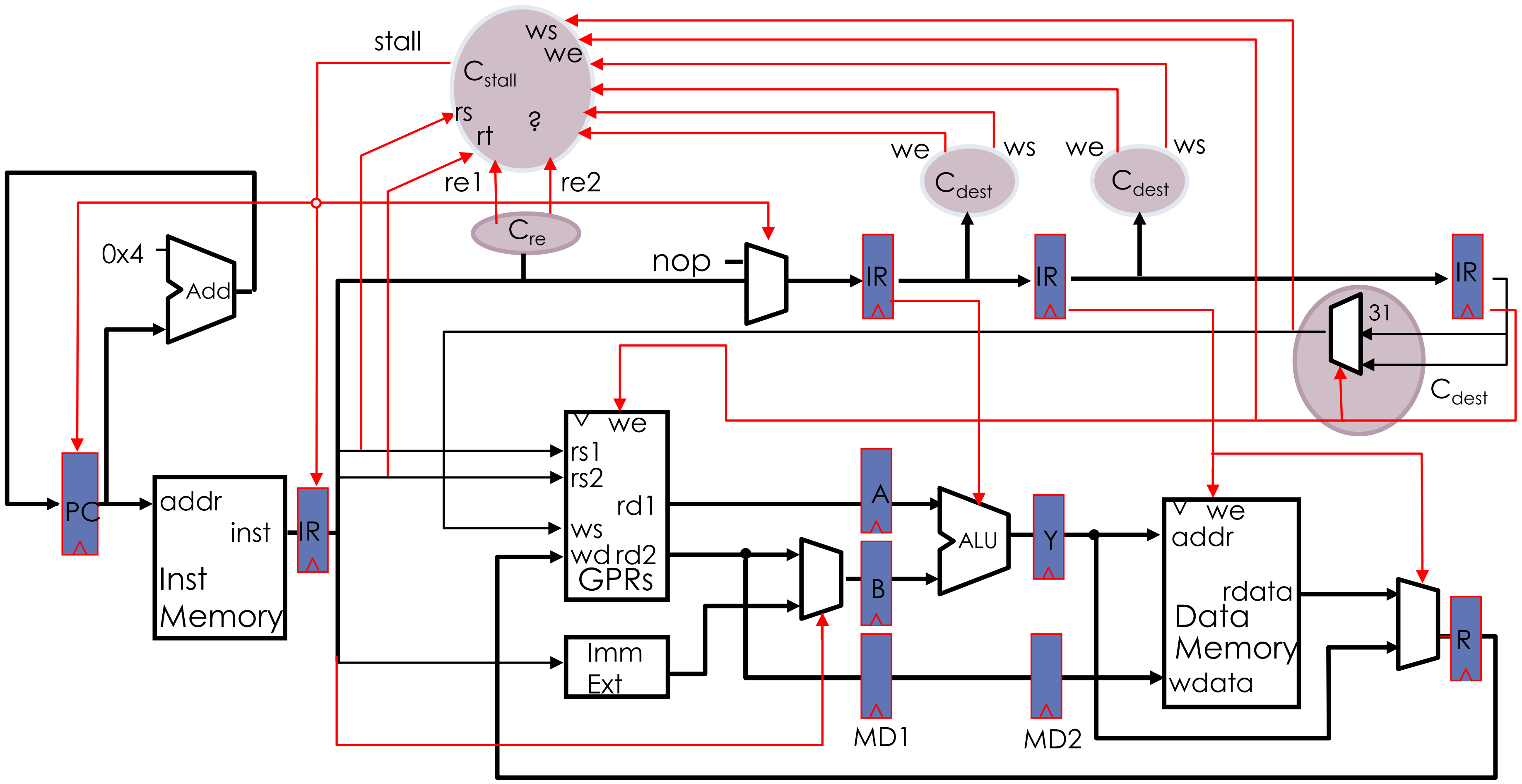
- Derive stall signal from conditions in the pipeline

Interlock Control Logic



Compare the *source registers* of the instruction in the decode stage (rs, rt) with the *destination register* of the *uncommitted* instructions (ws).

Interlock Control Logic



Should we always stall if RS/RT matches some WS? 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:

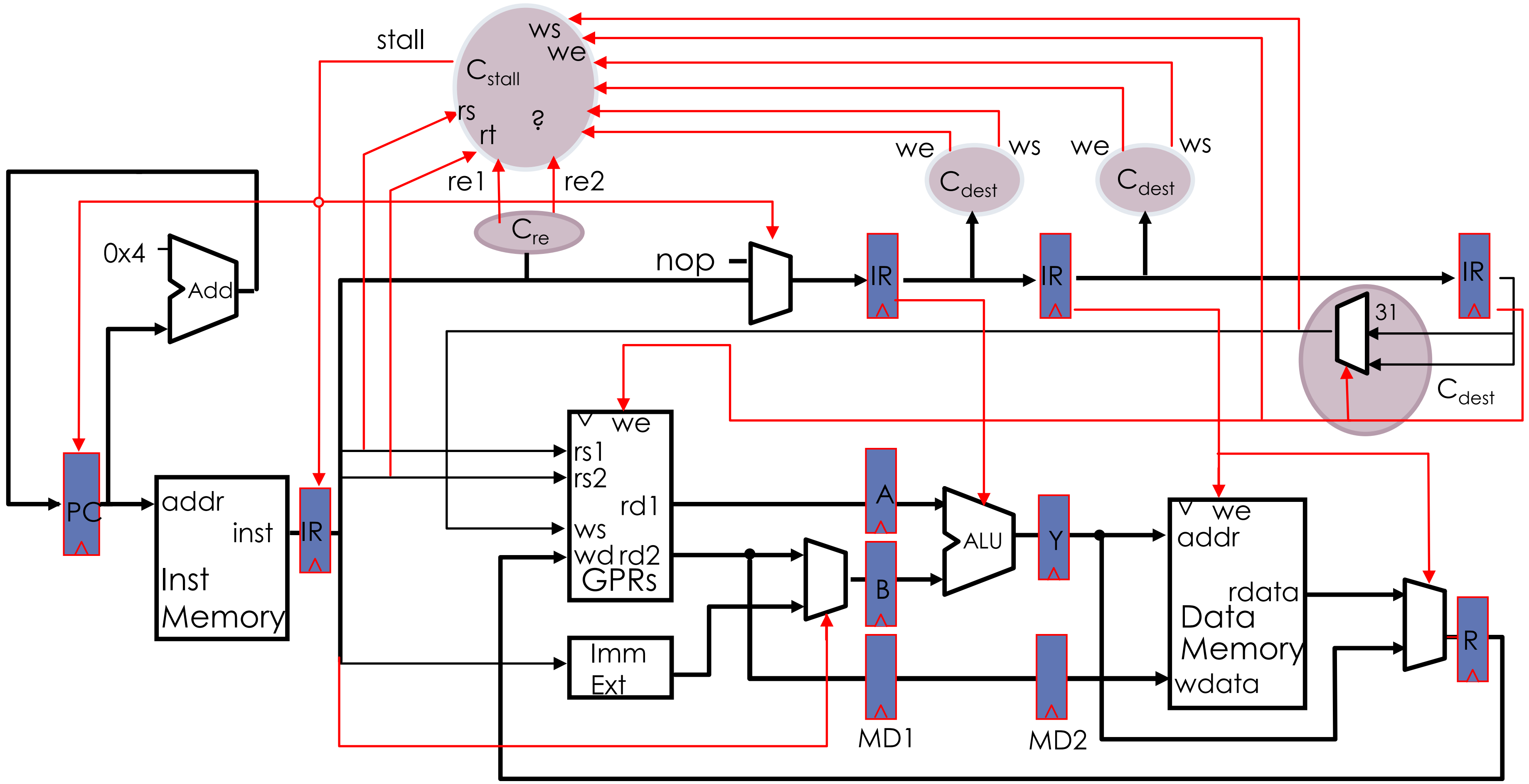
op	rs	rt	immediate16

J-type:

op	immediate26

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

<u>Cdest</u>	ws	Case(opcode)	
		ALU:	ws ← rd
		ALUi:	ws ← rt
		JAL, JALR:	ws ← R31
	we	Case(opcode)	
		ALU, ALUi, LW	we ← (ws != 0)
		JAL, JALR	we ← 1
		otherwise	we ← 0
<u>Cre</u>	re1	Case(opcode)	
		ALU, ALUi	re1 ← 1
		LW, SW, BZ	re1 ← 1
		JR, JALR	re1 ← 1
		J, JAL	re1 ← 0
	re2	Case(opcode)	
		<< same as re1 but for register rt>>	

Deriving the Stall Signal

Notation: [pipeline-stage][signal]
E.g., Drs – rs signal from decode stage
E.g., Ewe – we signal from execute stage

<u>Cstall</u>

```
stall-1 ← (     (Drs == Ews) & Ewe |
                (Drs == Mws) & Mwe |
                (Drs == Wws) & Wwe
          ) & Dre1

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

stall ←   stall-1 | 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

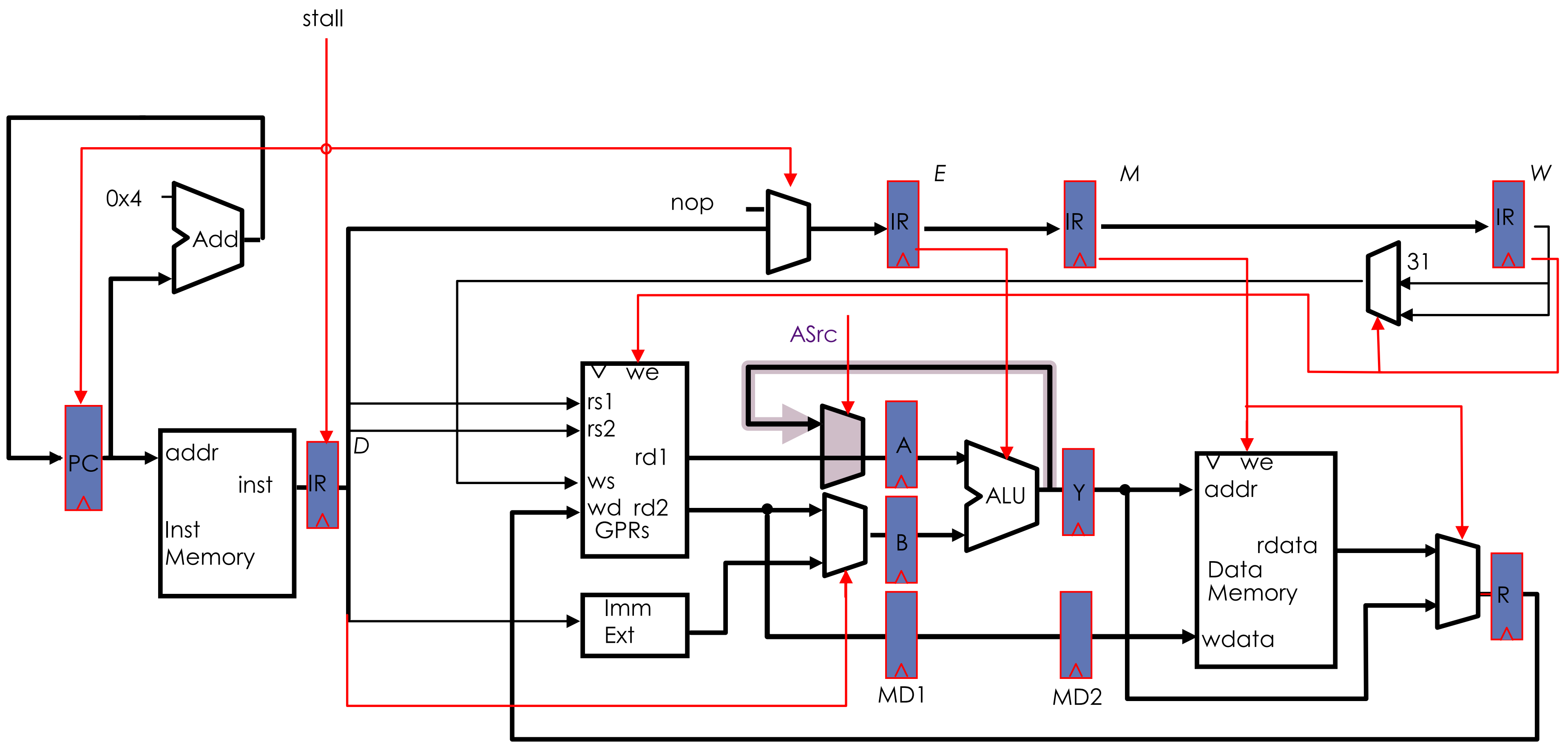
Strategy 2 – Forwarding (aka Bypasses)

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

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .	
(I_1) r1 ← r0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) r4 ← r1 + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2	
(I_3)				IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3
				stalled stages						
(I_4)								IF_4	ID_4	EX_4
(I_5)									IF_5	ID_5

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .	
(I_1) r1 ← r0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1					
(I_2) r4 ← r1 + 17		IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3)				IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)					IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)						IF_5	ID_5	EX_5	MA_5	WB_5

Example Forwarding Path



Deriving Forwarding Signals

This forwarding path only applies to the ALU operations…

```
Eforward        Case(Eopcode)
                ALU, ALUi       Eforward ← (ws != 0)
                otherwise       Eforward ← 0
```

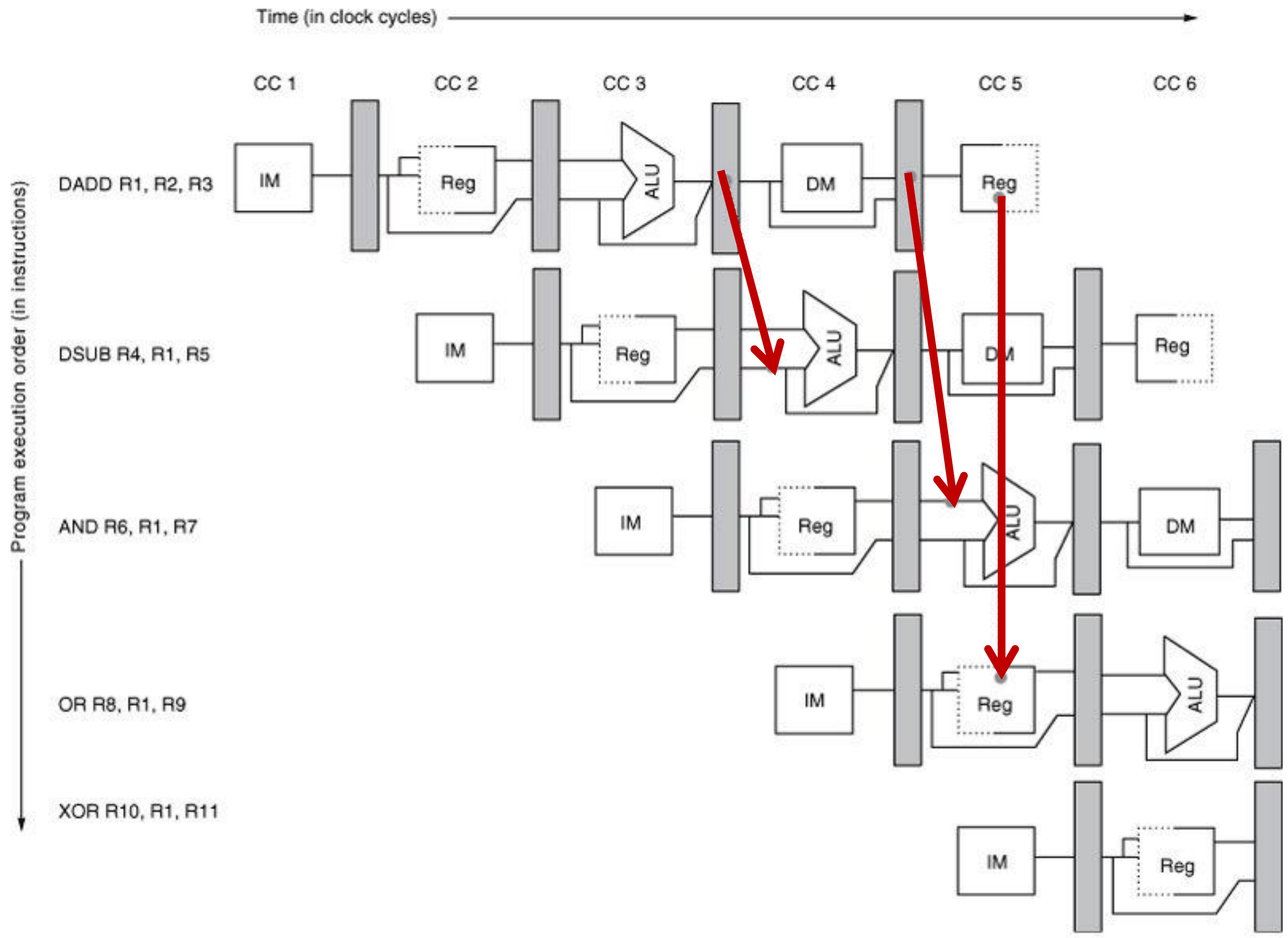
…and all other operations will need to stall as before

```
Estall          Case(Eopcode)
                LW              Estall ← (ws != 0)
                JAL, JALR       Estall ← 1
                otherwise       Estall ← 0
```

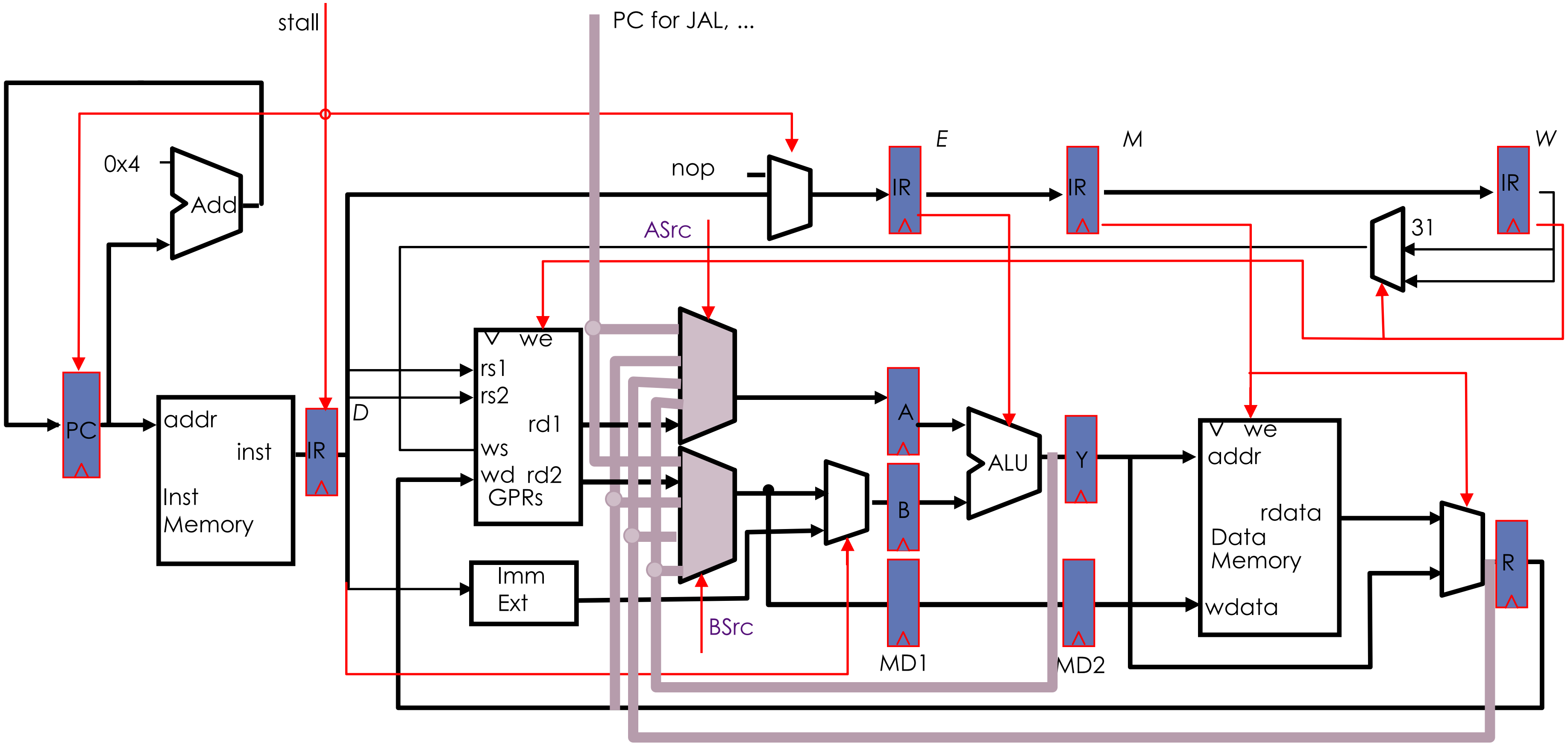
```
Asrc ← (Drs == Ews) & Dre1 & Eforward
```

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

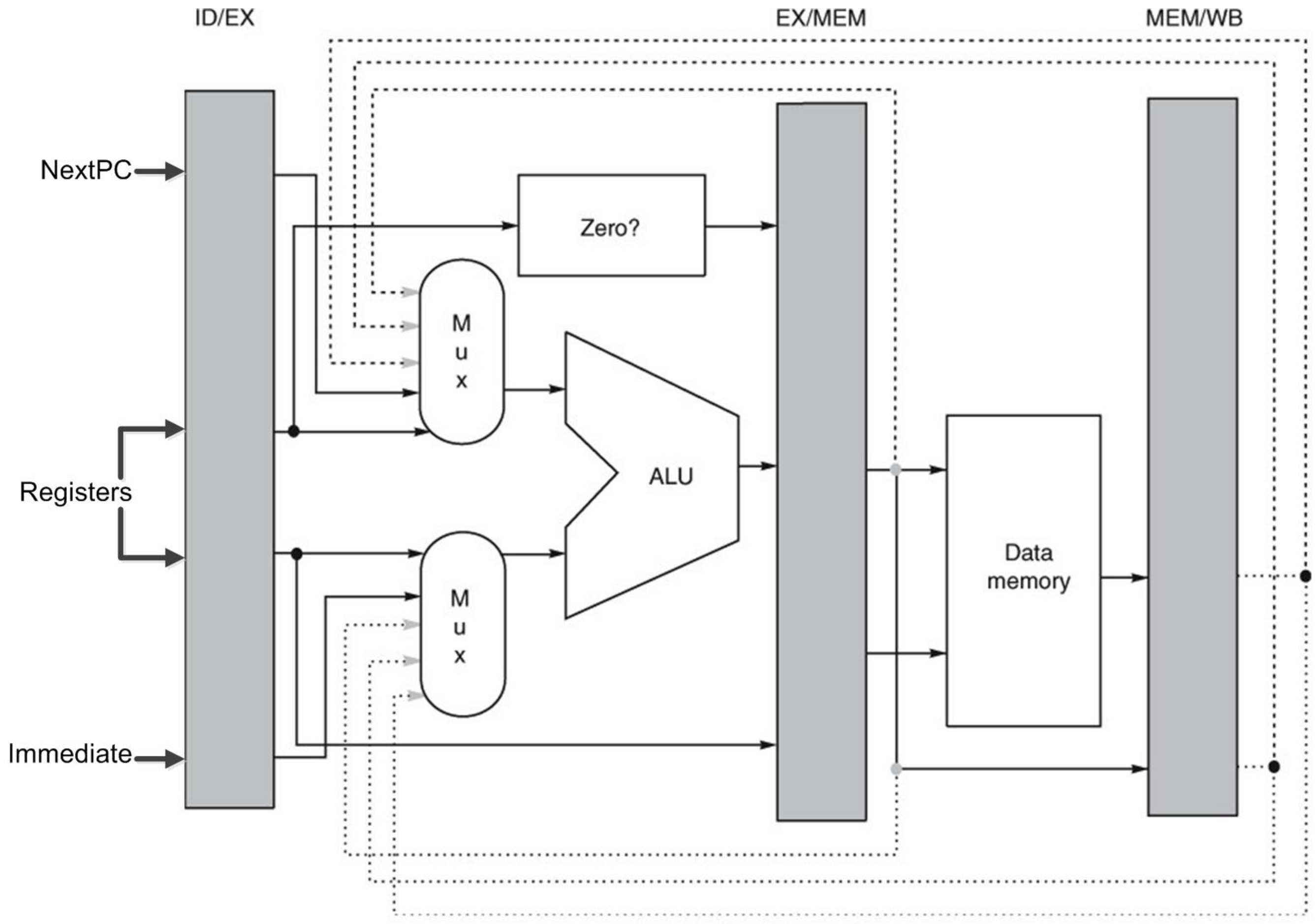
Multiple Forwarding Paths



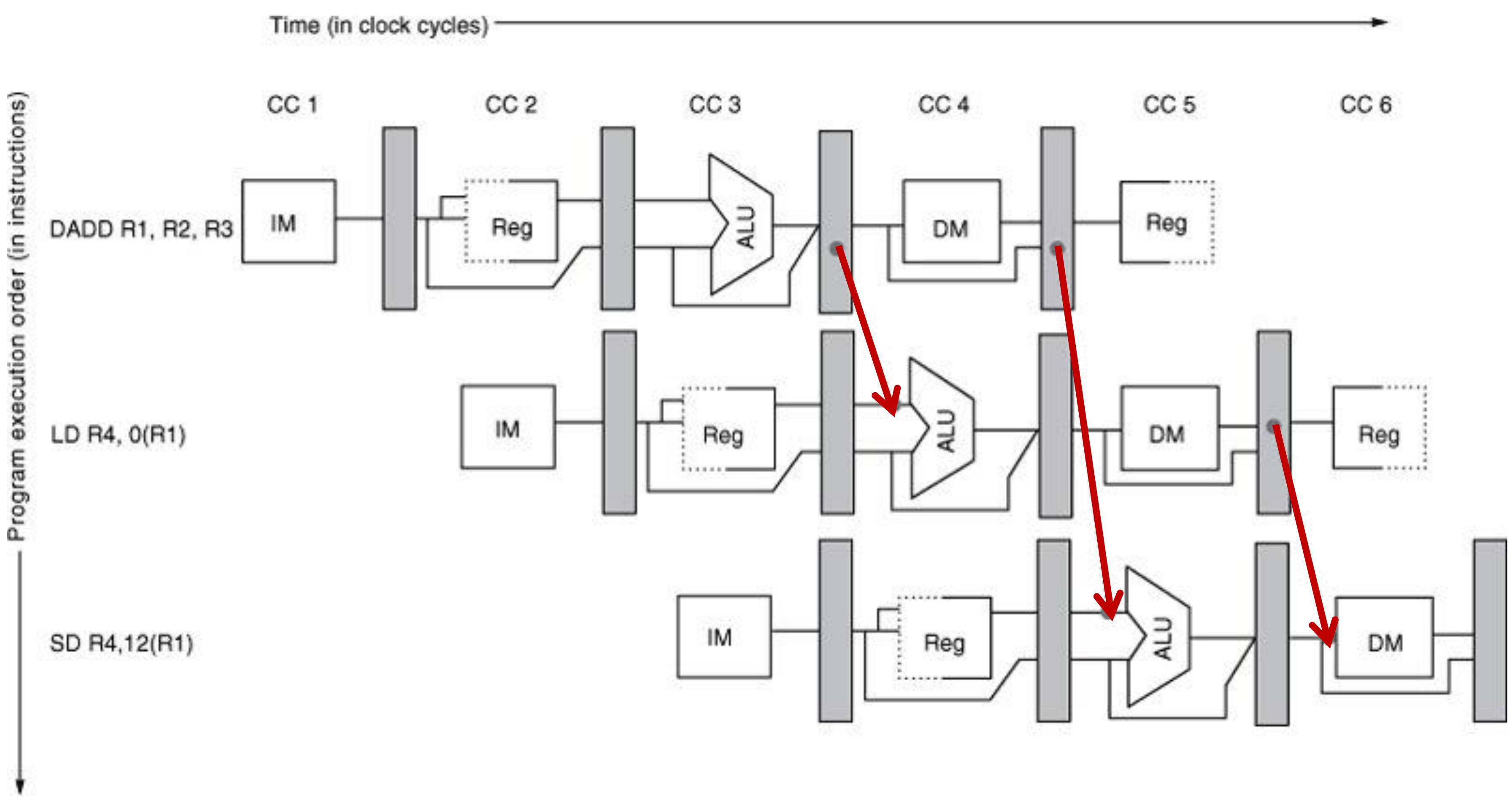
Multiple Forwarding Paths



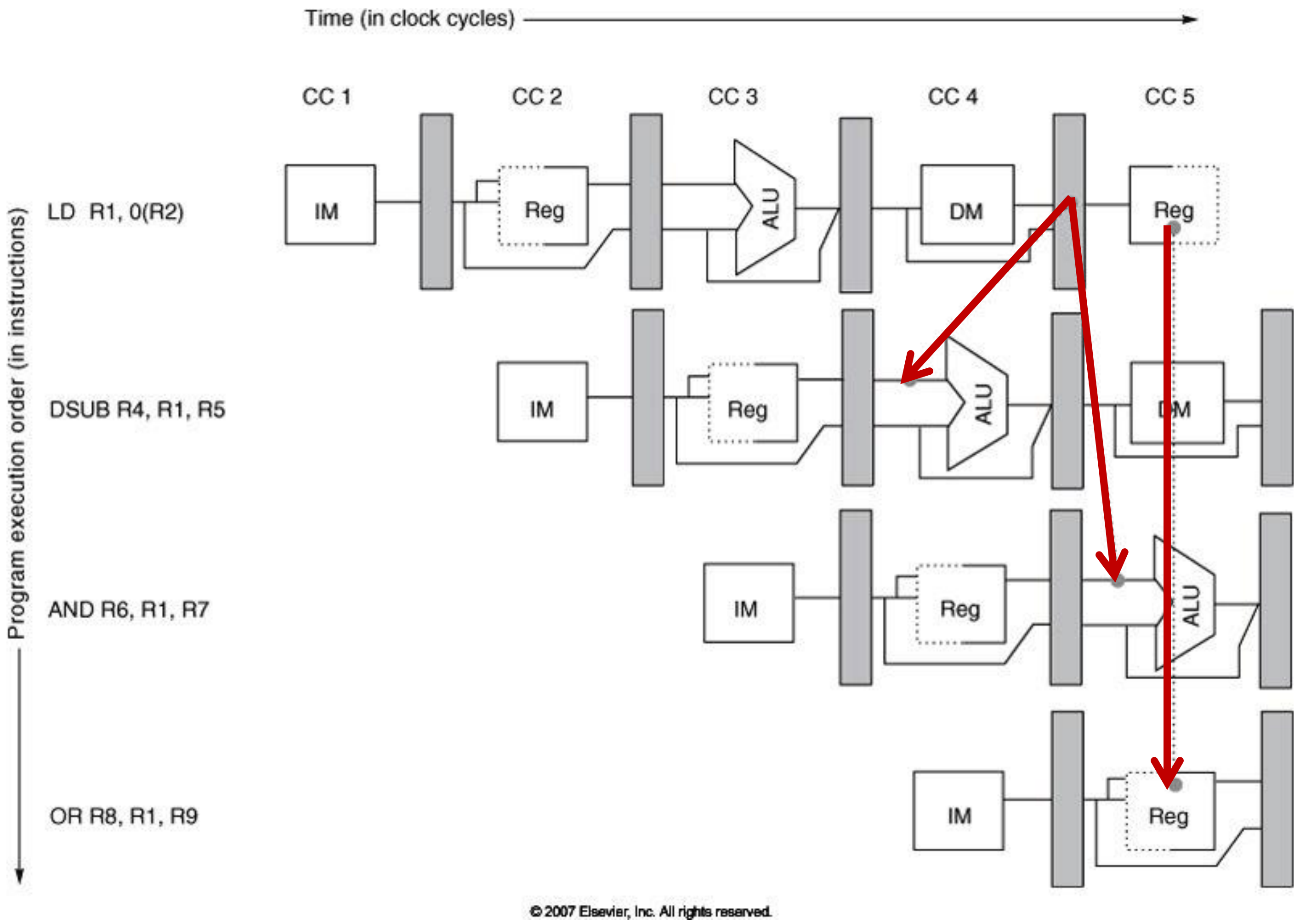
Forwarding Hardware



Forwarding Loads/Stores



Data Hazard Despite Forwarding



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		**Fast Code**	
LW	Rb, b	LW	Rb, b
LW	Rc, c	LW	Rc, c
ADD	Ra, Rb, Rc	LW	Re, e
SW	a, Ra	ADD	Ra, Rb, Rc
LW	Re e	LW	Rf, f
LW	Rf, f	SW	a, Ra
SUB	Rd, Re, Rf	SUB	Rd, Re, Rf
SW	d, RD	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 Kubiatowicz (UCB)
- Alvin Lebeck (Duke)
- David Patterson (UCB)
- Daniel Sorin (Duke)