Pipelining

CSCM601252 – Pengantar Organisasi Komputer

Instructor: Erdefi Rakun

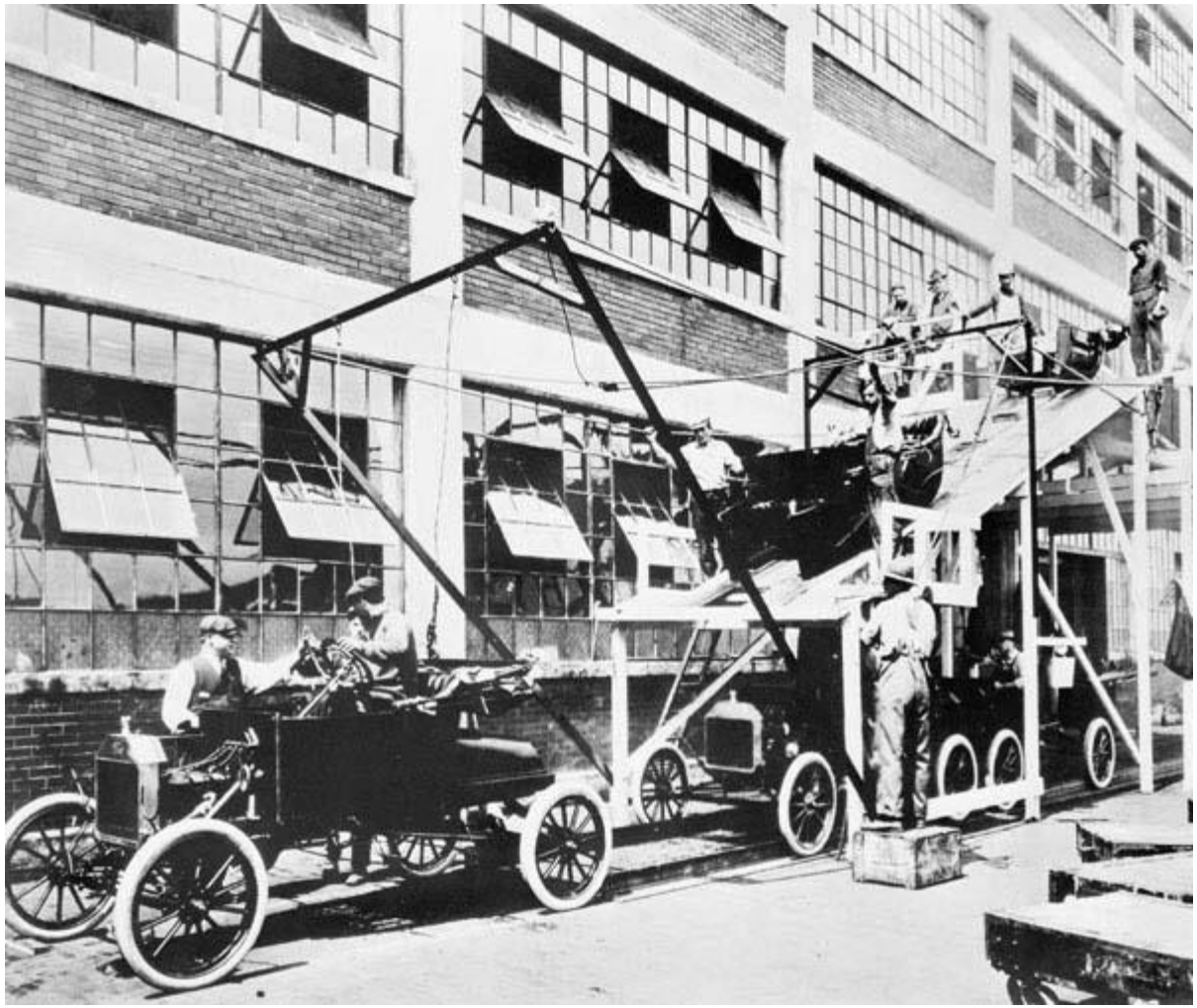
Fasilkom UI

Outline

- MIPS Pipeline Stages
- Pipelined Datapath
- Pipelined Execution
- Pipeline Control
- Pipelined Datapath with Control
- Pipeline Hazards
  - Structural
  - Data
  - Control

Note: These slides are taken from Aaron Tan's slide

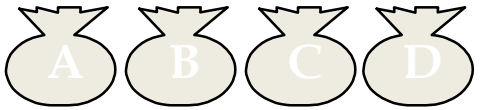
Assembly Line

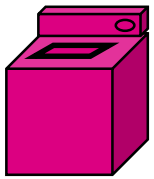


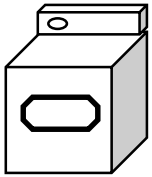
Henry Ford invented an assembly line (pipeline flow) for greater efficiency in 1913

Laundry Example

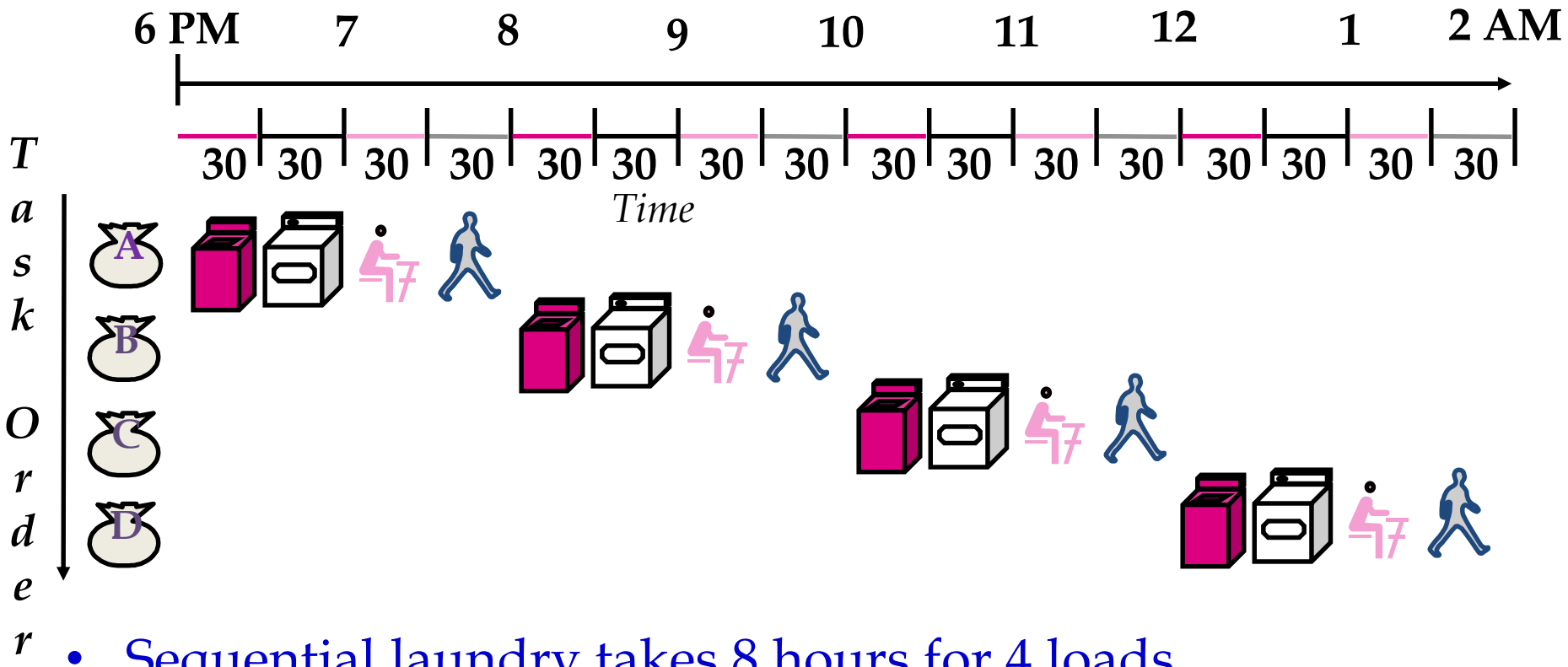
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- "Folder" takes 30 minutes
- "Stasher" takes 30 minutes to put clothes into drawers





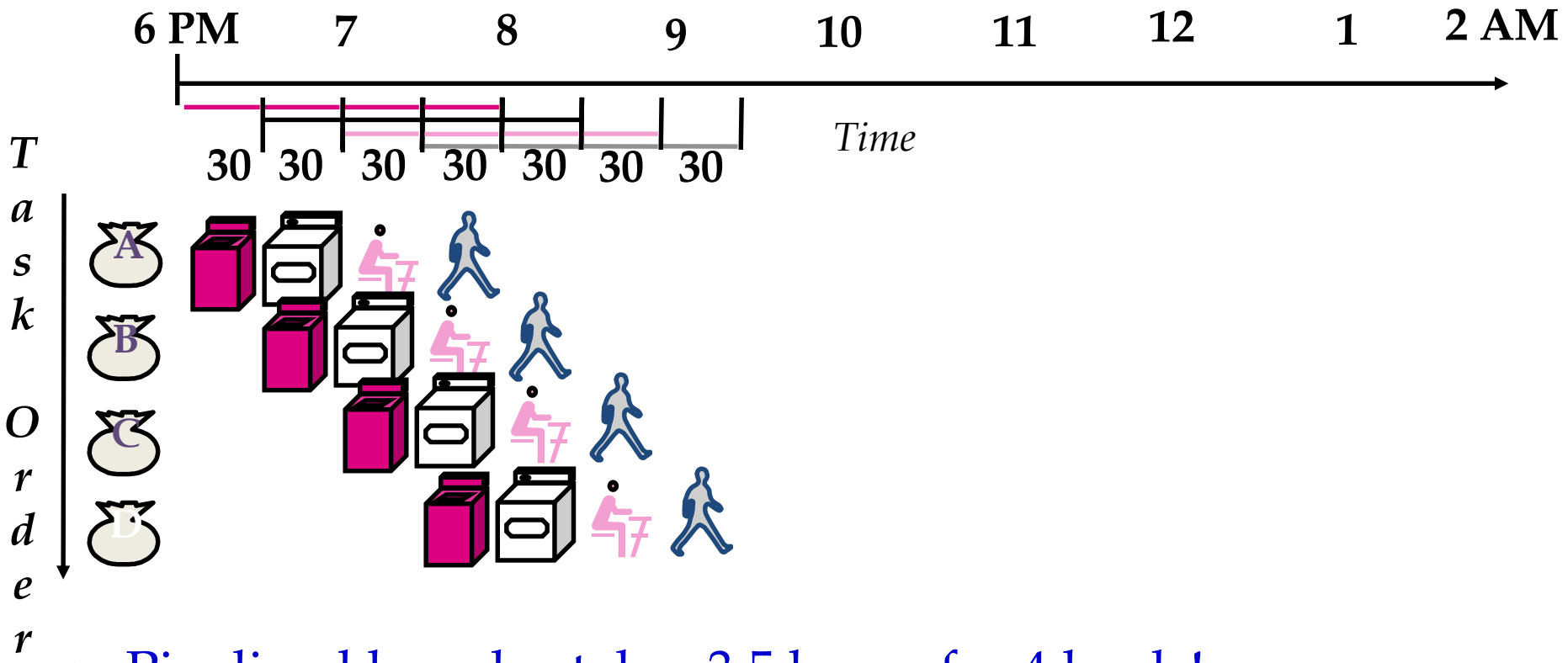


Sequential Laundry



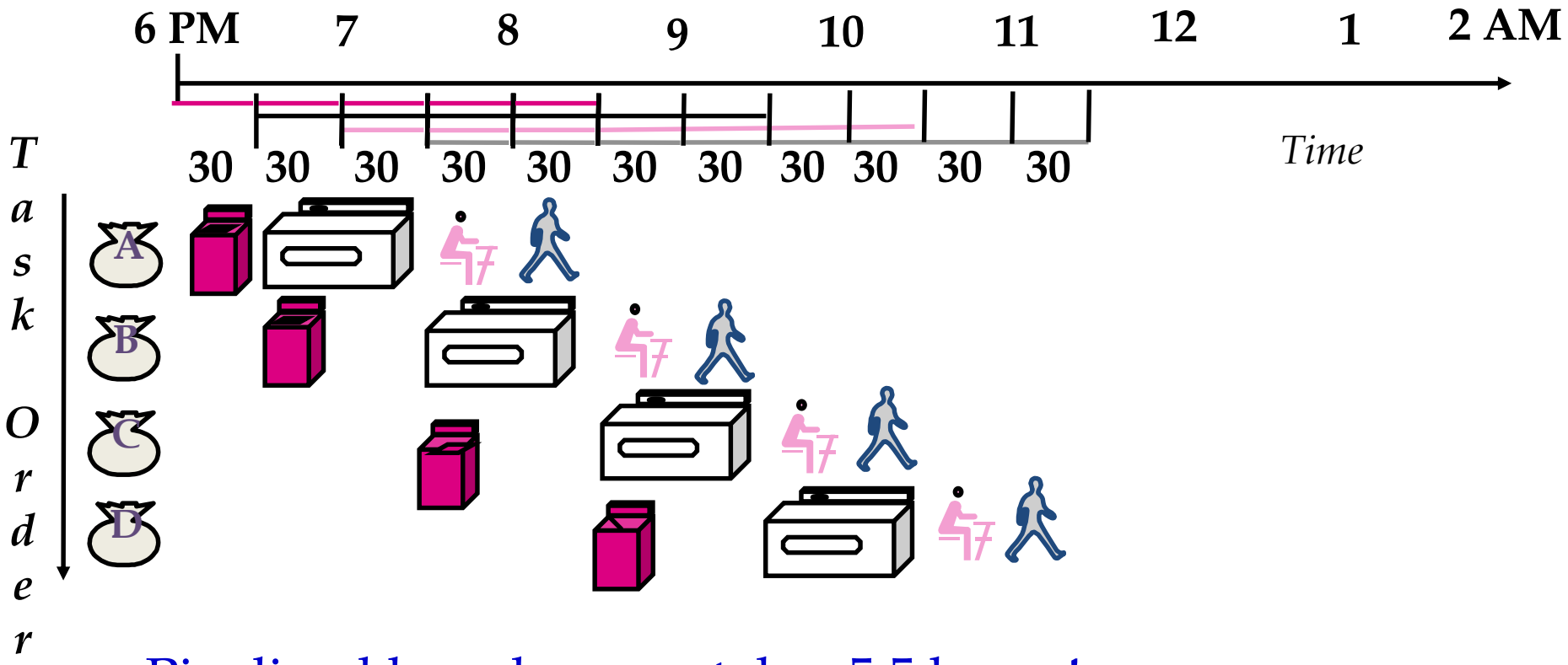
- Sequential laundry takes 8 hours for 4 loads
- Steady state: 1 load every 2 hours
- If they learned pipelining, how long would laundry take?

Pipelined Laundry



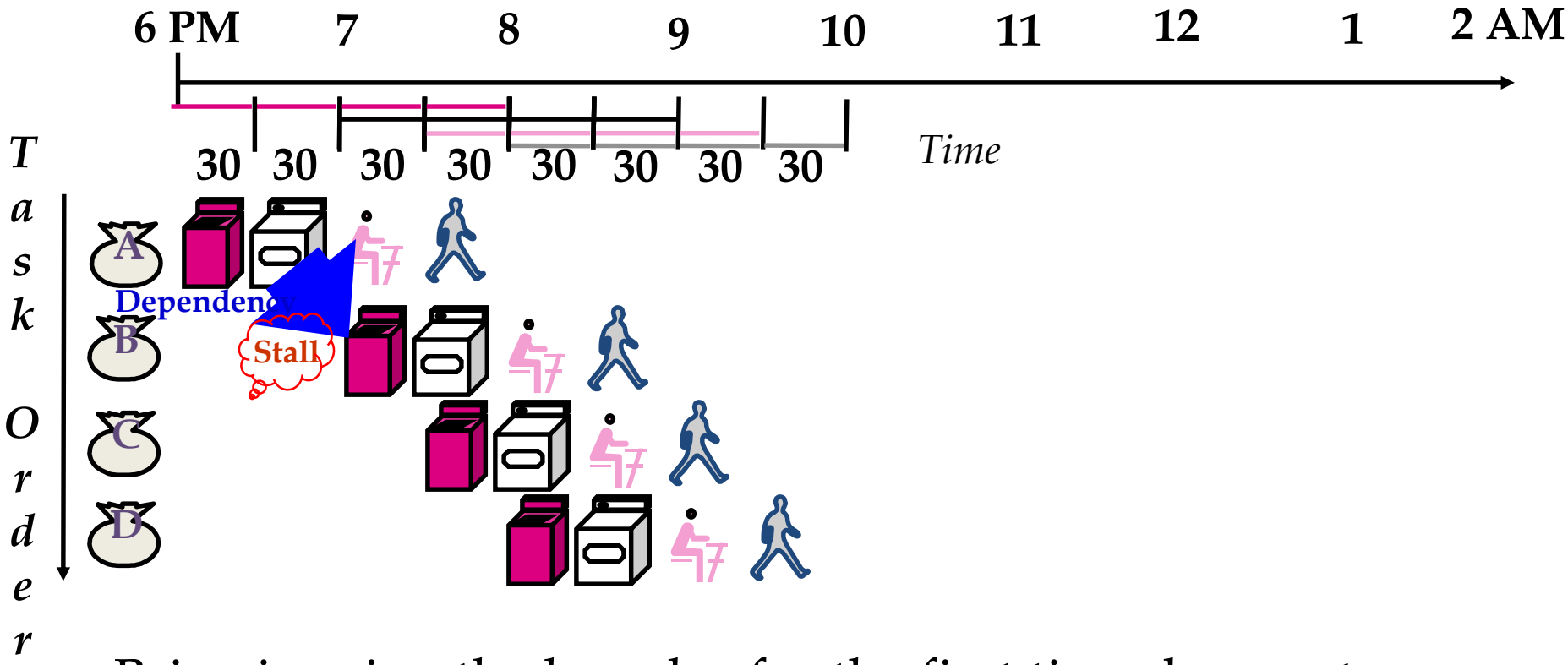
- Pipelined laundry takes 3.5 hours for 4 loads!
- Steady state: 1 load every 30 min
- Potential speedup = 2 hr/30 min = 4 (# stages)
- Time to fill pipeline takes 2 hours → speedup ↓

Slow Dryer



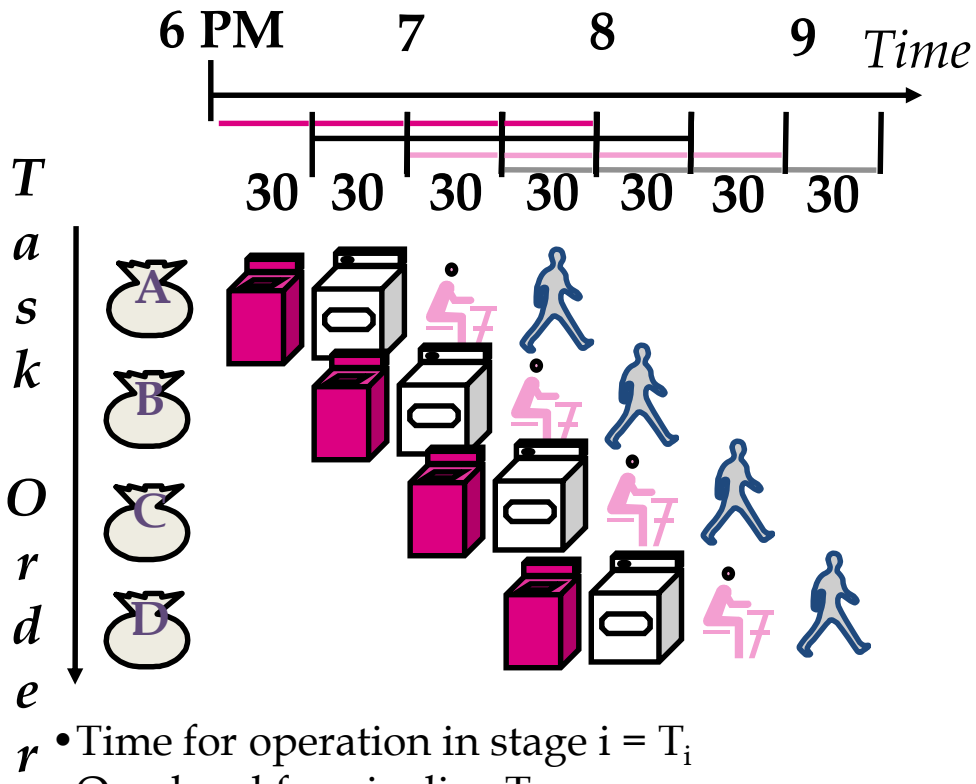
- Pipelined laundry now takes 5.5 hours!
- Steady state: One load every 1 hr (dryer speed)
- **Pipeline rate is limited by the slowest stage**

Dependency



- Brian is using the laundry for the first time; he wants to see the outcome of one wash + dry cycle first before putting in his clothes
- Pipelined laundry now takes 4 hours

Pipelining Lessons



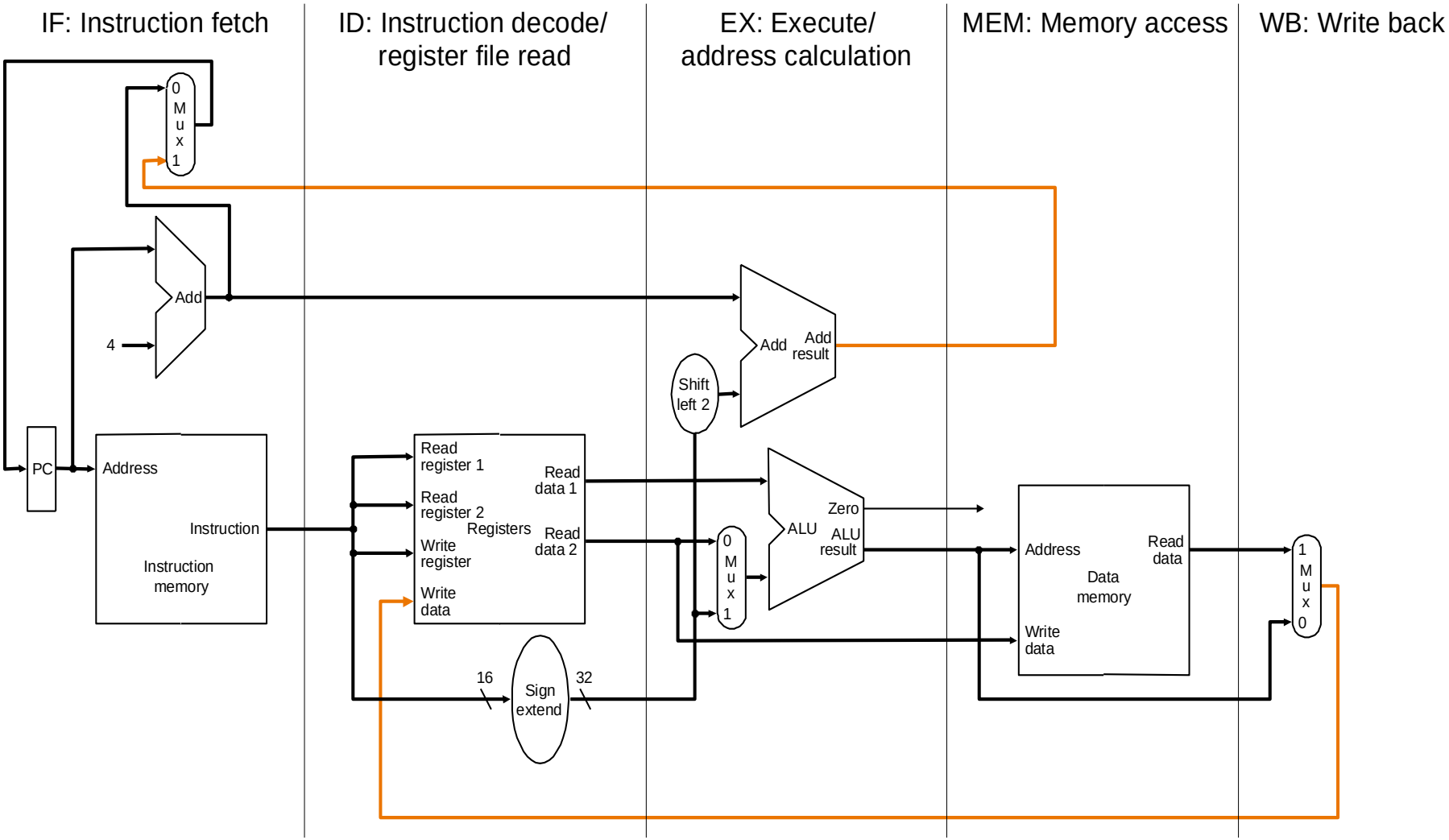
- Time for operation in stage i = T_i
- Overhead for pipeline T_d
- Time without pipelining $T_{seq} = \Sigma T_i$
- Time with pipelining
  $T_{pipe} = \max(T_i) + T_d$
- Speedup = Number of pipeline stages if evenly balanced and $T_d = 0$

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

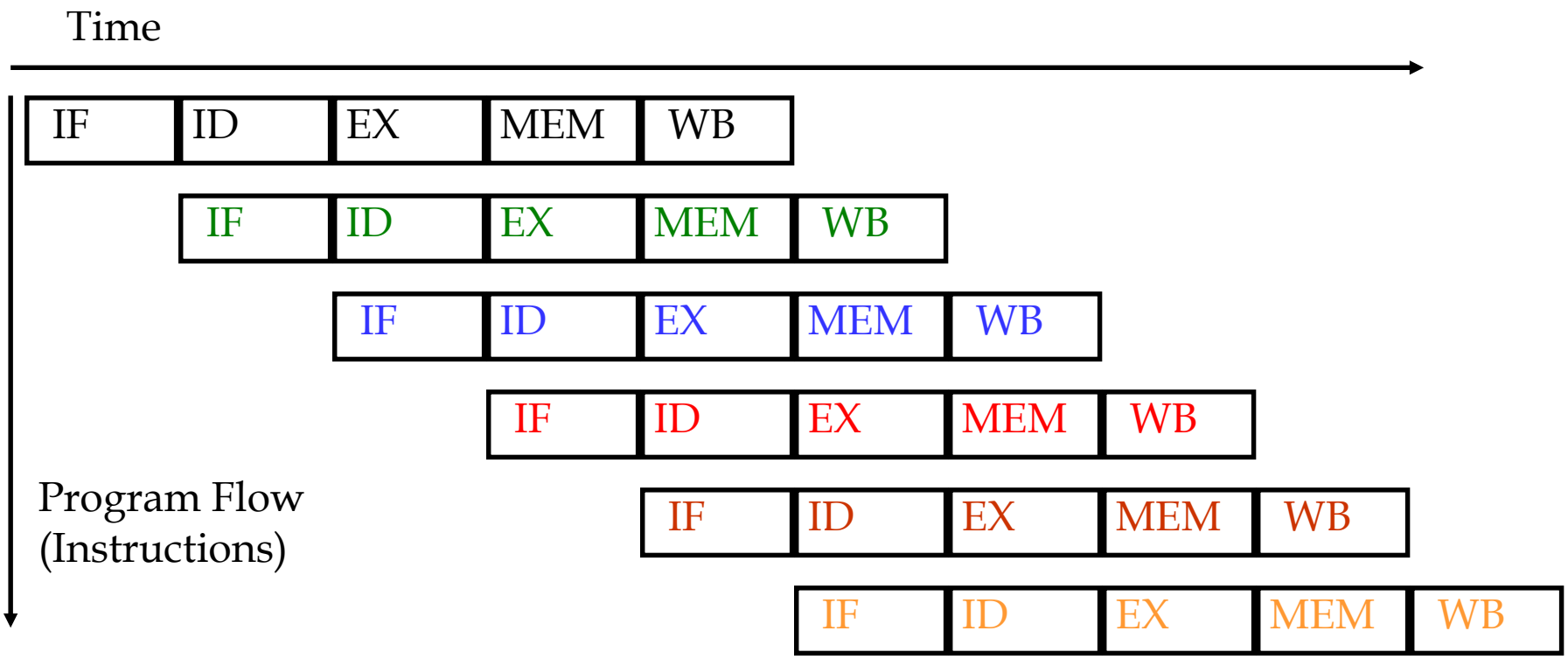
MIPS Pipeline Stages (1/2)

- Five Execution Stages
  - **IF**: Instruction Fetch
  - **ID**: Instruction Decode and Register Read
  - **EX**: Execute an operation or calculate an address
  - **MEM**: Access an operand in data memory
  - **WB**: Write back the result into a register
- Each execution stage takes 1 clock cycle
- General flow of data is from left to right
- Exceptions: Update of PC and write back of register file – more about this later…

MIPS Pipeline Stages (2/2)



Pipelined Execution Representation



Advantage Of Pipeline (1/2)

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

FIGURE 4.26 Total time for each instruction calculated from the time for each component. This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Advantage Of Pipeline (2/2)

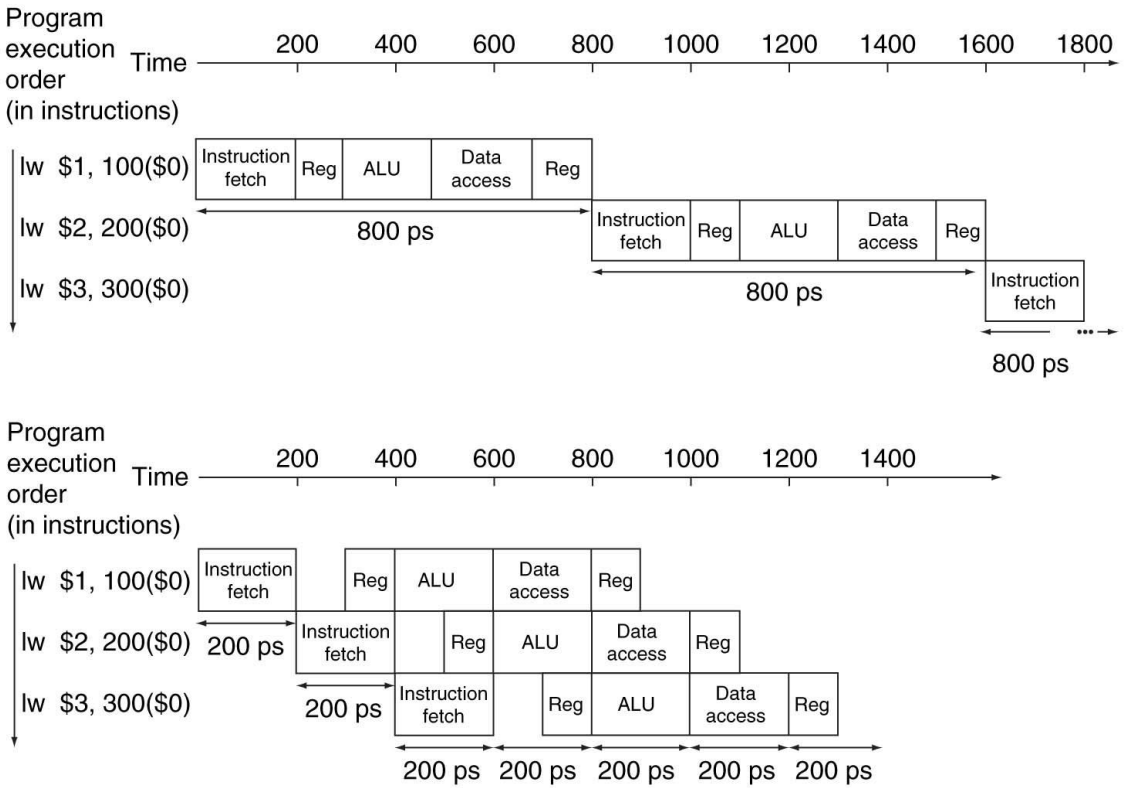
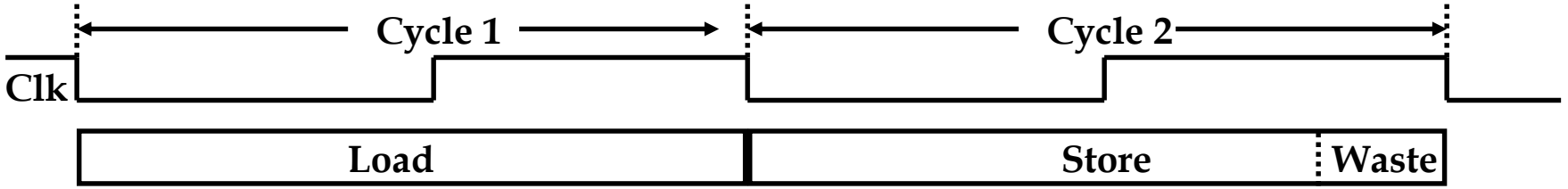


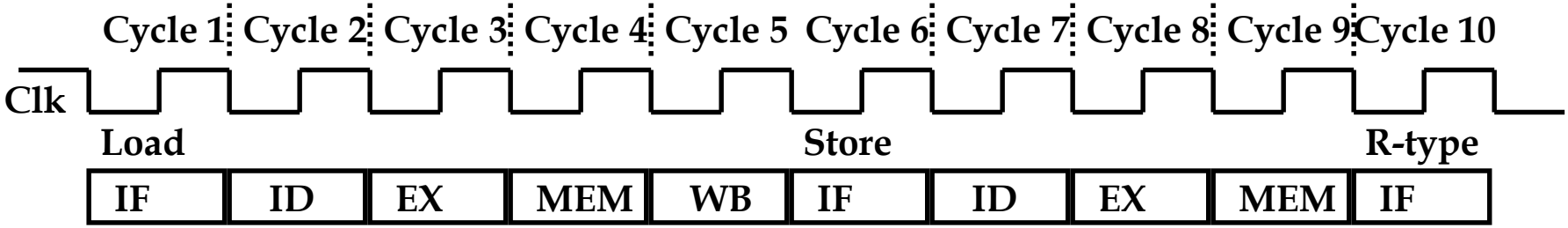
FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register fi le occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

Single-Cycle, Multi-Cycle, Pipeline (1/2)

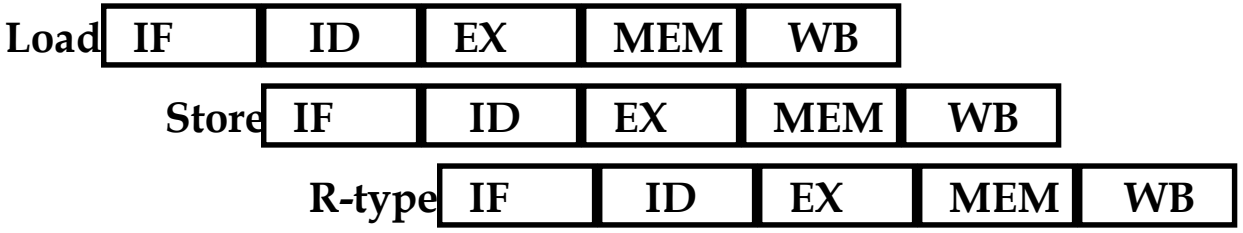
- **Single-cycle implementation**



- **Multi-cycle implementation**



- **Pipeline implementation**



Single-Cycle, Multi-Cycle, Pipeline (2/2)

- Suppose we execute 100 instructions
- Single Cycle Machine
  - 800ps /cycle × 1 CPI × 100 inst = 80000 ps = 80 ns
- Multicycle Machine
  - 200 ps/cycle × 4.6 CPI (due to inst mix) × 100 inst = 92.000 ps = 92 ns
- Ideal pipelined machine
  - 200 ps/cycle × (1 CPI × 100 inst + 4 cycle pipeline fill) = 20800 ps = 20.8 ns

Review Question

- Given this code:

```
add $t0, $s0, $s1
sub $t1, $s0, $s1
sll $t2, $s0, 2
srl $t3, $s1, 2
```

a) How many cycles will it take to execute the code on a single-cycle datapath?

b) How long will it take to execute the code on a single-cycle datapath, assuming a 100 MHz clock?

c) How many cycles will it take to execute the code on a 5-stage MIPS pipeline?

d) How long will it take to execute the code on a 5-stage MIPS pipeline? Assume pipeline is completely balanced and the single-cycle datapath could support 100MHz clock frequency.

Pipelined Datapath (1/3)

- Single-cycle implementation
  - Update all state elements (**PC**, register file, data memory) at the end of a clock cycle
- Pipelined implementation
  - One cycle per pipeline stage
  - All data that is used in subsequent clock cycles must be stored in state elements
  - May result in timing race if not stored properly – can you see why?

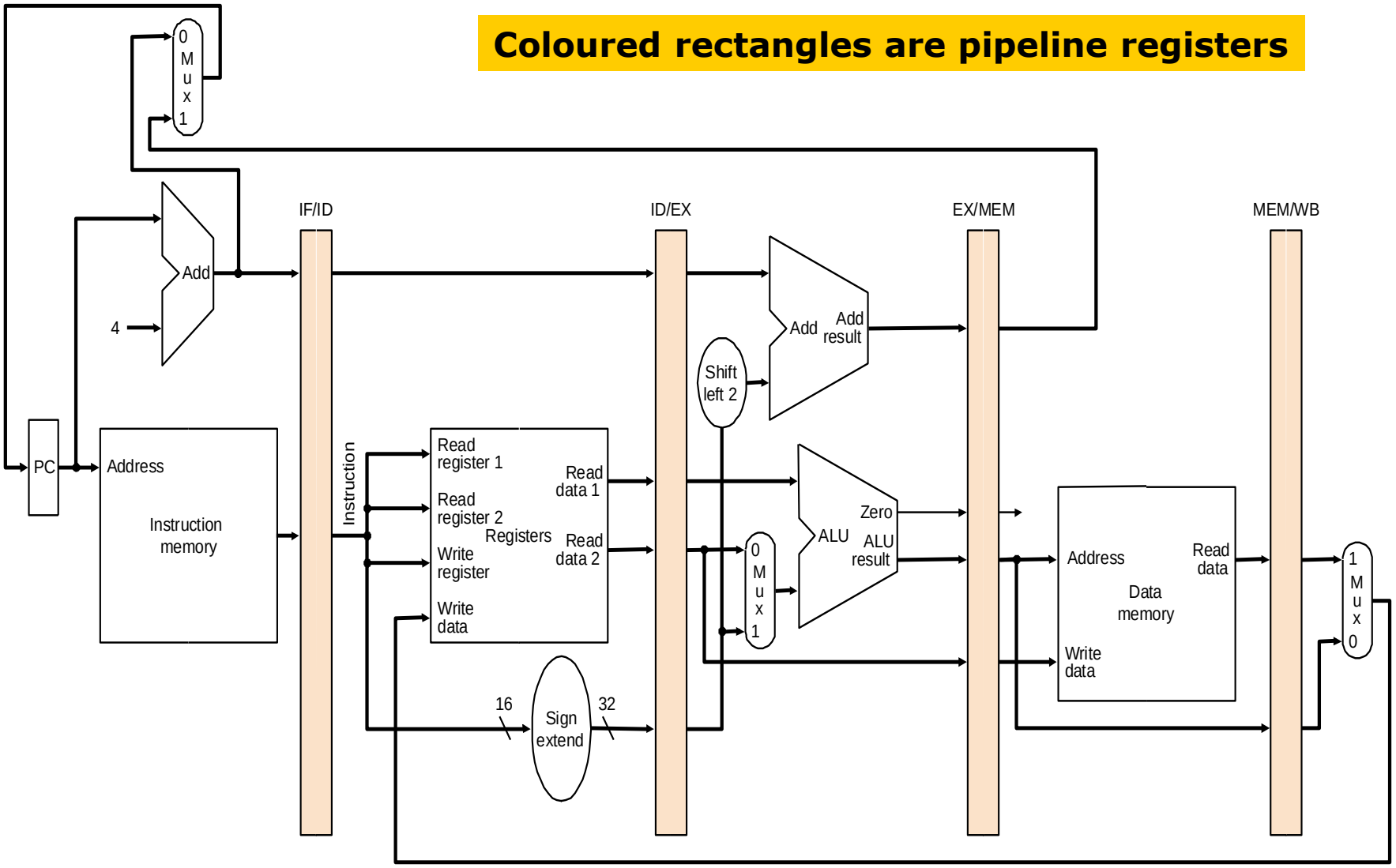
Ack: These slides are taken from Dr Tulika Mitra's CS1104 notes.

Pipelined Datapath (2/3)

- Data used by subsequent instructions in later clock cycles
  - Store in programmer-visible state elements: **PC**, register file and memory
- Data used by same instruction in later clock cycle
  - Additional registers in datapath called pipeline registers
  - **IF/ID:** register between **IF** and **ID**
  - **ID/EX:** register between **ID** and **EX**
  - **EX/MEM:** register between **EX** and **MEM**
  - **MEM/WB:** register between **MEM** and **WB**
- Why no register at the end of **WB** stage?

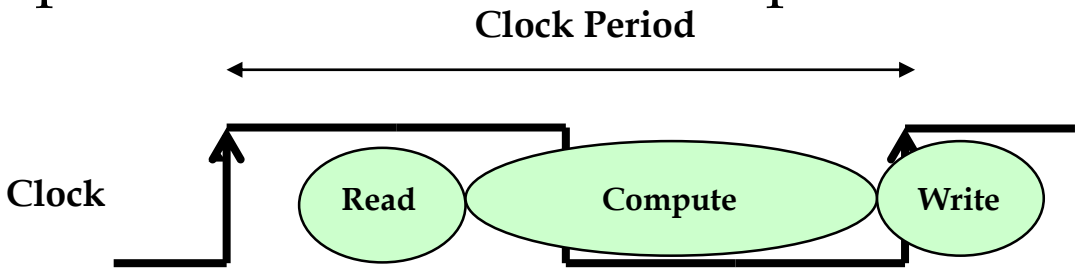
Pipelined Datapath (3/3)

Coloured rectangles are pipeline registers



Big Picture: Instruction Execution

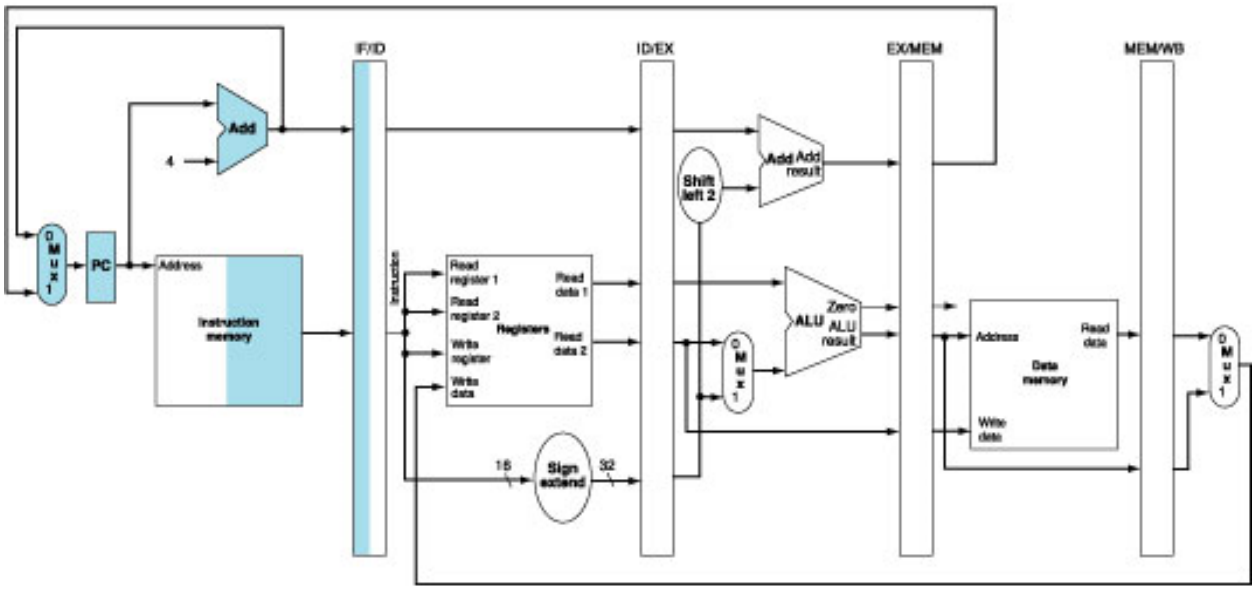
- Instruction execution is equivalent to
  - Read contents of one or more storage elements (register/memory)
  - Perform computation through some combinational logic
  - Write results to one or more storage elements (register/memory)
- All these performed within a clock period



Don't want to read a storage element when it is being written

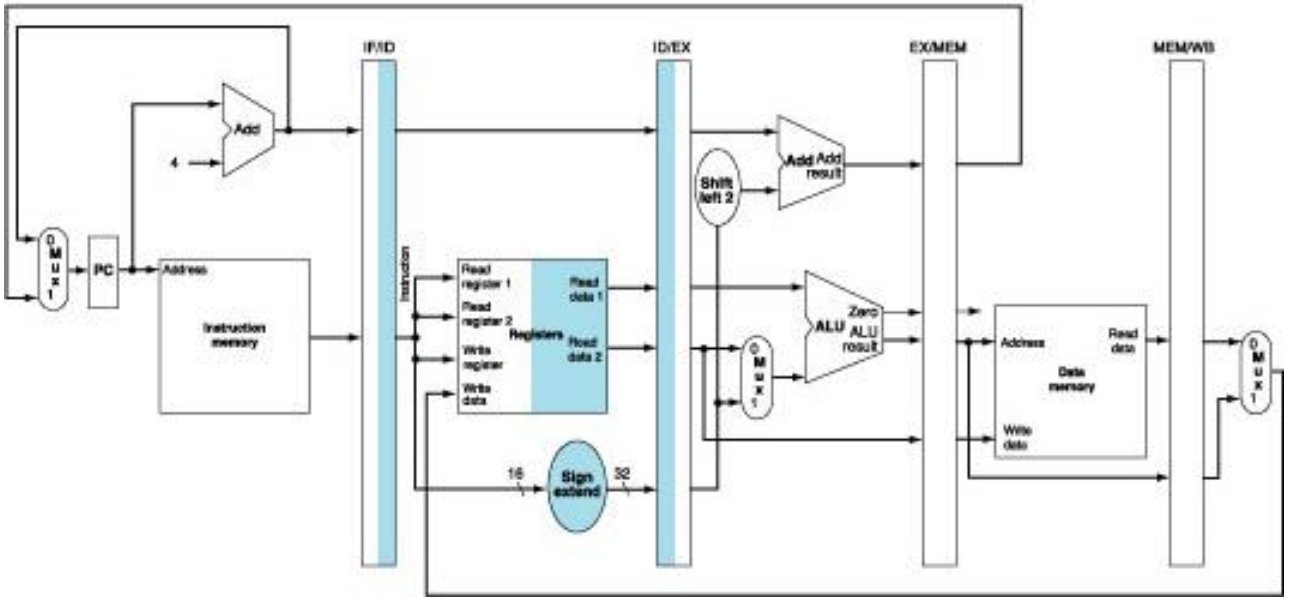
Pipelined Execution: IF Stage

- Read instruction from memory using the address in PC and put it in **IF/ID** register
- PC address is incremented by 4 and then written back to the PC for next instruction
- Incremented PC is also saved in IF/ID register – why?



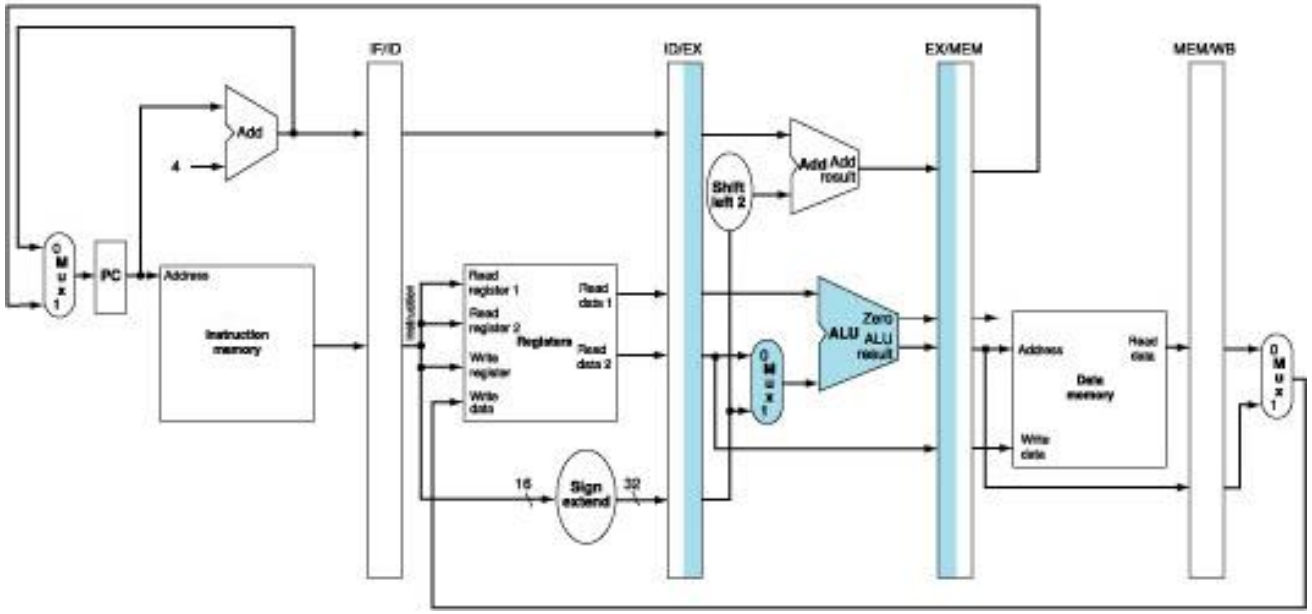
Pipelined Execution: ID Stage

- **IF/ID** register supplies the register numbers for reading two registers
- **IF/ID** register also supplies 16-bit offset to be sign-extended to 32-bit
- Data values read from register file, 32-bit offset and incremented PC stored in **ID/EX**



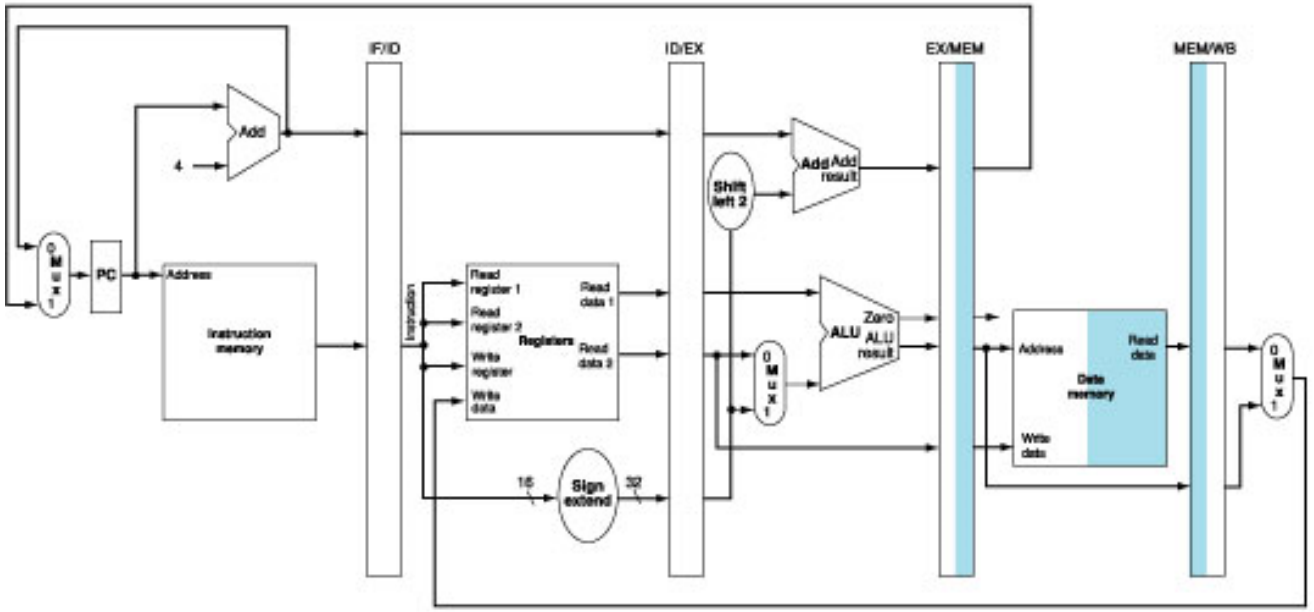
Pipelined Execution: EX Stage

- Example: **lw** instruction
- Reads content of register 1 and sign-extended offset from **ID/EX** register and adds them using ALU
- Sum is placed in **EX/MEM** register



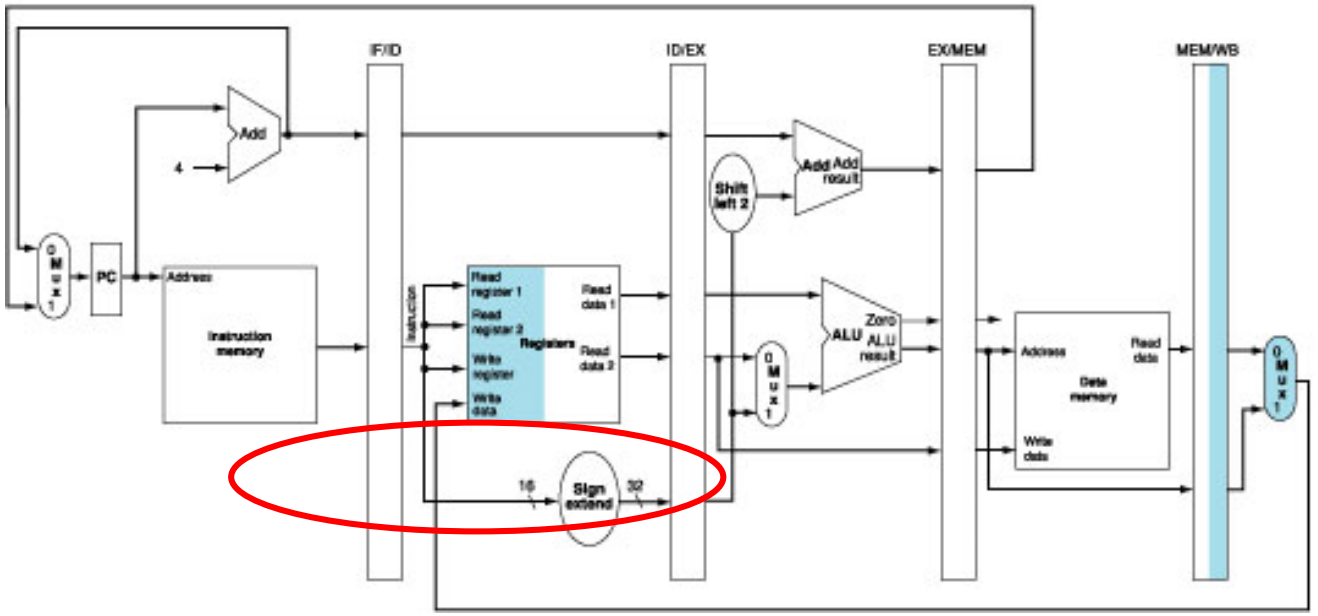
Pipelined Execution: MEM Stage

- Example: **lw** instruction
- Read data memory using address from **EX/MEM** register
- Put the data into the **MEM/WB** register



Pipelined Execution: WB Stage

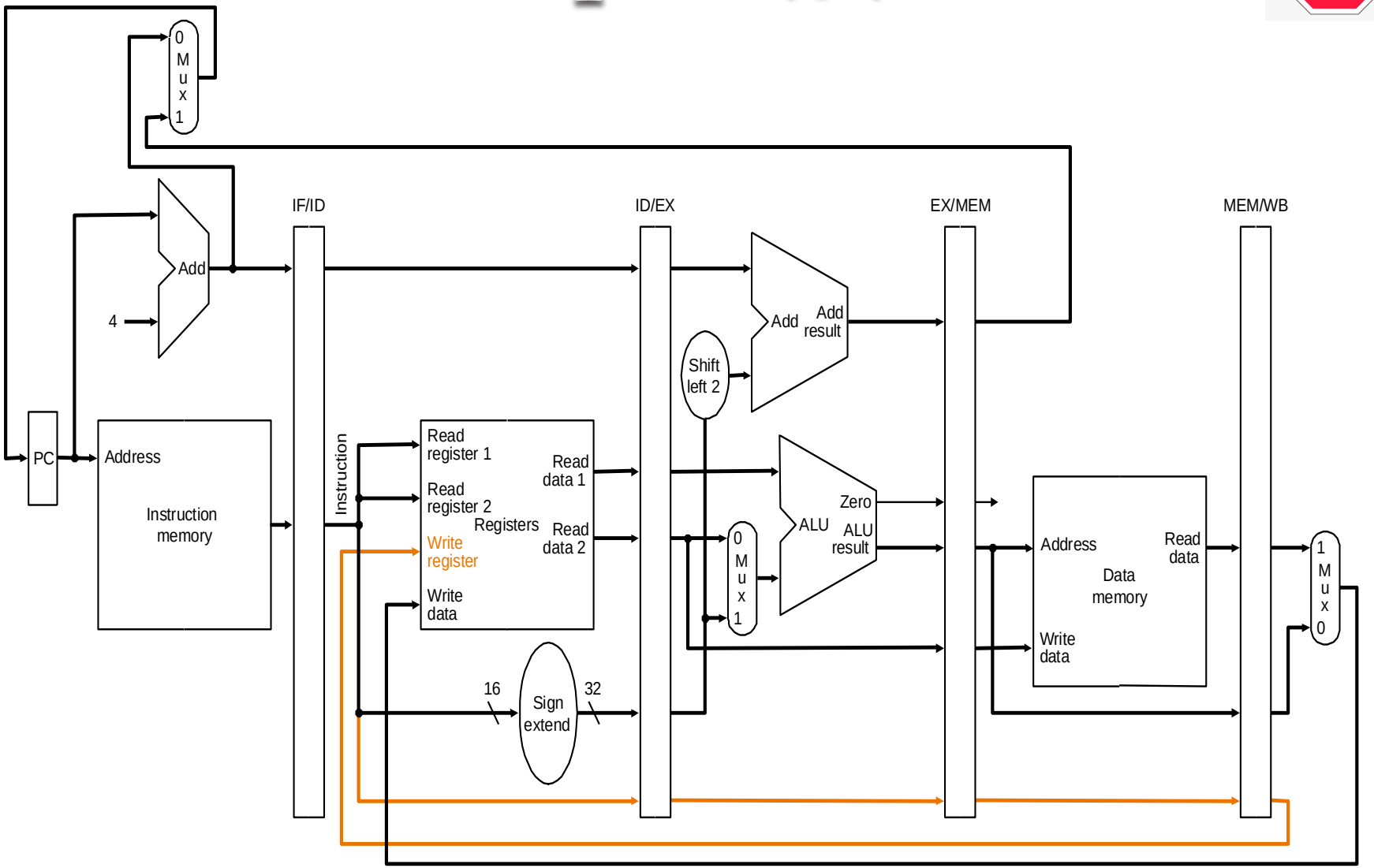
- Example: **lw** instruction
- Read data from **MEM/WB** pipeline register
- Write the data into the register file
- There is a bug in this datapath – can you detect it?



Corrected Datapath (1/2)

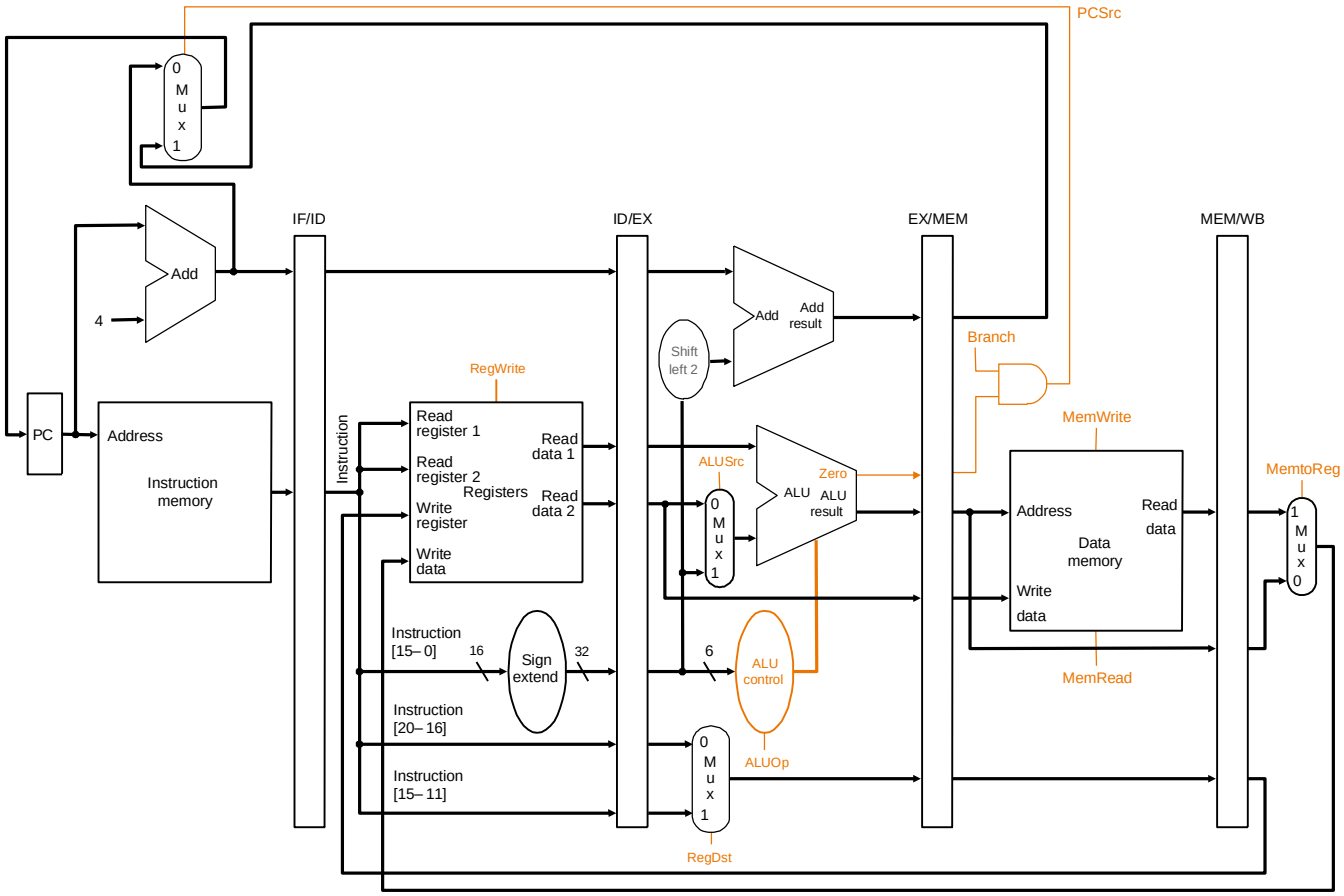
- How do we get "Write register" number?
- Instruction **X** in **IF/ID** supplies "Write register" number
- **X** is not the load instruction; **X** occurs after the load instruction
- Pass "Write register" number from **ID/EX** through **EX/MEM** to **MEM/WB** pipeline register for use in **WB** stage

Corrected Datapath (2/2)



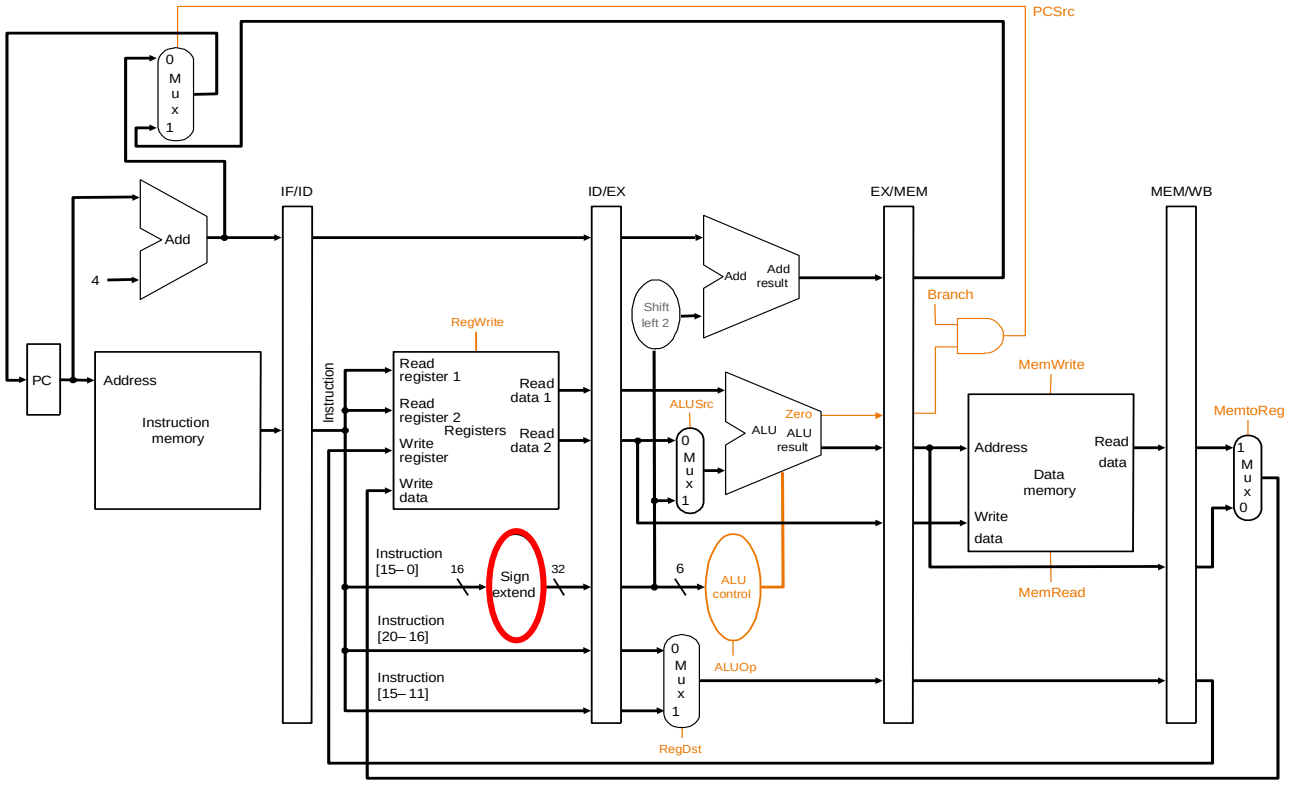
Pipeline Control (1/5)

- Same control signals as single-cycle datapath
- Difference: Each control signal belongs to a particular pipeline stage



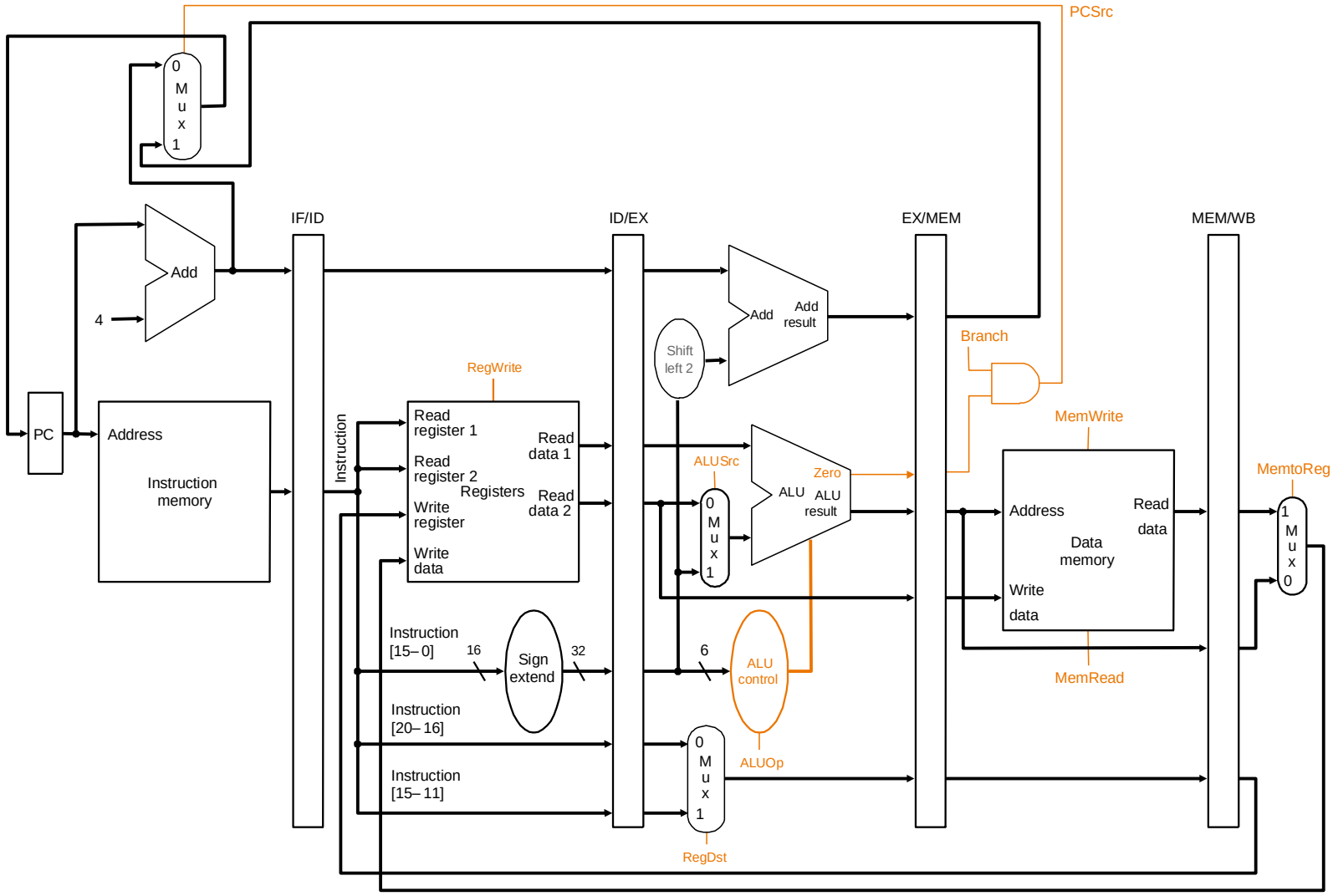
Pipeline Control (2/5)

- 6-bit funct field of the instruction in the EX stage is used as input to ALU control unit
- These bits are included in **ID/EX** register
- These bits are also the least significant bits of the 32-bit offset field already present in **ID/EX** register



Pipeline Control (3/5)

- Try to associate each control signal with a pipeline stage



Pipeline Control (4/5)

- Group control signals according to pipeline stage

	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop 1	ALUop 0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Pipeline Control (4/5)

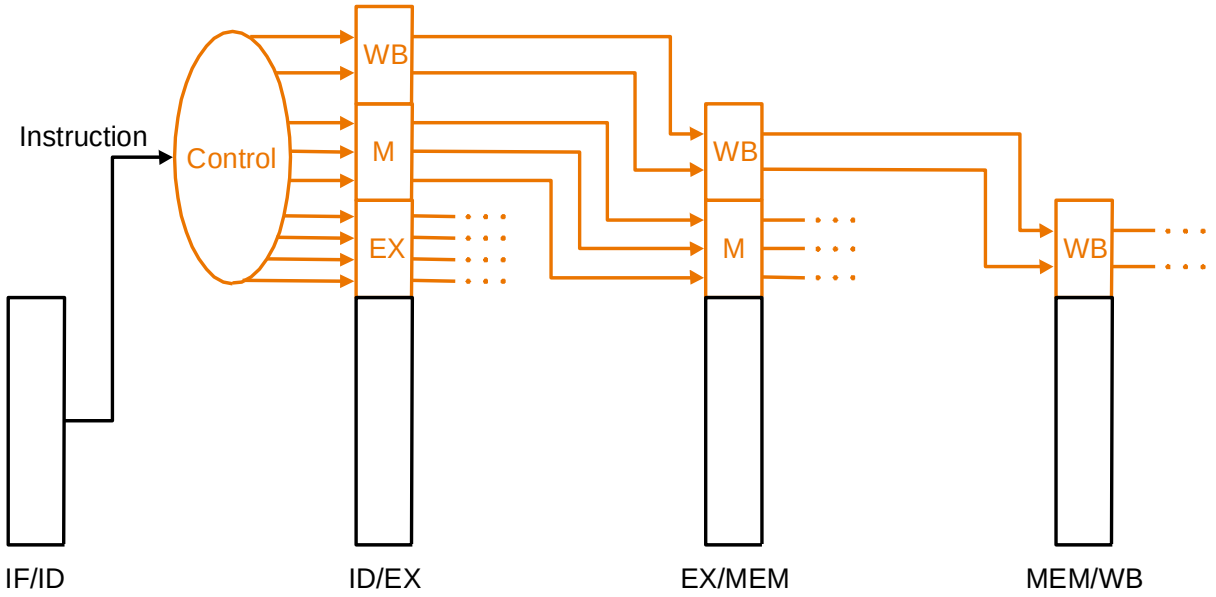
- Group control signals according to pipeline stage

	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop 1	ALUop 0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

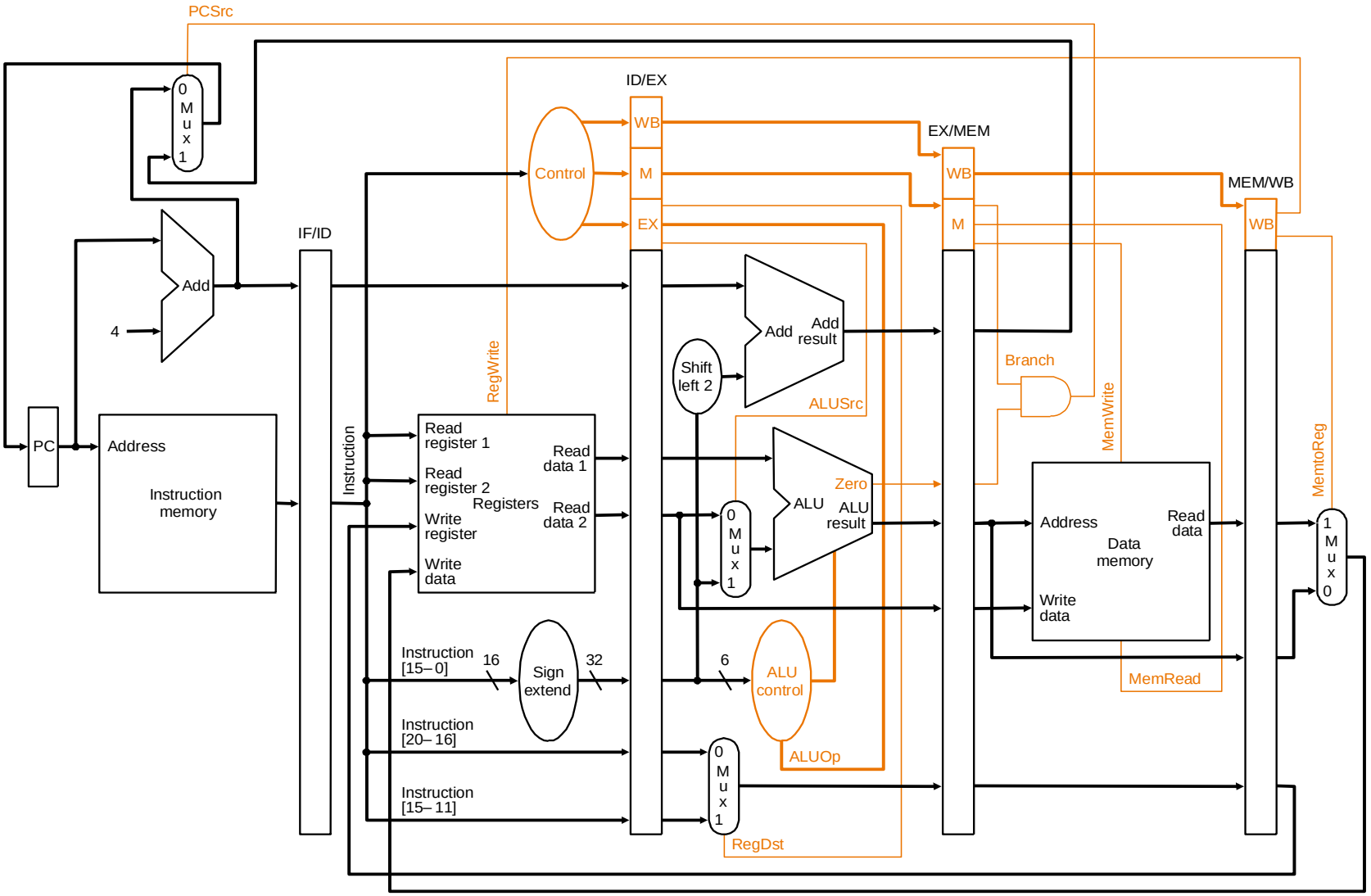
	EX stage				**MEM stage**			**WB stage**	
	RegDst	ALUSrc	ALU op1	ALU op0	Mem Read	Mem Write	Branch	Reg write	Memto Reg
R-type	1	0	1	0	0	0	0	1	0
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	0	X
beq	X	0	0	1	0	0	1	0	X

Pipeline Control (5/5)

- Control lines are needed starting from **EX**
- Create control signals during **ID** and pass them on using pipeline registers
- Control signals are used during appropriate pipeline stage as instruction moves down pipeline



Pipelined Datapath with Control



Try It Yourself #1

- Write down the control signals for the following sequence of instructions for 8 clock cycles: **add**, **sub**, **lw**, **sw**

Cycle	RegDst	ALUSrc	ALUOp1	ALUOp0	MemRead	MemWrite	RegWrite	MemtoReg
1								
2								
3	1	0	1	0				
4	1	0	1	0	0	0		
5								
6								
7								
8								

Representing Pipelines Graphically

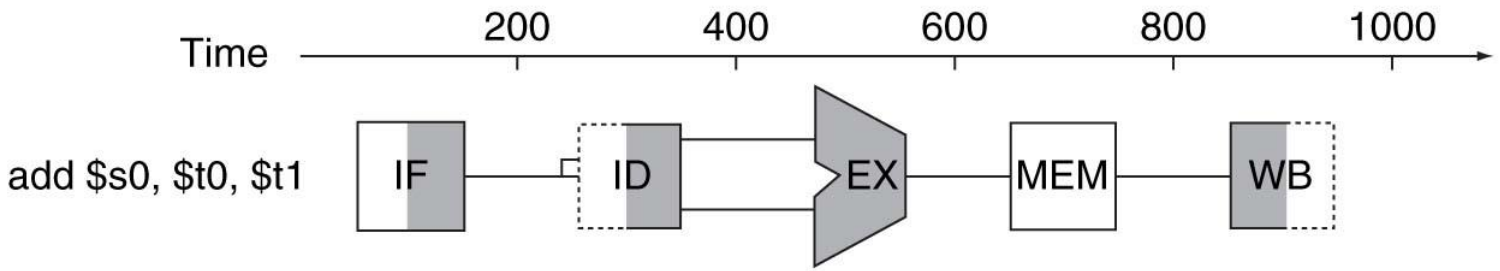


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction fetch stage, with the box representing instruction memory; *ID* for the instruction decode/register file read stage, with the drawing showing the register file being read; *EX* for the execution stage, with the drawing representing the ALU; *MEM* for the memory access stage, with the box representing data memory; and *WB* for the write-back stage, with the drawing showing the register fi le being written. The shading indicates the element is used by the instruction. Hence, MEM has a white back ground because add does not access the data memory. Shading on the right half of the register fi le or memory means the element is read in that stage, and shading of the left half means it is written in that stage. Hence the right half of ID is shaded in the second stage because the register file is read, and the left half of WB is shaded in the fifth stage because the register file is written.

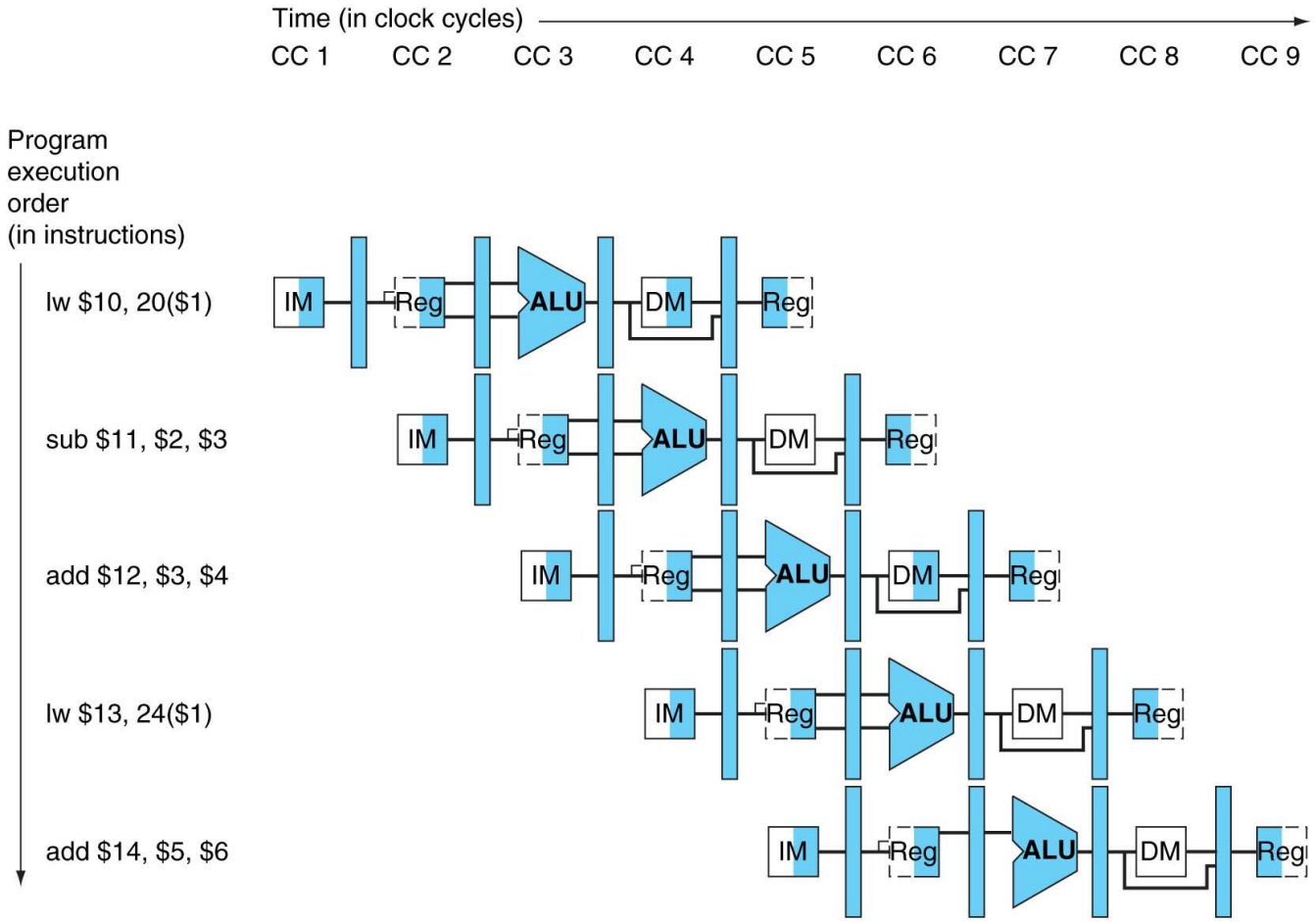


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram.

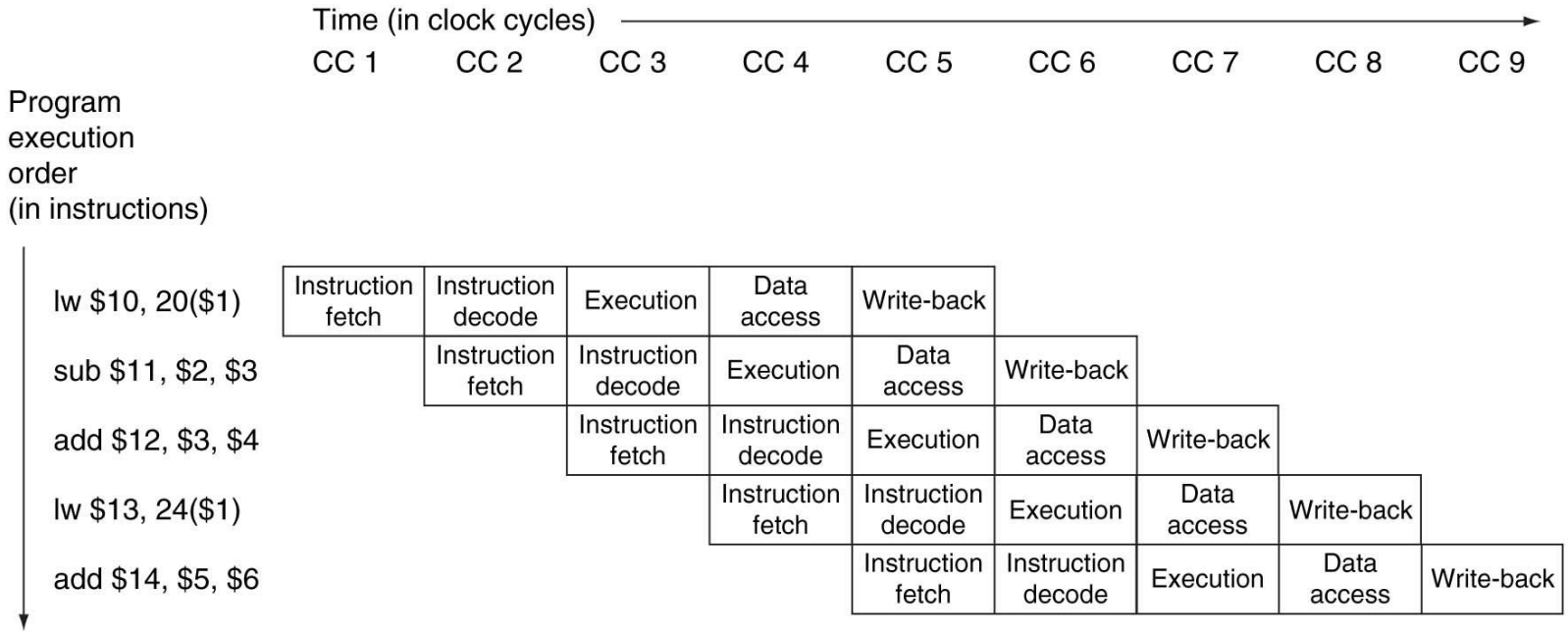


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43. Copyright © 2009 Elsevier, Inc. All rights reserved.

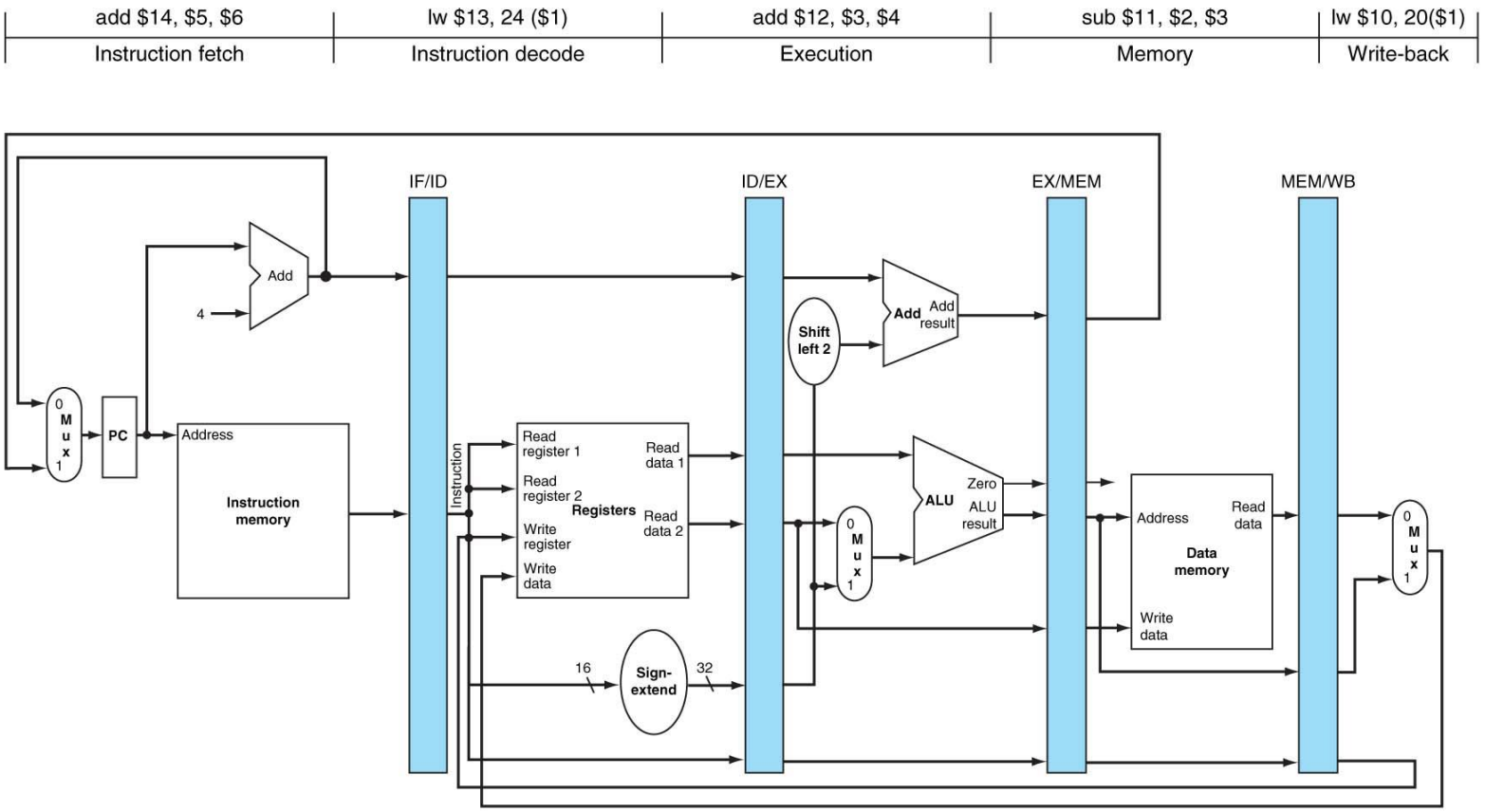
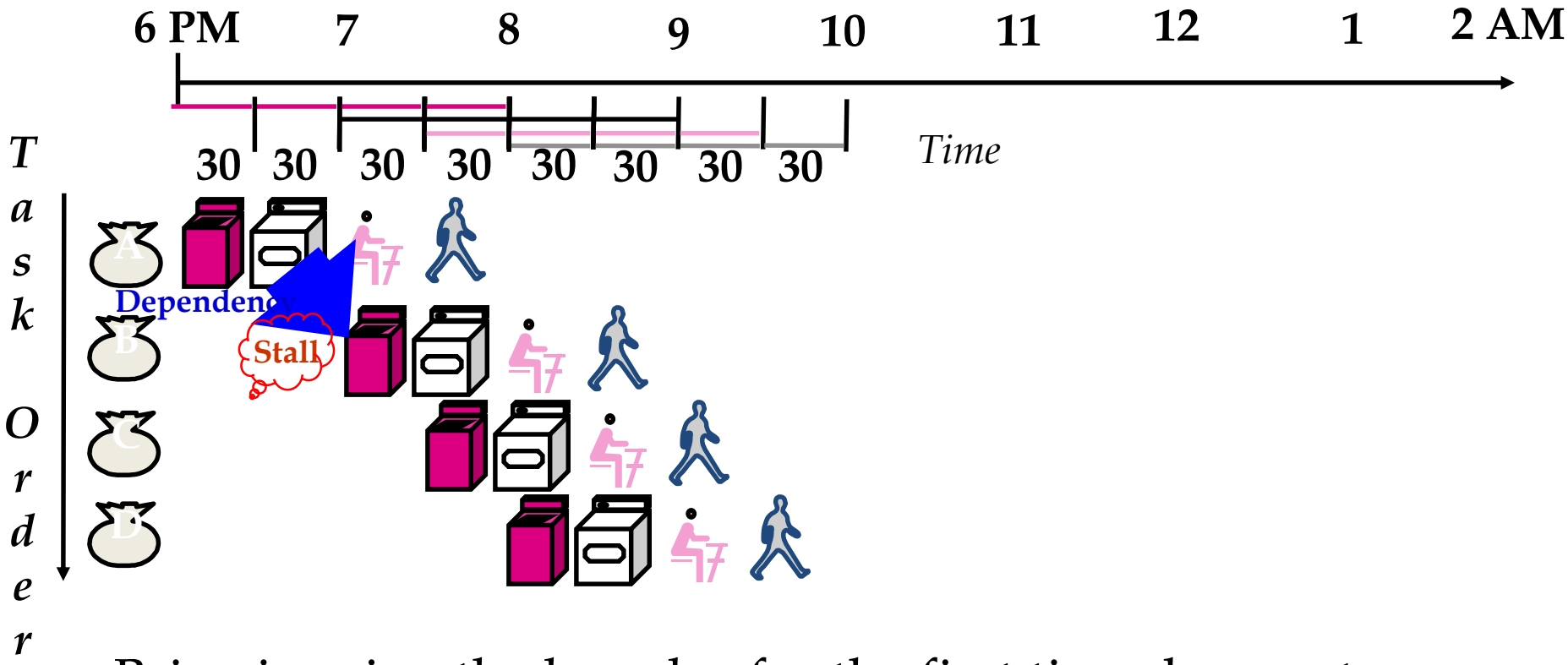


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram. Copyright © 2009 Elsevier, Inc. All rights reserved.

Pipeline Hazards

- Trouble for pipeline
- Hazards can prevent next instruction from immediately following previous instruction
- Structural hazards:
  - Simultaneous use of a hardware resource
- Data hazards:
  - Data dependencies between instructions
- Control hazards:
  - Change in program flow

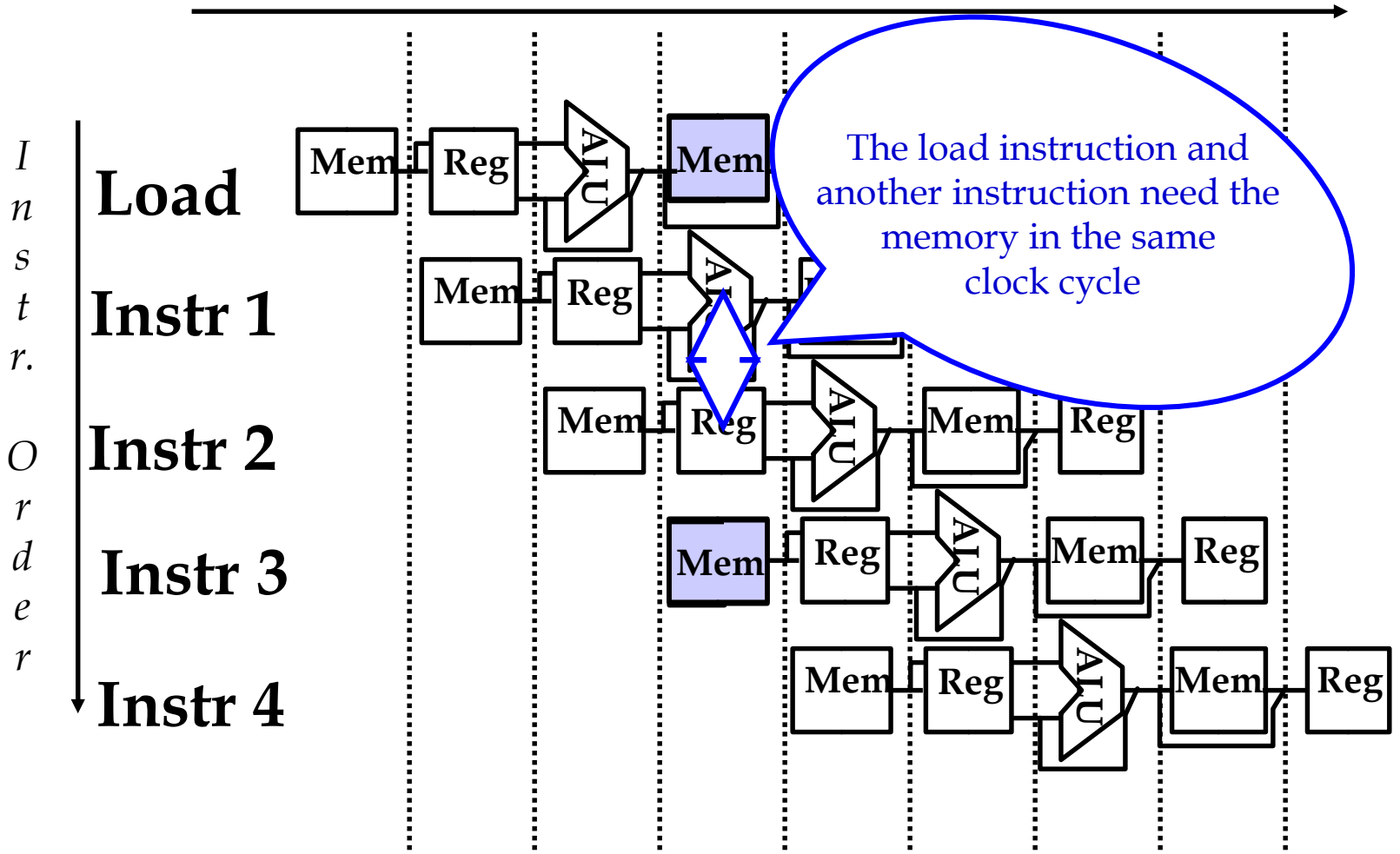
Dependency



- Brian is using the laundry for the first time; he wants to see the outcome of one wash + dry cycle first before putting in his clothes
- Pipelined laundry now takes 4 hours

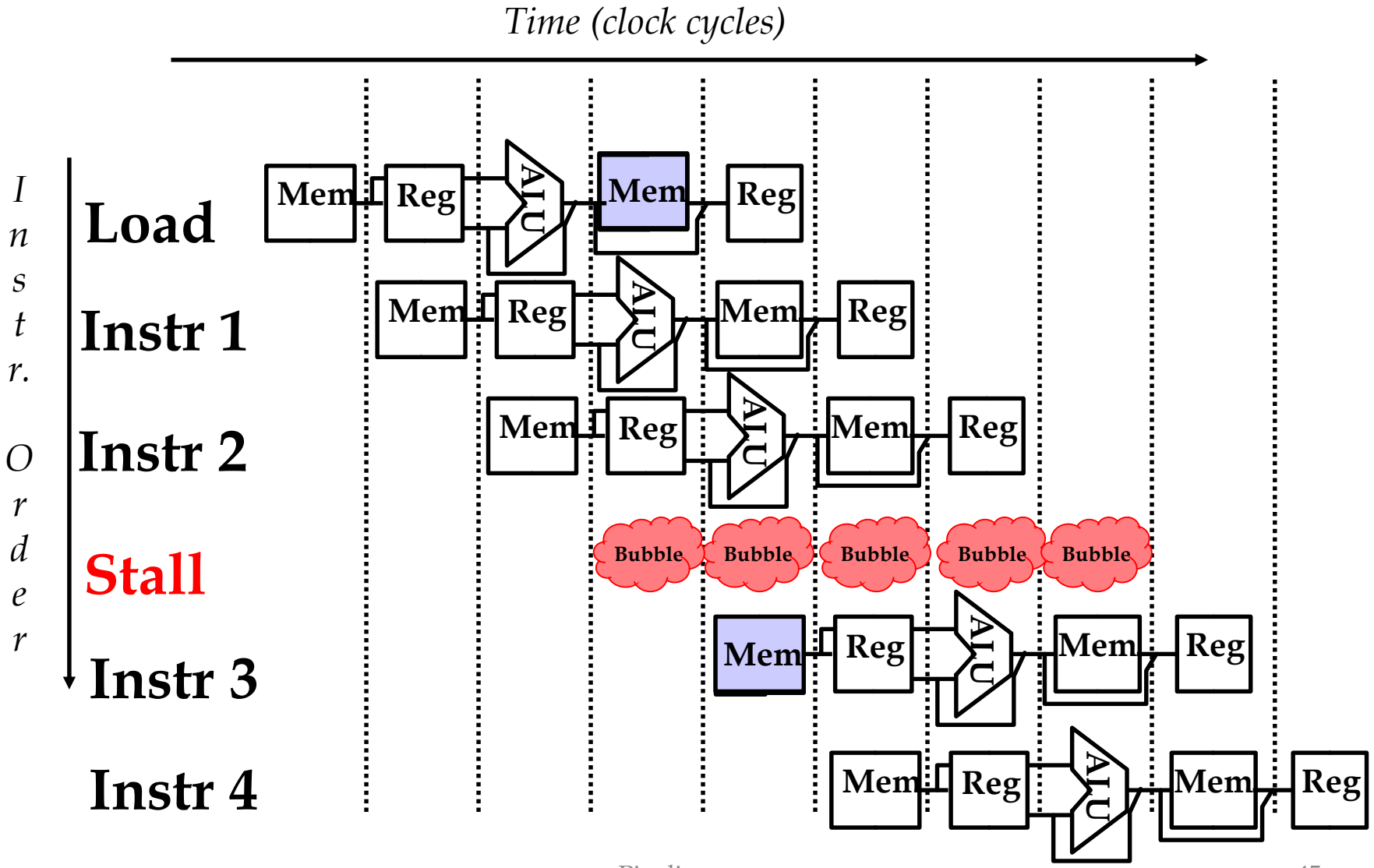
Example of Structural Hazard

- Only one memory

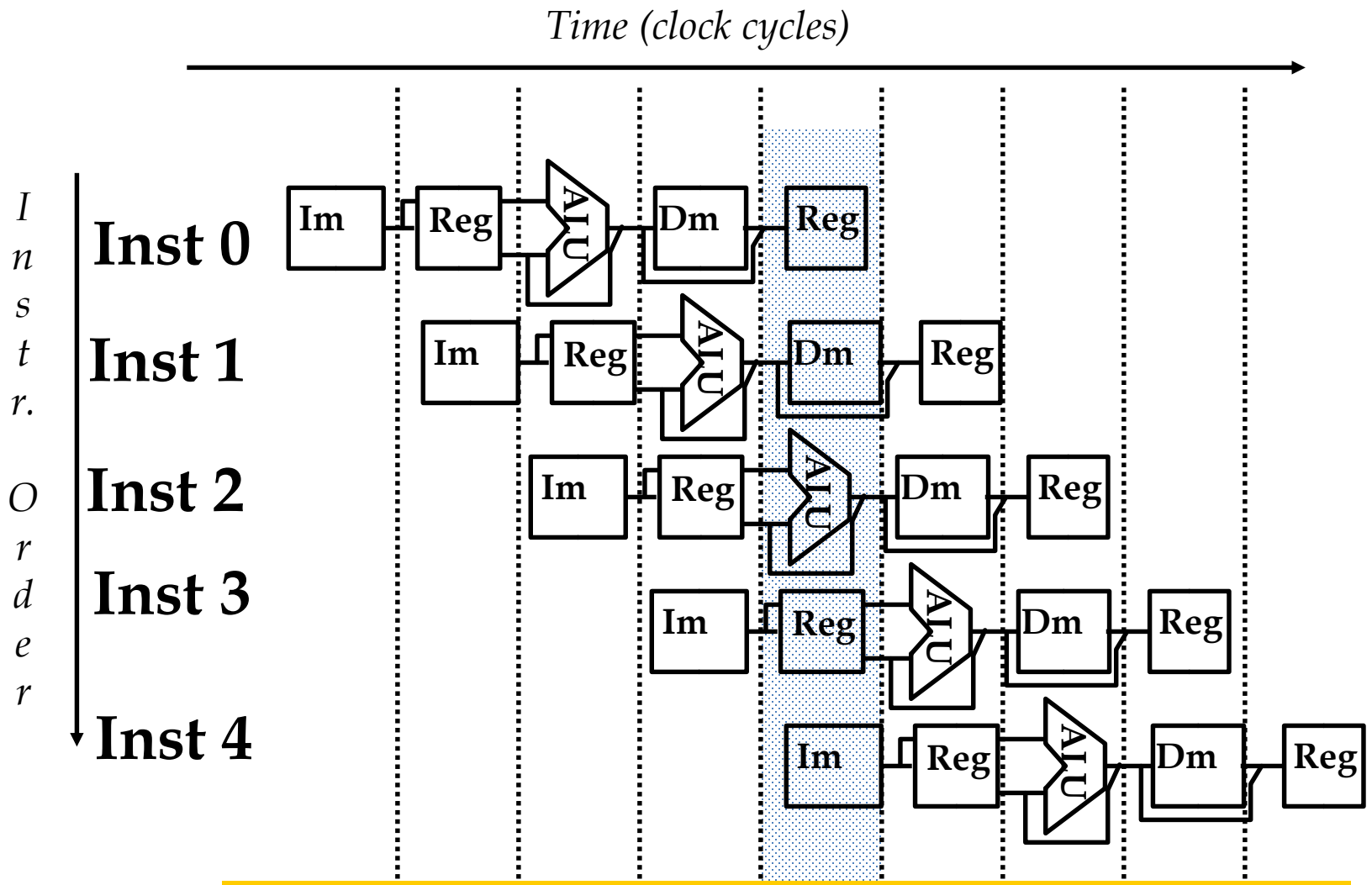


Example of Structural Hazard

- A simple solution: Stall the pipeline

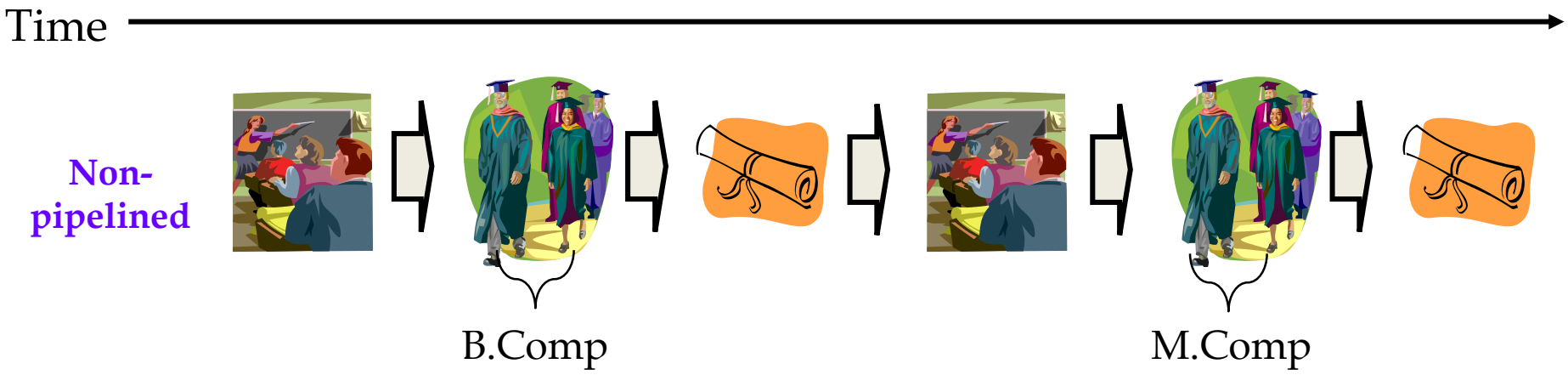


Wait a Minutes!



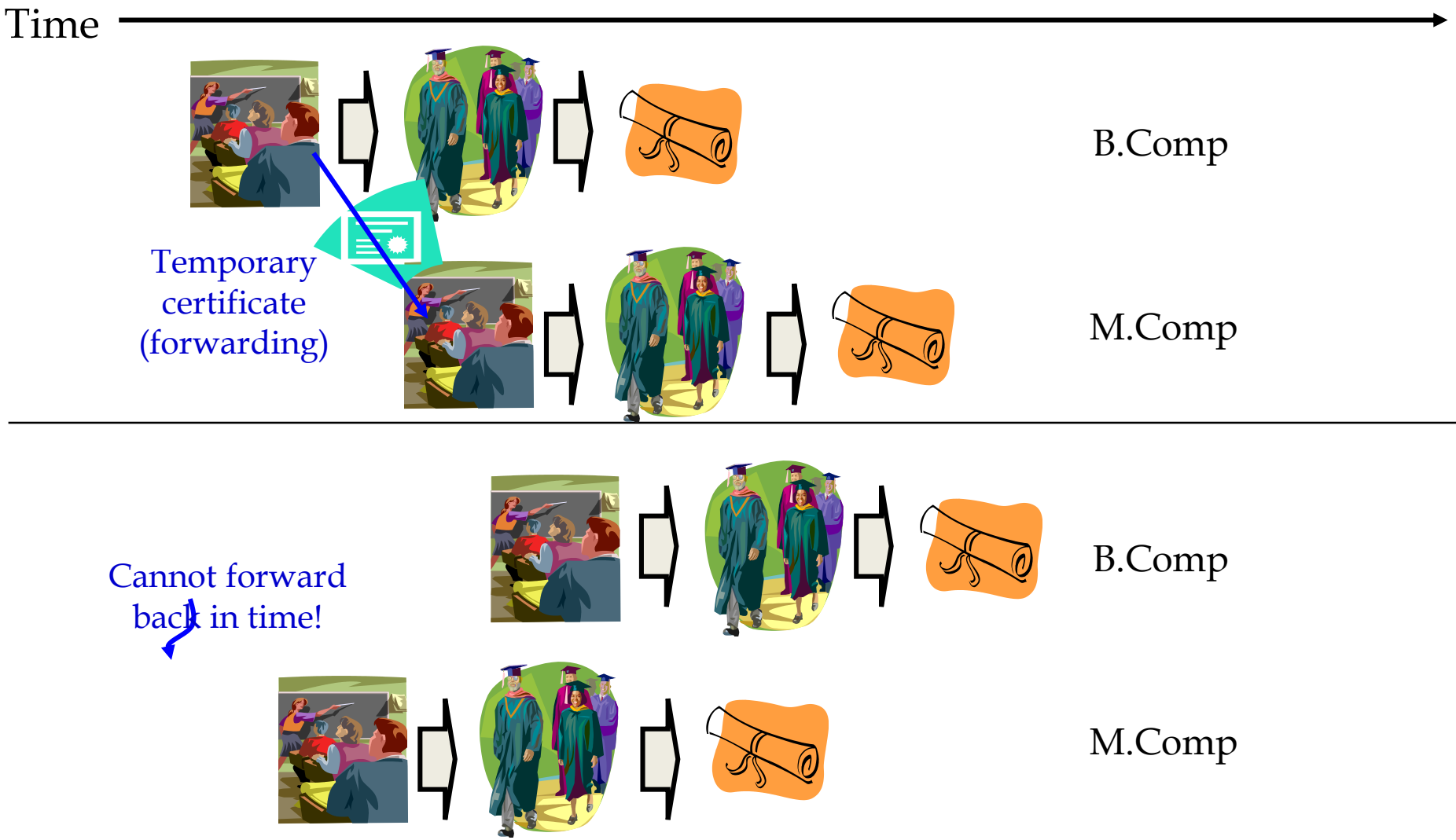
How come two instructions are using register file simultaneously?

Data Hazards: Analogy (1/2)



Dependency: M. Comp program requires B. Comp degree

Data Hazards: Analogy (2/2)



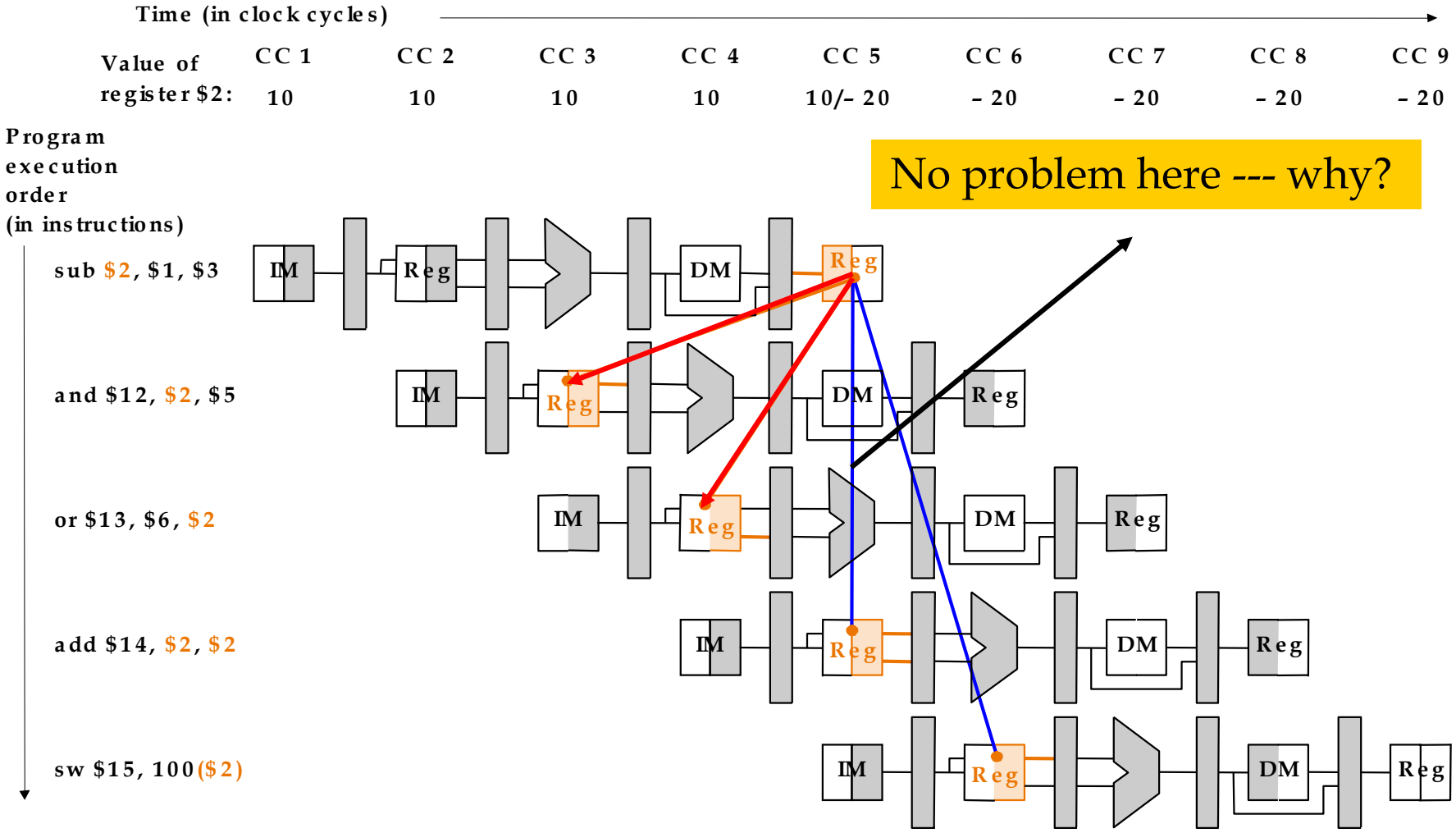
Data Hazards

- Due to data dependencies between instructions
- Instruction sequence

- Last four instructions are all dependent on the result produced by the first instruction (**sub**) in register $2

Data Hazards: Example

- Value from prior instruction is needed before write back



Solution: FORWARDING

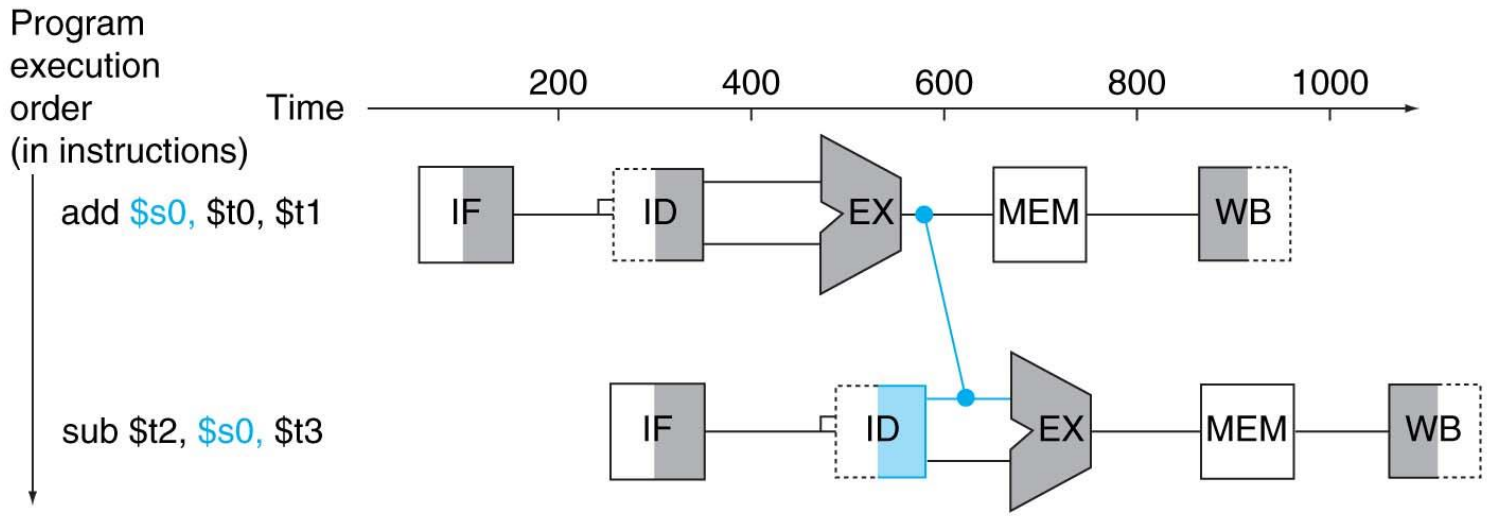
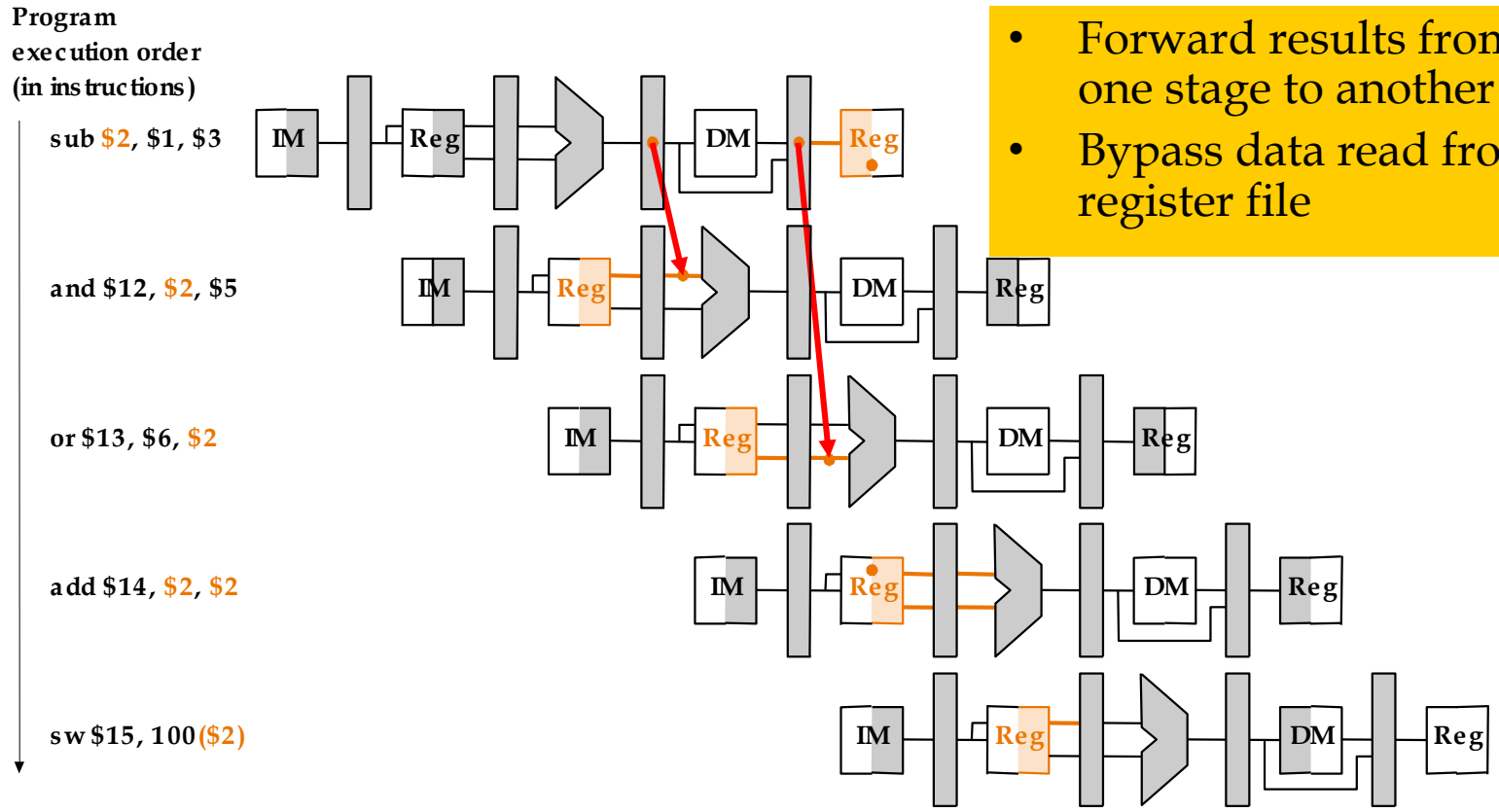


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Solution: FORWARDING

Time (in clock cycles) →

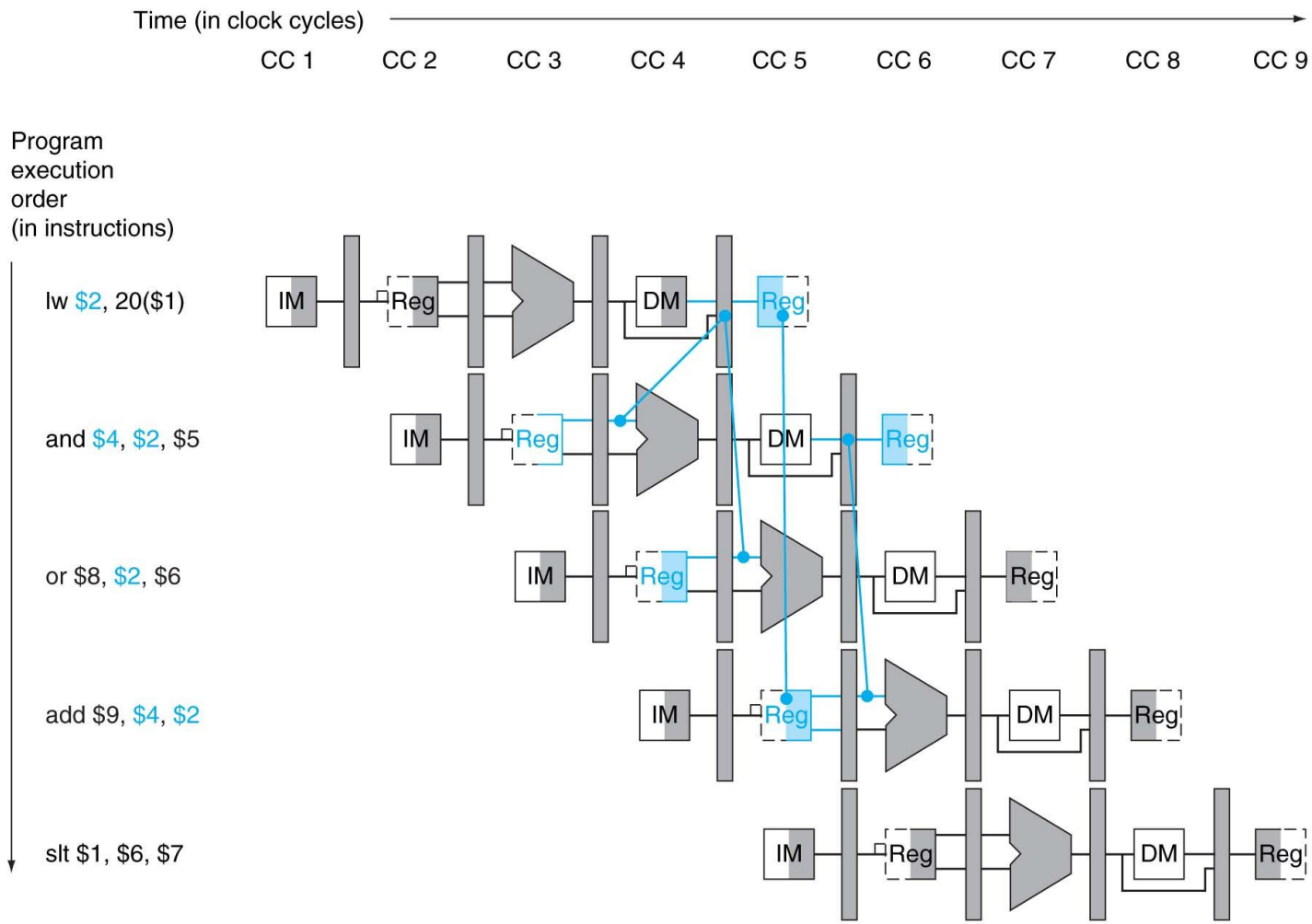
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM:	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X



- Forward results from one stage to another
- Bypass data read from register file

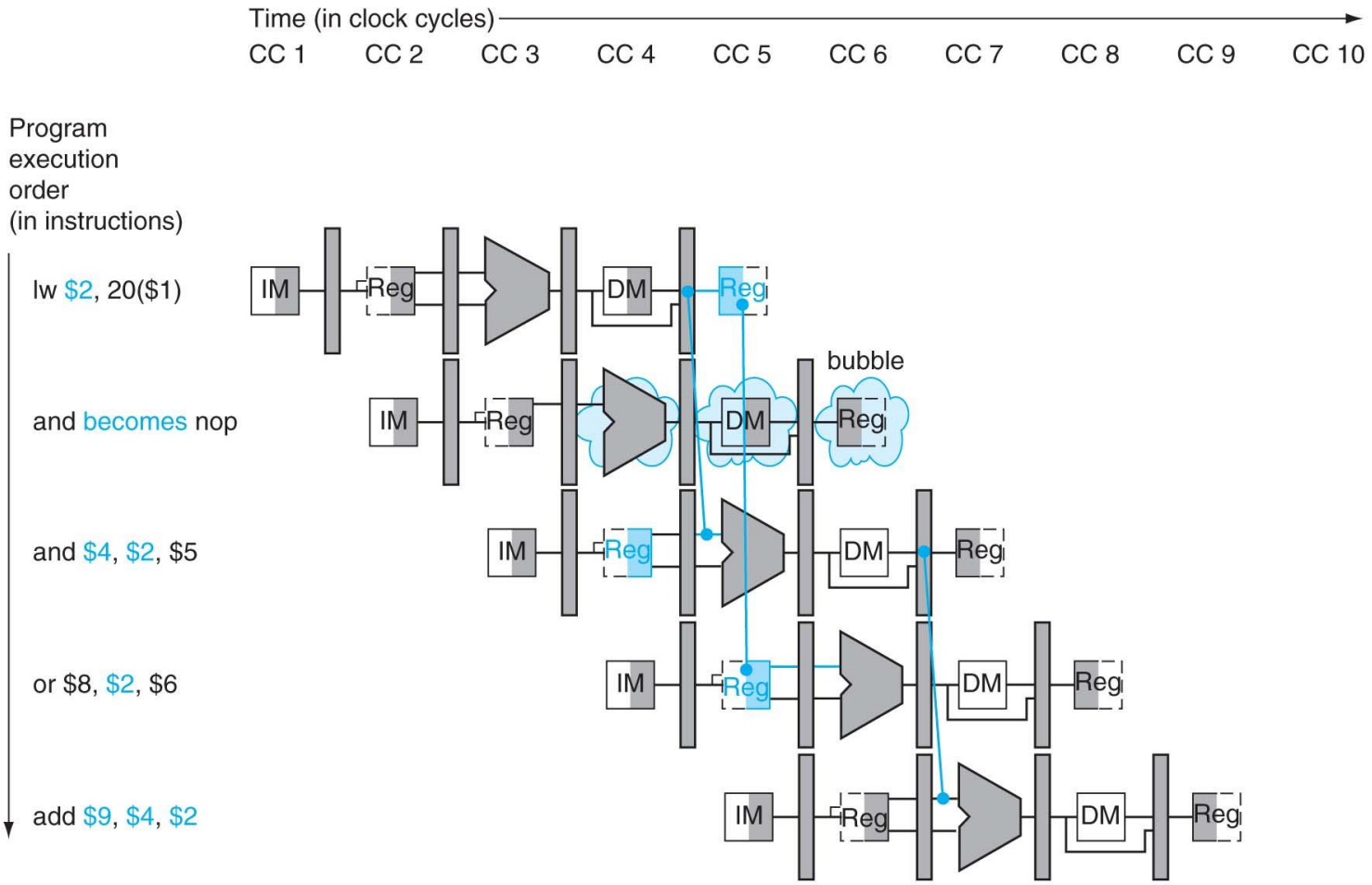
What About LOADS? (1/2)

- Cannot solve with forwarding: need the data even before it is produced



What About LOADS? (2/2)

- Stall the pipeline!



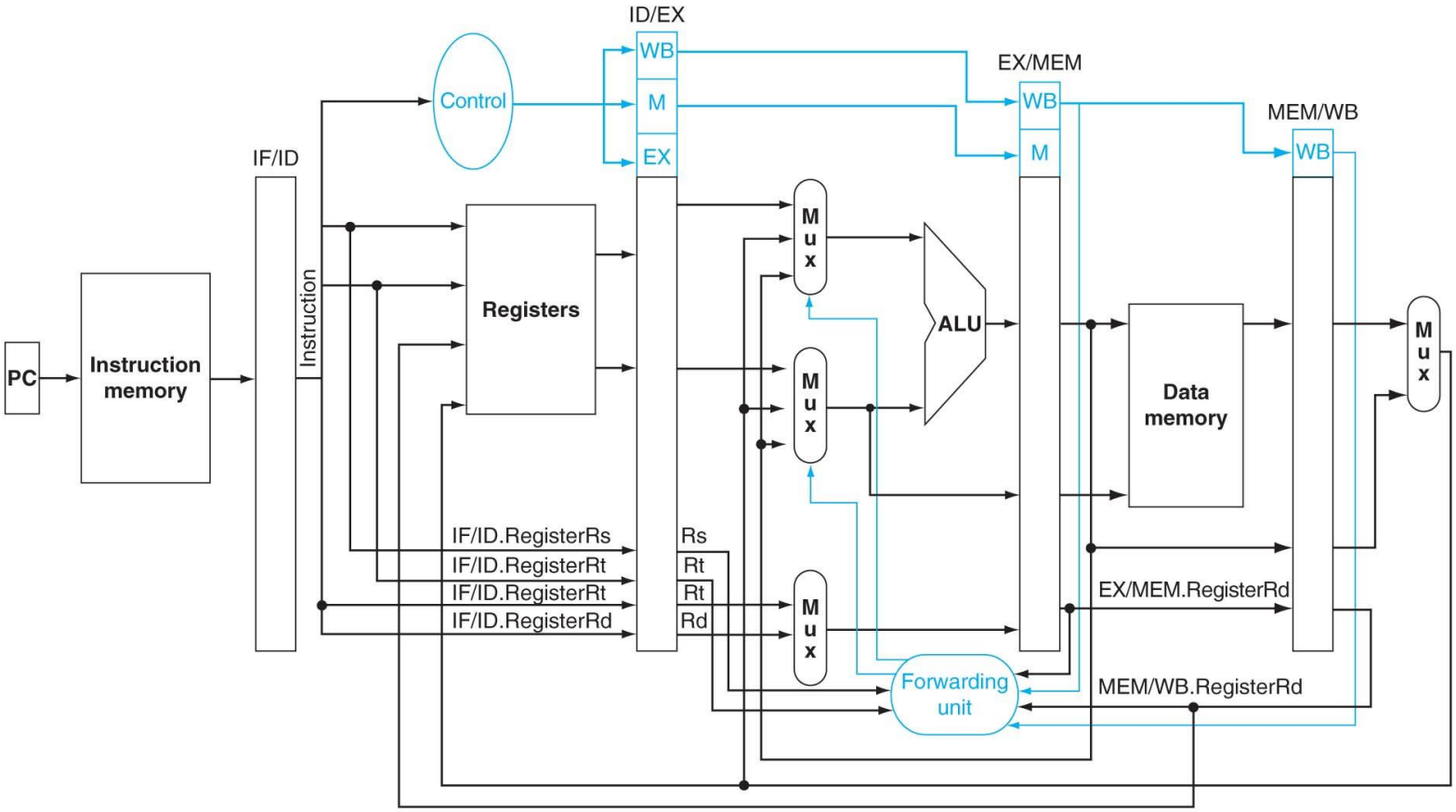


FIGURE 4.56 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.51, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

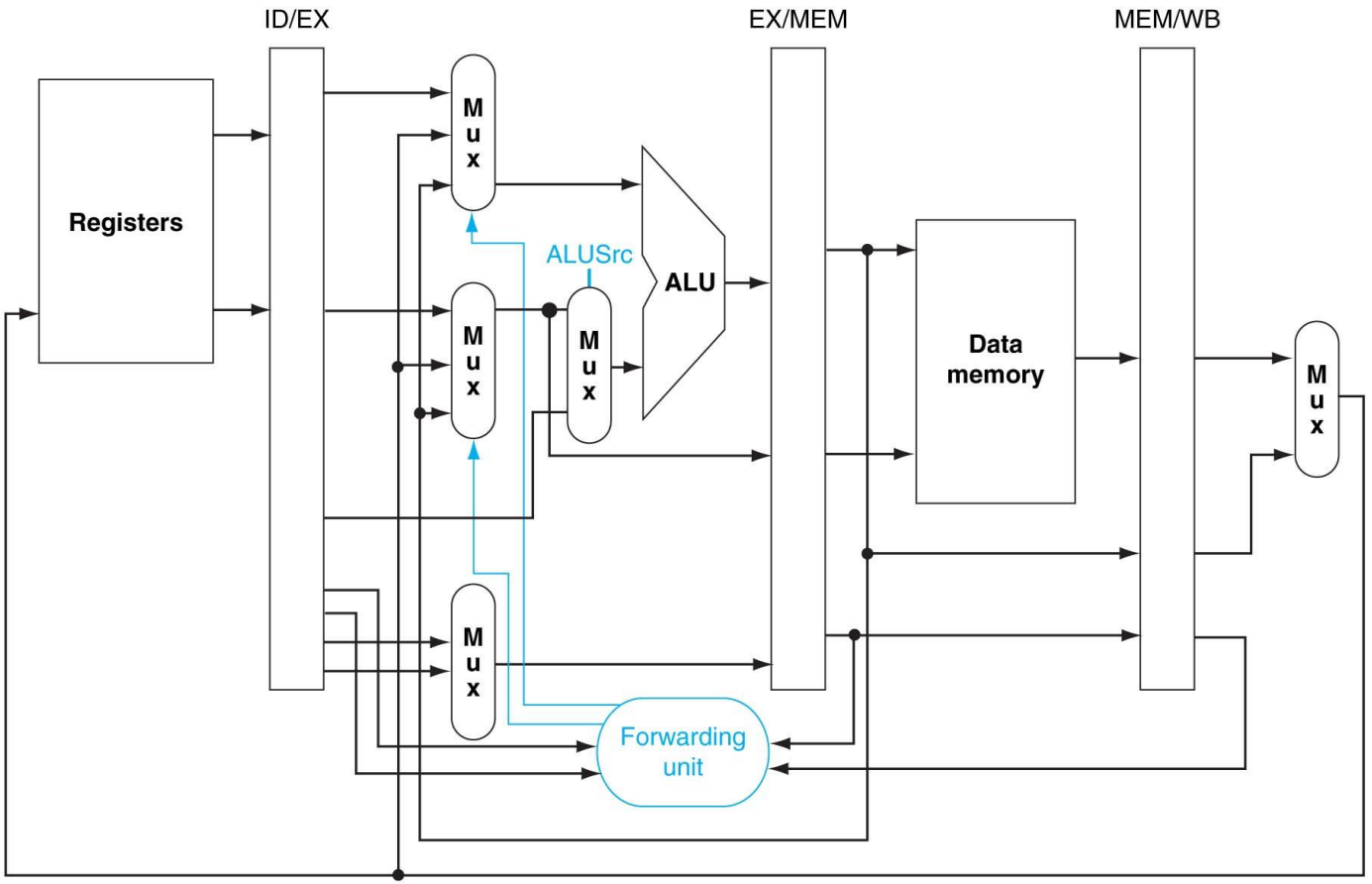
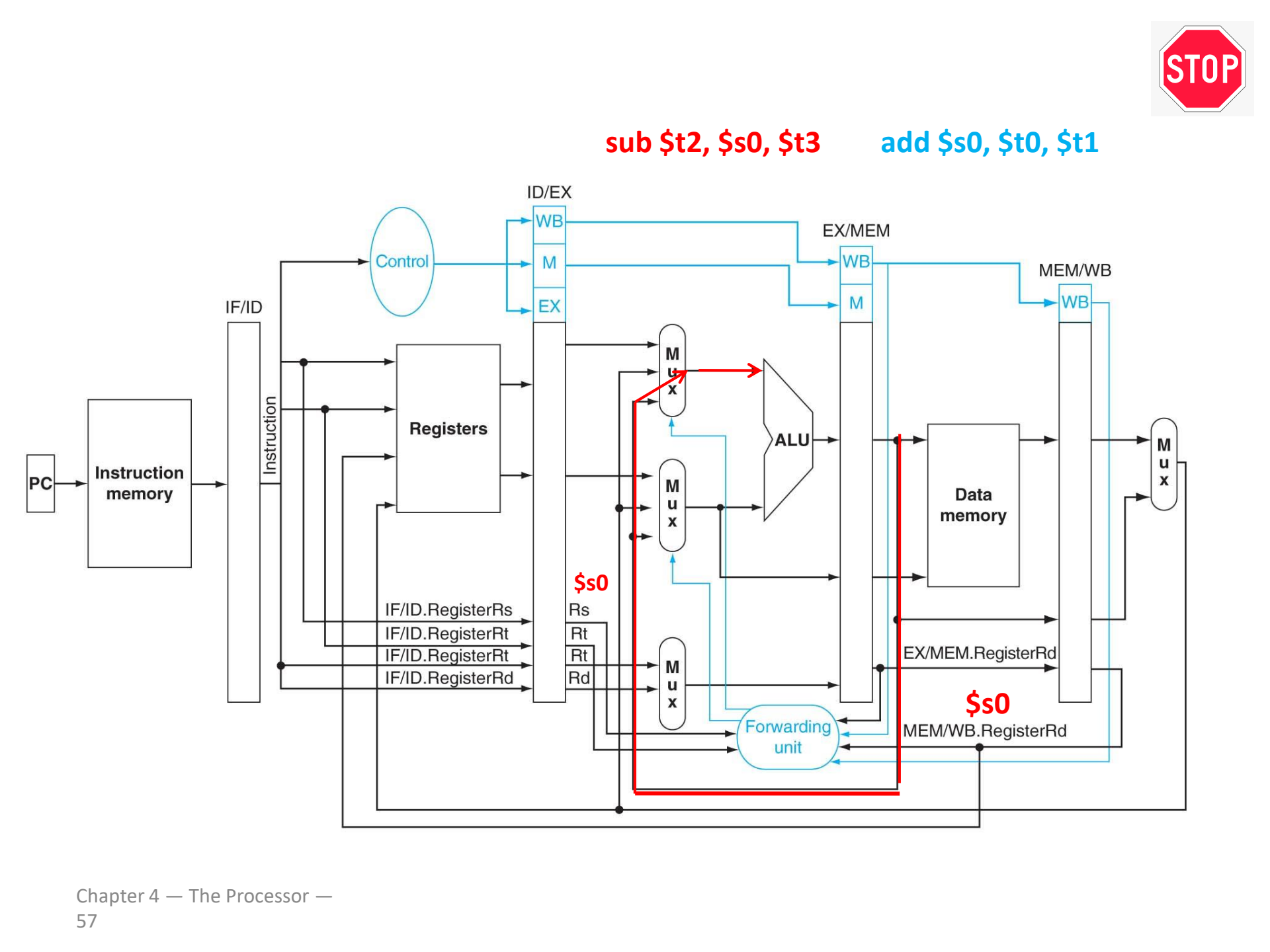
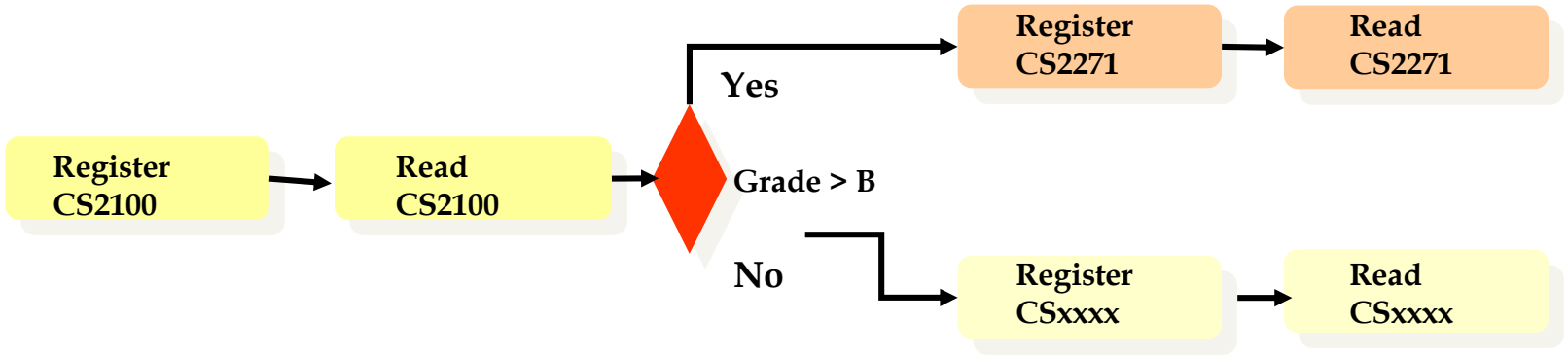


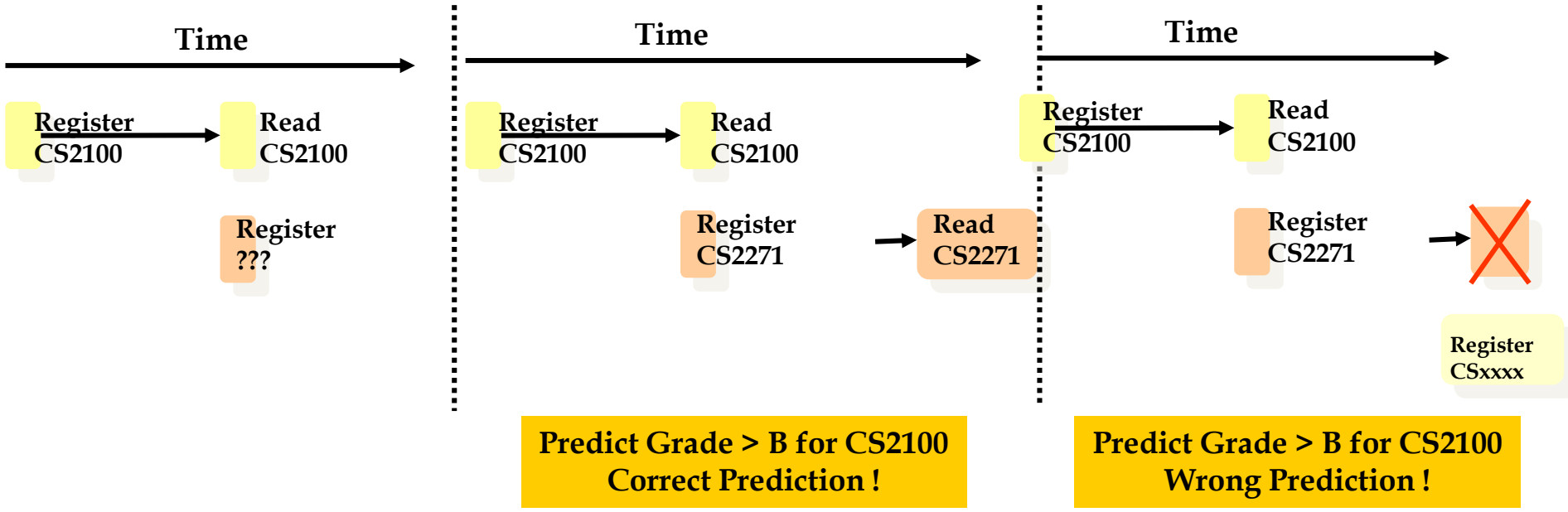
FIGURE 4.57 A close-up of the datapath in Figure 4.54 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.


STOP
sub $t2, $s0, $t3
add $s0, $t0, $t1
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
Instruction
PC
Instruction memory
Registers
ALU
Data memory
Mux
$s0
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
$s0
MEM/WB.RegisterRd
Forwarding unit

Control Hazards: Analogy

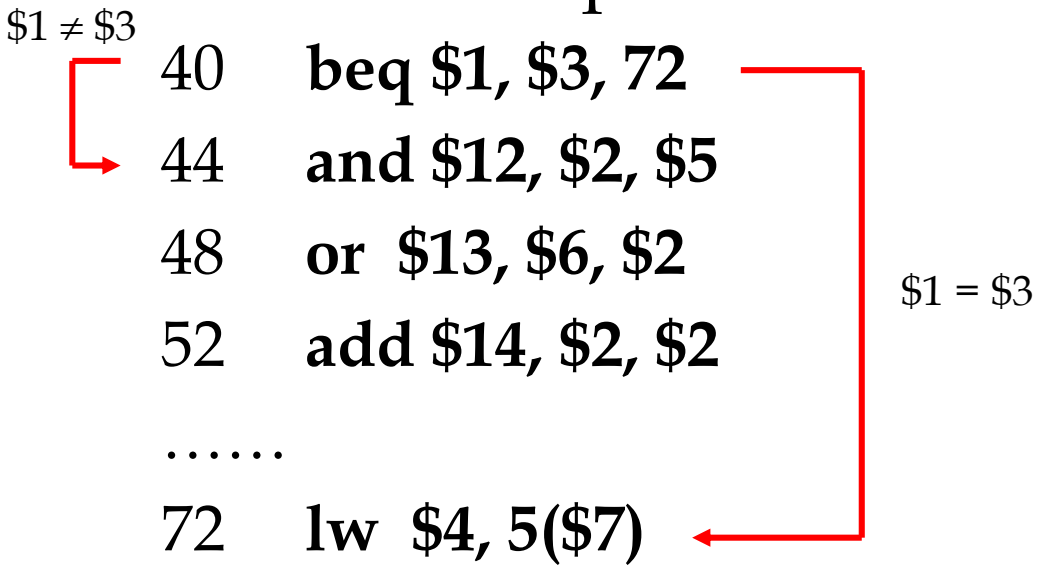
- Conditional execution





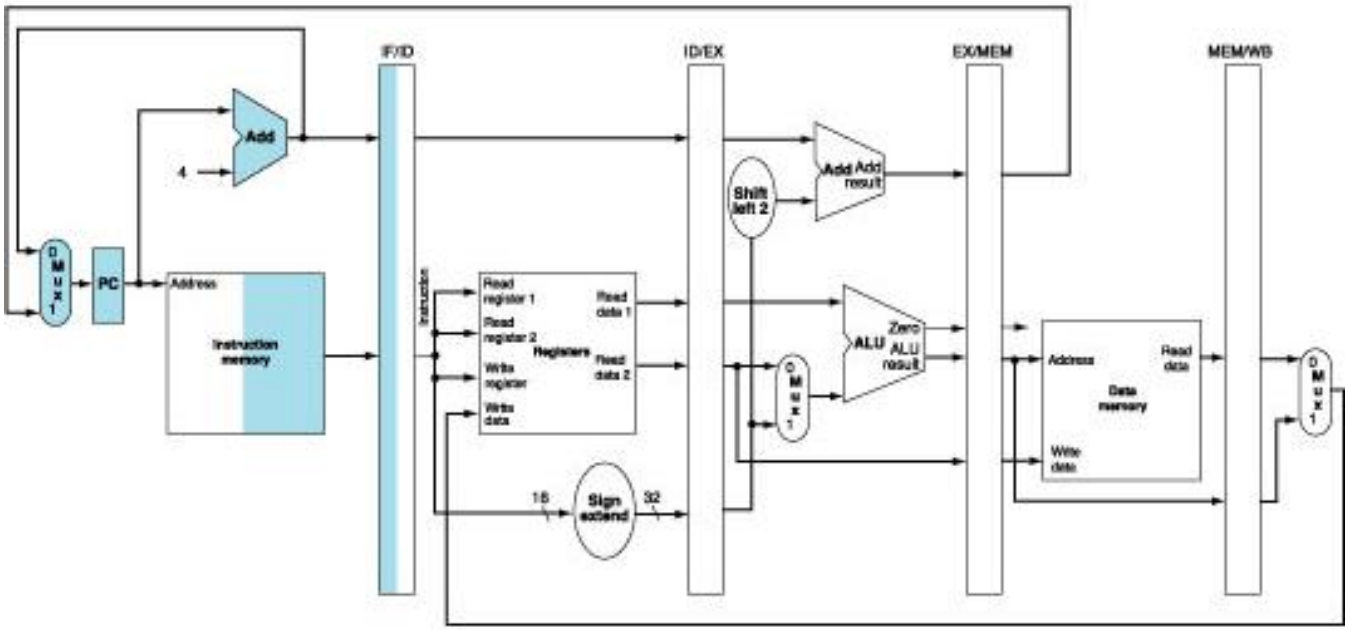
Control Hazards

- Make decision about which instruction to execute next based on result of an instruction
- Instruction sequence

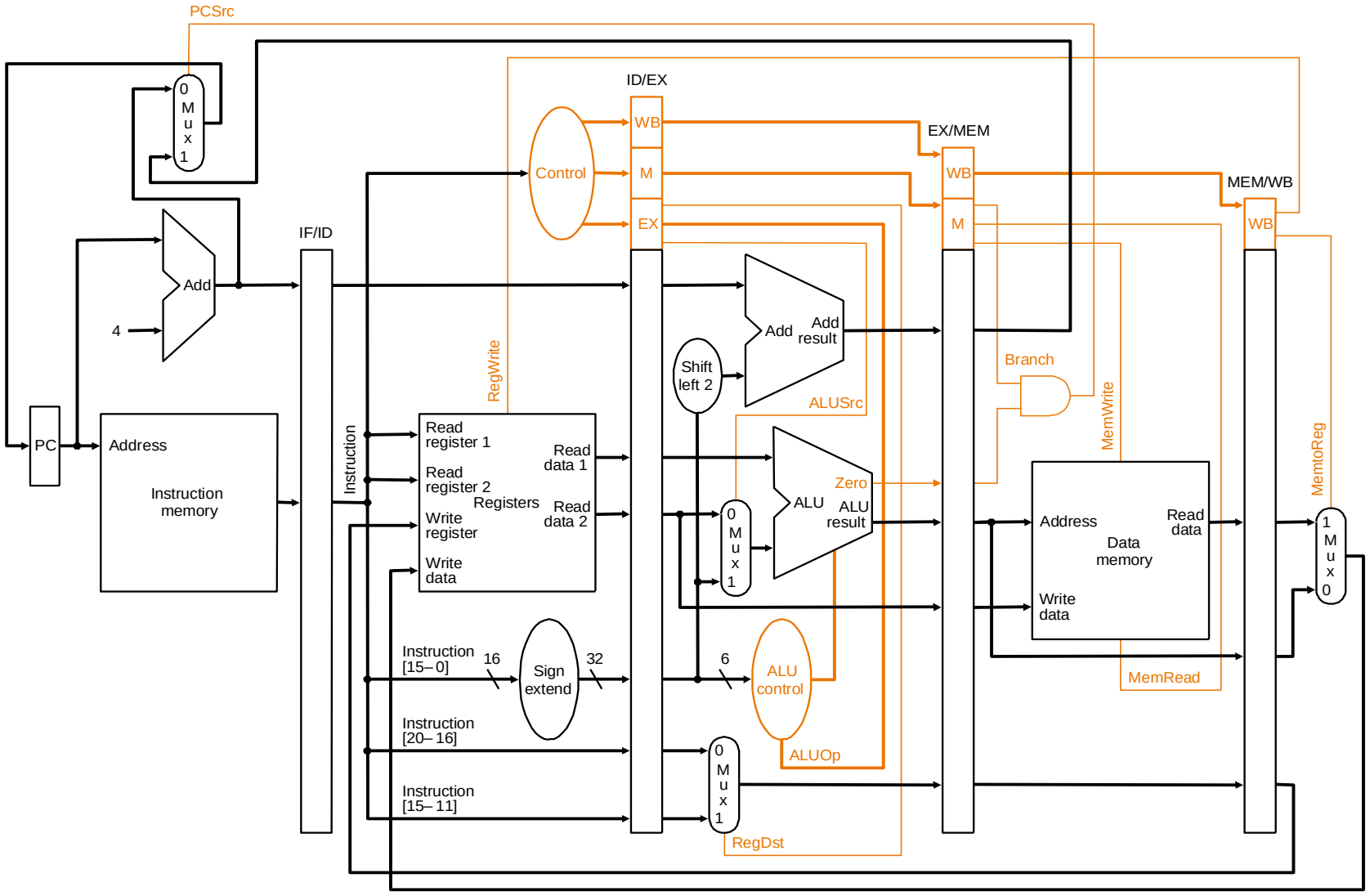


Pipelined Execution: IF Stage

- Read instruction from memory using the address in PC and put it in **IF/ID** register
- PC address is incremented by 4 and then written back to the PC for next instruction

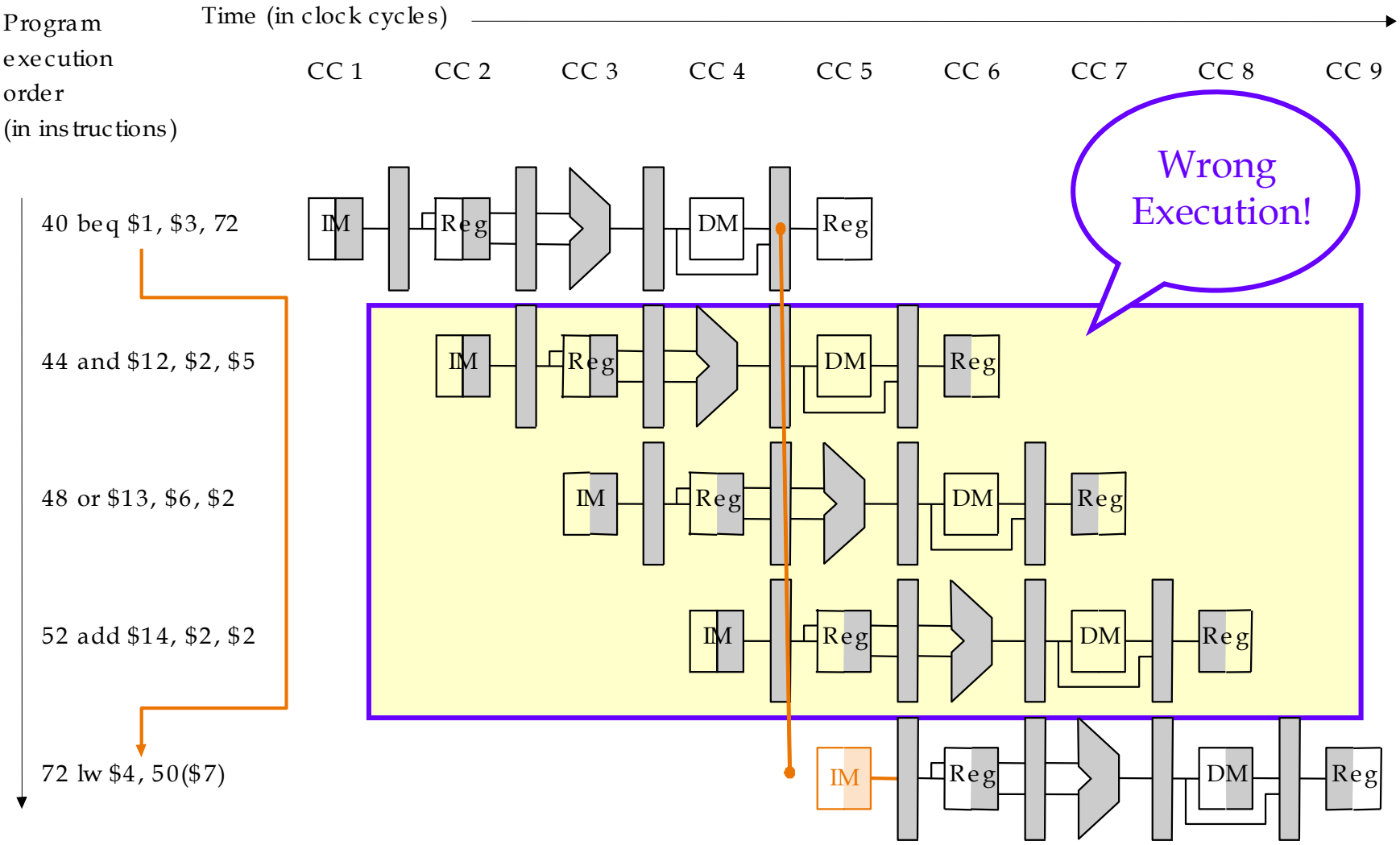


Control Hazards: Why?

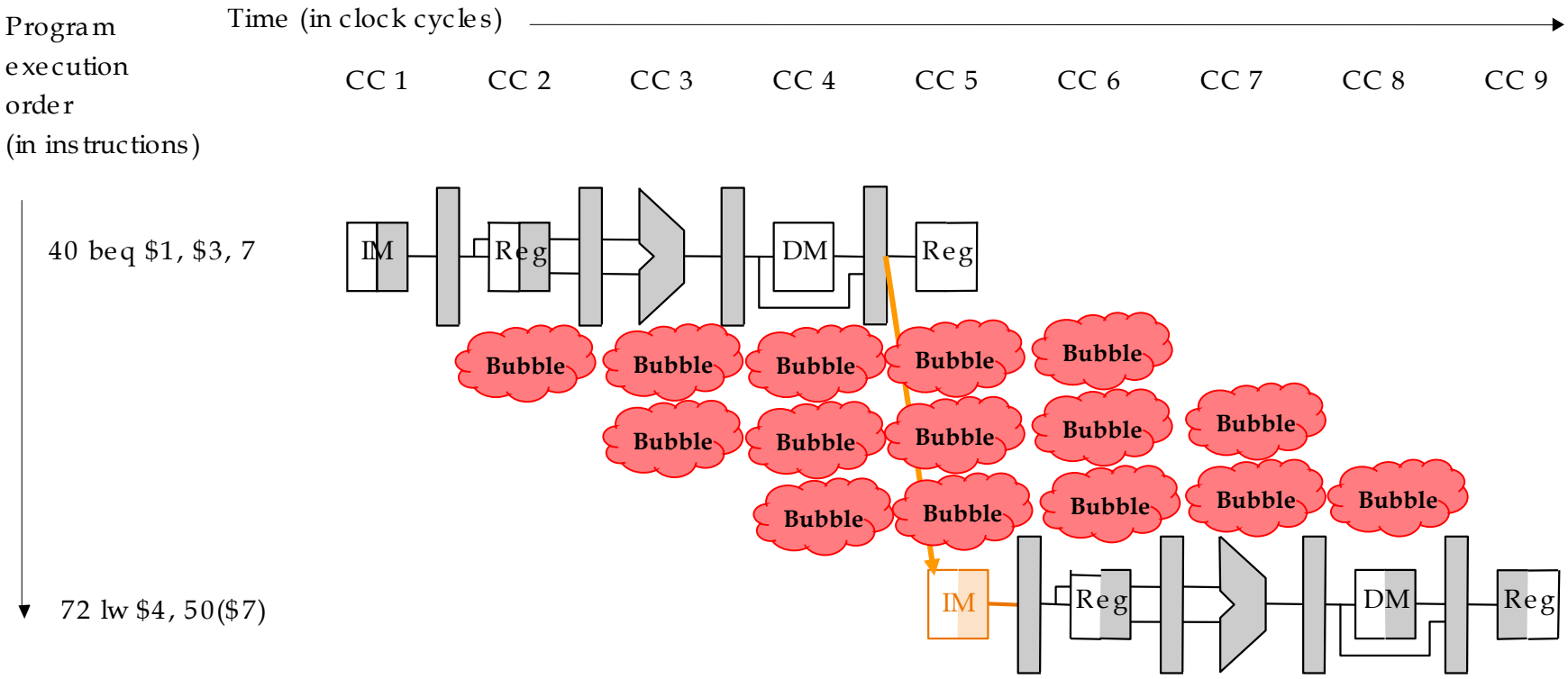


Decision is taken in **MEM** stage: too late!

Control Hazards: Example



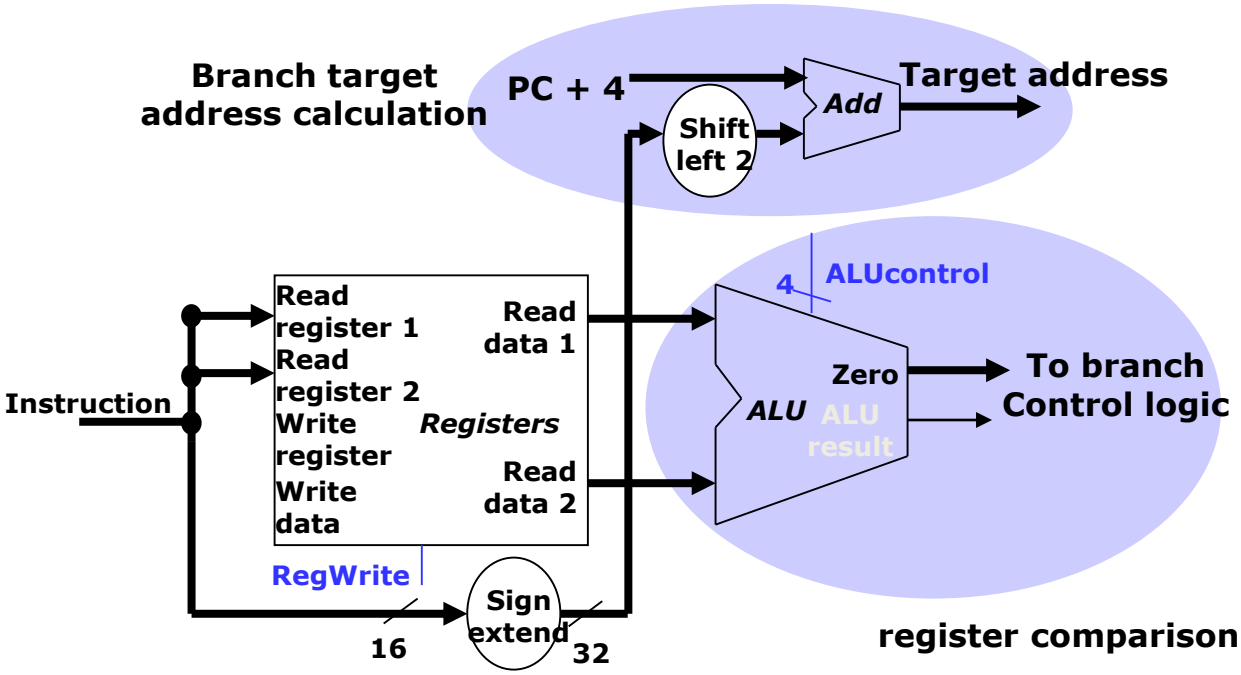
Control Hazards: Stall Pipeline



- Wait till you know the branch decision and then fetch the correct instructions
- Introduces 3 clock cycles delay

Control Hazards: Reduce Stalls (1/3)

- Make decision in ID stage instead of MEM
  - Move branch target address calculation
  - Move register comparison → cannot use ALU for register comparison any more



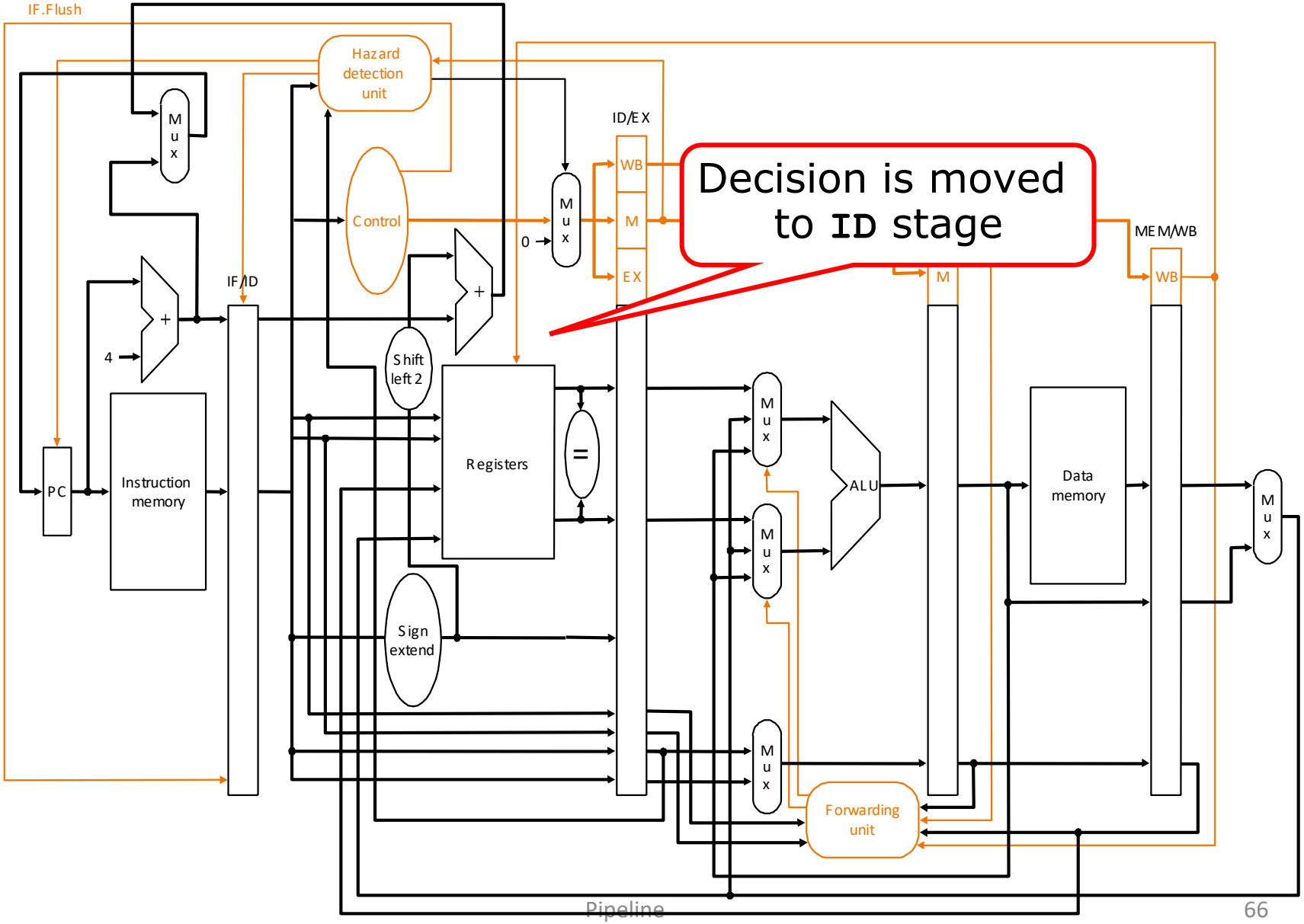
Pipeline Control

- Group control signals according to pipeline stage

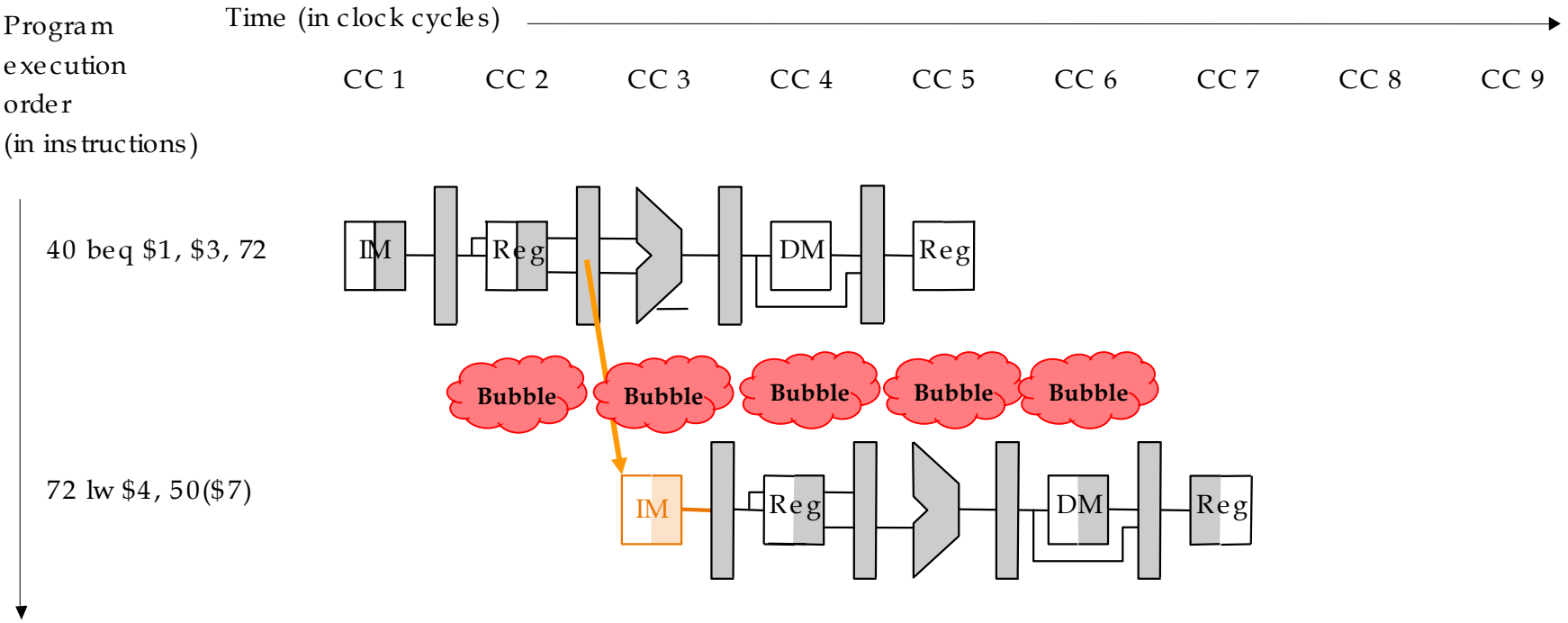
	RegDst	ALUSrc	Memto Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop 1	ALUop 0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

	EX stage				**MEM stage**			**WB stage**	
	RegDst	ALUSrc	ALU op1	ALU op0	Mem Read	Mem Write	Branch	Reg write	Memto Reg
R-type	1	0	1	0	0	0	0	1	0
lw	0	1	0	0	1	0	0	1	1
sw	X	1	0	0	0	1	0	0	X
beq	X	0	0	1	0	0	1	0	X

Control Hazards: Reduce Stalls (2/3)



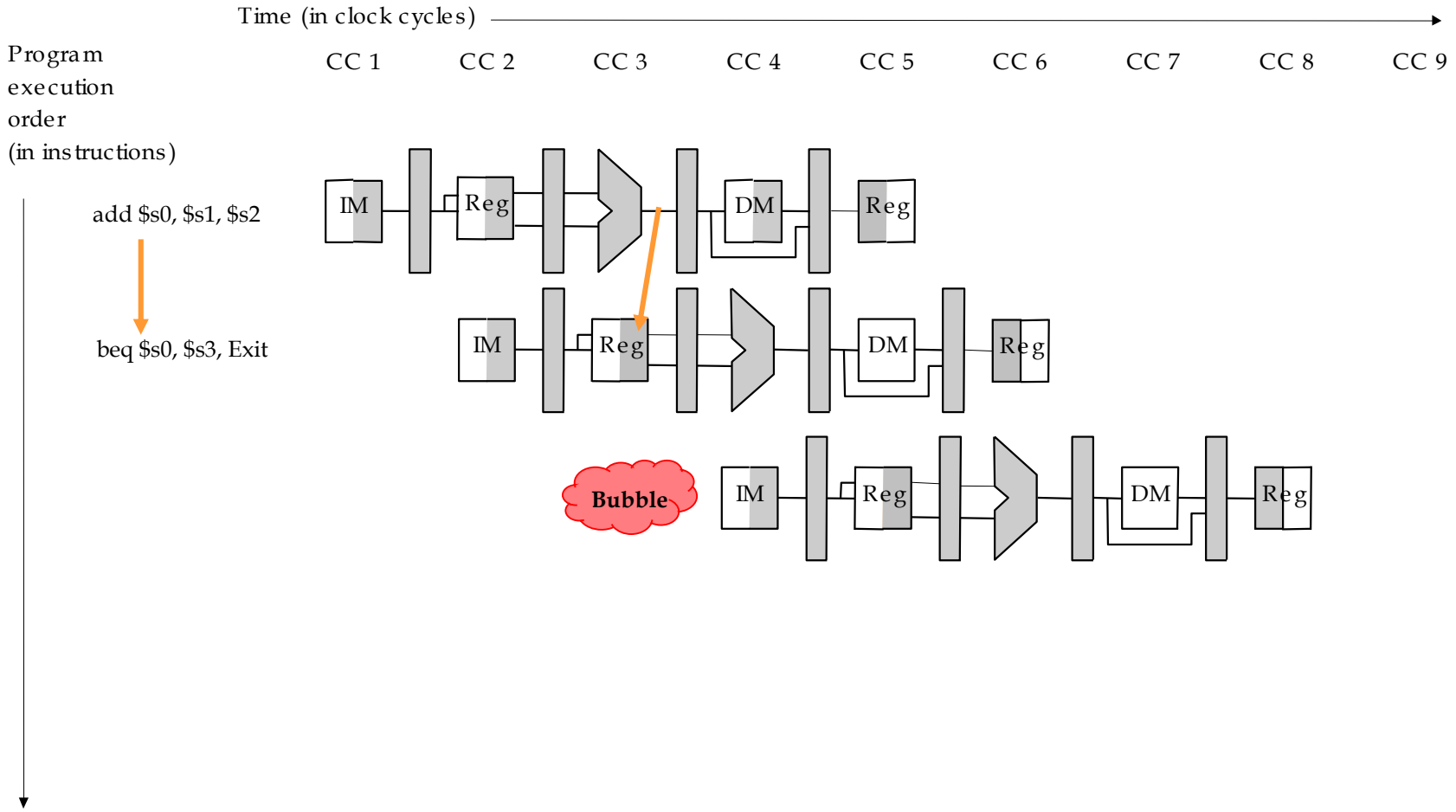
Control Hazards: Reduce Stalls (3/3)



- Wait till you know the branch decision and then fetch the correct instructions
- Introduces 1 clock cycle delay!

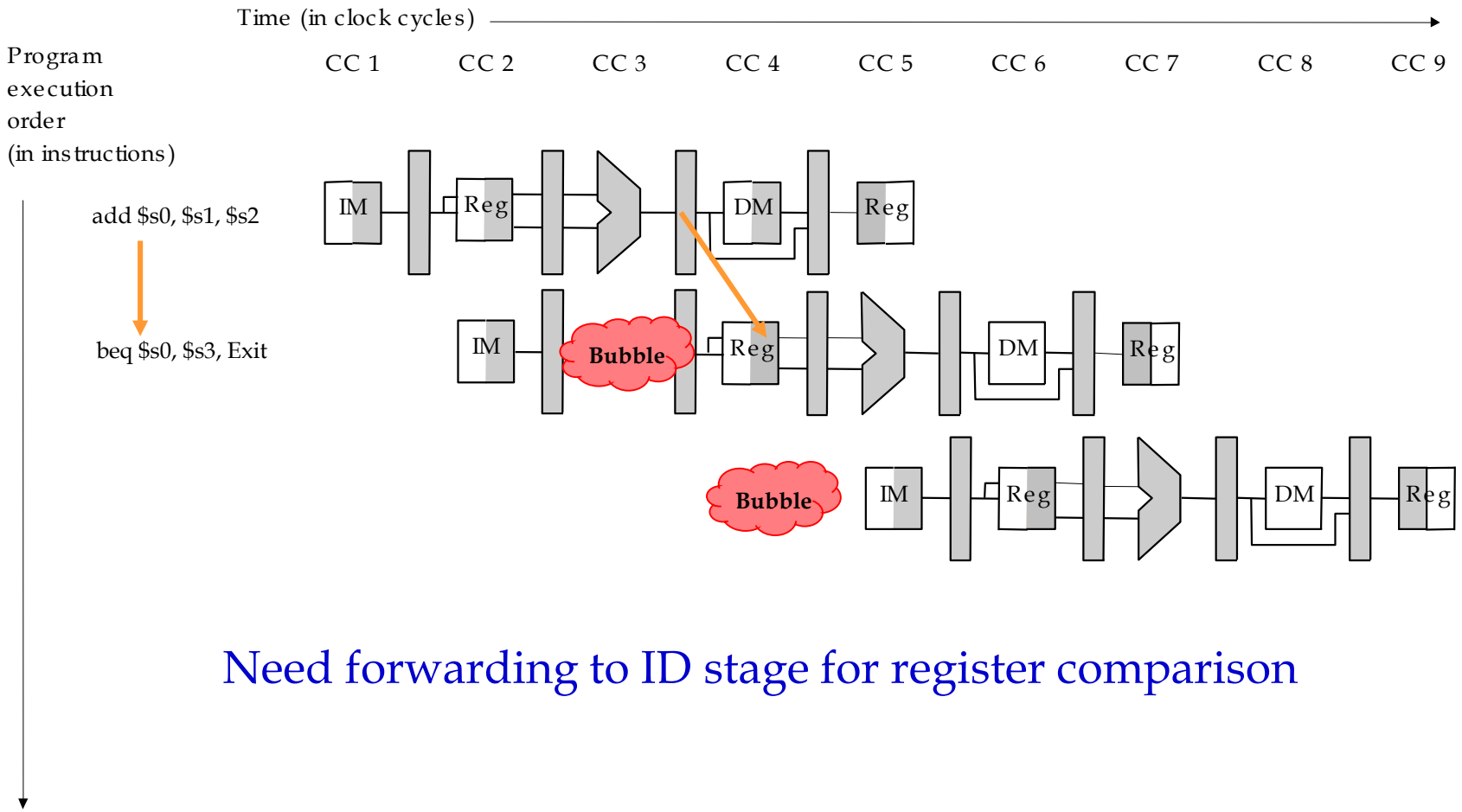
Interaction With Forwarding (1/3)

- Cannot solve with forwarding: need the data even before it is produced



Interaction With Forwarding (2/3)

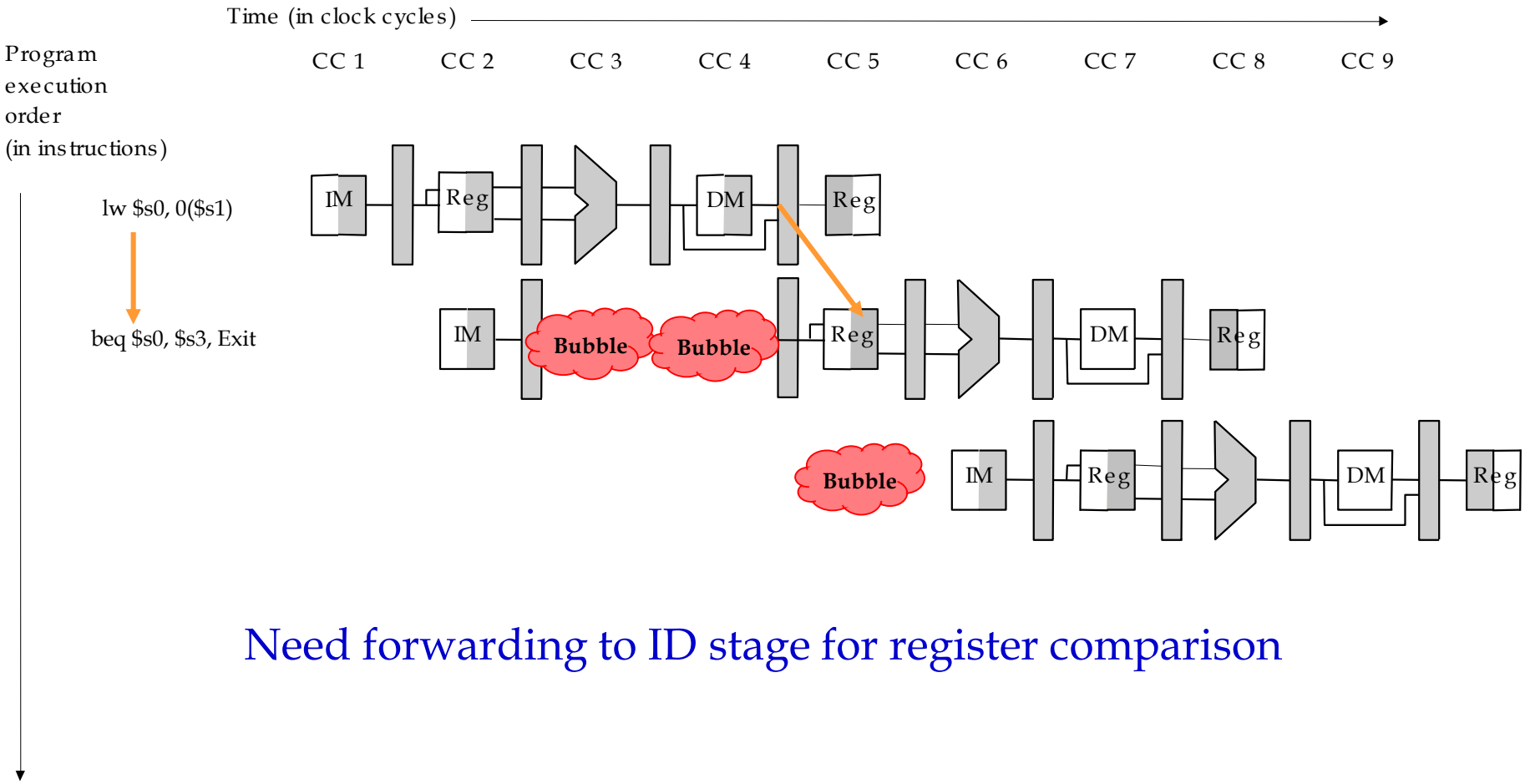
- Introduce one clock cycle delay



Need forwarding to ID stage for register comparison

Interaction With Forwarding (3/3)

- Even worse with load followed by branch

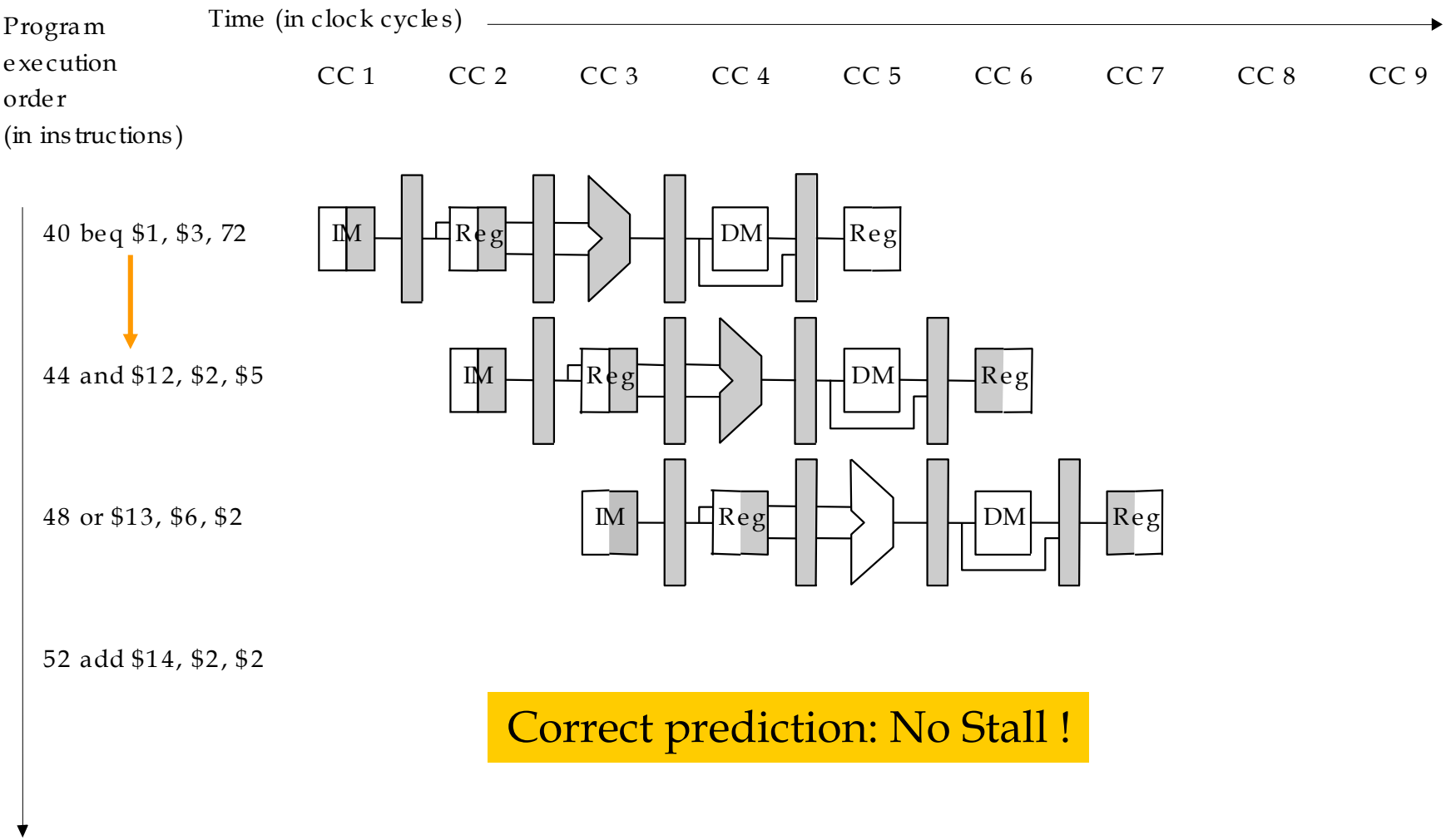


Need forwarding to ID stage for register comparison

Control Hazards: BRANCH Prediction (1/3)

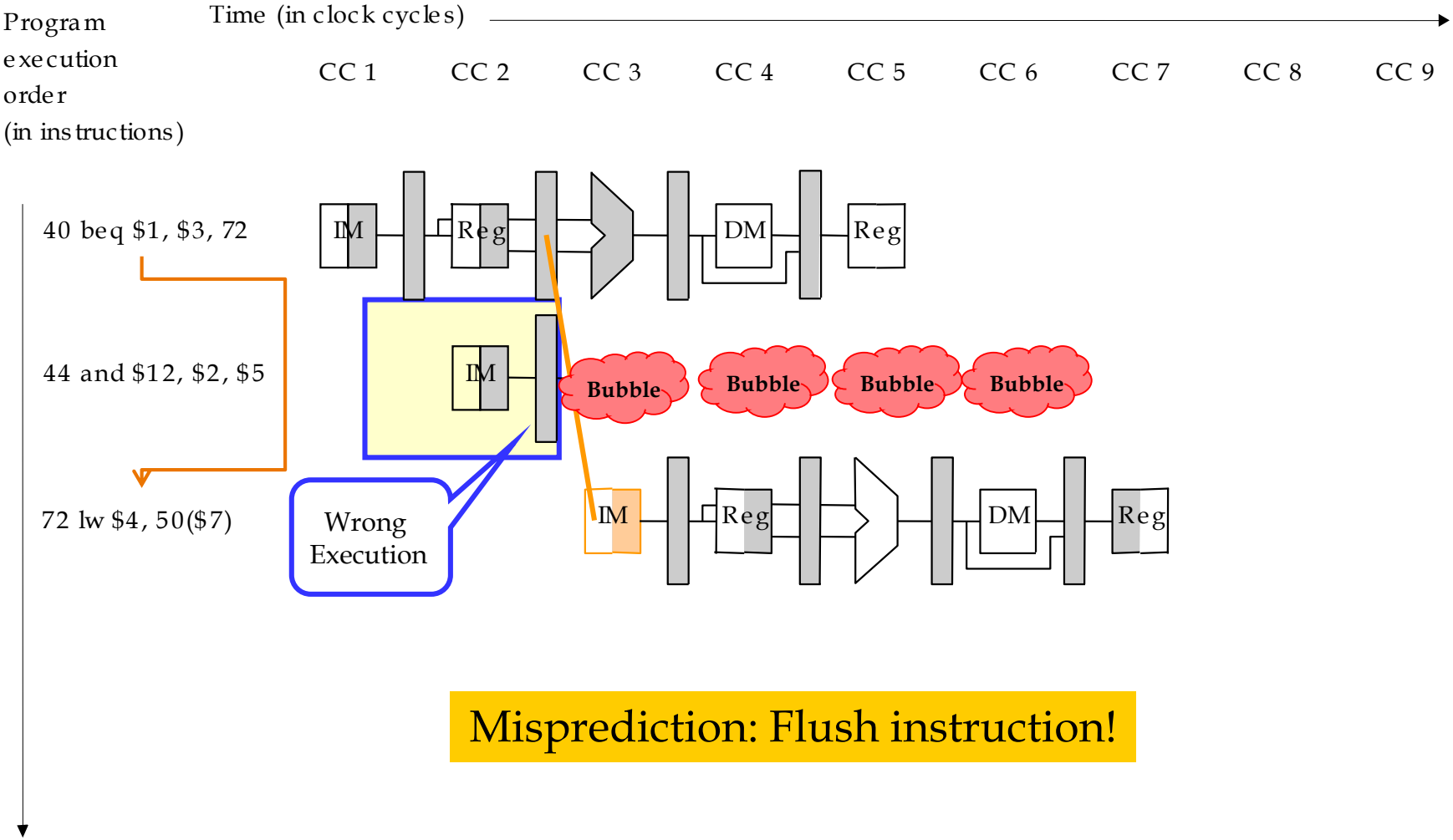
- Simple prediction: all branches untaken
- Fetch the successor instruction
- If you are right: No pipeline stall
- If you are wrong: Flush successor instruction from the pipeline

Control Hazards: BRANCH Prediction (2/3)



Correct prediction: No Stall !

Control Hazards: BRANCH Prediction (3/3)



Misprediction: Flush instruction!

Try It Yourself #6

- How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding but no branch prediction (decision making moved to ID stage)?

```
            addi  $s0, $zero, 10
Loop:       addi  $s0, $s0, -1
            bne   $s0, $zero, Loop
            xor   $t0, $t1, $t2
```

- Total instructions = 1 + 10×2 + 1 = 22
- Ideal pipeline = 4 + 22×1 = 26 cycles

Try It Yourself #7

- How many cycles will it take to execute the following code on a 5-stage pipeline with forwarding and branch prediction?

```
            addi  $s0, $zero, 10
Loop:       addi  $s0, $s0, -1
            bne   $s0, $zero, Loop
            xor   $t0, $t1, $t2
```

- Total instructions = 1 + 10×2 + 1 = 22
- Ideal pipeline = 4 + 22×1 = 26 cycles

Try It Yourself #6

	1	2	3	4	5	6	7	8	9	10	11
addi	IF	ID	EX	MEM	WB						
addi		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
addi						IF	ID	EX	MEM	WB	

Data dependency between (addi $s0, $s0, -1) and bne incurs 1 cycle of delay. There are 10 iterations, hence 10 cycles of delay.
Every bne incurs a cycle of delay to execute the next instruction. There are 10 iterations, hence 10 cycles of delay.
Total number of cycles of delay = 20
Total execution cycles = 26 + 20 = **46 cycles**

Try It Yourself #7

	1	2	3	4	5	6	7	8	9	10	11
addi	IF	ID	EX	MEM	WB						
addi		IF	ID	EX	MEM	WB					
bne			IF		ID	EX	MEM	WB			
addi						IF	ID	EX	MEM	WB	

Almost the same as Try It Yourself #6.
The data dependency remains, hence 10 cycles of delay for 10 iterations.
In the last iteration, the branch prediction is correct, hence saving 1 cycle of delay.
Total number of cycles of delay = 19
Total execution cycles = 26 + 19 = **45 cycles**

Branch-Delay (1/3)

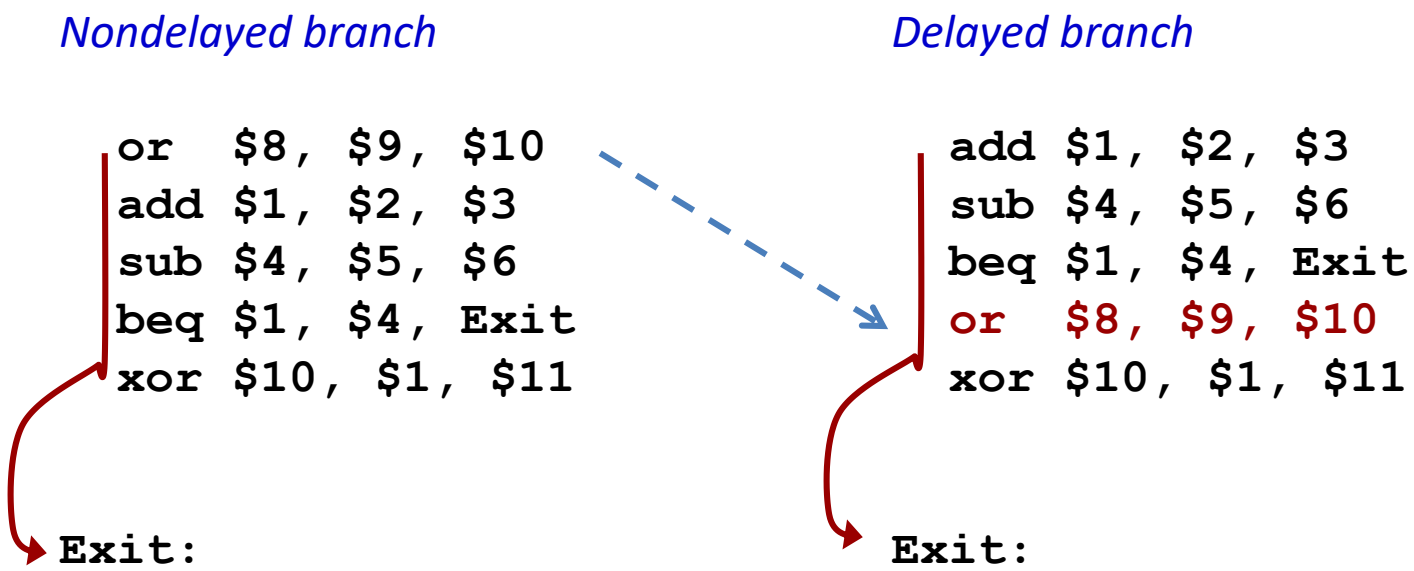
- Redefine branches
- Whether or not we take the branch, the instruction immediately following the branch gets executed
- This is called the branch-delay slot

Branch-Delay (2/3)

- Worst-case scenario
  - Add a no-op (nop) instruction in the branch-delay slot
- Better scenario
  - Find an instruction preceding the branch which can be placed in the branch-delay slot without affecting flow of the program
- Re-ordering instructions is a common method of speeding up programs
  - Compiler must be smart enough to do this
  - Usually can find such an instruction at least 50% of the time

Branch-Delay (3/3)

- Example:



SUMMARY

- Pipelining is a fundamental concept in computer systems!
  - Multiple instructions in flight
  - Limited by length of the longest stage
  - Hazards create trouble by stalling pipeline
- Pentium 4 has 22 pipeline stages!

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Sun UltraSPARC III	2003	1950 MHz	14	4	No	1	90	W
Sun UltraSPARC T1 (Niagara)	2005	1200 MHz	6	1	No	8	70	W

FIGURE 4.73 Record of Intel and Sun Microprocessors in terms of pipeline complexity, number of cores, and power. The Pentium 4 pipeline stages do not include the commit stages. If we included them, the Pentium 4 pipelines would be even deeper.

READING ASSIGNMENT

- The Processor: Pipeline
  - 4th , 5th edition: Chapter 4 Section 4.5 – 4.8