

CSEE 3827: Fundamentals of Computer Systems

Pipelined MIPS Implementation

Single-Cycle CPU Performance Issues

- Longest delay determines clock period
 - Critical path: load instruction
 - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary clock period for different instructions
- We will improve performance by pipelining

Pipelining Laundry Analogy

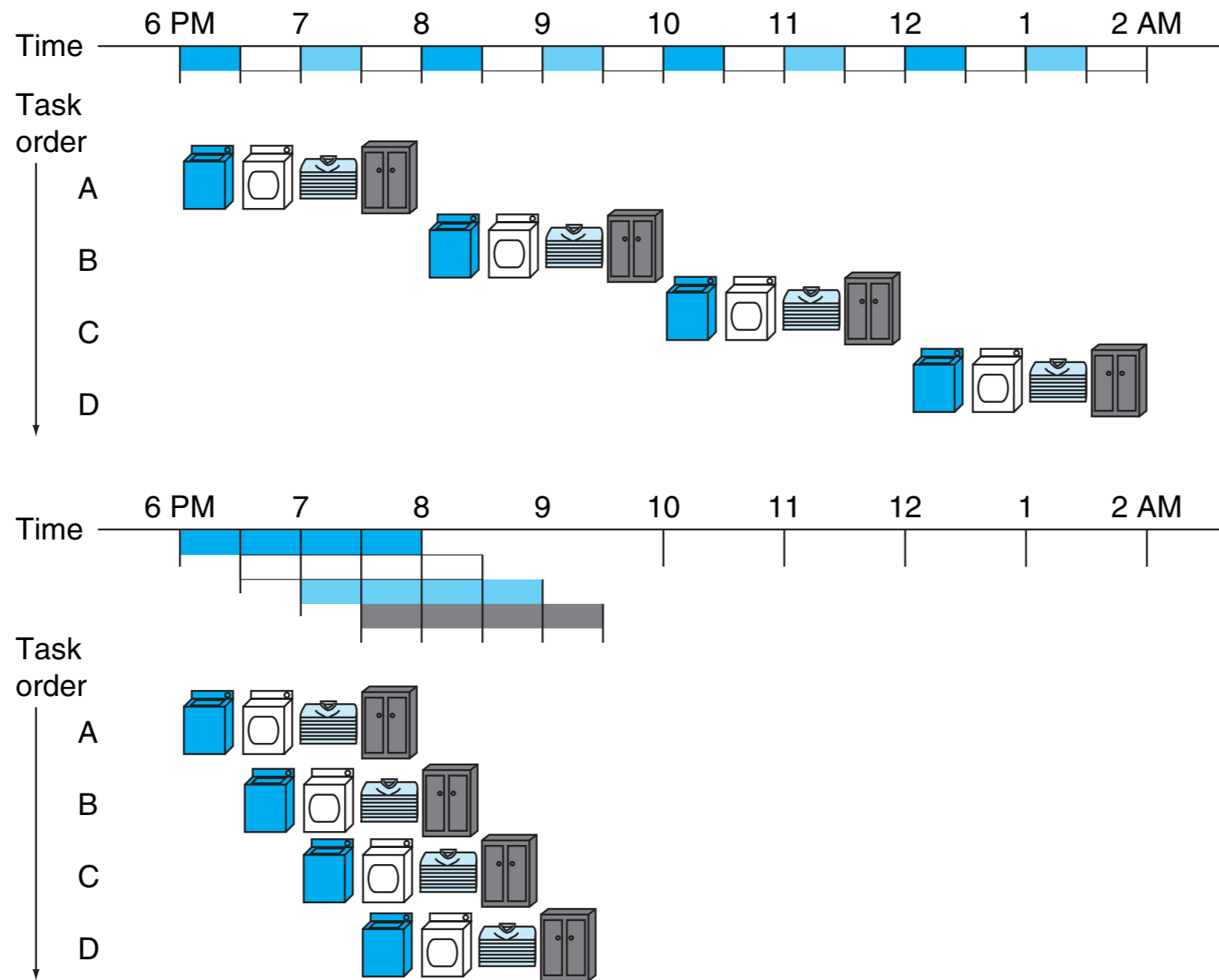


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storer” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource. Copyright © 2009 Elsevier, Inc. All rights reserved.

MIPS Pipeline

- Five stages, one step per stage
 - IF: Instruction fetch from (instruction) memory
 - ID: Instruction decode and register read (register file read)
 - EX: Execute operation or calculate address (ALU) or branch condition + calculate branch address
 - MEM: Access memory operand (memory) / adjust PC counter
 - WB: Write result back to register (reg file again)
- Note: not every instruction needs every stage

MIPS Pipeline Illustration 1

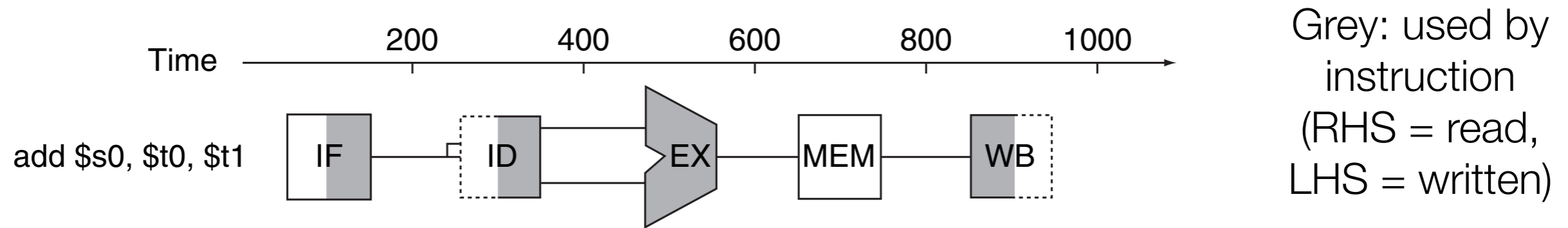


FIGURE 4.28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline in Figure 4.25. Here we use symbols representing the physical resources with the abbreviations for pipeline stages used throughout the chapter. The symbols for the five stages: *IF* for the instruction 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 file being written. The shading indicates the element is used by the instruction. Hence, MEM has a white background because `add` does not access the data memory. Shading on the right half of the register file 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. Copyright © 2009 Elsevier, Inc. All rights reserved.

MIPS Pipeline Illustration 2

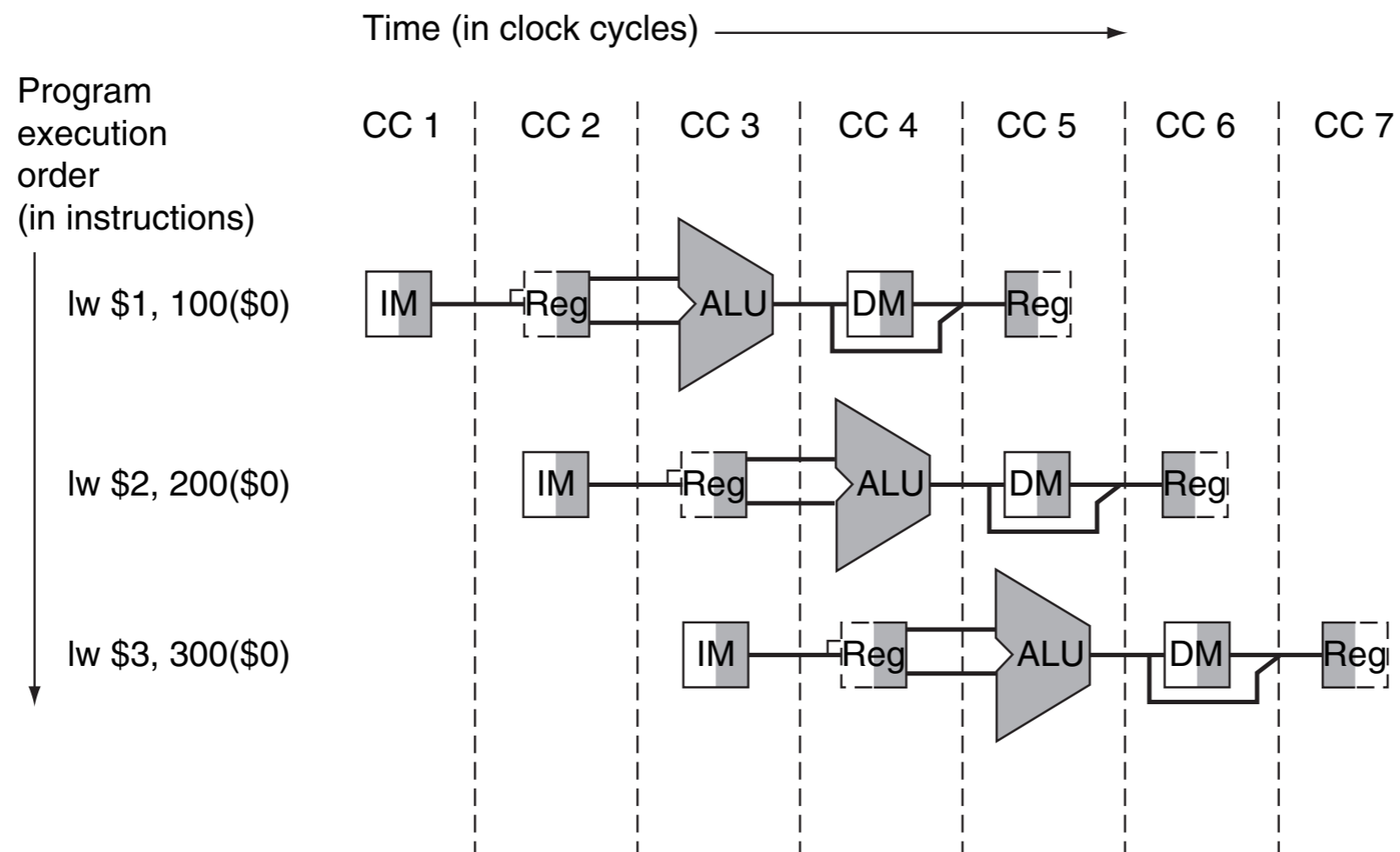


FIGURE 4.34 Instructions being executed using the single-cycle datapath in Figure 4.33, assuming pipelined execution. Similar to Figures 4.28 through 4.30, this figure pretends that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33. *IM* represents the instruction memory and the PC in the instruction fetch stage, *Reg* stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on. To maintain proper time order, this stylized datapath breaks the register file into two logical parts: registers read during register fetch (ID) and registers written during write back (WB). This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read. As before, we assume the register file is written in the first half of the clock cycle and the register file is read during the second half. Copyright © 2009 Elsevier, Inc. All rights reserved.

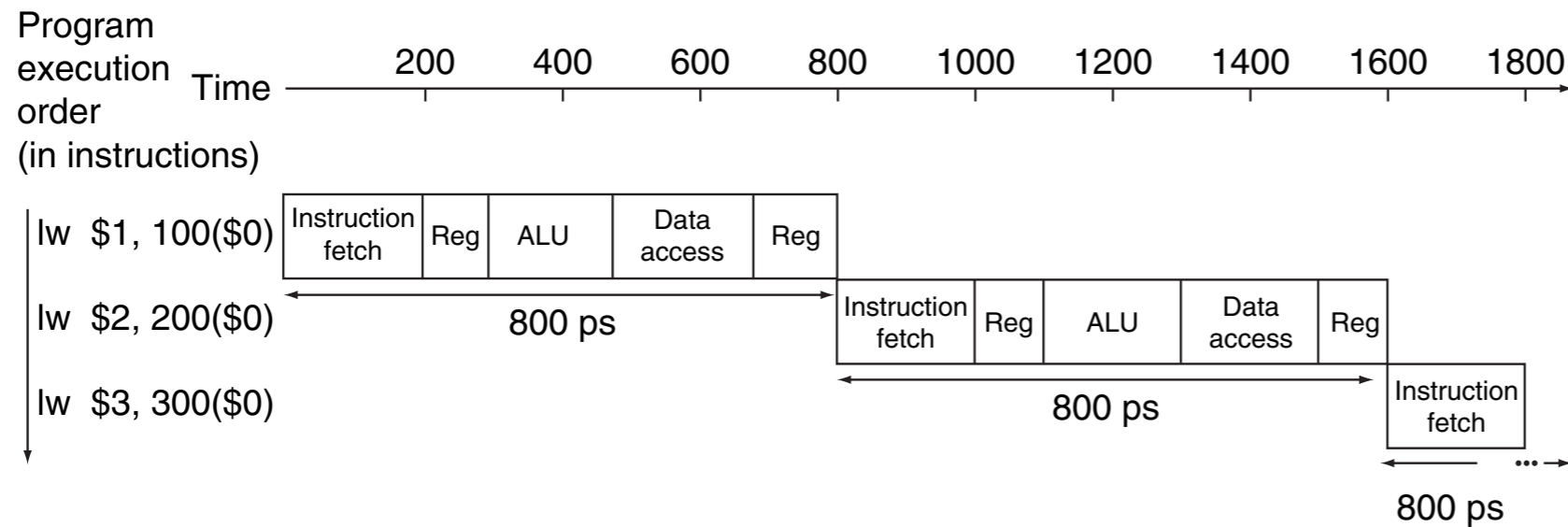
Pipeline Performance 1

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath to single-cycle datapath

Instr	IF	ID	EX	MEM	WB	Total (PS)
lw	200	100	200	200	100	800
sw	200	100	200	200		700
R-format	200	100	200		100	600
beq	200	100	200			500

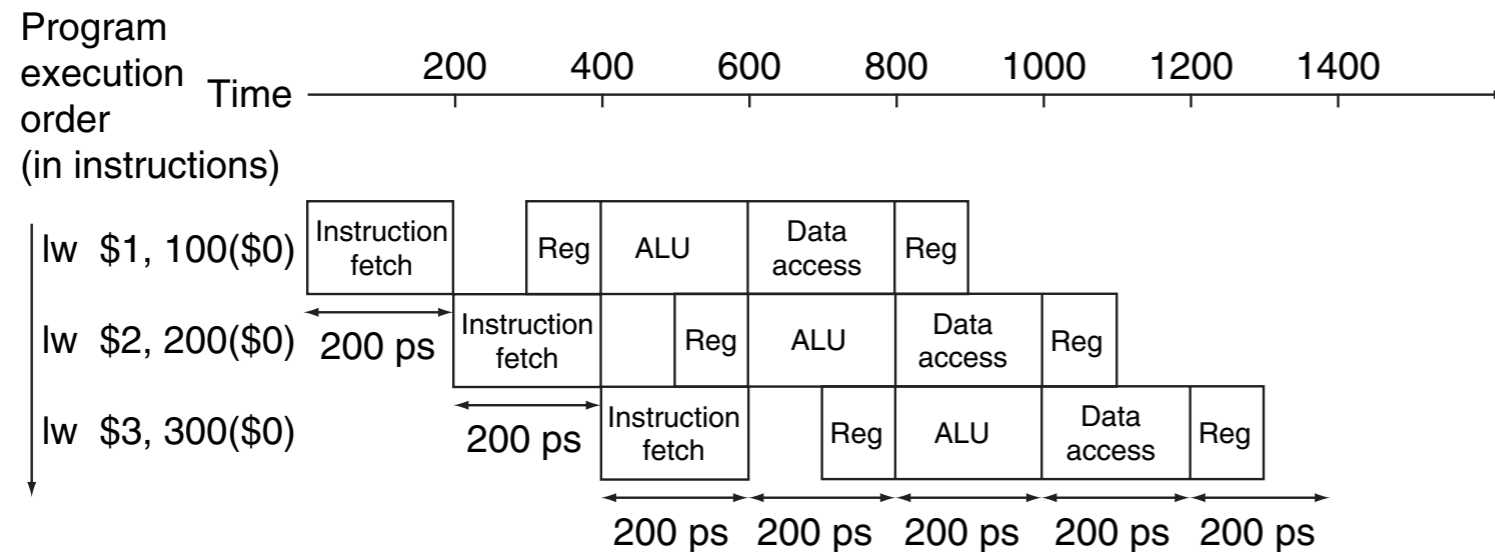


Pipeline Performance 2



Single-cycle $T_{\text{clock}} = 800\text{ps}$

Total time: 2400 ps



Pipelined $T_{\text{clock}} = 200\text{ps}$

Total time: 1400 ps

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 file 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. Copyright © 2009 Elsevier, Inc. All rights reserved.



Pipeline Speedup

- Speedup due to increased throughput.
- If all stages are balanced (i.e., all take the same time)

*Pipeline instr. completion rate = Single-cycle instr. completion rate * Number of stages*

- If not balanced, speedup is less

MIPS Pipelined Datapath

MIPS Pipelined Datapath

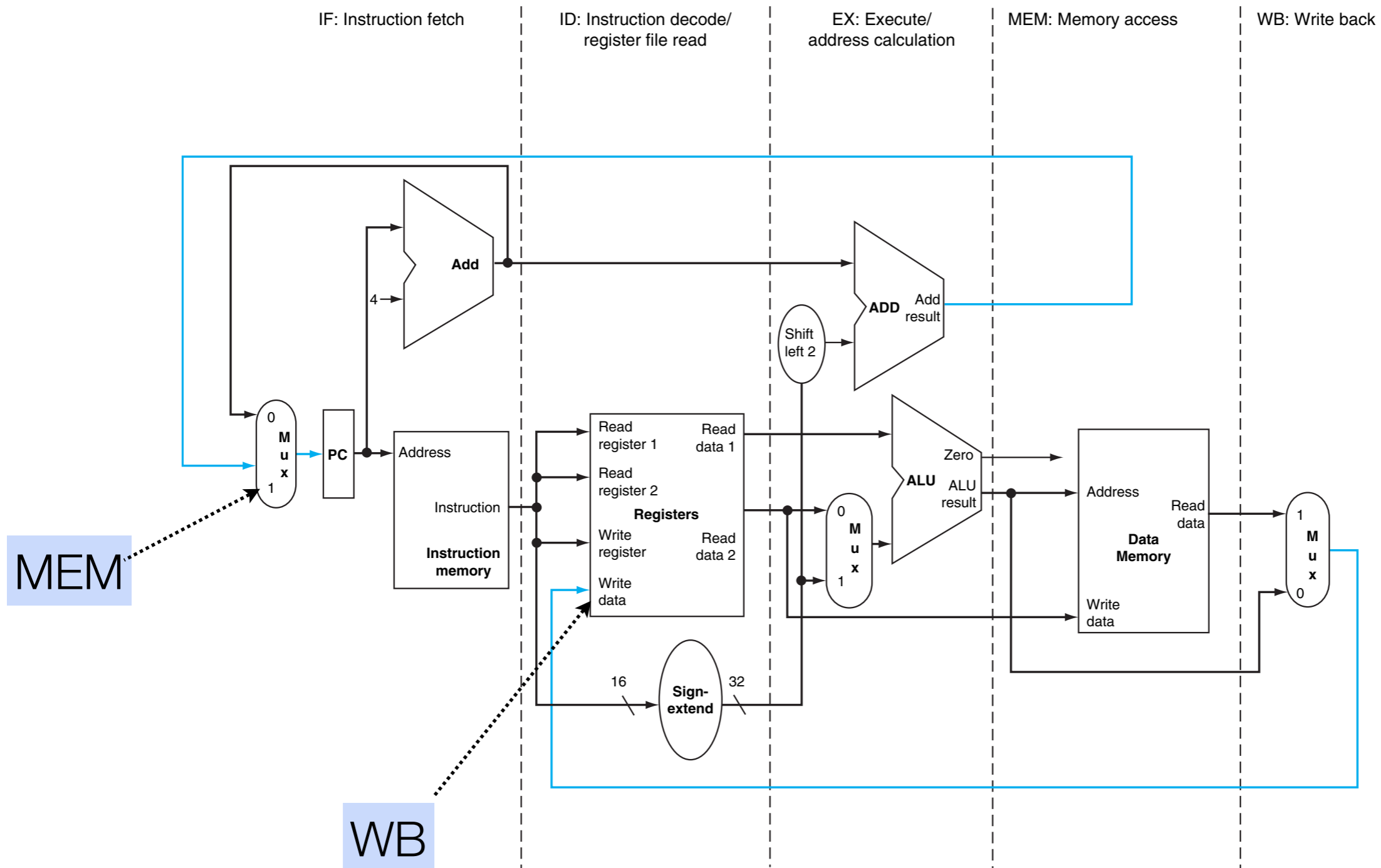


FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.) Copyright © 2009 Elsevier, Inc. All rights reserved.



Pipeline registers

- Need registers between stages, to hold information produced in previous cycle

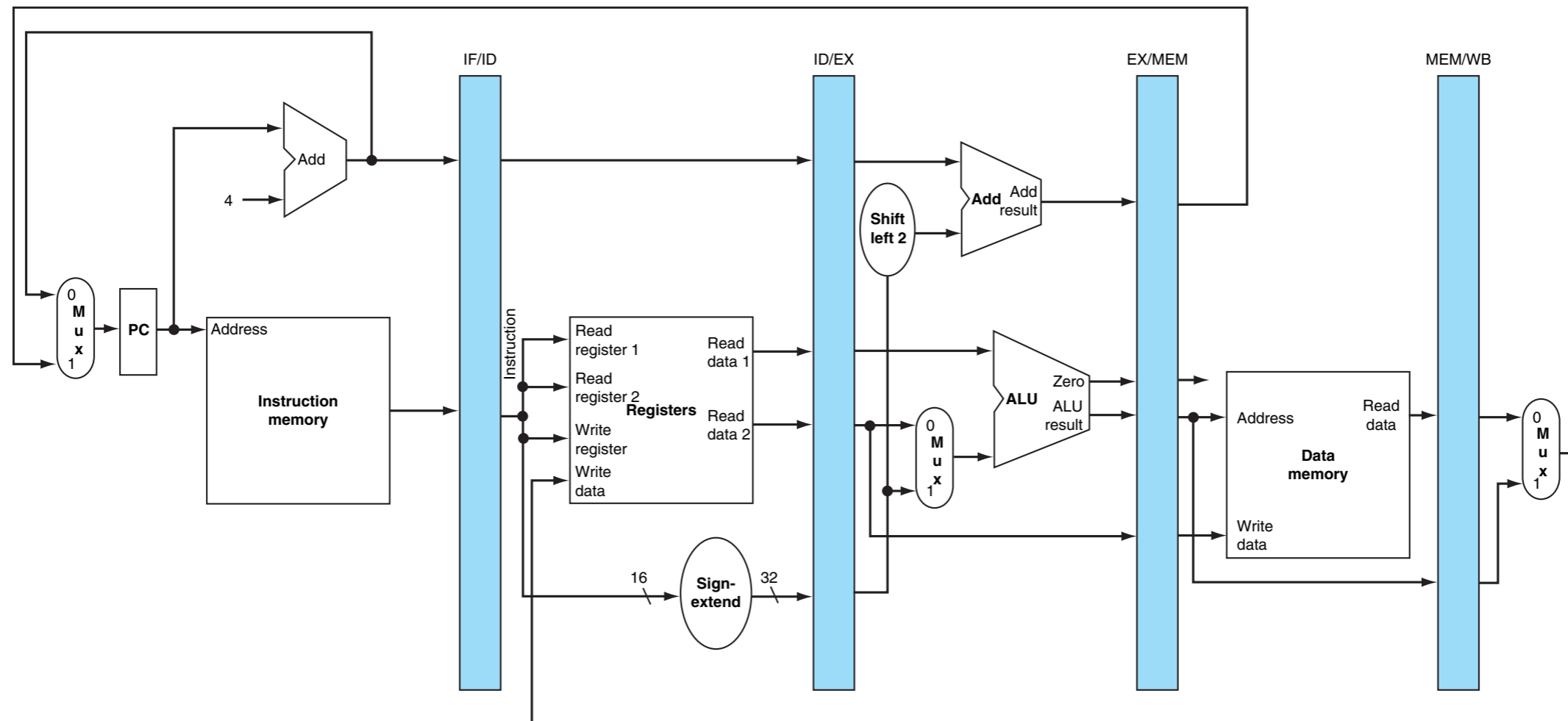
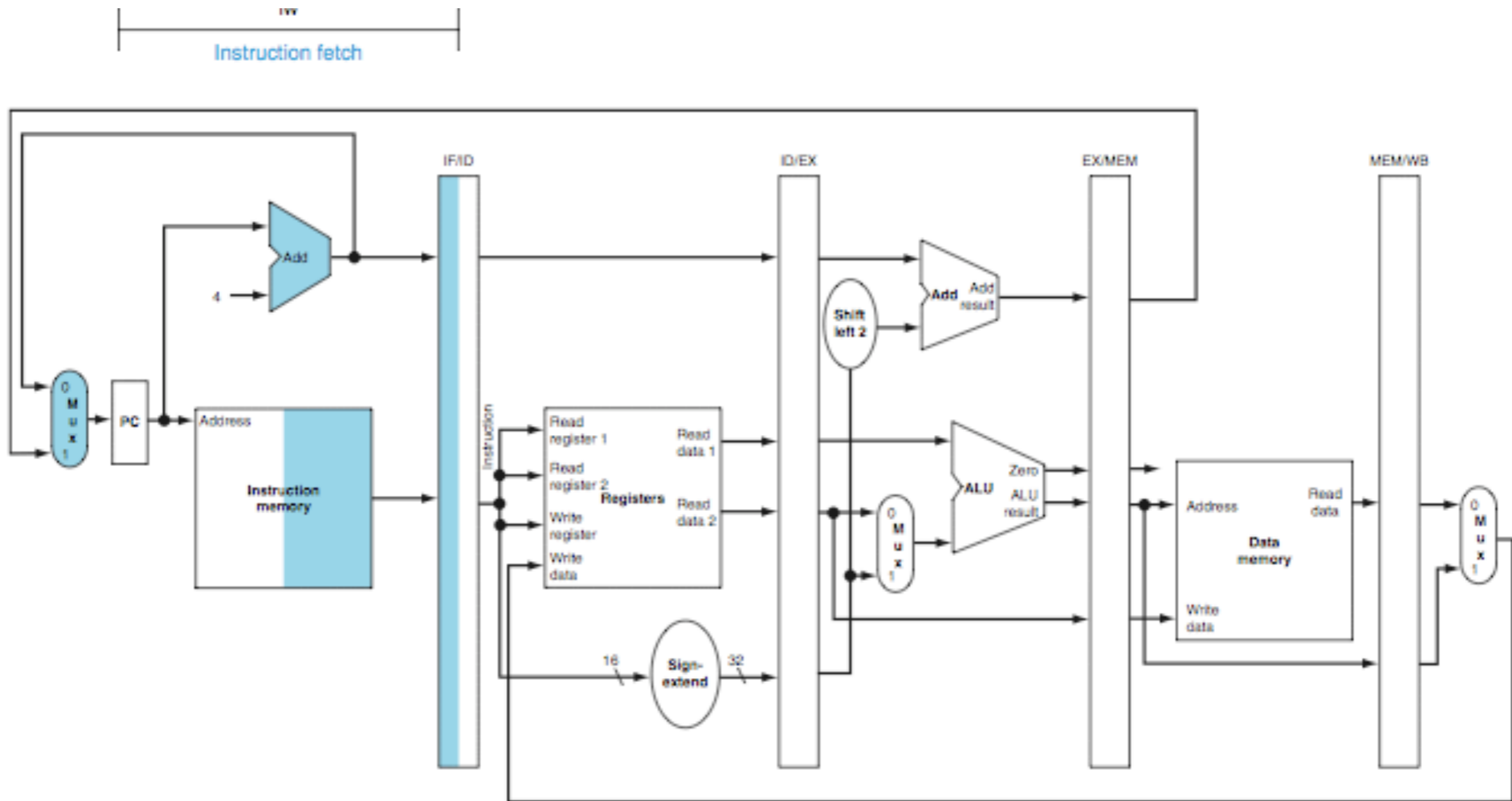


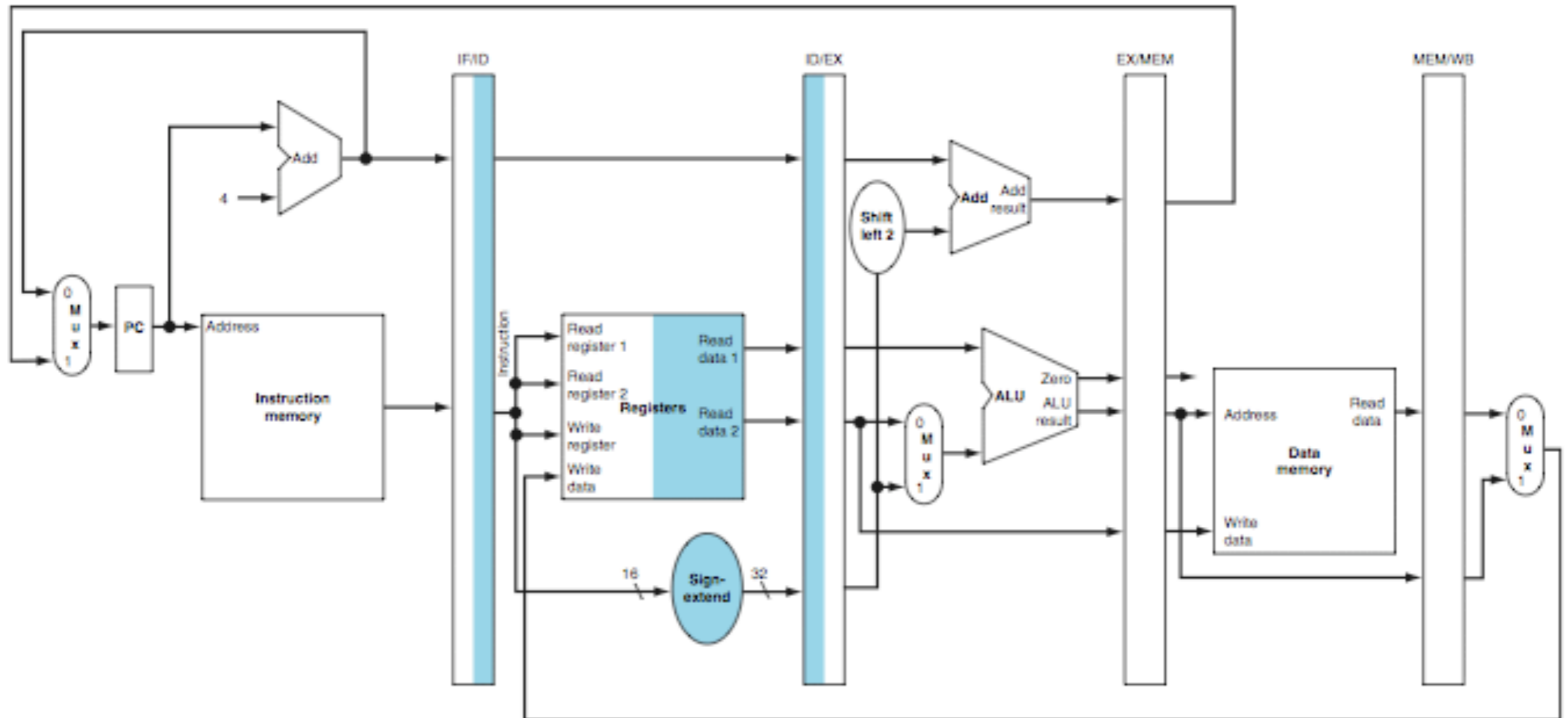
FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the *IF/ID* register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively. Copyright © 2009 Elsevier, Inc. All rights reserved.

IF for Load

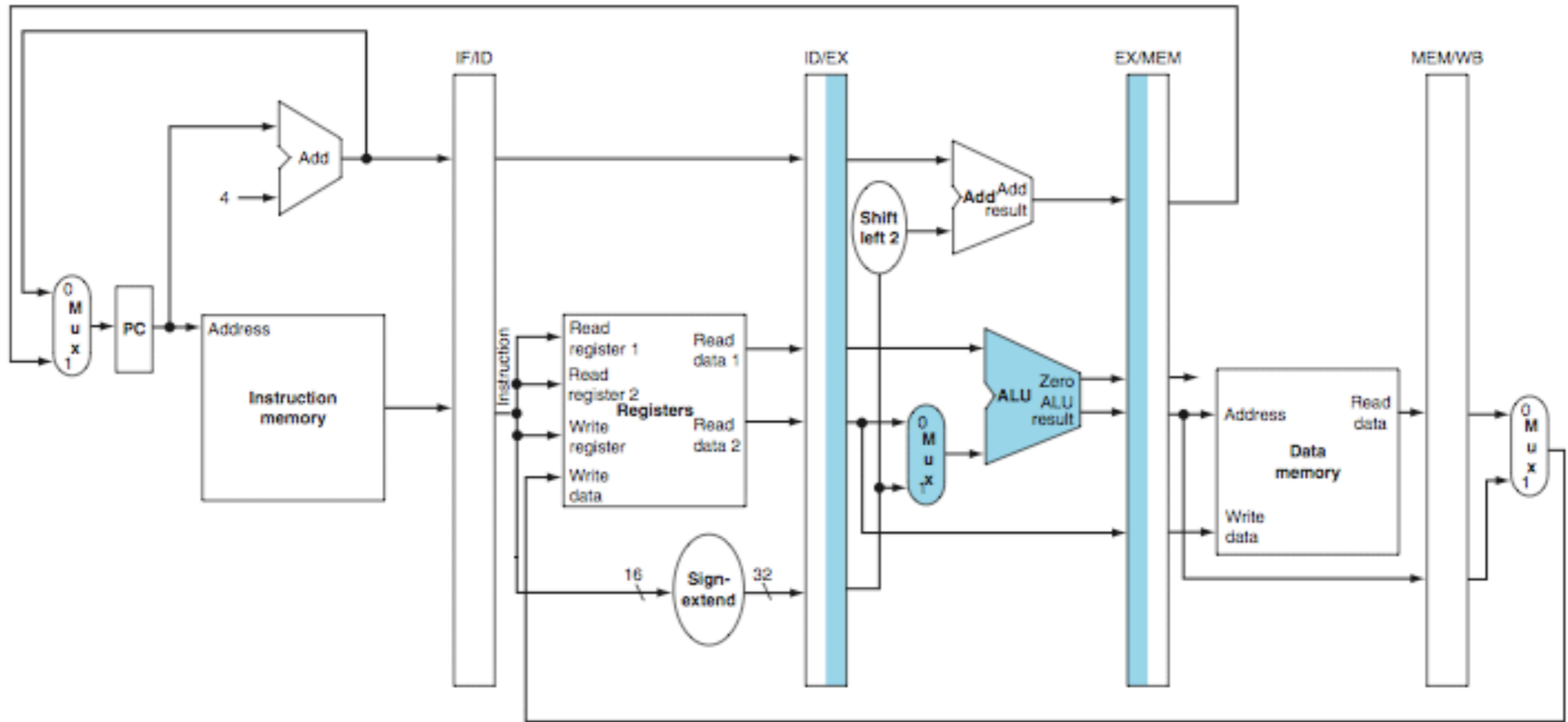


ID for Load

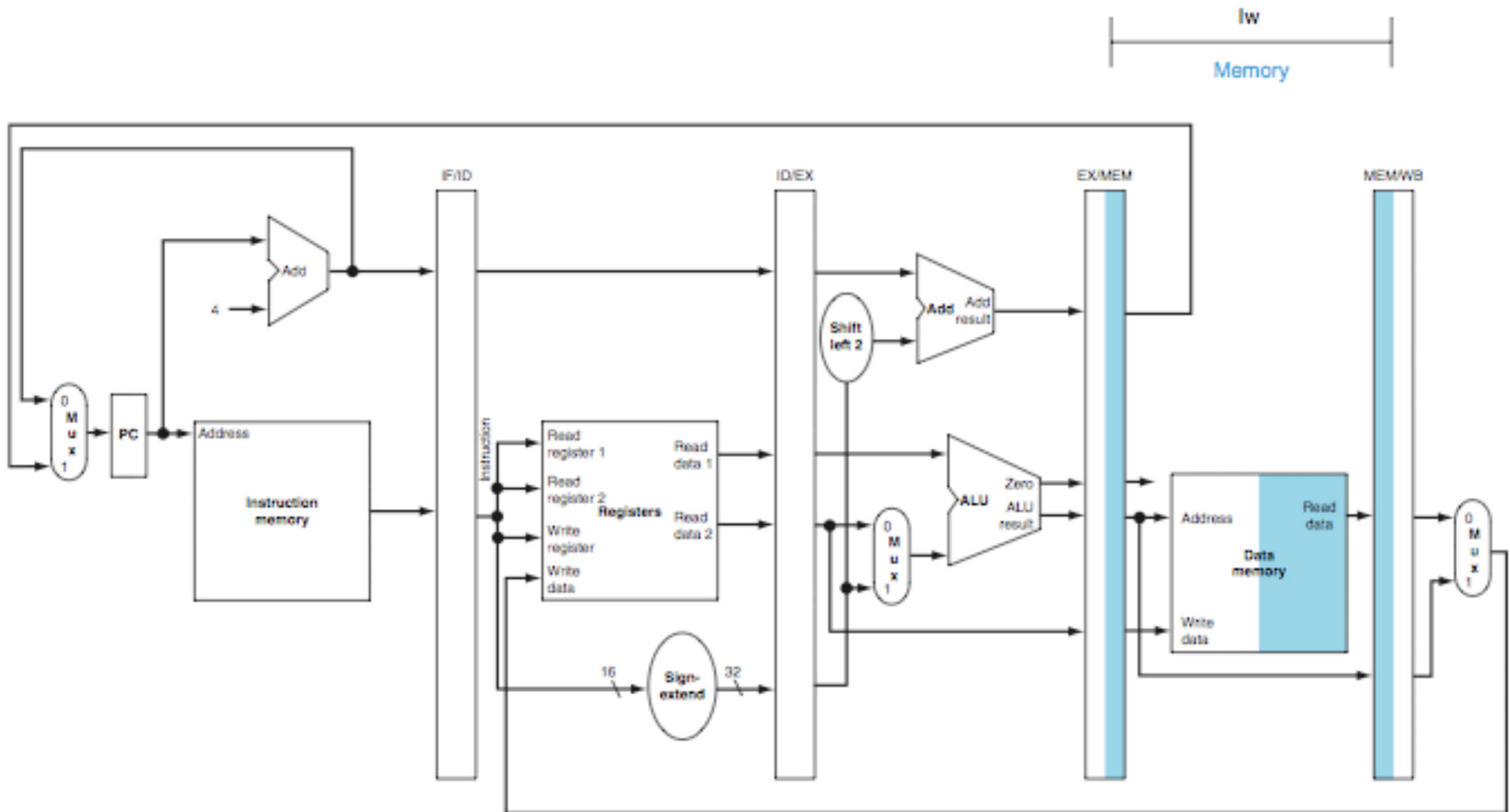
lw
Instruction decode



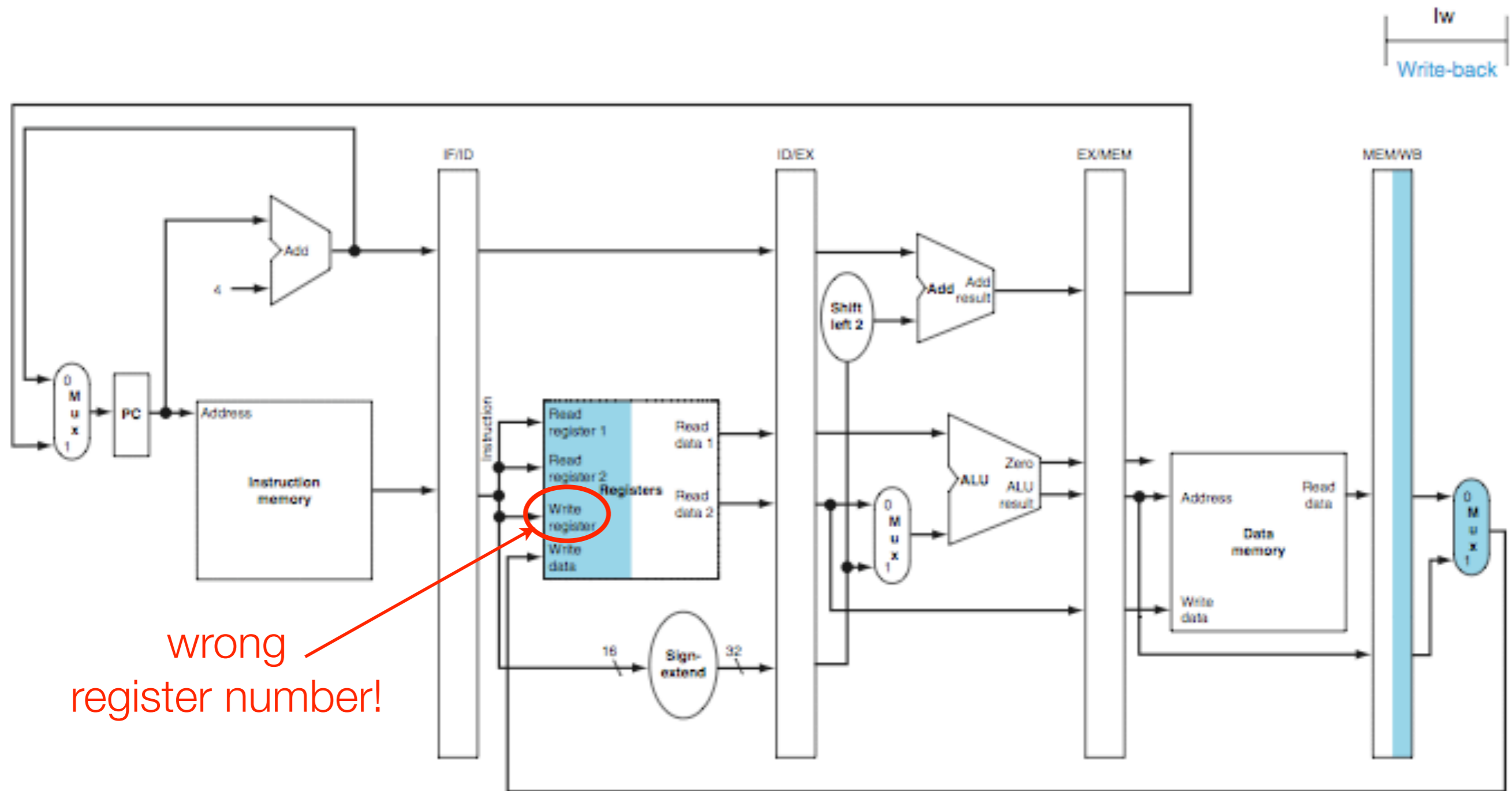
EX for Load



MEM for Load



WB for Load



wrong register number!

Corrected Datapath for Load

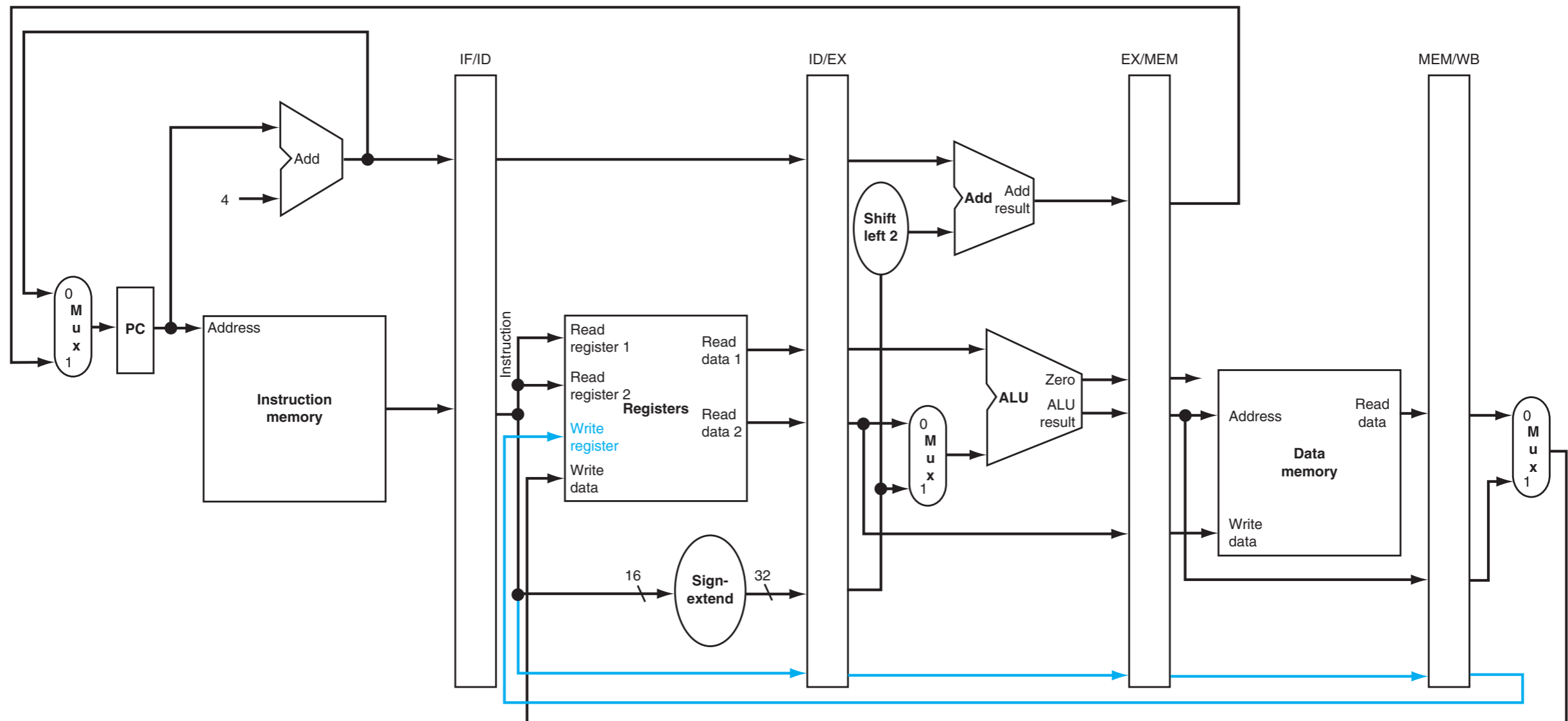


FIGURE 4.41 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color. Copyright © 2009 Elsevier, Inc. All rights reserved.

(A single-cycle pipeline diagram)



Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
 - “Single-clock-cycle” pipeline diagram
 - Shows pipeline usage in a single cycle
 - Highlight resources used
 - c.f. “multi-clock-cycle” diagram
 - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

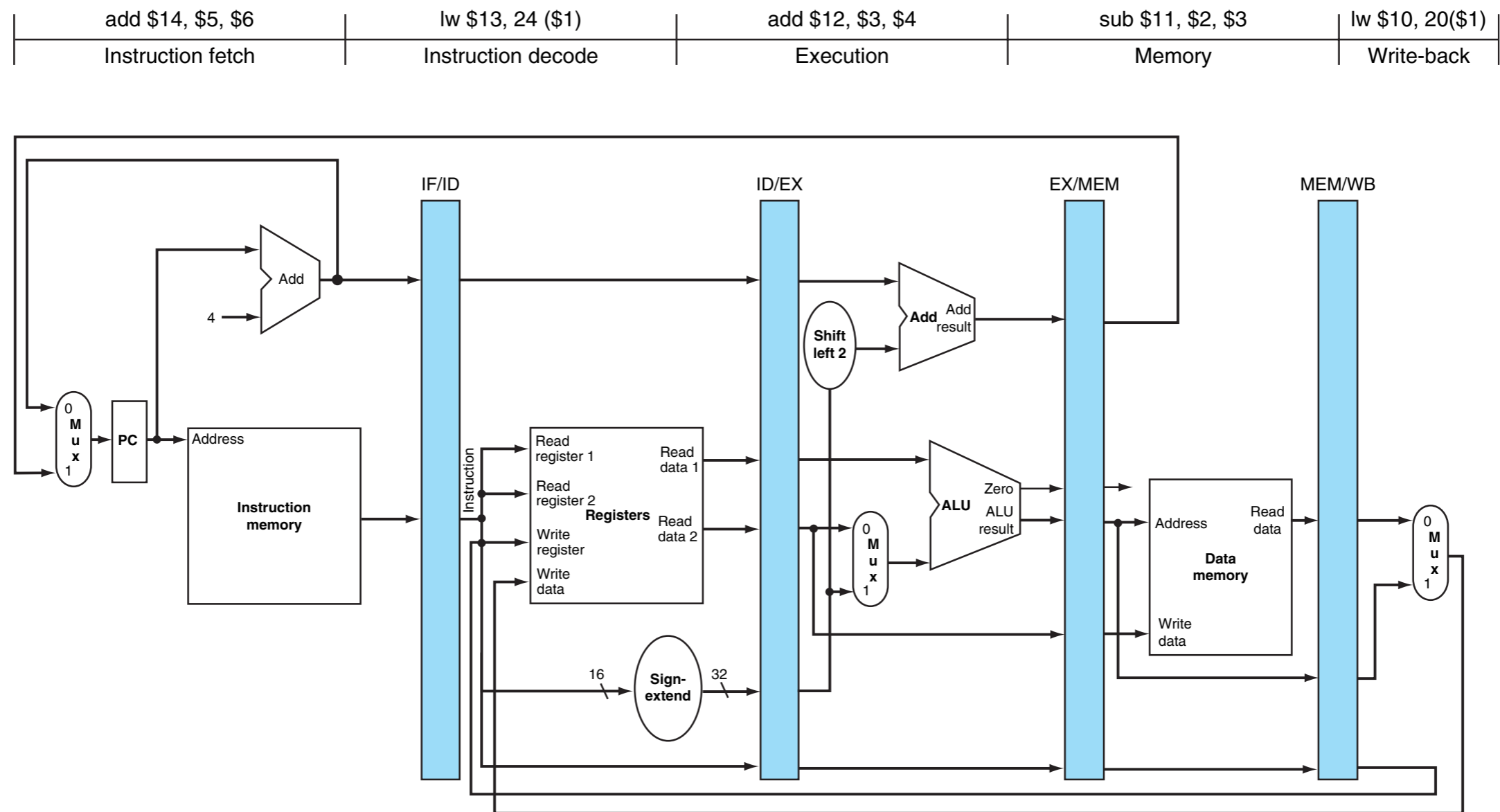


FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44.

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.



Multi-Cycle Pipeline Diagram 1

- Form showing resource usage over time

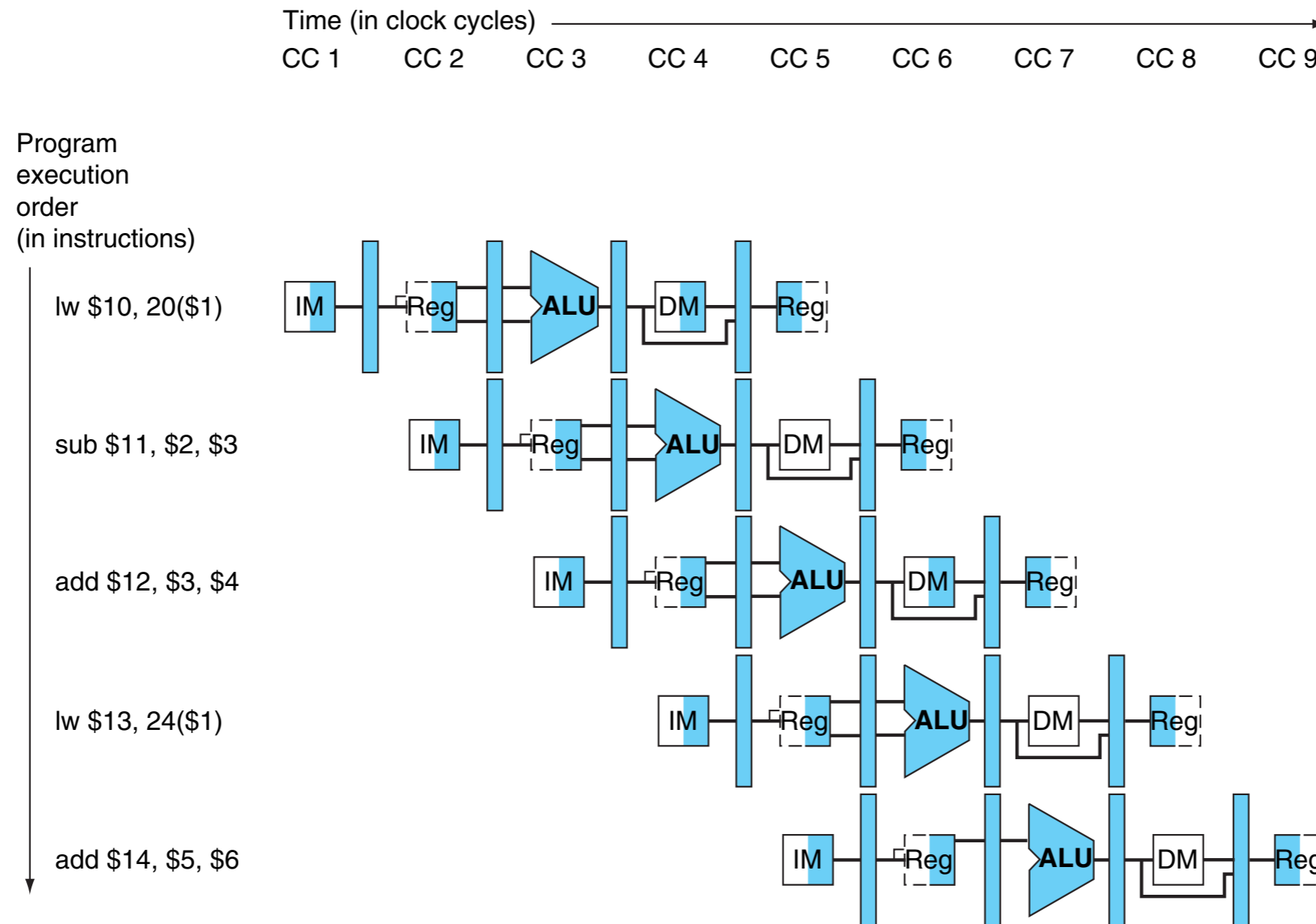


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram. Copyright © 2009 Elsevier, Inc. All rights reserved.

Multi-Cycle Pipeline Diagram 2

- Traditional form

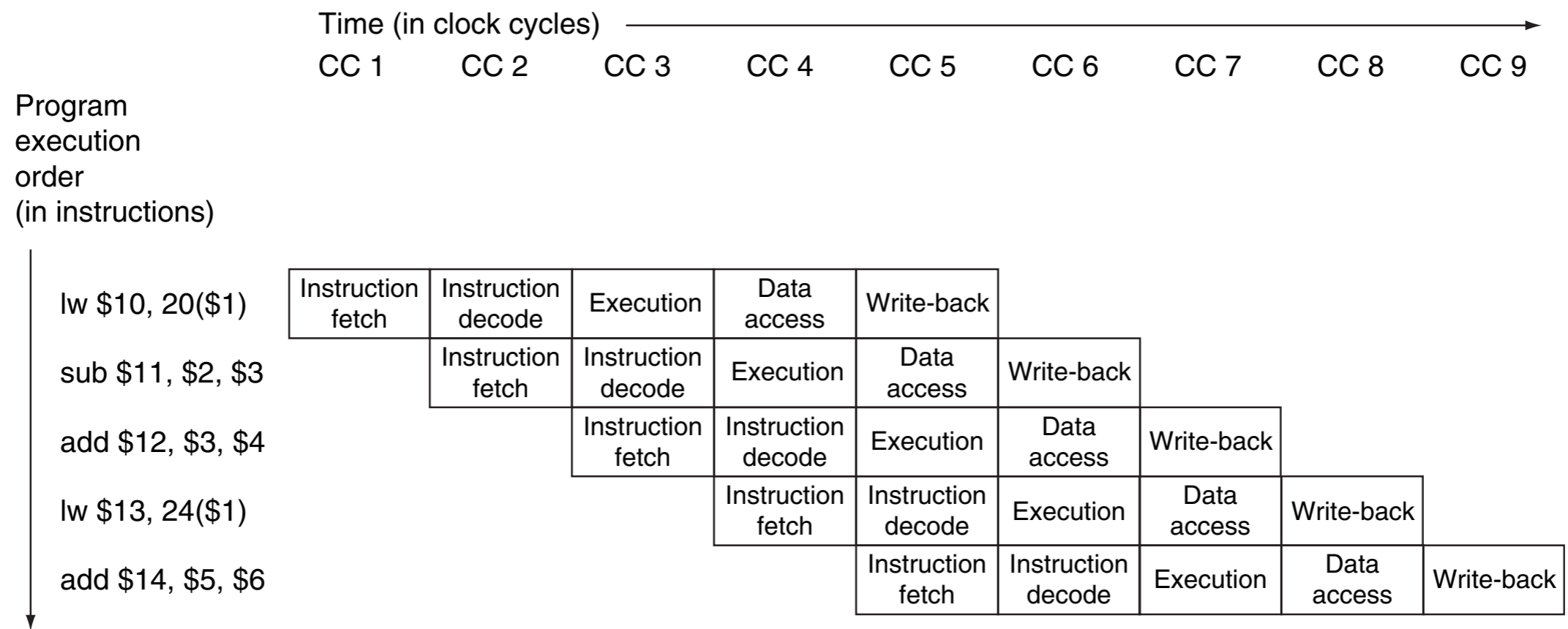


FIGURE 4.44 Traditional multiple-clock-cycle pipeline diagram of five instructions in Figure 4.43. Copyright © 2009 Elsevier, Inc. All rights reserved.

Pipelined Control (Simplified)

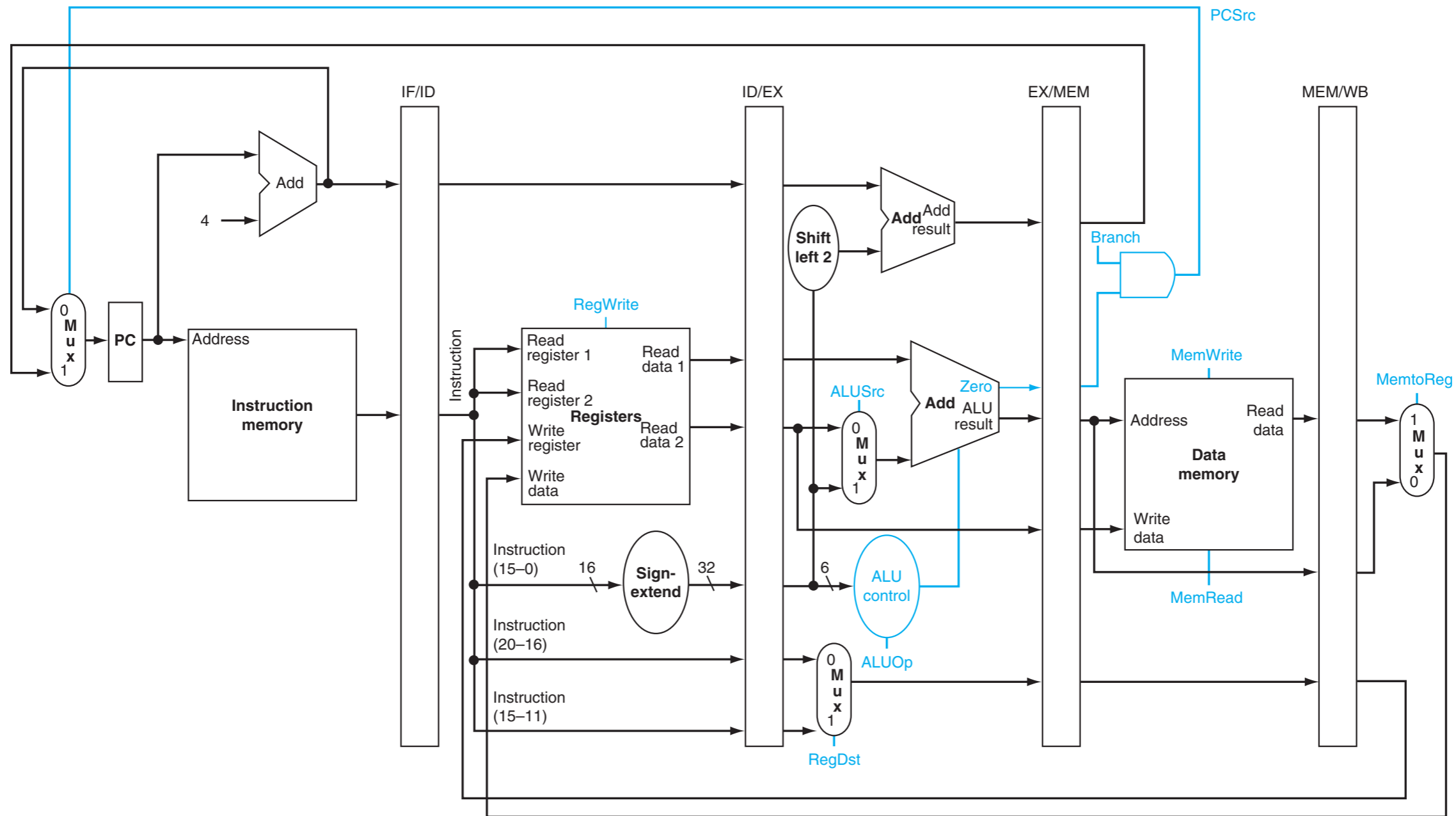


FIGURE 4.46 The pipelined datapath of Figure 4.41 with the control signals identified. This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so these bits must also be included in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged. Copyright © 2009 Elsevier, Inc. All rights reserved.



Pipelined Control Scheme

- As in single-cycle implementation, control signals derived from instruction

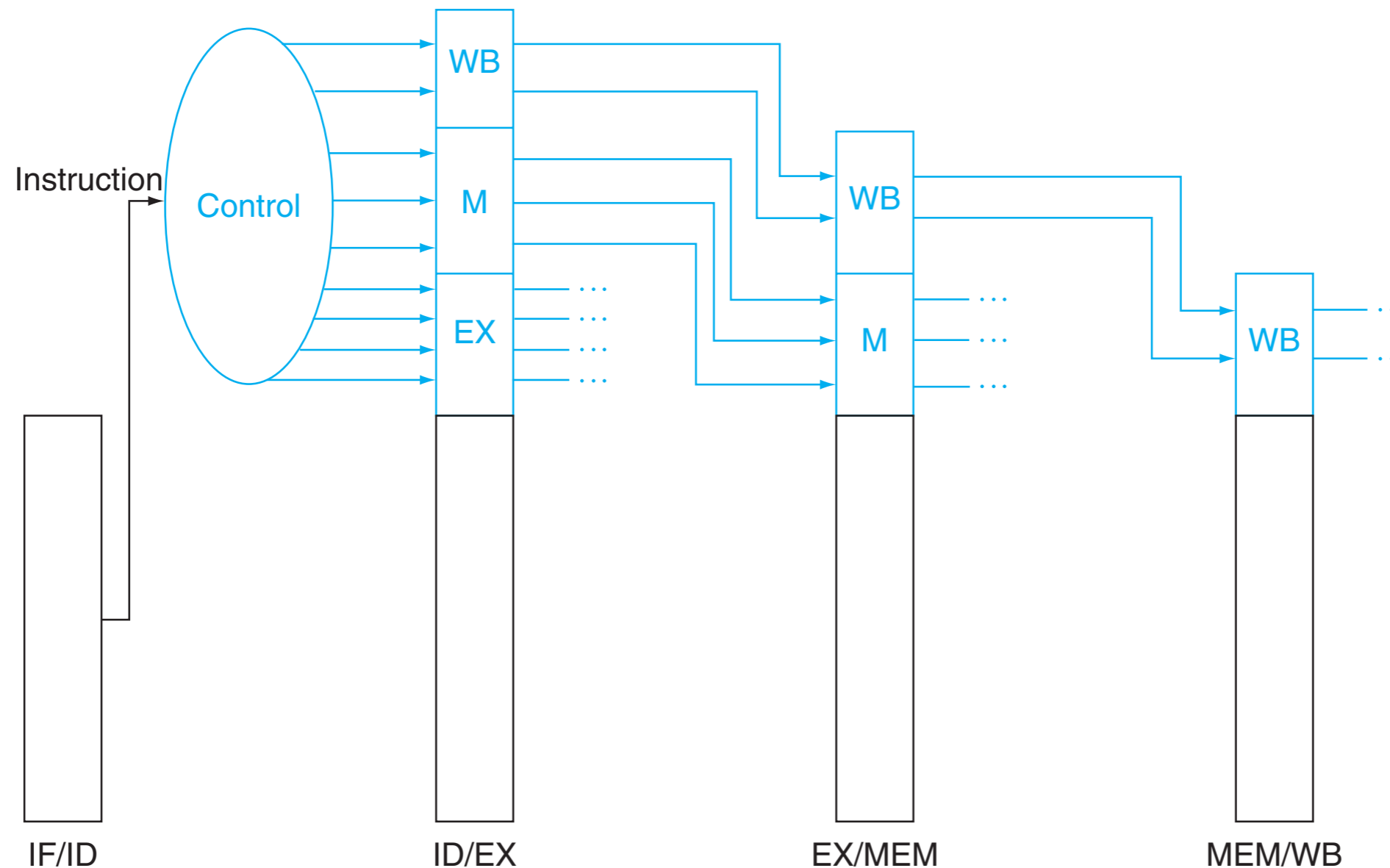


FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage. Copyright © 2009 Elsevier, Inc. All rights reserved.

Pipeline Control Values

- Control signals are conceptually the same as they were in the single cycle CPU.
- ALU Control is the same.
- Main control also unchanged. Table below shows same control signals grouped by pipeline stage

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

FIGURE 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages. Copyright © 2009 Elsevier, Inc. All rights reserved.

Controlled Pipelined CPU

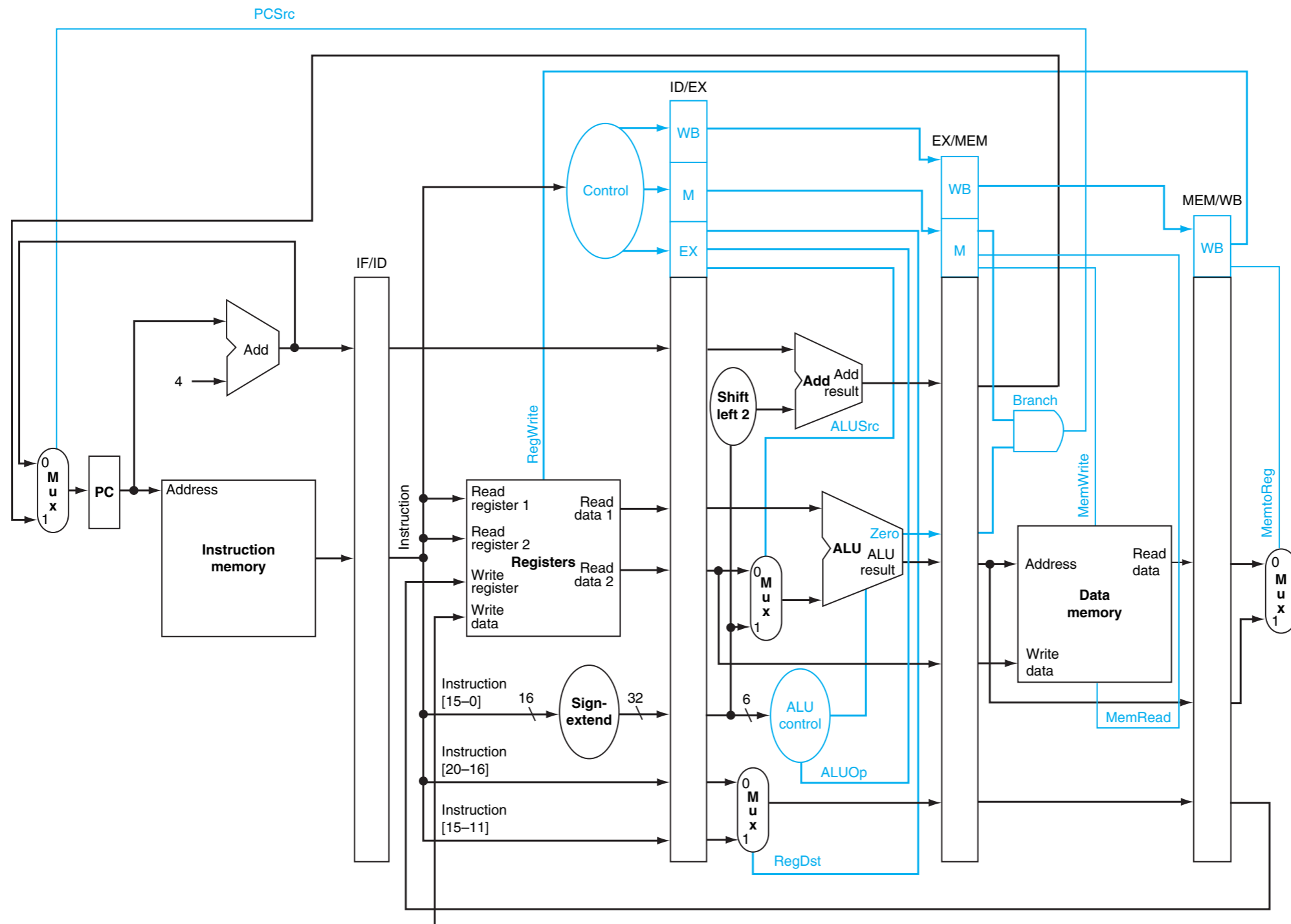


FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage. Copyright © 2009 Elsevier, Inc. All rights reserved.



Hazard

- A hazard is a situation that prevents starting the next instruction in the next cycle
- **Structure hazards** occur when a required resource is busy
- **Data hazards** occur when an instruction needs to wait for an earlier instruction to complete its data write
- **Control hazards** occur when the control action (i.e., next instruction to fetch) depends on a value that is not yet ready

Structure Hazard

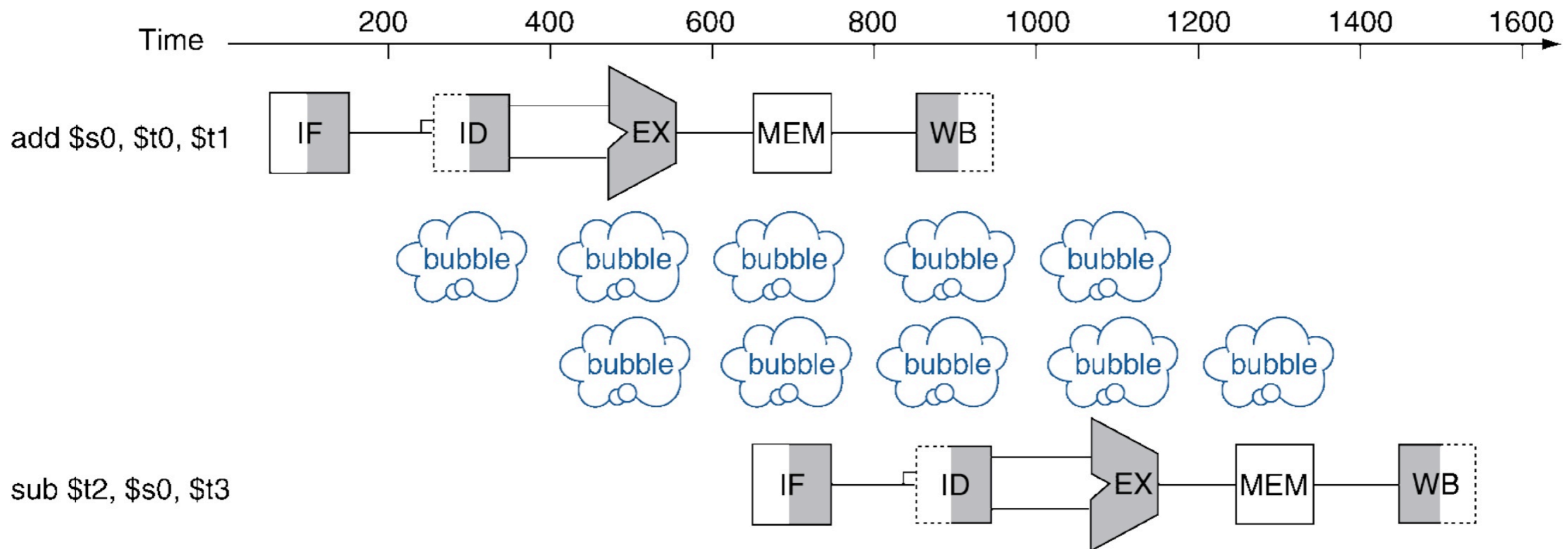
- Conflict for use of a resource
- In a MIPS pipeline with a single memory
 - Load/store requires memory access
 - Instruction fetch would have to **stall** for that cycle
 - This introduces a pipeline **bubble**
- Hence, pipelined datapaths require separate instruction and data memories (or separate instruction and data caches)

Data Hazards

- An instruction depends on completion of data access by a previous instruction

```
add $s0, $t0, $t1  
sub $t2, $s0, $t3
```

\$s0 set during WB phase
\$s0 read during ID phase



Forwarding (aka Bypassing)

- Use result when it is computed
- Don't wait for it to be stored in a register
- Requires extra connections in the datapath

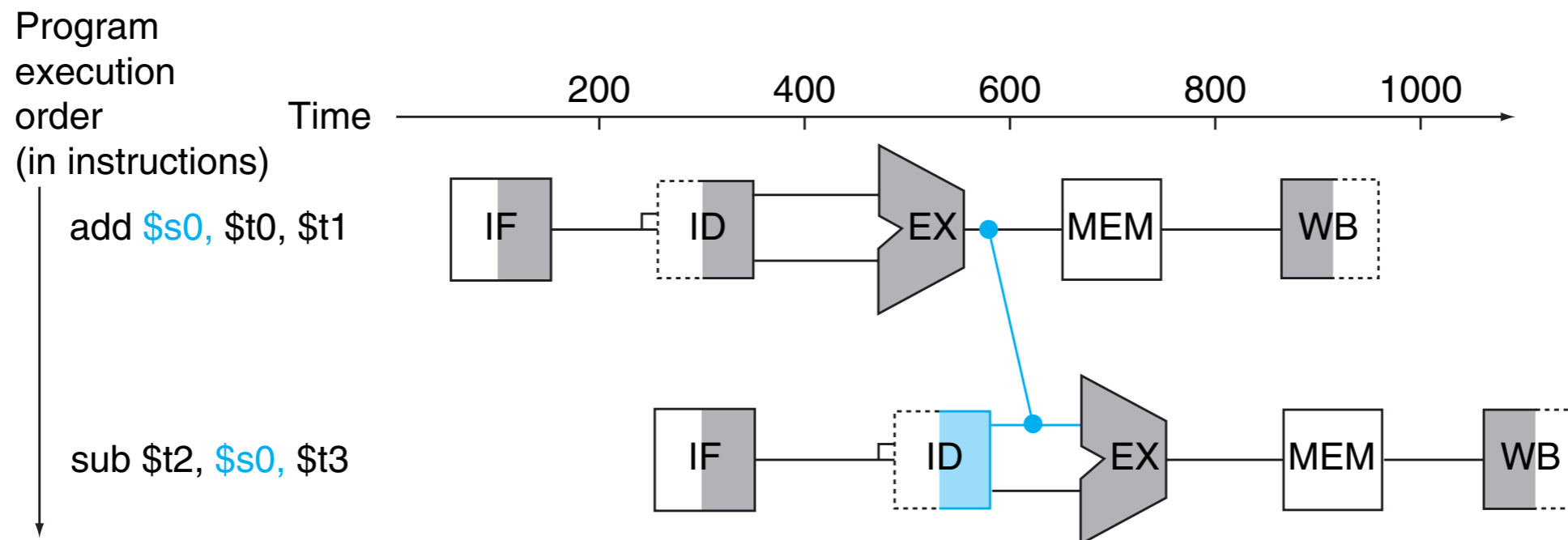


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register \$s0 read in the second stage of sub. Copyright © 2009 Elsevier, Inc. All rights reserved.

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
- If value not computed when needed
- Can't forward backward in time!

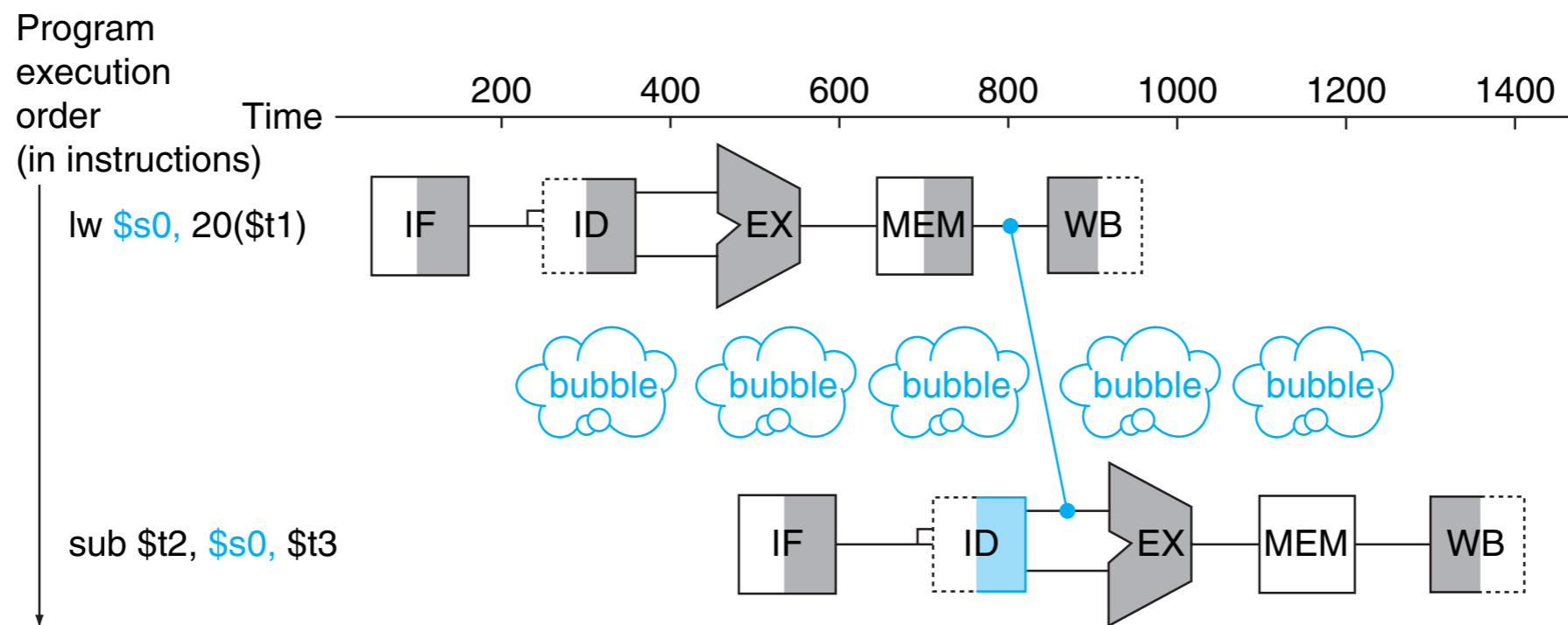


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard. Copyright © 2009 Elsevier, Inc. All rights reserved.

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction

MIPS assembly code for
 $A = B + E; C = B + F;$

stall

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
lw    $t4, 8($t0)
stall
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

13 cycles

```
lw    $t1, 0($t0)
lw    $t2, 4($t0)
lw    $t4, 8($t0)
add   $t3, $t1, $t2
sw    $t3, 12($t0)
add   $t5, $t1, $t4
sw    $t5, 16($t0)
```

11 cycles

Control Hazards

- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- In MIPS pipeline
 - Need to compare registers and compute target early in the pipeline
 - Add hardware to do it in ID stage (See Sec. 4.8)

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

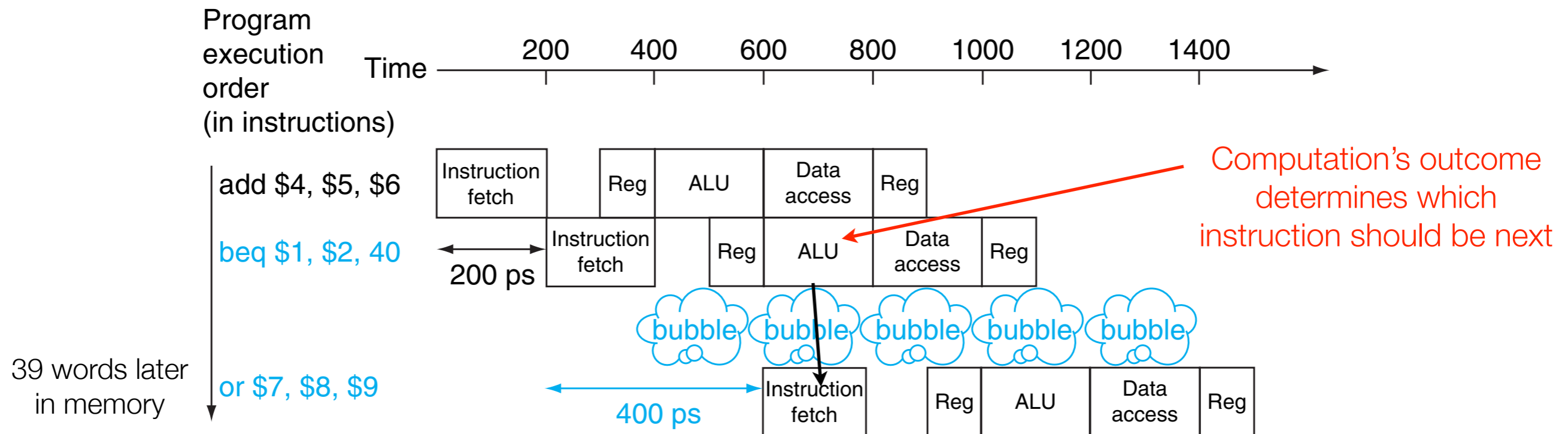


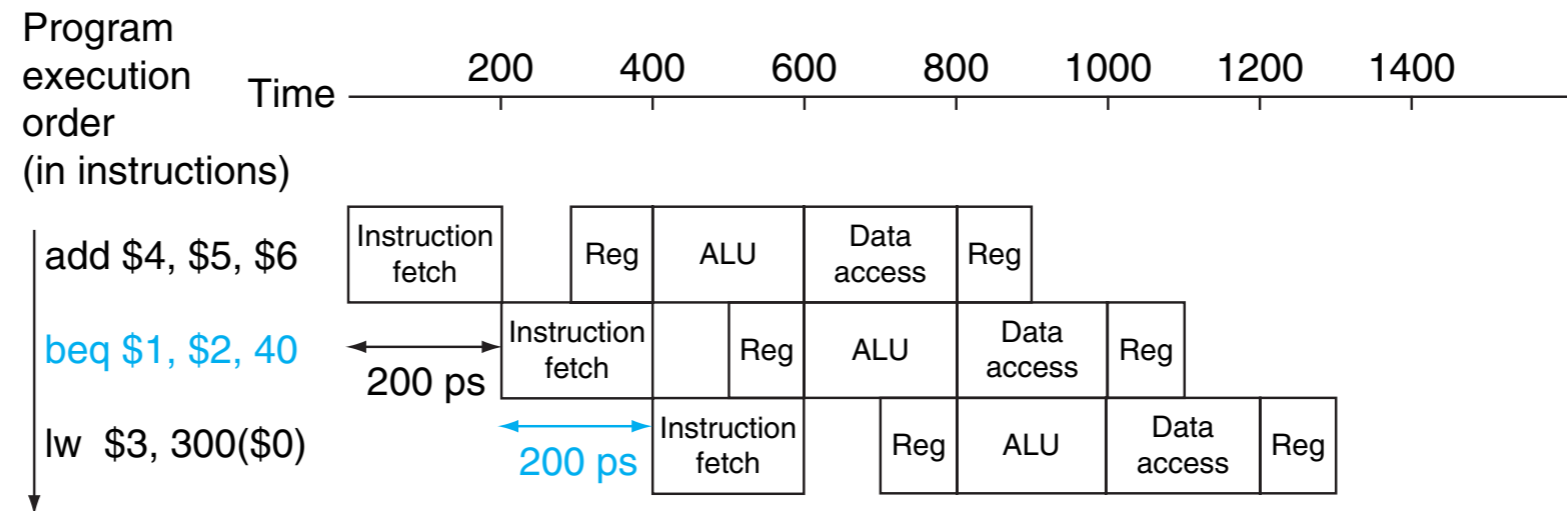
FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. The effect on performance, however, is the same as would occur if a bubble were inserted. Copyright © 2009 Elsevier, Inc. All rights reserved.

Branch Prediction

- Longer pipelines can't readily determine branch outcome early
 - Stall penalty becomes unacceptable
- Predict outcome of branch
 - Only stall if prediction is wrong
- In MIPS pipeline
 - Can predict branches not taken
 - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

prediction correct



prediction incorrect

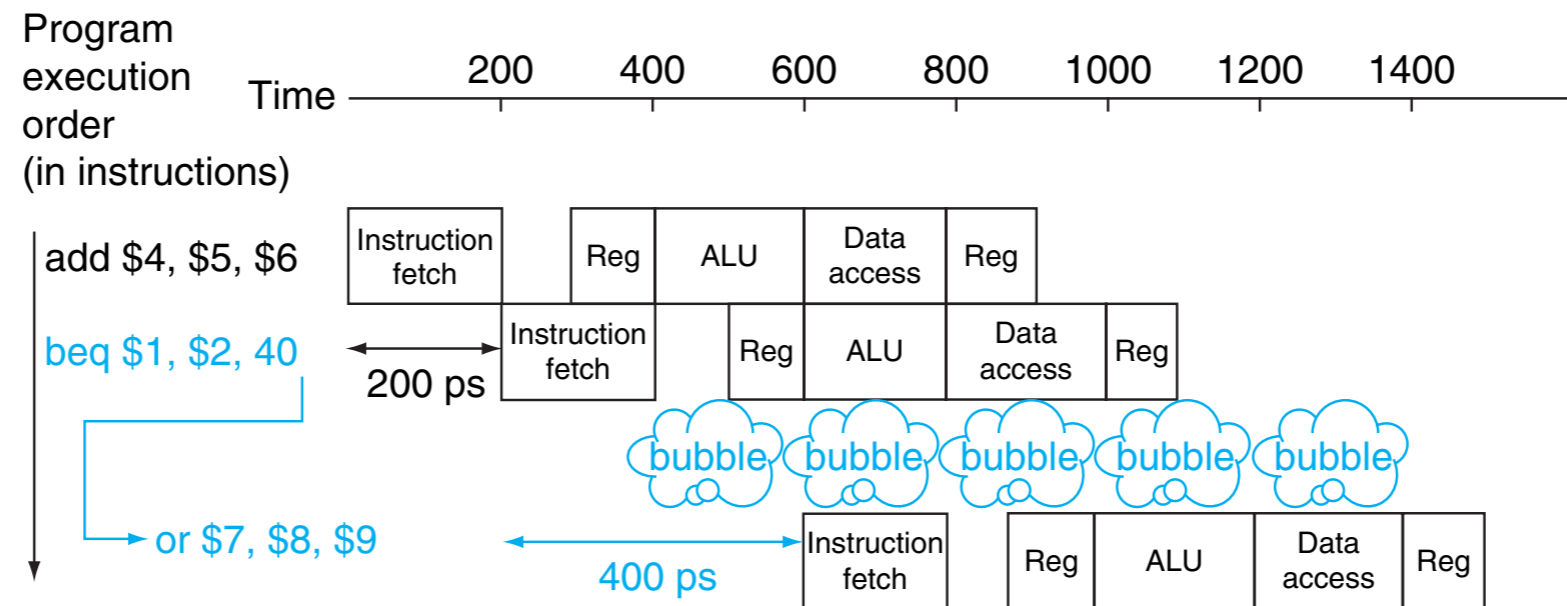


FIGURE 4.32 Predicting that branches are not taken as a solution to control hazard. The top drawing shows the pipeline when the branch is not taken. The bottom drawing shows the pipeline when the branch is taken. As we noted in Figure 4.31, the insertion of a bubble in this fashion simplifies what actually happens, at least during the first clock cycle immediately following the branch. Section 4.8 will reveal the details. Copyright © 2009 Elsevier, Inc. All rights reserved.



More-Realistic Branch Prediction

- Static branch prediction
 - Based on typical branch behavior
 - Example: loop and if-statement branches - default predictions:
 - backward branches taken (while / for loops run multiple iterations)
 - forward branches not taken (if/else usually “if”, not “else”)
- Dynamic branch prediction
 - Hardware measures actual branch behavior
 - e.g., record recent history of each branch
 - Assume future behavior will continue the trend
 - When wrong, stall while re-fetching, and update history

Pipeline Summary

- Pipelining improves performance by increasing instruction throughput
 - Executes multiple instructions in parallel
 - Each instruction has the same latency
- Subject to hazards
 - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

Data Hazards in ALU Instructions

- Consider this instruction sequence:

```
sub  $2, $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100($2)
```

- We can resolve hazards with forwarding
- How do we detect when to forward?

Dependencies & Forwarding

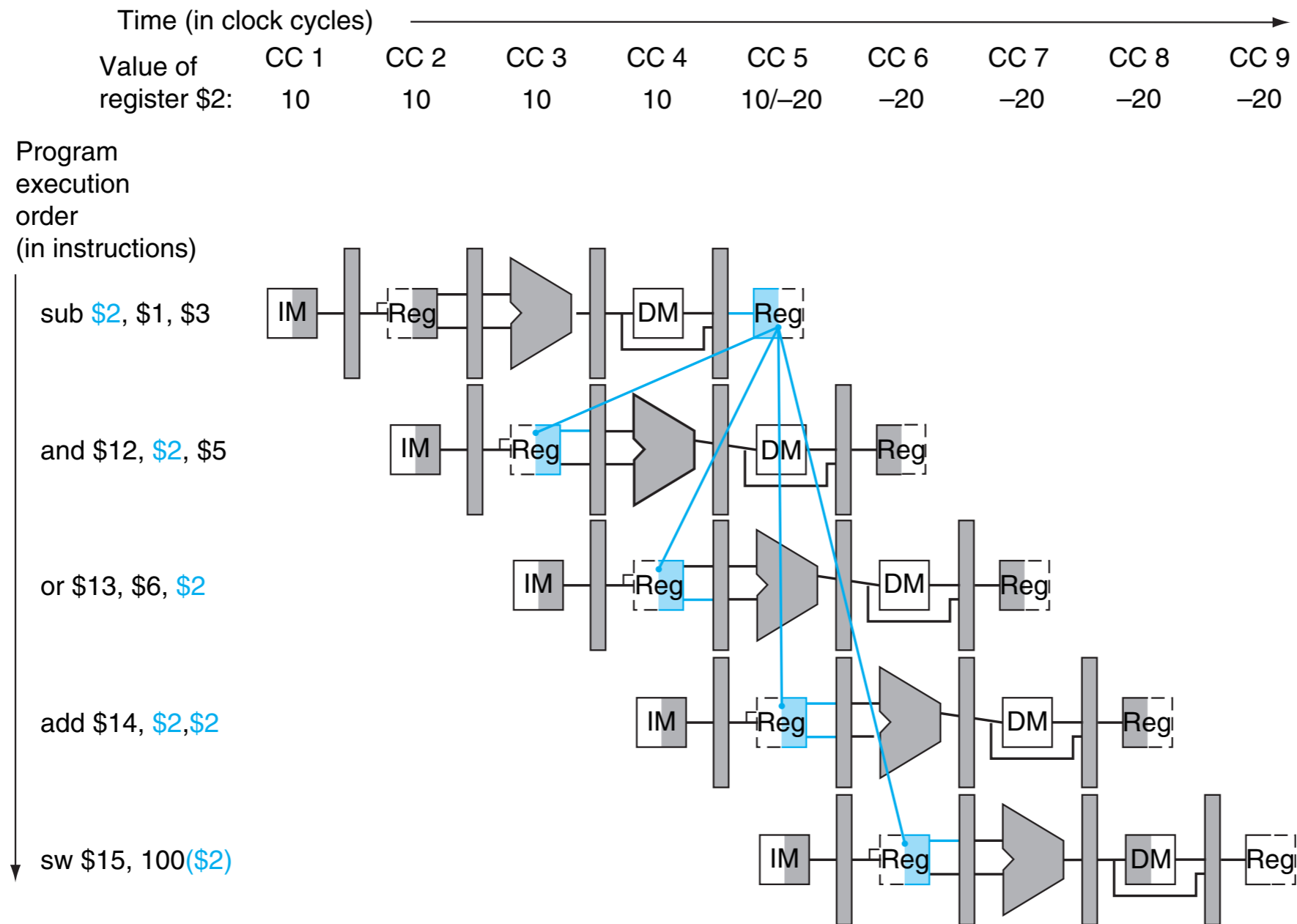


FIGURE 4.52 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences.

All the dependent actions are shown in color, and “CC 1” at the top of the figure means clock cycle 1. The first instruction writes into \$2, and all the following instructions read \$2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards*. Copyright © 2009 Elsevier, Inc. All rights reserved.



Detecting the Need to Forward

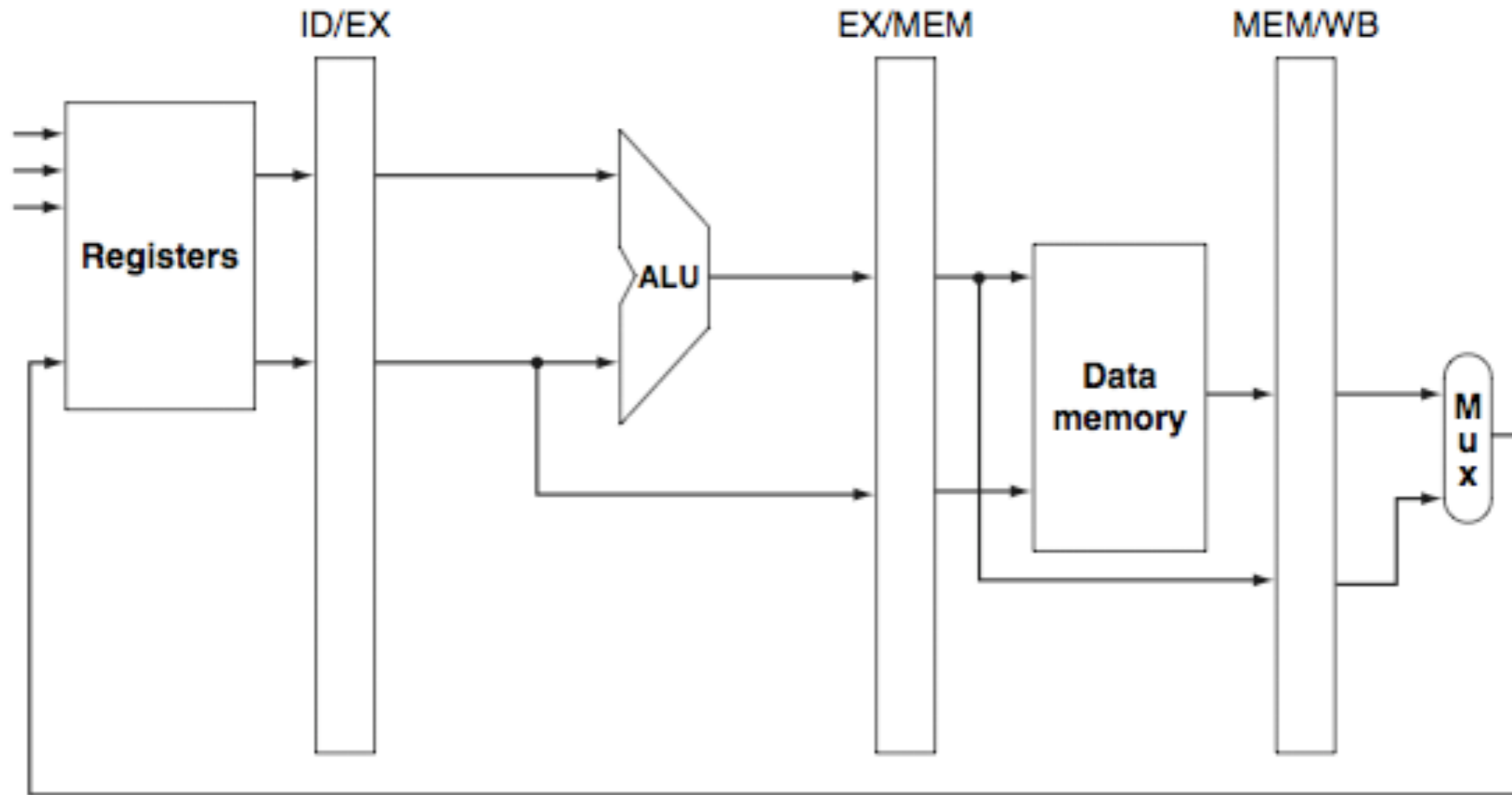
- Pass register numbers along pipeline: `Stage.RegIdentifier`
 - e.g., `ID/EX.RegisterRs` = register number for `Rs` sitting in ID/EX pipeline register
 - ALU operand register numbers in EX stage are given by `ID/EX.RegisterRs`, `ID/EX.RegisterRt`
- Data hazards when
 - (e.g., `add Rd, Rs, Rt`)
 - 1a. `EX/MEM.RegisterRd` = `ID/EX.RegisterRs`
 - 1b. `EX/MEM.RegisterRd` = `ID/EX.RegisterRt`(e.g., `lw Rs, C(Rt)`)
 - 2a. `MEM/WB.RegisterRd` = `ID/EX.RegisterRs`
 - 2b. `MEM/WB.RegisterRd` = `ID/EX.RegisterRt`

Fwd from
EX/MEM
pipeline reg

Fwd from
MEM/WB
pipeline reg

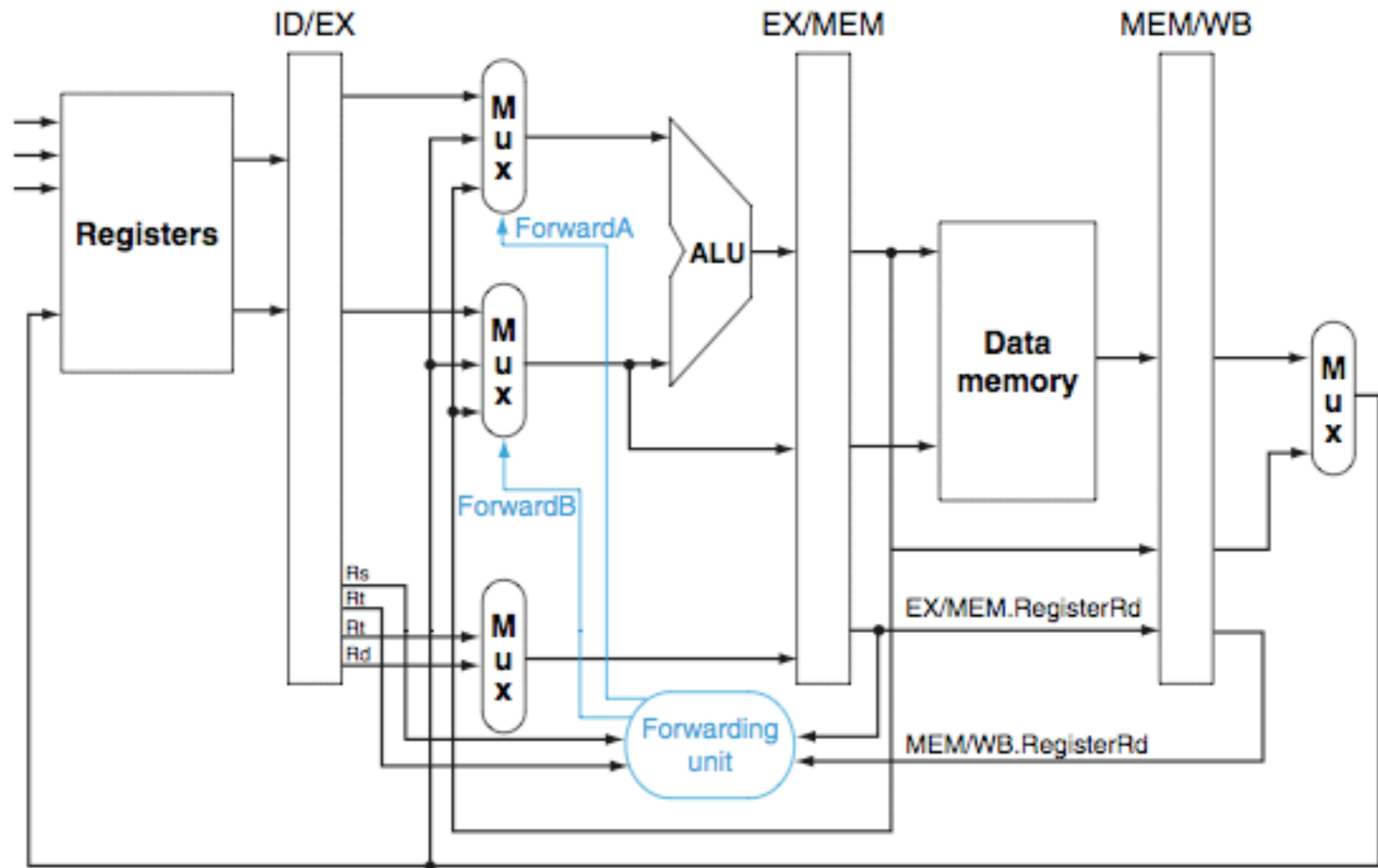


Simplified Pipeline w. No Forwarding



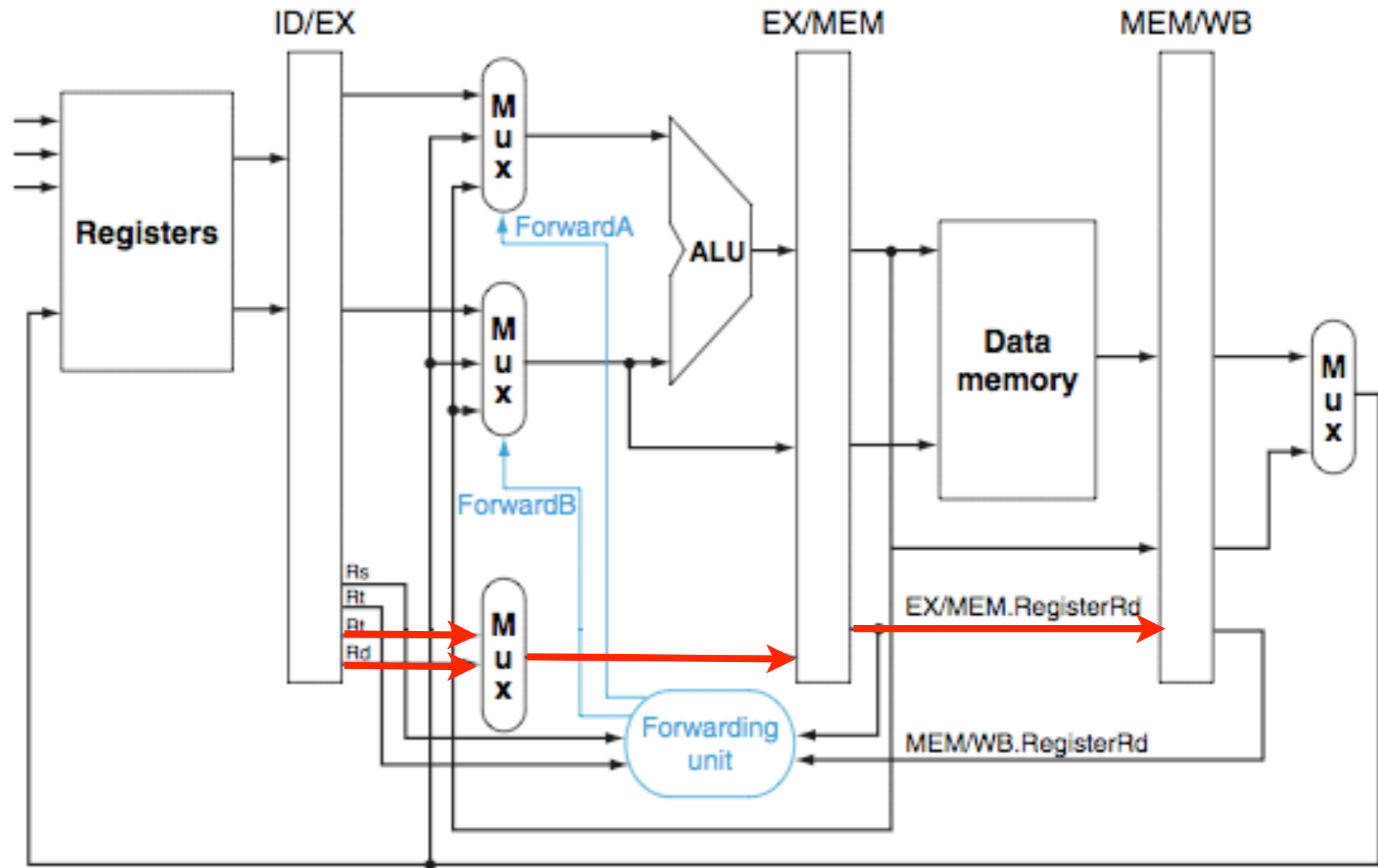
a. No forwarding

Simplified Pipeline w. Forwarding Paths



b. With forwarding

Simplified Pipeline w. Forwarding Paths 1

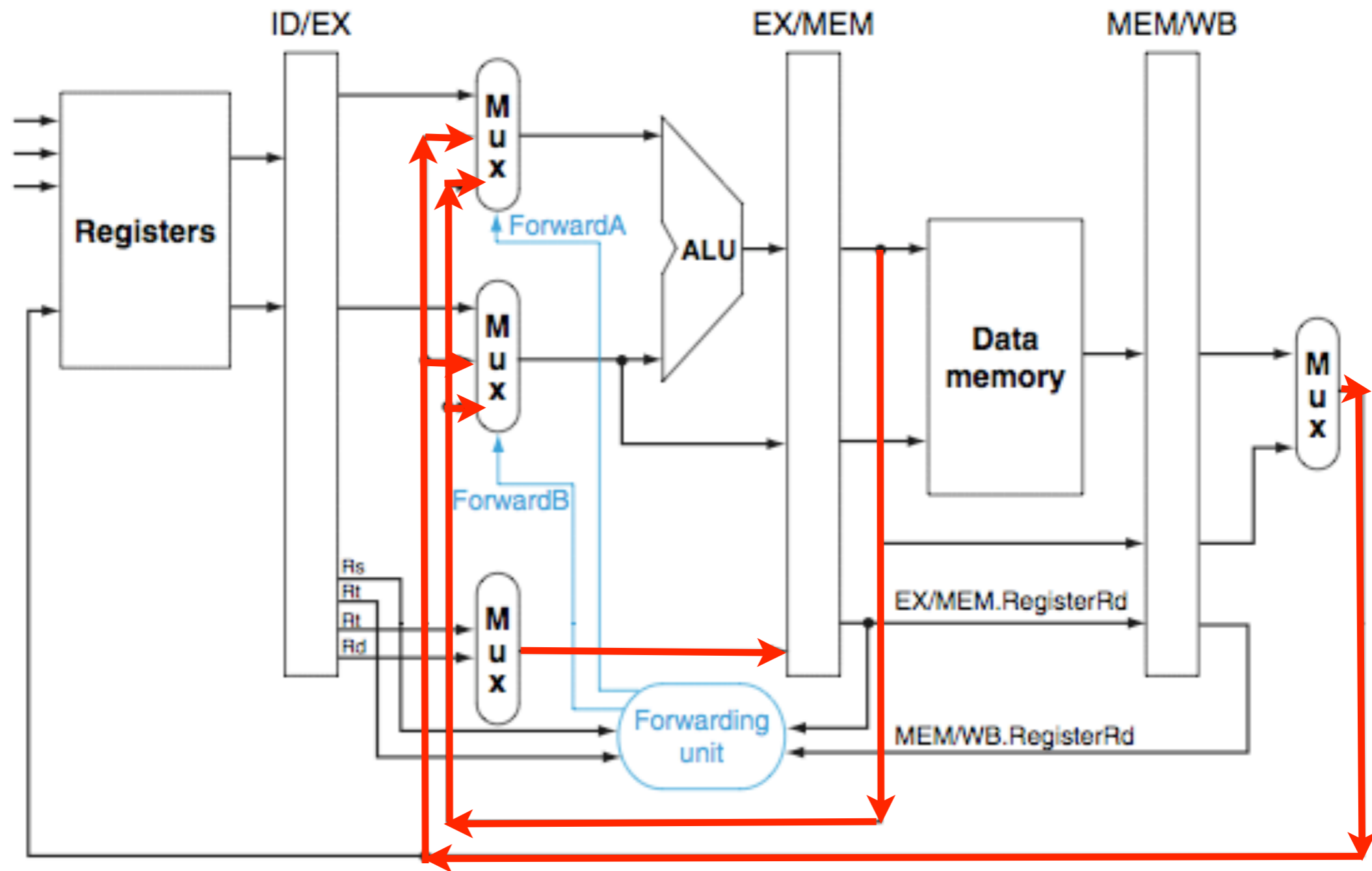


b. With forwarding

keep track of register sources/targets for in-flight instructions



Simplified Pipeline w. Forwarding Paths 2

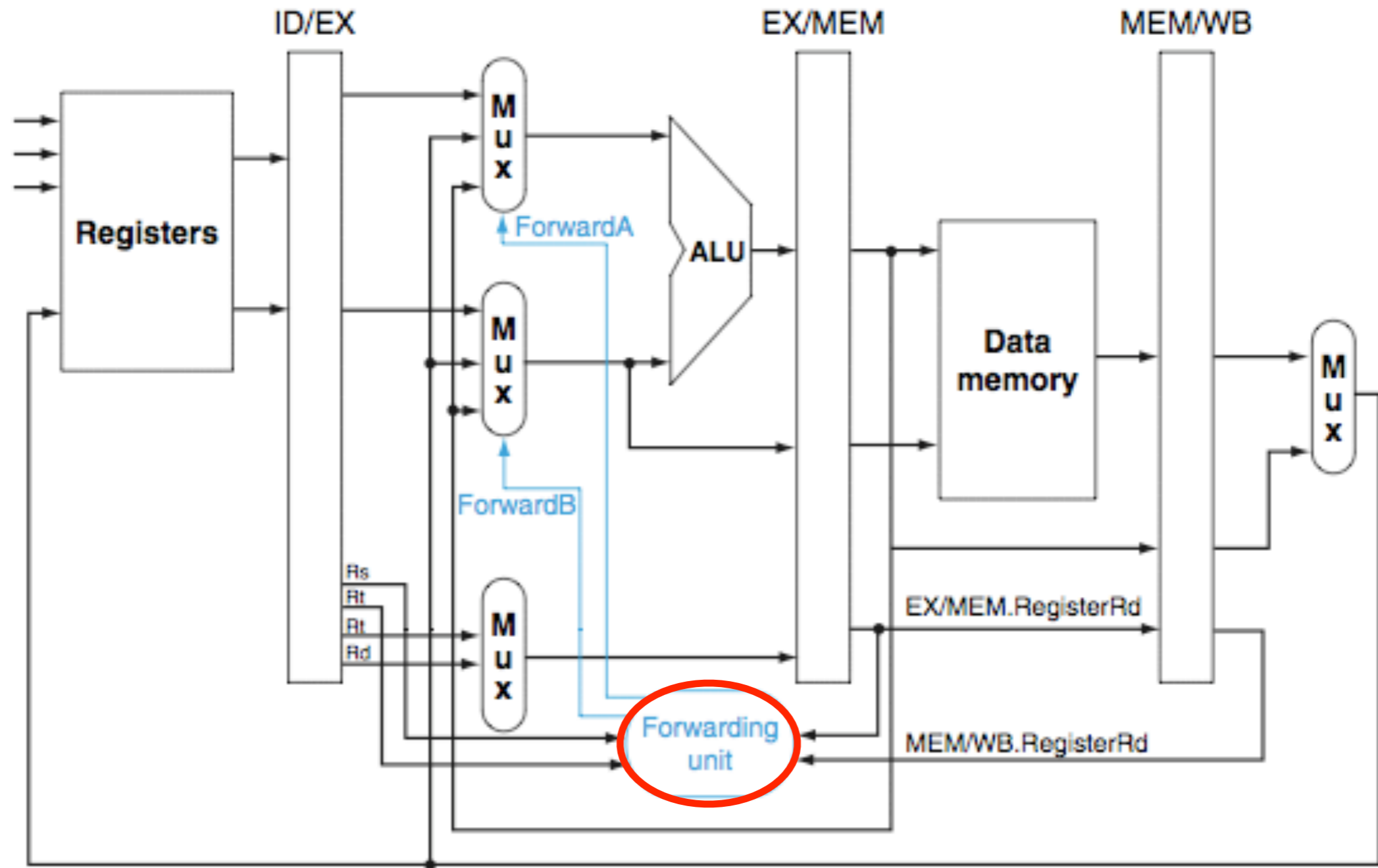


b. With forwarding

option of routing previously calculated values directly to ALU



Simplified Pipeline w. Forwarding Paths 3



b. With forwarding

Operand forwarding (aka register bypass) controlled by forwarding unit



Forwarding Conditions

- EX hazard

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
ForwardA = 10

- if (EX/MEM.RegWrite and (EX/MEM.RegisterRd \neq 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
ForwardB = 10

- MEM hazard

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01

- if (MEM/WB.RegWrite and (MEM/WB.RegisterRd \neq 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01



Double Data Hazard

- Consider the sequence:

add \$1, \$1, \$2

add \$1, \$1, \$3

add \$1, \$1, \$4

- Both hazards occur
 - Want to use the most recent
- Revise MEM hazard condition
 - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard

Condition for
EX hazard on
RegisterRs

- `if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
ForwardA = 01`

Condition for
EX hazard on
RegisterRt

- `if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
ForwardB = 01`



Datapath with Forwarding

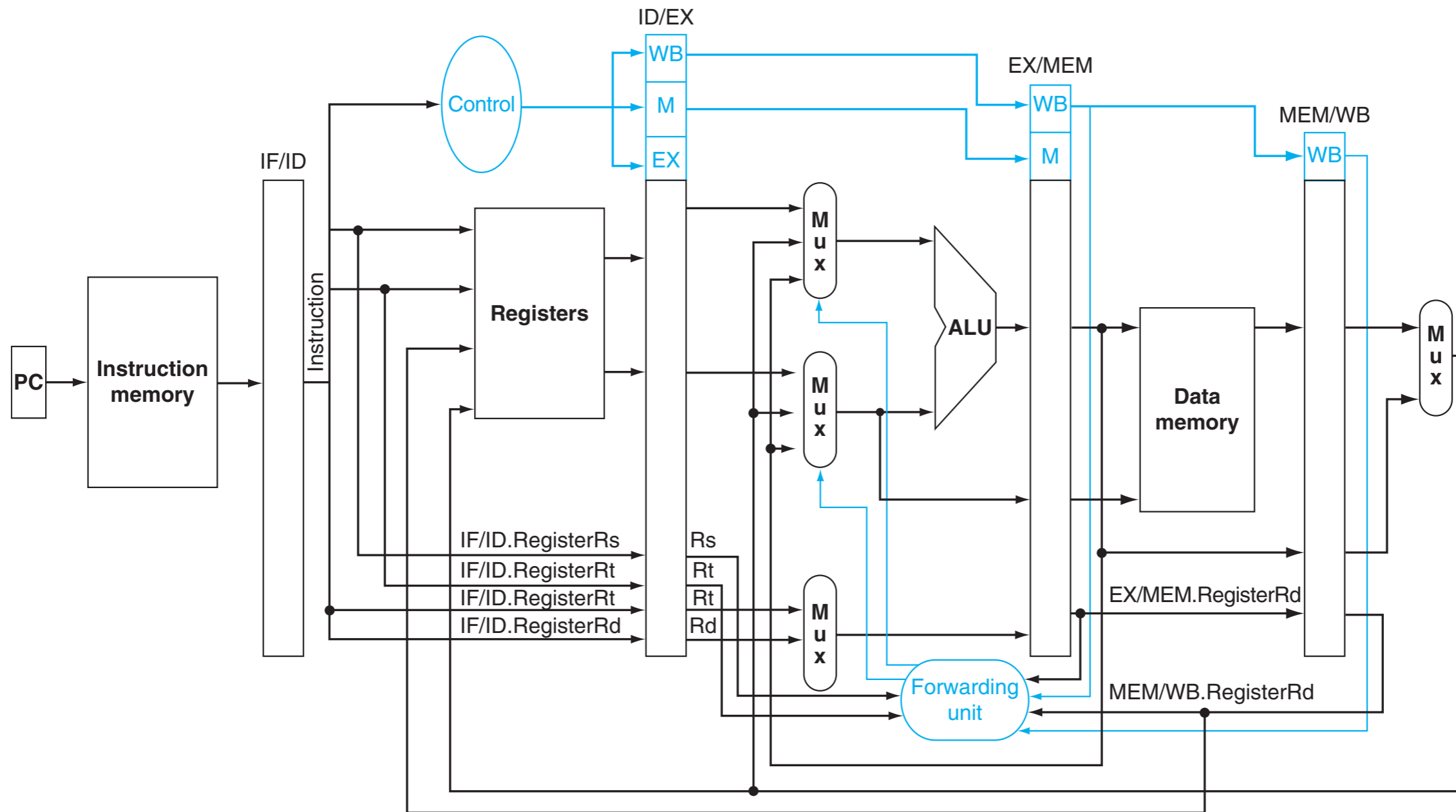


FIGURE 4.56 The datapath modified to resolve hazards via forwarding. Compared with the datapath in Figure 4.51, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware. Copyright © 2009 Elsevier, Inc. All rights reserved.



Load-Use Data Hazard

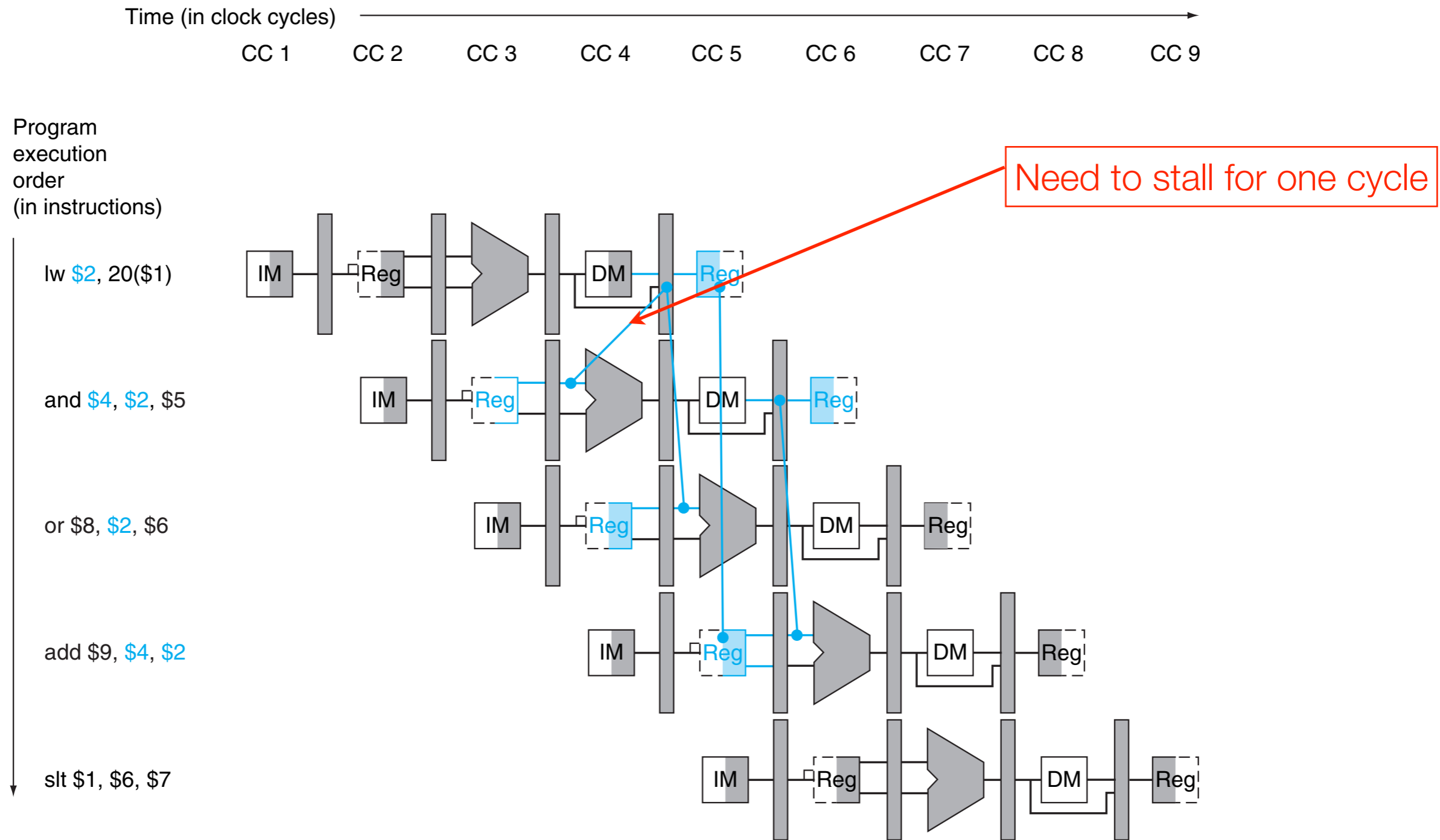


FIGURE 4.58 A pipelined sequence of instructions. Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit. Copyright © 2009 Elsevier, Inc. All rights reserved.



Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by

- IF/ID.RegisterRs, IF/ID.RegisterRt

- Load-use hazard when

```
if ID/EX.MemRead
and ((ID/EX.RegisterRt = IF/ID.RegisterRs)
    or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
```

Reg to write to

- If detected, stall and insert bubble

How to Stall the Pipeline

- Force control values in ID/EX register to 0
- Prevent update of PC and IF/ID register
 - Using instruction is decoded again
 - Following instruction is fetched again
- 1-cycle stall allows MEM to read data for lw
 - Can subsequently forward to EX stage

Stall/Bubble in the Pipeline

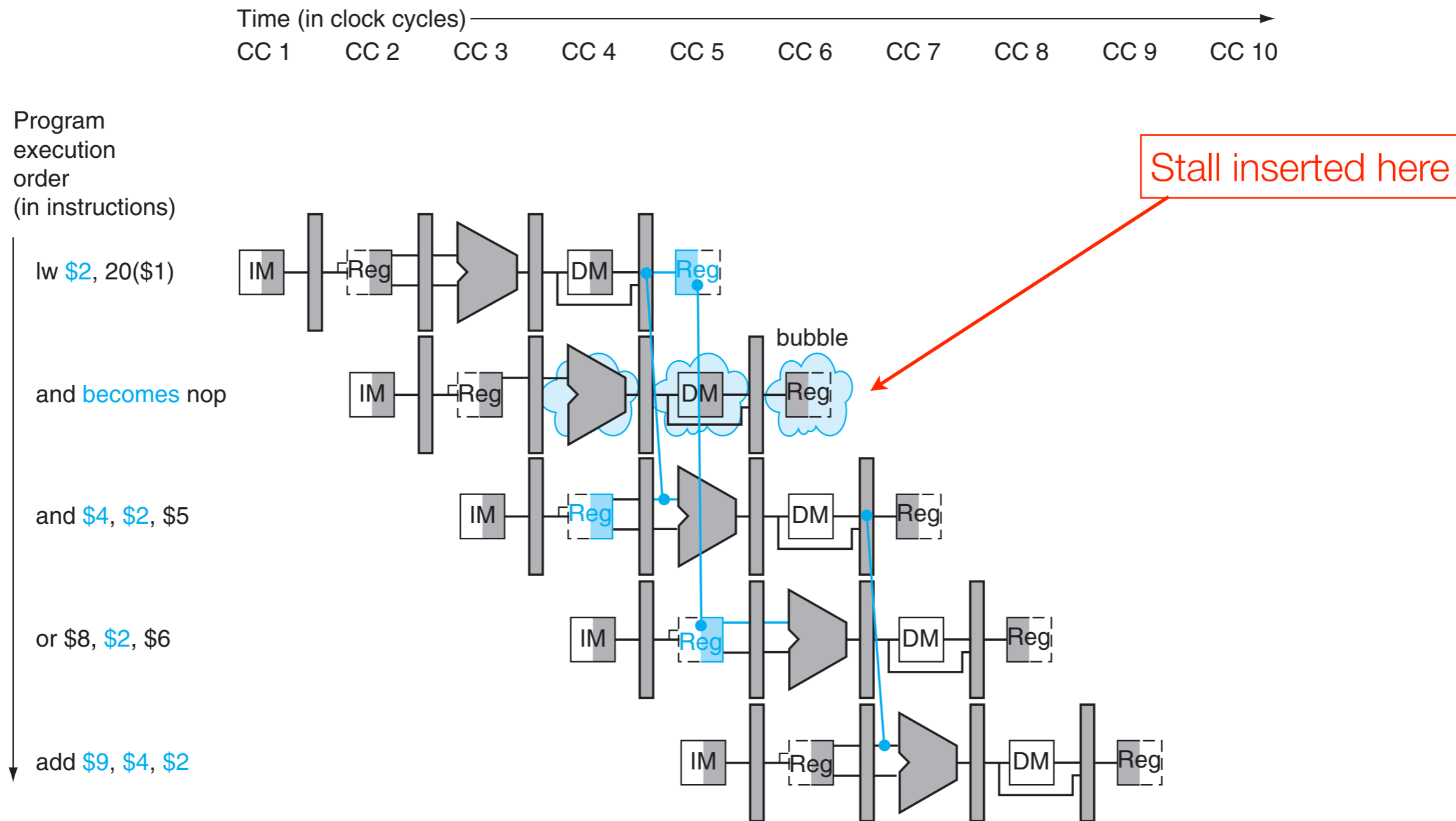


FIGURE 4.59 The way stalls are really inserted into the pipeline. A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop. Note that the and instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise the OR instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur. Copyright © 2009 Elsevier, Inc. All rights reserved.



Datapath with Hazard Detection

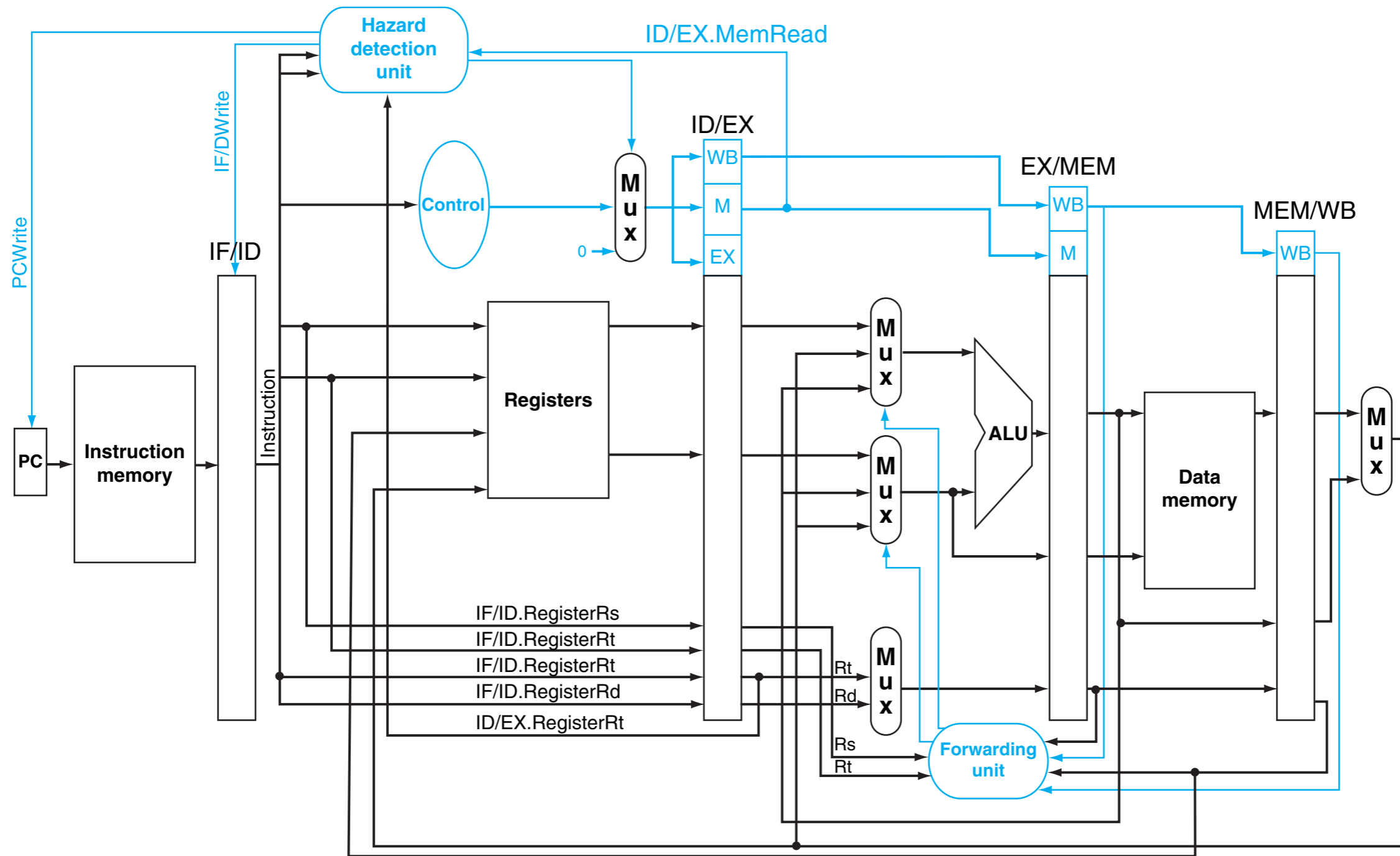


FIGURE 4.60 Pipelined control overview, showing the two multiplexers for forwarding, the hazard detection unit, and the forwarding unit. Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements. Copyright © 2009 Elsevier, Inc. All rights reserved.



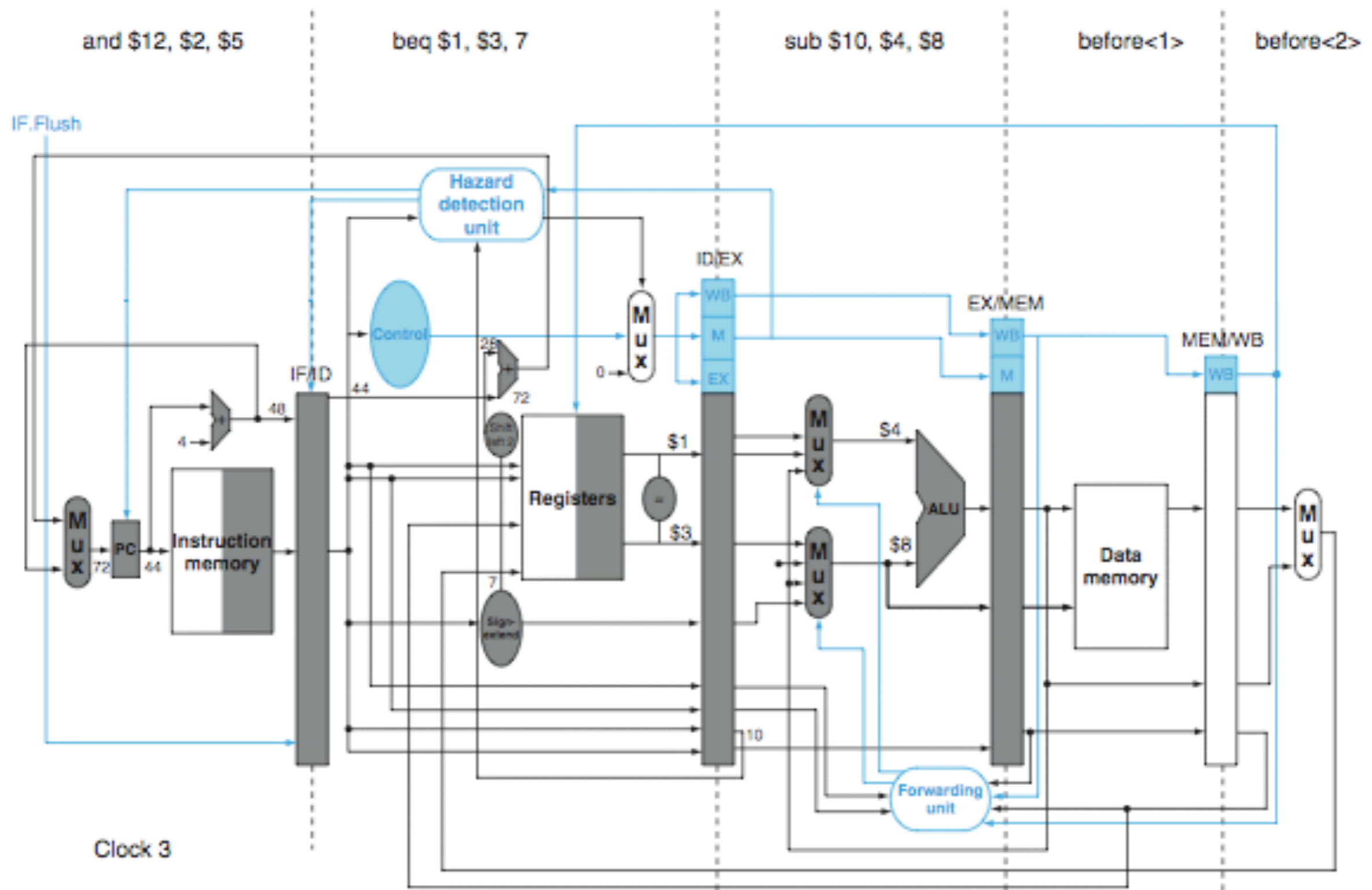
Stalls and Performance

- Stalls reduce performance
 - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
 - Requires knowledge of the pipeline structure

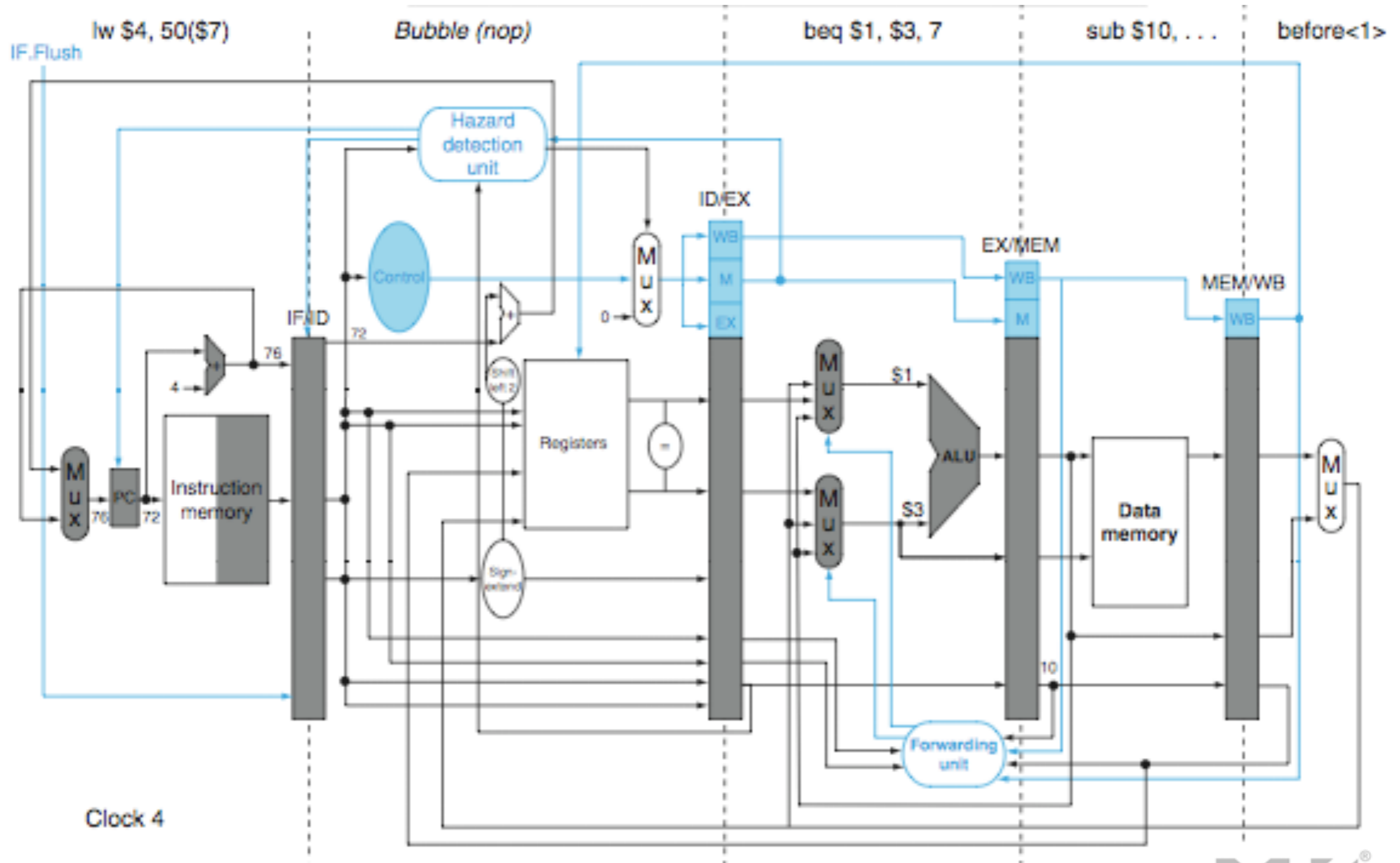
Branch Hazards

- Determine branch outcome and target as early as possible
- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator

Branch Taken 1



Branch Taken 2



Clock 4



Data Hazards for Branches 1

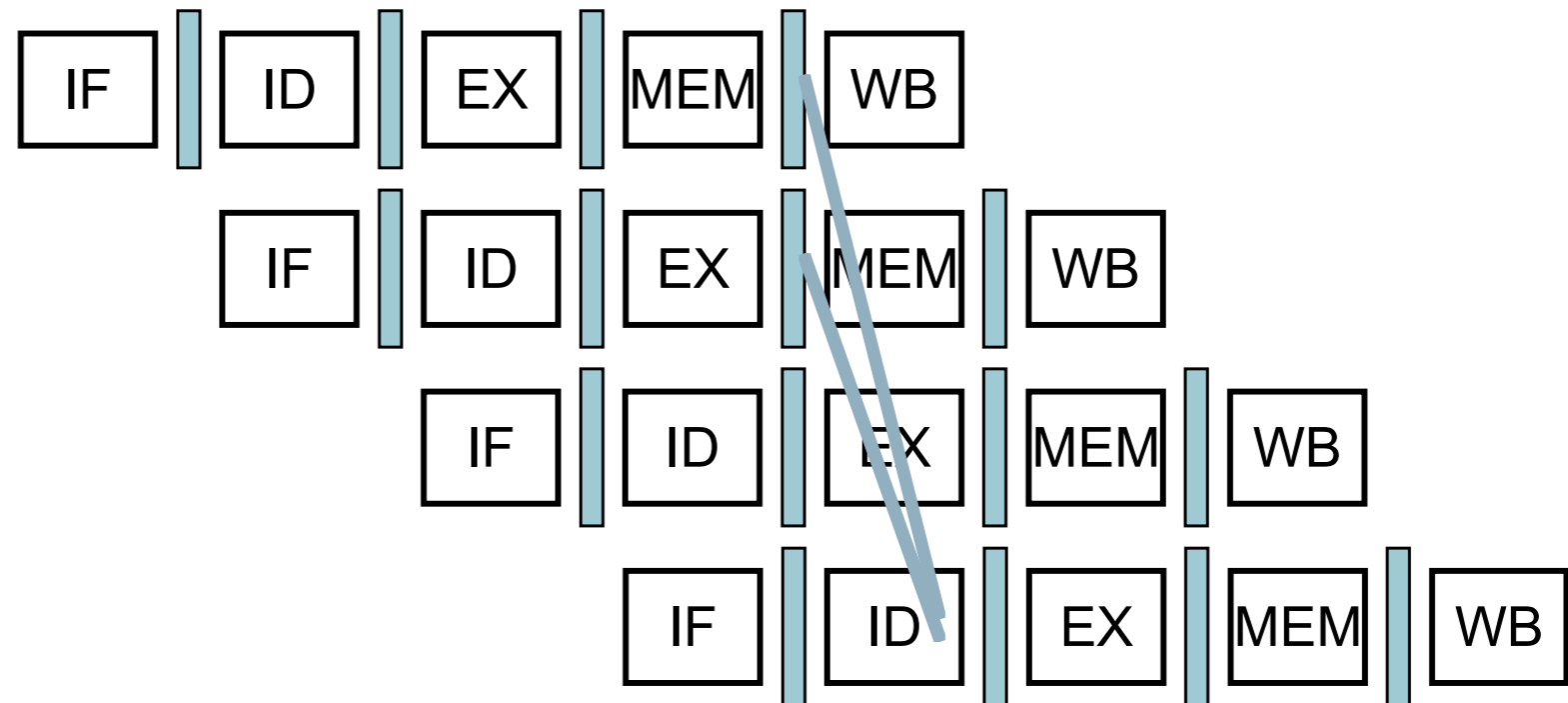
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction → can resolve using forwarding

add \$1, \$2, \$3

add \$4, \$5, \$6

...

beq \$1, \$4, target



Data Hazards for Branches 2

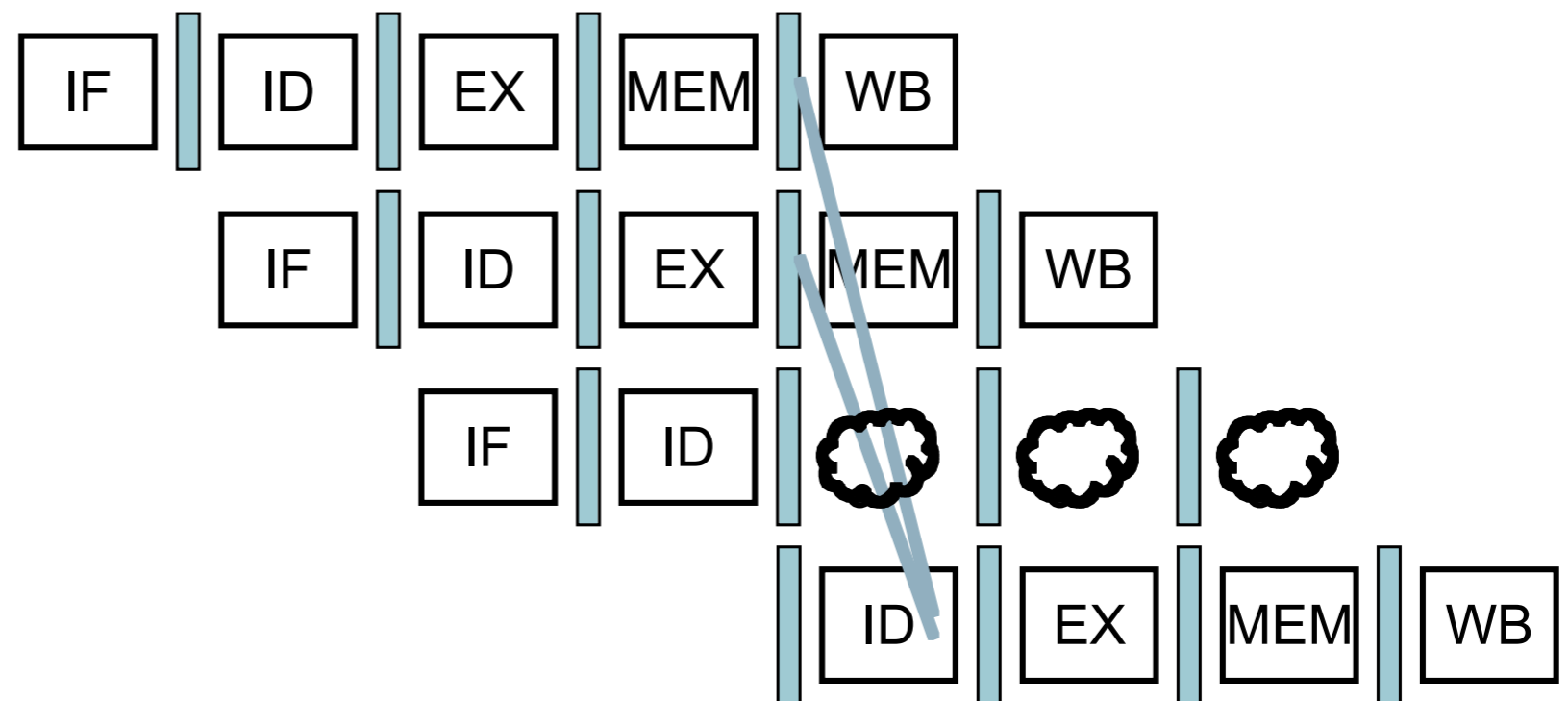
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction → need 1 stall cycle

lw \$1, addr

add \$4, \$5, \$6

beq stalled

beq \$1, \$4, target



Data Hazards for Branches 3

- If a comparison register is a destination of immediately preceding load instruction → need 2 stall cycles

lw \$1, addr

beq stalled

beq stalled

beq \$1, \$0, target

