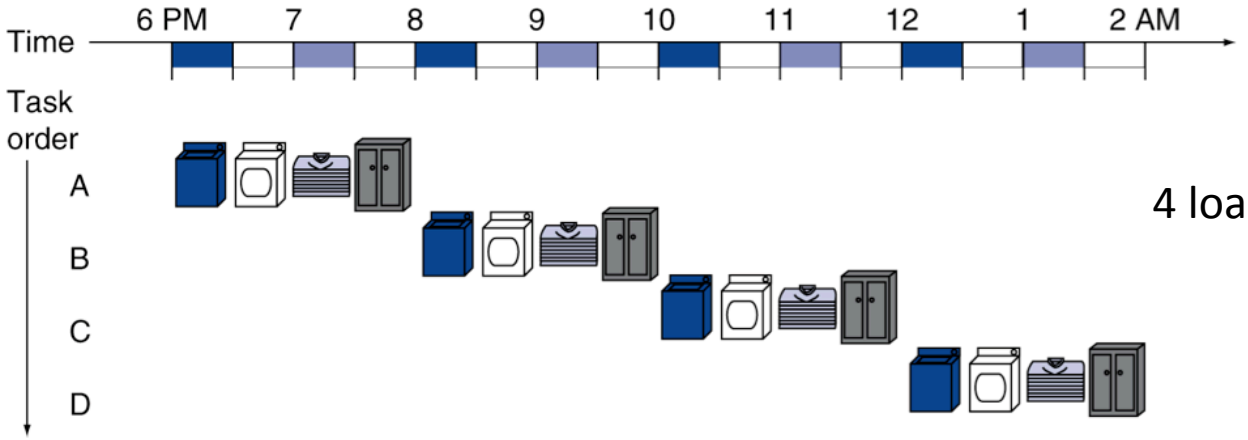
Pipelining Analogy

Doing multiple loads of laundry

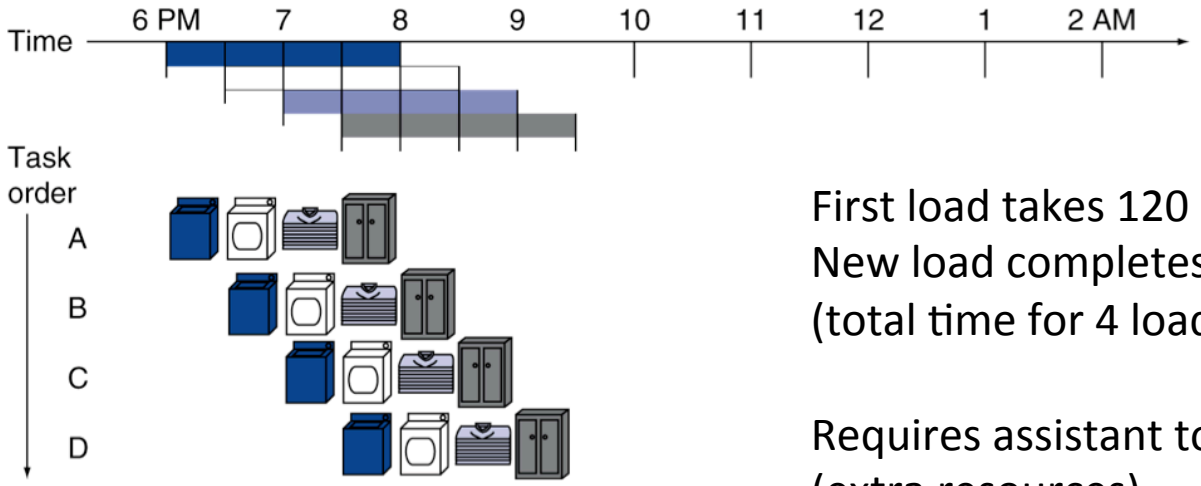


4 loads take 4 *120 = 480 minutes

1. Use washer for next dirty load of clothes (30 min.)
2. When done, use dryer to dry clothes (30 min.)
3. When done, fold the clean dry clothes (30 min)
4. When done, put the clothes away (30 min)

Pipelined Laundry Analogy

Parallelism improves performance by overlapping

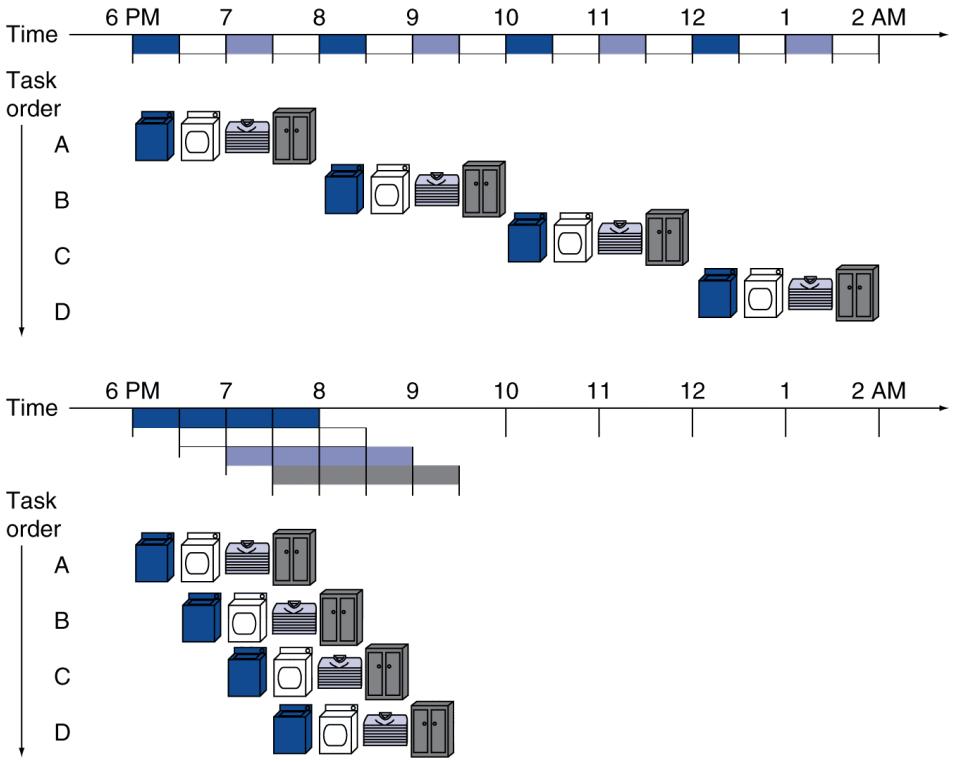


First load takes 120 minutes
New load completes every 30 min thereafter
(total time for 4 loads = 120 + 90 = 210 min)

Requires assistant to work in parallel
(extra resources)

1. Wash 1st load
2. When done place 1st into dryer and wash 2nd
3. When done, fold 1st load, dry 2nd and wash 3rd
4. When done, put away 1st, fold 2rd, dry 3rd and wash 4th

Pipelining improves throughput



- Four loads:
  - Speedup
    = 480/210 = 2.3
- Non-stop:
  - Speedup
    = 120n/(120+30n) ≈ 4
    = number of stages

Each load still takes 120 minutes (2 hours)

But more loads are completed per hour

MIPS Pipelining

Five stages, one step per stage

1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register

MIPS ISA & Pipelining

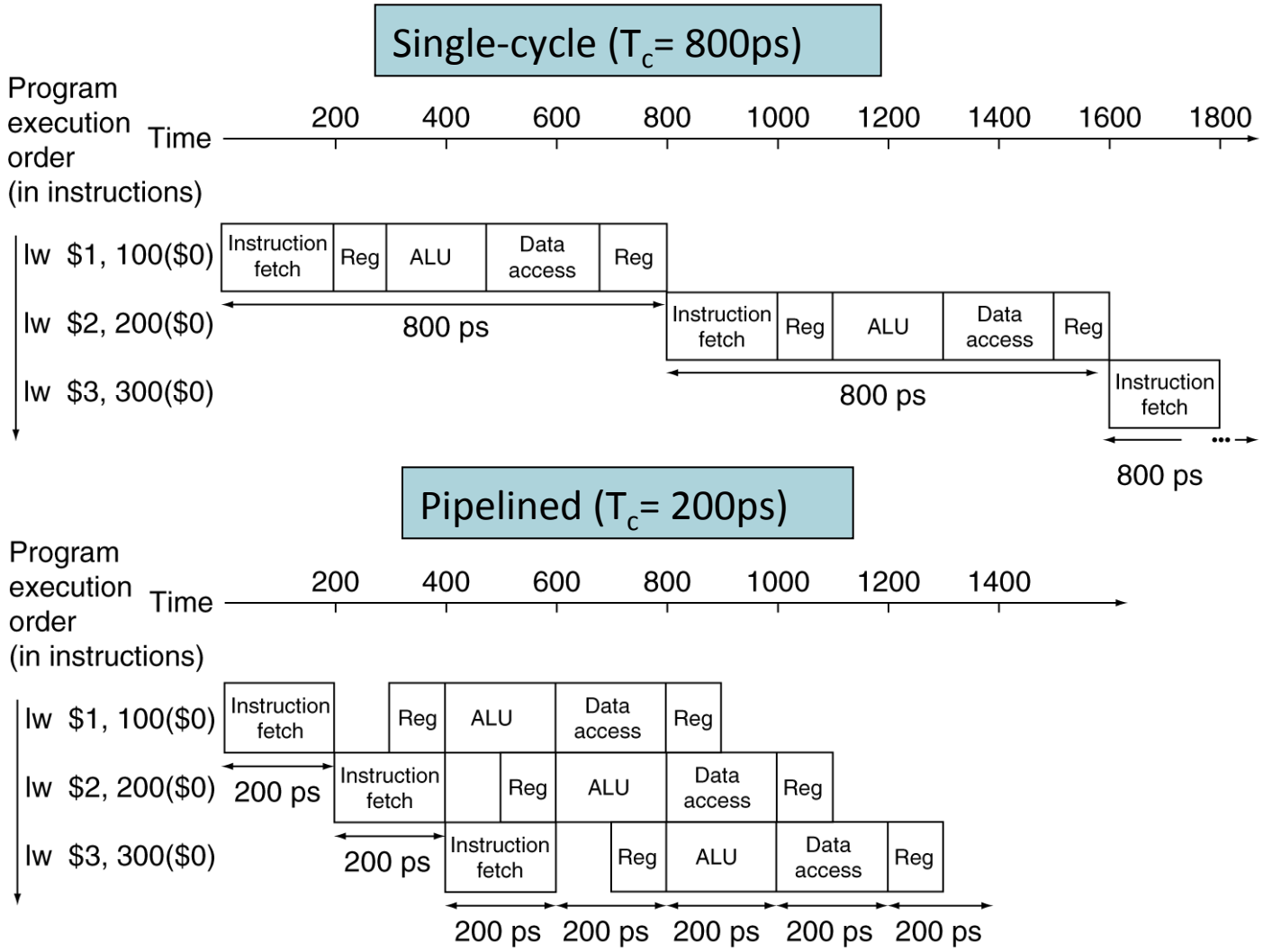
- All instructions are 32-bits
  - Easier to fetch and decode in one cycle
  - c.f. x86: 1- to 17-byte instructions
- Few and regular instruction formats
  - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle

MIPS Pipelining

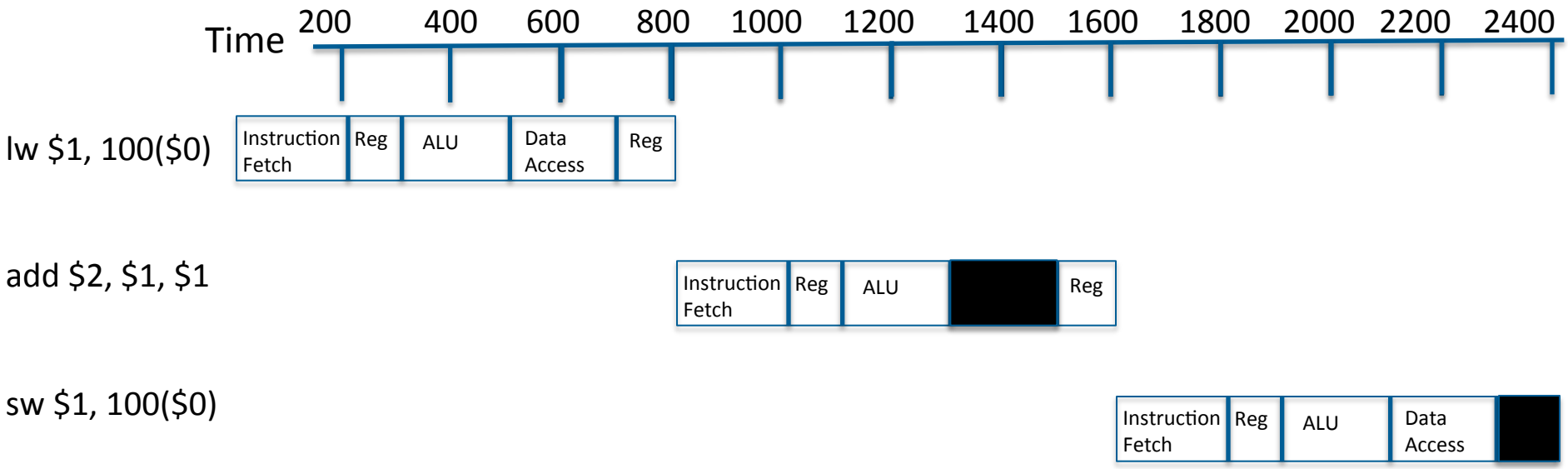
- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

MIPS Pipelining



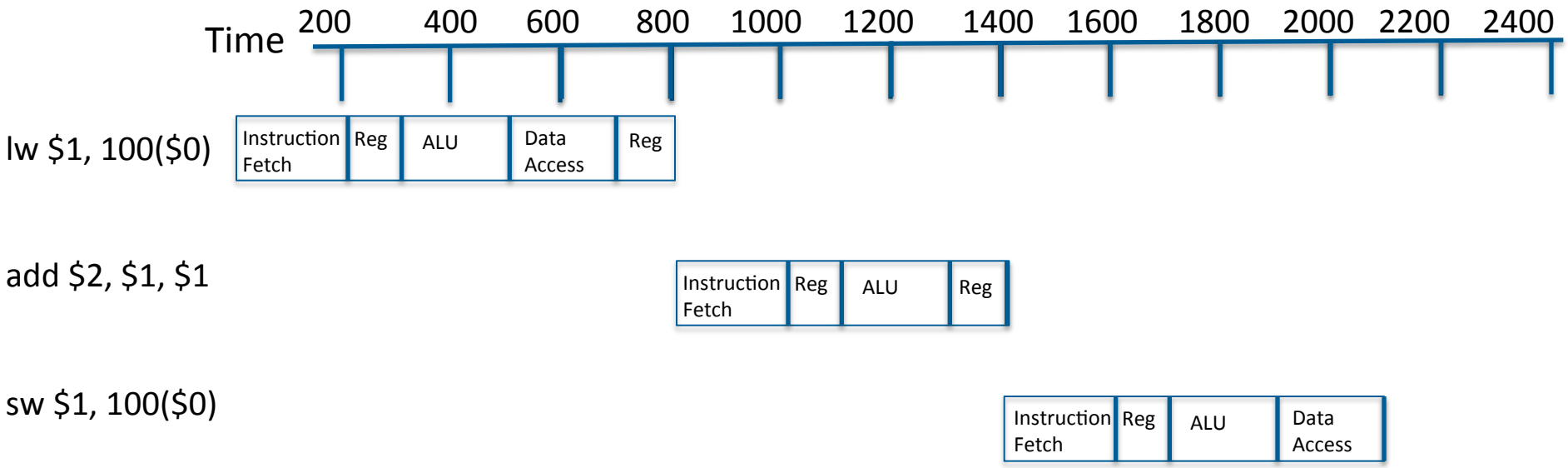
Single-Cycle Performance



Single-cycle (T_c= 800ps)
T_{total} = 2400 ps

Cycle time matches longest instruction

Multi-Cycle Performance



Multi-cycle with variable cycle time

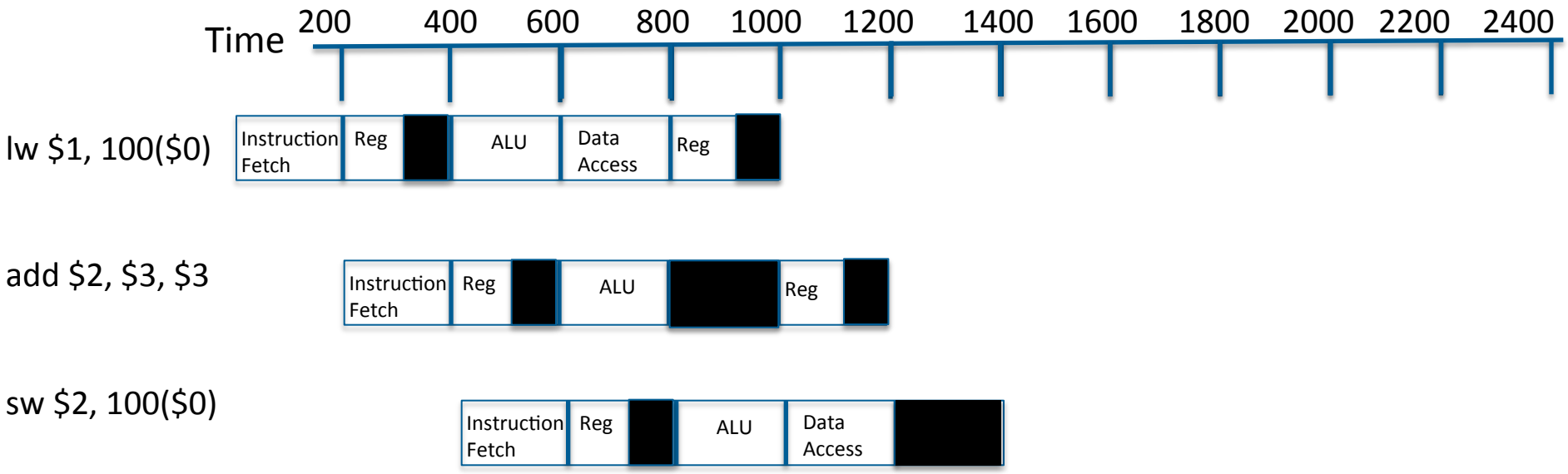
T_{total} = 800 + 600 + 700 =2100 ps

Multi-cycle with fixed cycle time (200ps)

T_{total} = 1000 + 800 + 800 =2600 ps

Cycle time matches longest step

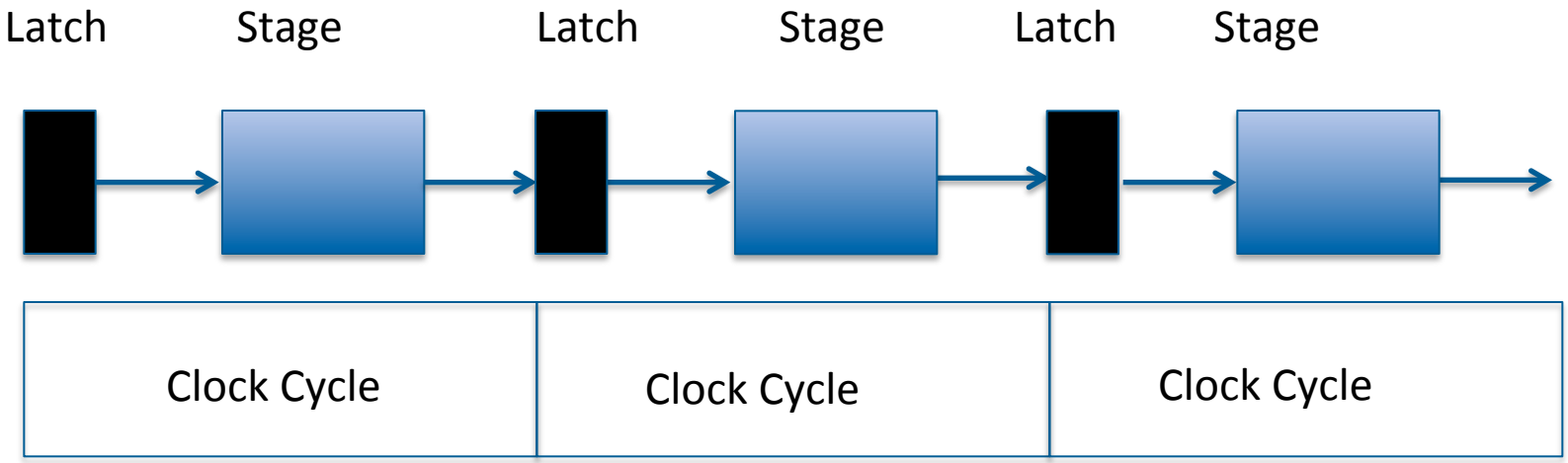
Pipelined Performance



Pipelined-cycle (T_c= 200ps)
T_{total} = 1000 + 200 + 200 = 1400 ps

Cycle time matches longest step

Pipelining



- Although it is possible to divide the work in to approximately equal stages, they are unlikely to be exactly equal
- Latches are added to synchronize the stages
- The latches themselves may add a small latency to each stage (folded into cycle time)

Pipeline Speedup

$T_{pipelined} = T_{nonpipelined} \div$ Number of stages

Speedup ≈ N assuming:

- independent instructions
- balanced pipeline stages

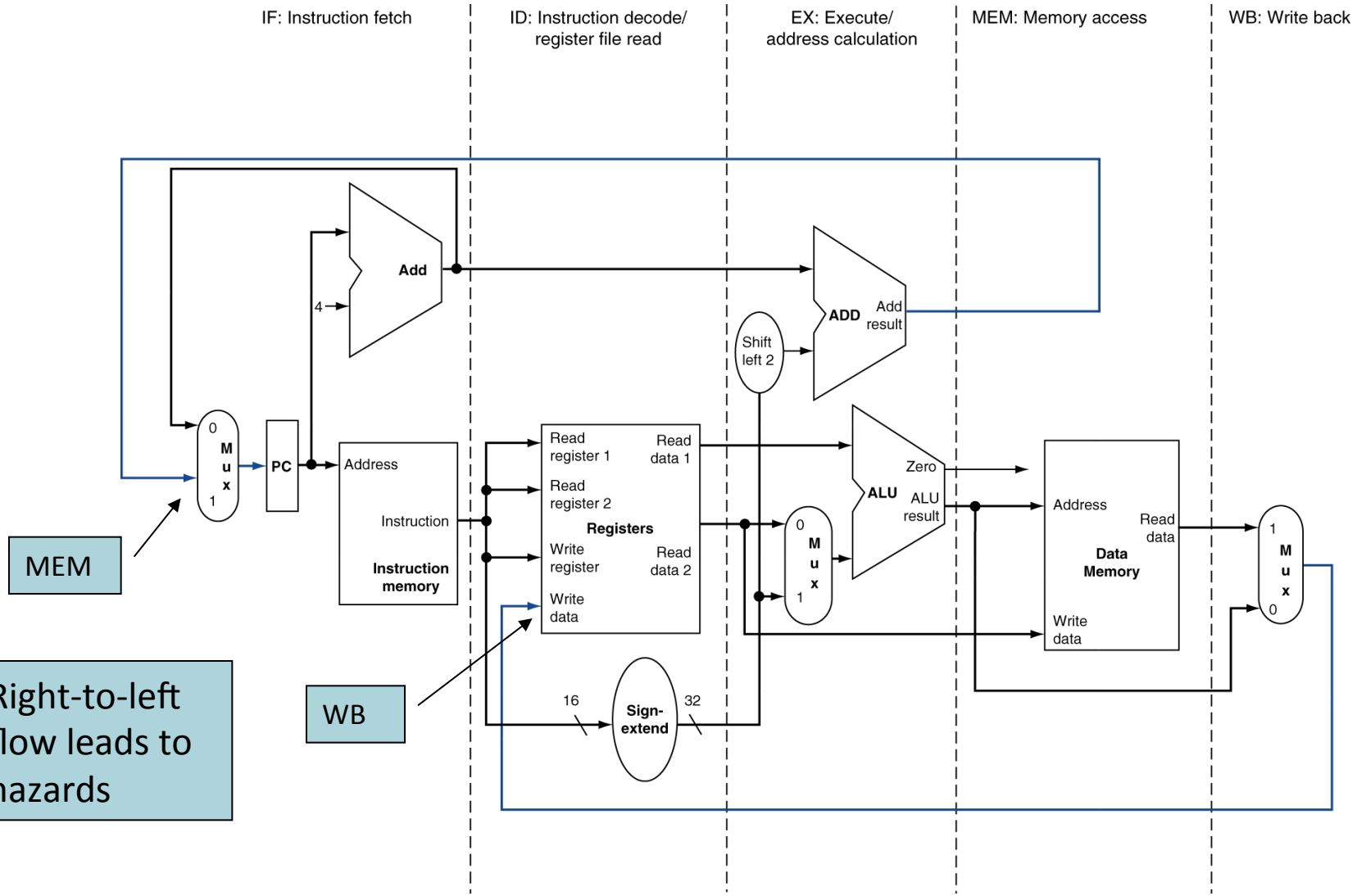
Dependencies cause stalls and reduce speedup

It is throughput that is increased

Latency or fill time is 5 cycles

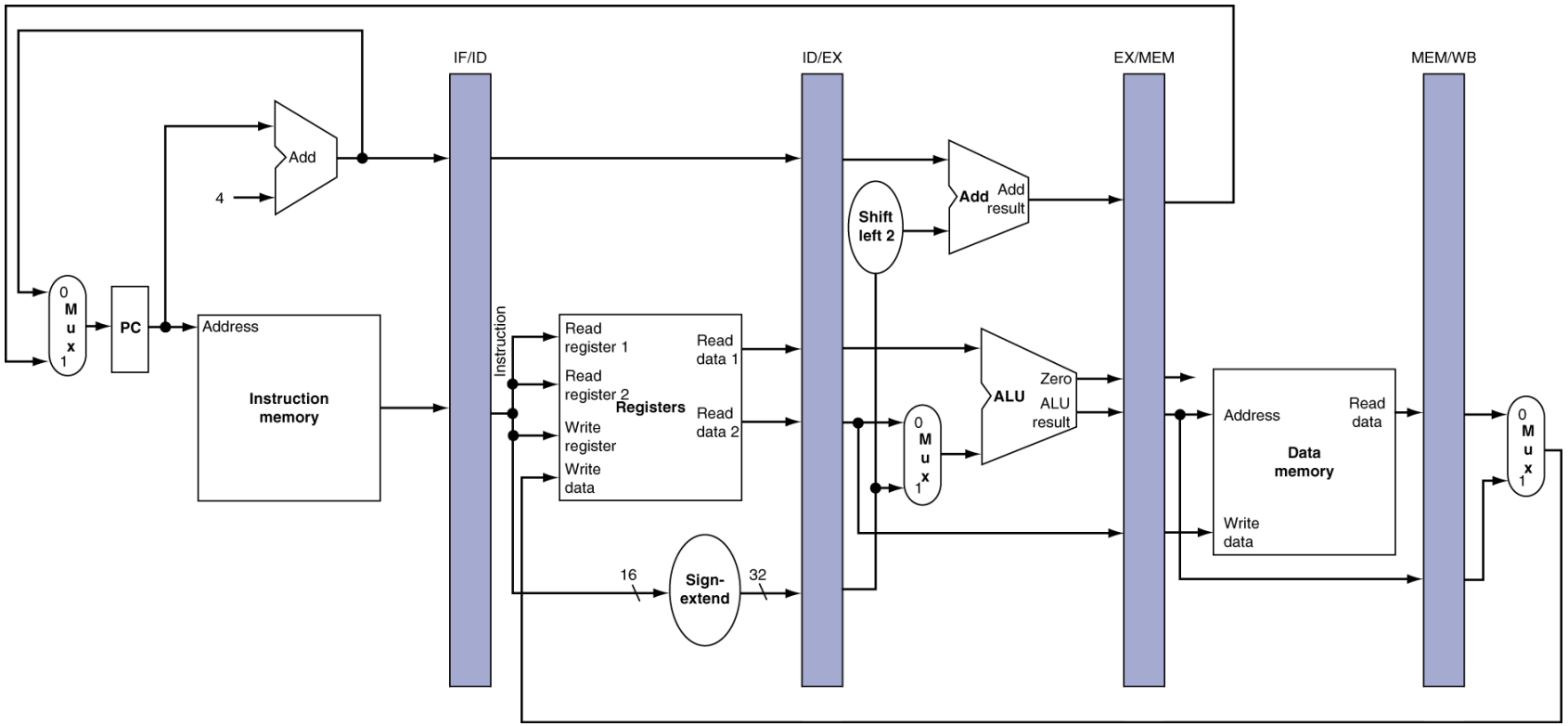
- A new instruction completes for each cycle thereafter
- Each instruction still takes 5 cycles

MIPS Pipelined Datapath



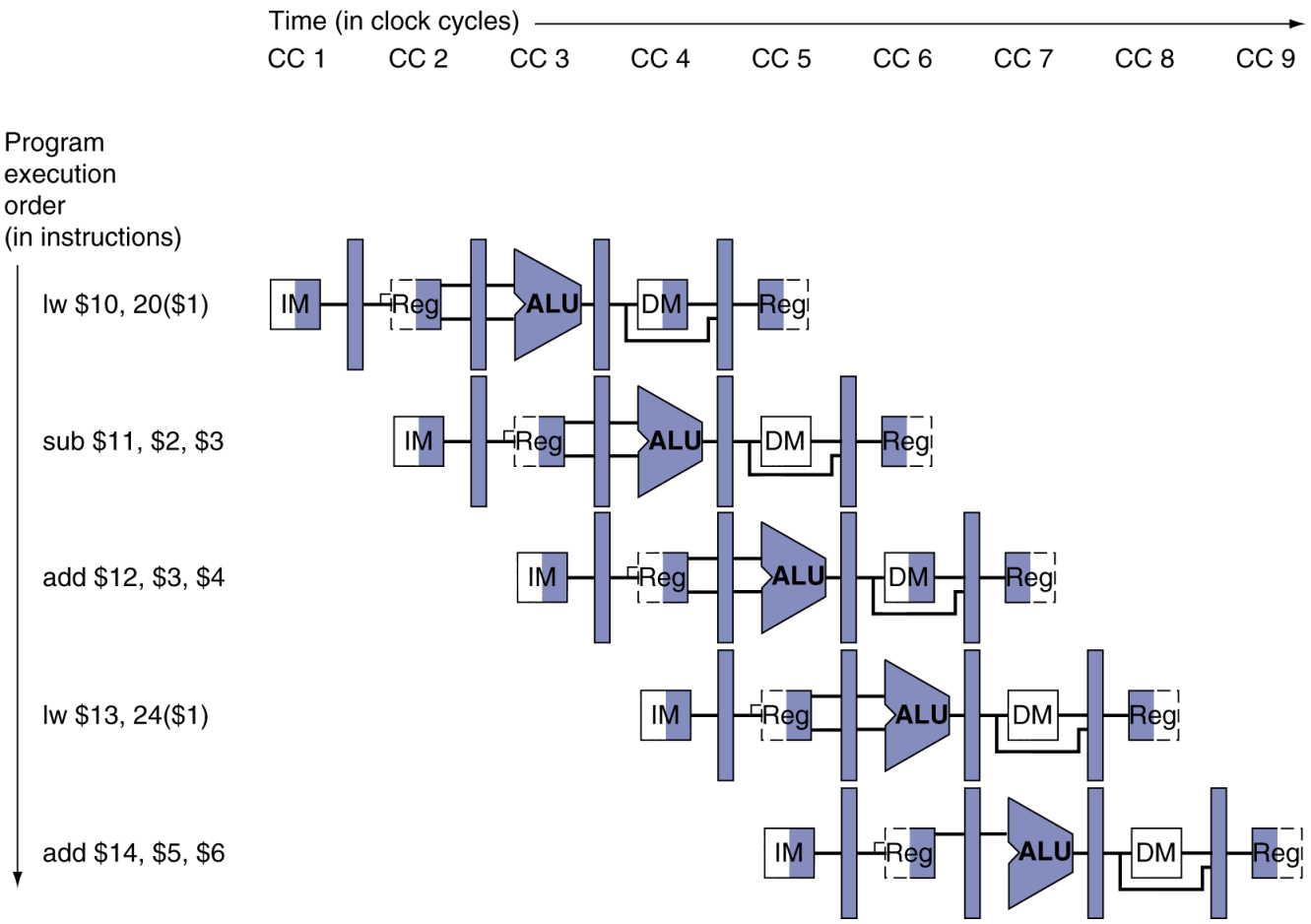
Pipeline Registers

- Need registers between stages
  - To hold information produced in previous cycle



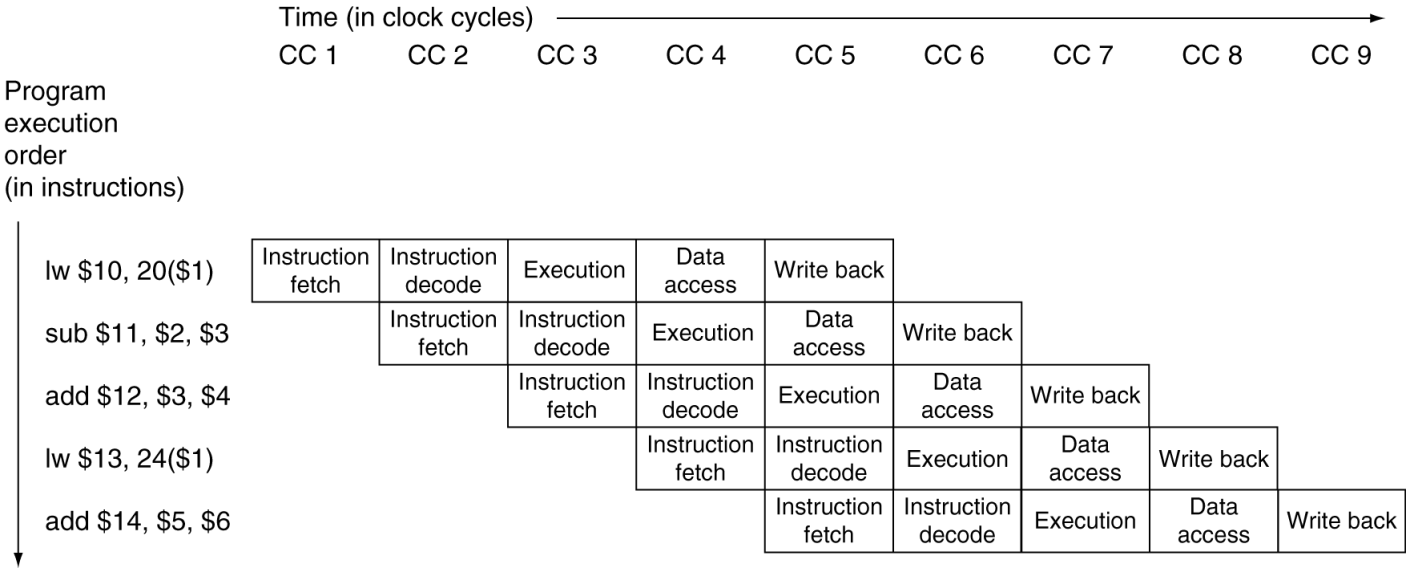
Pipeline Diagram

- Form showing resource usage



Pipeline Diagram

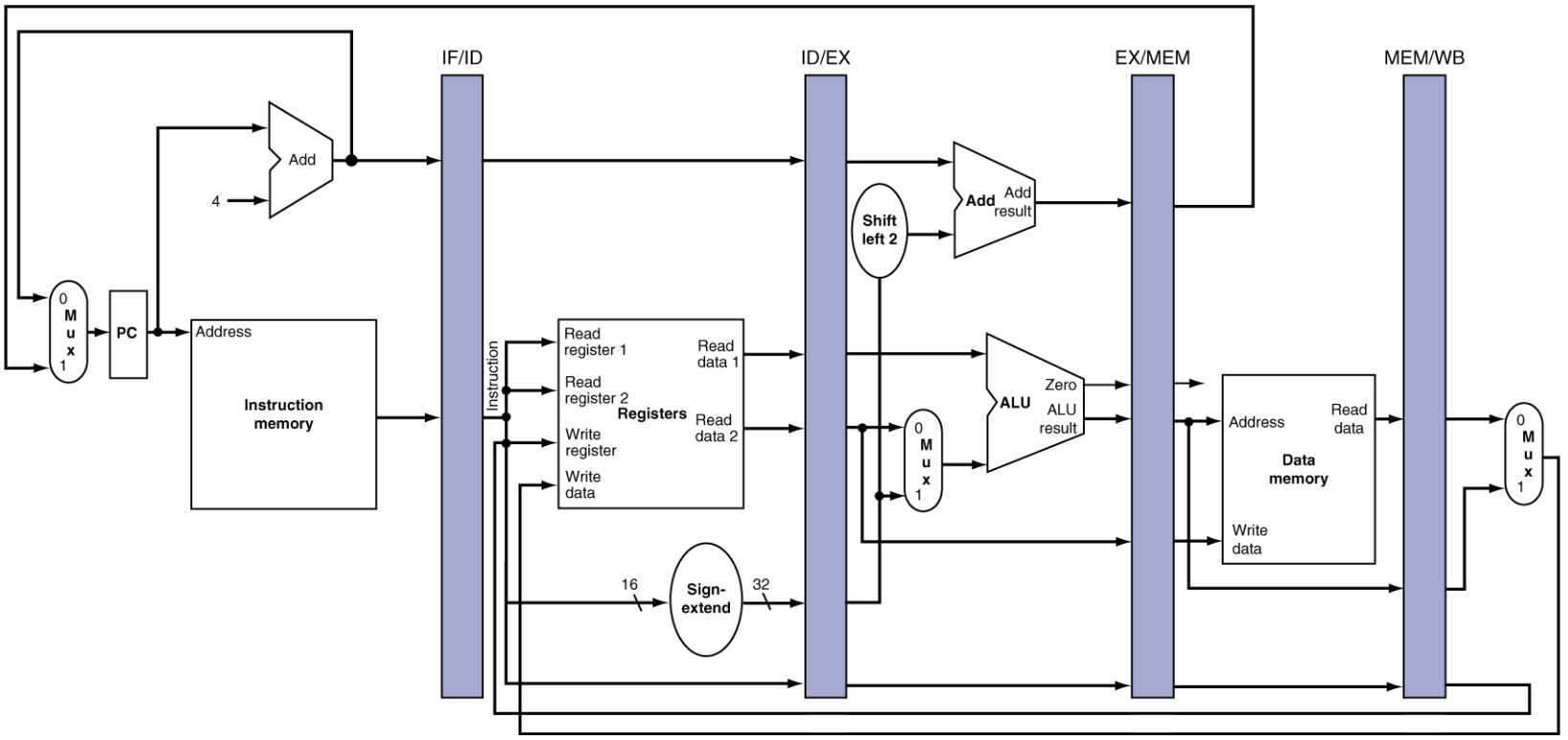
- Traditional form



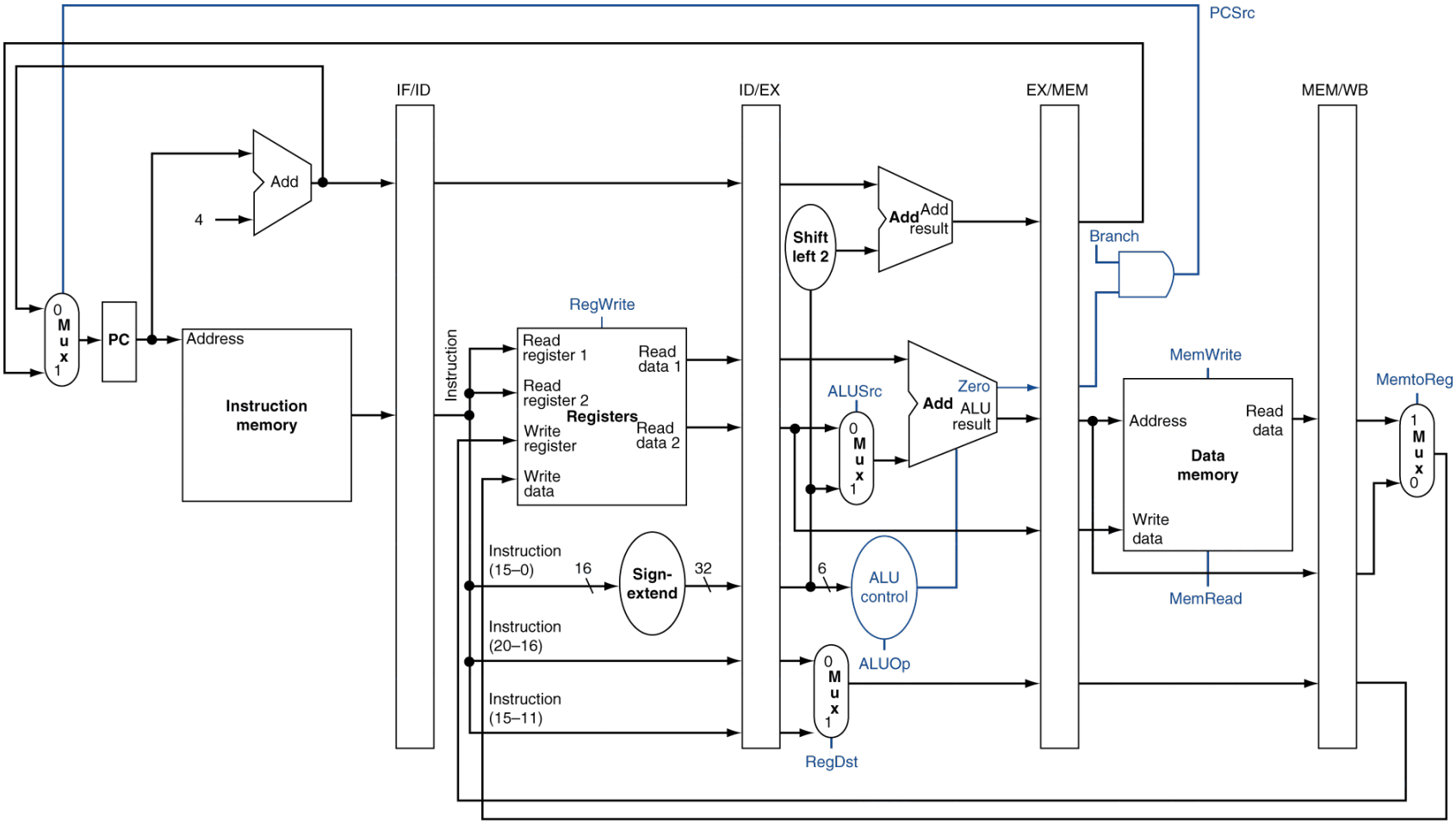
Pipeline Diagram

- State of pipeline in a given cycle

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

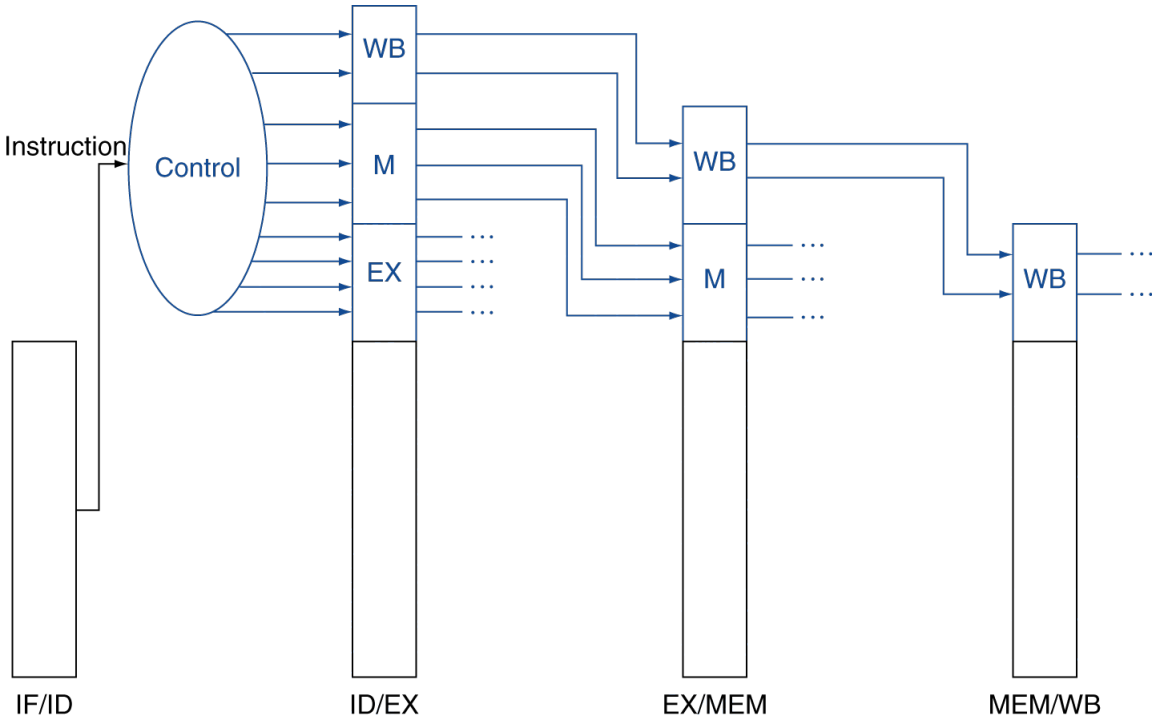


Pipelined Control

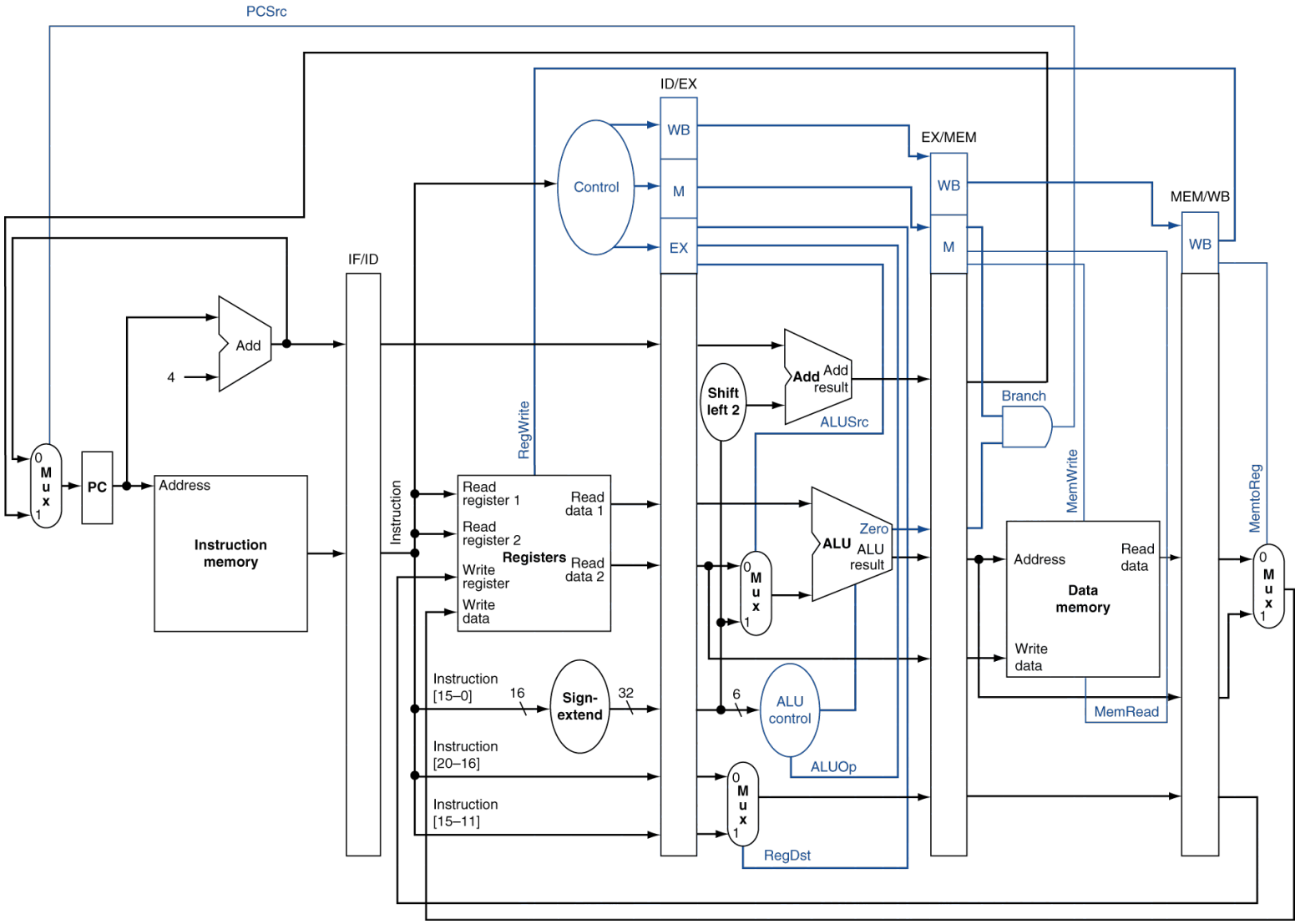


Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



Pipelining with no hazards

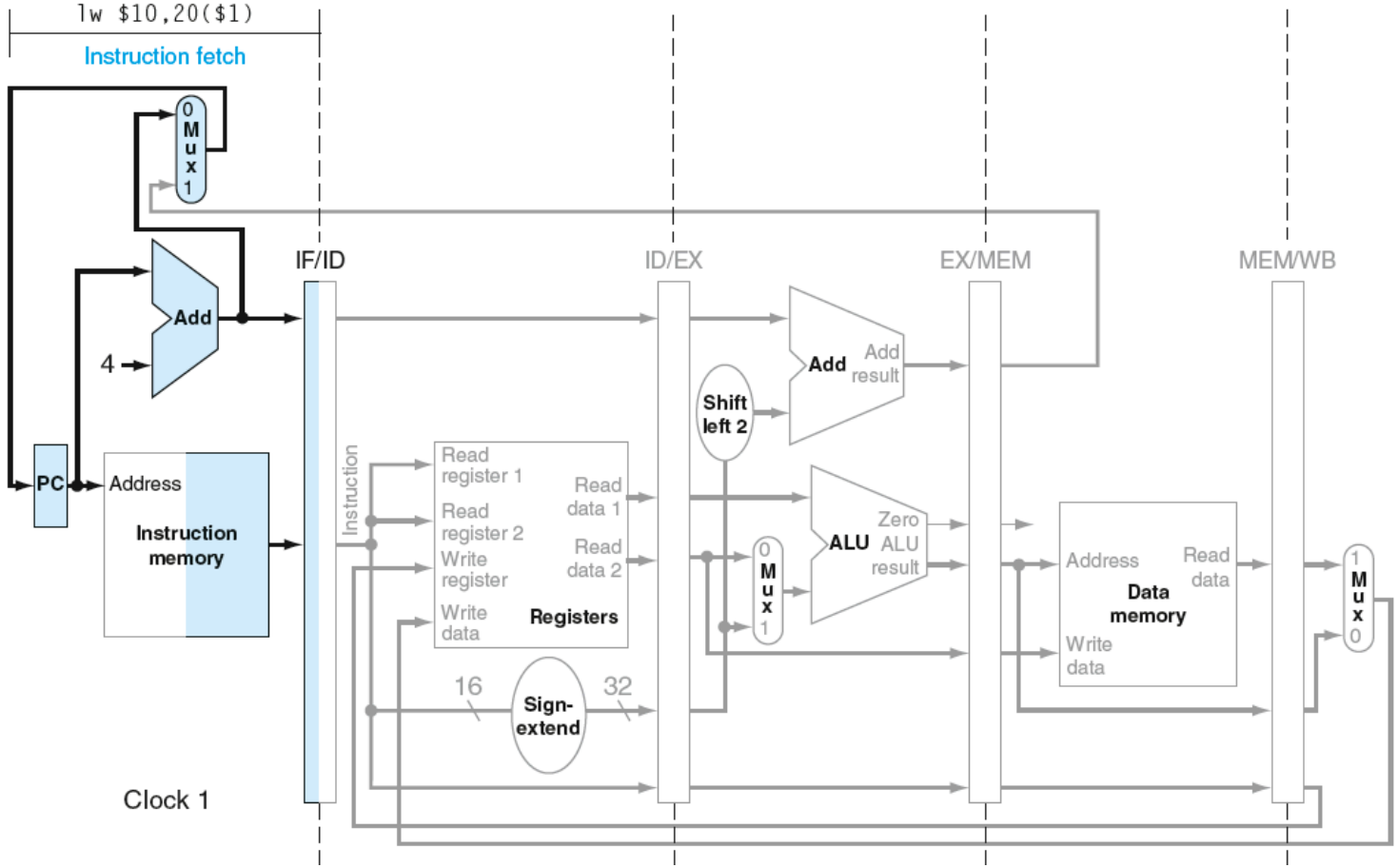
Pipelining the instructions below produces the expected results

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

There are no dependencies, so no stalls are needed

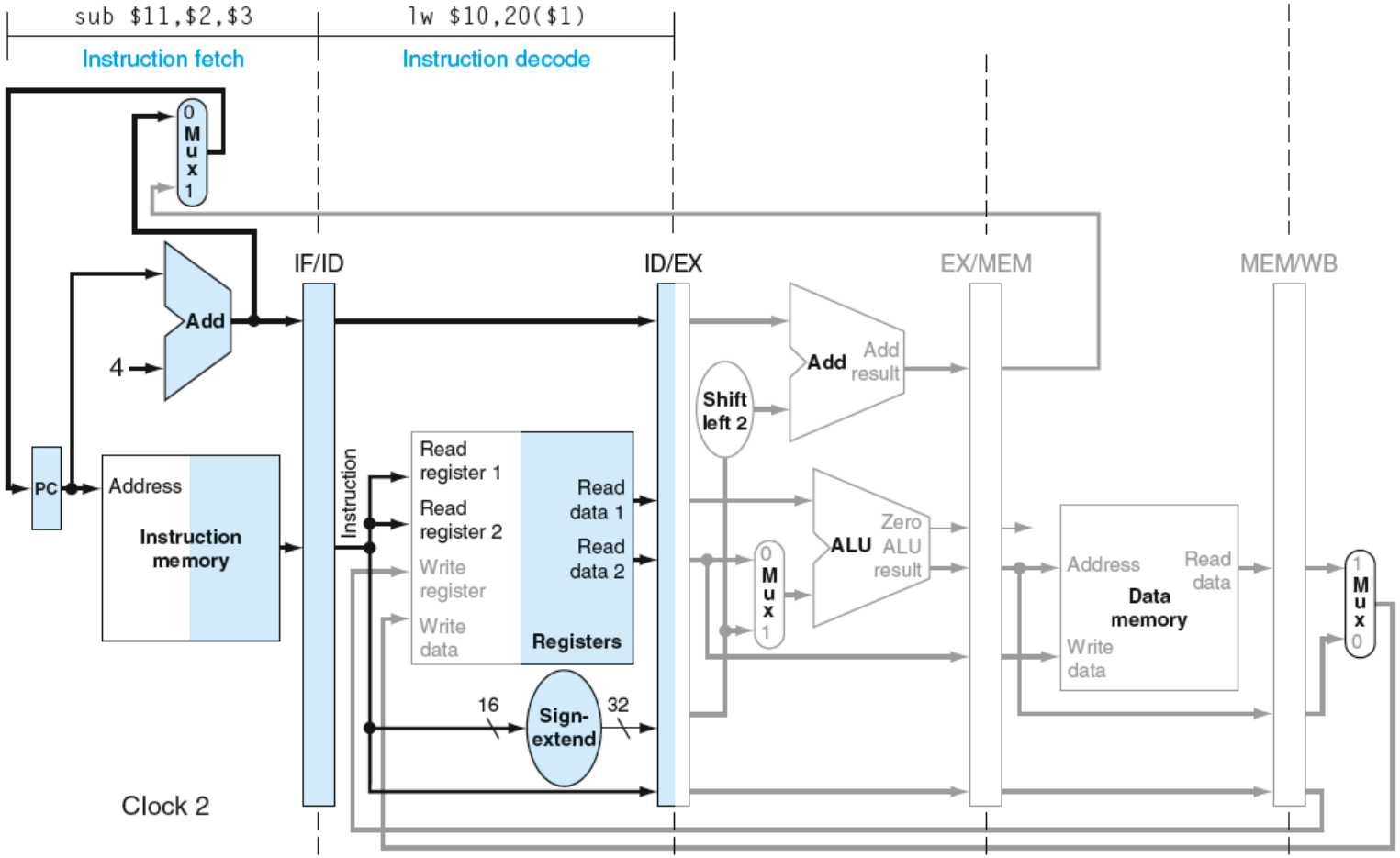
As a simple example, we can trace the first 2 instructions

Pipelining with no hazards



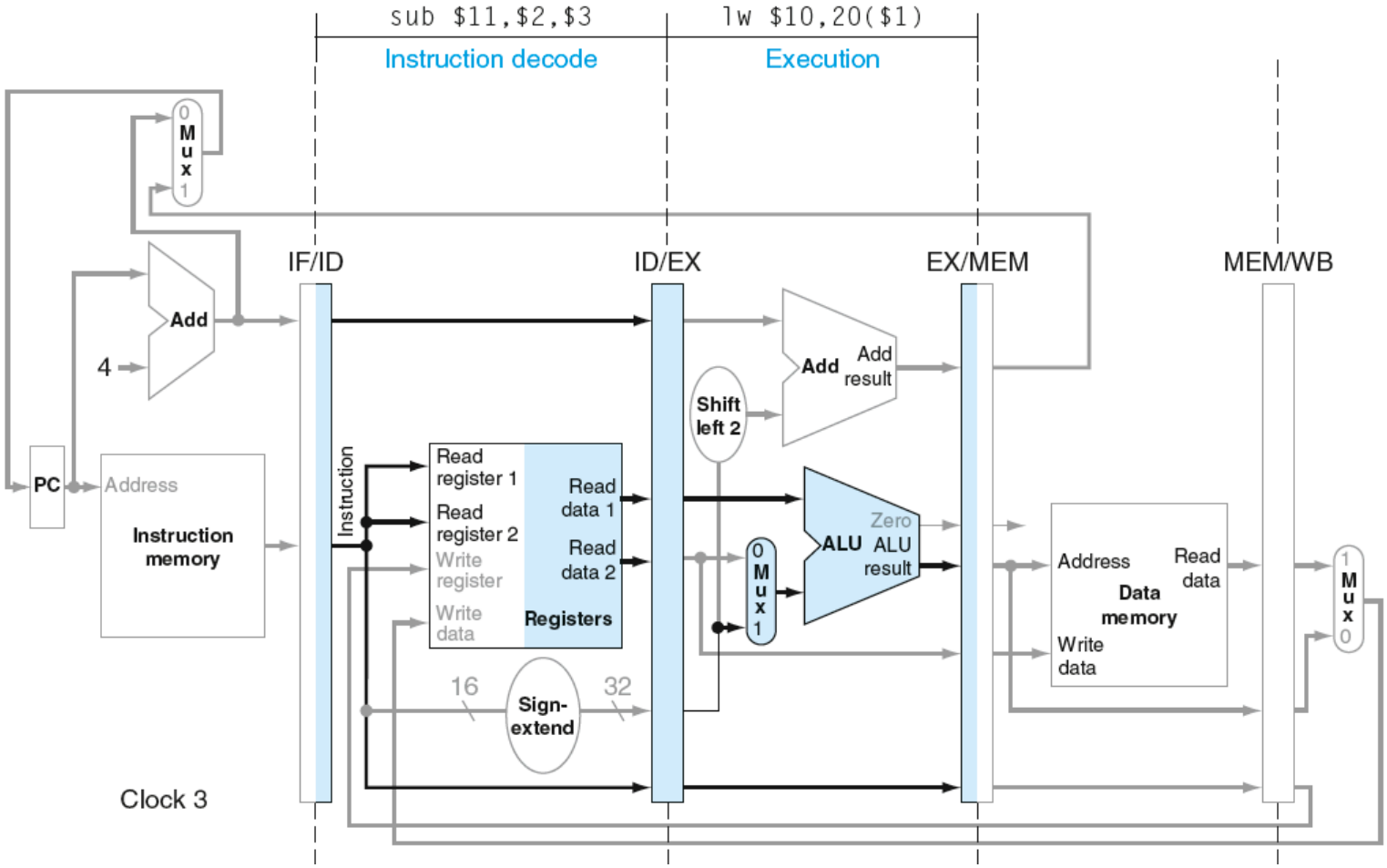
LW is fetched in cycle 1

Pipelining with no hazards



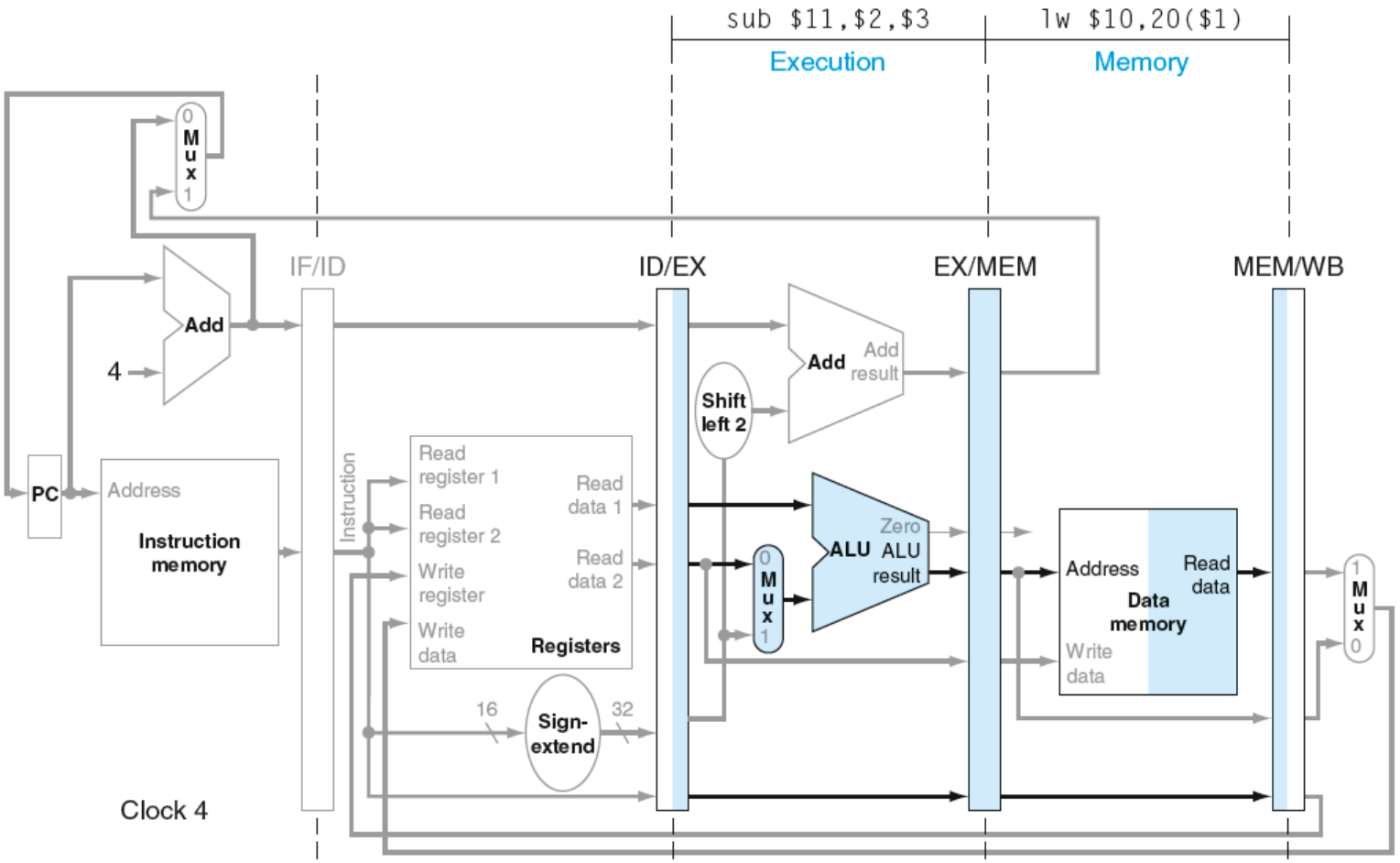
In cycle 2, LW advances to the decode stage and SUB is fetched

Pipelining with no hazards



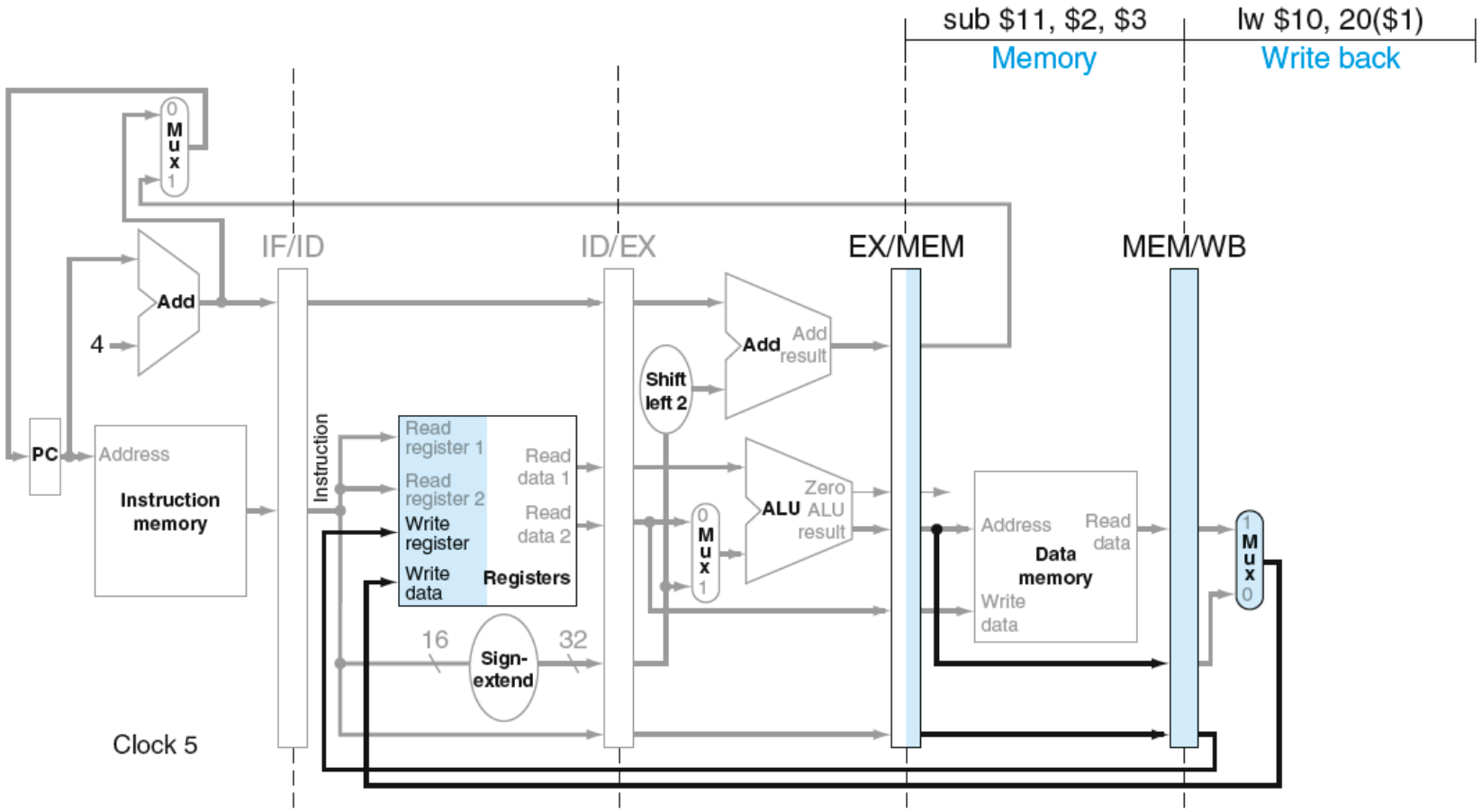
By cycle 3, LW is in the execute stage and SUB is in the decode stage

Pipelining with no hazards



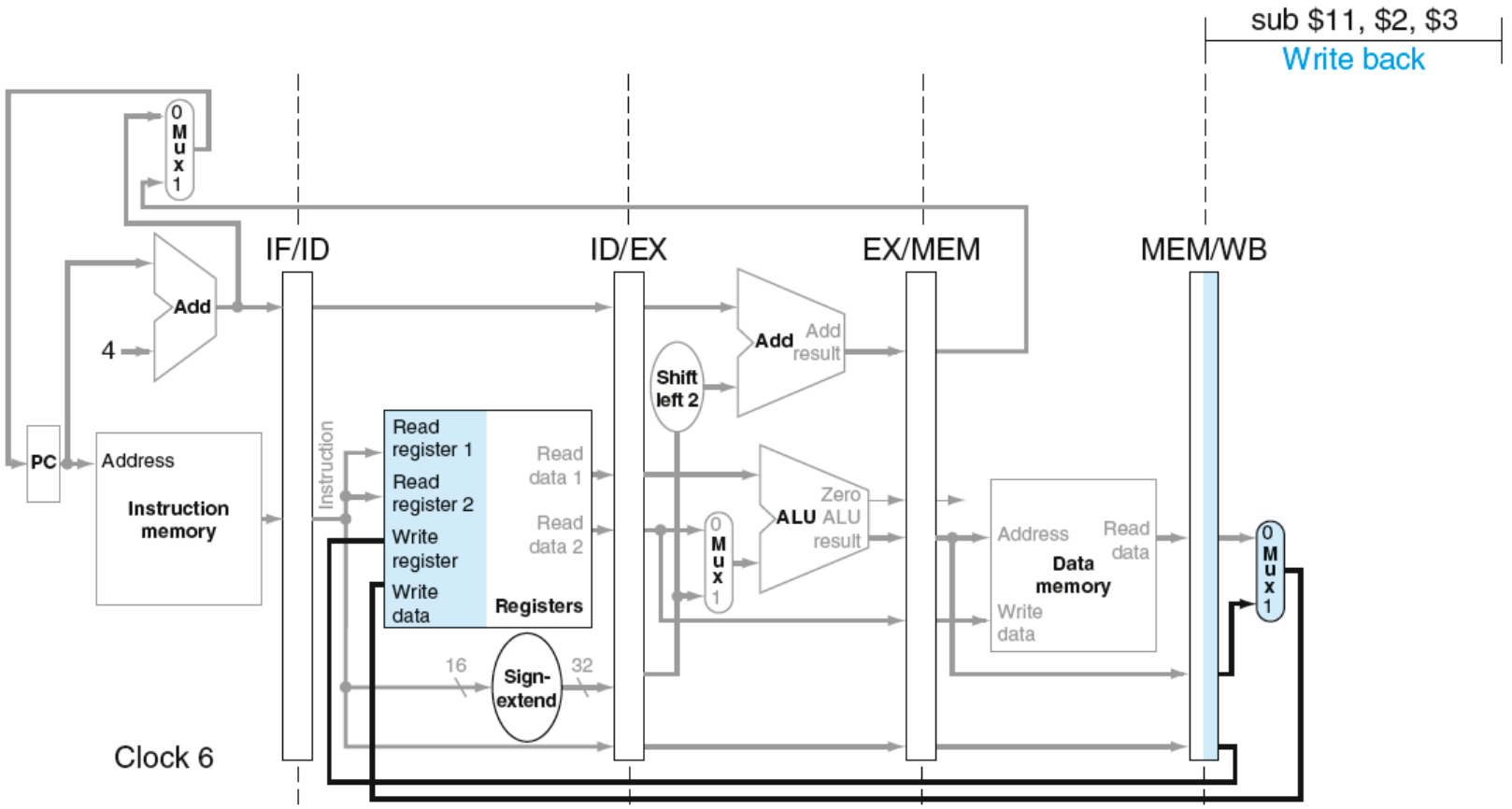
LW reads from the data memory in cycle 4, while SUB executes in stage 3

Pipelining with no hazards



LW completes in cycle 5 by writing its result into $10
SUB simply idles in stage 4 since it does not access memory

Pipelining with no hazards



SUB completes in cycle 6, writing its result into $11

The two instructions require a total of 6 cycles to complete

Pipelining with no hazards

- Pipelining provides a higher throughput
  - More instructions complete per unit time
  - Each instruction takes 5 cycles
  - But instructions are overlapped

- On the non-pipelined multi-cycle system
  - lw takes 5 cycles
  - sub takes 4 cycles
  - Total for this example = 9 cycles instead of 6
  - Each instruction completes before the next starts

Filling the Pipeline

This example traces 5 instructions through the pipeline:

```
lw      $10, 20($1)
sub     $11, $2, $3
and     $12, $4, $5
or      $13, $6, $7
add     $14, $8, $9
```

There are 5 stages, so it takes 5 instructions to fill the pipeline

The result registers $10 through $14 are not used as inputs, so there are no dependencies or data hazards

Filling the Pipeline

Five instructions are required to fill the pipeline

The first instruction completes after 5 cycles

- Hence the pipeline *latency* is 5 cycles
- This is also called the *fill time*

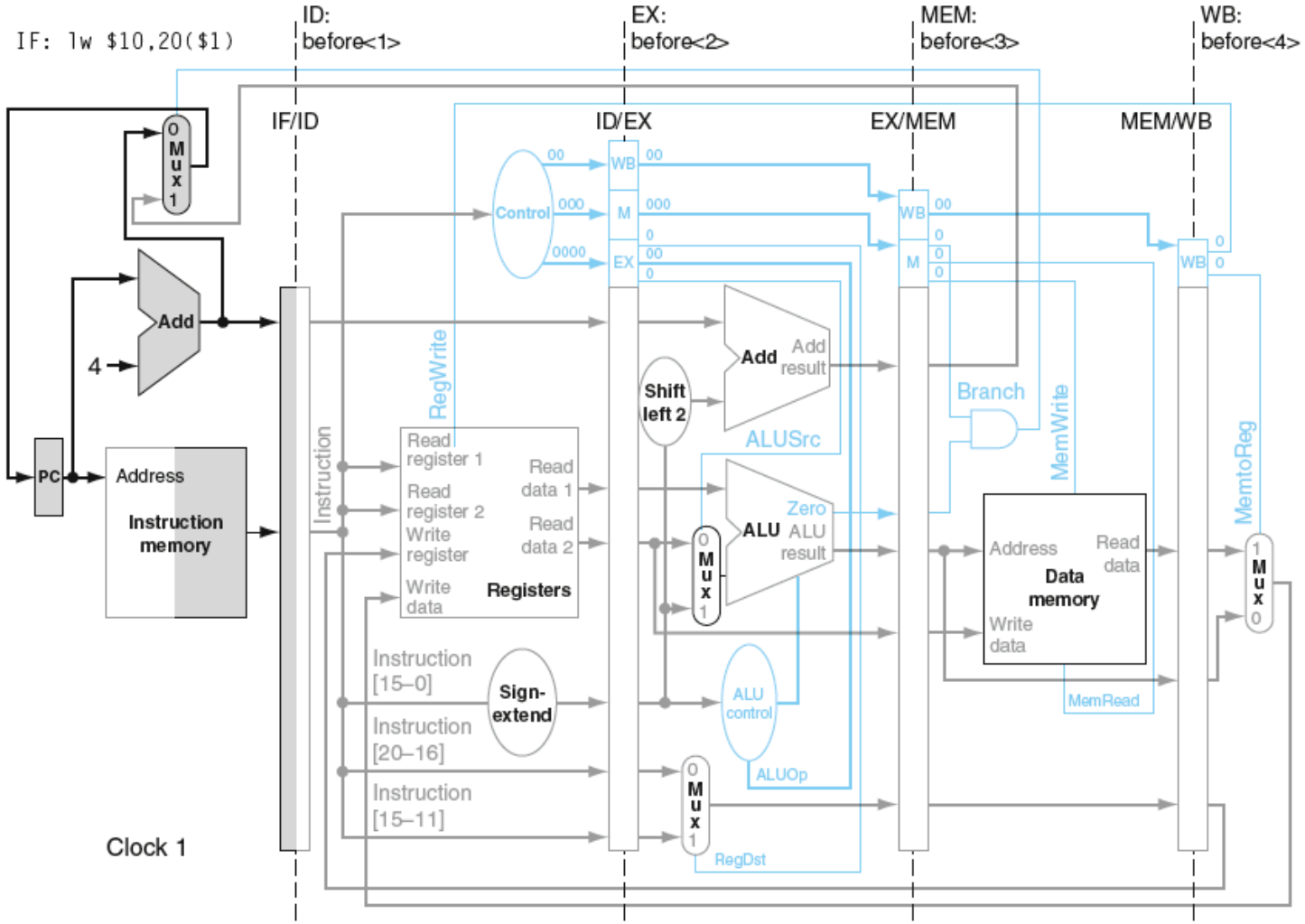
One instruction completes for each subsequent cycle

- pipeline stalls would be required if there were hazards

"before <i>" refers to the i^{th} instruction before those in the example 5-instruction sequence

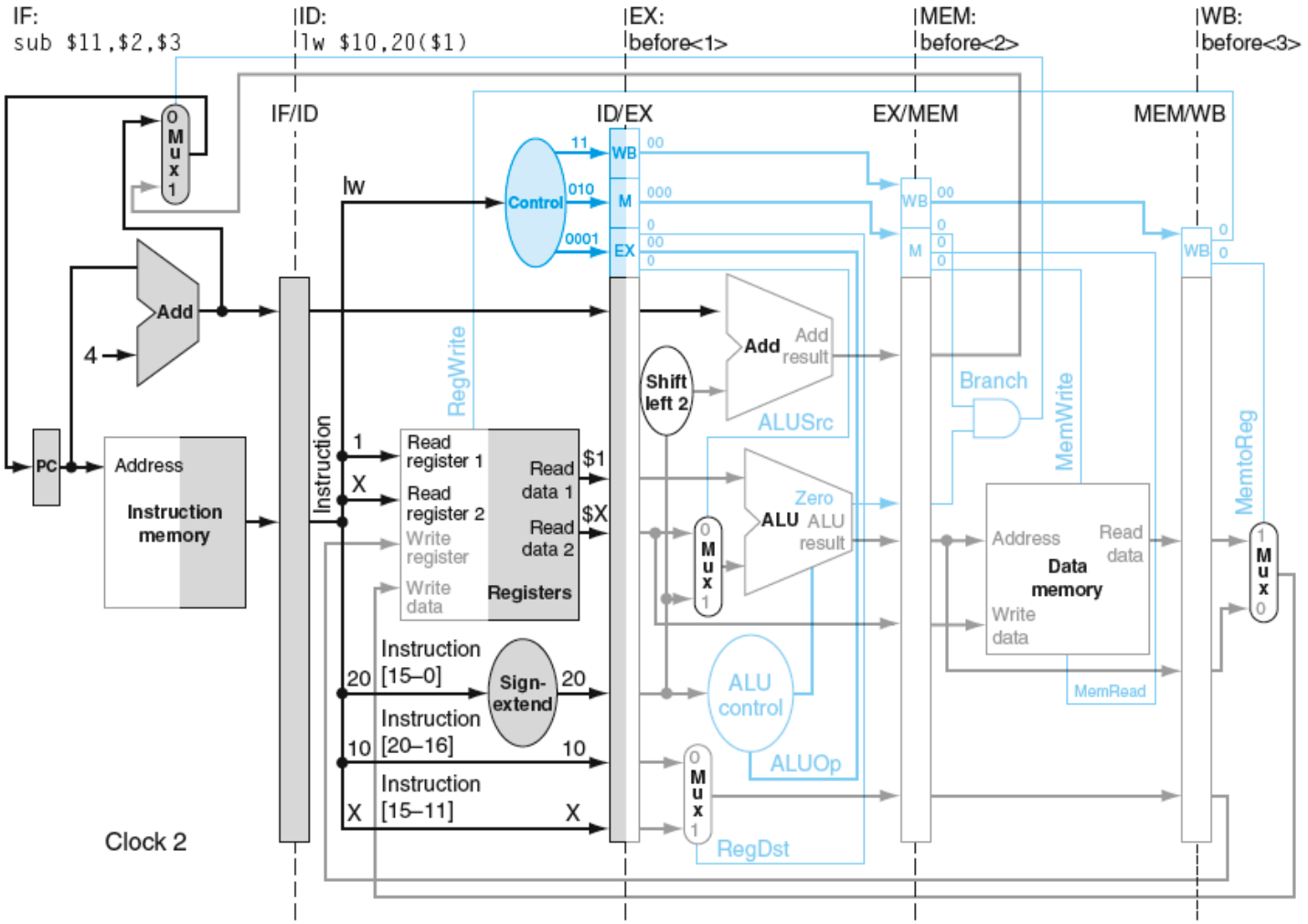
"after <i>" will refer to the i^{th} instruction following those in the example sequence

Pipelining with no hazards



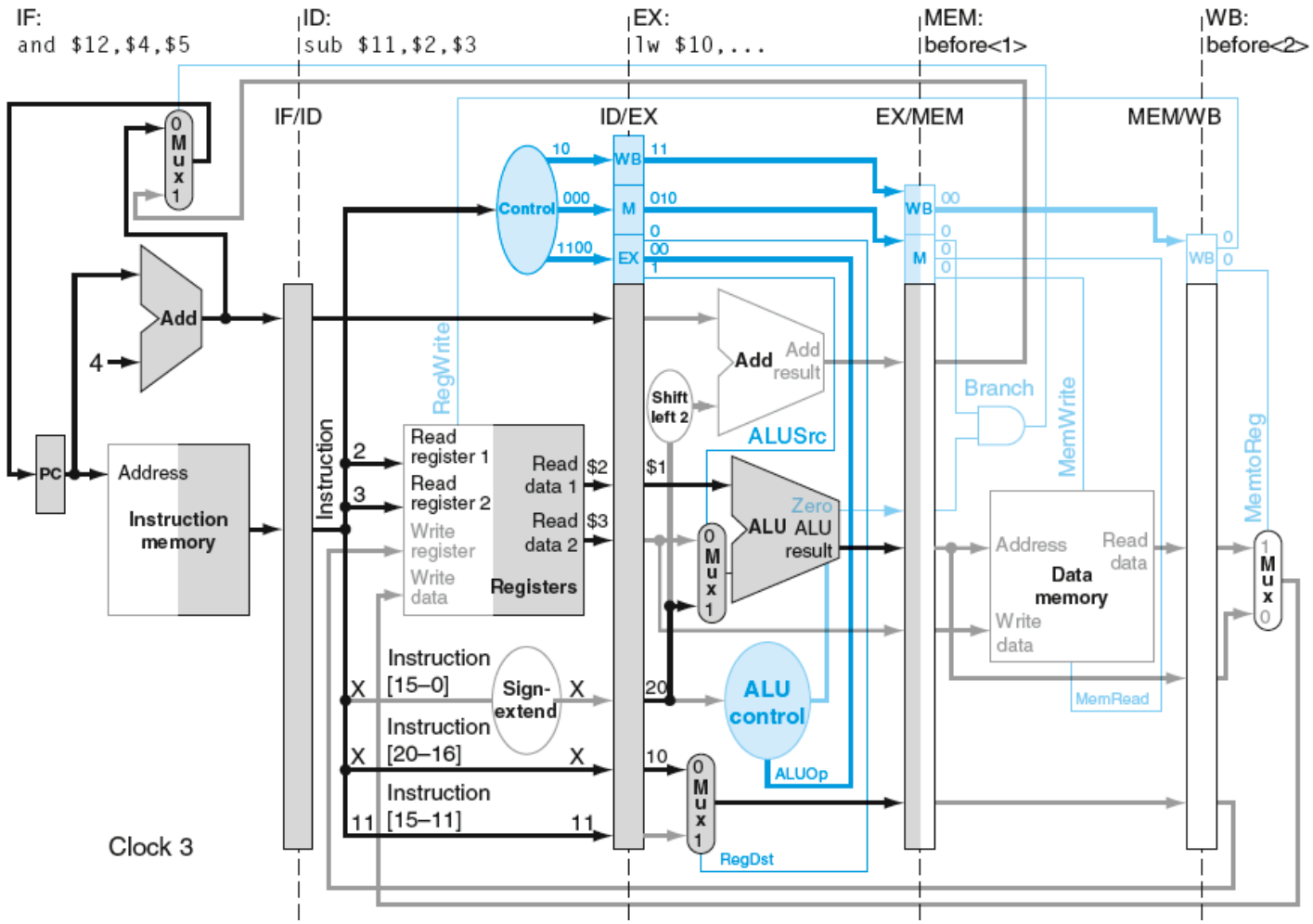
LW is fetched in cycle 1

Pipelining with no hazards



In cycle 2, the control signals for LW are generated and SUB is fetched

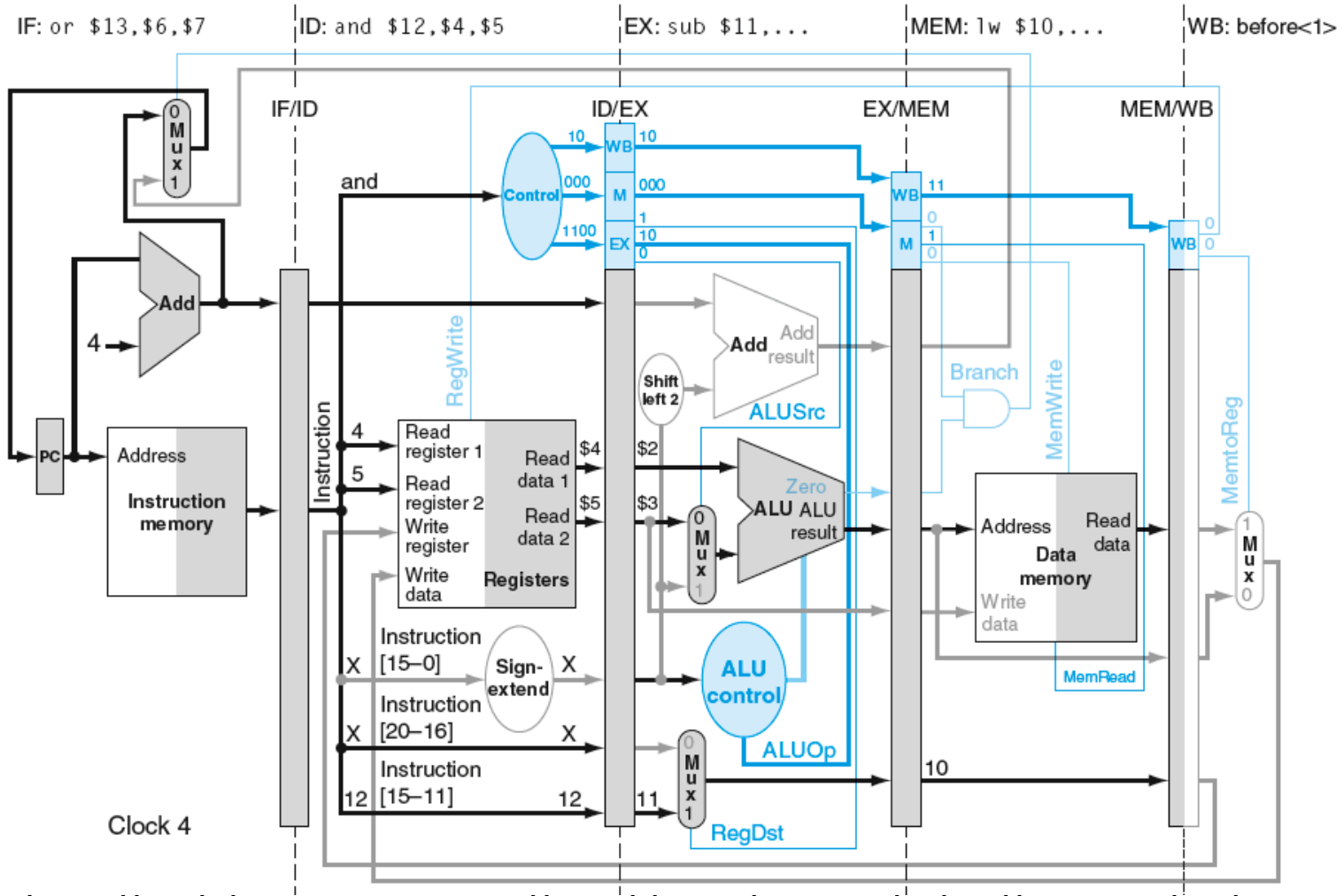
Pipelining with no hazards



Each instruction reads its input registers in the decode stage

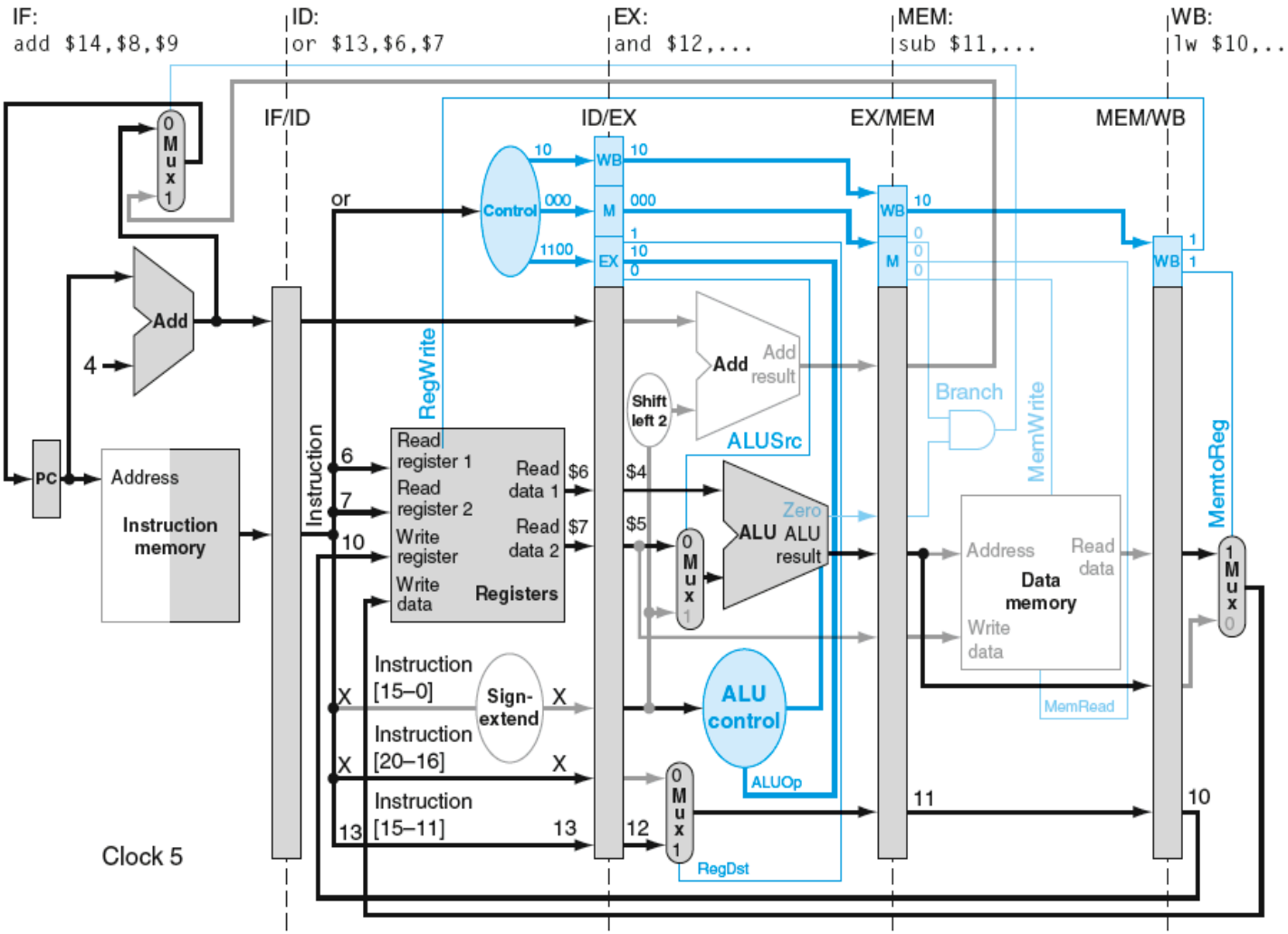
In cycle 3, LW executes, SUB is decoded while AND is fetched

Pipelining with no hazards



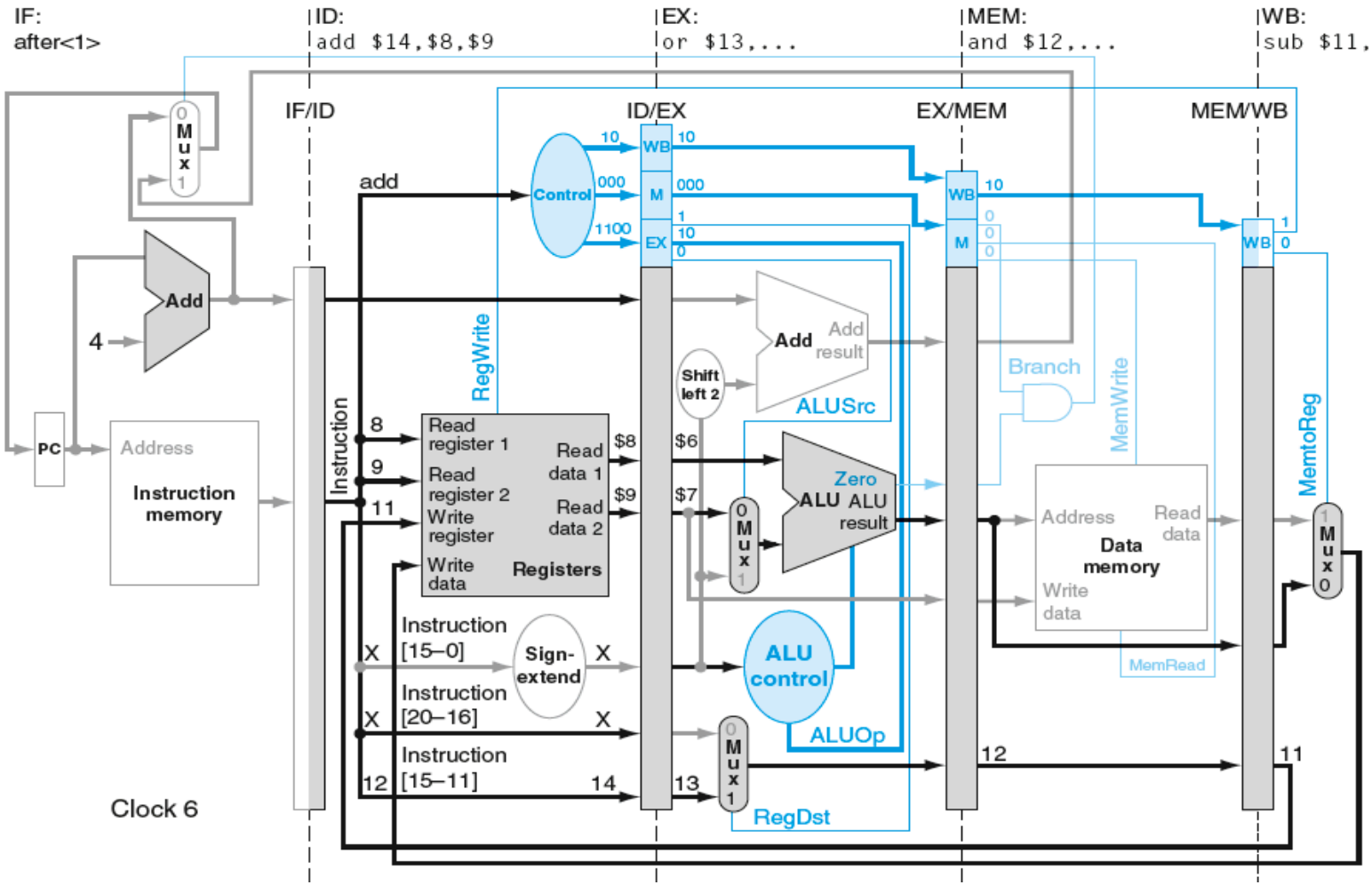
LW reads from the data memory using the address it computed in the execute stage
SUB computes $2 - $3, AND is decoded and reads its input registers ($4 and $5)

Pipelining with no hazards



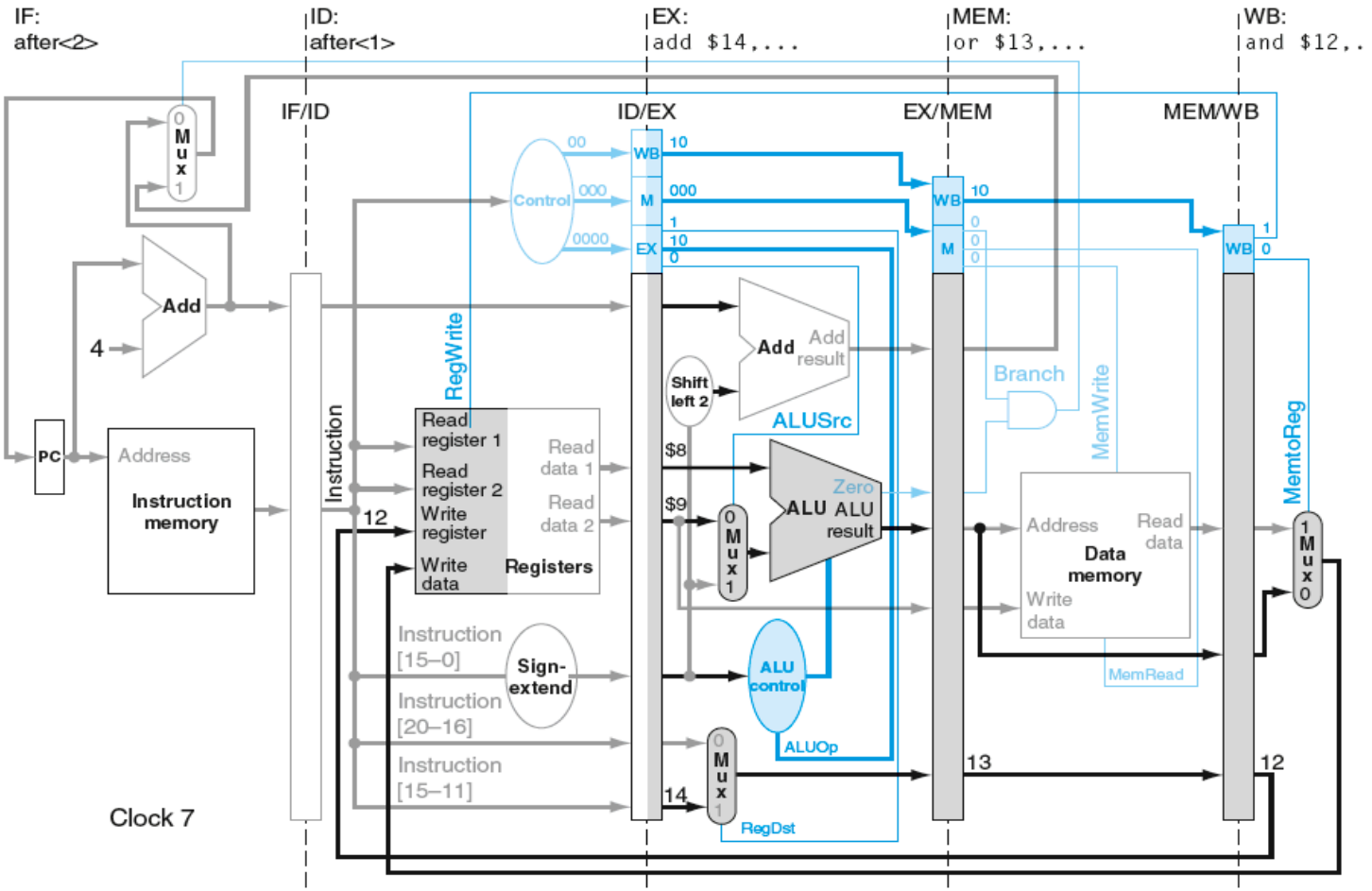
LW completes after writing $10, SUB idles, AND executes, OR is decoded and reads $6 & $7

Pipelining with no hazards



SUB writes $11 and completes, OR executes, ADD is decoded while $8 & $9 are read

Pipelining with no hazards

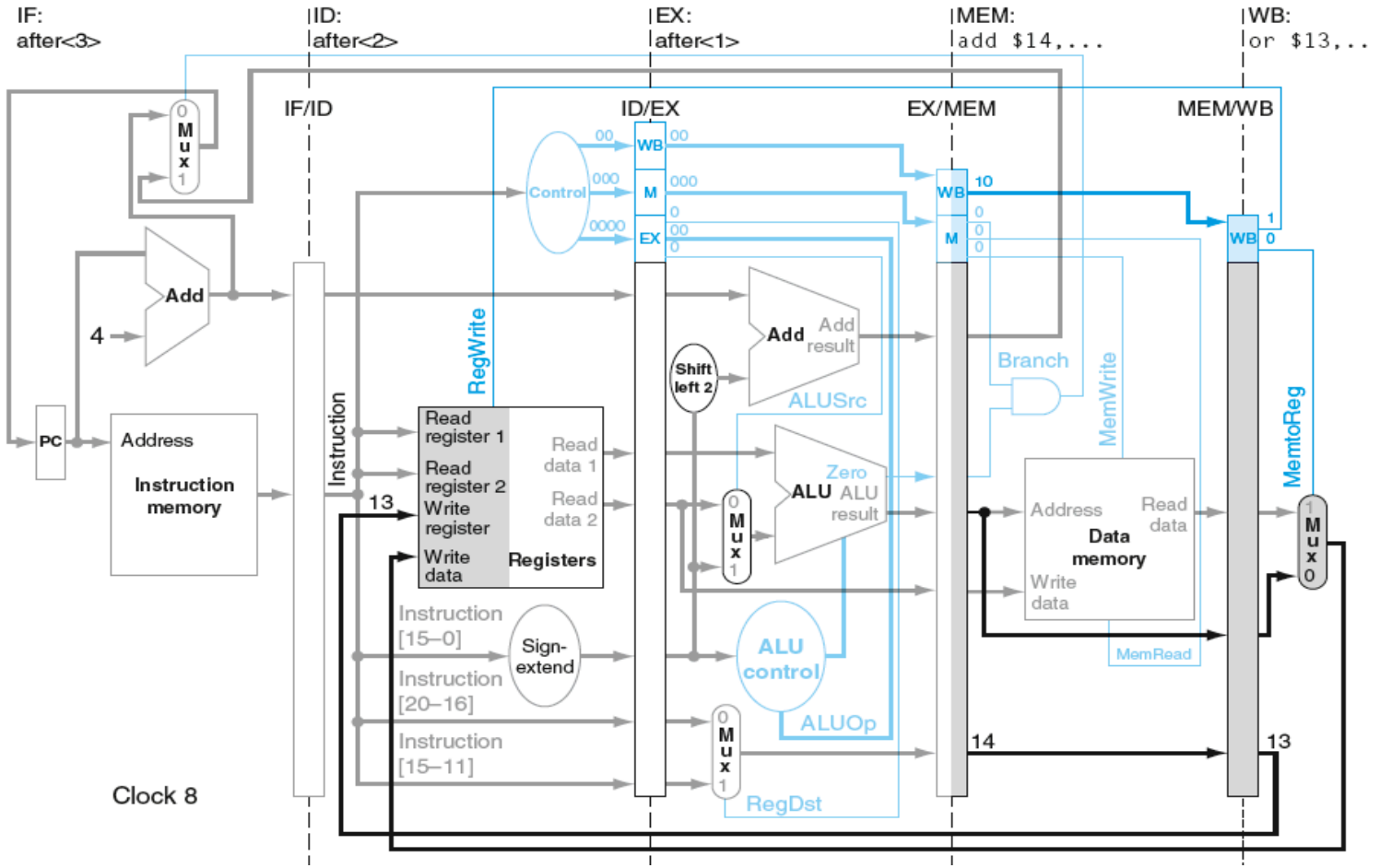


Every instruction must pass through each pipeline stage

Instructions that do not access the data memory idle in stage 4

Hence all instructions take 5 cycles to complete

Pipelining with no hazards

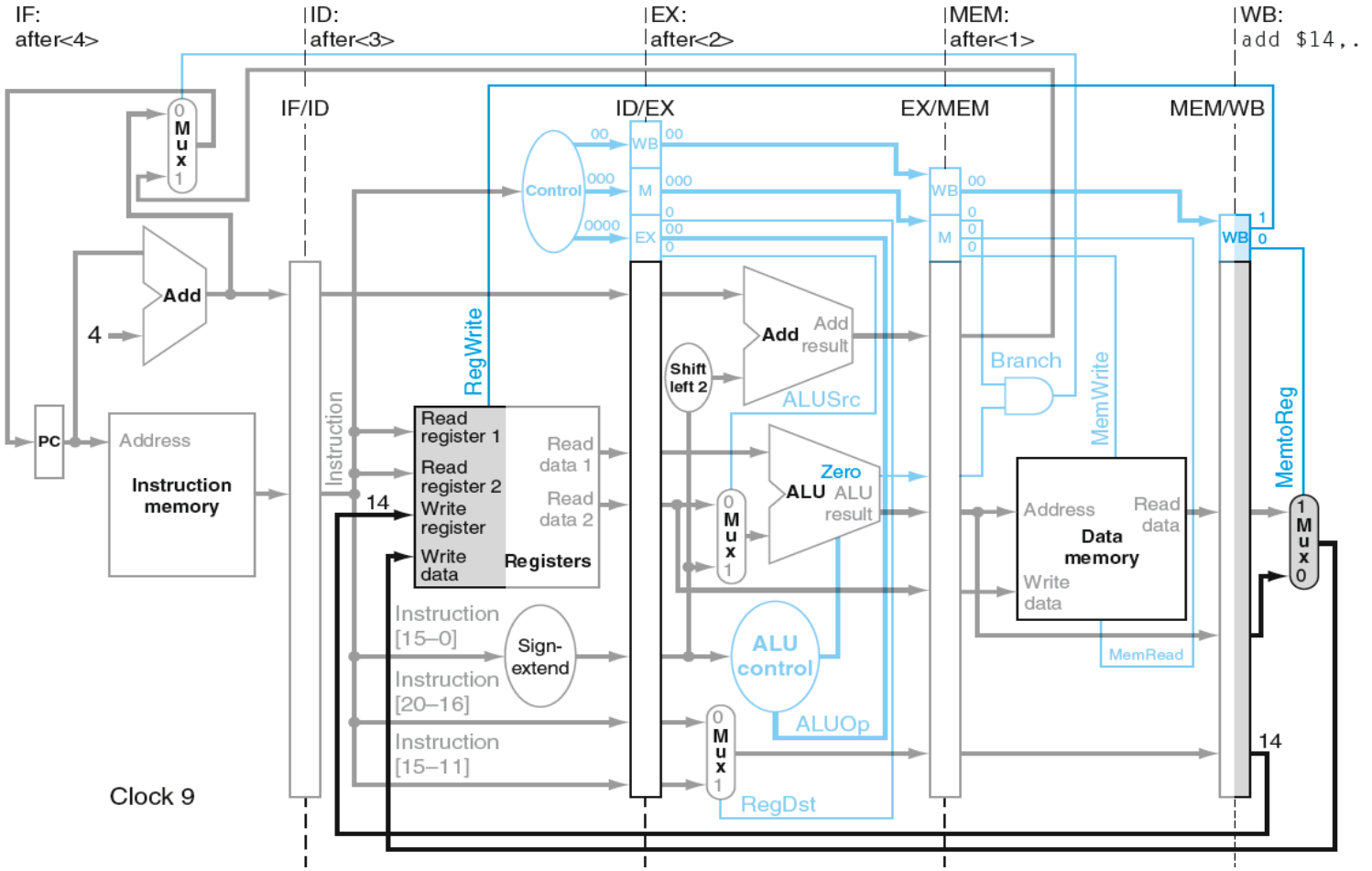


Result registers are written in stage 5 (the write-back stage)

Instructions such as SW and BEQ, do not produce results

In cycle 8, the OR instruction completes

Pipelining with no hazards



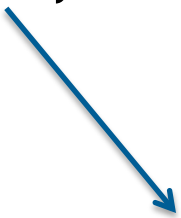
The final instruction in the 5-instruction sequence completes in cycle 9

Pipelining with no hazards

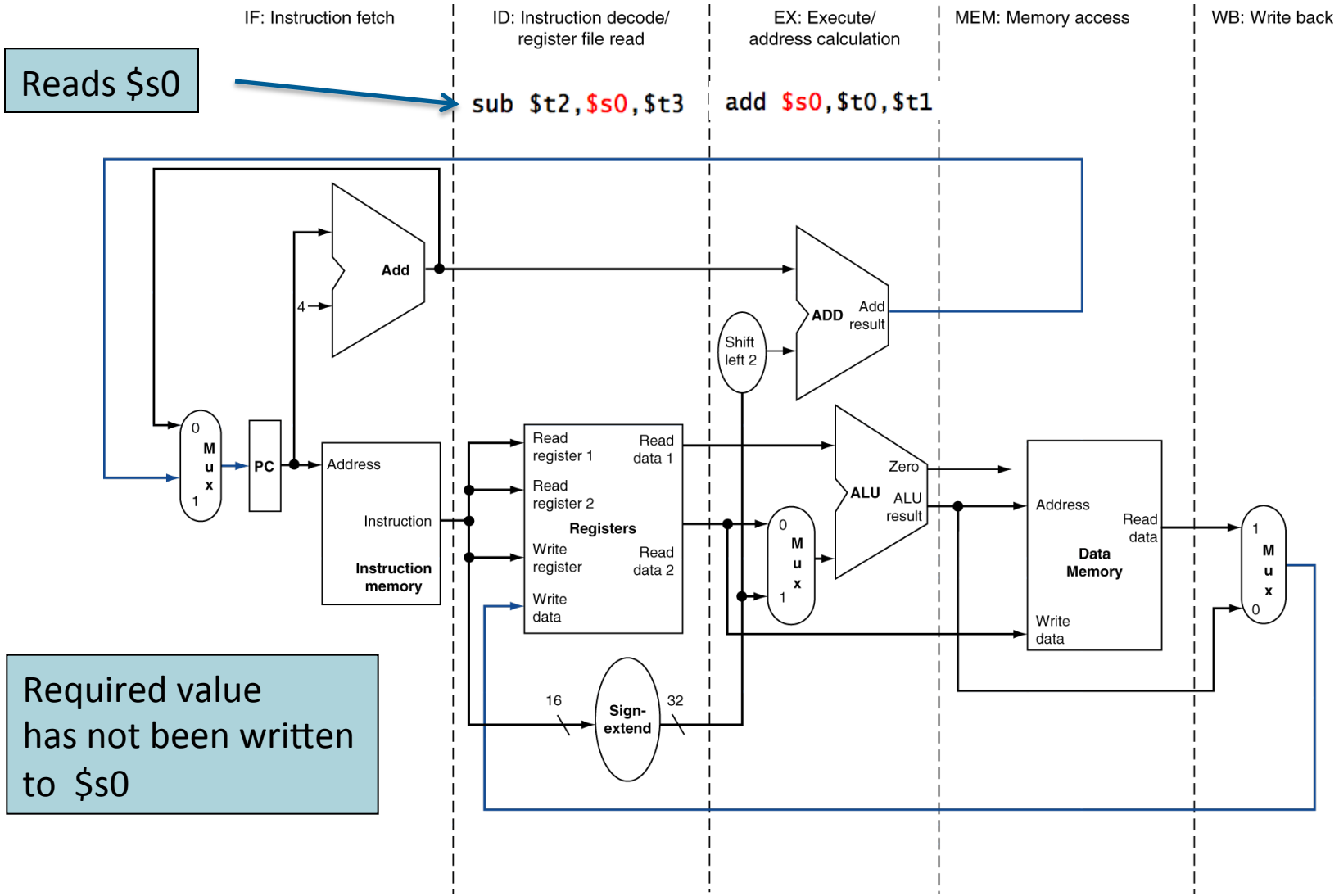
- The 5-instruction sequence completes in 9 cycles
- On the non-pipeline multi-cycle system
  - LW takes 5 cycles
  - SUB, AND, OR & AND each takes 4 cycles
  - Total = 21 cycles without pipelining
- Pipelining provides a speedup
  - In this case speedup = 21/9 = 2.33
  - Stalls to cope with data hazards would reduce the speedup

Data Hazards

- An instruction depends on a result from a previous instruction
  - `add  $s0, $t0, $t1`
  - `sub  $t2, $s0, $t3`

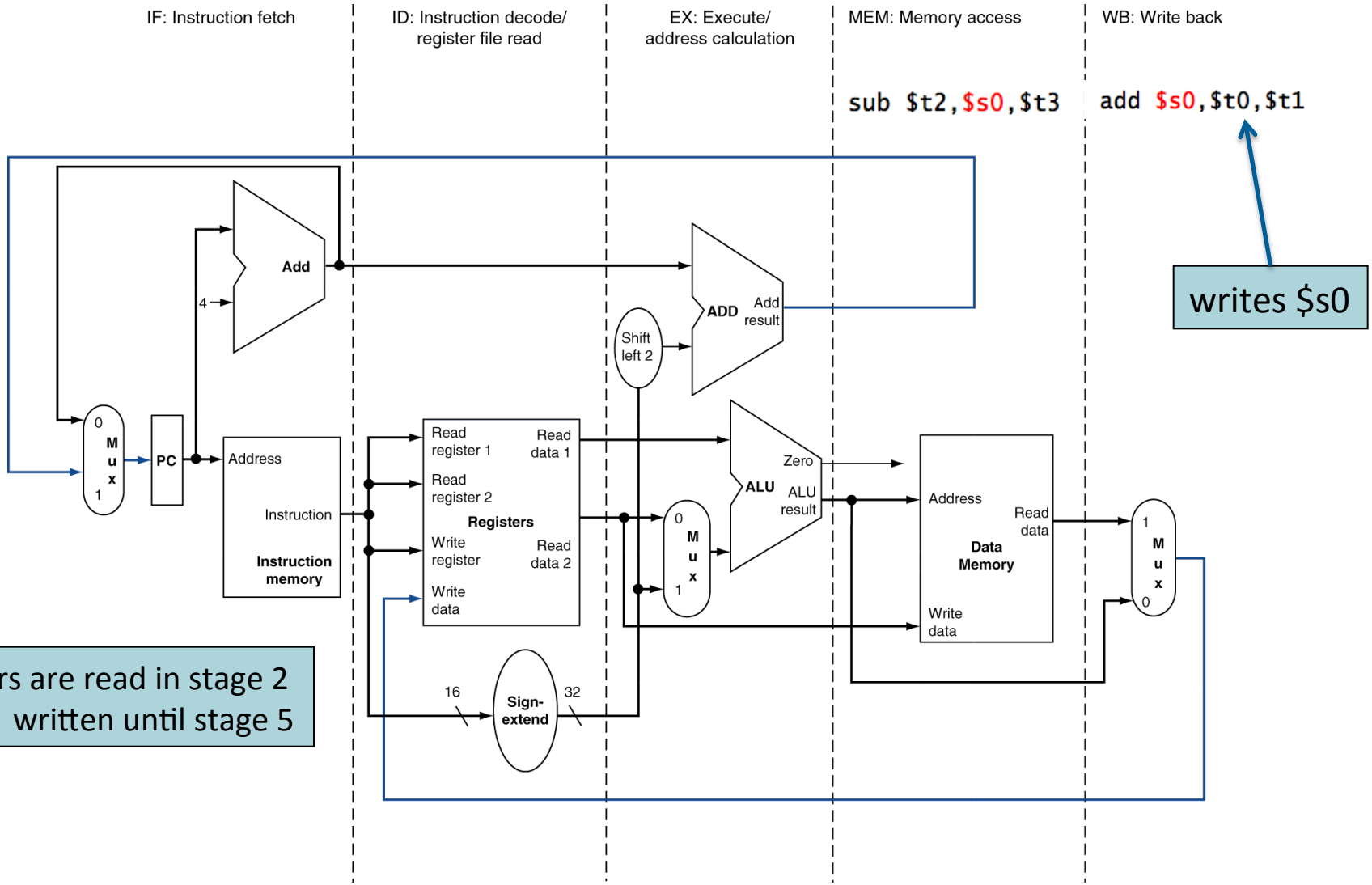


Data Hazards



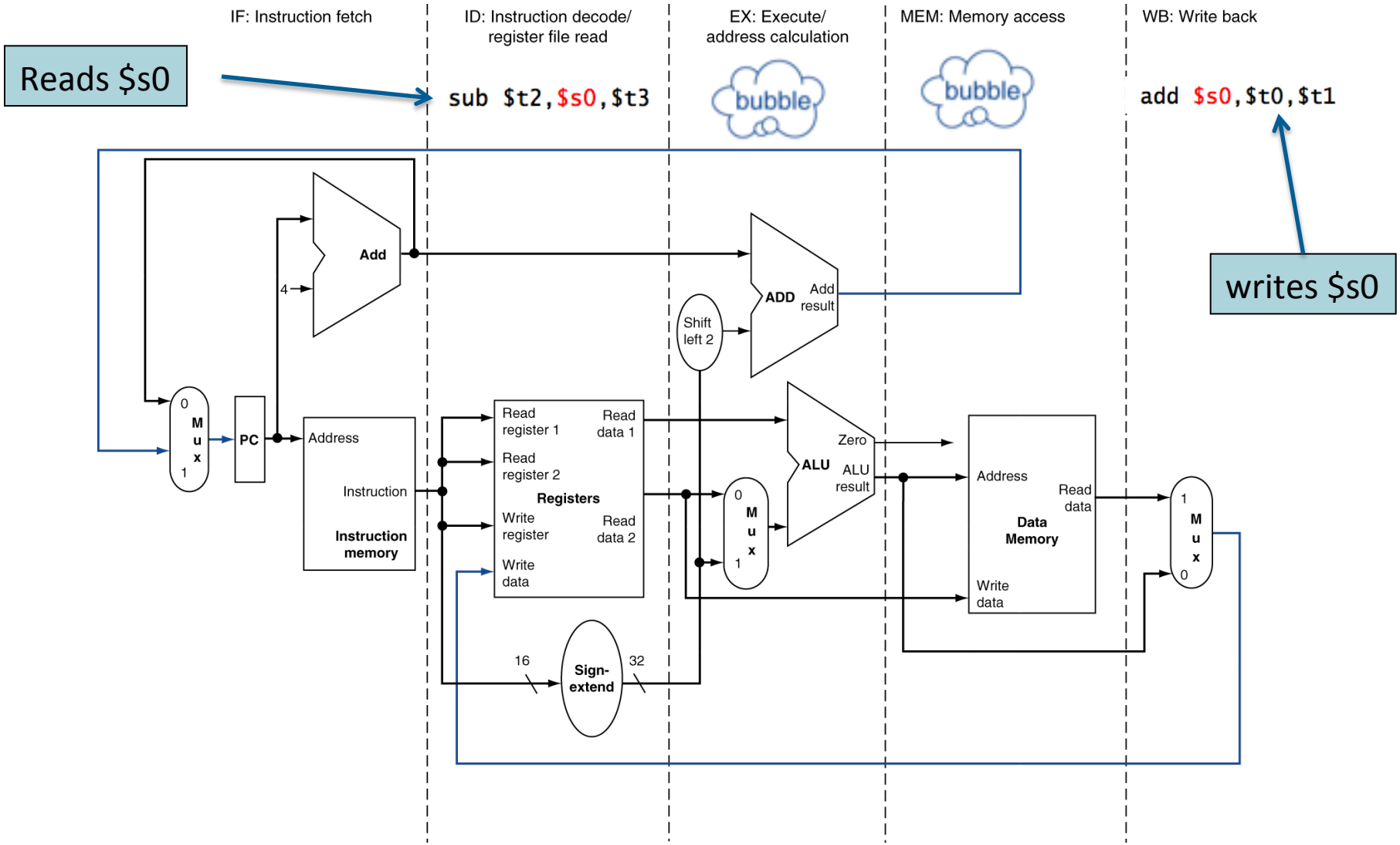
Required value has not been written to $s0

Data Hazards



Registers are read in stage 2 but not written until stage 5

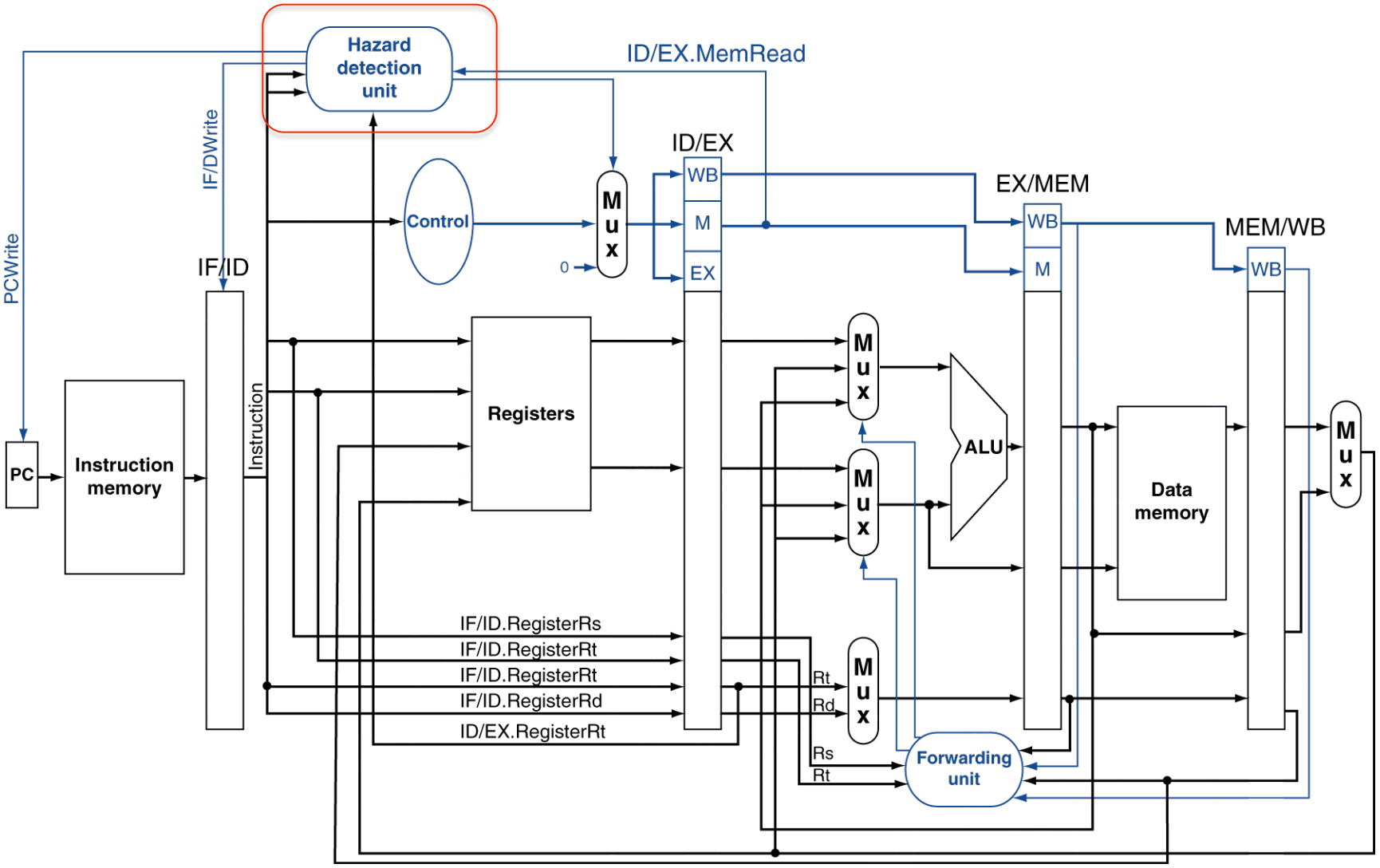
Stalling dependent instruction



Pipeline Interlock

- Stalling is also known as *pipeline interlock*
  - Instructions in front continue to advance
  - Creates empty stages or *bubbles*
  - Consumes 2 extra clock cycles
- Without stalling, dependent instructions use stale data
  - Extra hardware is required to implement stalls
  - Hazard detection unit controls when stalls occur

Hazard Detection



Hazard Detection

Instruction in decode may be dependent

- If either of its 2 input registers matches:
- reg written by instruction in EXE or in MEM stage
- Hazard detection unit outputs:
  - 0 to allow normal control signals to be passed
  - 1 to instead substitute zero control signals (i.e. bubble)
- Instructions must write a result to a cause data hazard
  - Neither sw nor beq write a result

Hazard detection unit operation

If (EX/MEM.RegWrite or MEM/WB.RegWrite)

- Does instruction in stage 4 or 5 write a register?

If (EX/MEM.Rd == ID/EX.Rs or EX/MEM.Rd == ID/EX.Rt) or
If (MEM/WB.Rd == ID/EX.Rs or MEM/WB.Rd == ID/EX.Rt)

- Result reg. match an input reg. for instruction in stage 2?

Hazard exits if both conditions are true

Example

```
add $s0,$t0,$t1
sub $t2,$s0,$t3
slt $t4,$t2,$0
```

- Hazard detection unit compares register numbers
- Sub must be stalled until add writes $s0
- Slt must be stalled until sub writes $t2
- Registers are written during the first half of a cycle
- Registers are read during the second half of a cycle
  - otherwise 3 rather than 2 bubbles would be required

Example

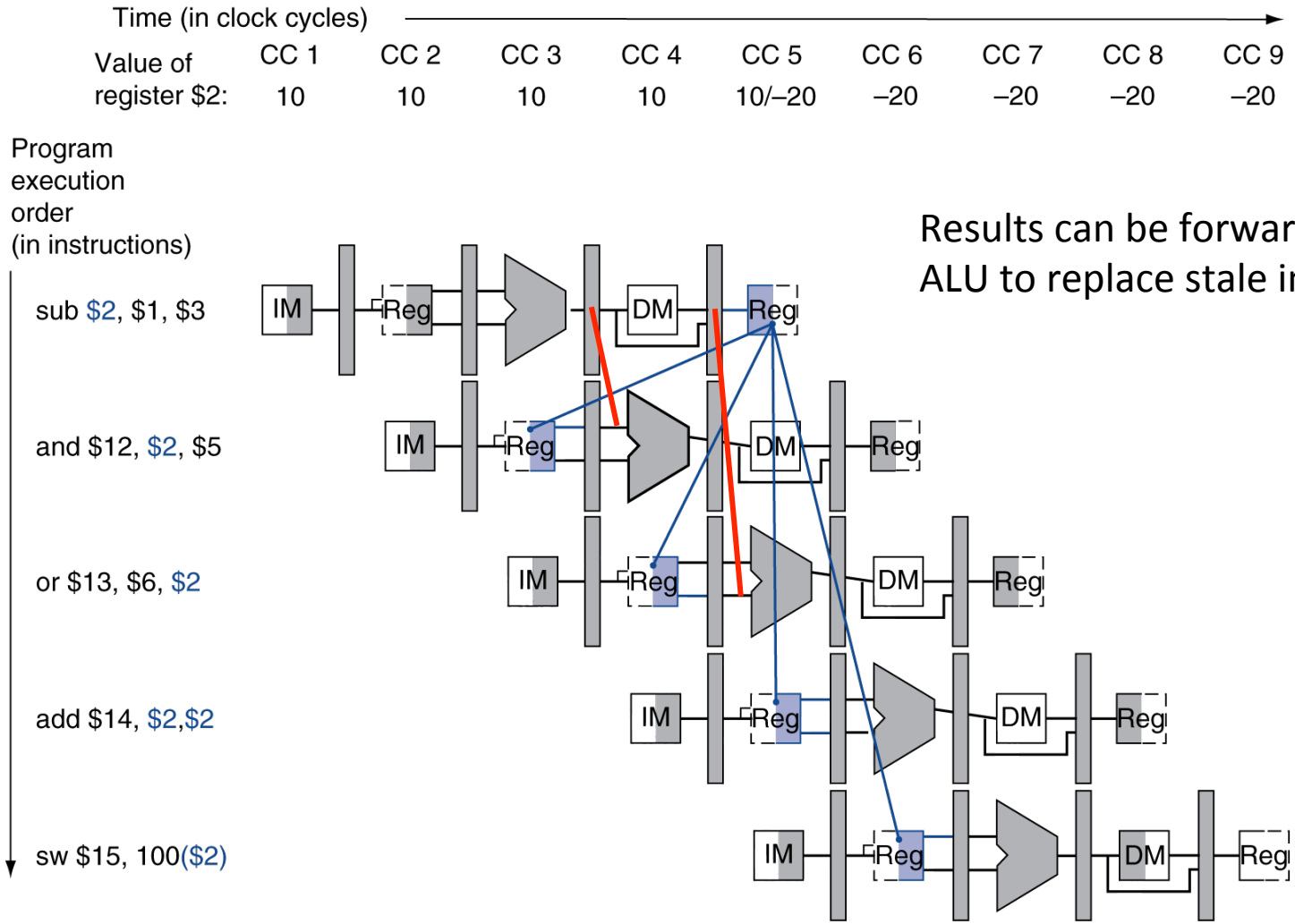
Cycle	IF	ID	EX	MEM	WB
1	add $s0,$t0,$t1				
2	sub $t2,$s0,$t3	add $s0,$t0,$t1			
3	slt $t4,$t2,$0	sub $t2,$s0,$t3	add $s0,$t0,$t1		
4	slt $t4,$t2,$0	sub $t2,$s0,$t3	bubble	add $s0,$t0,$t1	
5	slt $t4,$t2,$0	sub $t2,$s0,$t3	bubble	bubble	add $s0,$t0,$t1
6		slt $t4,$t2,$0	sub $t2,$s0,$t3	bubble	bubble
7		slt $t4,$t2,$0	bubble	sub $t2,$s0,$t3	bubble
8		slt $t4,$t2,$0	bubble	bubble	sub $t2,$s0,$t3
9			slt $t4,$t2,$0	bubble	bubble
10				slt $t4,$t2,$0	bubble
11					slt $t4,$t2,$0

Stalls add 4 extra cycles to the time required for these 3 instructions.

Alternatives to Stalls

- Software
  - Compiler inserts useful instructions
    - 2 independent instructions between dependent instructions (code rearrangement)
    - 2 NOP instructions if useful ones can't be found
- Hardware
  - Required value is forwarded to dependent instruction
  - Just-in-time replacement of stale ALU inputs

Dependencies & Forwarding



Results can be forwarded to the ALU to replace stale inputs

Detecting the Need to Forward

- Pass register numbers along pipeline
  - 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
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

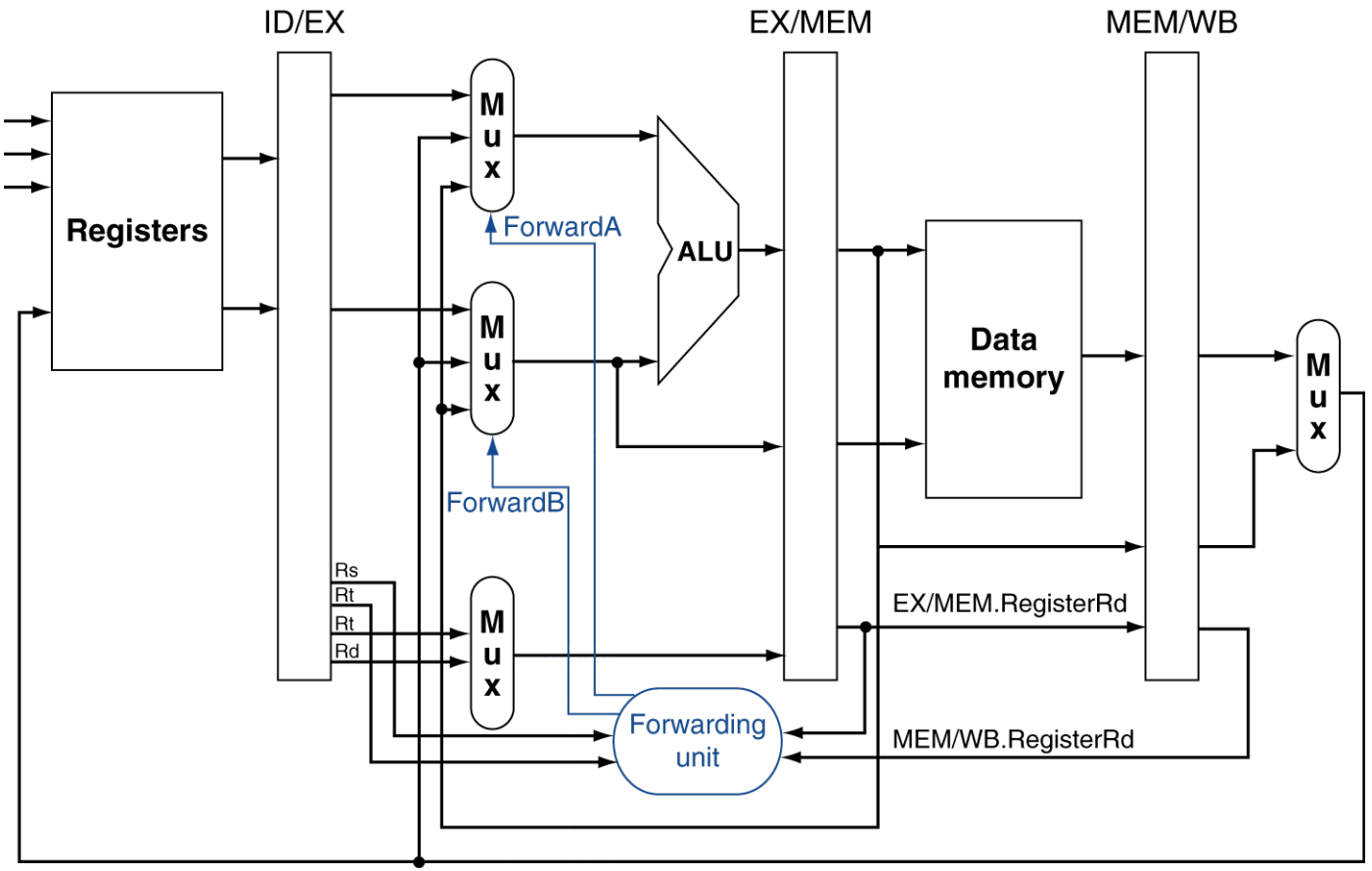
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- Only if forwarding instruction writes a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0,
    MEM/WB.RegisterRd ≠ 0

Forwarding Paths



Forwarding replaces stale ALU inputs

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Forwarding Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	First ALU operand comes from the register file.
ForwardA = 01	MEM/WB	First ALU operand is forwarded from data memory or earlier ALU result.
ForwardA = 10	EX/MEM	First ALU operand comes from the prior ALU result.
ForwardB = 00	ID/EX	Second ALU operand comes from the register file.
ForwardB = 01	MEM/WB	Second ALU operand is forwarded from data memory or earlier ALU result.
ForwardB = 10	EX/MEM	Second ALU operand comes from the prior ALU result.

Forwarding within the Pipeline

- 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
  - 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
  - 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

Forwarding within the Pipeline

This sequence will demonstrate how forwarding works

```
sub     $2, $1, $3
and     $4, $2, $5
or      $4, $4, $2
add     $9, $4, $2
```

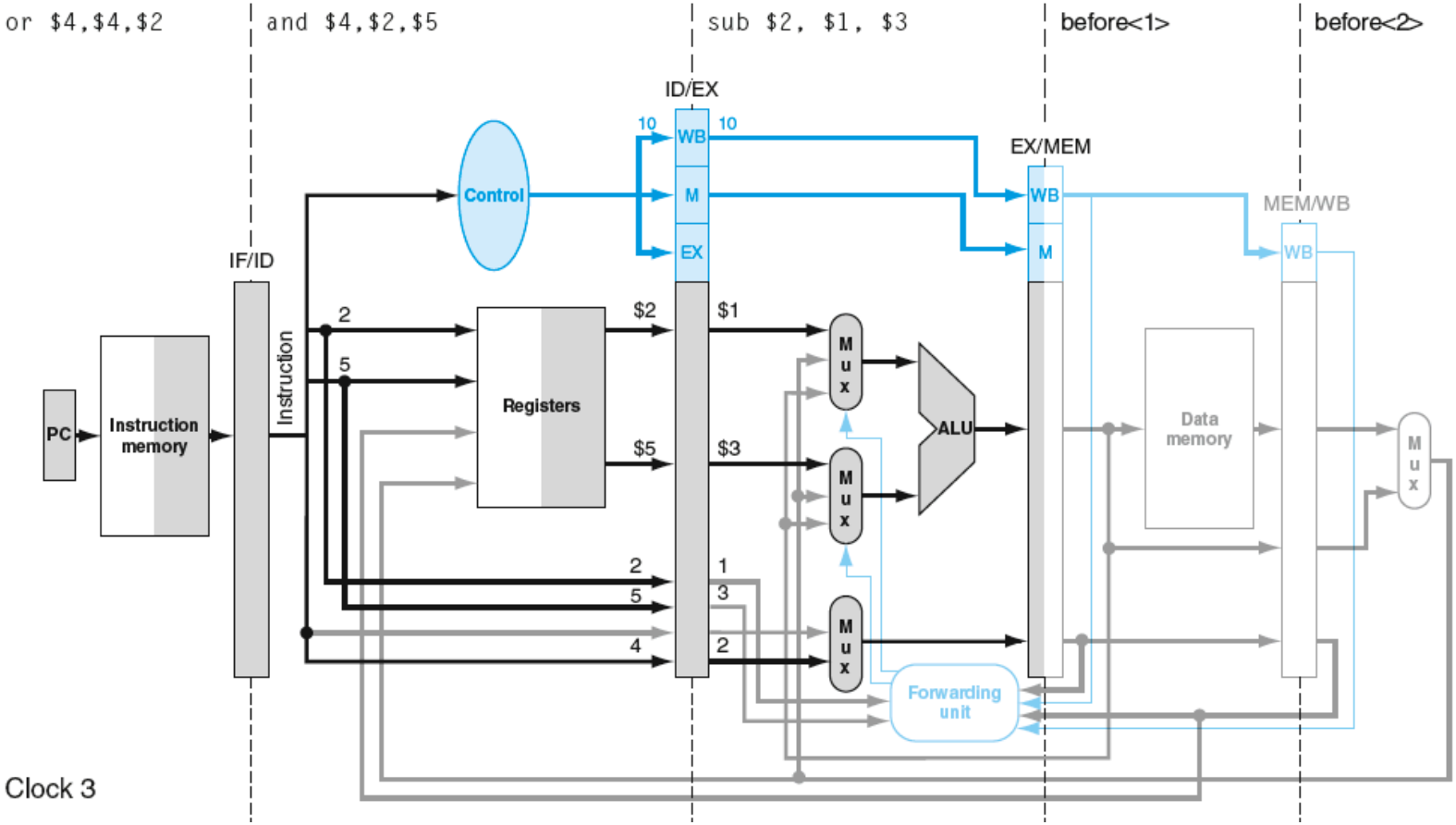
The SUB produces a result in $2 needed by the AND

The result in $4 produced by AND in needed by the OR

OR updates $4 which is then used by ADD

Forwarding within the Pipeline

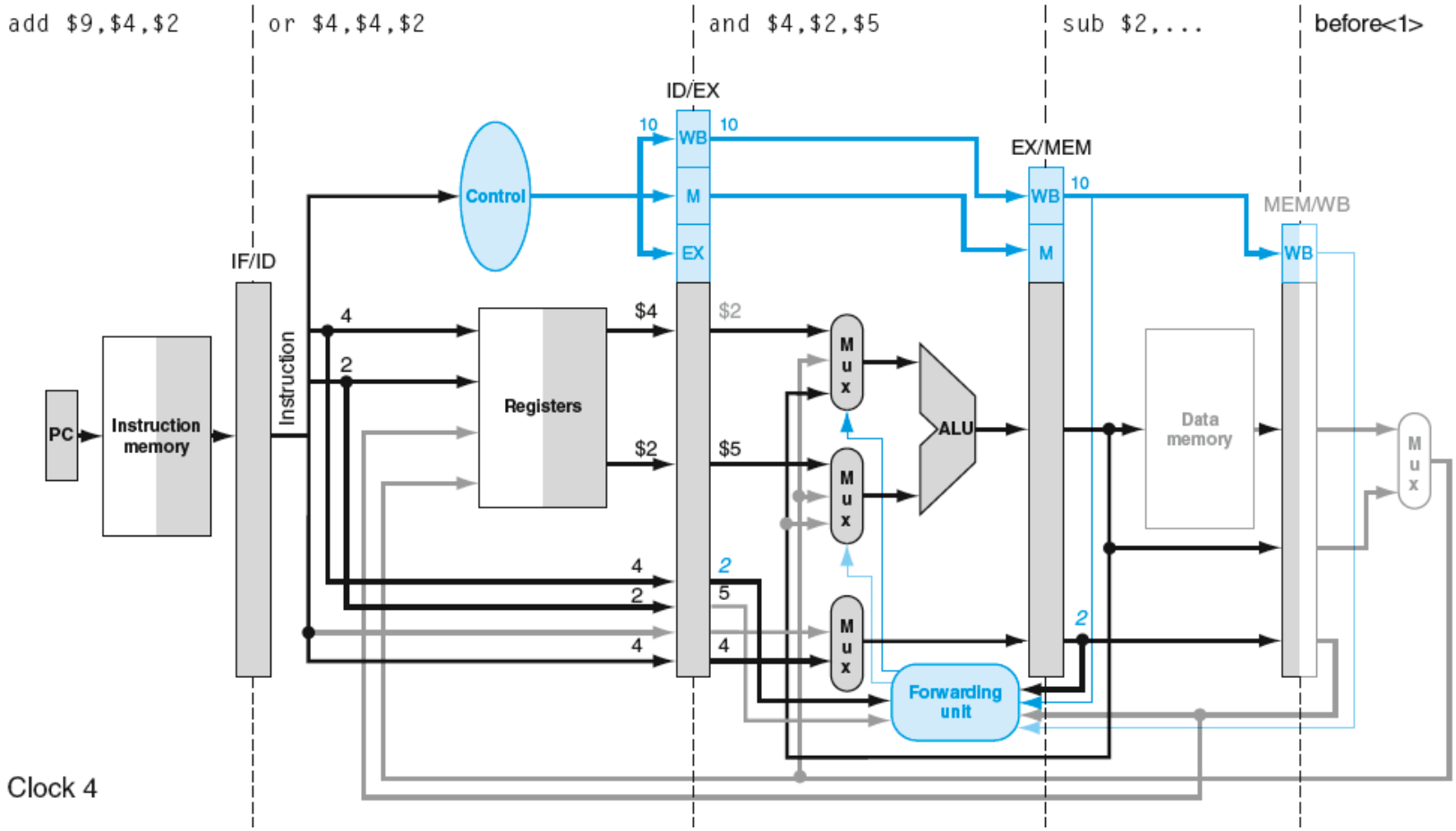
By cycle 3, the pipeline contains:



The decode stage is where instructions read their input registers
The write-back stage is where instructions write their result registers

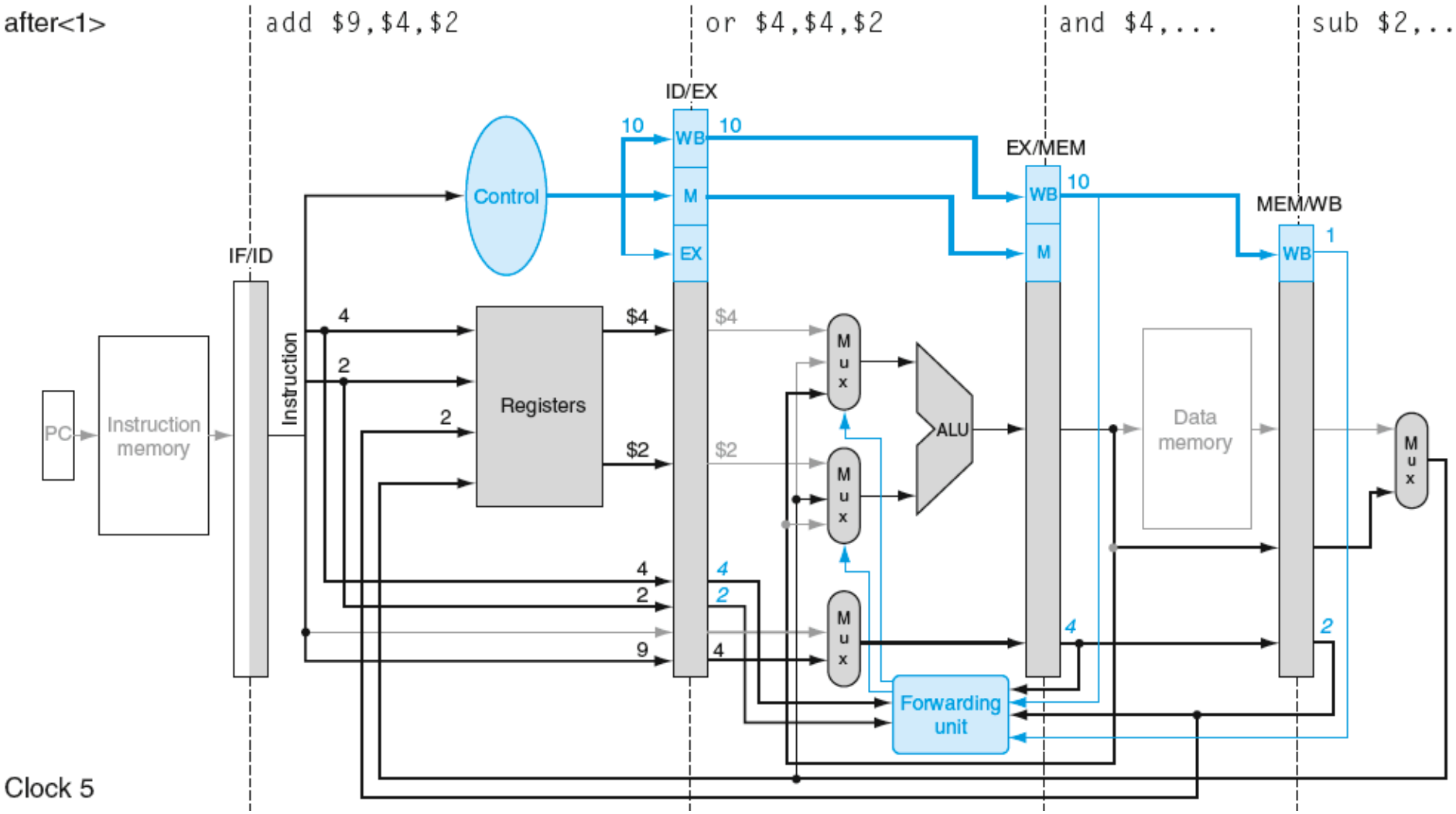
Forwarding within the Pipeline

By cycle 4, the pipeline contains:



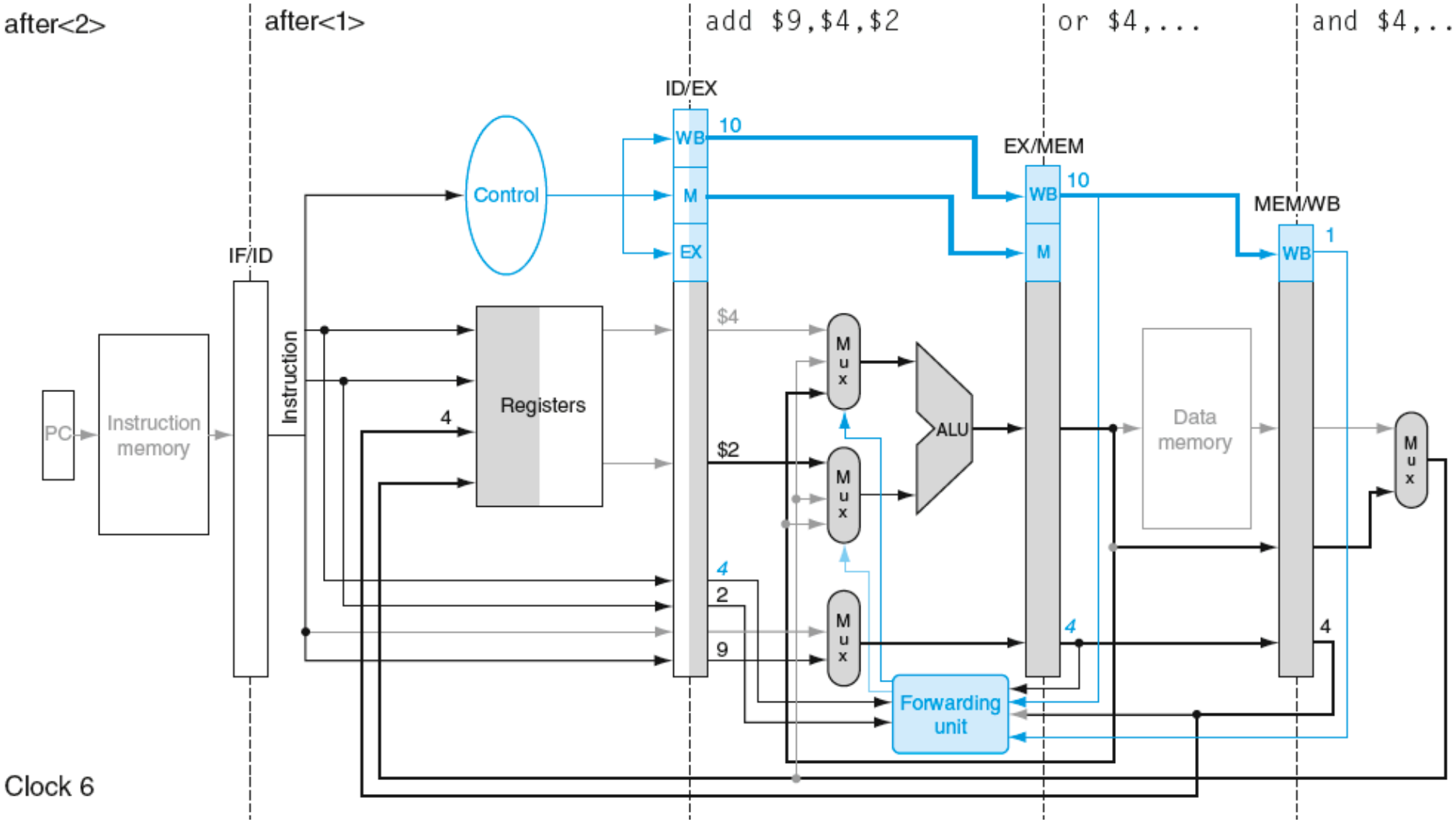
The upper ALU input for the AND is replaced by the forwarded value from EX/MEM ($2)

Forwarding within the Pipeline



The upper ALU input for the OR is replaced by the forwarded value from EX/MEM

Forwarding within the Pipeline



The upper ALU input for the ADD is replaced by the forwarded value from EX/MEM

Forwarding within the Pipeline

With forwarding, fewer cycles are consumed

- The above sequence completes in 8 cycles
- It takes 14 cycles without forwarding

Without forwarding, stalls are needed (bubbles)

- Dependent instruction is held in decode stage
- Instruction producing result must reach stage5
- Register writes occur in first half of clock cycle
- Register reads occur in second half of clock cycle
- Otherwise one extra stall would be needed

Forwarding within the Pipeline

Forwarding avoids having to stall dependent instructions

- By just-in-time replacement of stale ALU inputs

Some instructions cannot cause data hazards

- These instruction don't write a result register
- Sw writes to memory, not to a register
- Beq compares 2 registers without writing either

Load Delays

Forwarding avoids having to stall dependent instructions

- Except for values read from memory
- *Delay slot* is the slot following a load instruction
- Instructions in the delay slot can't use the load result
- otherwise they must be stalled for one cycle

Hazard detection unit is still required for this special case

Load Delays

This sequence illustrates the effect of a load delay:

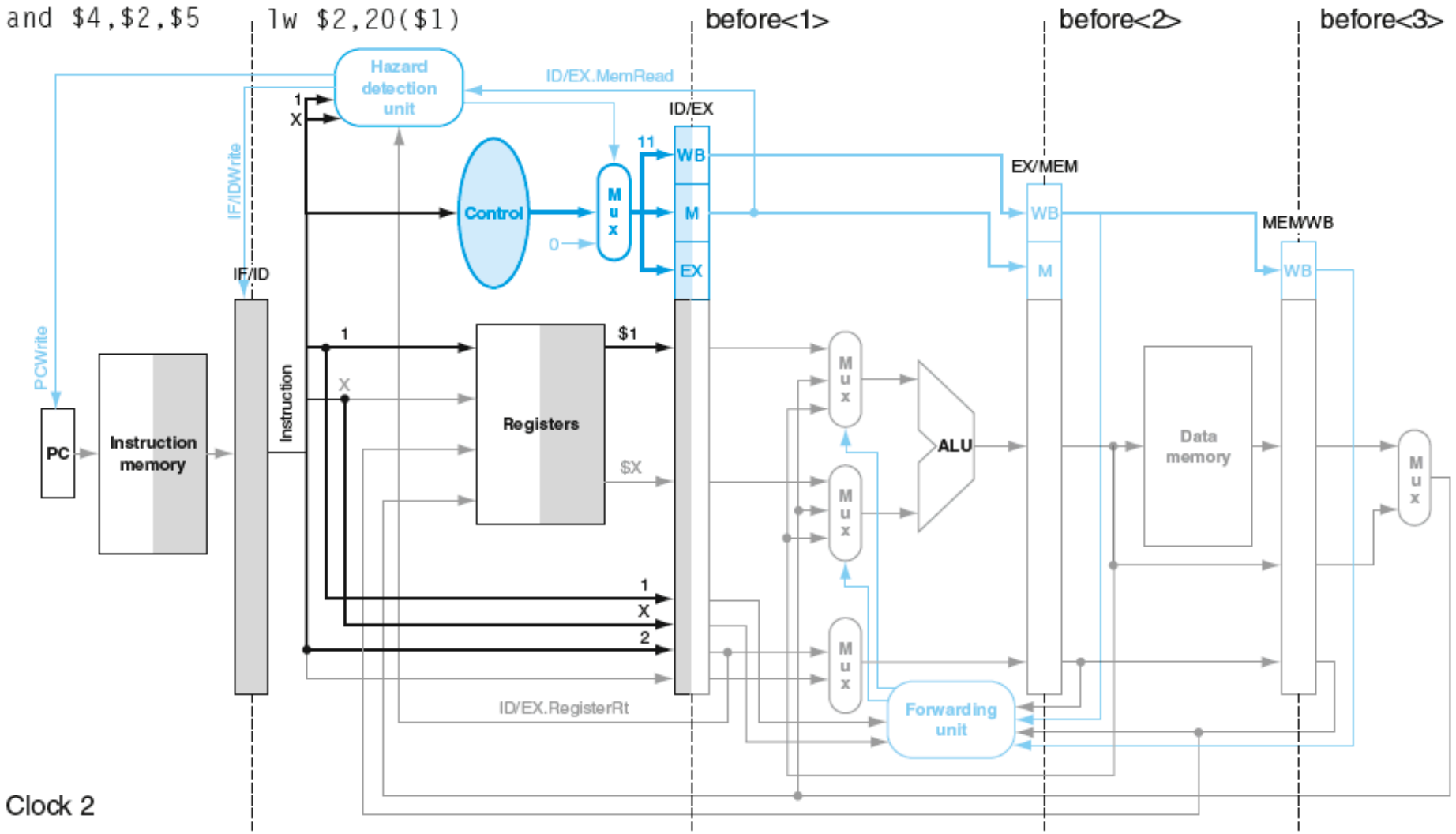
```
lw      $2, 20($1)
and     $4, $2,$5
or      $4, $4,$2
add     $9, $4,$2
```

The AND needs the result in $2 read from memory by LW

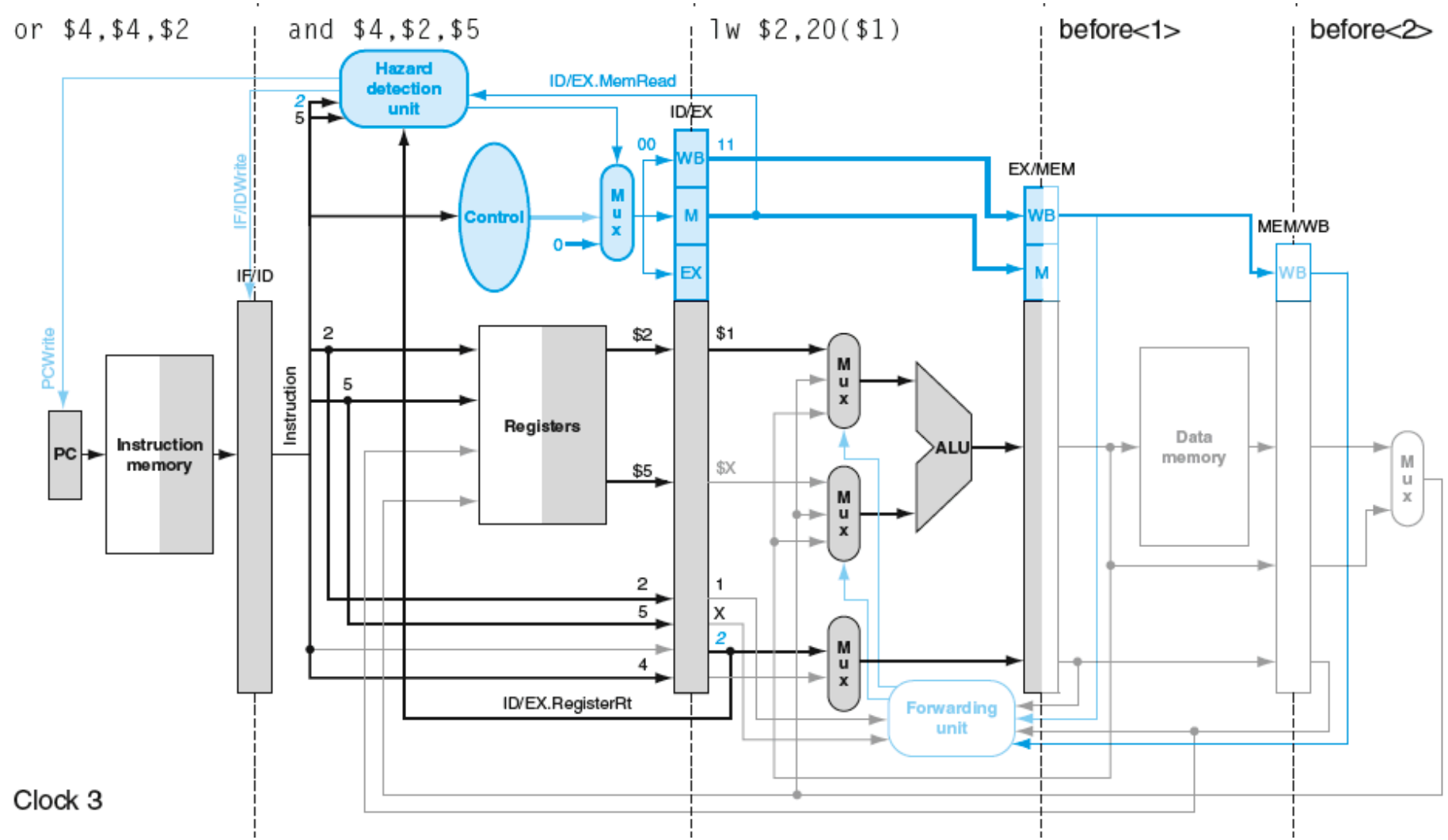
The result in $4 produced by AND in needed by the OR

OR updates $4 which is then used by ADD

Load Delays

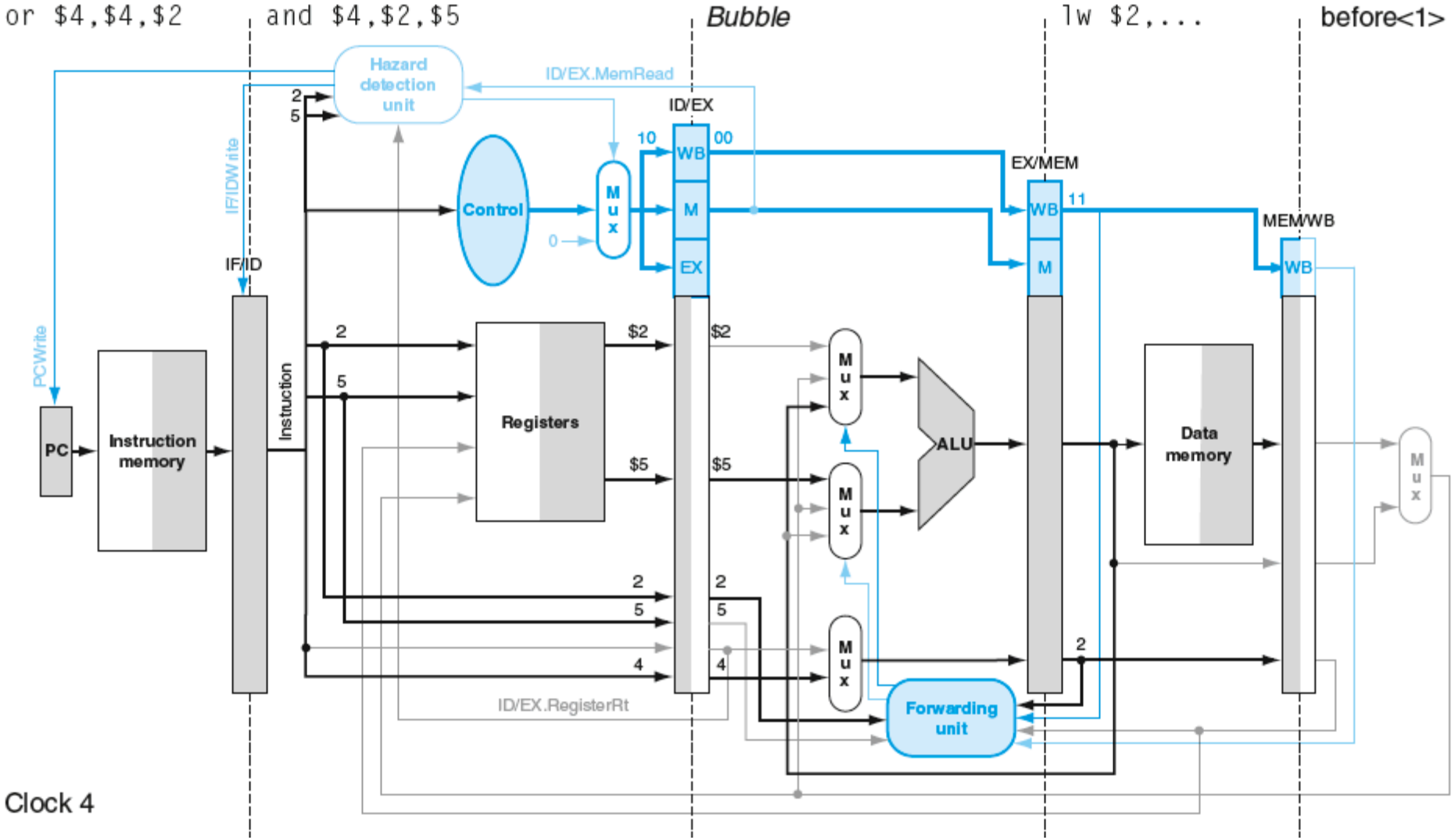

and $4,$2,$5
lw $2,20($1)
before<1>
before<2>
before<3>
Hazard detection unit
ID/EX.MemRead
ID/EX
IF/IDWrite
Control
Mux
WB
M
EX
EX/MEM
MEM/WB
IF/ID
PCWrite
PC
Instruction memory
Instruction
Registers
$1
$X
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 2

Load Delays



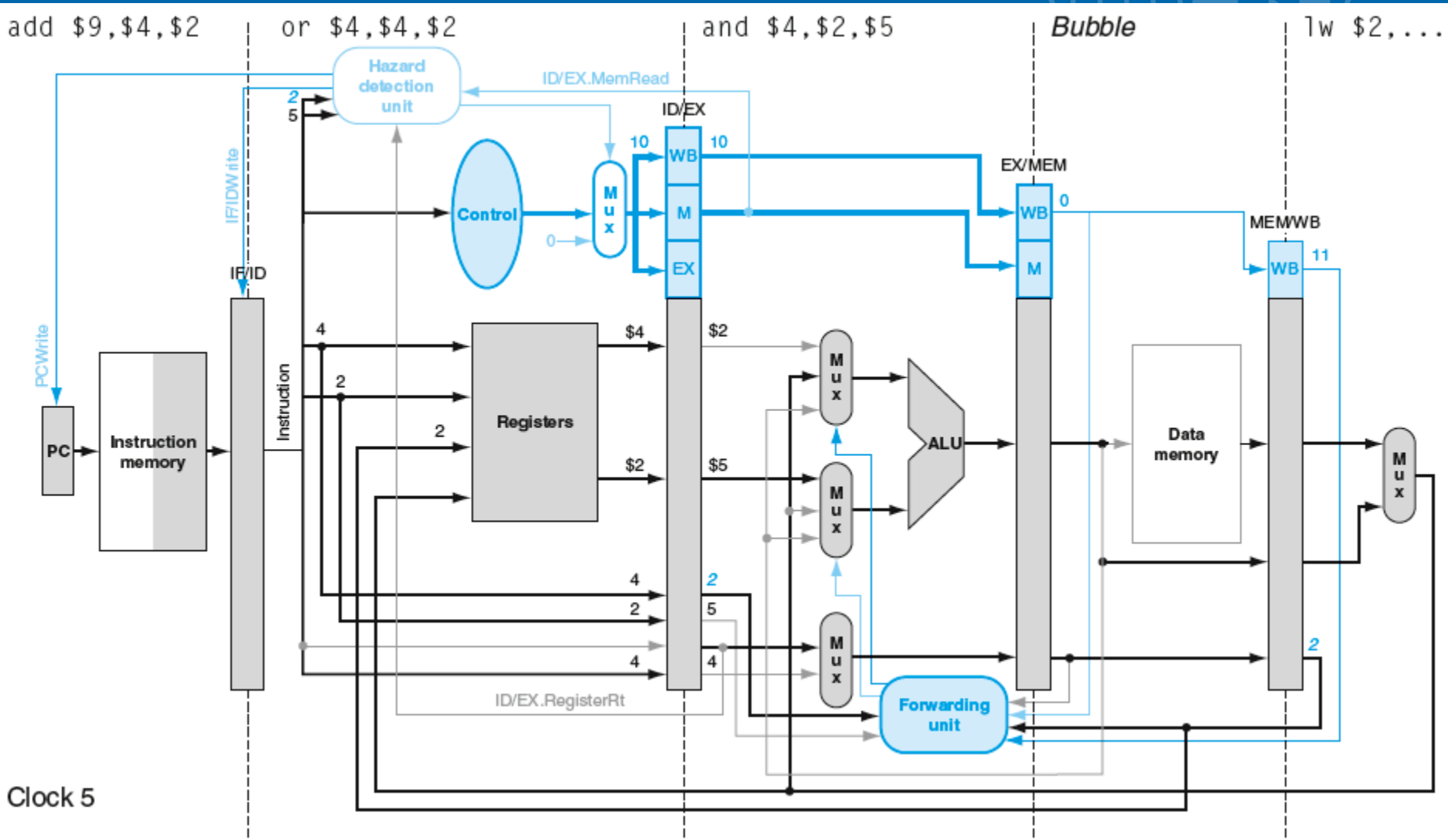
AND uses $2, so it must be stalled to allow LW time to read from the data memory

Load Delays



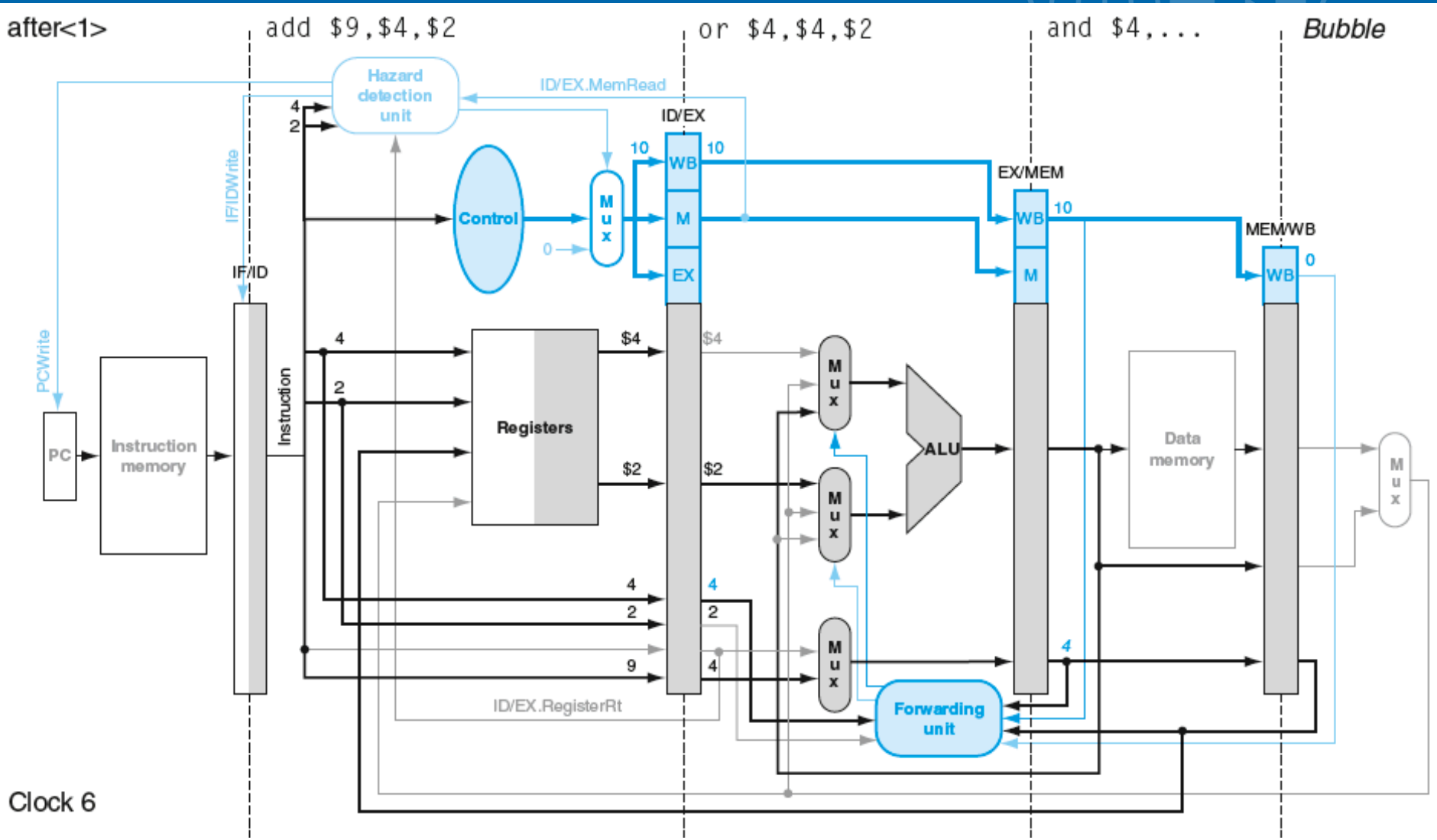
Bubble occupies stage 3 while LW reads from the data memory

Load Delays



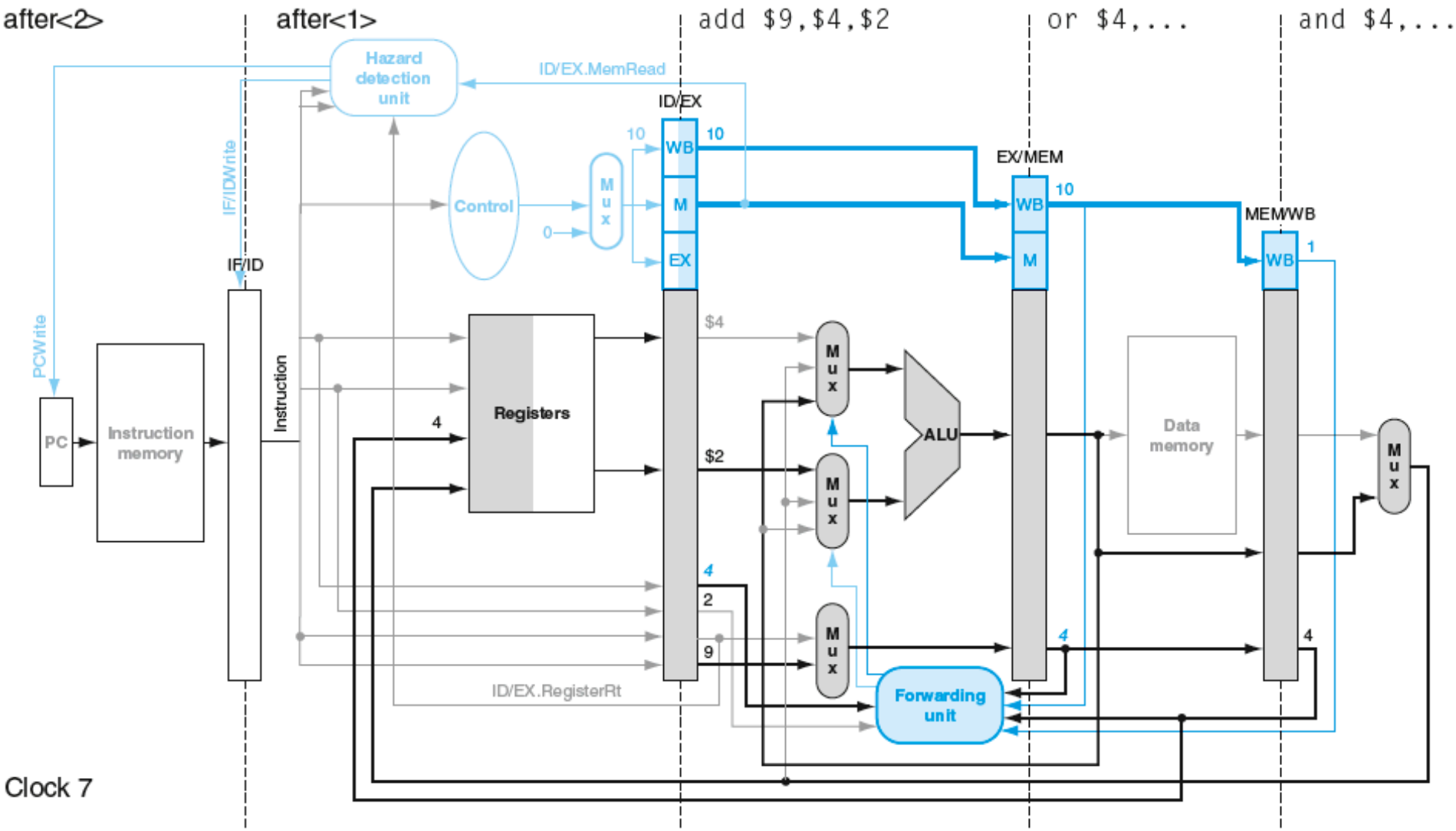
AND receives forwarded value from MEM/WB pipeline register in cycle 5

Load Delays



Forwarding takes care of the dependencies for the OR and ADD instructions.

Load Delays



All instructions complete by cycle 9.

Load Delays

The stall due to load delay added one extra cycle

Some compilers rearrange machine instructions

- This can avoid extra cycles due to stalls
- Rearrangement must not change program behavior
- Assembly language programmers can fill delay slot
- NOP instruction can be used as last resort