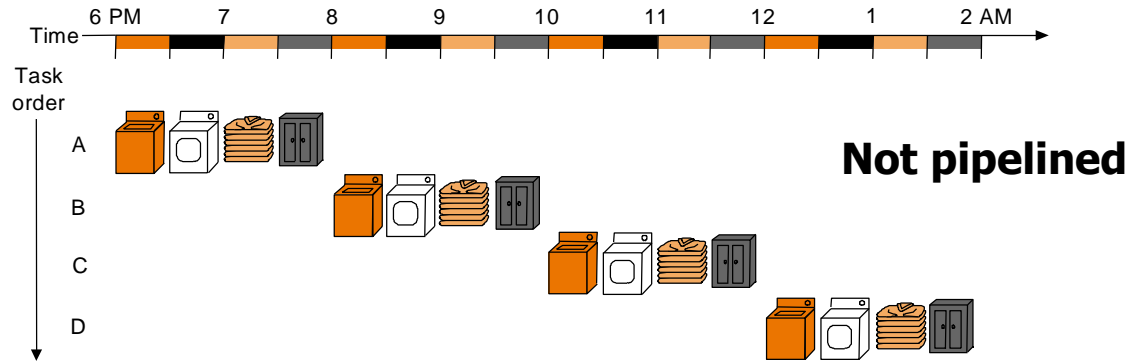


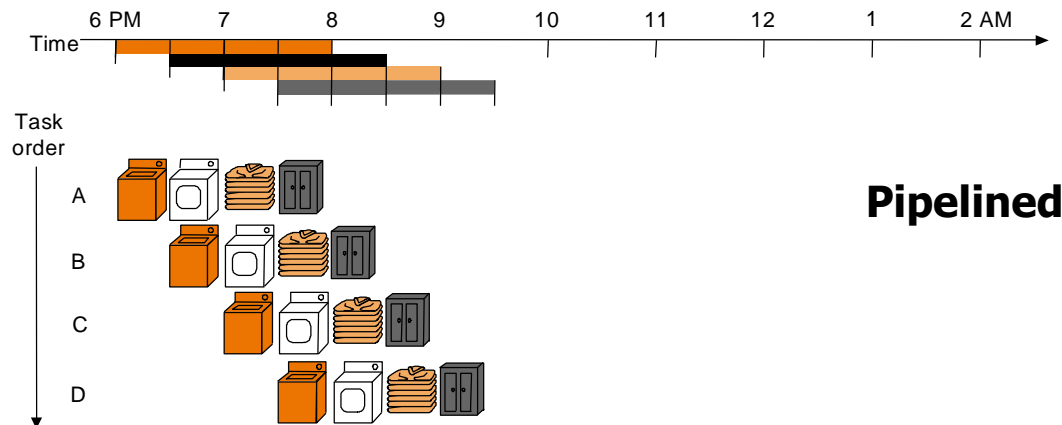
# Enhancing Performance with Pipelining

# Pipelining

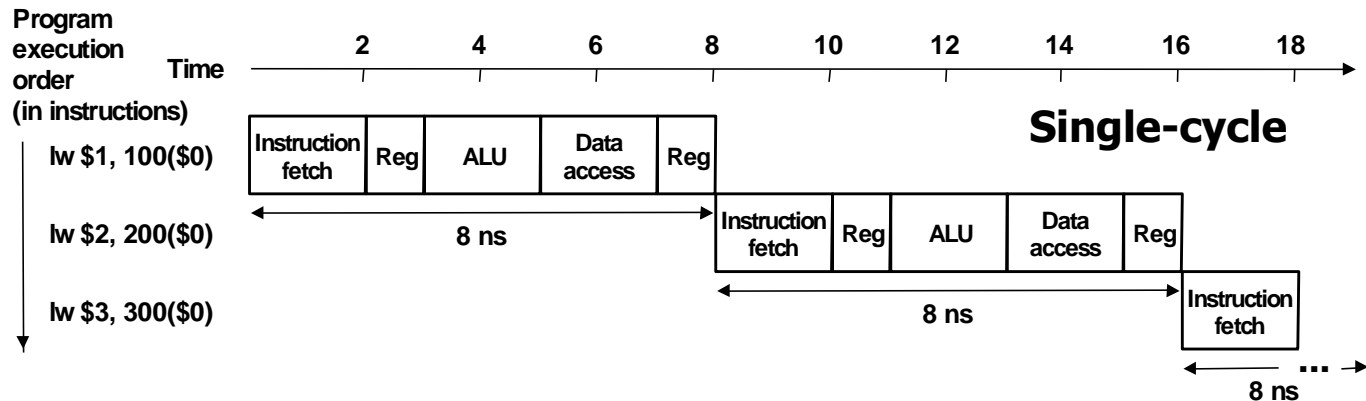
- Start work ASAP!! Do not waste time!



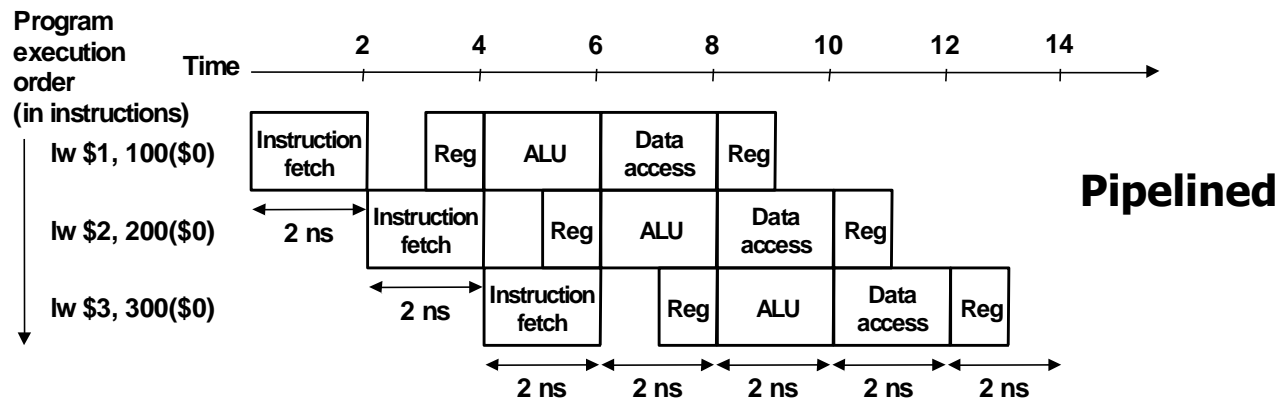
**Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped**



# Pipelined vs. Single-Cycle Instruction Execution: the Plan



**Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.**



# Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

# Pipelining MIPS

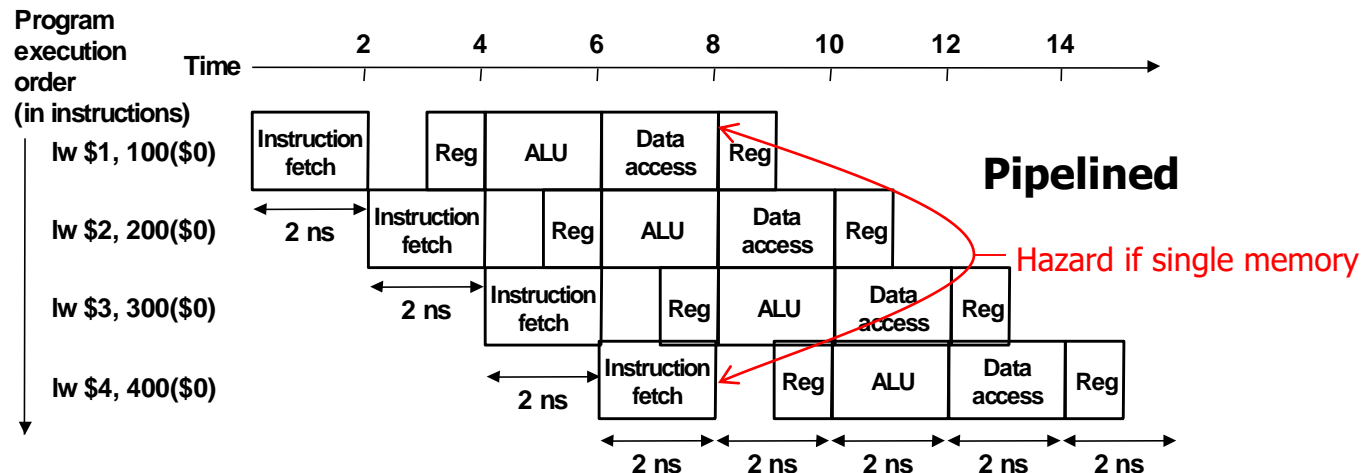
- What makes it easy with MIPS?
  - *all instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - *just a few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

# Pipelining MIPS

- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline
- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually...

# Structural Hazards

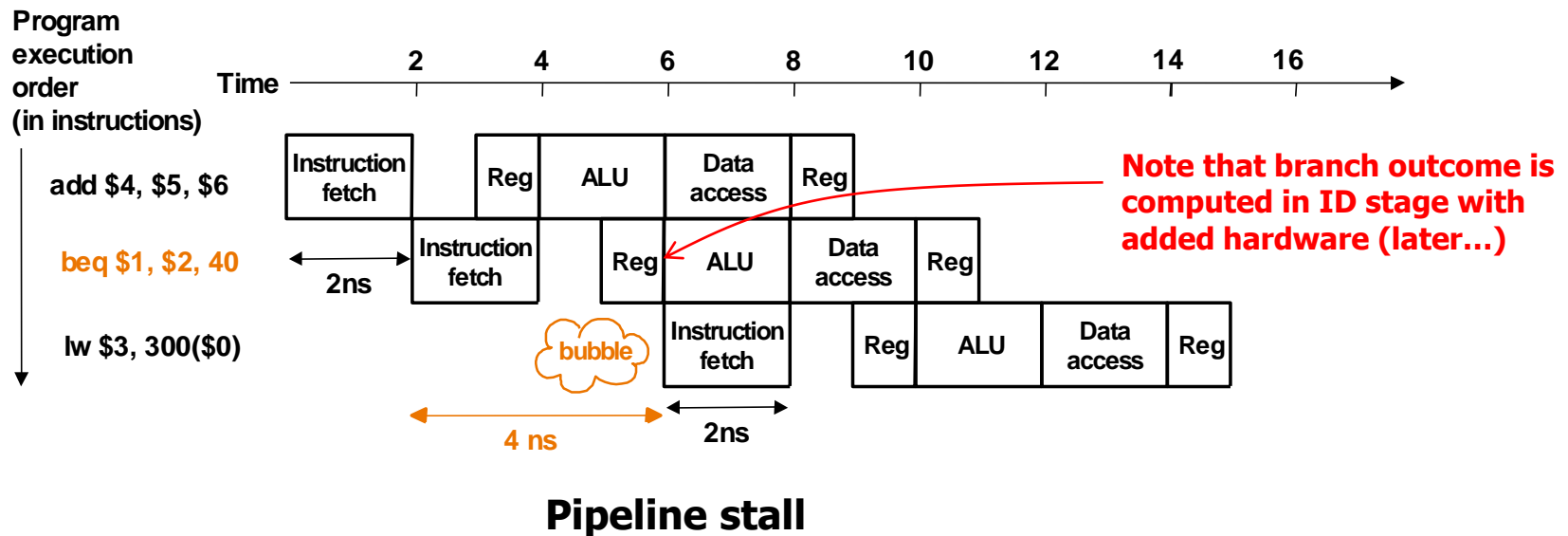
- **Structural hazard:** inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth  $lw$  instructions



- *MIPS was designed to be pipelined:* structural hazards are easy to avoid!

# Control Hazards

- **Control hazard:** need to make a decision based on the result of a previous instruction still executing in pipeline
- **Solution 1** *Stall* the pipeline

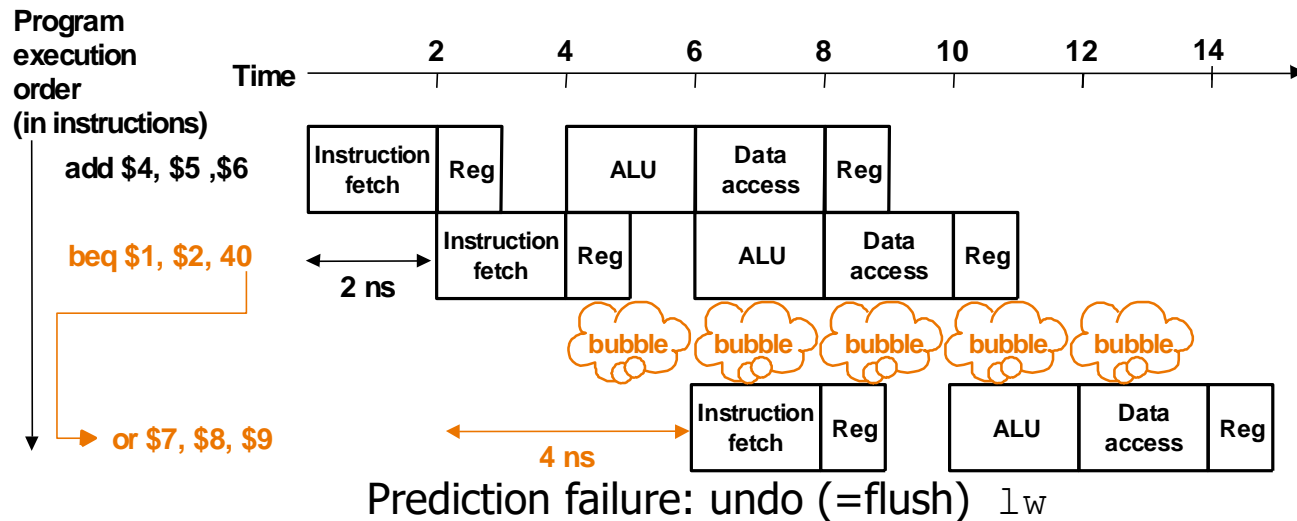
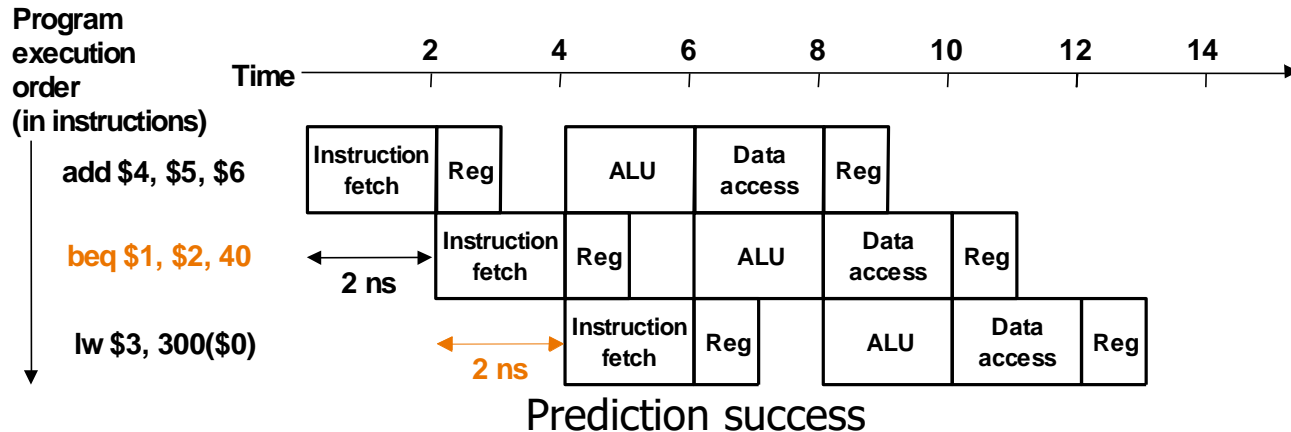




# Control Hazards

- Solution 2 *Predict* branch outcome

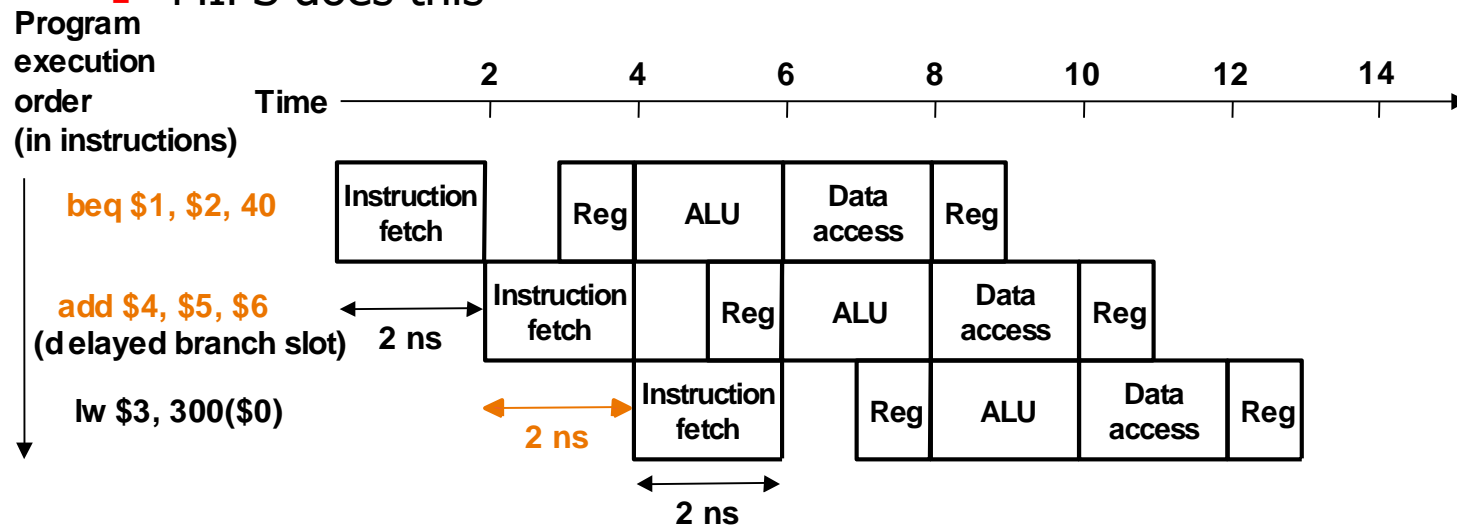
- e.g., predict *branch-not-taken* :



# Control Hazards

- Solution 3 Delayed branch: always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome

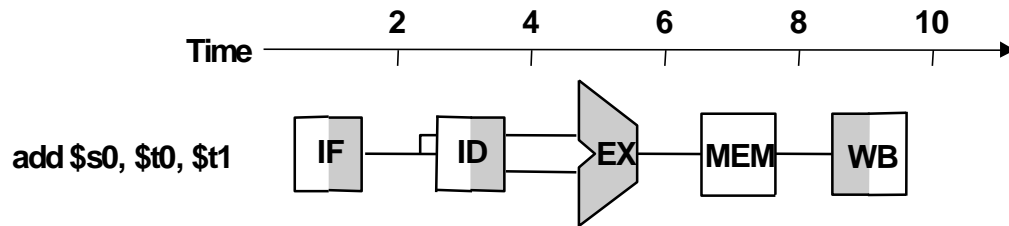
■ MIPS does this



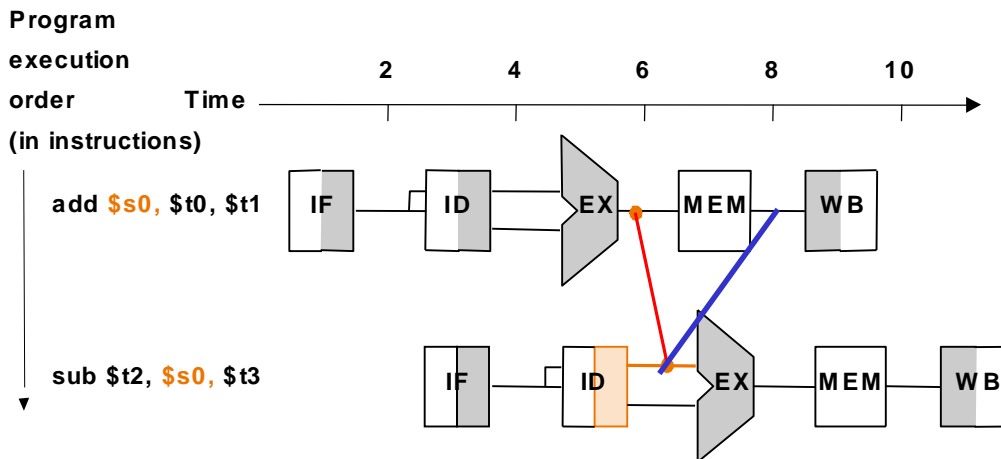
**Delayed branch beq is followed by add that is independent of branch outcome**

# Data Hazards

- **Data hazard:** instruction needs data from the result of a previous instruction still executing in pipeline
- Solution *Forward* data if possible...



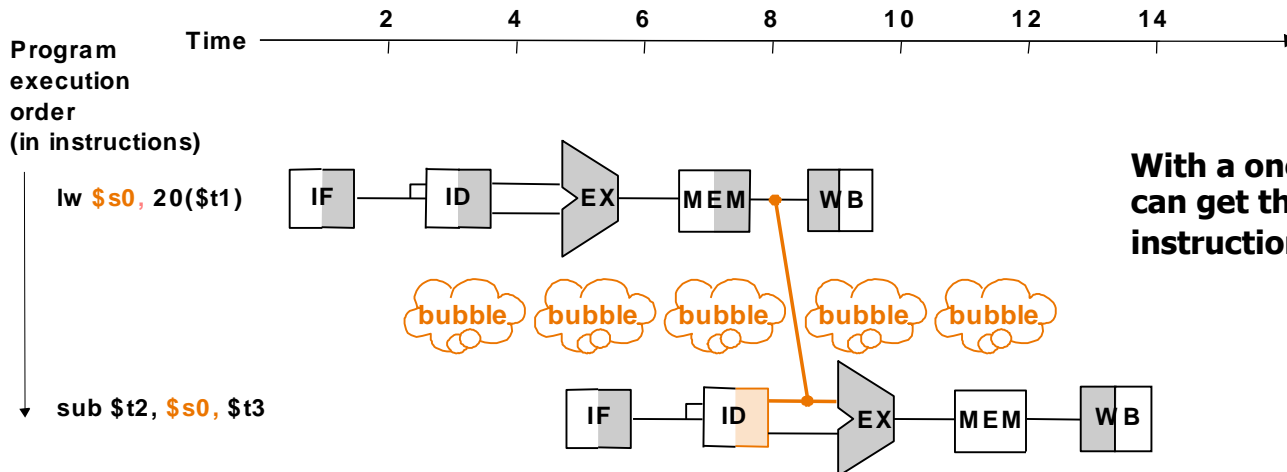
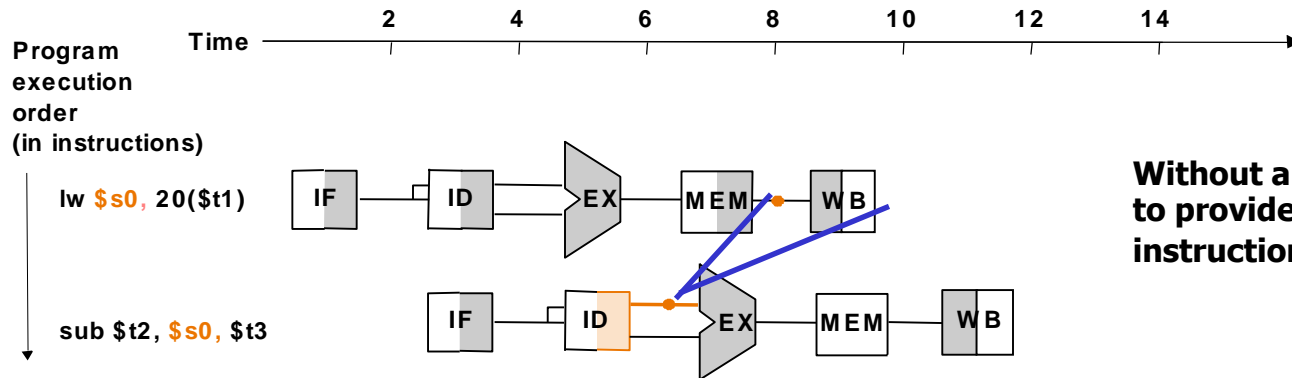
Instruction pipeline diagram:  
shade indicates use –  
**left=write, right=read**



**Without forwarding – blue line –**  
data has to go back in time;  
**with forwarding – red line –**  
data is available in time

# Data Hazards


- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called *load-use data hazard*



# Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```




A red curved arrow points from the `4($t1)` in the second instruction (`lw $t2, 4($t1)`) to the `0($t1)` in the third instruction (`sw $t2, 0($t1)`), indicating a data hazard where the second instruction writes to a register that the third instruction reads from before it has been updated.

Data hazard

- Reordered code:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

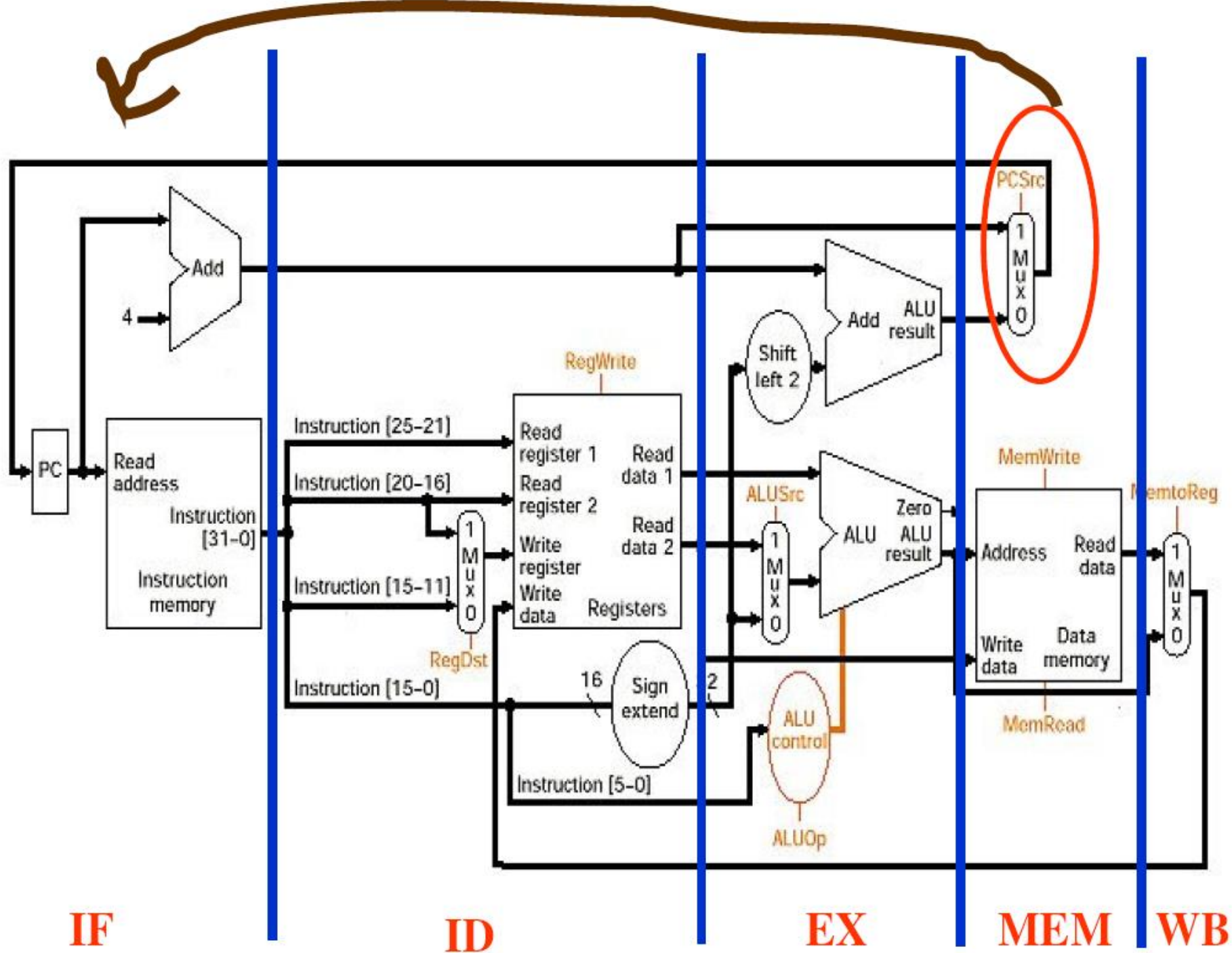


A red curved arrow points from the `4($t1)` in the third instruction (`sw $t0, 4($t1)`) to the `0($t1)` in the fourth instruction (`sw $t2, 0($t1)`), indicating that the two instructions have been reordered to eliminate the hazard.

Interchanged

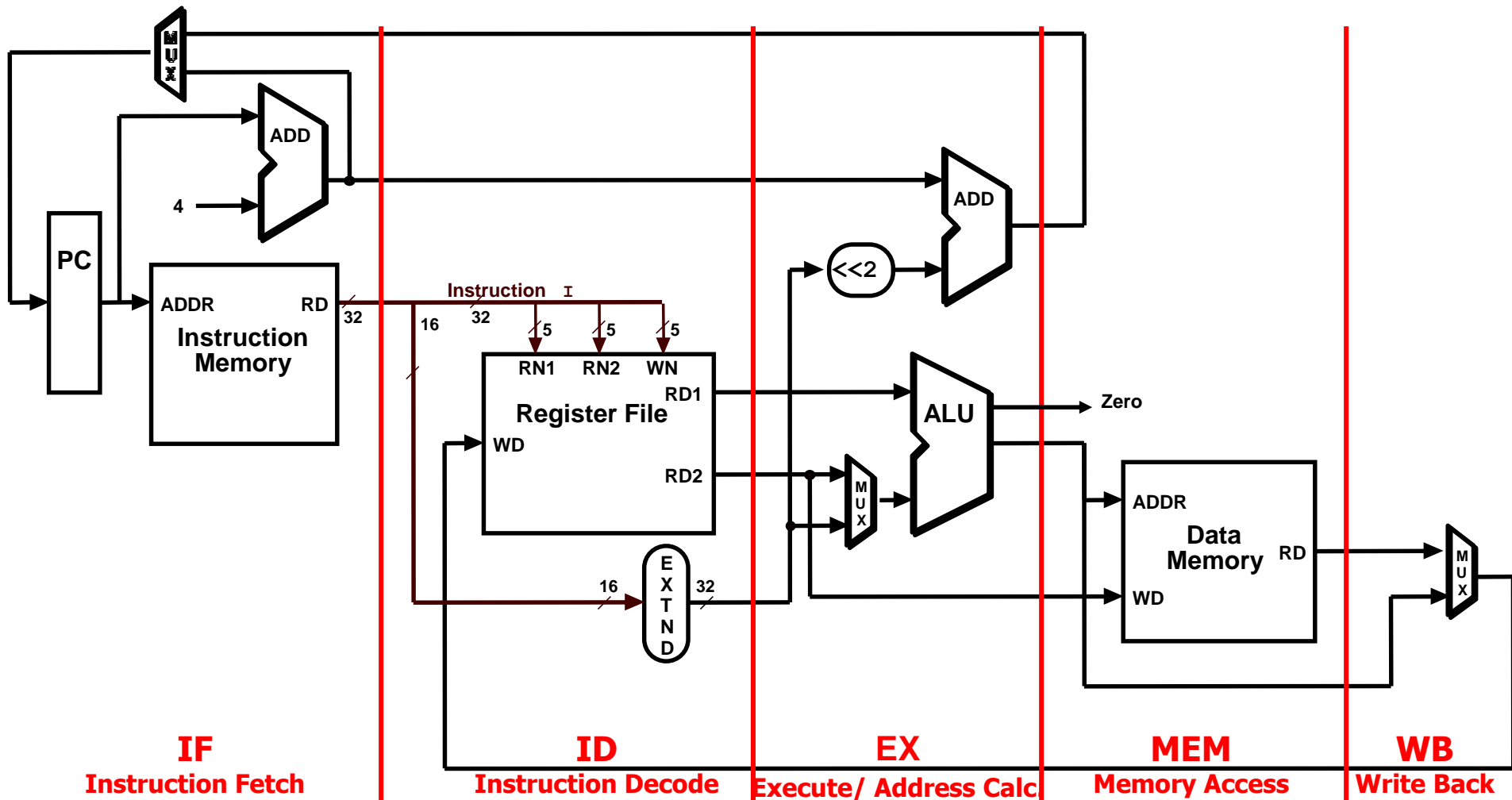
# Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step
- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*



# Review - Single-Cycle Datapath

## "Steps"

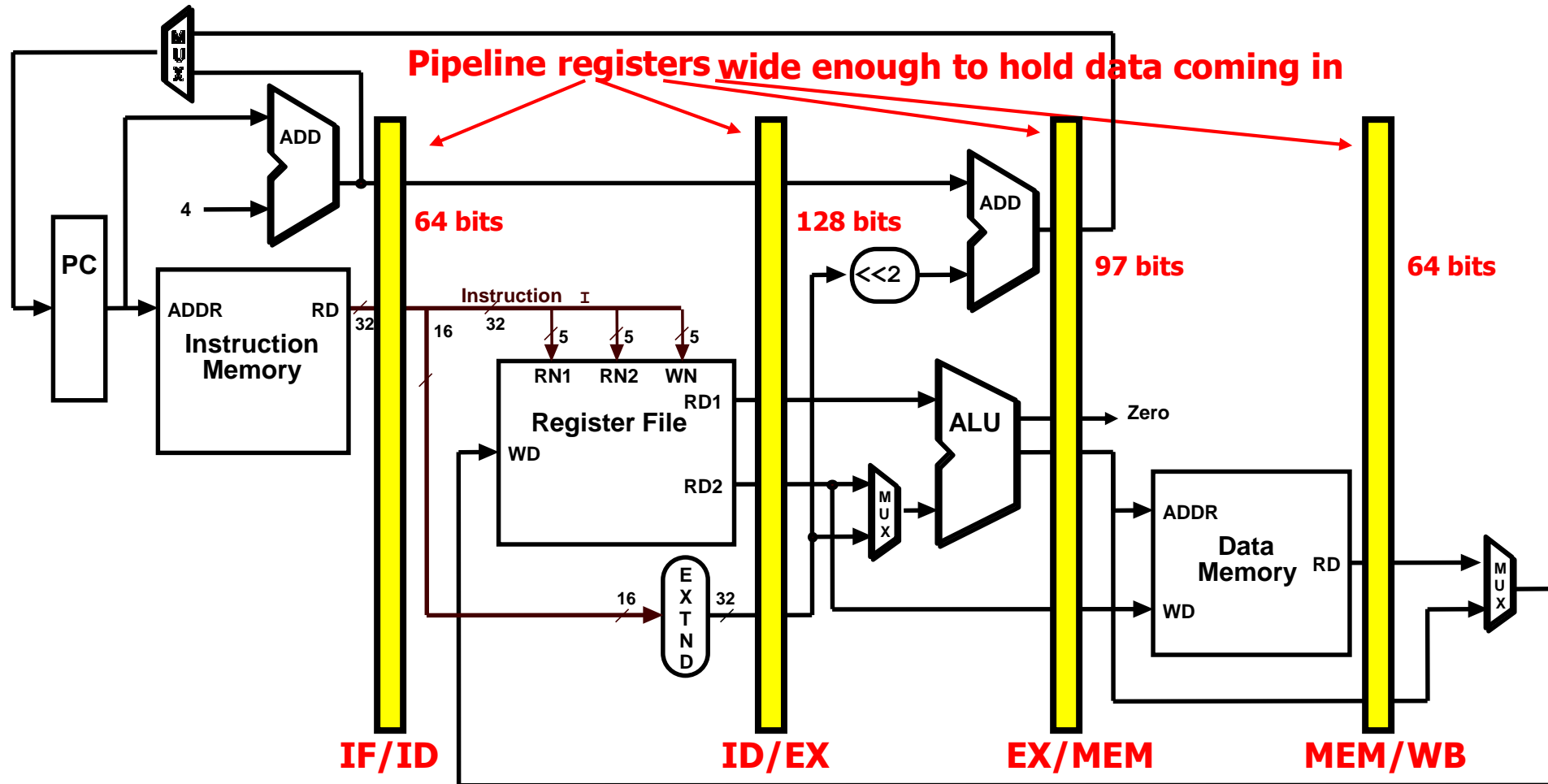




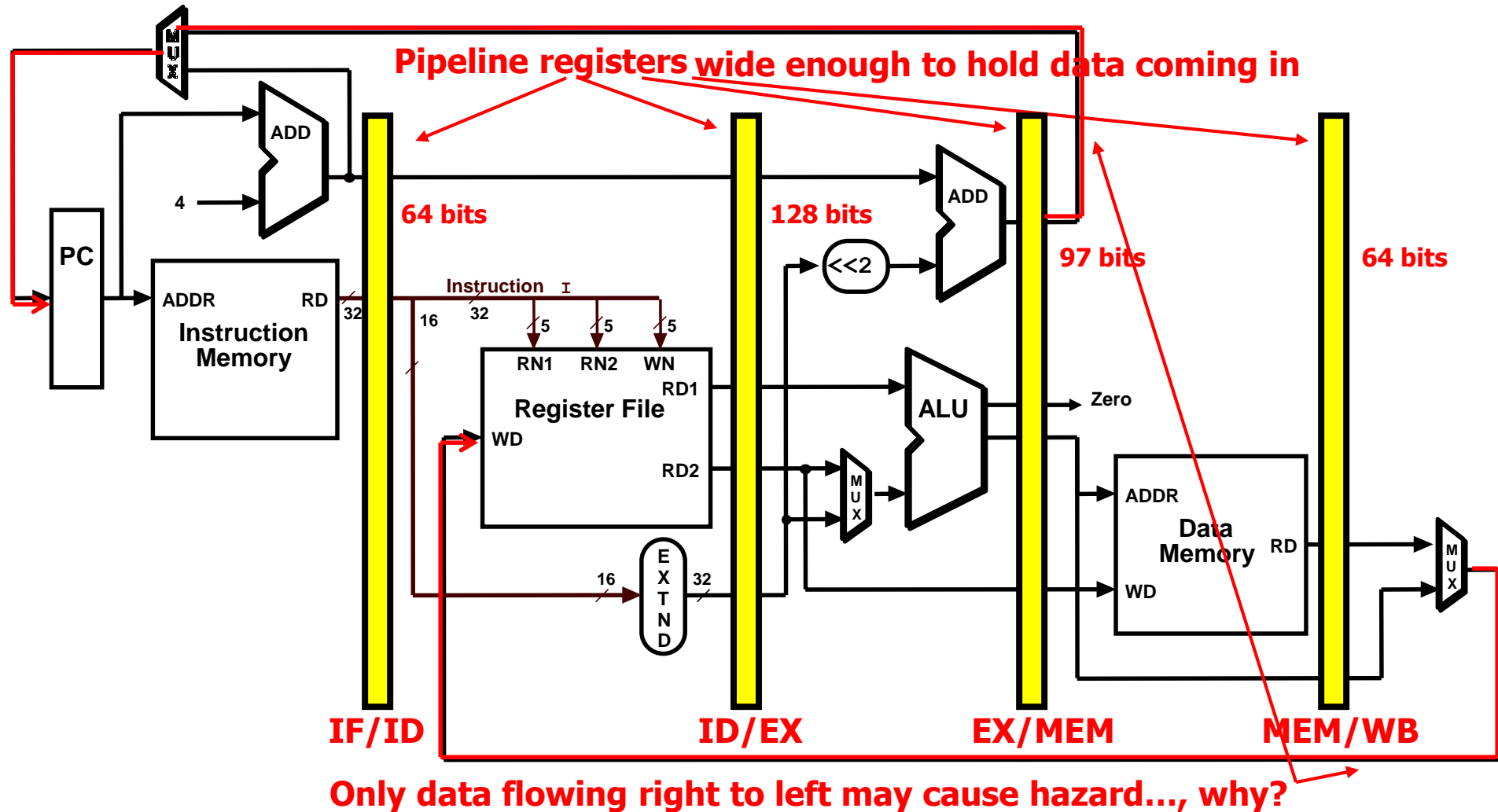
# Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - *Answer: We may be able to start executing a new instruction at each clock cycle - pipelining*
- ...but we shall need *extra* registers to hold data between cycles
  - *pipeline registers*

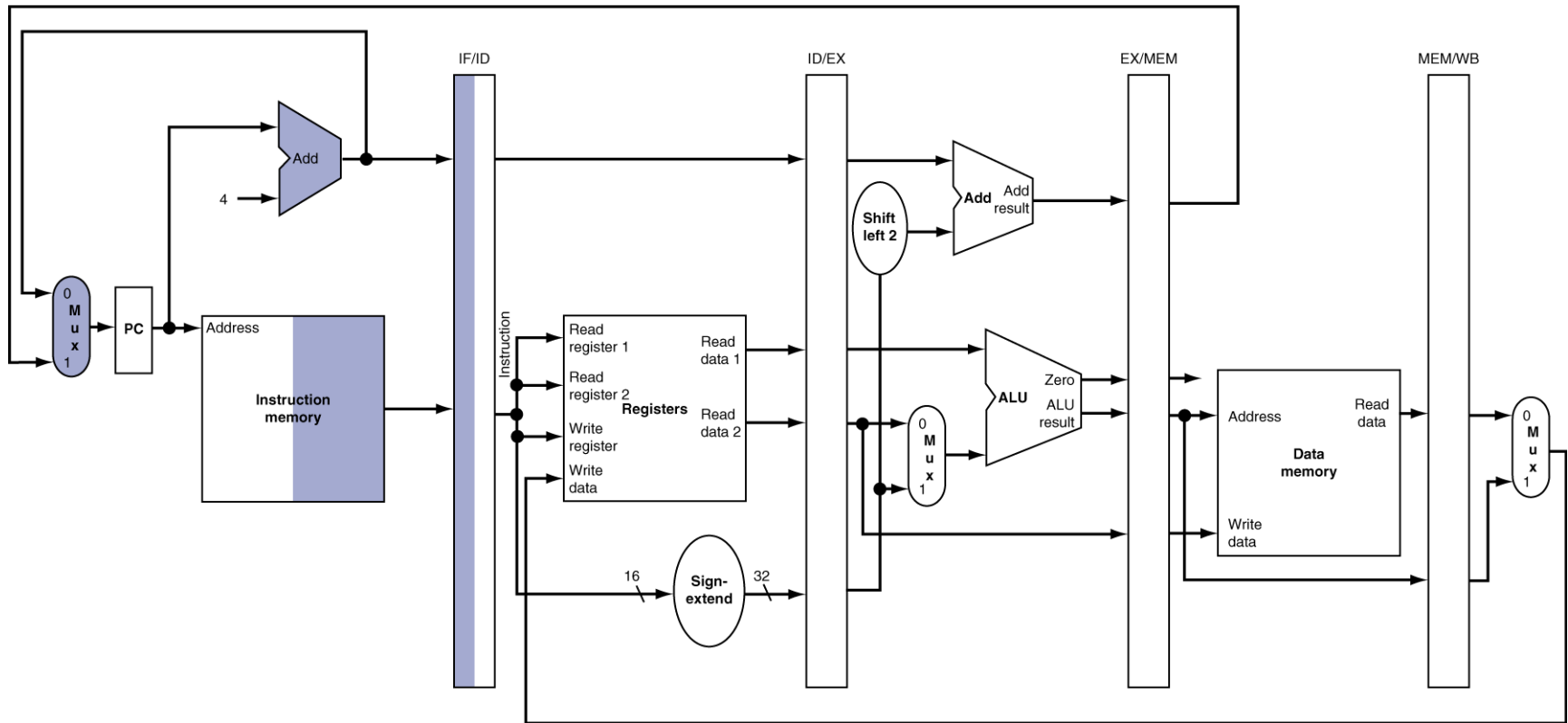
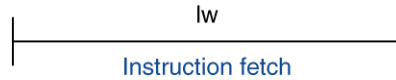
# Pipelined Datapath



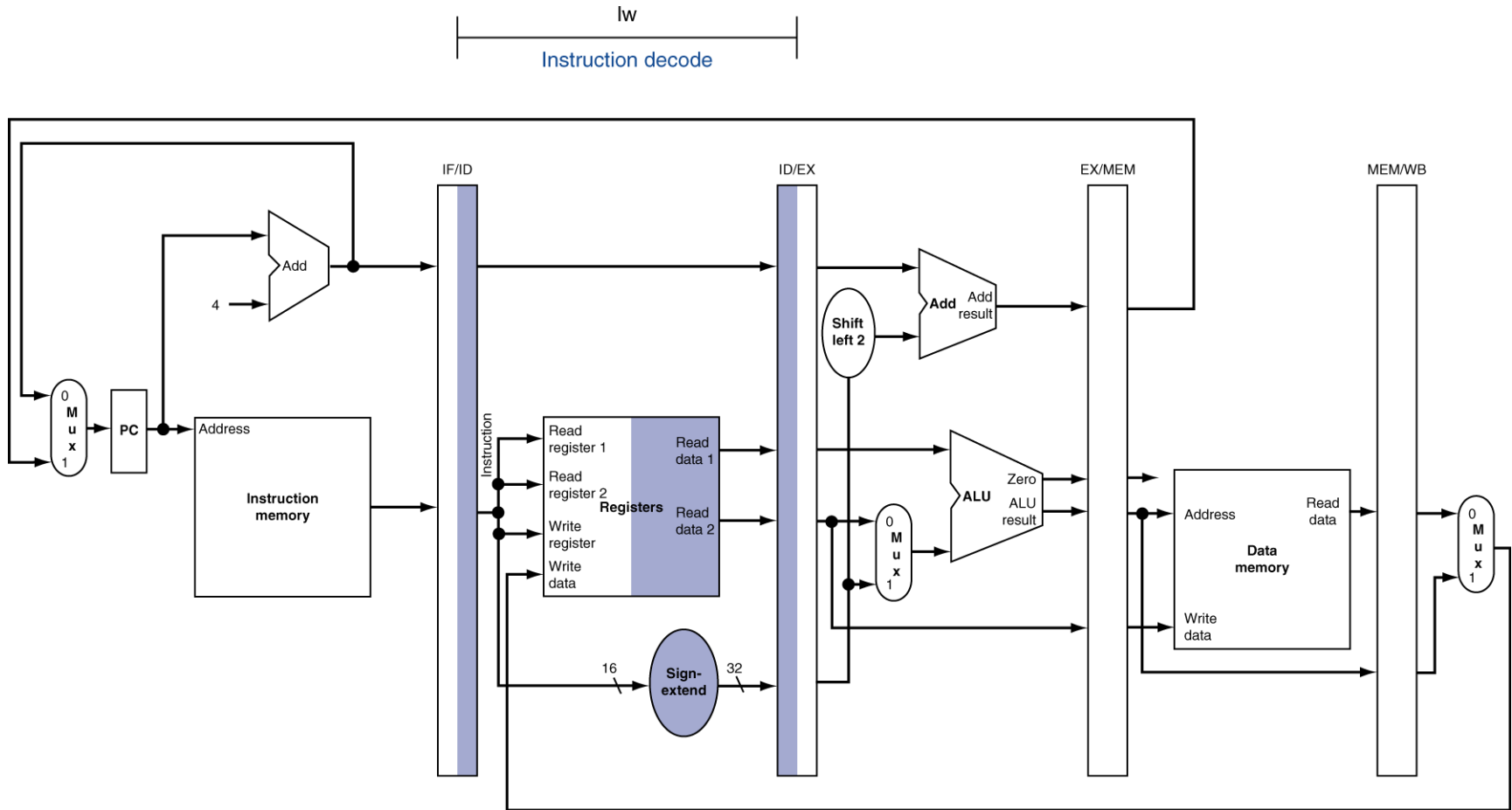
# Pipelined Datapath



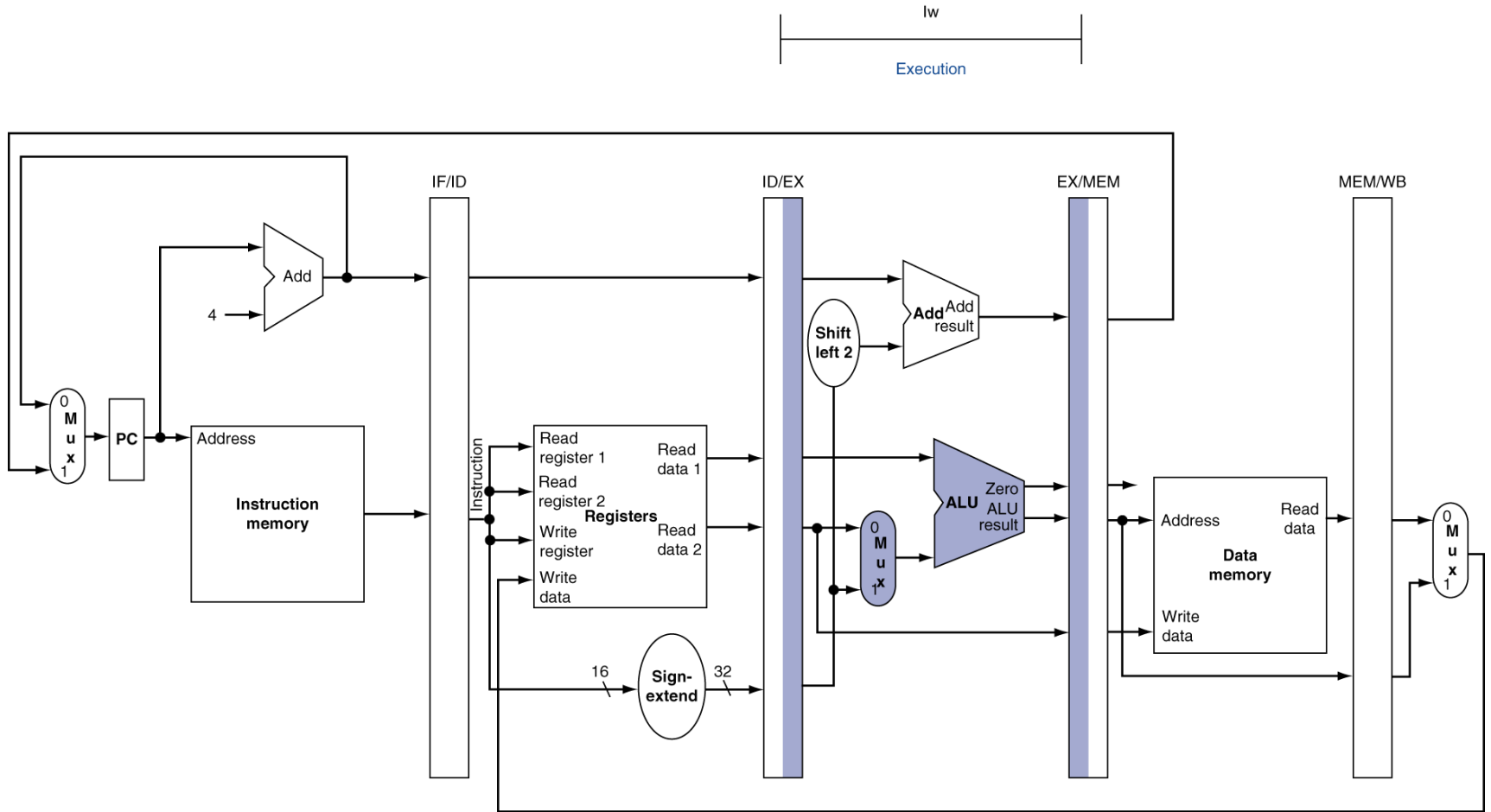
# IF for Load, Store, ...



# ID for Load, Store, ...

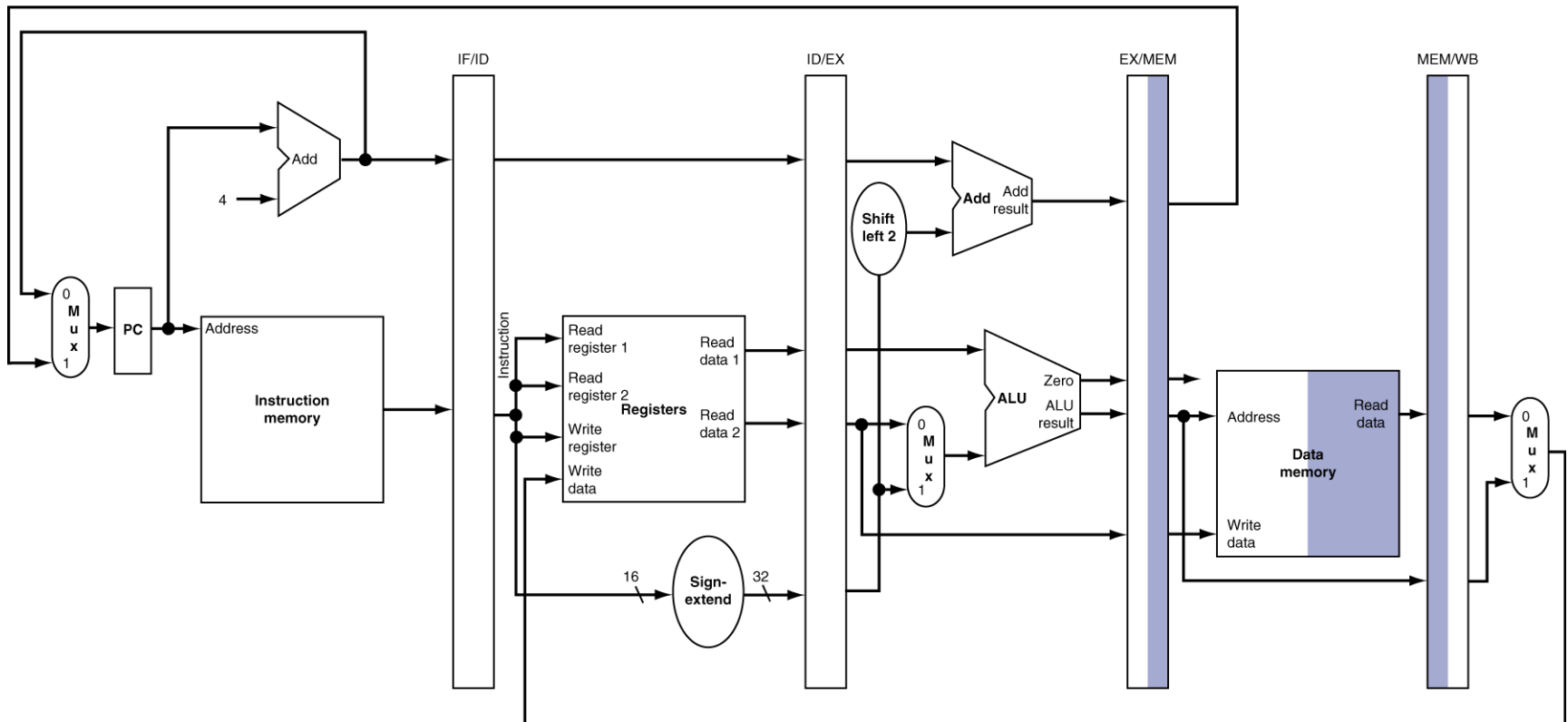


# EX for Load

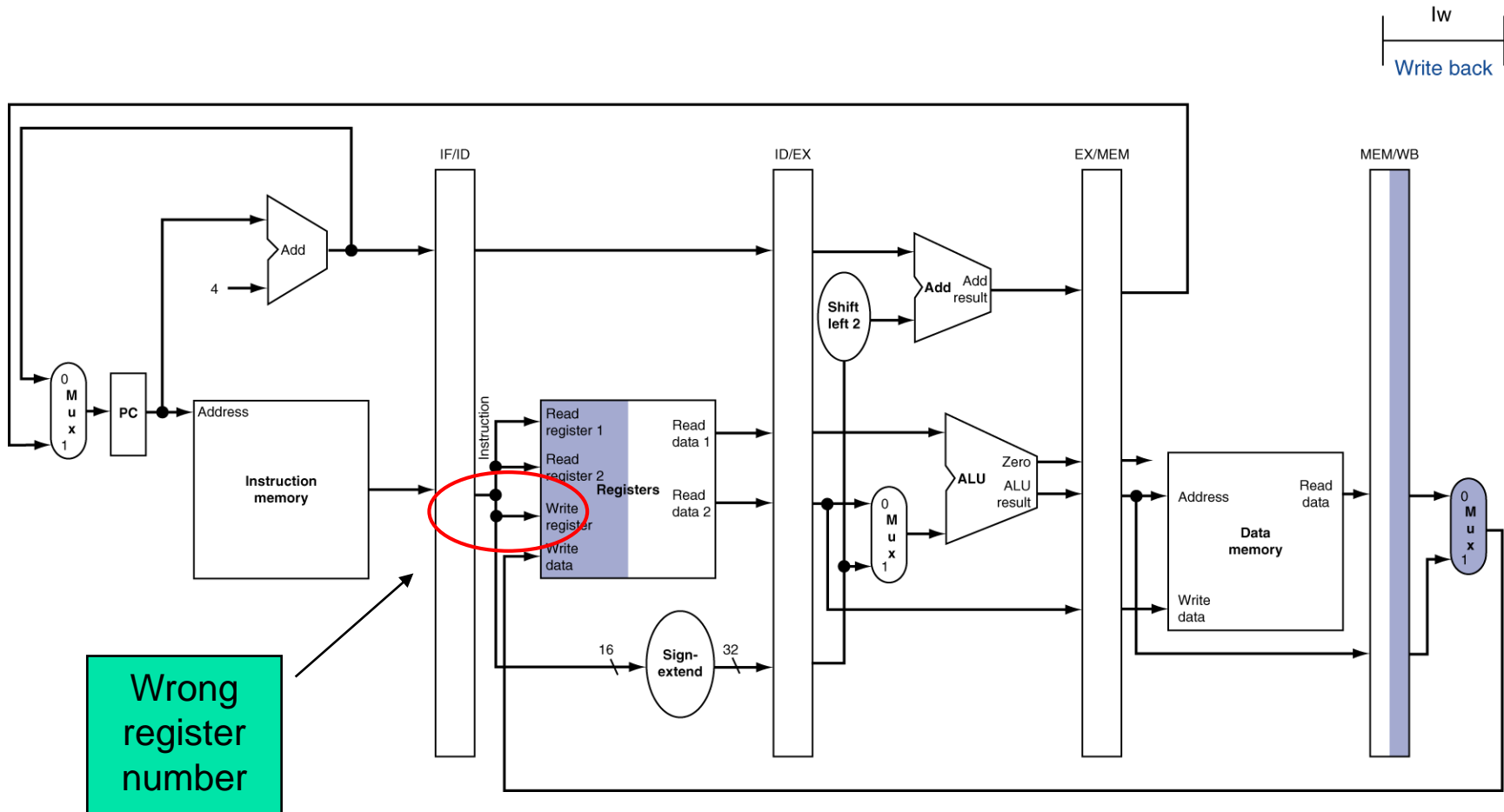


# MEM for Load

lw  
Memory

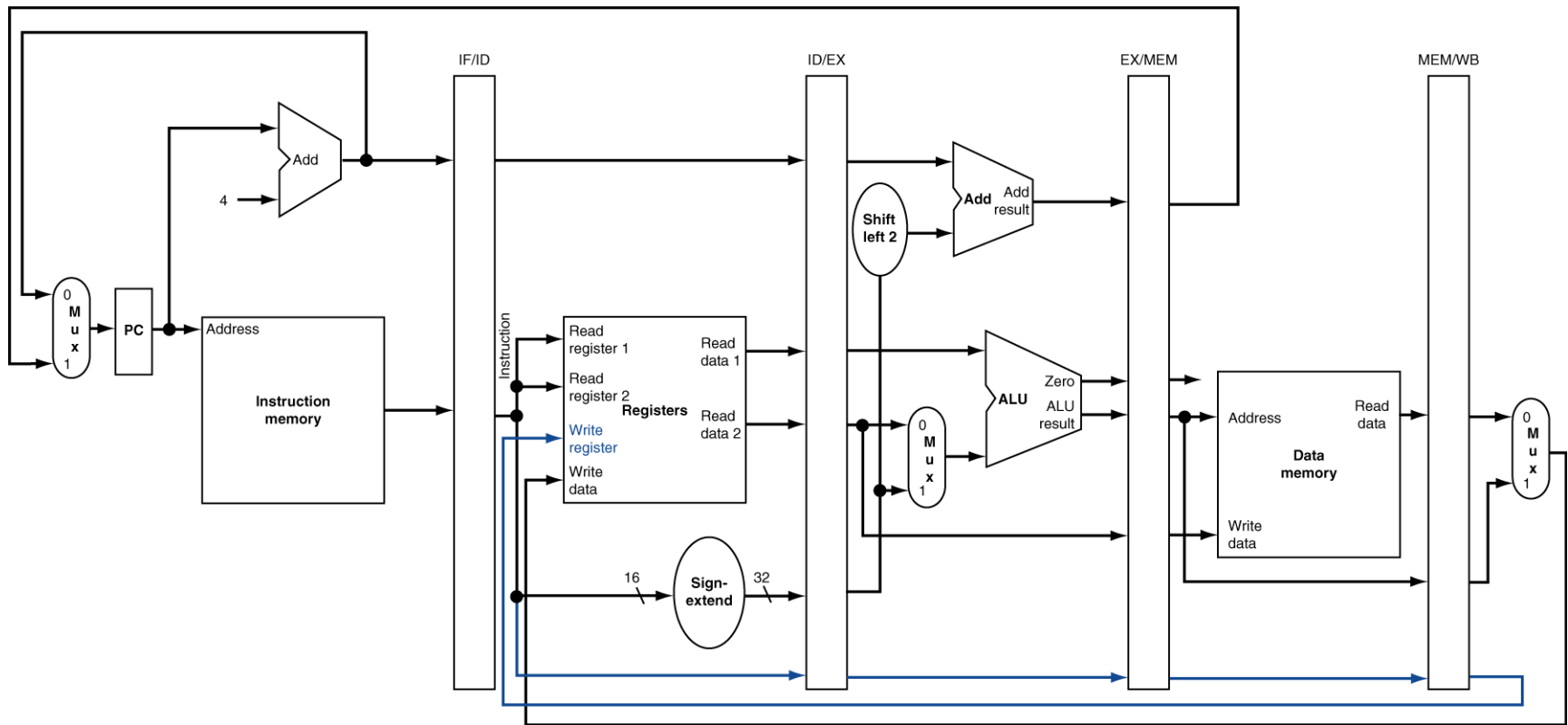


# WB for Load

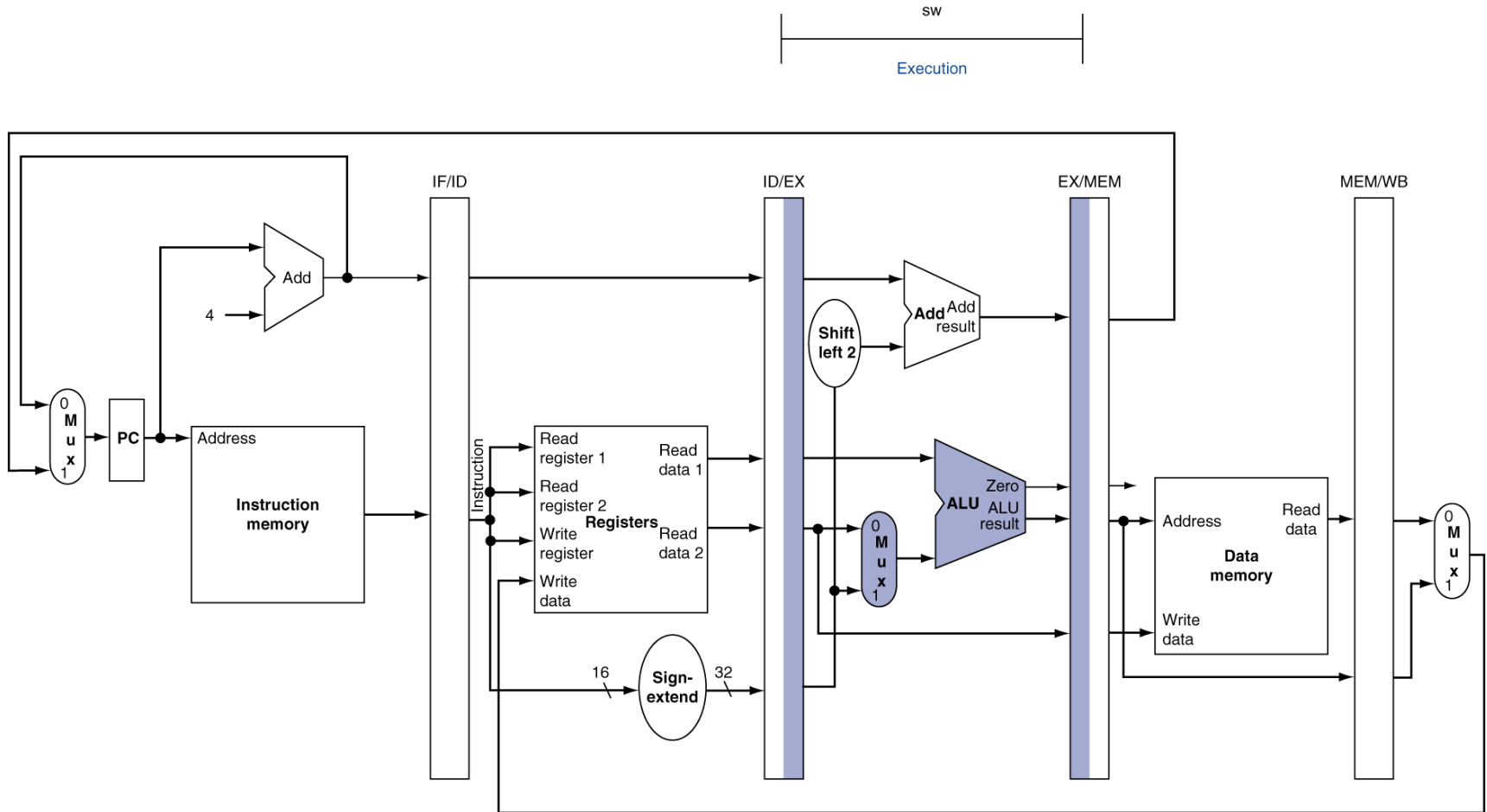




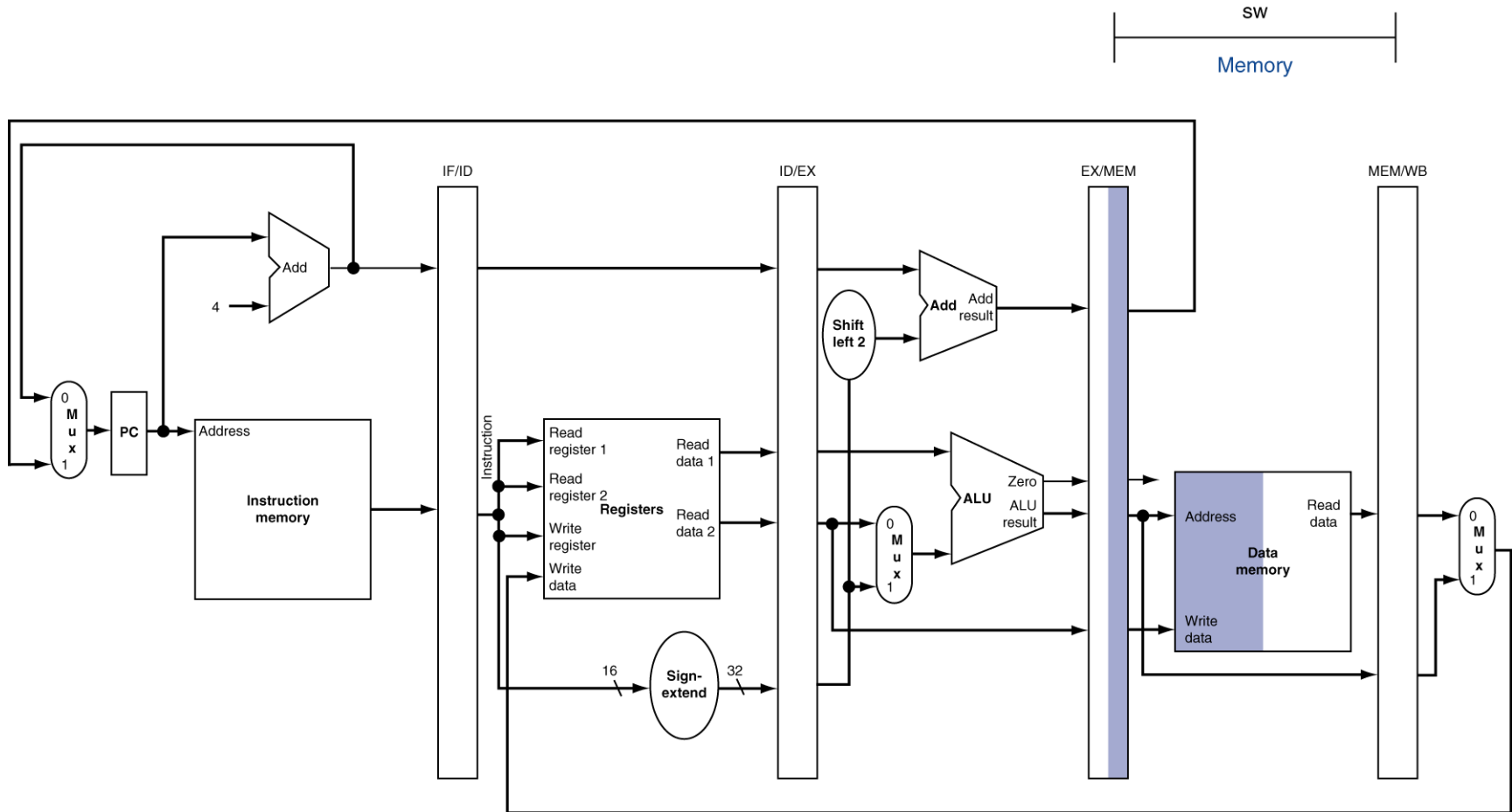
# Corrected Datapath for Load



# EX for Store

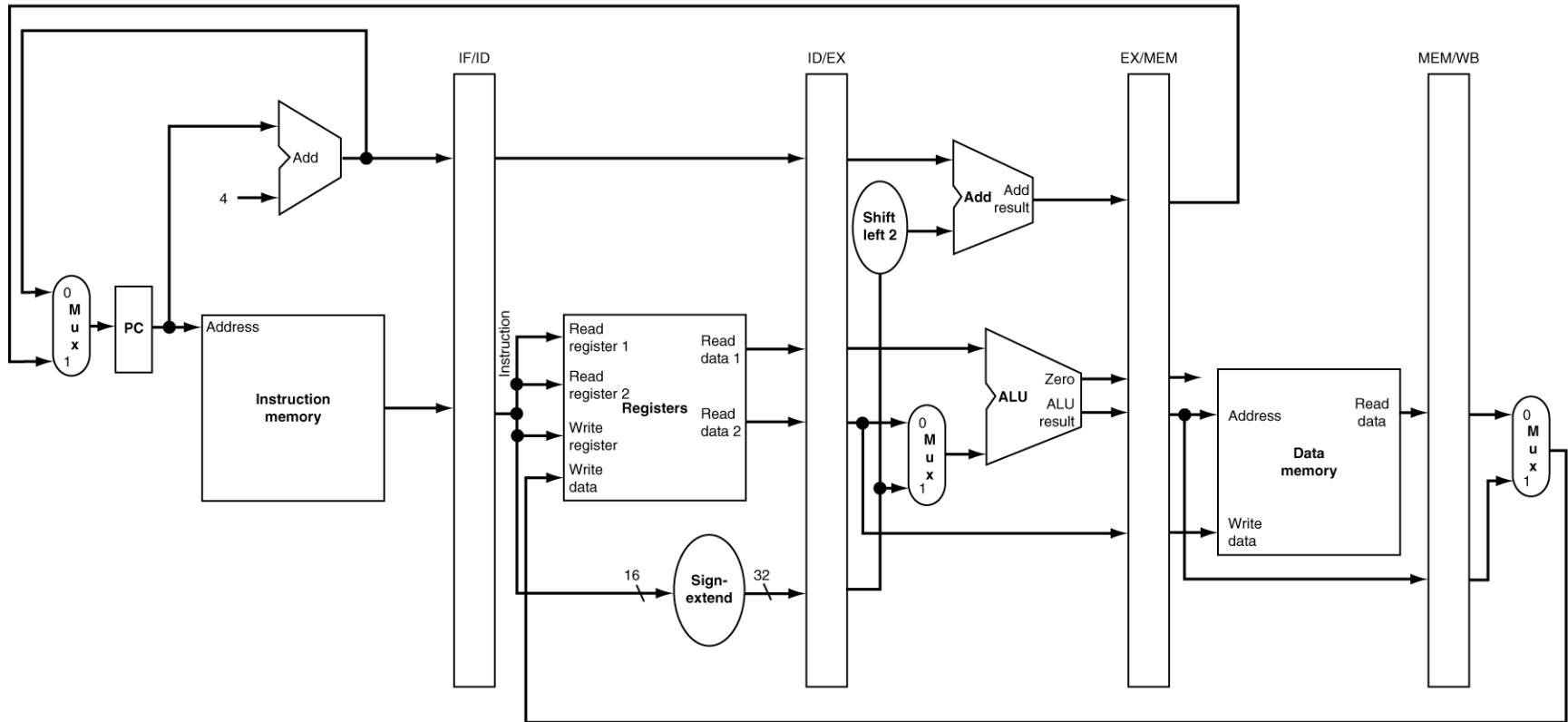


# MEM for Store

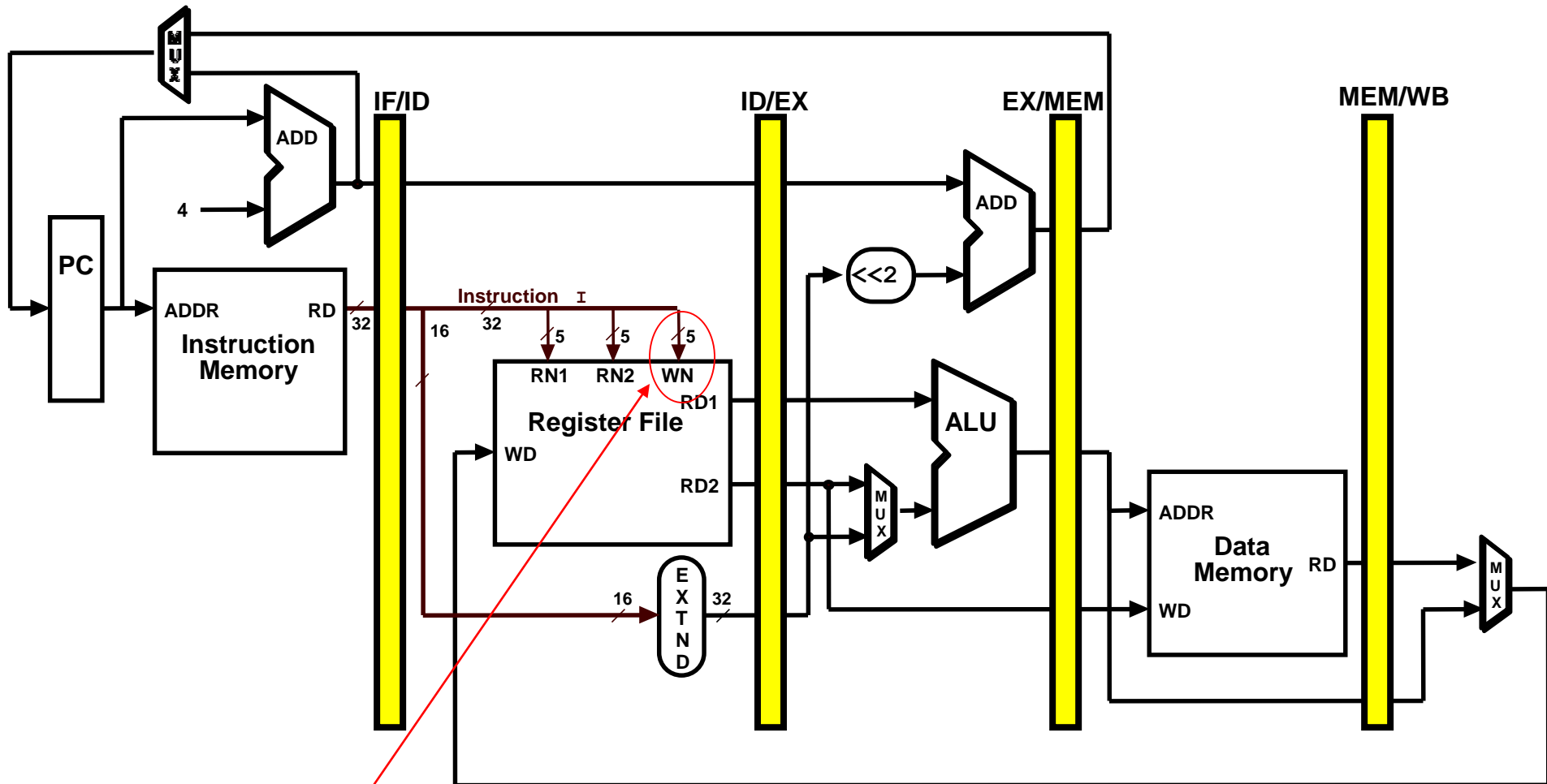


# WB for Store

SW  
Write-back

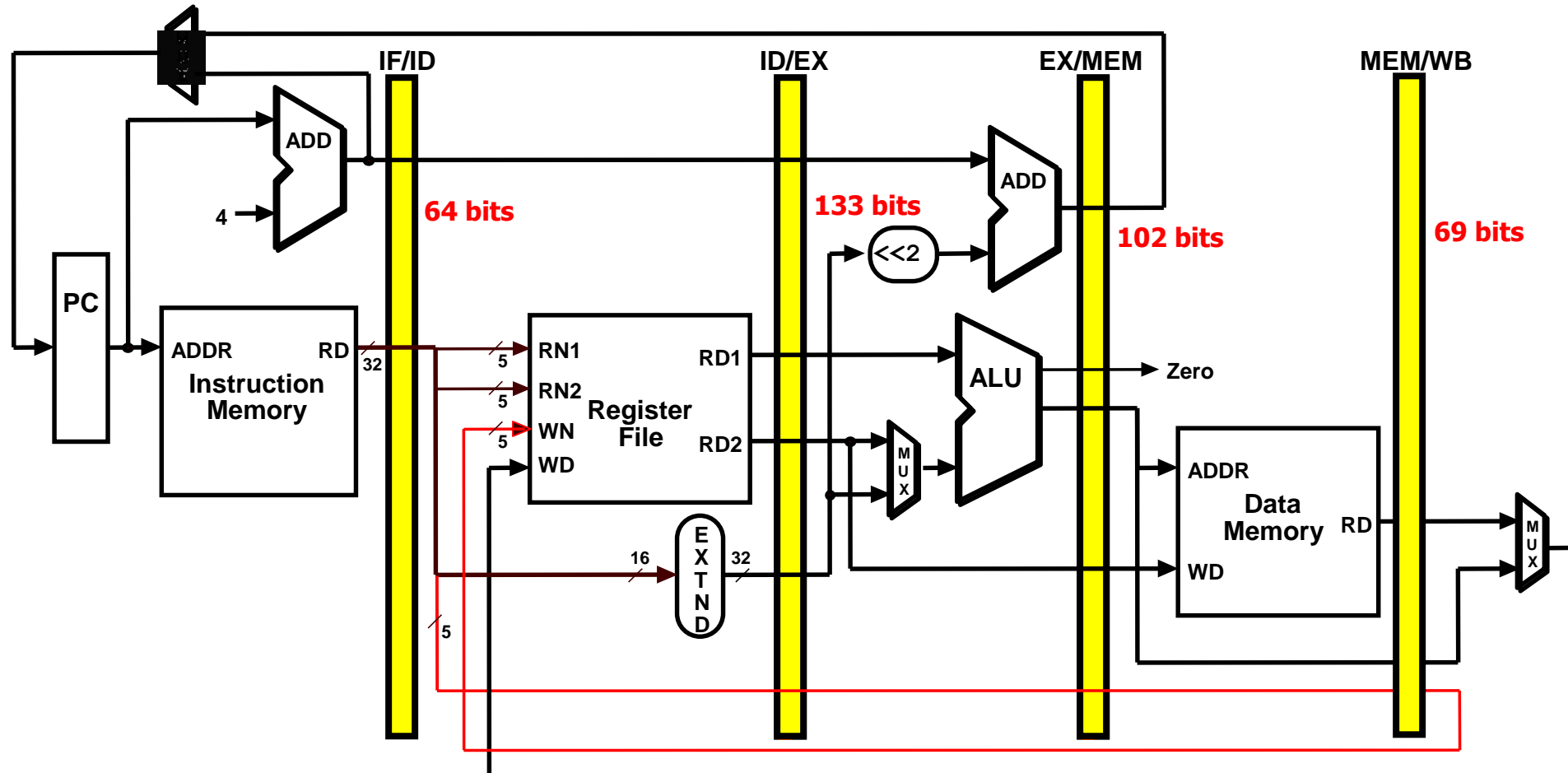


# Bug in the Datapath



**Write register number comes from another *later* instruction!**

# Corrected Datapath



**Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits**

# Pipelined Example

- Consider the following instruction sequence:

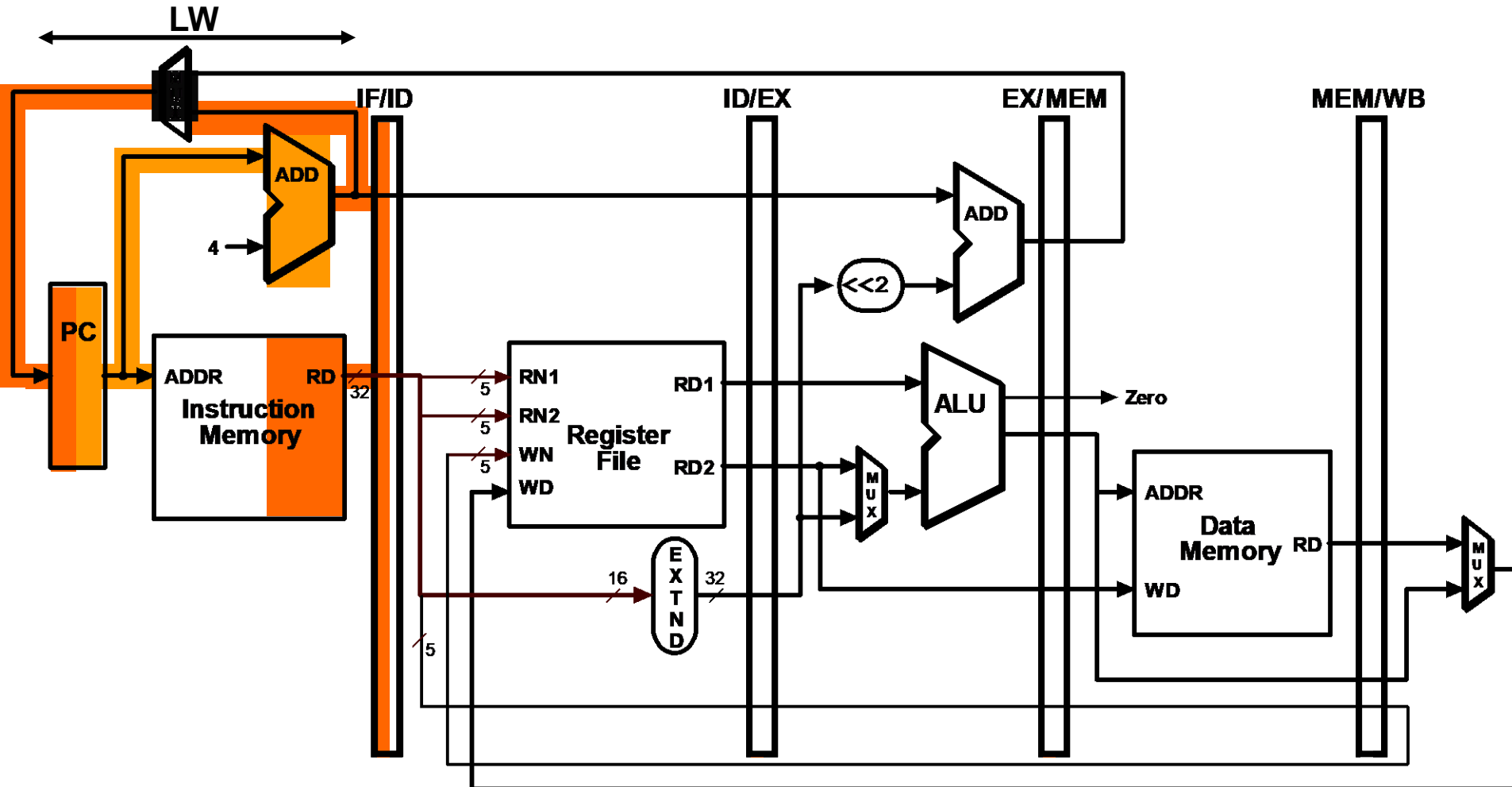
```
lw    $t0, 10($t1)
```

```
sw    $t3, 20($t4)
```

```
add   $t5, $t6, $t7
```

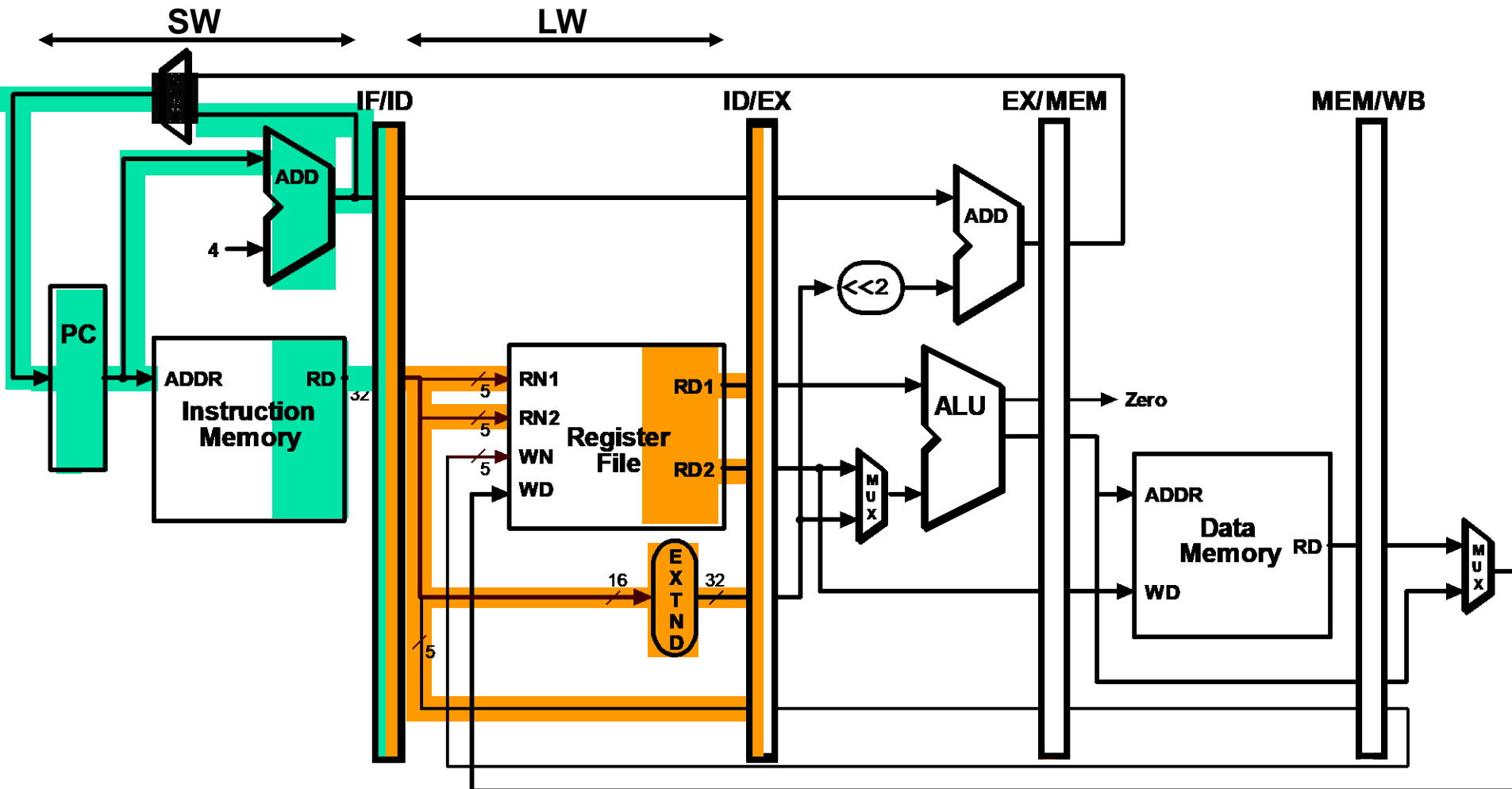
```
sub   $t8, $t9, $t10
```

# Single-Clock-Cycle Diagram: Clock Cycle 1

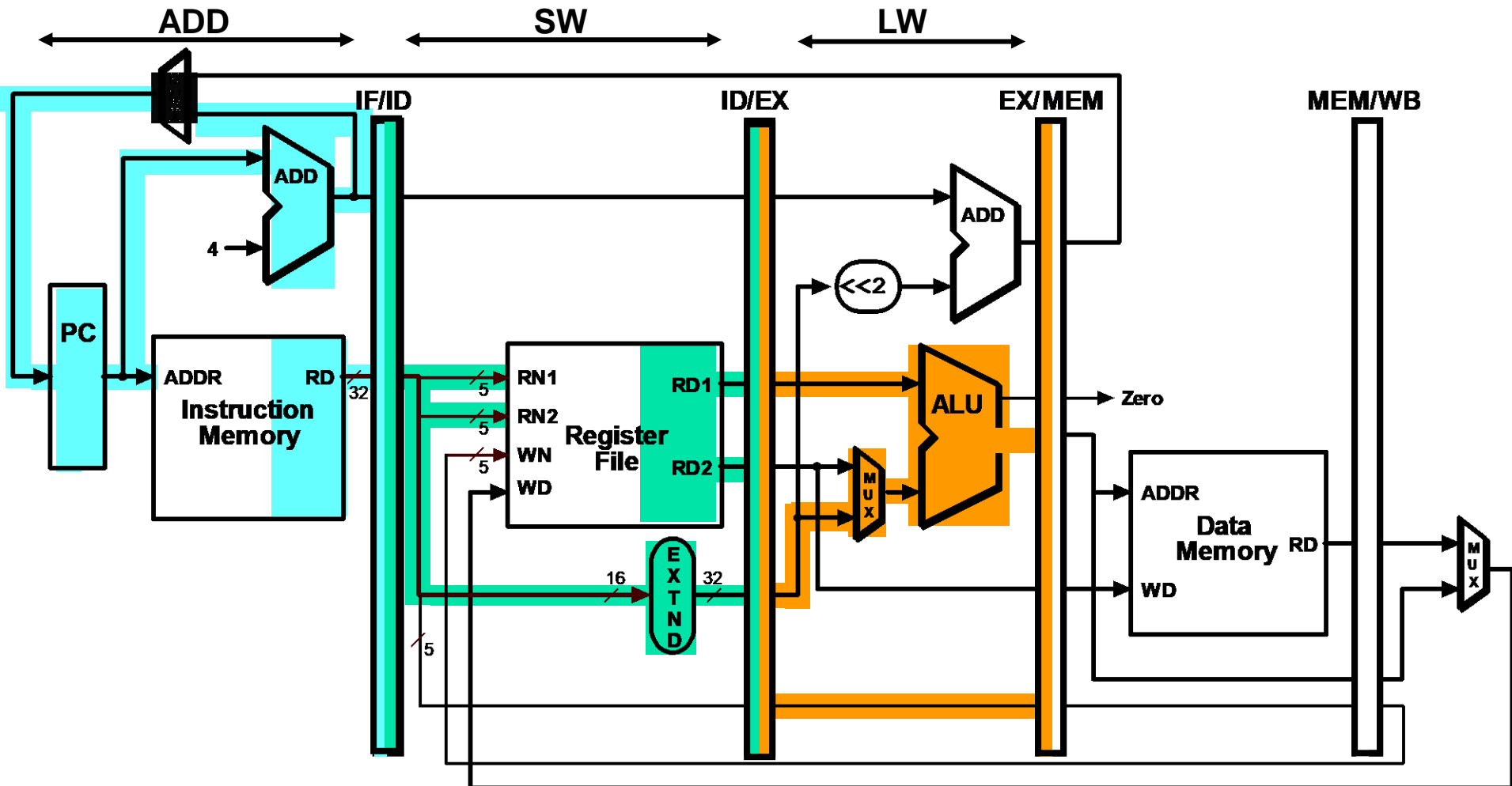




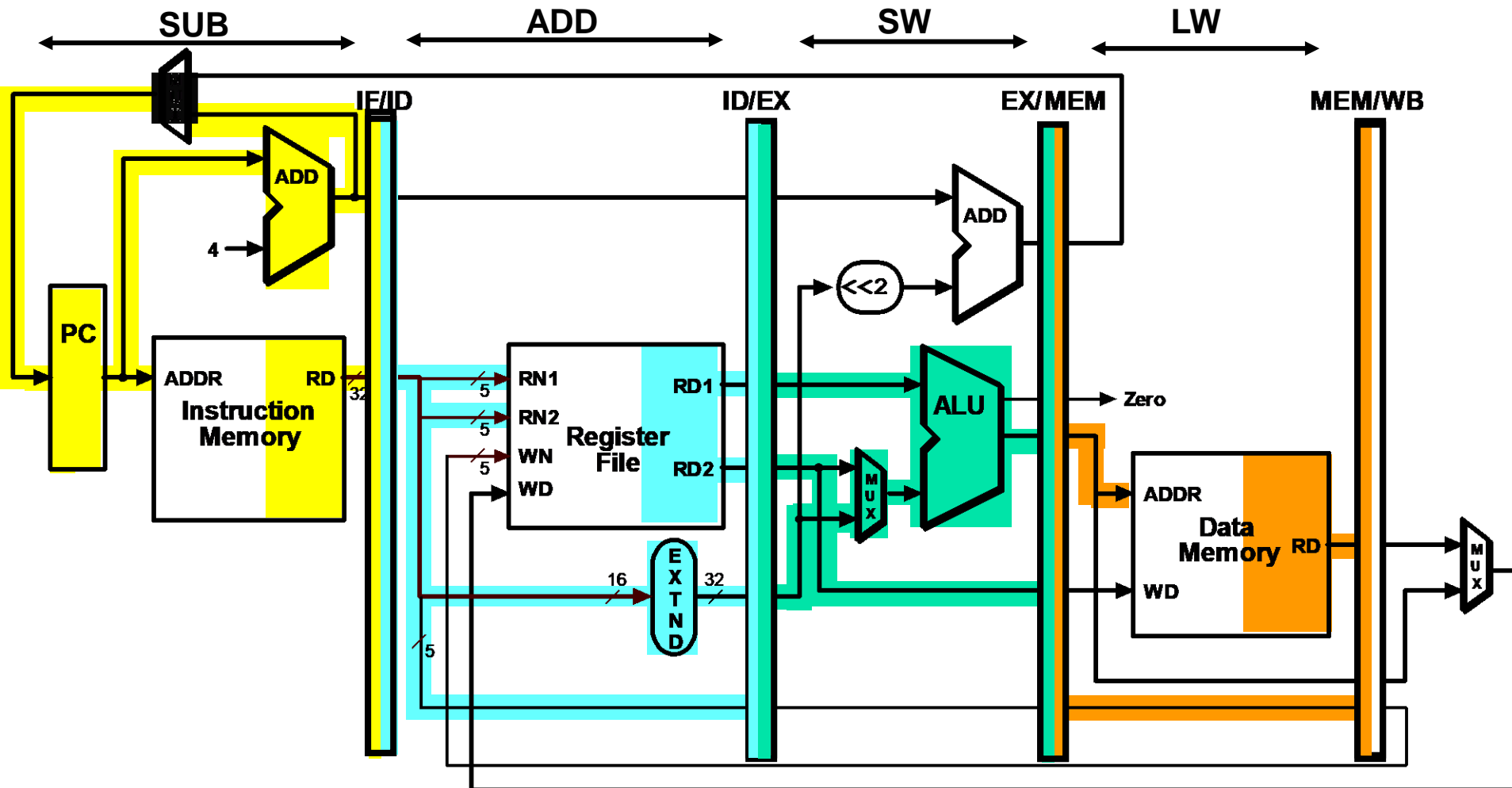
# Single-Clock-Cycle Diagram: Clock Cycle 2



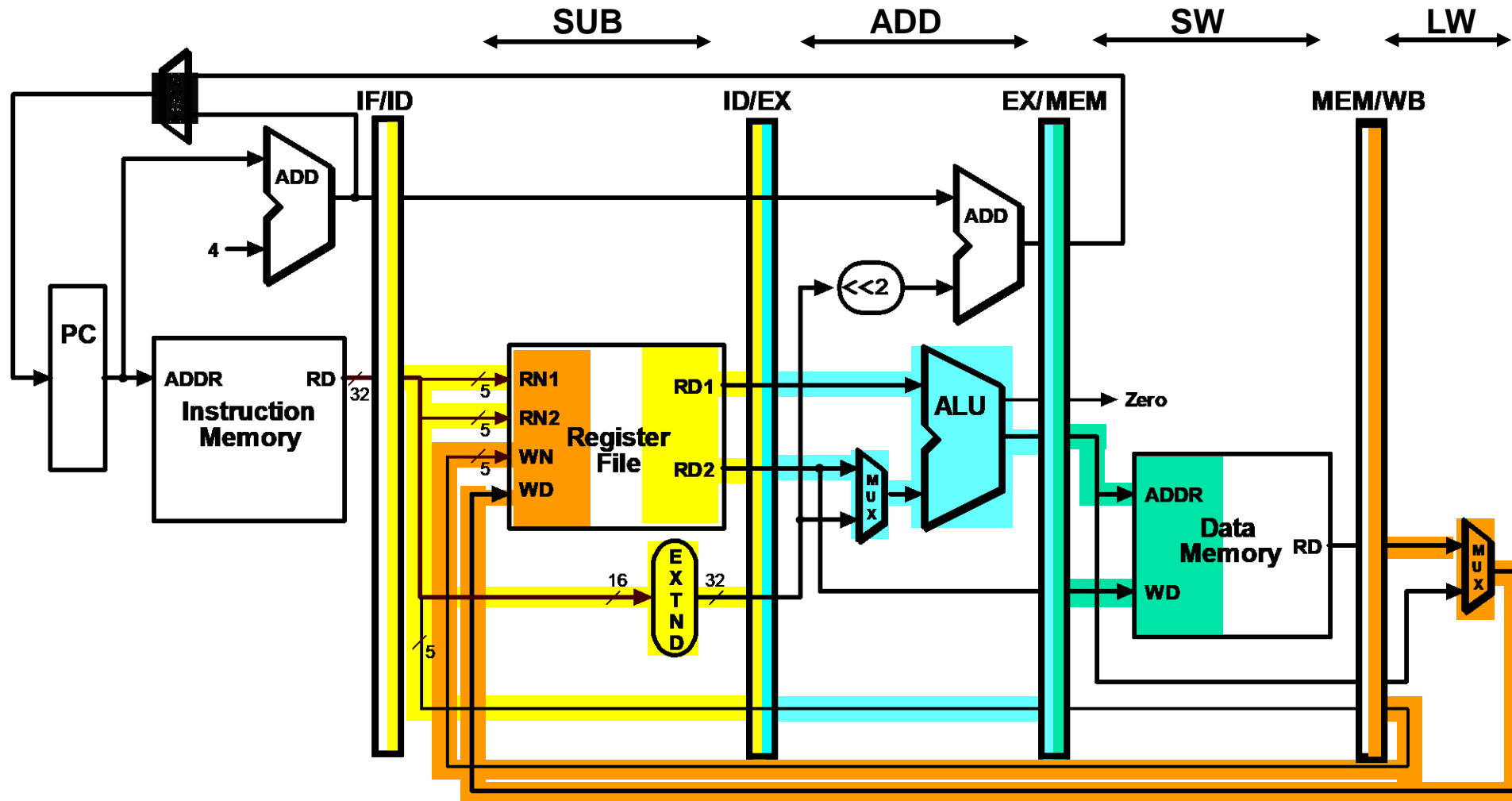
# Single-Clock-Cycle Diagram: Clock Cycle 3



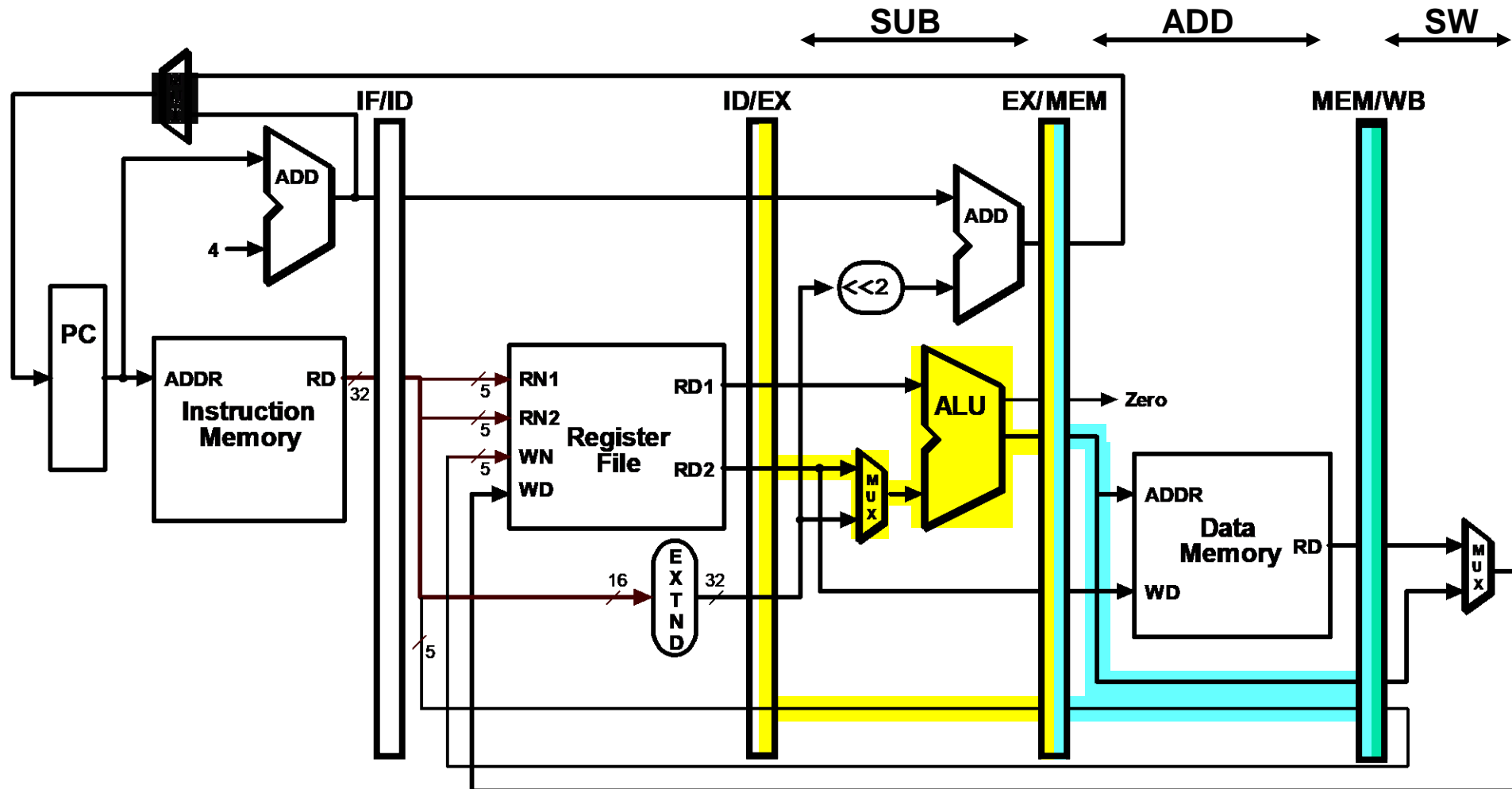
# Single-Clock-Cycle Diagram: Clock Cycle 4



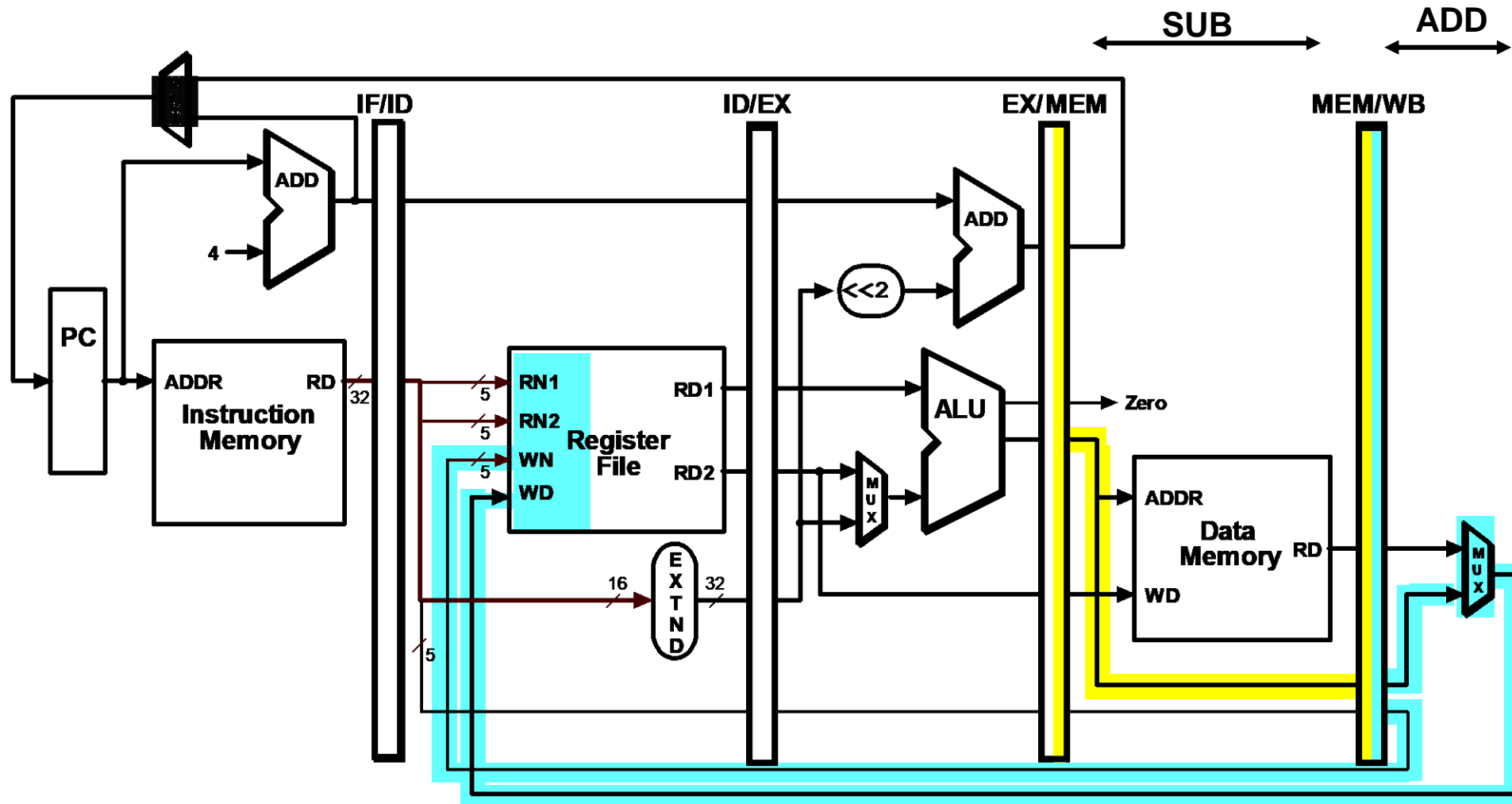
# Single-Clock-Cycle Diagram: Clock Cycle 5



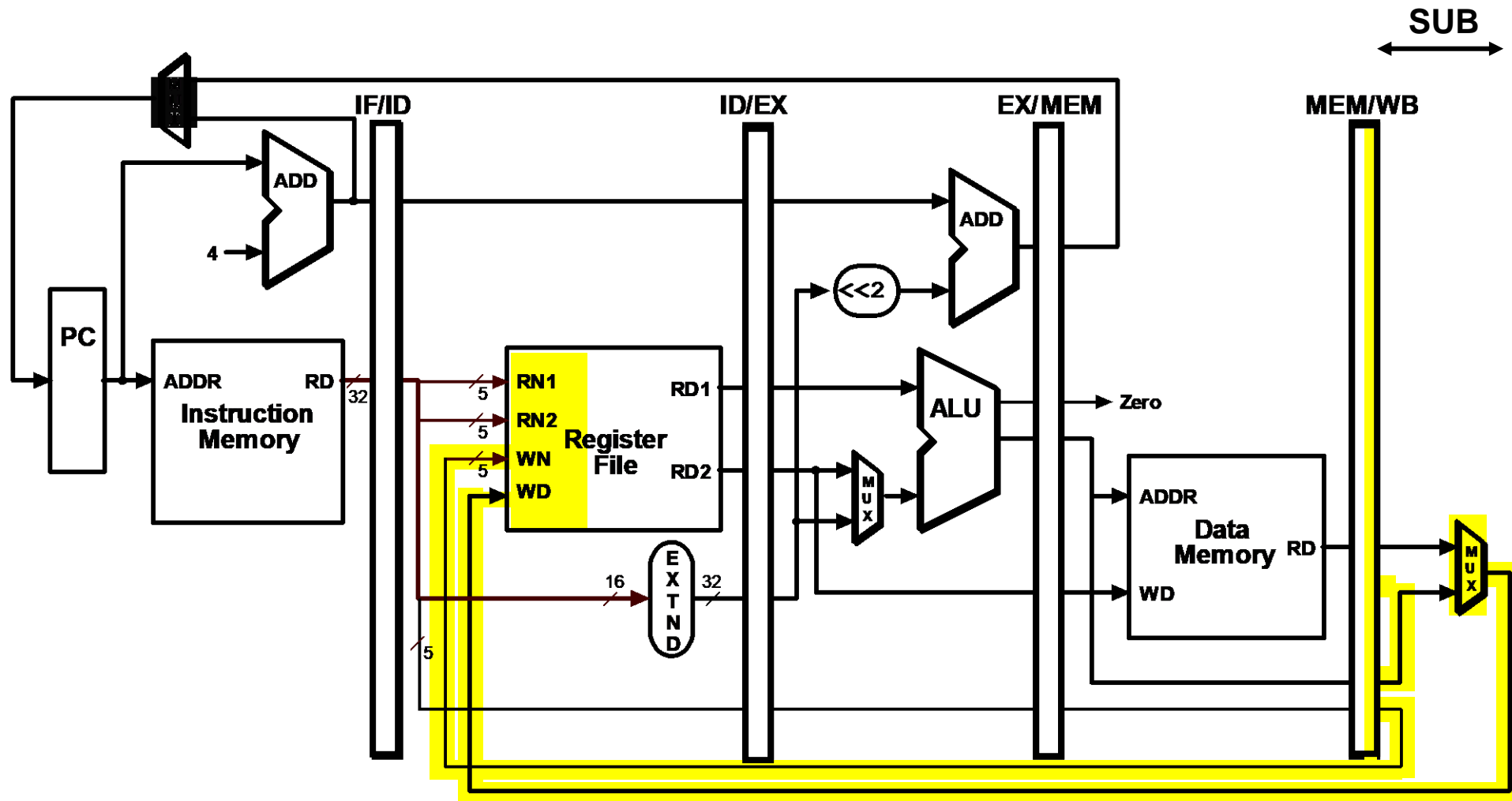
# Single-Clock-Cycle Diagram: Clock Cycle 6



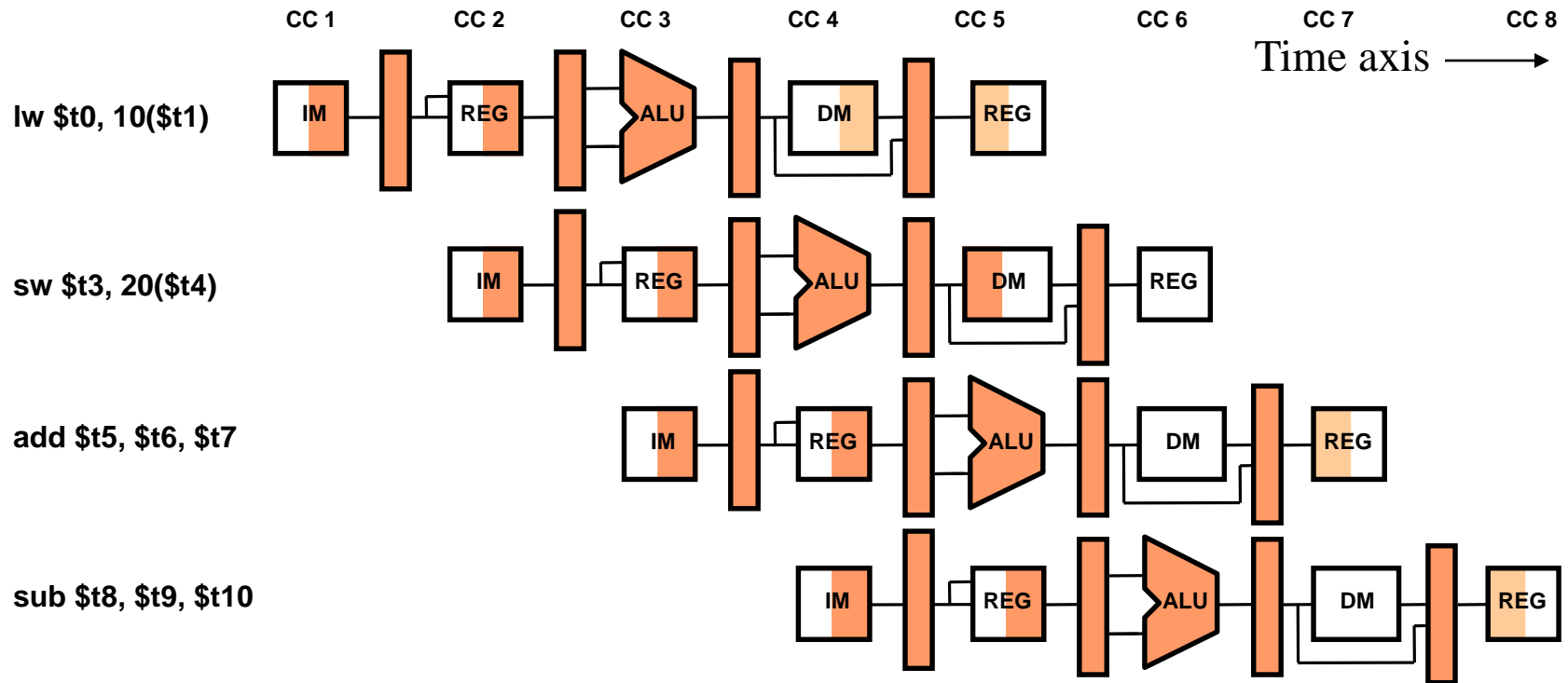
# Single-Clock-Cycle Diagram: Clock Cycle 7



# Single-Clock-Cycle Diagram: Clock Cycle 8



# Multiple-Clock-Cycle Diagram

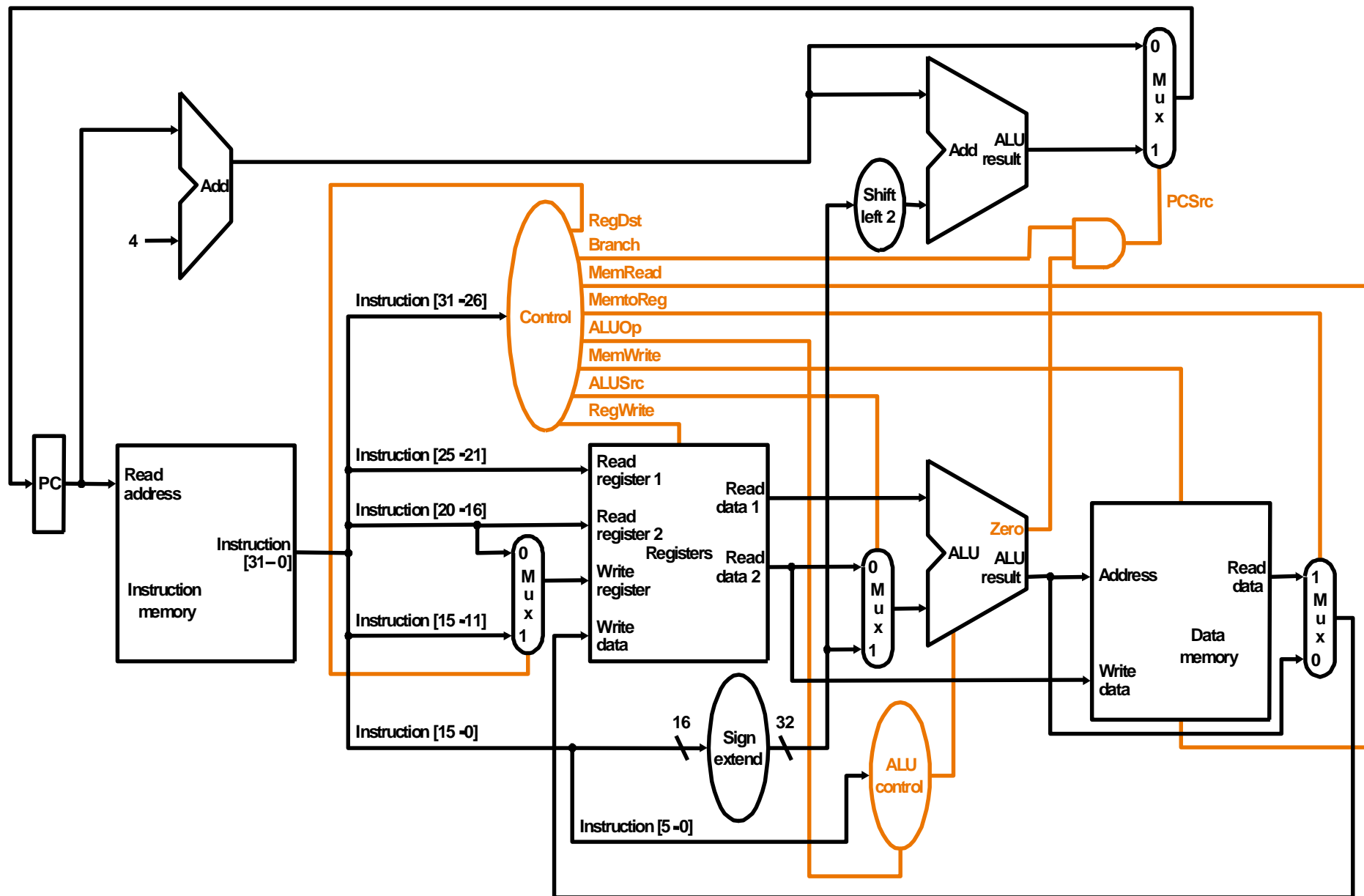




# Notes

- One significant **difference** in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5<sup>th</sup> (the last write-back) pipeline stage vs. the 4<sup>th</sup> stage for the multicycle implementation. *Why?*
  - think of *structural hazards* when writing to the register file...
- Worth repeating: the *essential difference* between the pipeline and multicycle implementations is the insertion of pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1. *Why?*
- As we develop control for the pipeline keep in mind that we are *not considering* `jump` – should not be too hard to implement!

# Recall Single-Cycle Control – the Datapath



# Recall Single-Cycle – ALU Control

Instruction opcode	AluOp	Instruction operation	Funct Field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type	10	OR	100101	or	001
R-type	10	set on less	101010	set on less	111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

**Truth table for ALU control bits**

# Recall Single-Cycle – Control Signals

## Effect of control bits

Signal Name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20-16)	The register destination number for the Write register comes from the rd field (bits 15-11)
RegWrite	None	The register on the Write register input is written with the value on the Write data input
ALUSrc	The second ALU operand comes from the second register file output (Read data 2)	The second ALU operand is the sign-extended, lower 16 bits of the instruction
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4	The PC is replaced by the output of the adder that computes the branch target
MemRead	None	Data memory contents designated by the address input are put on the first Read data output
MemWrite	None	Data memory contents designated by the address input are replaced by the value of the Write data input
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory

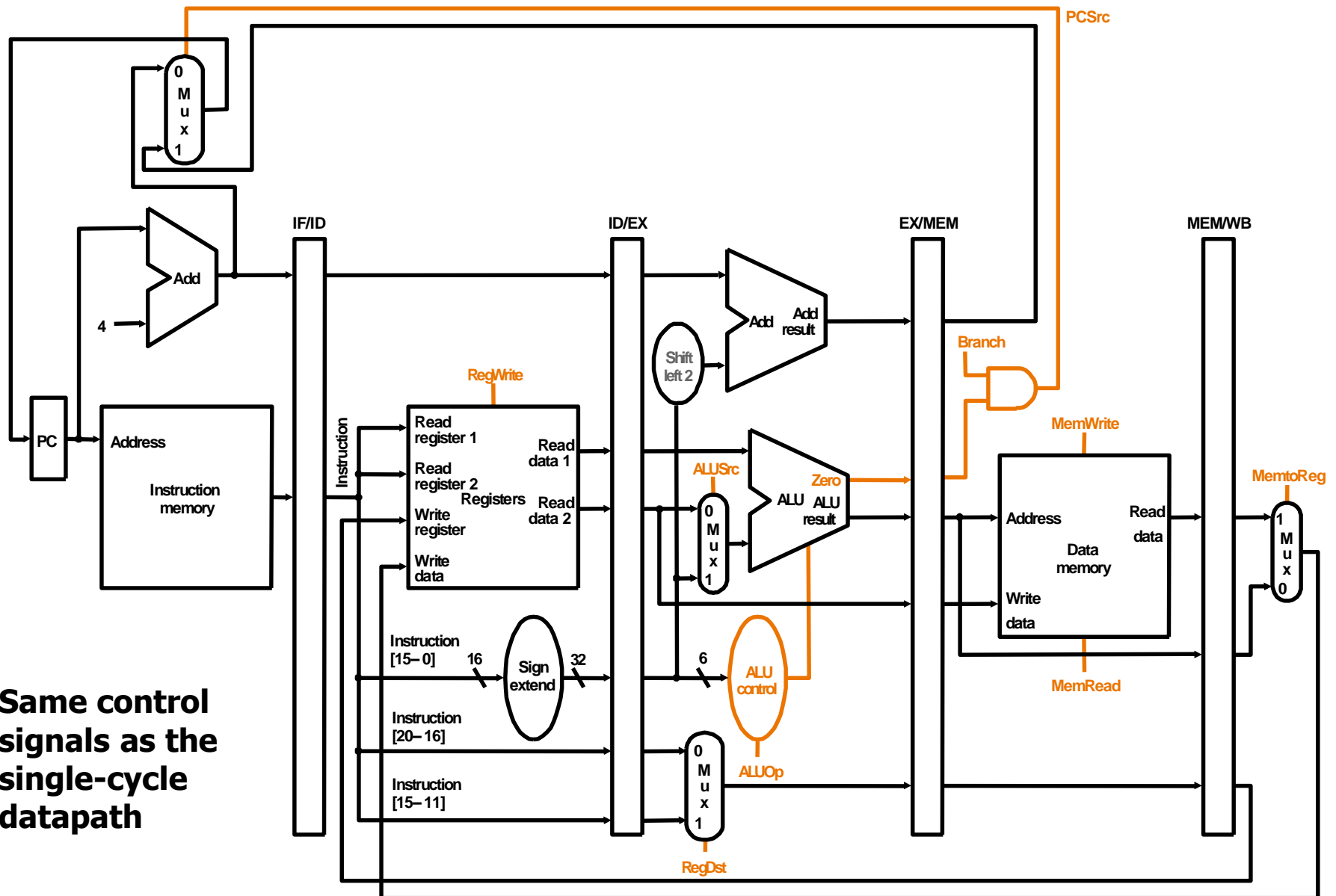
## Determining control bits

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	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

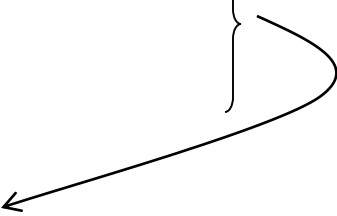
# Pipeline Control

- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
  - Observe:
    - No separate write signal for the PC as it is written every cycle
    - No separate write signals for the pipeline registers as they are written every cycle
    - *No separate read signal for instruction memory* as it is read every clock cycle
    - *No separate read signal for register file* as it is read every clock cycle
  - Need to *set control signals during each pipeline stage*
  - Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*
- } Will be modified by hazard detection unit!!

# Pipelined Datapath with Control I



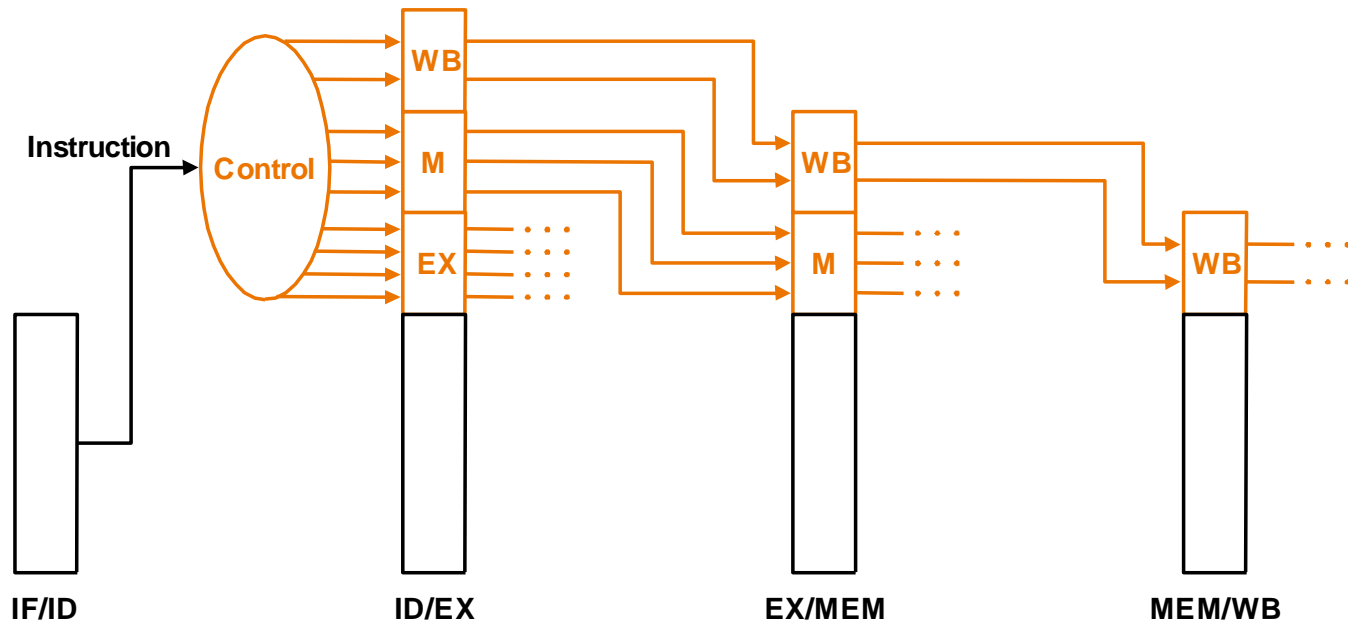
# Pipeline Control Signals

- There are five stages in the pipeline
    - *instruction fetch / PC increment*
    - *instruction decode / register fetch*
    - *execution / address calculation*
    - *memory access*
    - *write back*
- } Nothing to control as instruction memory read and PC write are always enabled
- 

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branc h	Mem Read	Mem Write	Reg write	Mem to 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

# Pipeline Control Implementation

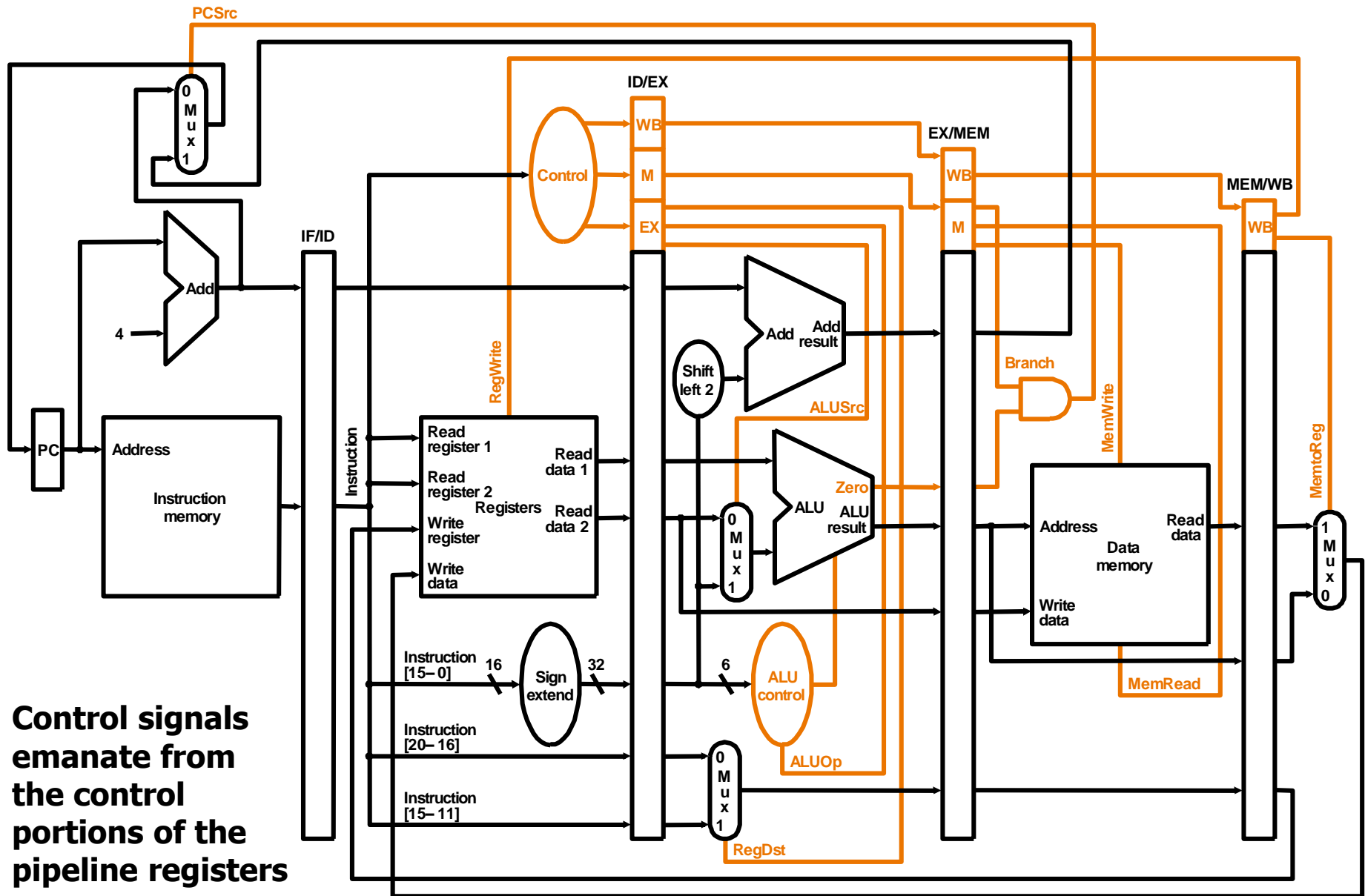
- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages



- *Note:* The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register



# Pipelined Datapath with Control II



# Pipelined Execution and Control

- Instruction sequence:

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

**Label “before<i>” means  
i th instruction before  
lw**

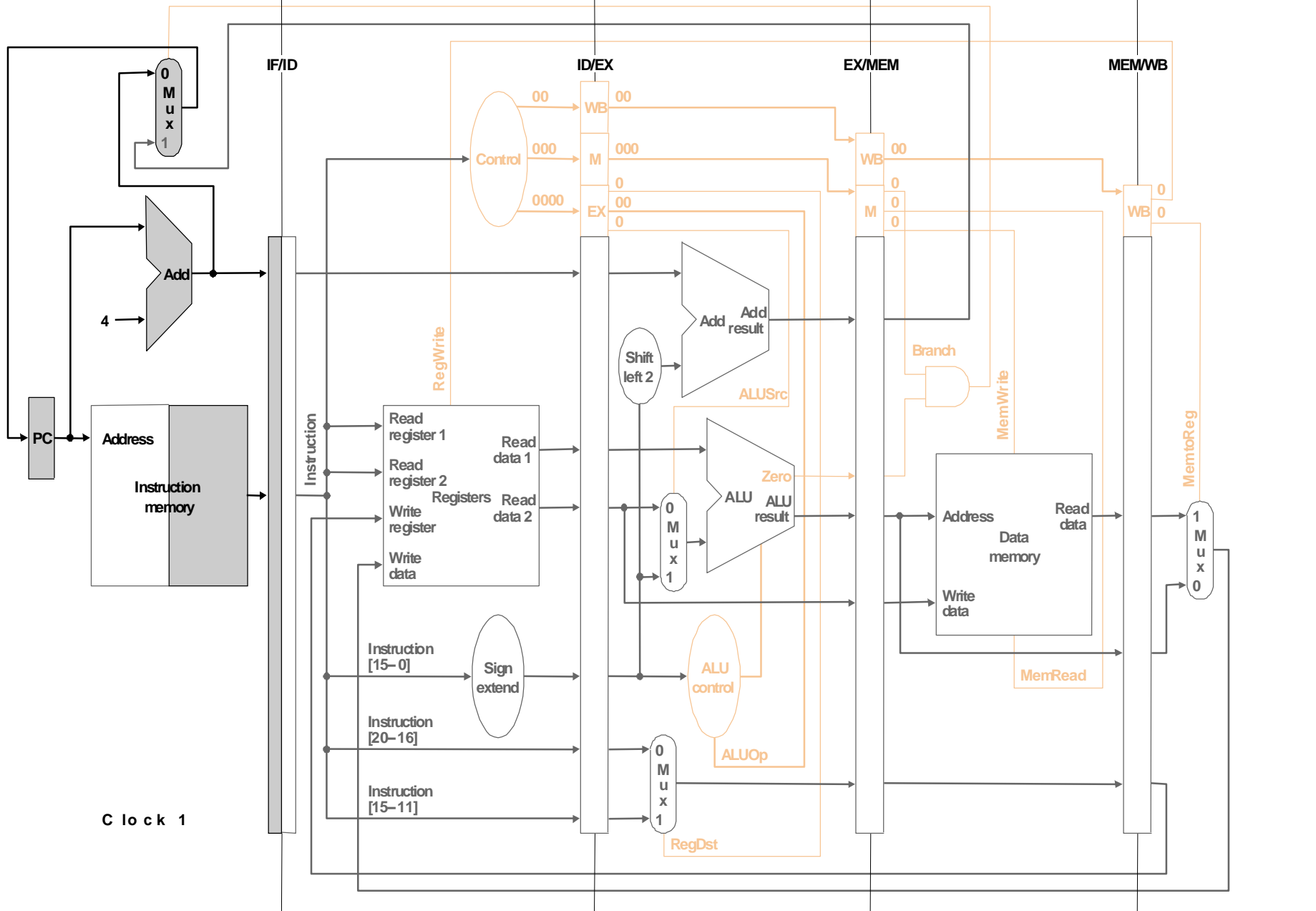
IF : lw \$t0, 20(\$t1)

ID : before <1>

EX : before <2>

MEM : before <3>

WB : before <4>



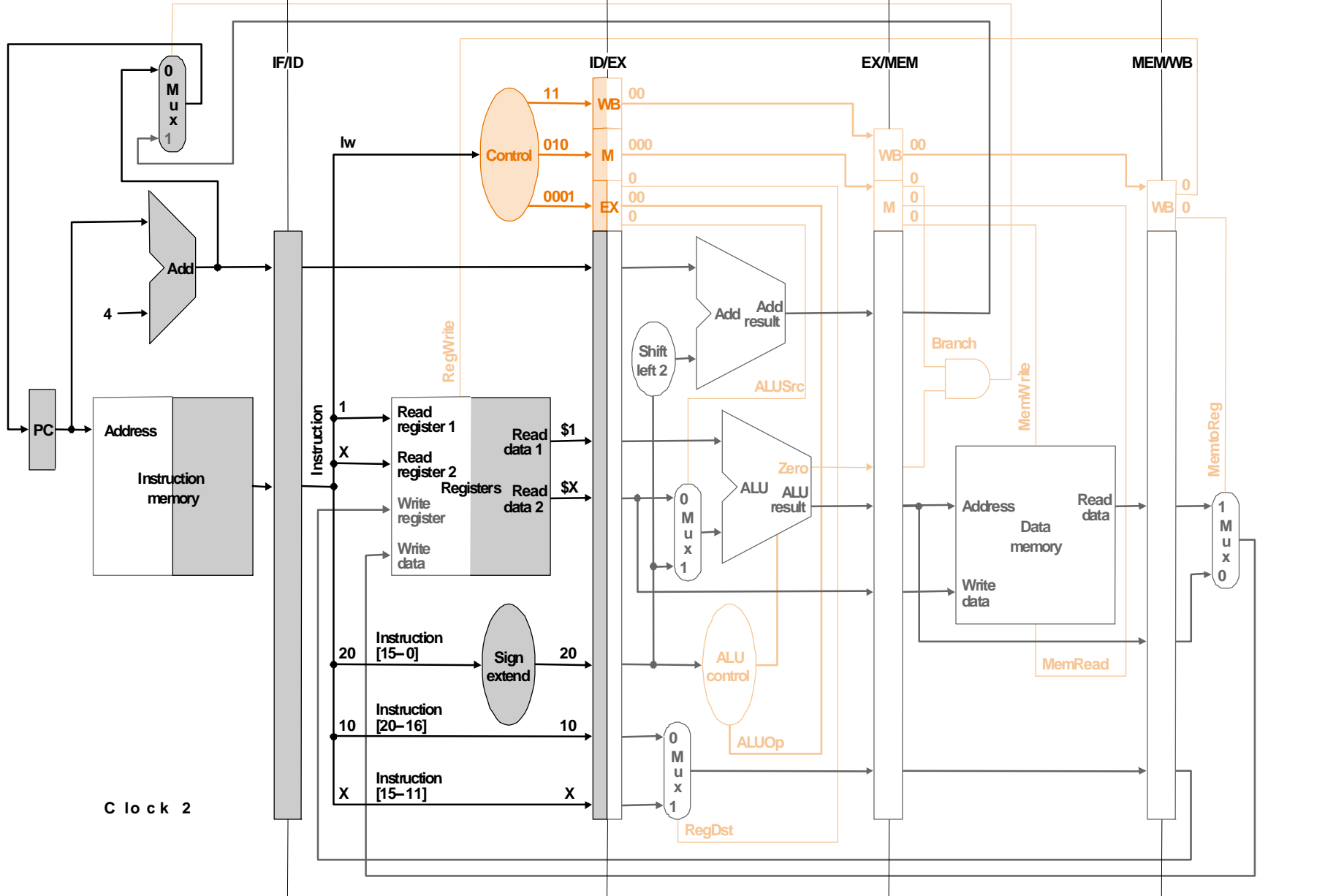
IF : sub \$11, \$2, \$3

ID : lw \$10, 20(\$1)

EX : before <1>

MEM : before <2>

WB : before <3>



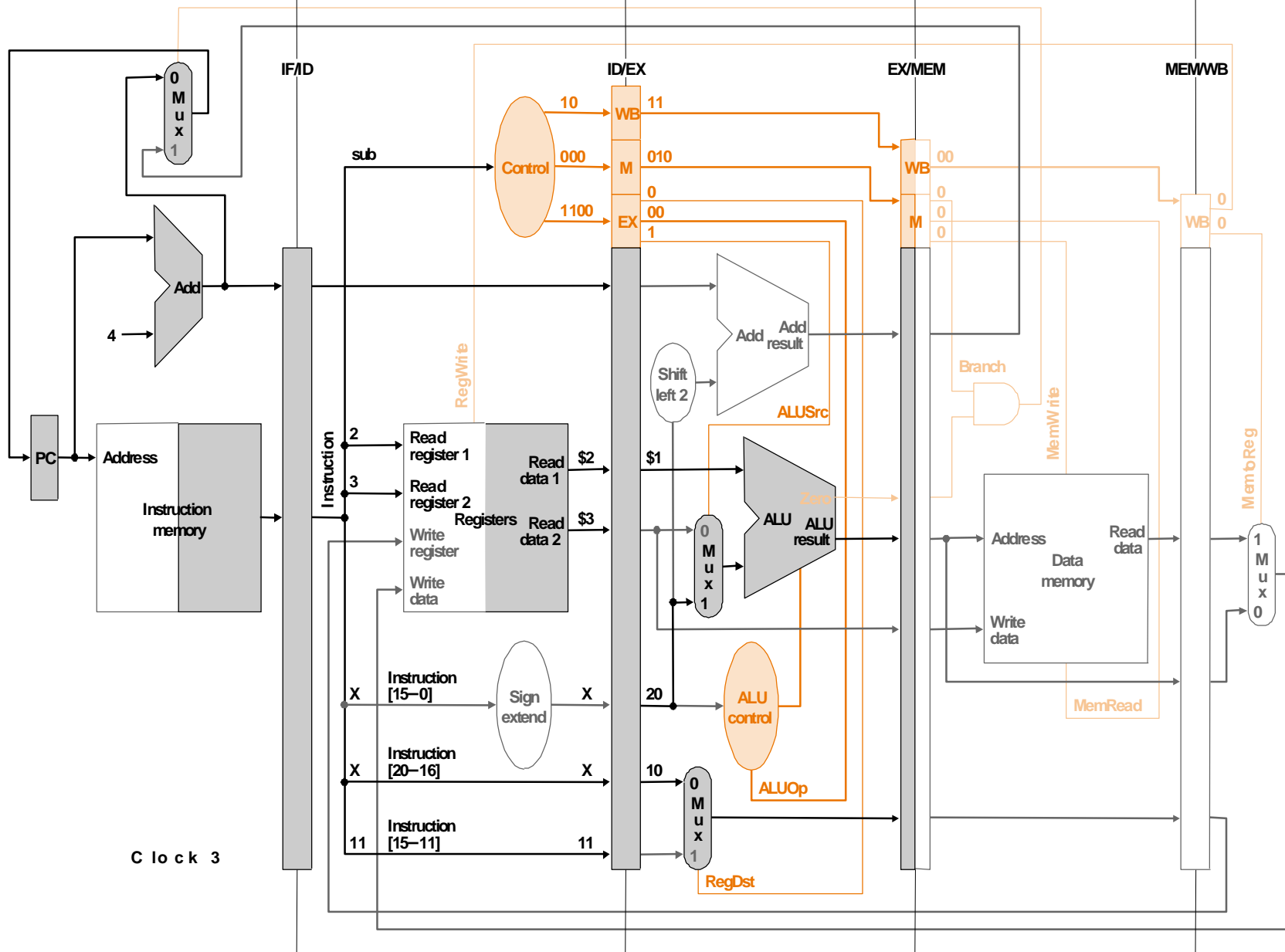
IF: and \$12, \$4, \$5

ID: sub \$11, \$2, \$3

EX: lw \$10, ...

MEM: before <1>

WB: before <2>



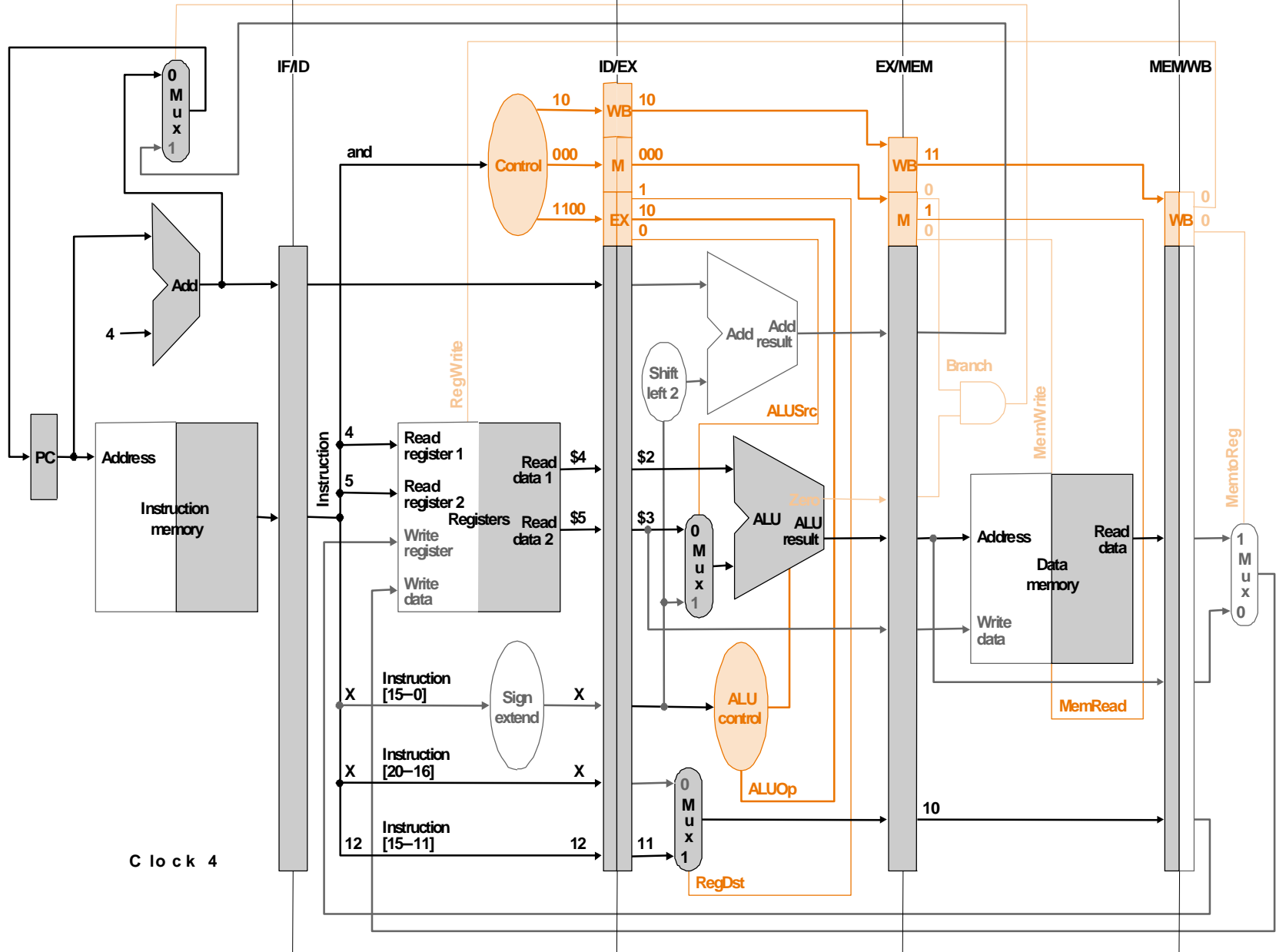
IF: or \$13, \$6, \$7

ID: and \$12, \$2, \$3

EX: sub \$11, ...

MEM: lw \$10, ...

WB: before <1>



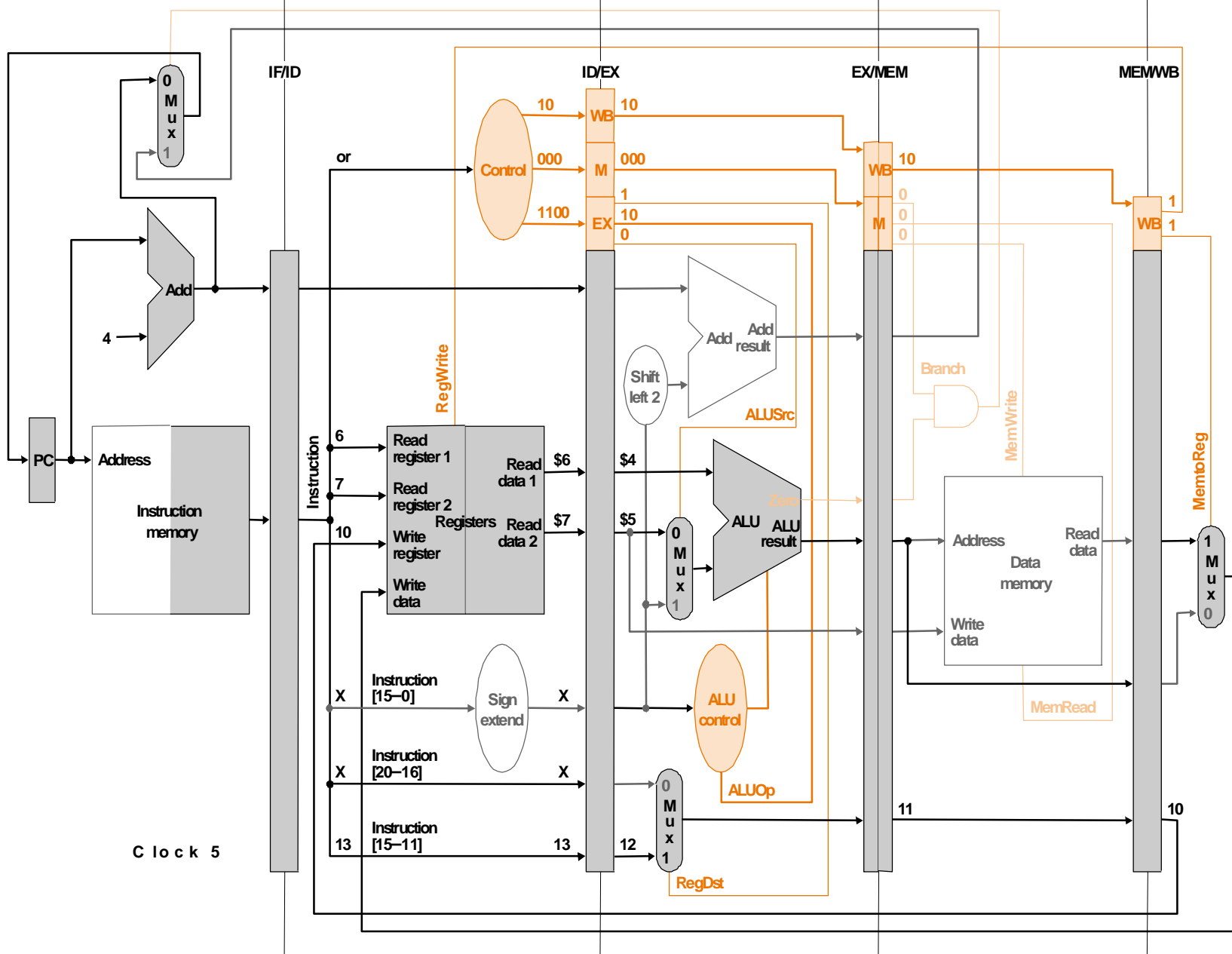
IF: add \$14, \$8, \$9

ID: or \$13, \$6, \$7

EX: and \$12, ...

MEM: sub \$11, ...

WB: lw \$10, ...



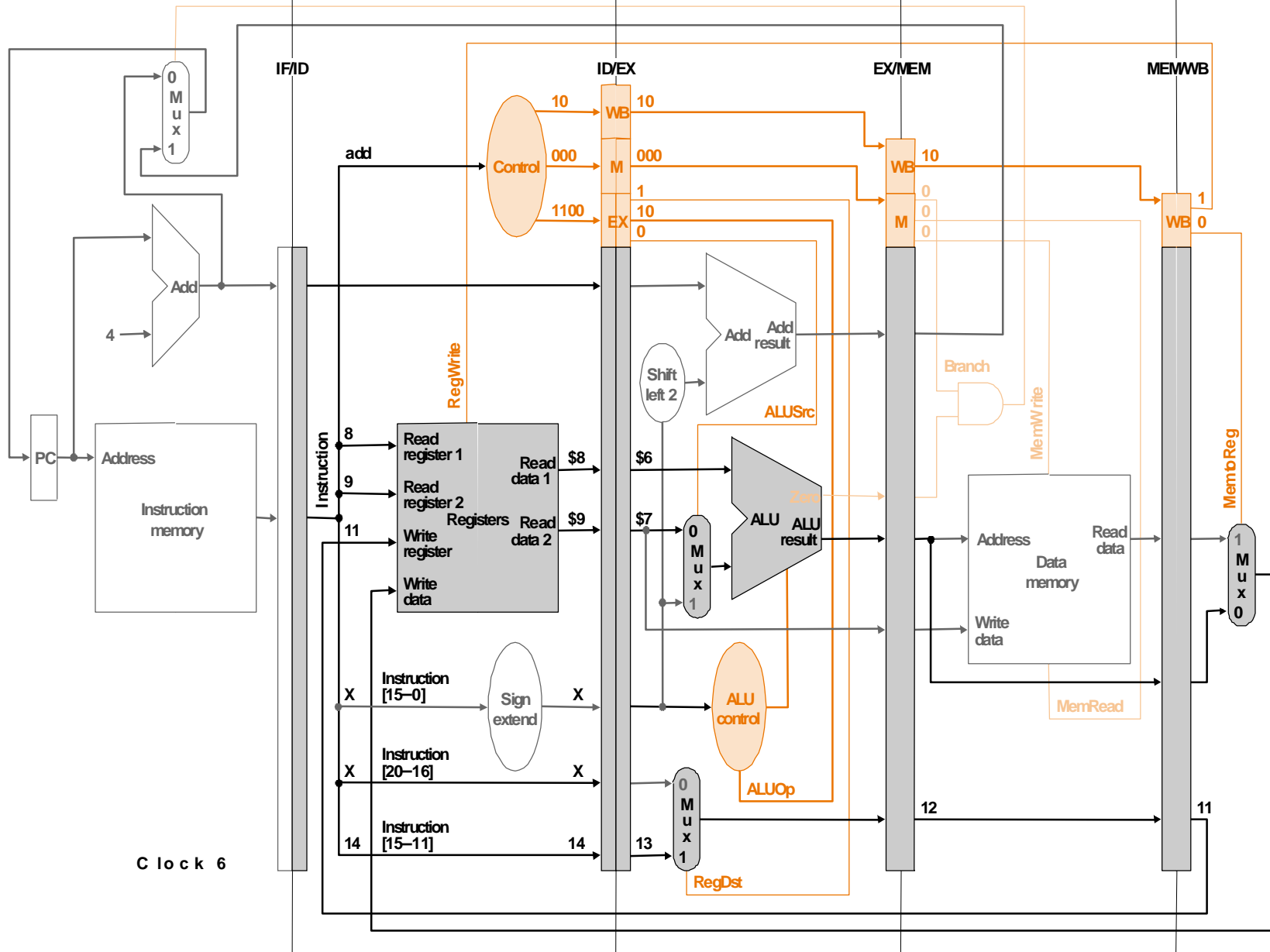
IF: after<1>

ID: add \$14, \$8, \$9

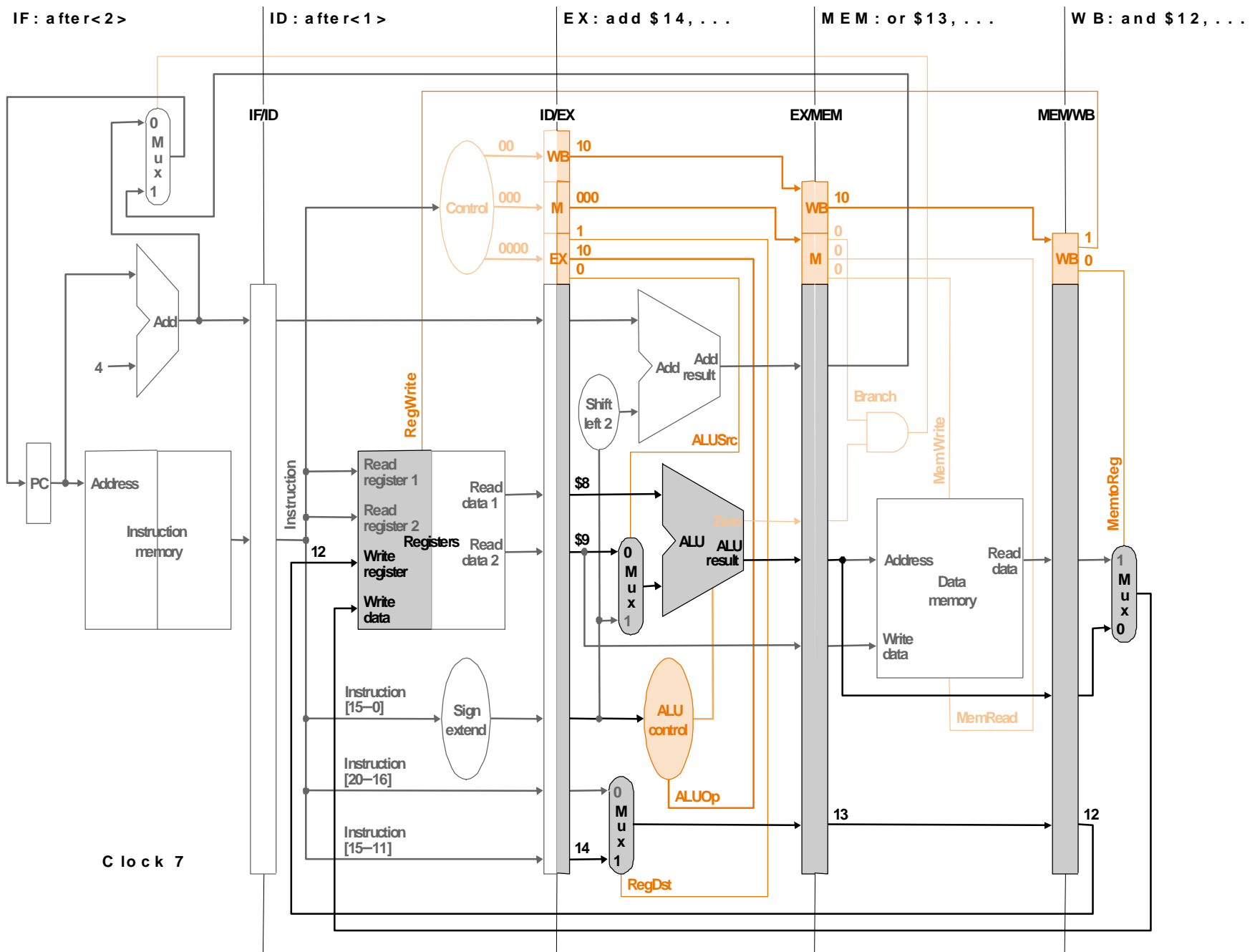
EX: or \$13, ...

MEM: and \$12, ...

WB: sub \$11, ...







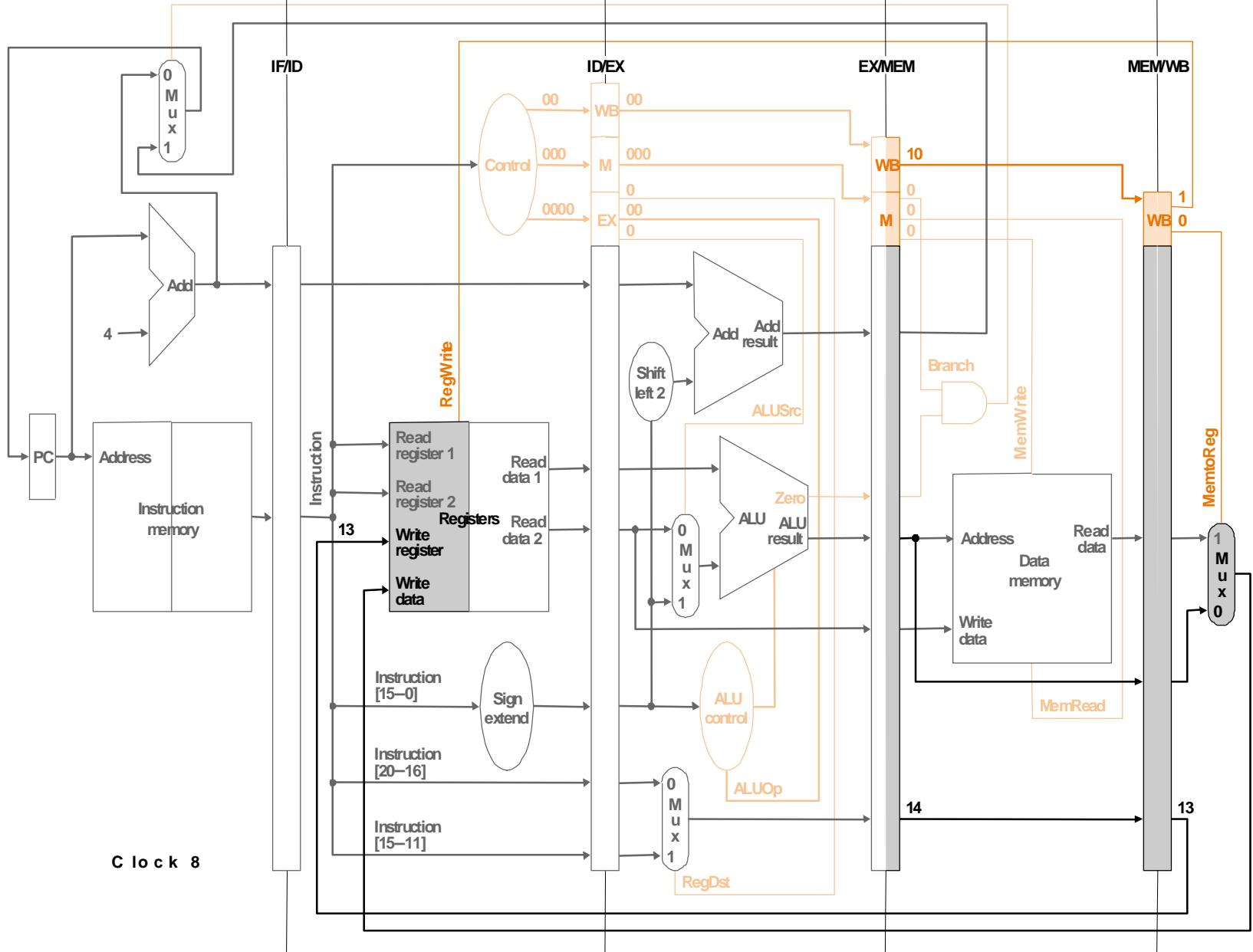
IF : after<3>

ID : after<2>

EX : after<1>

MEM : add \$14, ...

WB : or \$13, ...



# Pipelined Execution and Control

