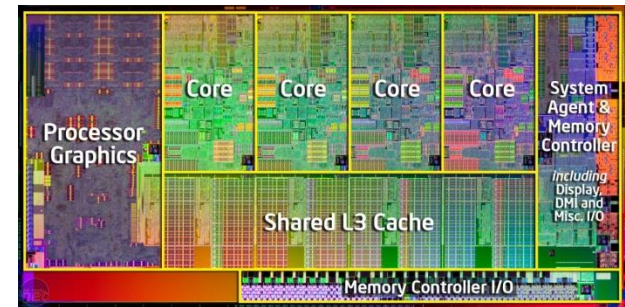




COSE222, COMP212 Computer Architecture

Lecture 5. MIPS Processor Design

Pipelined MIPS #2

Prof. Taeweon Suh

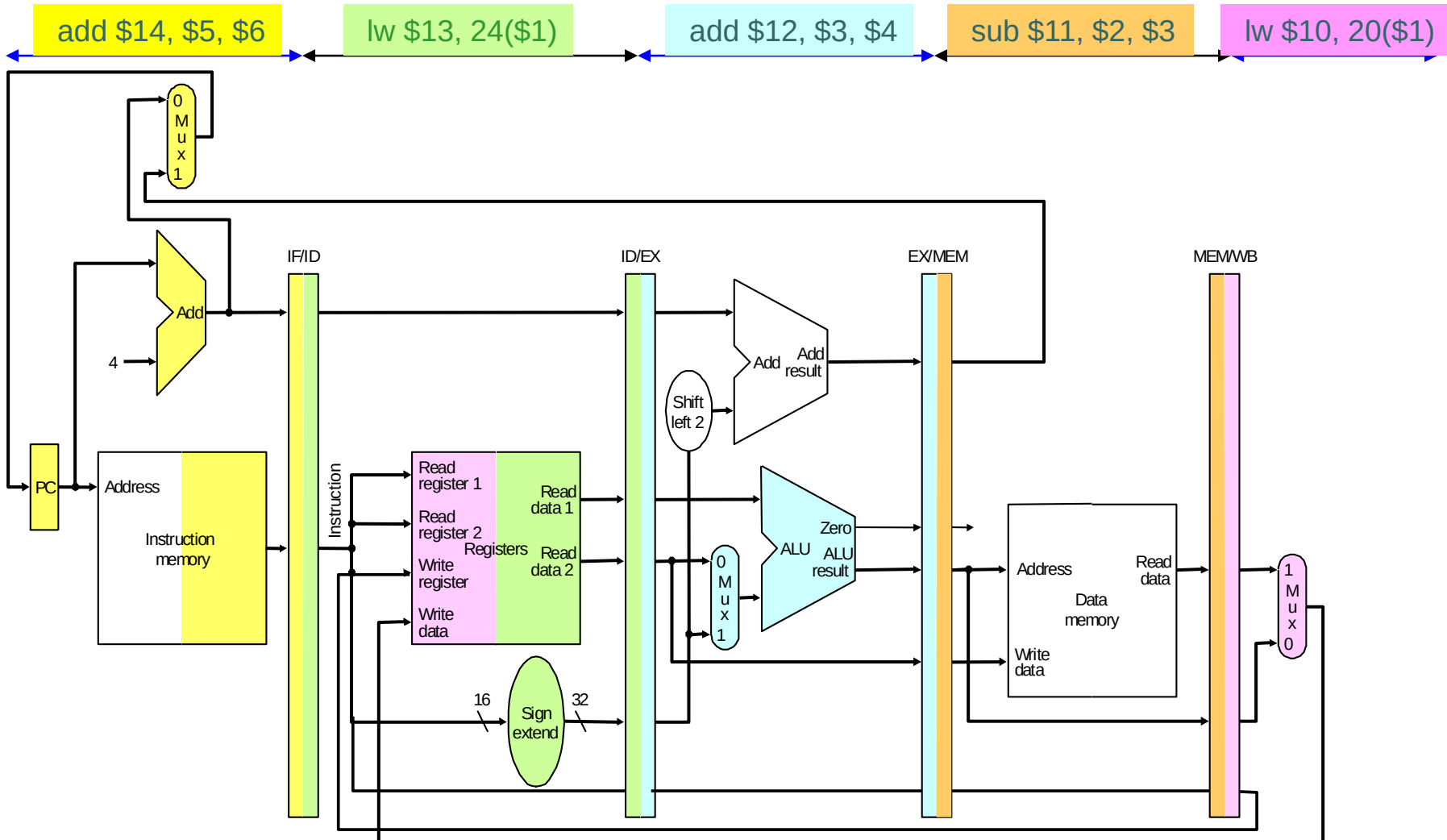
Computer Science & Engineering

Korea University

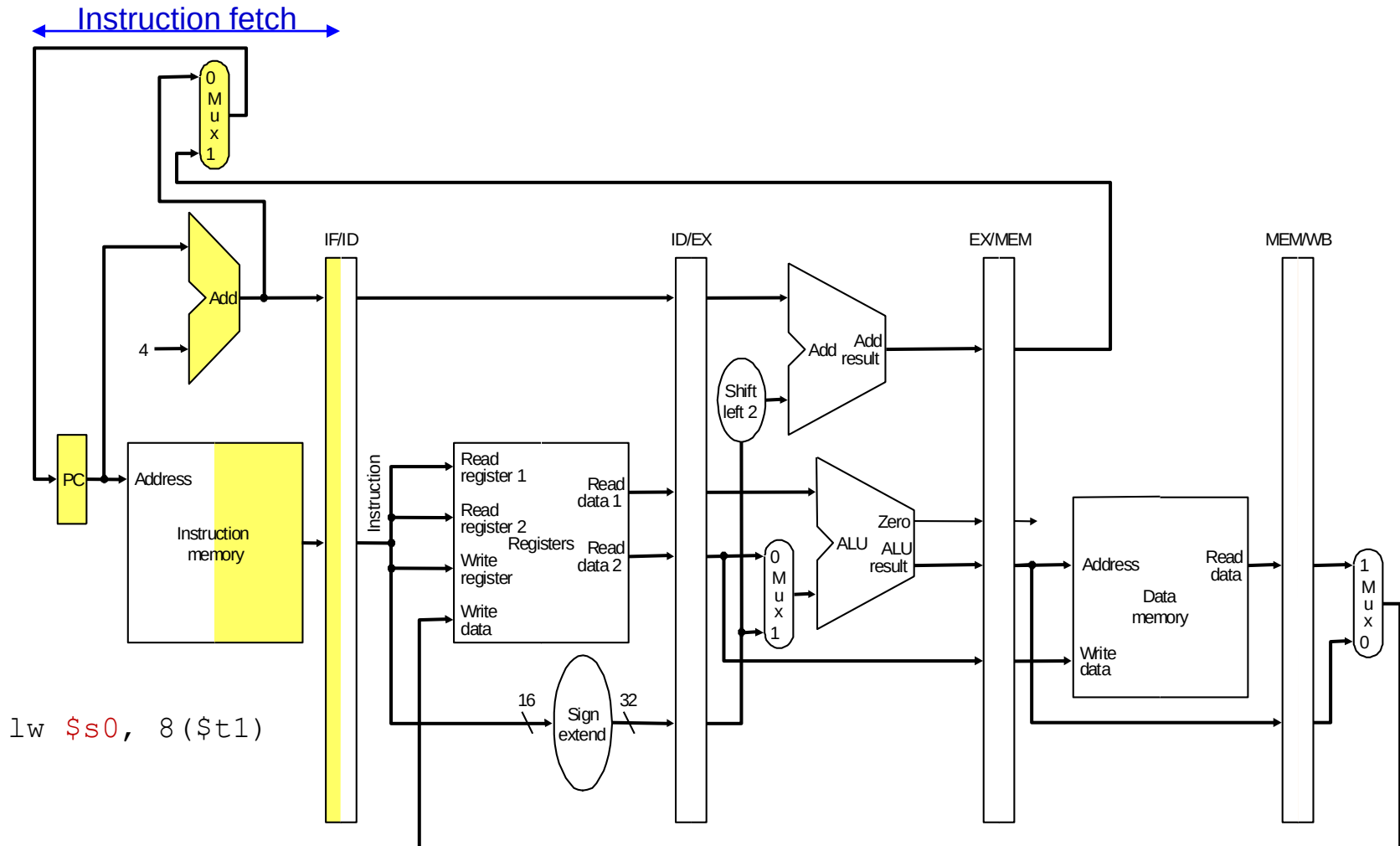
Korea Univ



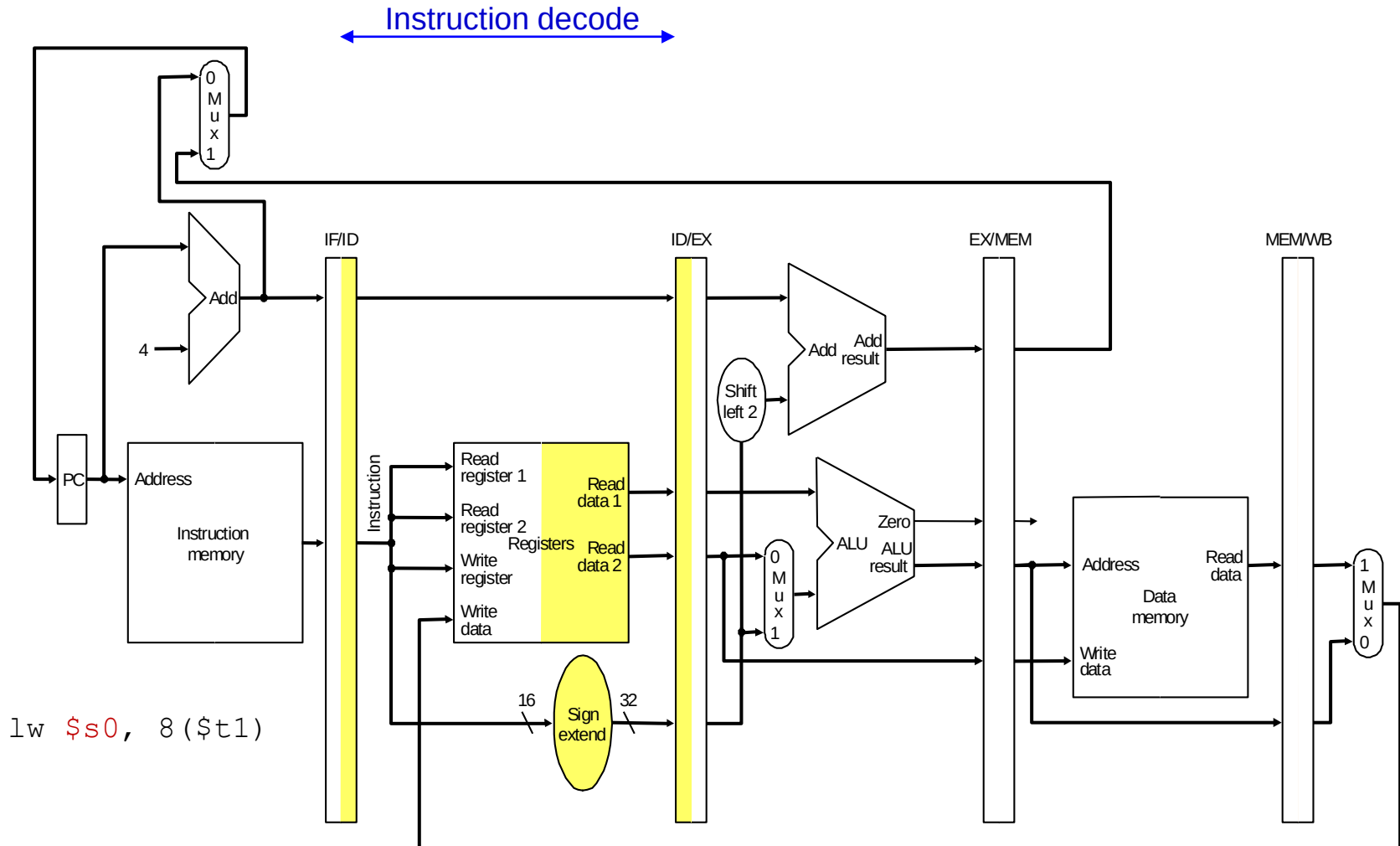
Pipelining Example



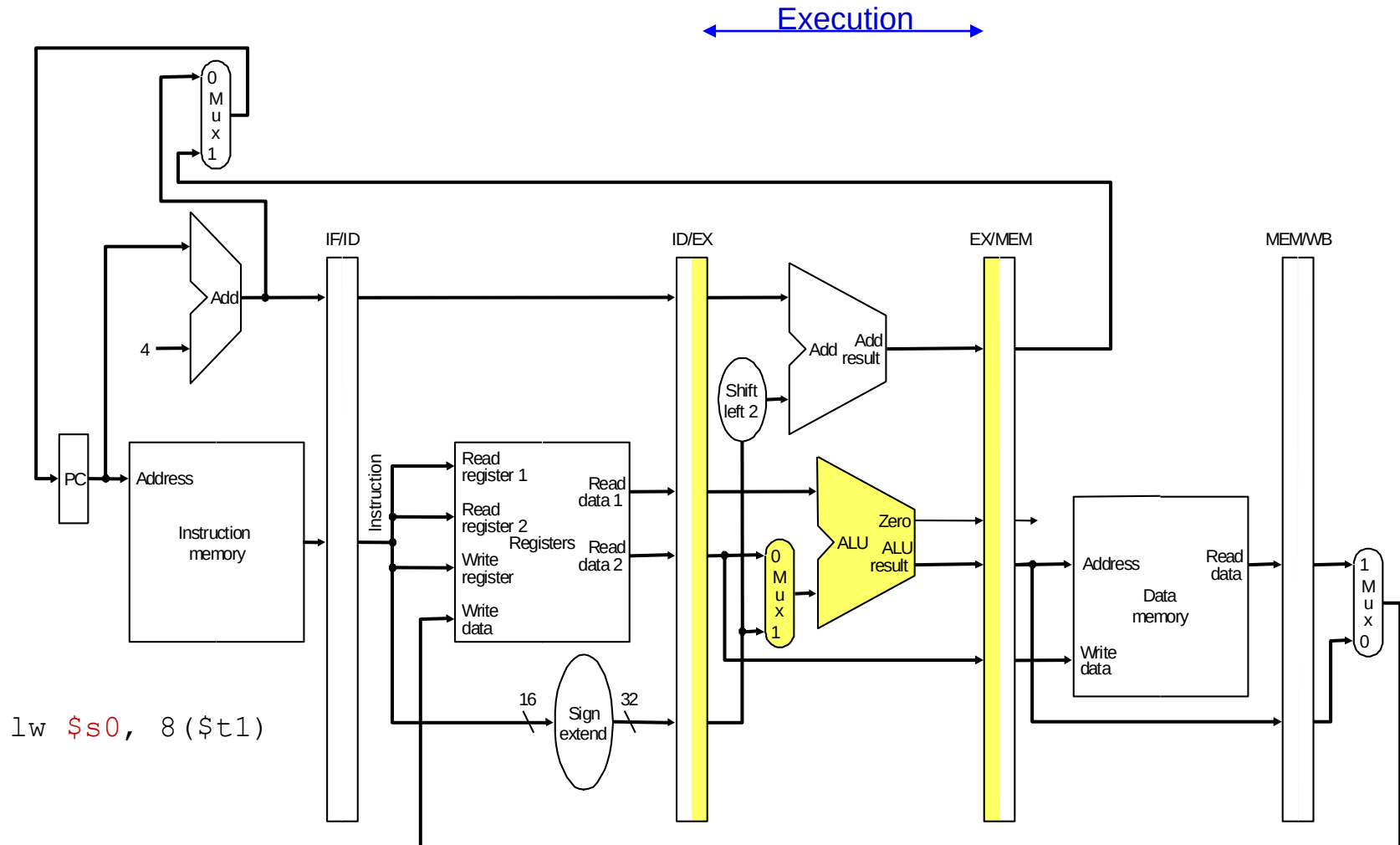
lw: Instruction Fetch (IF)



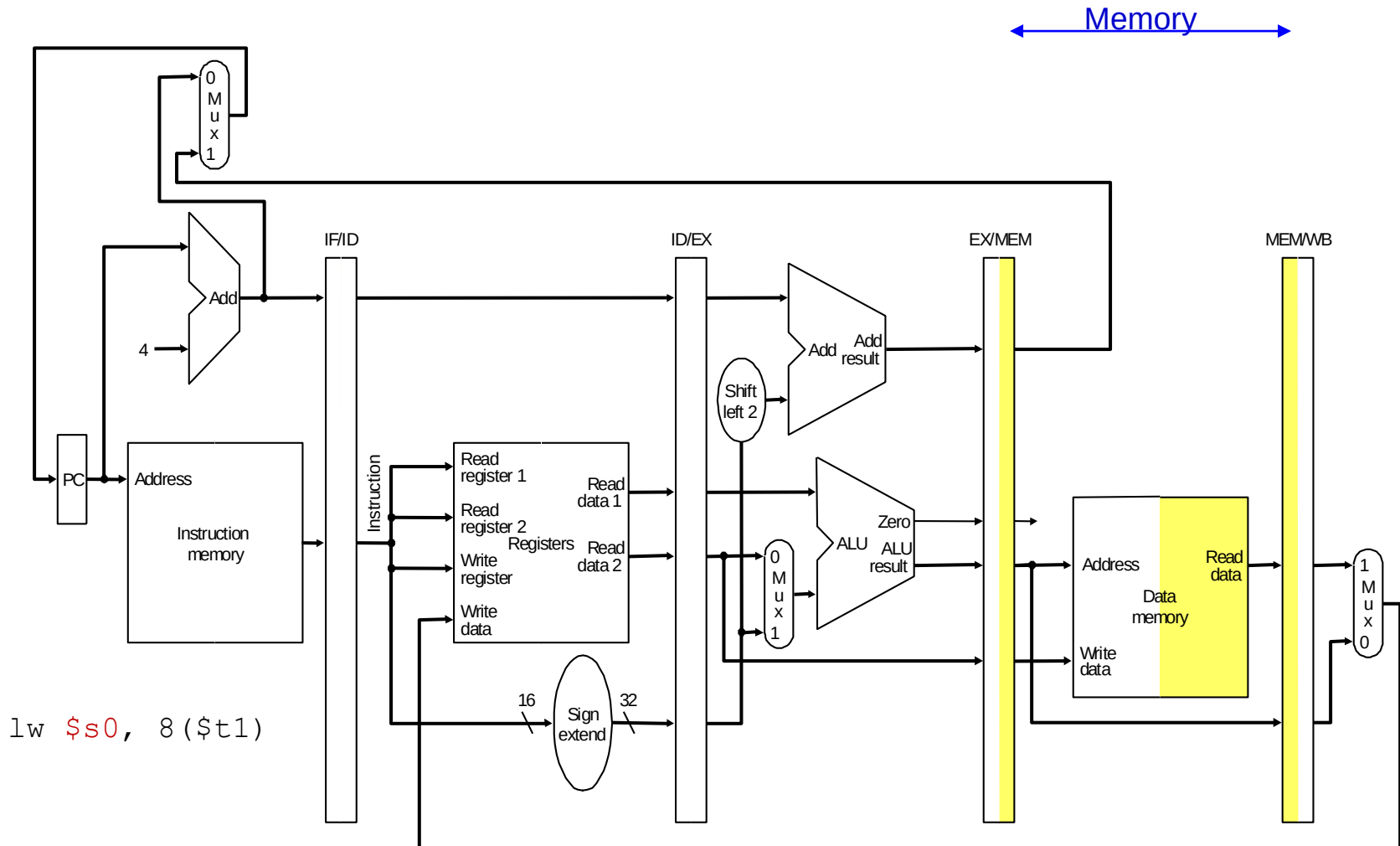
1w: Instruction Decode (ID)



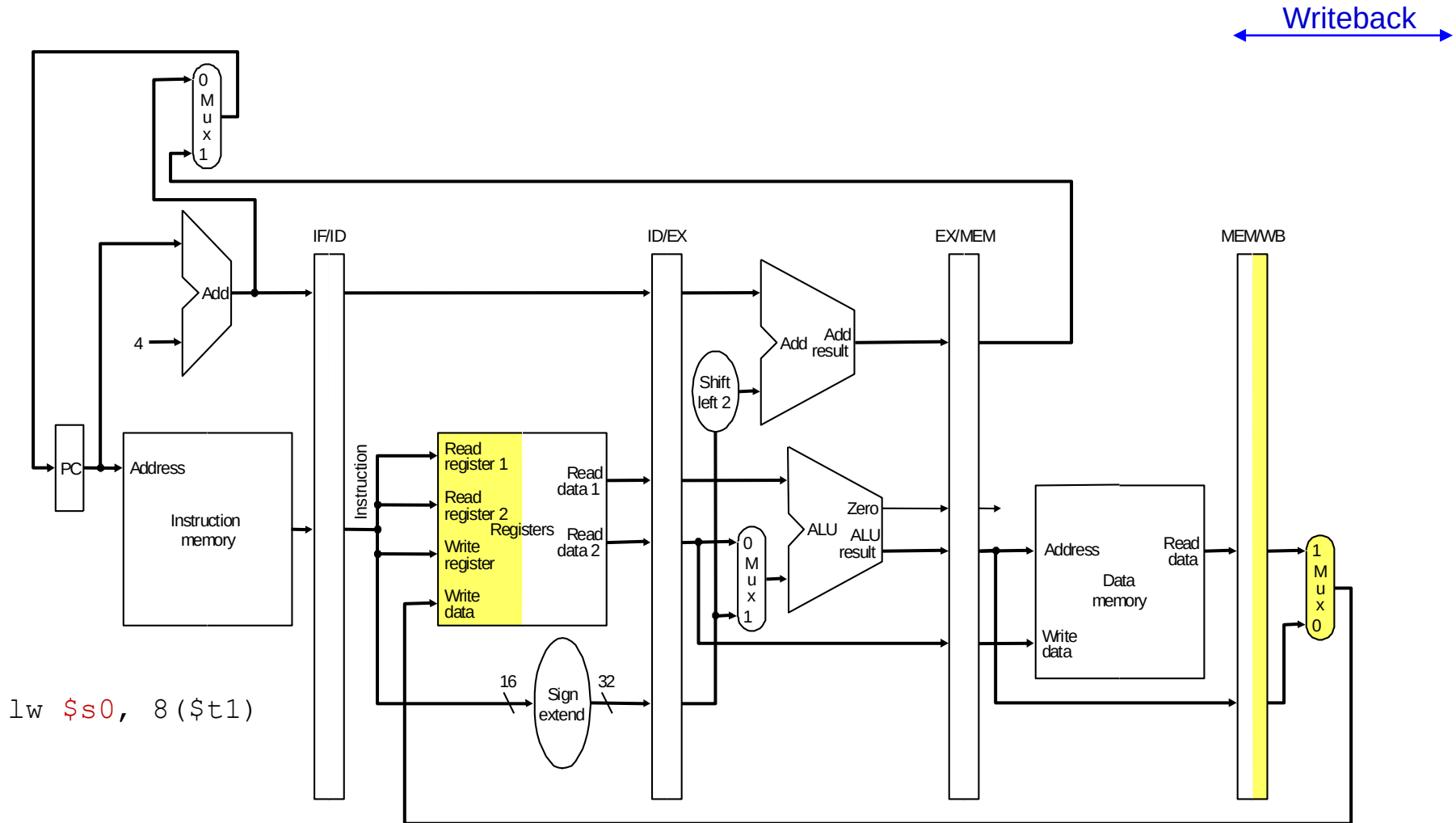
1w: Execution (EX)



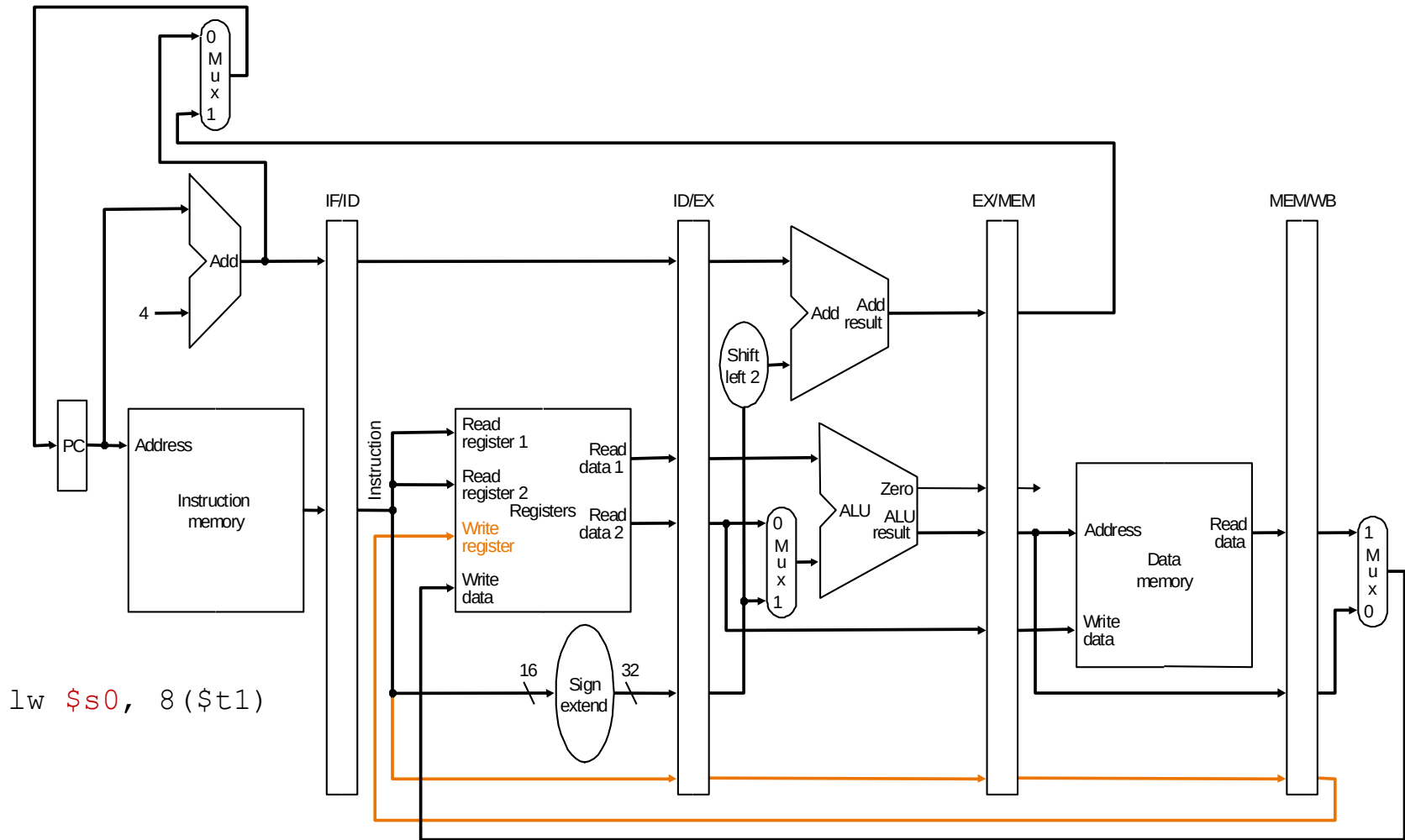
1w: Memory (MEM)



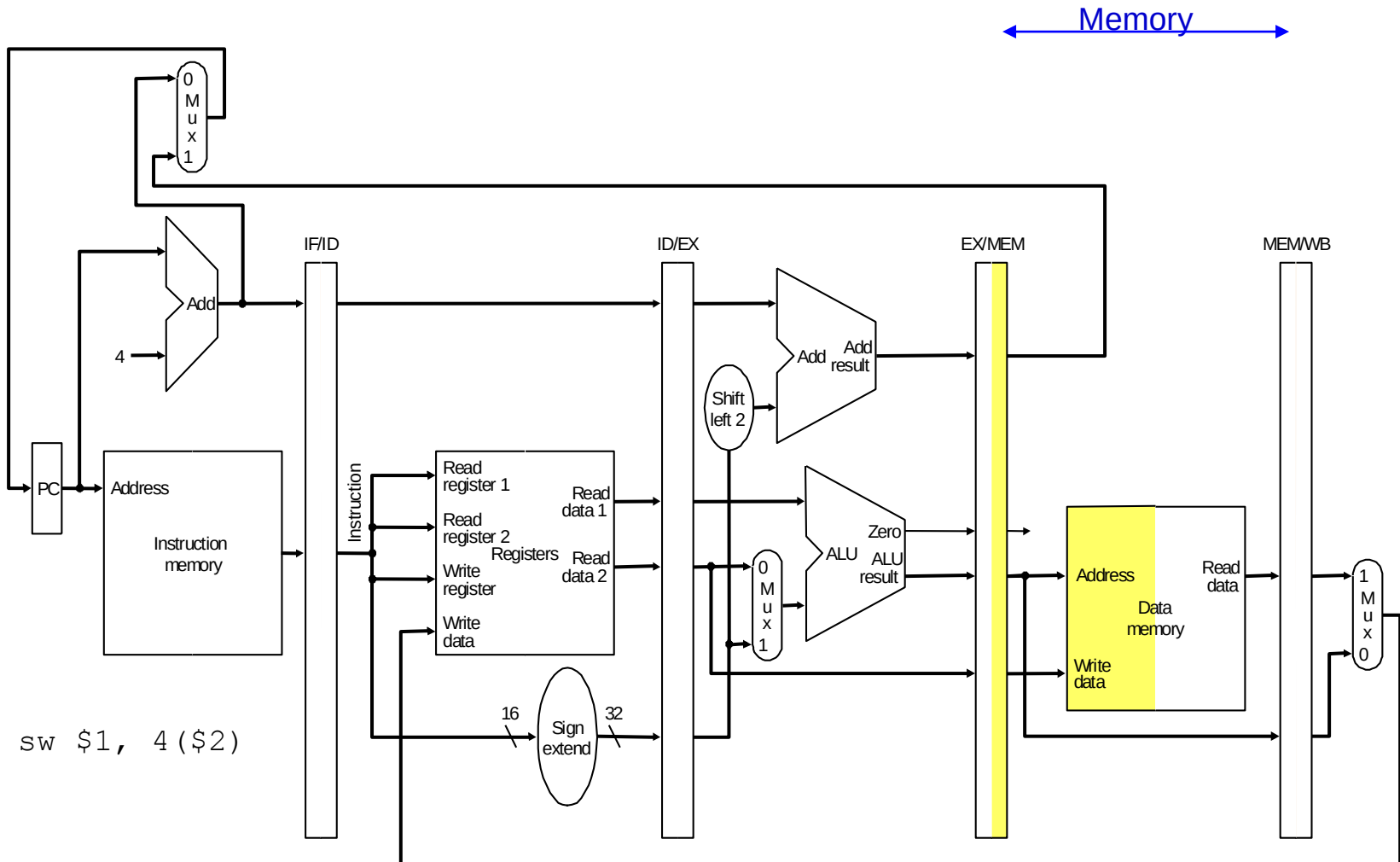
lw: Writeback (WB)



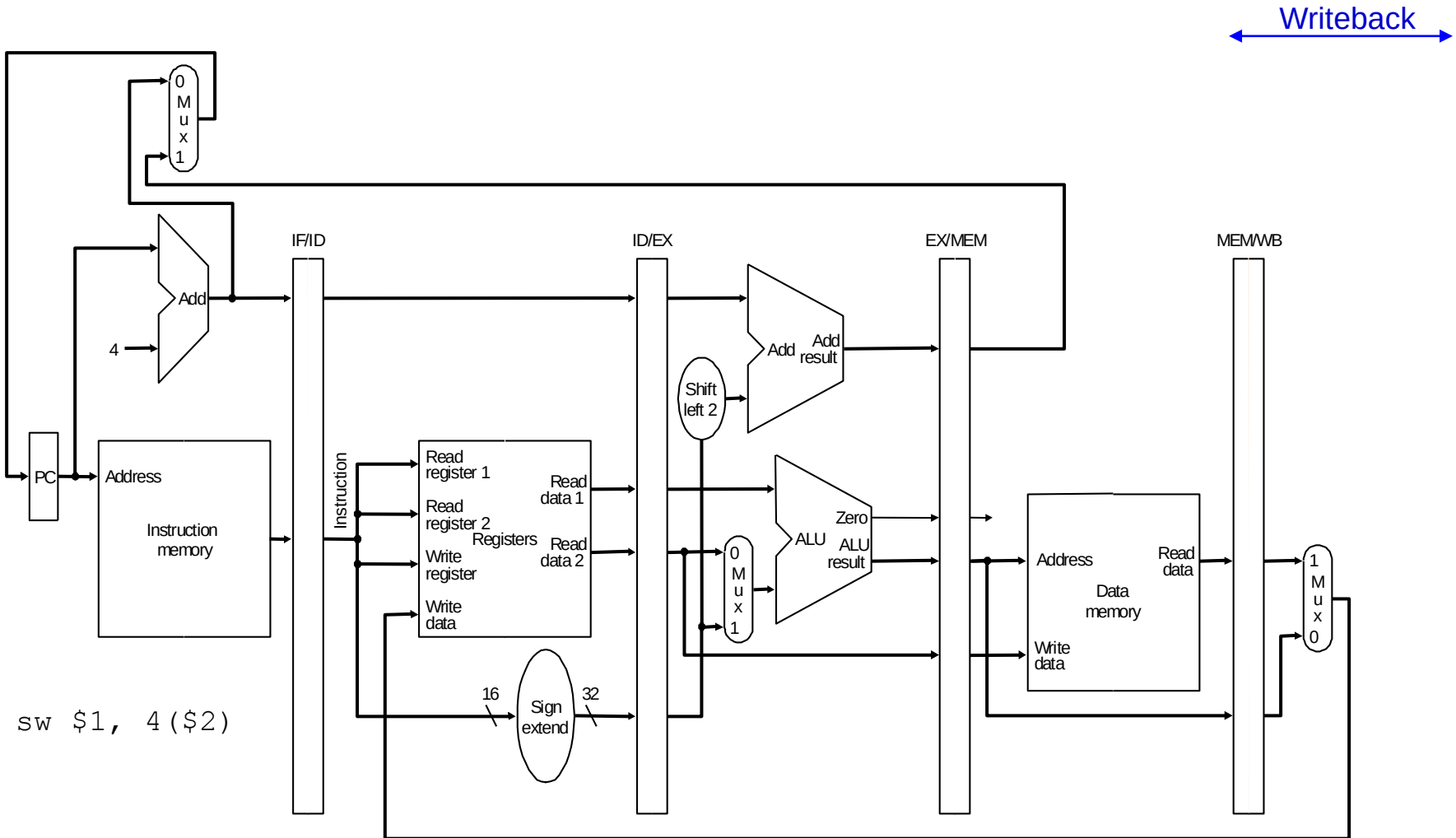
Corrected Datapath (for lw)



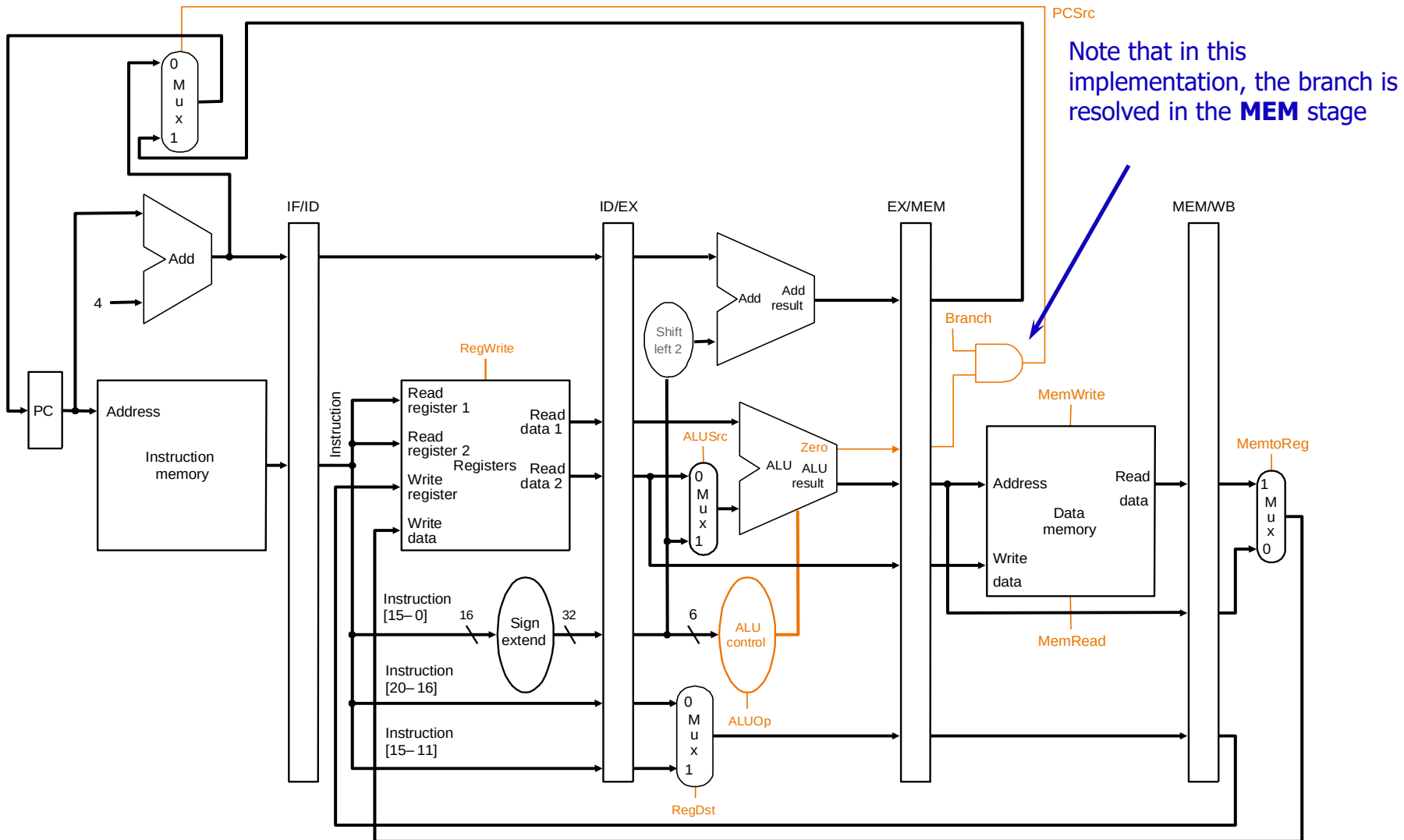
sw: Memory (MEM)



sw: Writeback (WB): do nothing



Pipeline Control



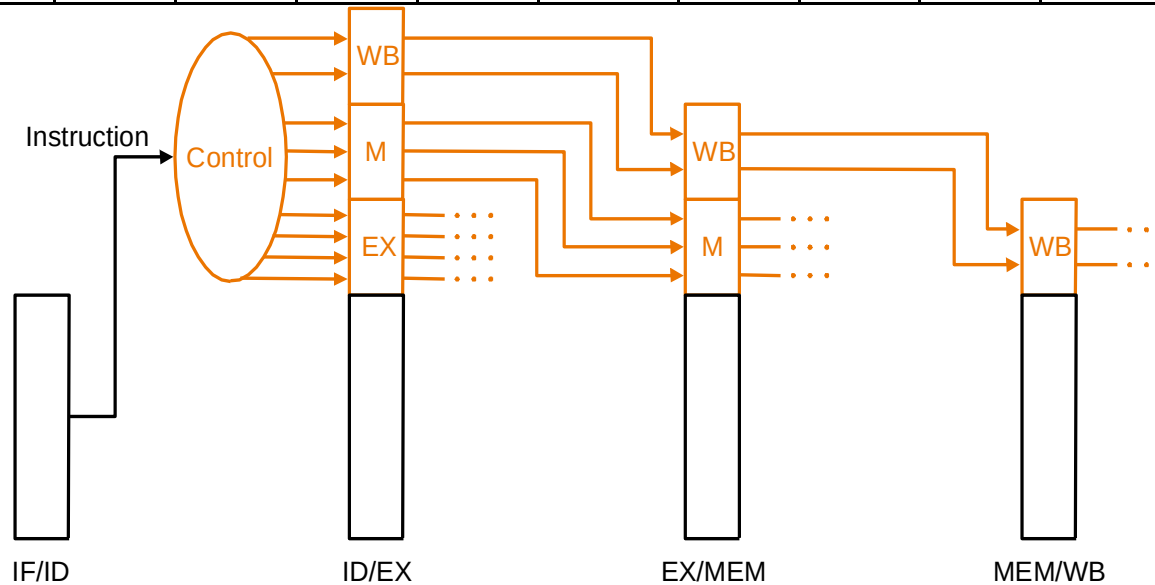
Pipeline Control

- What needs to be controlled in each stage (IF, ID, EX, MEM, WB)?
 - IF: Instruction fetch and PC increment
 - ID: Instruction decode and operand fetch from register file and/or immediate
 - EX: Execution stage
 - RegDst
 - ALUOp[1:0]
 - ALUSrc
 - MA: Memory stage
 - Branch
 - MemRead
 - MemWrite
 - WB: Writeback
 - MemtoReg
 - RegWrite (note that this signal is in ID stage)

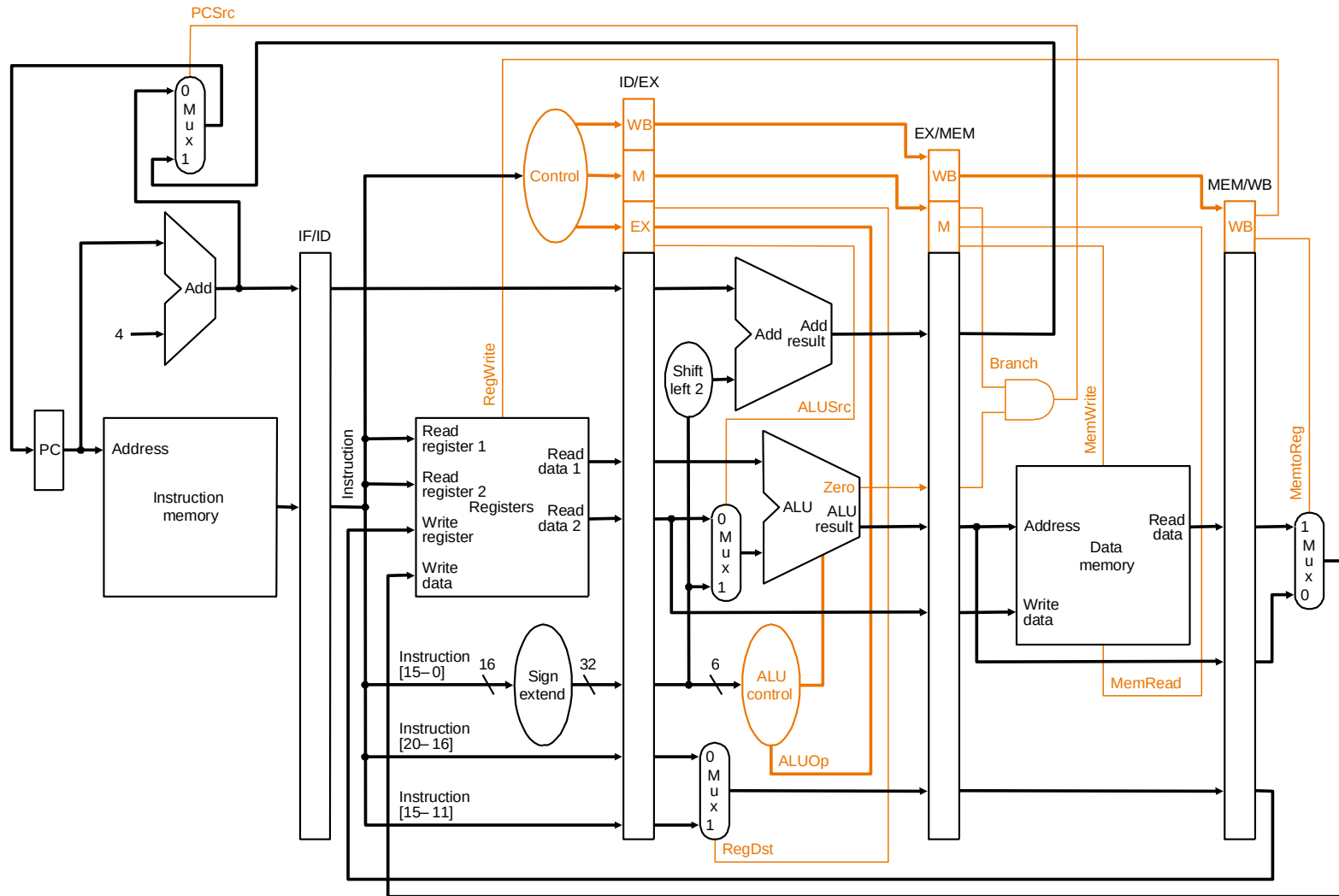
Pipeline Control

- Extend pipeline registers to include control information created in ID stage
- Pass control signals along just like the data

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	Branch	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

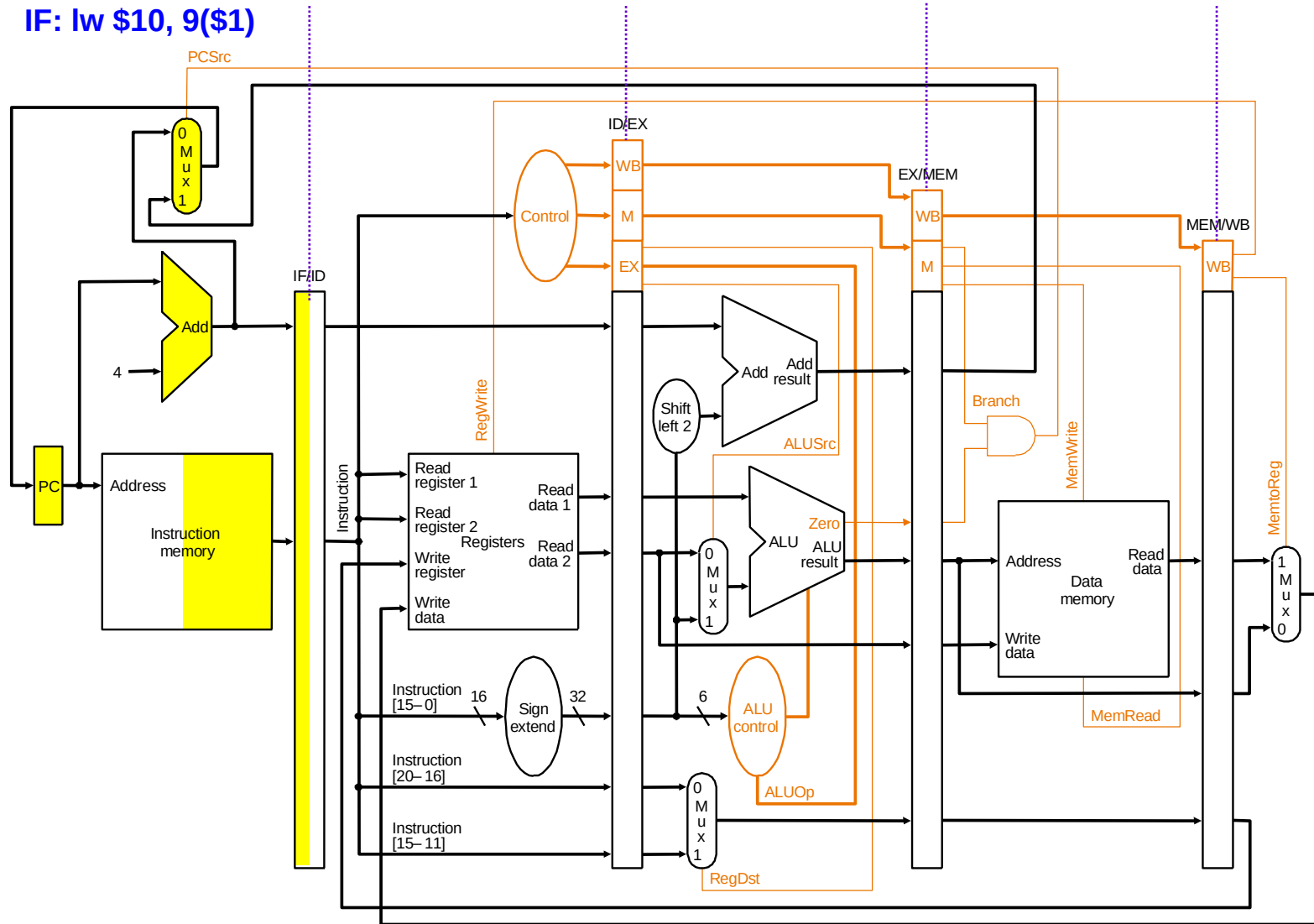


Datapath with Control



Datapath with Control

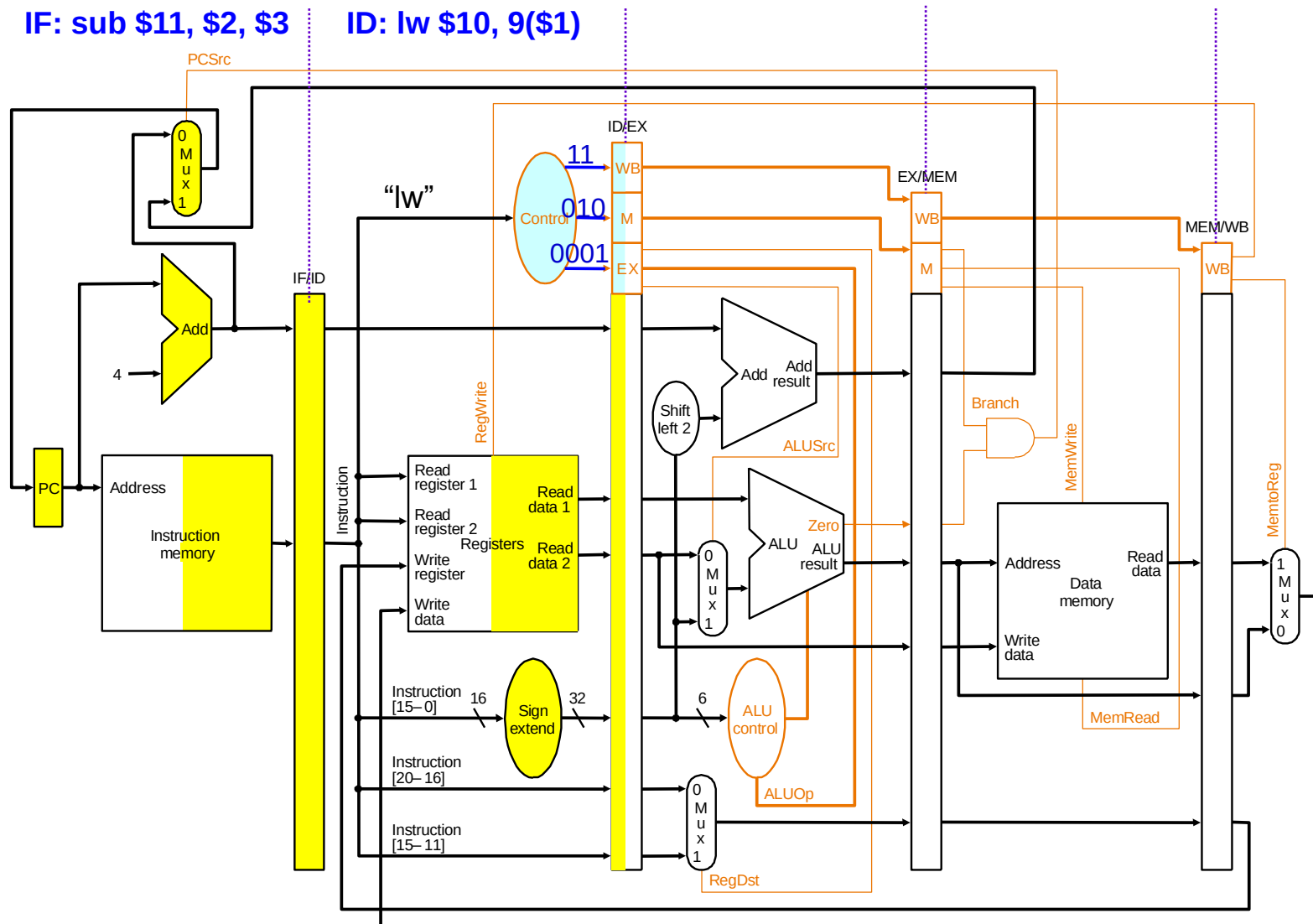
IF: lw \$10, 9(\$1)



Datapath with Control

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

ID: lw \$10, 9(\$1)

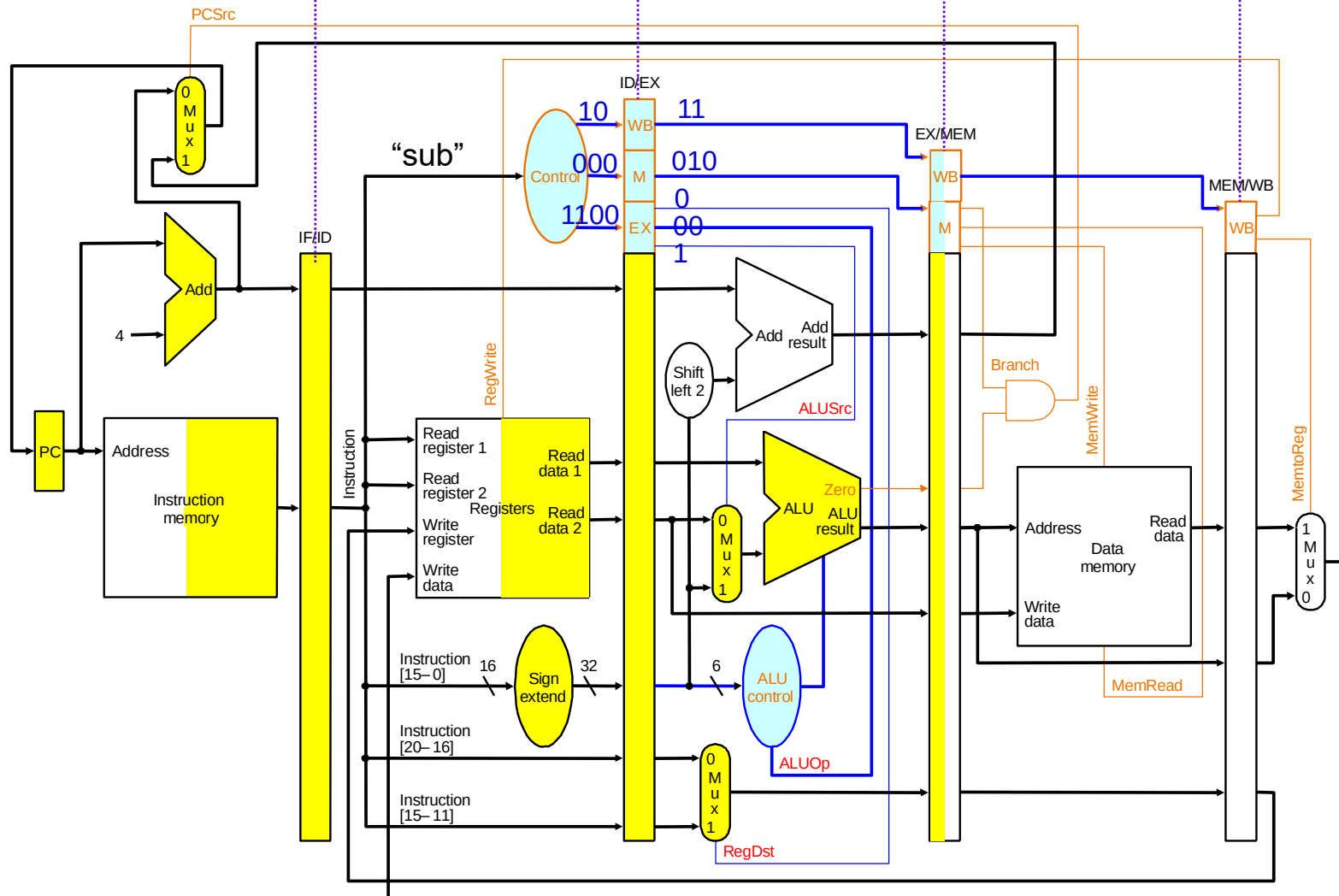


Datapath with Control

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

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

EX: lw \$10, 9(\$1)



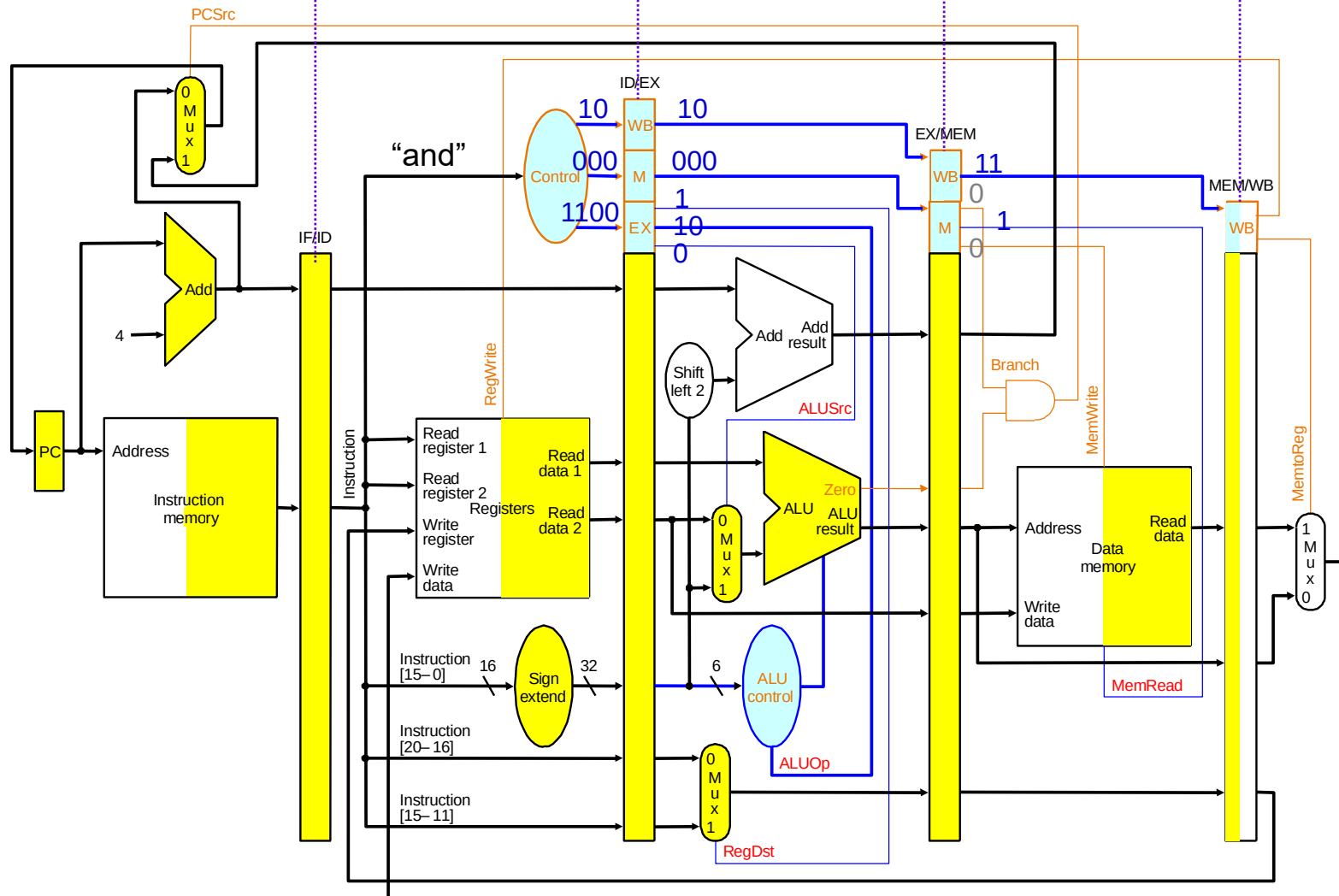
Datapath with Control

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

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

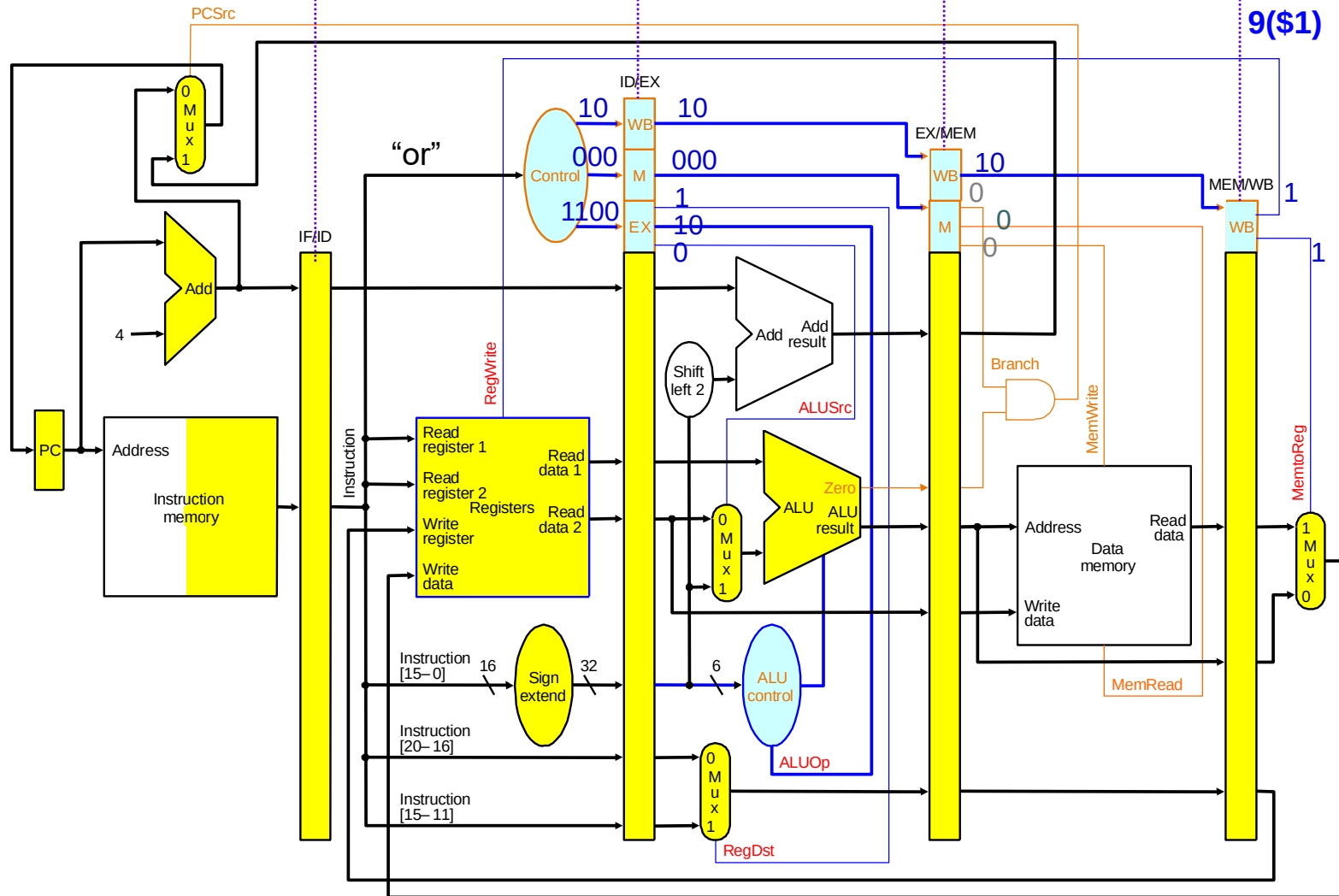
EX: sub \$11, \$2, \$3

MEM: lw \$10, 9(\$1)

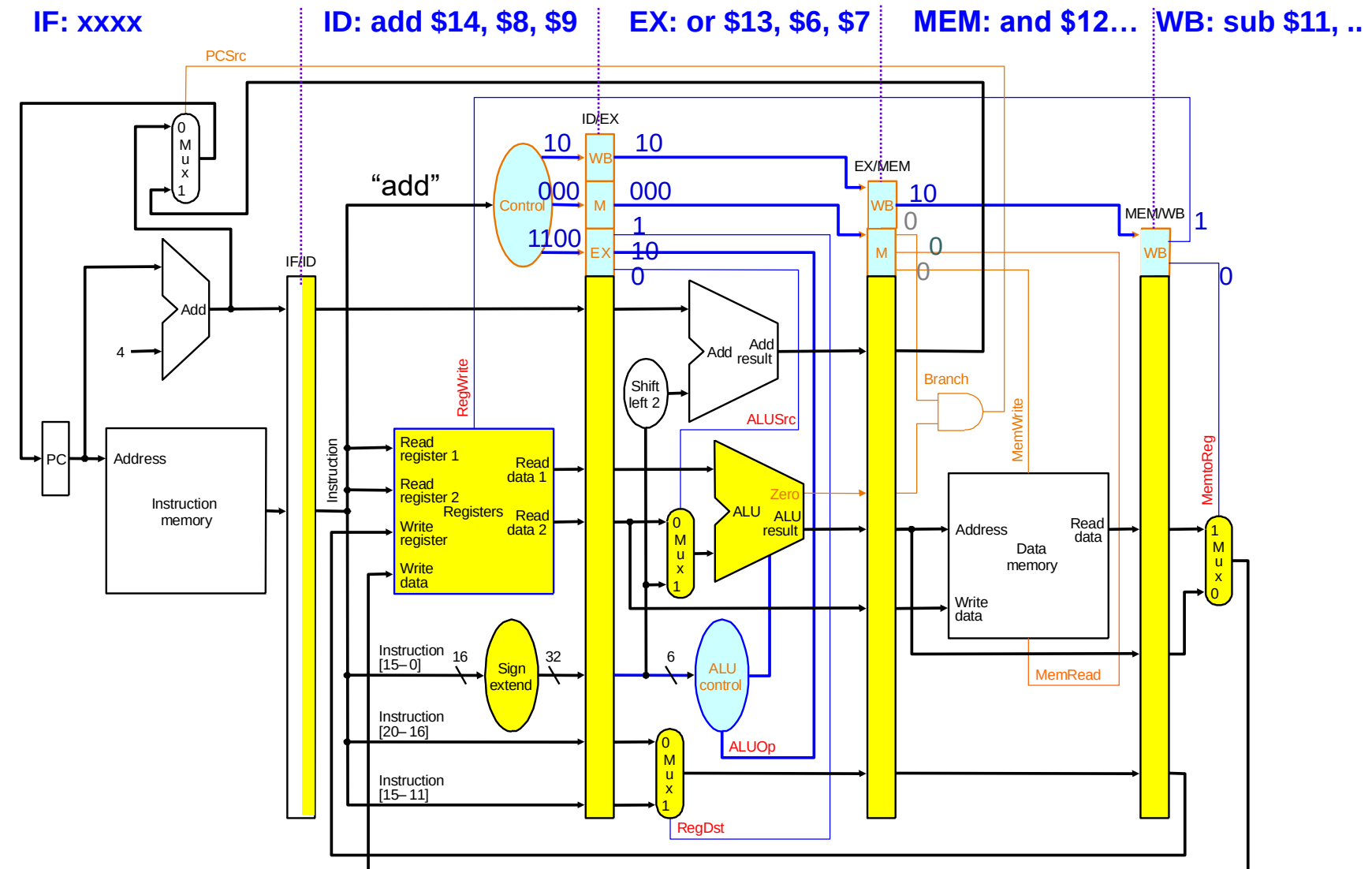


Datapath with Control

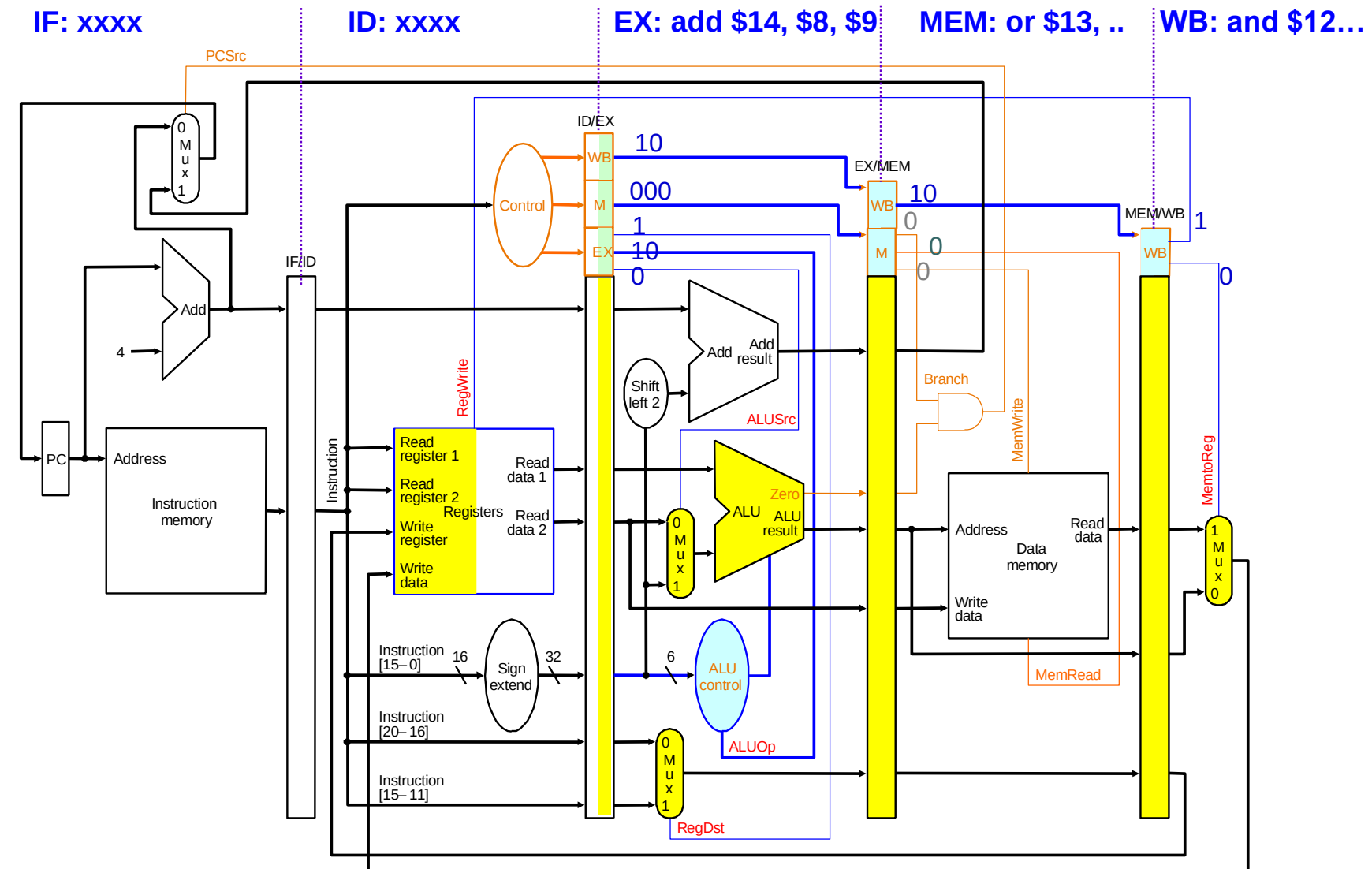
IF: add \$14, \$8, \$9 ID: or \$13, \$6, \$7 EX: and \$12, \$4, \$5 MEM: sub \$11, .. WB: lw \$10, 9(\$1)



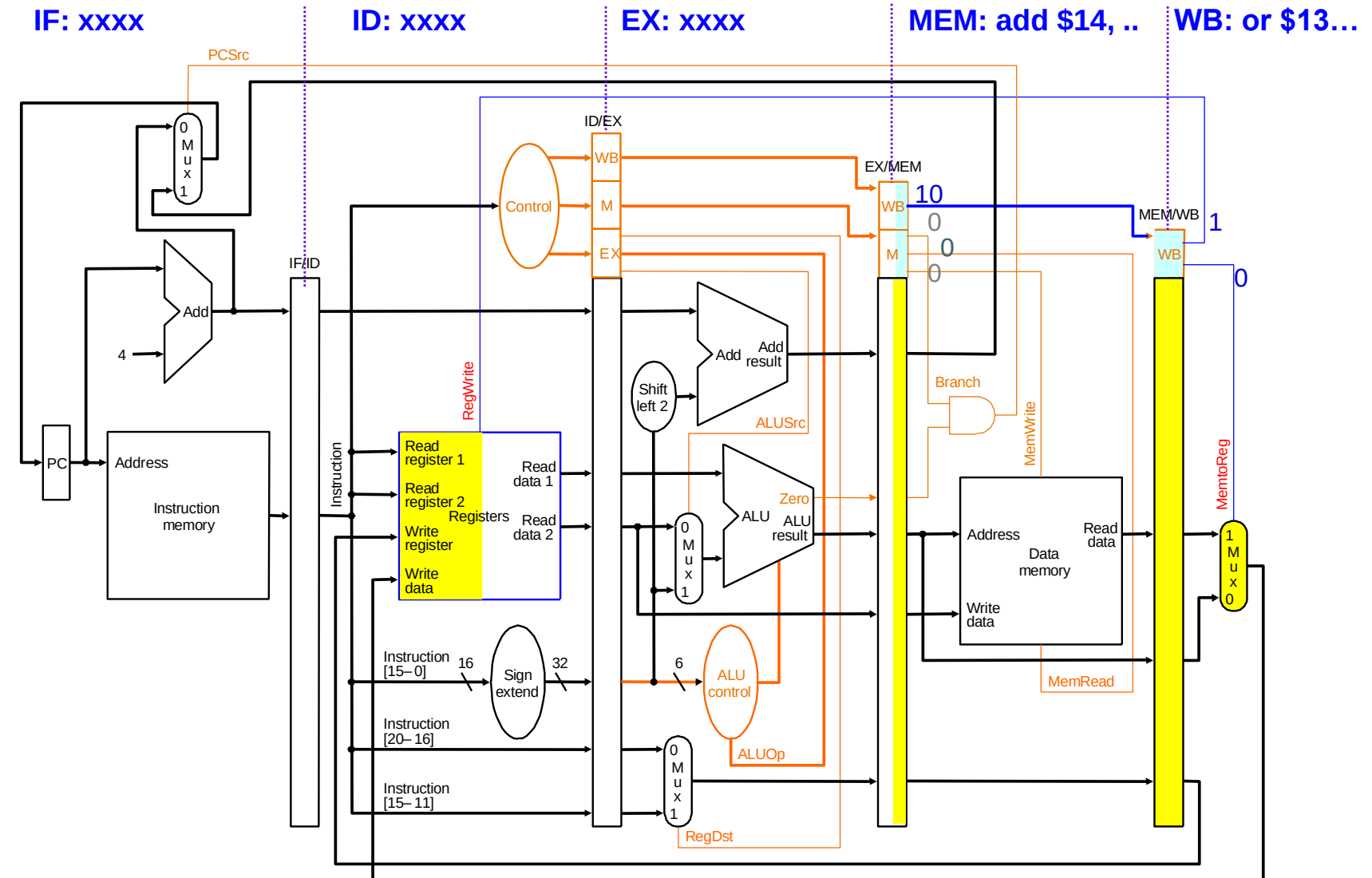
Datapath with Control



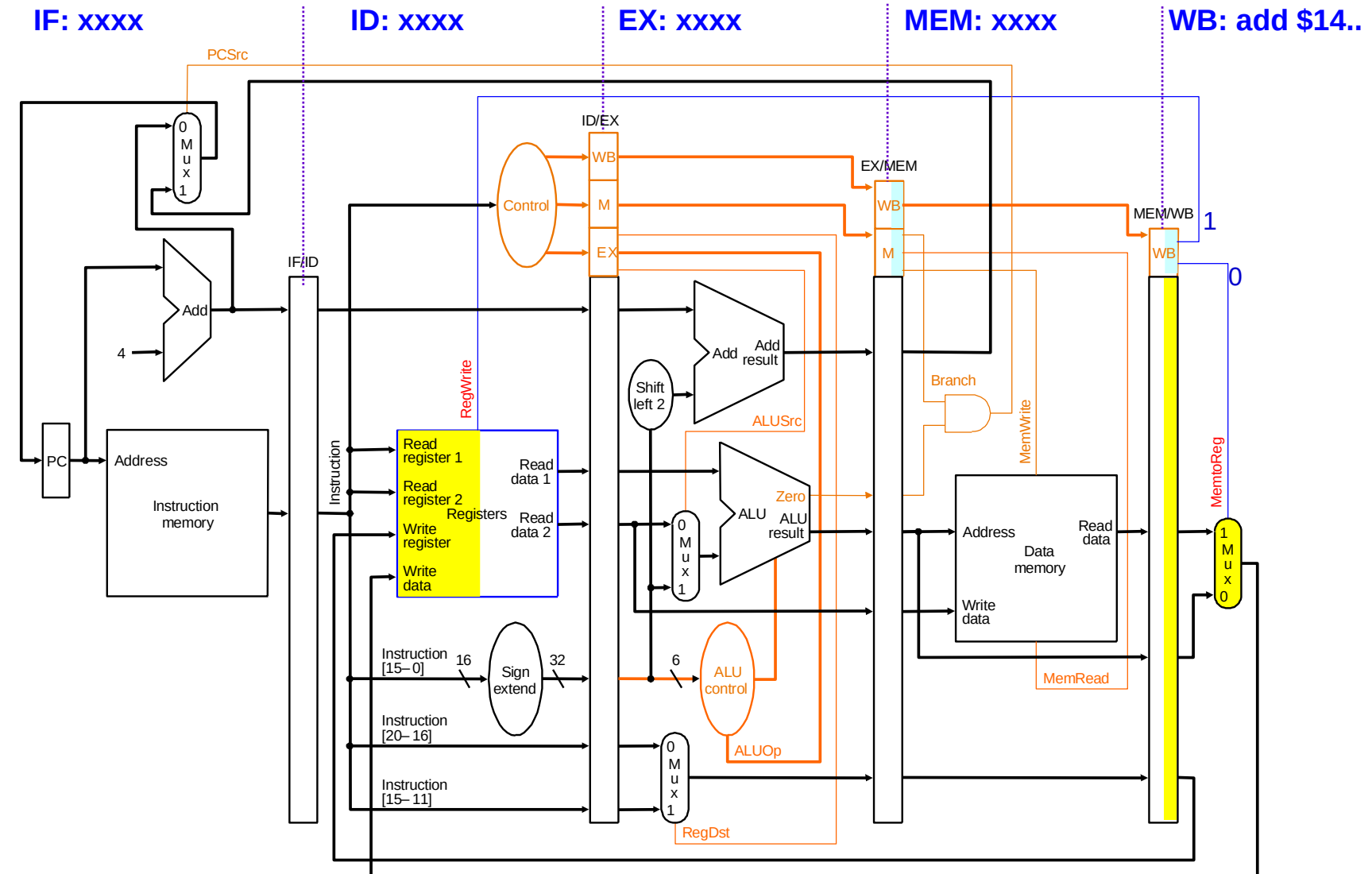
Datapath with Control



Datapath with Control

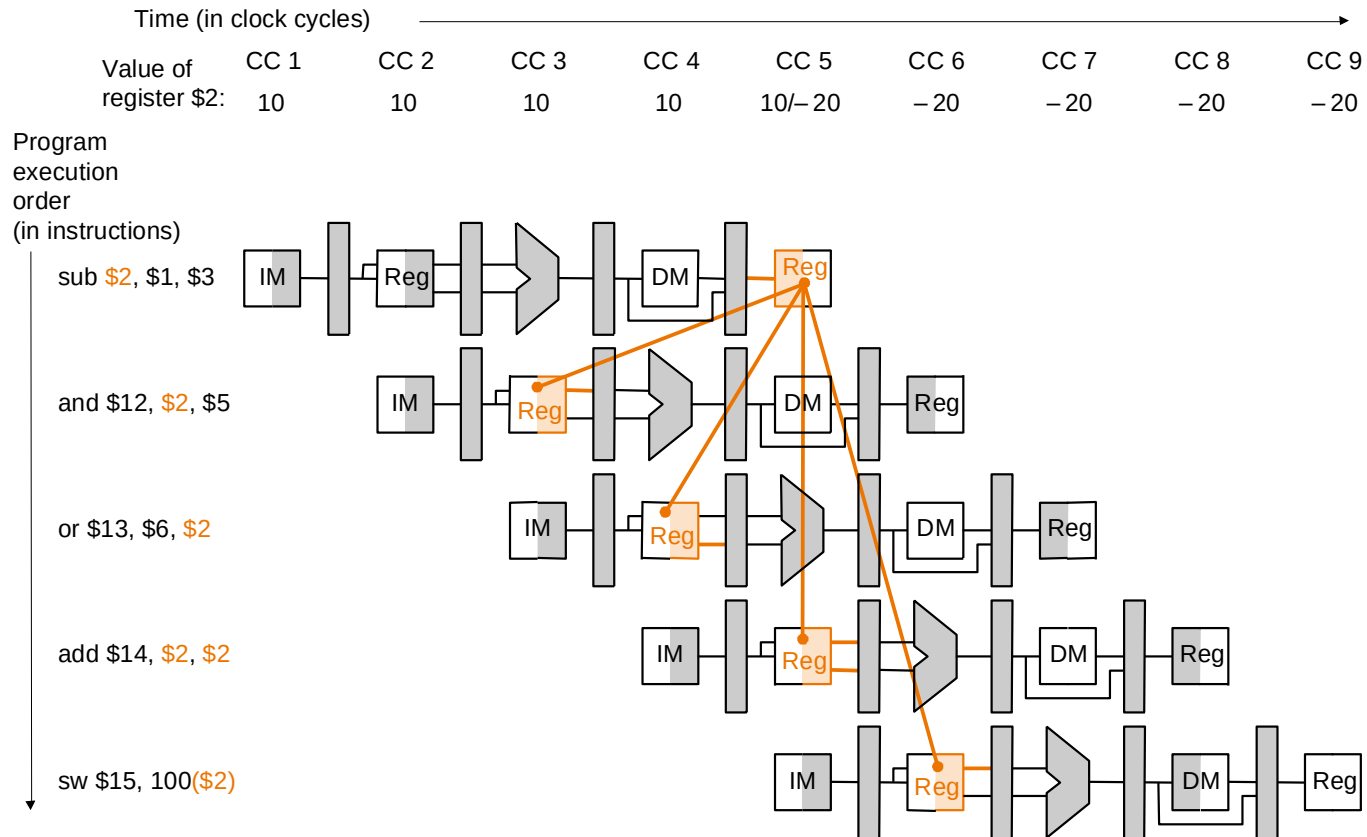


Datapath with Control



Dependencies

- Dependencies incur data and control hazards



Data Hazard - Software Solution

- Compiler techniques
 - Insert **nop** (0x0000_0000) between instructions
 - Where do we insert nops in the following example?

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

- However, it really slows us down!
- Code scheduling reorganizes the code so that it relieves the dependencies between instructions

Data Hazard - Forwarding

- Don't wait for them to be written to the register file
 - Use temporary results

	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

Program execution order (in instructions)

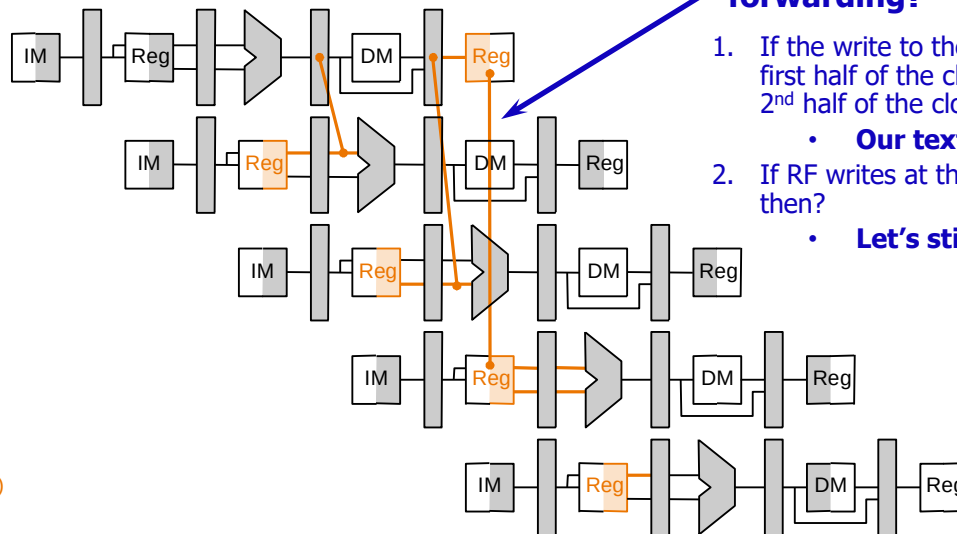
sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2

add \$14, \$2, \$2

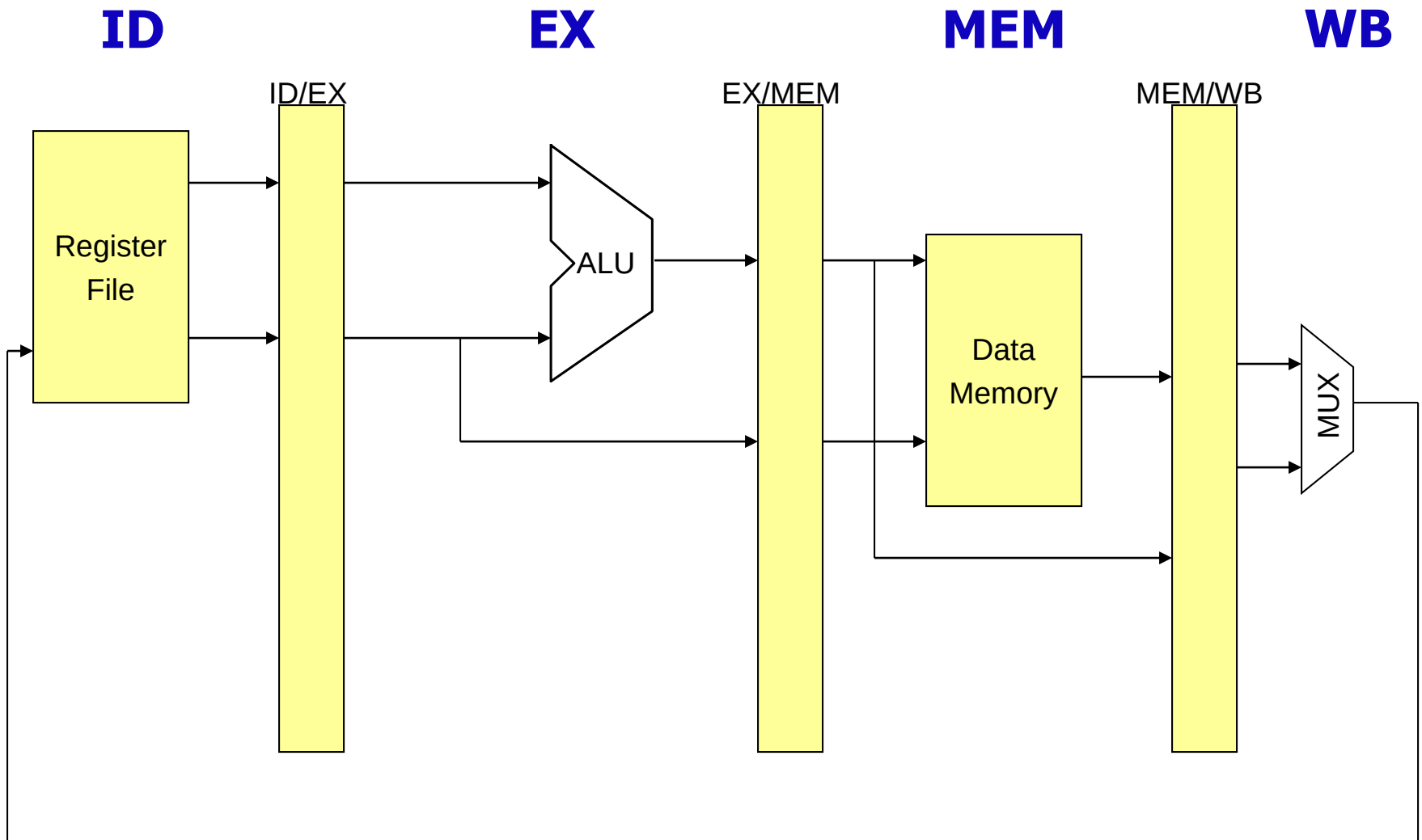
sw \$15, 100(\$2)



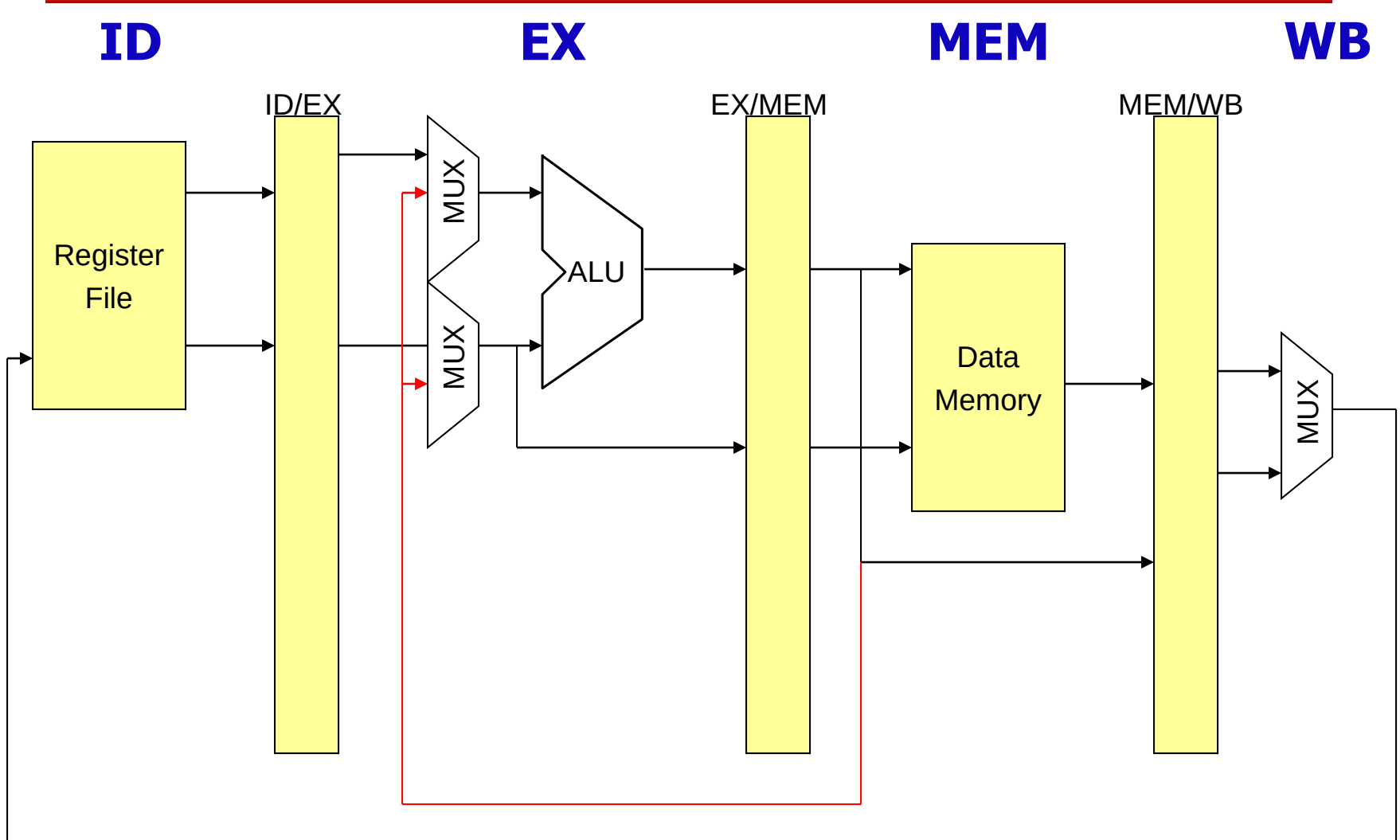
Ok.. Then, do we have to do this forwarding?

1. If the write to the register file occurs in the first half of the clock, and read occurs in the 2nd half of the clock, then?
 - **Our textbook follows this**
2. If RF writes at the rising-edge of the clock, then?
 - **Let's stick to this for our project**

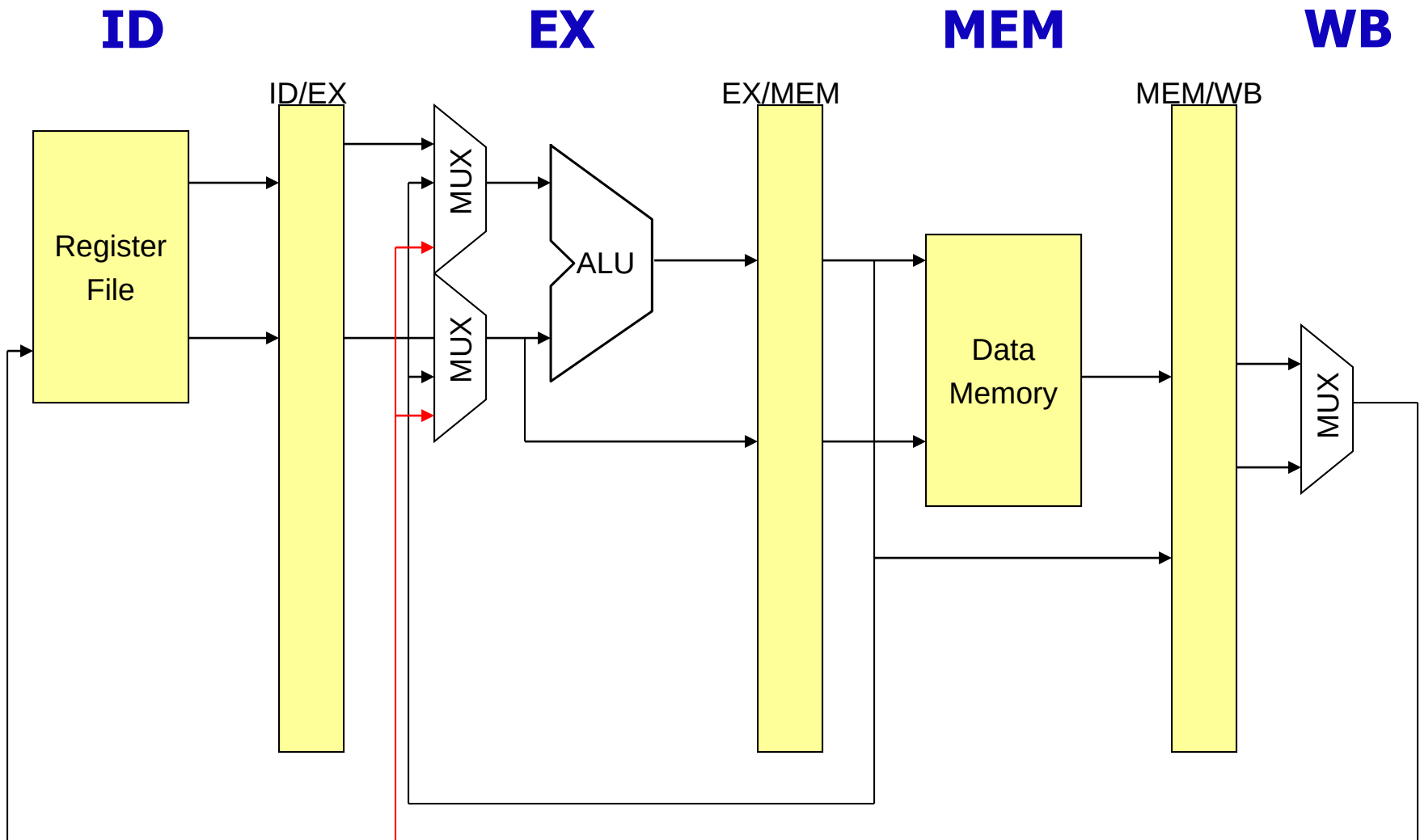
Forwarding



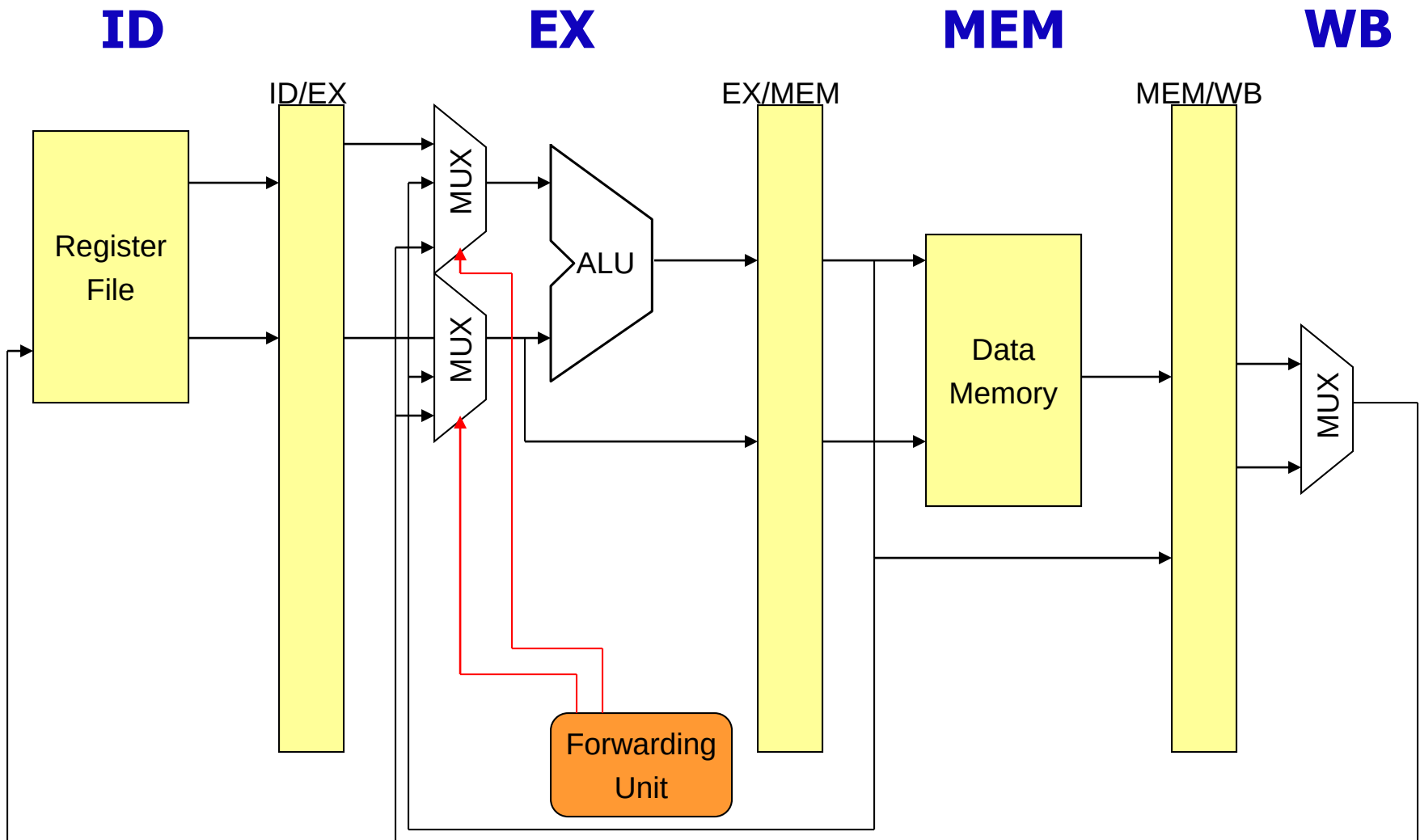
Forwarding (from EX/MEM)



Forwarding (from MEM/WB)



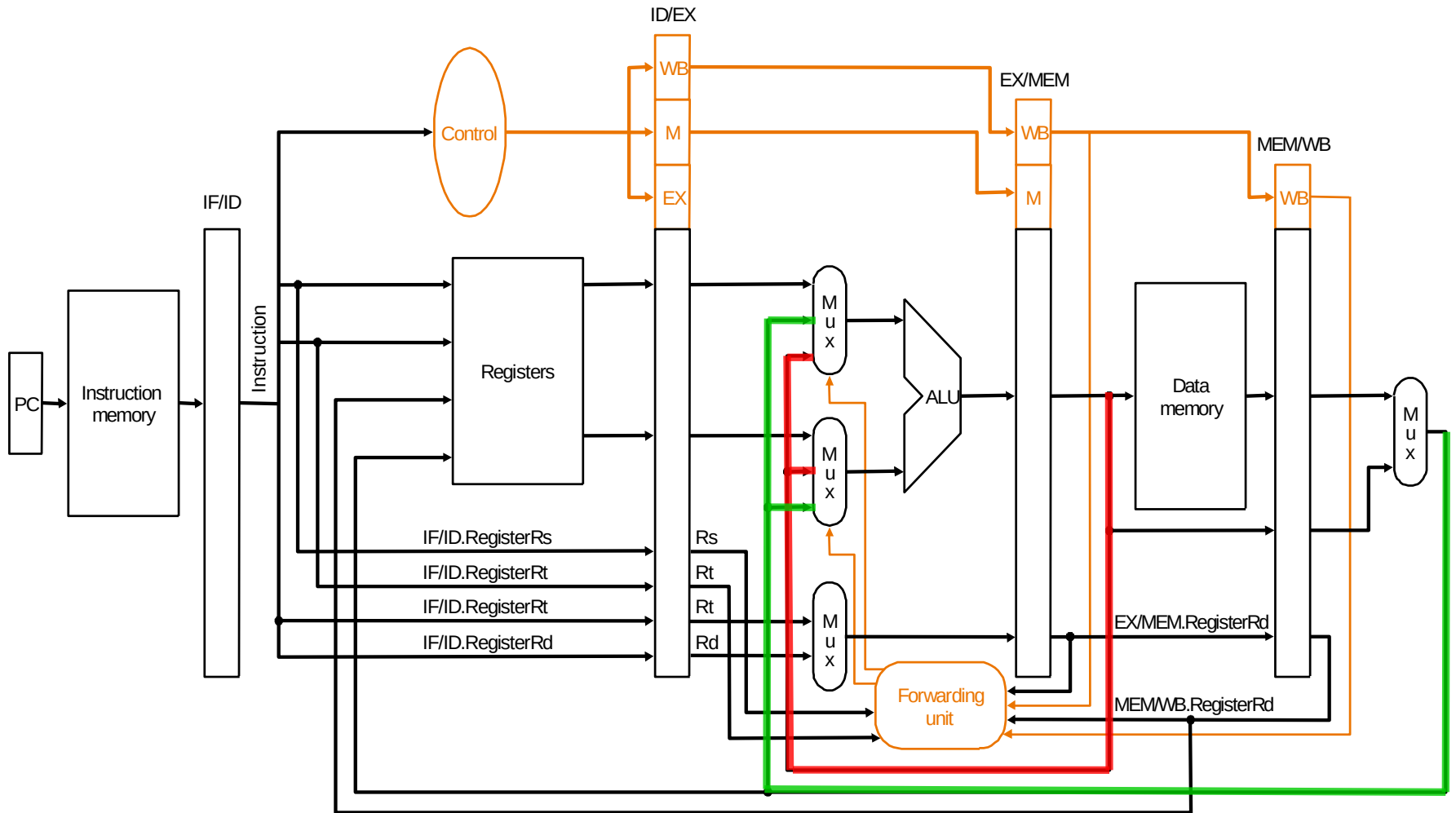
Forwarding (operand selection)



WB

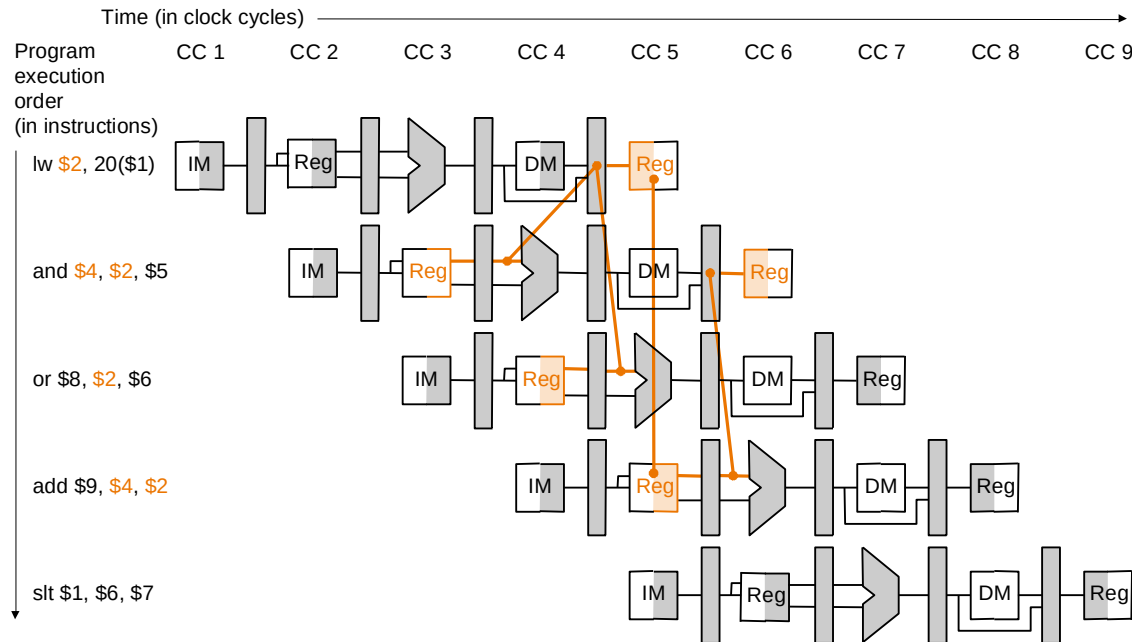


Forwarding



Can't always forward

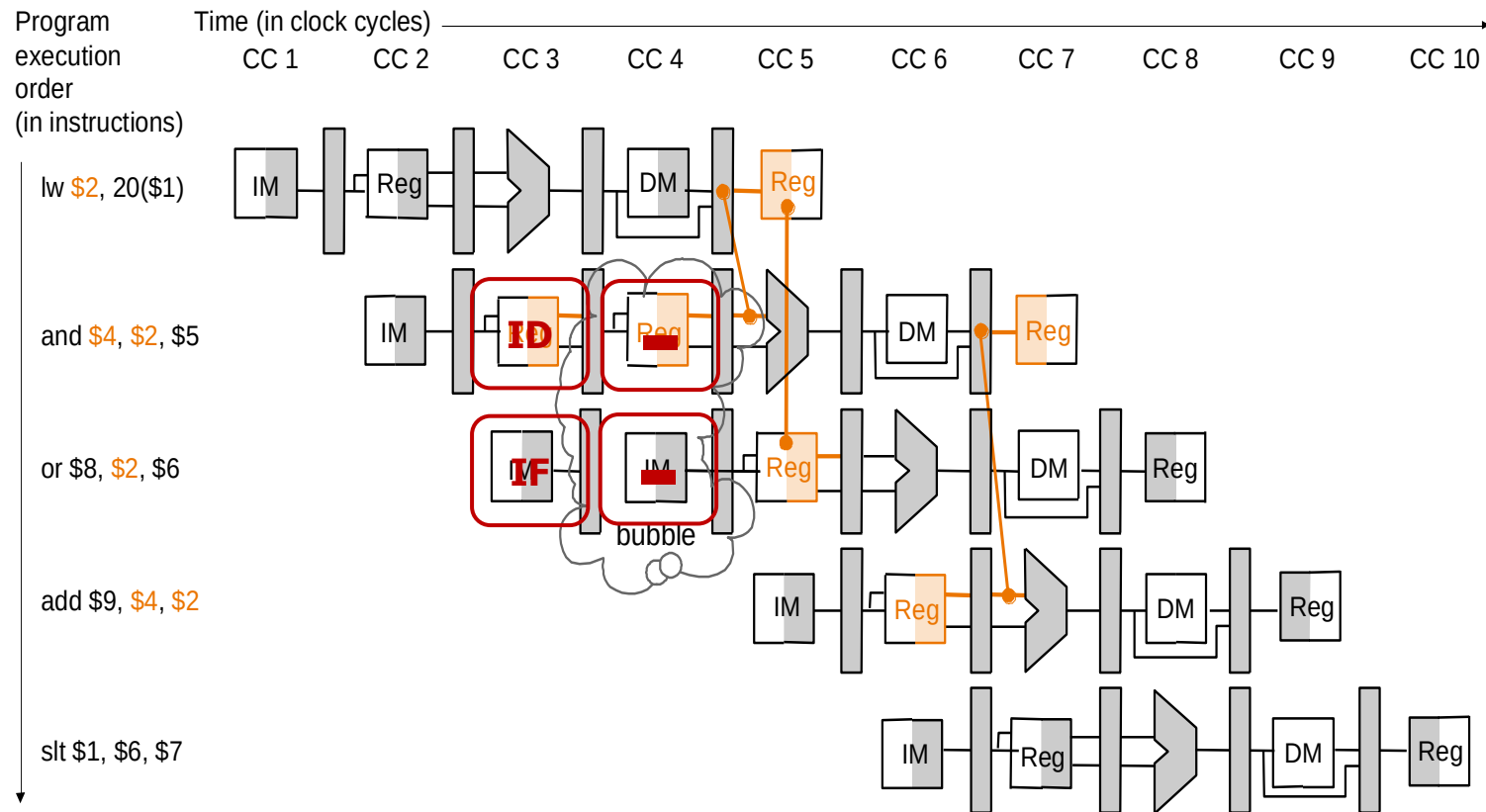
- lw (load word) can still cause a hazard
 - An instruction tries to read a register following a load instruction that writes to the same register



- Thus, we need a hazard detection unit to “stall” the pipeline after the load instruction

Stalling

- We can stall the pipeline by keeping an instruction in the same stage



Data Hazard - Load-Use Case

at cc2

IF

ID

EX

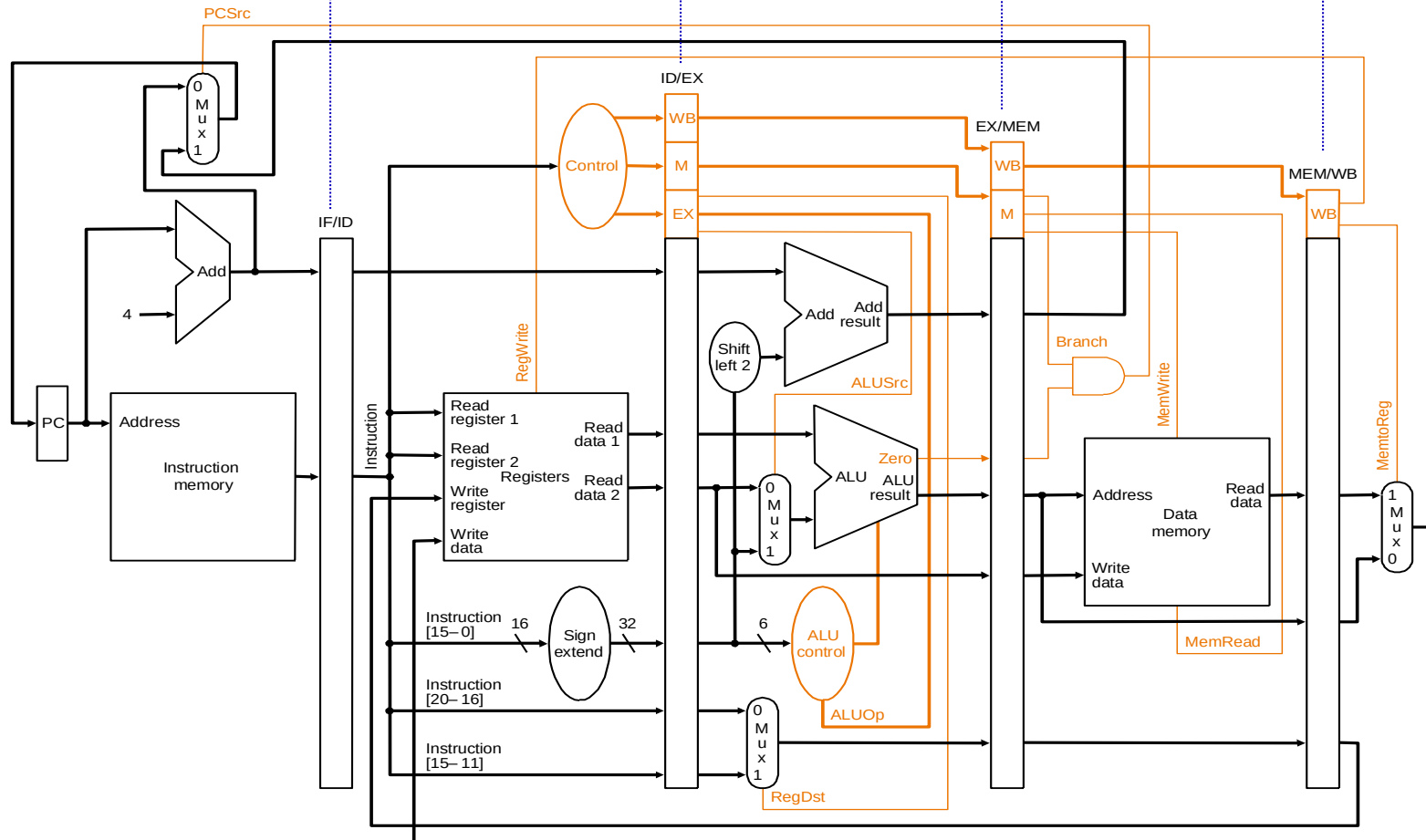
MEM

WB

or \$8, \$2, \$6

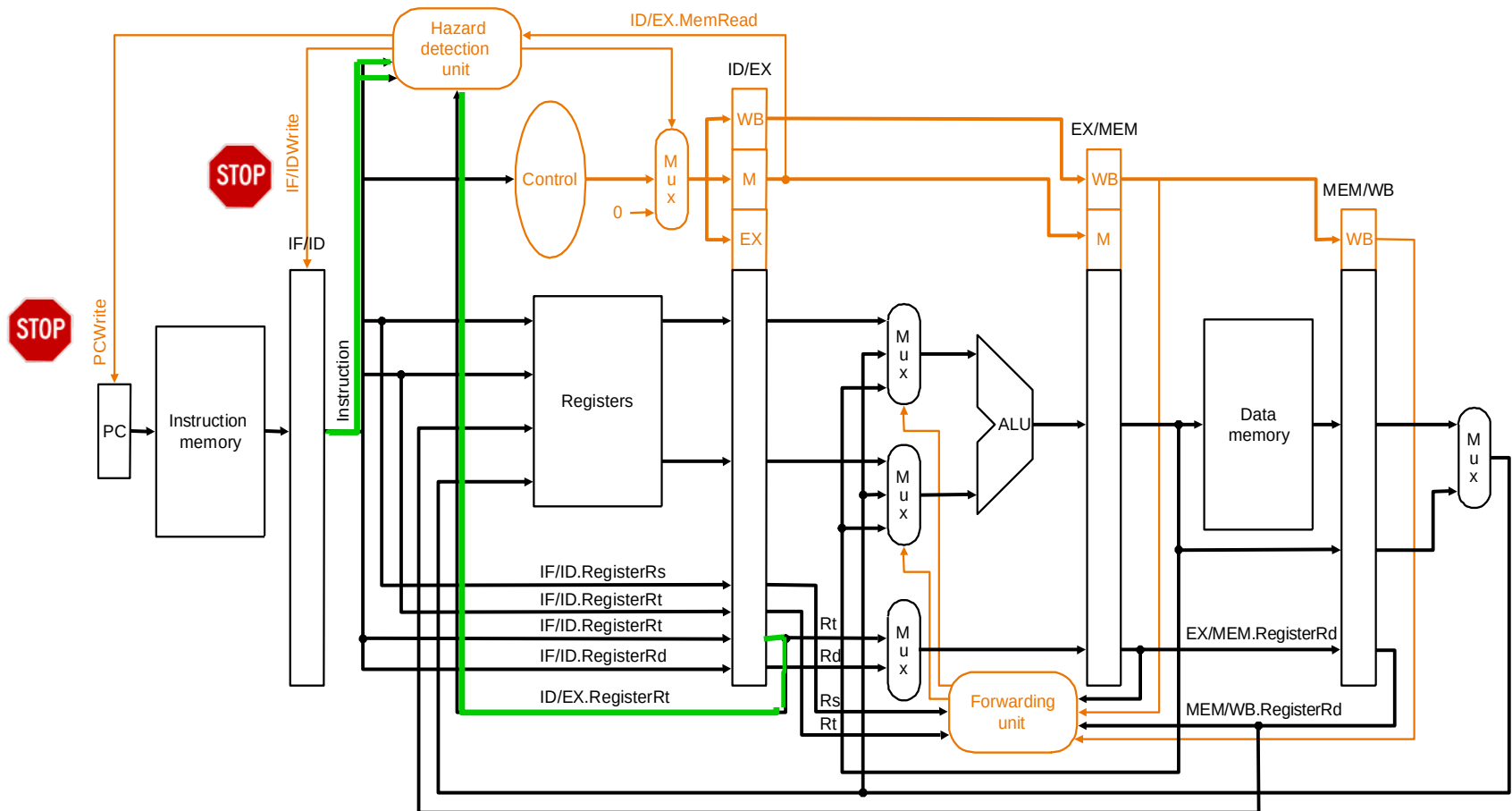
and \$4, \$2, \$5

lw \$2, 20(\$1)



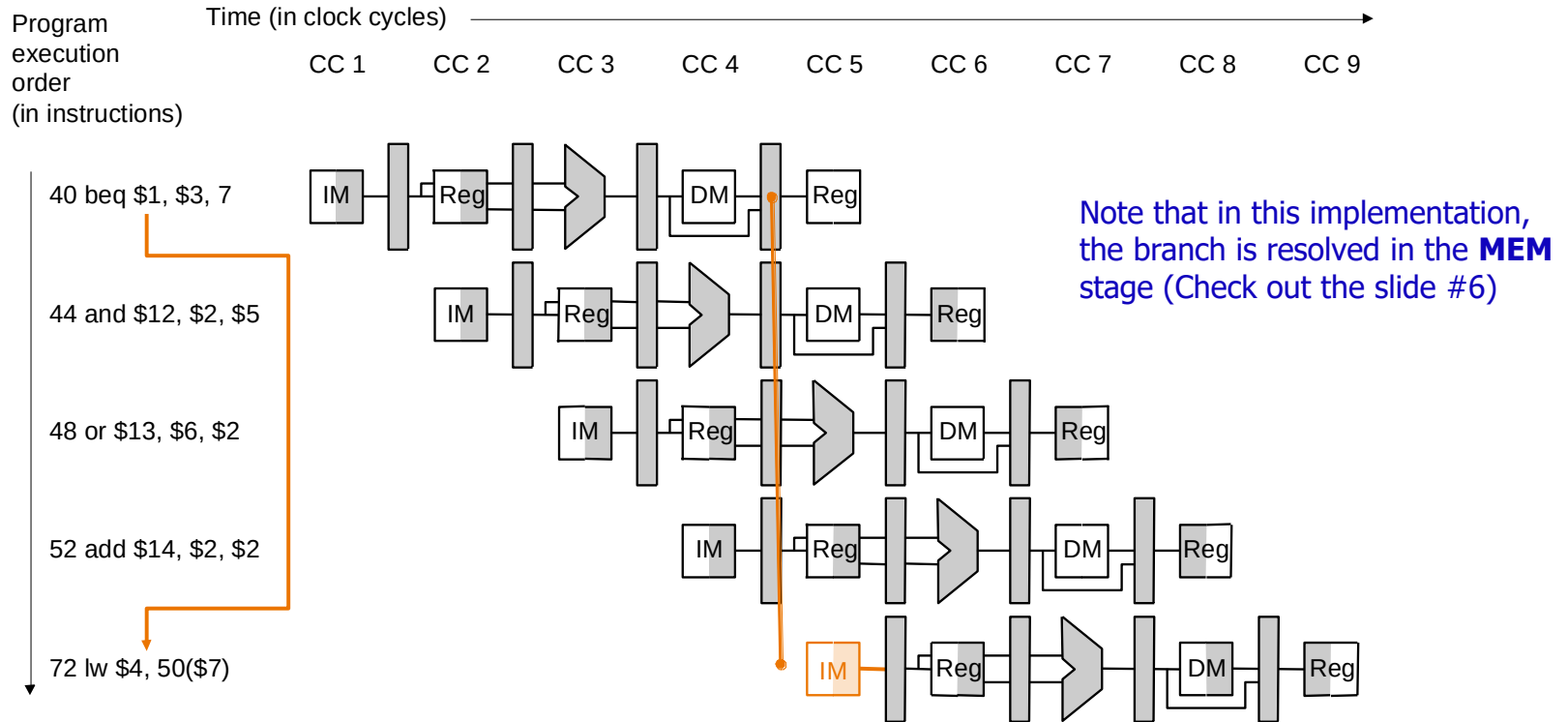
Hazard Detection Unit

- Stall the pipeline if both ID/EX is a load and ($rt=IF/ID.rs$ or $rt=IF/ID.rt$)
 - Stall by letting an instruction (that won't write anything) go forward



Control Hazards - Branch

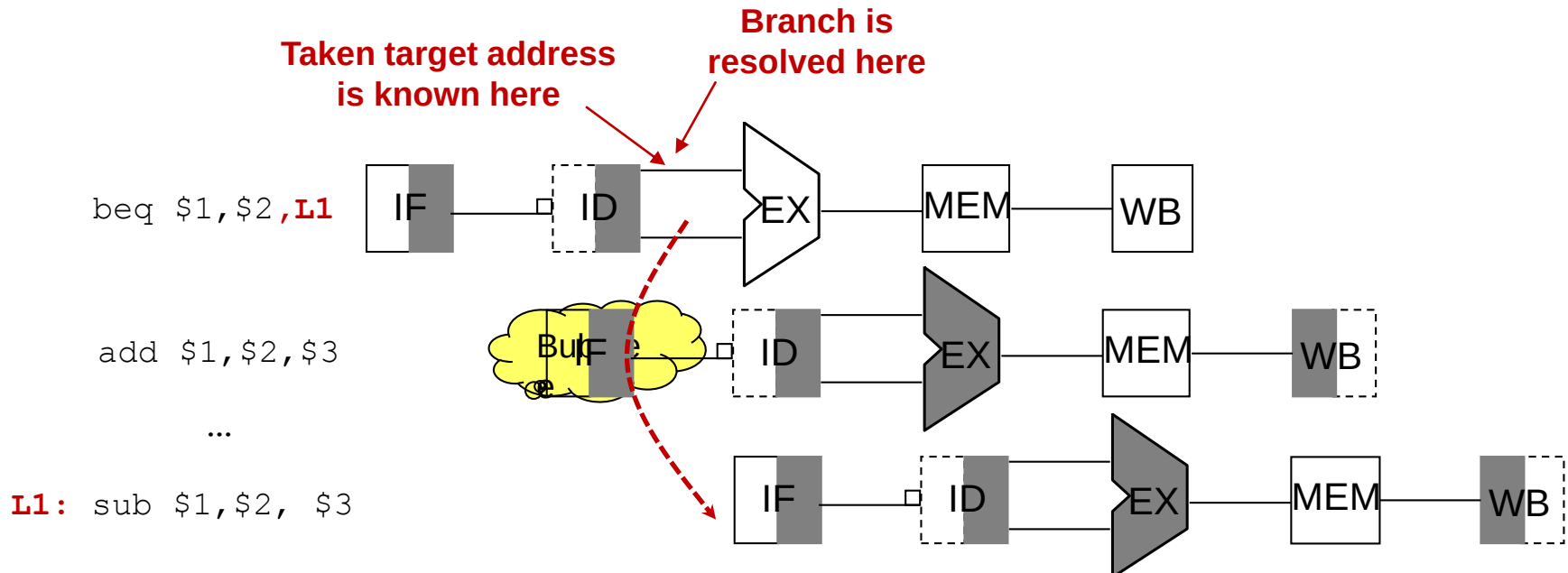
- When the branch condition is resolved, other instructions are in the pipeline



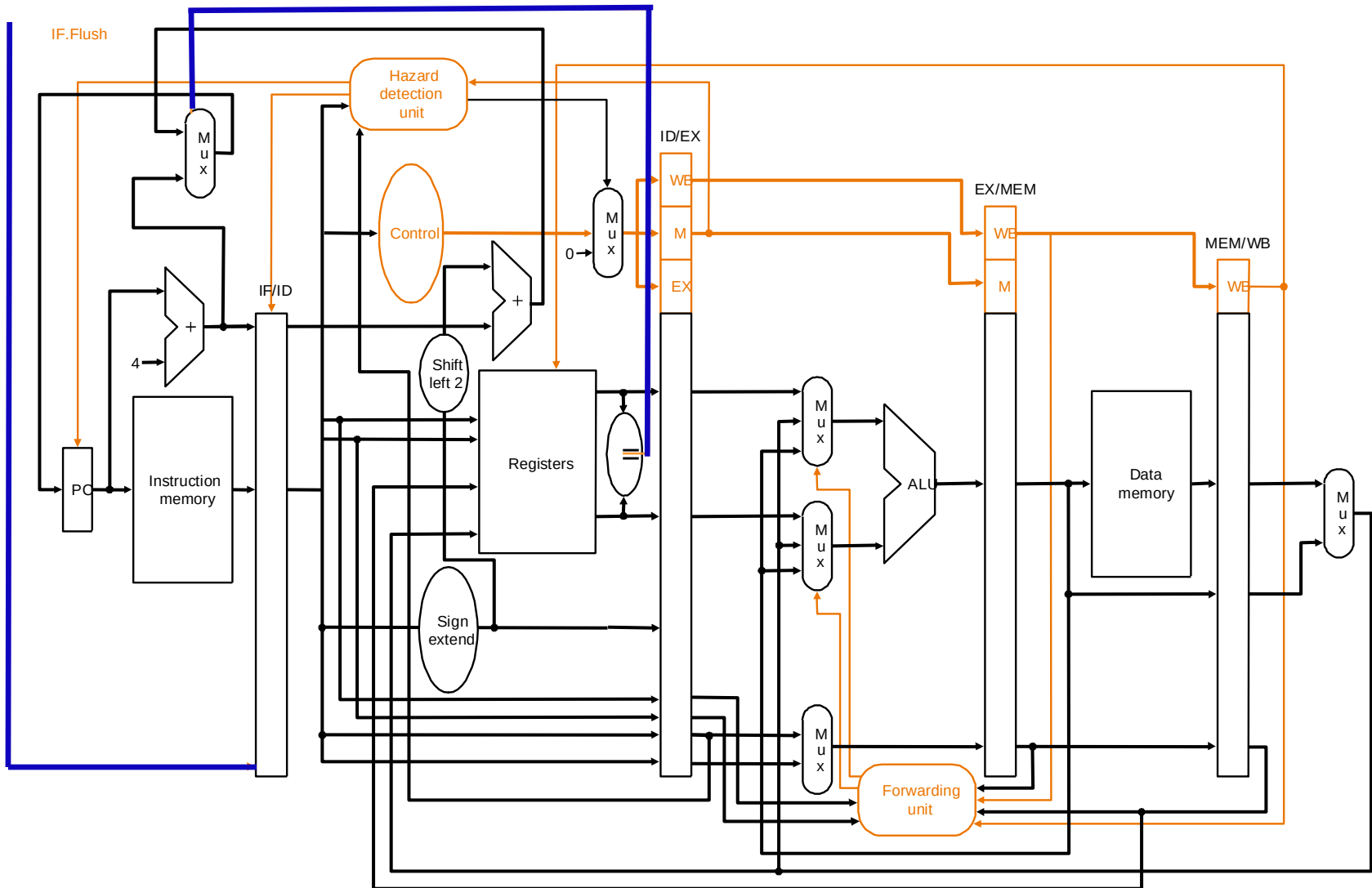
- It works like “not taken” prediction
- If branch turns out to be taken, flush instructions

Alleviate Branch Hazards

- Reduce penalty to 1 cycle
 - Move the branch compare to the ID stage of pipeline
 - Add an adder to calculate the branch target in ID stage
 - Add the IF.flush signal that zeros the instruction (or squash) in IF/ID pipeline register



Flushing Instructions



Control Hazard Handling Logic

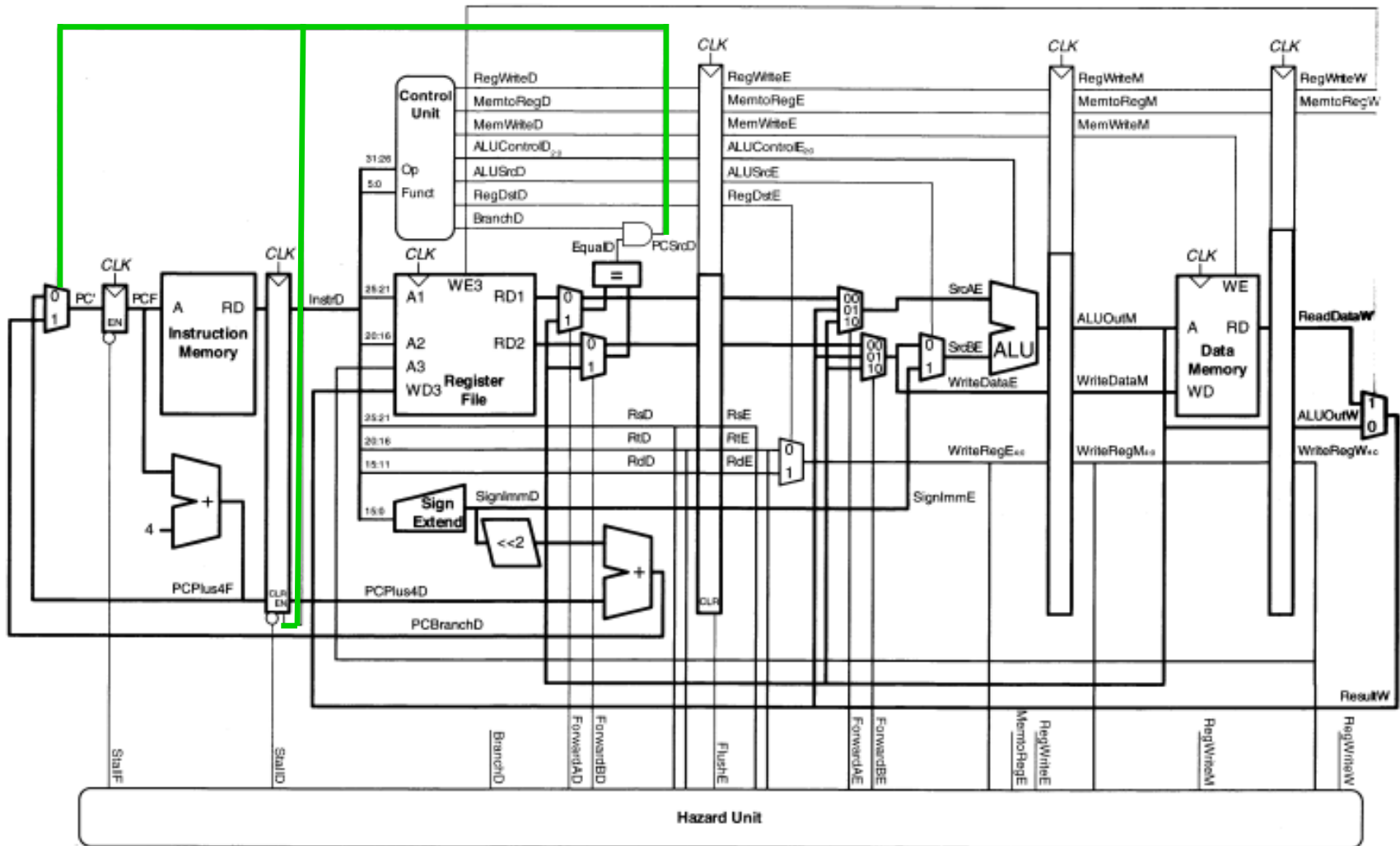
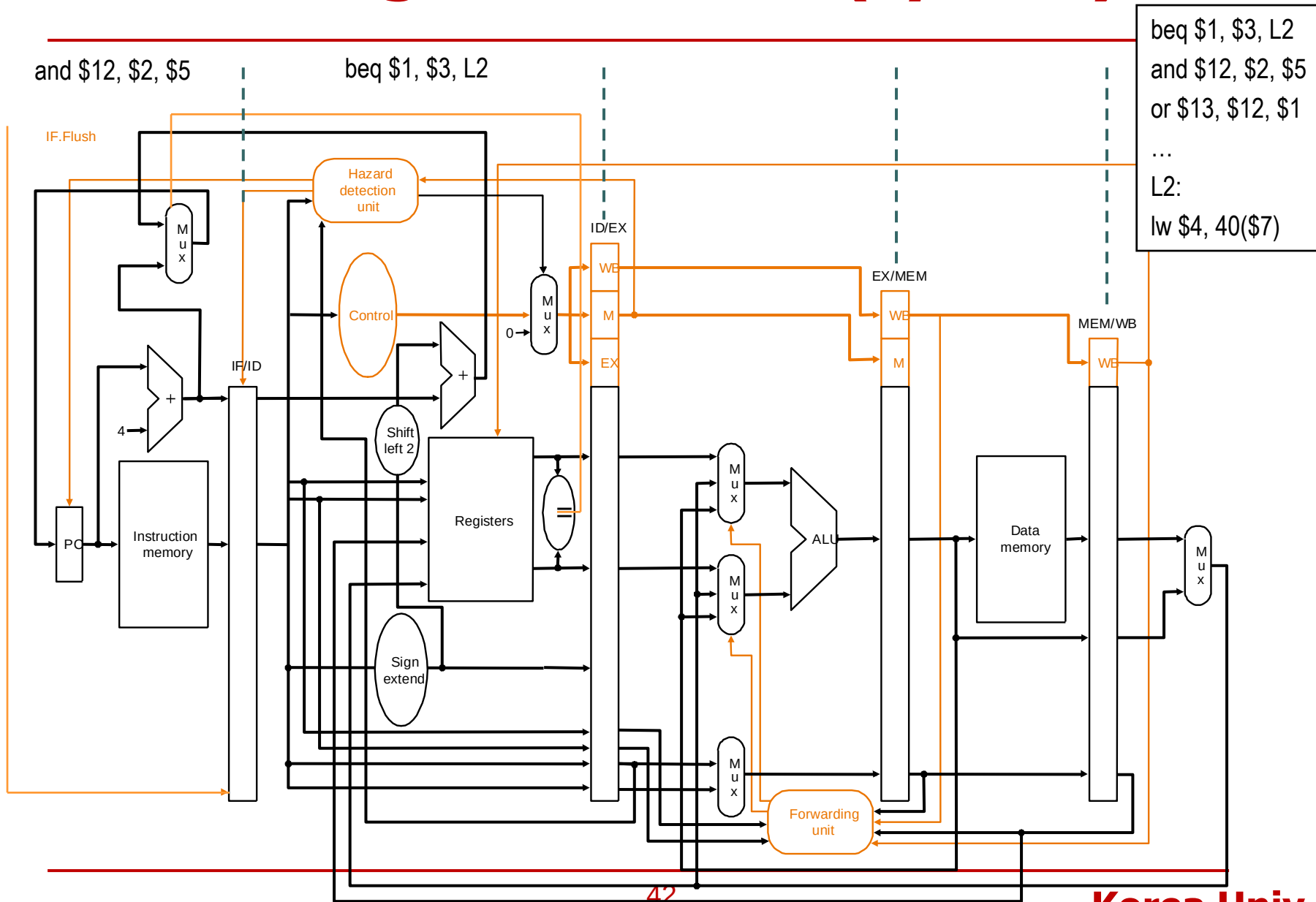
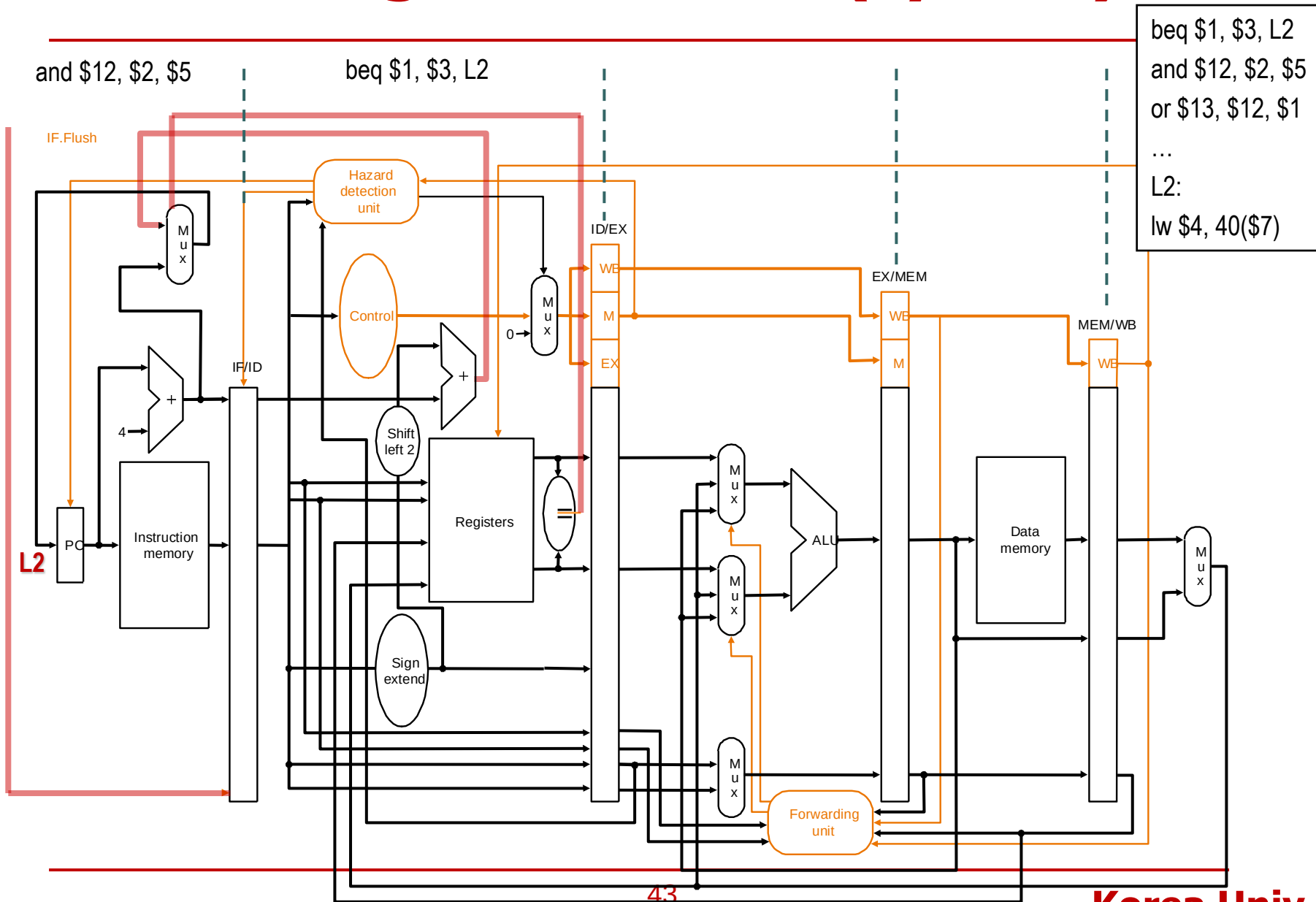


Figure 7.58 Pipelined processor with full hazard handling

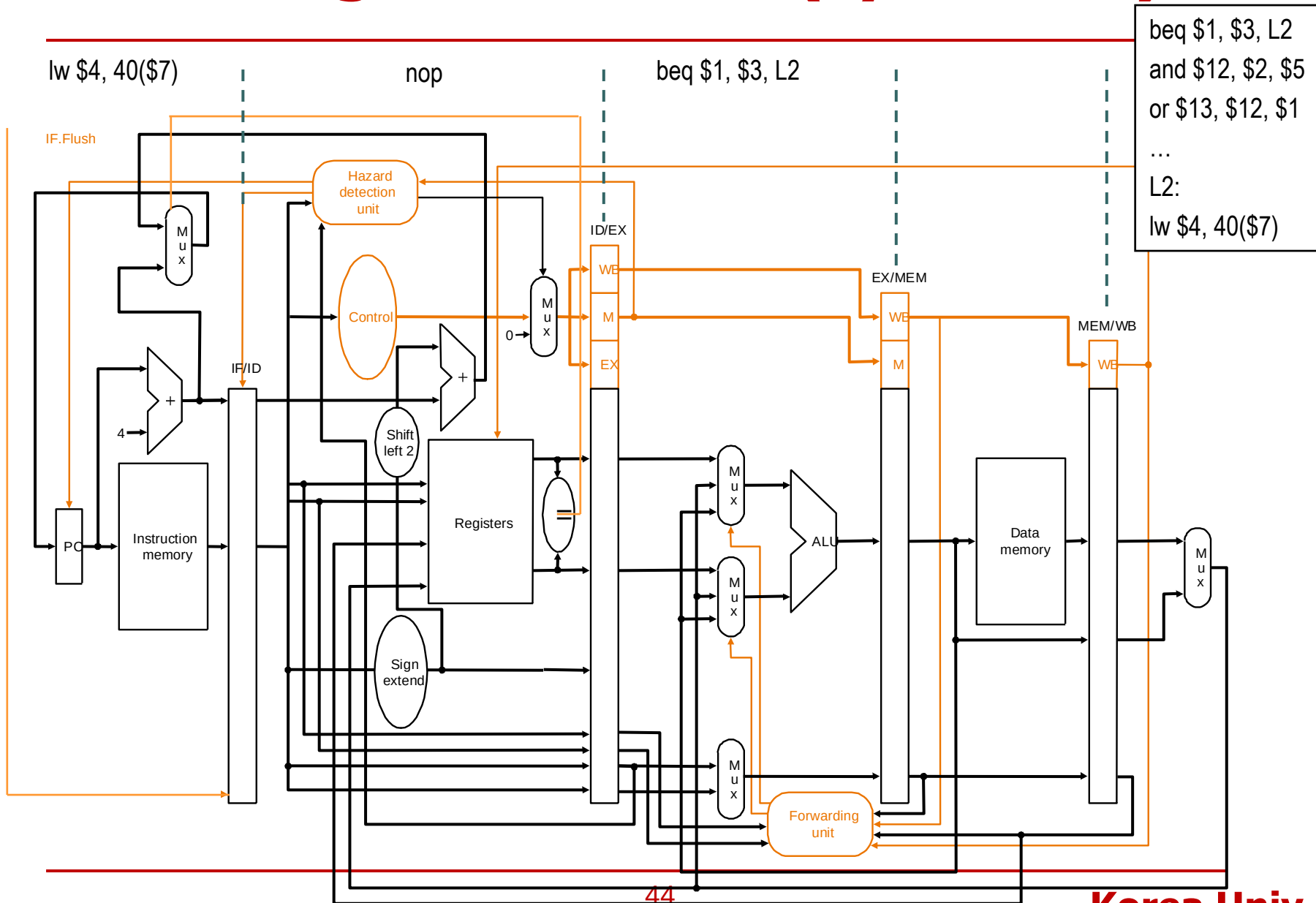
Flushing Instructions (cycle N)



Flushing Instructions (cycle N)



Flushing Instructions (cycle N+1)



Improving Performance

- Try and avoid stalls using hardware/software techniques
 - Software technique
 - Reorder instructions
 - Utilize the delay slot of branch
 - Hardware technique
 - Implement the delayed branch

Performance of Pipelined CPU

- Assuming that the CPU executes 100 billion instructions to run your program, what is the execution time of the program on a pipelined MIPS?

$$\text{CPU Time} = \# \text{ insts} \times \text{CPI} \times \text{clock cycle time (T)} = \# \text{ insts} \times \text{CPI} / f$$

CPI Example

- Ideally $CPI = 1$. But, need to handle stallings (by loads and branches)
- SPECINT2000 benchmark:
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- Suppose
 - 40% of loads are used by next instruction
 - 25% of branches are mispredicted
- What is the average CPI?

CPI Example

- SPECINT2000 benchmark:
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- If there is no stall in the pipelined MIPS, how would you calculate CPI?
 - Average CPI = $(0.25)(1 \text{ CPI}) + (0.10)(1 \text{ CPI}) + (0.11)(1 \text{ CPI}) + (0.02)(1 \text{ CPI}) + (0.52)(1 \text{ CPI}) = 1$
- Suppose
 - 40% of loads are used by next instruction
 - 25% of branches are mispredicted
 - All jumps flush next instruction
- What is the average CPI?
 - Load/Branch CPI = 1 when no stalling, 2 when stalling. Thus
 - $\text{CPI}_{\text{lw}} = 1(0.6) + 2(0.4) = 1.4$
 - $\text{CPI}_{\text{beq}} = 1(0.75) + 2(0.25) = 1.25$
 - $\text{CPI}_{\text{jump}} = 2(1) = 2$
- Average CPI = $(0.25)(1.4) + (0.1)(1) + (0.11)(1.25) + (0.02)(2) + (0.52)(1) = 1.15$

Critical Path

- Critical path of the pipelined MIPS:

$$T_c = \max \left\{ \begin{array}{l} t_{pcq} + t_{mem} + t_{setup}, \\ t_{pcq} + t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}, \\ t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup}, \\ t_{pcq} + t_{memread} + t_{setup}, \\ t_{pcq} + t_{mux} + t_{RFsetup} \end{array} \right.$$

// IF stage
 // ID stage
 // EX stage
 // MEM stage
 // WB stage

- If the write to the register file occurs in the first half of the clock, and read occurs in the 2nd half of the clock, then?
 - Our textbook follows this**
- If RF writes at the rising-edge of the clock, then?
 - Let's stick to this for our project**

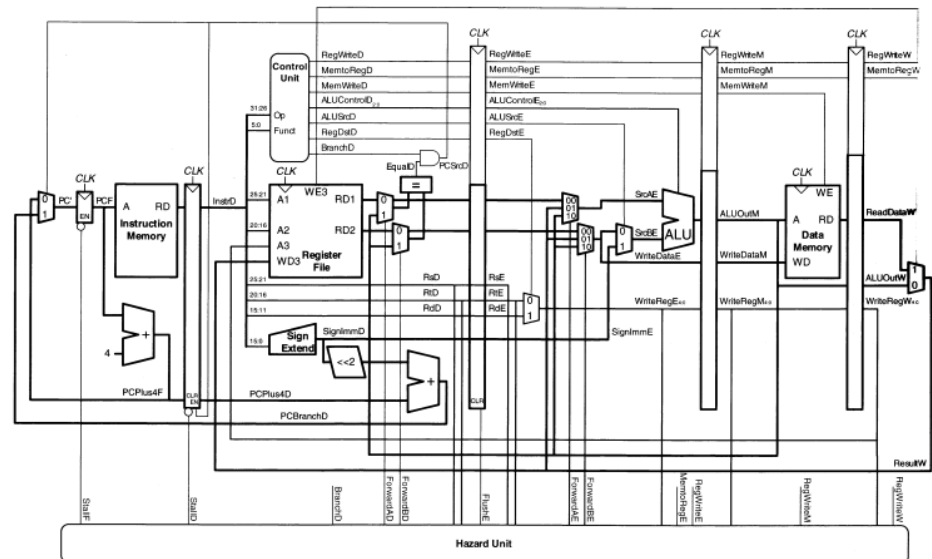


Figure 7.58 Pipelined processor with full hazard handling

Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20
Equality comparator	t_{eq}	40
AND gate	t_{AND}	15
Memory write	$T_{memwrite}$	220

$$T_c = \max \{$$

$$t_{pcq} + t_{mem} + t_{setup},$$

$$t_{pcq} + t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup},$$

$$t_{pcq} + t_{mux} + t_{mux} + t_{ALU} + t_{setup},$$

$$t_{pcq} + t_{memread} + t_{setup},$$

$$t_{pcq} + t_{mux} + t_{RFsetup}$$

$$\}$$

$$\text{IF: } 30 + 250 + 20 = 300 \text{ ps}$$

$$\text{ID: } 30 + 150 + 25 + 40 + 15 + 25 + 20 = 305 \text{ ps}$$

$$\text{EX: } 30 + 25 + 25 + 200 + 20 = 300 \text{ ps}$$

$$\text{MA: } 30 + 250 + 20 = 300 \text{ ps}$$

$$\text{WB: } 30 + 25 + 20 = 75 \text{ ps}$$

$$T_c = (t_{pcq} + t_{RFread} + t_{mux} + t_{eq} + t_{AND} + t_{mux} + t_{setup}) \longrightarrow$$

$$= 305 \text{ ps}$$

$$f_c = 1/T_c$$

$$f_c = 1/0.305 \text{ ns}$$

$$= 3.279 \text{ GHz}$$

Example

- Assuming that the CPU executes 100 billion instructions to run your program, what is the execution time of the program on a pipelined MIPS (CPI = 1.15, $T_c = 305$ ps)?

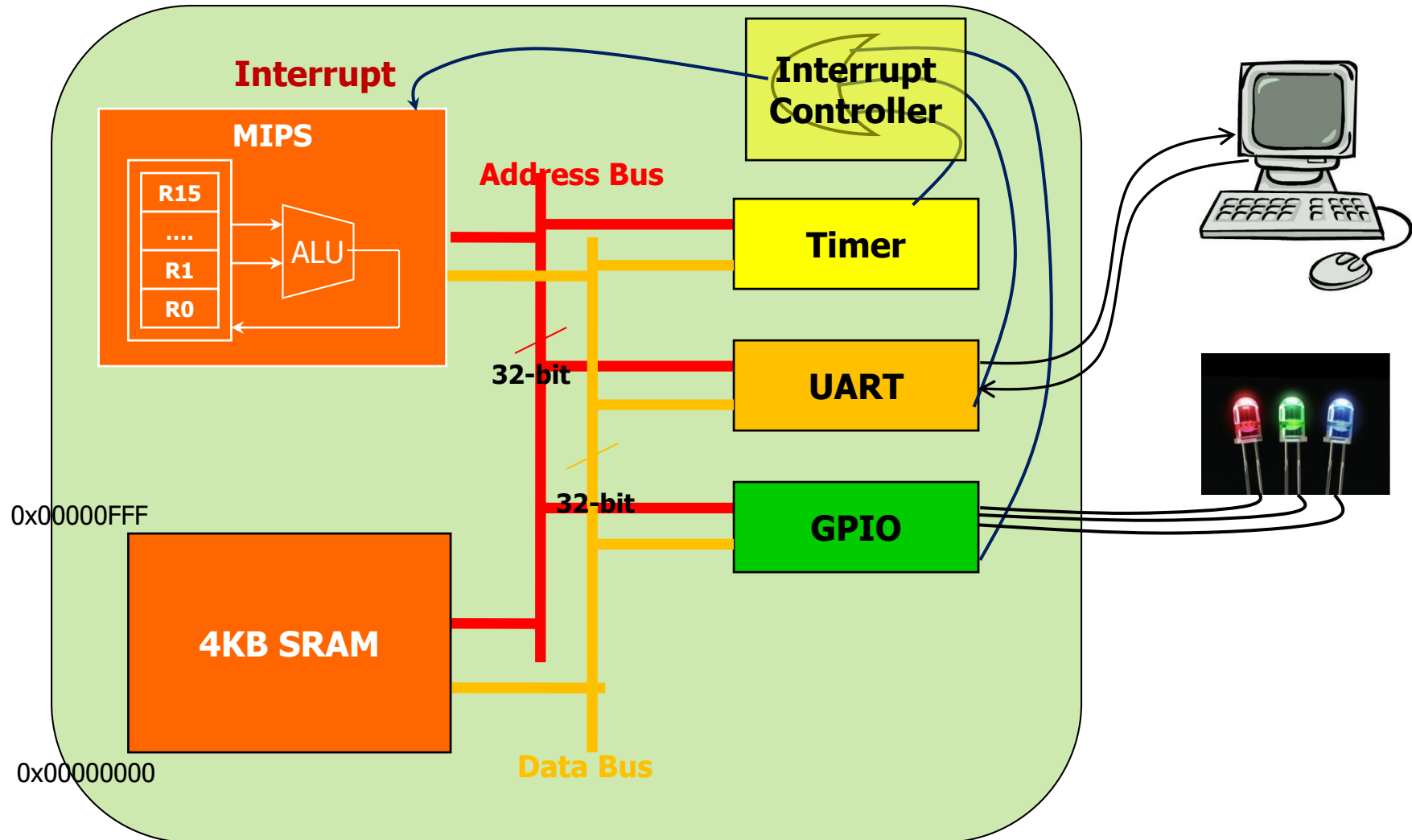
$$\begin{aligned}\text{Execution Time} &= (\# \text{instructions})(\text{cycles/instruction})(\text{seconds/cycle}) \\ &= (100 \times 10^9)(1.15)(0.305 \times 10^{-9} \text{ s}) \\ &= \mathbf{40.25 \text{ seconds}}\end{aligned}$$

Processor	Execution Time (seconds)	Speedup (single-cycle is baseline)
Single-cycle	95	1
Multicycle	133	0.71
Pipelined	40.25	2.36

Exceptions & Interrupts

- CPU has to prepare for all possible situations it could face
 - “Unexpected” events require change in flow of control
 - MIPS CPU jumps to 0x8000_0180 when an exception occurs
- **Exceptions** arise within the CPU
 - Undefined opcode
 - Divide-by-zero
 - Arithmetic overflow in MIPS
 - Some other architectures (such as x86 and ARM) do **not** generate exception on arithmetic overflow. Instead, set bits of the flag register inside CPU
- **Interrupts** are from external I/O devices
 - Keyboard, Mouse, Network card etc
- Many architectures and authors do not distinguish between interrupts and exceptions
 - Often use the term “interrupt” to refer to both types of events

Simplified Hardware System



- UART: Universal Asynchronous Receiver and Transmitter
- GPIO: General Purpose Input/Output

Backup Slides

Superscalar

- Superscalar
 - Start more than one instruction in the same cycle
 - Hardware performs the dynamic scheduling of instructions
 - Hardware tries to find instructions to execute
 - Out of order (OOO) execution is possible
 - Speculative execution
 - Dynamic branch prediction with branch history table
 - Many modern processors (including x86) are superscalars
 - Core i7: 14-stage pipeline with 4 instruction issue
 - DEC Alpha 21264: 9-stage pipeline with 6 instruction issue
- Compiler technology is important

Exception Handling in MIPS and Handler Actions

- Exception handling in MIPS **Hardware** (CPU)
 - CPU saves PC of offending (or interrupted) instruction to the “Exception Program Counter (EPC)” register
 - CPU saves indication of the problem to the “Cause” register
 - Jump to handler at 0x8000 00180
- Exception Handler in **Software**
 - Read cause, and transfer to relevant handler
 - If restartable,
 - Take corrective action
 - Use EPC to return to program
 - Otherwise
 - Terminate program
 - Report error using EPC, cause, ...

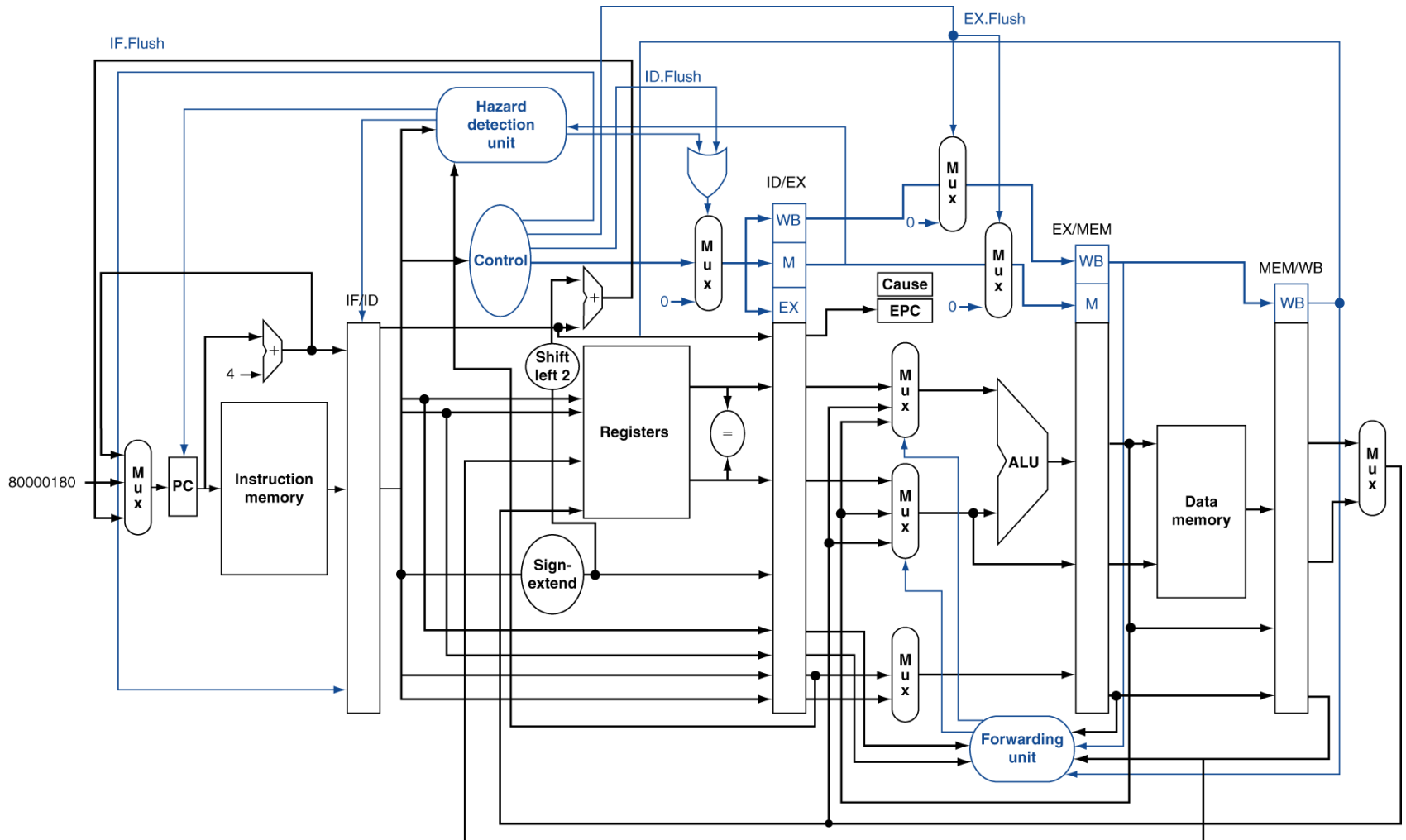
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage

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

- Prevent \$1 from being clobbered
 - Complete previous instructions
 - Flush add and subsequent instructions
 - Set Cause and EPC register values
 - Transfer control to handler
- Similar to mispredicted branch
 - Use much of the same hardware

Pipeline with Exceptions



Exception Example

- Exception on **add** in

40	sub	\$11,	\$2,	\$4
44	and	\$12,	\$2,	\$5
48	or	\$13,	\$2,	\$6
4C	add	\$1,	\$2,	\$1
50	slt	\$15,	\$6,	\$7
54	lw	\$16,	50(\$7)	

...

- Handler

80000180	sw	\$25,	1000(\$0)
80000184	sw	\$26,	1004(\$0)

...

Korea Univ



Exception Example

