



고려대학교
KOREA UNIVERSITY

COM515 Advanced Computer Architecture

Lecture 2. Pipelining

- COMP212 Review -

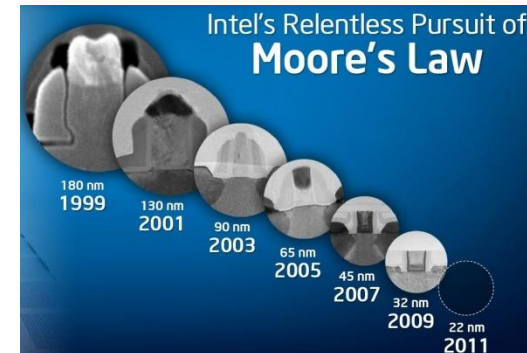
Prof. Taeweon Suh

Computer Science Education

Korea University

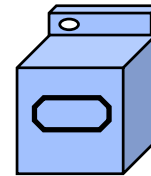
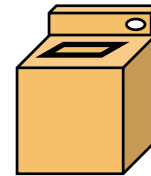
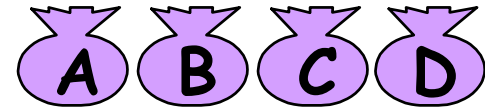
Processor Performance

- Performance of single-cycle processor is limited by the **long** critical path delay
 - The critical path limits the operating clock frequency
- Can we do better?
 - New semiconductor technology will reduce the critical path delay by manufacturing small-sized transistors
 - Core 2 Duo is manufactured with 65nm technology
 - Core i7 is manufactured with 45nm technology
 - Next semiconductor technology is 32nm technology
 - Can we increase the processor performance with a different microarchitecture?
 - **Yes! Pipelining**

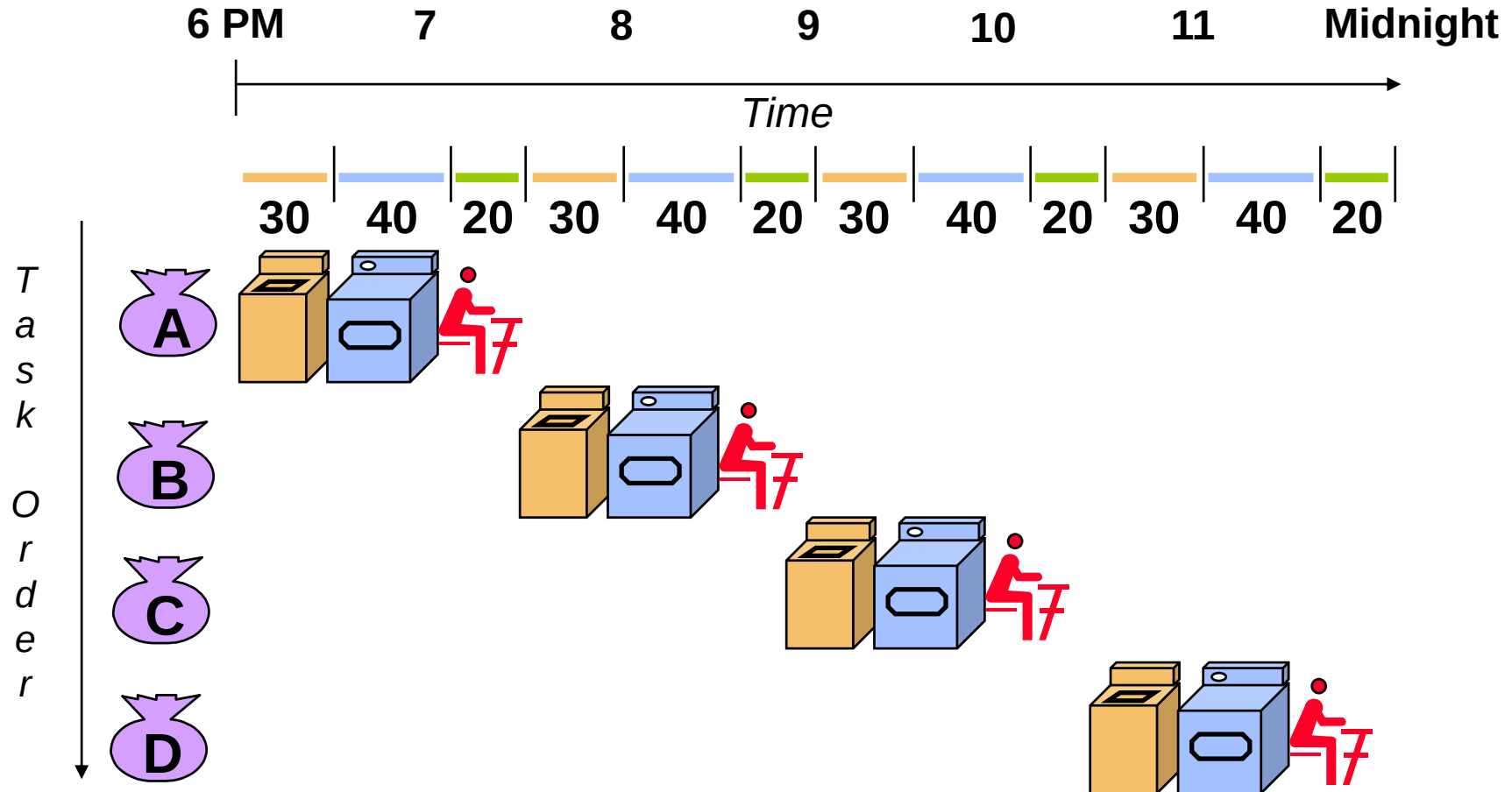


Revisiting Performance

- Laundry Example
 - Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
 - **Washer** takes 30 minutes
 - **Dryer** takes 40 minutes
 - **Folder** takes 20 minutes

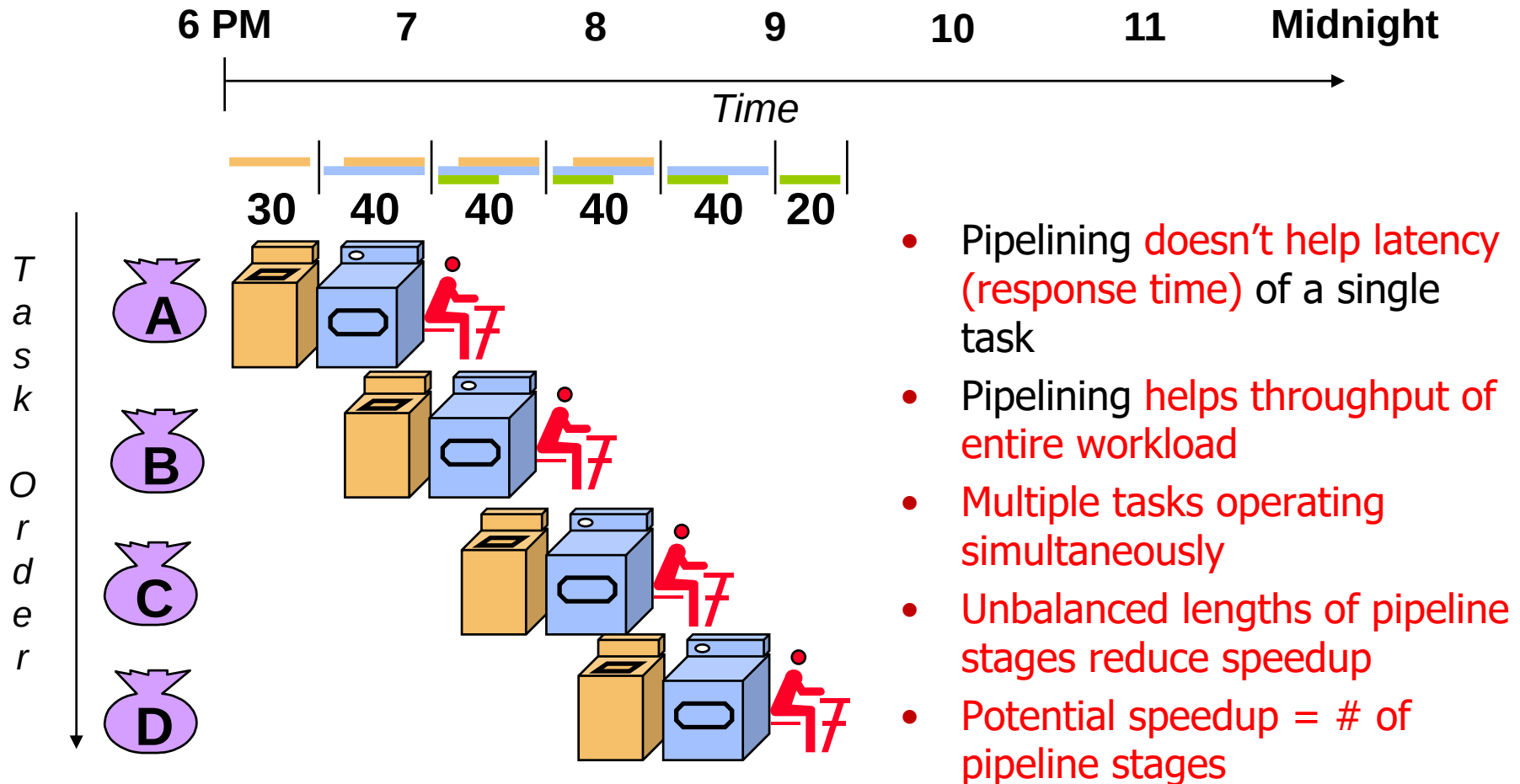


Sequential Laundry



- Response time: 90 mins
- Throughput: 0.67 tasks / hr (= 90mins/task, 6 hours for 4 loads)

Pipelined Laundry



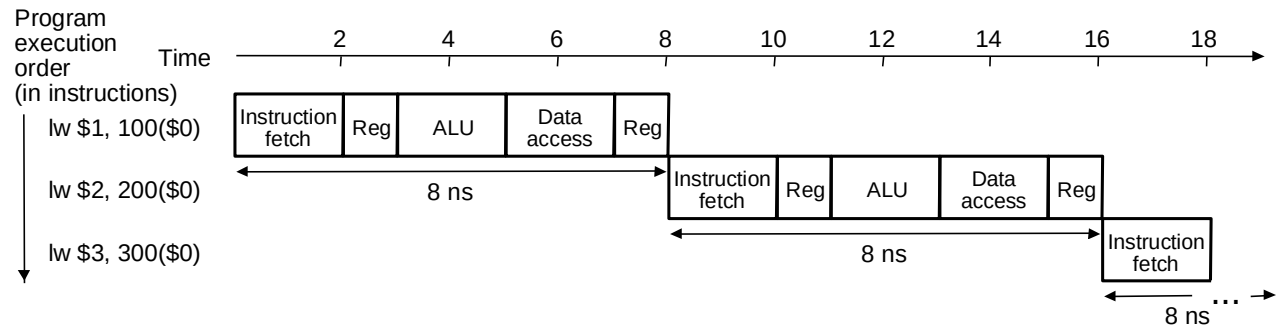
- Response time: 90 mins
- Throughput: 1.14 tasks / hr (= 52.5 mins/task, 3.5 hours for 4 loads)

Pipelining

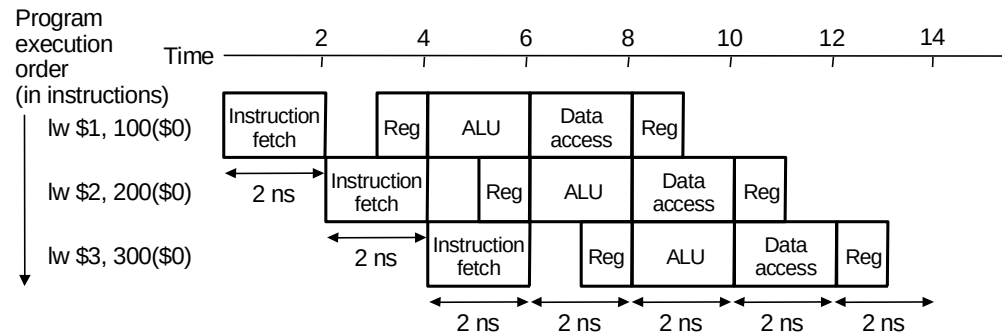
- Improve performance by increasing instruction **throughput**

Instruction Fetch	Register File Access (Read)	ALU Operation	Data Access	Register Access (Write)
2ns	1ns	2ns	2ns	1ns

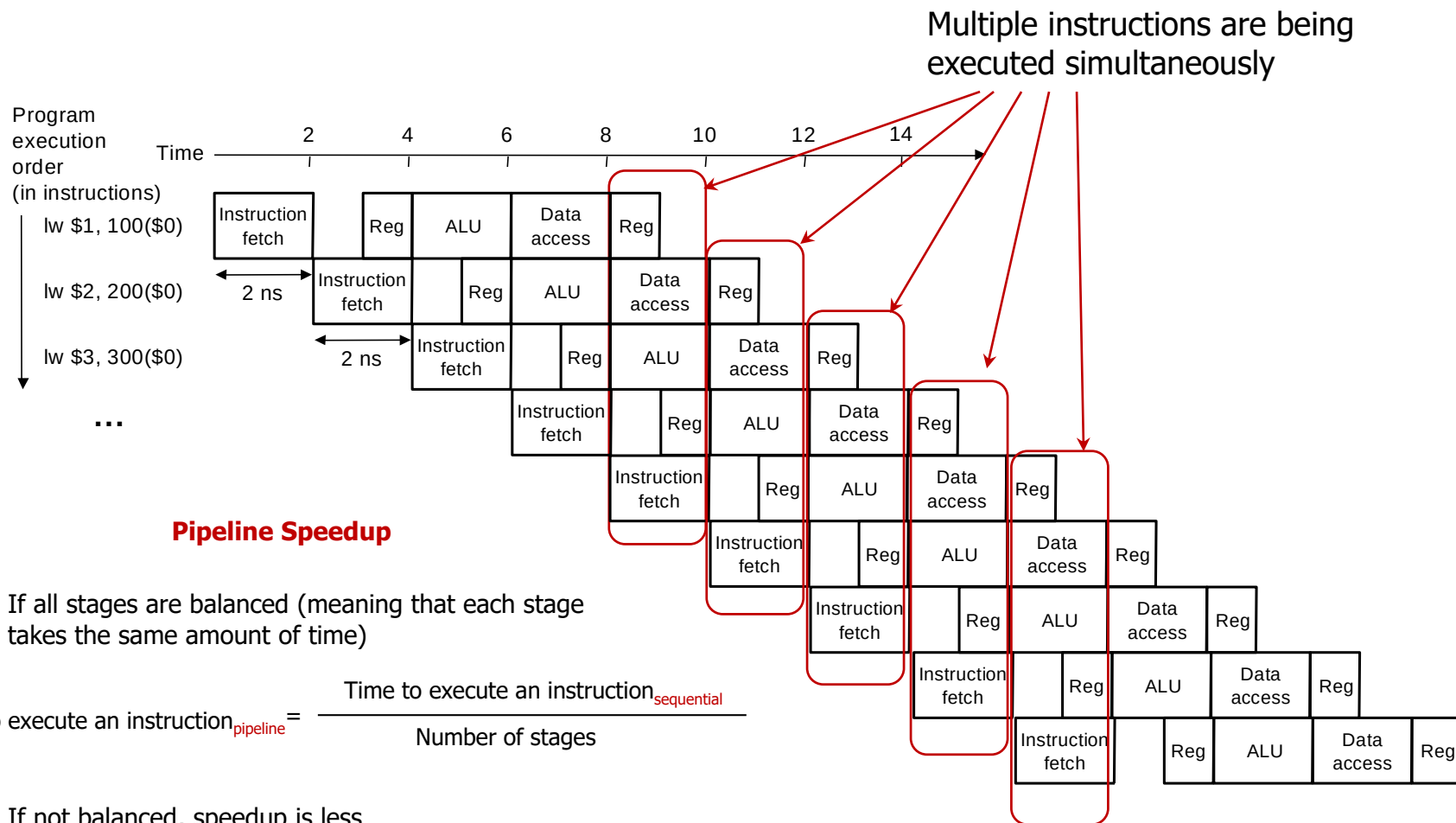
Sequential Execution



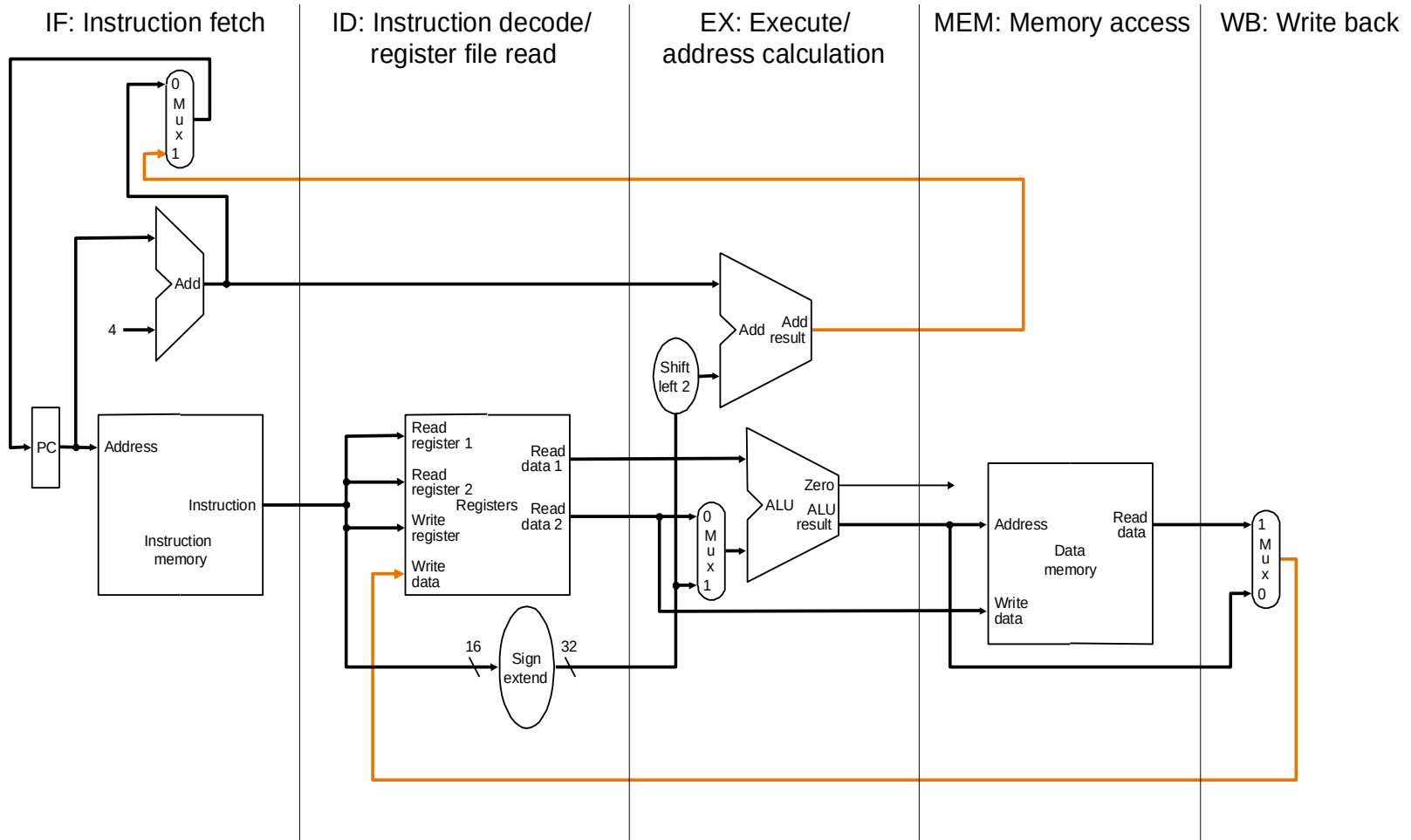
Pipelined Execution



Pipelining (Cont.)

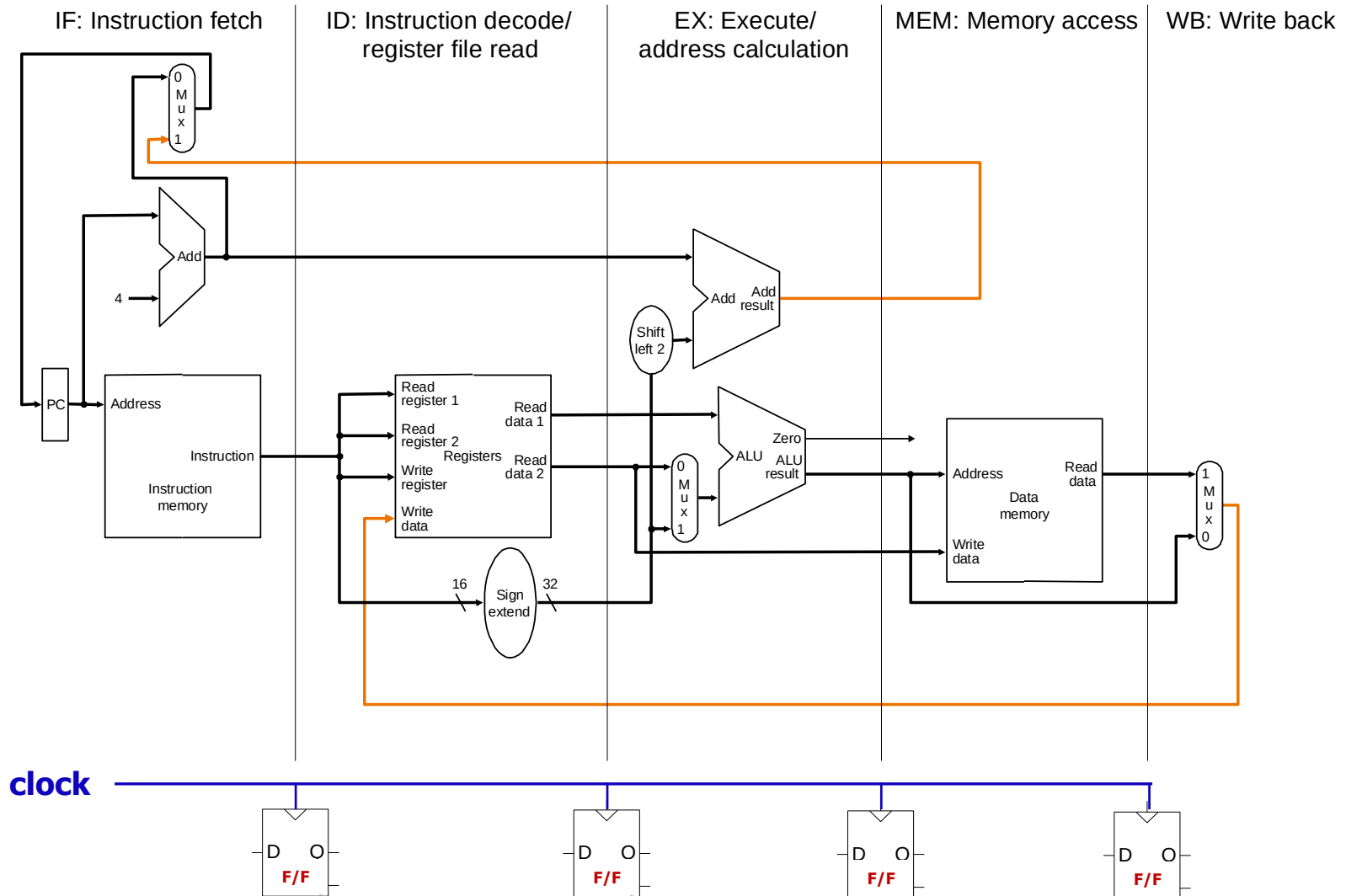


Basic Idea

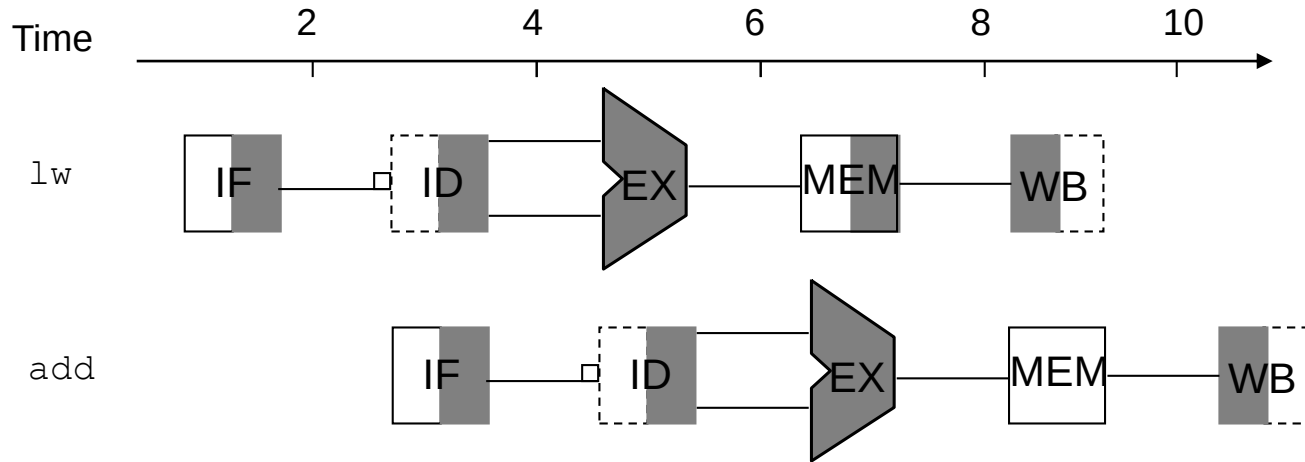


- What do we have to add to actually split the datapath into stages?

Basic Idea

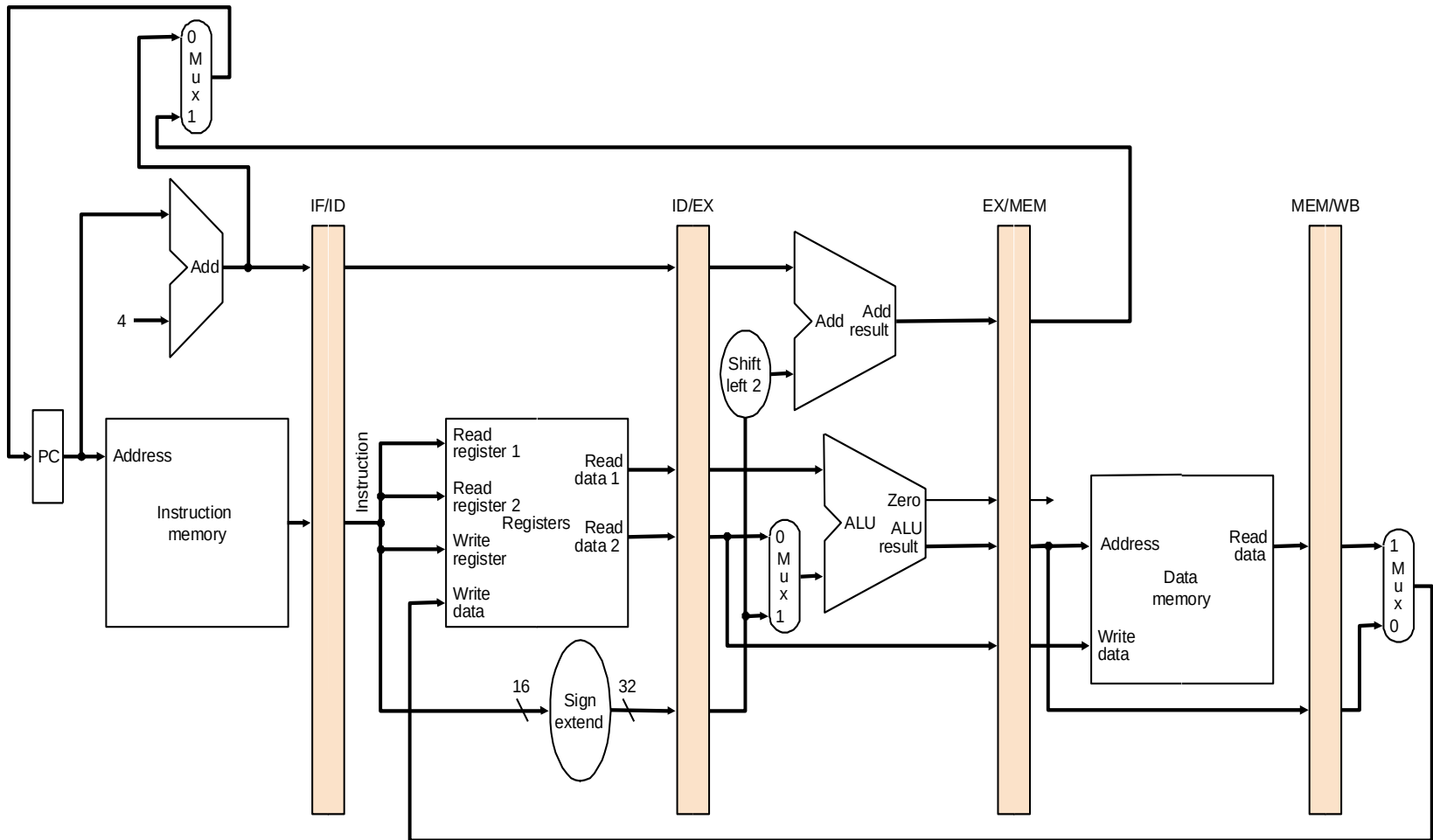


Graphically Representing Pipelines

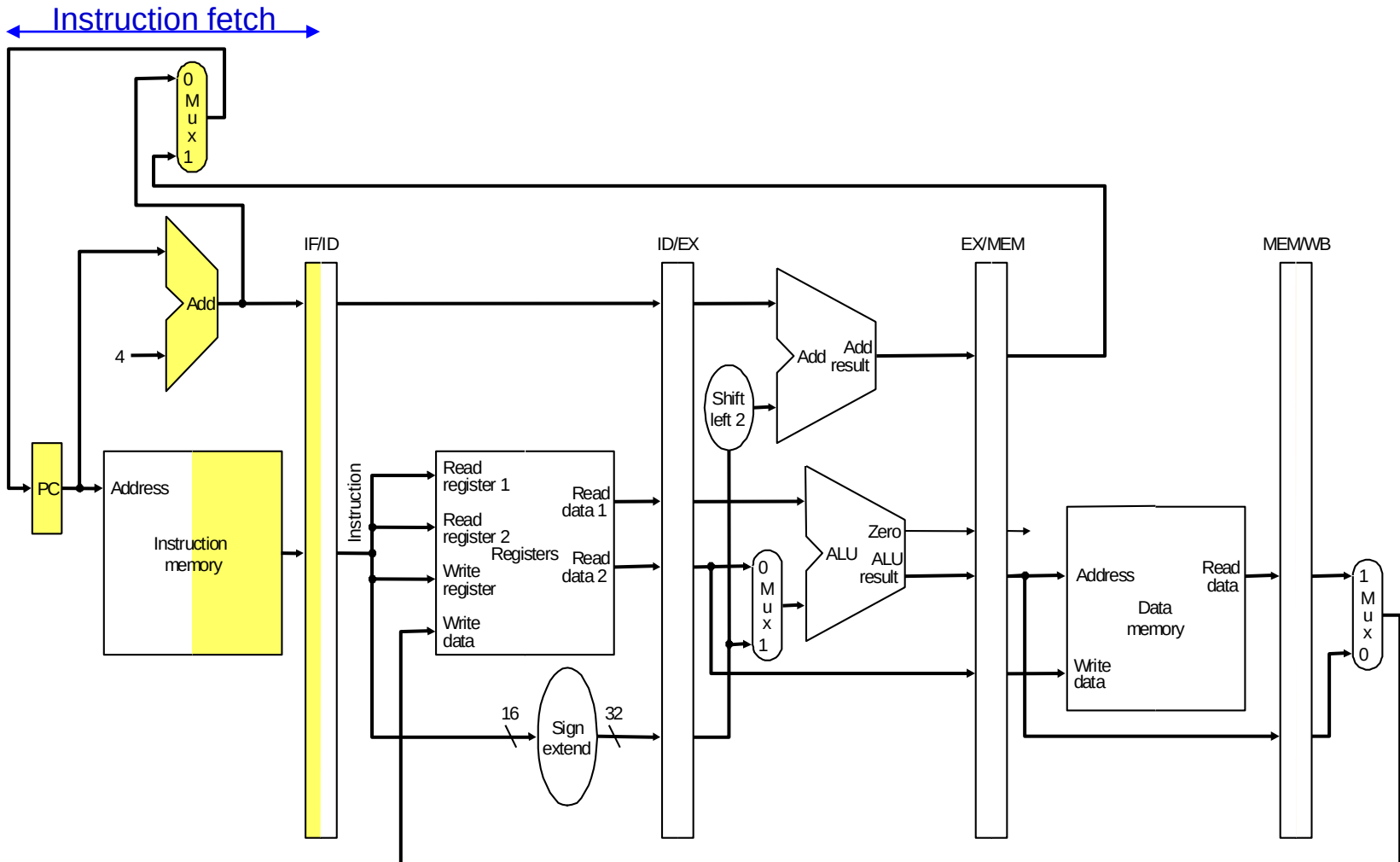


- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

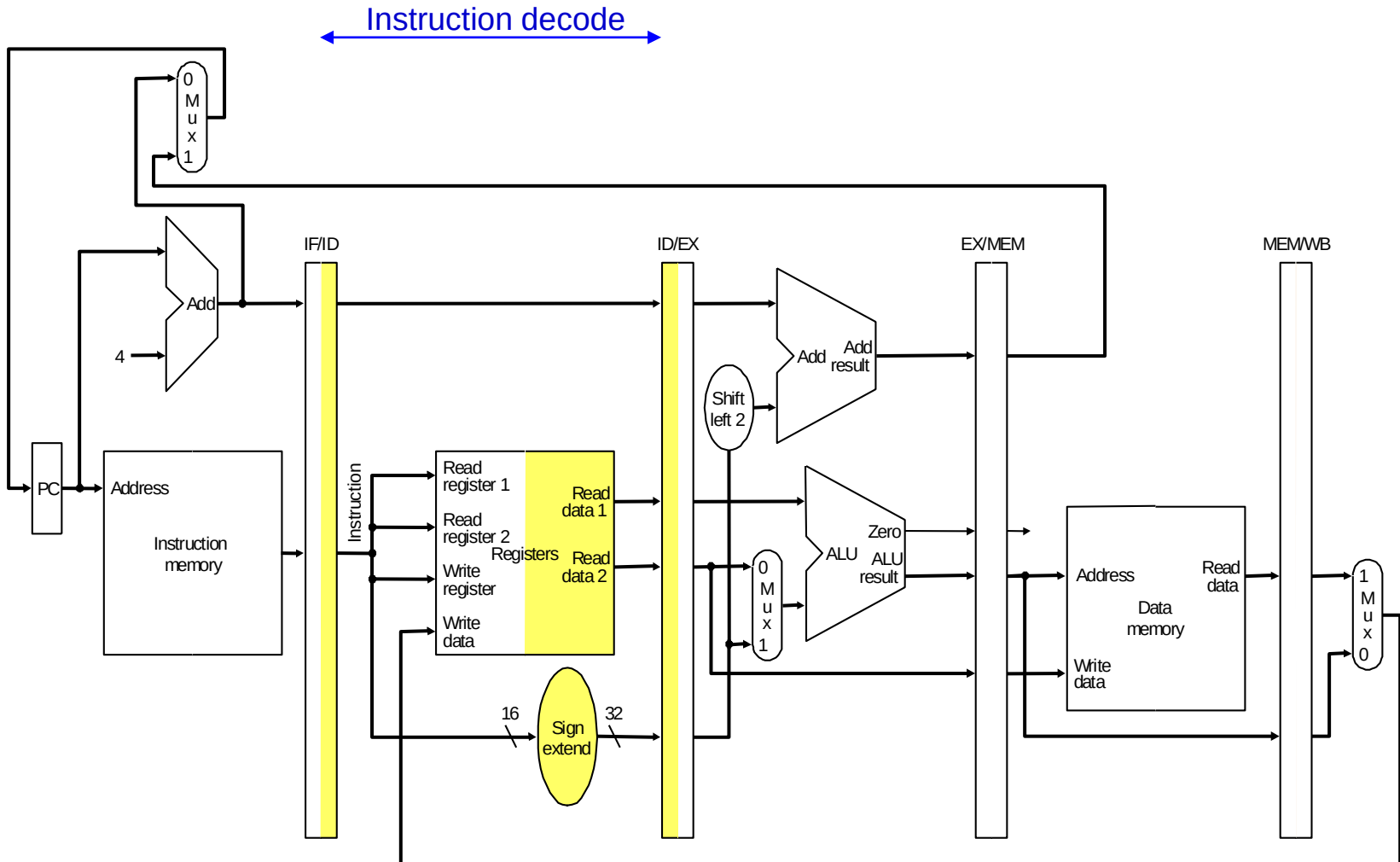
Pipelined Datapath



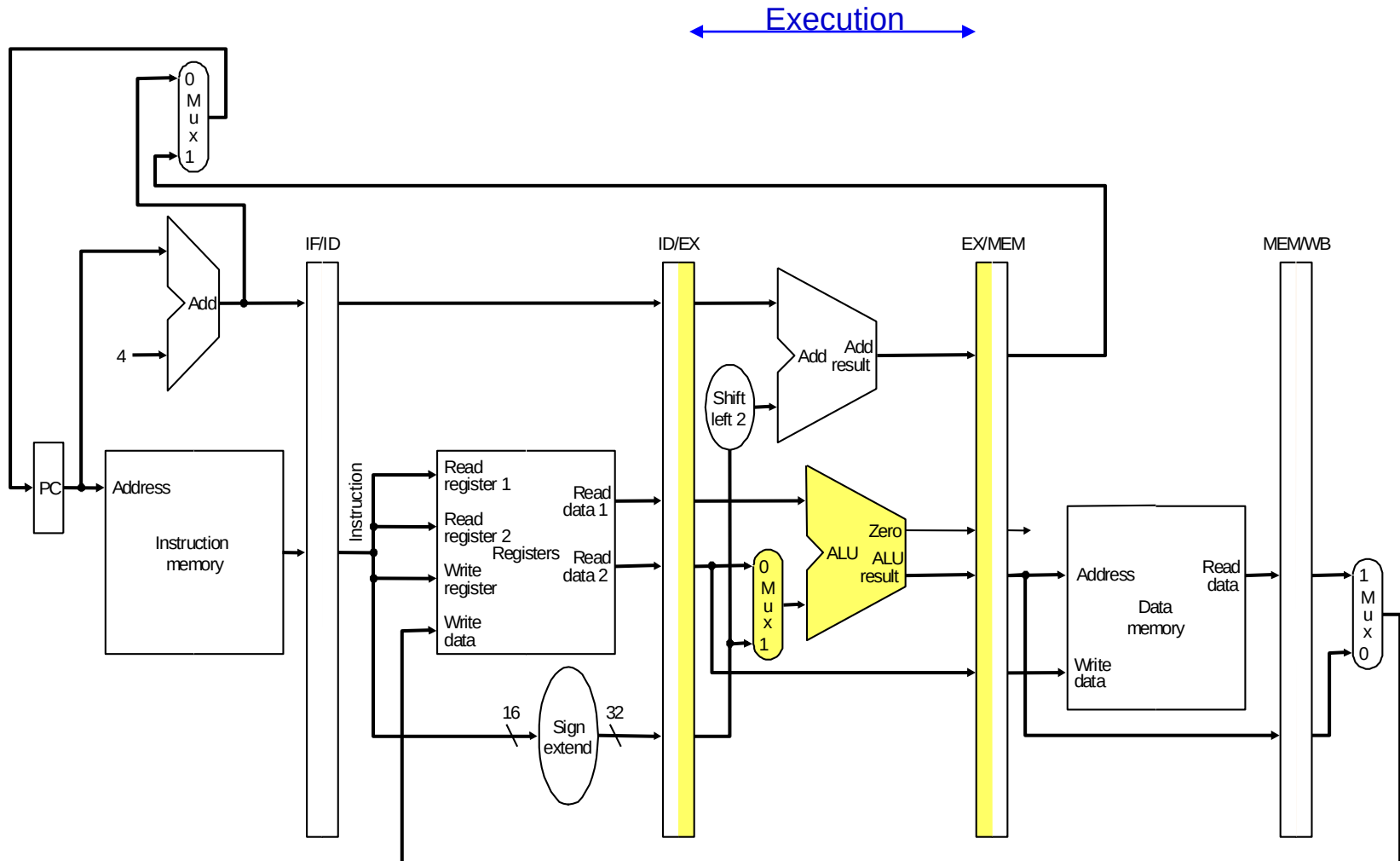
1w: Instruction Fetch (IF)



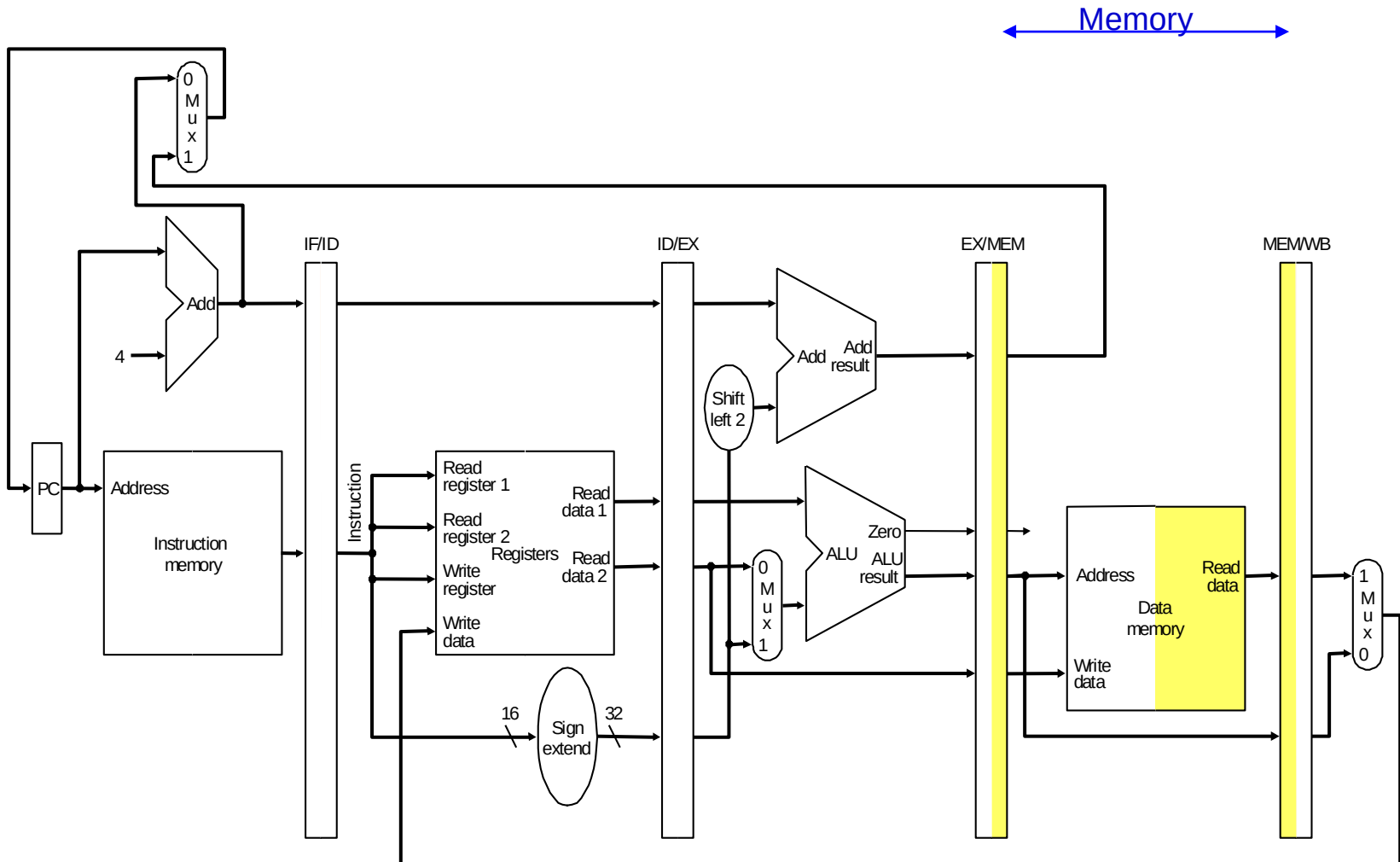
1w: Instruction Decode (ID)



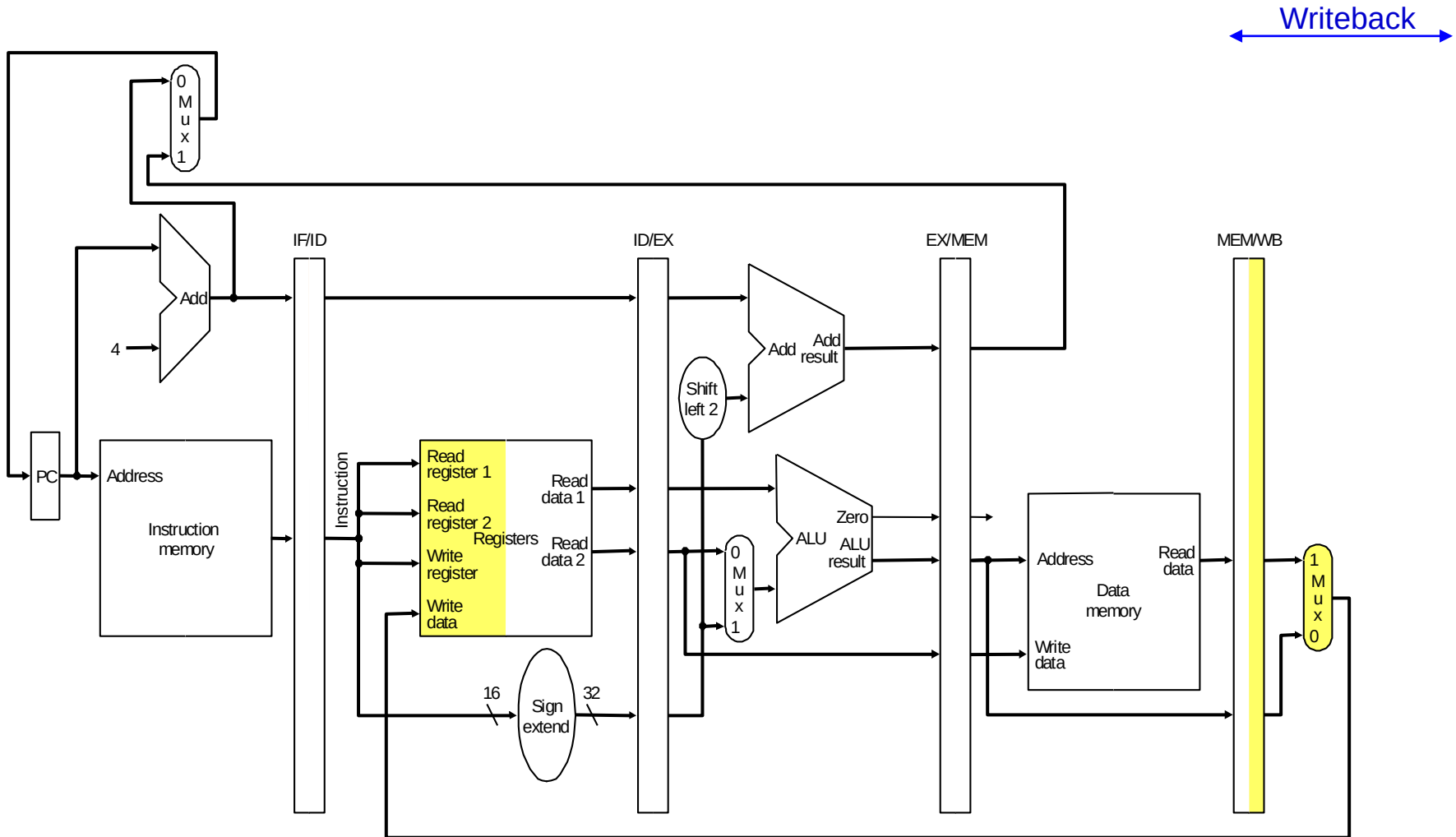
1w: Execution (EX)



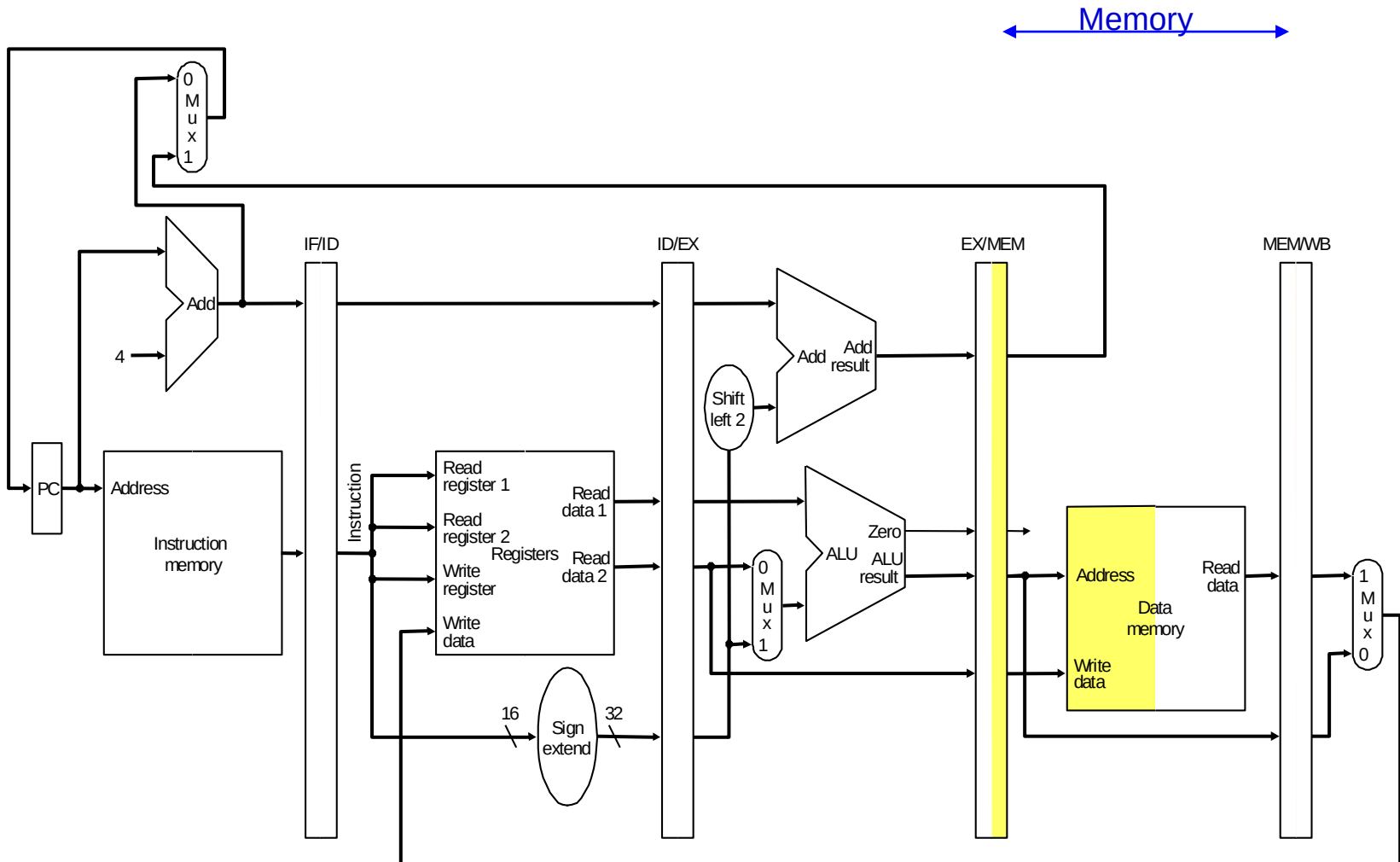
1w: Memory (MEM)



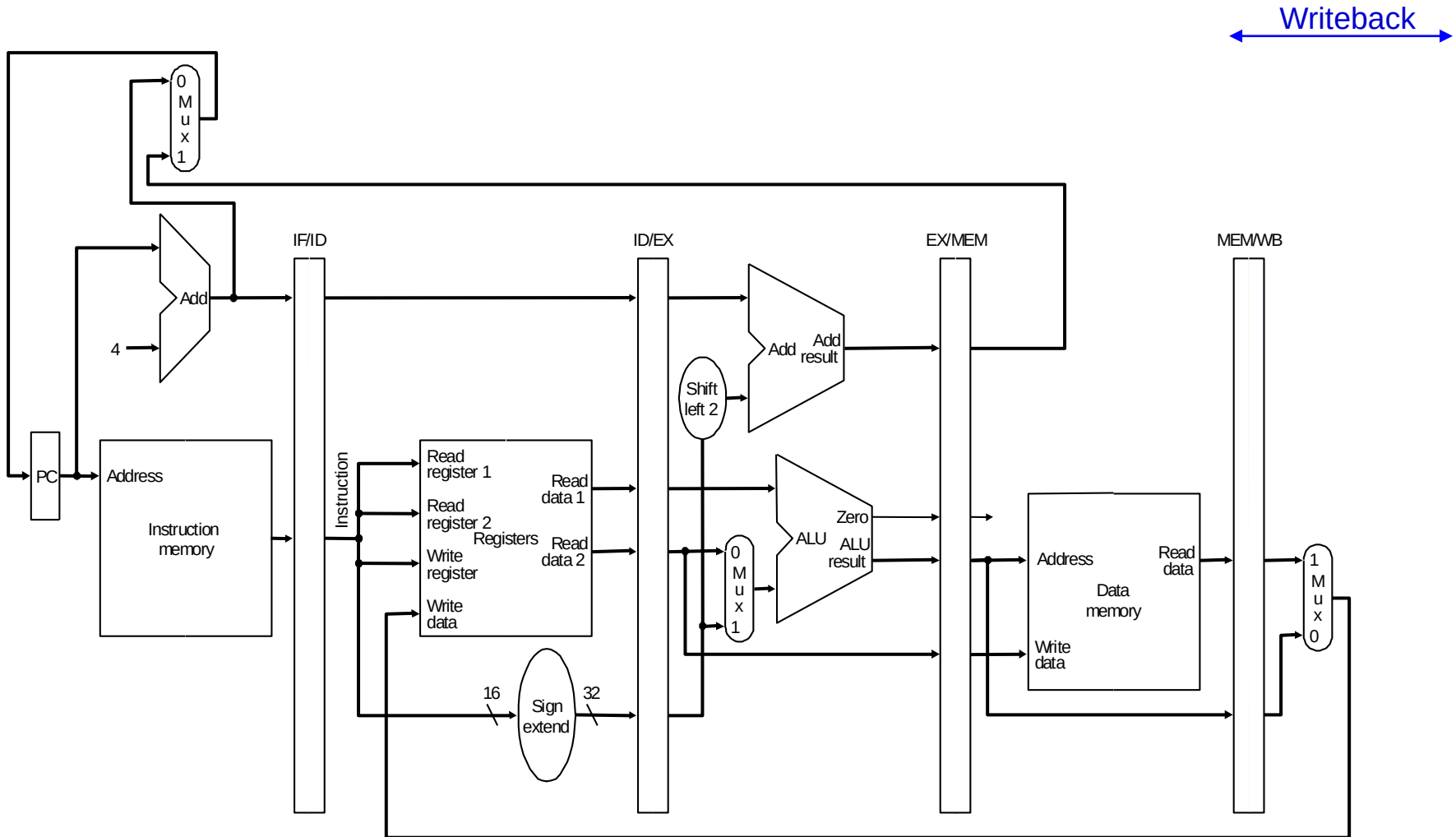
1w: Writeback (WB)



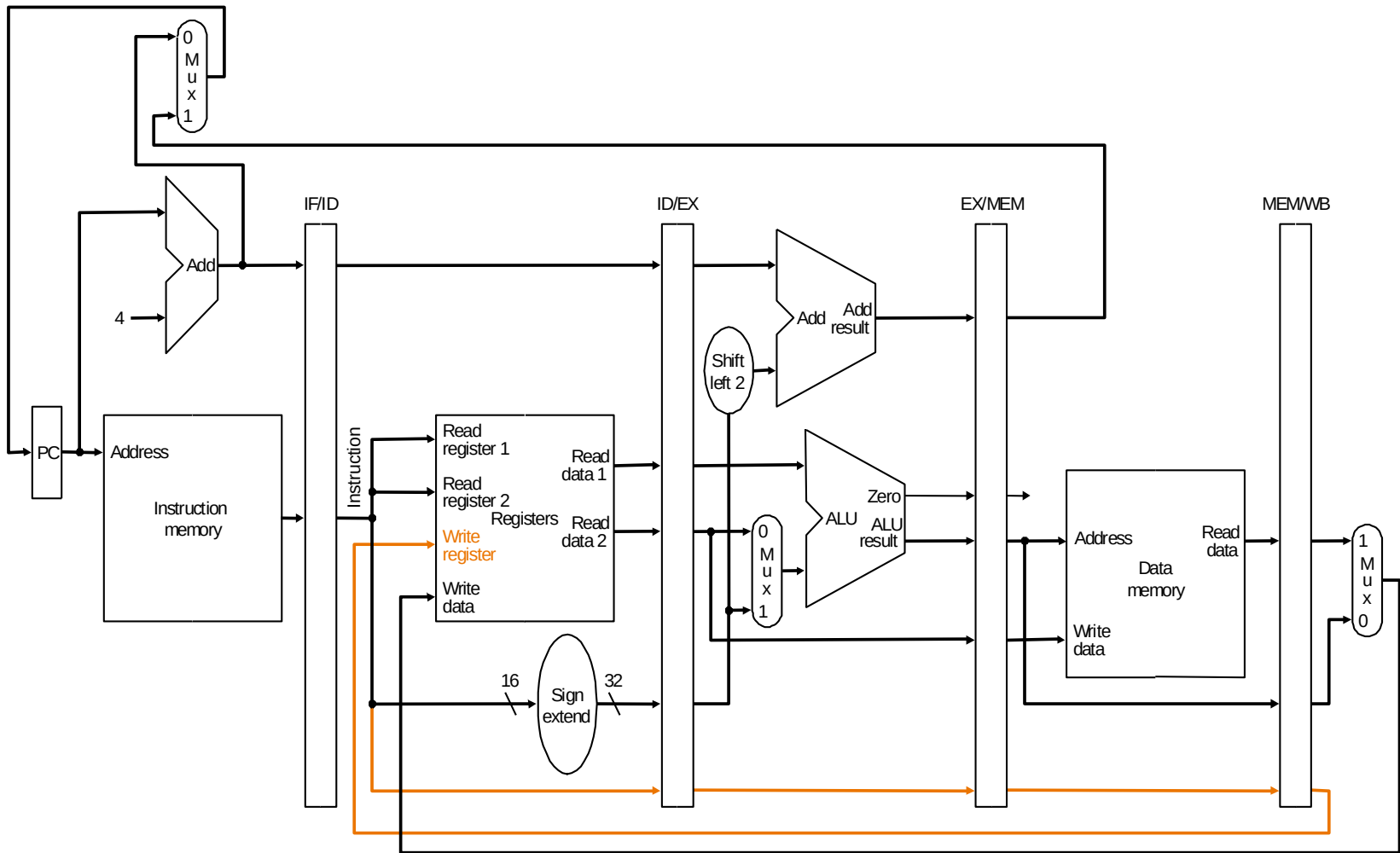
sw: Memory (MEM)



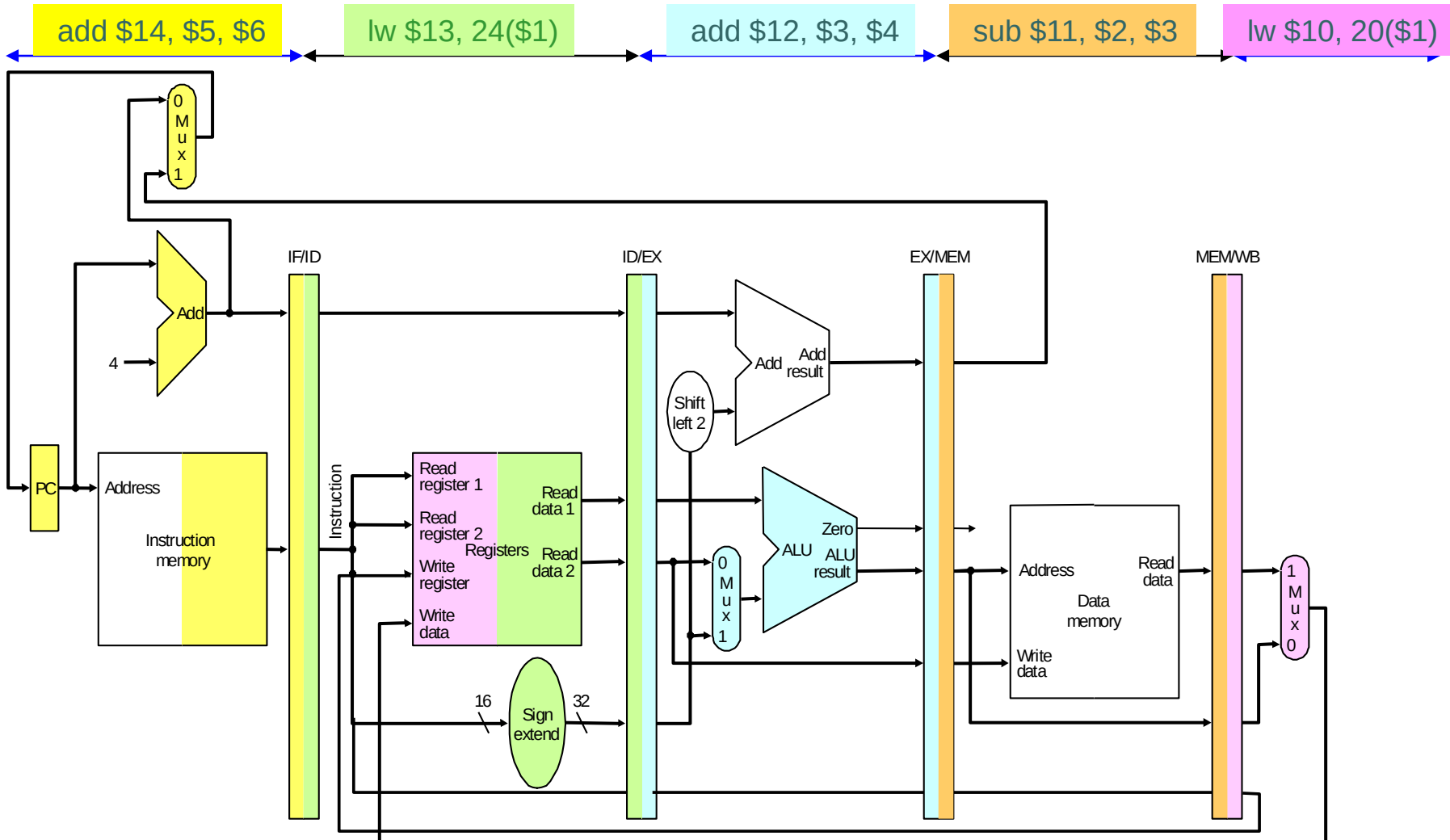
sw: Writeback (WB): do nothing



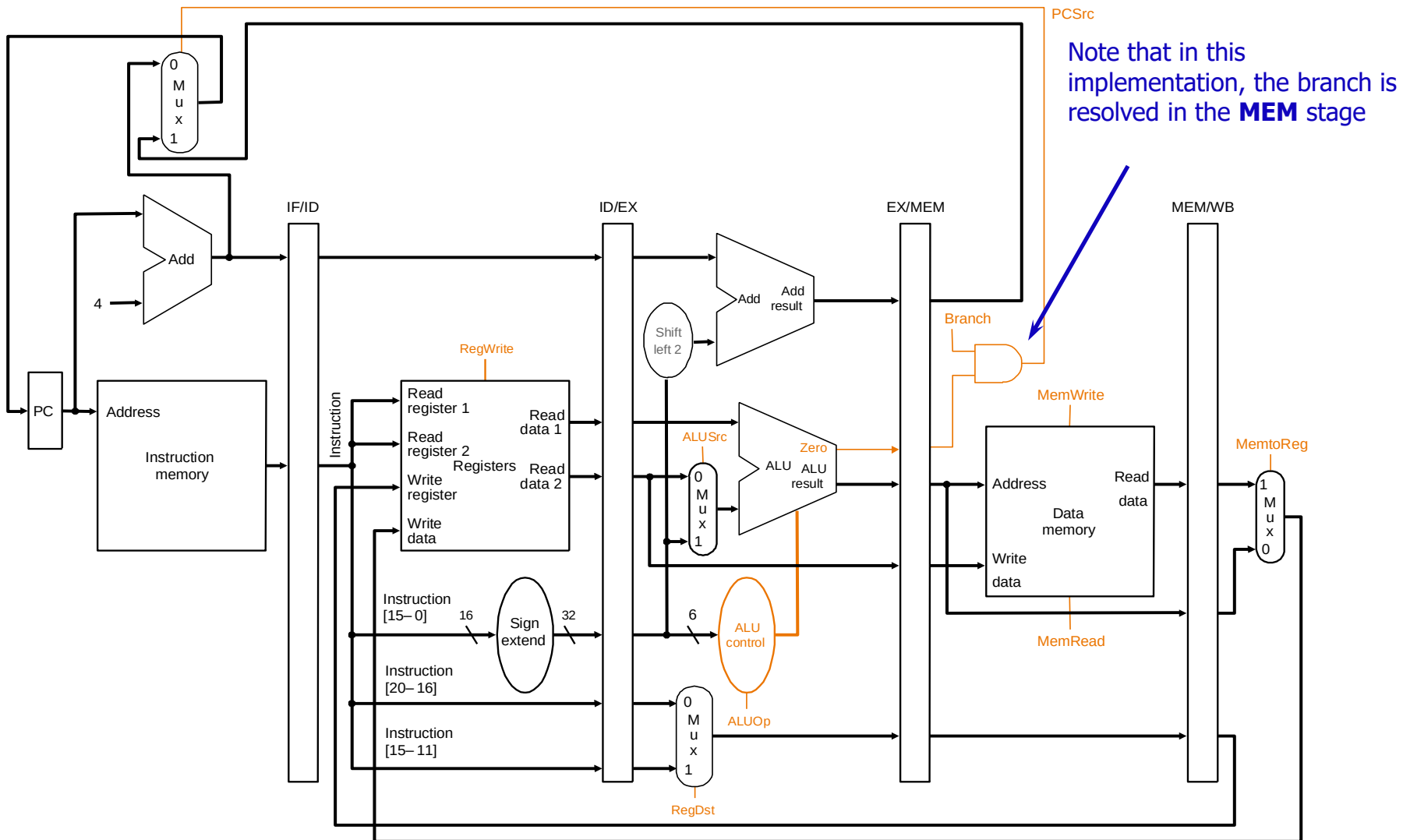
Corrected Datapath (for 1w)



Pipelining Example



Pipeline Control



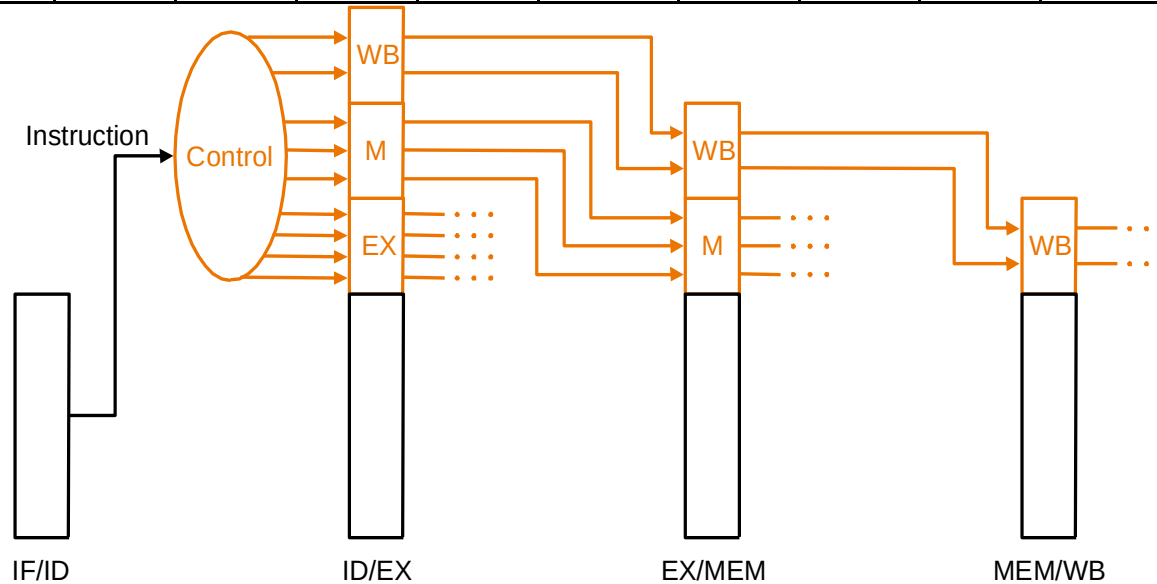
Pipeline Control

- 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

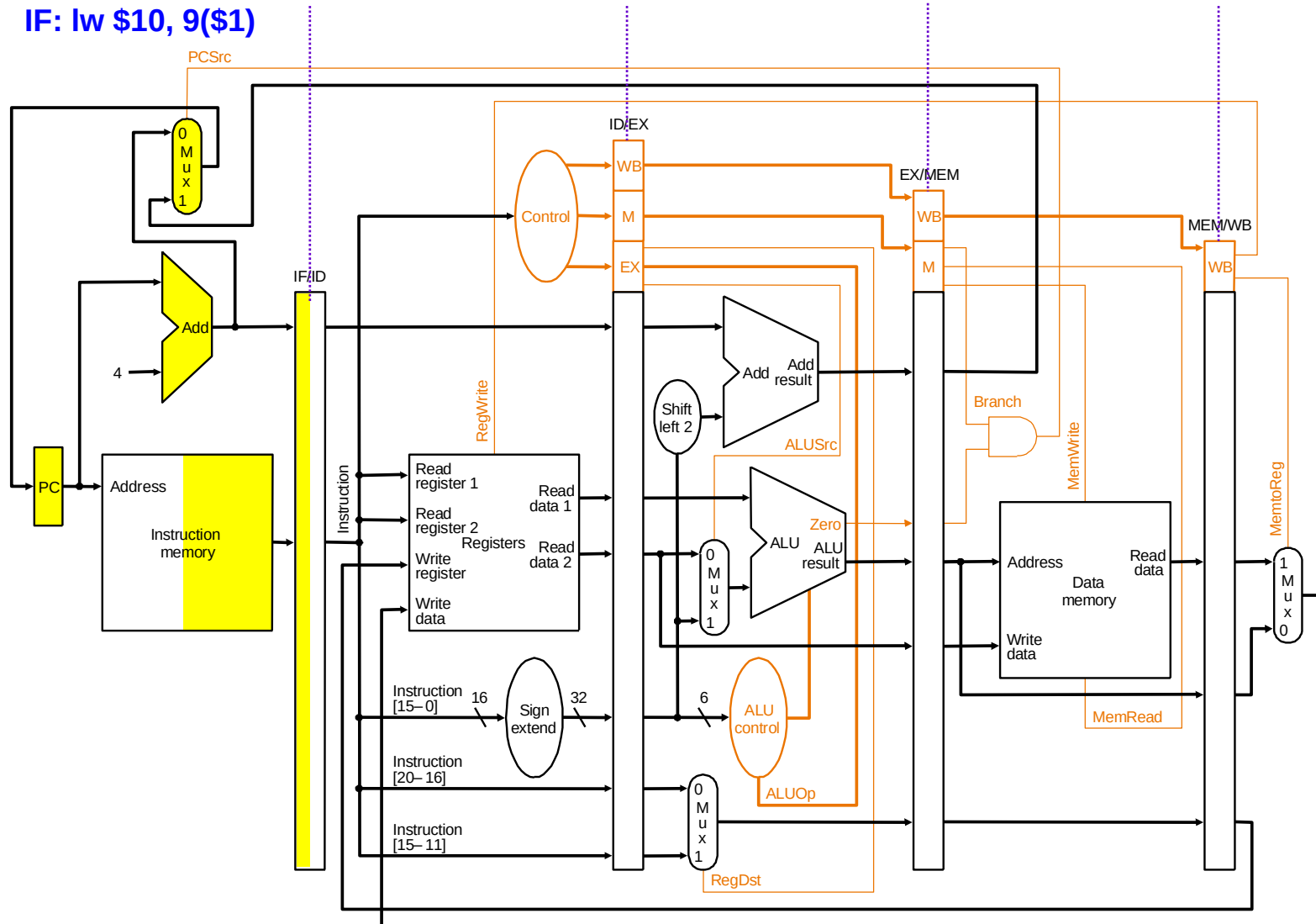


Korea Univ

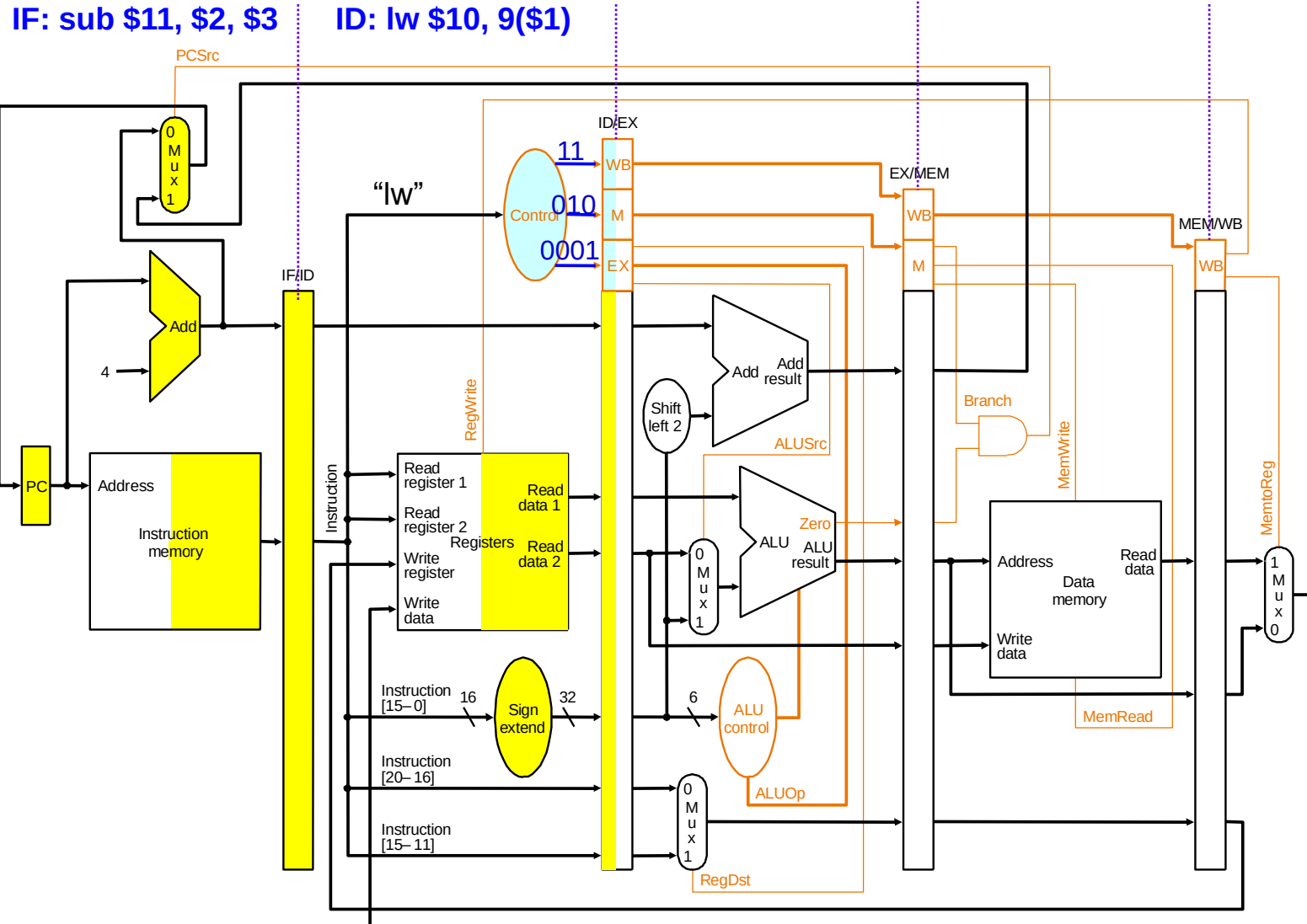


Datapath with Control

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



Datapath with Control

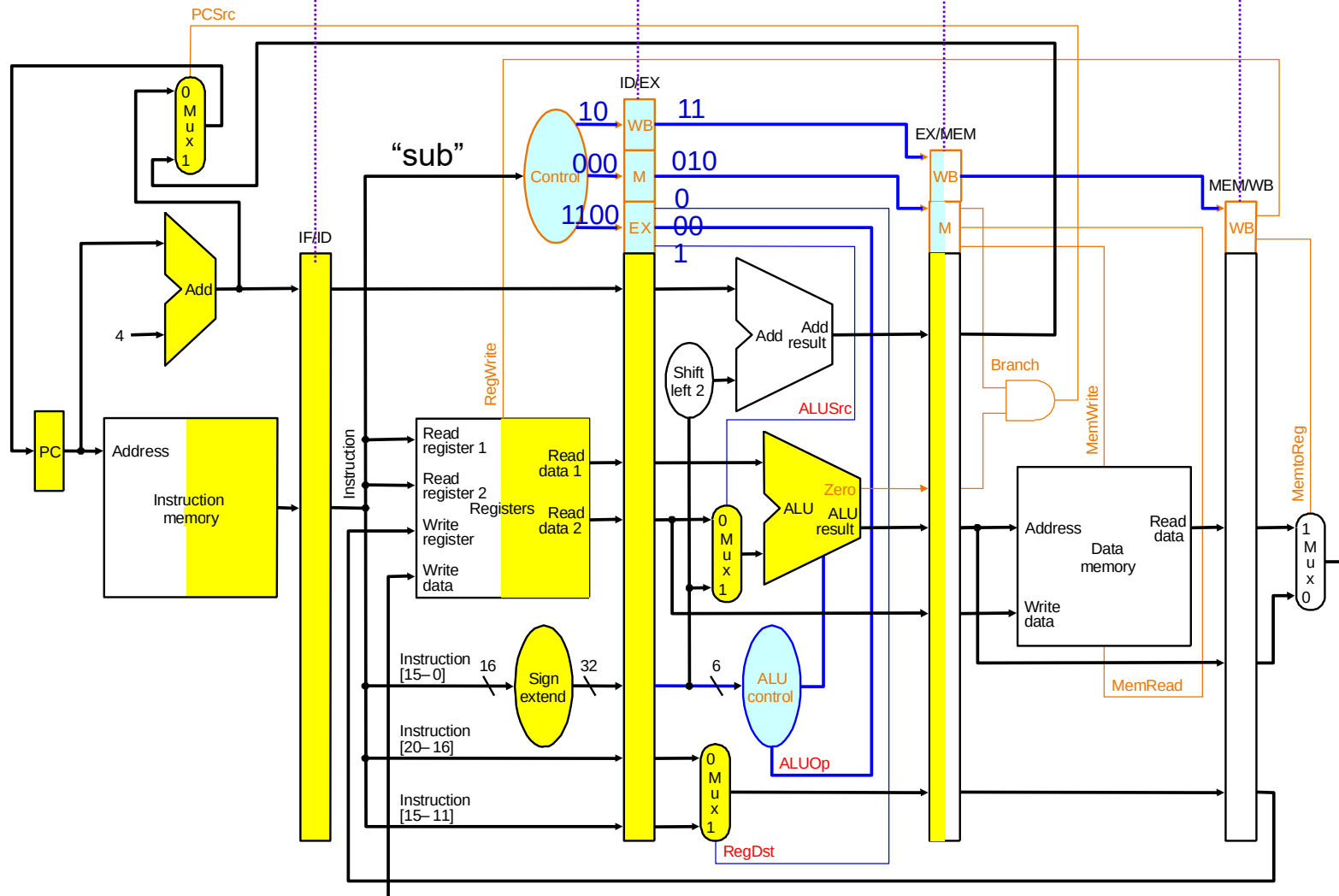


Datapath with Control

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

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

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



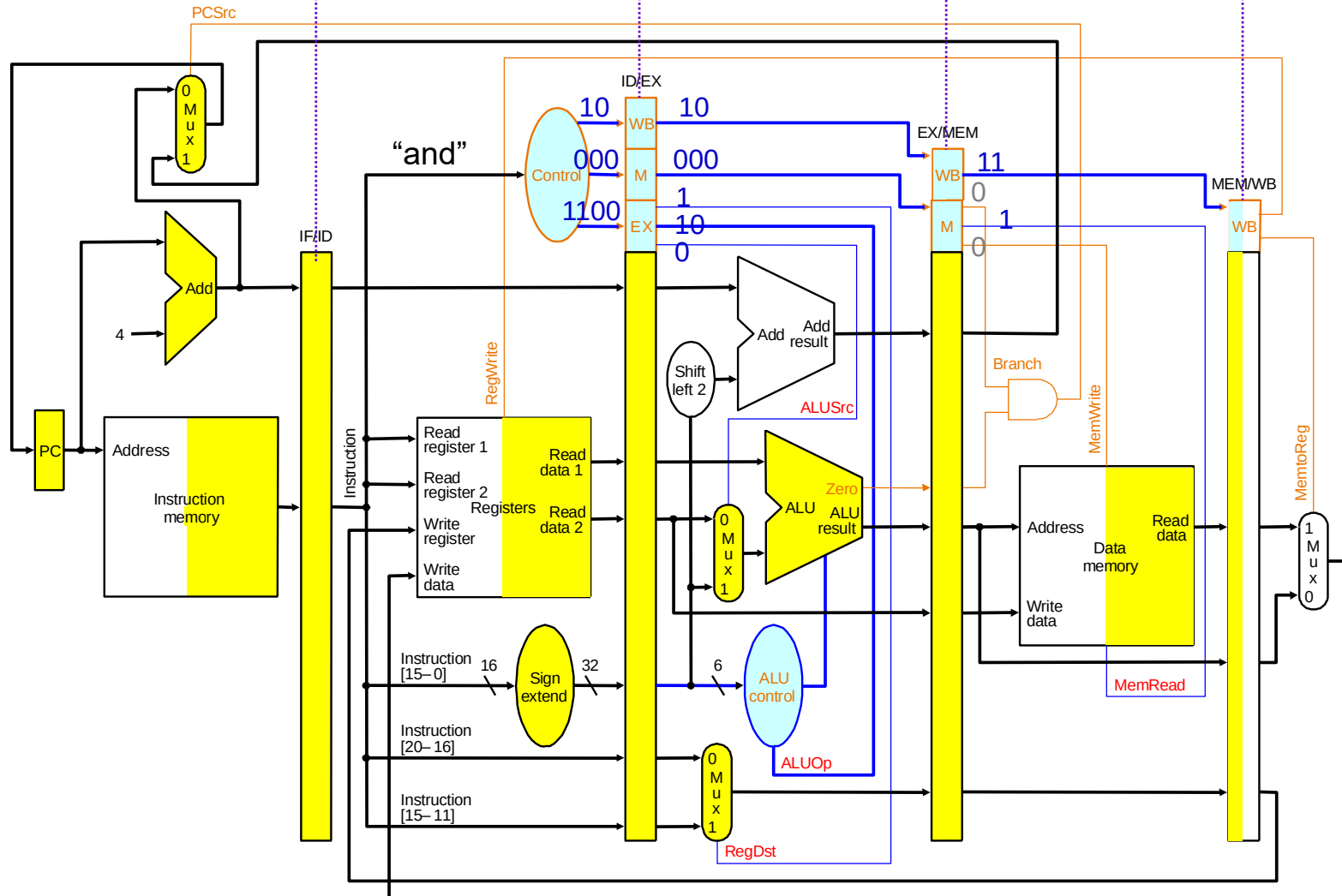
Datapath with Control

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

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

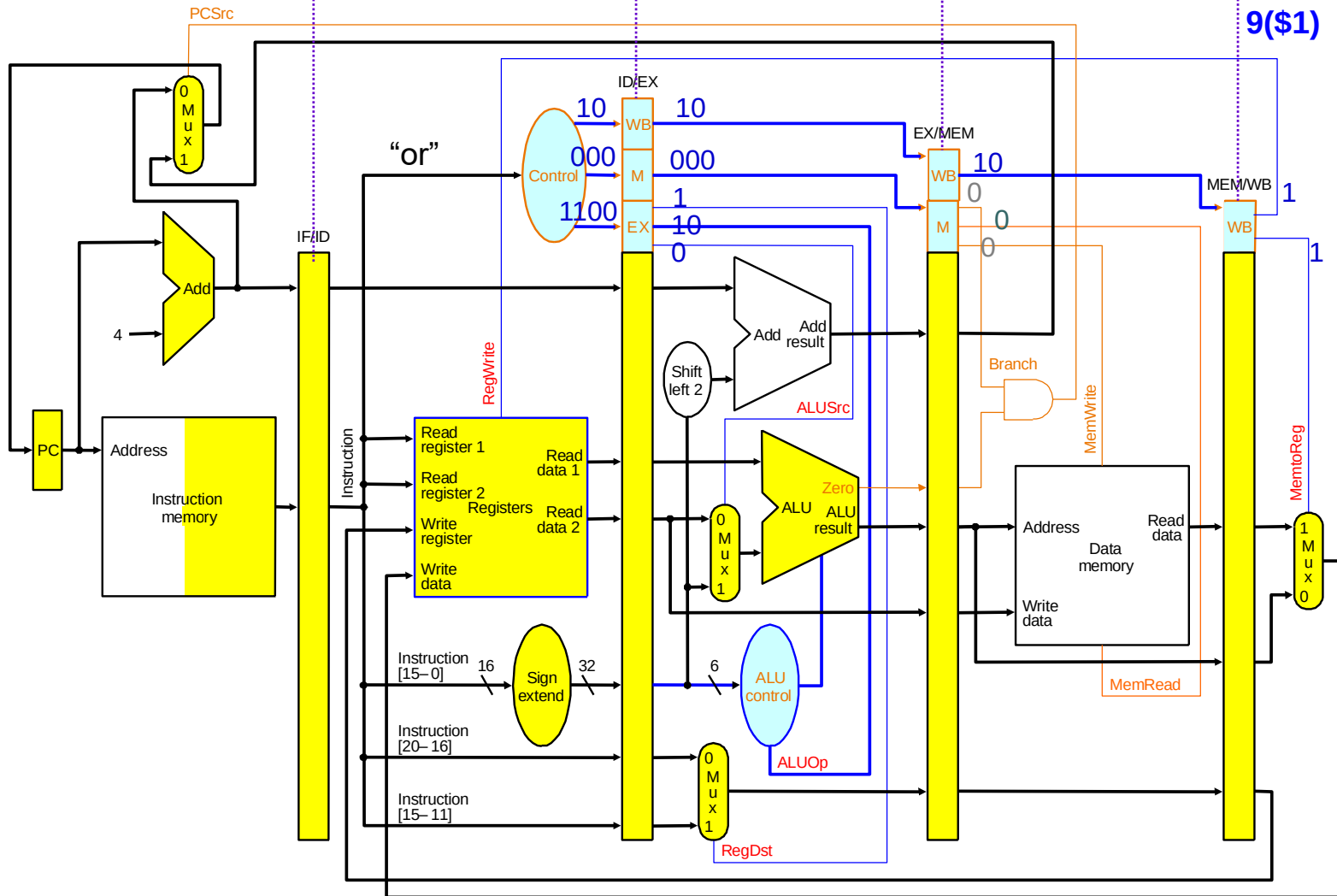
EX: sub \$11, \$2, \$3

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

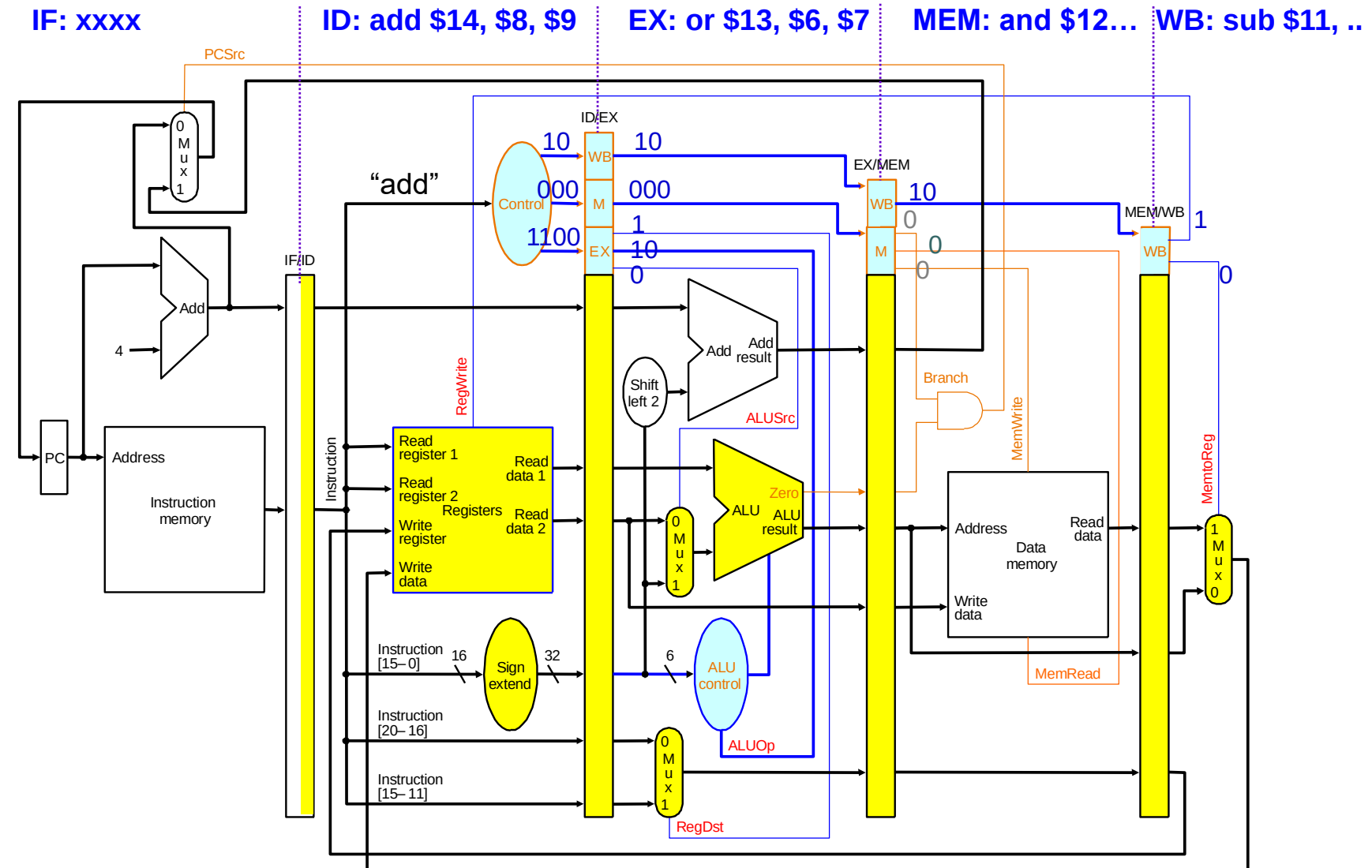


Datapath with Control

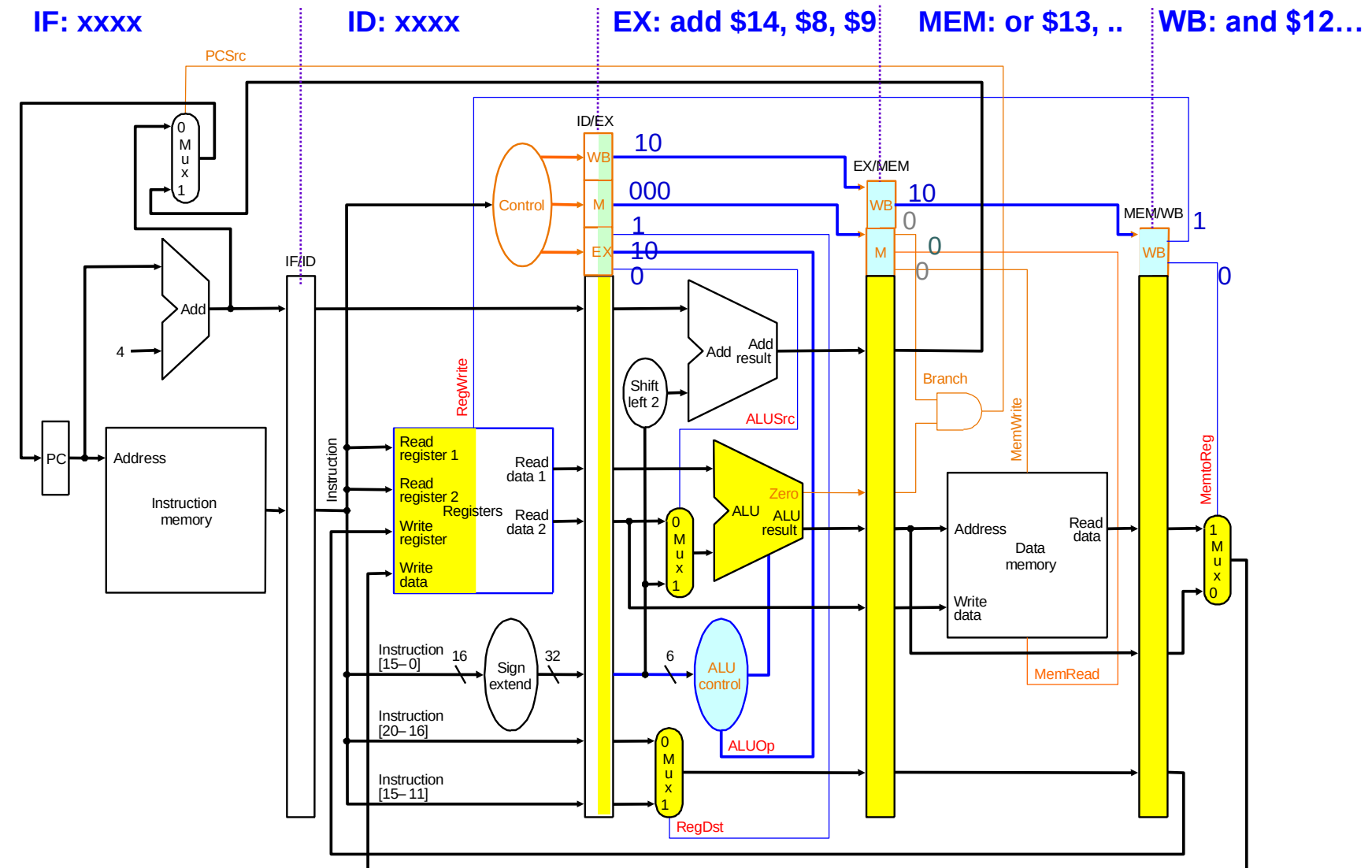
IF: add \$14, \$8, \$9 ID: or \$13, \$6, \$7 EX: and \$12, \$4, \$5 MEM: sub \$11, .. WB: lw \$10, 9(\$1)



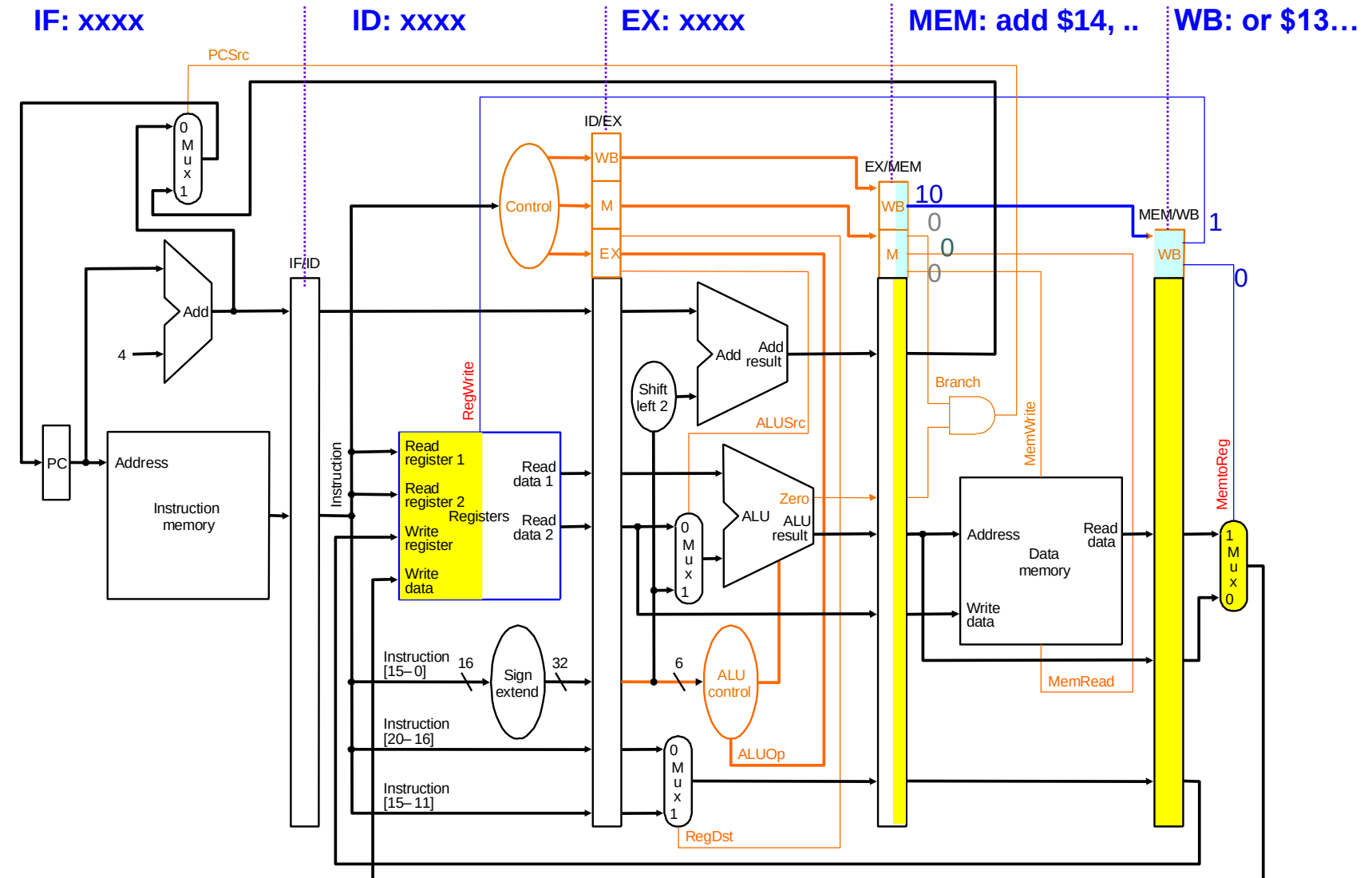
Datapath with Control



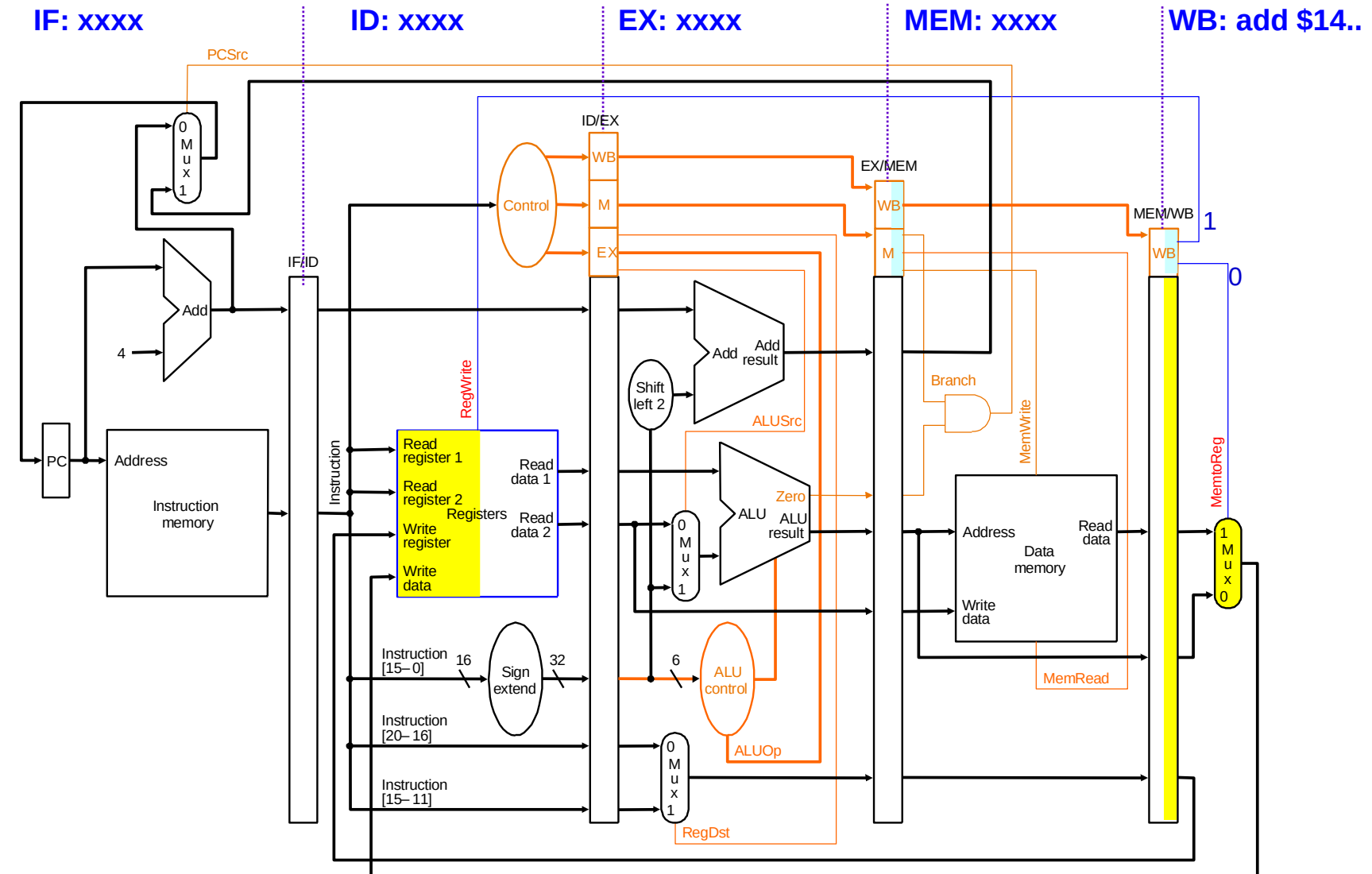
Datapath with Control



Datapath with Control



Datapath with Control

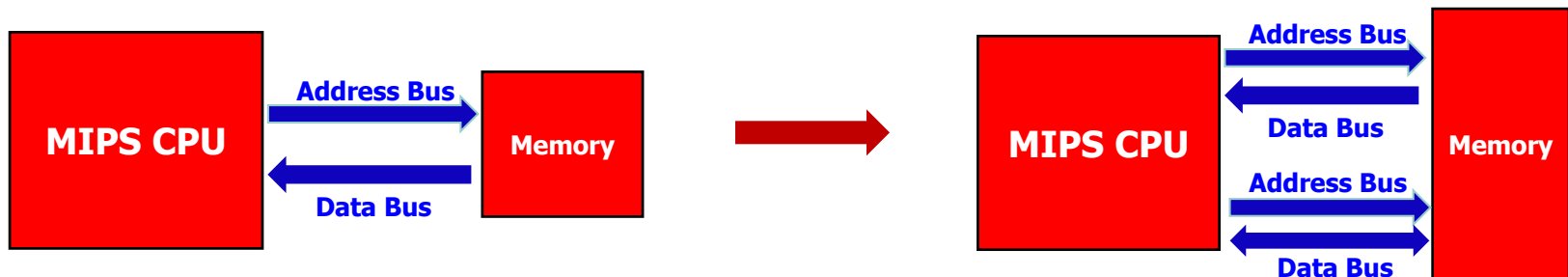


Hazards

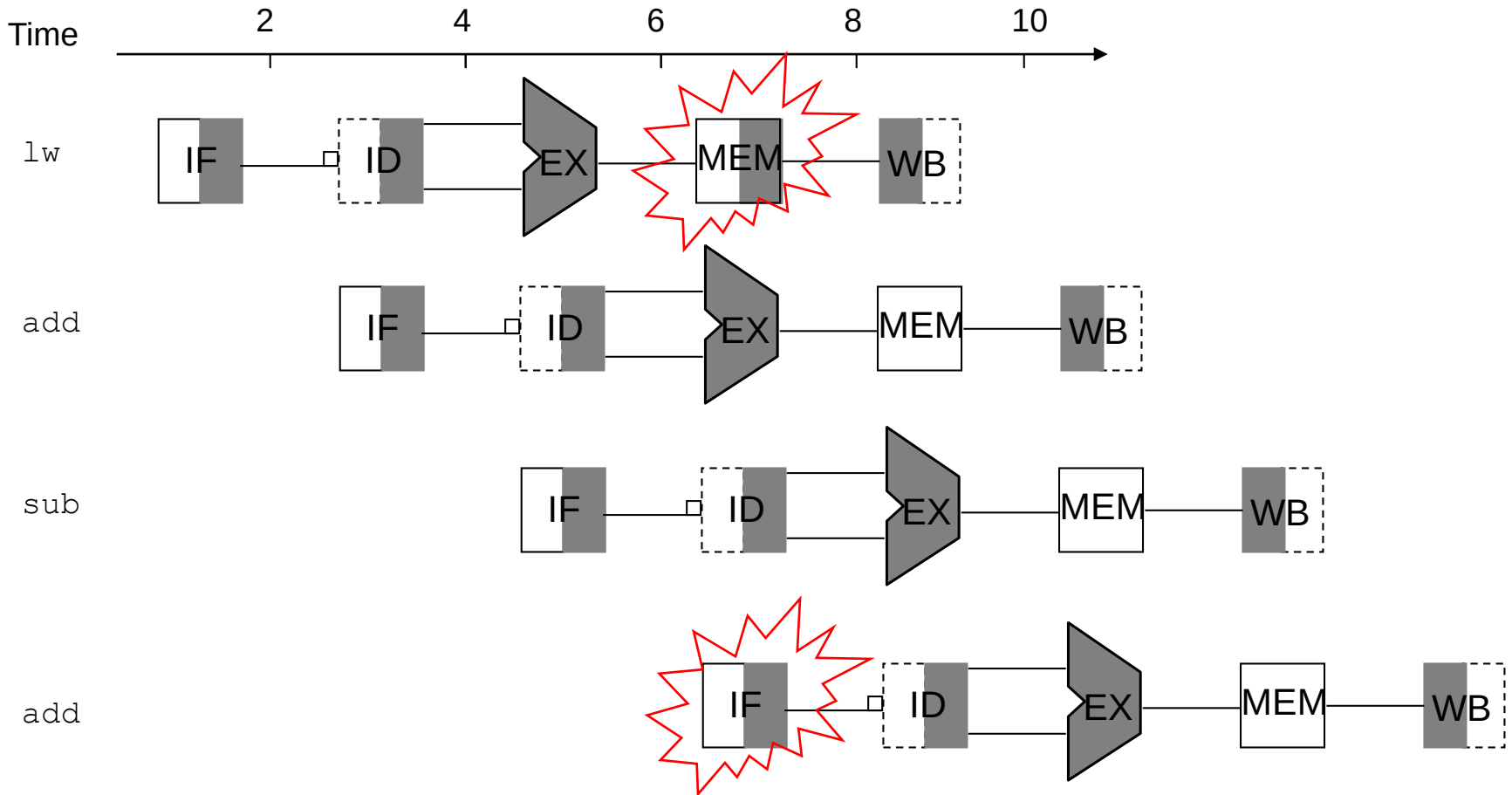
- It would be happy if we split the datapath into stages and the CPU works just fine
 - But, things are not that simple as you may expect
 - There are **hazards**!
- Hazard is a situation that prevents starting the next instruction in the next cycle
 - **Structure hazards**
 - Conflict over the use of a resource at the same time
 - **Data hazard**
 - Data is not ready for the subsequent dependent instruction
 - **Control hazard**
 - Fetching the next instruction depends on the previous branch outcome

Structure Hazards

- Structural hazard is a conflict over the use of a resource at the same time
- Suppose the MIPS CPU with a single memory
 - Load/store requires data access in MEM stage
 - Instruction fetch requires instruction access from the same memory
 - Instruction fetch would have to **stall** for that cycle
 - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require either separate ports to memory or separate memories for instruction and data



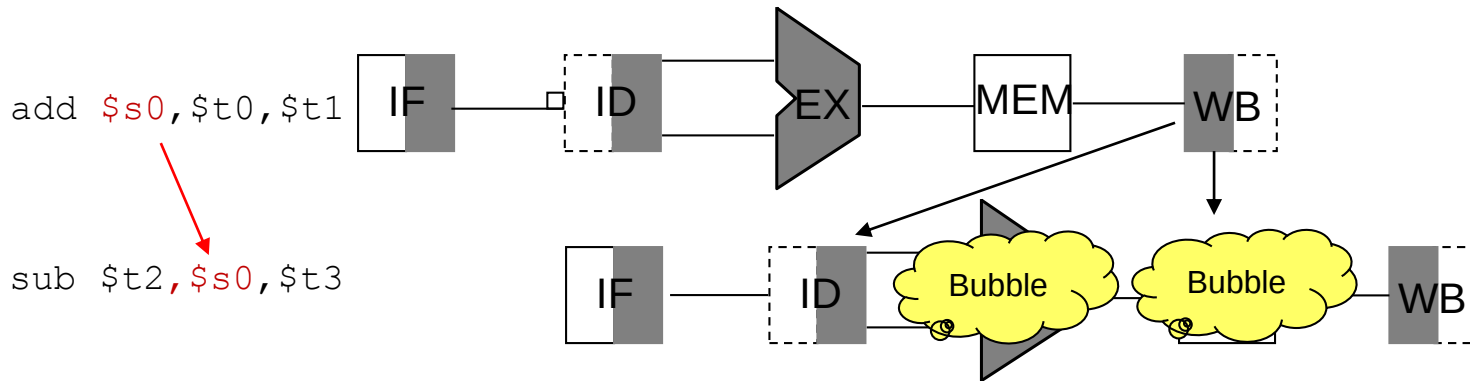
Structure Hazards (Cont.)



Either provide separate ports to access memory or to provide instruction memory and data memory separately

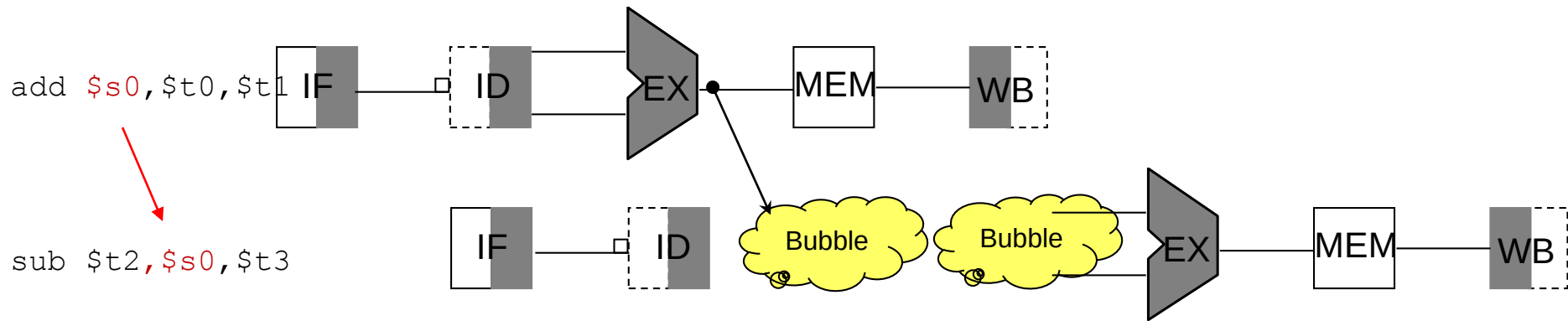
Data Hazards

- Data is not ready for the subsequent dependent instruction



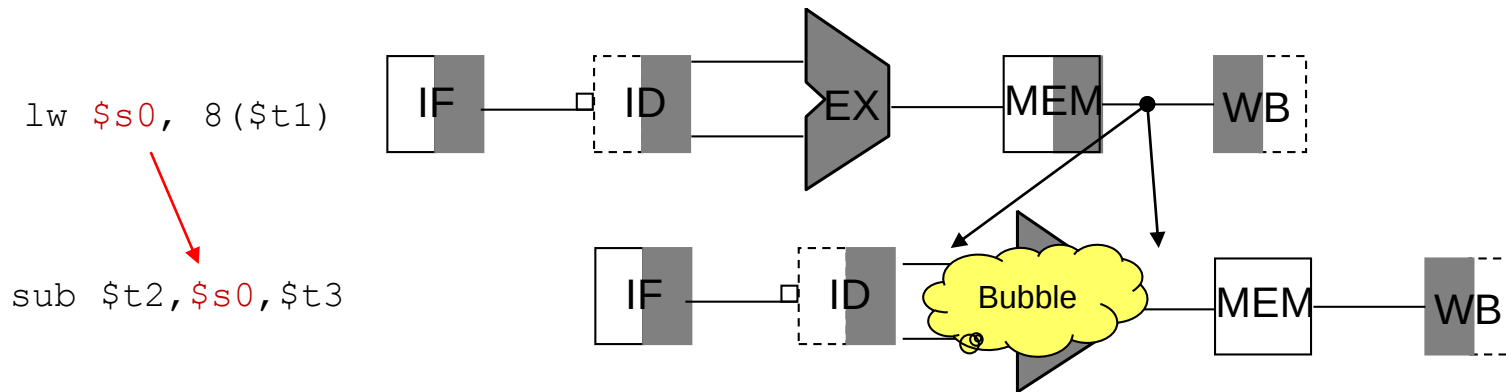
- To solve the data hazard problem, the pipeline needs to be stalled (typically referred to as "bubble")
 - Then, the performance is penalized
- A better solution?
 - **Forwarding** (or Bypassing)

Forwarding



Data Hazard - Load-Use Case

- Can't always avoid stalls by forwarding
 - Can't forward backward in time!
- **Hardware interlock** is needed for the pipeline stall

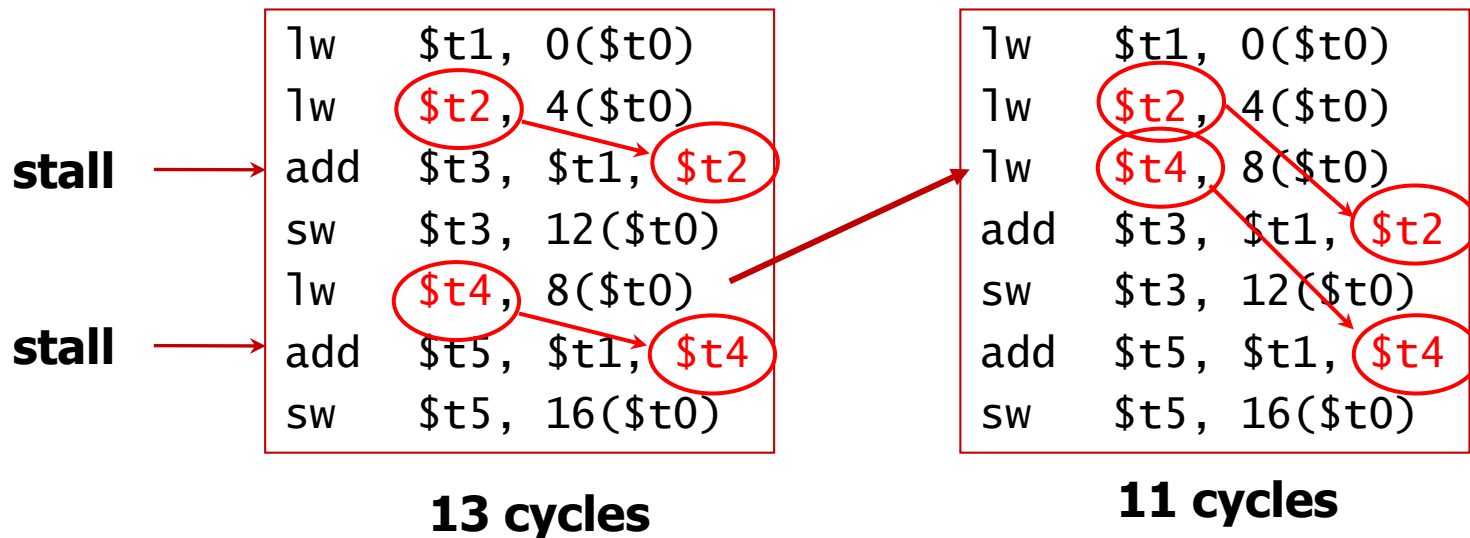


- This bubble can be hidden by proper instruction scheduling

Code Scheduling to Avoid Stalls

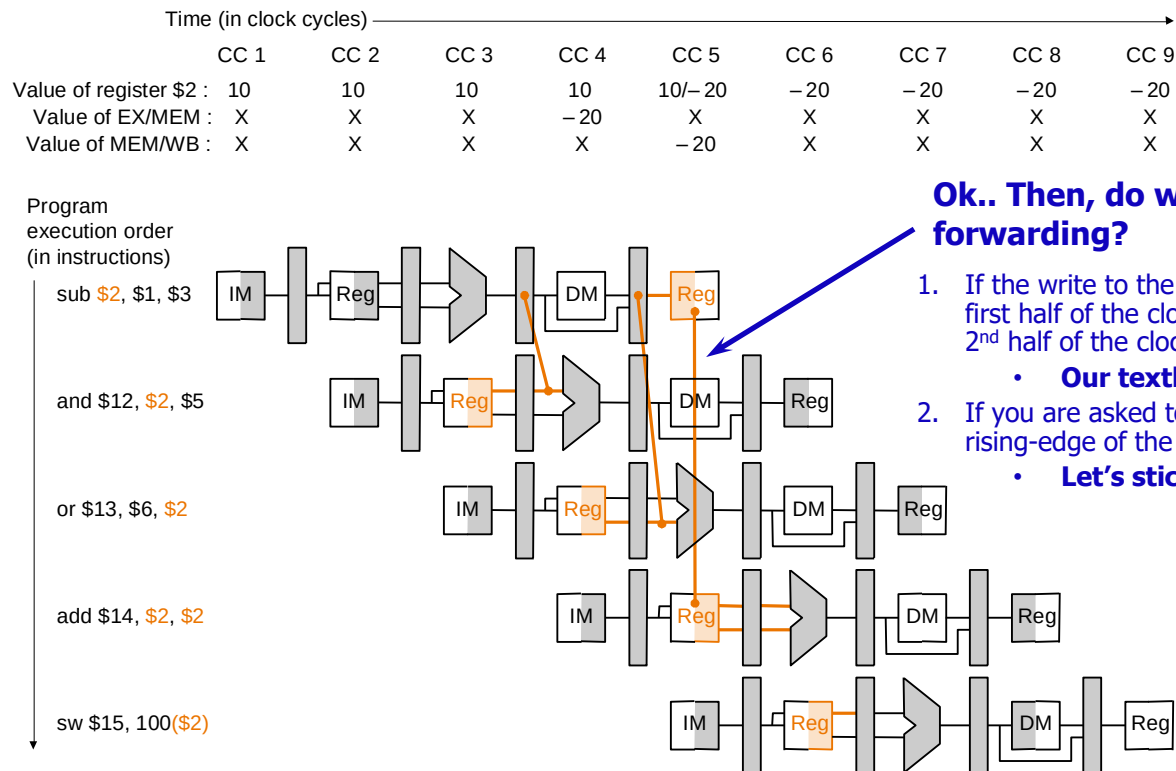
- Reorder code to avoid use of load result in the next instruction

`A = B + E; // B is loaded to $t1, E is loaded to $t2`
`C = B + F; // F is loaded to $t4`



Data Hazard - Forwarding

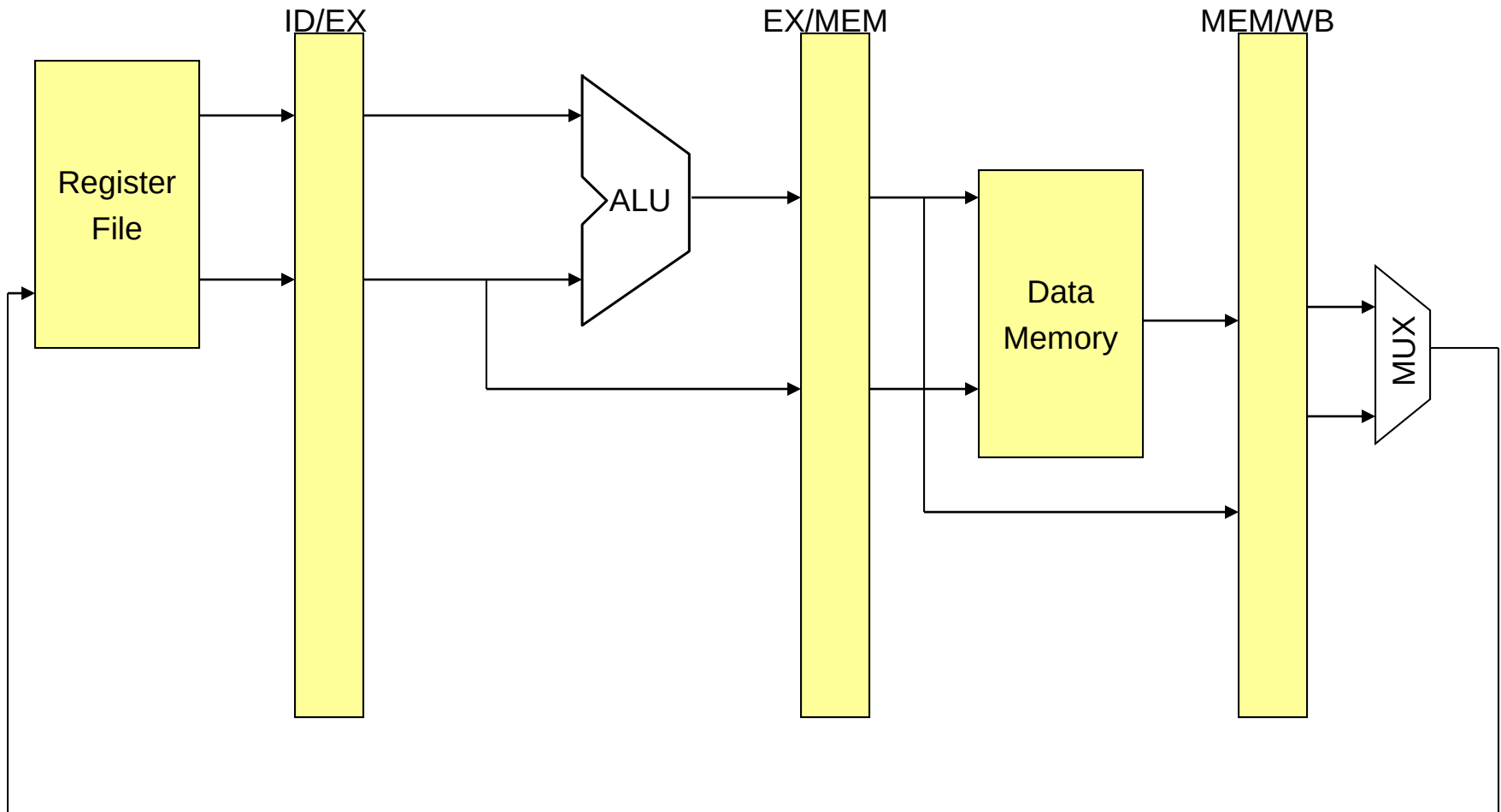
- Don't wait for them to be written to the register file
 - Use temporary results



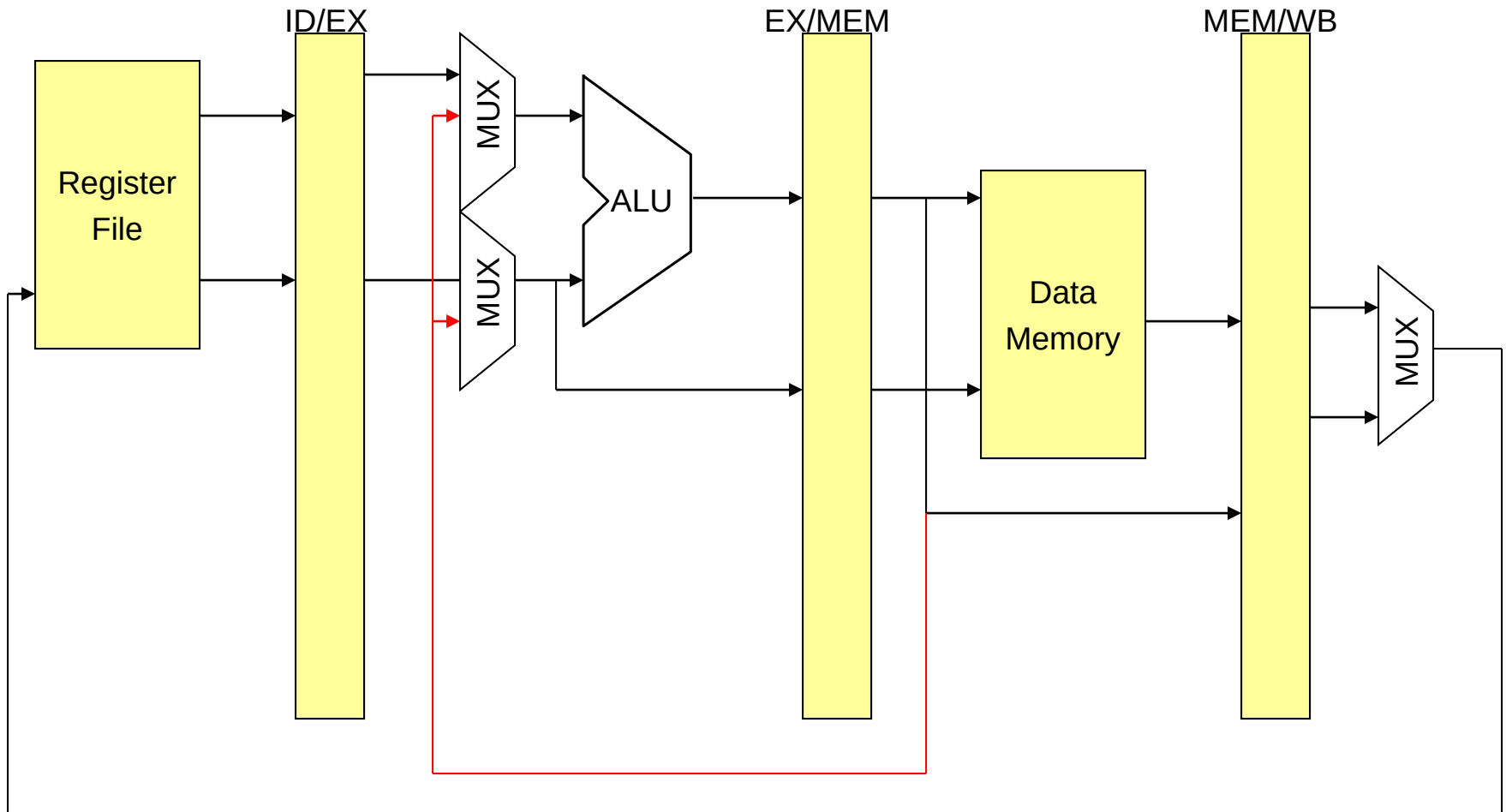
Ok.. Then, do we have to do this forwarding?

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 you are asked to design CPU using only rising-edge of the clock, then?
 - **Let's stick to this for our project**

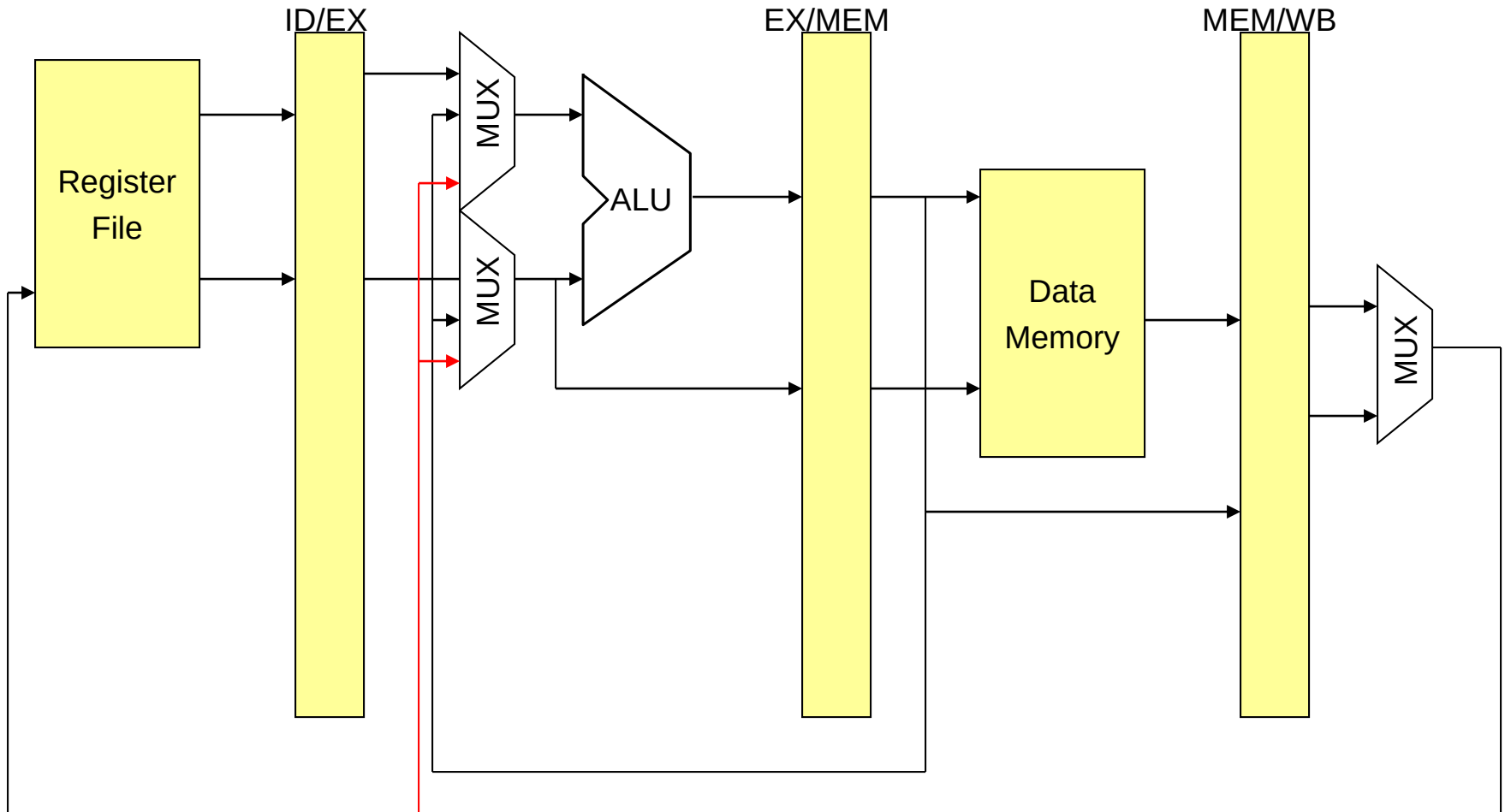
Forwarding



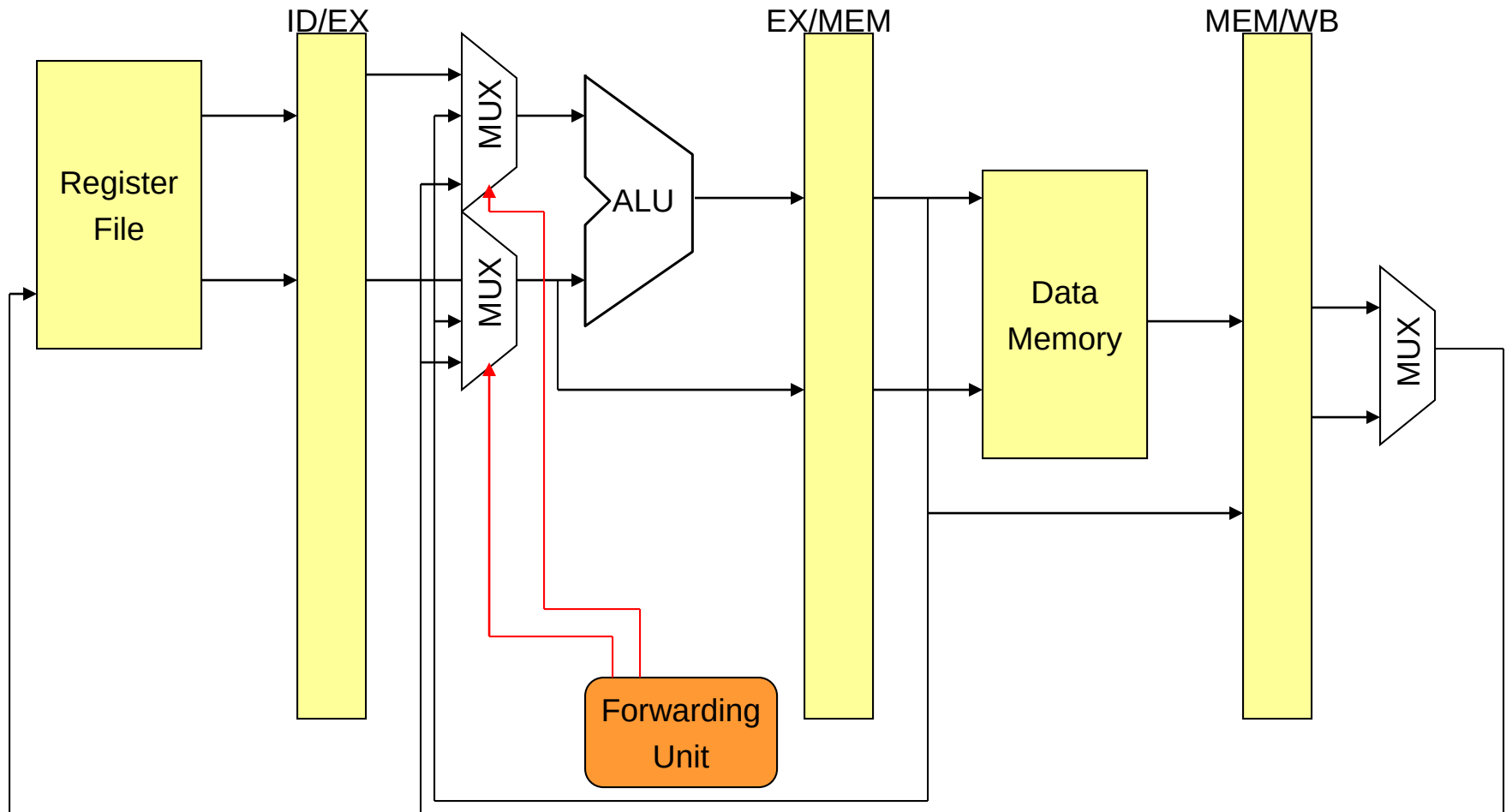
Forwarding (from EX/MEM)



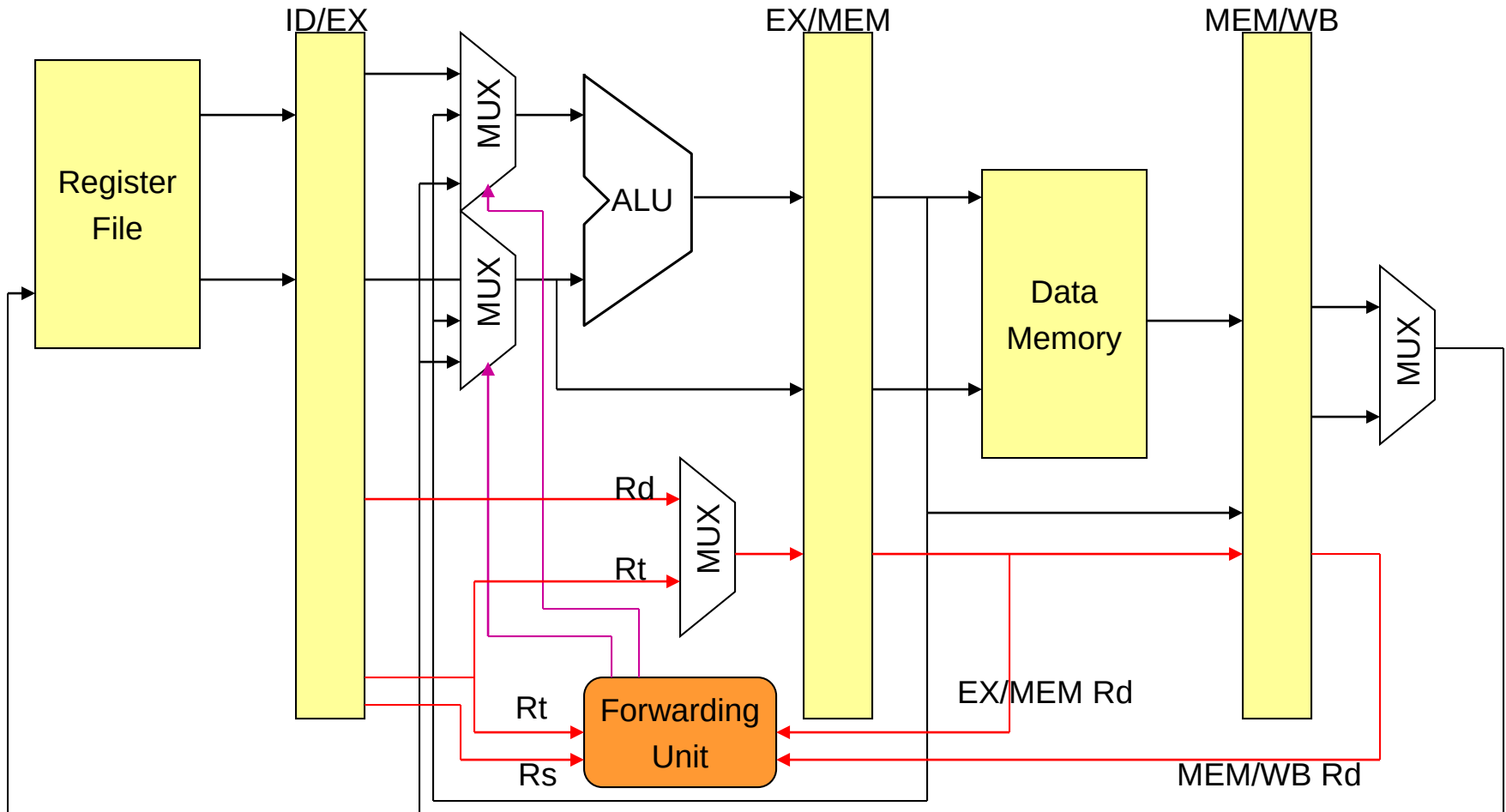
Forwarding (from MEM/WB)



Forwarding (operand selection)

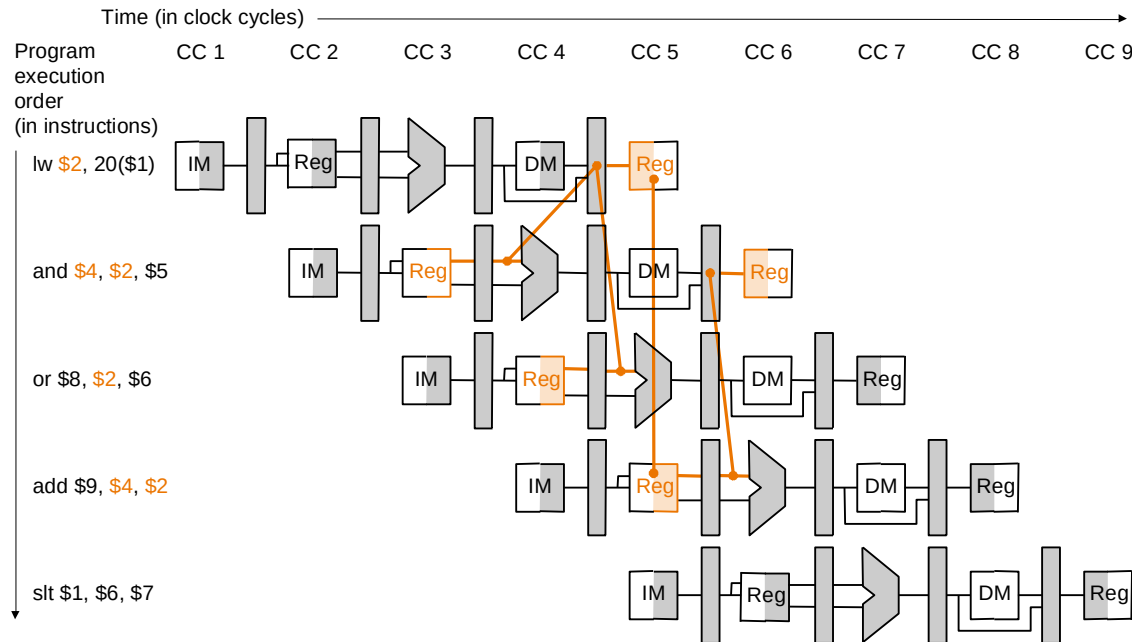


Forwarding (operand propagation)



Can't always forward

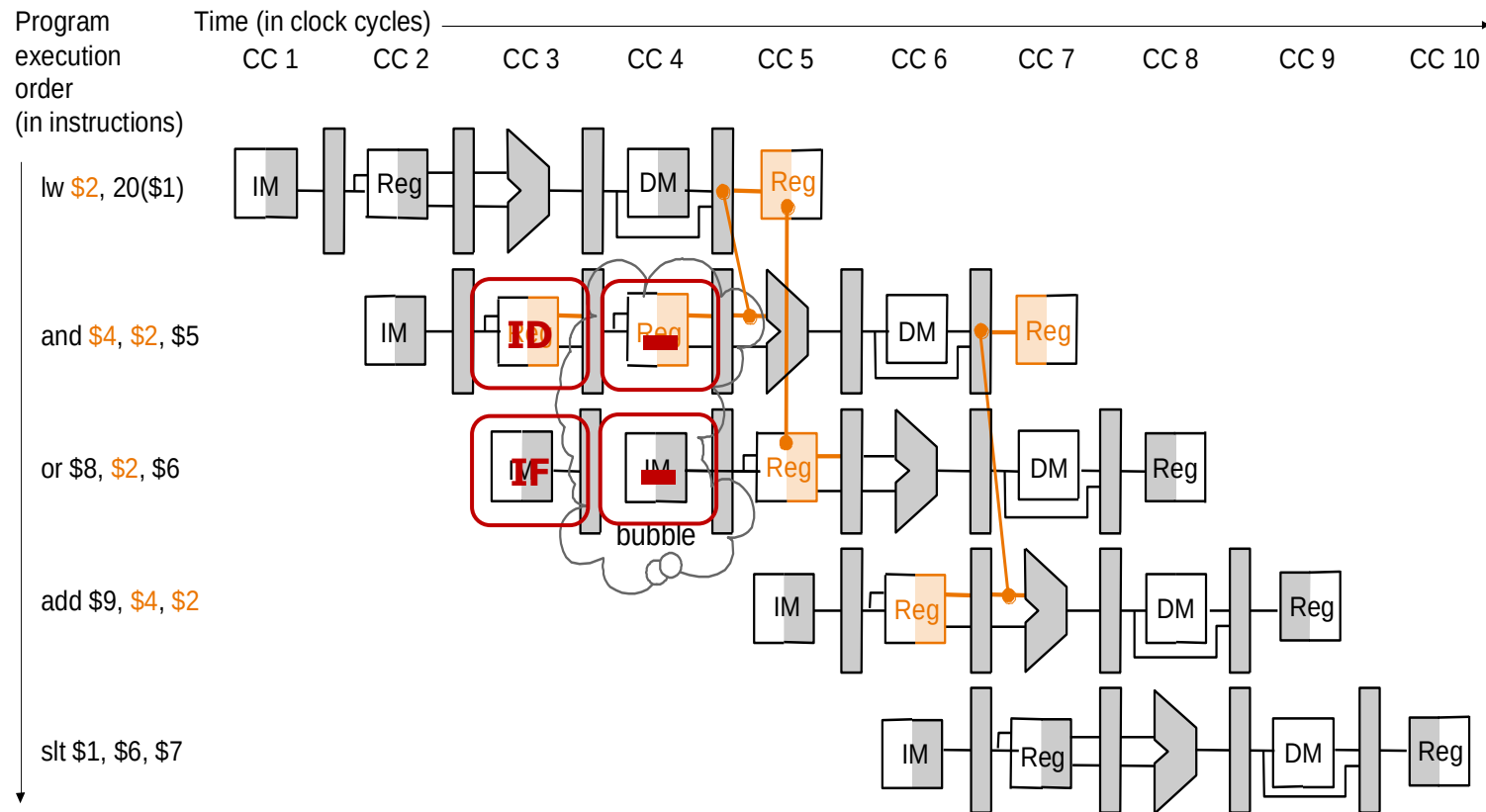
- lw (load word) can still cause a hazard
 - An instruction tries to read a register following a load instruction that writes to the same register



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

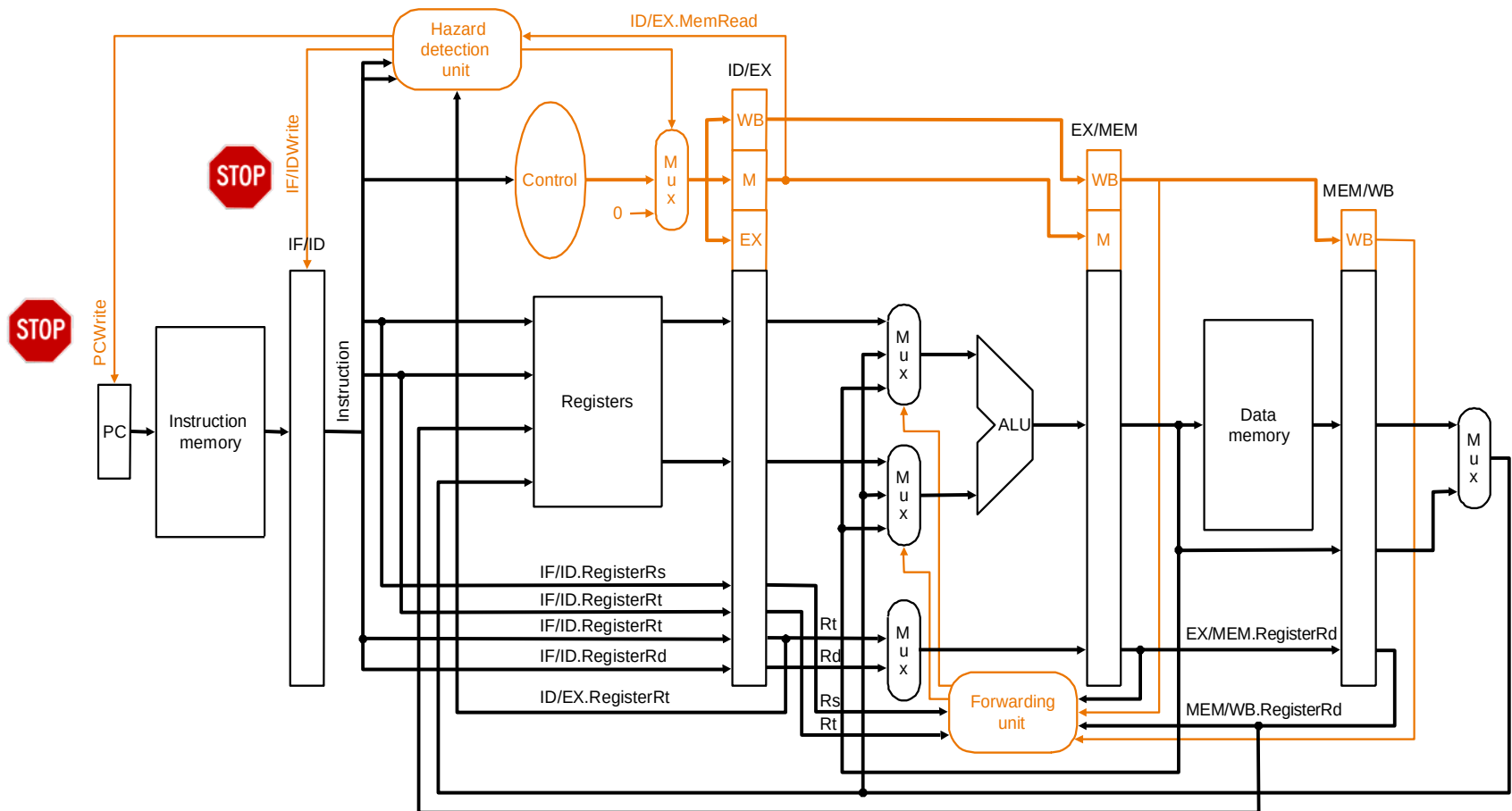
Stalling

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



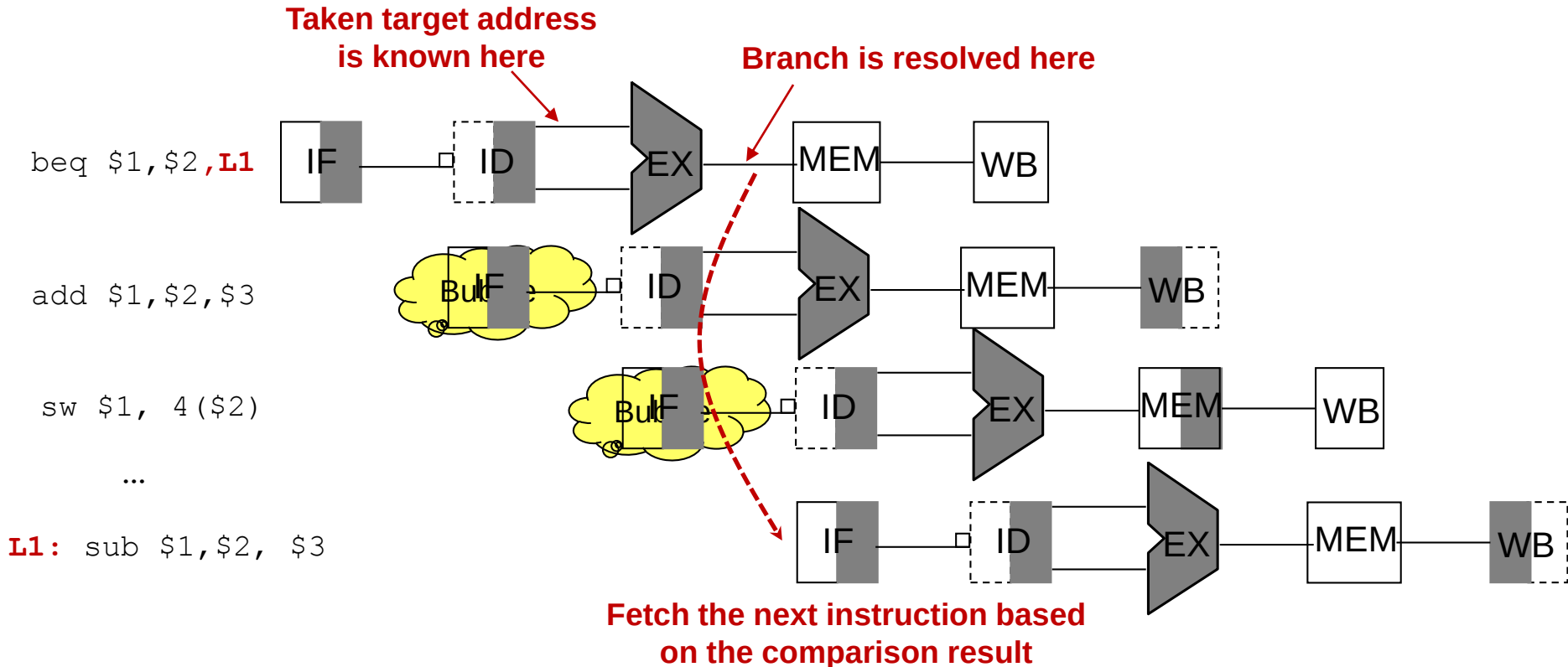
Hazard Detection Unit

- 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



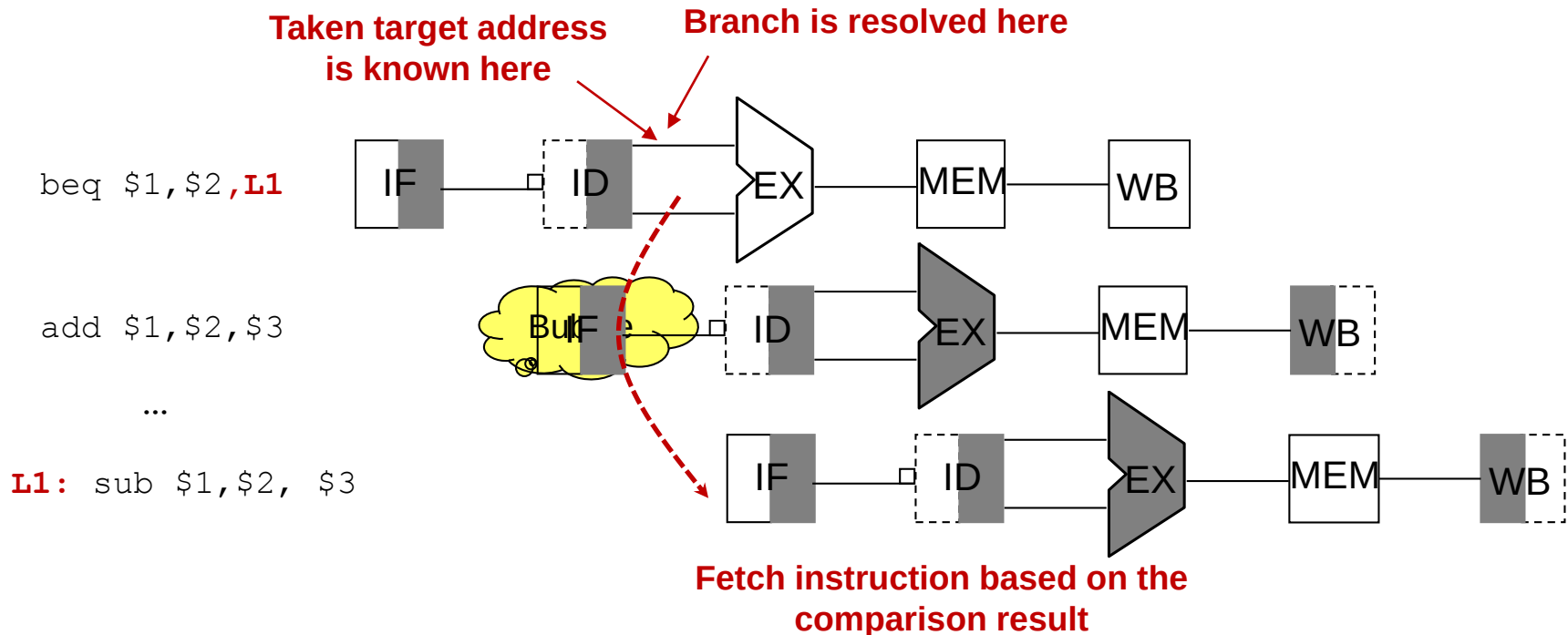
Control Hazard

- Branch determines the flow of instructions
- Fetching the next instruction depends on the branch outcome
 - Pipeline can't always fetch correct instruction
 - Branch instruction is still working on ID stage when fetching the next instruction



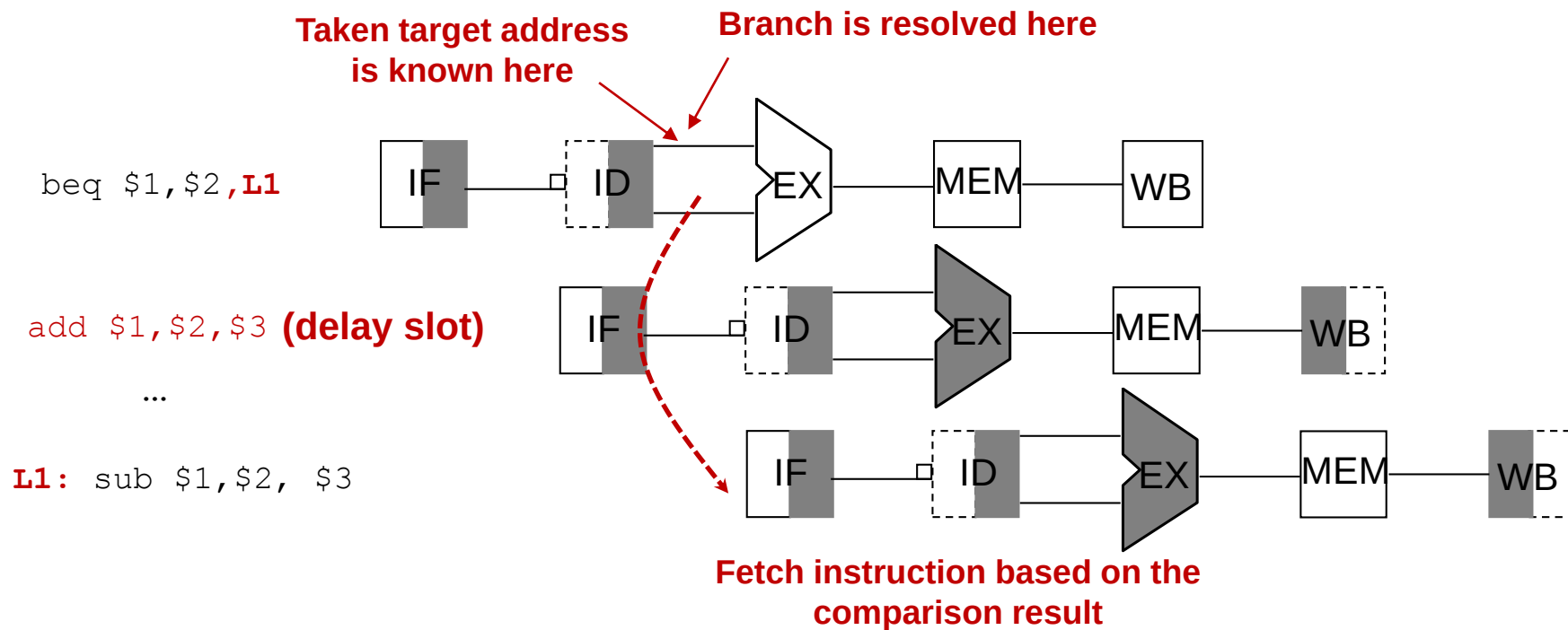
Reducing Control Hazard

- To reduce 2 bubbles to 1 bubble, add hardware in ID stage to compare registers (and generate branch condition)
 - But, it requires additional forwarding and hazard detection logic – Why?



Delayed Branch

- Many CPUs adopt a technique called the delayed branch to further reduce the stall
 - Delayed branch** always executes the next sequential instruction
 - The branch takes place after that one instruction delay
 - Delay slot** is the slot right after a delayed branch instruction



Delay Slot (Cont.)

- Compiler needs to schedule a useful instruction in the delay slot, or fills it up with `nop` (no operation)

```
// $s1 = a, $s2 = b, $s3 = c
// $t0 = d, $t1 = f
a = b + c;
if (d == 0) {f = f + 1;}
f = f + 2;
```



```
add $s1,$s2, $s3
bne $t0,$zero, L1
nop    //delay slot
addi $t1, $t1, 1
L1: addi $t1, $t1, 2
```

Can we do better?

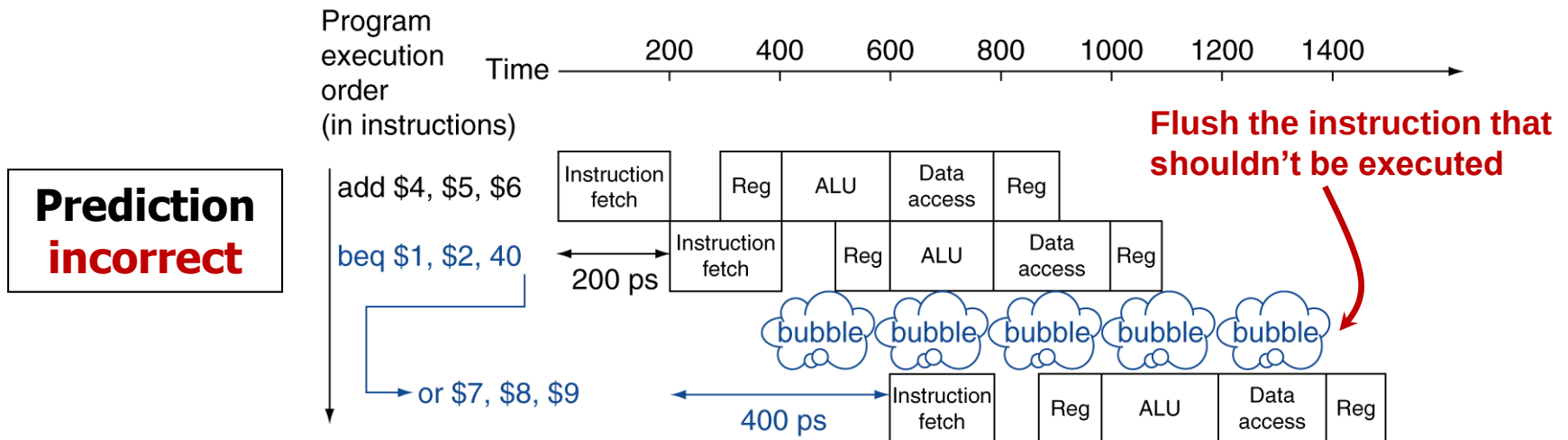
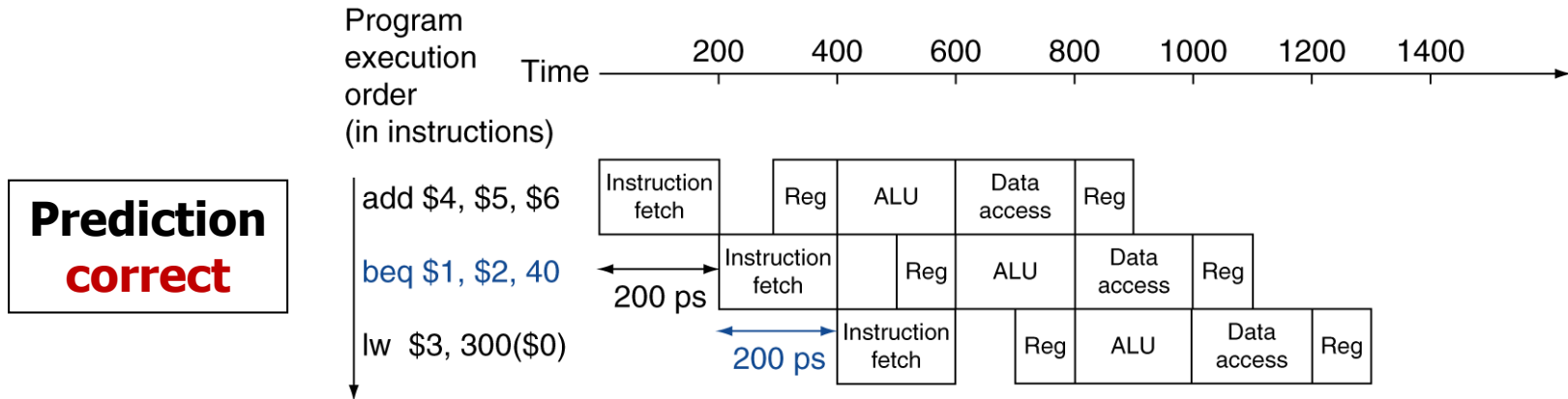
```
bne $t0, $zero, L1
add $s1,$s2,$s3 // delay slot
addi $t1, $t1, 1
L1: addi $t1, $t1, 2
```

Fill the delay slot with a useful and valid instruction

Branch Prediction

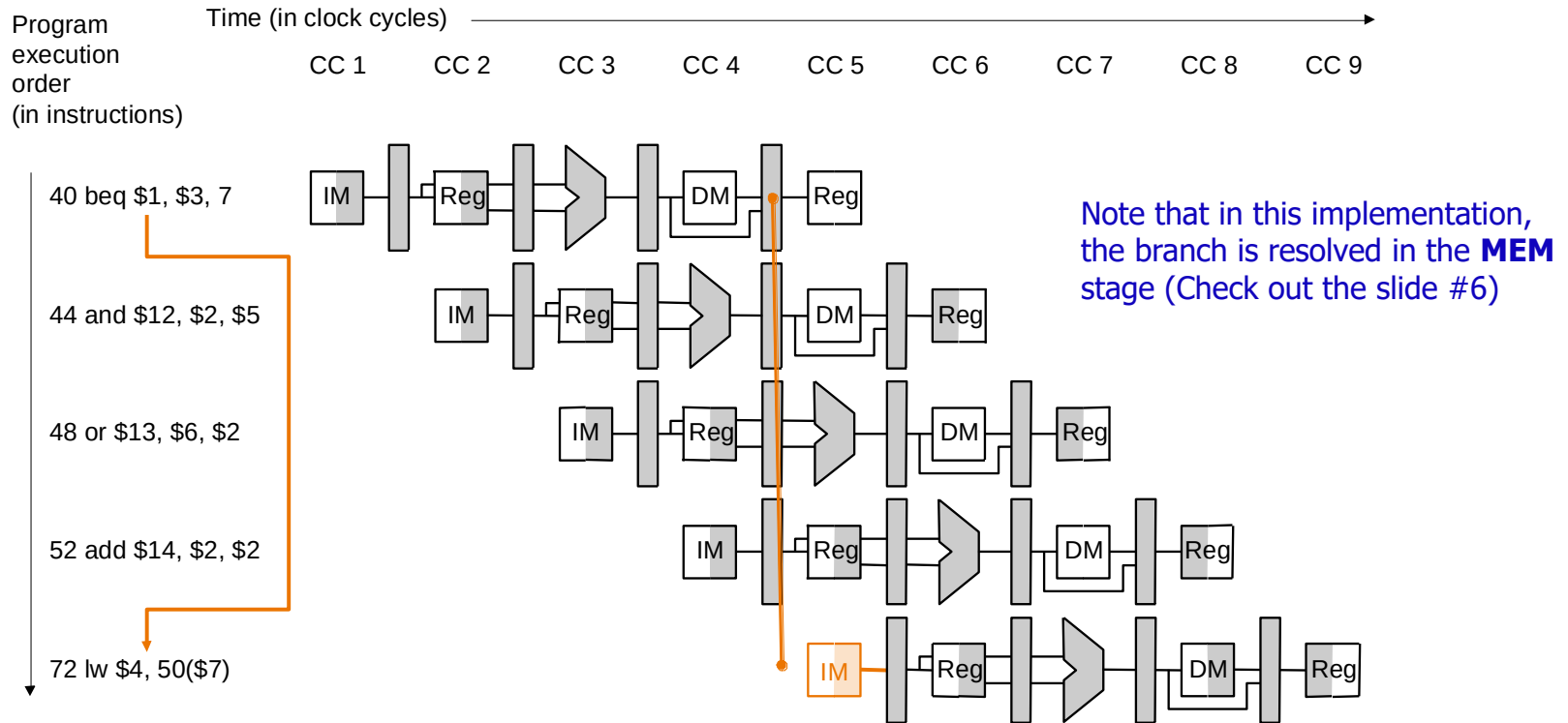
- Longer pipelines (implemented in Core 2 Duo, for example) can't readily determine branch outcome early
 - **Stall penalty** becomes unacceptable since branch instructions are used so frequently in the program
- **Solution: Branch Prediction**
 - Predict the branch outcome in hardware
 - Flush the instructions (that shouldn't have been executed) in the pipeline if the prediction turns out to be wrong
 - Modern processors use sophisticated branch predictors

MIPS with Predict-Not-Taken



Control Hazards - Branch

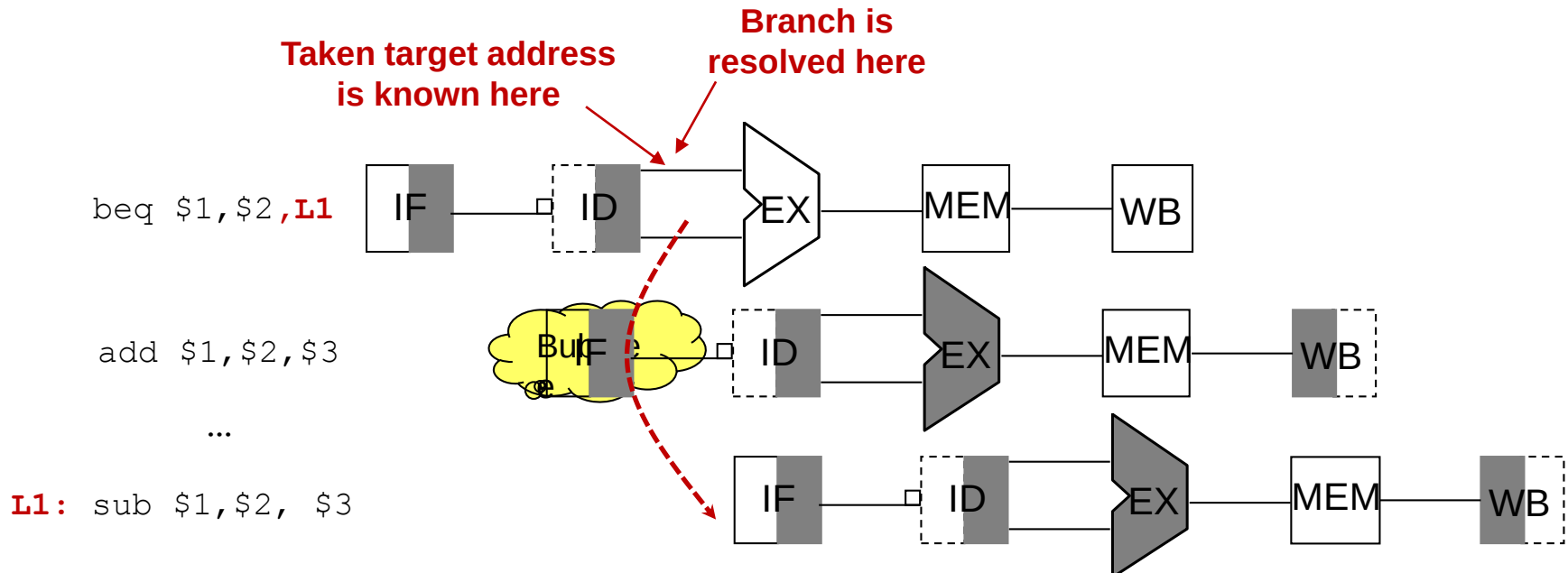
- When the branch condition is resolved, other instructions are in the pipeline



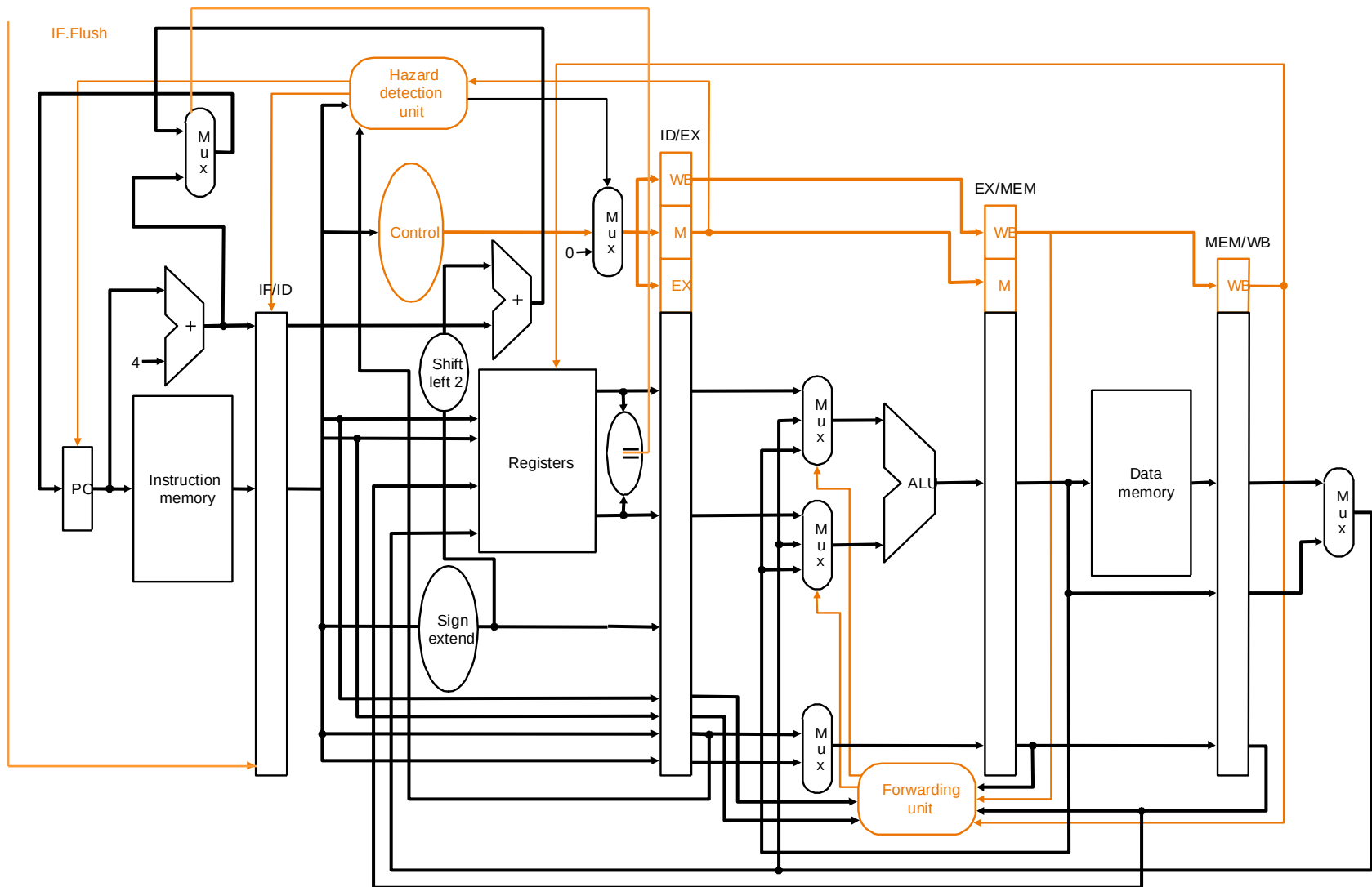
- We are predicting “branch not taken”
- If we are wrong (if branch is taken), flush instructions

Alleviate Branch Hazards

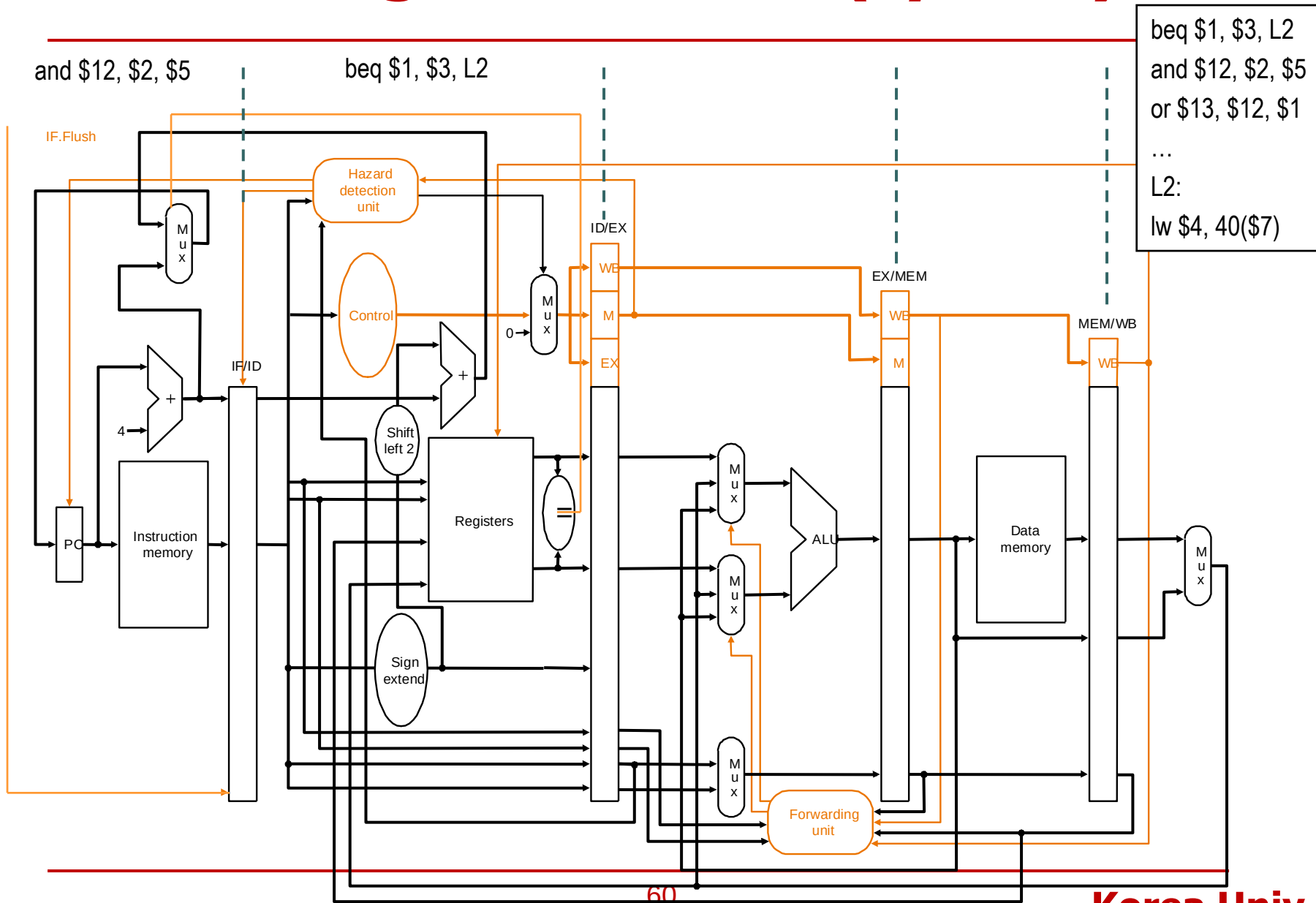
- Reduce penalty to 1 cycle
 - Move the branch compare to the ID stage of pipeline
 - Add an adder to calculate the branch target in ID stage
 - Add the IF.flush signal that zeros the instruction (or squash) in IF/ID pipeline register



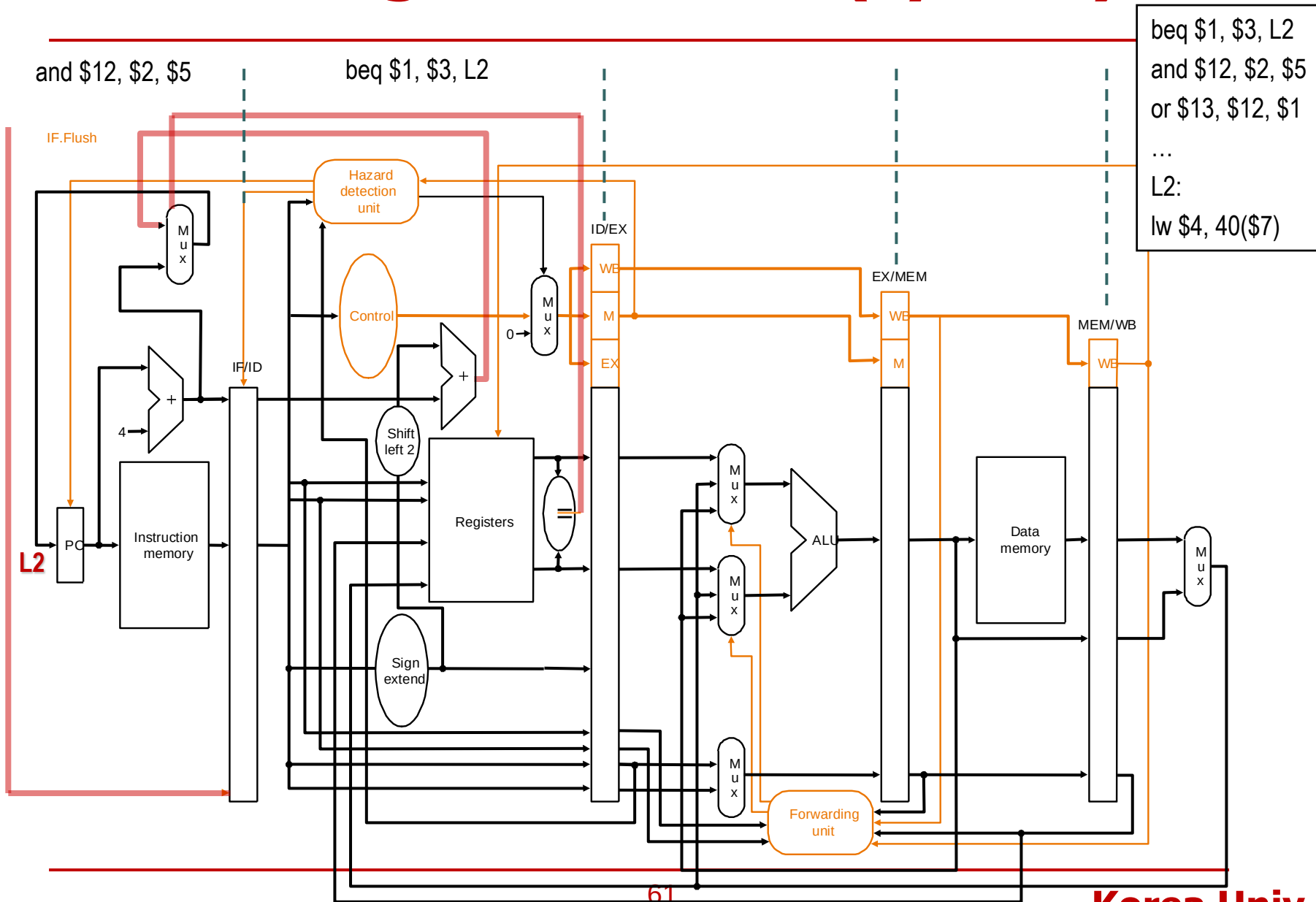
Flushing Instructions



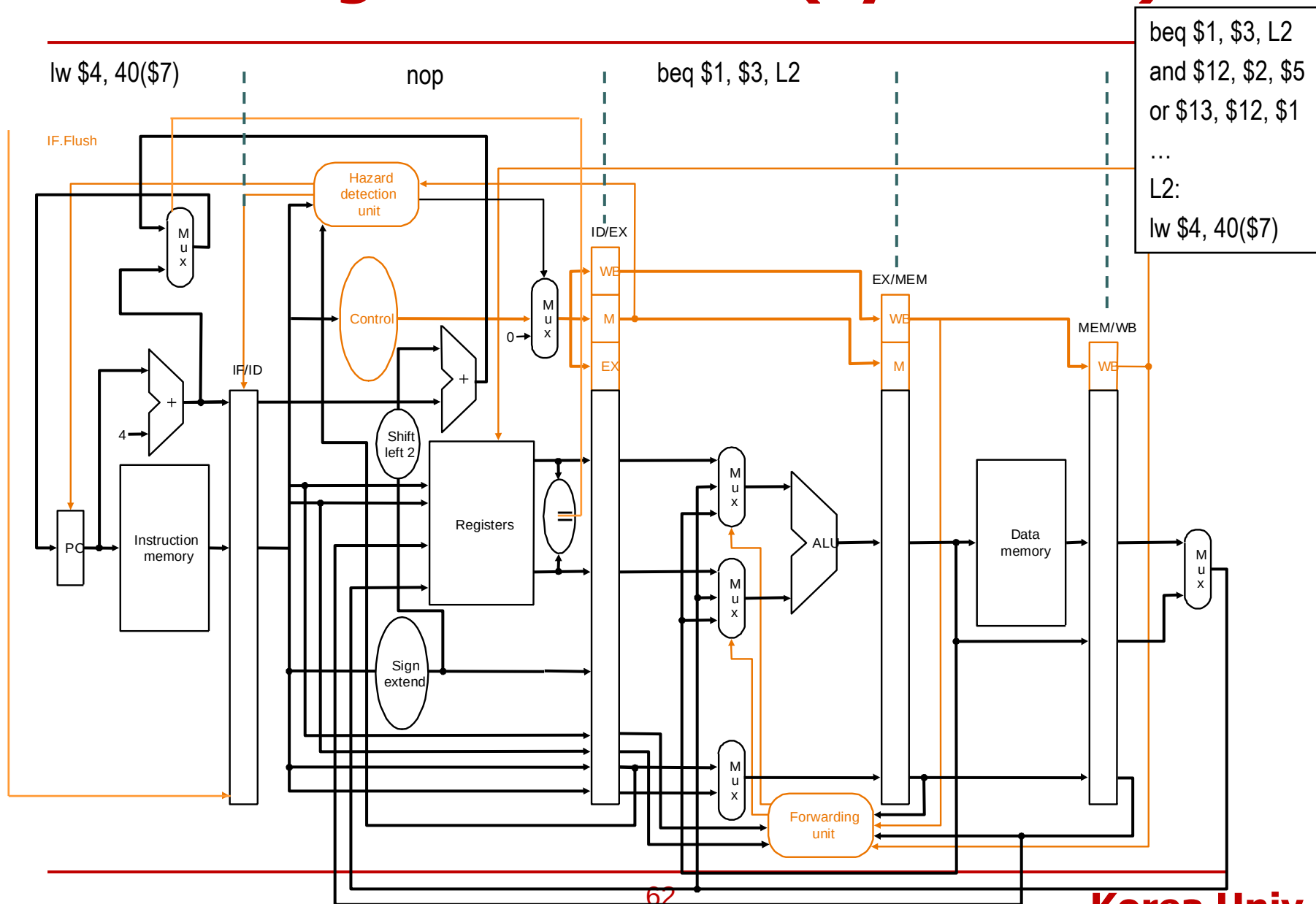
Flushing Instructions (cycle N)



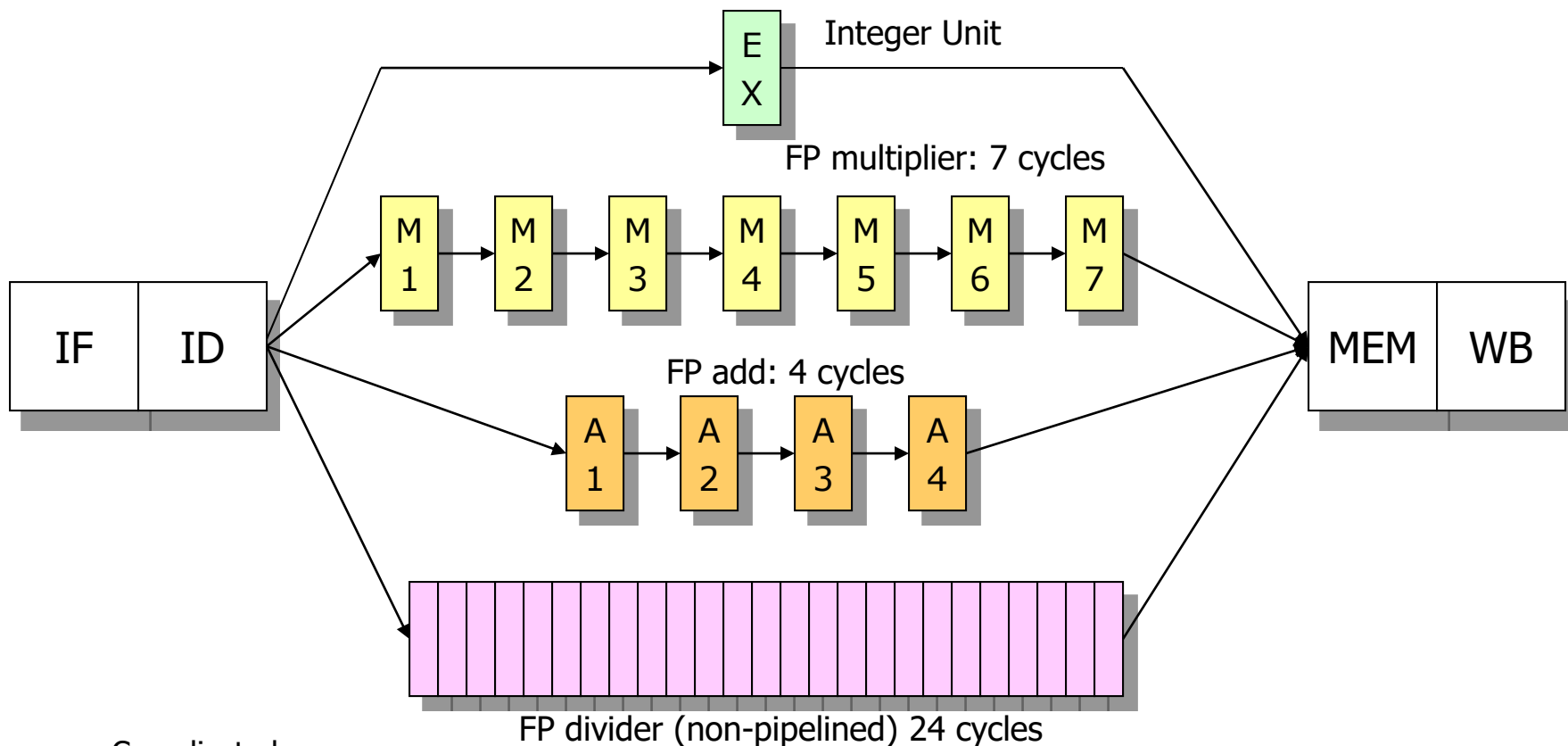
Flushing Instructions (cycle N)



Flushing Instructions (cycle N+1)

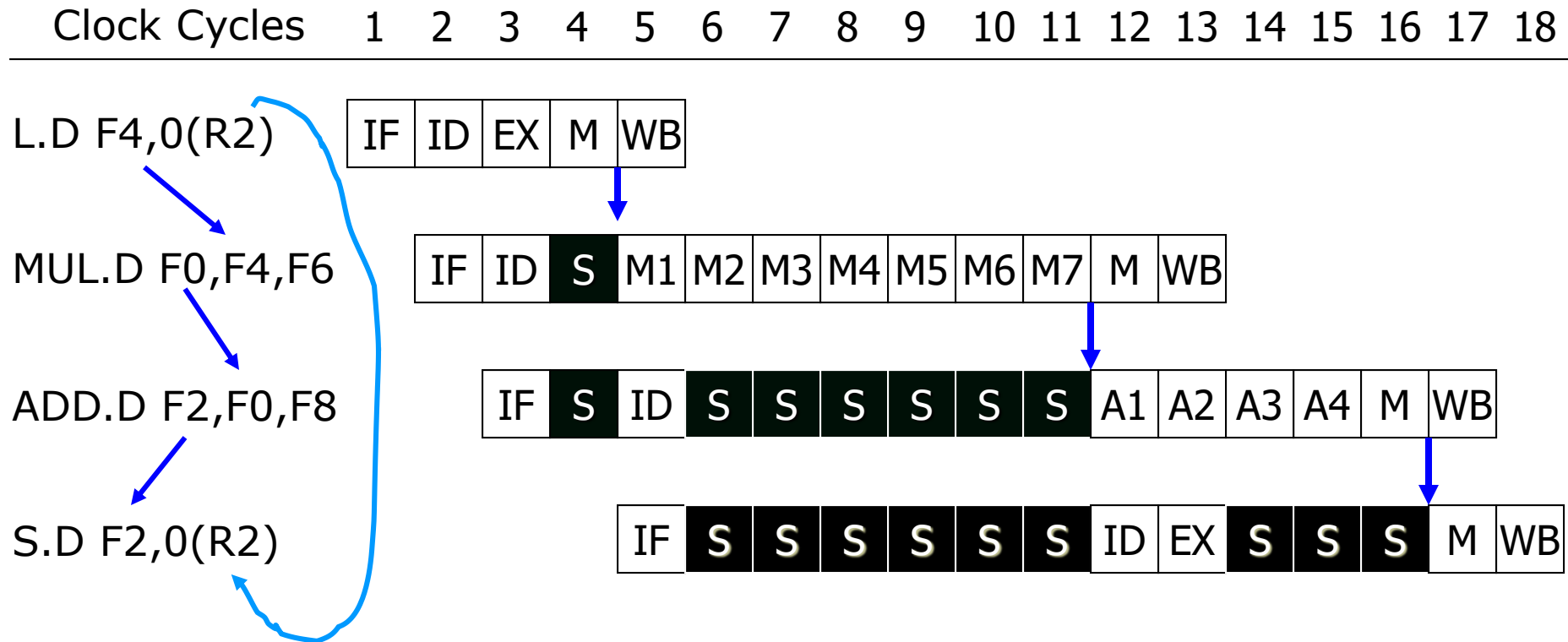


Supporting Multiple FP Operations

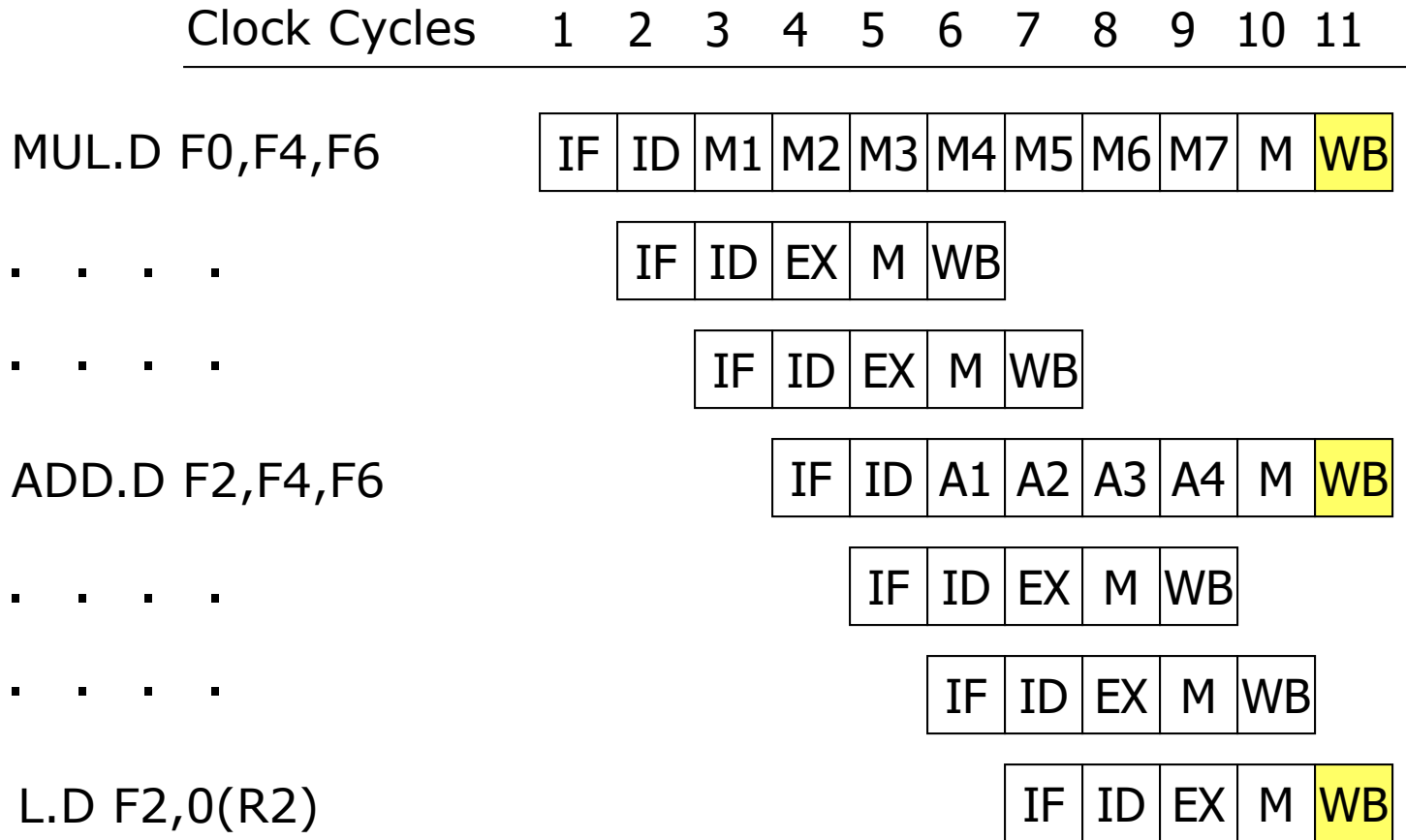


- Complicate bypass
- Potential structural hazard
- Multiple (FP) instructions can complete at the same time
 - RF might need to be multi-ported
 - Ordering issue, who gets to update the register?
- Out-of-order completion/retirement: Precise exception issue

Bypassing & Forwarding



Structural Hazards



- Write to register file at the same cycle (cc11)
- Write to the same register (WAW)
- MEM in cc10

Precise Exception Issue

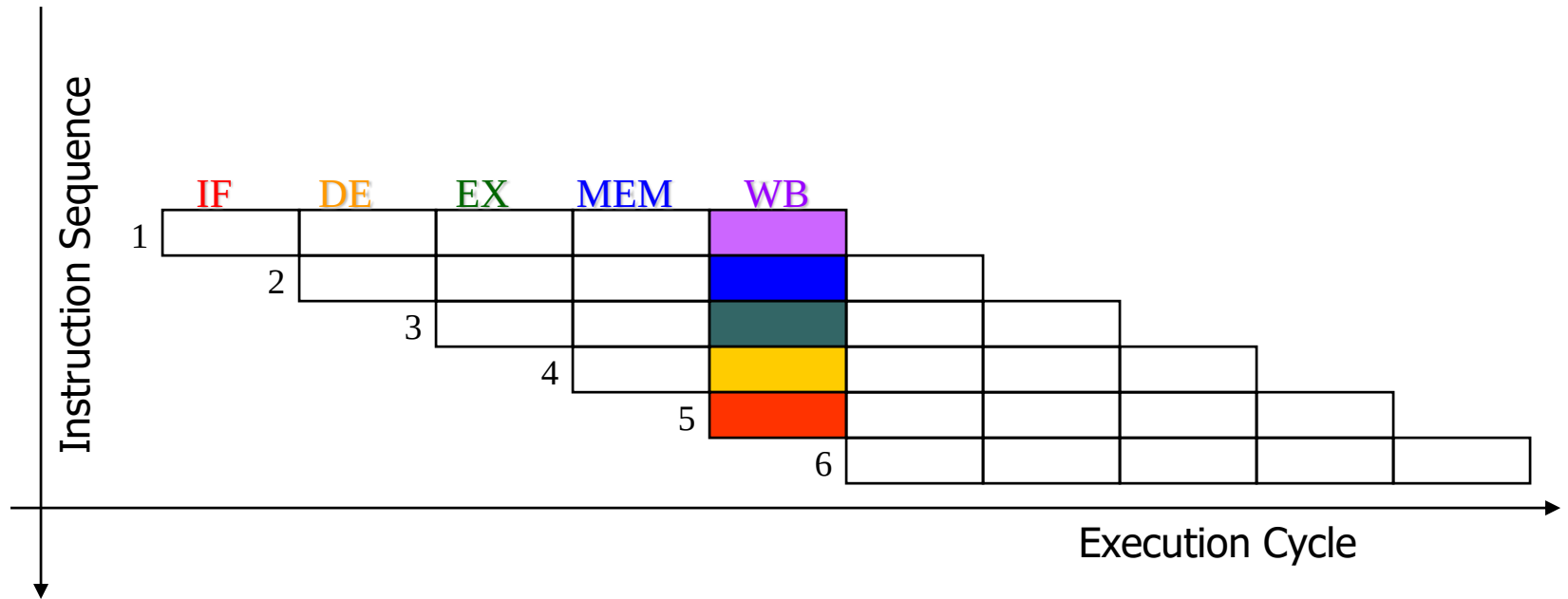
DIV.D F0, F2, F4 (exception!)

ADD.D F3, F10, F8 (completed)

SUB.D F12, F12, F14 (completed)

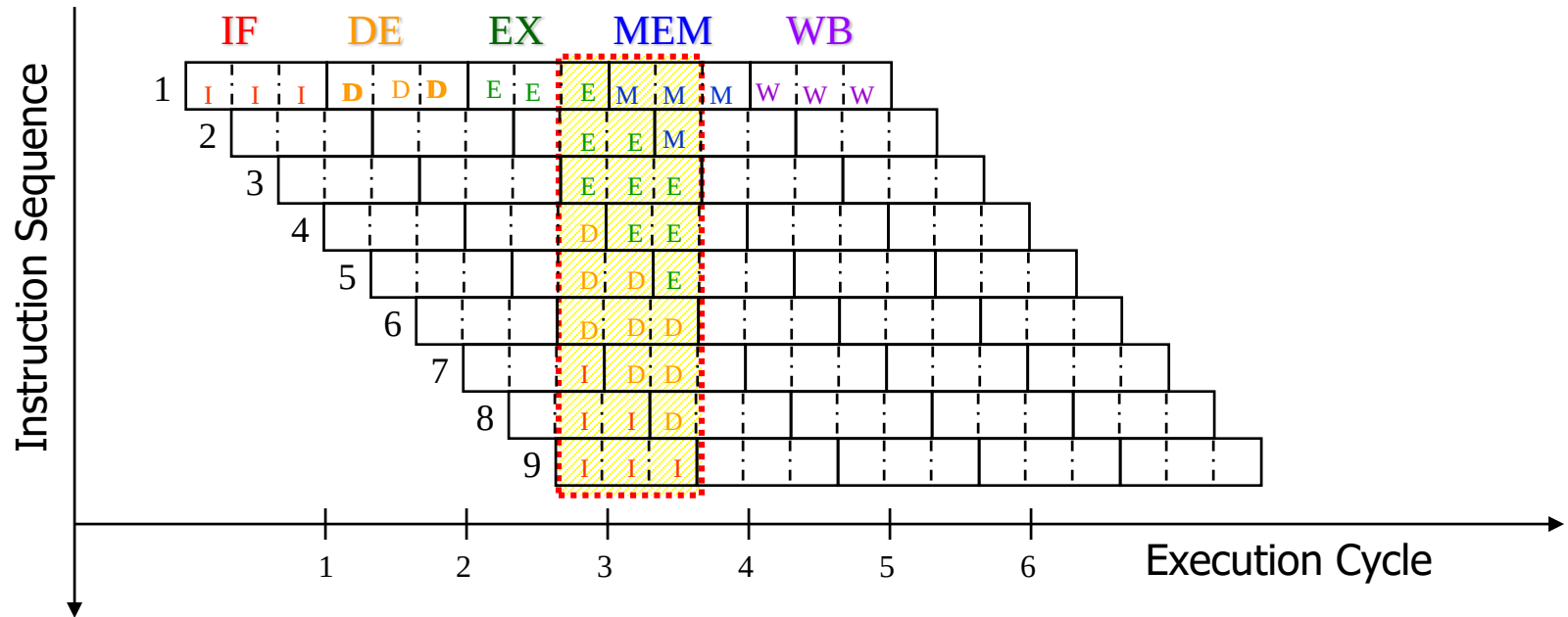
- Precise exception: If the pipeline can (or must) be stopped
 - All the instructions before the faulty (or intended) instruction must be completed
 - All the instructions after it must not be completed
 - Restart the execution from the faulty (or intended) instruction
- State must be consistent with the original program order
- Not straightforward with out-of-order completion
- Simple solution: Stalling until no exception of prior long-latency instruction is guaranteed
- Other modern solution: ROB (will dedicate a lecture to it)

Scalar Pipeline (Baseline)



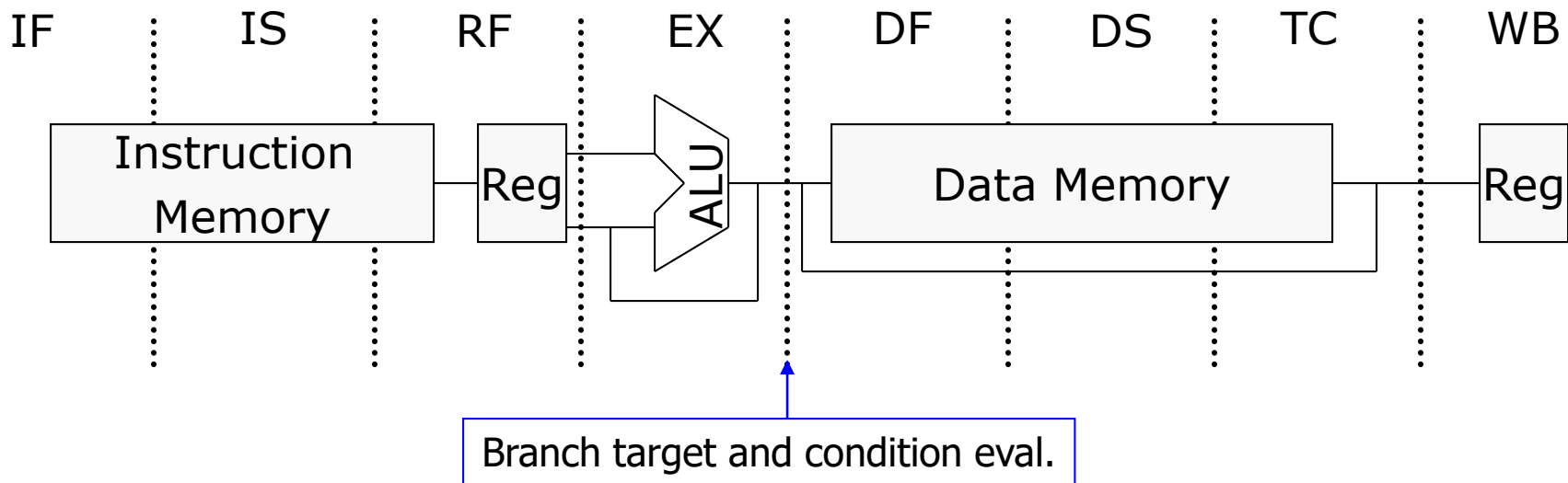
Superpipeline

- Deeper pipelining is called **superpipelining**
- Cache access is particularly time critical, so the extra pipeline stages come from decomposing the memory access
- Deeper pipeline allows for achieving higher clock rates

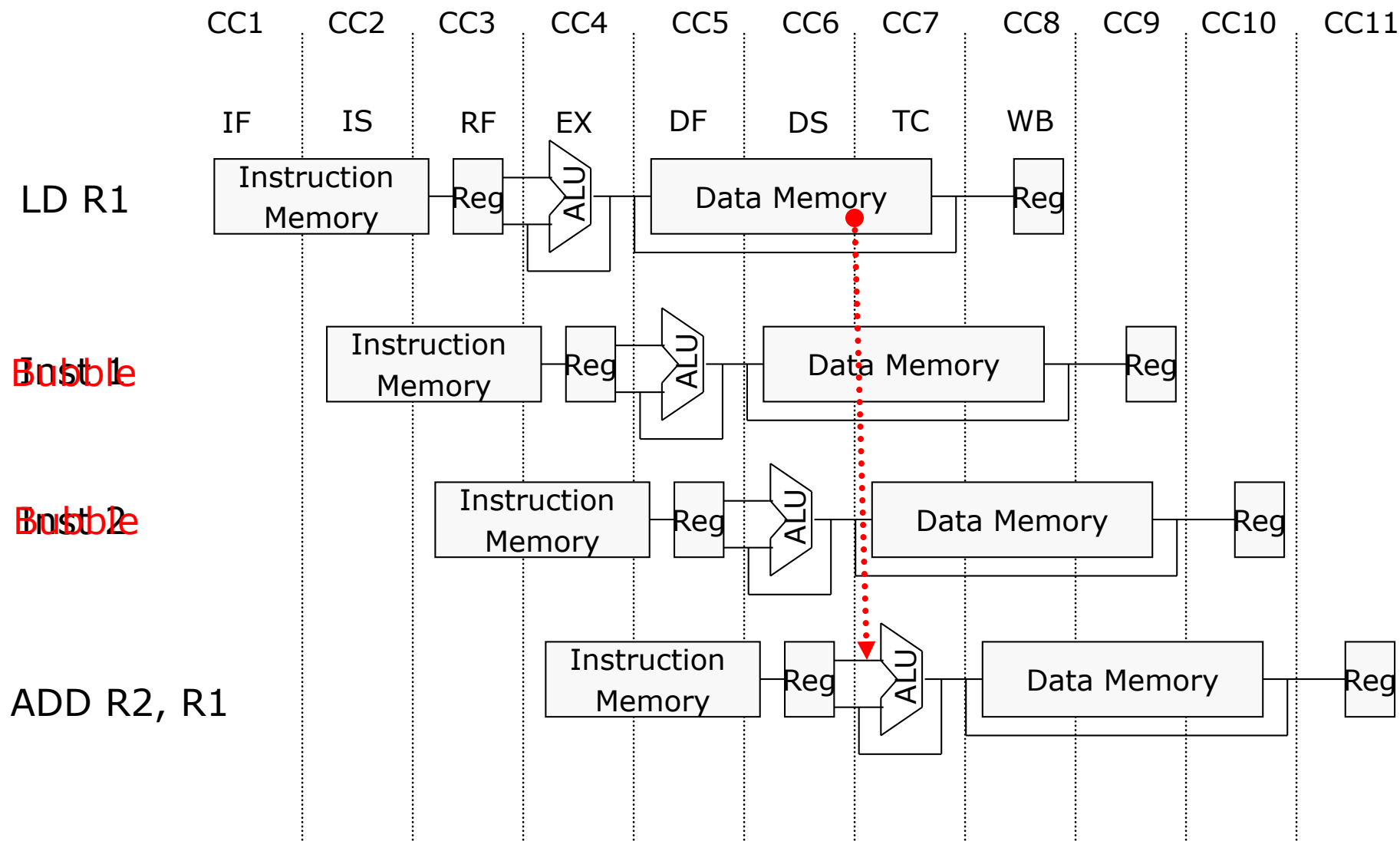


MIPS R4000 Pipeline

- Deeper Pipeline (superpipelining)
- 2 cycle delays for load
- Predicted-Not-Taken strategy
 - Not-taken (fall-through) branch : 1 delay slot
 - Taken branch: 1 delay slot + 2 idle cycles



Load delay (2 cycles)



Branches (Predicted-not-taken)

