Pipelining

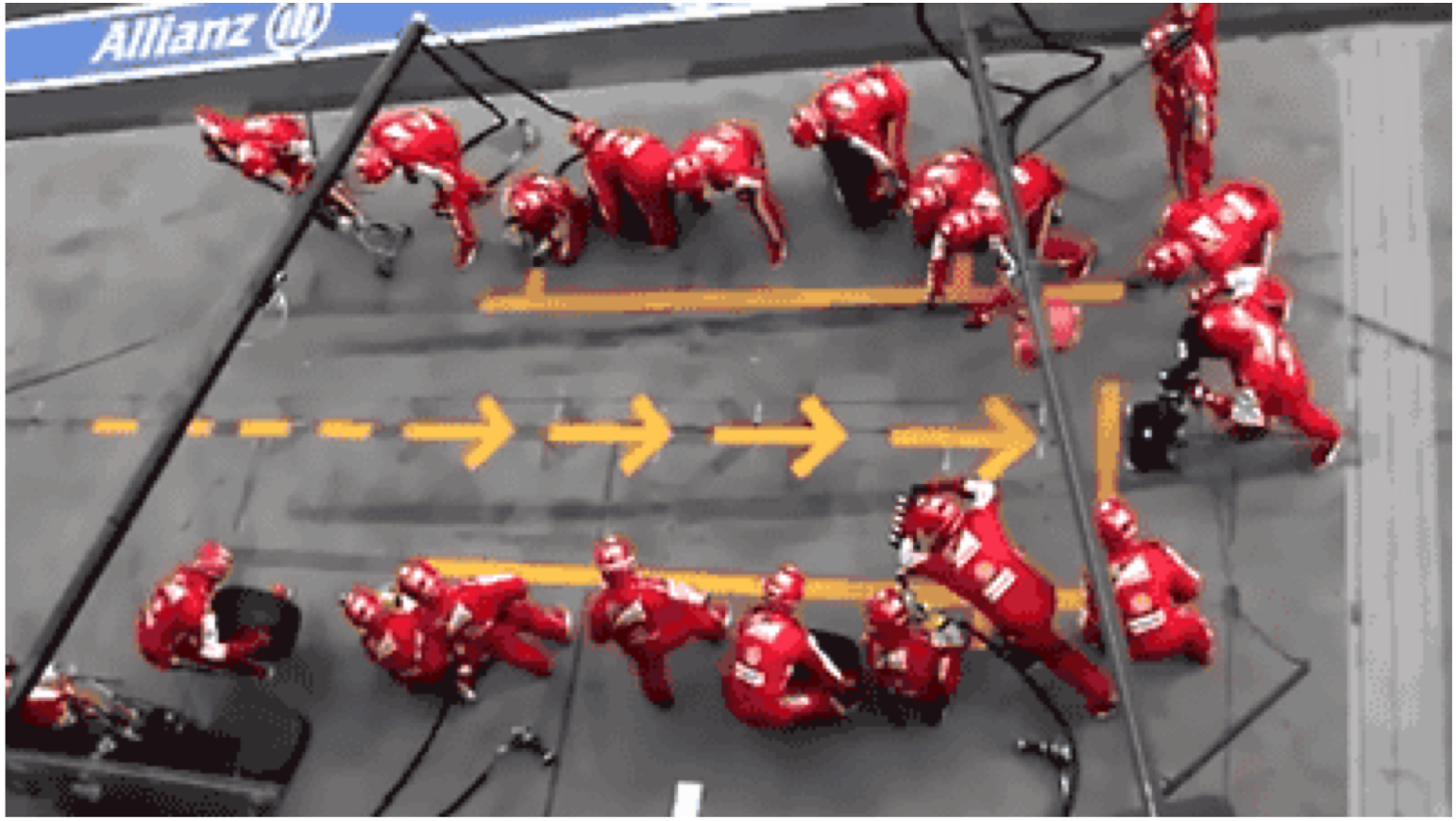
(Computer Architecture: Chapter 3 & Appendix C)

Yanyan Shen

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Parallel Processing



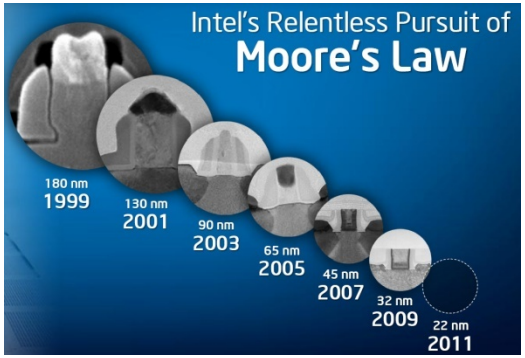
Outline

- C.1 Introduction to Pipelining
- How Pipeline is Implemented
- Pipeline Hazards
- Exceptions
- Handling Multicycle Operations

Processor Performance

- Performance of single-cycle processor is limited by the **long** critical path delay
  - The critical path limits the operating clock frequency

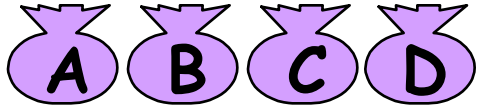
- **Can we do better?**
  - New semiconductor technology will reduce the critical path delay by manufacturing small-sized transistors
    - Core 2 Duo is manufactured with 65nm technology
    - Core i7 is manufactured with 45nm technology
    - Next semiconductor technology is 32nm technology

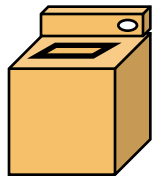


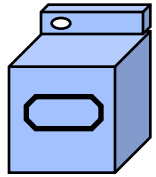
  - **Can we increase the processor performance with a different microarchitecture?**
    - **Yes! Pipelining**

Revisiting Performance

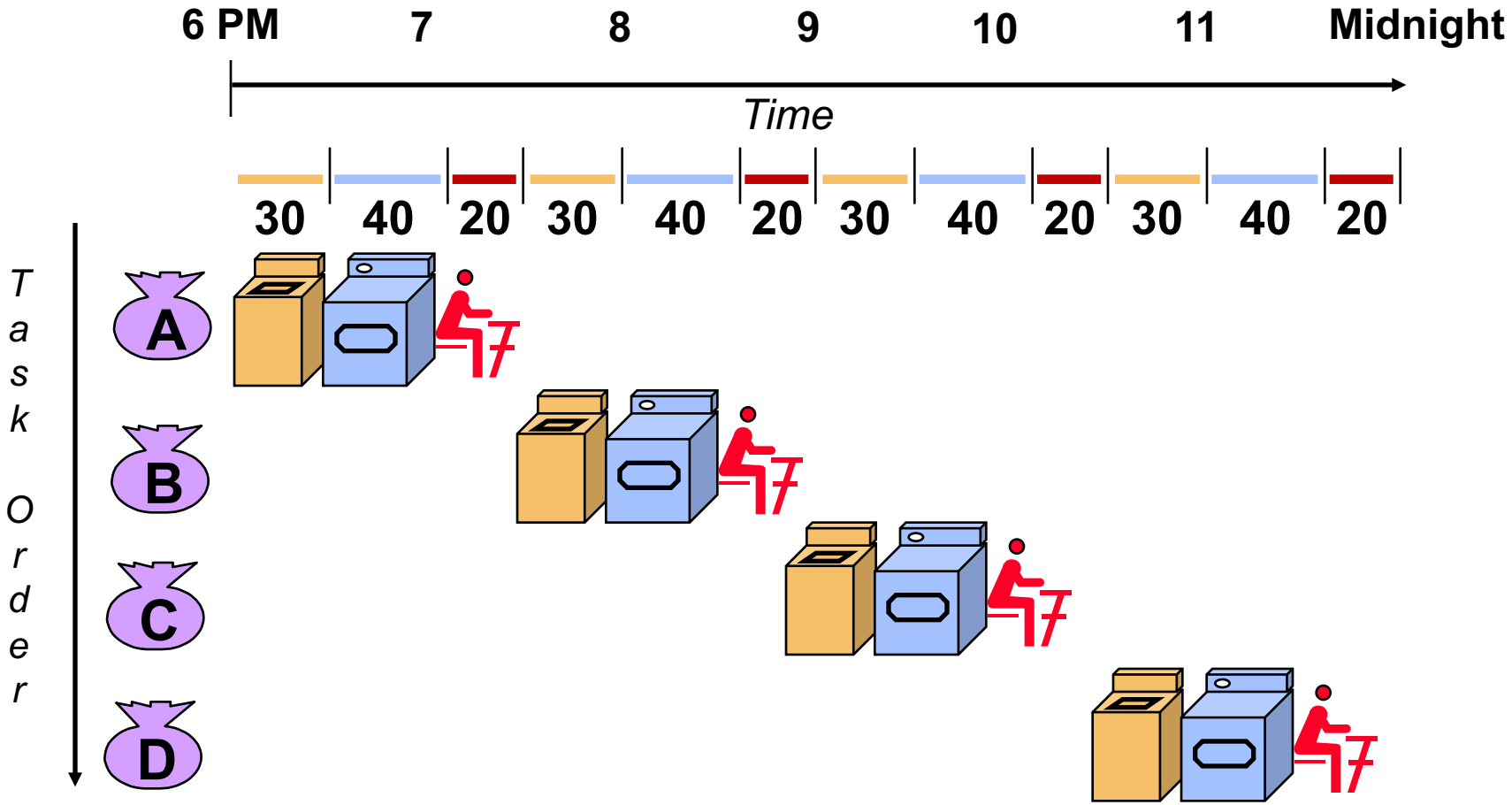
- Laundry Example
  - Ann, Brian, Cathy, Dave each has one load of clothes to wash, dry, and fold
  - **Washer** takes 30 minutes
  - **Dryer** takes 40 minutes
  - **Folder** takes 20 minutes





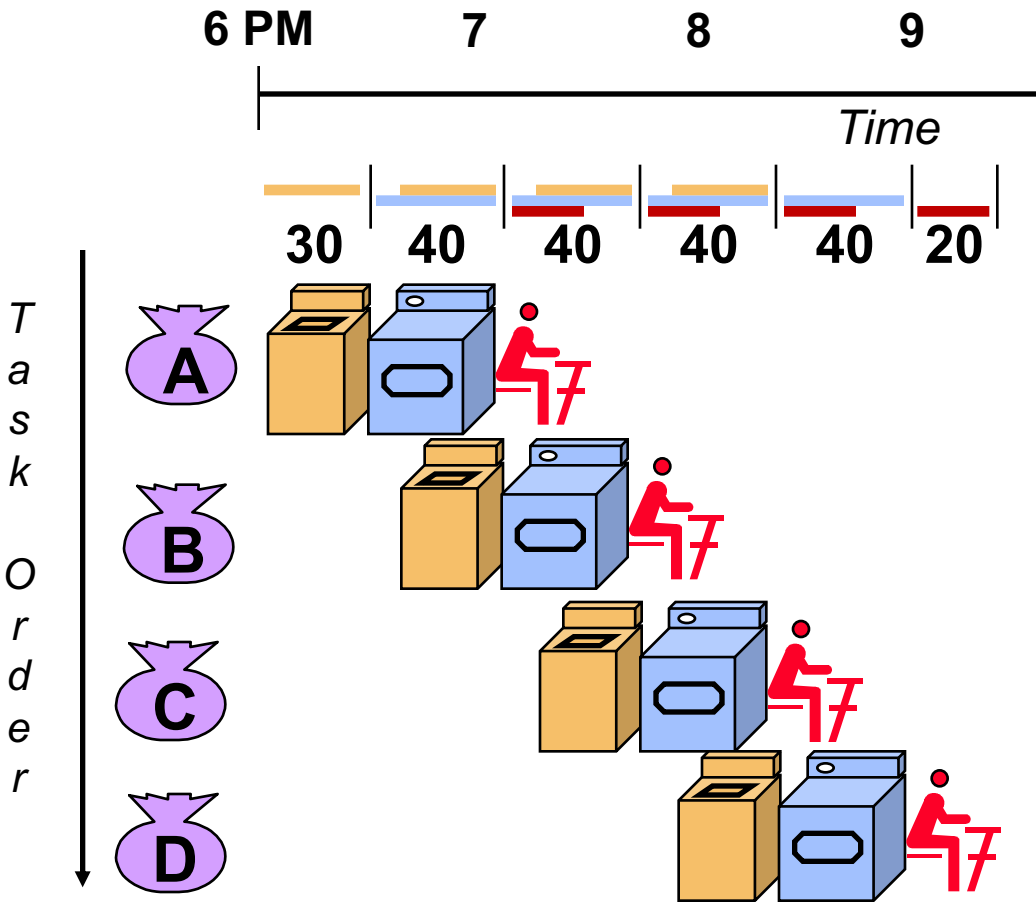


Sequential Laundry



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

Pipelined Laundry



Notes

- Pipelining doesn't help latency (response time) of a single task
- Pipelining helps throughput of entire workload
- Multiple tasks operating simultaneously

Potential speedup = # of pipeline stages

Unbalanced lengths of pipeline stages reduce speedup

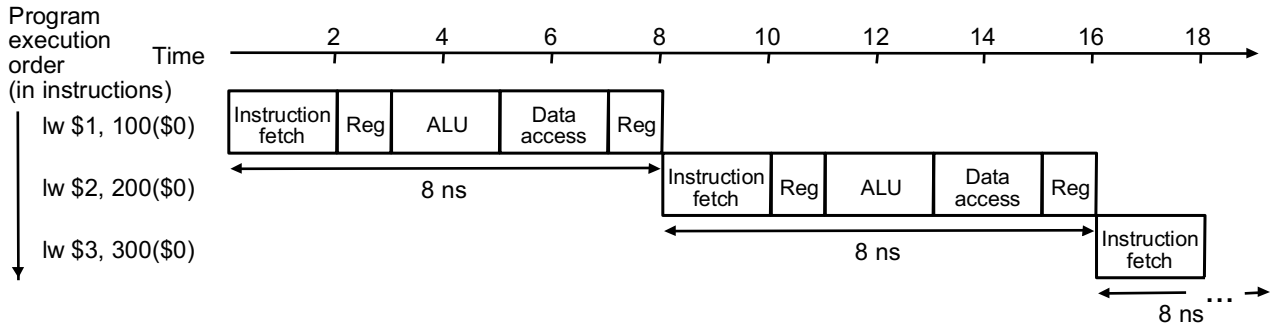
- Response time: 90 mins
- Throughput: 1.14 tasks / hr (= 52.5 mins/task, 3.5 hours for 4 loads)

Pipelining

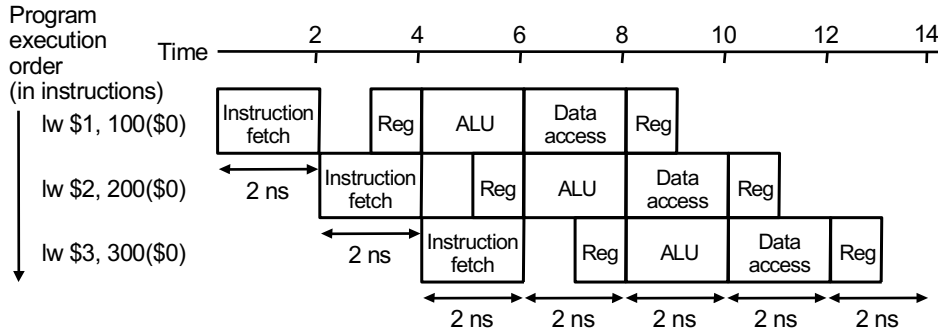
Improve performance by increasing instruction **throughput**

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

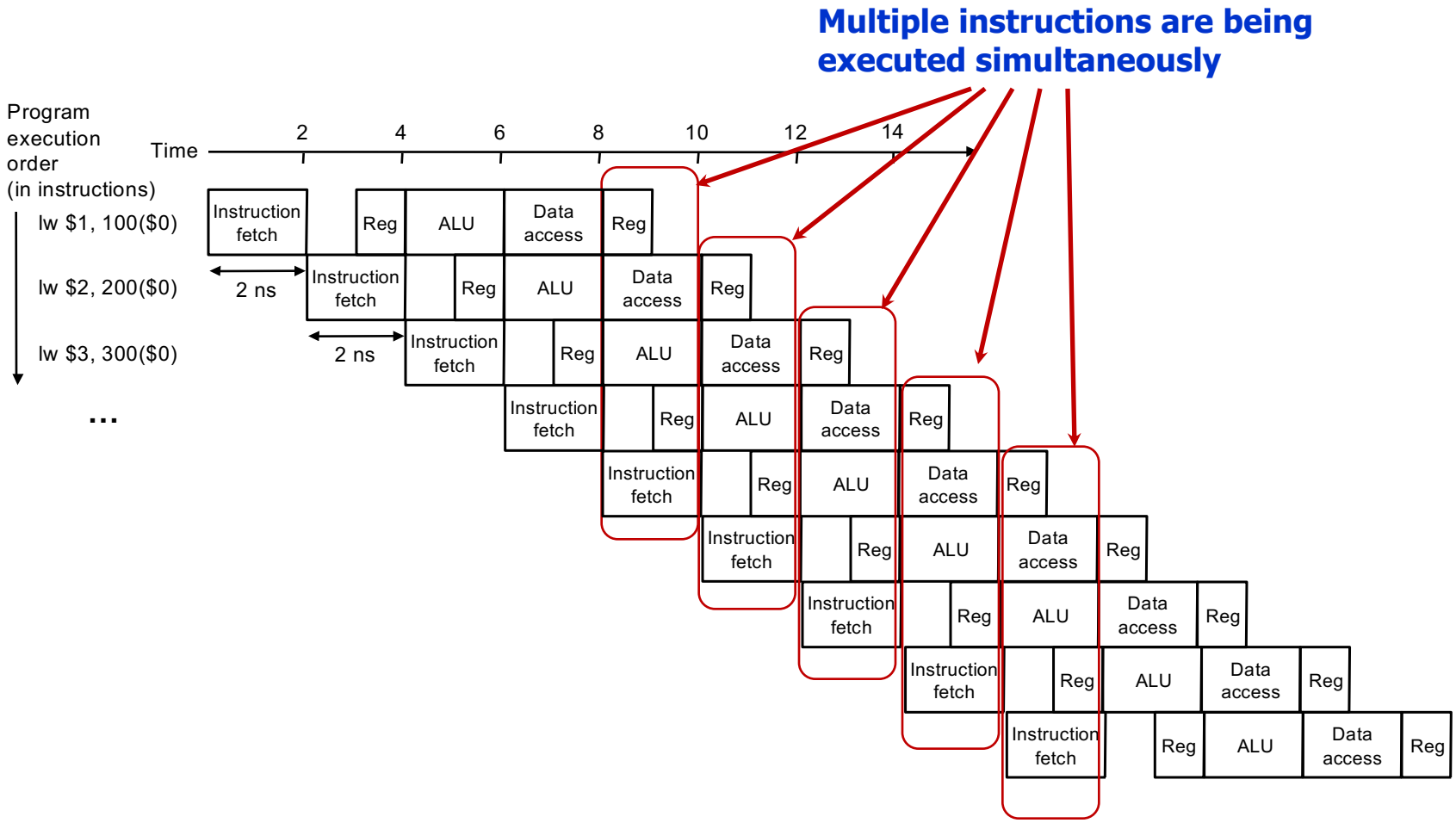
Sequential Execution



Pipelined Execution



Pipelining (Cont.)



Pipelining (Cont.)

Pipeline Speedup

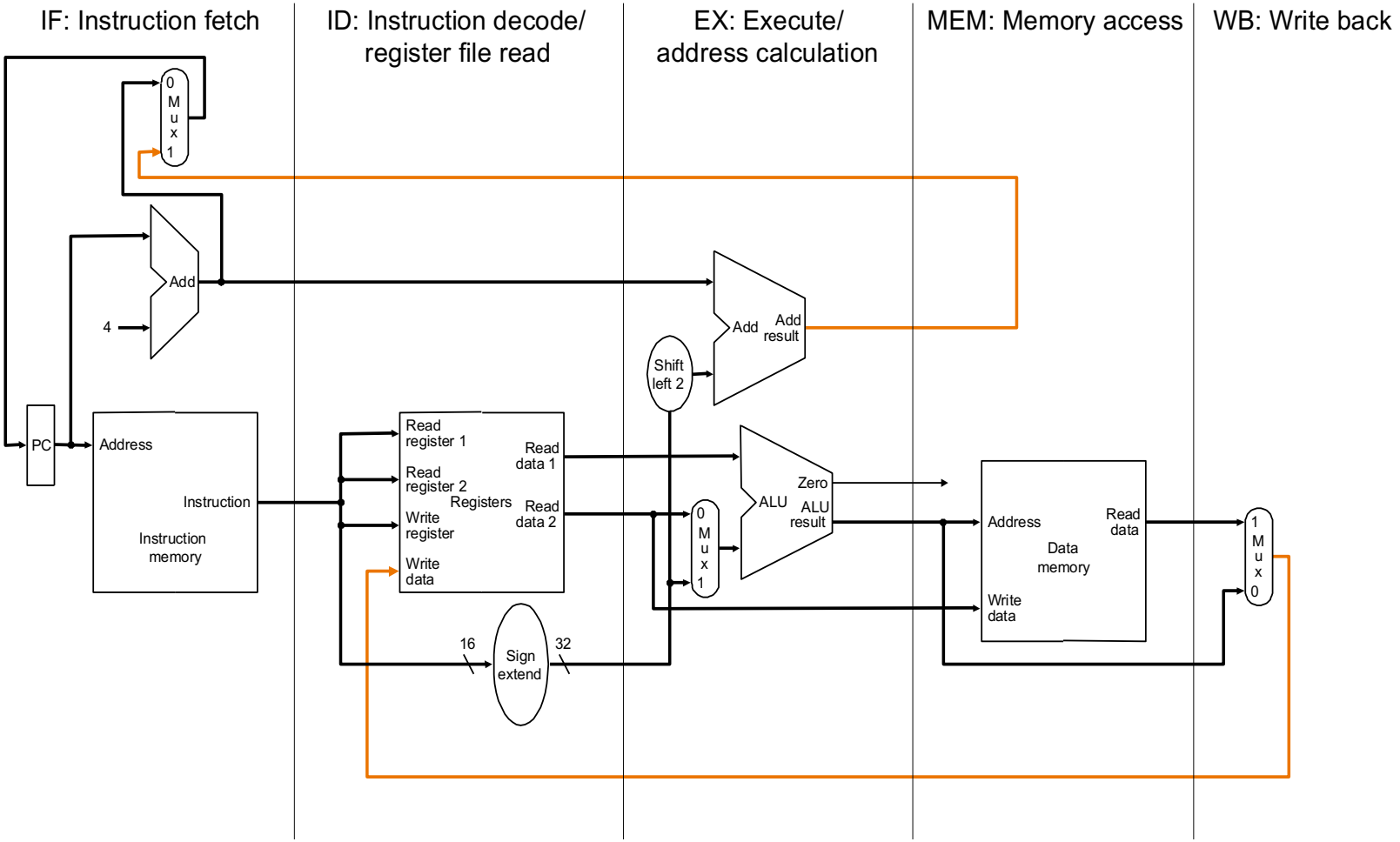
$$\text{Time to execute an instruction}_{\text{pipeline}} = \frac{\text{Time to execute an instruction}_{\text{sequential}}}{\text{Number of stages}}$$

- If not balanced, speedup is less
- Speedup comes from increased **throughput**
- the **latency** of instruction does not decrease

Outline

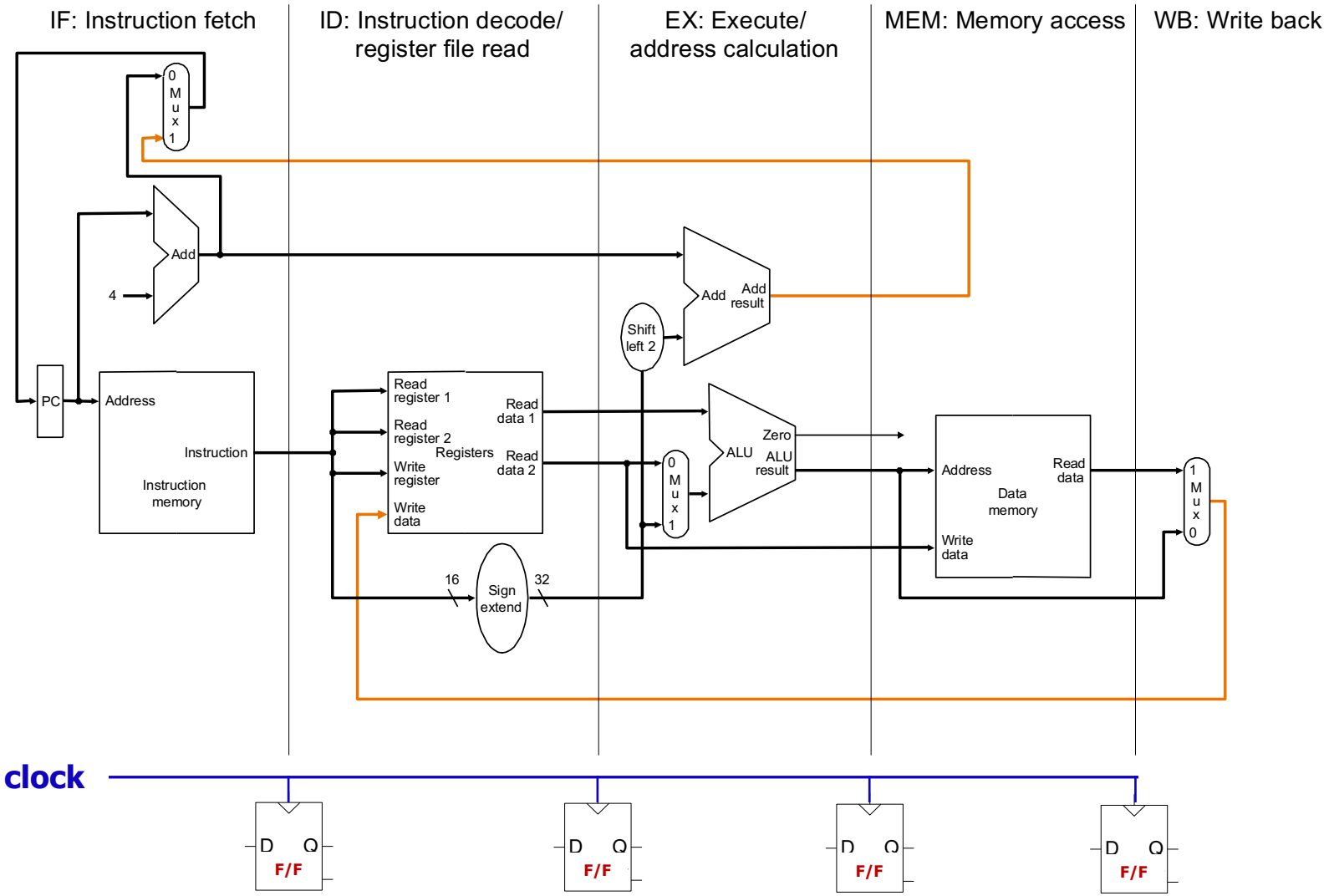
- Introduction to Pipelining
- C.2 How Pipeline is Implemented
- Pipeline Hazards
- Exceptions
- Handling Multicycle Operations

Basic Idea

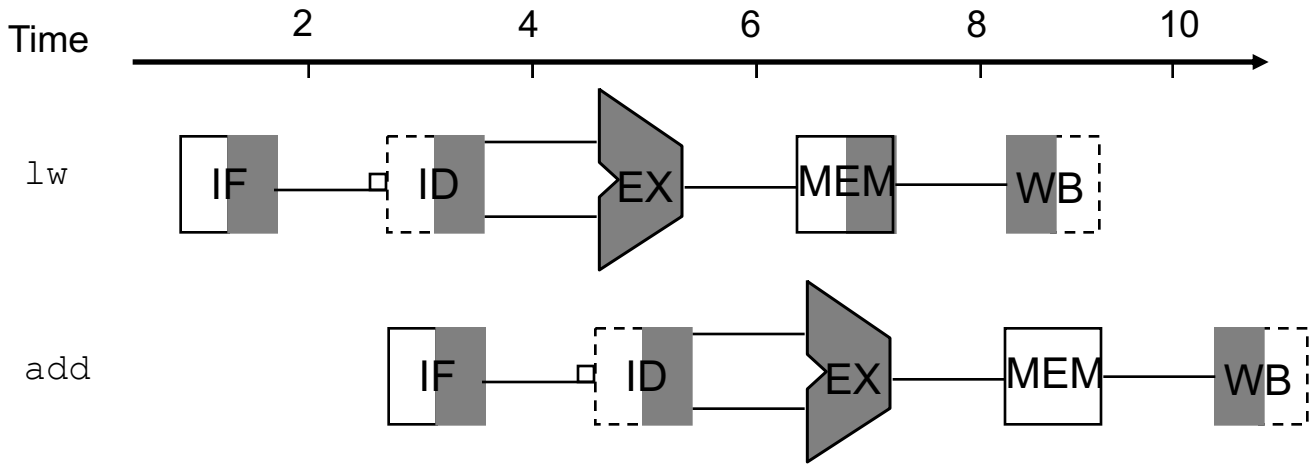


What do we have to add to actually split the datapath into stages?

Basic Idea

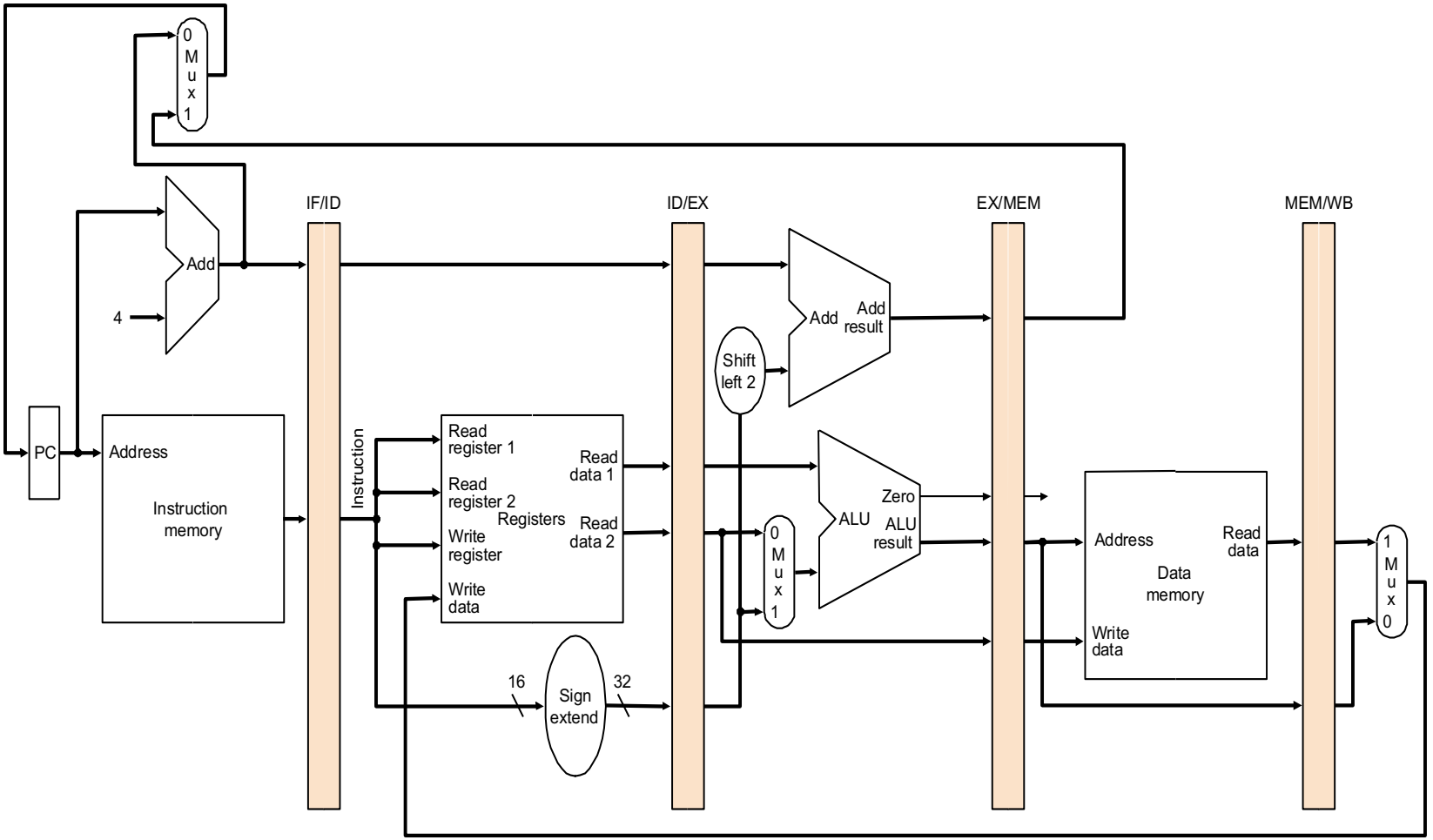

IF: Instruction fetch
ID: Instruction decode/ register file read
EX: Execute/ address calculation
MEM: Memory access
WB: Write back
0 M u x 1
Add
4
PC
Address
Instruction
Instruction memory
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
1 M u x 0
16
Sign extend
32
clock
D Q
F/F

Graphically Representing Pipelines

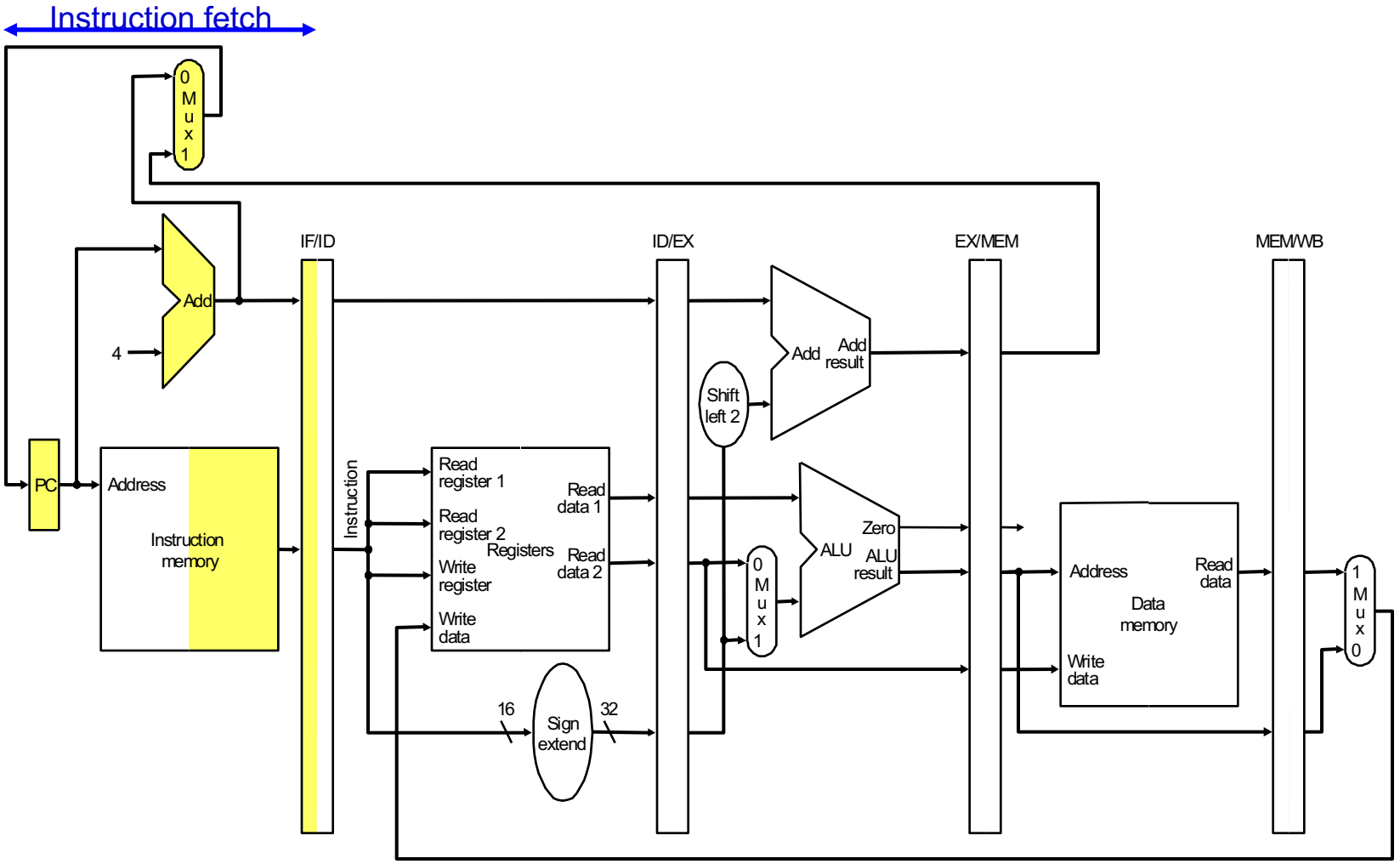


- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

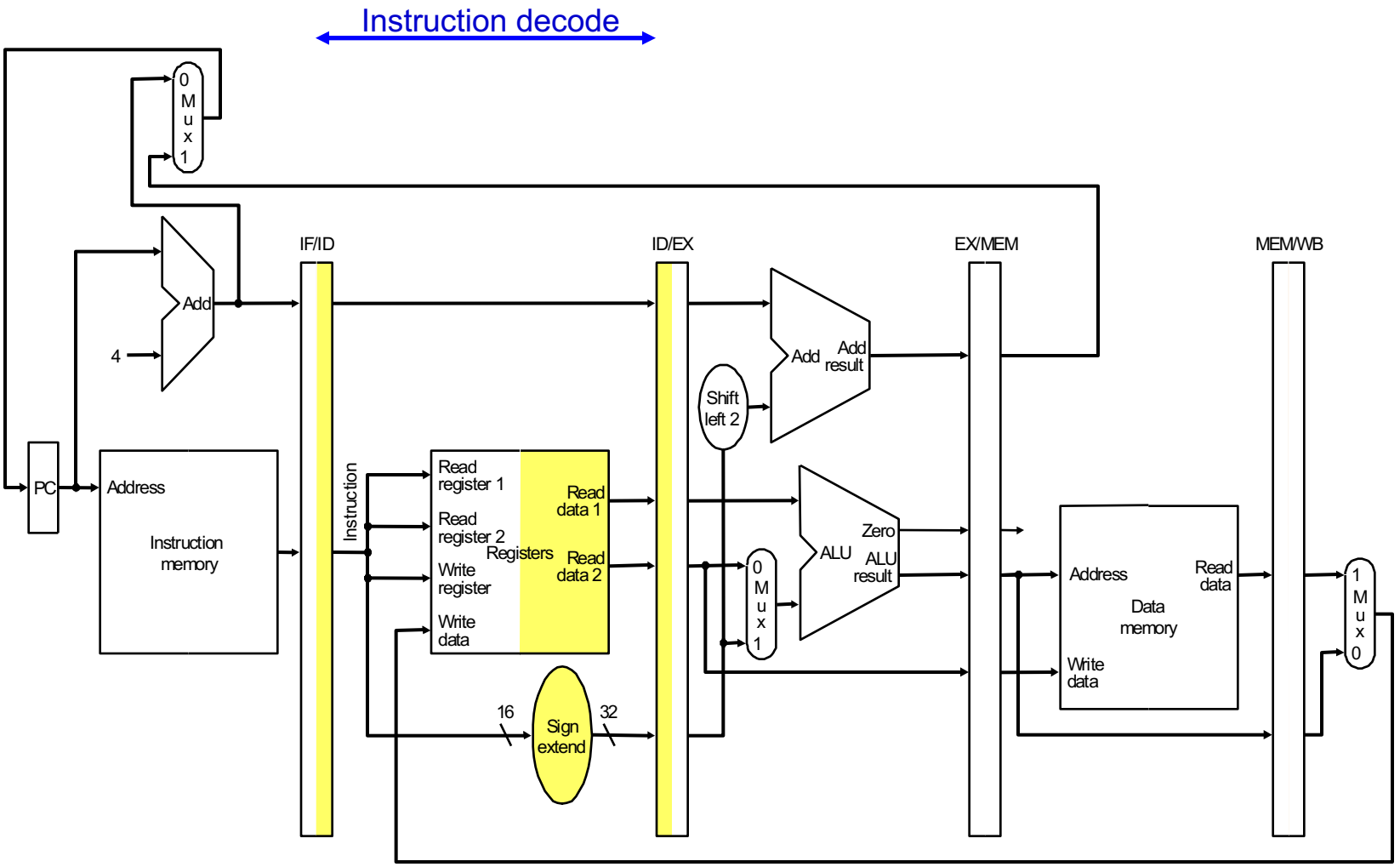
Pipelined Datapath



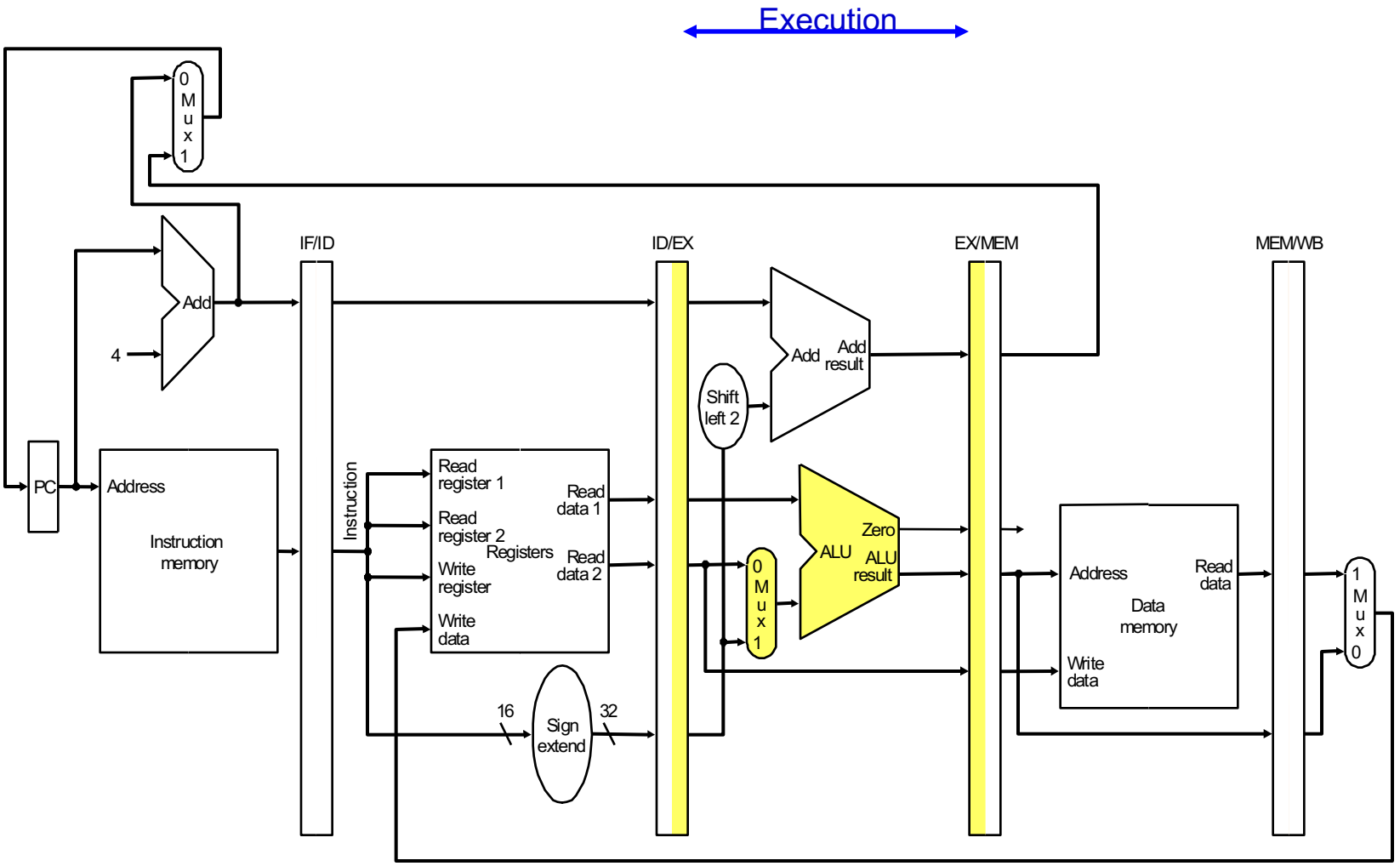
`lw`: Instruction Fetch (IF)



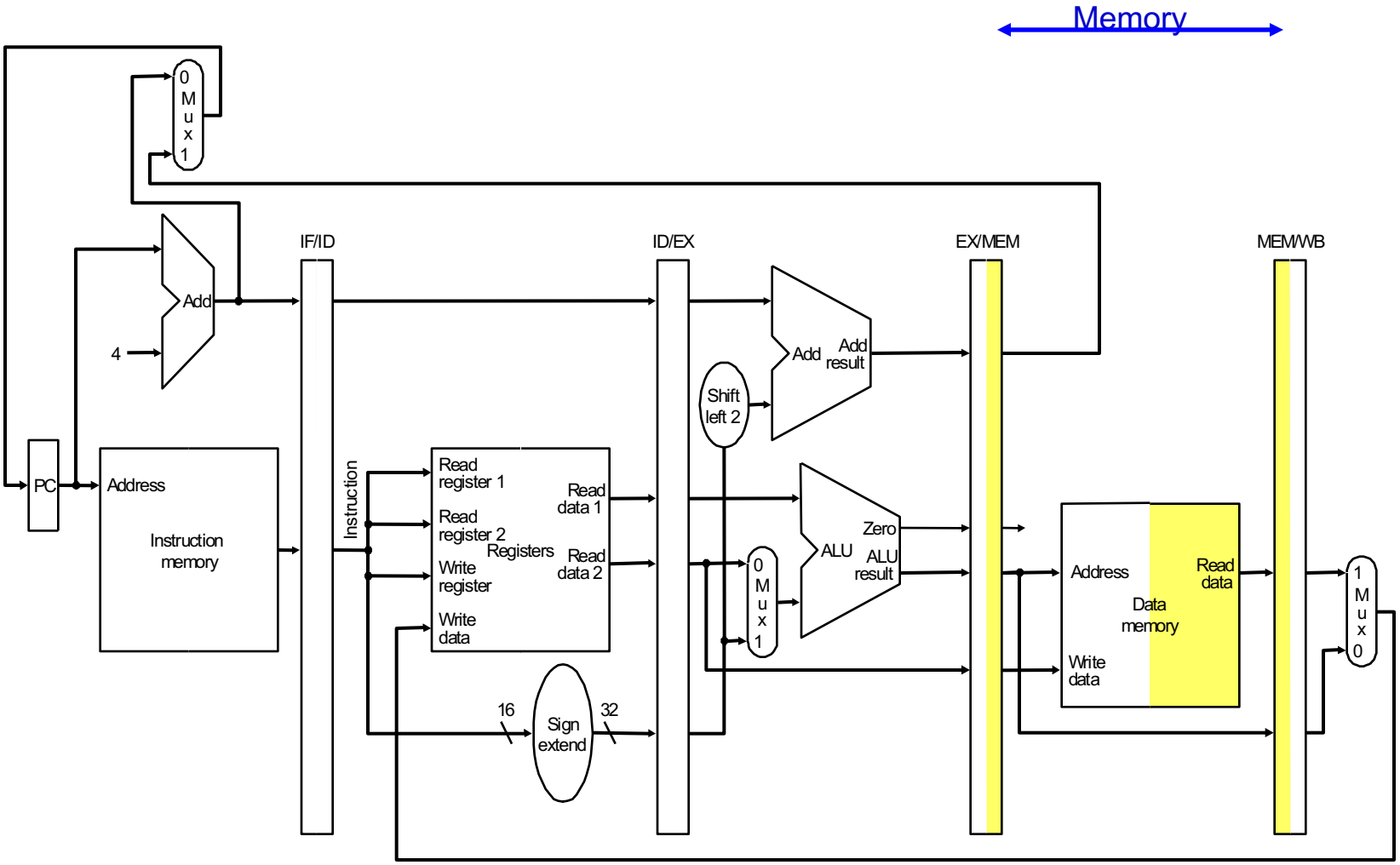
`lw`: Instruction Decode (ID)



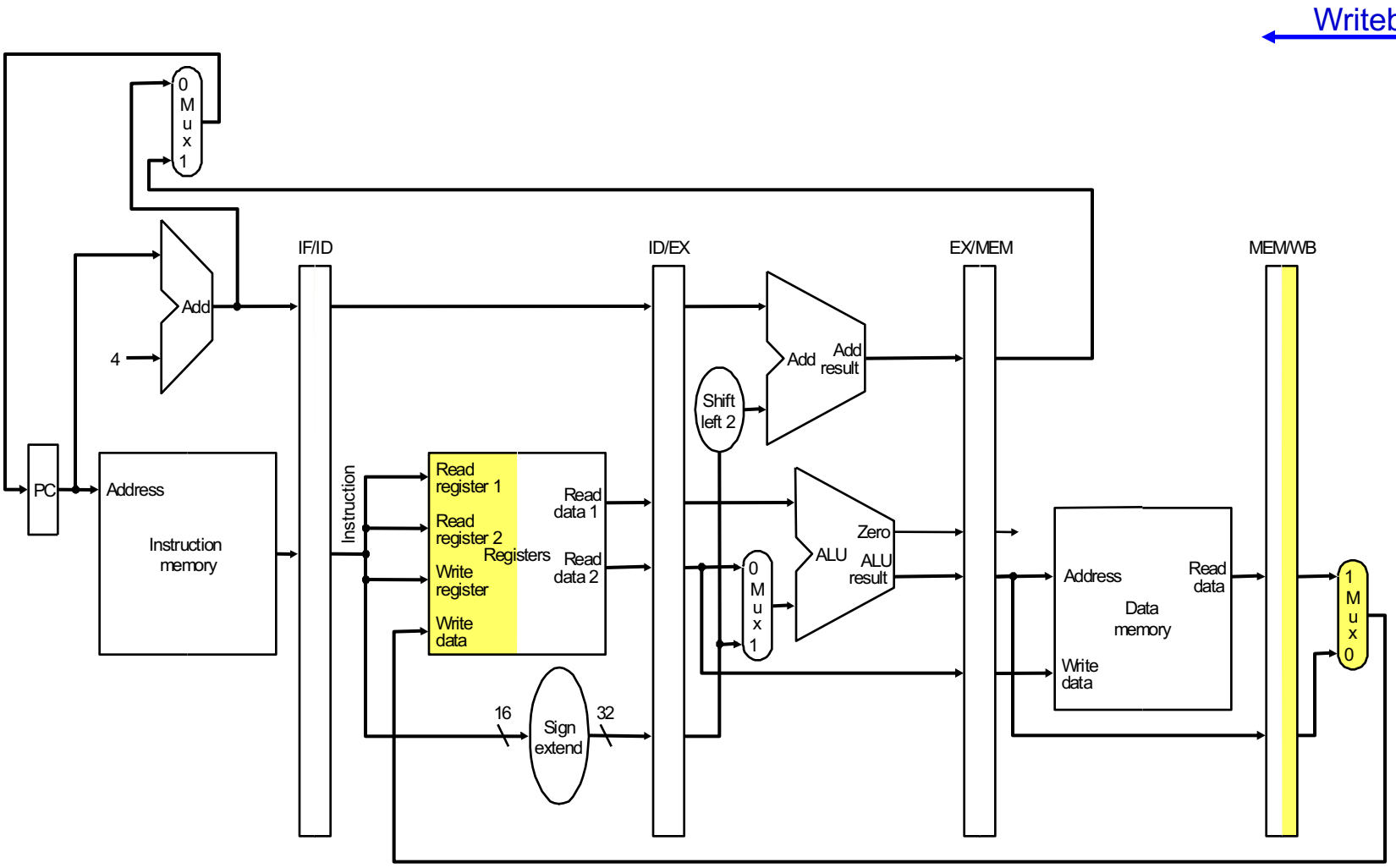
lw: Execution (EX)



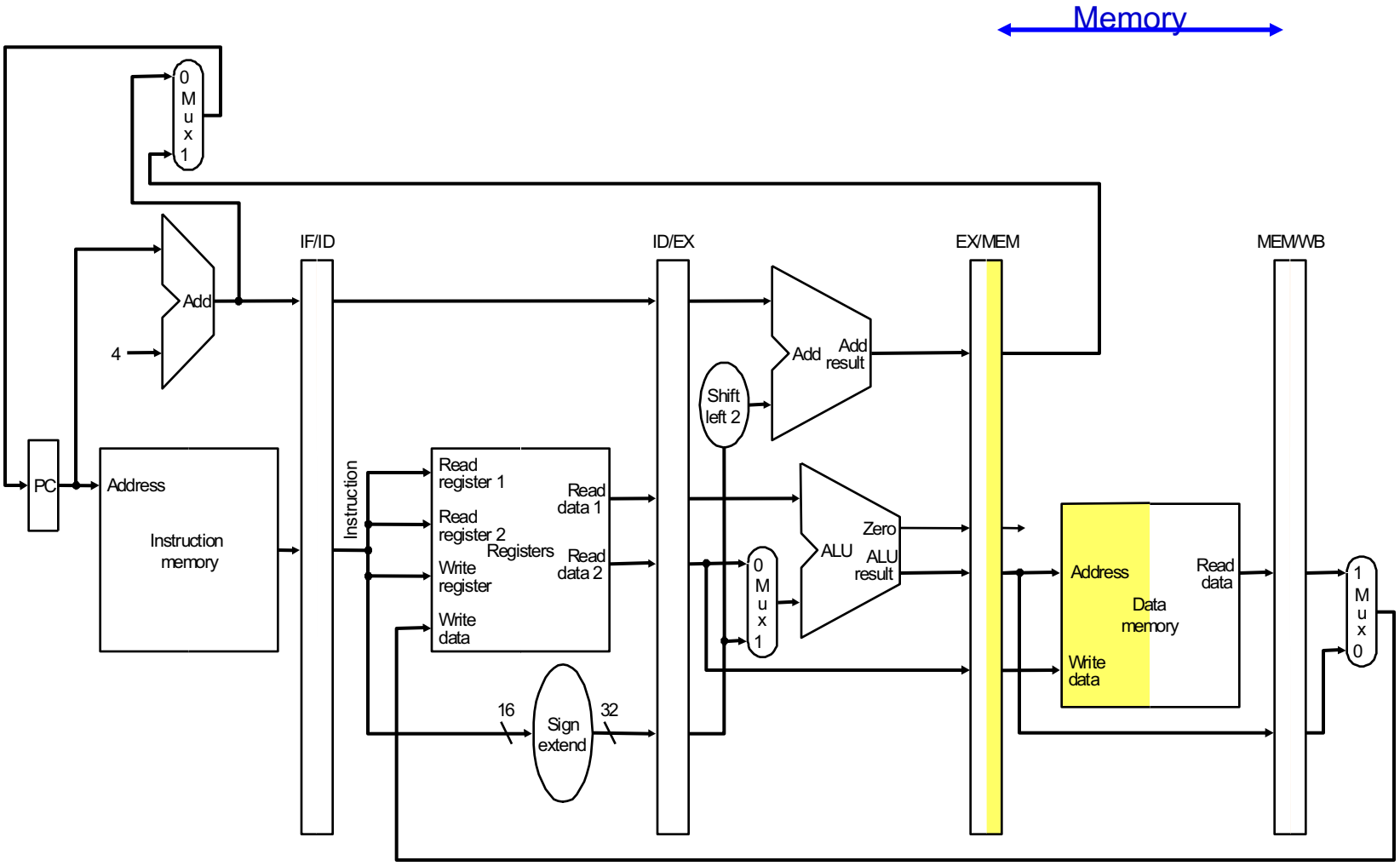
lw: Memory (MEM)



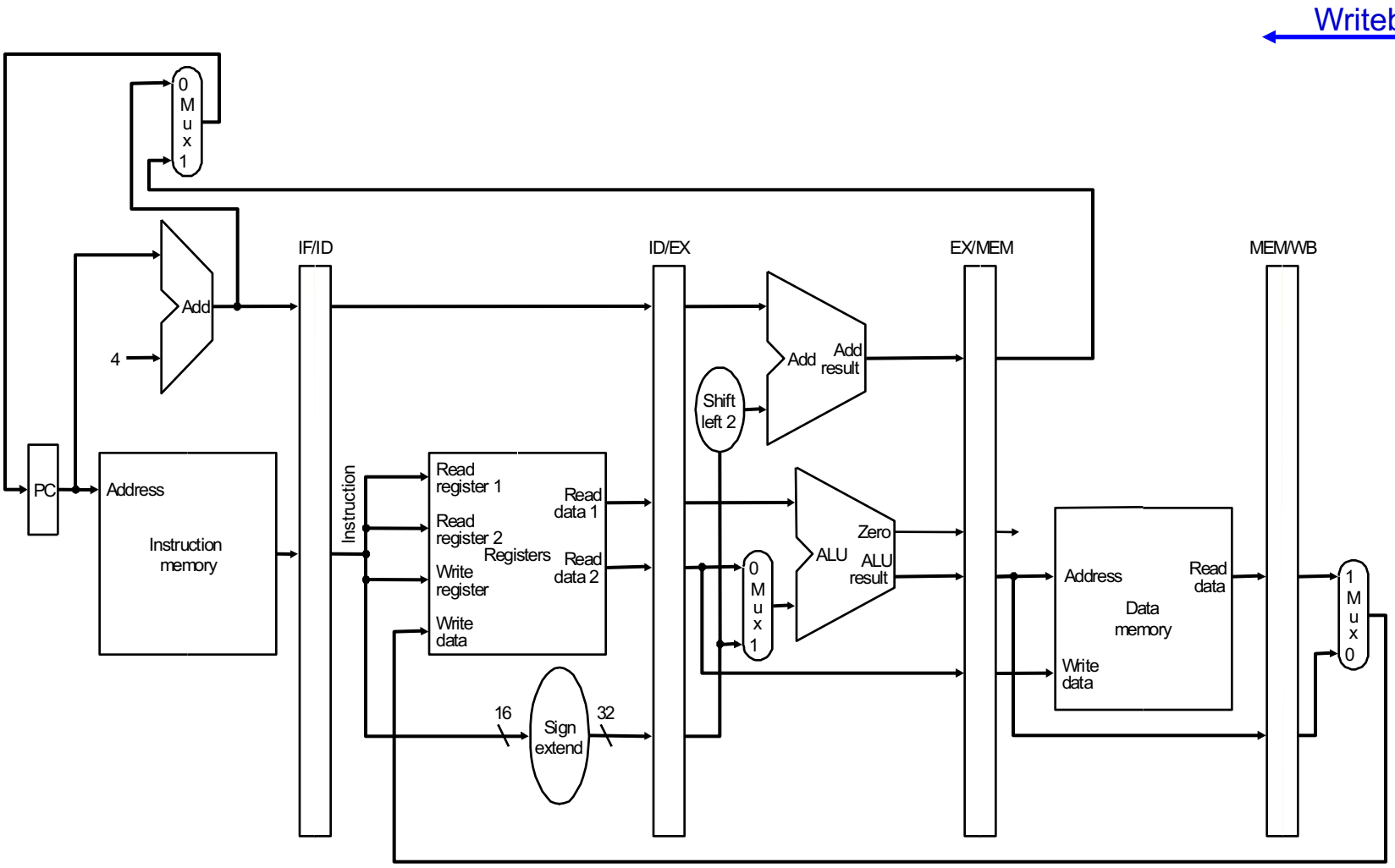
lw: Writeback (WB)



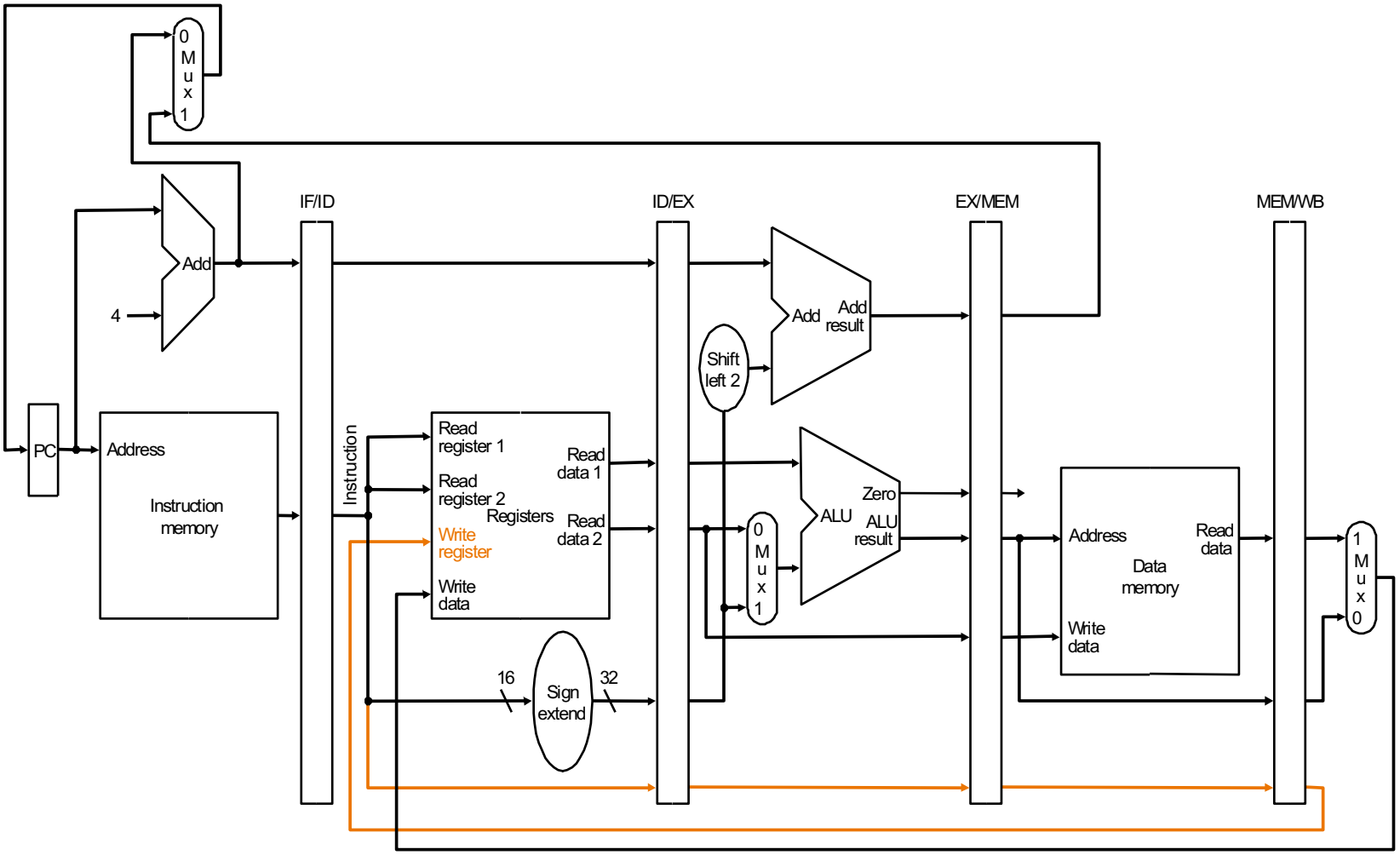
sw: Memory (MEM)



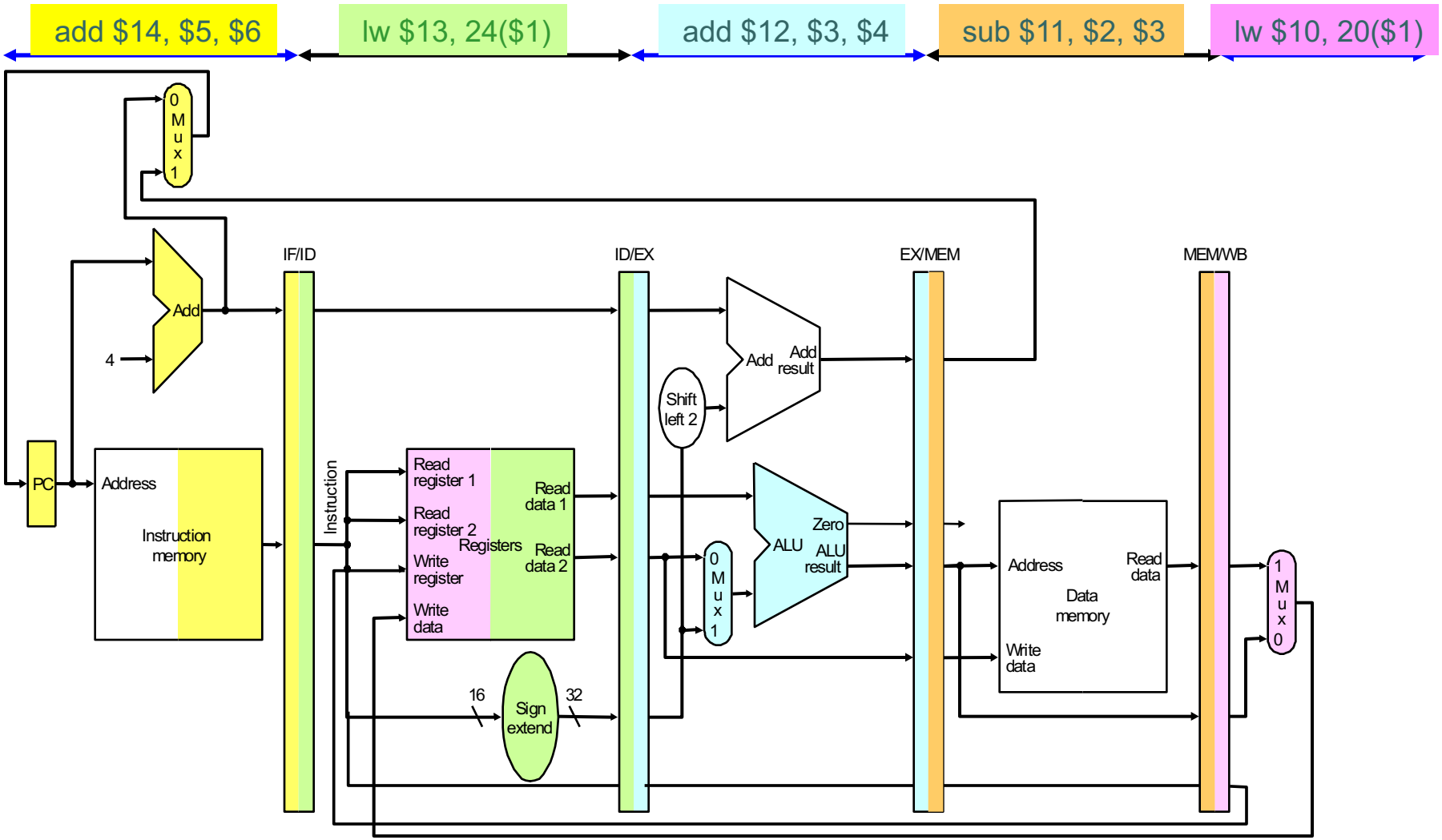
`sw`: Writeback (WB): do nothing



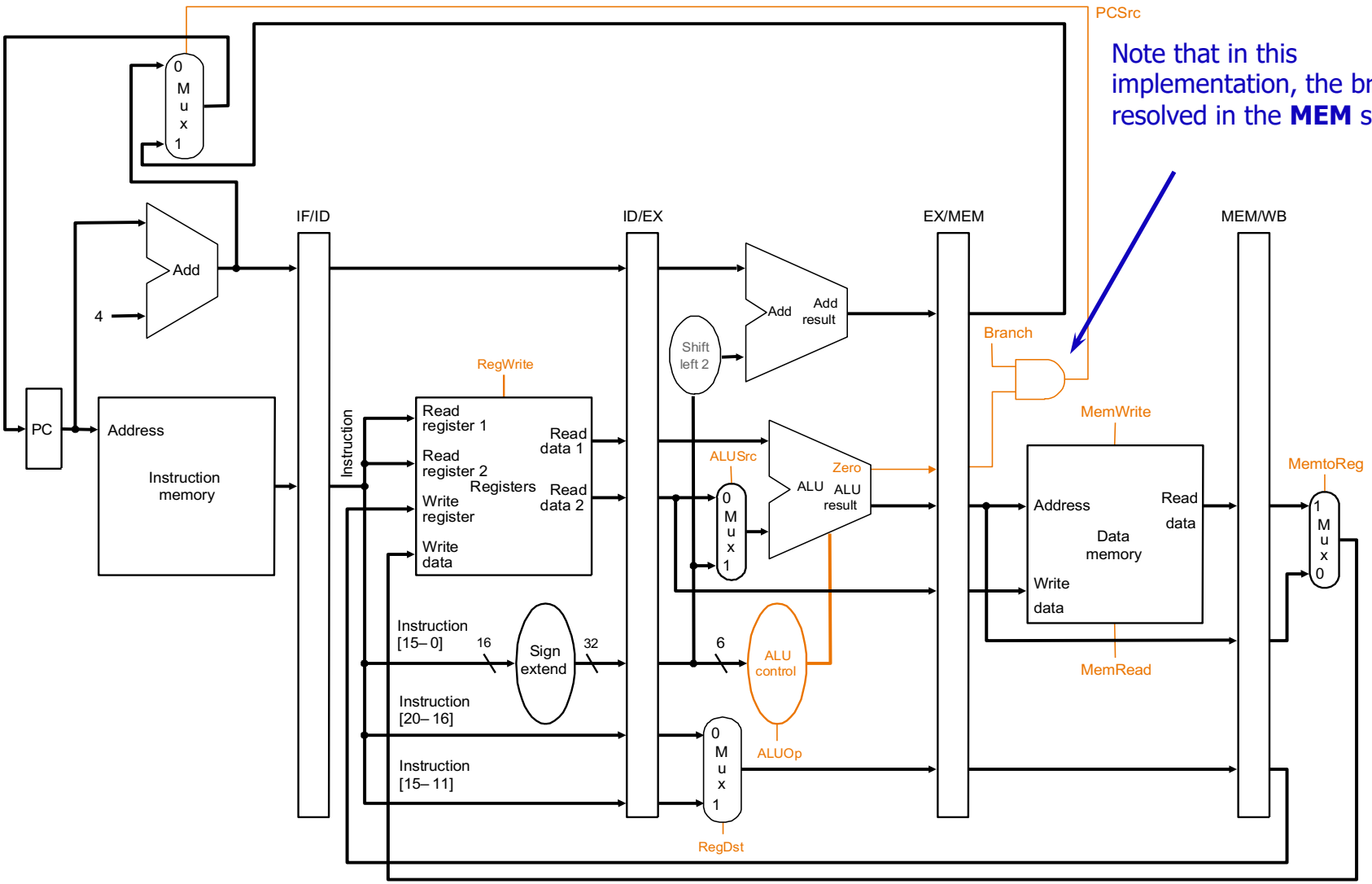
Corrected Datapath (for `lw`)



Pipelining Example



Pipeline Control



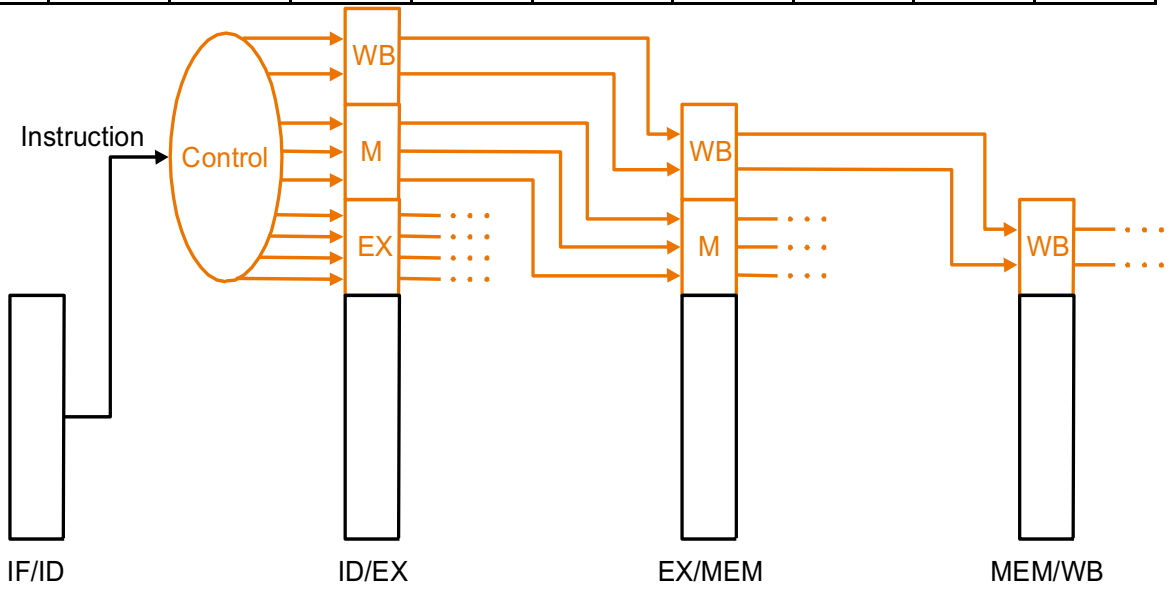
Pipeline Control

- 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)

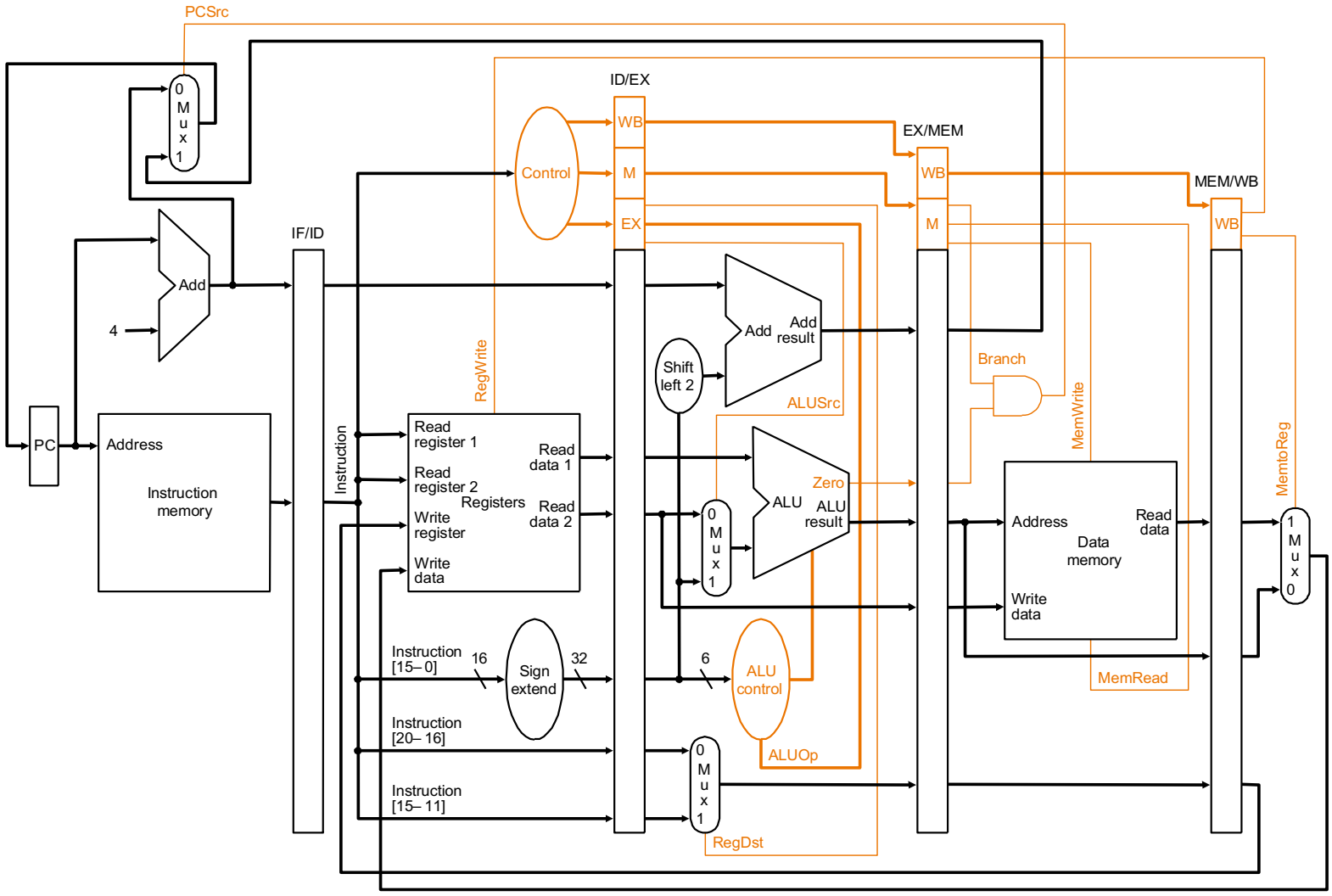
Pipeline Control

- 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

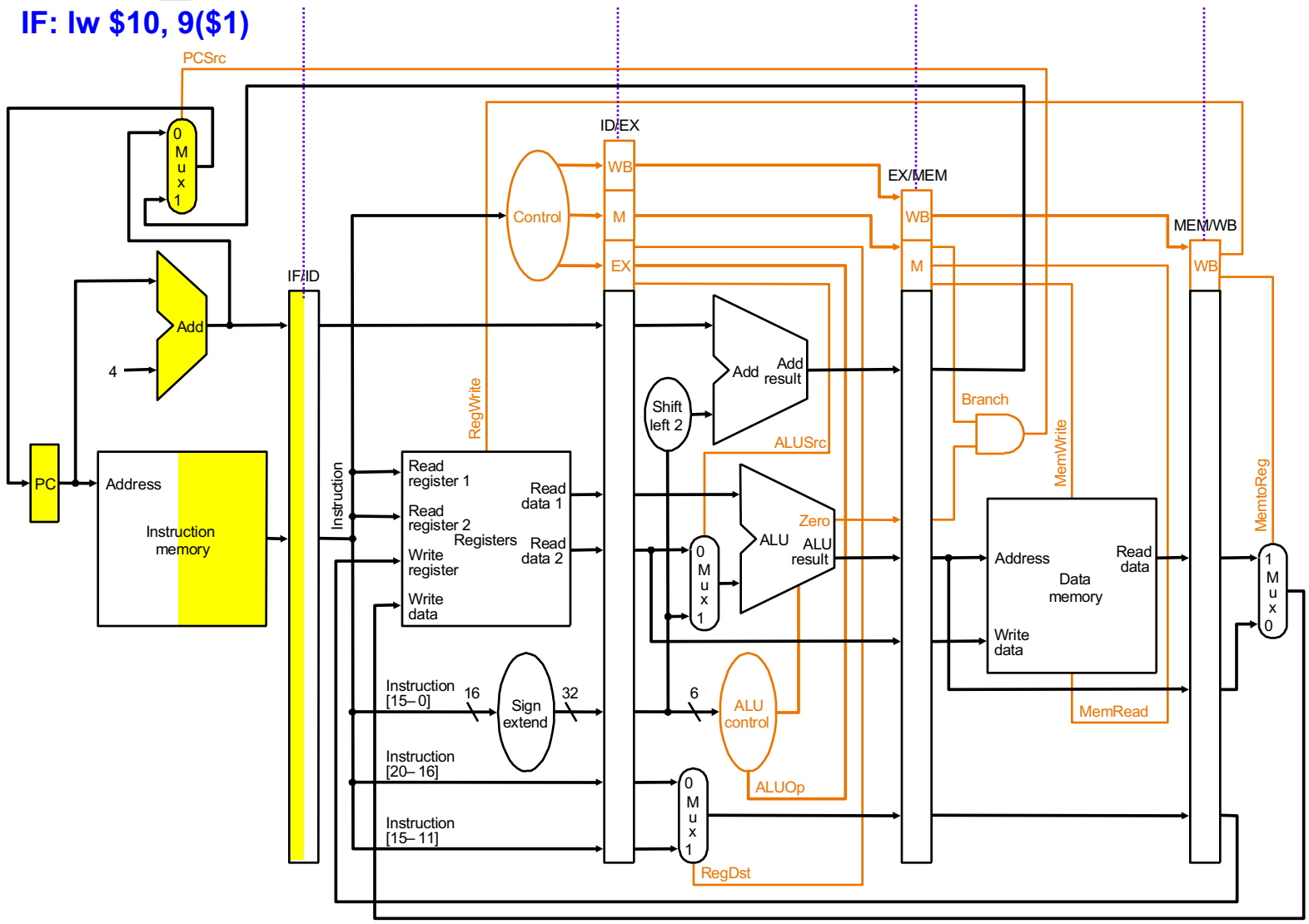


Datapath with Control

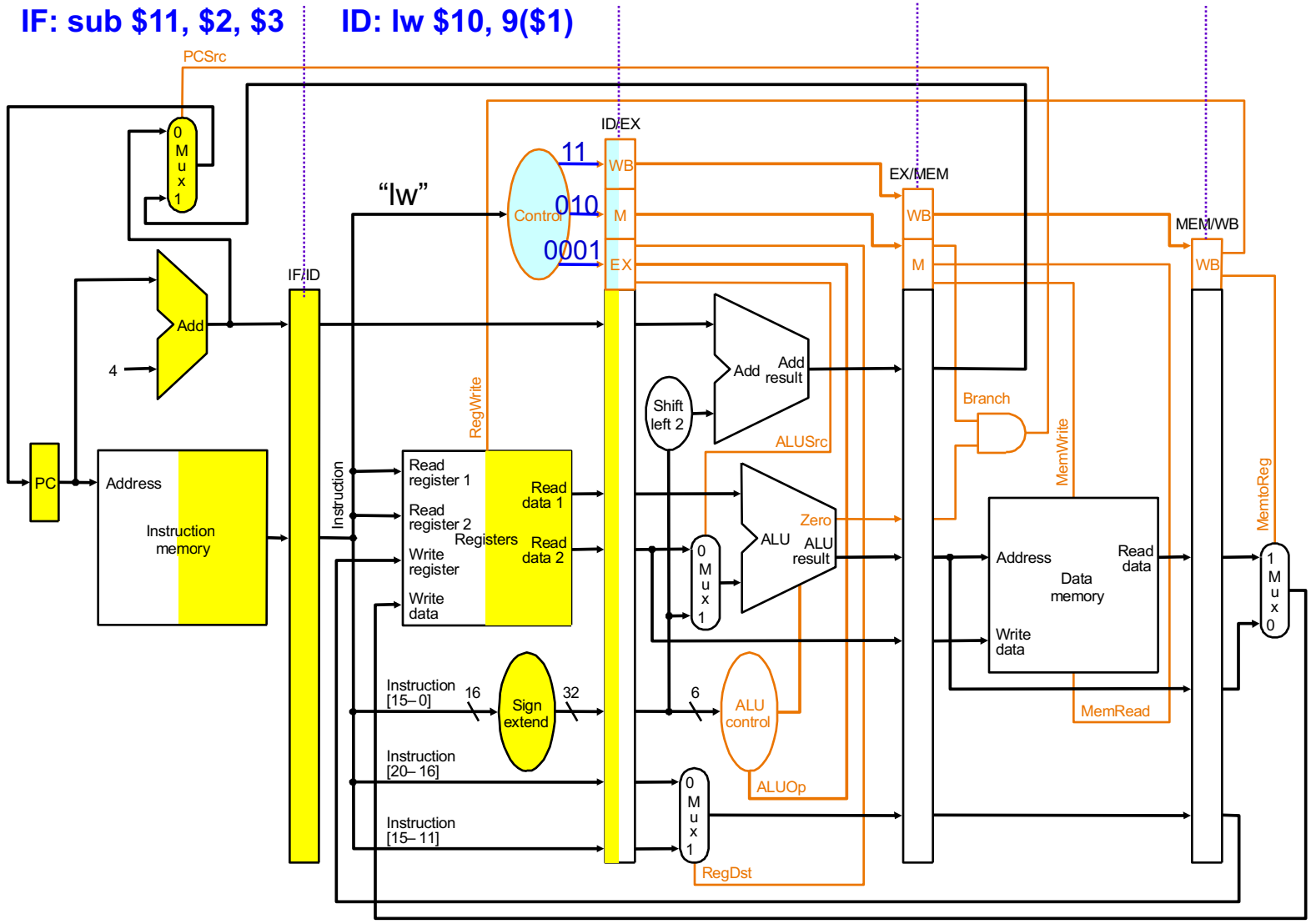


Datapath with Control

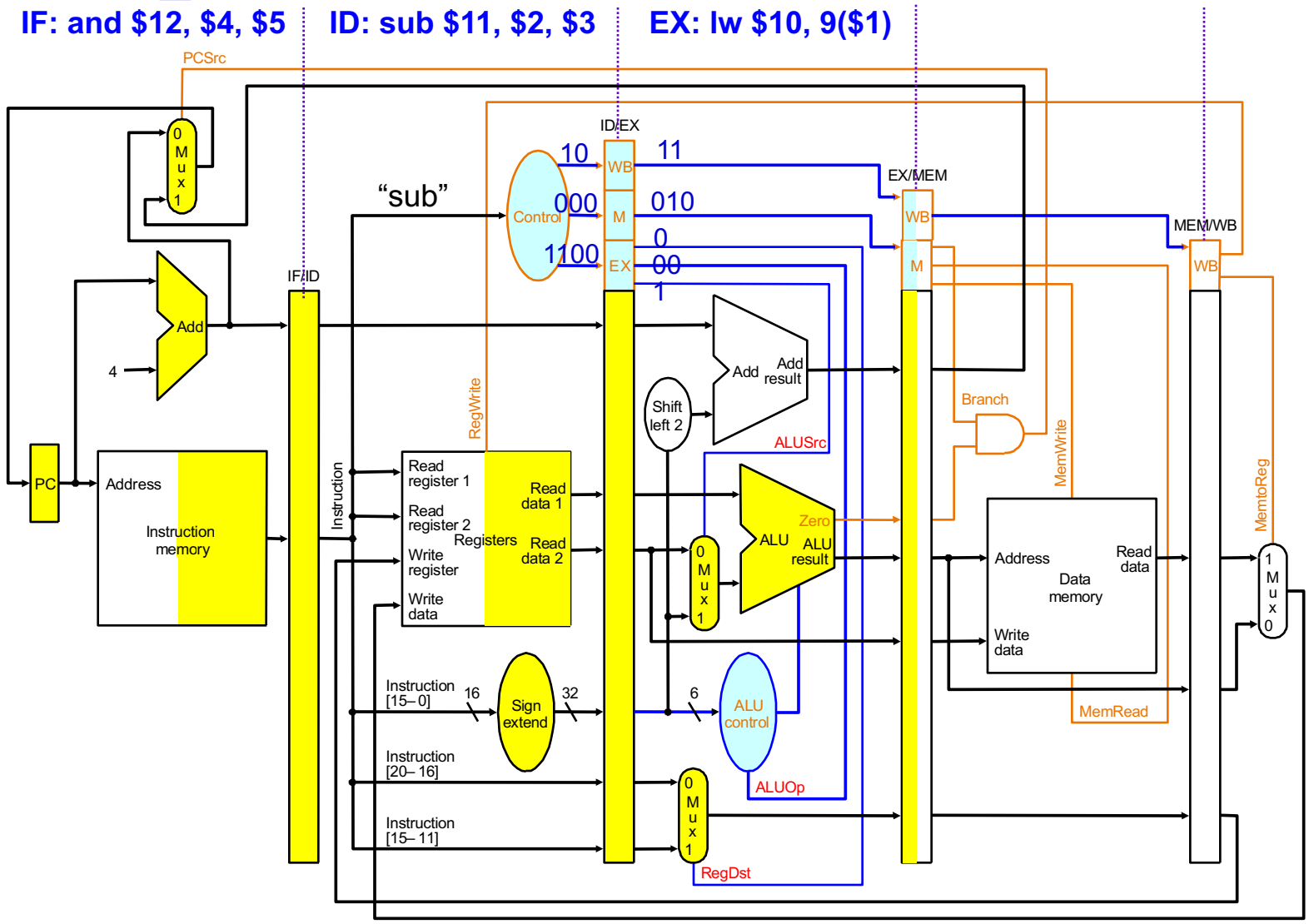
IF: lw $10, 9($1)



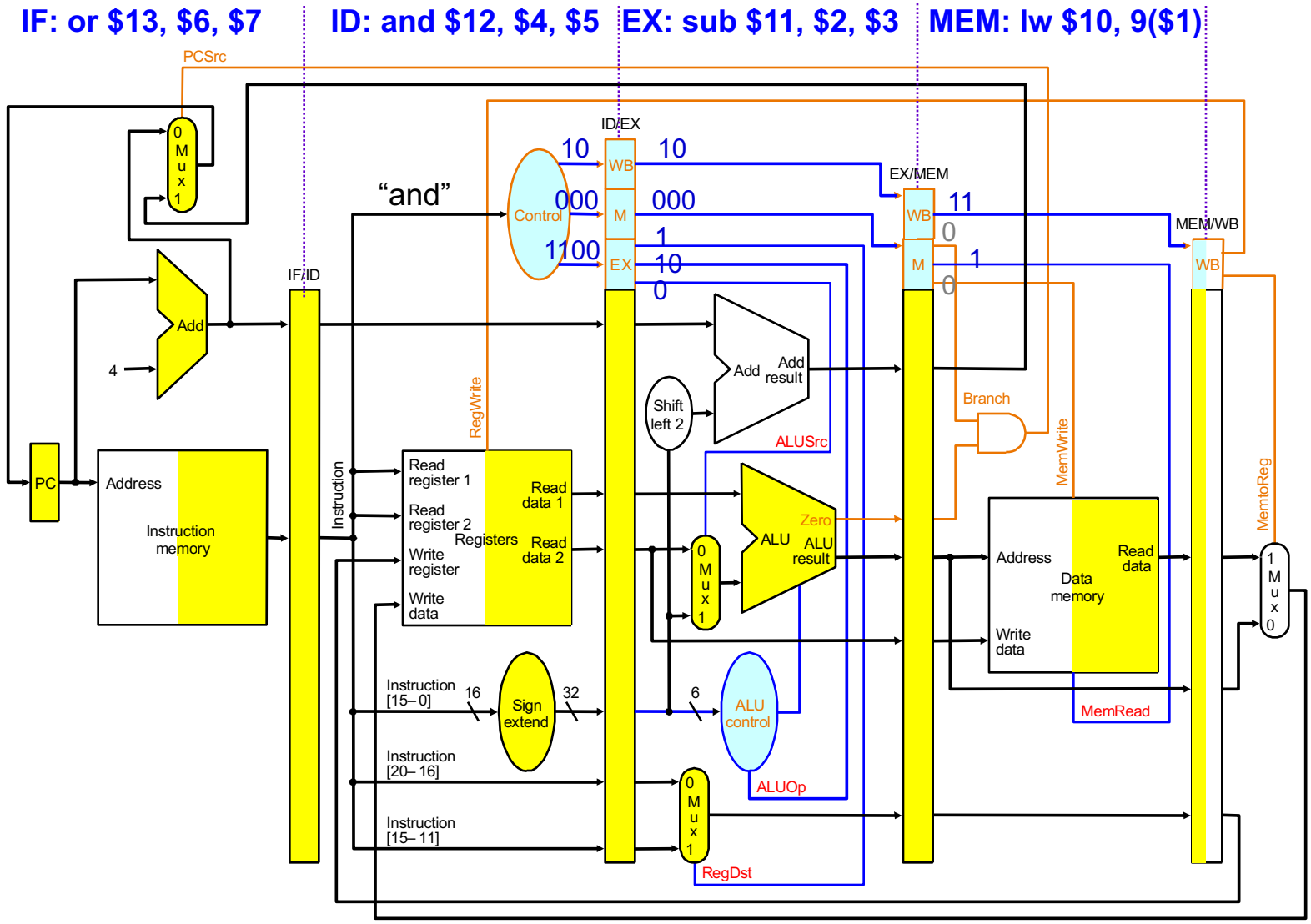
Datapath with Control



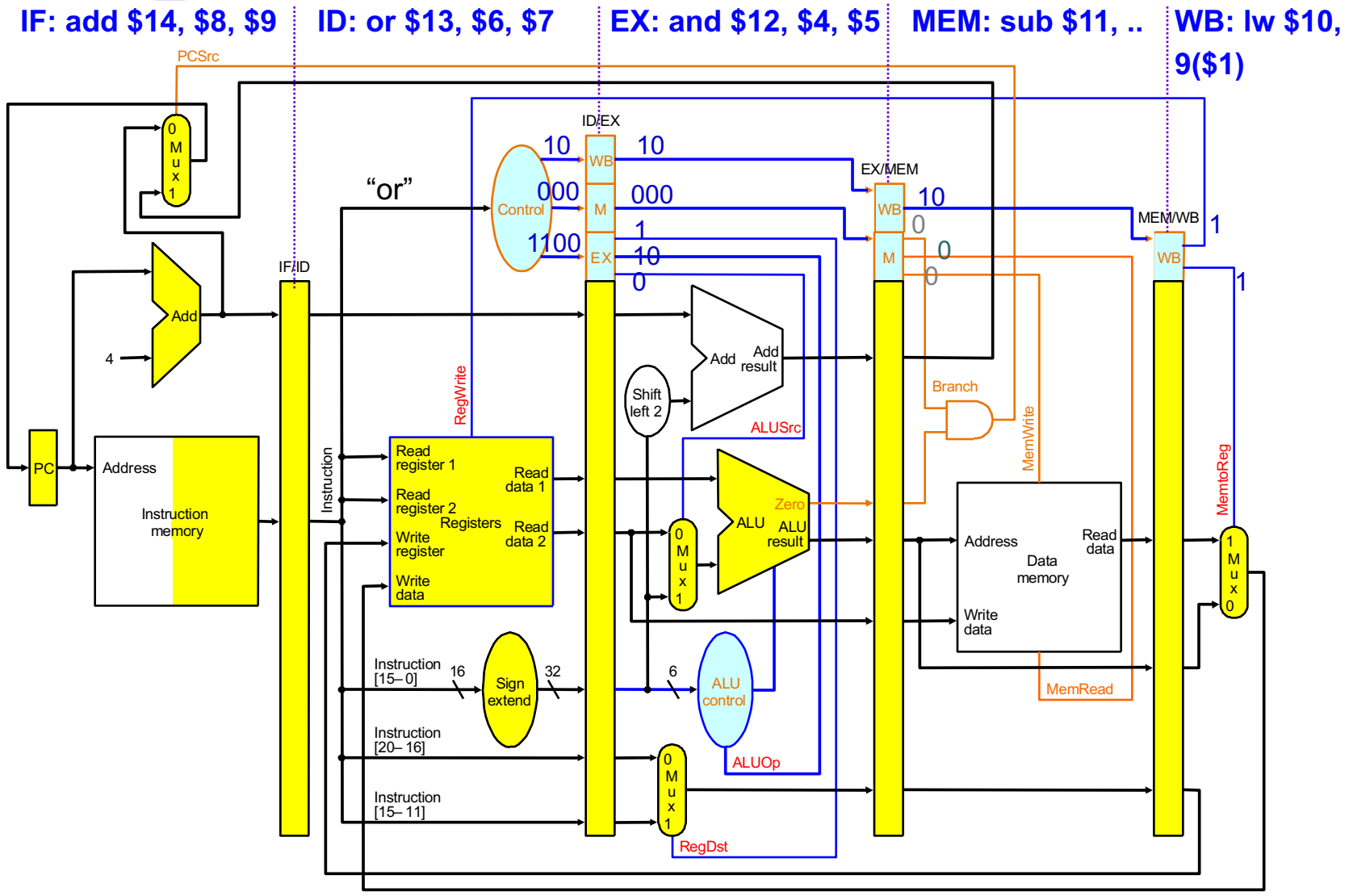
Datapath with Control



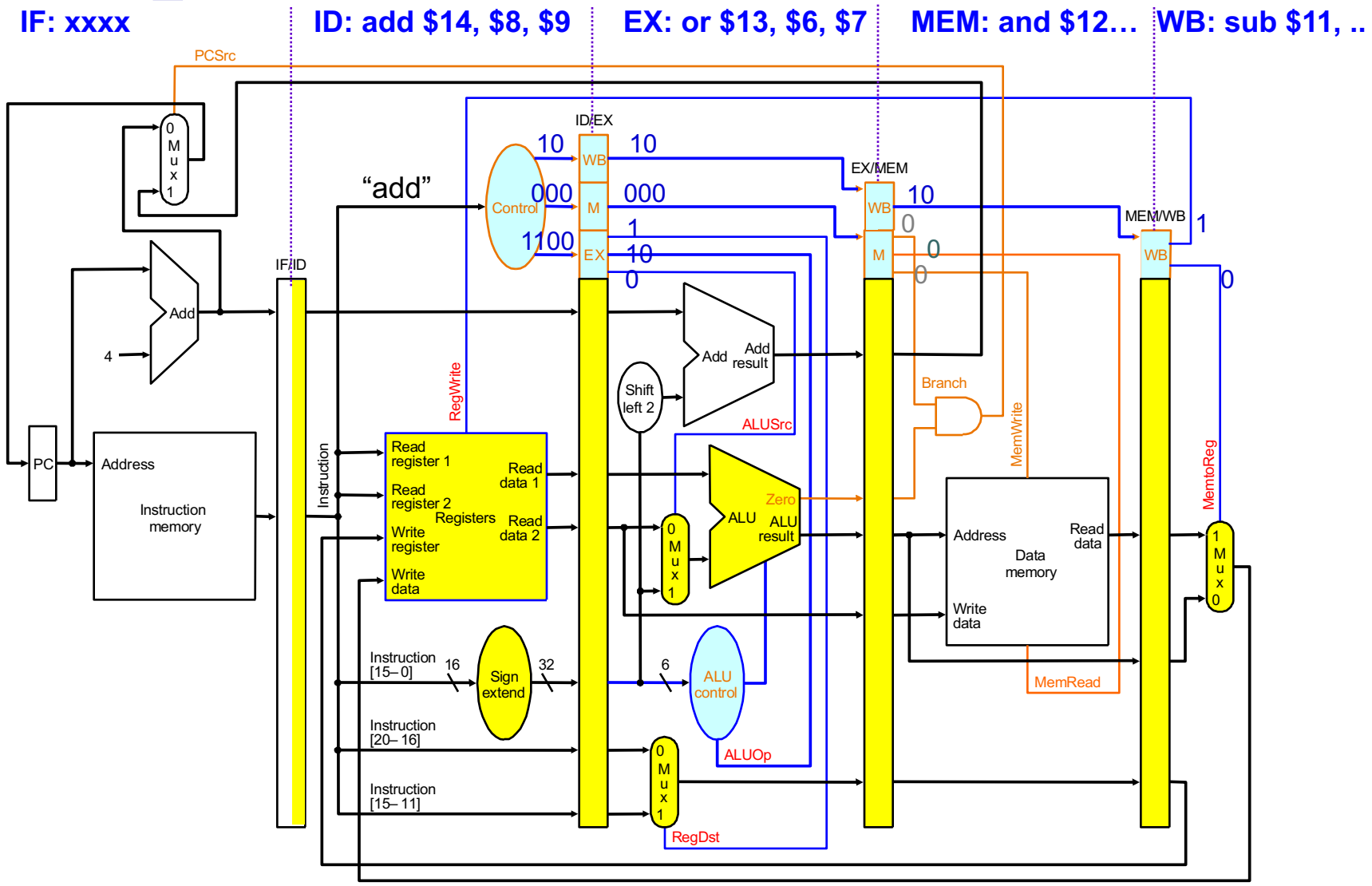
Datapath with Control



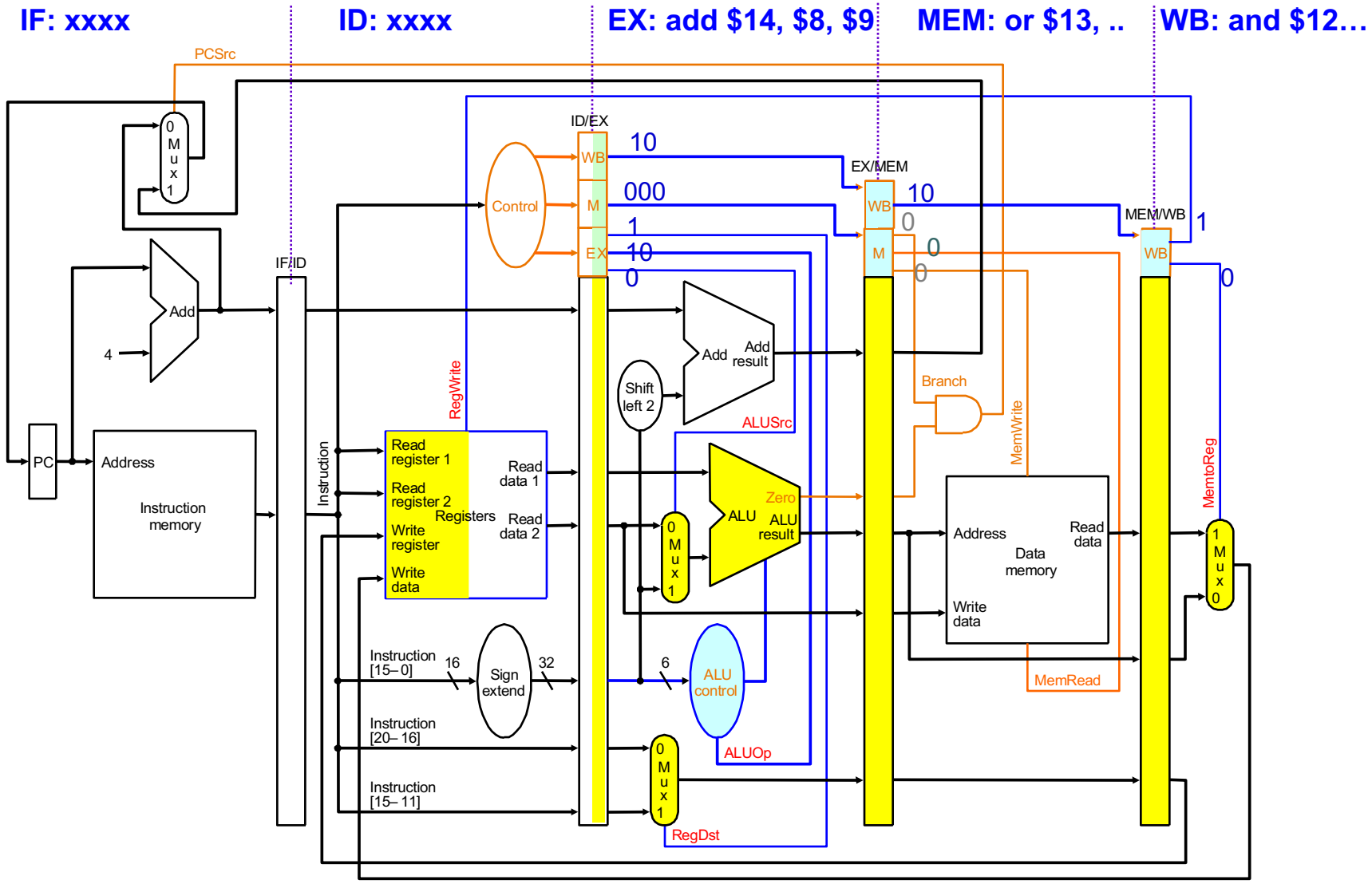
Datapath with Control



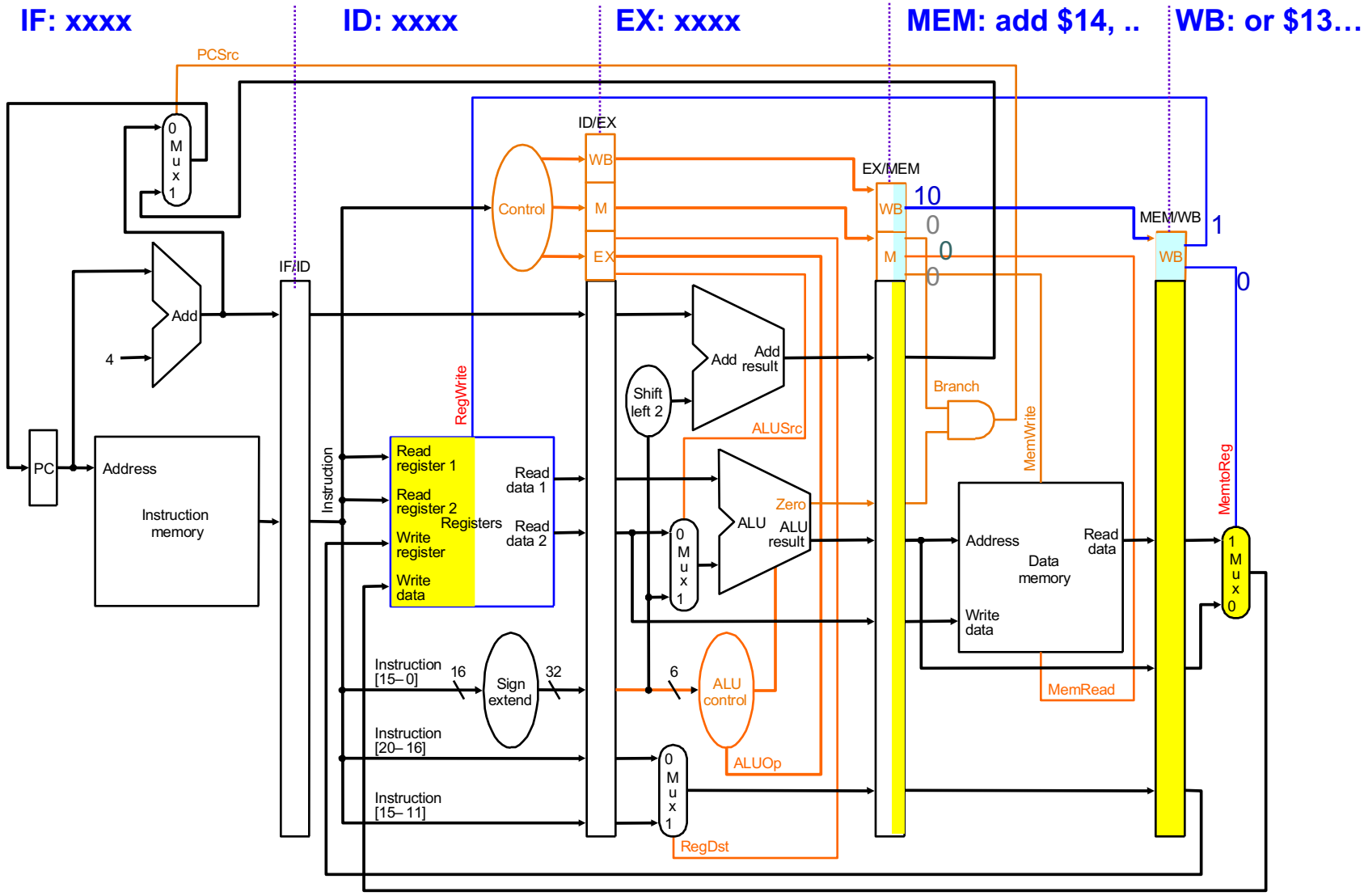
Datapath with Control



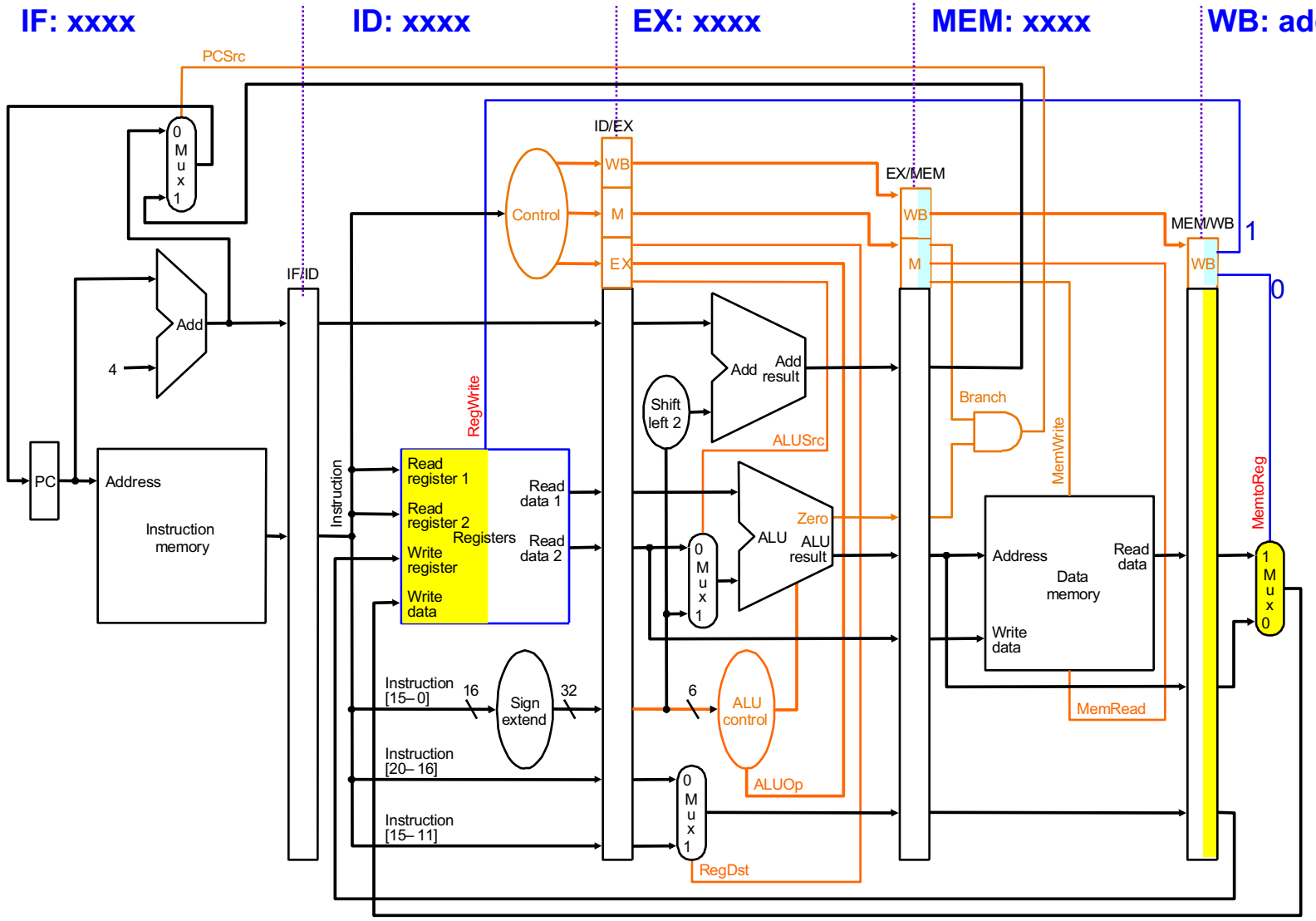
Datapath with Control



Datapath with Control



Datapath with Control



Outline

- Introduction to Pipelining
- How Pipeline is Implemented
- C.3 Pipeline Hazards
- Exceptions
- Handling Multicycle Operations

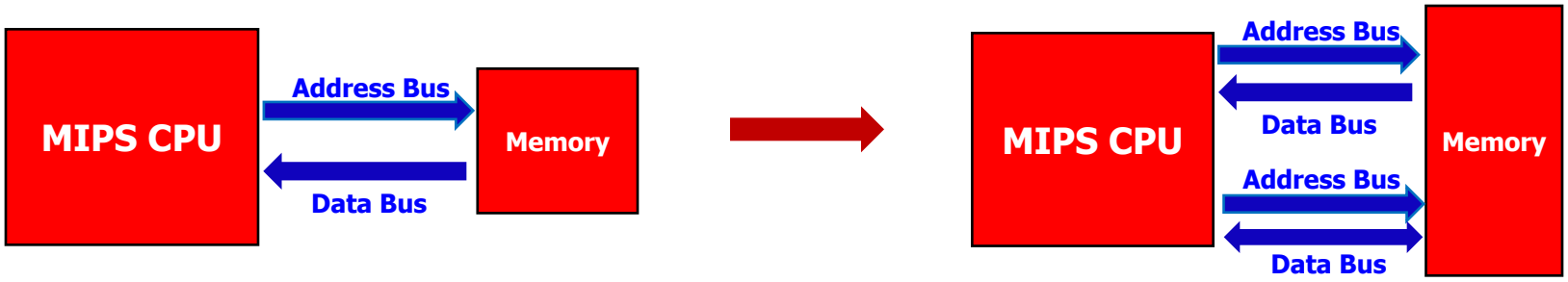
Hazards

- 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**!

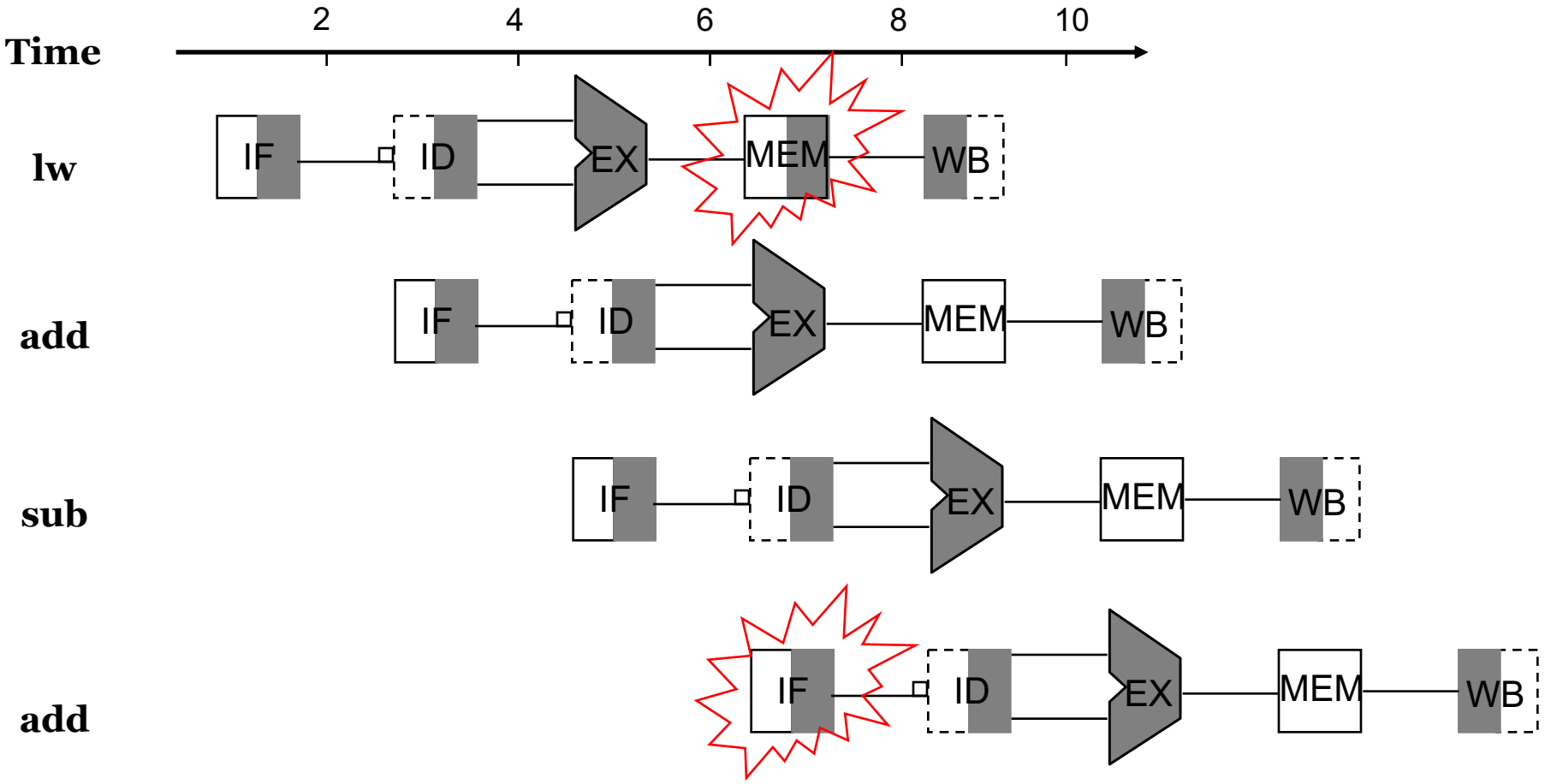
- Hazard is a situation that prevents starting the next instruction in the next cycle
  - **Structure hazard**
    - Conflict over the use of a resource at the same time
  - **Data hazard**
    - Data is not ready for the subsequent dependent instruction
  - **Control hazard**
    - Fetching the next 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



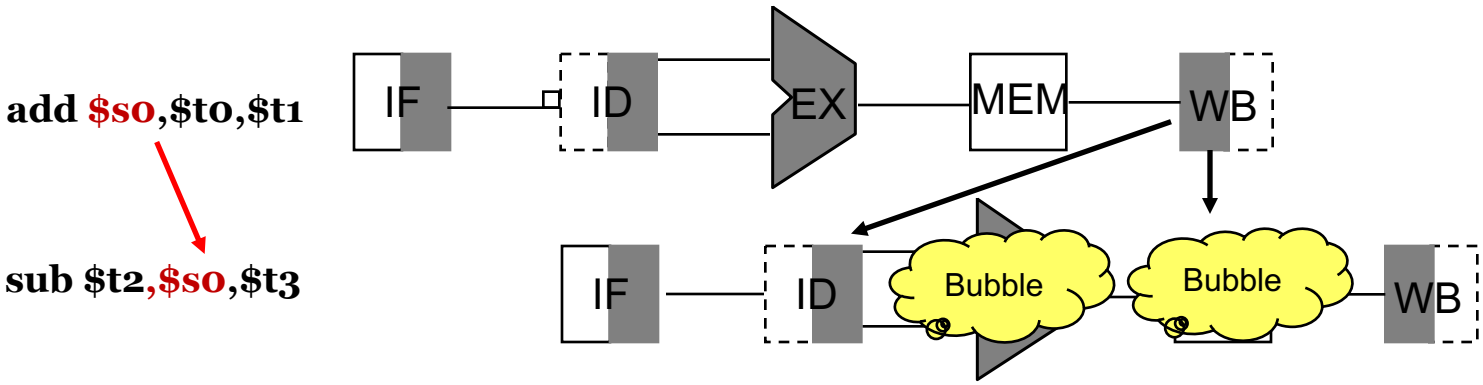
Structure Hazards (Cont.)



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

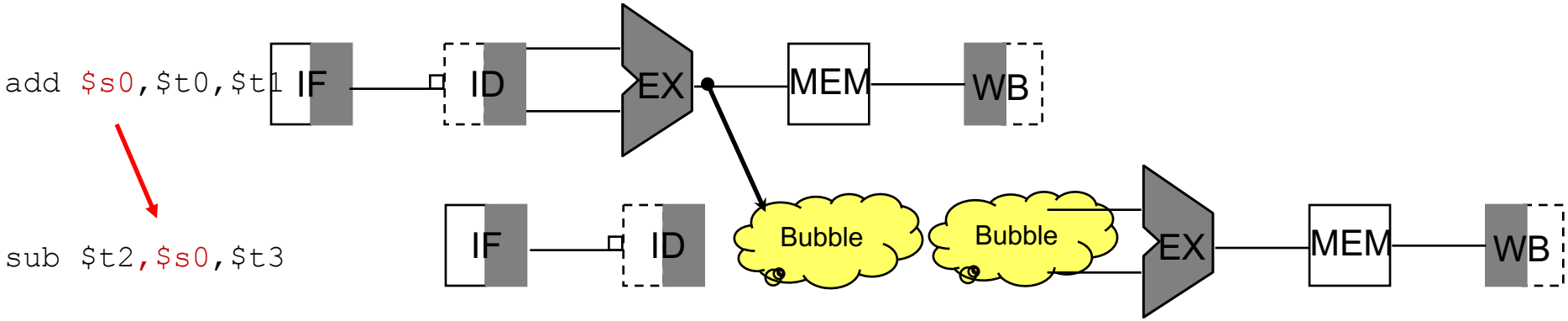
Data Hazards

Data is not ready for the subsequent dependent instruction



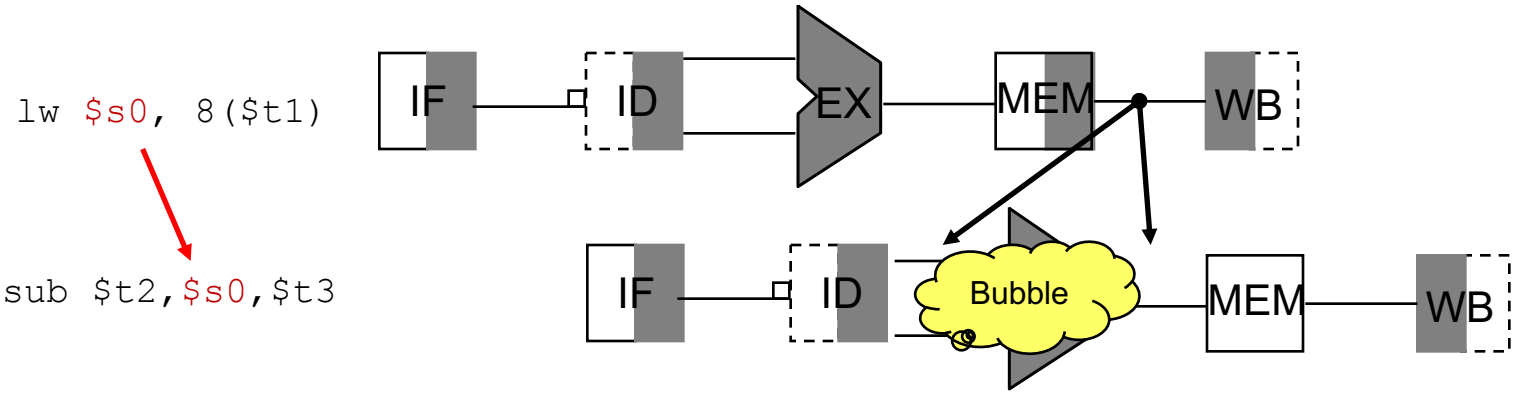
- To solve the data hazard problem, the pipeline needs to be stalled (typically referred to as "**bubble**")
  - Then, the performance is penalized
- A better solution?
  - **Forwarding** (or Bypassing)

Forwarding


add $s0,$t0,$t1
IF
ID
EX
MEM
WB
sub $t2,$s0,$t3
IF
ID
Bubble
Bubble
EX
MEM
WB

Data Hazard - Load-Use Case

- Can't always avoid stalls by forwarding
  - Can't forward backward in time!

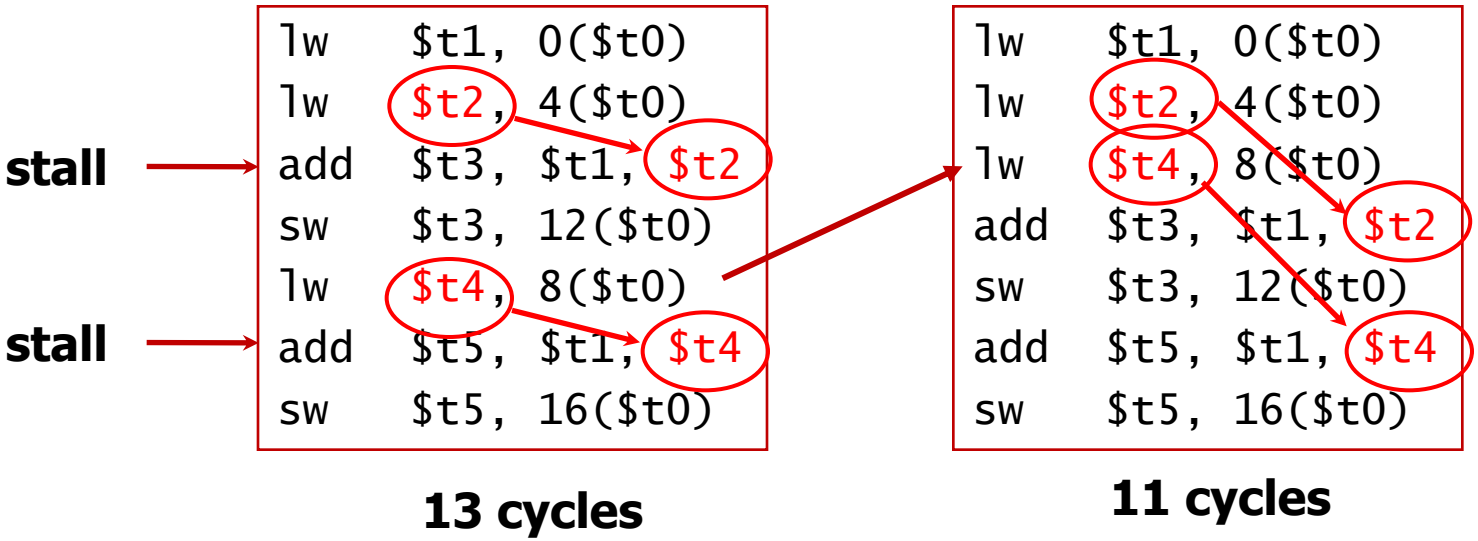


- This bubble can be hidden by proper instruction scheduling

Code Scheduling to Avoid Stalls

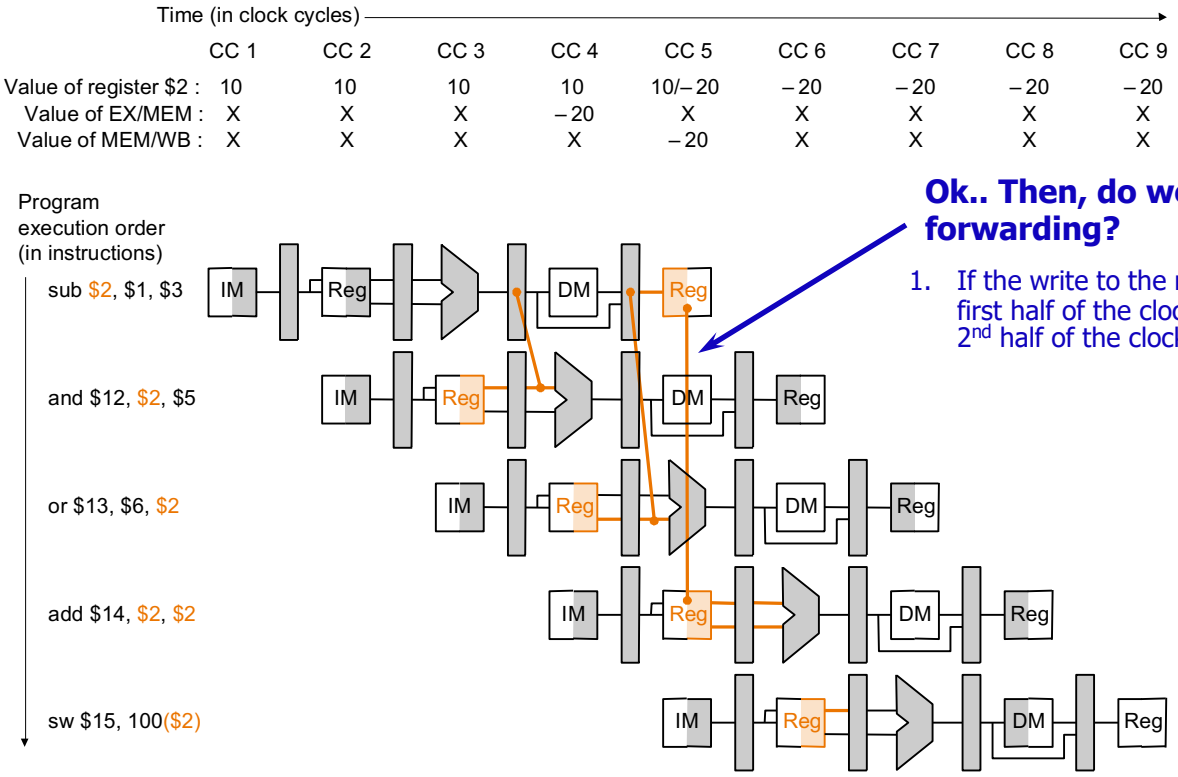
- Reorder code to avoid `use of load` result in the next instruction

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

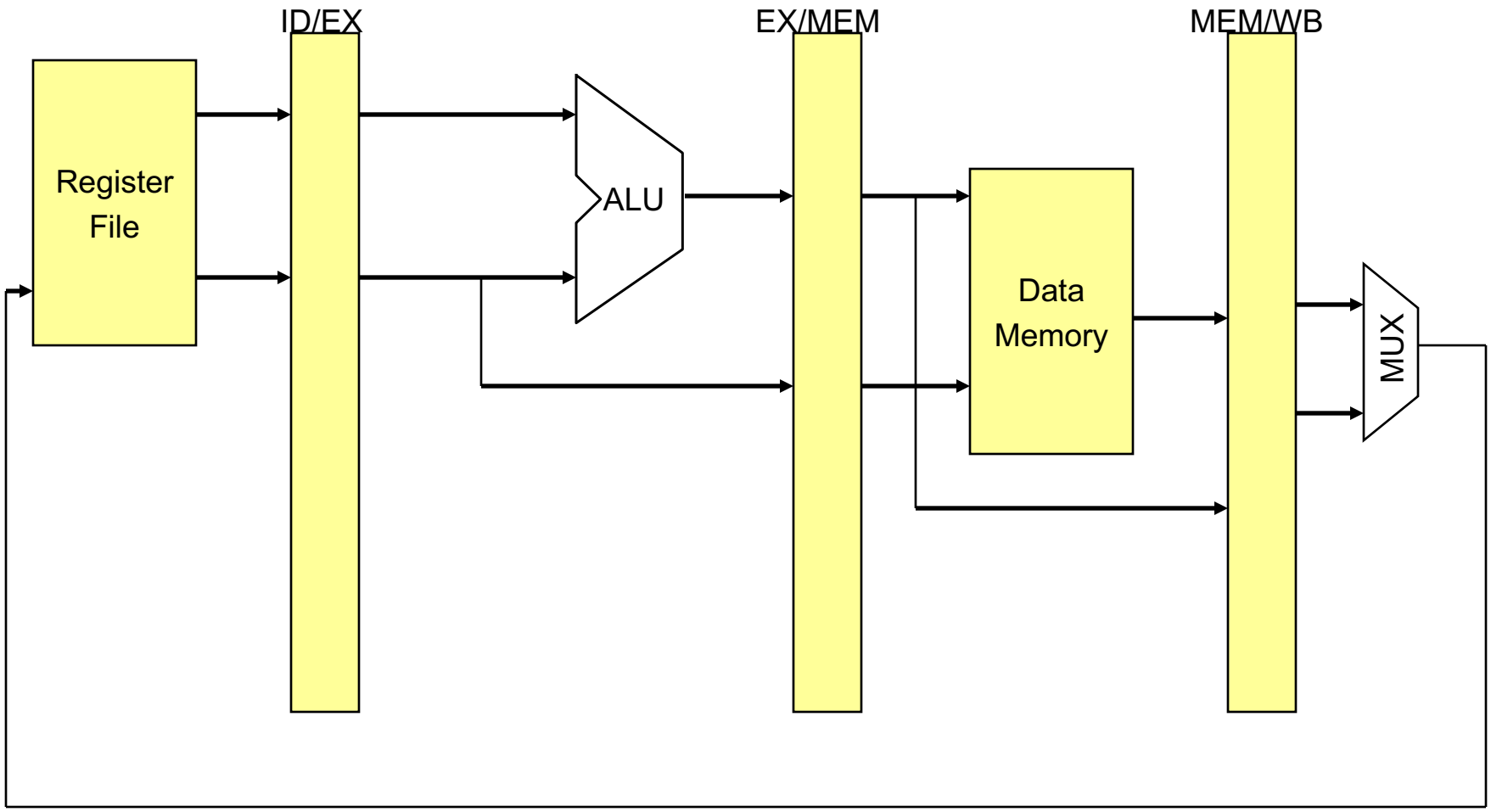


Data Hazard - Forwarding

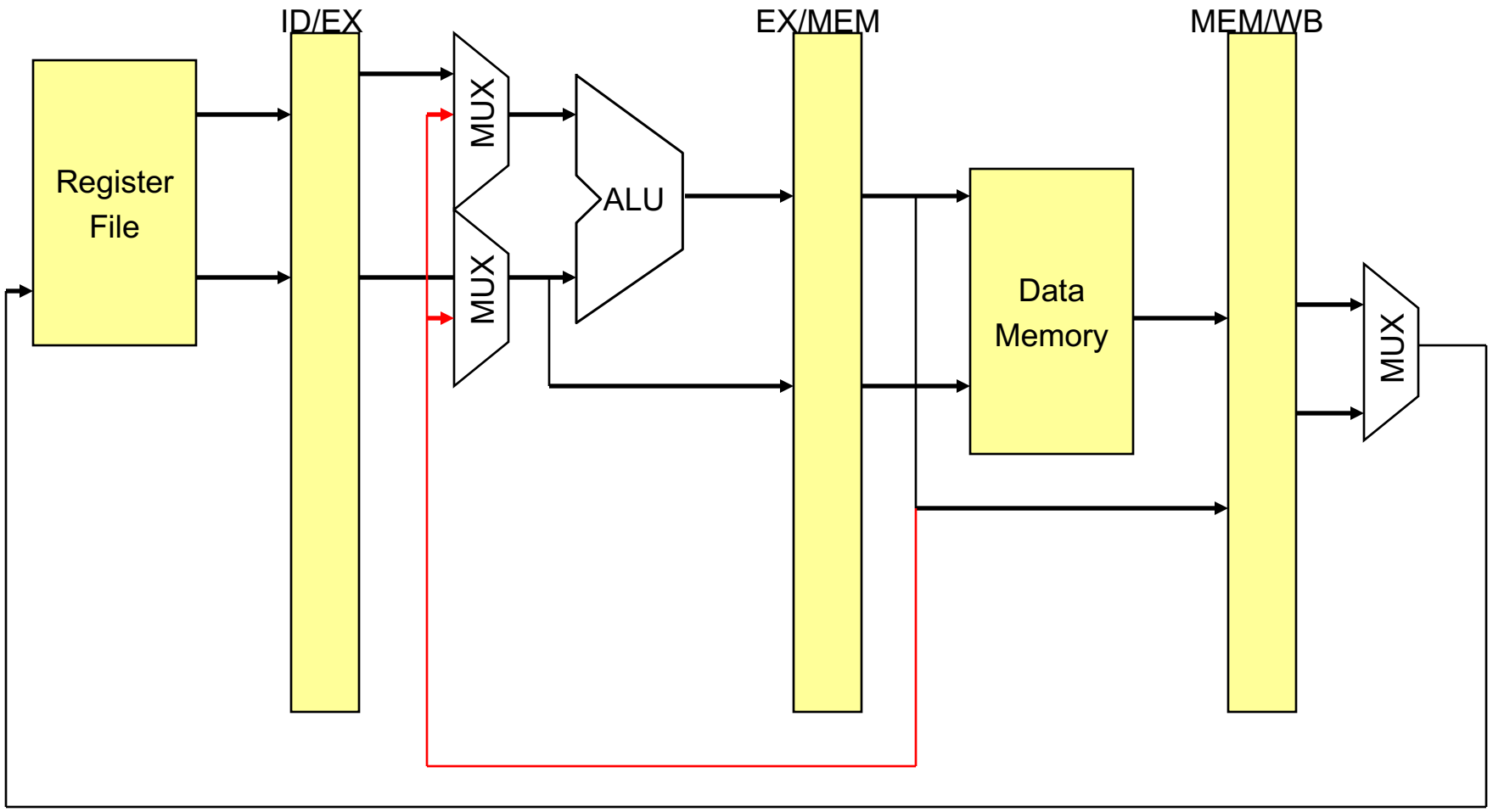
- Don't wait for them to be written to the register file
  - Use temporary results



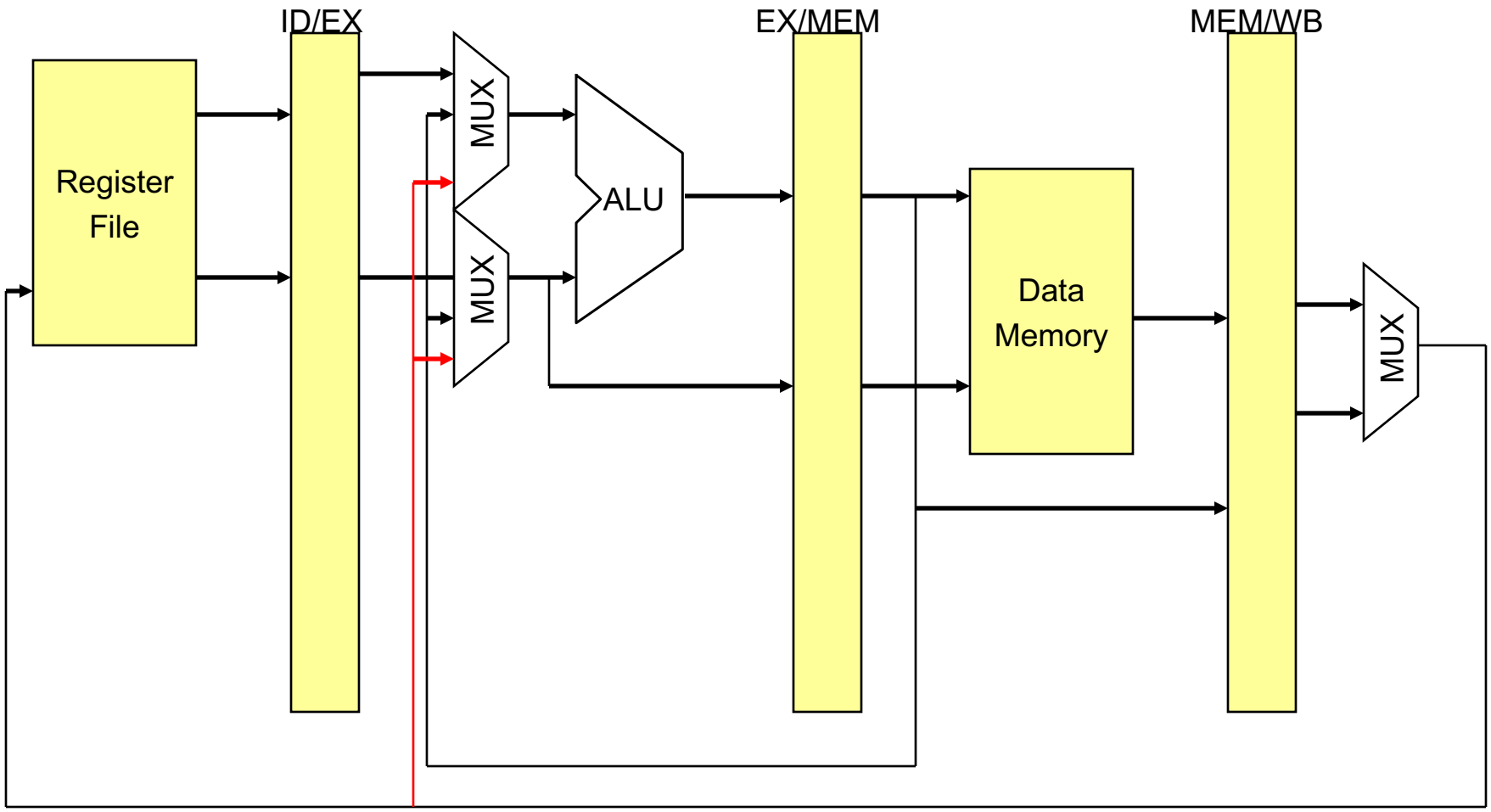
Forwarding



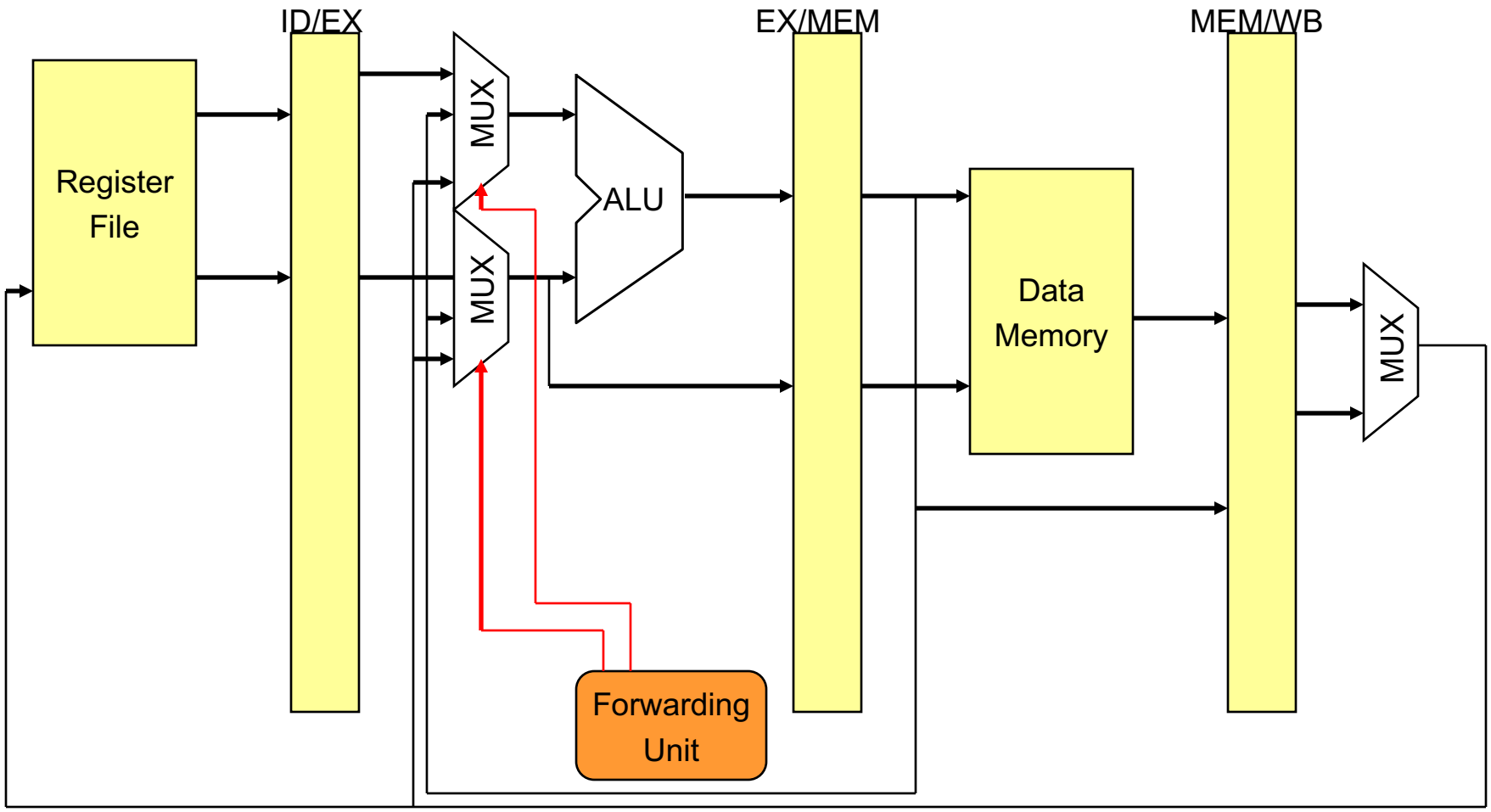
Forwarding (from EX/MEM)



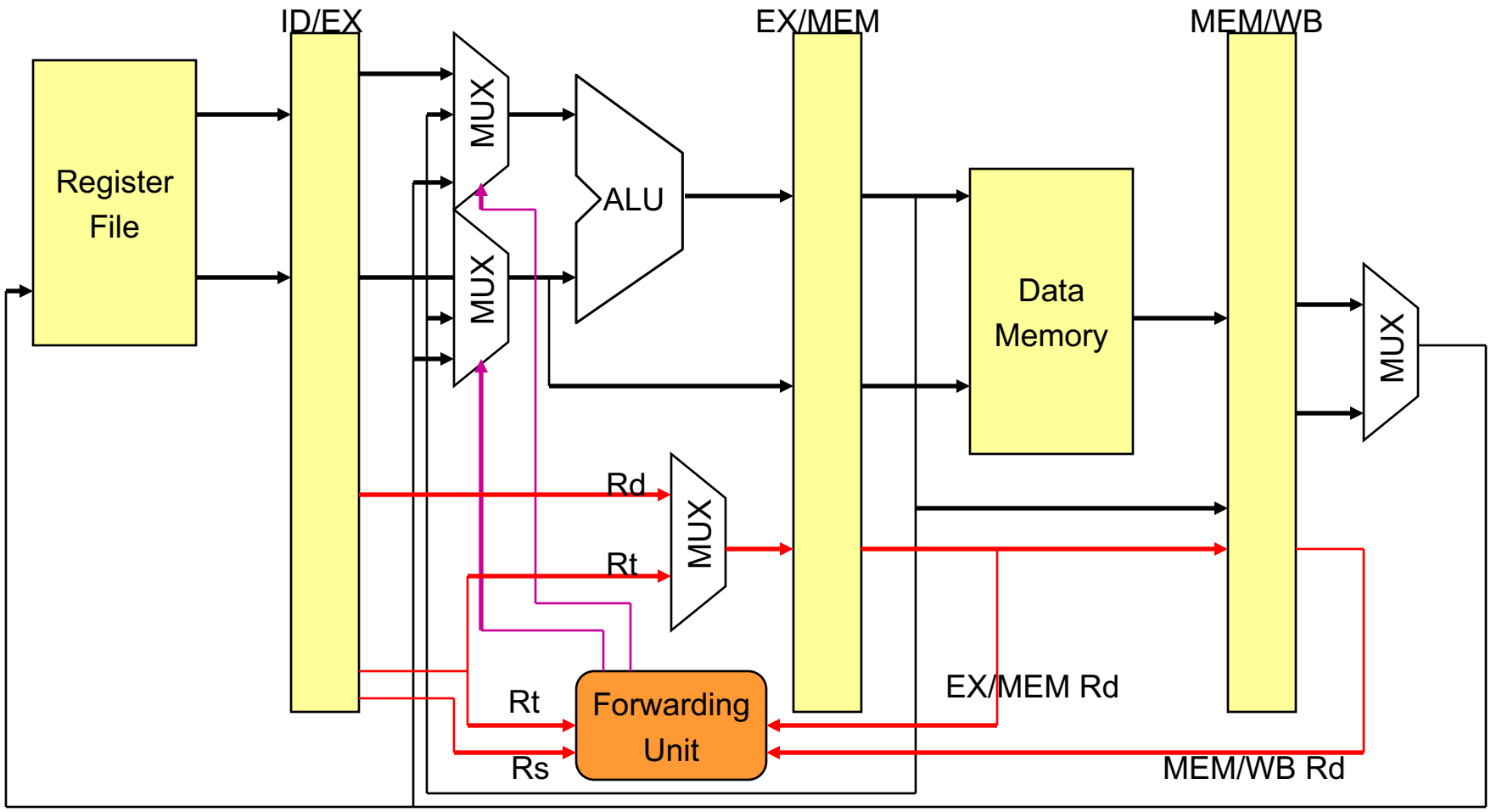
Forwarding (from MEM/WB)



Forwarding (operand selection)

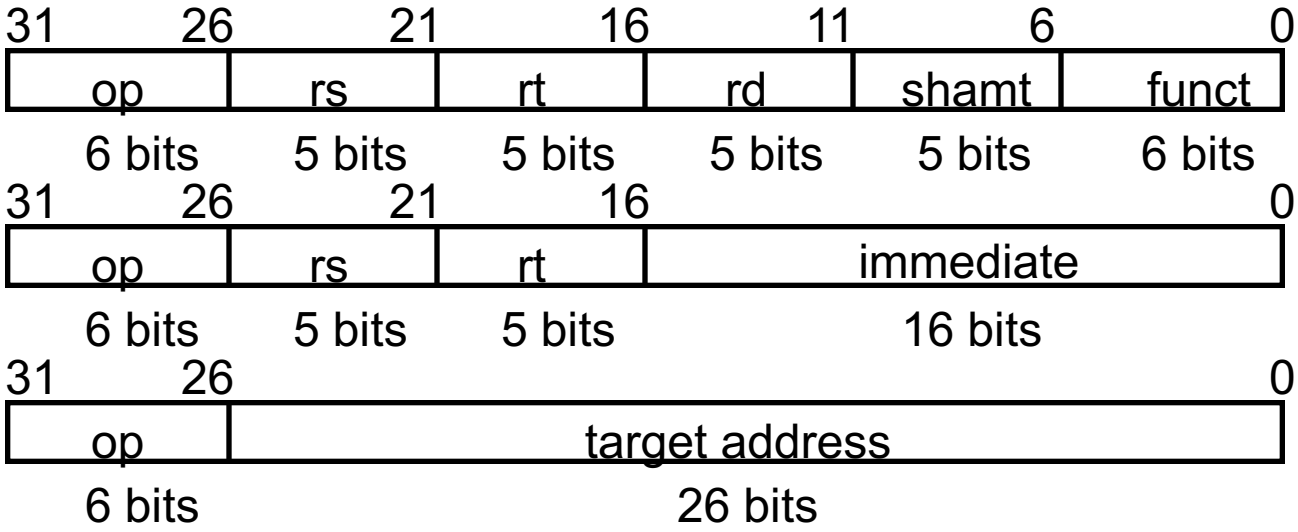


Forwarding (operand propagation)



Review: The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. The three instruction formats:

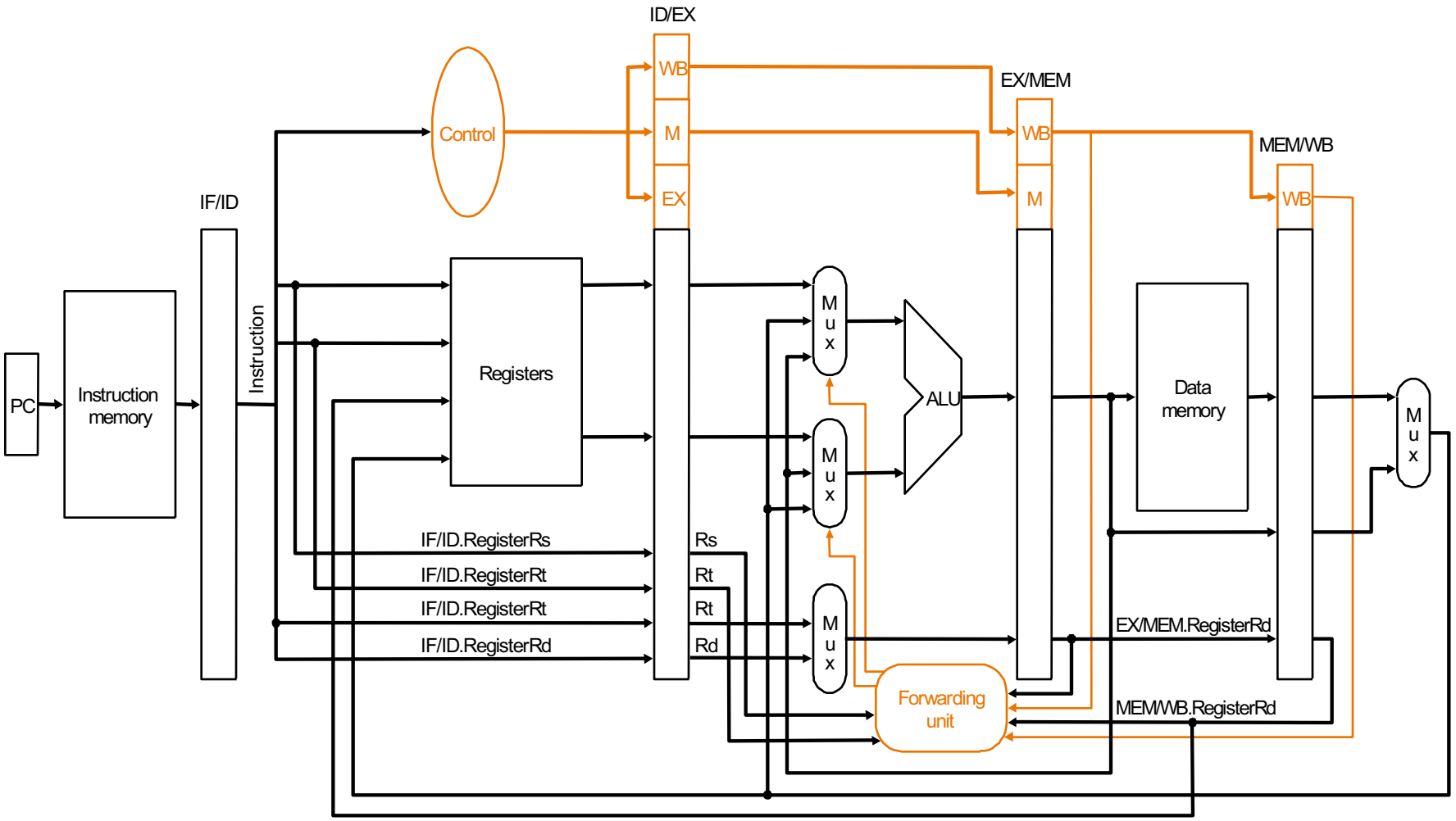


- The different fields are:
  - op: operation of the instruction
  - rs, rt, rd: the source and destination register specifiers
  - shamt: shift amount
  - funct: selects the variant of the operation in the “op” field
  - address / immediate: address offset or immediate value
  - target address: target address of the jump instruction

Forwarding Logic Implementation

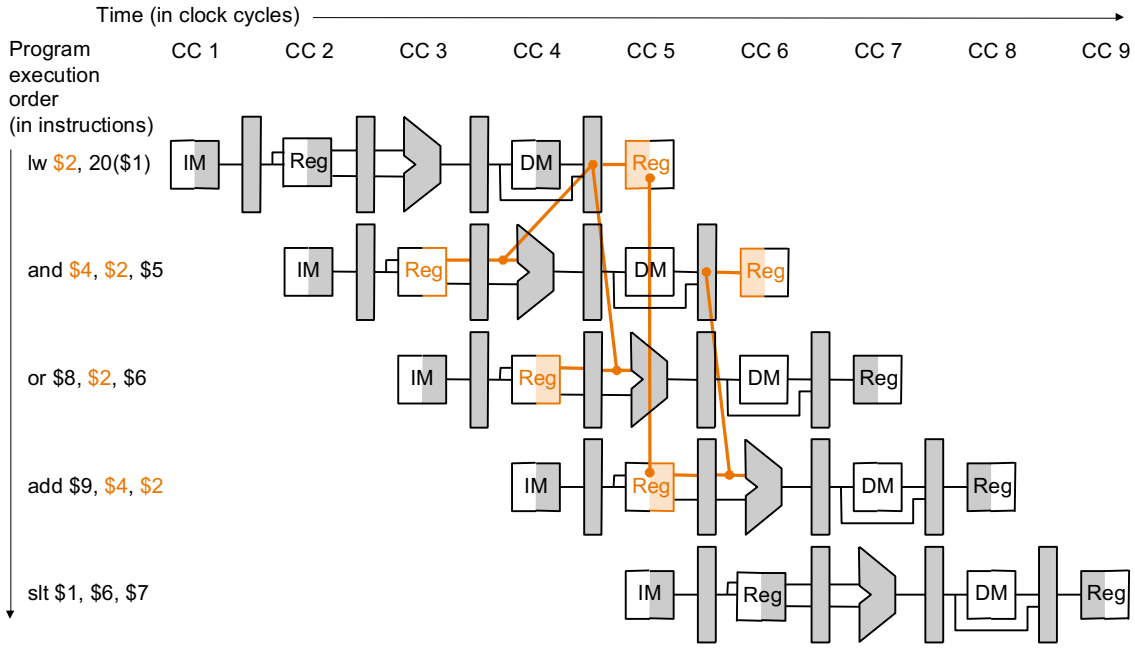
Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rd] == ID/EX.IR[rs]
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rd] == ID/EX.IR[rt]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rd] == ID/EX.IR[rs]
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rd] == ID/EX.IR[rt]
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR[rt] == ID/EX.IR[rs]
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR[rt] == ID/EX.IR[rt]
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR[rt] == ID/EX.IR[rs]
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR[rt] == ID/EX.IR[rt]

Forwarding


ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
PC
Instruction memory
Instruction
Registers
M u x
ALU
Data memory
M u x
M u x
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
Forwarding unit
MEM/WB.RegisterRd

Can't always forward

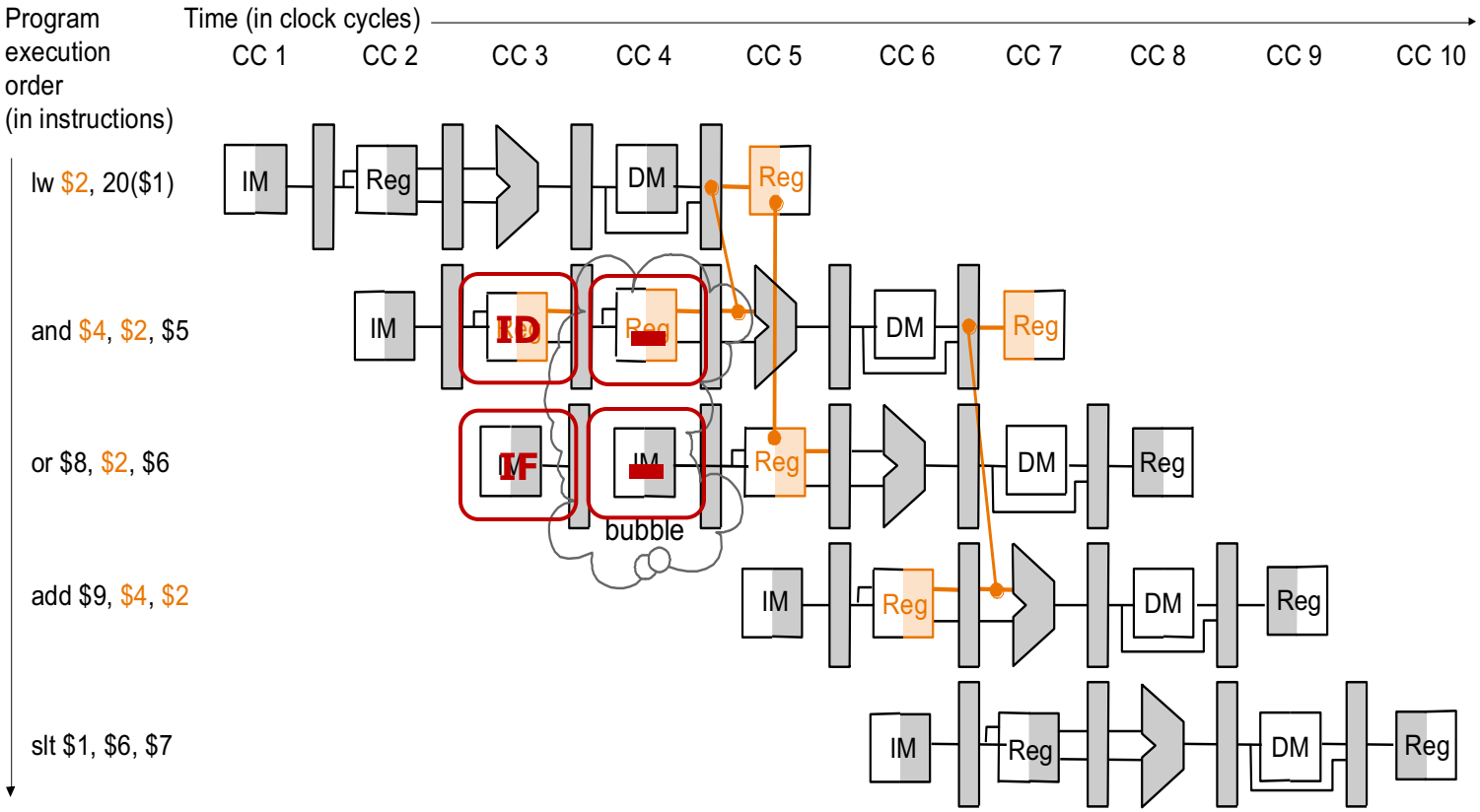
- `lw` (load word) can still cause a hazard
  - An instruction tries to read a register following a load instruction that writes to the same register



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

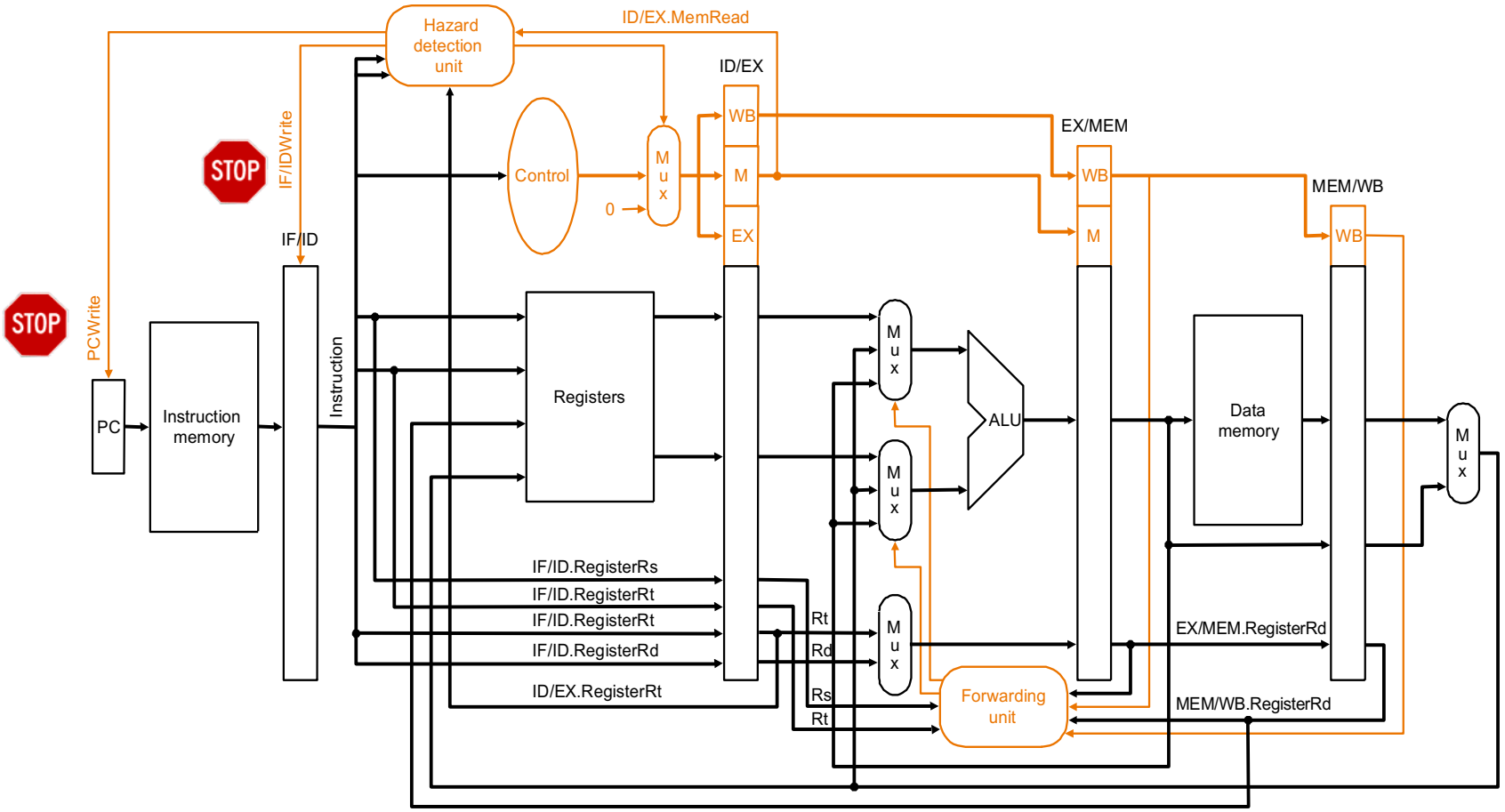
Stalling

- 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



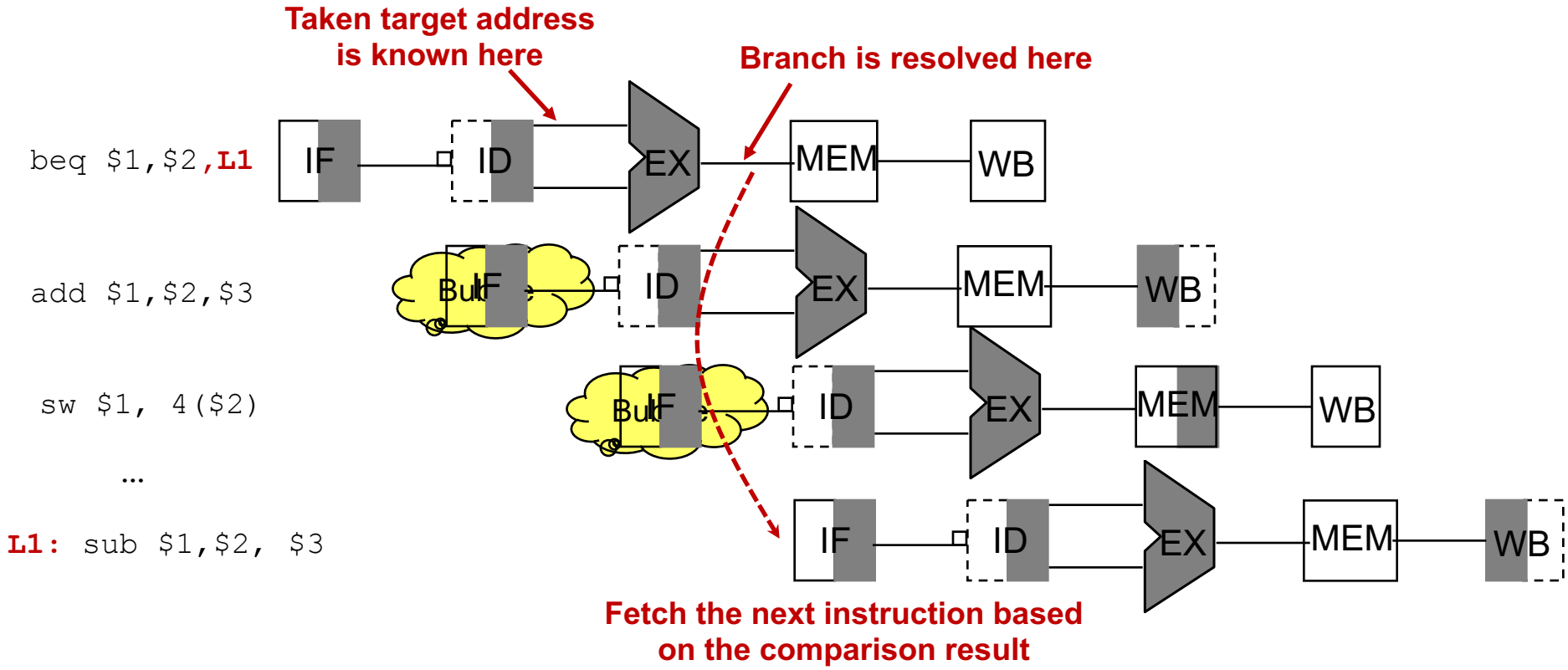
Data Hazard Detection Logic

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	ID/EX.IR[rt] == IF/ ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ ID.IR[rs]

The logic to detect the need for load interlocks during the ID stage of an instruction

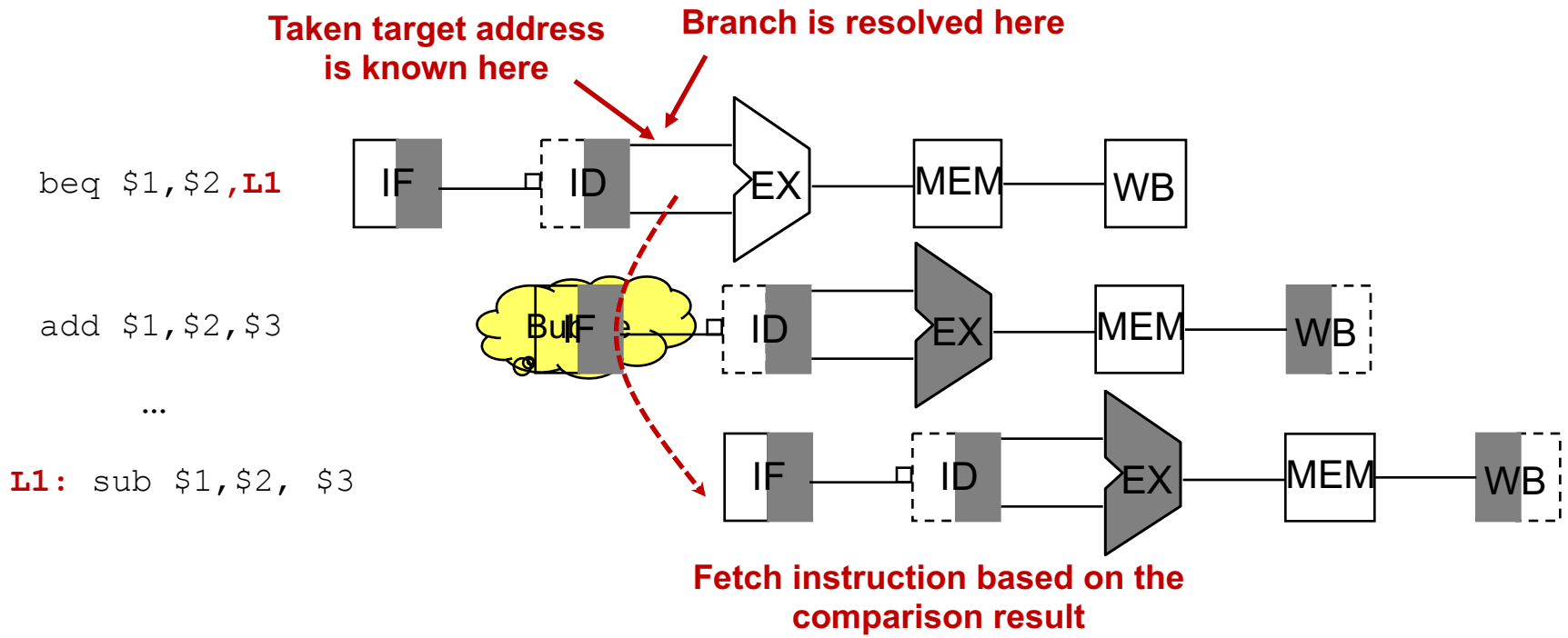
Control Hazard

- 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



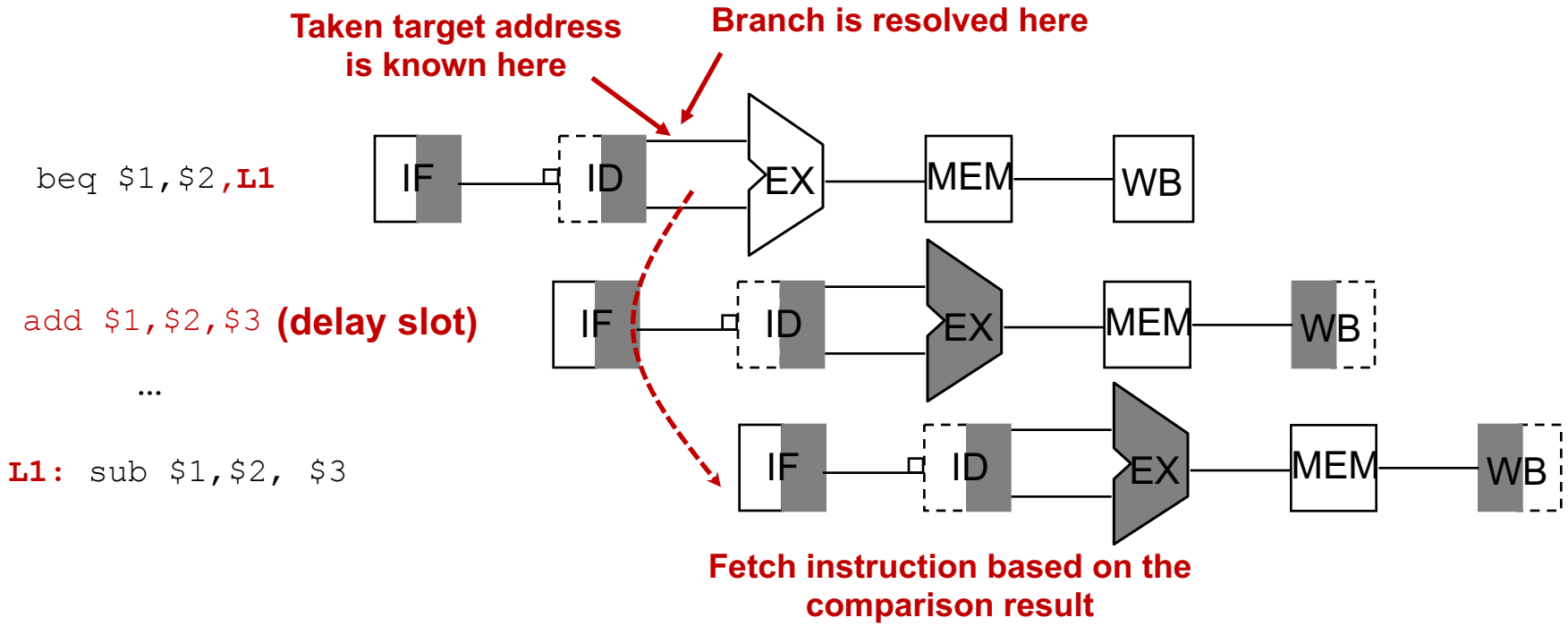
Reducing Control Hazard

- To reduce 2 bubbles to 1 bubble, add hardware in ID stage to compare registers (and generate branch condition)



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, $3 = 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

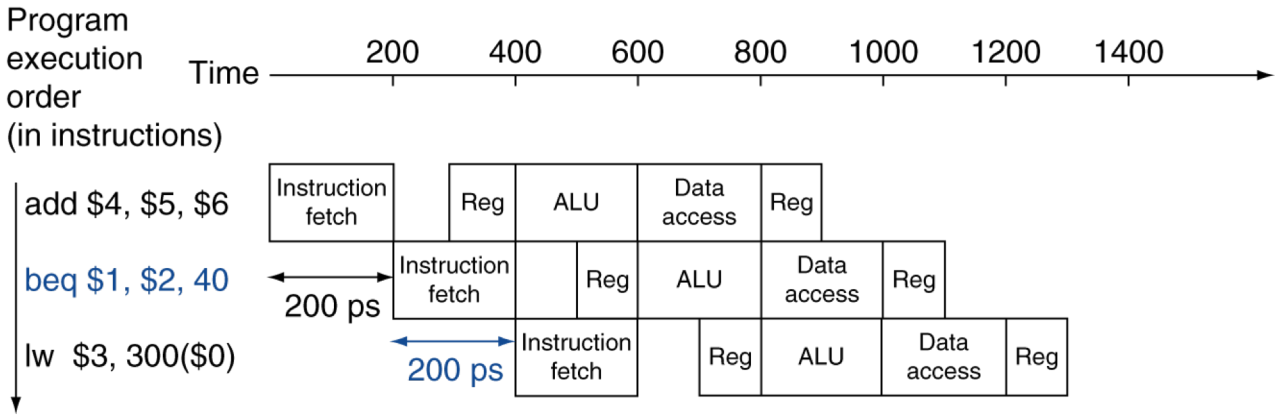
Branch Prediction

- 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

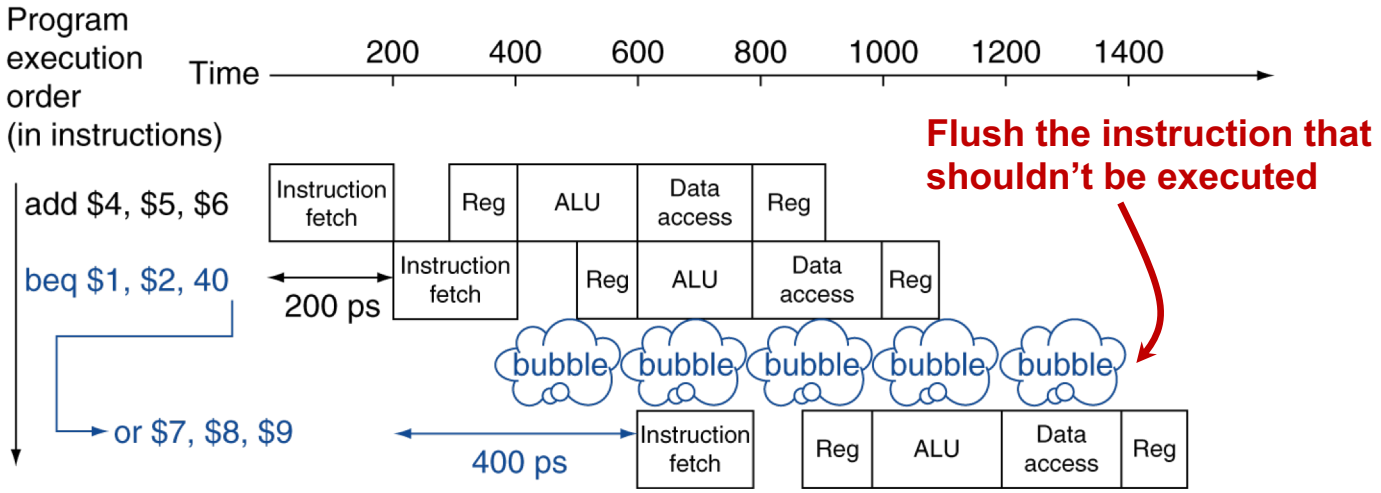
- **Solution: Branch Prediction**
  - Predict the branch outcome in hardware
  - **Flush** the instructions (that shouldn't have been executed) in the pipeline if the prediction turns out to be wrong
  - Modern processors use **sophisticated branch predictors**

MIPS with Predict-Not-Taken

Prediction correct

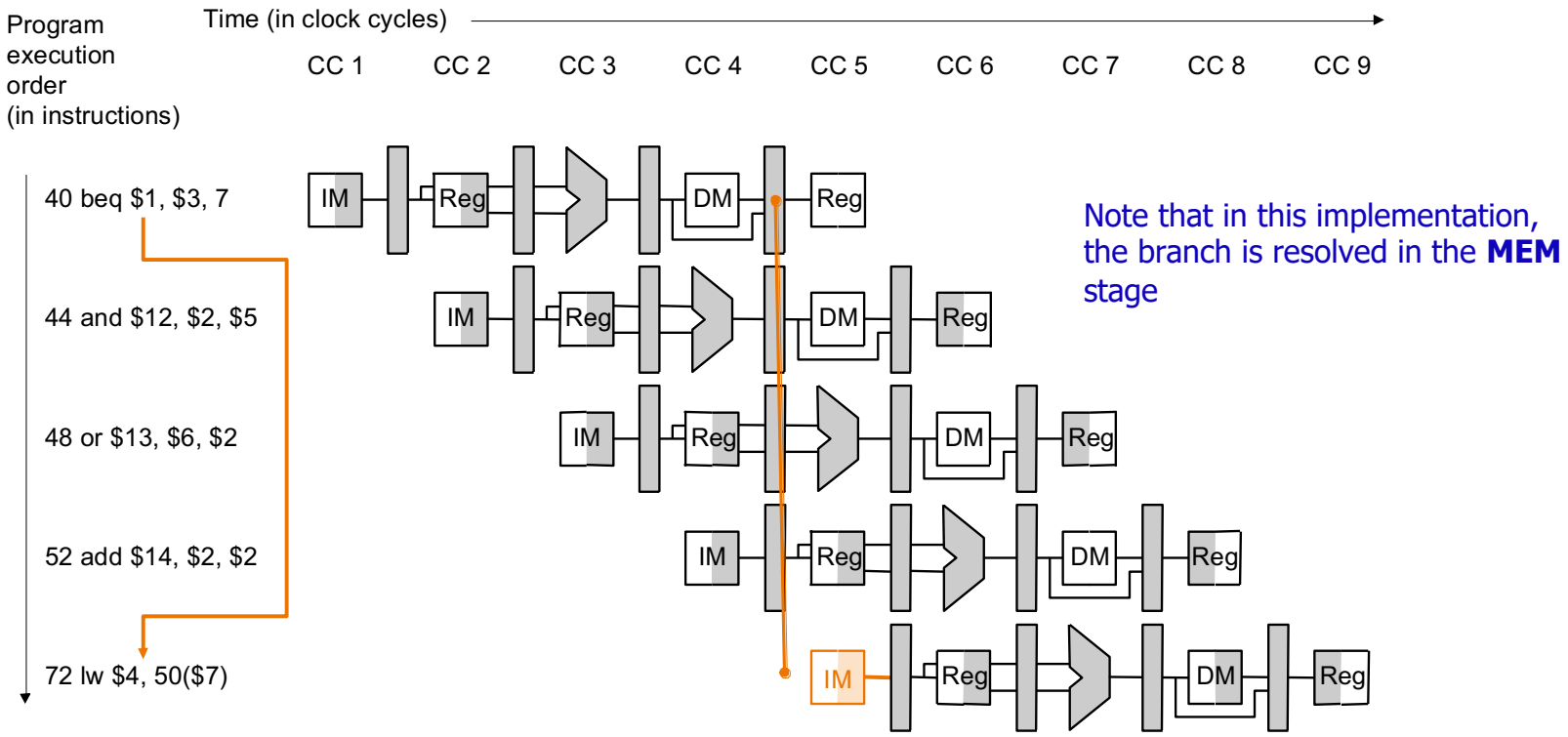


Prediction incorrect



Control Hazards - Branch

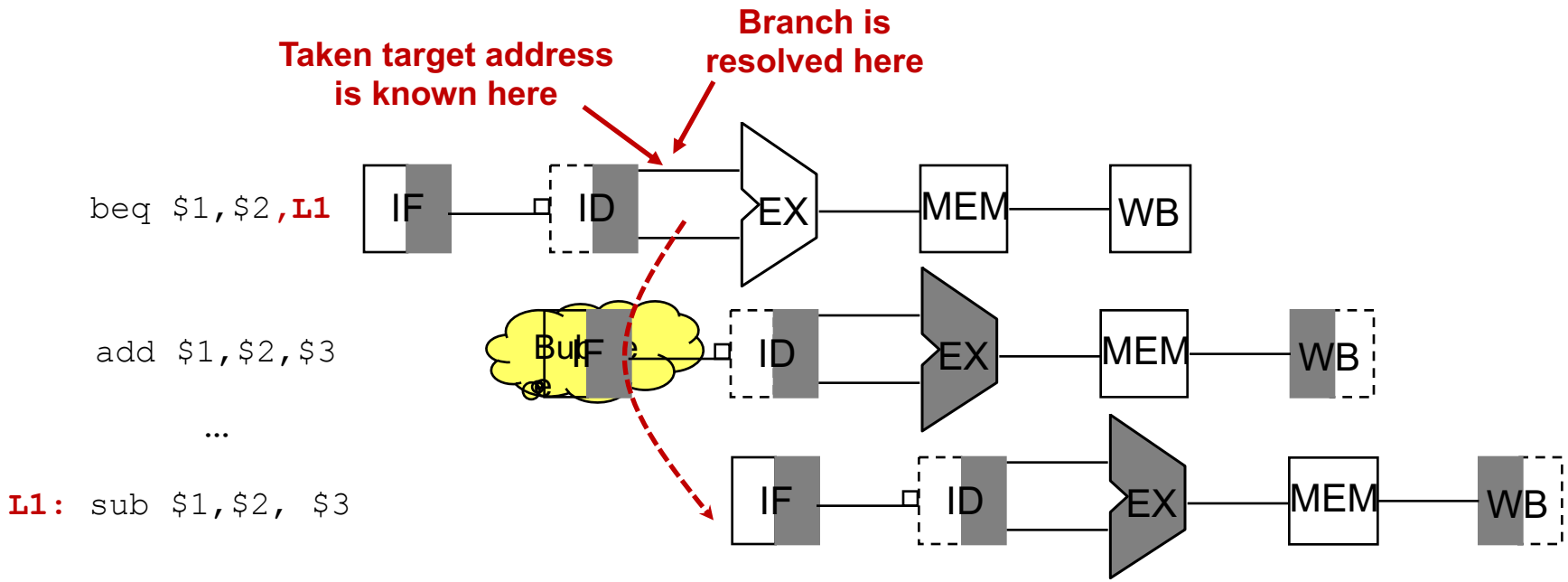
- When the branch condition is resolved, other instructions are in the pipeline



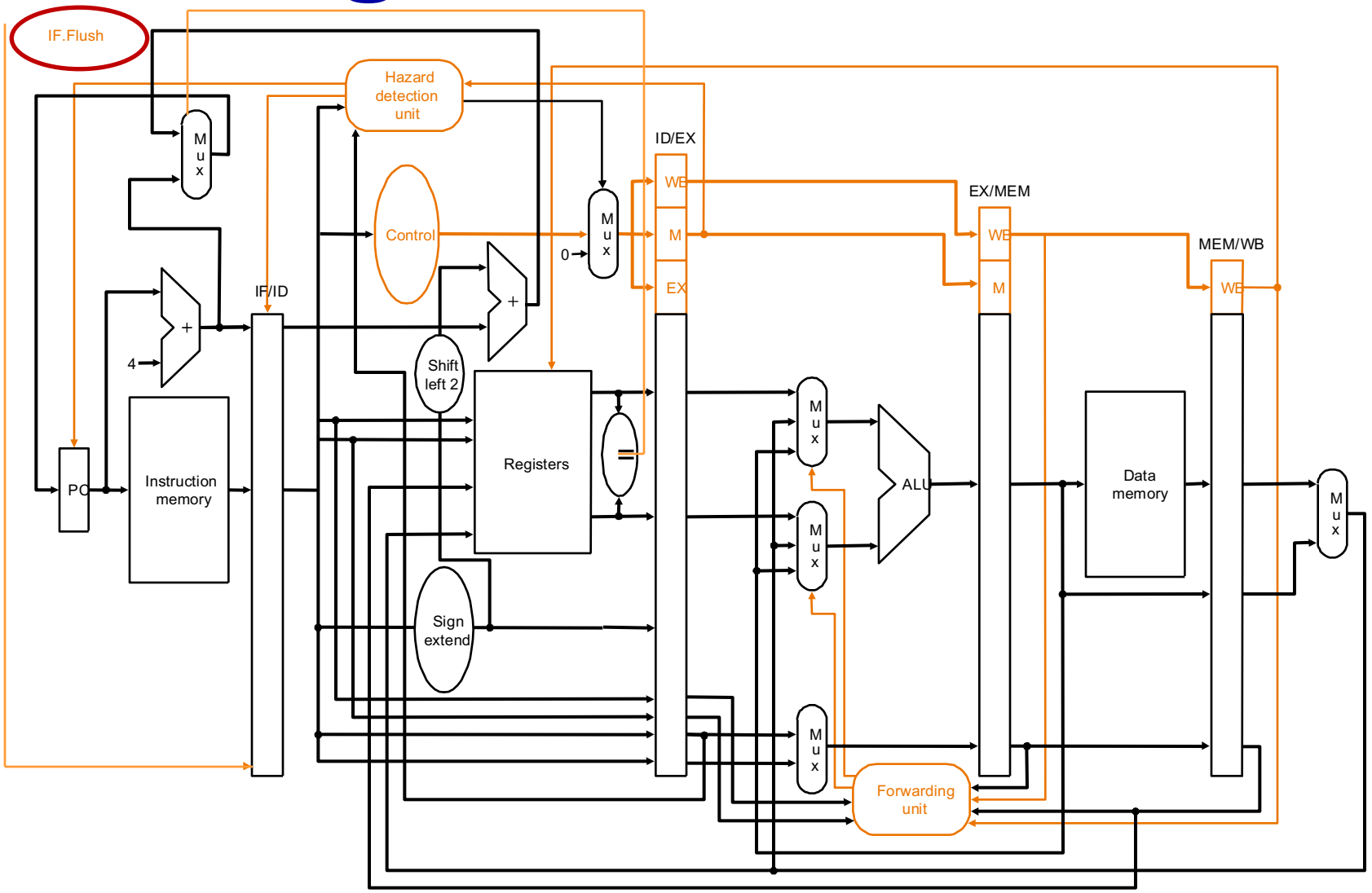
- We are predicting "branch not taken"
- If we are wrong (if branch is taken), flush instructions

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

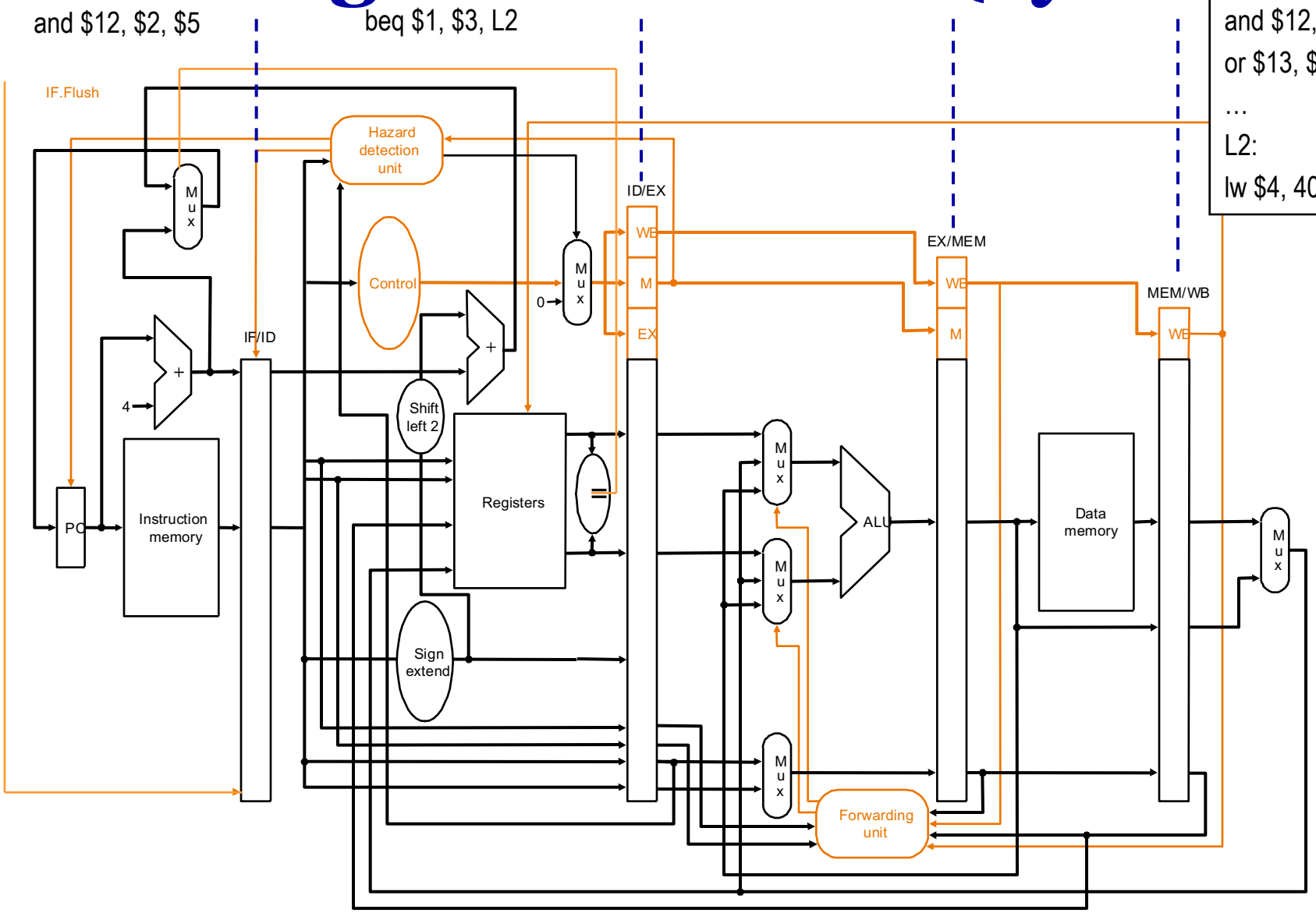


Flushing Instructions



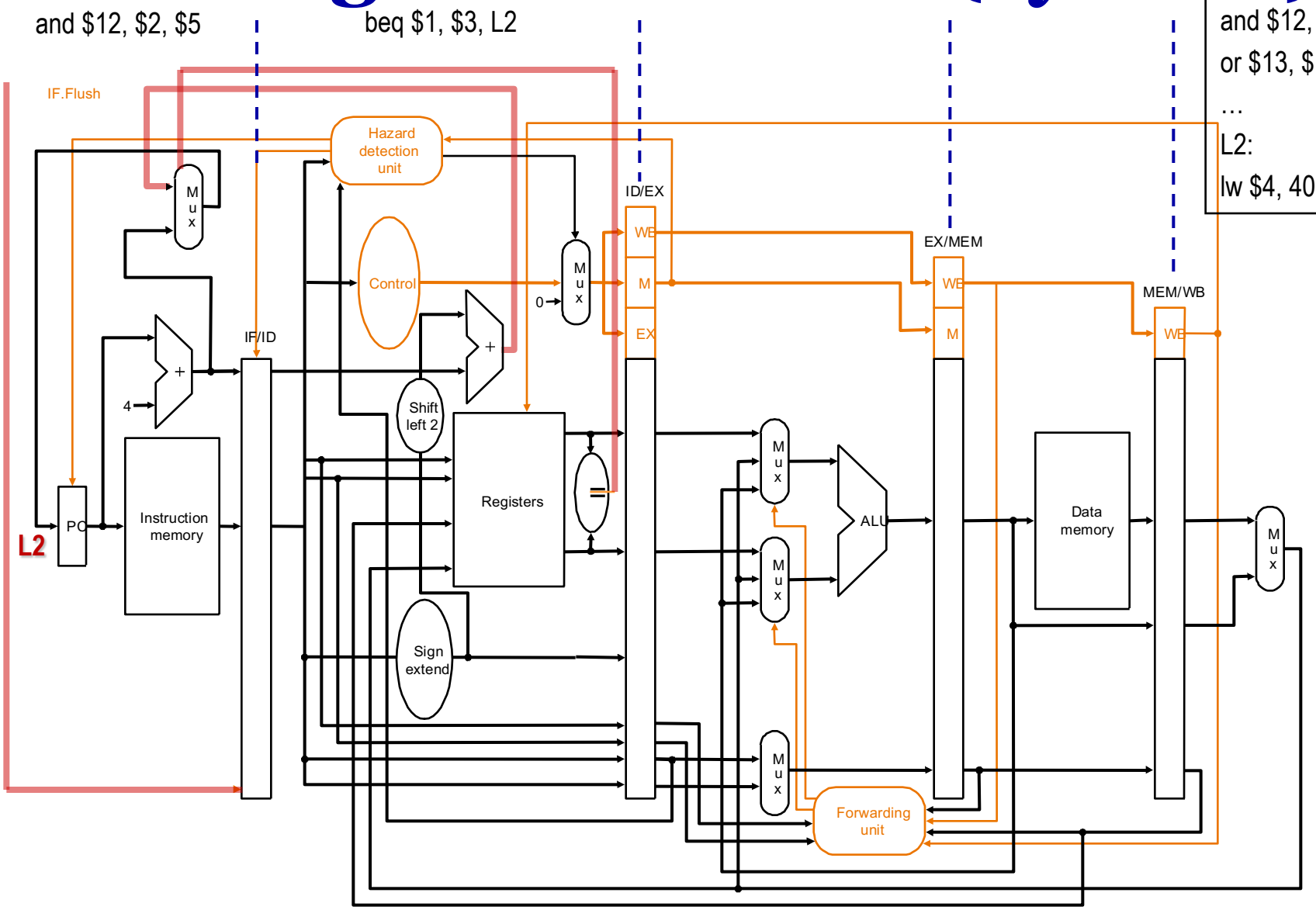
Flushing Instructions (cycle N)

beq $1, $3, L2
and $12, $2, $5
or $13, $12, $1
…
L2:
lw $4, 40($7)

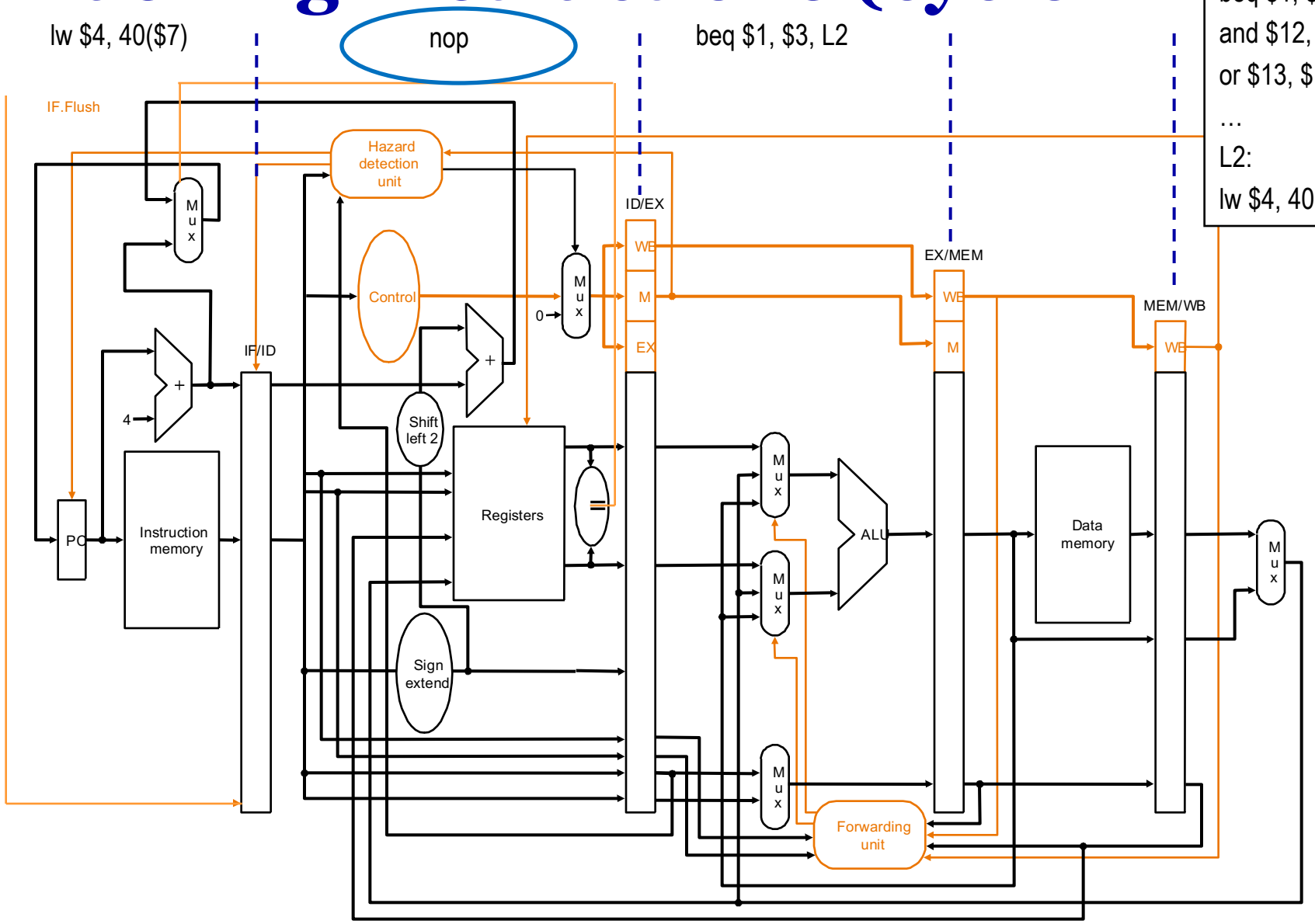


Flushing Instructions (cycle N)

beq $1, $3, L2
and $12, $2, $5
or $13, $12, $1
…
L2:
lw $4, 40($7)



Flushing Instructions (cycle N+1)



Outline

- Introduction to Pipelining
- How Pipeline is Implemented
- Pipeline Hazards
- C.4 Exceptions
- Handling Multicycle Operations

Exceptions

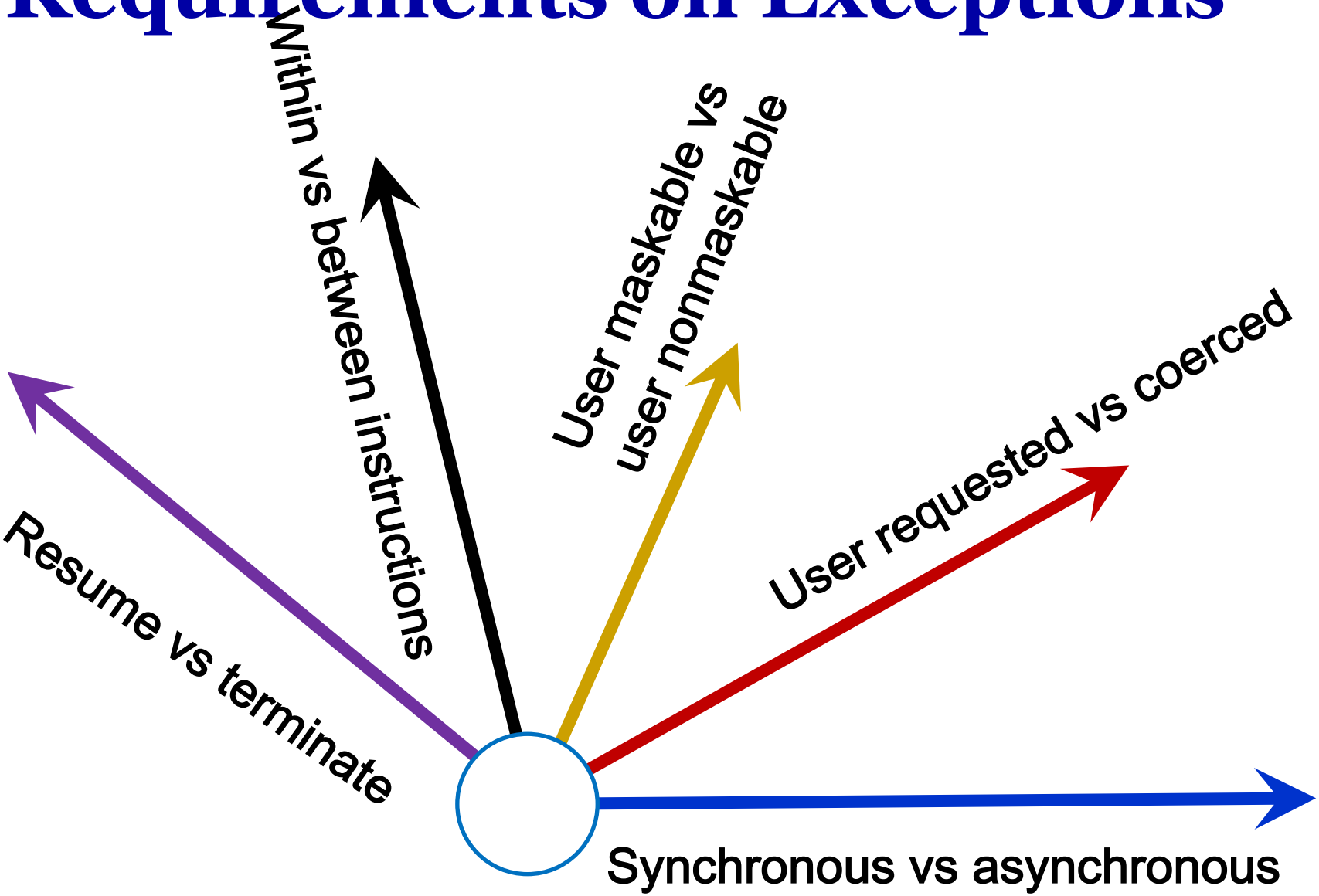
- **Exceptions** describe those situations where the normal execution order of instruction is changed!
  - may force the CPU to **abort** the *instructions in the pipeline* before they complete!

- Some other used terminologies for "exception"
  - Interrupt
  - Fault

Types of Exceptions

- I/O device request
- Invoking an OS service for a user program
- Tracing instruction execution
- Breakpoint (programmer requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses
- Memory protection violation
- Using an undefined instruction
- Hardware malfunctions
- Power failure

Requirements on Exceptions



Classifications

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Stopping and Restarting Execution

- The most ***difficult*** exceptions have two properties

- **(1)** they occur **within instructions** (i.e., in the middle of the instruction execution corresponding to EX or MEM pipe stages

- **(2)** they must be **restartable**

Steps to Save Pipeline State

- **(1)** Force a **trap instruction** into the pipeline on the next IF
- **(2)** Until the trap is taken, turn off all writes *for the faulting instruction* and *for all instructions that follow in the pipeline*
  - This can be done by placing zeros into the pipeline latches of all instructions, starting with the instruction that generates the exception, but not those that precede that instruction
- **(3)** After the exception-handling routine in the OS receives control, it immediately *saves the PC of the faulting instruction*
  - This value will be used to return from the exception later

Precise vs Imprecise Exceptions

If the pipeline can be stopped so that *the instructions just before the faulting instruction* are completed and *those after it* can be restarted from scratch, the pipeline is staid to have precise exceptions

- Supporting precise exceptions is a requirement in many systems
- Any processor with demand paging or IEEE arithmetic trap handlers must make its exceptions precise

Exceptions in MIPS Pipeline

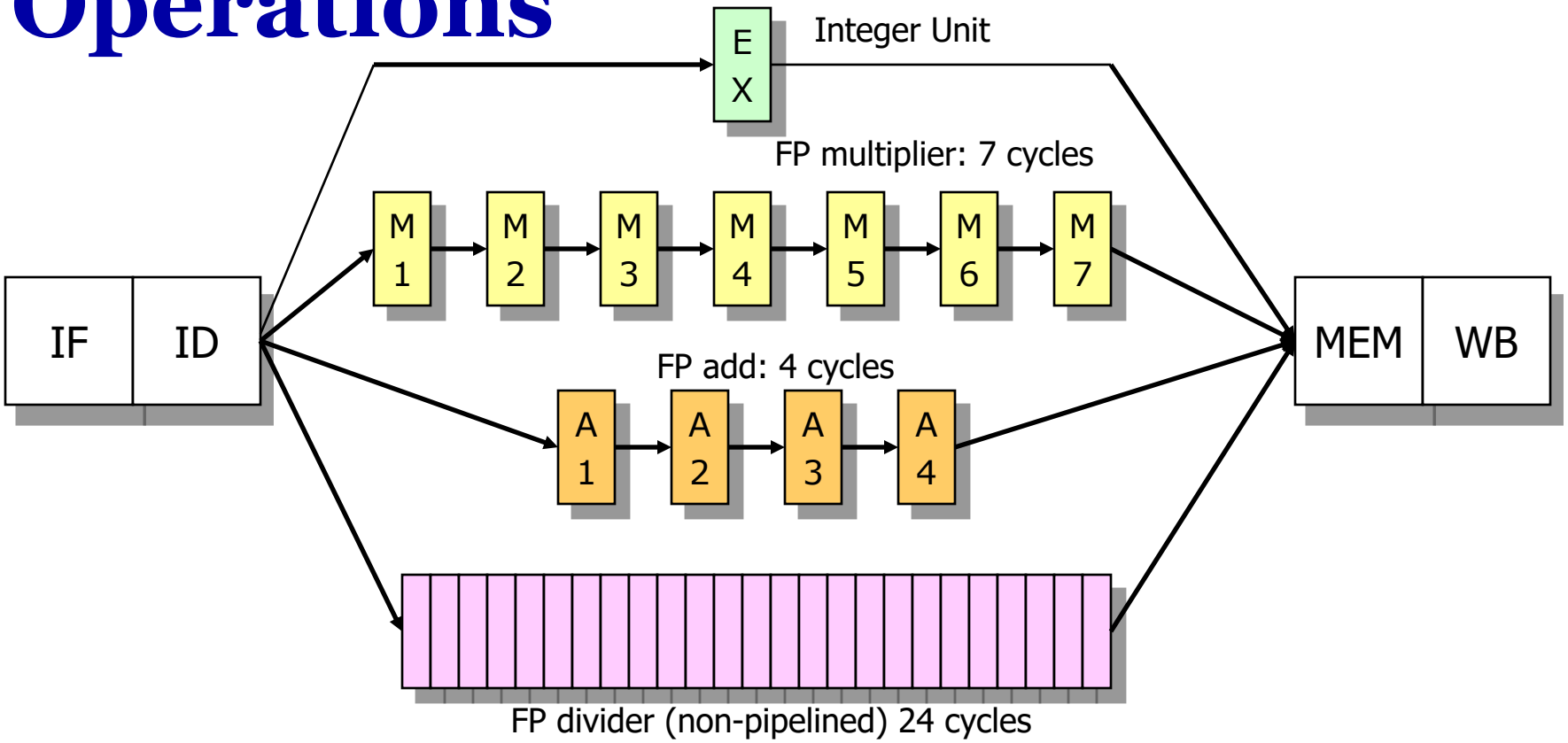
Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Exceptions may occur in different stages of a pipeline

Outline

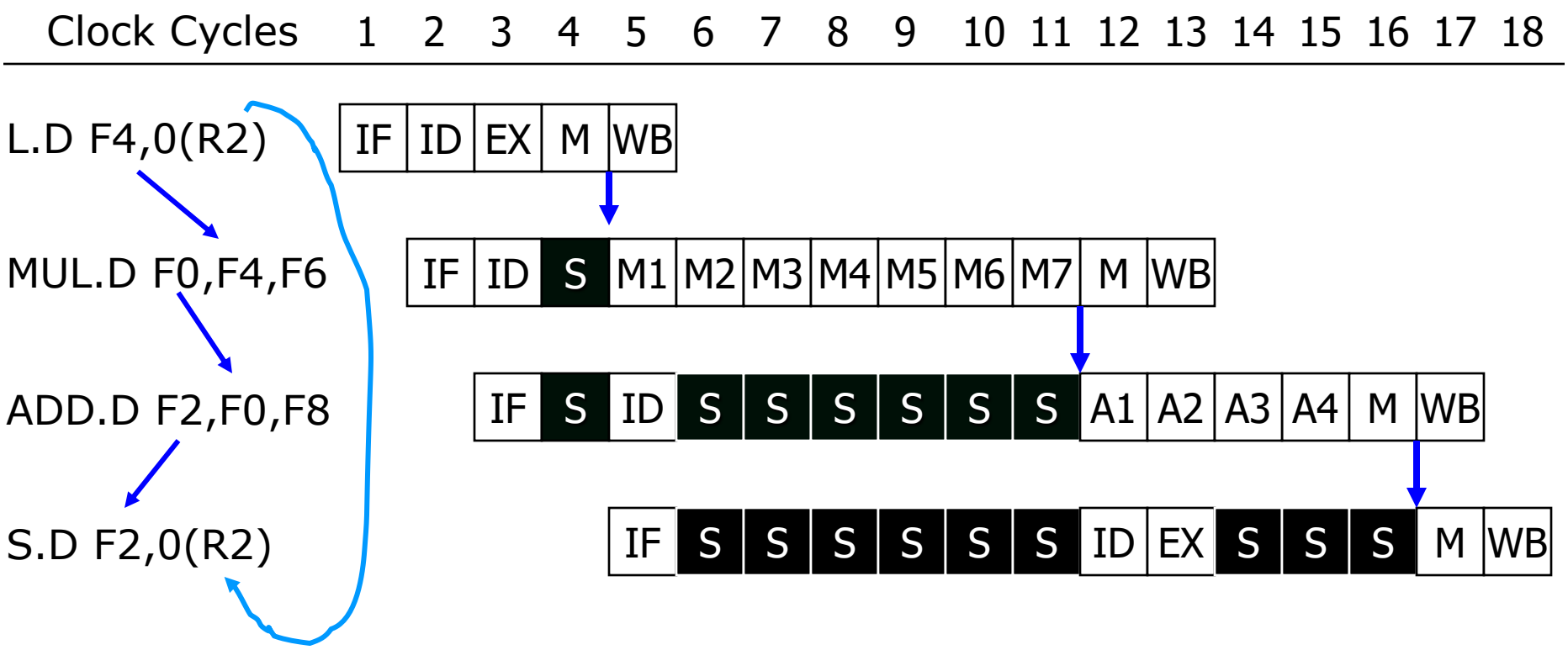
- Introduction to Pipelining
- How Pipeline is Implemented
- Pipeline Hazards
- Exceptions
- C.5 Handling Multicycle Operations

Supporting Multiple FP Operations

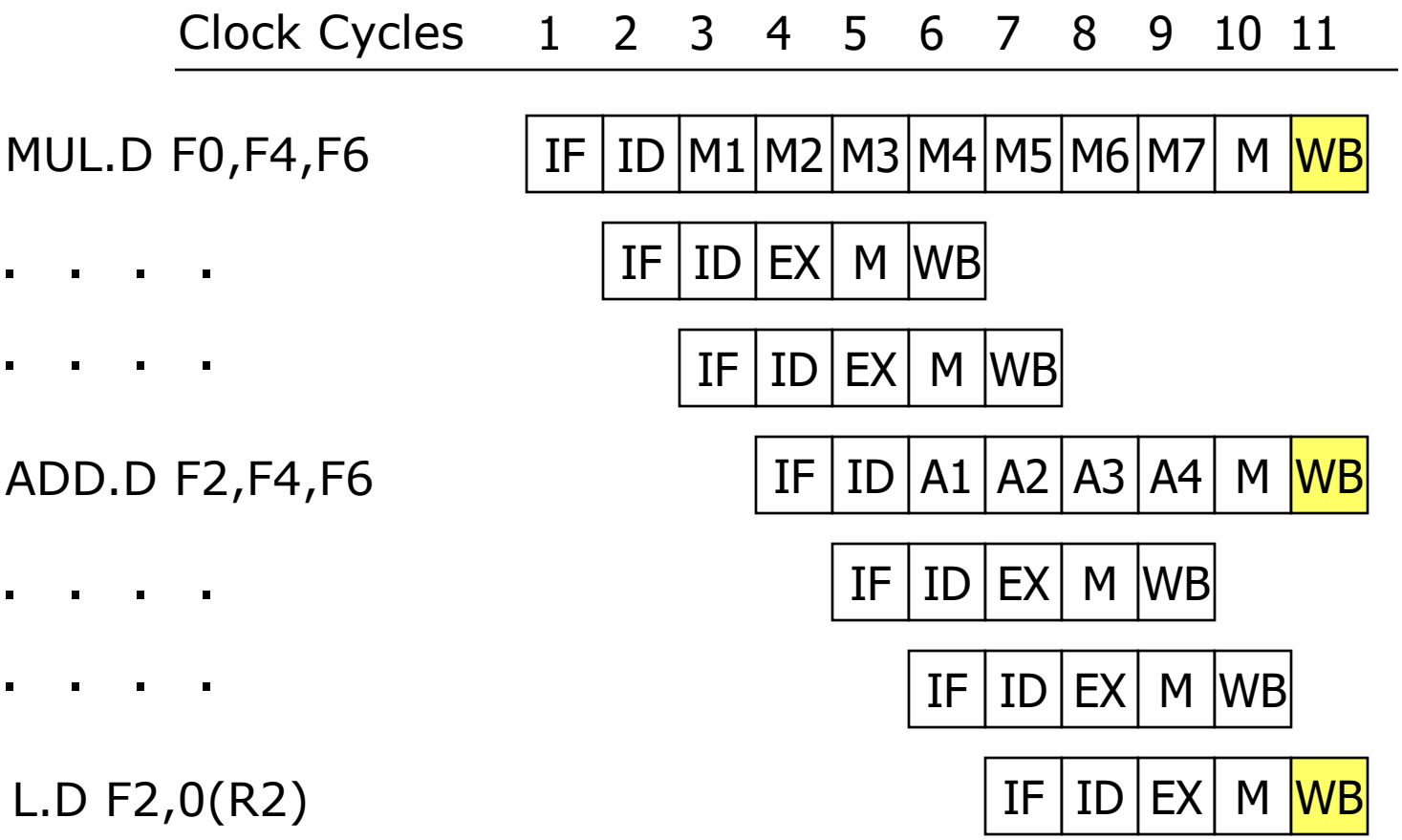


- **Complicate bypass or forwarding**
- **Potential structural hazard**
- **Multiple (FP) instructions can complete at the same time**
  - **RF might need to be multi-ported**
  - **Ordering issue, who gets to update the register?**
- **Out-of-order completion/retirement: Precise exception issue**

Bypassing & Forwarding



Structural Hazards



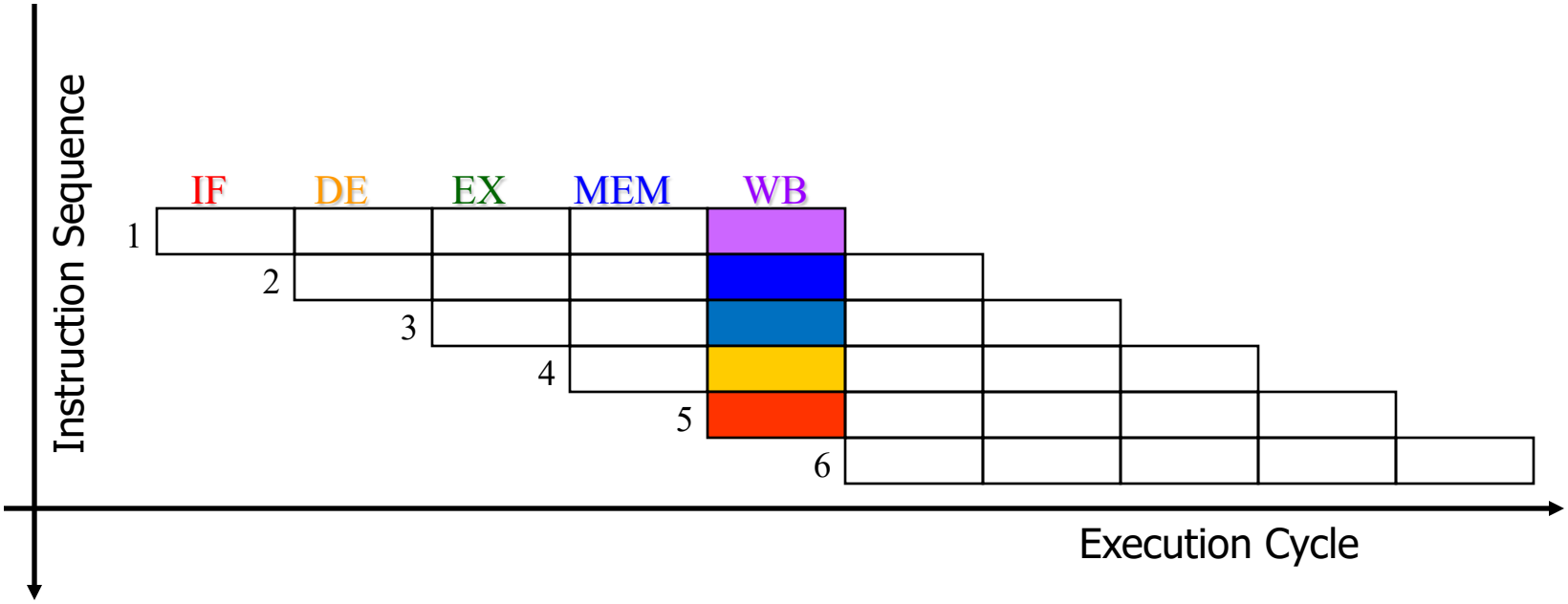
- Write to register file at the same cycle (cc11)
- Write to the same register (WAW)
- MEM in cc10

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

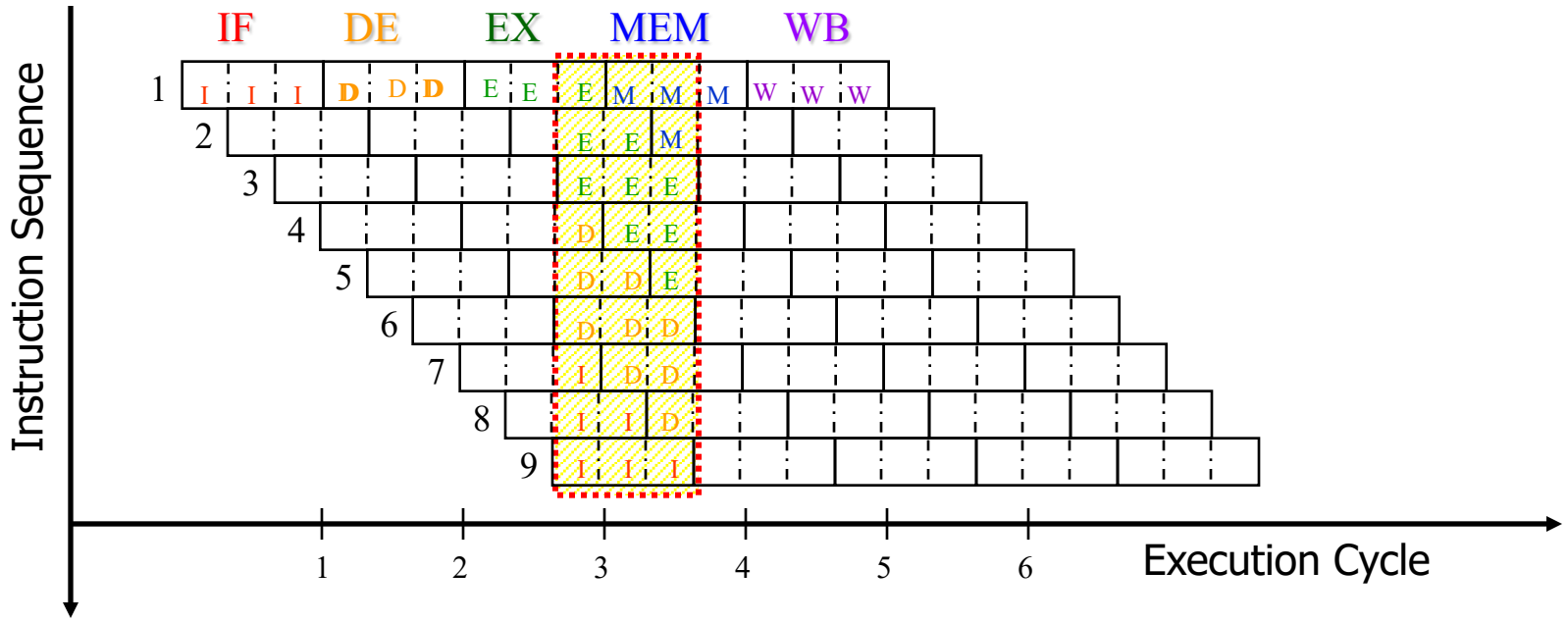
Scalar Pipeline (Baseline)



Modified from Prof Sean Lee's Slide

Superpipeline

- Deeper pipelining is called superpipelining
- Deeper pipeline allows for achieving higher clock rates



Modified from Prof Sean Lee's Slide