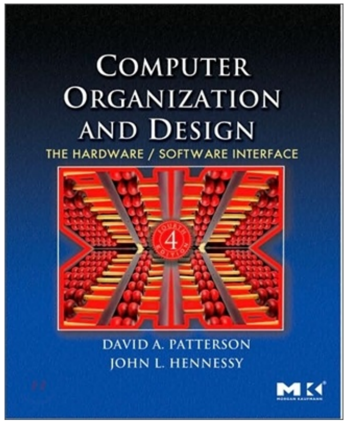
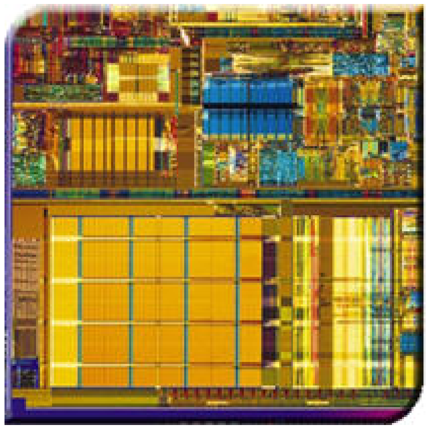
# Computer Architecture

## Lecture 7  Pipelining



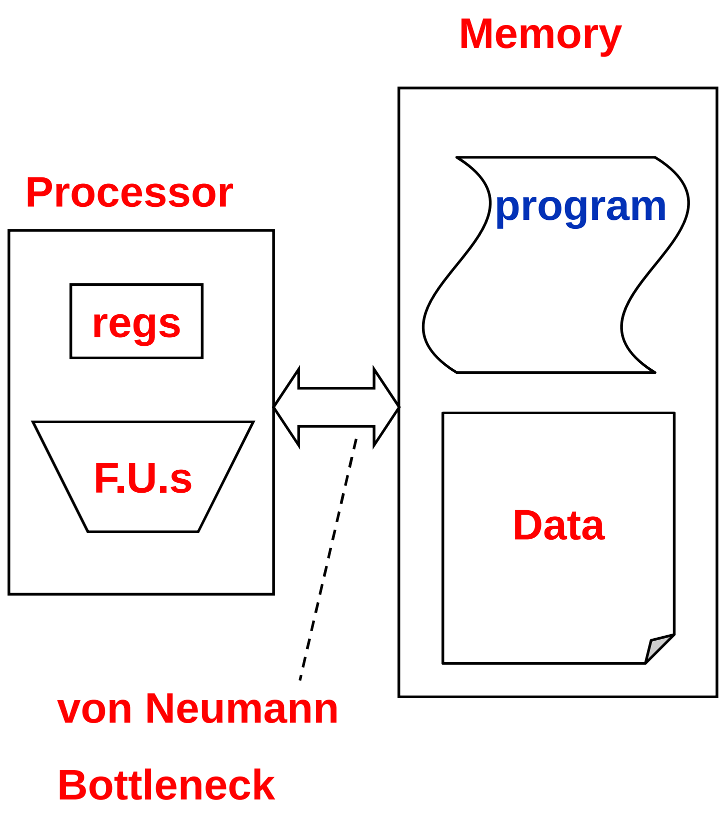



**Prof. Jongmyon Kim**

울산대학교
UNIVERSITY OF ULSAN

# Fundamental Execution Cycle

| | |
|---|---|
| ***Instruction Fetch*** | **Obtain instruction from program storage** |
| ***Instruction Decode*** | **Determine required actions and instruction size** |
| ***Operand Fetch*** | **Locate and obtain operand data** |
| ***Execute*** | **Compute result value or status** |
| ***Result Store*** | **Deposit results in storage for later use** |
| ***Next Instruction*** | **Determine successor instruction** |



# Why Pipelining?

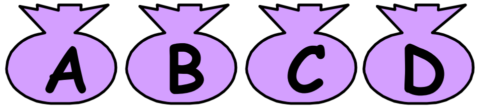
| Instruction Class | Instruction Fetch | Register Read | ALU Operation | Data Access | Register Write | Total Time |
|---|---|---|---|---|---|---|
| Load word | 2ns | 1ns | 2ns | 2ns | 1ns | 8ns |
| Store word | 2ns | 1ns | 2ns | 2ns | | 7ns |
| R-format | 2ns | 1ns | 2ns | | 1ns | 6ns |
| Branch | 2ns | 1ns | 2ns | | | 5ns |

- Single cycle datapath
  - Design for the worst case
  - Need to make the cycle time = 8ns per cycle
- Multi-cycle datapth
  - Design for each individual instruction class
  - For the above example: cycle time = 2ns
  - Lw=10ns (5 cycles), sw=8ns (4 cycles), R-format=8ns(4 cycles), beq=6ns (3 cycles)
- Can we do better?

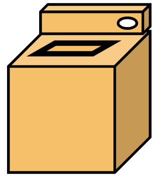
# Pipelining: It's Natural!

- Laundry Example

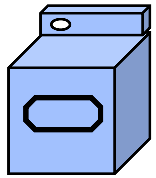
- Ann, Brian, Cathy, Dave
  each have one load of clothes
  to wash, dry, and fold




- Washer takes 30 minutes

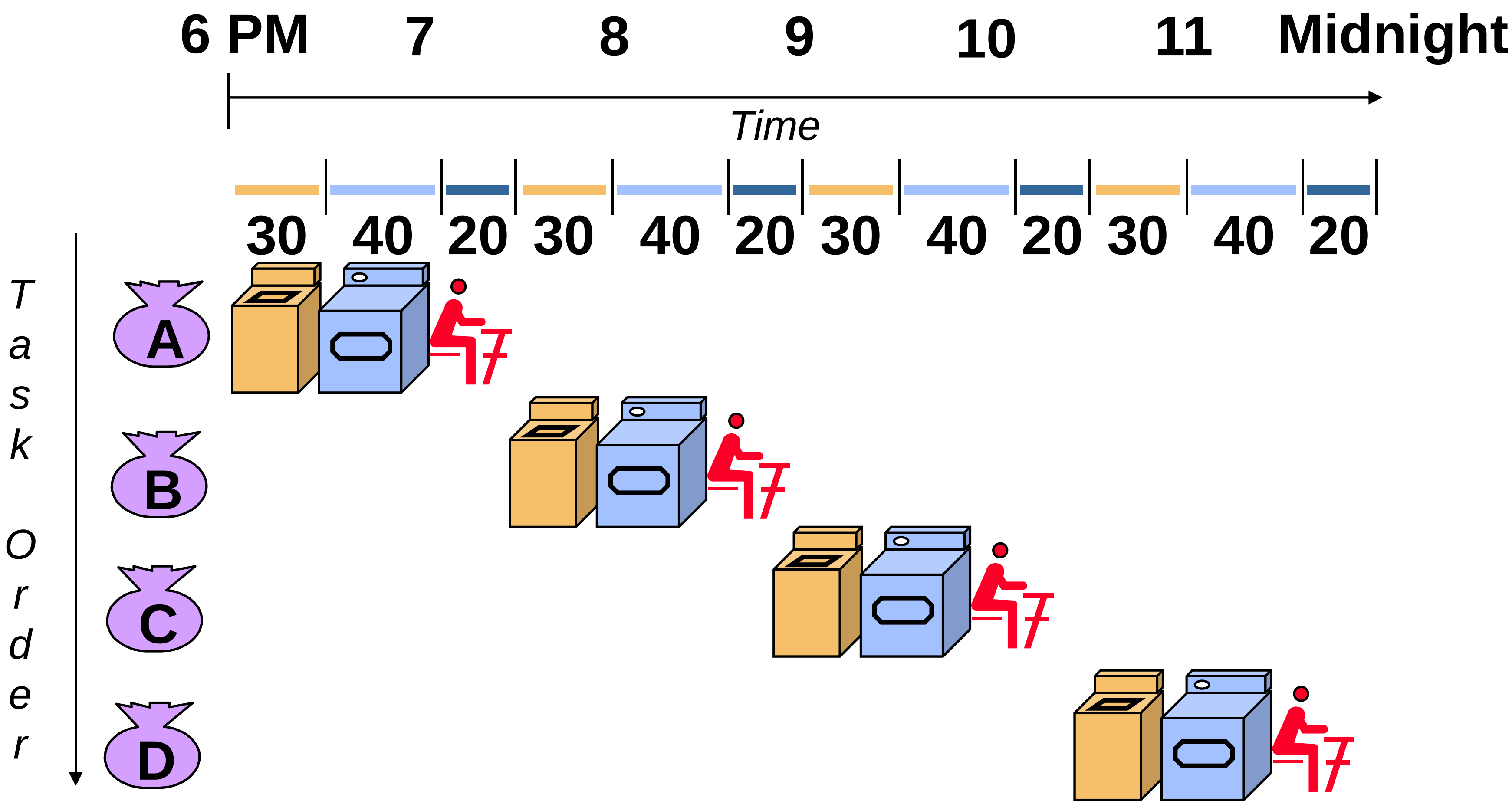


- Dryer takes 40 minutes



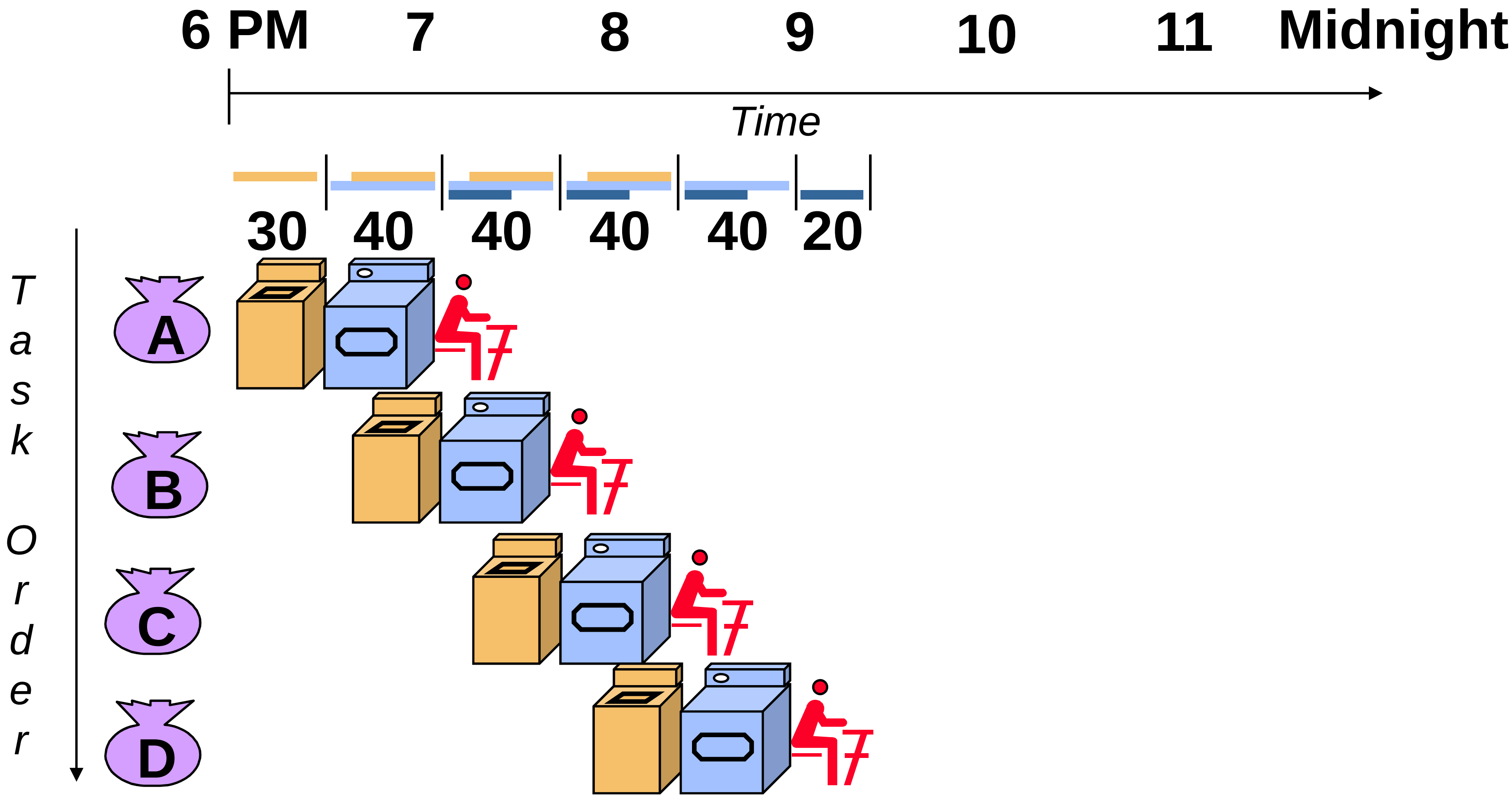
- "Folder" takes 20 minutes

# Sequential Laundry



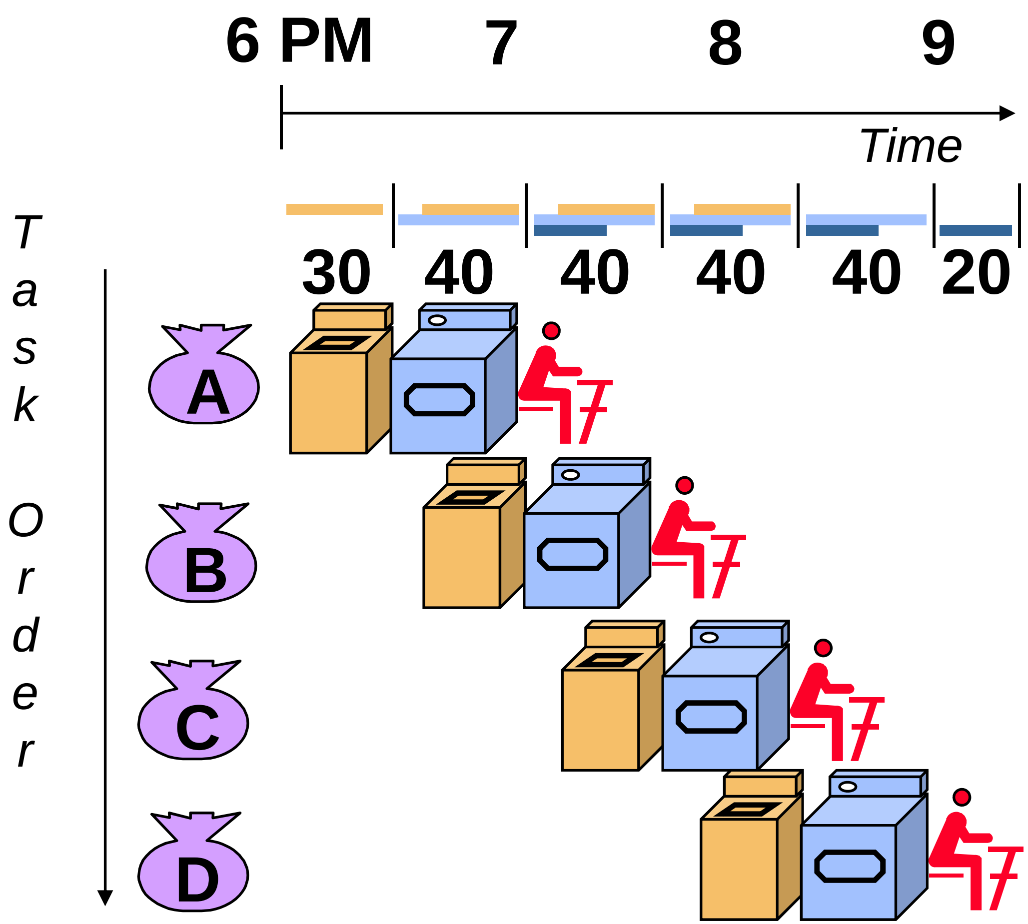

- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Pipelined Laundry: Start work ASAP




- Pipelined laundry takes 3.5 hours for 4 loads

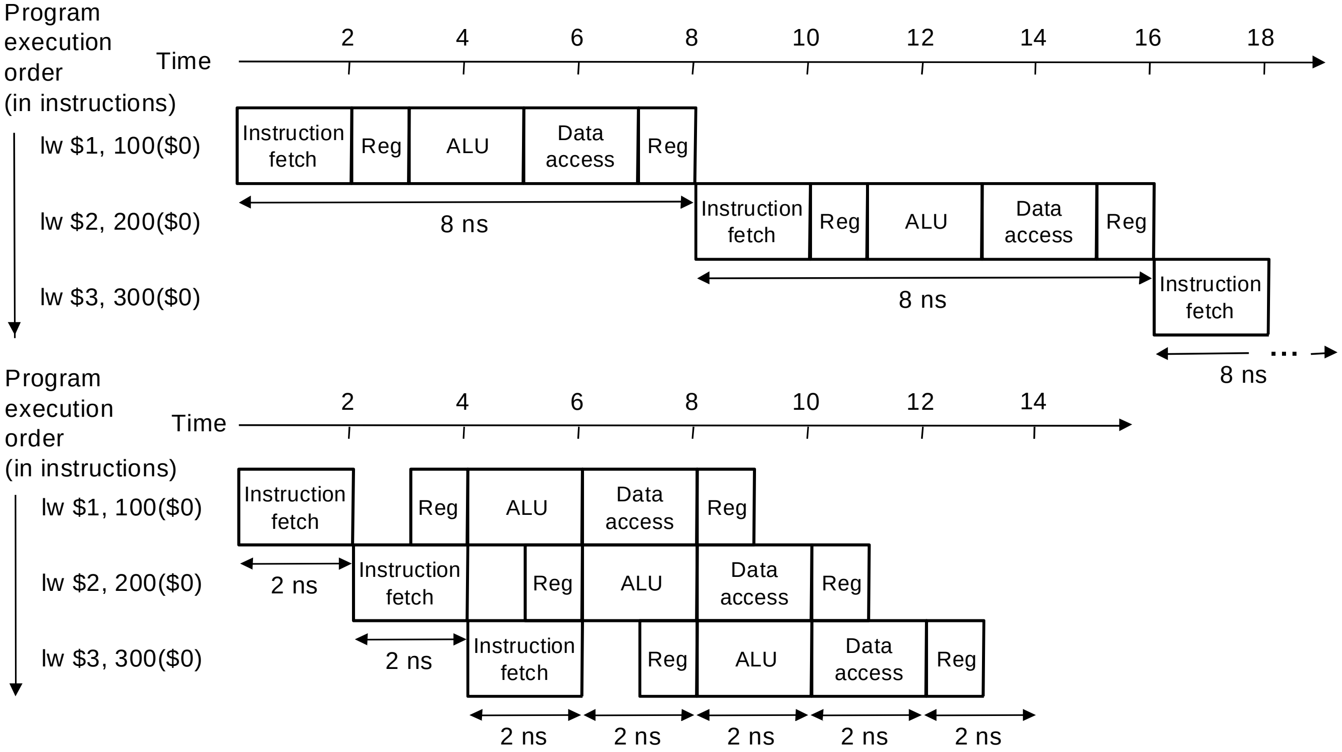
# Pipelining Lessons




- Pipelining doesn't help latency of a single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

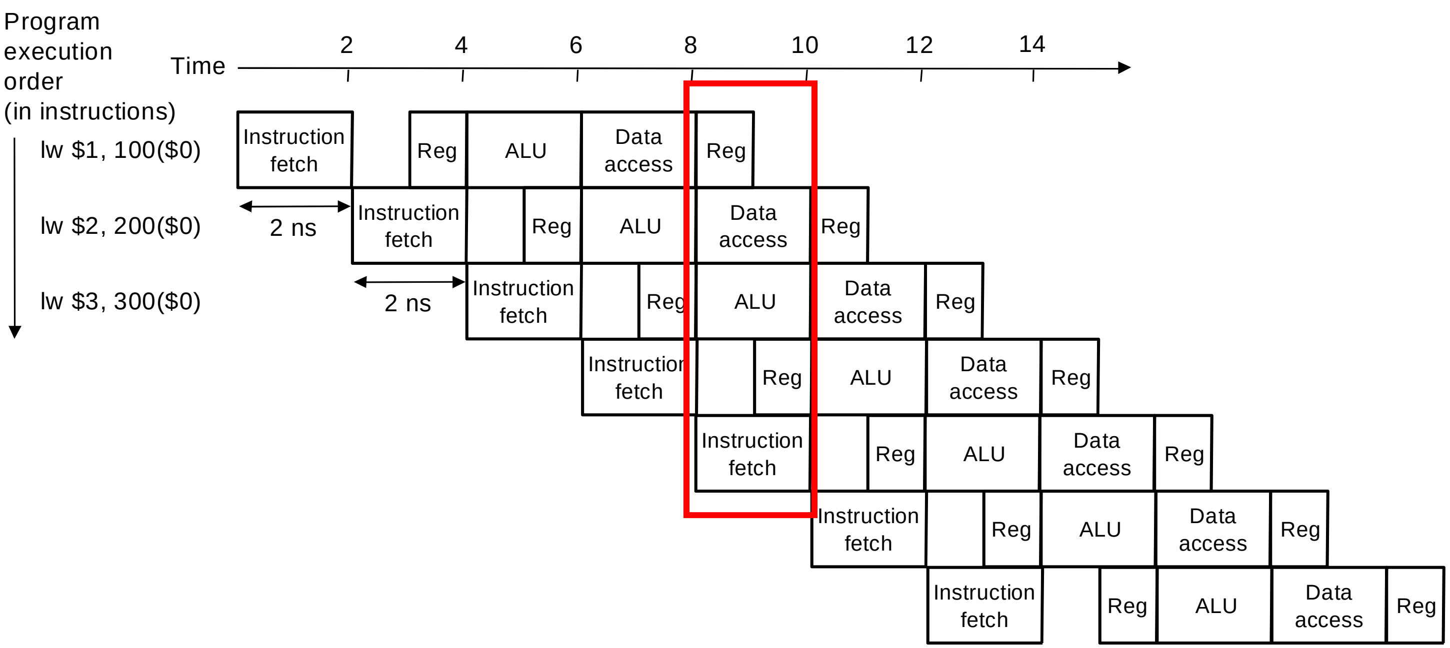
# Pipelining

- Improve performance by increasing instruction throughput




*Ideal speedup is number of stages in the pipeline. Do we achieve this?*

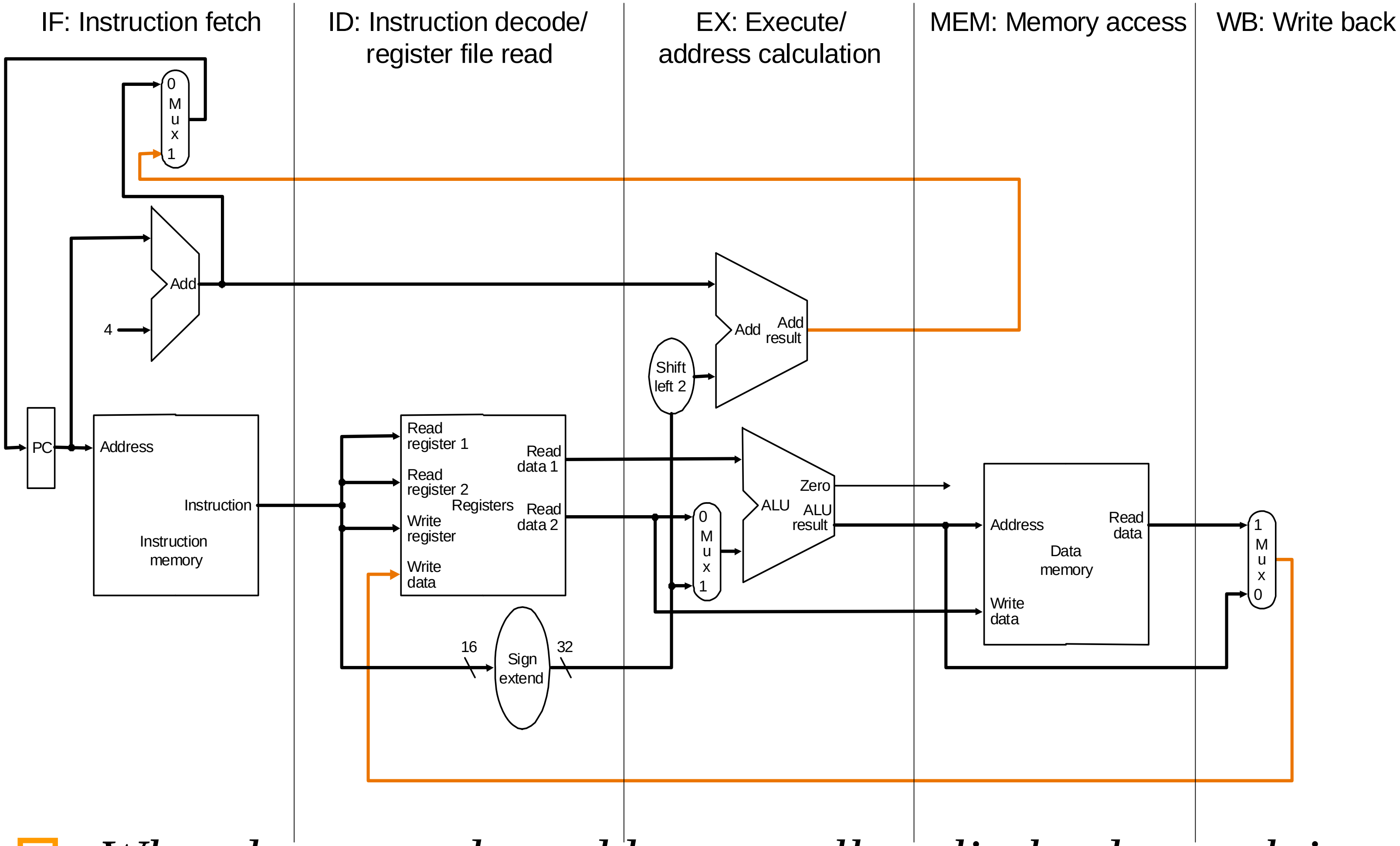
# Ideal Pipelining



# Pipelining

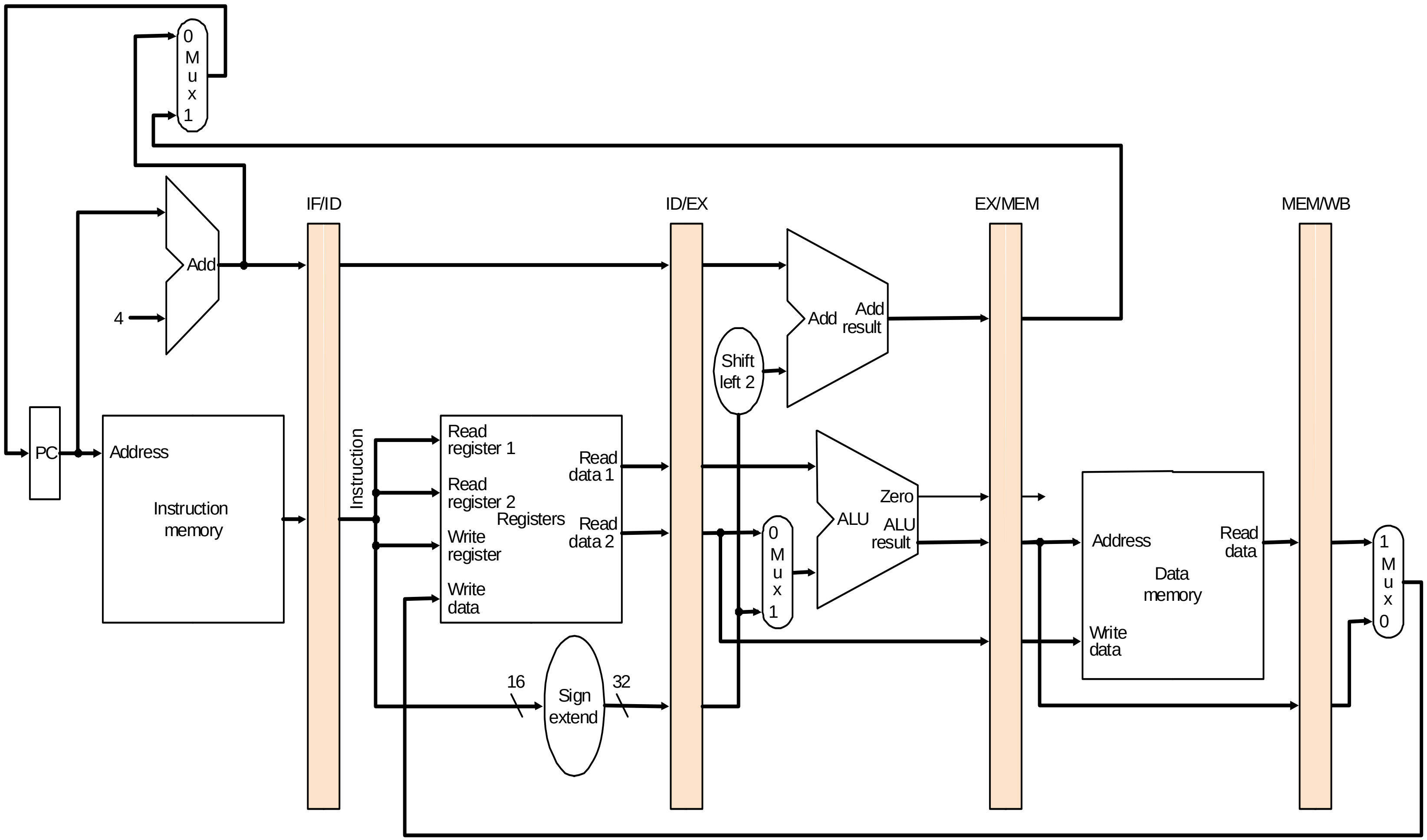
- What makes it easy
  - All instructions are the same length
  - Simple instruction formats
  - Memory operands appear only in loads and stores
- What makes it hard?
  - structural hazards: suppose we had only one memory
  - control hazards: need to worry about branch instructions
  - data hazards: an instruction depends on a previous instruction
- We'll build a simple pipeline and look at these issues
- We'll talk about modern processors and what really makes it hard:
  - exception handling
  - trying to improve performance with out-of-order execution, etc.

# Basic Idea

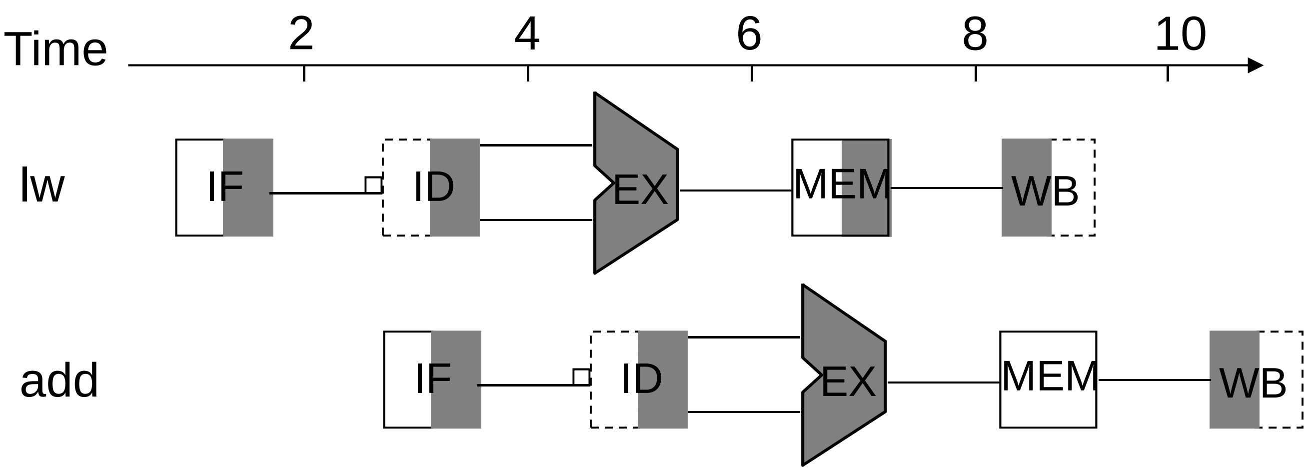



- *What do we need to add to actually split the datapath into stages?*

# Pipelined Datapath



# Graphically Representing Pipelines

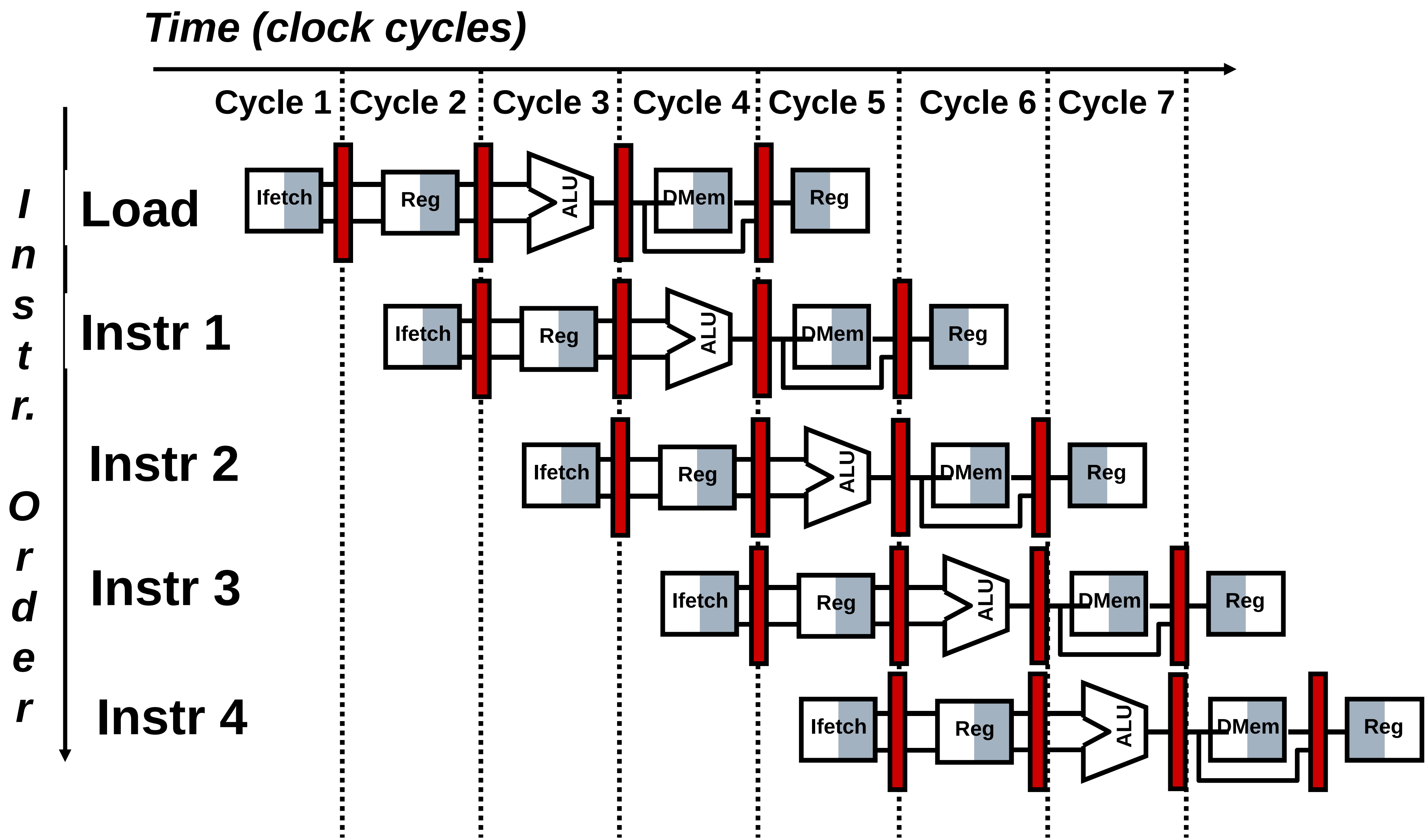



- Shading indicates the unit is being used by the instruction
- Shading on the right half of the register file (ID or WB) or memory means the element is being read in that stage
- Shading on the left half means the element is being written in that stage

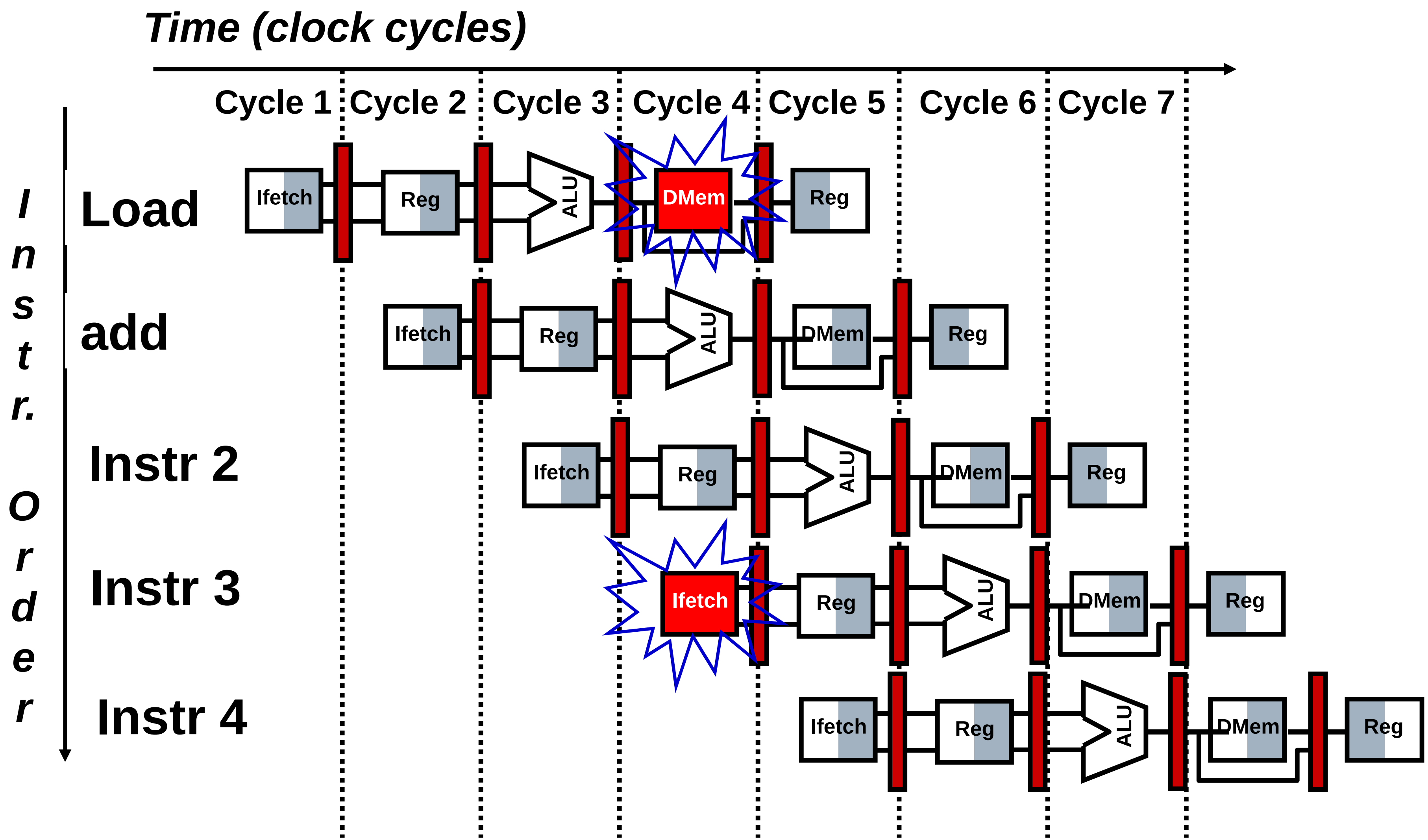
# Pipelining is not quite that straightforward !

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

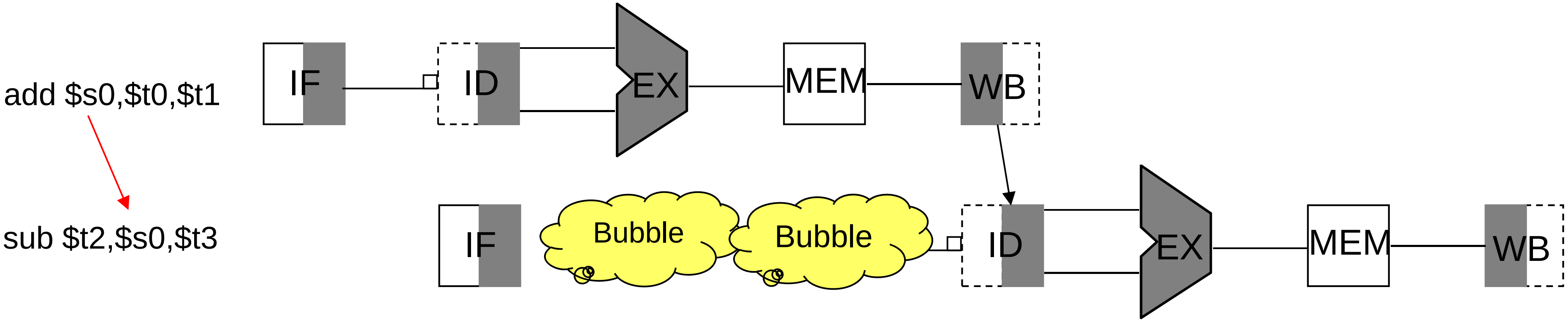
# Single Memory Port



# Single Memory Port / Structural Hazard

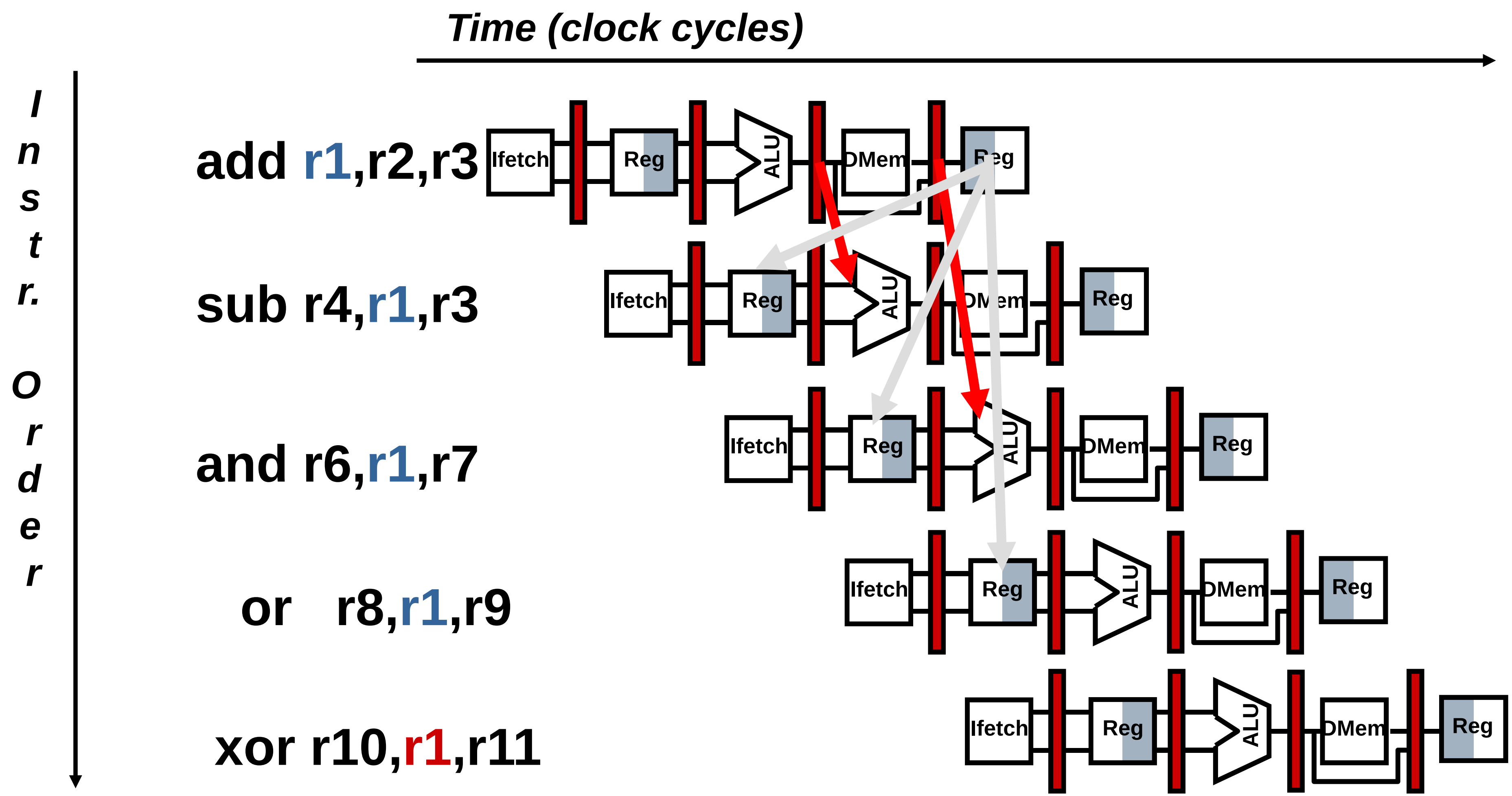


# Data Hazard

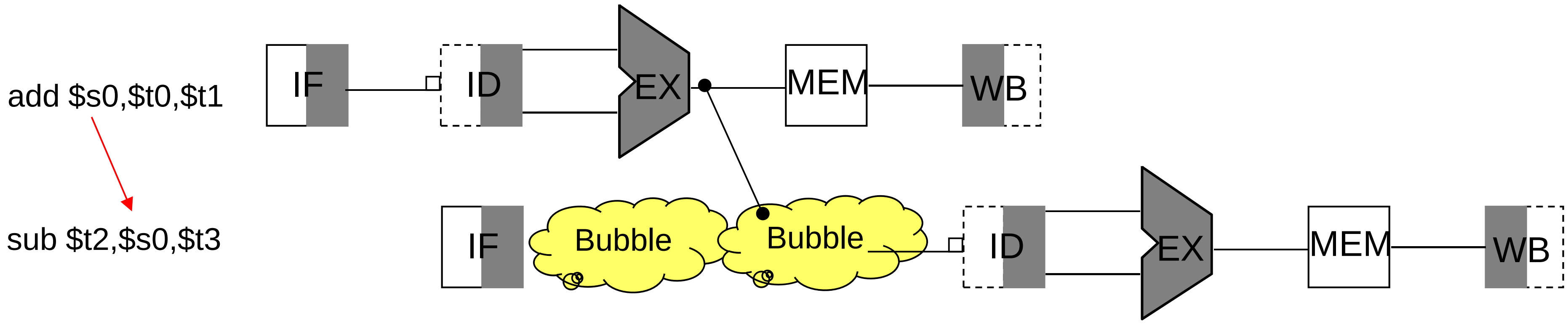



- Data is not ready for the subsequent dependent instruction
- Pipeline stall, typically referred to as “bubble”
- Performance is penalized
- One solution — forwarding (or bypassing)

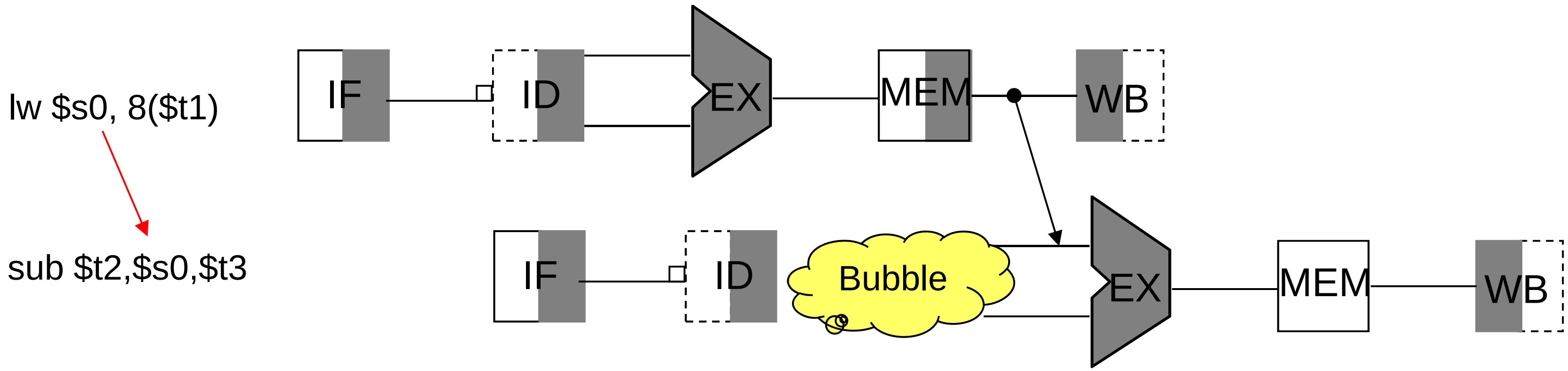
# Forwarding to Avoid Data Hazard



# Reducing Data Hazard: Forwarding (Bypassing)

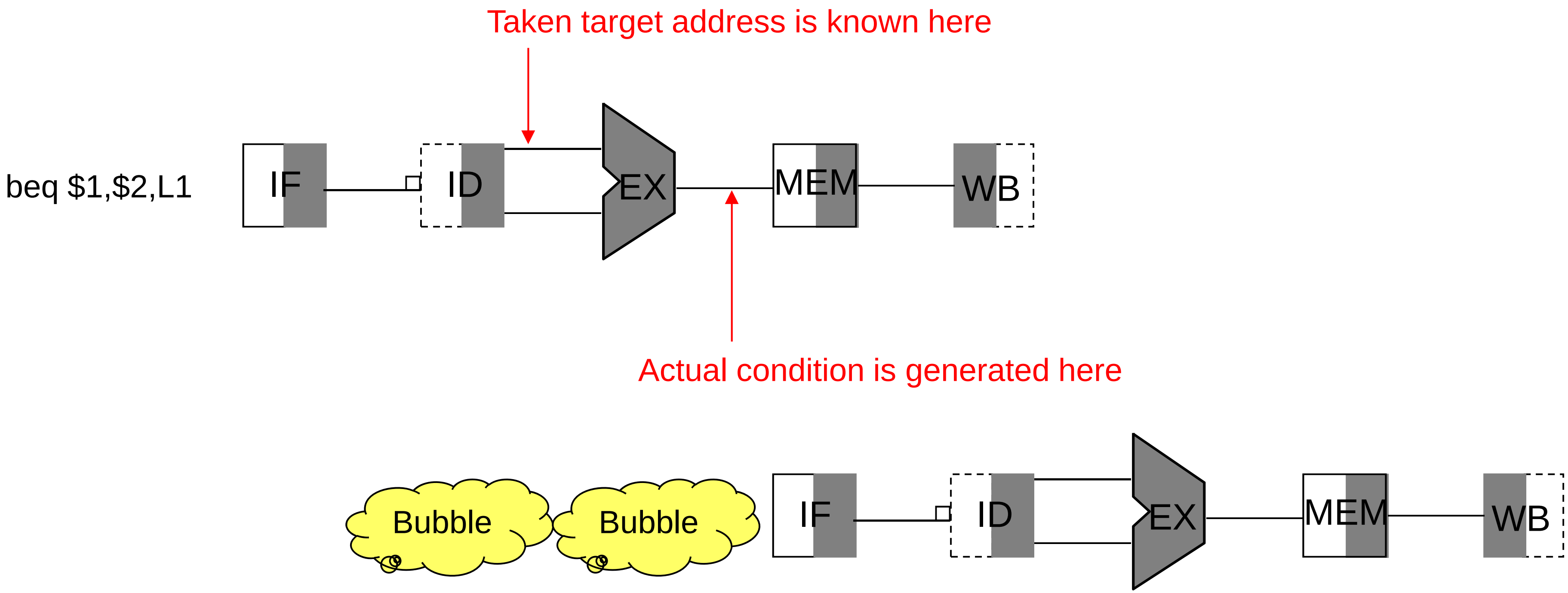


# Load-to-Use Data Hazard



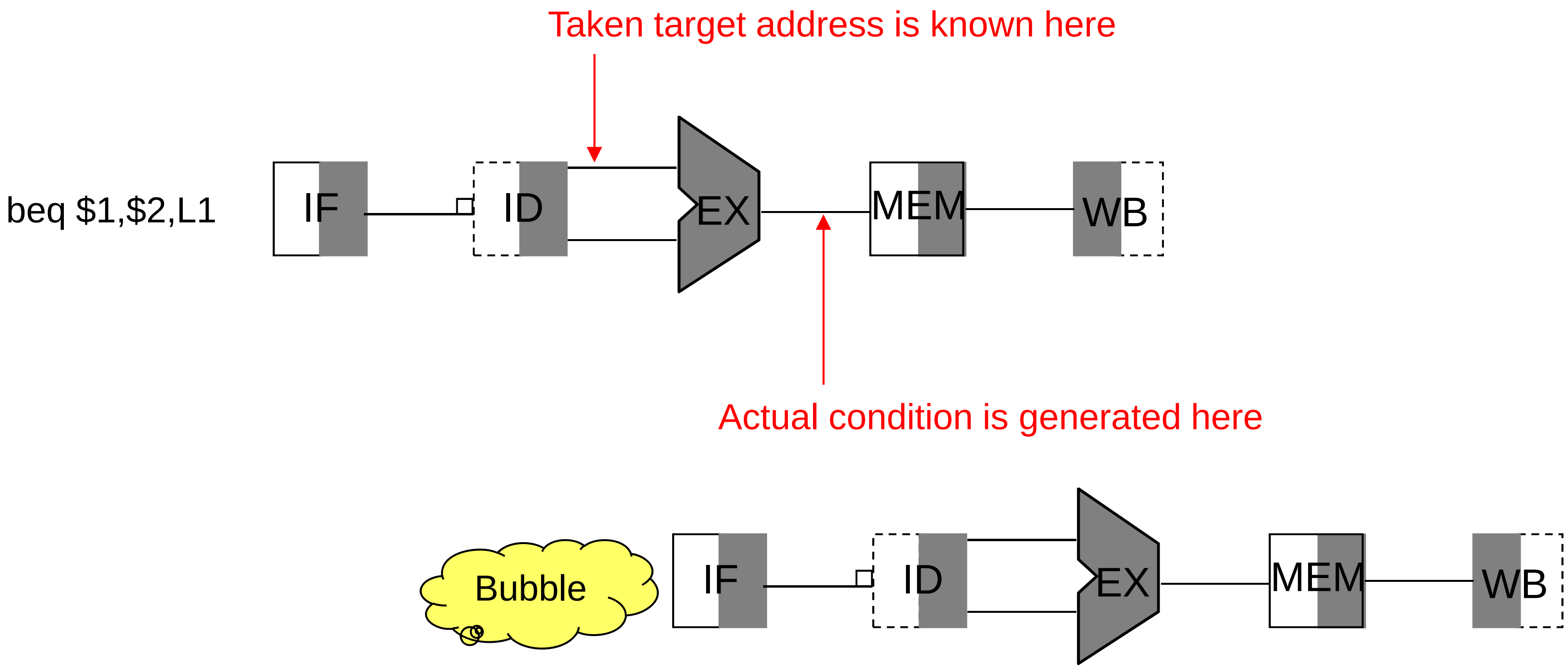

- This bubble can be hidden by proper instruction scheduling
- Hardware interlock is needed to install the pipeline stall

# Control Hazard



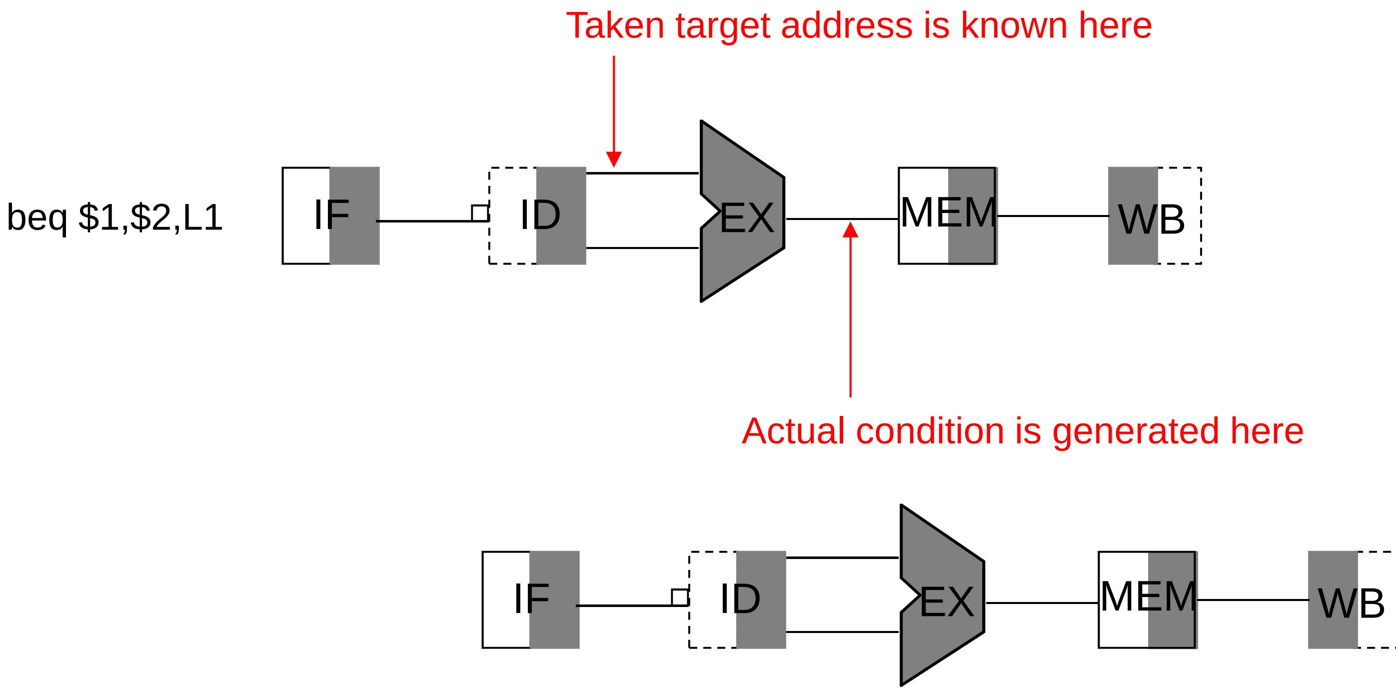

- Solution — branch prediction

# Control Hazard




- Branch prediction — Predict "branch direction"

# Control Hazard




- Branch prediction — Predict "branch target" and "branch direction"
- Static Branch Prediction
  - Use opcode
  - Use branch offset (+/-), e.g. backward taken, forward not taken,
- Dynamic Branch Predictor

울산대학교 UNIVERSITY OF ULSAN

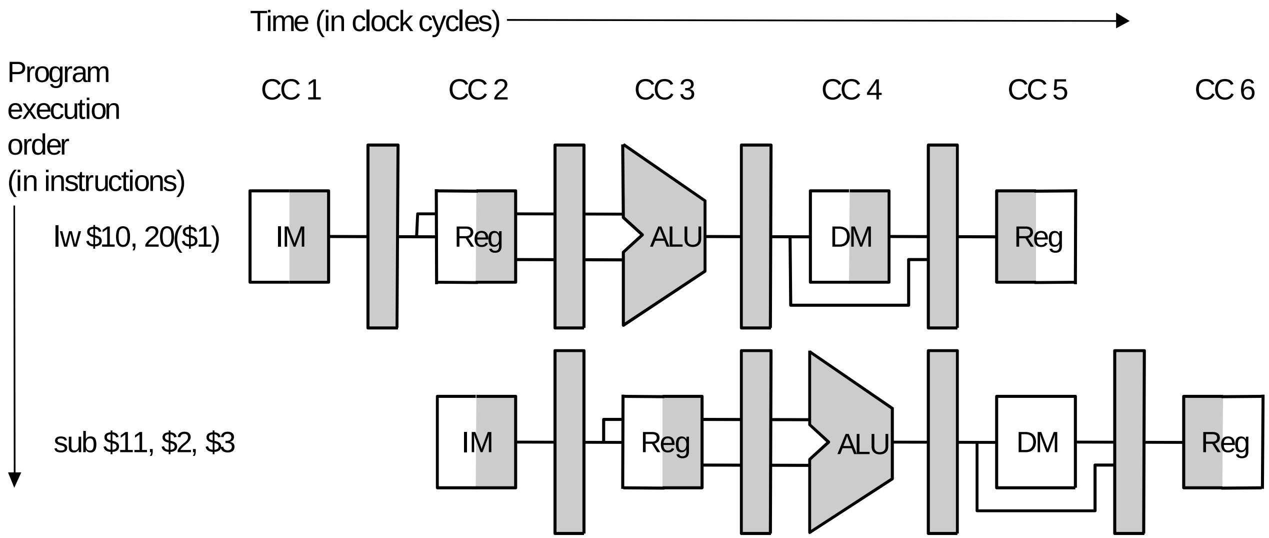
# Delay Slot (MIPS)

- Expose pipeline into the ISA design
- Load and jump/branch entail a "delay slot" (e.g. delayed branch)
  - No need for load delay slot in R4000 and after for hardware interlock was implemented
- The instruction right after the jump or branch is executed before the jump/branch

```
jal    function_A
add    $4, $5, $6    ; executed before jmp
lw     $12, 8($4)    ; executed after return
```

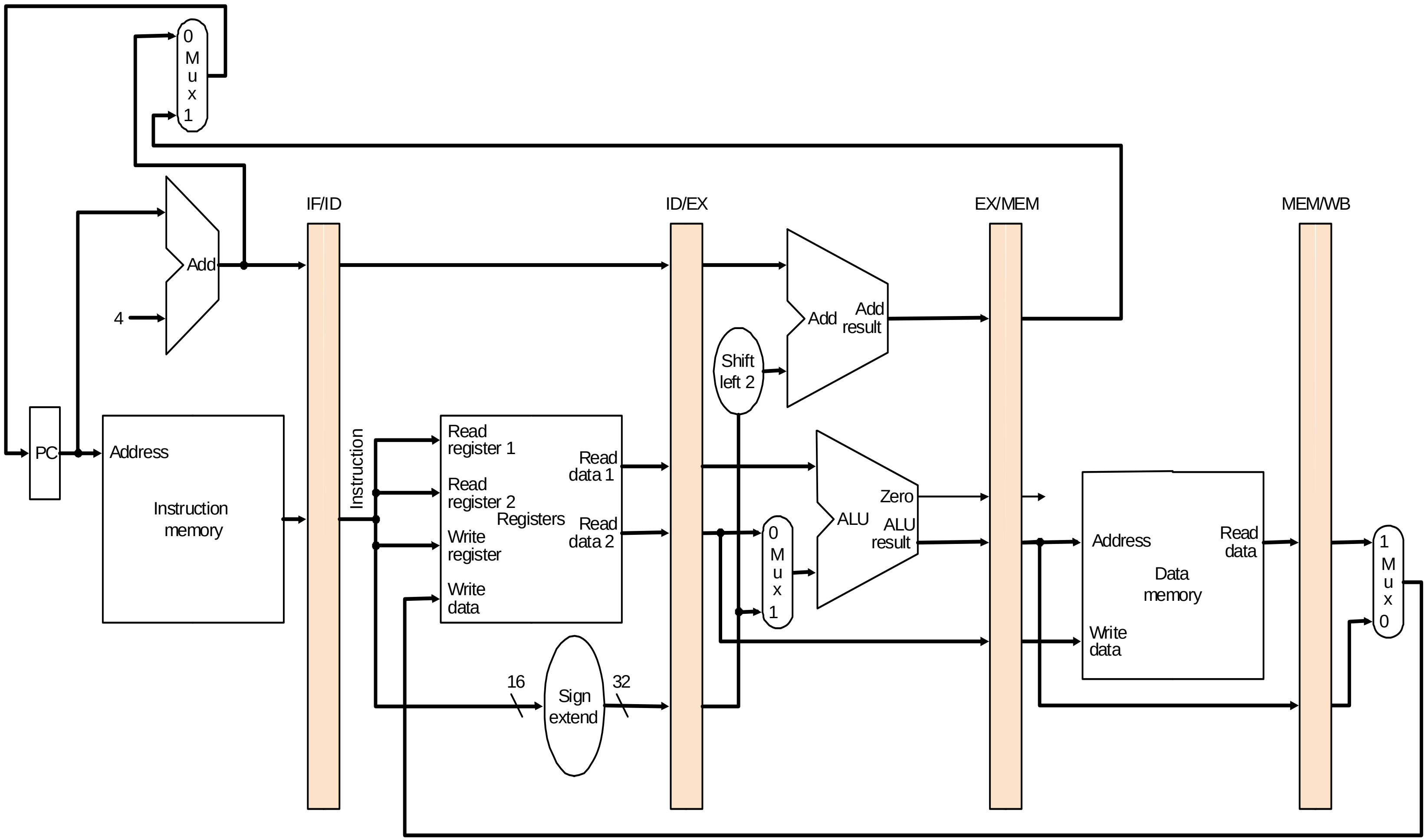
- Jump/branch and the delay slot instruction are considered "indivisible"
- In the delay slot, the compiler needs to schedule
  - A useful instruction (either before the jmp, or after the jmp w/o side effect)
  - otherwise a NOP

# Graphically Representing Pipelines

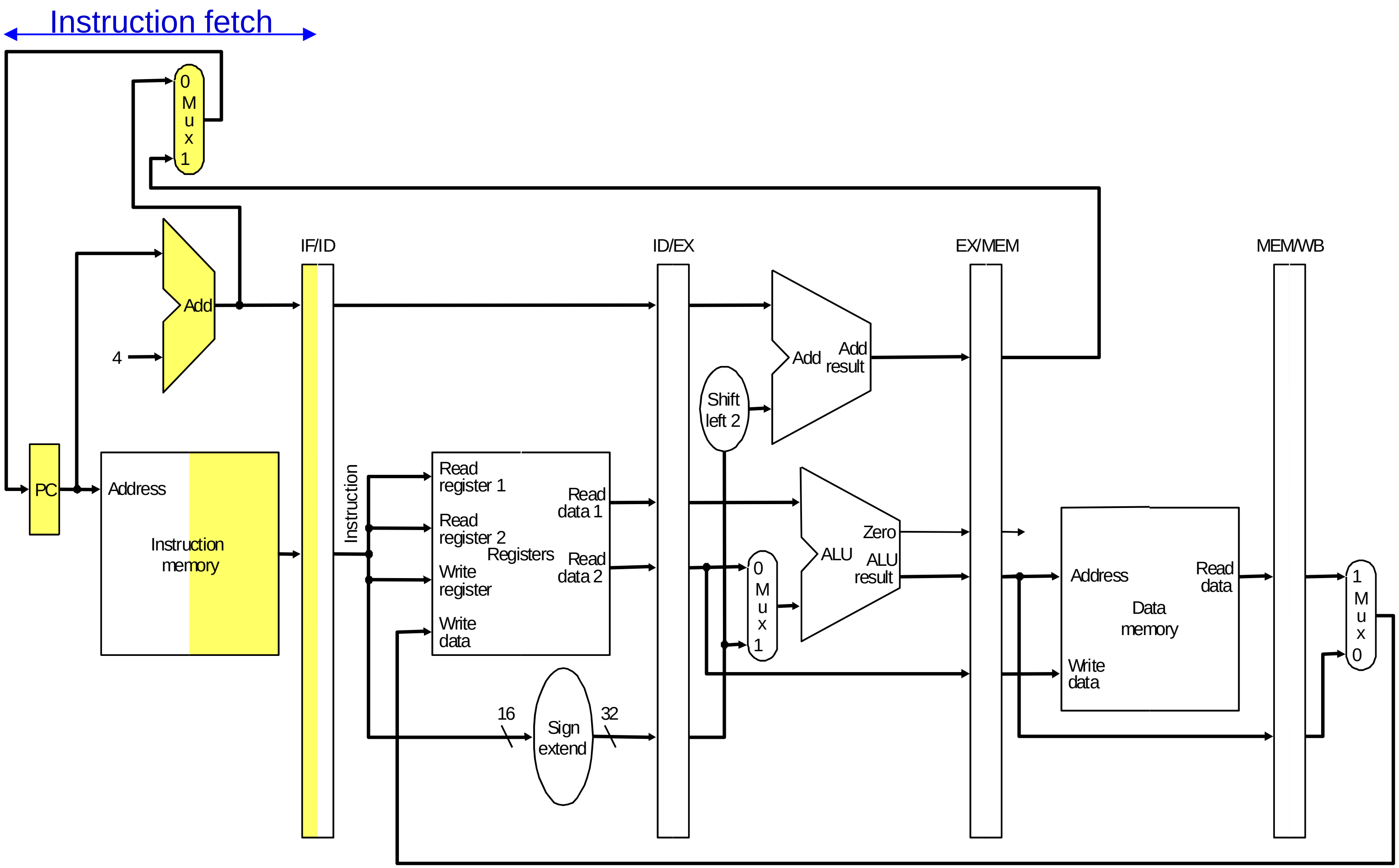



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

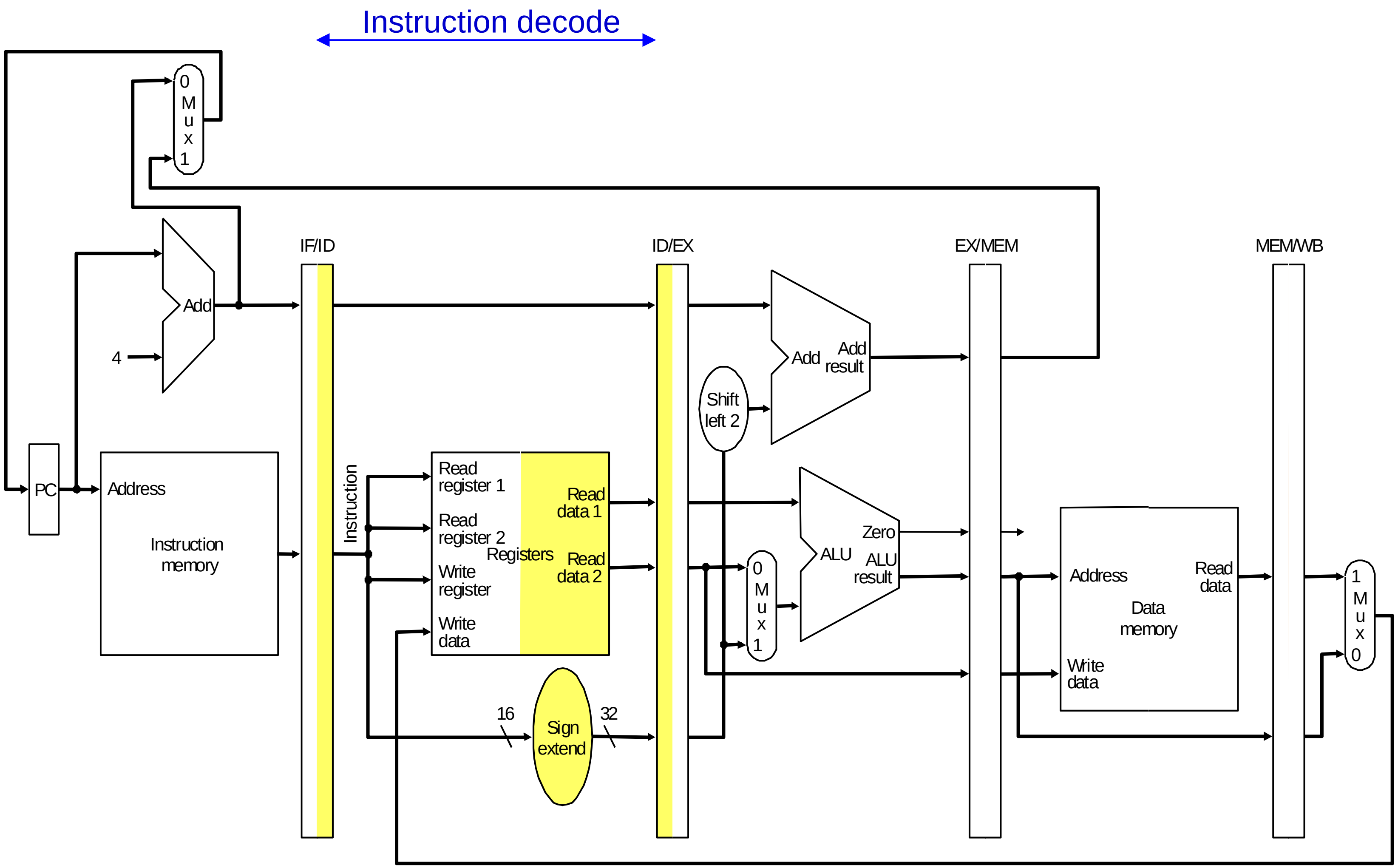
# Pipelined Datapath



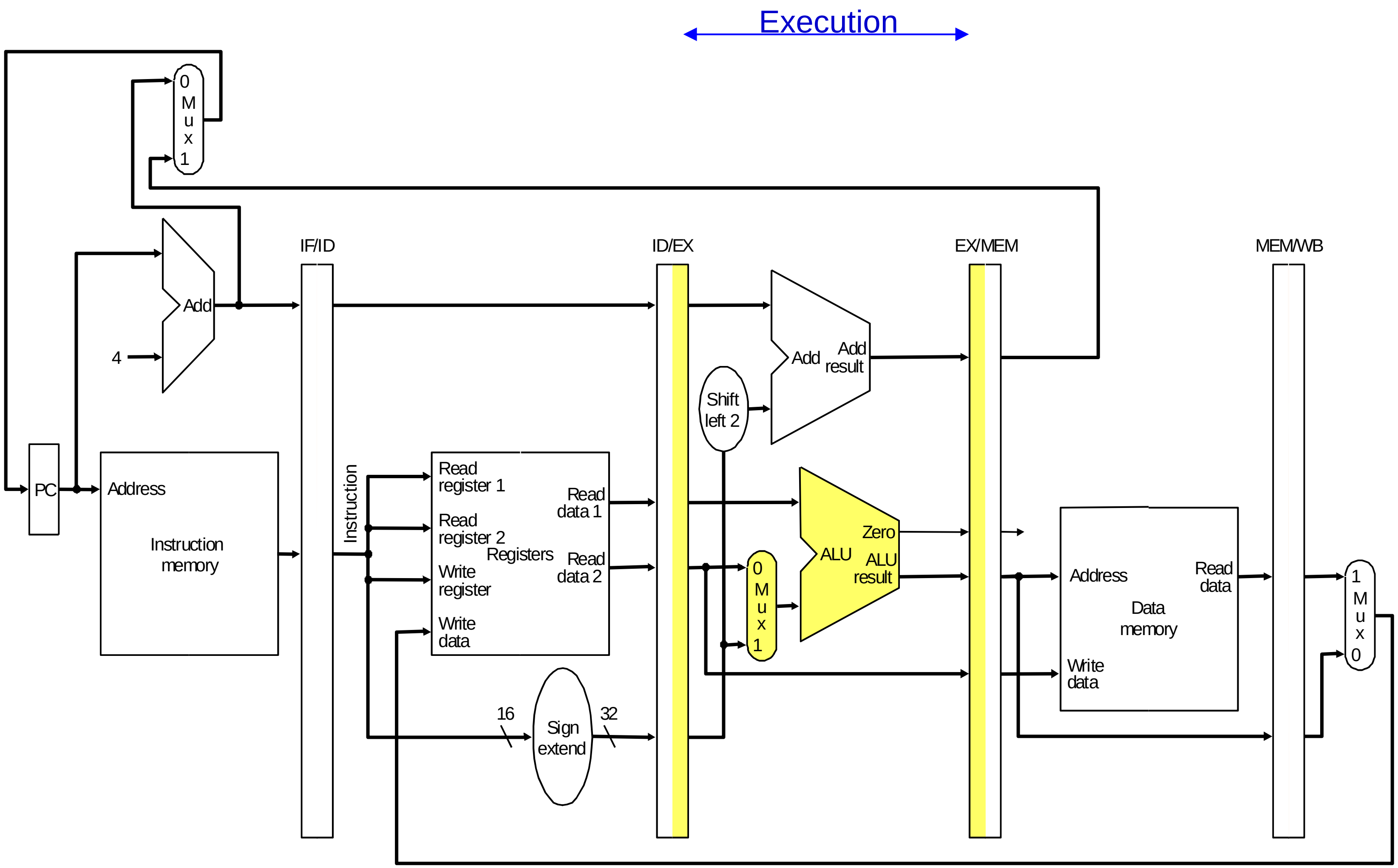
# Example for lw instruction: Instruction Fetch (IF)



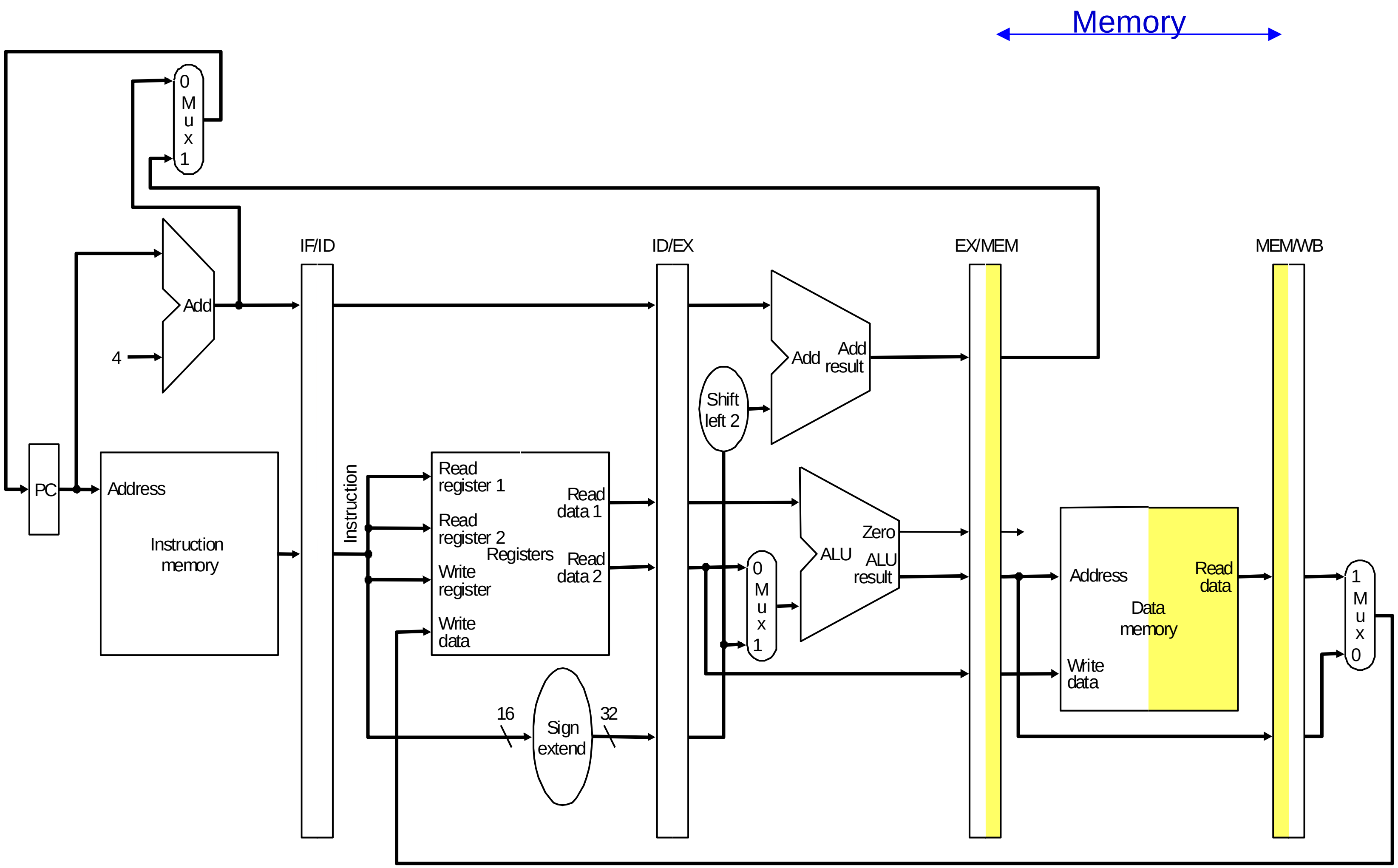
# Example for lw instruction: Instruction Decode (ID)



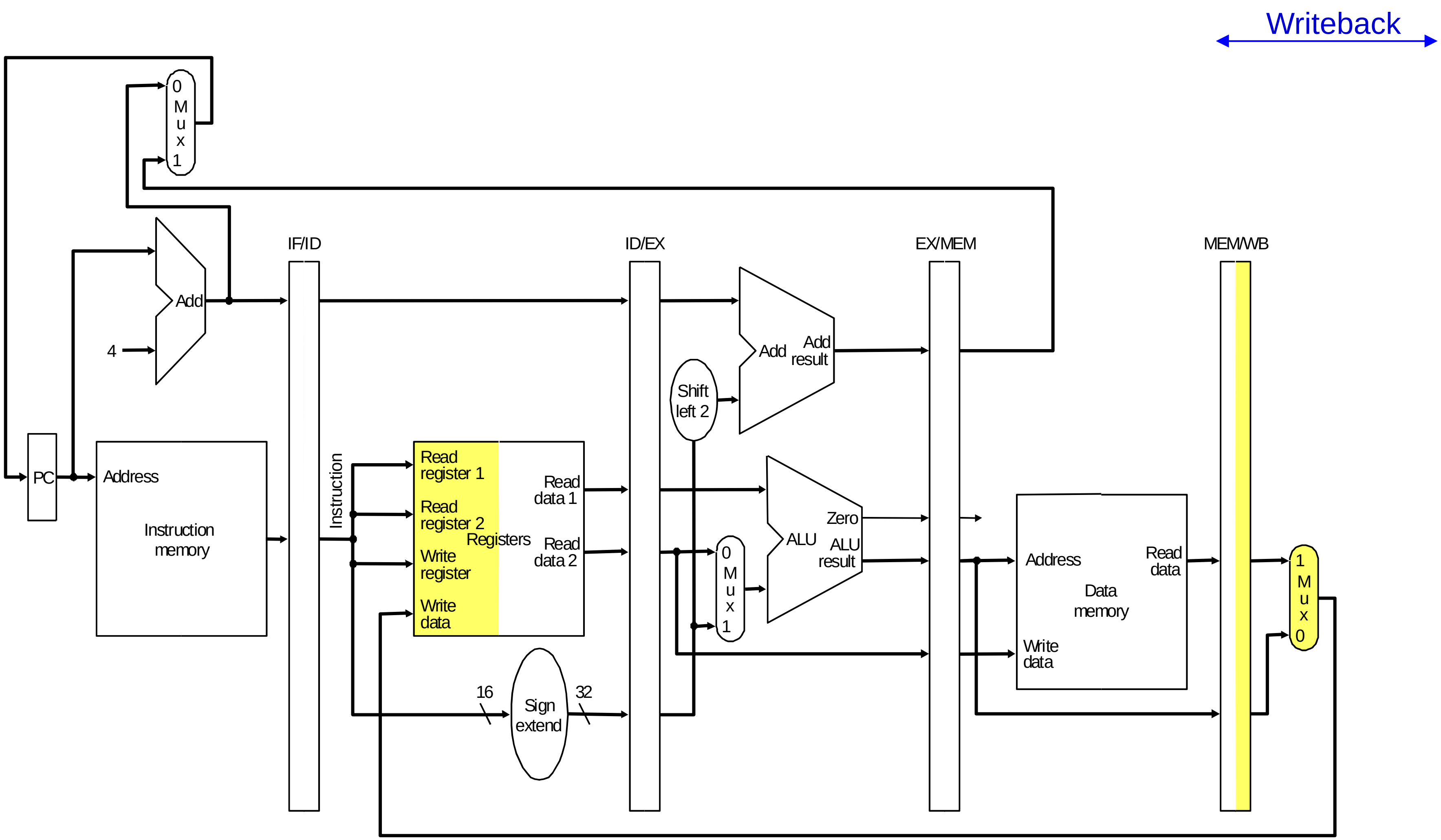
# Example for lw instruction: Execution (EX)



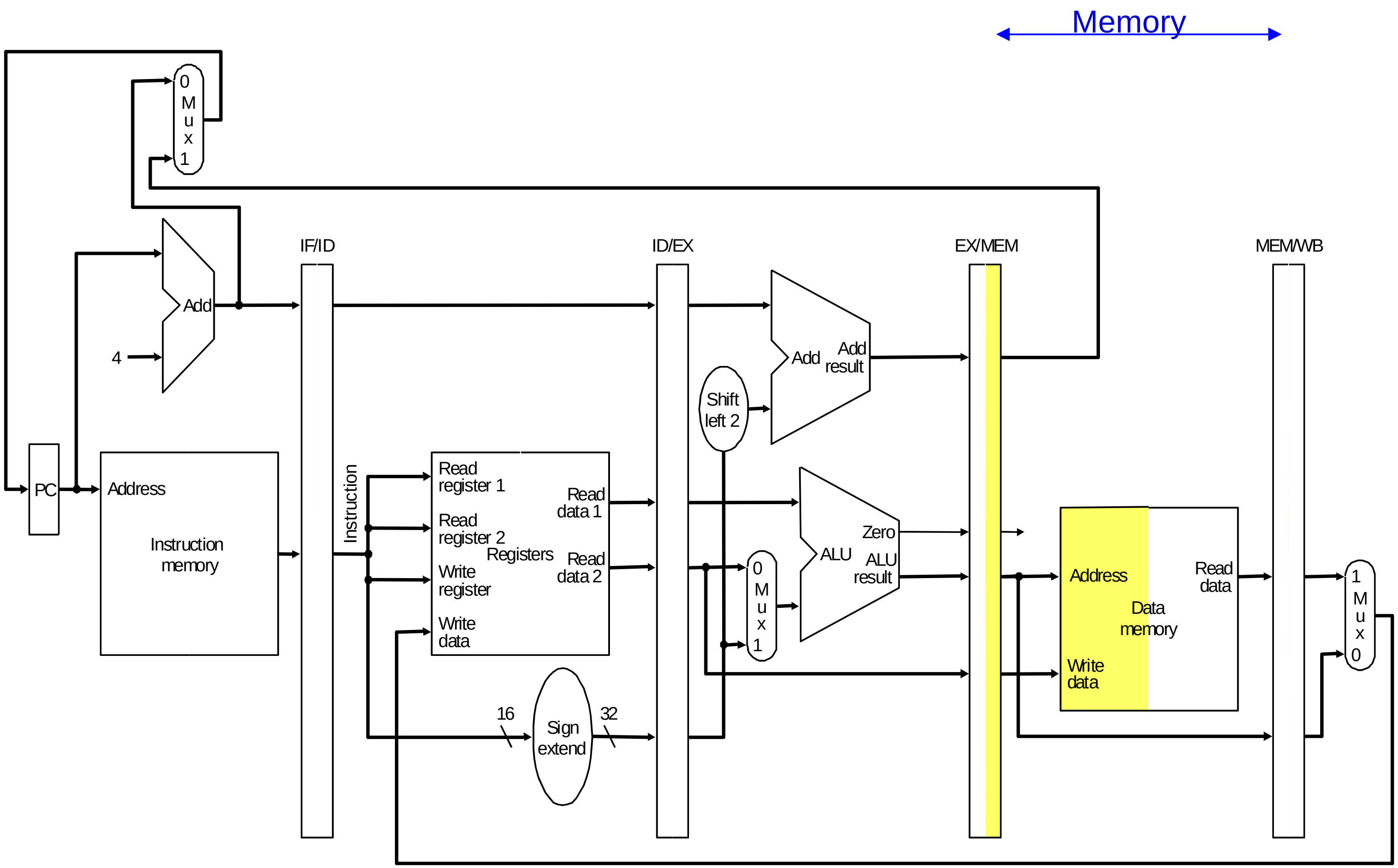
# Example for lw instruction: Memory (MEM)



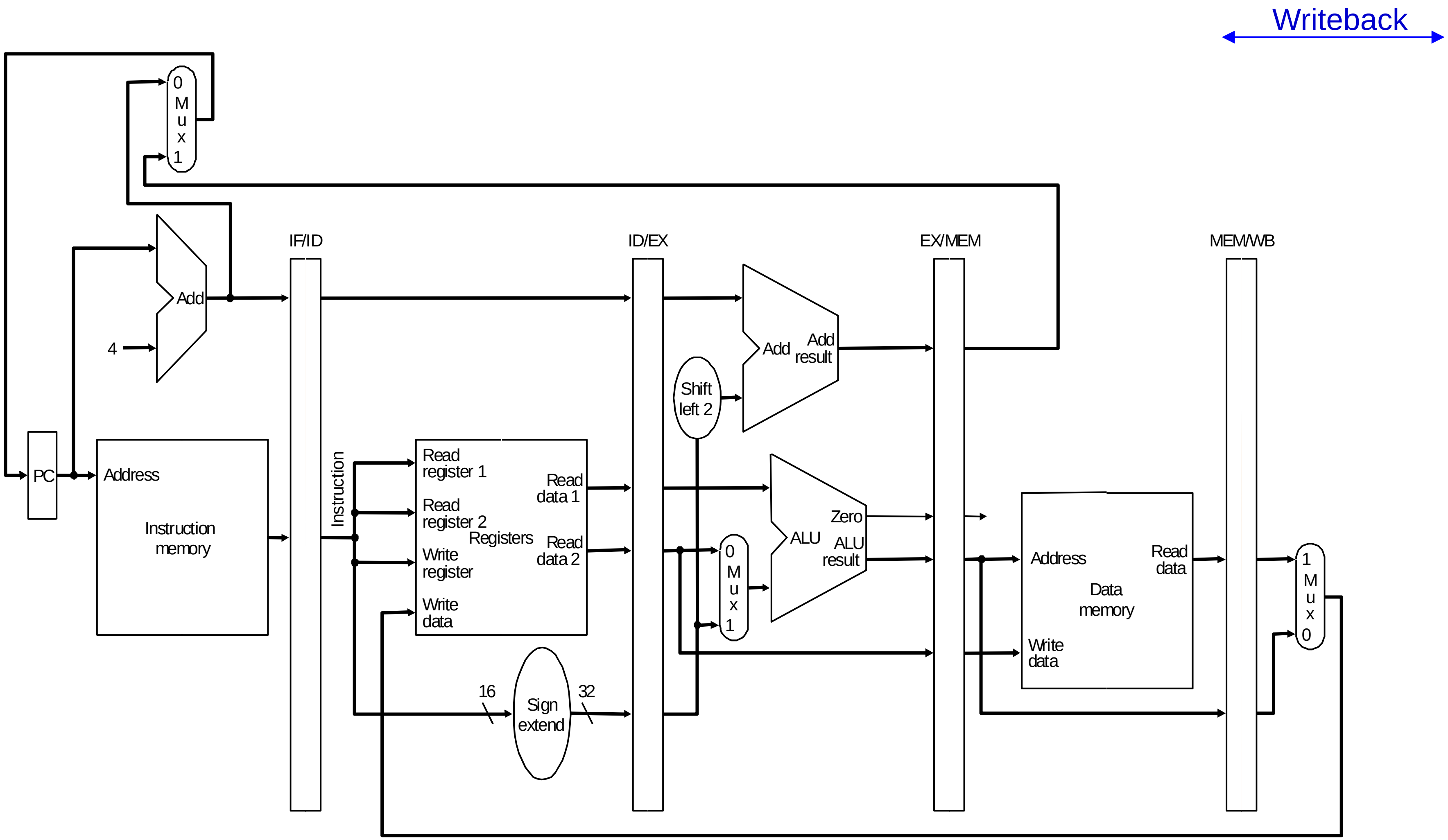
# Example for lw instruction: Writeback (WB)



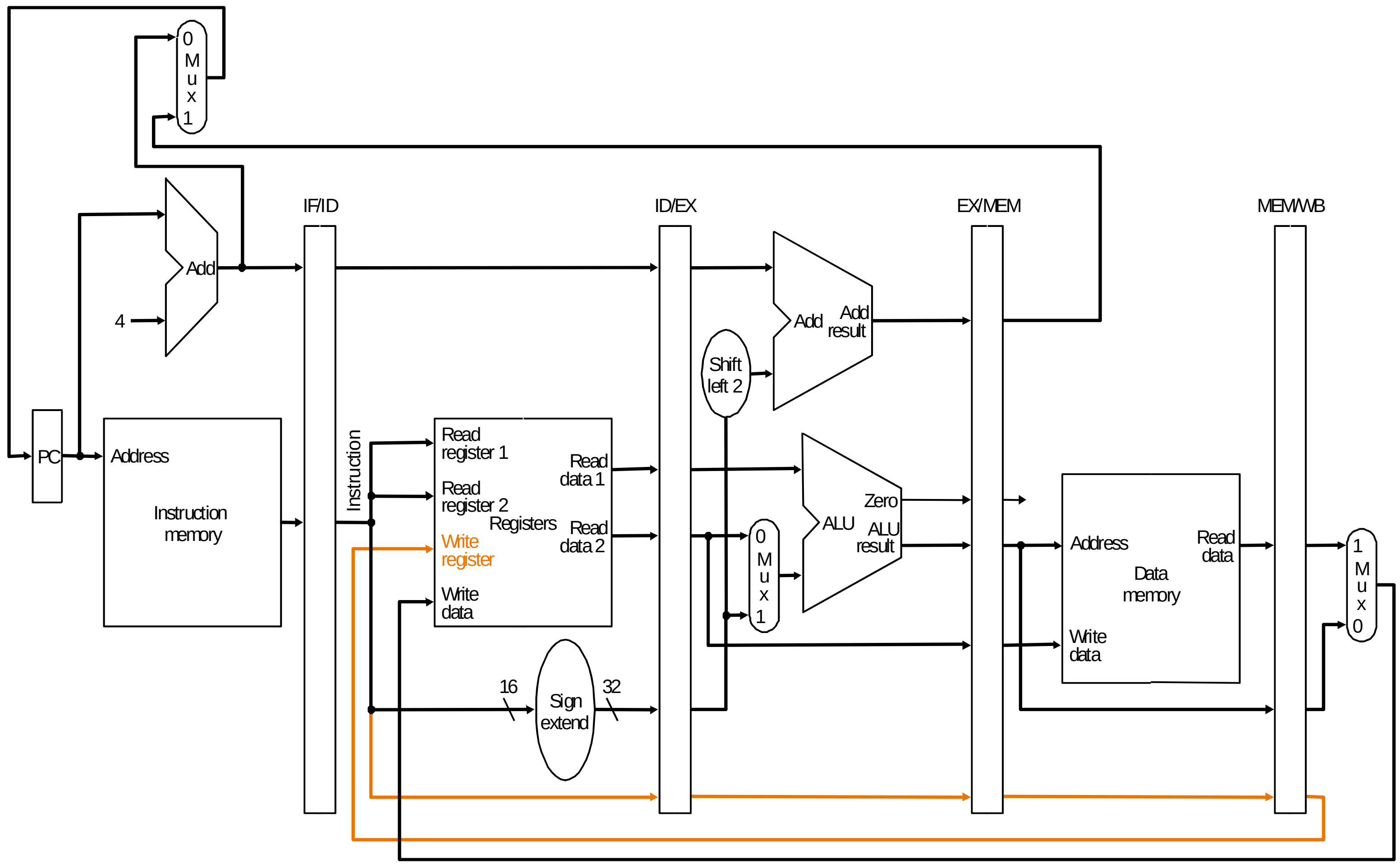
# Example for sw instruction: Memory (MEM)



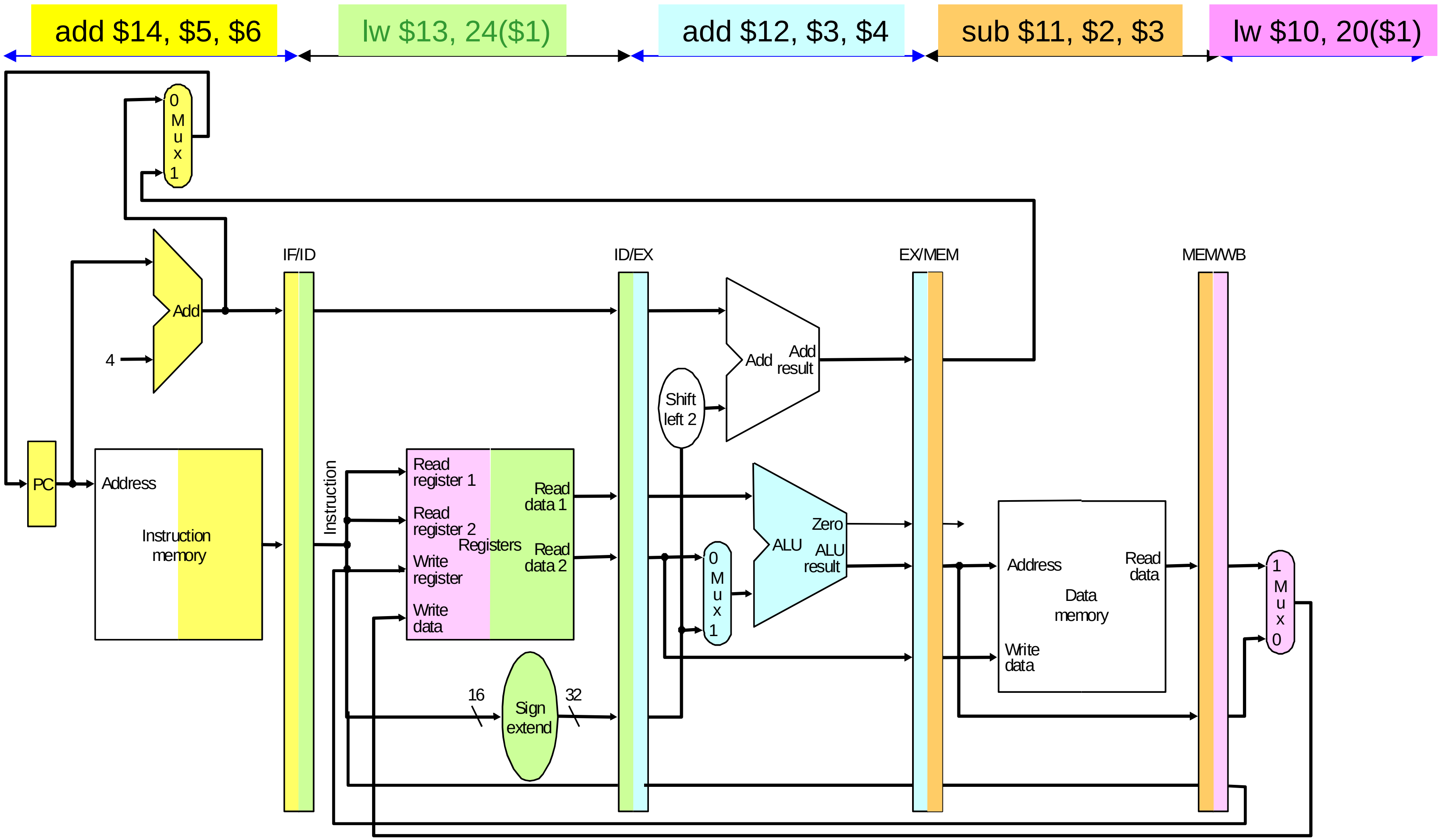
# Example for sw instruction: Writeback (WB): do nothing



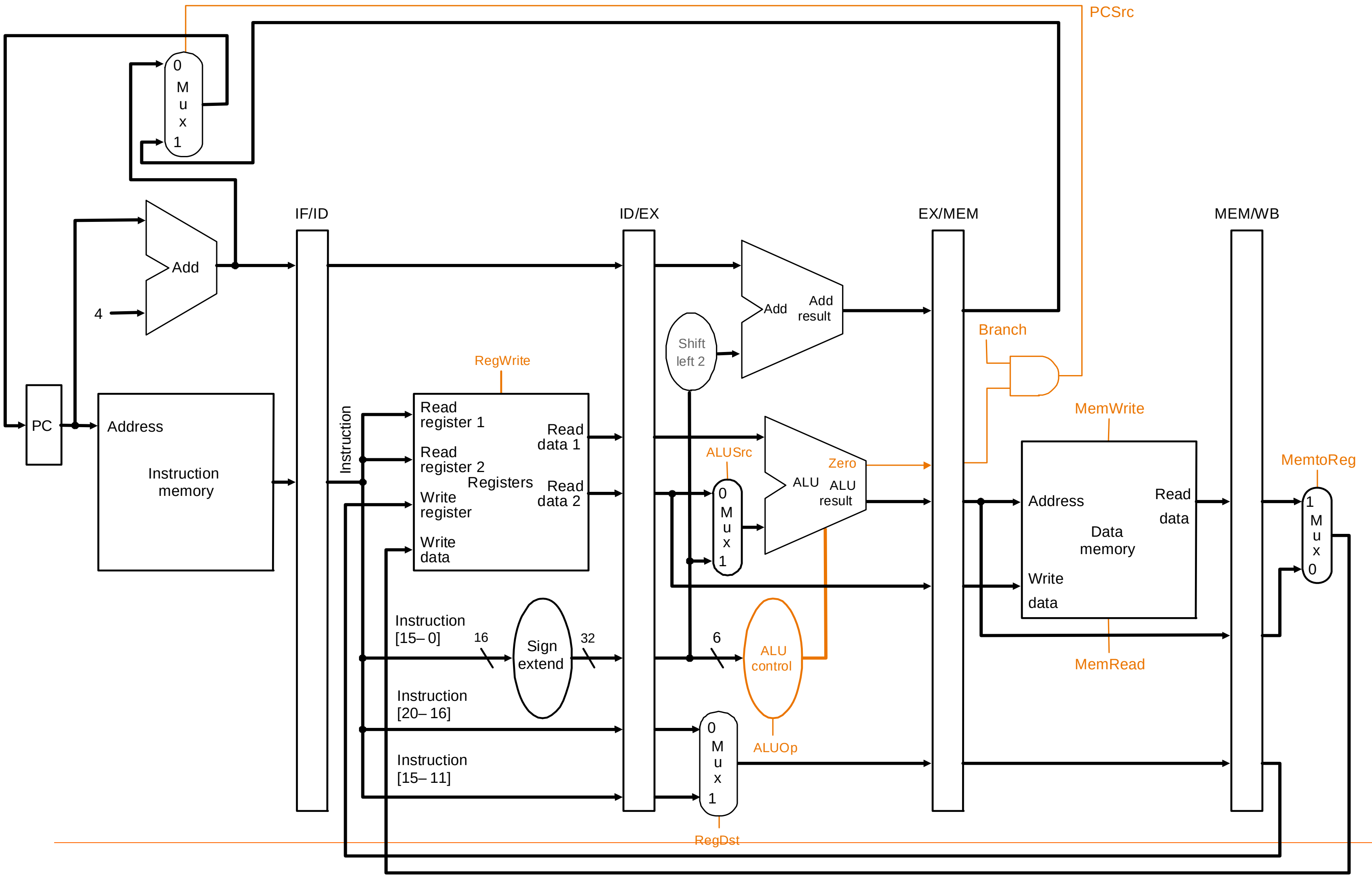
# Corrected Datapath (for lw)



# Pipelining Example



# Pipeline Control



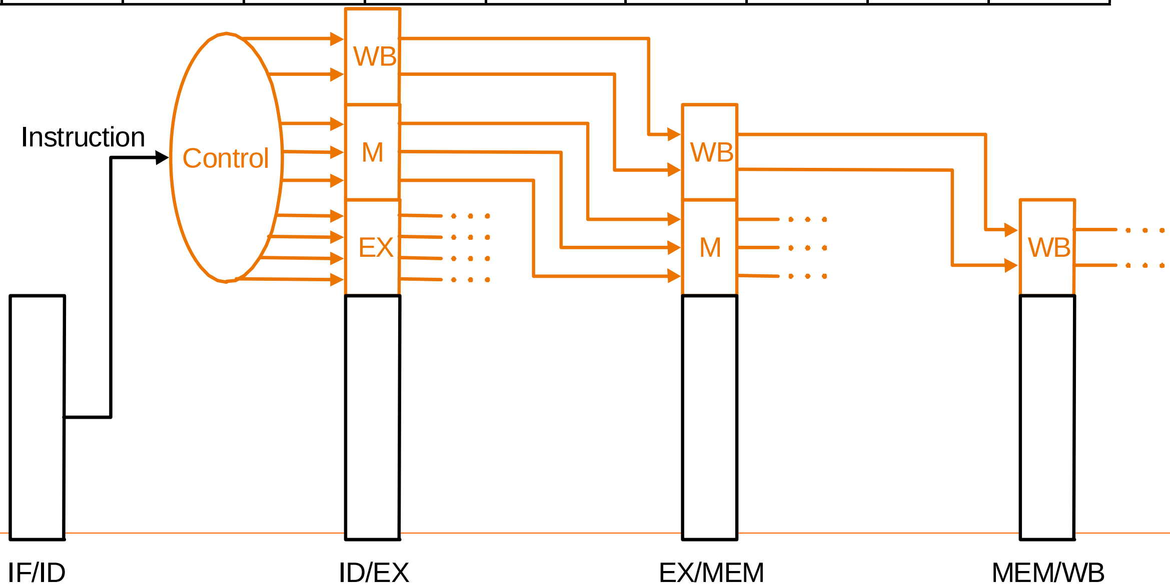
# Pipeline control

- We have 5 stages.  What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution (4 lines)
    - RegDst
    - ALUop[1:0]
    - ALUSrc
  - Memory Stage (3 lines)
    - Branch
    - MemRead
    - MemWrite
  - Write Back (2 lines)
    - MemtoReg
    - RegWrite (note that this signal is in ID stage)

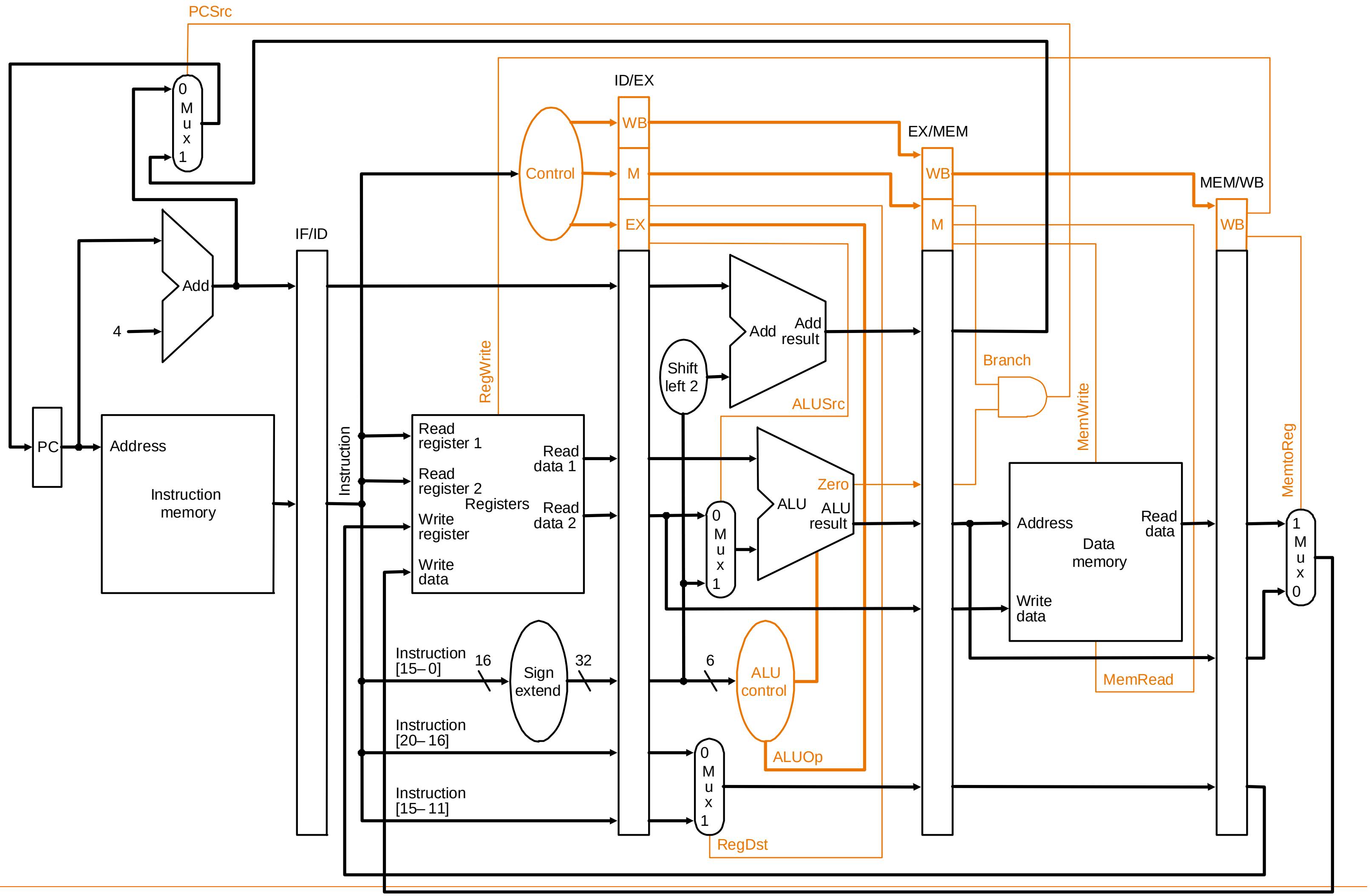
# Pipeline Control

- Extend pipeline registers to include control information (created in ID)
- 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 |



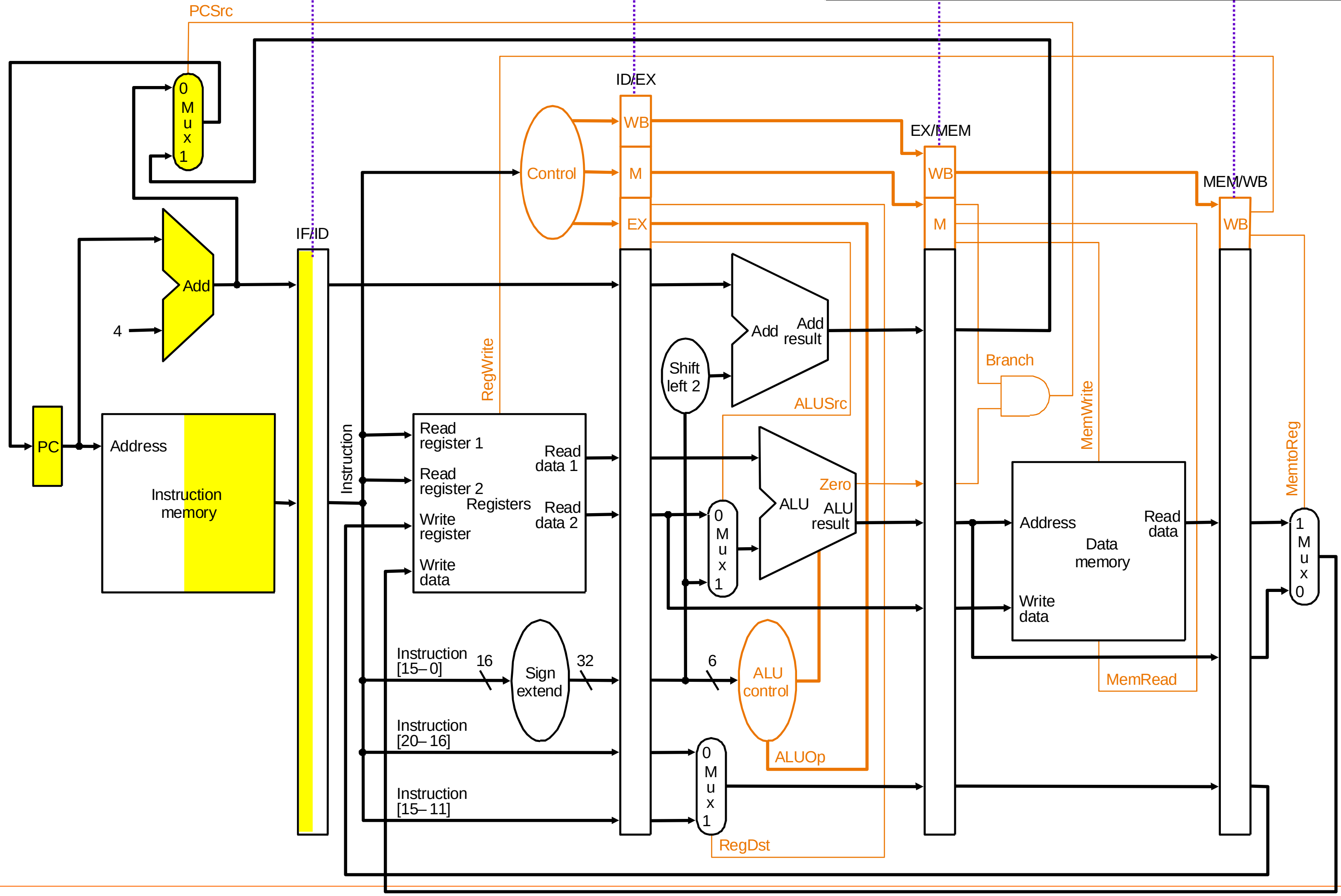
# Datapath with Control



# Datapath with Control

| 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 |

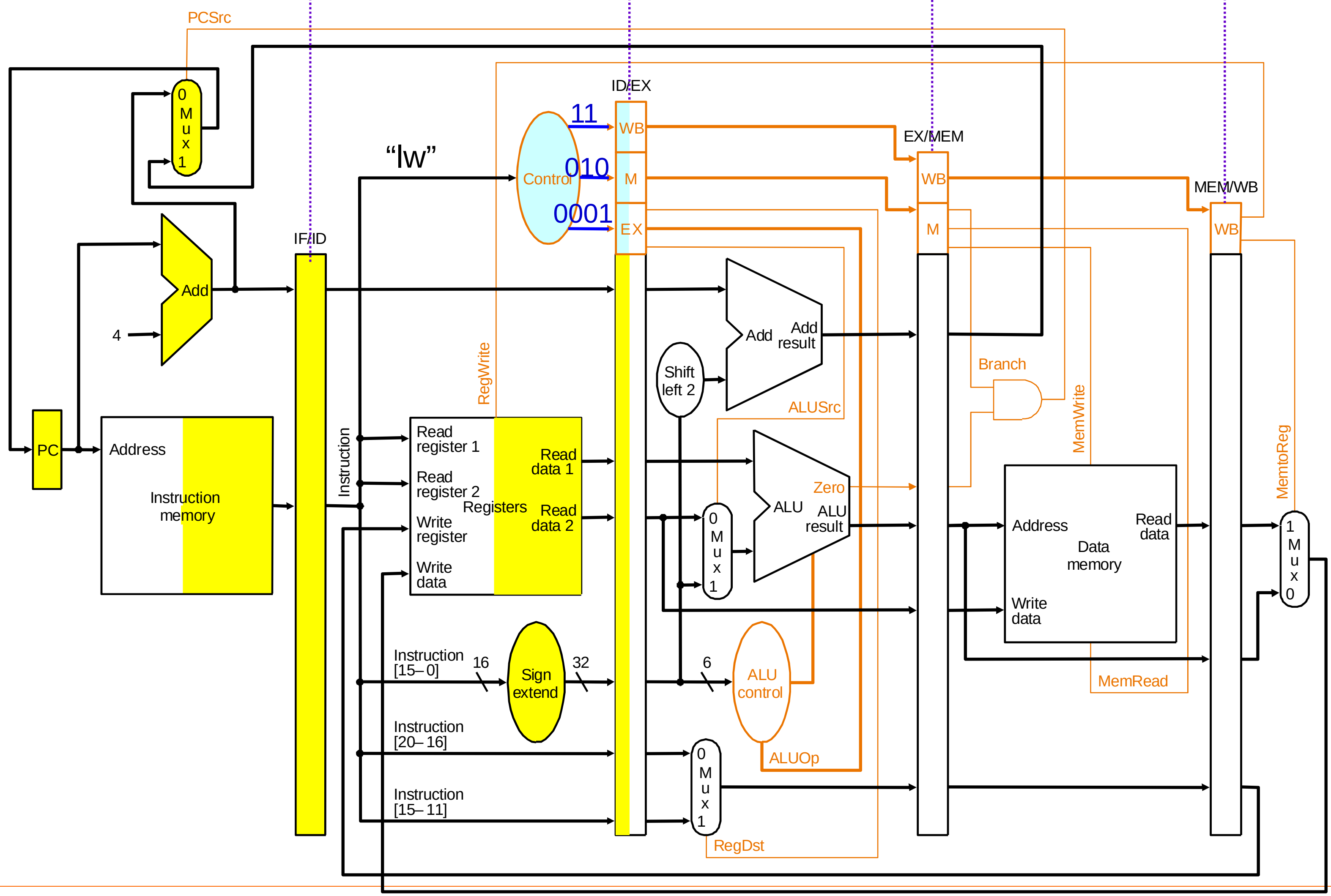
**IF: lw $10, 9($1)**



# Datapath with Control

| 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 |

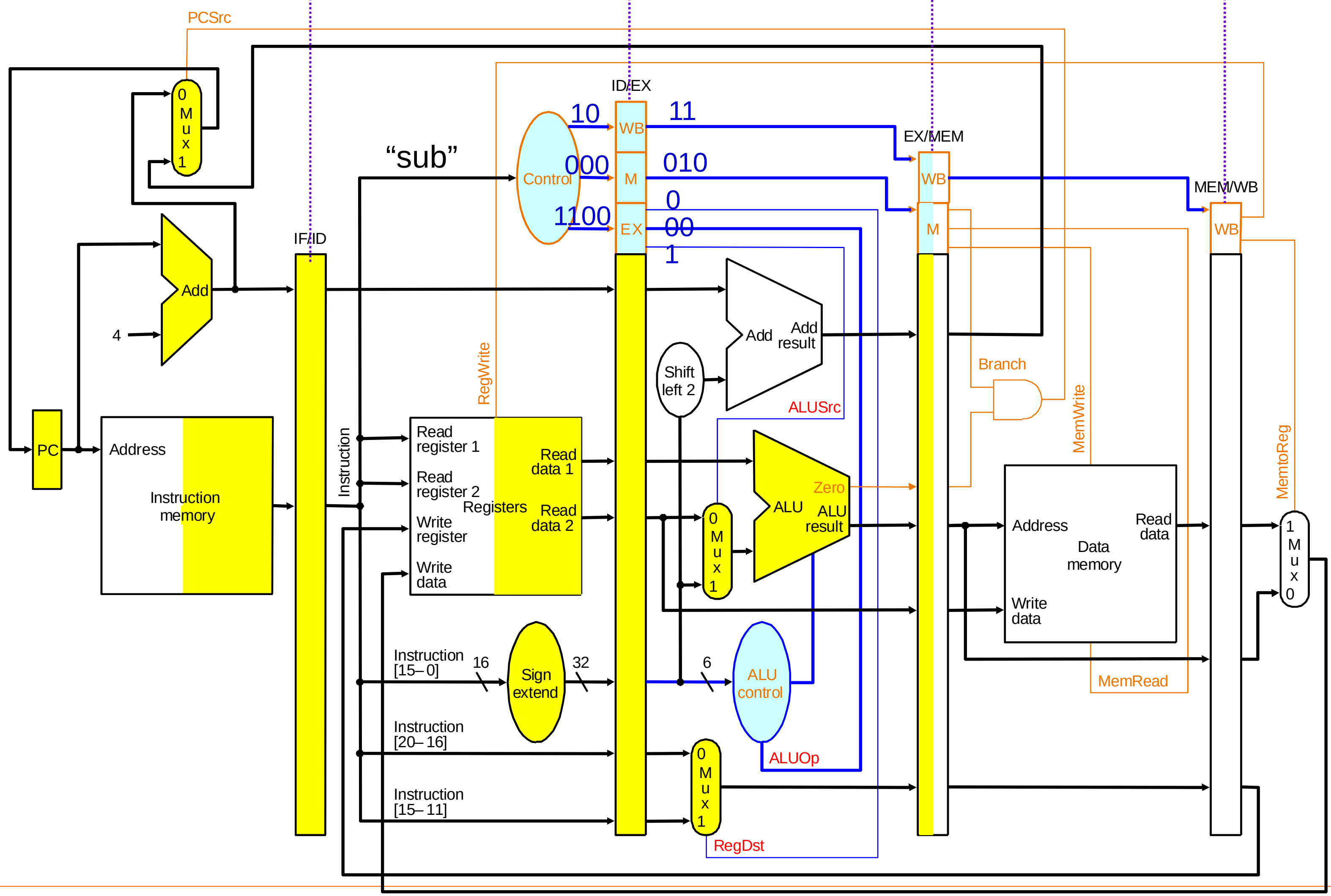
**IF: sub $11, $2, $3** **ID: lw $10, 9($1)**



# Datapath with Control

| 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 |

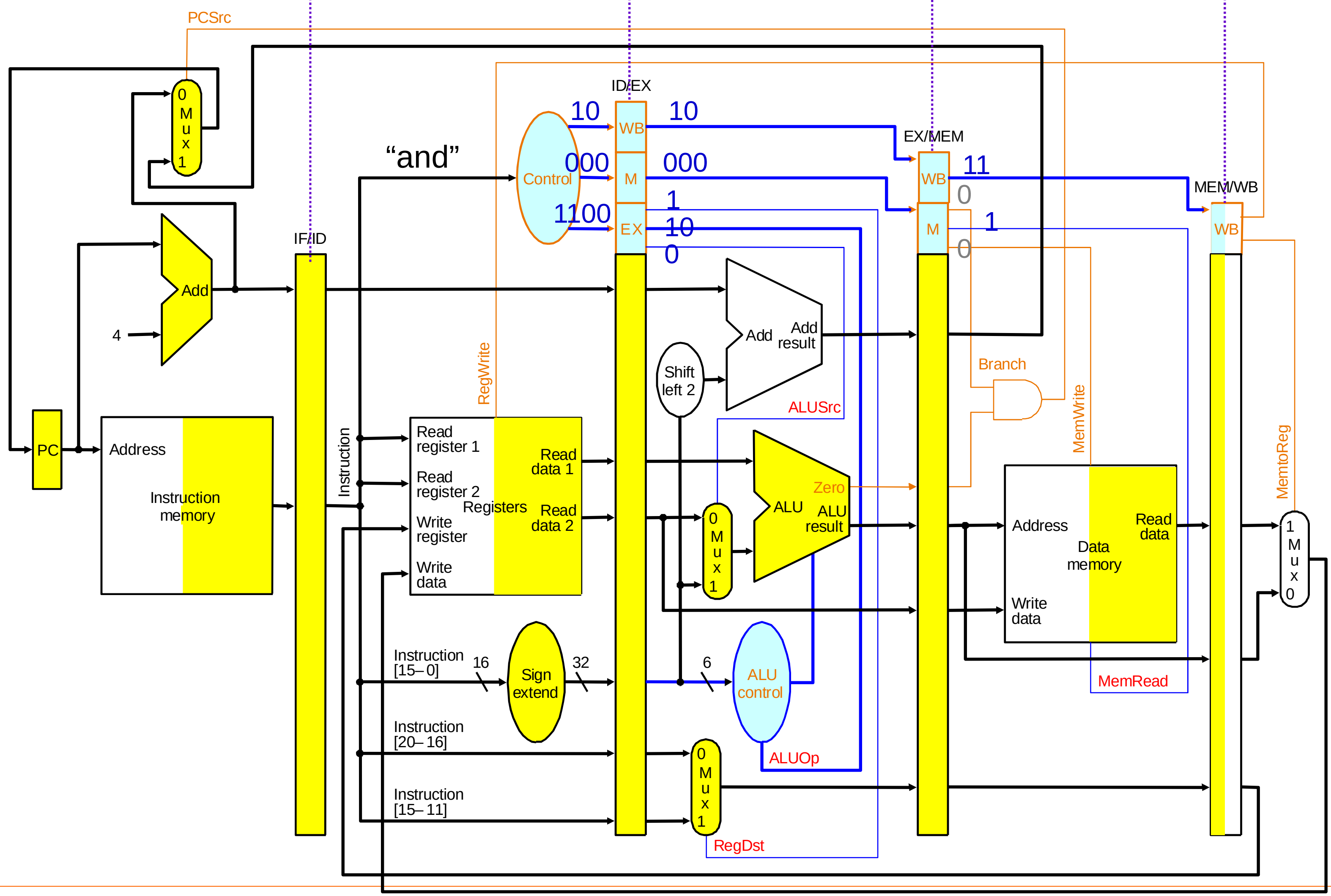
**IF: and $12, $4, $5** **ID: sub $11, $2, $3** **EX: lw $10, 9($1)**



# Datapath with Control

| 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 |

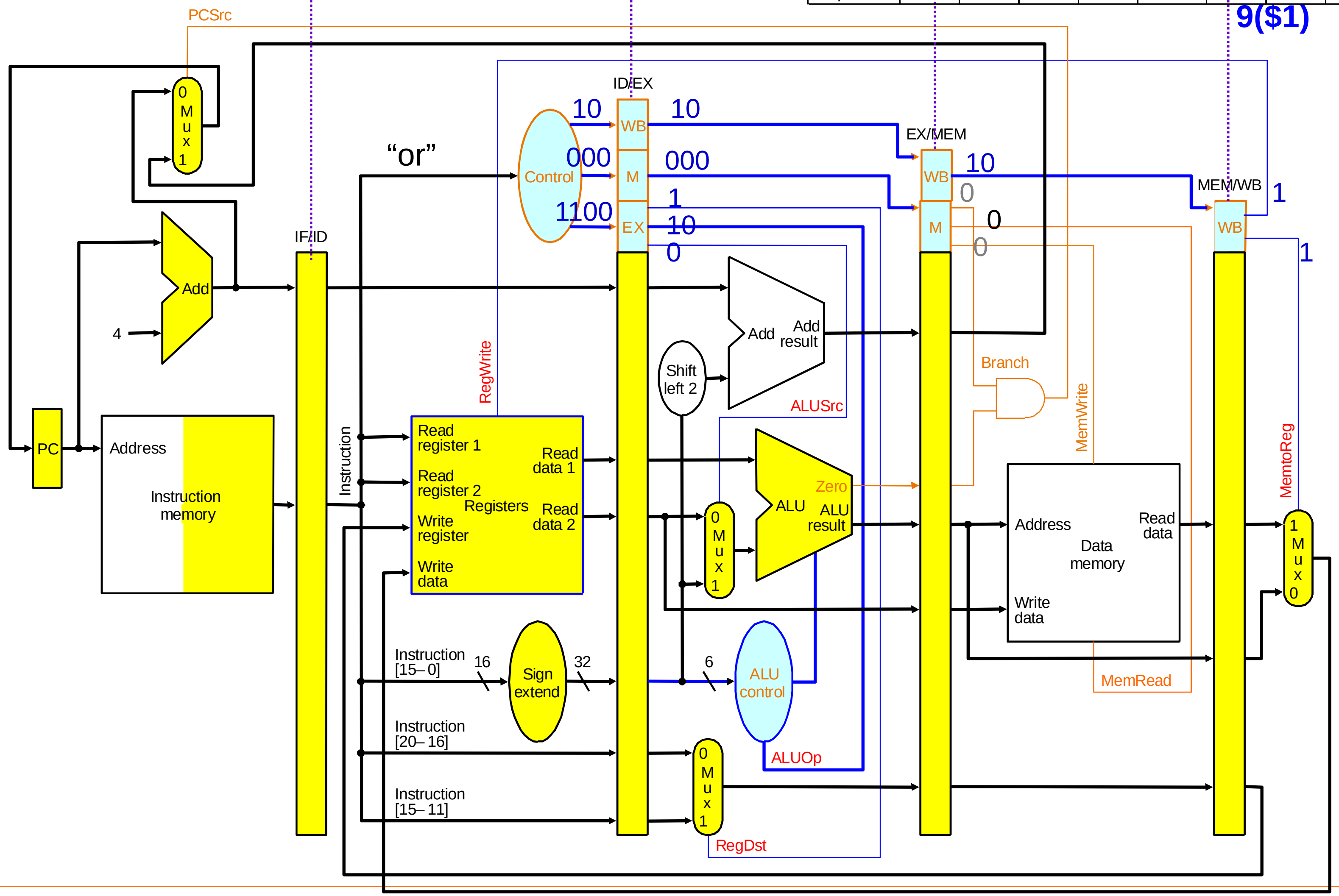
**IF: or $13, $6, $7　ID: and $12, $4, $5　EX: sub $11, $2, $3　MEM: lw $10, 9($1)**



# Datapath with Control

| 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 |

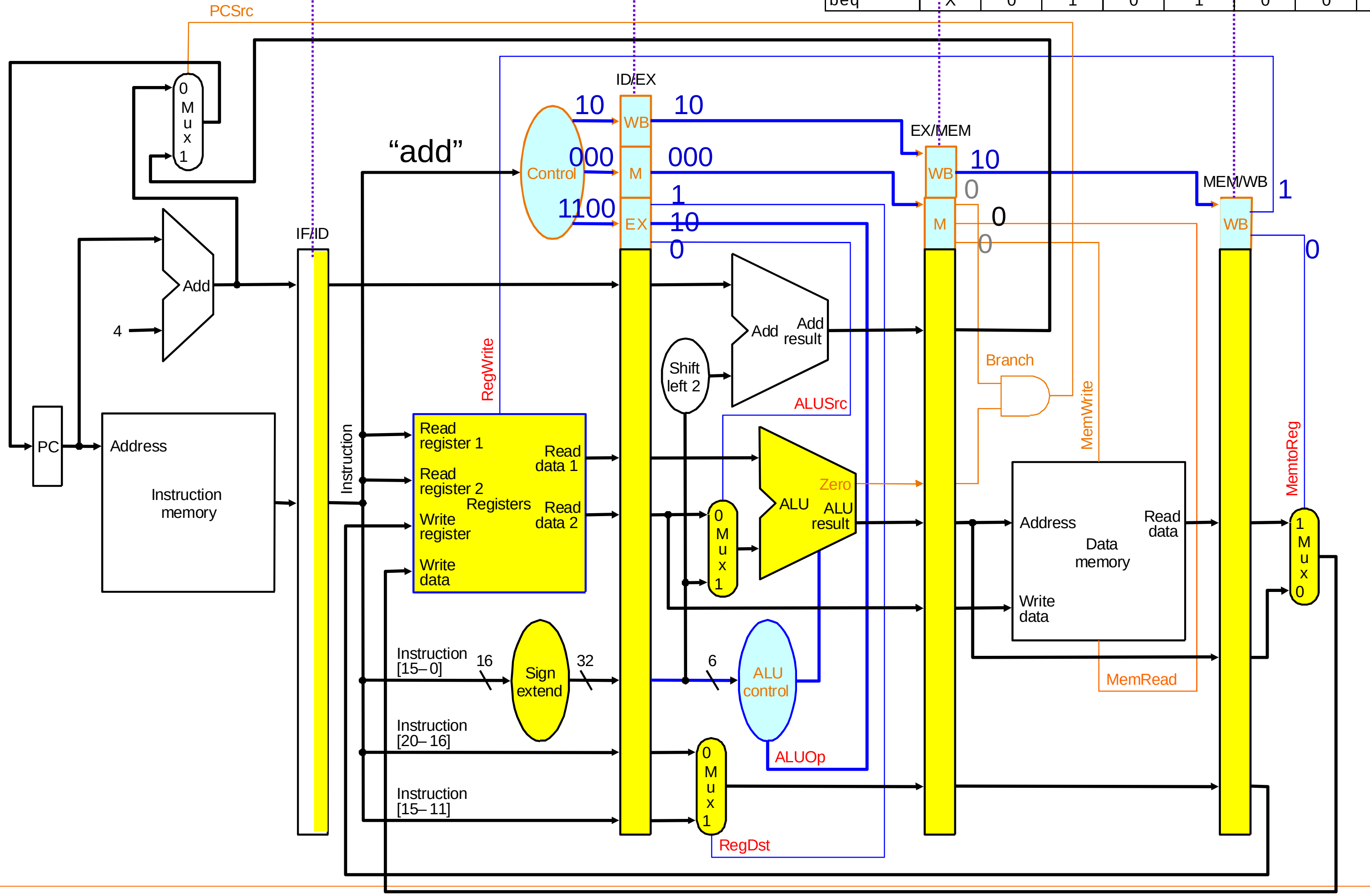
IF: add $14, $8, $9 | ID: or $13, $6, $7 | EX: and $12, $4, $5 | MEM: sub $11, .. | WB: lw $10, 9($1)



# Datapath with Control

| 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 |

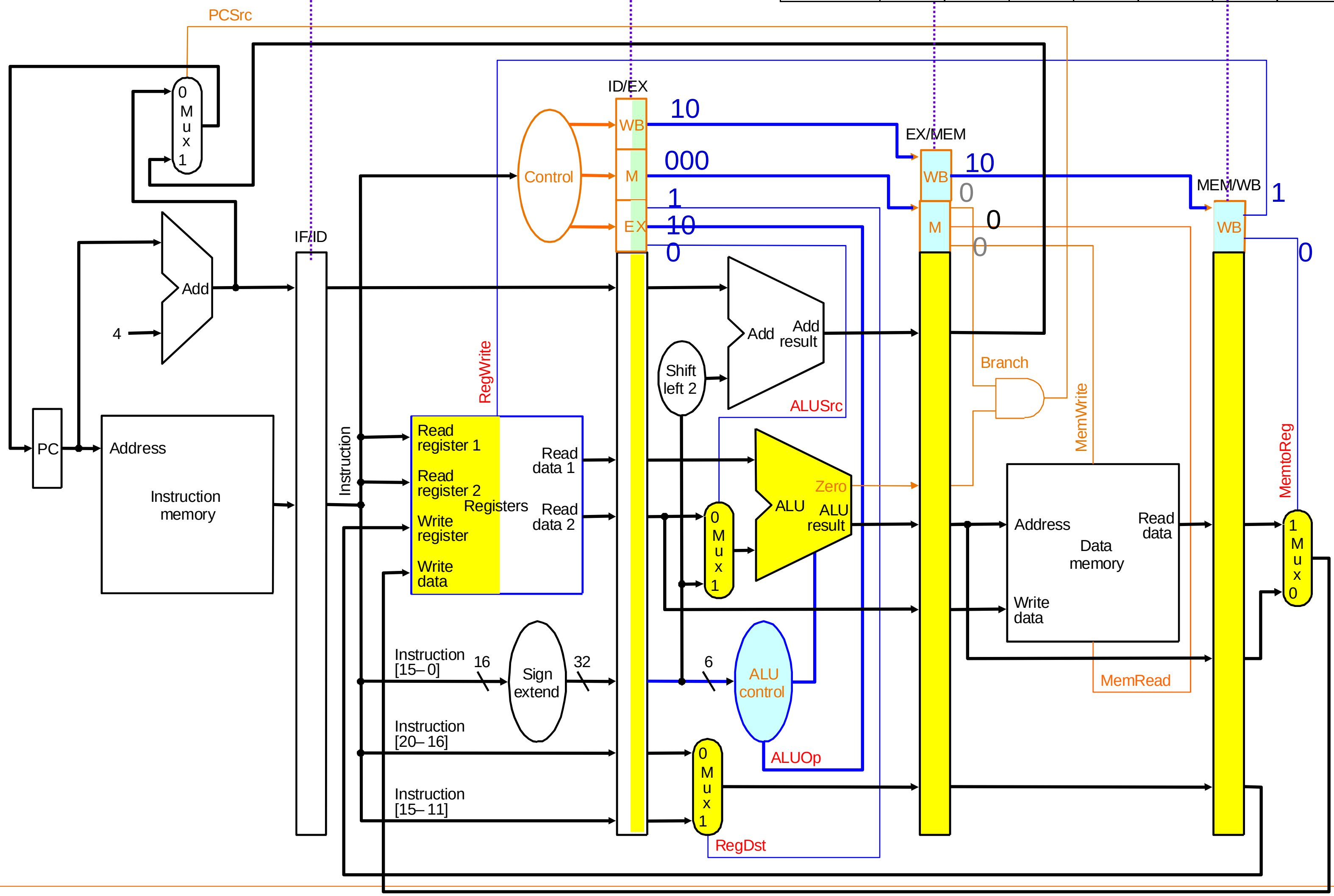
**IF: xxxx** **ID: add \$14, \$8, \$9** **EX: or \$13, \$6, \$7** **MEM: and \$12..** **WB: sub \$11, ..**



# Datapath with Control

| 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 |

IF: xxxx ID: xxxx EX: add $14, $8, $9 MEM: or $13,.. WB: and $12,..



# Datapath with Control

| 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 |

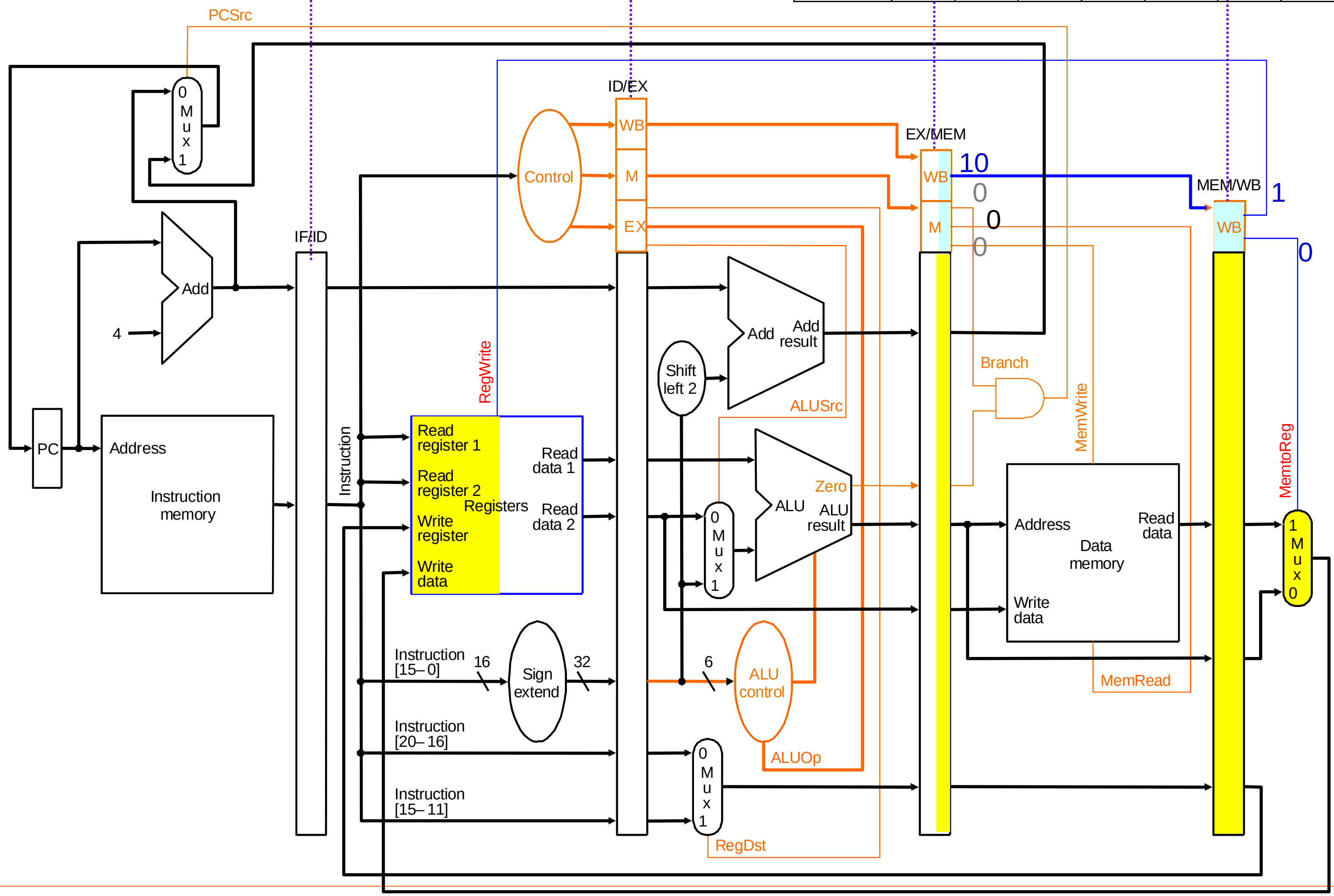
IF: xxxx

ID: xxxx

EX: xxxx

MEM: add $14, ..

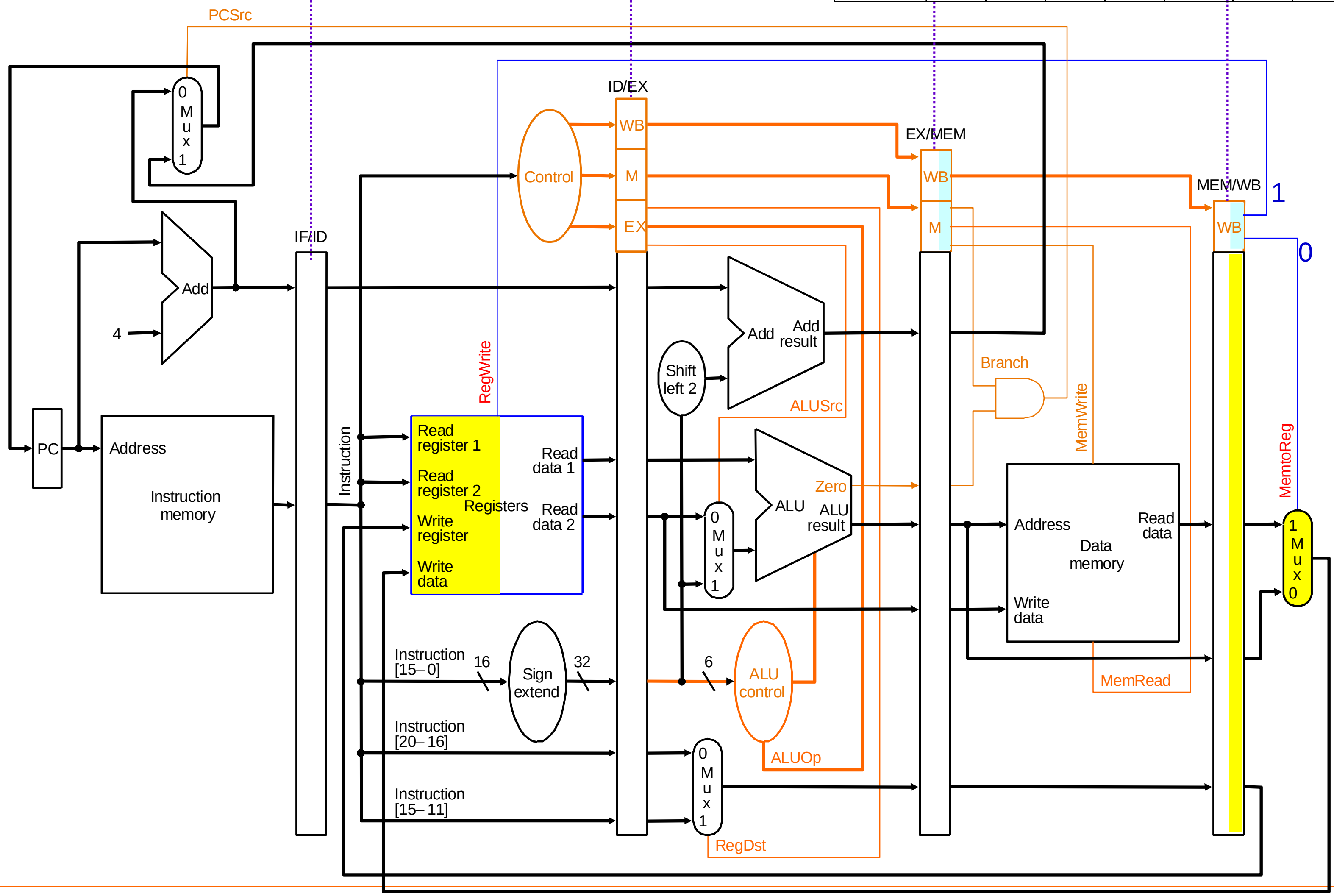
WB: or $13…



# Datapath with Control

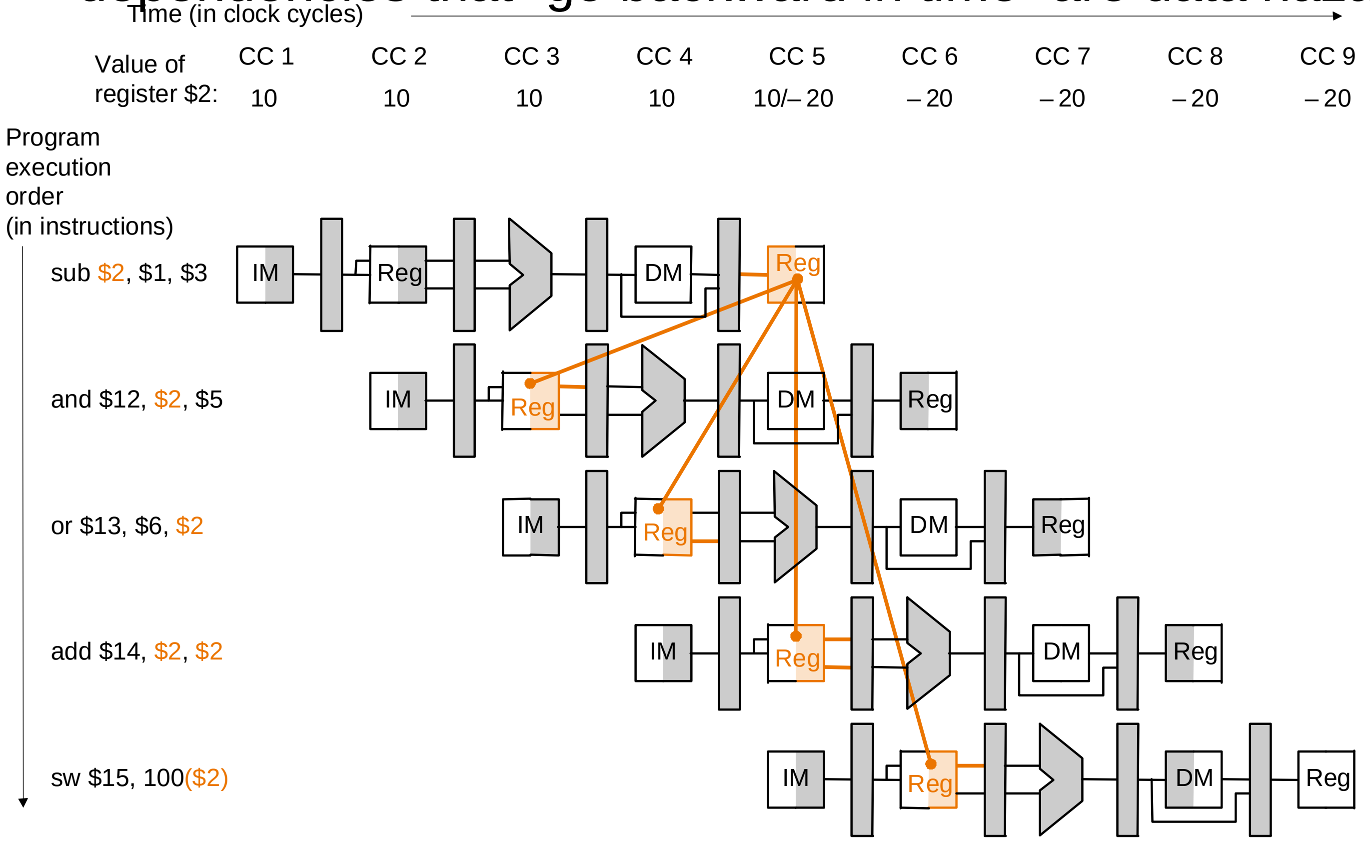
| Instruction | Execution/Address Calculation stage control lines: Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Memory access stage control lines: Branch | Mem Read | Mem Write | Write-back stage control lines: Reg write | Mem to Reg |
|---|---|---|---|---|---|---|---|---|---|
| R-format | 1 | 1 | 0 | 0 | 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 |

**IF: xxxx** **ID: xxxx** **EX: xxxx** **MEM: xxxx** **WB: add $14..**



# Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards



# Software Solution

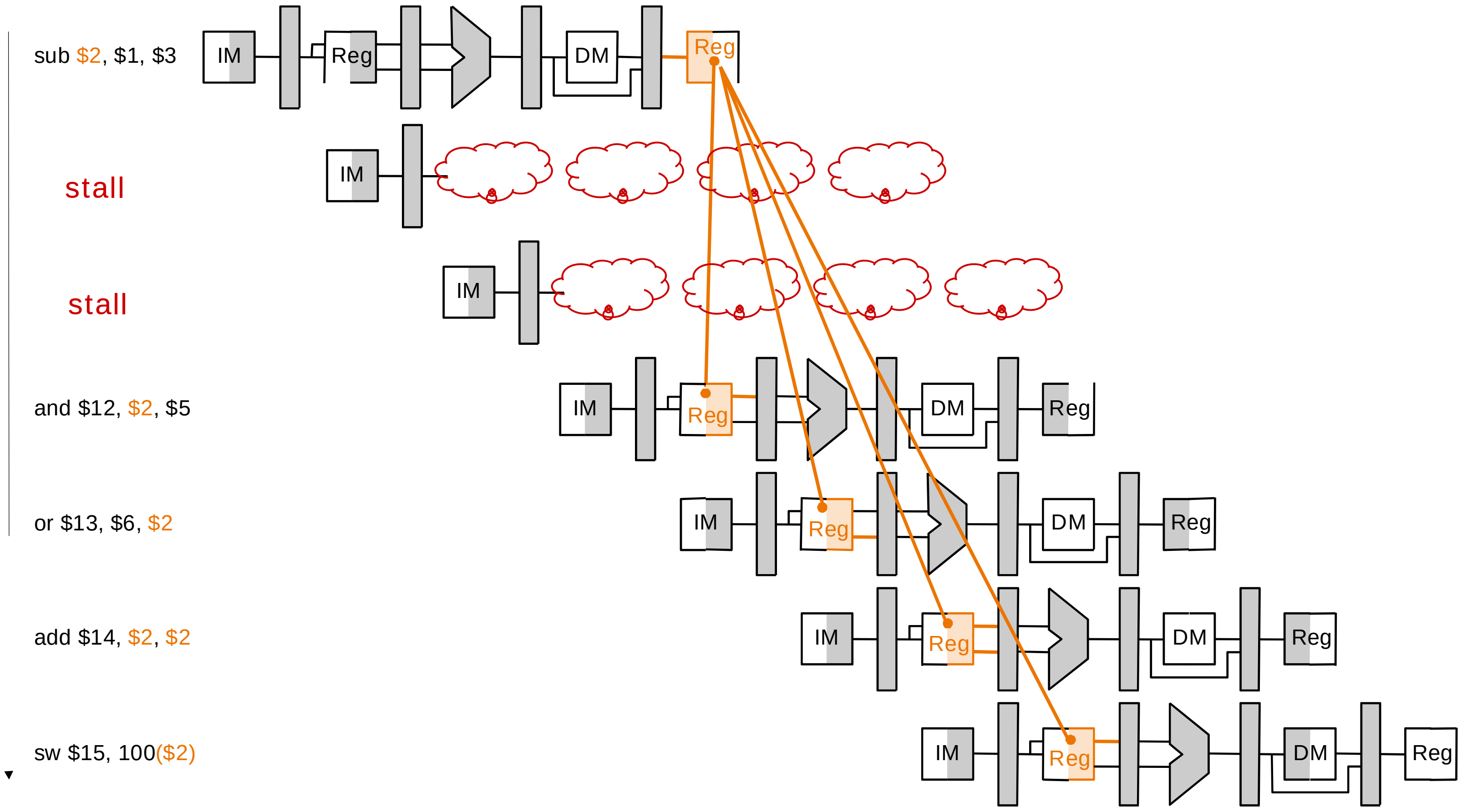
- Have compiler guarantee no hazards
- Where do we insert the “nops” ?

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

- Problem:  this really slows us down!

# Or Pipeline Stalls

bubble

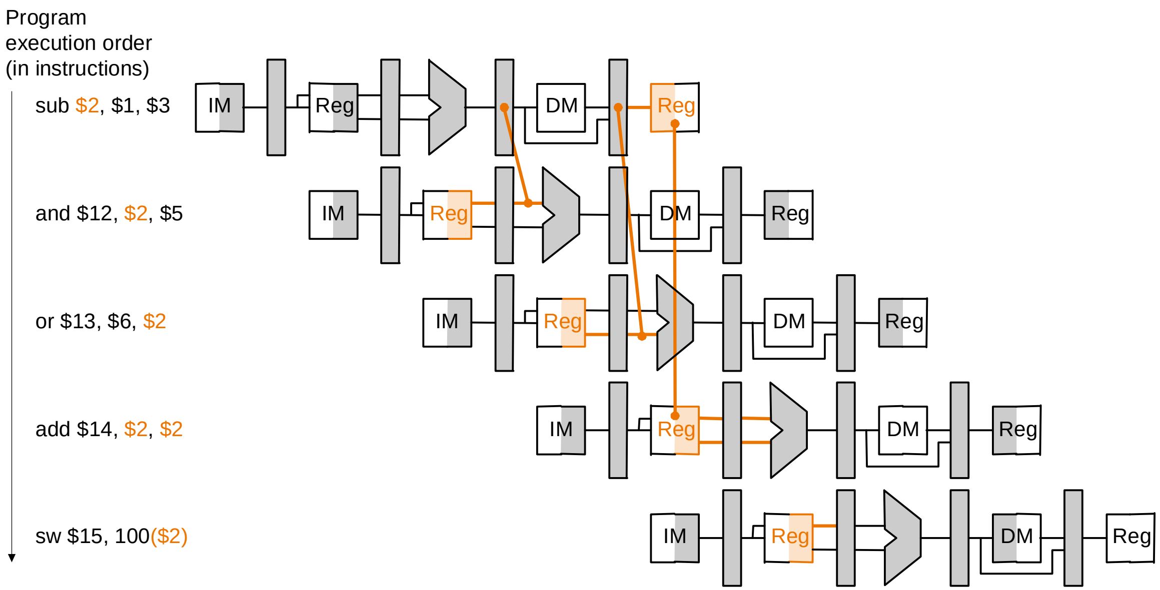


# Forwarding

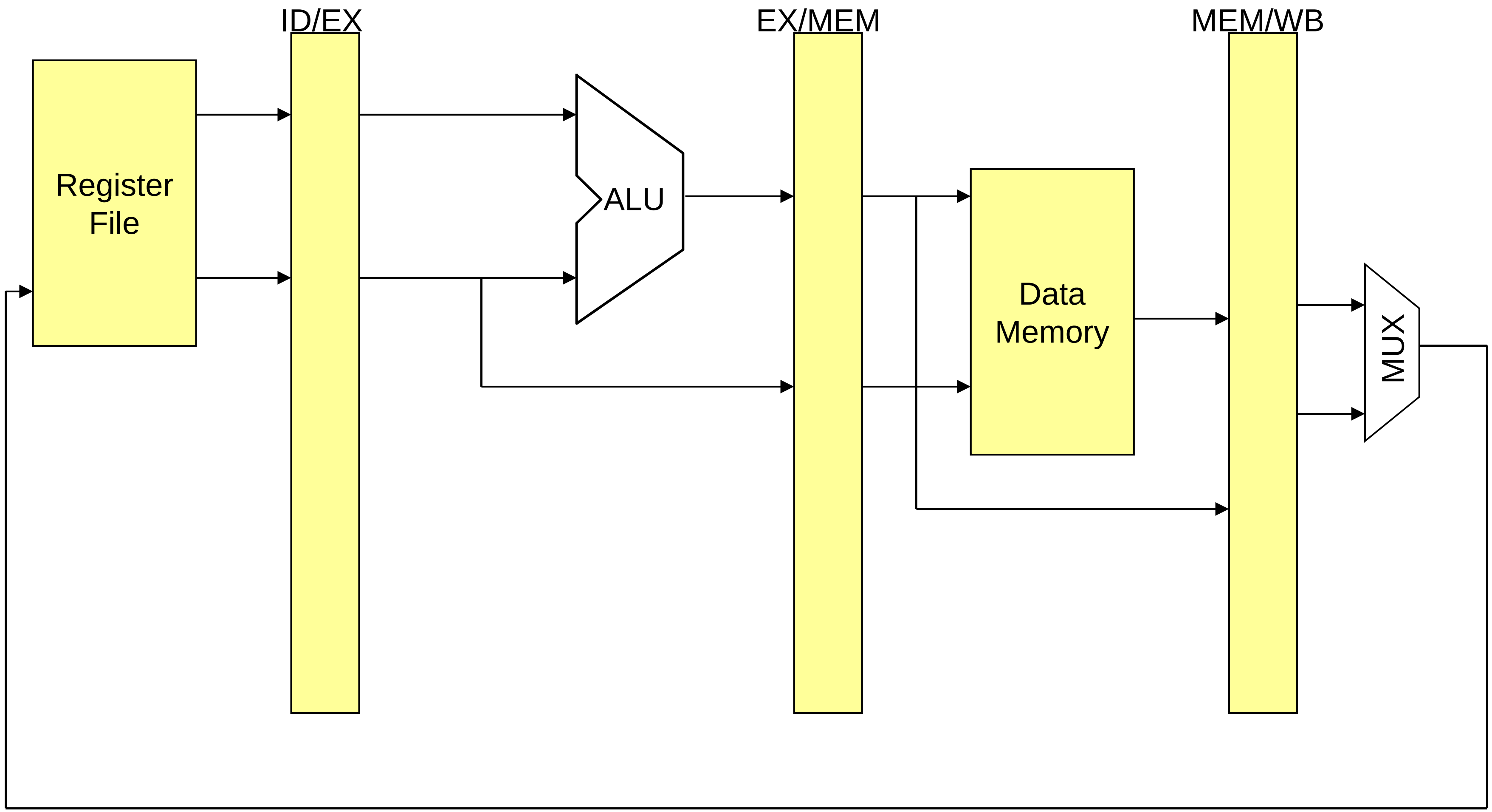
- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding

Time (in clock cycles) →

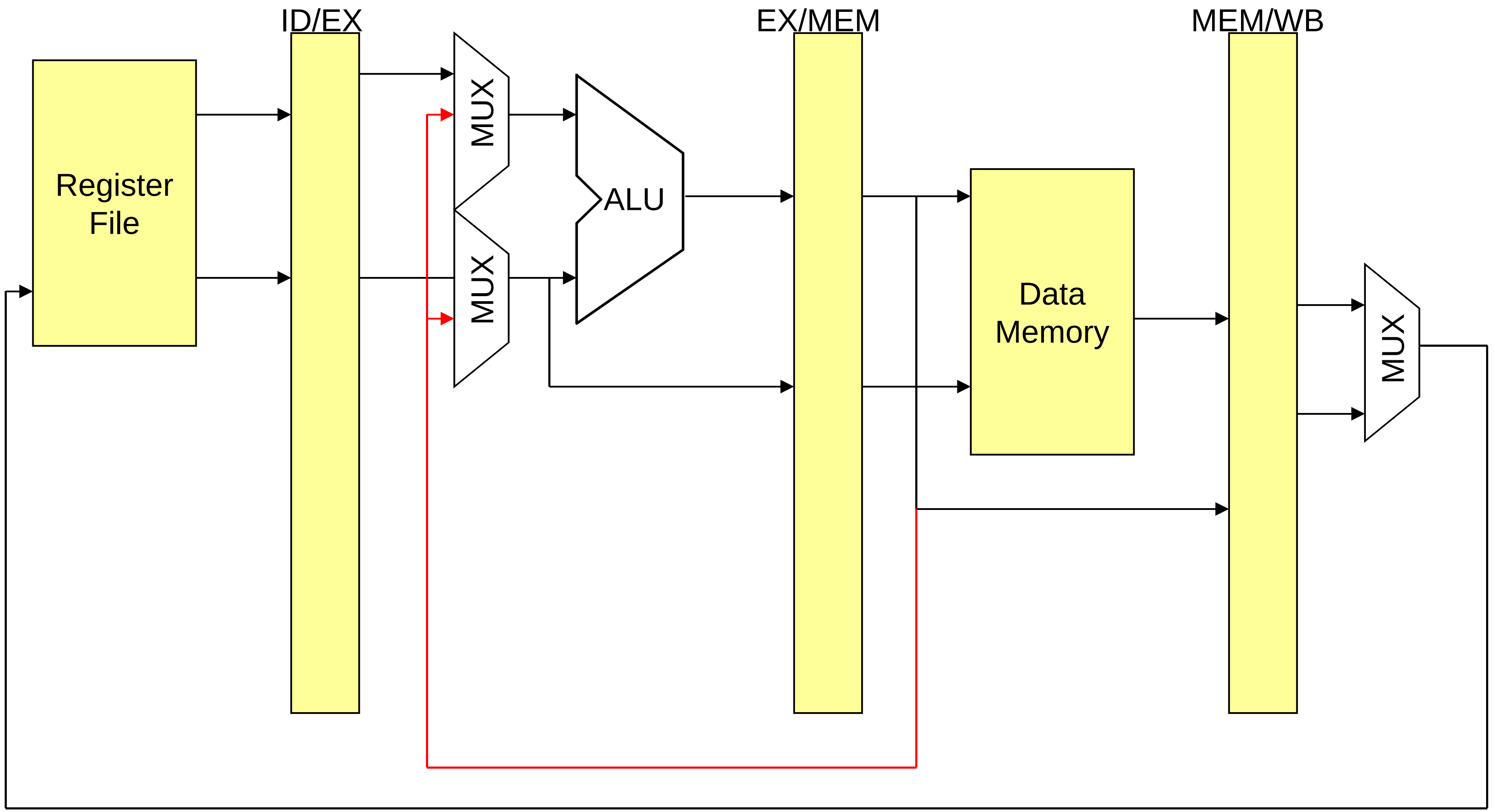
| | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value of register $2 : | 10 | 10 | 10 | 10 | 10/– 20 | – 20 | – 20 | – 20 | – 20 |
| Value of EX/MEM : | X | X | X | – 20 | X | X | X | X | X |
| Value of MEM/WB : | X | X | X | X | – 20 | X | X | X | X |



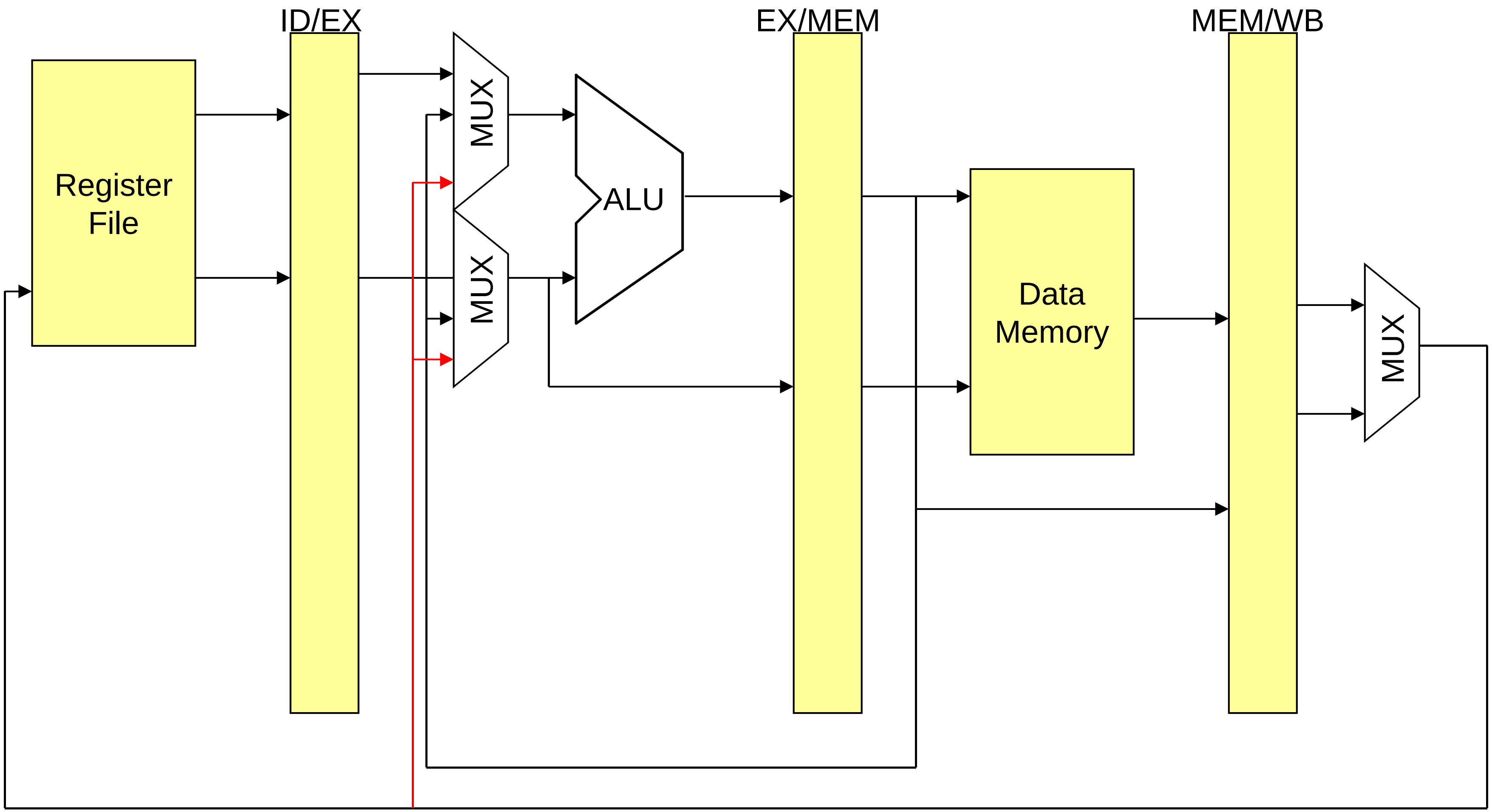
# Forwarding (simplified)



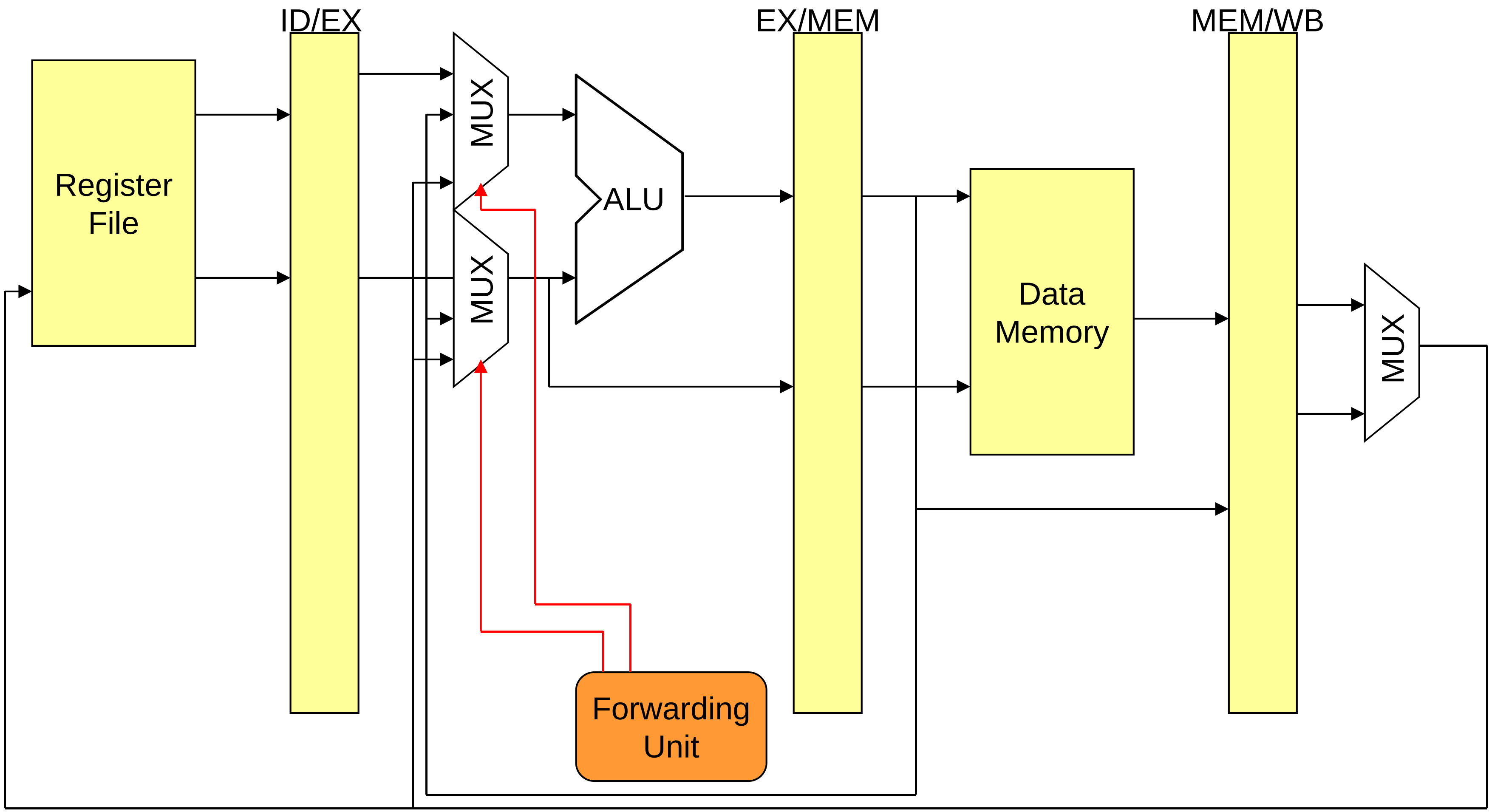
# Forwarding (from EX/MEM)



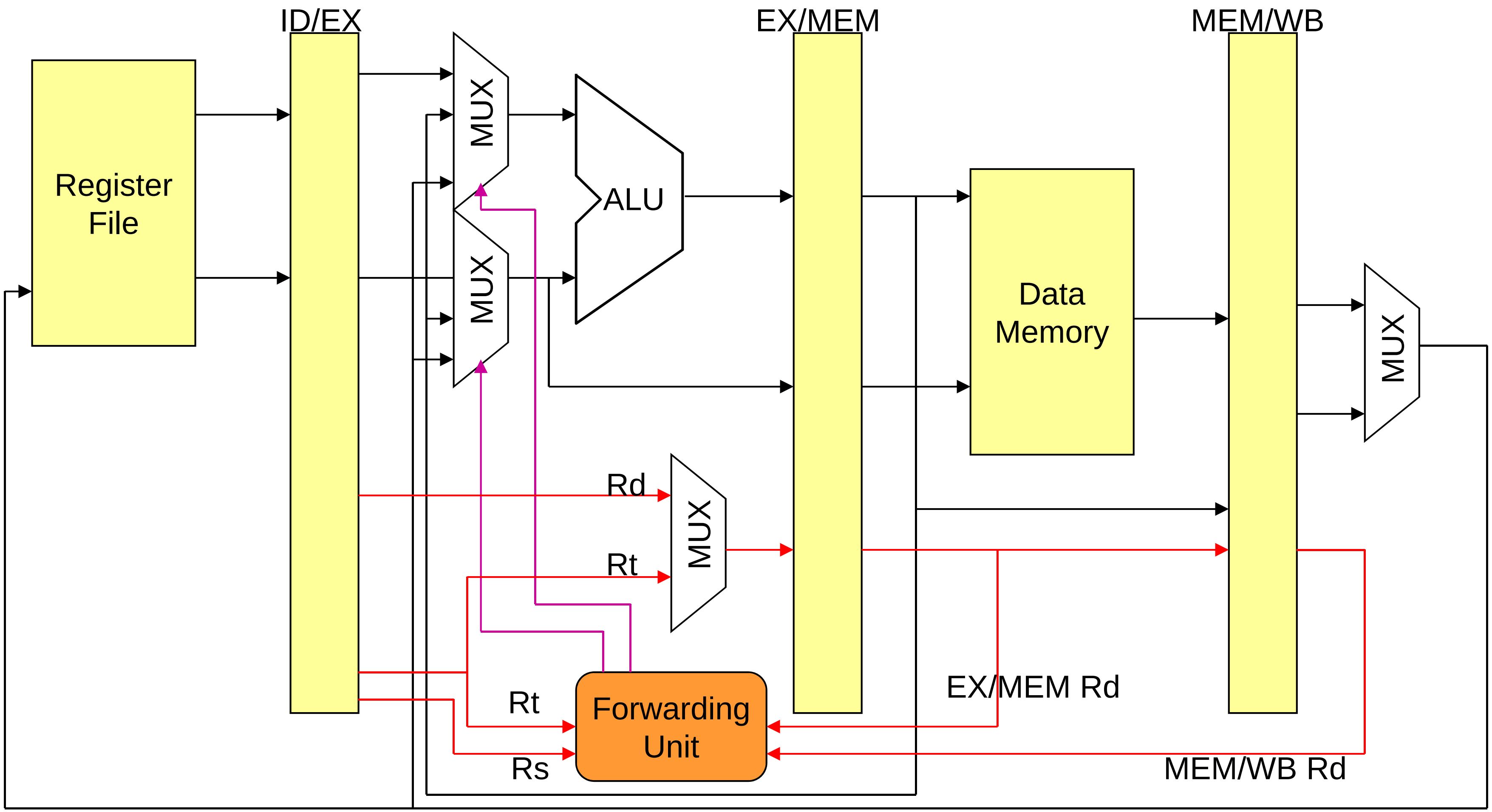
# Forwarding (from MEM/WB)



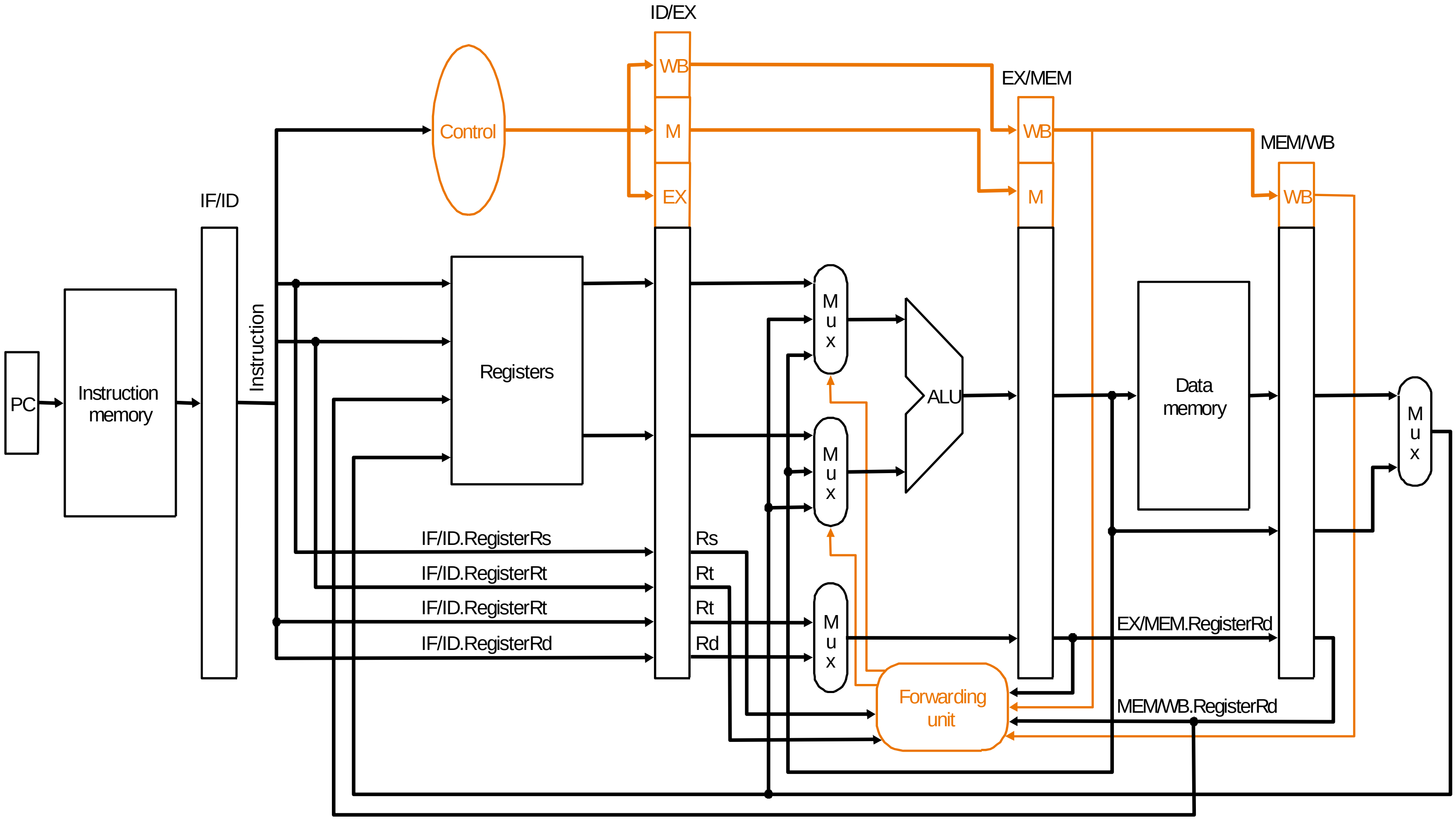
# Forwarding (operand selection)



# Forwarding (operand propagation)



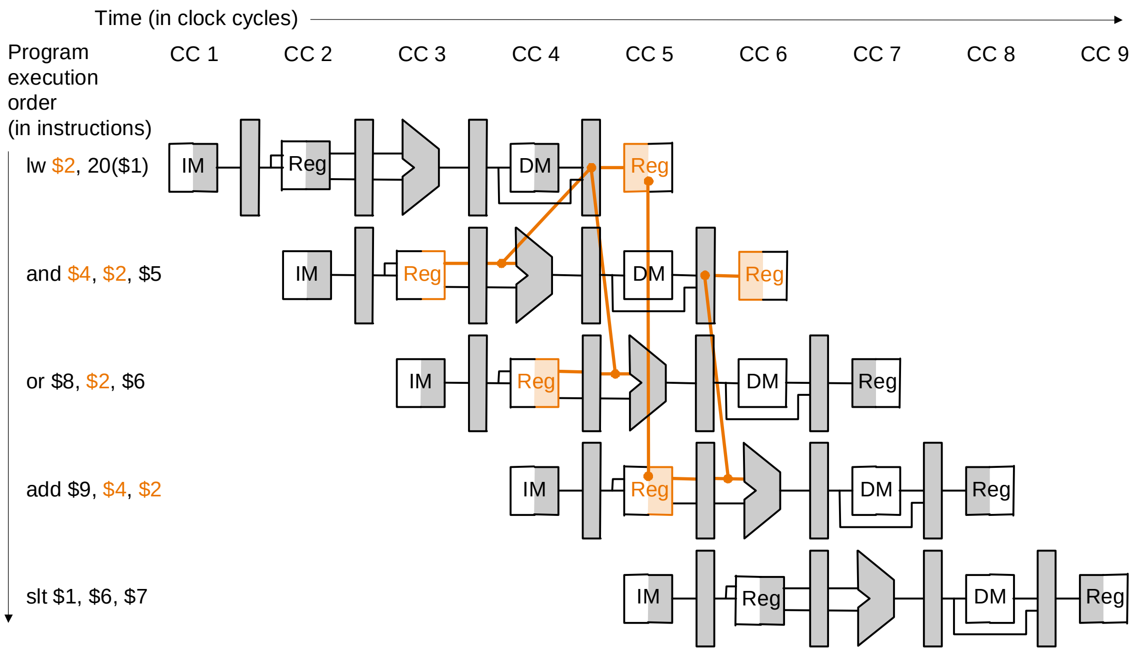
# Forwarding



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

# Can't always forward

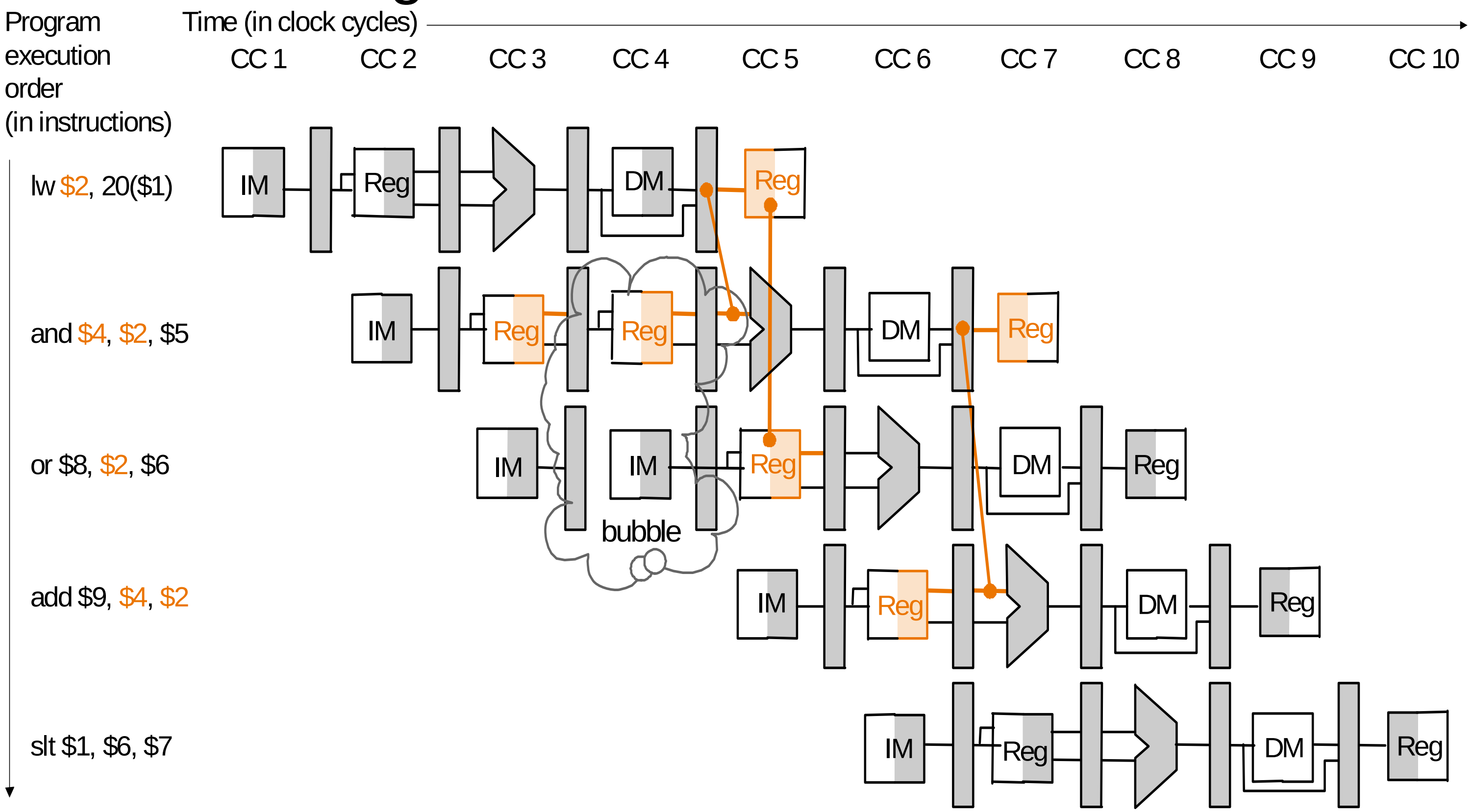
- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.




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

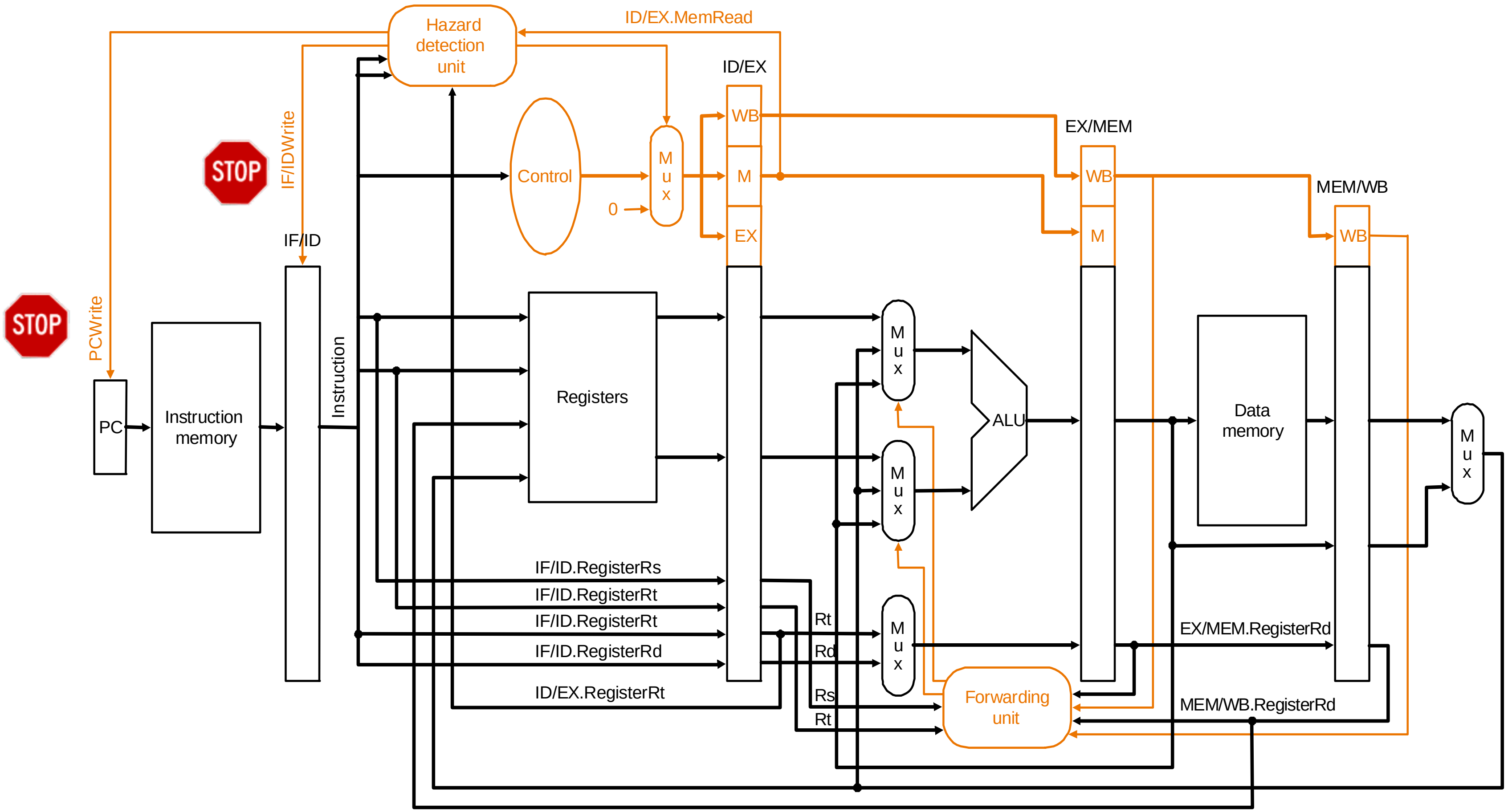
# Stalling

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



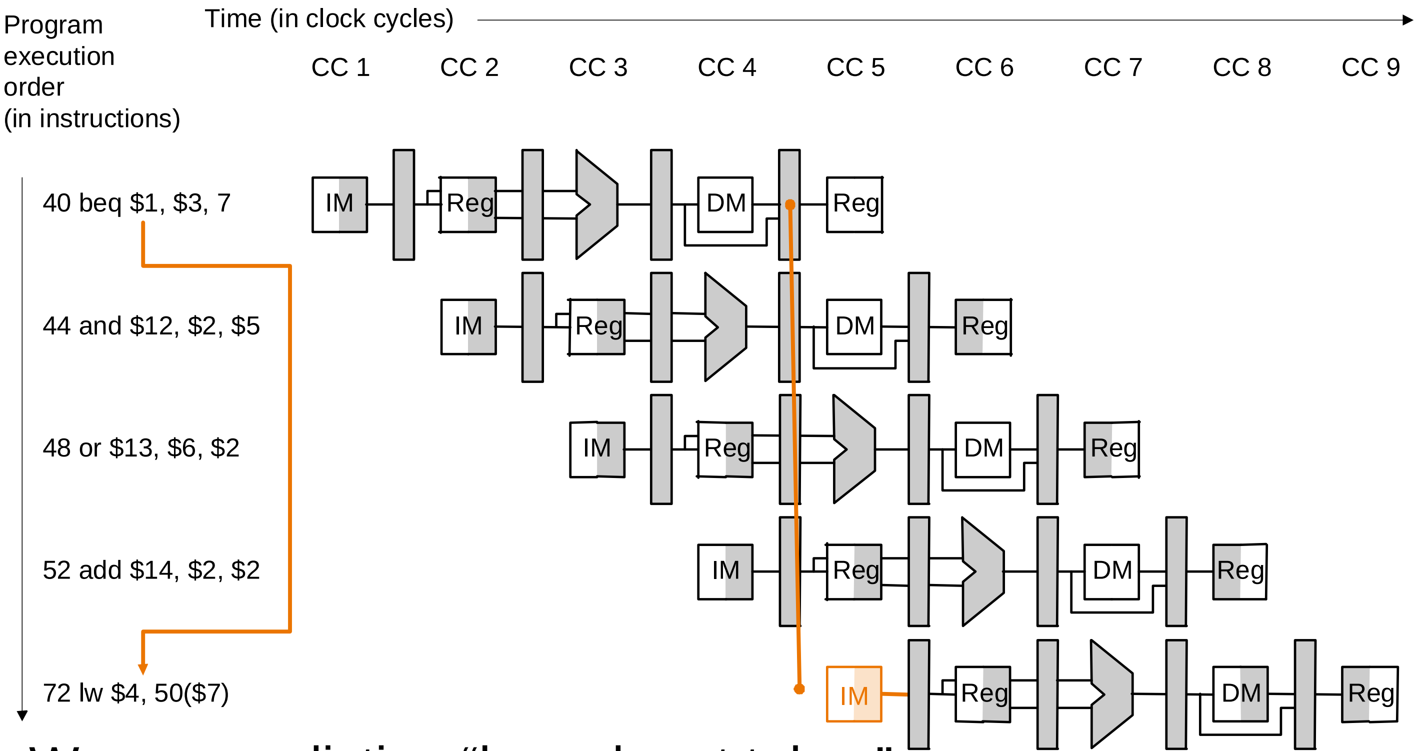
# Hazard Detection Unit

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



# Branch Hazards

- When we decide to branch, other instructions are in the pipeline!
- Assume: branch is not taken
- When assumption failed, flush 3 instructions

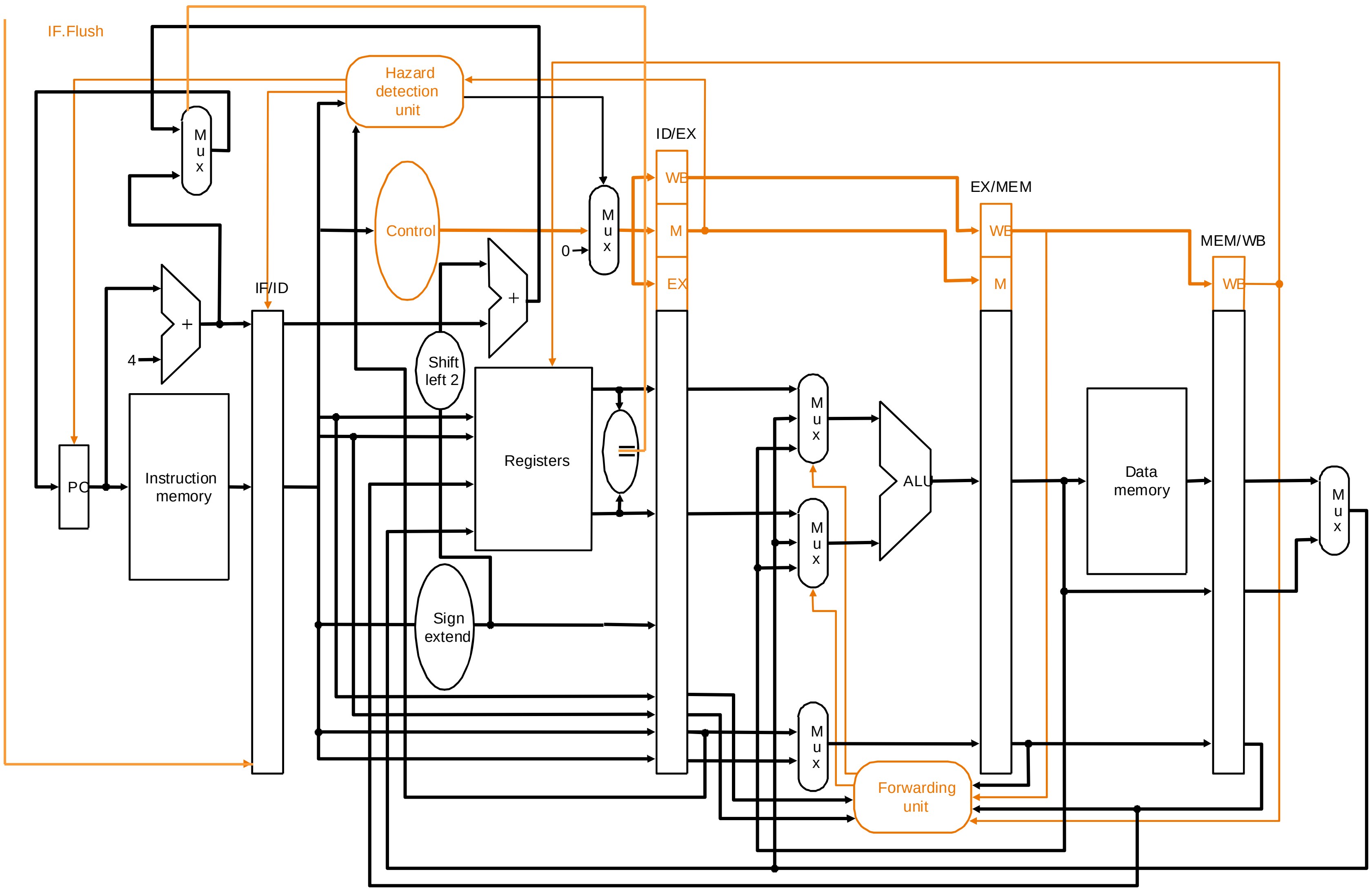



- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong

# Alleviate Branch Hazards

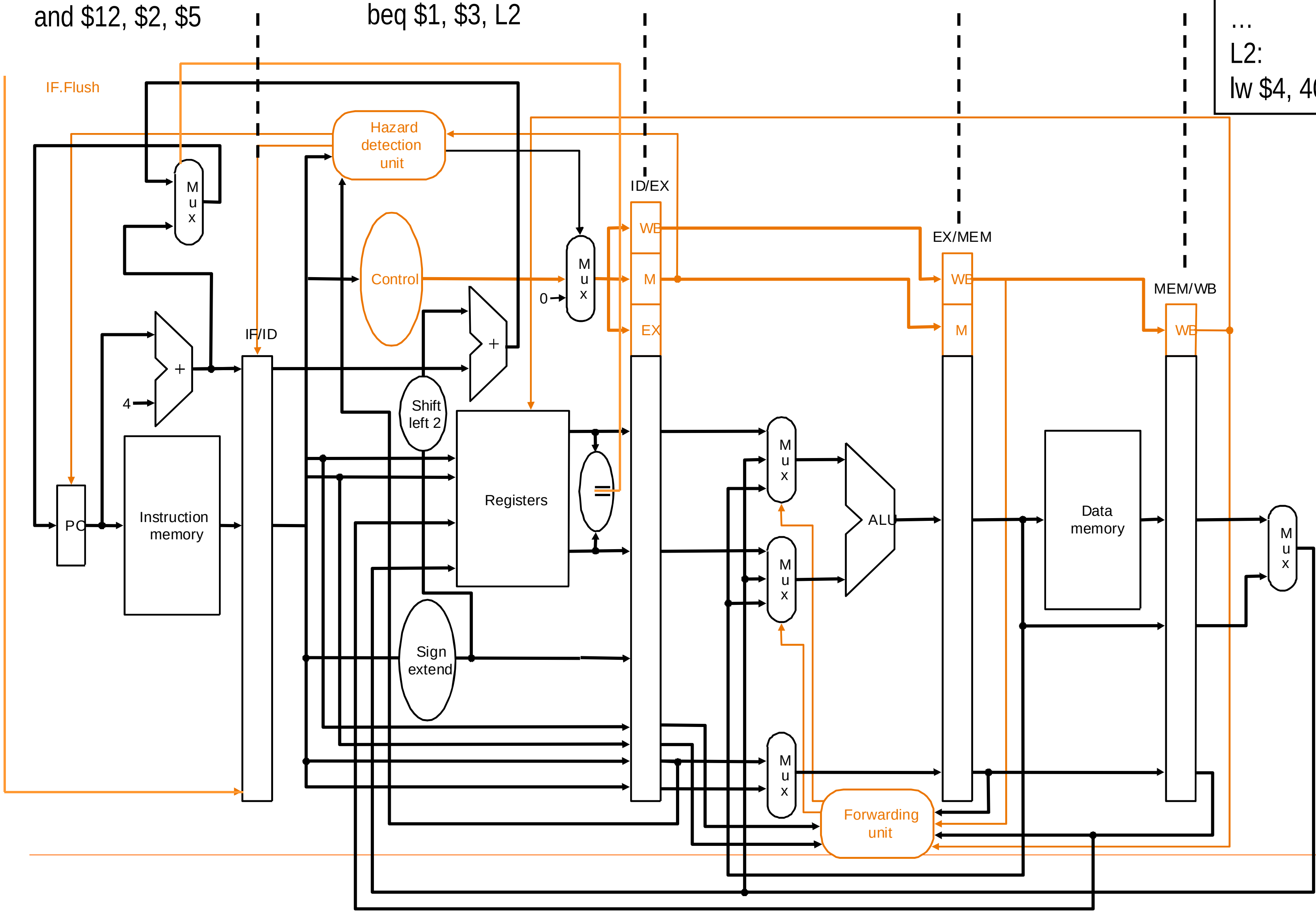
- Move branch compare to ID stage of the pipeline
- Add adder to calculate branch target in ID stage
- Add IF.flush signal that zeros the instruction (or squash) in IF/ID pipeline register
- Reduce penalty to 1 cycle

# Flushing Instructions



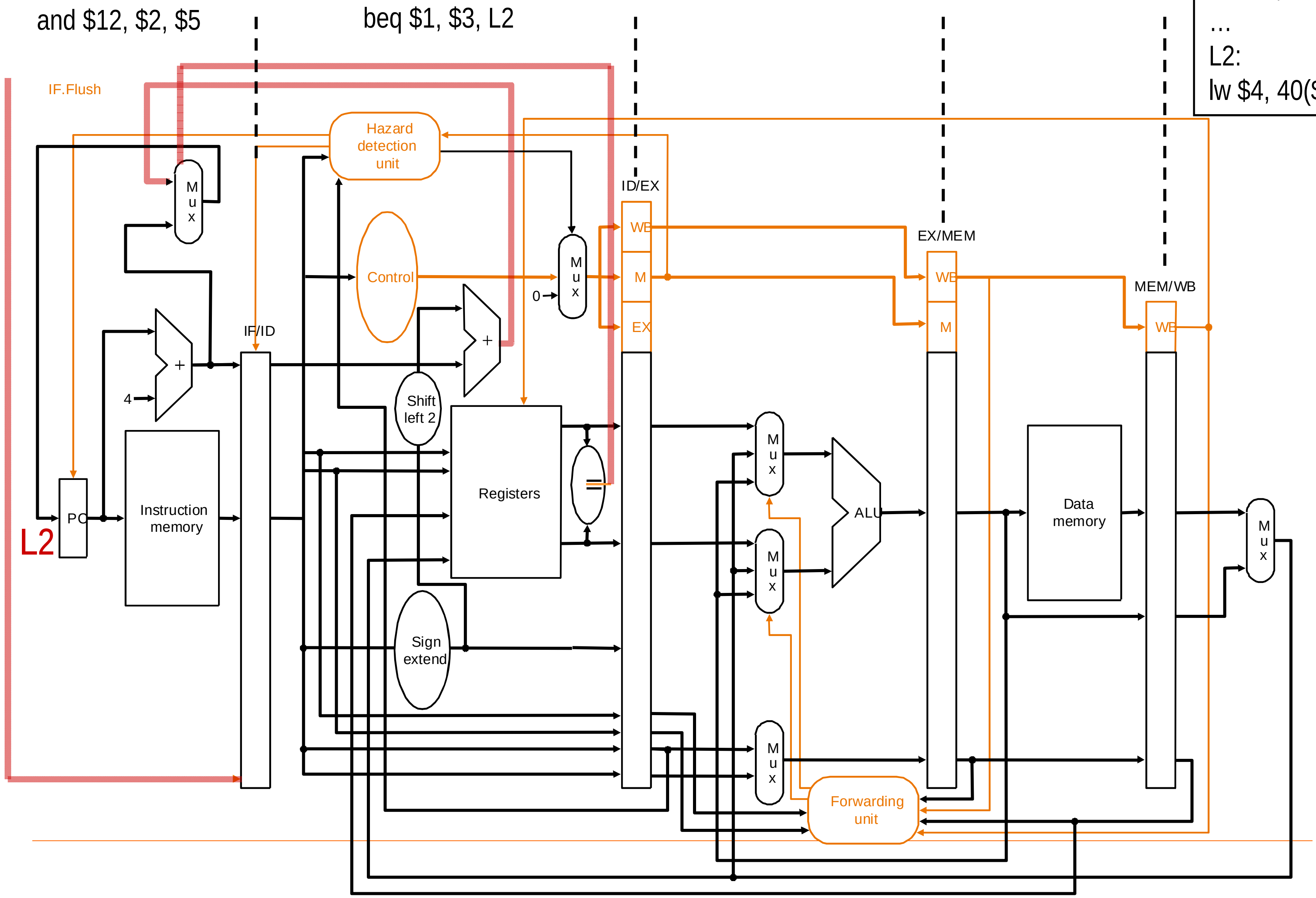
# Flushing Instructions (cycle N)

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



# Flushing Instructions (cycle N)

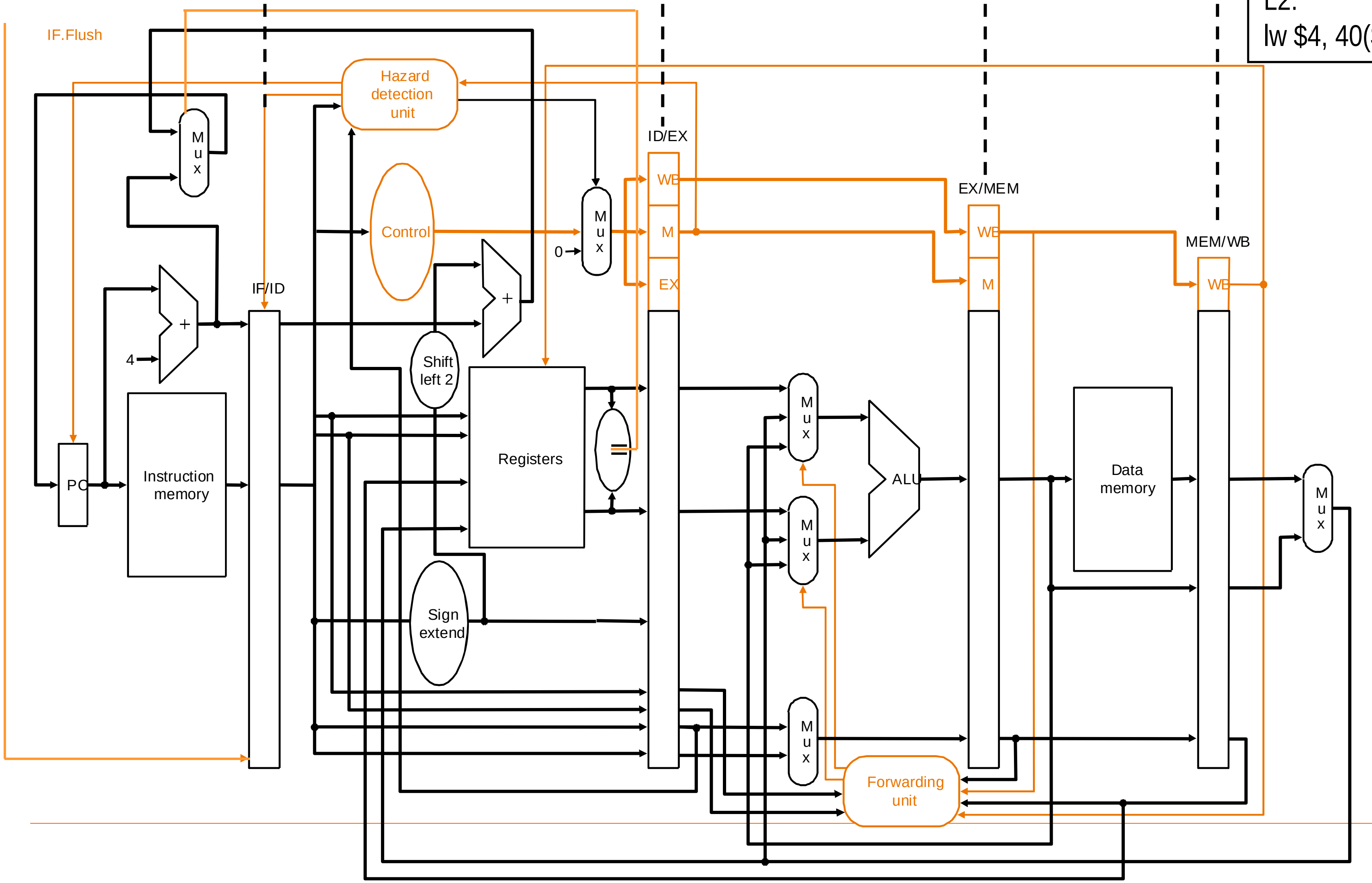
beq $1, $3, L2
and $12, $2, $5
or $13, $12, $1
…
L2:
lw $4, 40($7)



# Flushing Instructions (cycle N+1)

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

lw \$4, 40(\$7) | nop | beq \$1, \$3, L2



# Improving Performance

- Try and avoid stalls! E.g., reorder these instructions:

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

- Superscalar: start more than one instruction in the same cycle

- Most all processors are now pipelined and Superscalar

# Dynamic Scheduling

- The hardware performs the “scheduling”
  - hardware tries to find instructions to execute
  - out of order execution is possible
  - speculative execution and dynamic branch prediction
- All modern processors are very complicated
  - DEC Alpha 21264:  9 stage pipeline, 6 instruction issue
  - PowerPC and Pentium:  branch history table
  - Compiler technology important