# EEL 4768
# Computer Architecture

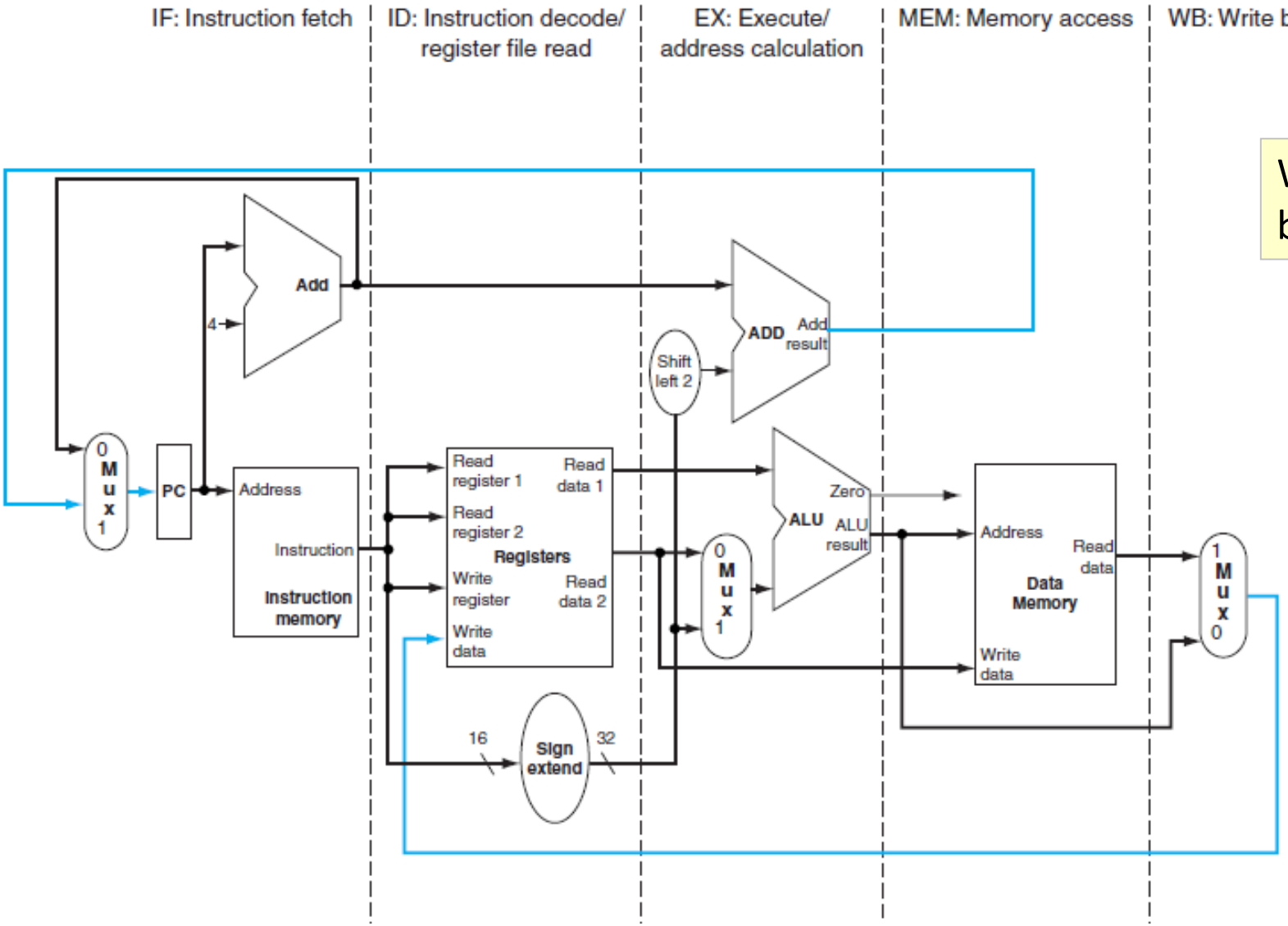
## Pipelined Datapath

# Outline

- Pipelined Datapath Implementation
- Forwarding Detection

# Pipelined Datapath

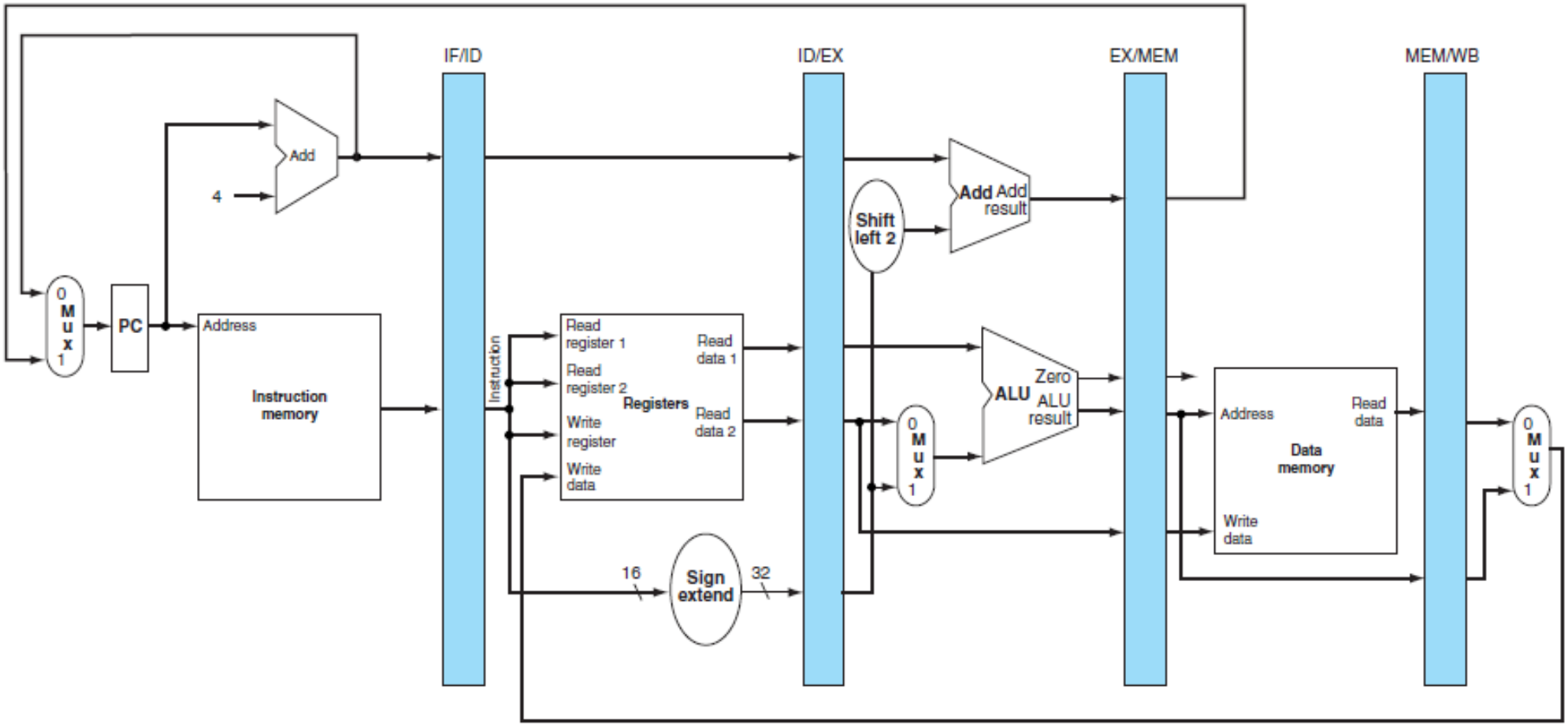
- Up to 5 instructions can be in the datapath at the same time
- Separate memories: Instruction Memory and Data Memory




We're doing branch in 4th stage

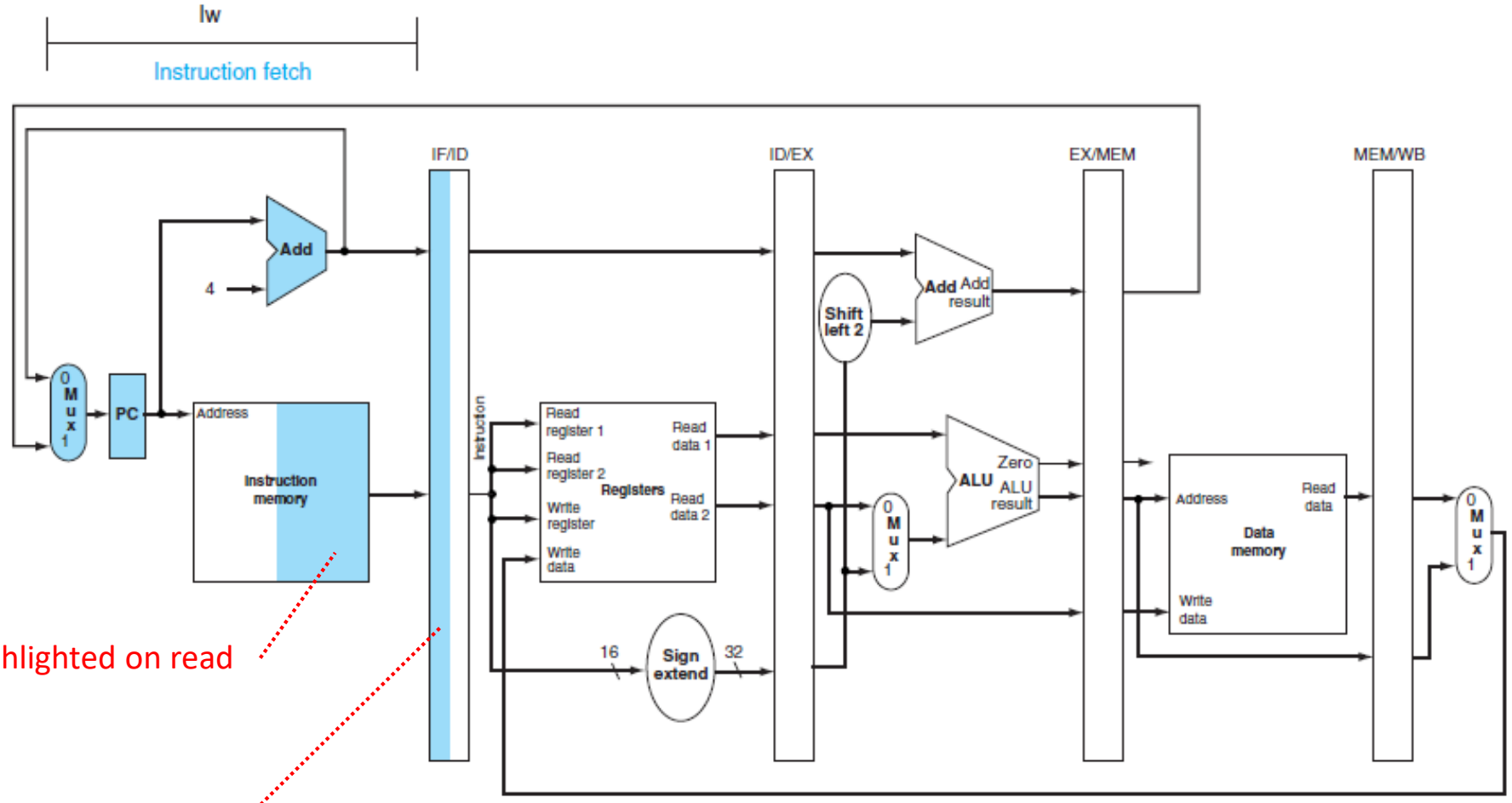
# Pipelined Datapath

- Each stage contains a different instruction
- We use registers between the stages



# Load Word (lw) Instruction in the IF Stage

- PC+4 is written to PC
- PC+4 is also passed in the IF/ID register in case it's needed for the branch address

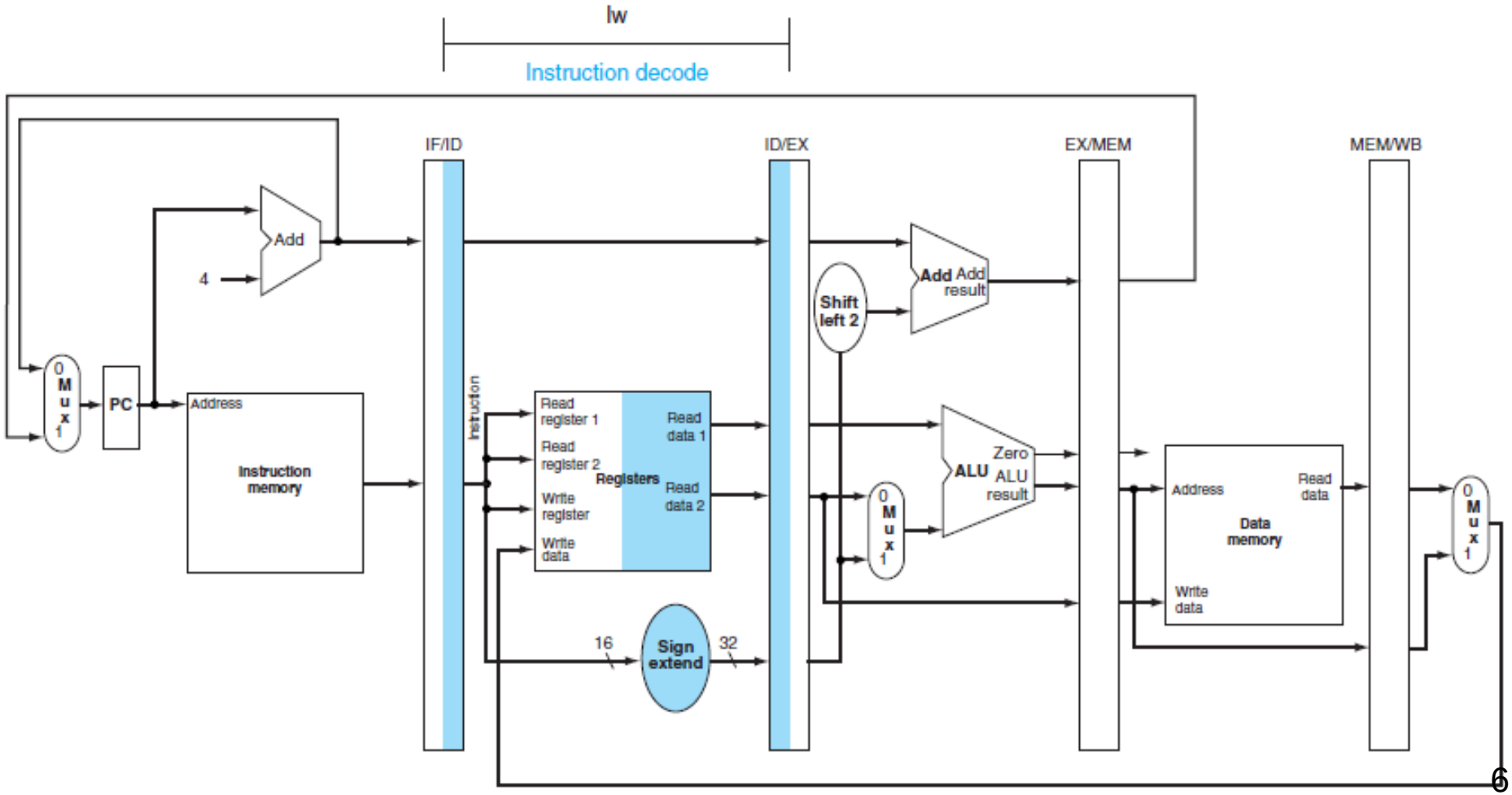



Right side highlighted on read

Left side highlighted on write

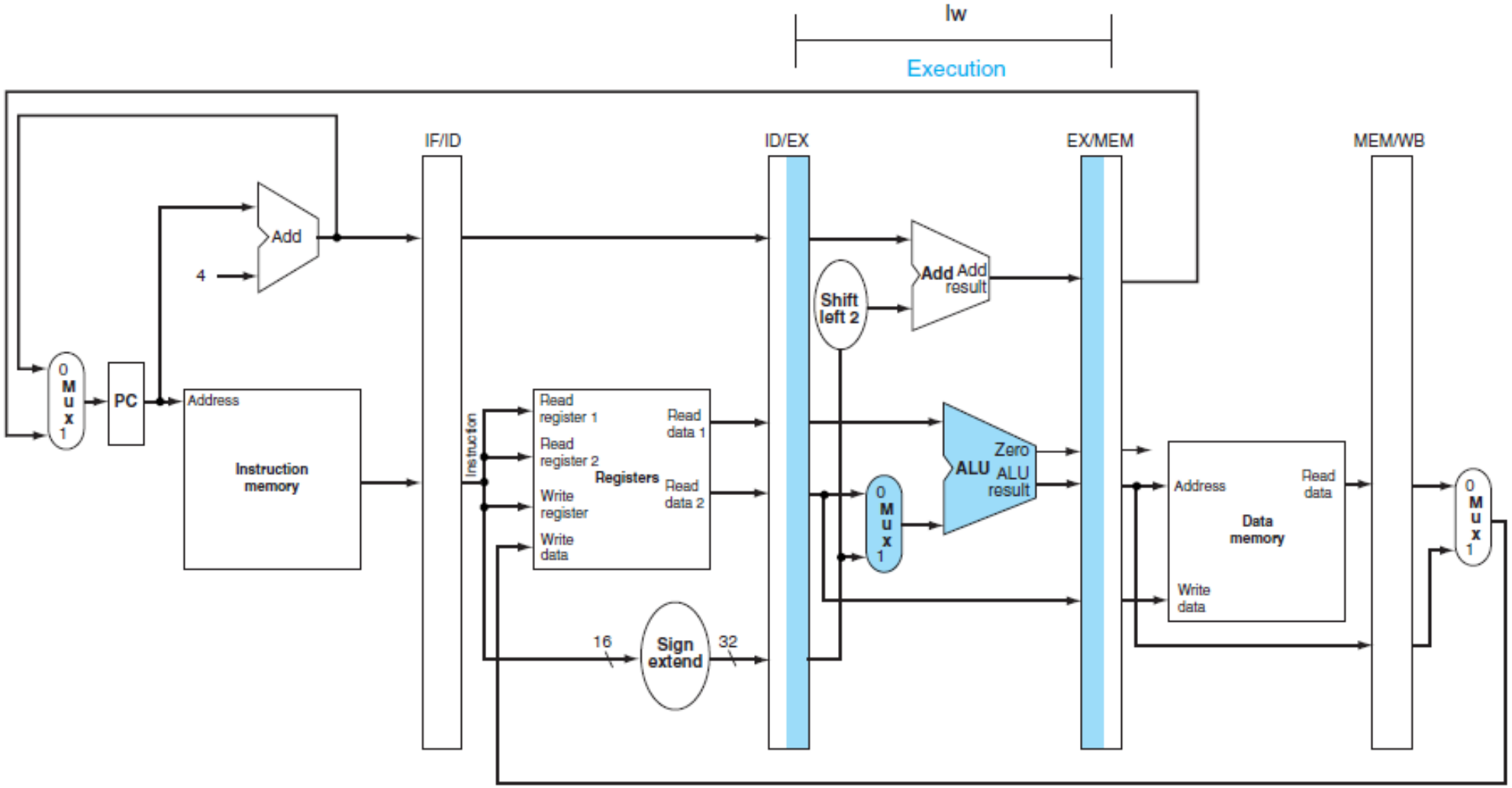
# Load Word (lw) Instruction in the ID Stage

- 16-bit number, sign-extended is written in ID/EX register
- Two read registers are written in ID/EX register
- PC+4 is written to ID/EX register



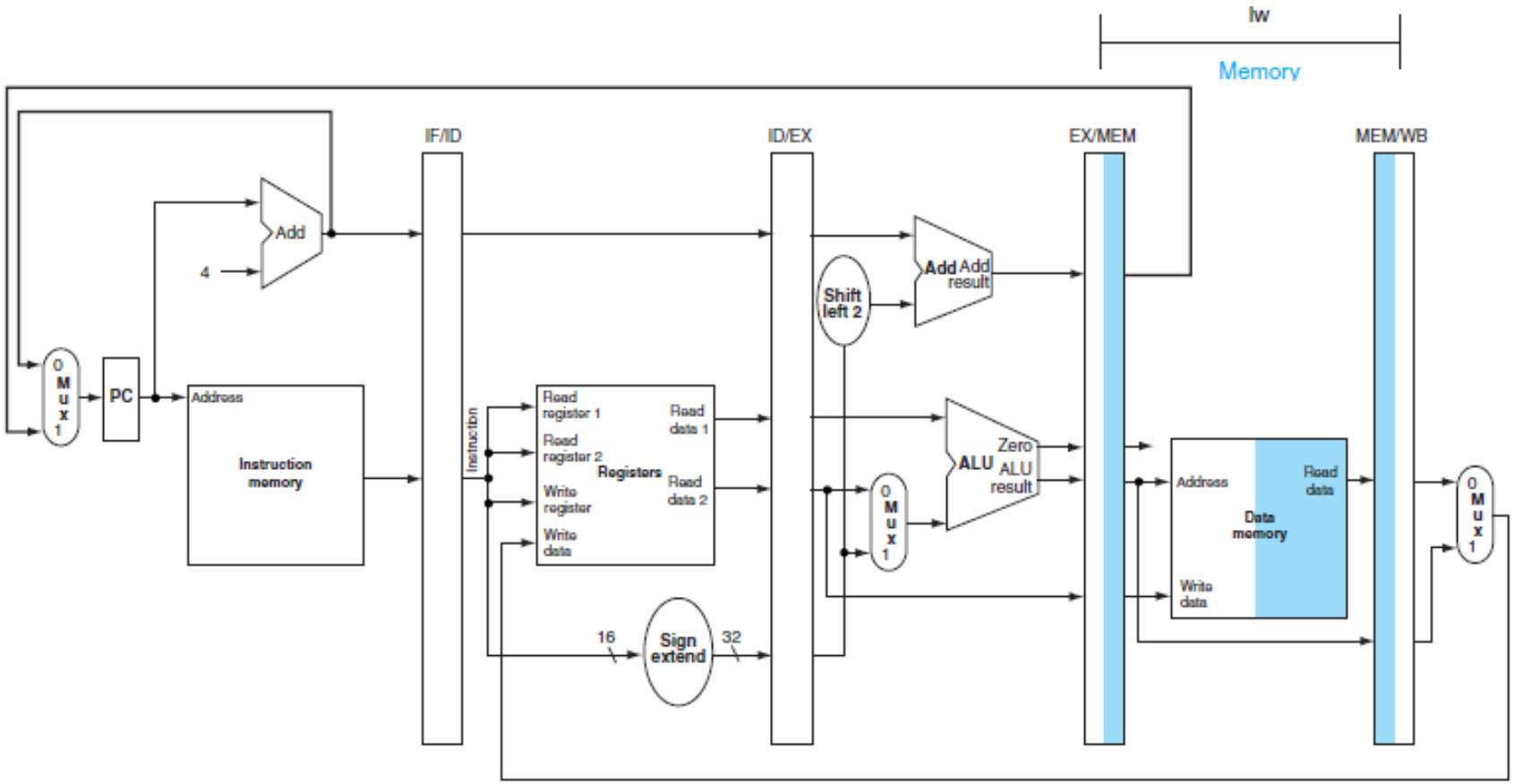
# Load Word (lw) Instruction in the EX Stage

- ALU gets the register containing the base address
- It adds to it the sign-extended 16-bit offset
- The computed address is written in the EX/MEM register



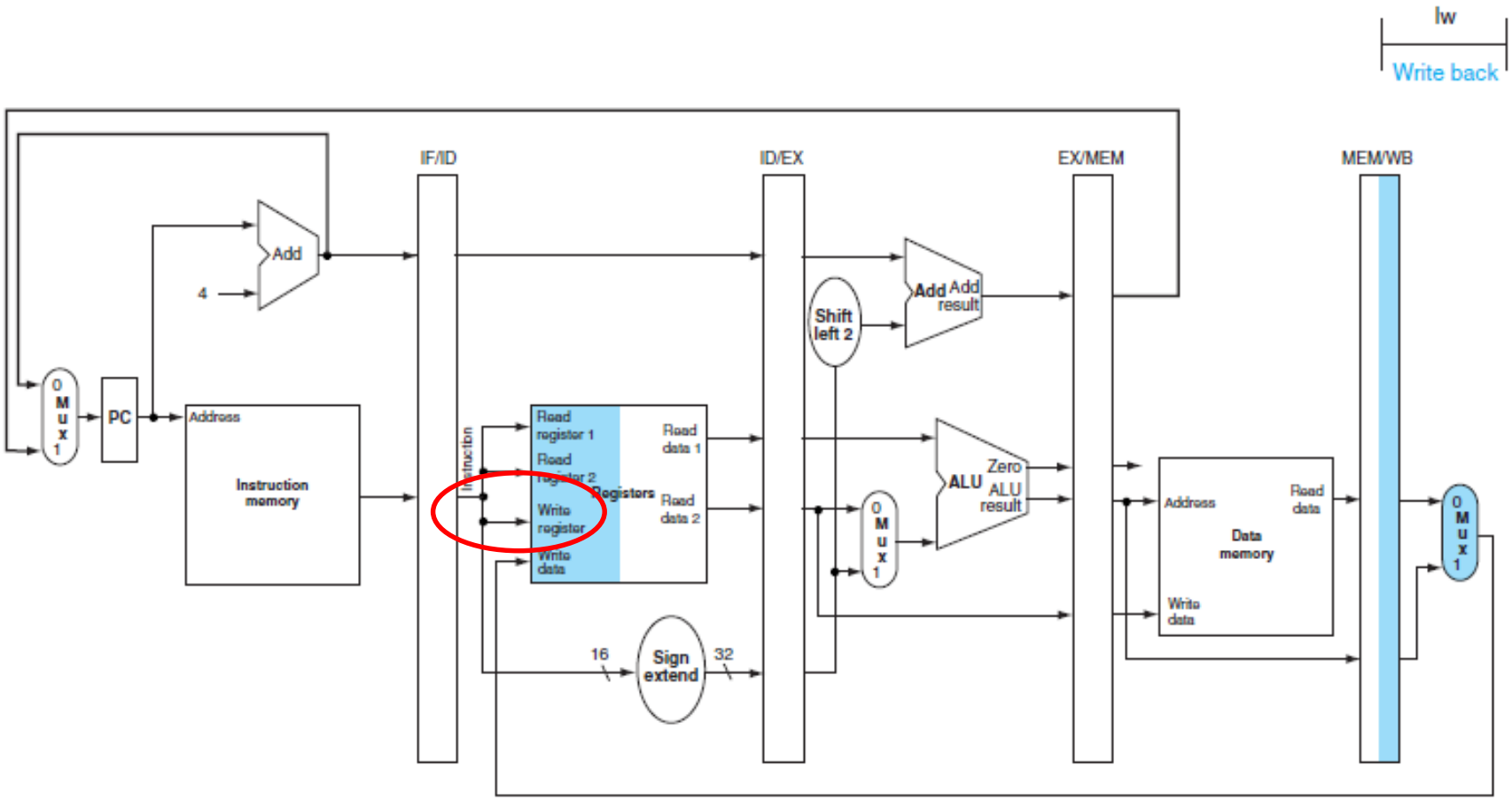
# Load Word (lw) Instruction in the MEM Stage

- The address goes out of the EX/MEM register and into the memory
- The read data is written into the MEM/WB register



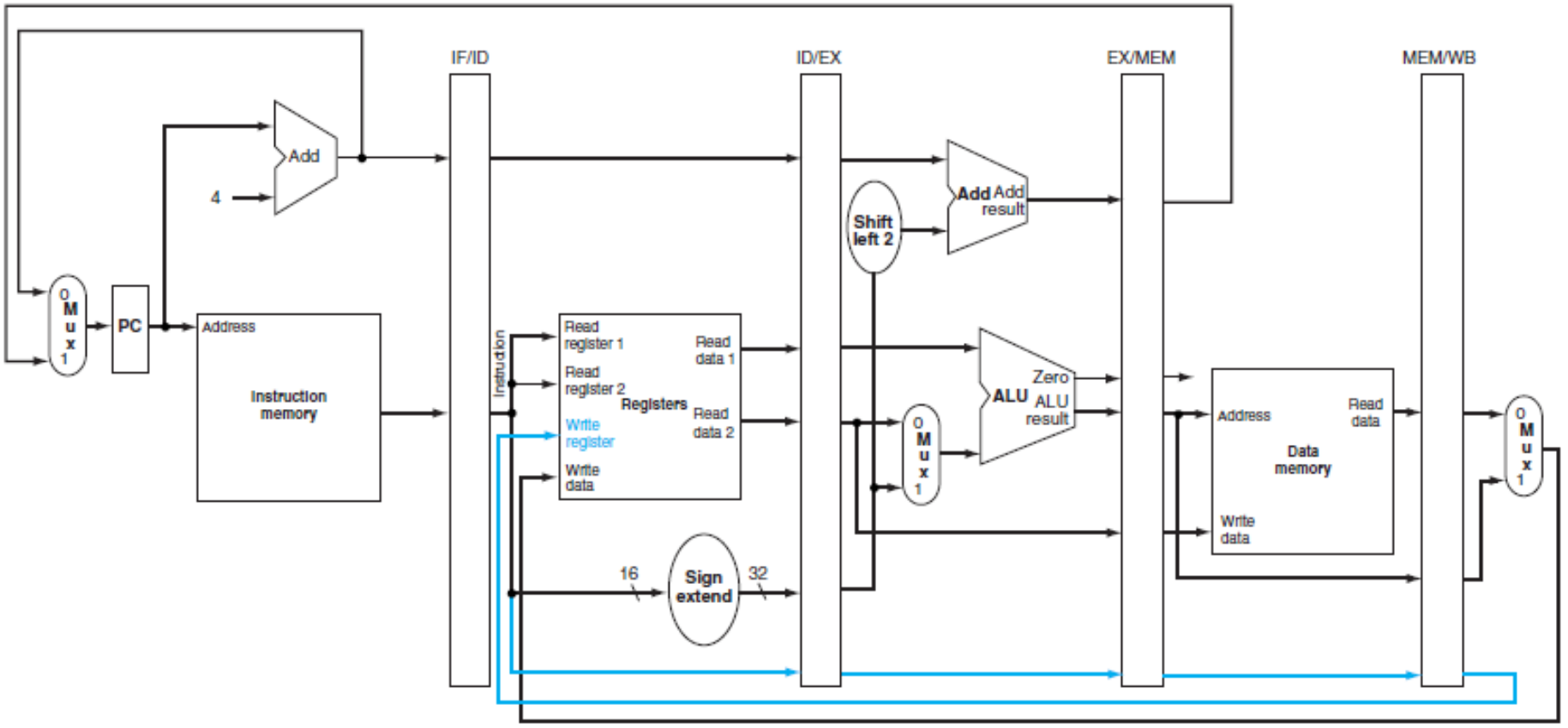
# Load Word (lw) Instruction in the WB Stage

- The data goes out of he MEM/WB register
- It is written in the register file




Problem:
The “write register” is coming from the instruction that’s in the IF/ID stage.
This is not the load instruction!

# Load Word (lw) Instruction in the WB Stage




When the load instruction is in ID stage, it sends the “write register” field to the EX, MEM and WB stages through the stage registers.

# What Should we Pass to the Next Stage?

- The five stages are, in order of execution:
  - IF, ID, EX, MEM, WB
- Anything that is needed at a later stage should be passed to the next stage in the stage register
- Example:
  - Register address, operands, 16-bit number, PC+4, …

# Representing the Pipelined Datapath

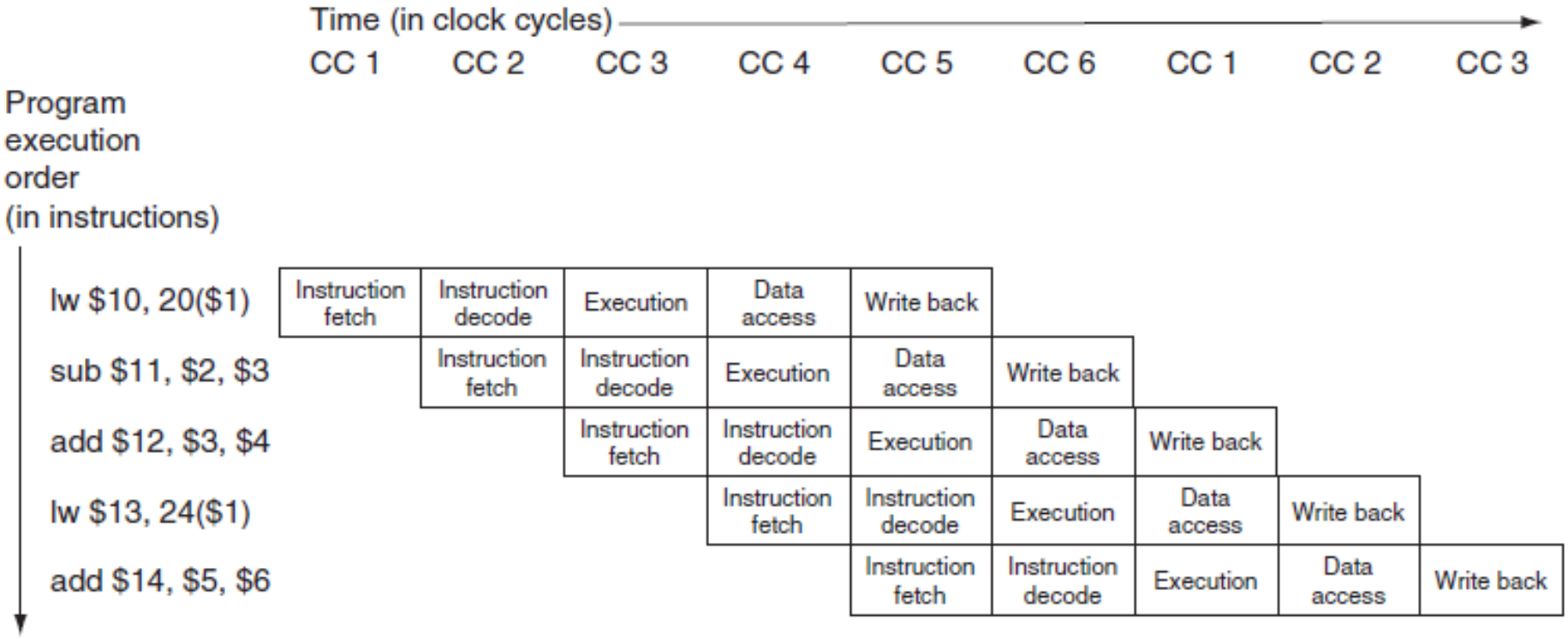
```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

- We'll see two ways to represent this code on the pipelined datapath

# Representing a Code Execution in a Diagram

The figure below is one way to represent this code in a diagram

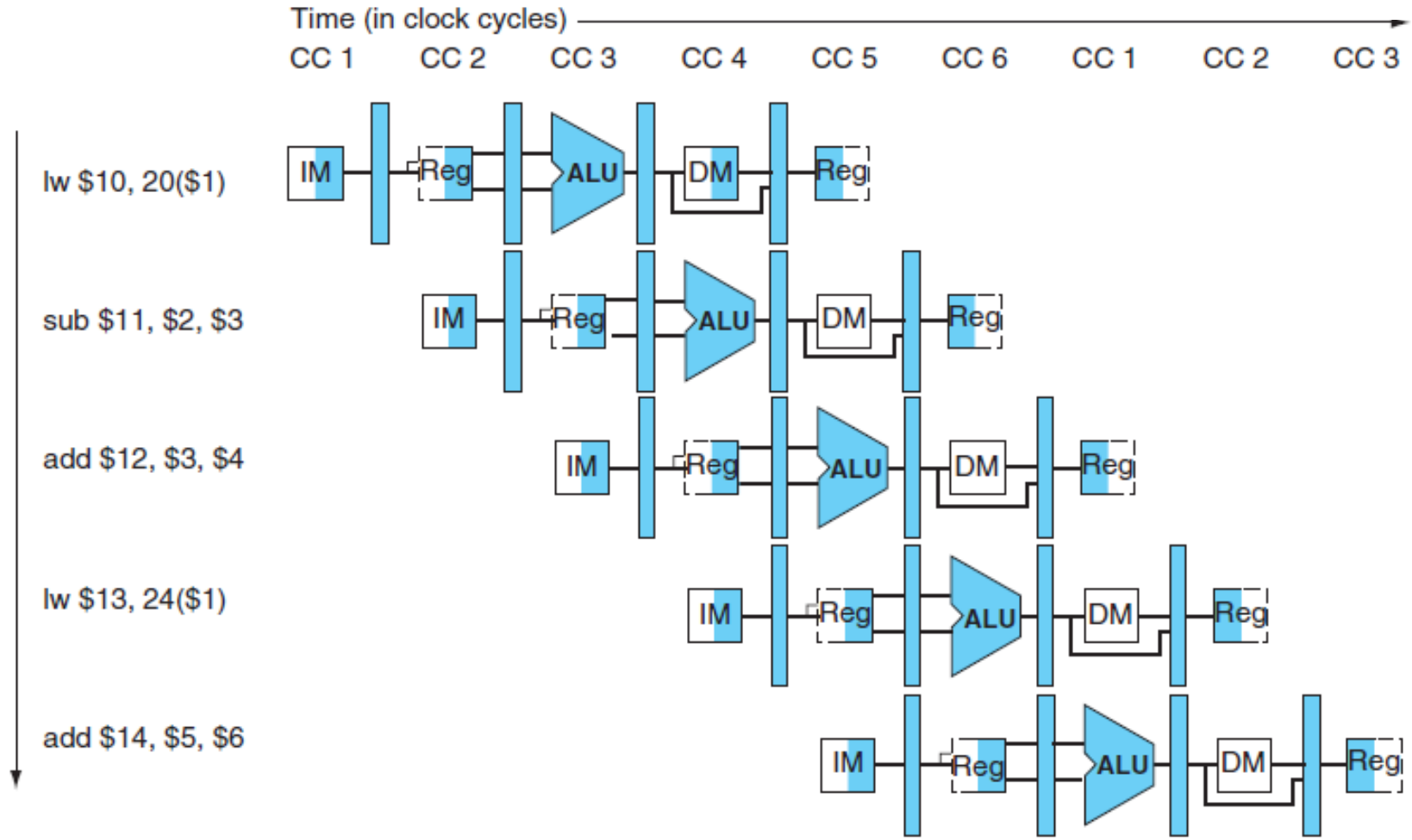
```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```



# Representing a Code Execution in a Diagram

The figure below is another way to represent this code in a diagram

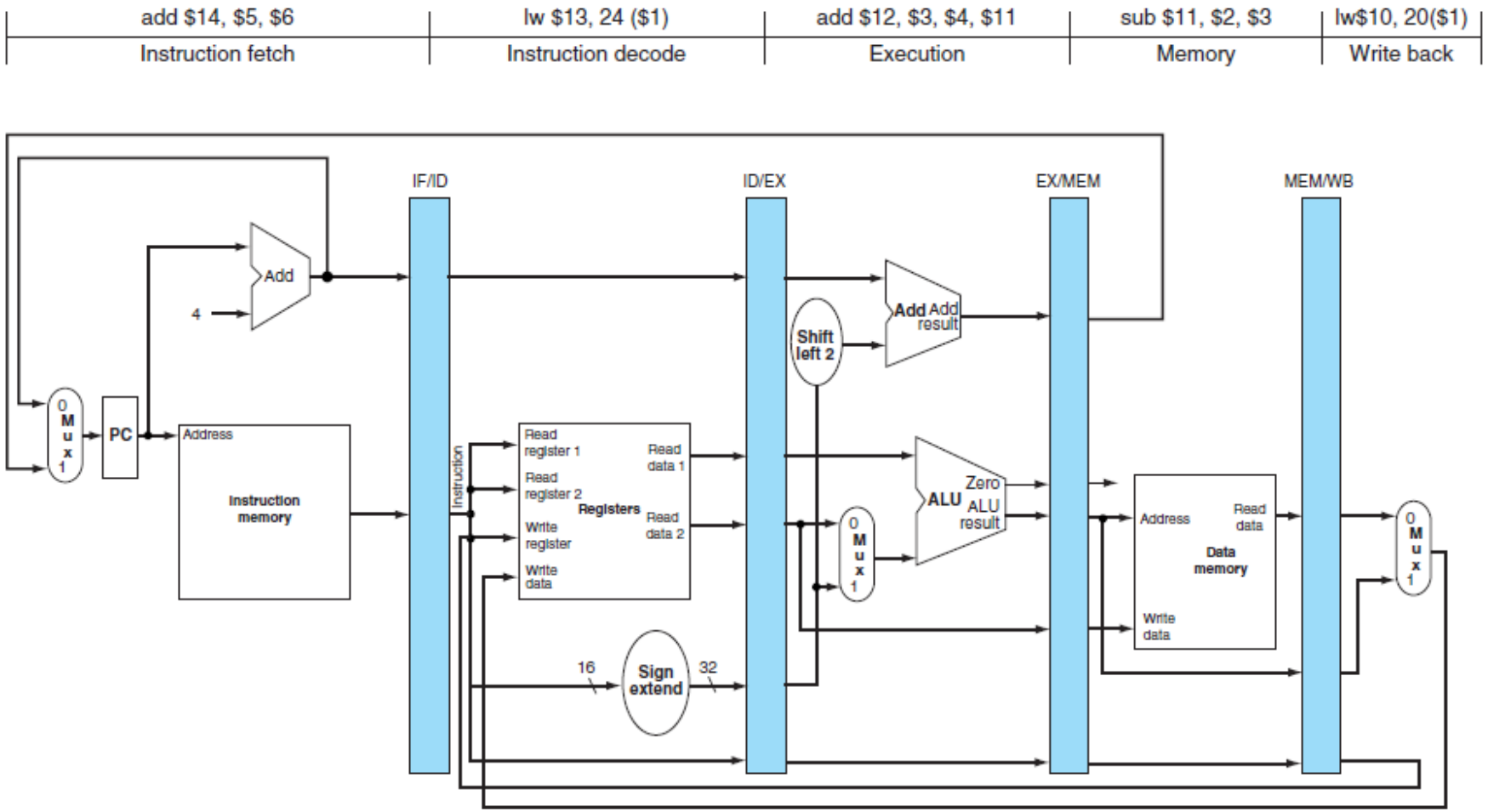
```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```



# Snapshot of the Datapath during a Clock Cycle

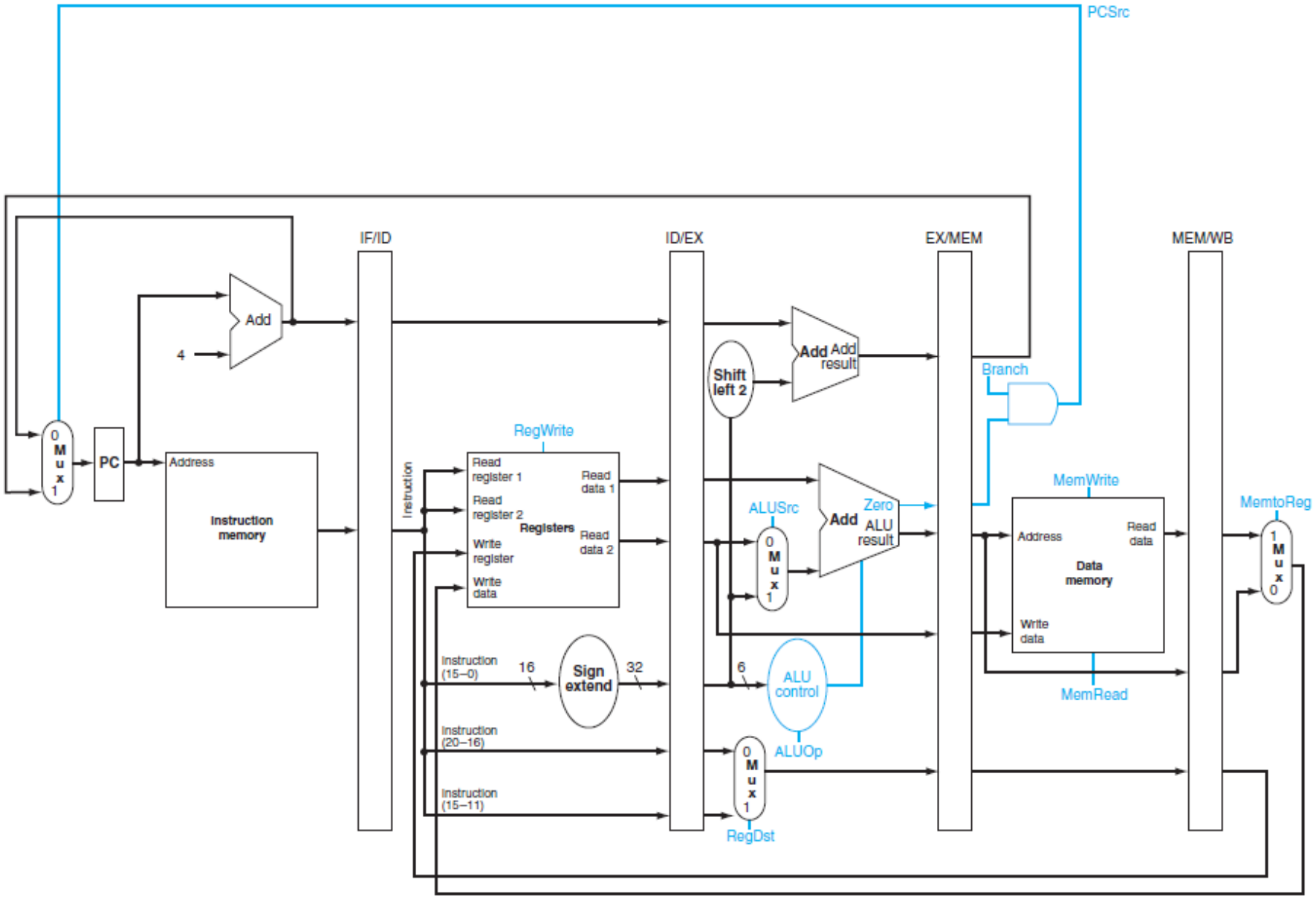
During one clock cycle, the datapath contains all the five instructions of the code.

```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```



# Control Signals

- We add the control signals
- The control signals are split over the five stages

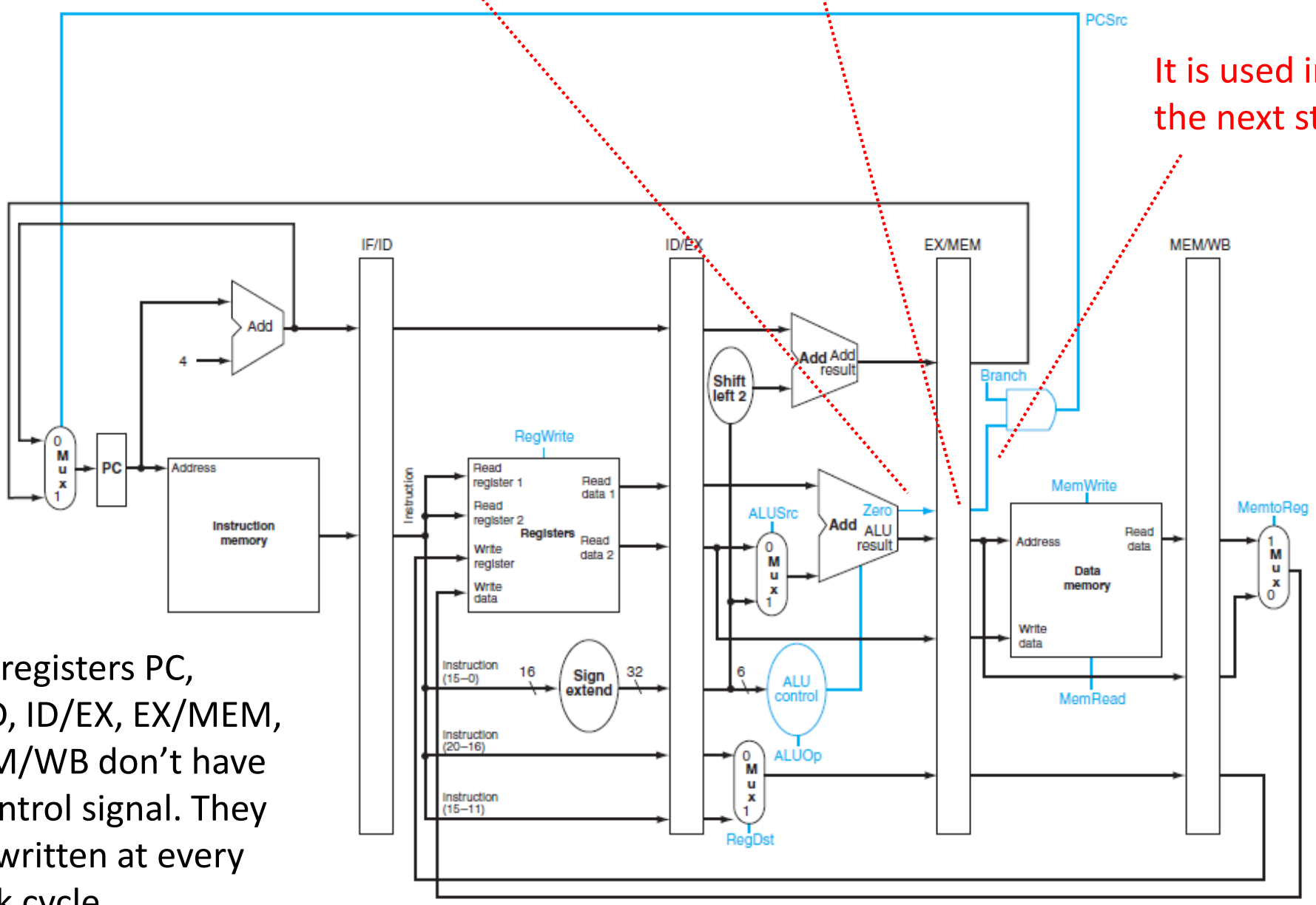


# Control Signals

Here, we know the result of branch (beq)

The signal goes into the stage register

It is used in the next stage



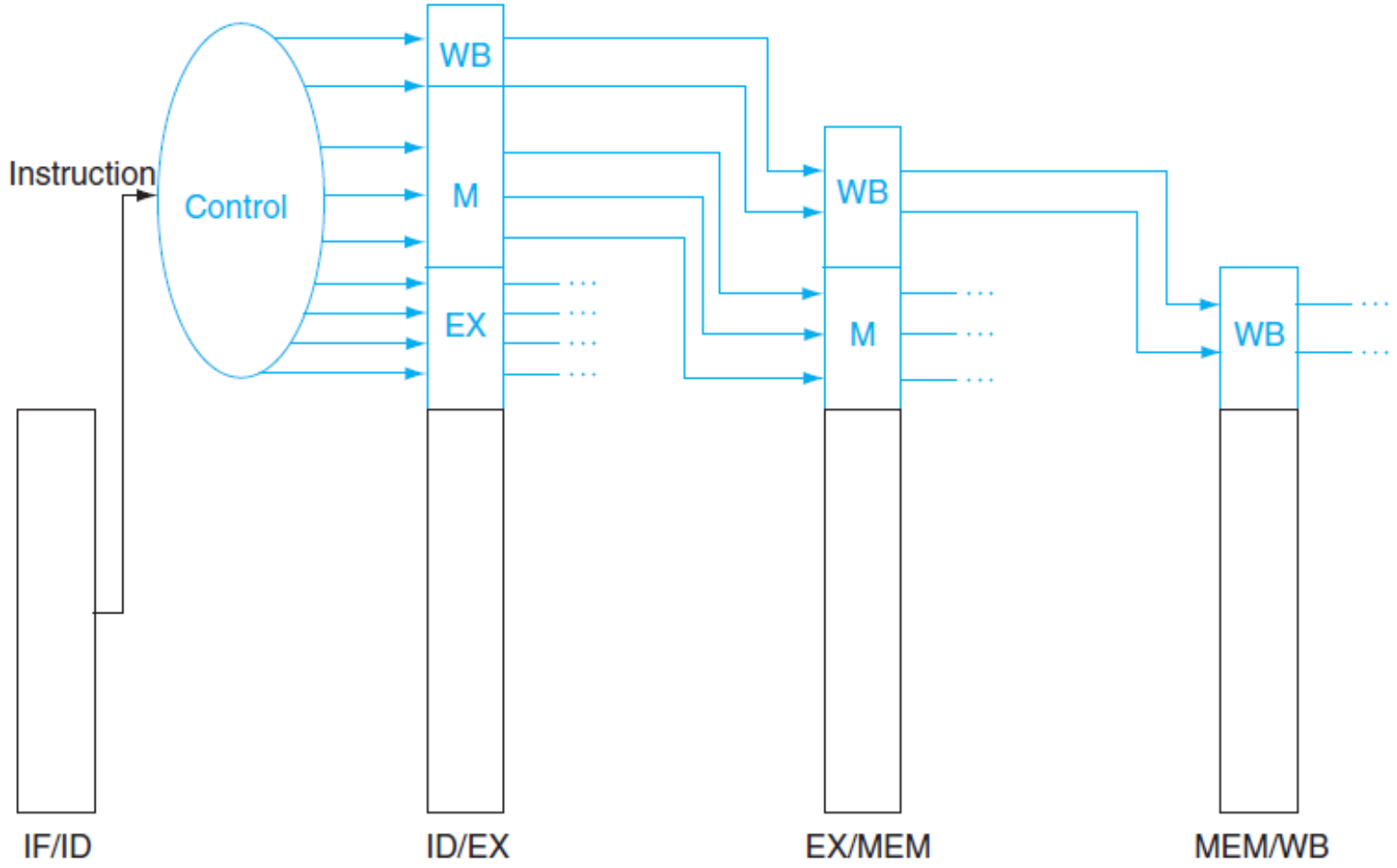

The registers PC, IF/ID, ID/EX, EX/MEM, MEM/WB don't have a control signal. They are written at every clock cycle.

# Control Signals in the Stages

- Instruction Fetch (IF) Stage:
  - The same is done for all instructions (read instruction from memory, PC=PC+4)
  - There are no control signals
- Instruction Decode (ID) Stage:
  - The same thing is done for all instructions (read registers)
  - There are no control signals
- Execution(EX)/Address Computation Stage:
  - RegDst: for results register (bits [20:16] or [15:11] of the instruction)
  - ALUOp: operation of the ALU
  - ALUSrc: read register or 16-bit sign-extended
- Memory (MEM) Access Stage:
  - Branch: 1 if it's a beq instruction
  - MemRead: 1 to read from memory (for load word (lw))
  - MemWrite: 1 to write to memory (for store word (sw))
- Write Back (WB) Stage:
  - MemtoReg: write the ALU result or the data from the memory
  - RegWrite: 1 to write into a register

# Setting the Control Lines

- The instructions travel through the stages from left to right
- The control signals go along with the instructions
  - They are used in the corresponding stage

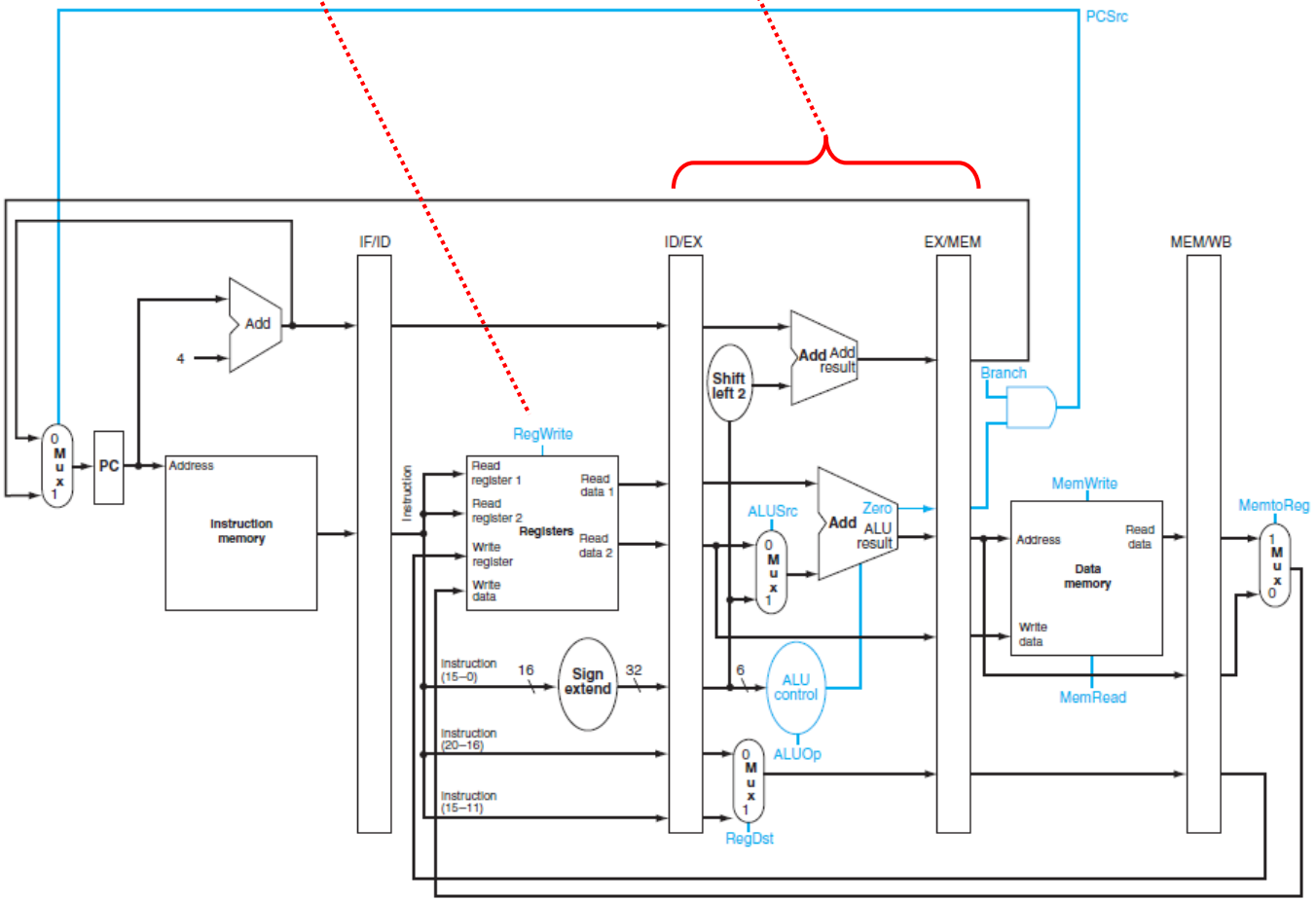


# Control Signals

The control signals start in the EX stage

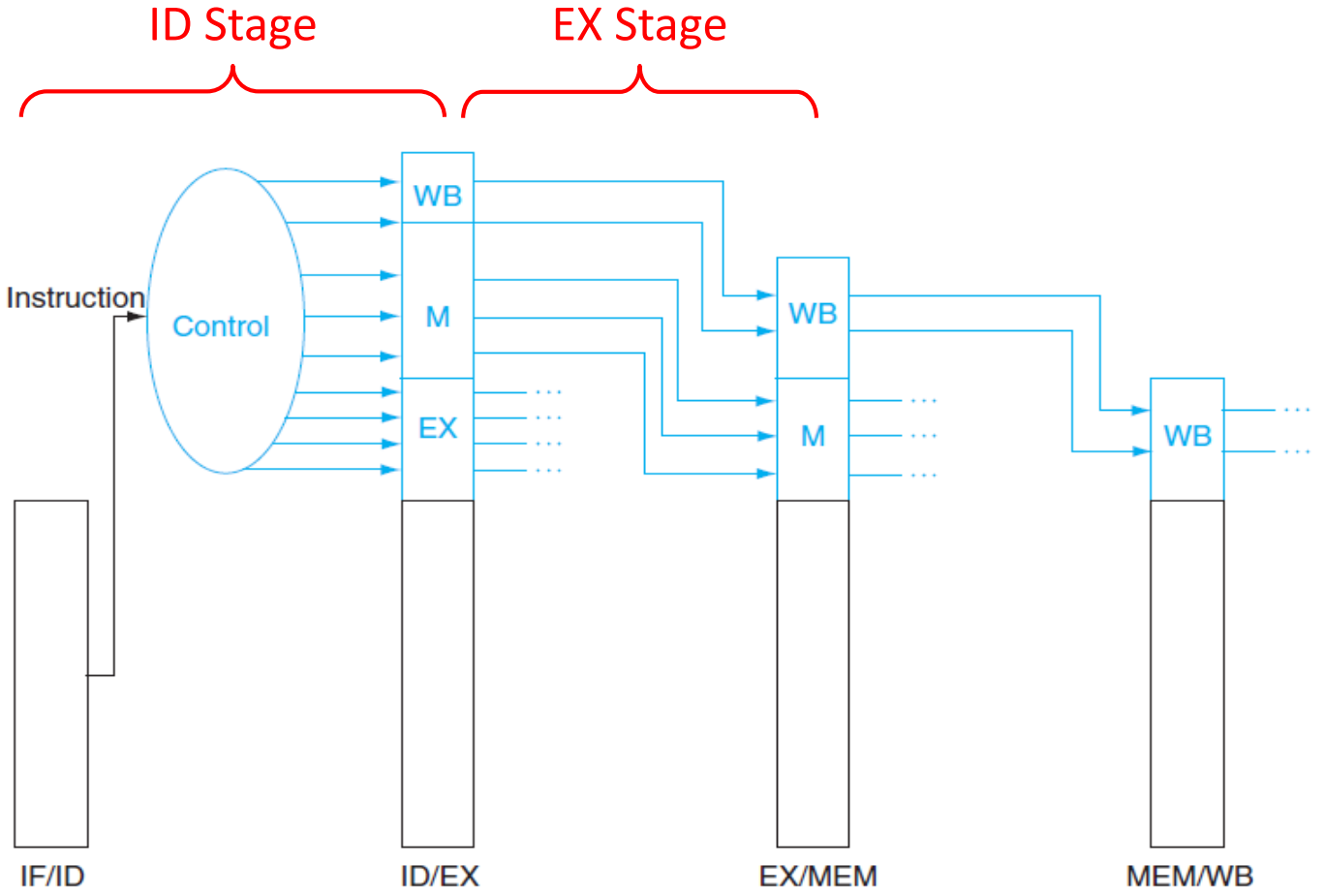
We create the control information during the Instruction Decode (ID) stage

RegWrite is in the WB stage (last stage)

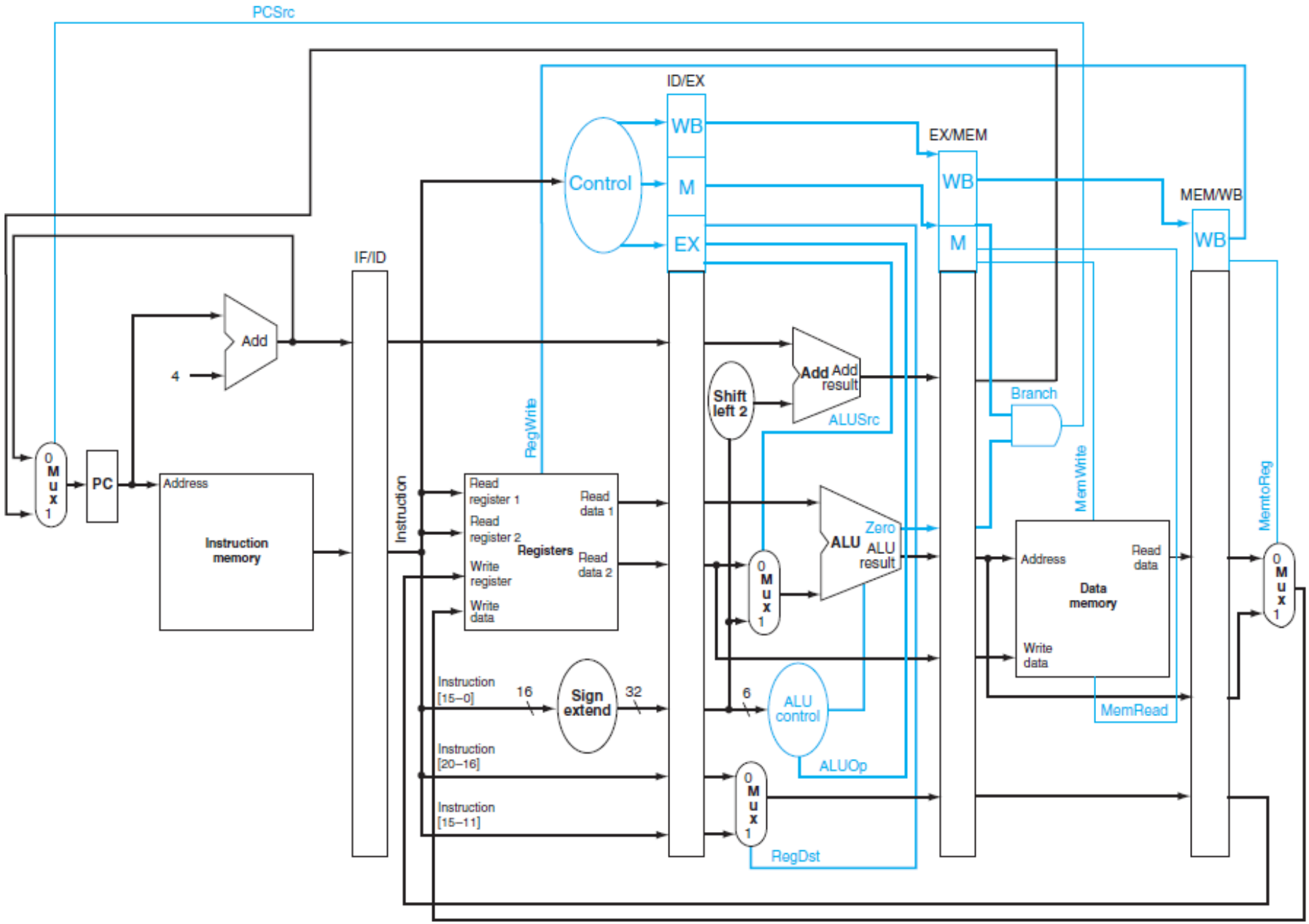


# Control Created During ID … then Propagate

- The use of control starts in the EX stage
- The control signals are set in the ID stage
- They propagate in the stage registers
- They are used in the corresponding stage

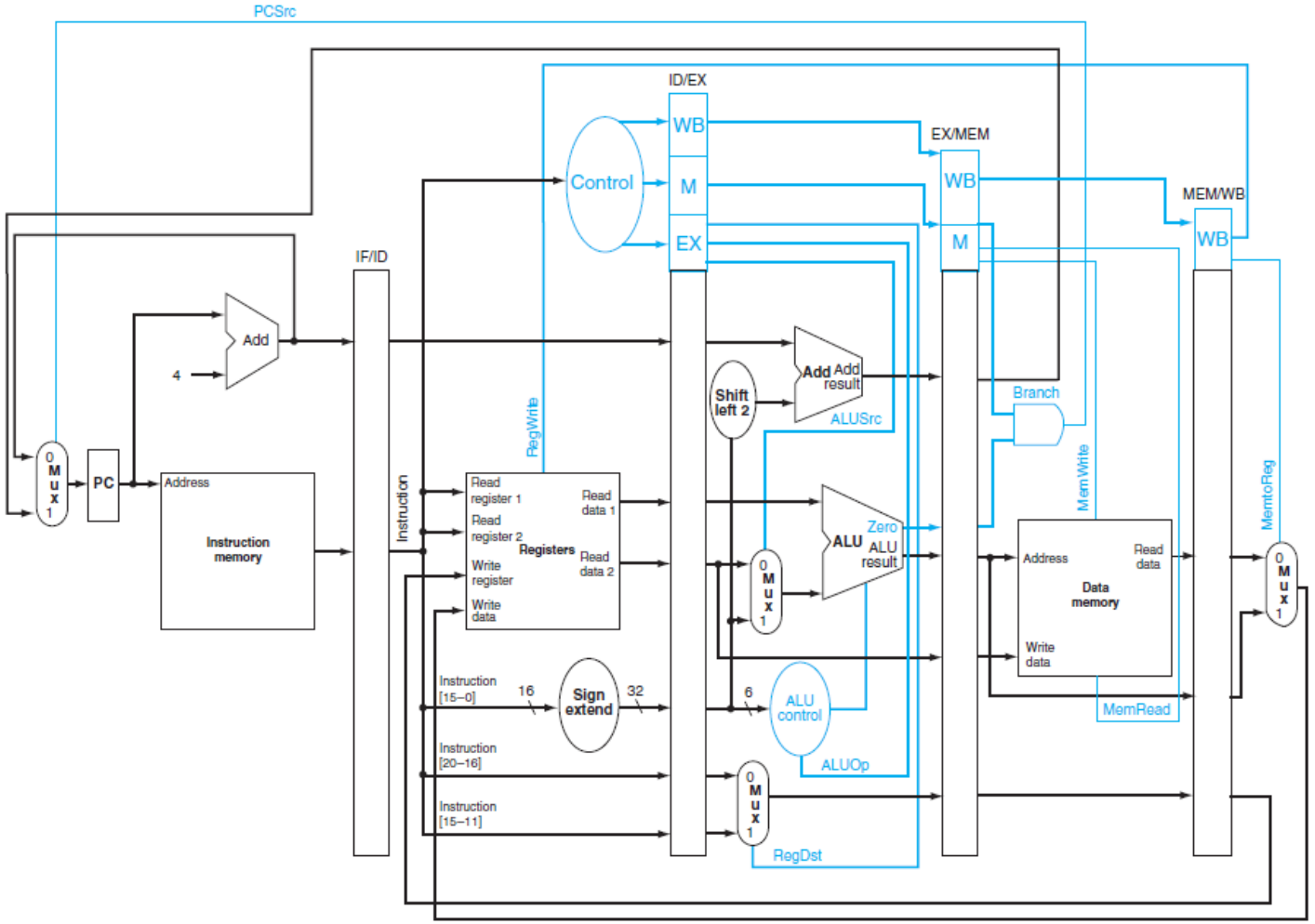


# Datapath with the Control Signals



# Pipelined Datapath So Far

- Questions remaining:
  - How are the data dependences handled in this datapath?
  - Is there a way to insert nops? Who would insert them?
  - Can the data be forwarded between stages?
  - How should the branches be treated?



# Pipelined Datapath So Far

- How are the data dependences handled in this datapath?

*By using nops (separating data dependences by at least two instructions)*

- Is there a way to insert nops? Who would insert them?

*This hardware can't insert a nop; the compiler (or assembly programmer) would do it*

- Can the data be forwarded between stages?

*No. There is no hardware component that forwards*

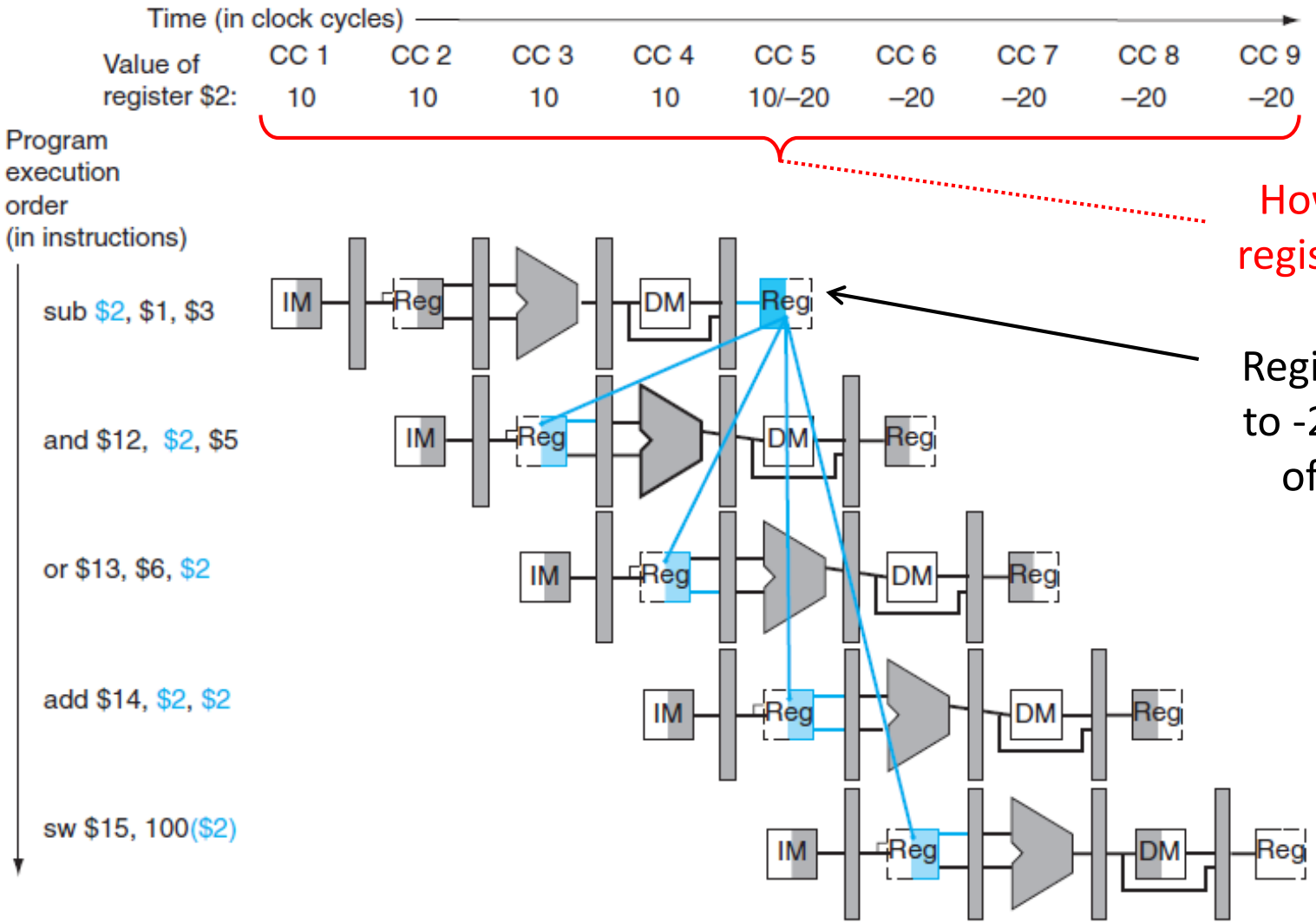
- How should the branches be treated?

*(1) We can do the stall-on-branch strategy by following each branch with 3 nops.*

*(2) Alternatively, we can do the delayed-branch strategy by following the branch with instruction that won't need to be flushed (and that don't have data dependence with the operands of the branch)*

# Execution of Code with Dependencies

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```

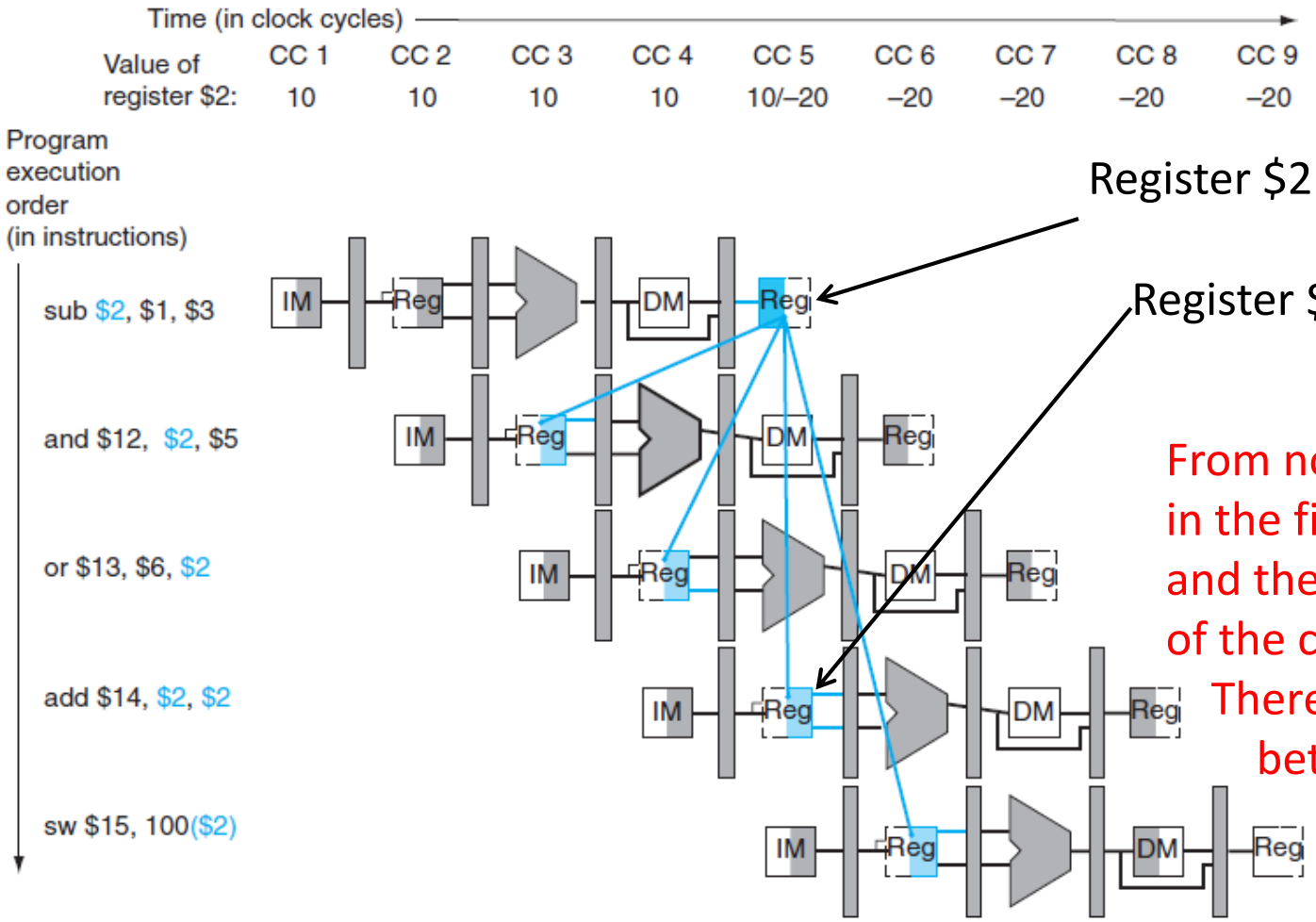



How the value of register $2 changes

Register $2 changes to -20 in the middle of Clock Cycle 5

# Execution of Code with Dependencies

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```



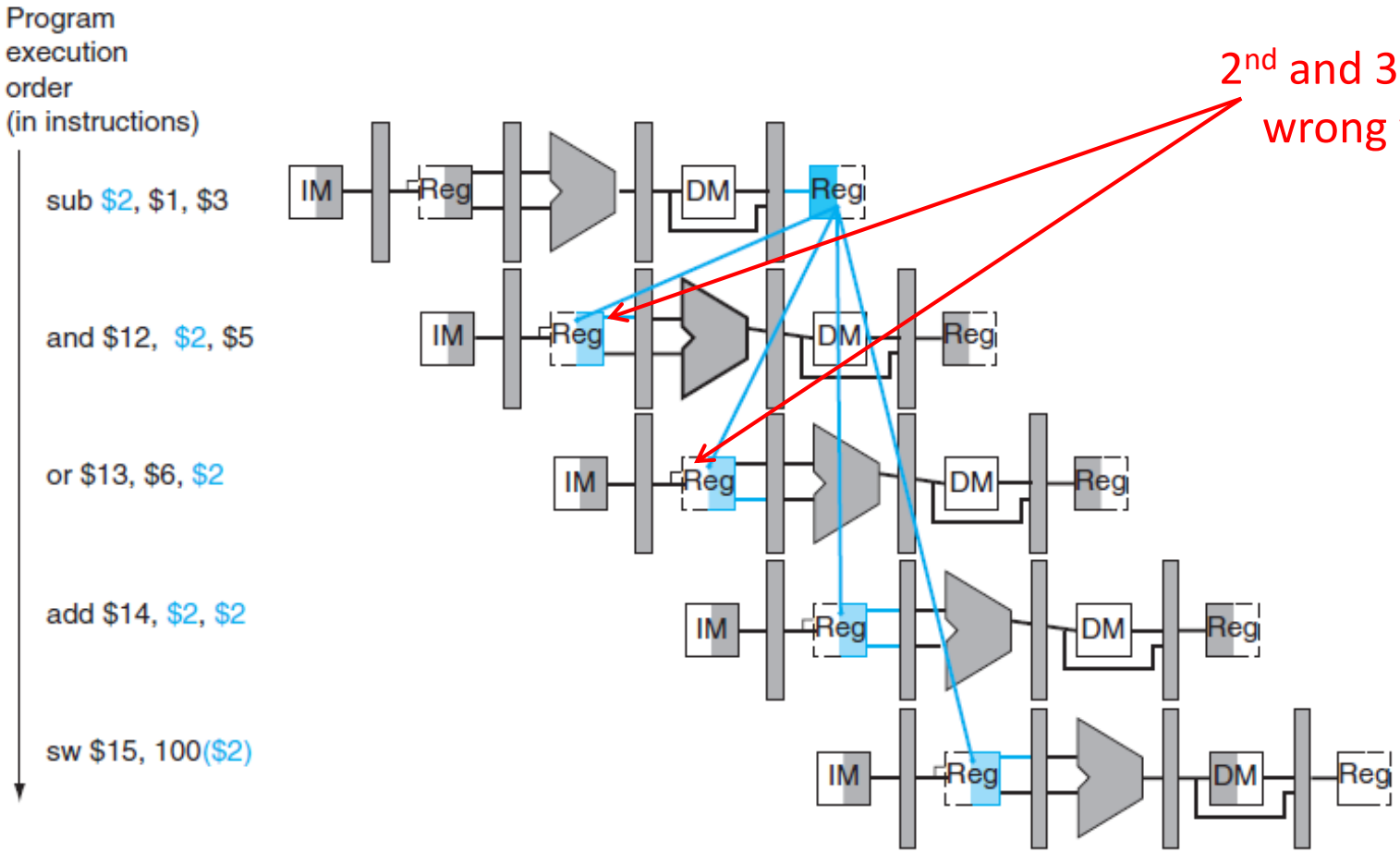

From now on, we make the 'write' in the first half of the clock cycle and the 'read' in the second half of the clock cycle.

There will be no conflict between the 1st and 4th instruction

# Execution of Code with Dependencies

```
sub   $2, $1,$3      # Register $2 written by sub
and   $12,$2,$5      # 1st operand($2) depends on sub
or    $13,$6,$2      # 2nd operand($2) depends on sub
add   $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)    # Base ($2) depends on sub
```

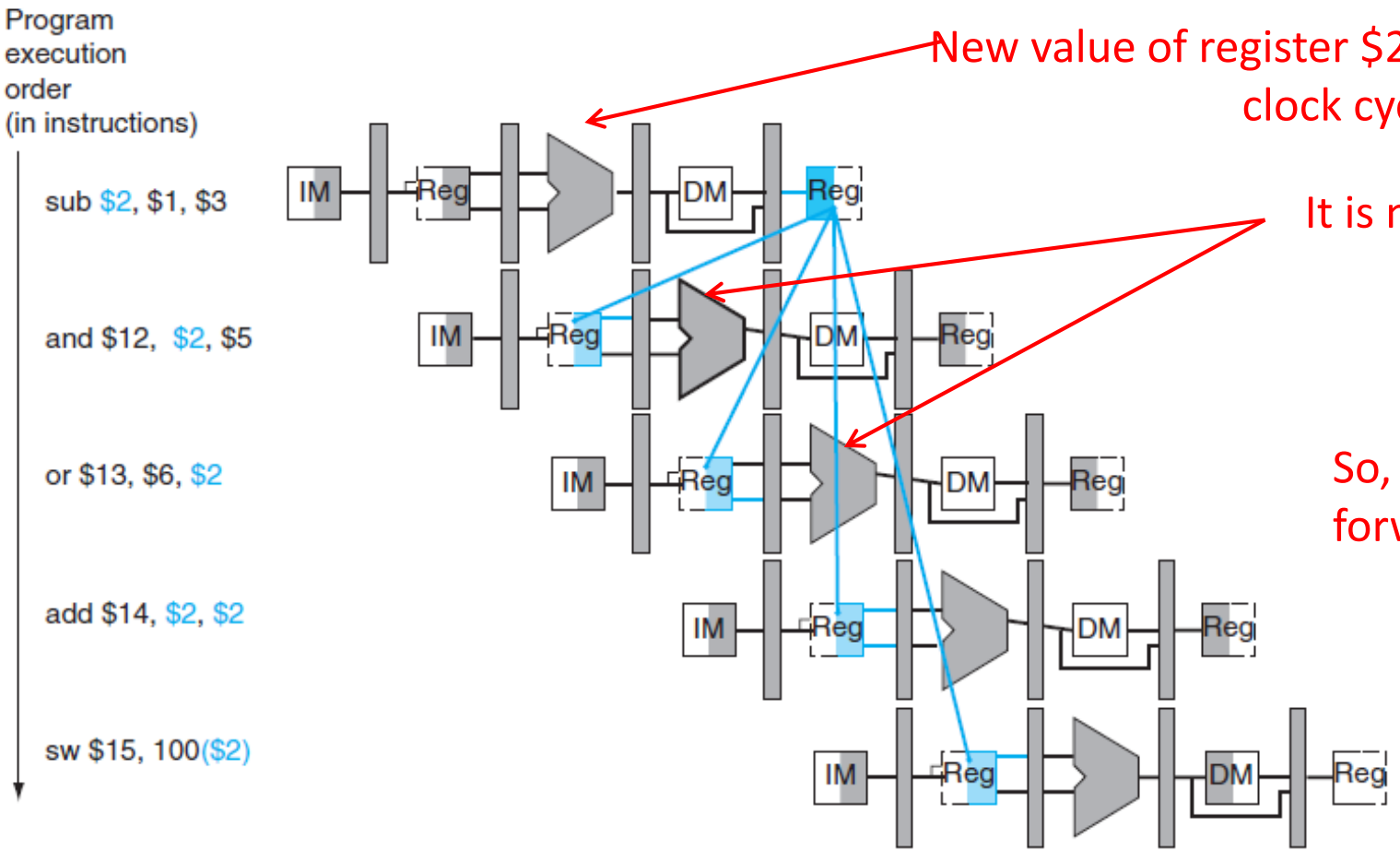
| Time (in clock cycles) | CC 1 | CC 2 | CC 3 | CC 4 | CC 5 | CC 6 | CC 7 | CC 8 | CC 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value of register $2: | 10 | 10 | 10 | 10 | 10/–20 | –20 | –20 | –20 | –20 |



# Execution of Code with Dependencies

```
sub   $2, $1,$3     # Register $2 written by sub
and   $12,$2,$5     # 1st operand($2) depends on sub
or    $13,$6,$2     # 2nd operand($2) depends on sub
add   $14,$2,$2     # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)   # Base ($2) depends on sub
```

| 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 |



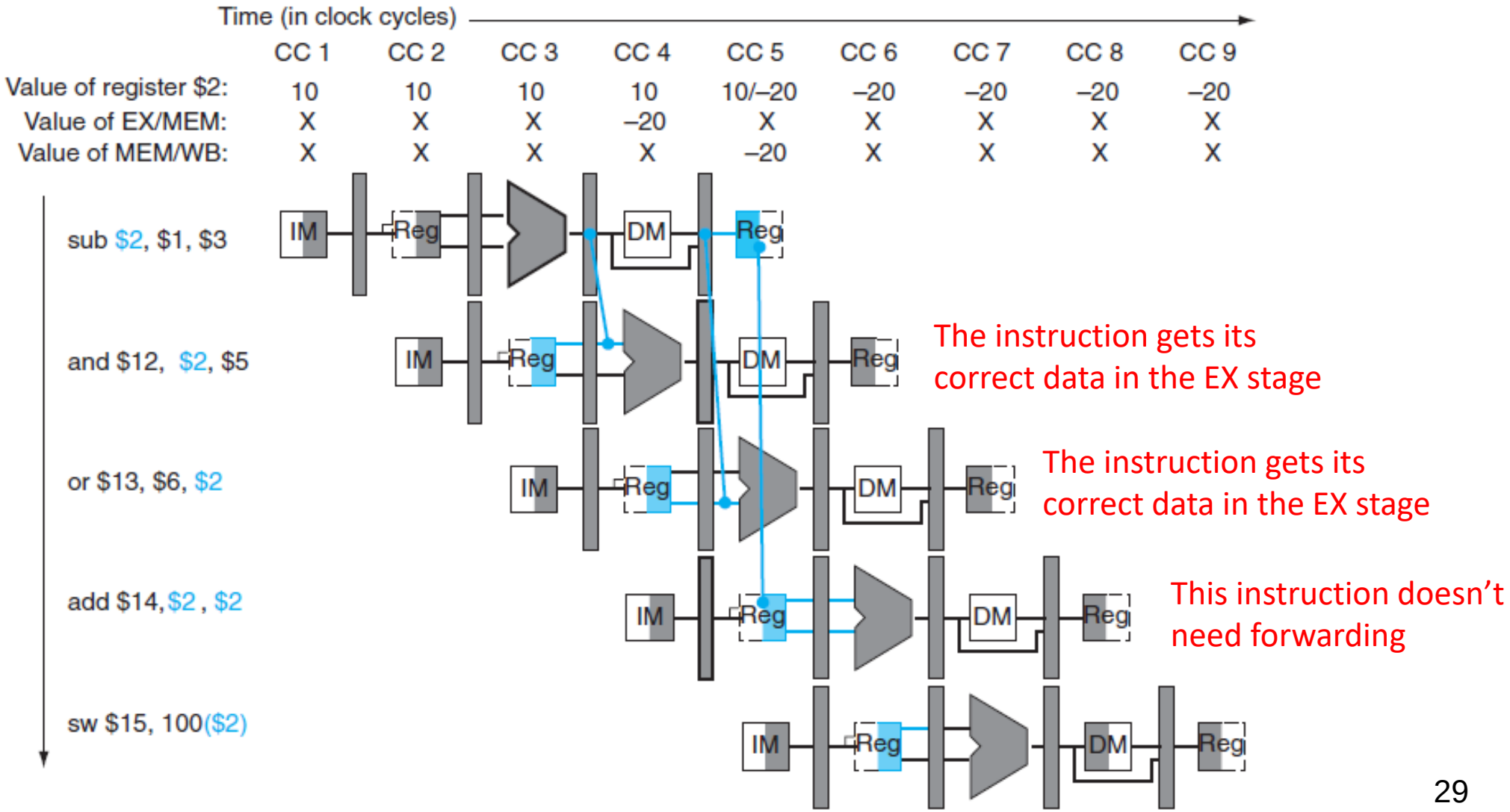

New value of register $2 is produced here in clock cycle 3

It is needed here in clock cycles 4 and 5

So, it is possible to forward the data.

# Data Hazard: Detecting and Forwarding

- We need to detect the hazard and forwarding in the EX stage



# Data Hazard: Detecting and Forwarding

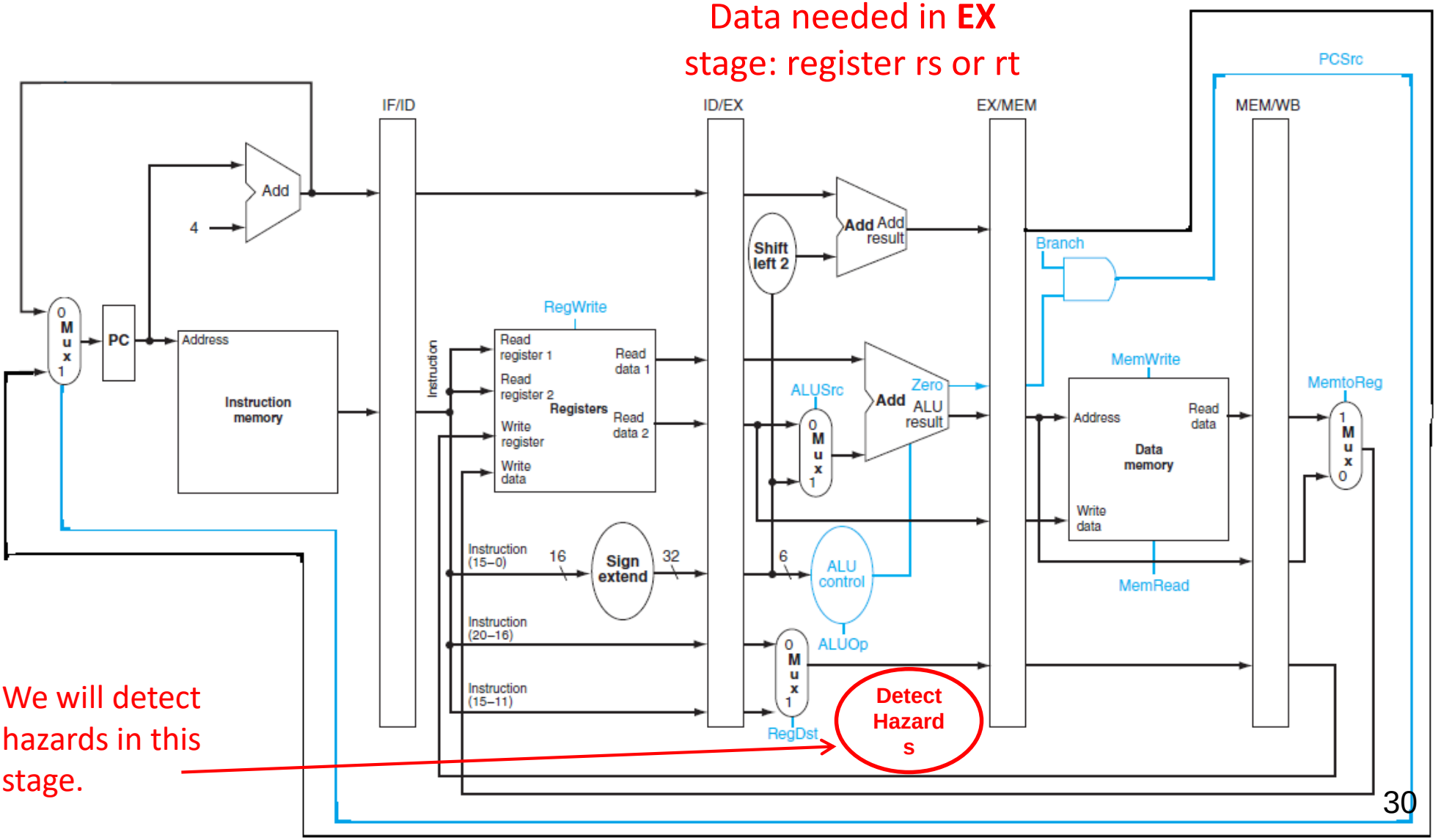
***Forwarding from MEM to EX:***

Hazard: If ID/EX.RegisterRs = EX/MEM.RegisterRd

Hazard: If ID/EX.RegisterRt = EX/MEM.RegisterRd

Updated data is **MEM** stage: register rd

Data needed in **EX** stage: register rs or rt



# Data Hazard: Detecting and Forwarding

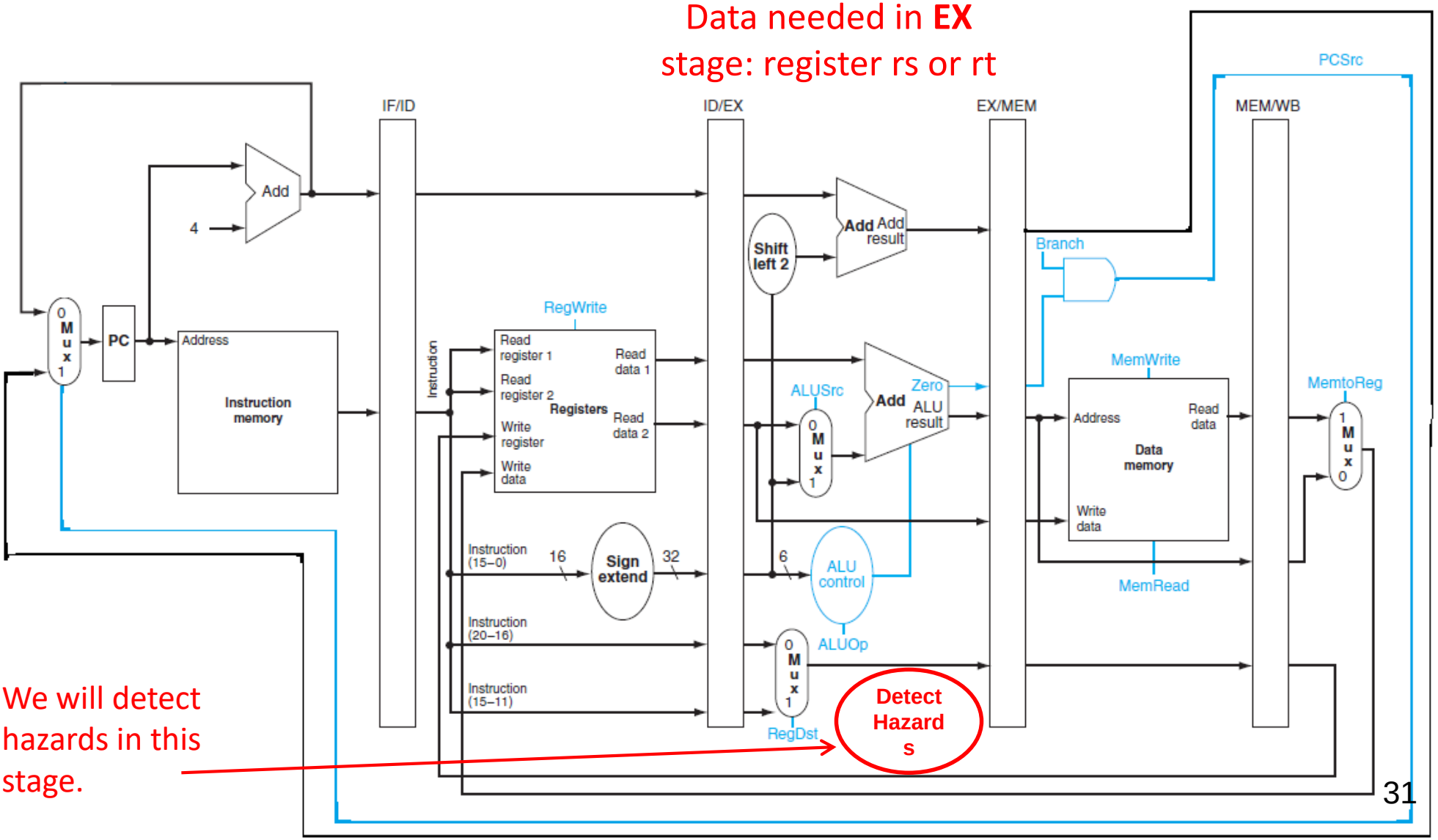
***Forwarding from WB to EX***

Hazard: If ID/EX.RegisterRs = MEM/WB.RegisterRd

Hazard: If ID/EX.RegisterRt = MEM/WB.RegisterRd

Updated data is in **WB** stage: register rd

Data needed in **EX** stage: register rs or rt




We will detect hazards in this stage.

# Data Hazard Detection

- Data hazard detection is done in the EX (ALU) stage
- There is a hazard if:

  ID/EX.RegisterRs = EX/MEM.RegisterRd (Type 1a)

  ID/EX.RegisterRt = EX/MEM.RegisterRd (Type 1b)

  ID/EX.RegisterRs = MEM/WB.RegisterRd (Type 2a)

  ID/EX.RegisterRt = MEM/WB.RegisterRd (Type 2b)

```
sub     $2,   $1, $3   # Register $2 set by sub
and     $12,  $2, $5   # 1st operand($2) set by sub
or      $13,  $6, $2   # 2nd operand($2) set by sub
add     $14,  $2, $2   # 1st($2) & 2nd($2) set by sub
sw      $15,  100($2) # Index($2) set by sub
```

- Between 1st and 2nd instruction: Hazard type 1a
- Between 1st and 3rd instruction: Hazard type 2b

Hint:

If instructions follow each other, it's Type 1. If it's the first (source rs) read register, it's (a).

If instructions are separated by one instruction, it's Type 2. If it's the second read (rt) register, it's (b).

# Data Hazard Detection

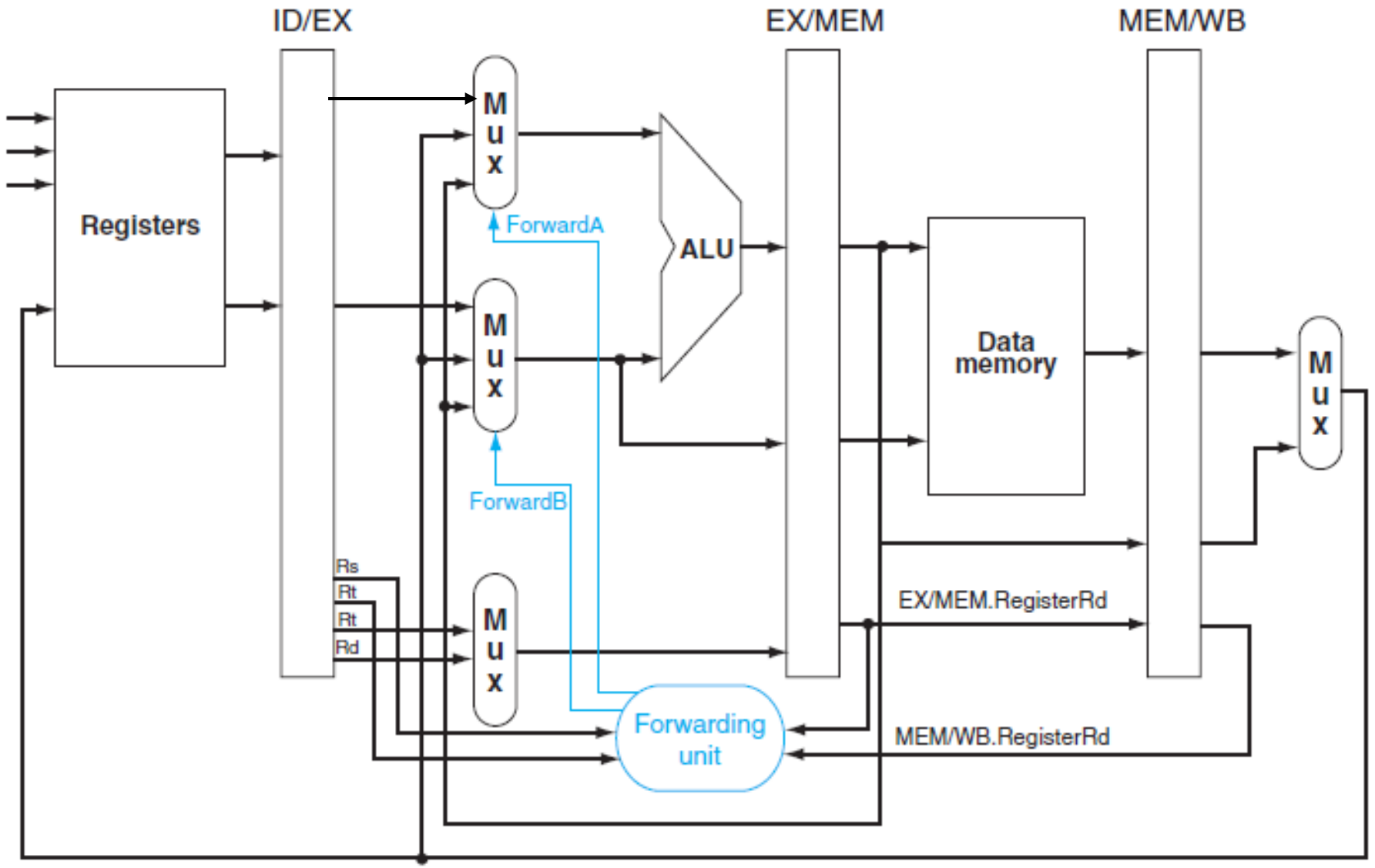
- Data hazard detection is done in the EX (ALU) stage
- There is a hazard if:

  ID/EX.RegisterRs = EX/MEM.RegisterRd (Type 1a)
  ID/EX.RegisterRt = EX/MEM.RegisterRd (Type 1b)
  ID/EX.RegisterRs = MEM/WB.RegisterRd (Type 2a)
  ID/EX.RegisterRt = MEM/WB.RegisterRd (Type 2b)

- The conditions above are not sufficient:
- RegWrite signal should be 1 (the data in the next stage is going to be written to the register), i.e.:
  - EX/MEM.RegWrite == 1
  - EX/WB.RegWrite == 1
- Also, if the destination register is $0, we should not forward since register $0 should always be equal to ‘zero’
  - If an instruction tried to modify register $0, e.g. add $0, $1, $2, we should not forward, i.e.:
  - EX/MEM.RegisterRd ≠ 0
  - MEM/WB.RegisterRd ≠ 0

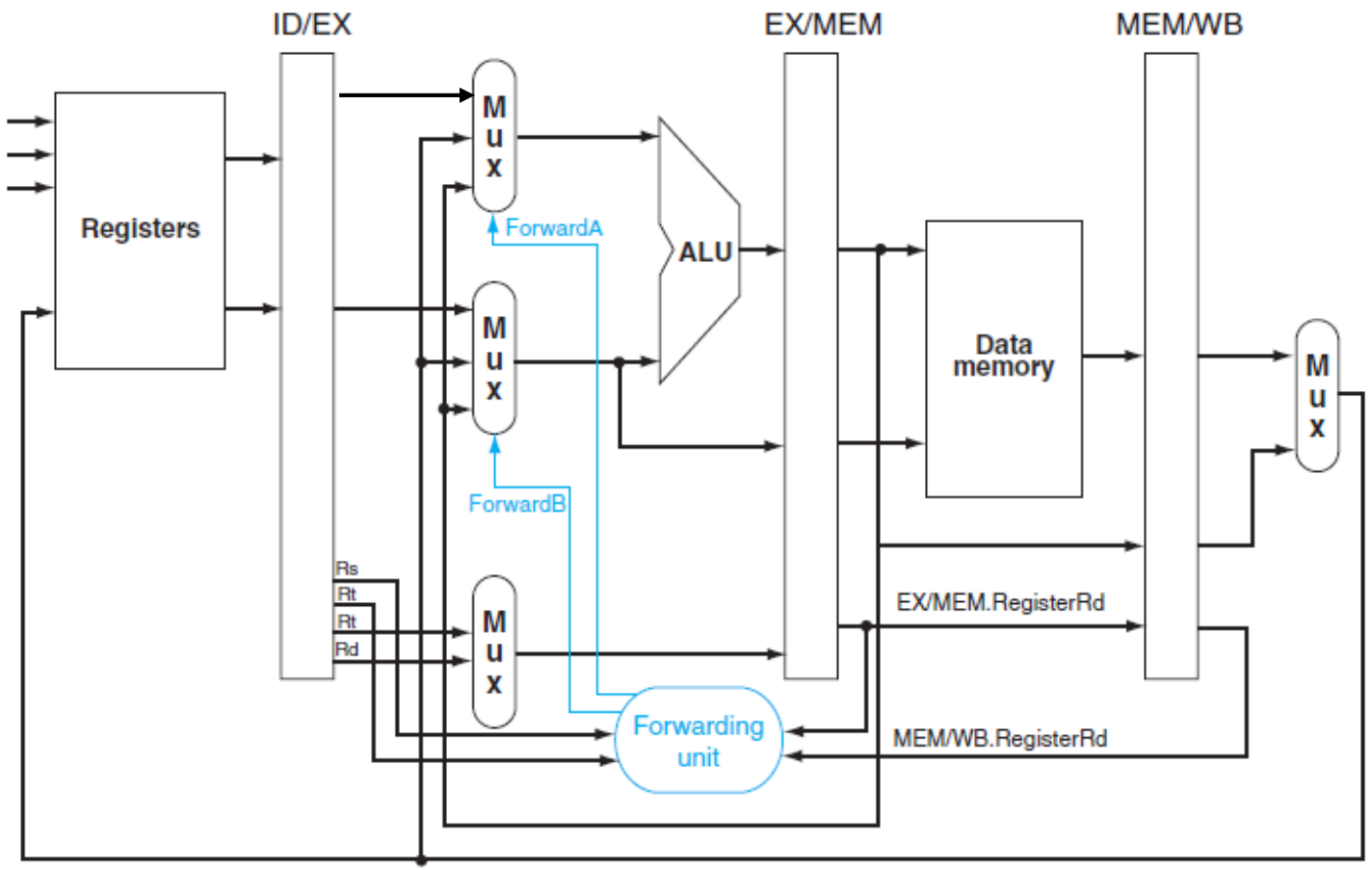
# Forwarding Unit

- The forwarding unit is in the EX stage
- It forwards the data from the MEM stage or from the WB stage



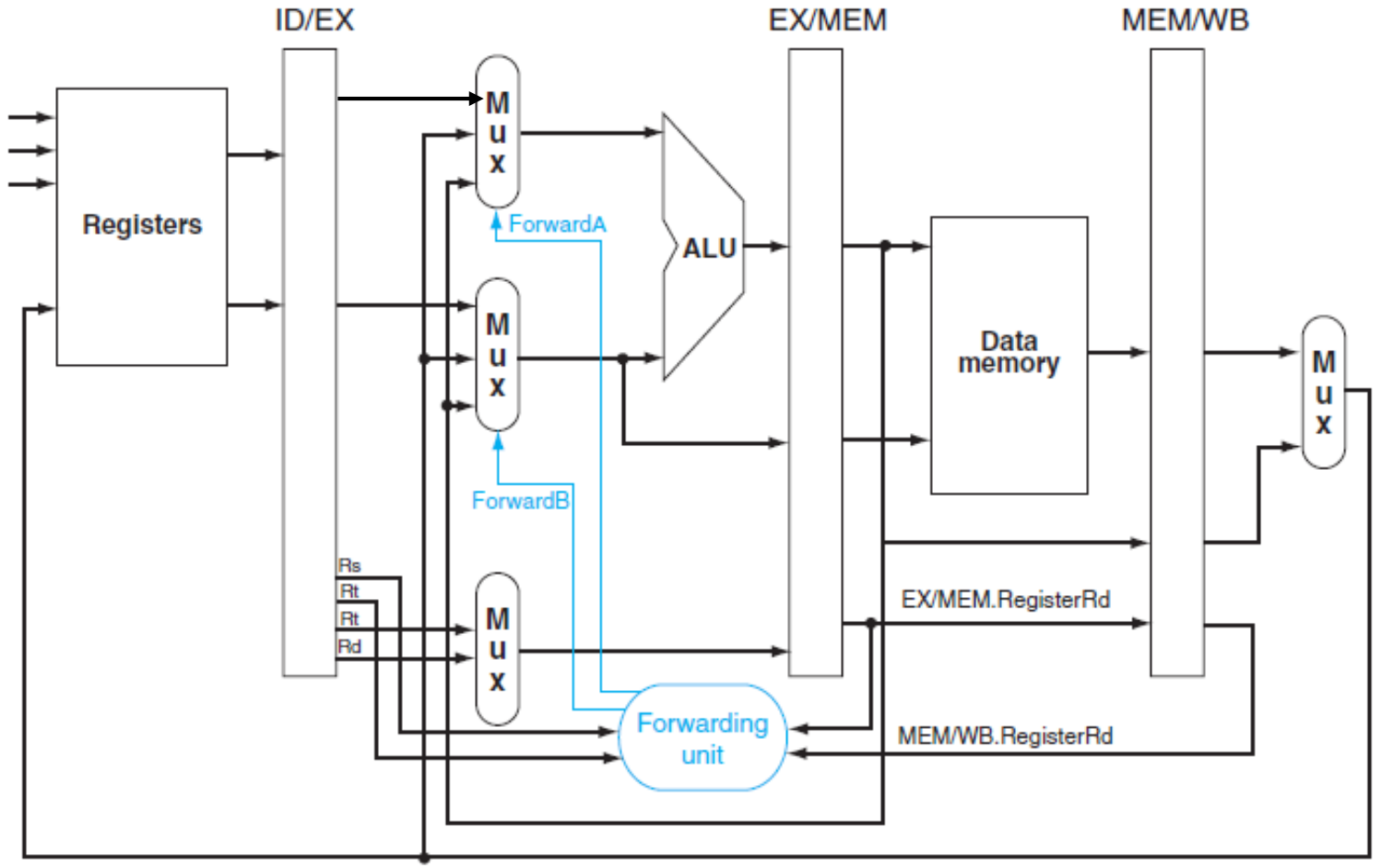
# Forwarding from MEM

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10
```



# Forwarding from WB

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

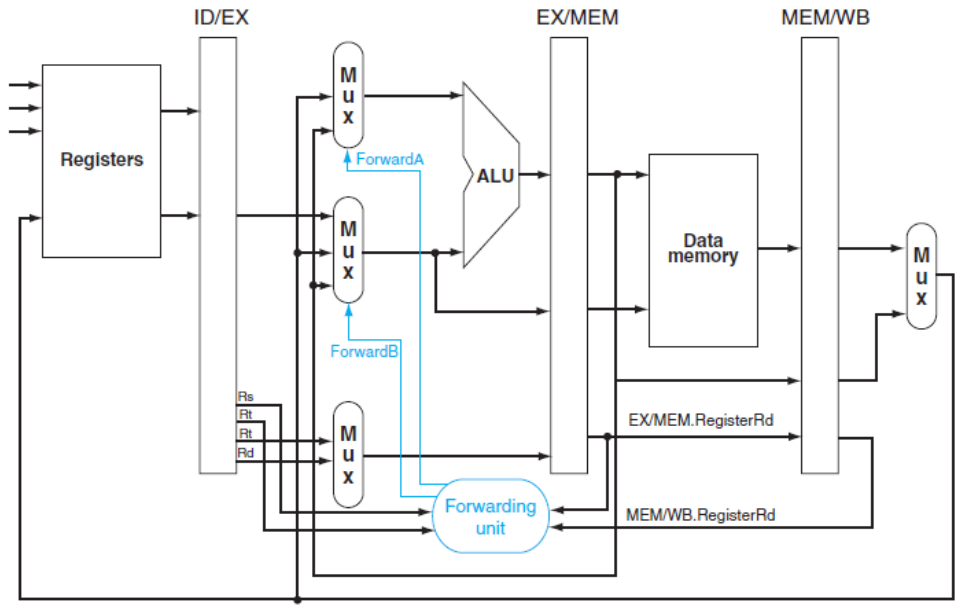


# Forwarding: Multiple Data Hazards

- Hazard with MEM and WB stages simultaneously!:
  - Forward from the MEM stage since it contains the most recent result
- We forward from WB stage, only if there is no hazard with MEM stage

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt)
and (MEM/WB.RegisterRd = ID/EX.RegisterRt)) ForwardB = 01
```

Forward from WB stage if there's no hazard with MEM stage. MEM stage has priority.




Example

```
add $1,$1,$2
add $1,$1,$3
add $1,$1,$4
```

The 3rd add has a data hazard with the 2nd add and the 1st add.

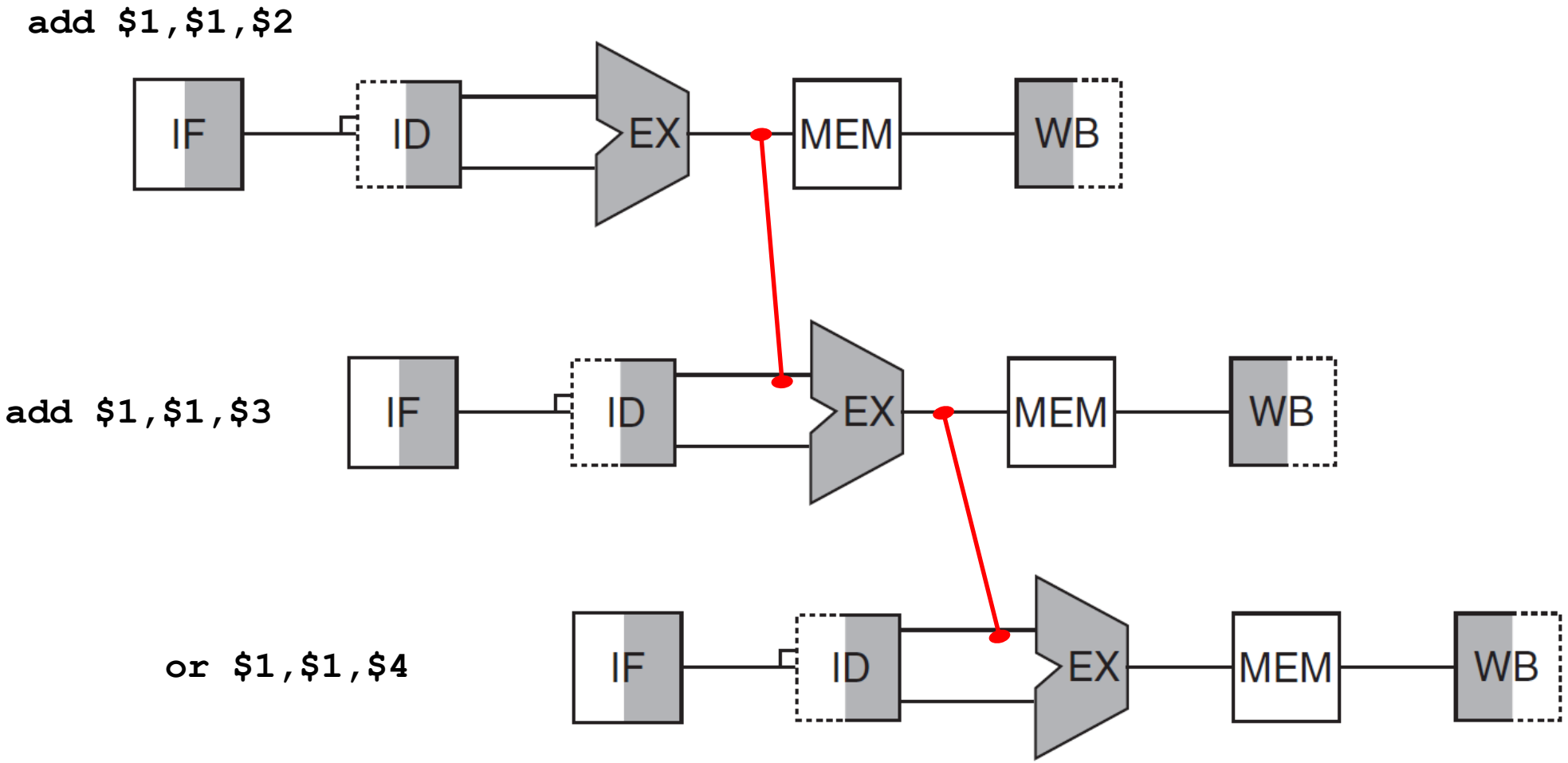
The 3rd add is in EX stage, the 2nd is in MEM stage, the 1st is in WB stage.

Forward from MEM stage.

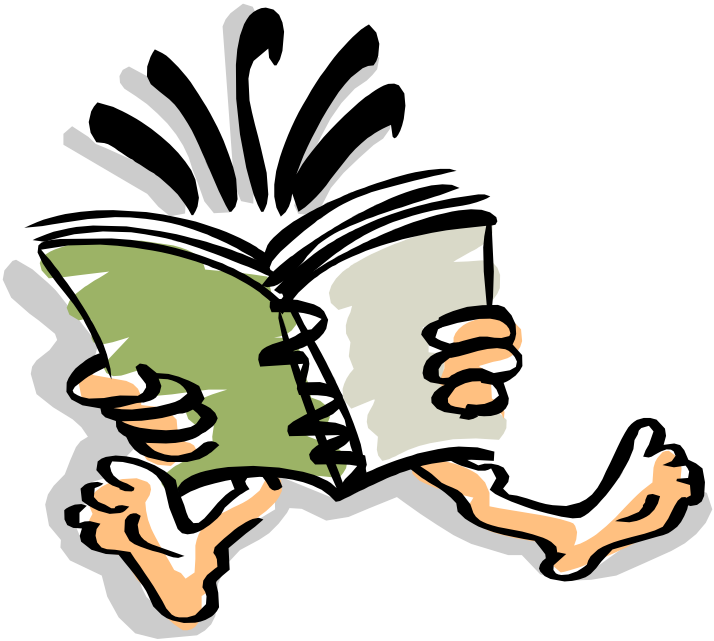
# Data Forwarding

**Code:**

```
add     $1,$1,$2
add     $1,$1,$3
or      $1,$1,$4
```



# Readings



- H&P COD
  - Chapter 4