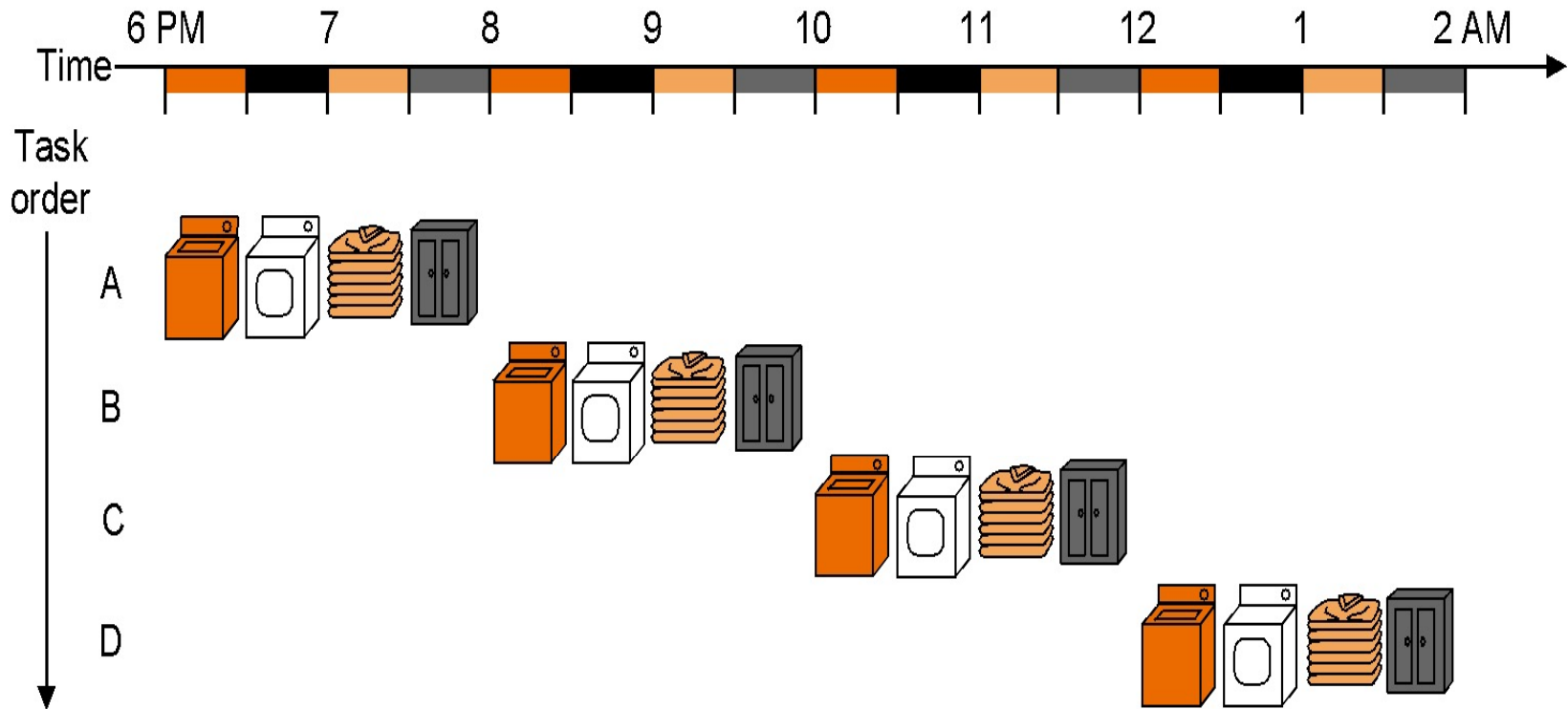


# Pipelining - 1

# Pipelining: Definition

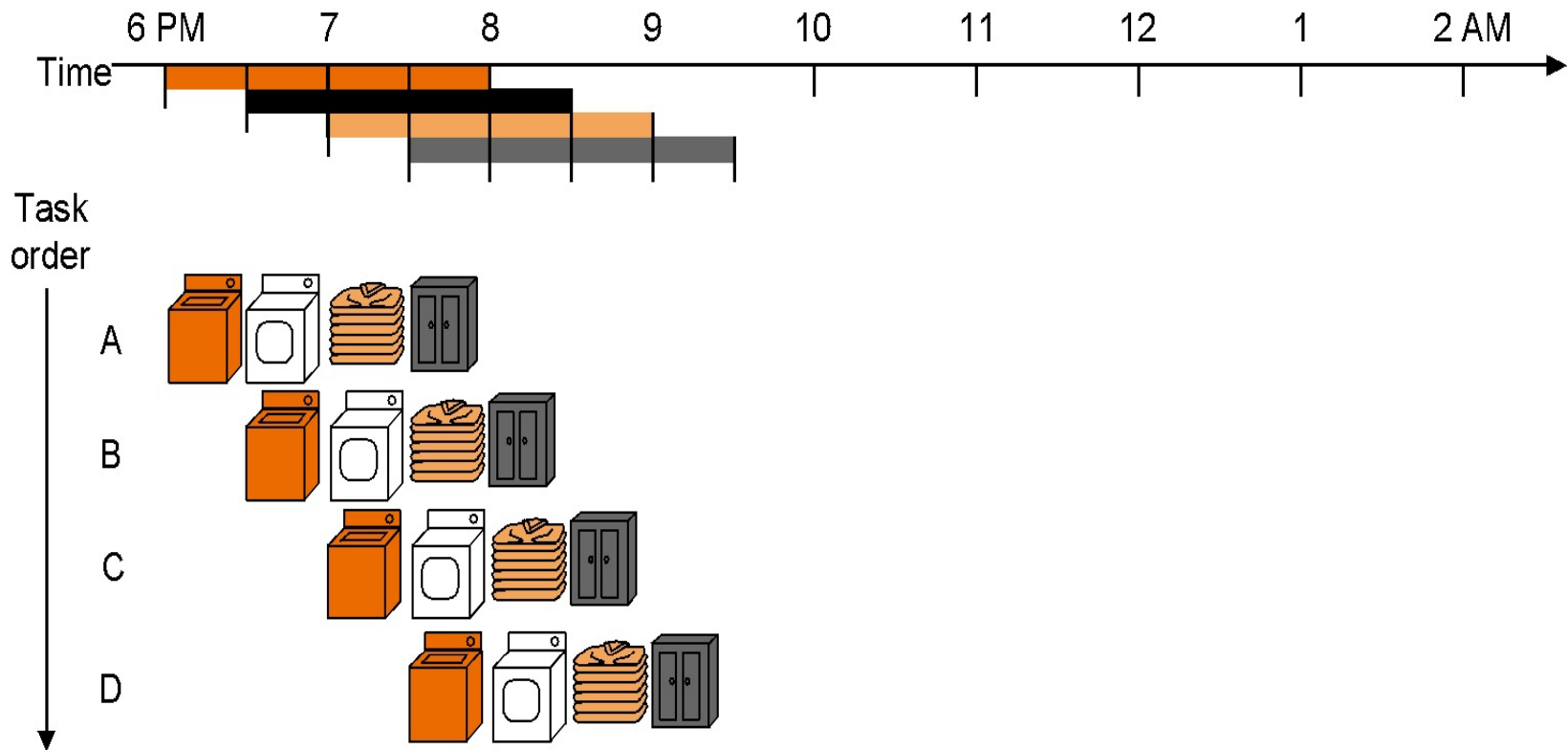
- An implementation technique
  - instructions are overlapped in execution.
- A key technique for improving performance in many, if not all, contemporary processors.
- In multiple cycle implementation,
  - steps of instructions are executed in different stages.
  - functional units for different purposes
- In pipelined datapath
  - Some units used more efficiently
  - Some units must be replicated

# Laundry Analogy

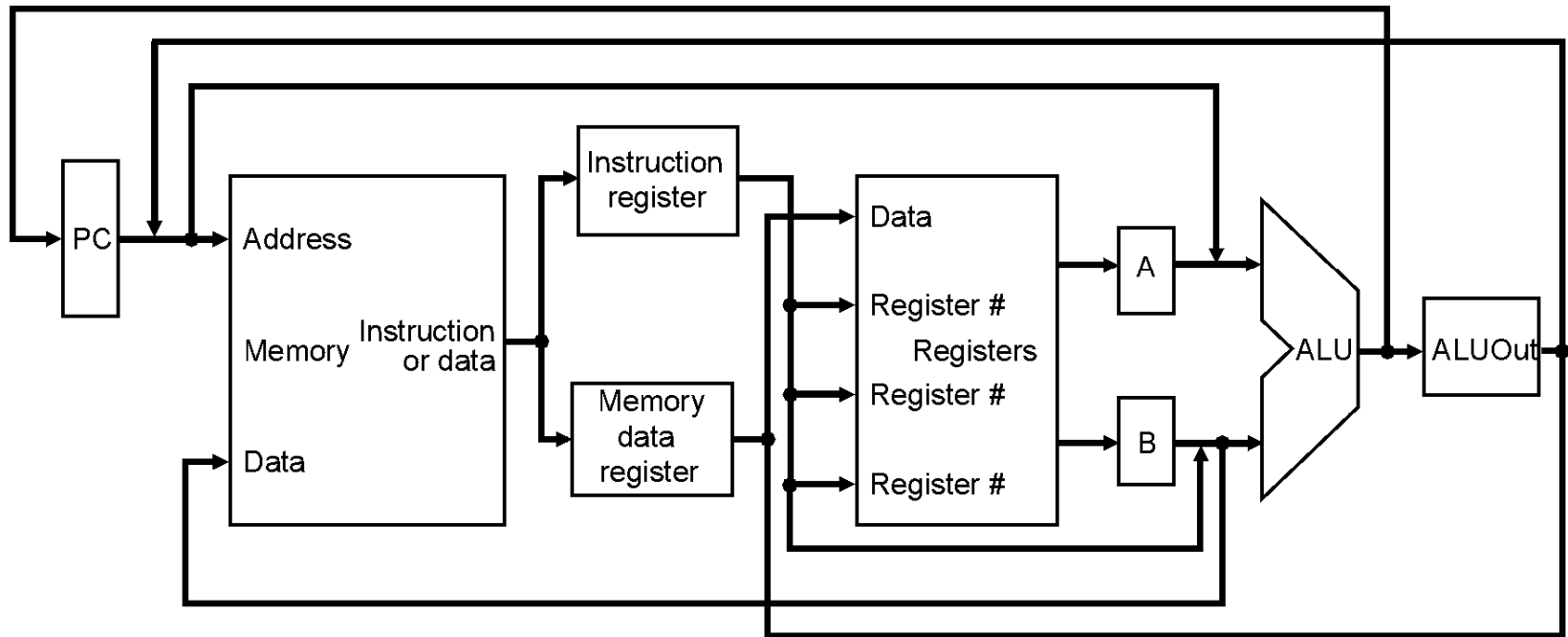


- Four different stages
- Identical time (30 min)

# Pipelined Laundry



# Multicycle Datapath



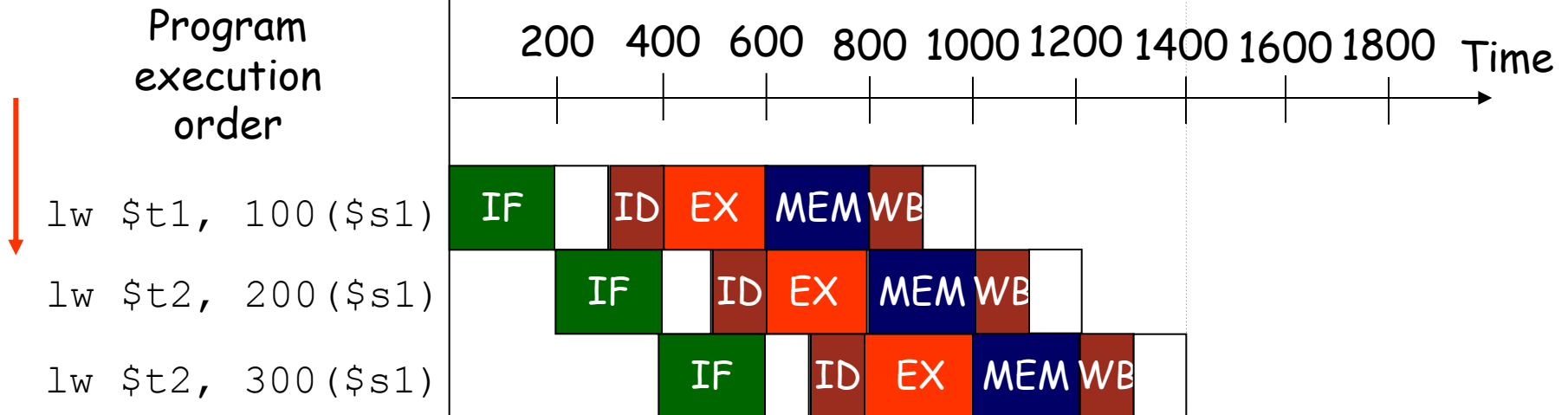
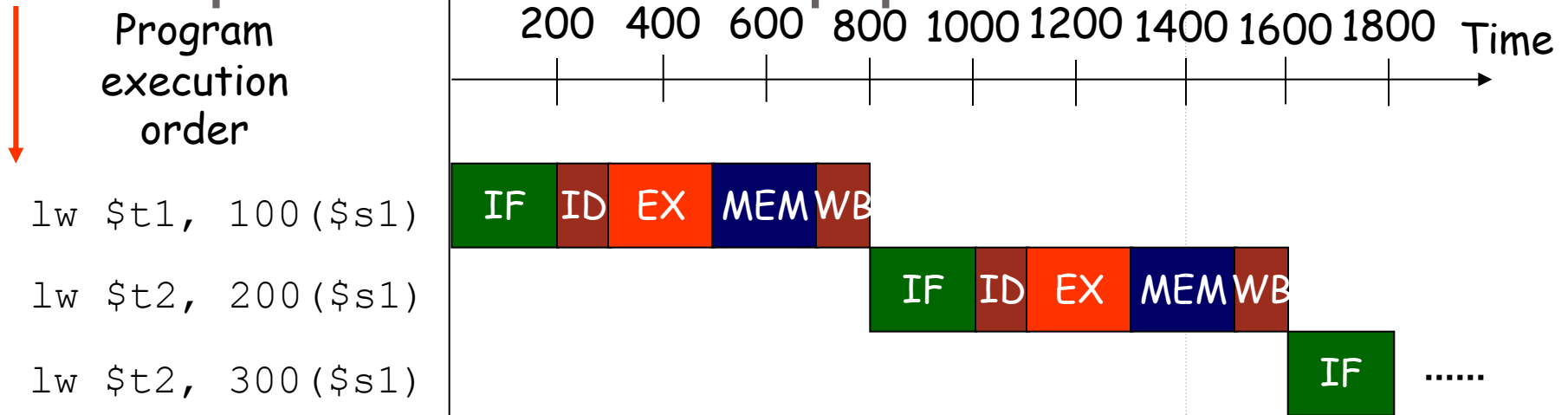
# Pipelined Datapath

- MIPS instructions classically take five steps:
  1. IF - Fetch instruction from memory
  2. ID - Read registers while decoding the instruction
  3. EX - Execute the operation or calculate an address
  4. MEM - Access an operand in memory
  5. WB - Write result into a register

# Times of Instruction Steps

Instruction class	Inst. Fetch	Register Read	ALU op	Data access	Register write	Total time
lw	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
sw	200 ps	100 ps	200 ps	200 ps		700 ps
R-format	200 ps	100 ps	200 ps		100 ps	600 ps
Branch	200 ps	100 ps	200 ps			500 ps

# Pipelined vs. Nonpipelined





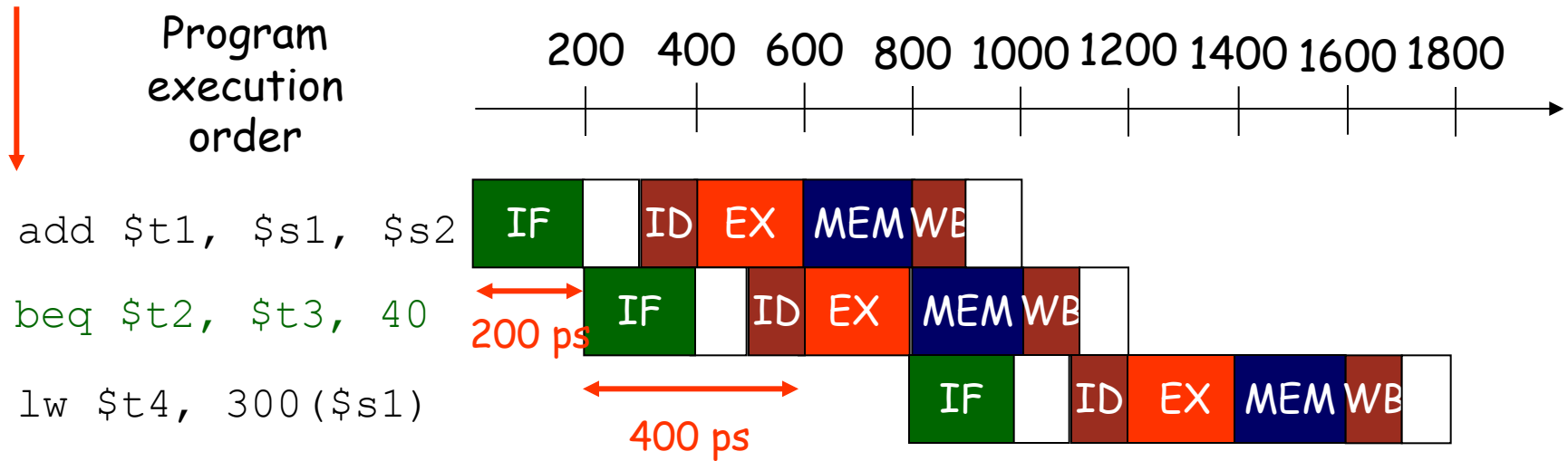
# Aspects of Pipelining

- Latency is the same (usually a little worse)
- Instruction throughput increases.
- What is the ideal speedup due to pipelining?
  - In practice, it is hard to achieve the theoretical speedup because of 1) imperfectly balanced instruction steps and 2) hazards.
- Features of MIPS that make the pipelining easy:
  1. Instructions are of the same length.
  2. Few instruction formats
  3. Load/store architecture
  4. Operands are aligned in memory

# Pipelining Problems

- Structural hazards:
  - Hardware cannot support the combination of instructions in the same clock cycle.
  - one memory for instruction and data.
- Control hazards:
  - branch instructions change the flow of instructions
- Data hazards:
  - an instruction depends on the result of a previous instruction
  - The result may not be available when needed by the current instruction

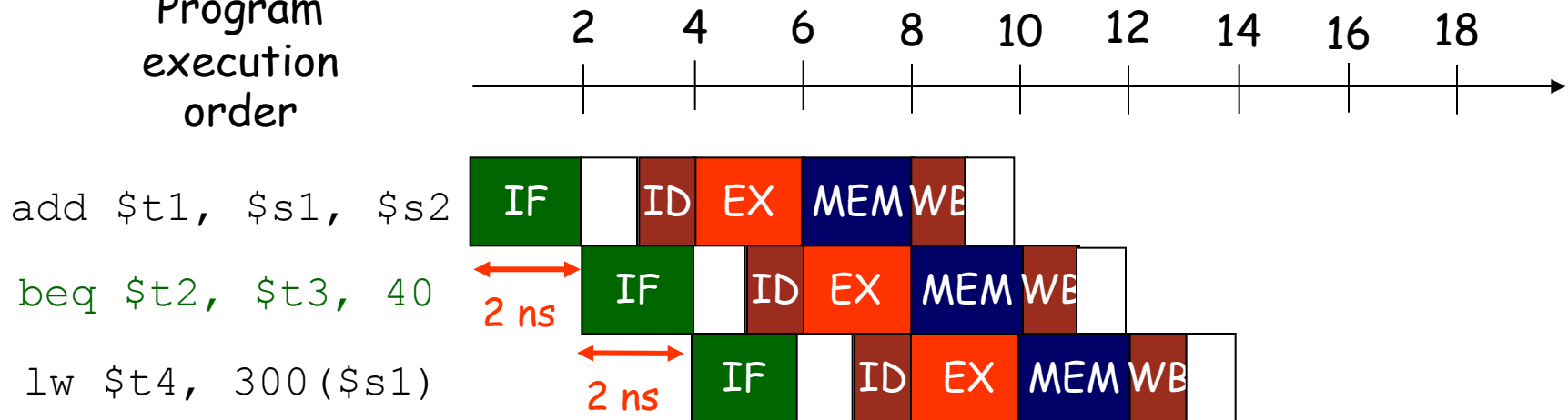
# Pipeline Stalls on Control Hazards



- Assume that we can resolve branch in the second step
- Next instruction cannot start until the end of second cycle.
- The pipeline stops issuing new instruction, or it stalls.

# Predict the Outcome

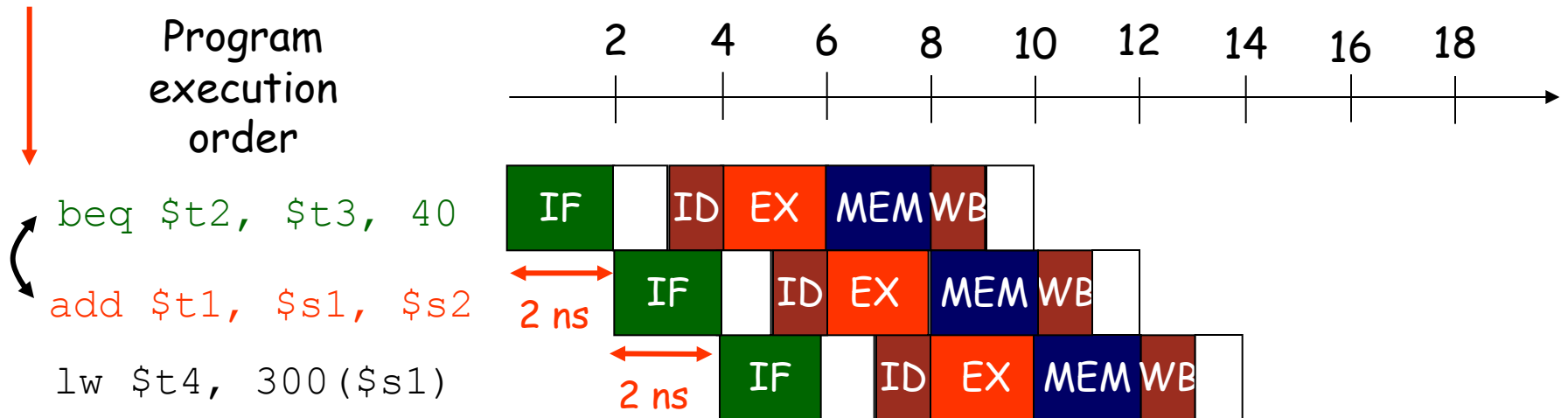
Program  
execution  
order



- One simple approach is to predict that branches will always fail.
- When otherwise occurs (branch is indeed taken), then started instruction is discarded (flushed out) and the instruction at the branch address will start executing.

# Delayed Branches

- A solution actually used by the MIPS architecture.
- Delayed branches always execute next sequential instruction
  - MIPS software places an instruction immediately after the delayed branch instruction that is not affected by the branch
- Compilers fill about 50% of the branch delay slots with useful instructions



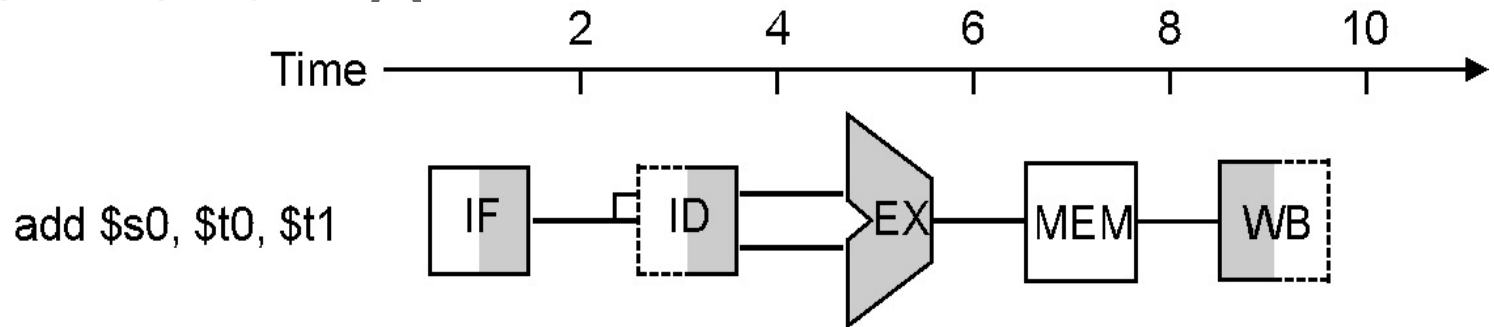
# Data Hazards

- An instruction depends on the results of a previous instruction still in the pipeline.
- Example:  
add \$s0, \$t0, \$t1  
sub \$t2, \$s0, \$t3
- This is also known as data dependency.
- Normally, these types of data hazards could severely stall the pipeline
  - We have to add three bubbles in the pipeline.

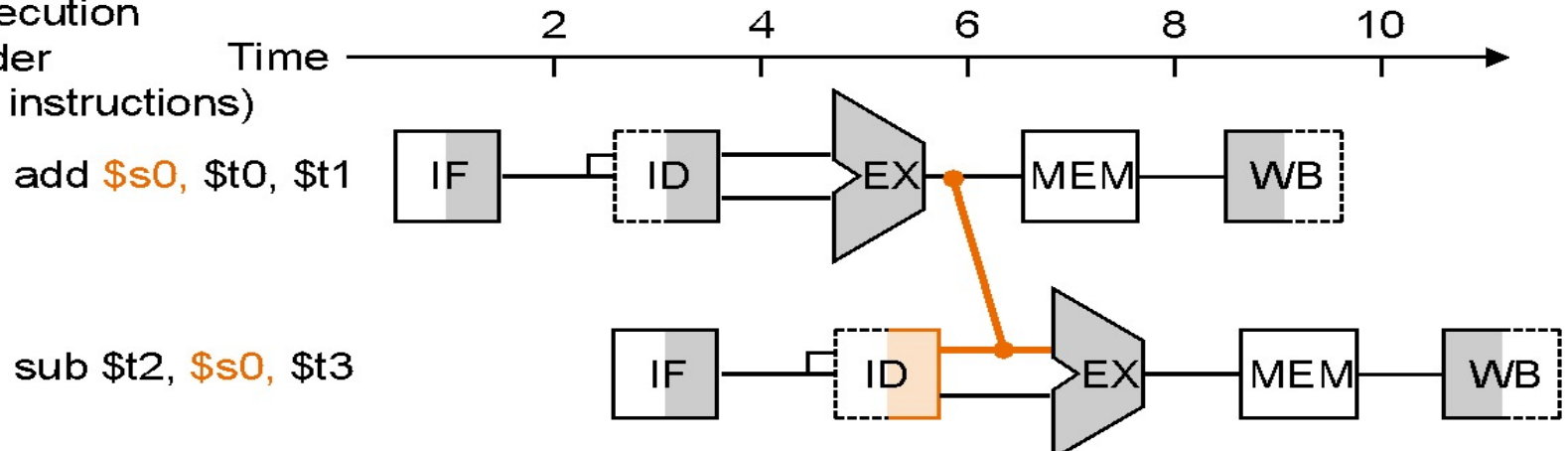
# Forwarding

- First solution: *compiler optimizations* rearrange the instruction sequence.
- Second solution: *forwarding* (or *bypassing*).
  - ALU creates the result much sooner than the result is actually written back to a register.
  - As soon as the ALU creates it, the result immediately can be forwarded to the next instruction.
  - Direct use of ALU results.

# Forwarding



Program  
execution  
order  
(in instructions)

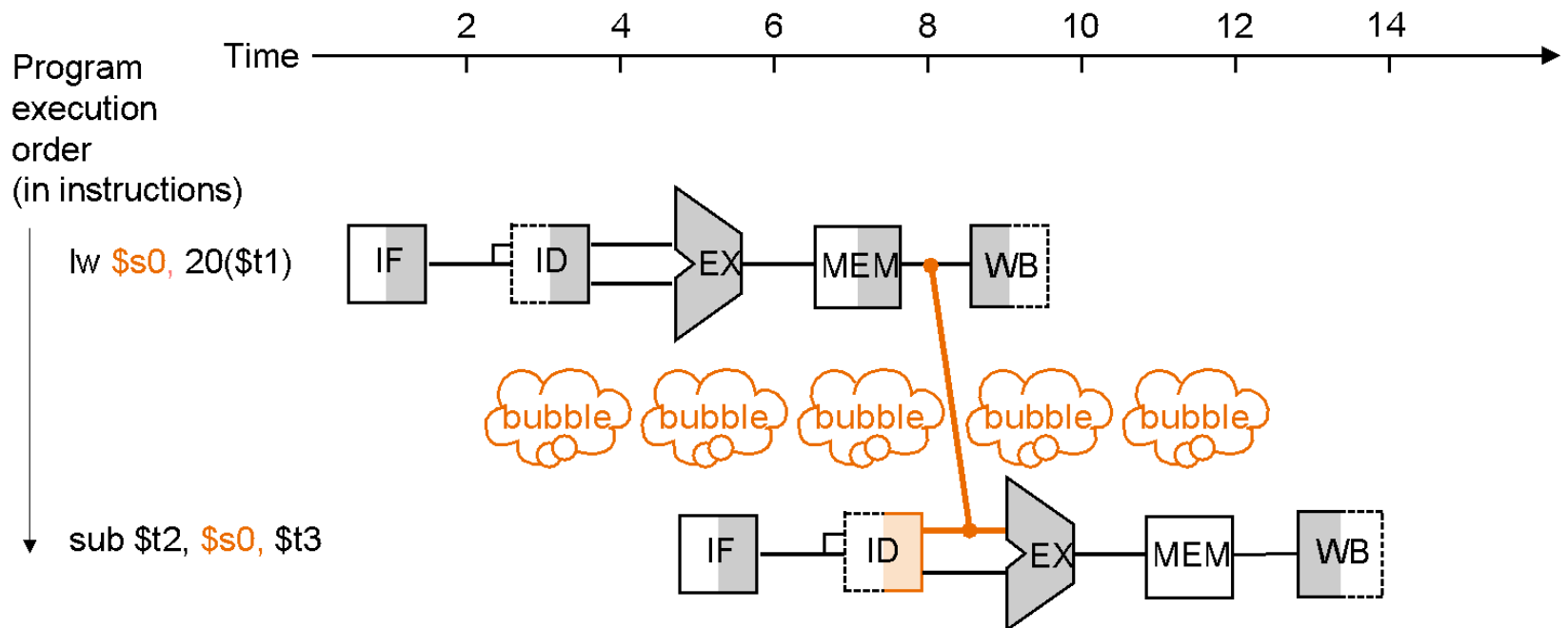


- Direct use of ALU result



# Forwarding: Examples


- Forwarding when an R-format instruction following a load instruction.



# Reordering Code to Avoid Pipeline Stalls

**Code that causes a pipeline stall due to data hazard.**

```
lw    $t0, 0($t1)
lw    $t2, 4($t1)
addi  $s0, $t2, 0x5
addi  $s1, $t0, 0xA
```



**Reordered code. No data hazard.**

```
lw    $t0, 0($t1)
lw    $t2, 4($t1)
addi  $s1, $t0, 0xA
addi  $s0, $t2, 0x5
```

# Recall: Single Cycle Databath

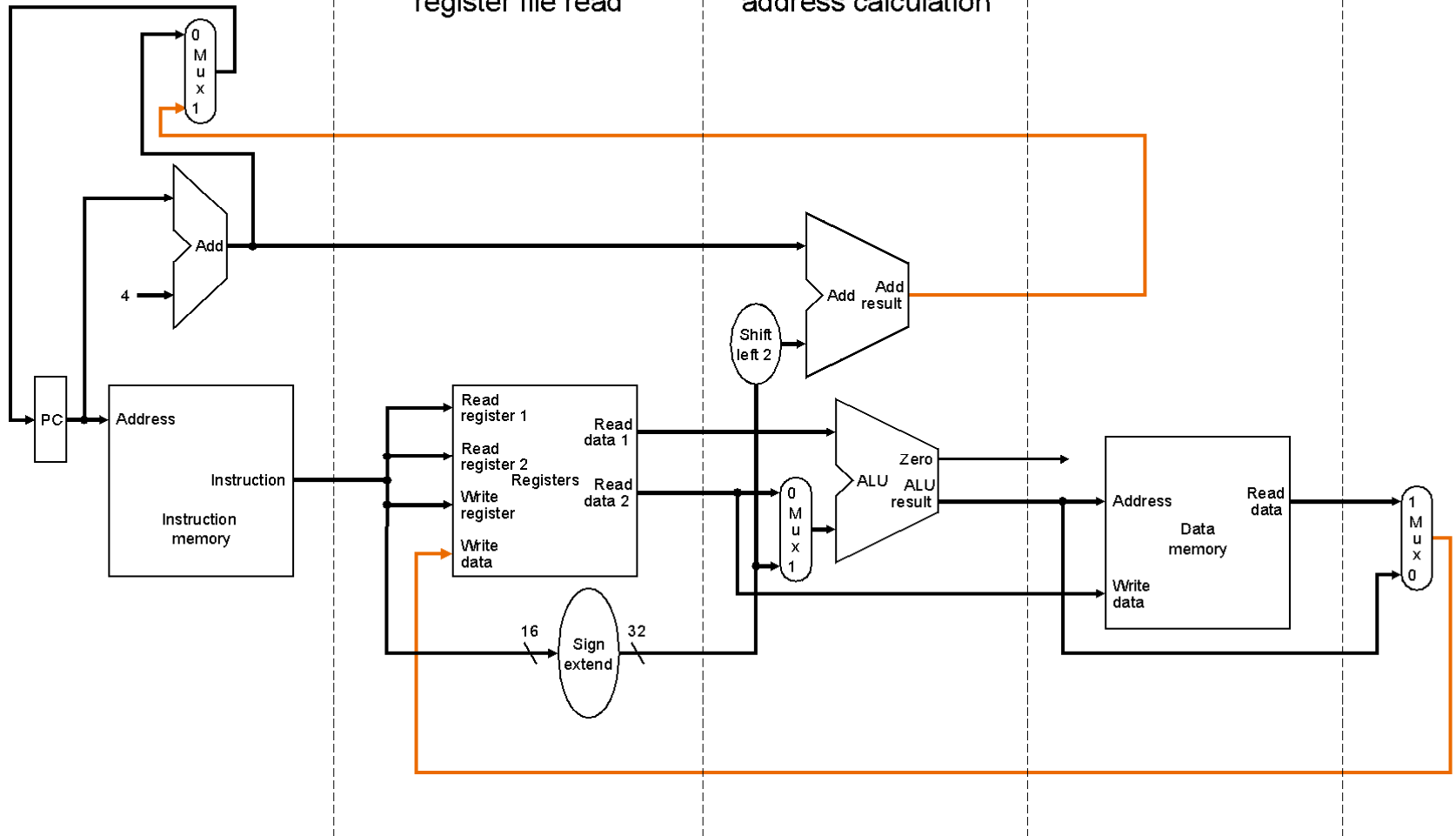
IF: Instruction fetch

ID: Instruction decode/  
register file read

EX: Execute/  
address calculation

MEM: Memory access

WB: Write back



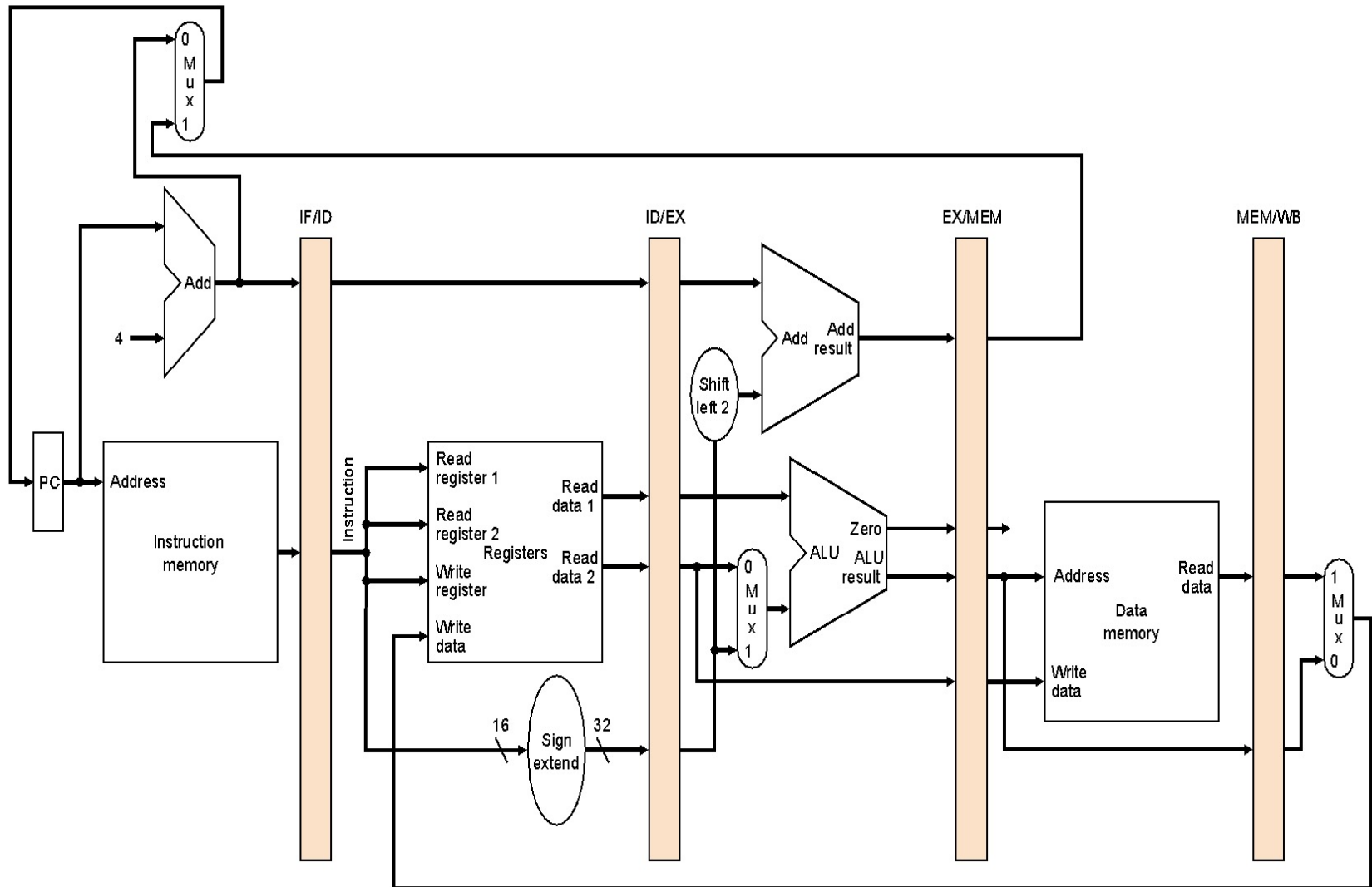
# Instruction and Data Flow

- Instruction and data move generally from left to right with two exceptions:
- *Write-back stage*, which places the result back into the register file in the middle of the data path => data hazard
- The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage => control hazard

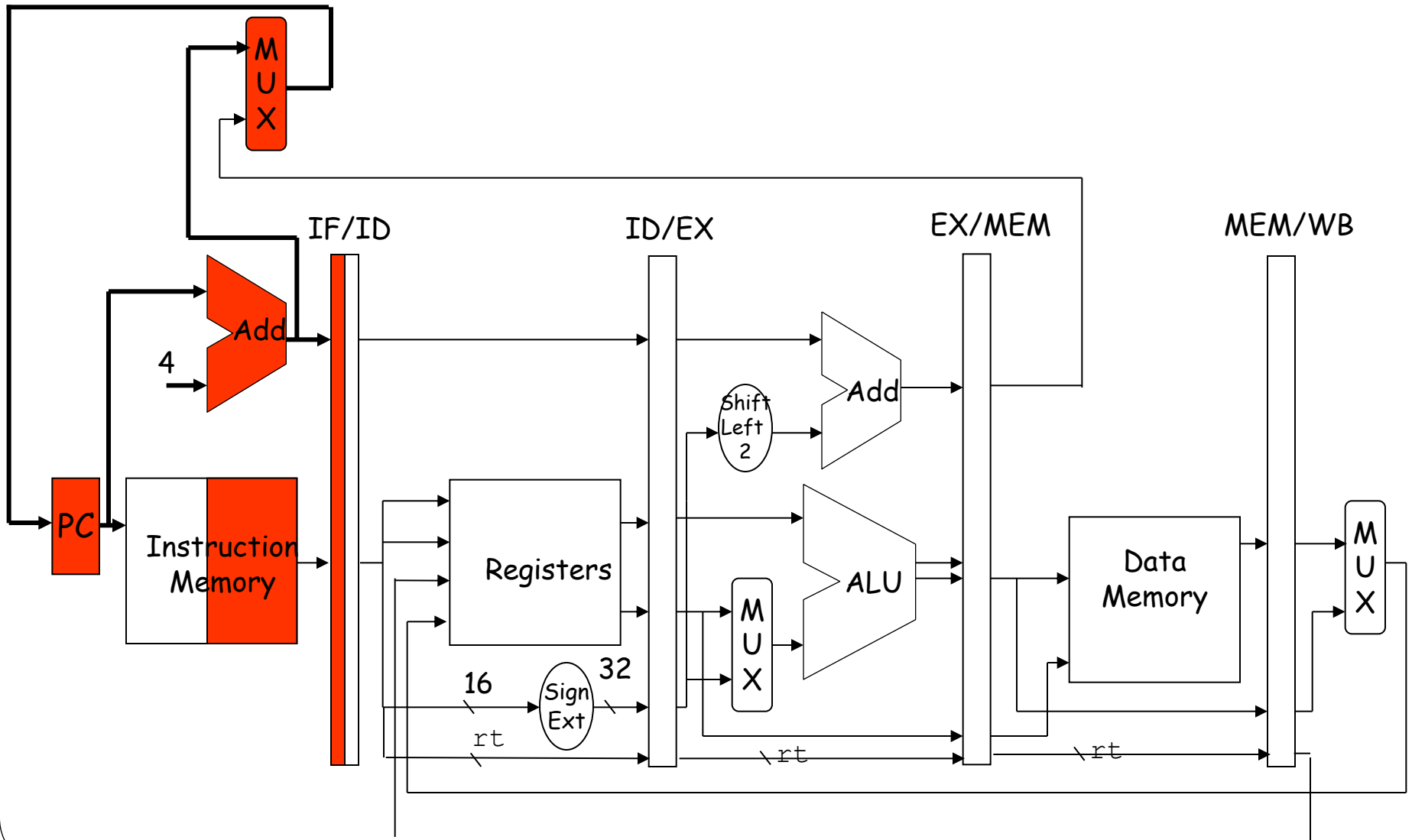
# Design of Pipelined Datapath

- Break the datapath into smaller segments
- These segments can be shared by instructions
- Use registers between two consecutive segments of the datapath to hold the intermediate results.
- Pipeline registers

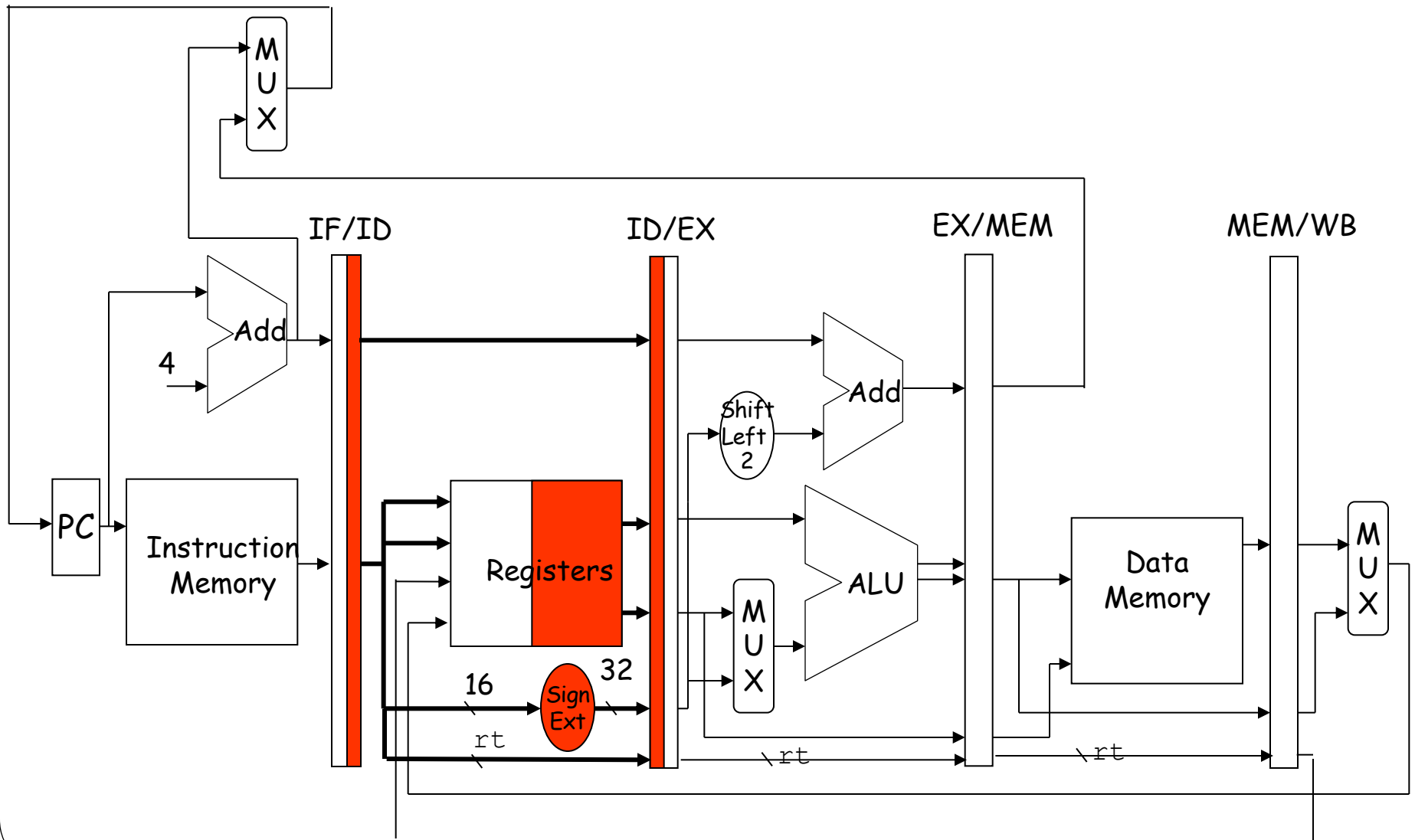
# Pipelined Datapath



# Example: IF of lw

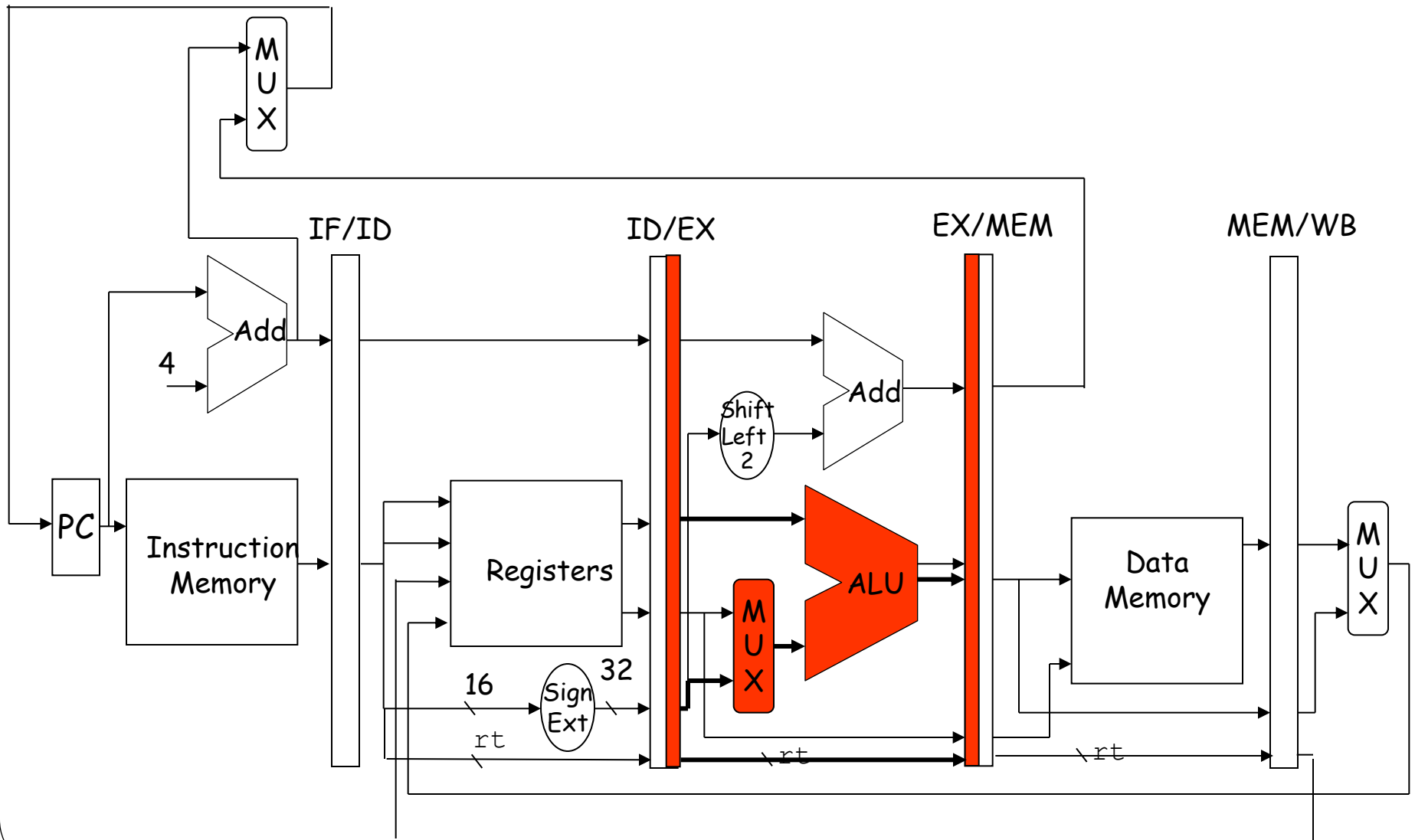


# Example: ID of $lw$

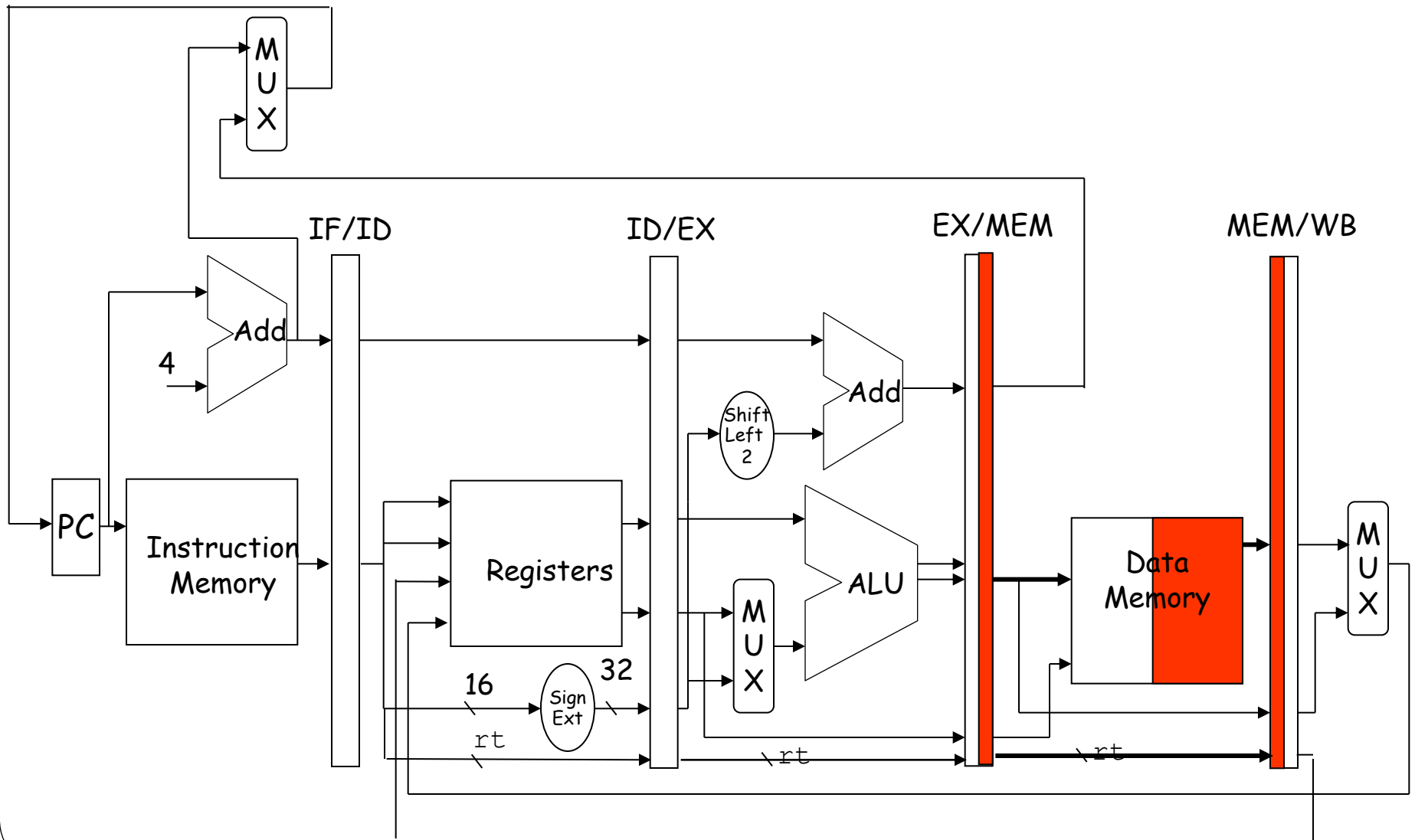




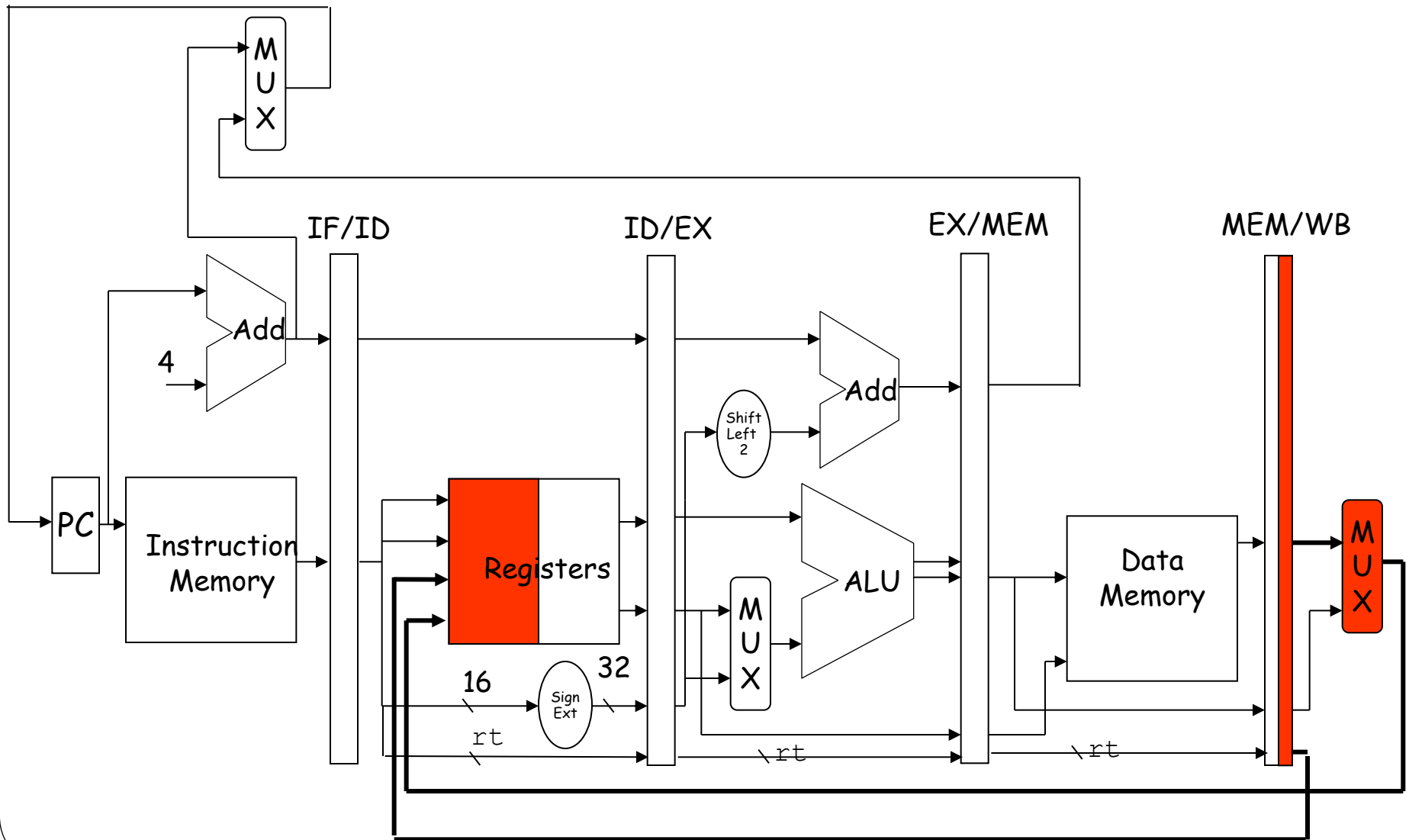
# Example: EX of $lw$



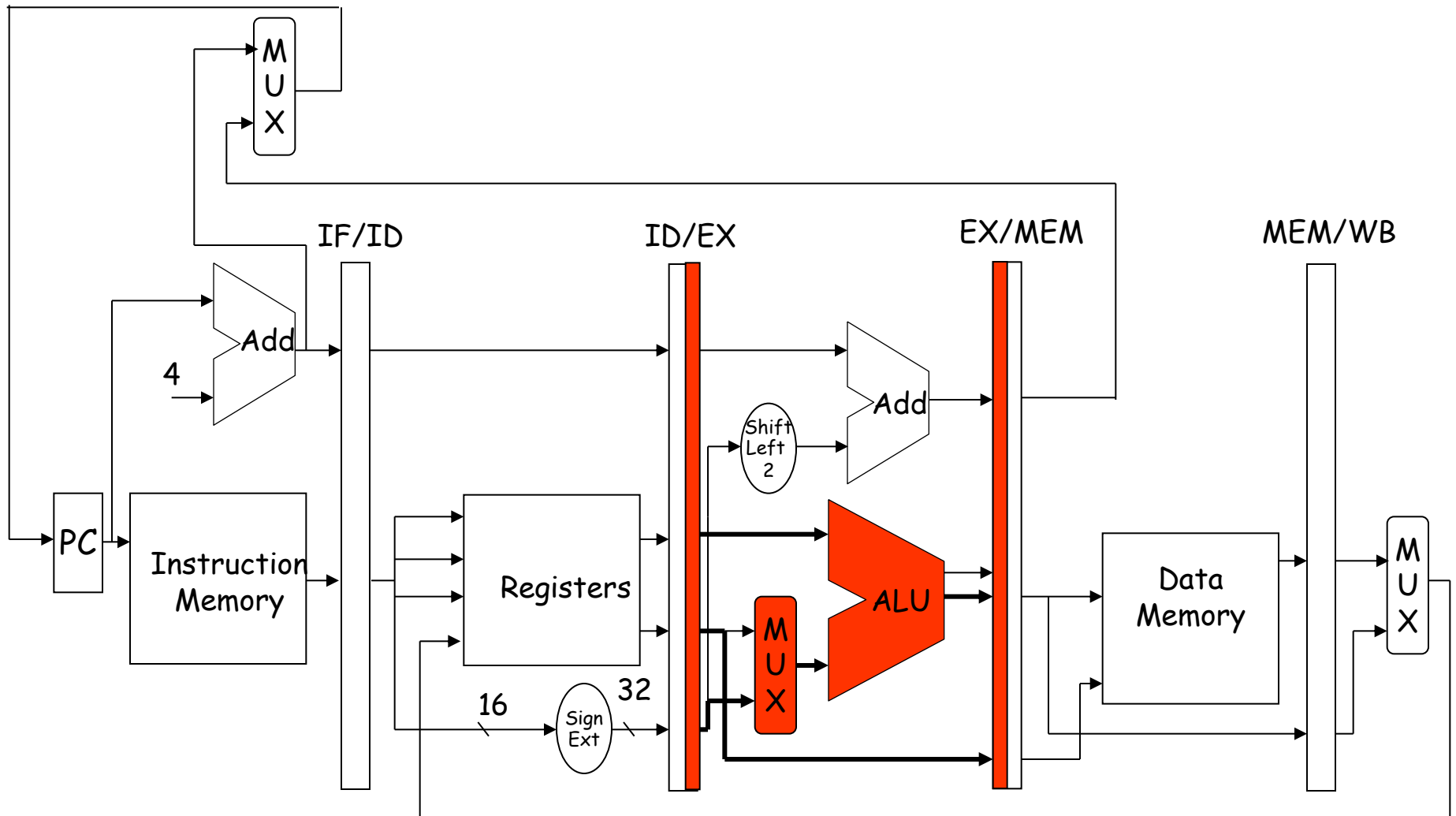
# Example: MEM of $lw$



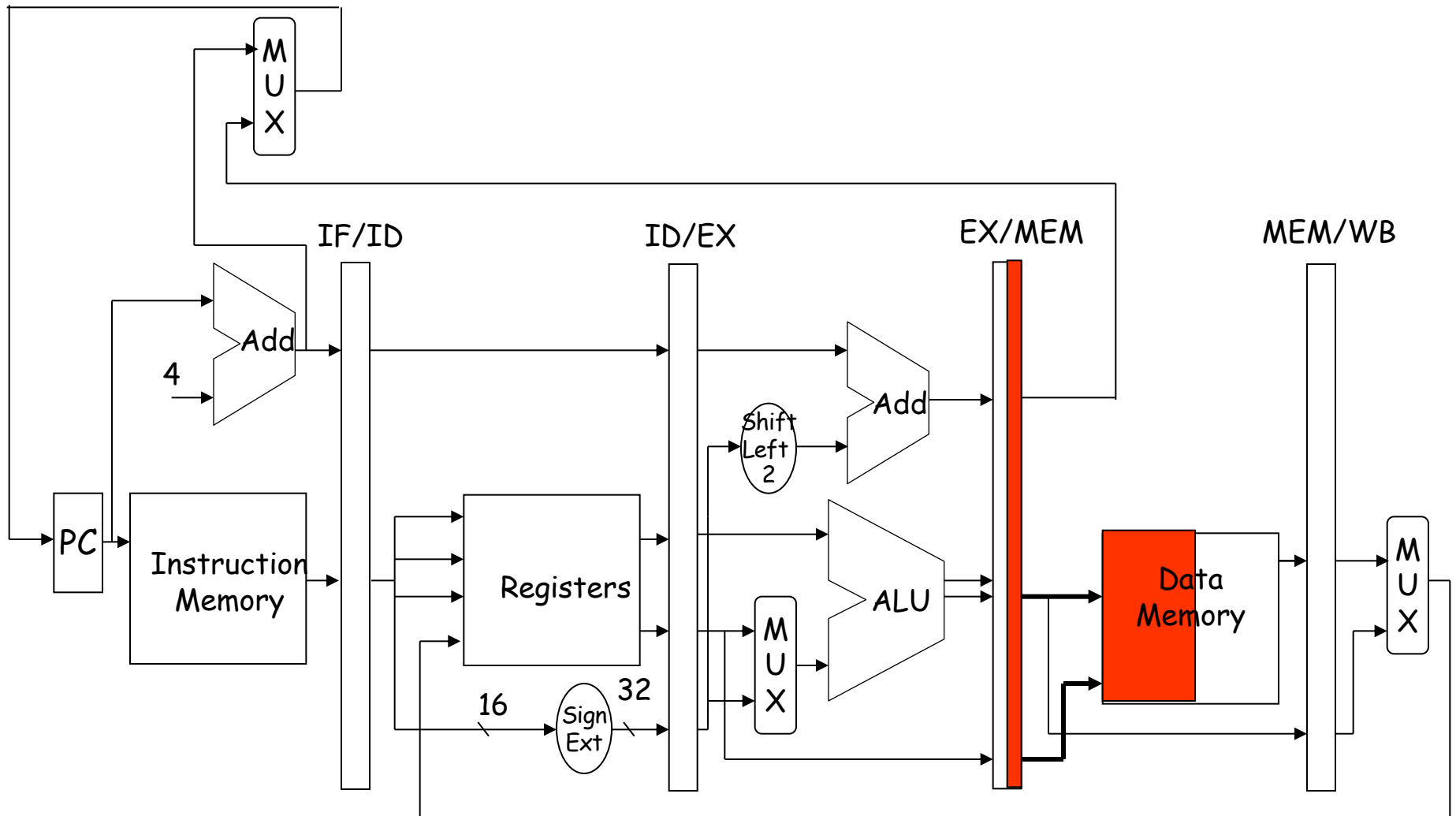
# Example: WB of $lw$



## Example 2: $s_w$ , EX stage

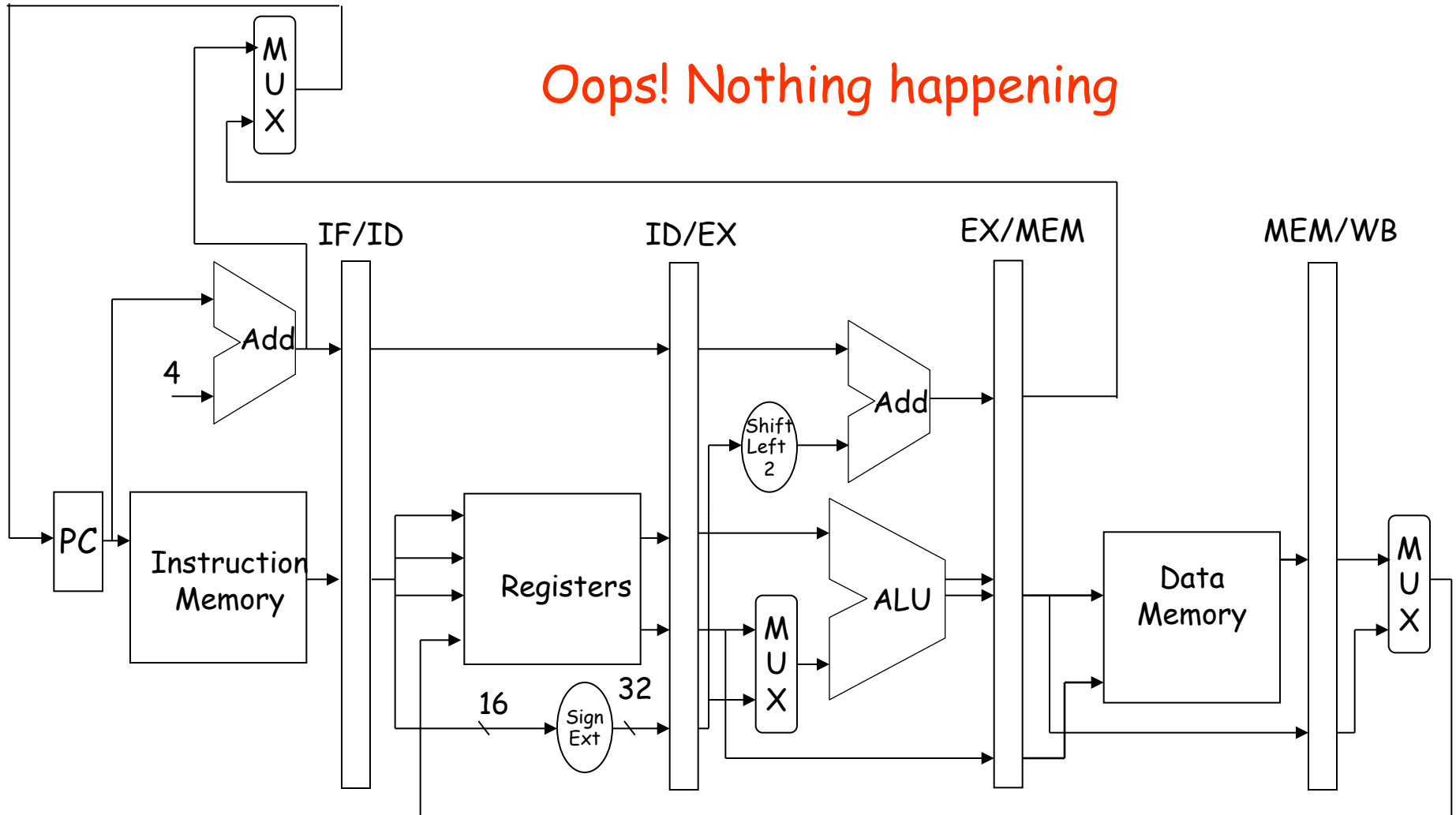


# Example 2: MEM of sw



# Example 2: WB of sw

Oops! Nothing happening



# Summary So Far

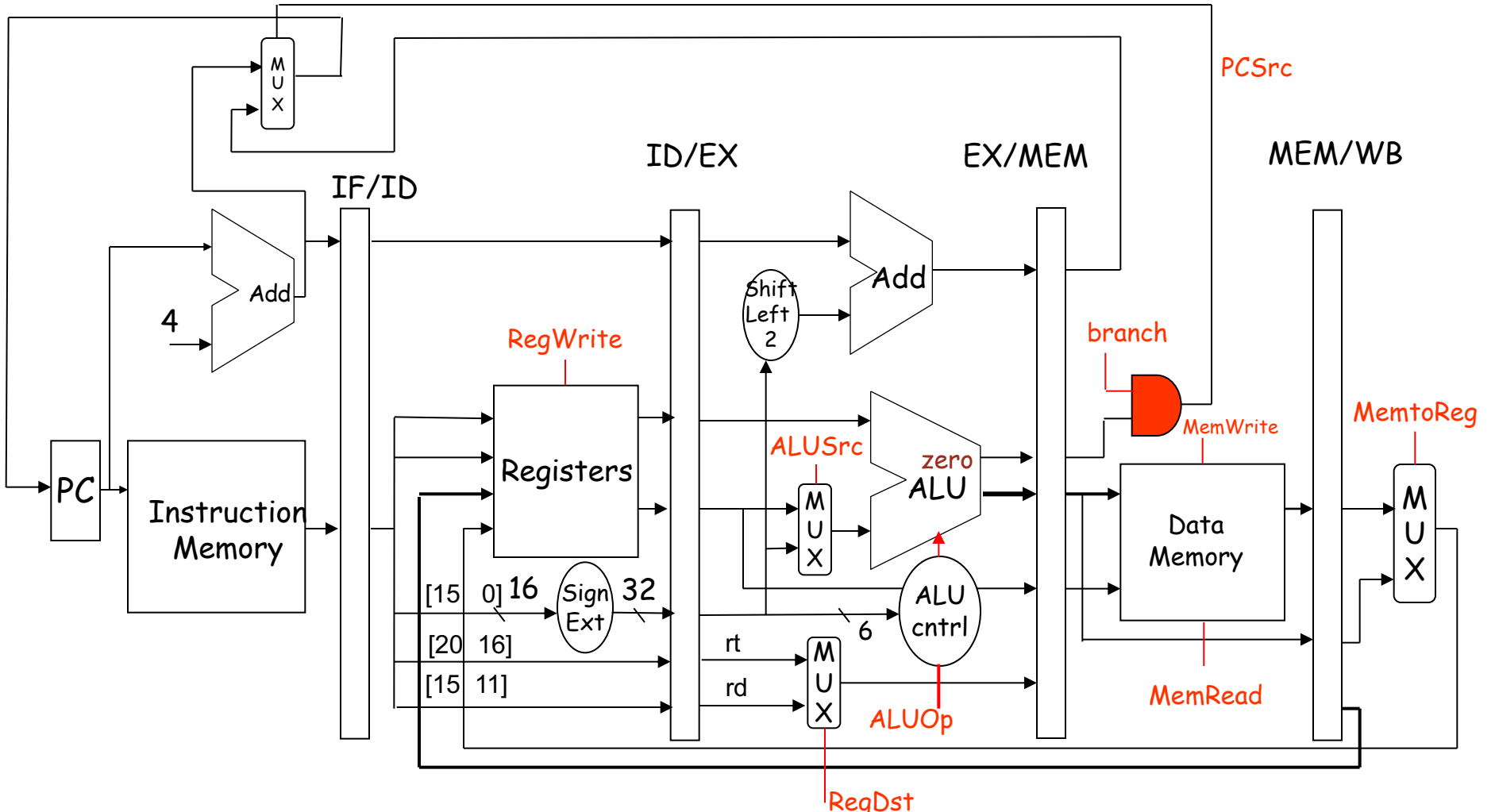
- Not all instructions require the complete datapath.
- No information transfer from one pipeline stage to the next is possible except through pipeline registers
- Each component can be used only in a single pipeline stage
- Everything that happened in the previous stage will be overwritten.

# Pipelined Control

- PC is written on each clock cycle (no write signal)
- No write signals for pipeline registers.
- IF stage: no control signal since instruction is read and PC is updated each cycle
- ID stage: No control signals
- EX stage: RegDst, ALUOp, ALUSrc
- MEM stage: branch, MemRead, MemWrite
- WB stage: MemtoReg, RegWrite



# Pipelined Datapath with Control Signals

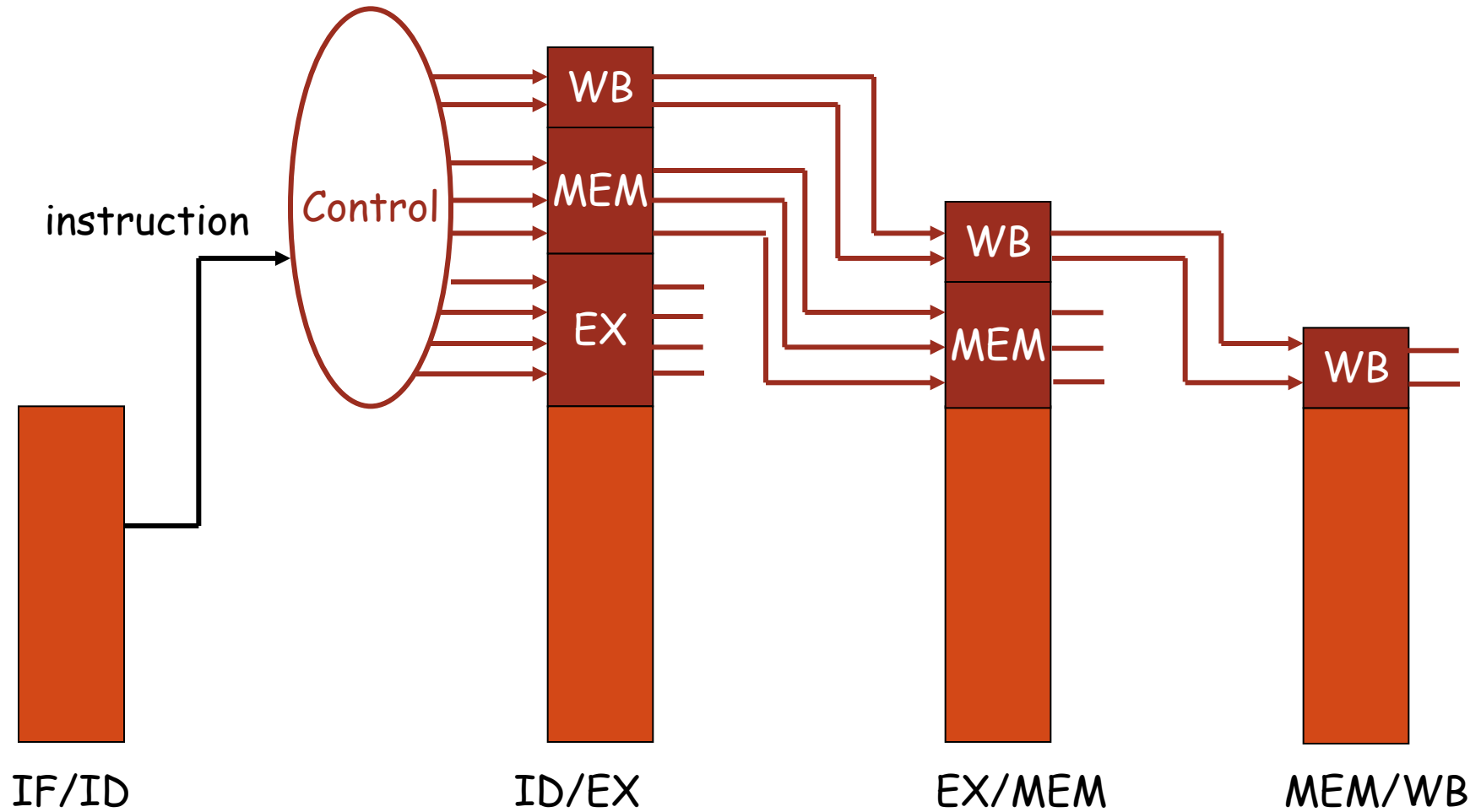


# Control Signals for Instructions

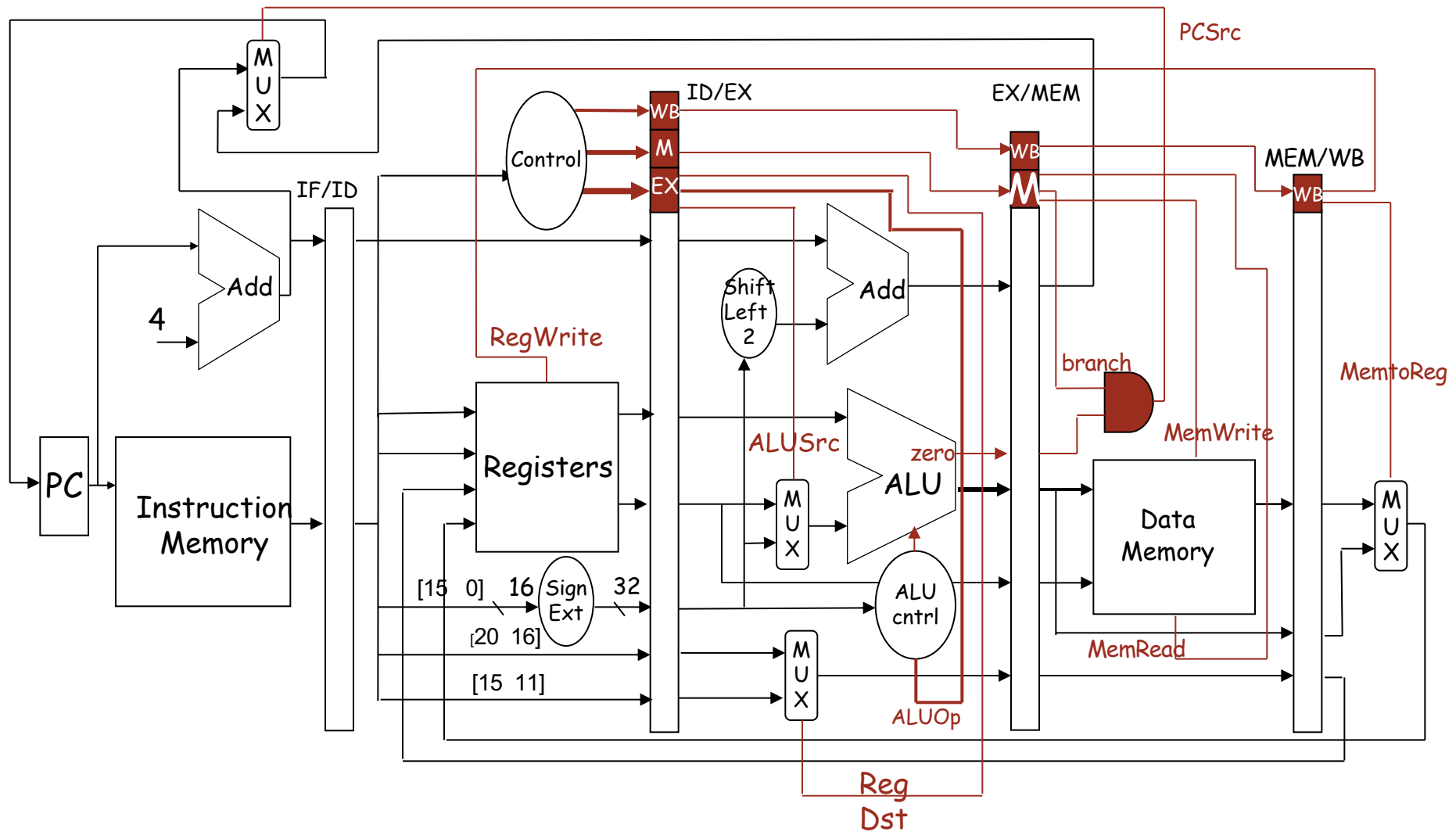
- Nine control signals that start in the EX stage
- Main control unit generates the control signals during the ID stage.

Instruction	EX Stage				MEM Stage			WB Stage	
	RegDest	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto-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

# Control Lines for the Last Three Stages



# Pipelined Datapath with Control Signals



# Example

- Show how these five instructions will go through the pipeline:

```
lw  $10, 20($1)
```

```
sub $11, $2, $3
```

```
and $12, $4, $5
```

```
or      $13, $6, $7
```

```
add $14, $8, $9
```

# Clock 1

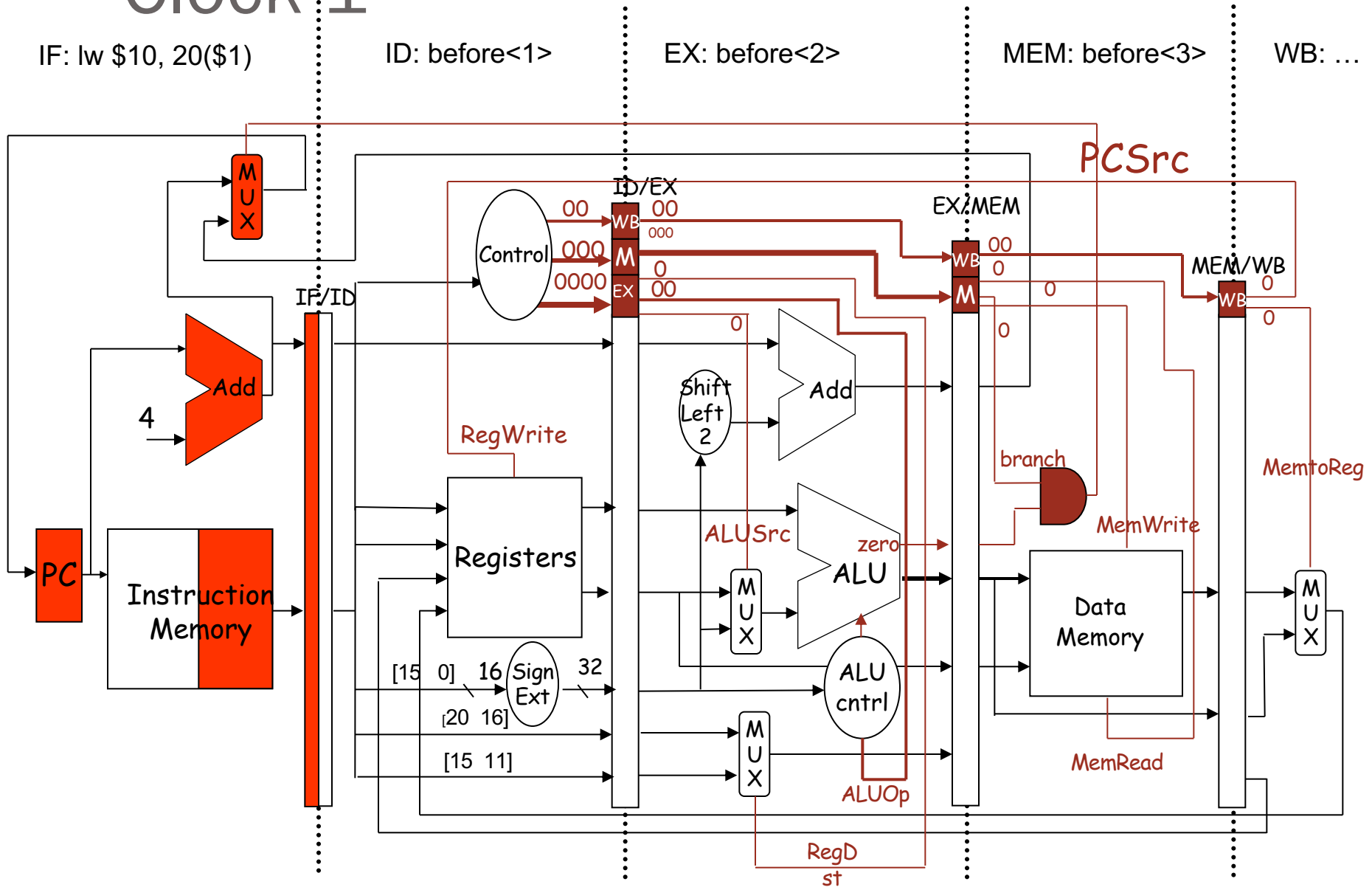
IF: lw \$10, 20(\$1)

ID: before<1>

EX: before<2>

MEM: before<3>

WB: ...



# Clock 2

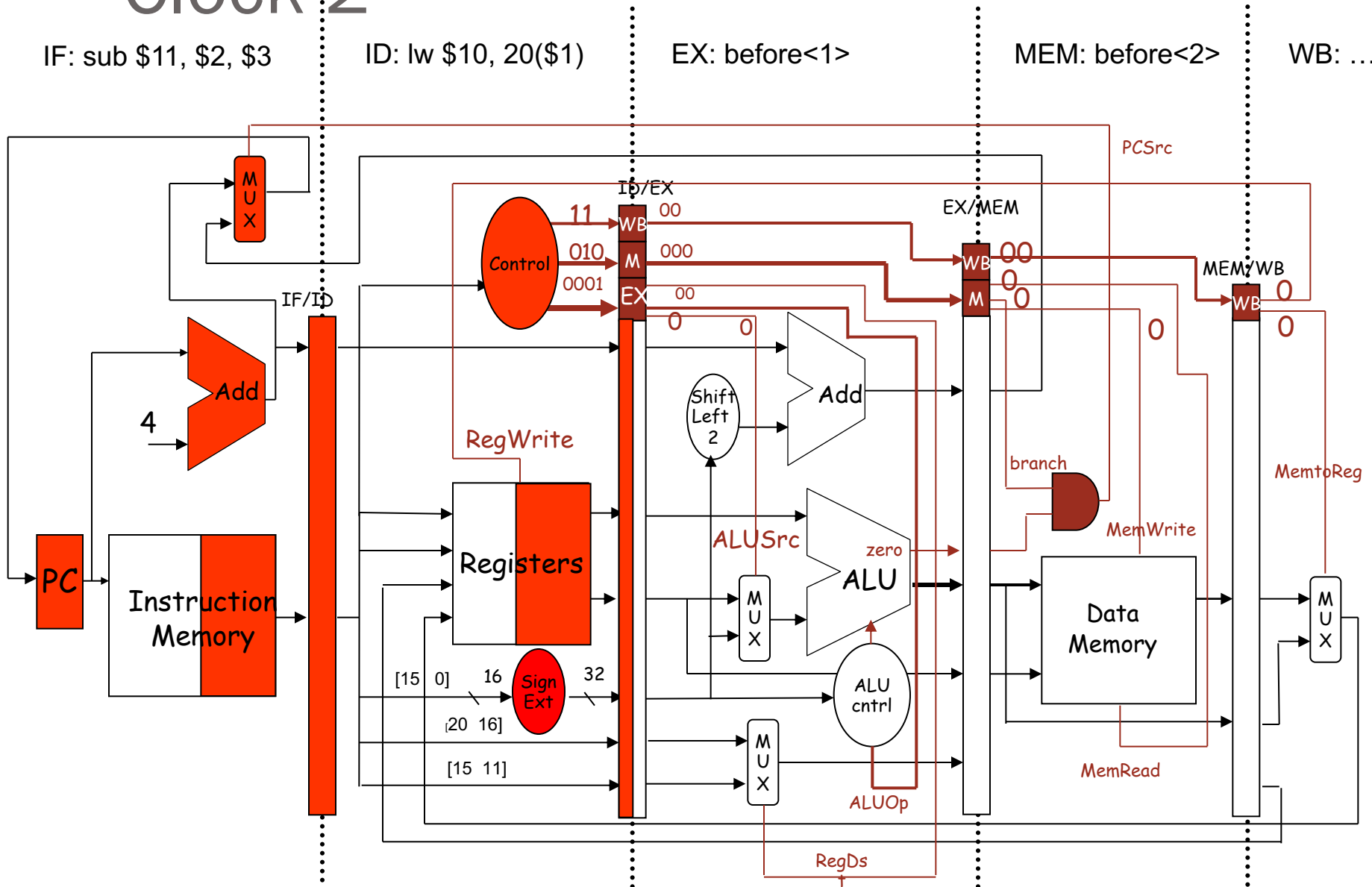
IF: sub \$11, \$2, \$3

ID: lw \$10, 20(\$1)

EX: before<1>

MEM: before<2>

WB: ...



# Clock 3

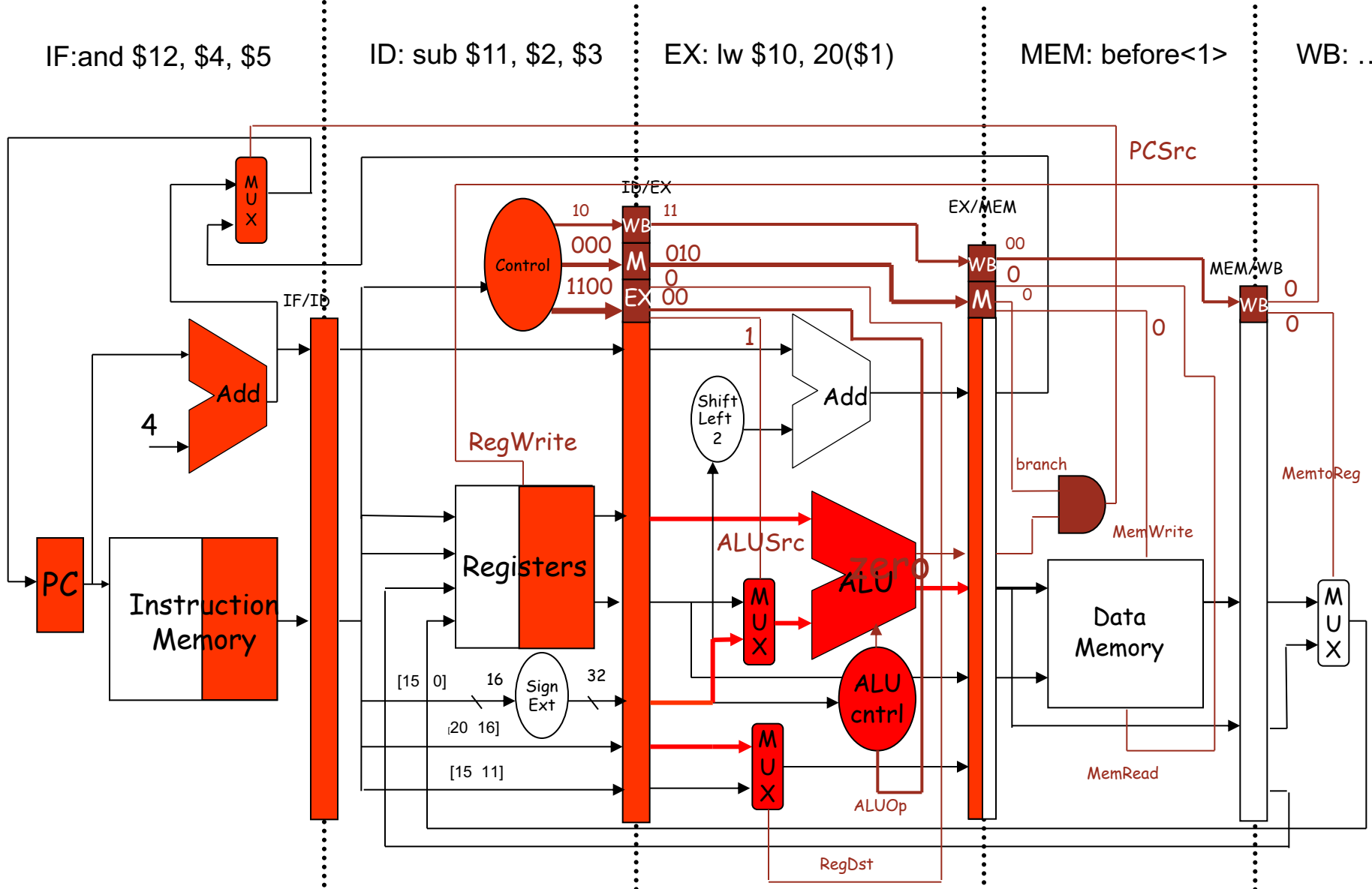
IF: and \$12, \$4, \$5

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

EX: lw \$10, 20(\$1)

MEM: before<1>

WB: ...





# Clock 4

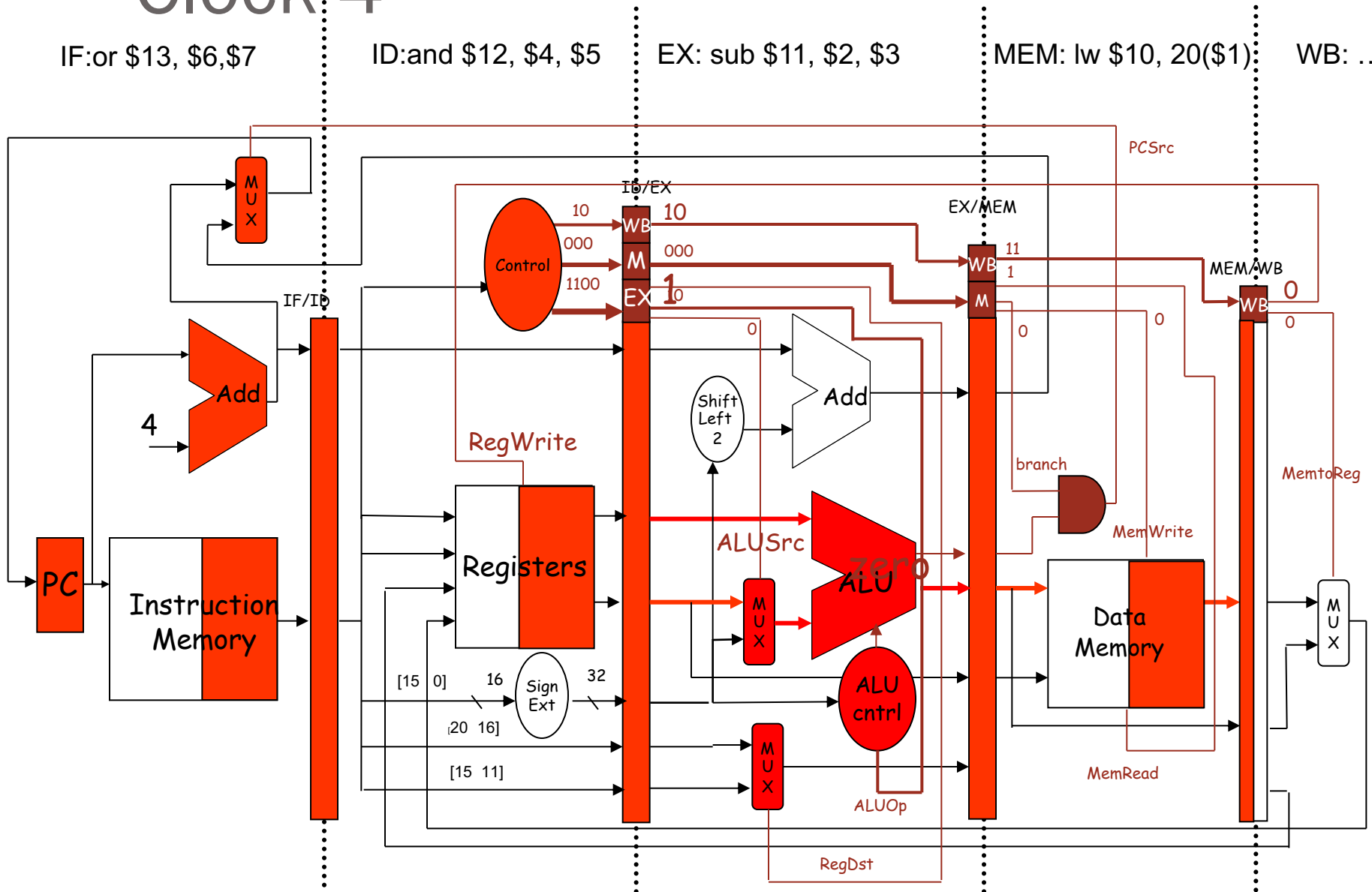
IF: or \$13, \$6, \$7

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

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

MEM: lw \$10, 20(\$1)

WB: ...



# Clock 5

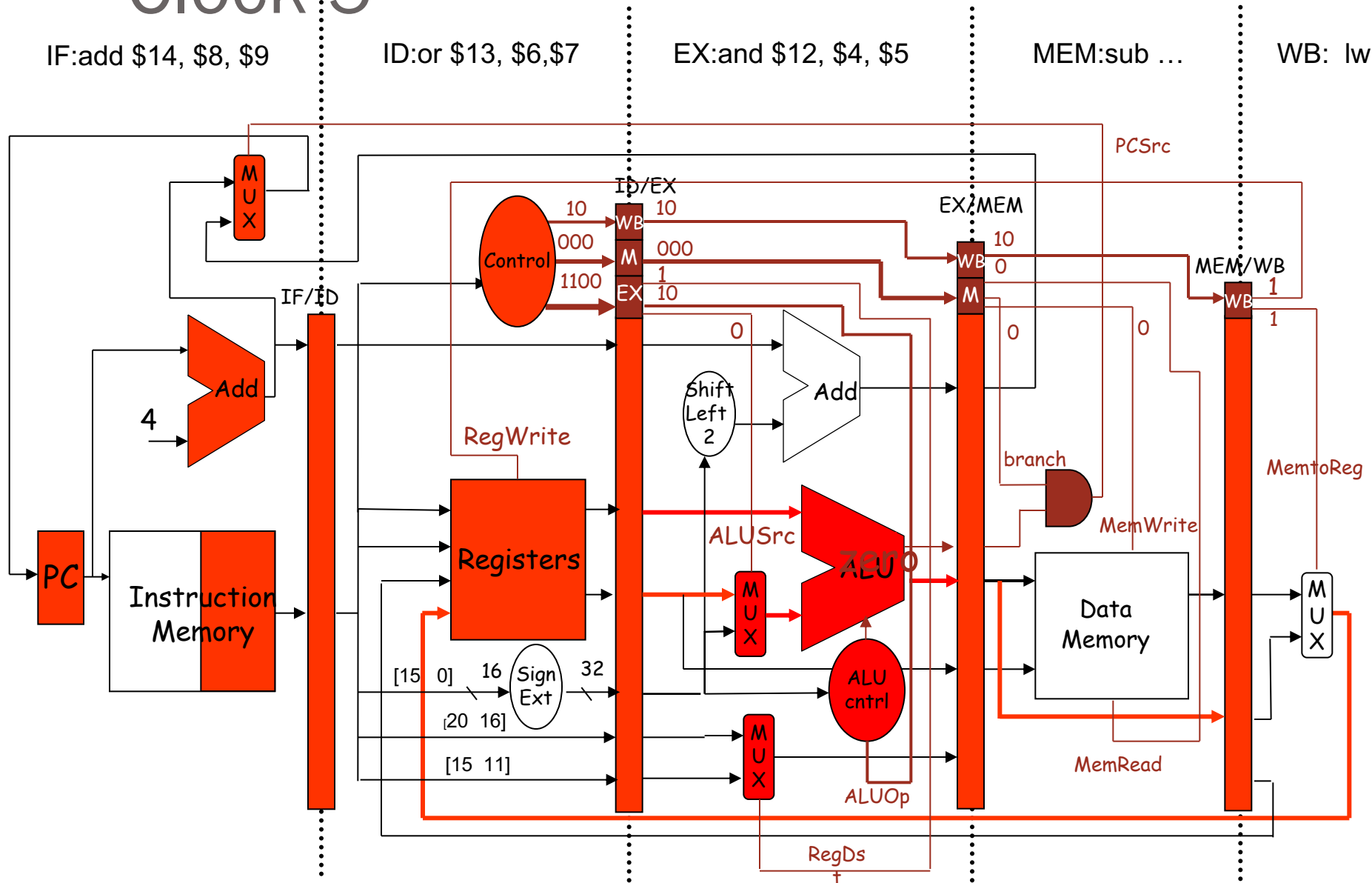
IF:add \$14, \$8, \$9

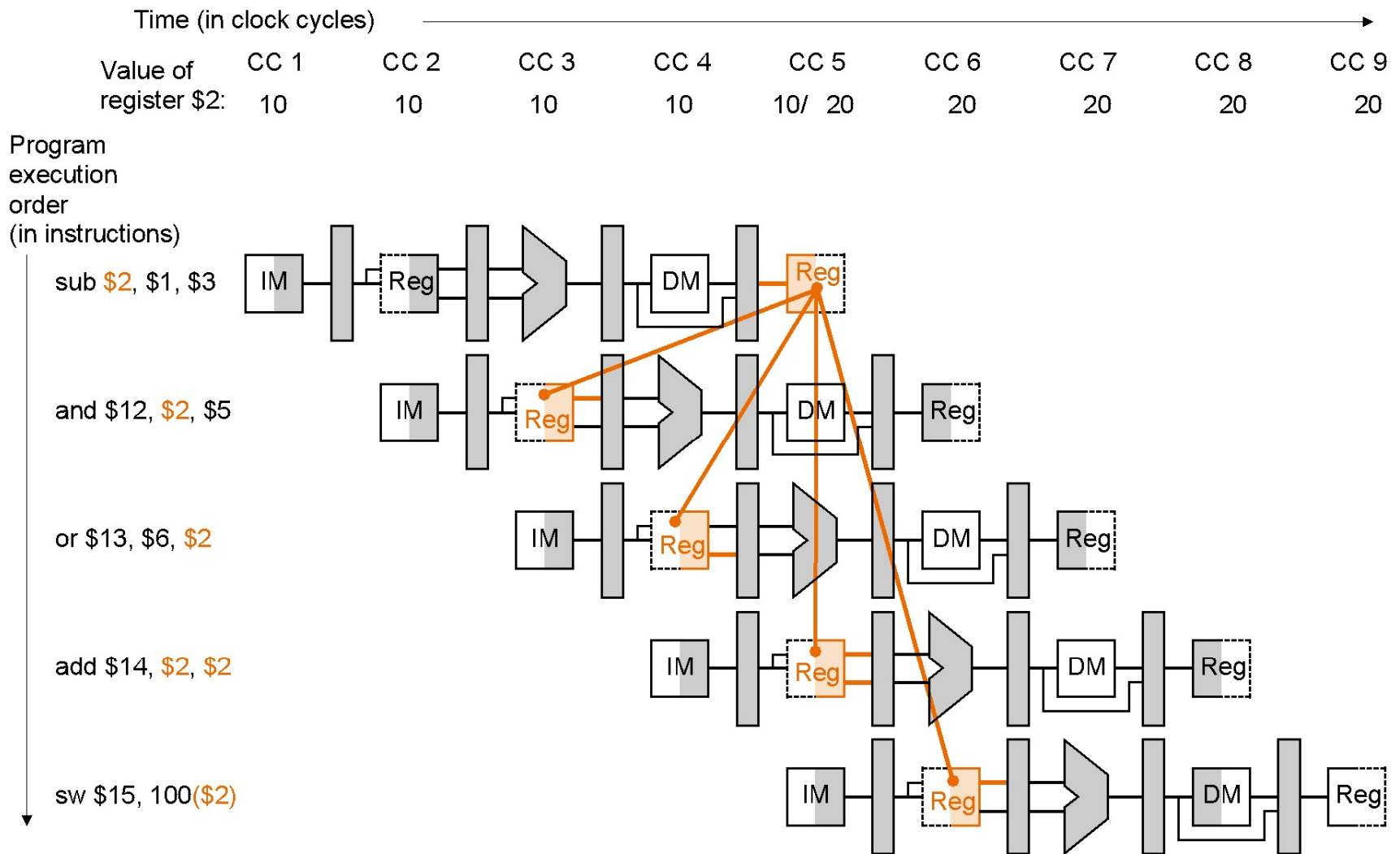
ID:or \$13, \$6,\$7

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

MEM:sub ...

WB: lw





Assumption: a register can be read and written in the same clock cycle.

# Data Hazards & Forwarding

- Software solution for data hazards:

```
sub $2, $1, $3
```

```
nop
```

```
nop
```

```
and $12, $2, $5
```

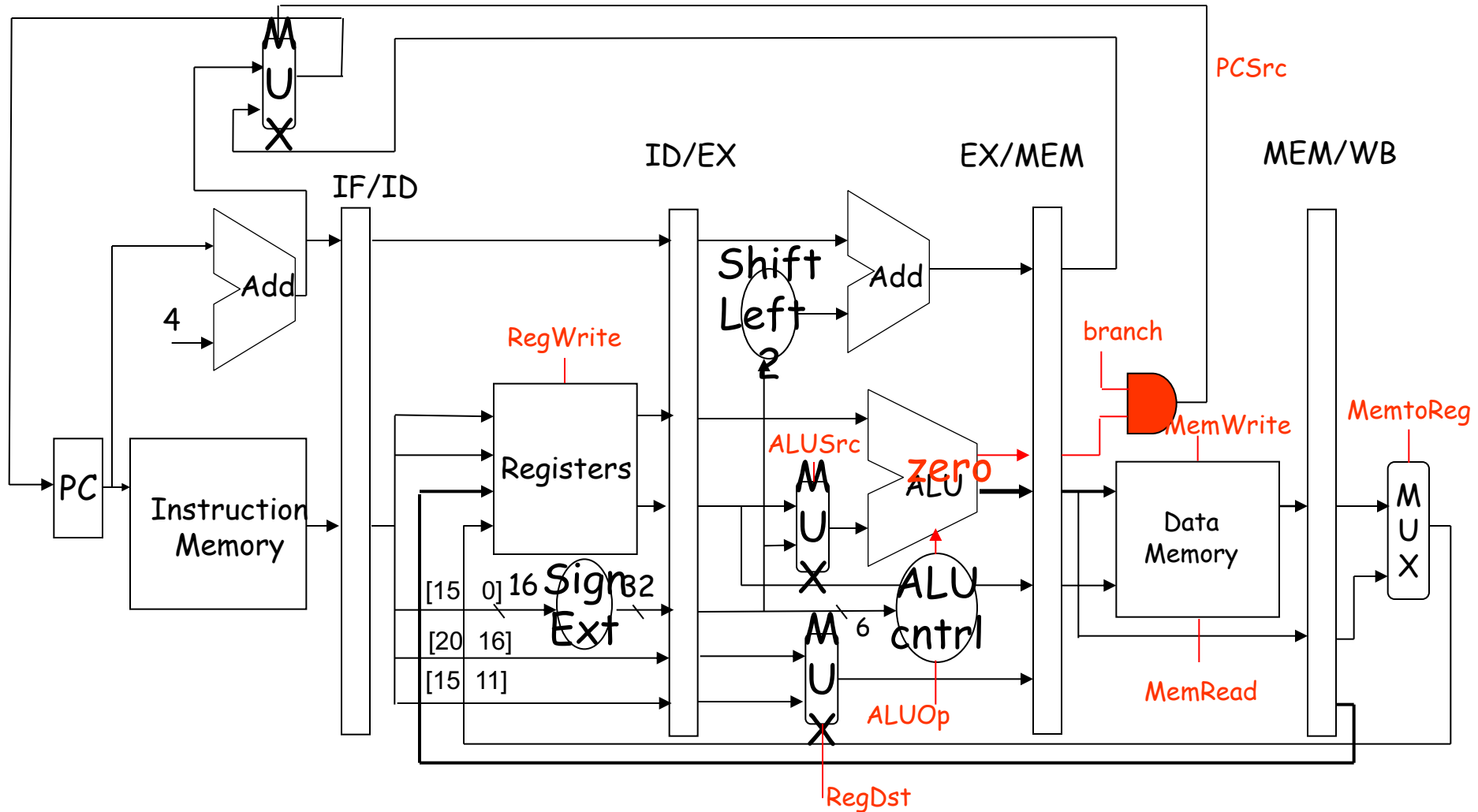
```
or      $13, $6, $2
```

```
add $14, $2, $2
```

```
sw      $15, 100($2)
```

- But, these dependencies happen too often to rely on compilers

# Pipelined Datapath with Control Signals



# A Notation for Detecting Data Hazards

- Two types of data hazards.

1.  $\text{EX/MEM.rd} = \text{ID/EX.rs}$   
 $\text{EX/MEM.rd} = \text{ID/EX.rt}$

2.  $\text{MEM/WB.rd} = \text{ID/EX.rs}$   
 $\text{MEM/WB.rd} = \text{ID/EX.rt}$

- Easy to detect data hazards using this notation

```
sub    $2,    $1,    $3
and    $12,   $2,    $5
or     $13,   $6,    $2
```

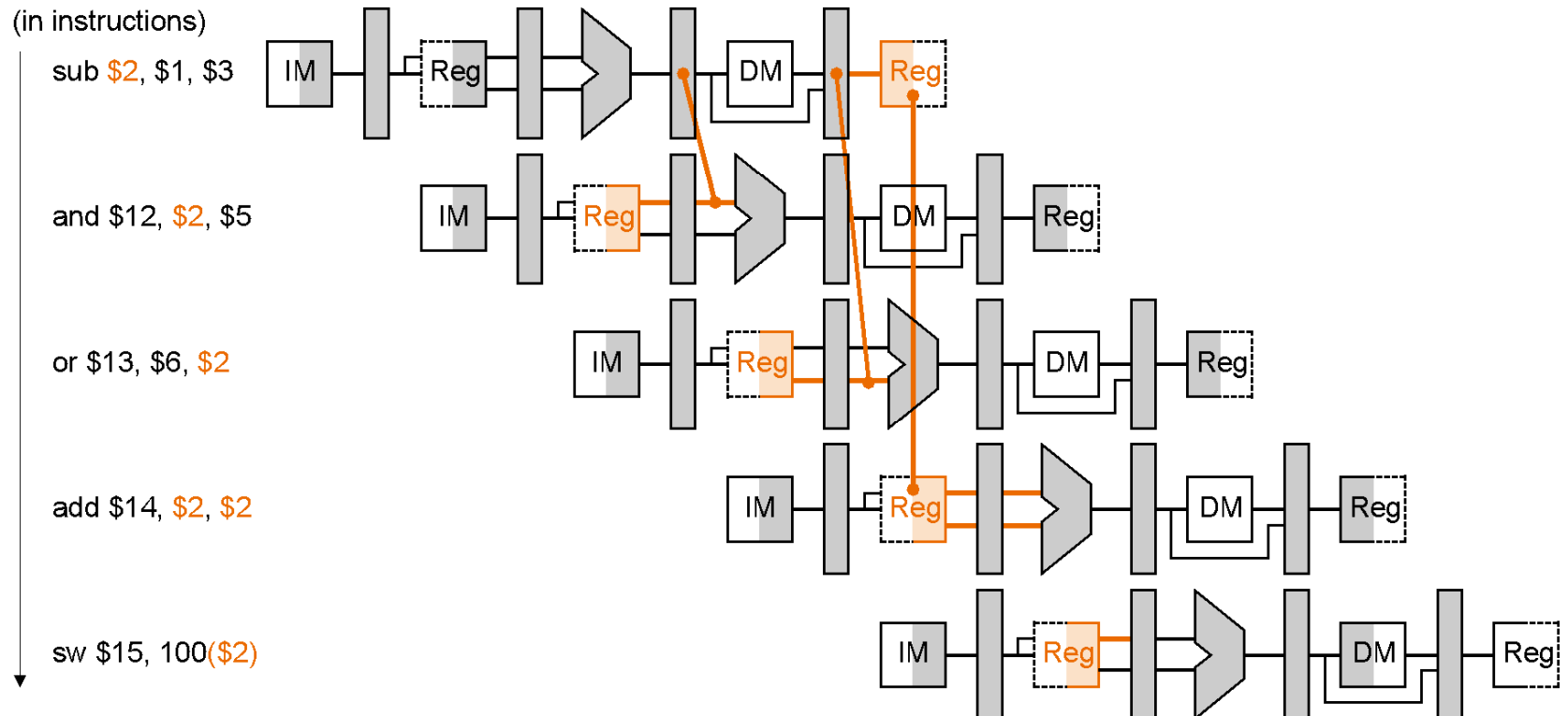
- sub-and:  $\text{EX/MEM.rd} = \text{ID/EX.rs} = \$2$
- sub-or:  $\text{MEM/WB.rd} = \text{ID/EX.rt} = \$2$

# Forwarding

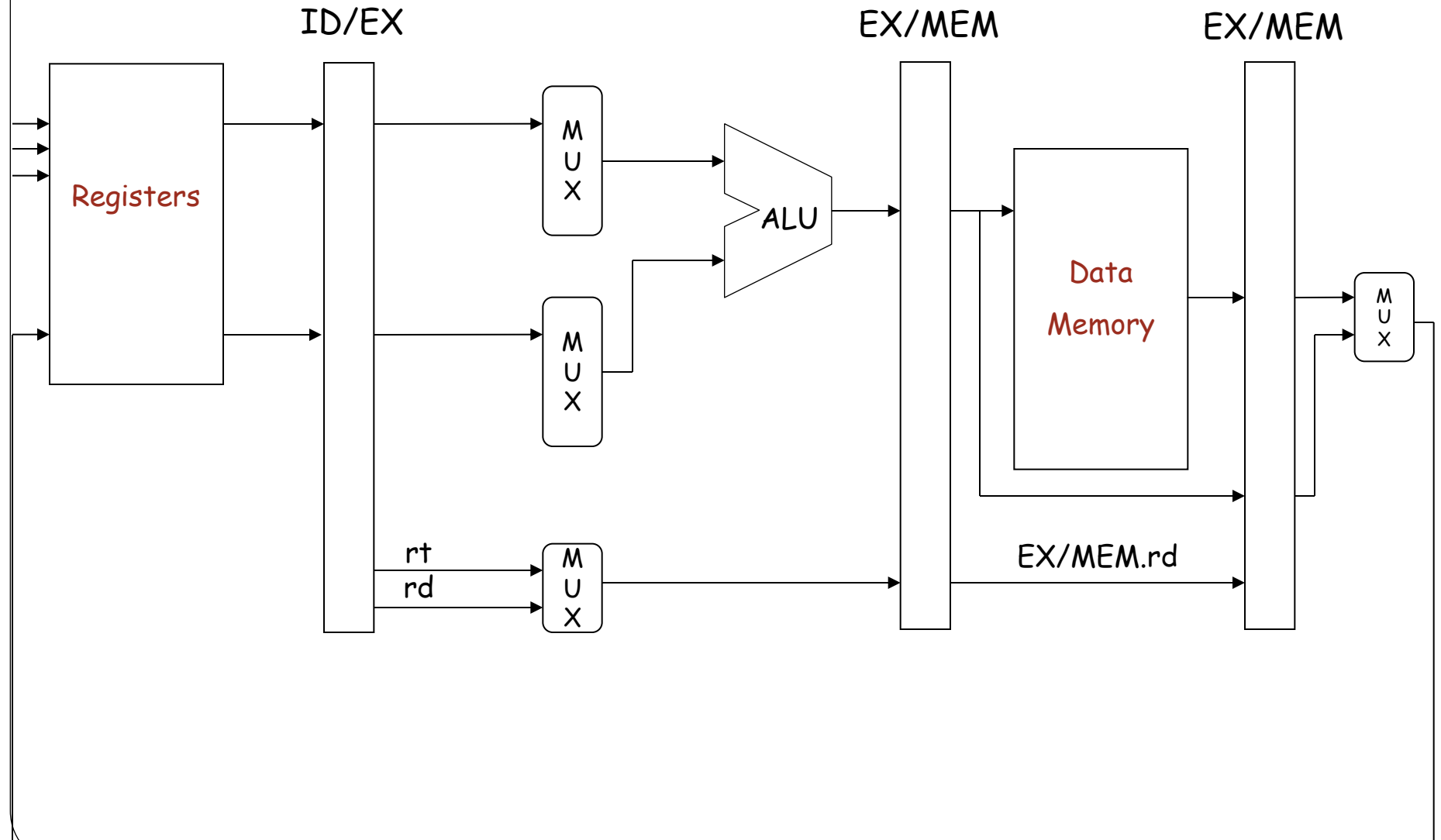
Time (in clock cycles)

	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

Program  
execution order  
(in instructions)

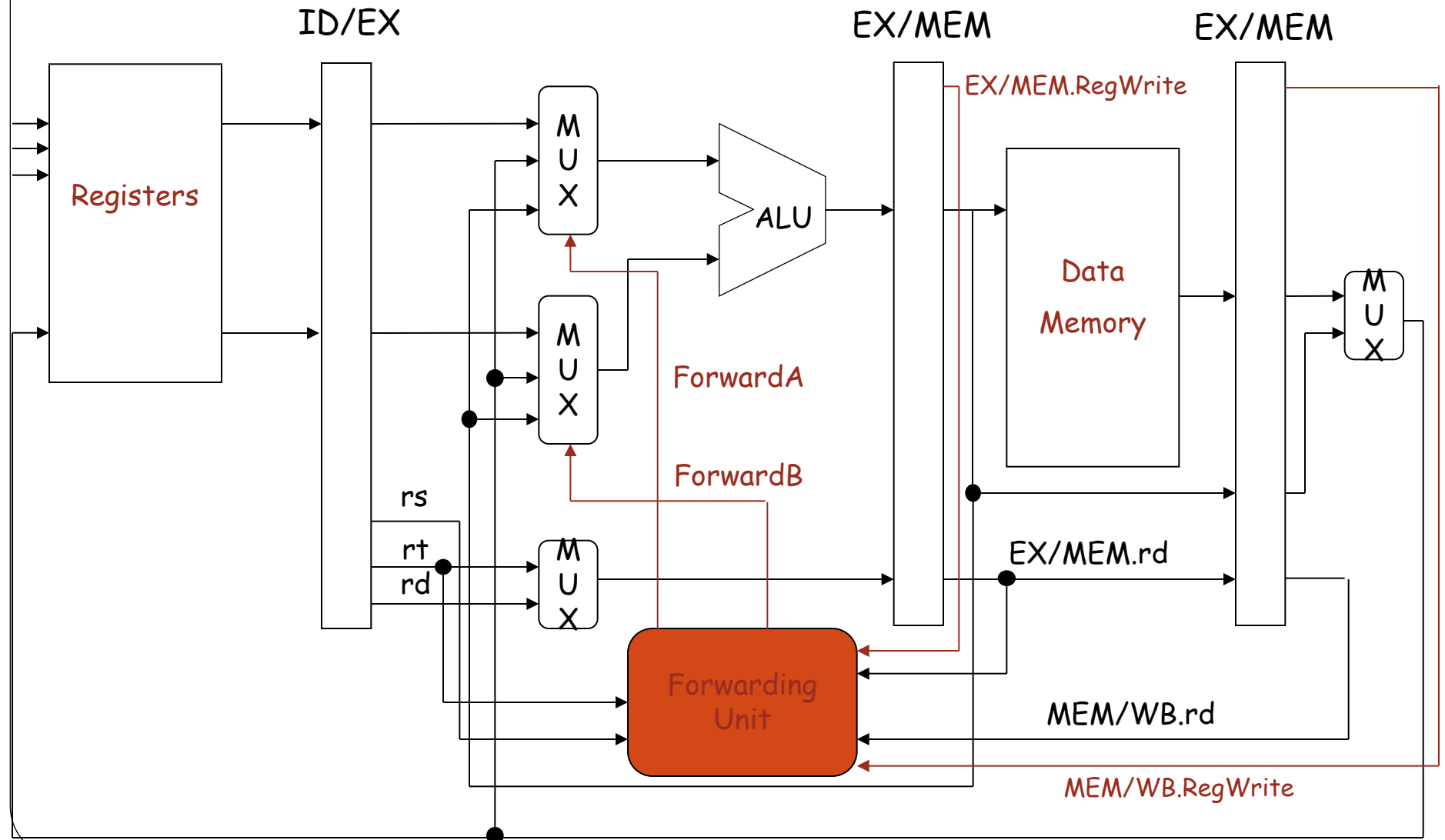


# Pipeline Without Forwarding





# Pipeline with Forwarding Unit



# Control Signals to Resolve Data Dependencies

- EX Hazard:

```
if (EX/MEM.RegWrite and (EX/MEM.rd ≠ 0)
    and (EX/MEM.rd = ID/EX.rs) )
```

**ForwardA = 10**

```
if (EX/MEM.RegWrite and (EX/MEM.rd ≠ 0)
    and (EX/MEM.rd = ID/EX.rt) )
```

**ForwardB = 10**

- MEM Hazard:

```
if (MEM/WB.RegWrite and (MEM/WB.rd ≠ 0)
    and (MEM/WB.rd = ID/EX.rs) )
```

**ForwardA = 01**

```
if (MEM/WB.RegWrite and (MEM/WB.rd ≠ 0)
    and (MEM/WB.rd = ID/EX.rt) )
```

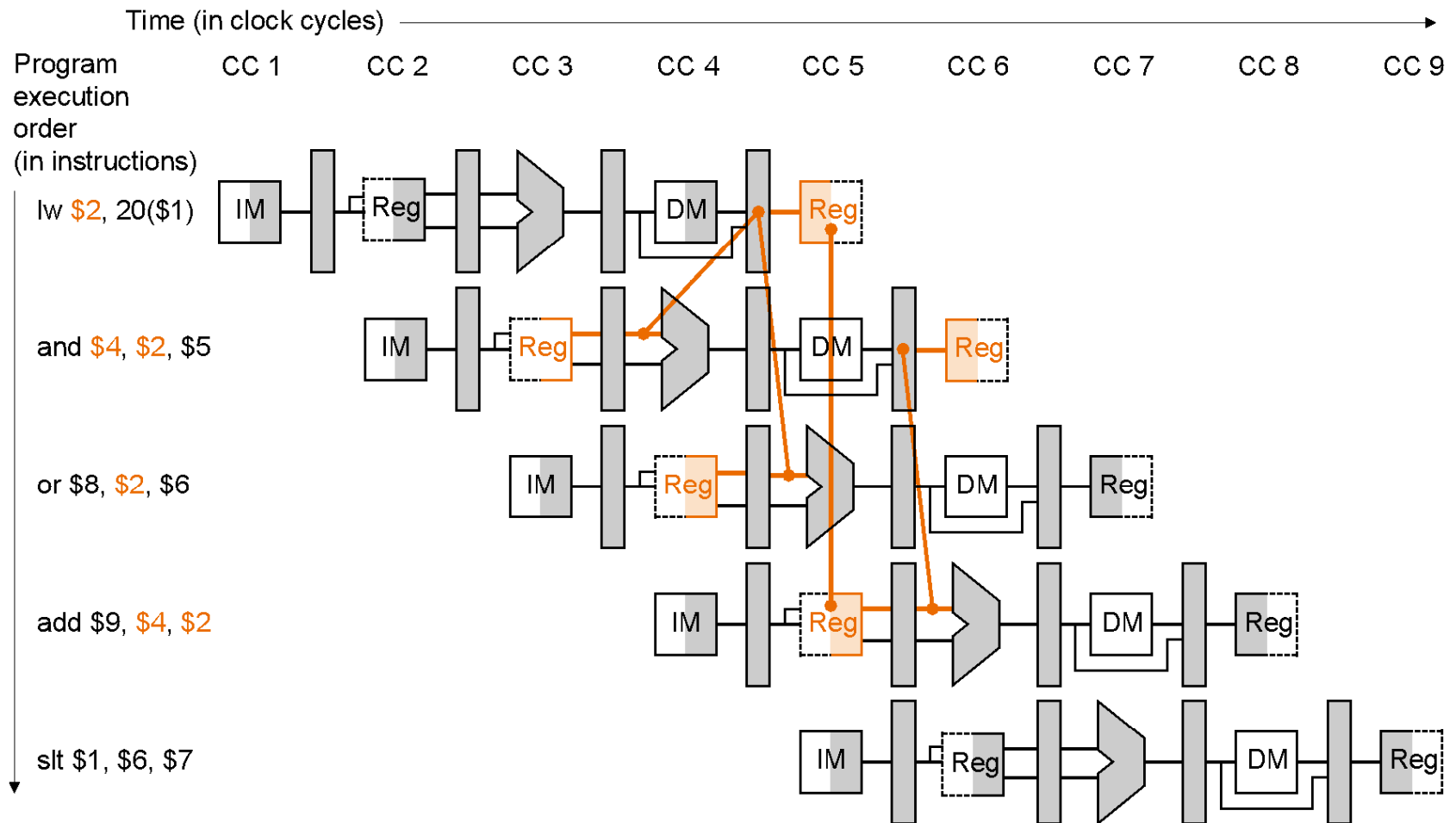
**ForwardB = 01**

# Control Values for Forwarding MUXes

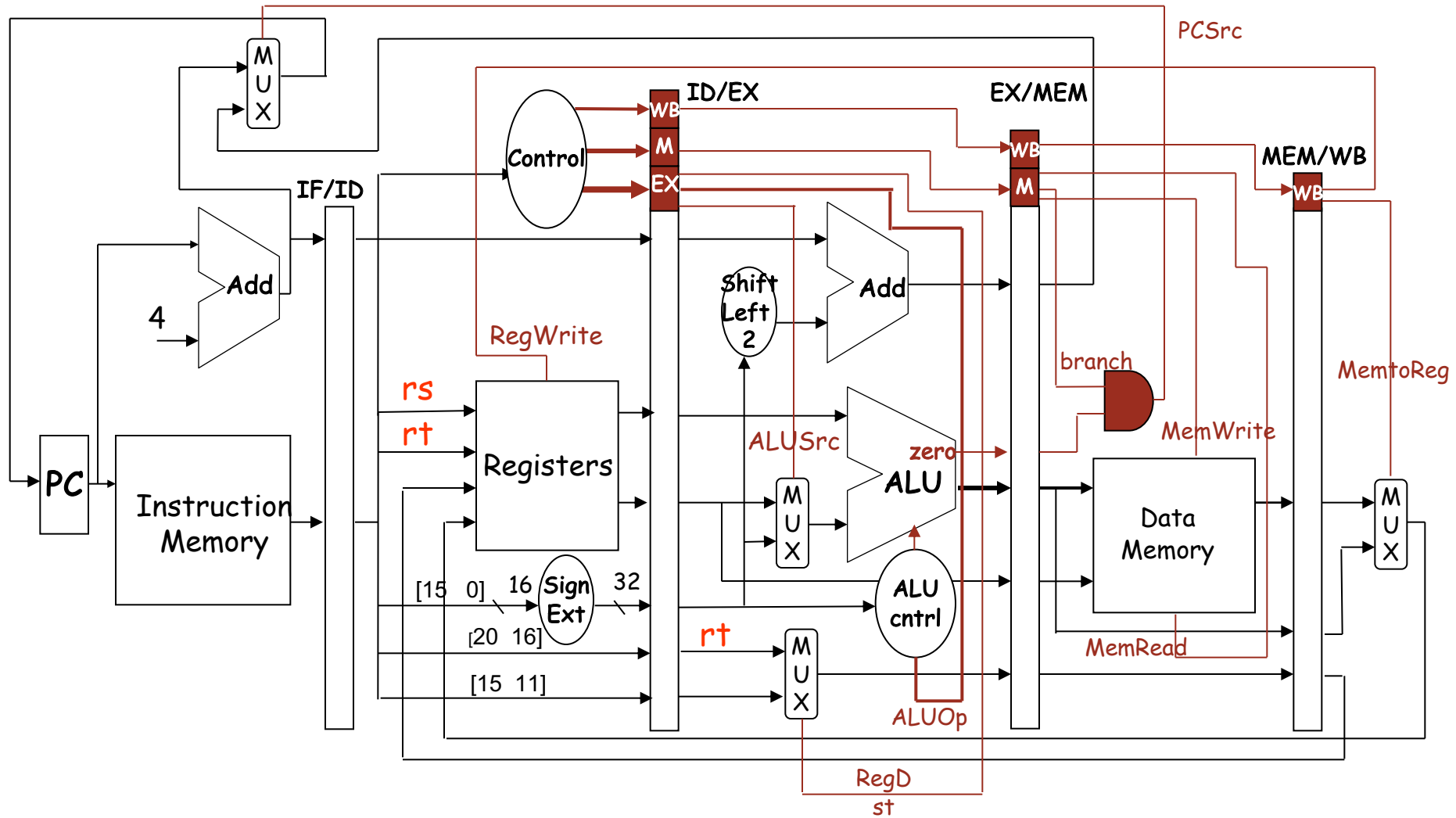
MUX control	Source	Explanation
ForwardA=00	ID/EX	The first ALU operand comes from the register file
ForwardA=10	EX/MEM	The first ALU operand is forwarded from the prior ALU result
ForwardA=01	MEM/WB	The first ALU operand is forwarded from the data memory or an earlier ALU result
ForwardB=00	ID/EX	The second ALU operand comes from the register file
ForwardB=10	EX/MEM	The second ALU operand is forwarded from the prior ALU result
ForwardB=01	MEM/WB	The second ALU operand is forwarded from the data memory or an earlier ALU result

# Can't Always Forward - Stalls

Load word can still cause a hazard:



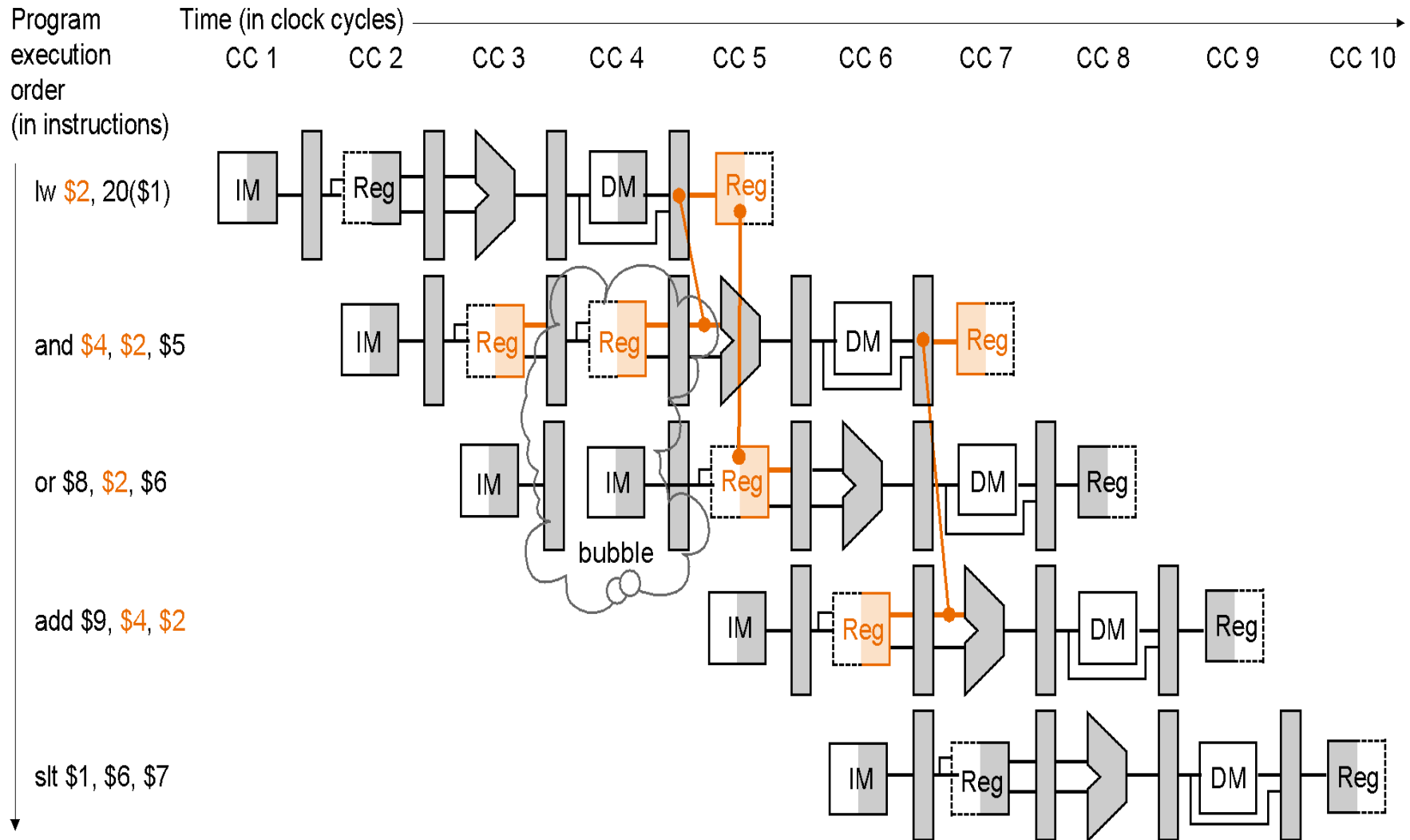
# Pipelined Datapath with Control



# Hazard Detection Unit

- **If** (ID/EX.MemRead **and**  
((ID/EX.rt = IF/ID.rs)  
**or**  
(ID/EX.rt = IF/ID.r**t**)))  
    stall the pipeline

# Stalling the Pipeline



# How to Stall the Pipeline?

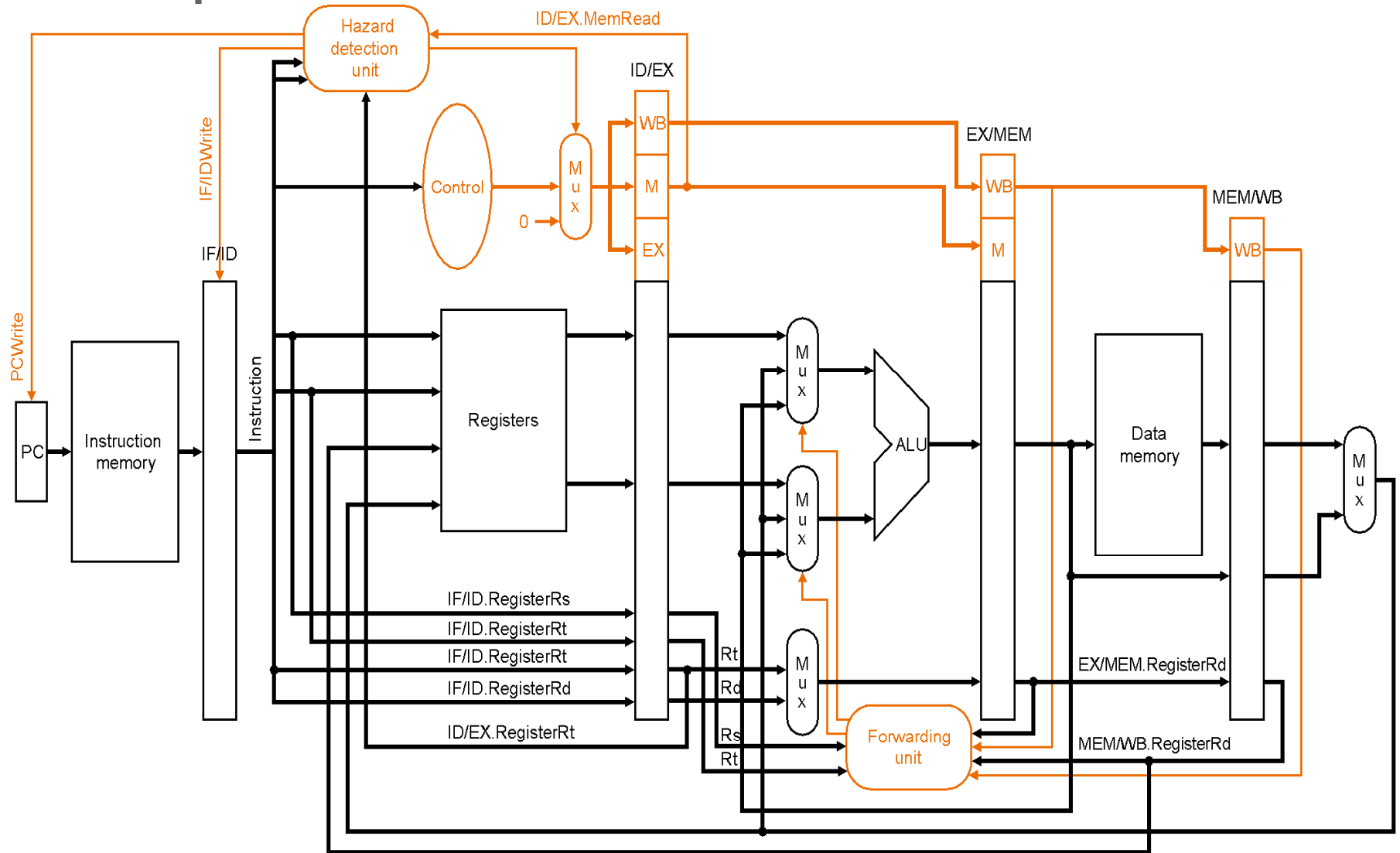
- The instructions in the IF and ID stages must be stalled; (i.e. not allowed to make progress)
- IF Stage
  - PC and IF / ID registers have write enable inputs
  - Prevent the PC register and the IF / ID pipeline register from changing (i.e. set their write enable inputs to 0).
  - The instruction in the IF stage continue to read instructions using the same PC.



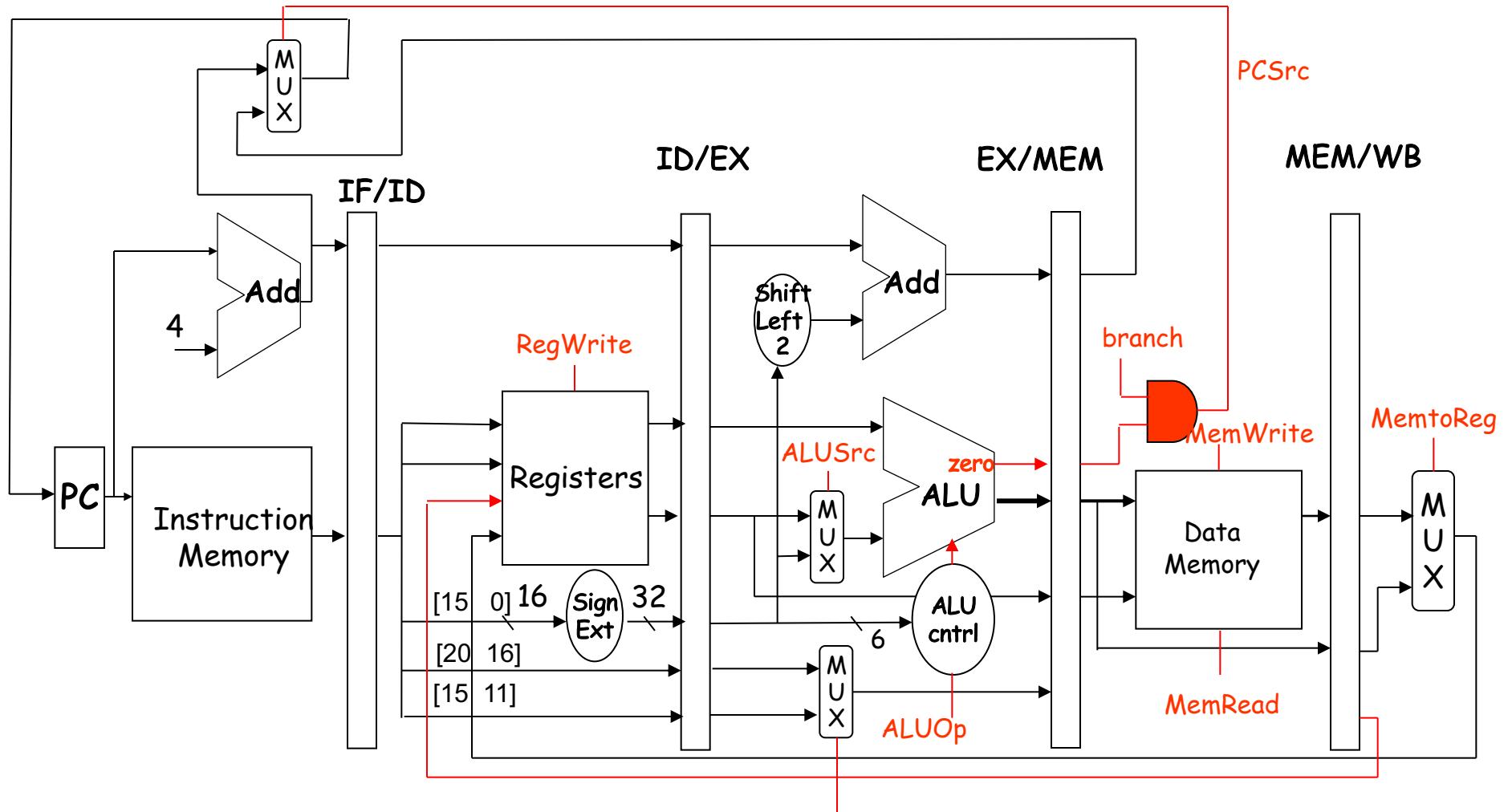
# How to Stall the Pipeline

- ID Stage
  - Since the data in the IF / ID register doesn't change, the same instruction is decoded
  - The registers in the ID stage will continue to be read using the same instruction fields in the IF / ID pipeline register.
- Hazard is detected in the ID stage, thus we insert a bubble in the EX stage.
- Inserting a bubble means
  - deasserting all the control signals in the ID stage in order to prevent the bubble writing into anything.
  - Recall that control signals are generated in this stage

# Pipelined Control

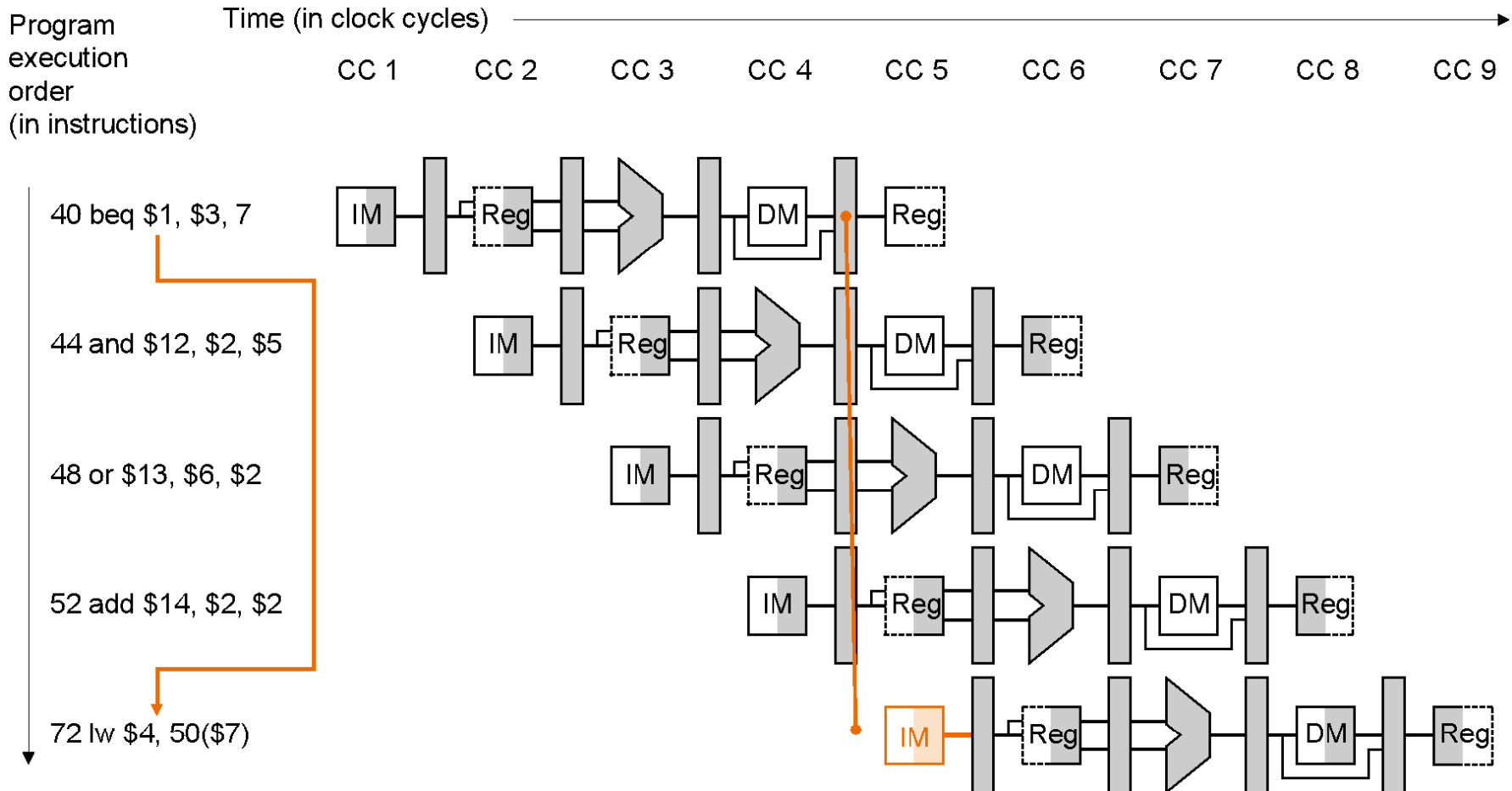


# Pipelined Datapath with Control



# Branch (or Control) Hazards

- When the branch is decided to be taken, successive instructions following branch are already in the pipeline.



# Branch (or Control) Hazards

- Note that the branch instruction decides whether to branch in the MEM stage
  - different than multicycle desing
- Control hazards are relatively simpler to understand than data hazards
- They occur less frequently.
- There is nothing as effective against control hazards as forwarding is for data hazards.
- Simpler schemes to overcome misprediction or ease of penalties.

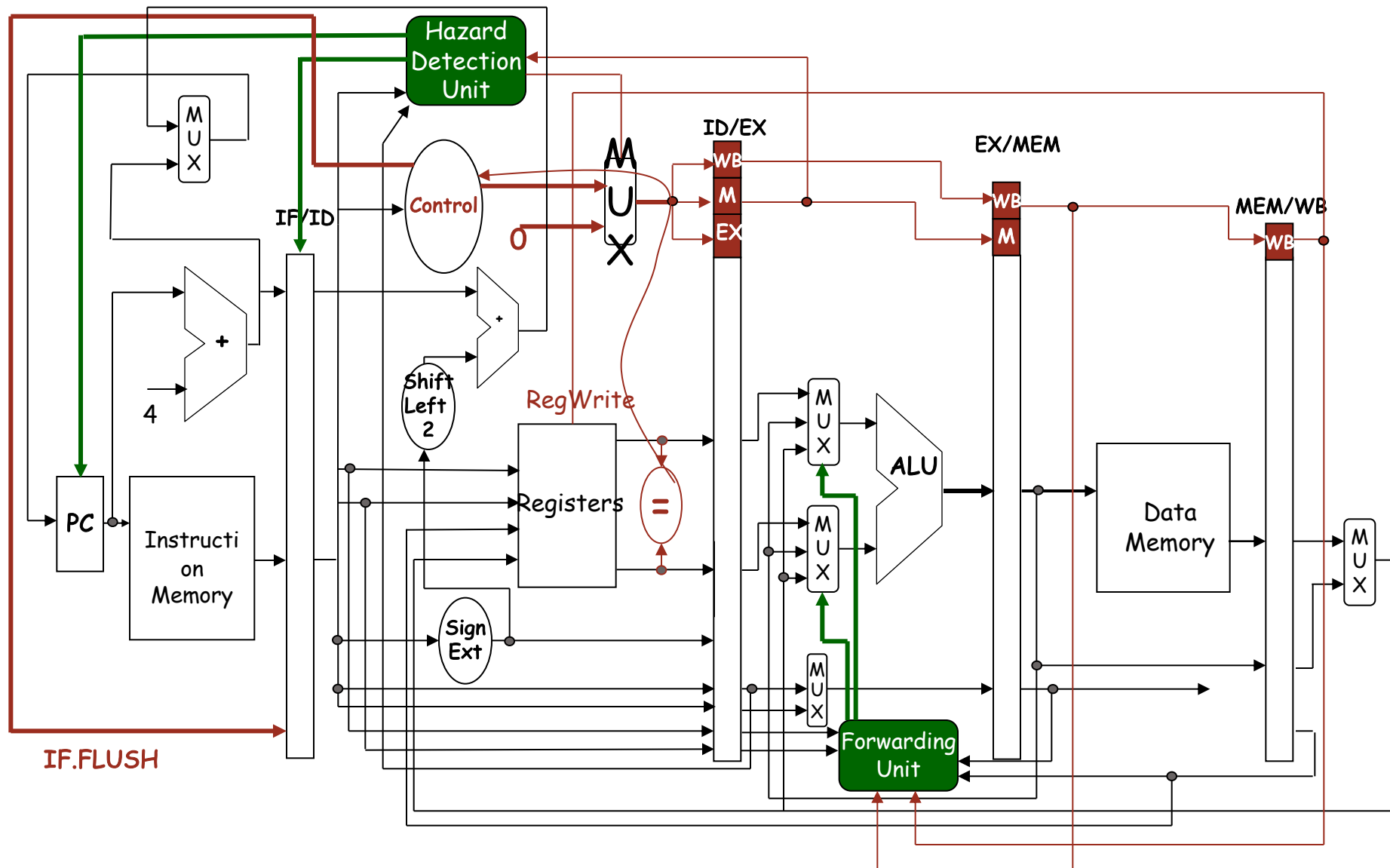
# Assume Branch Not Taken

- “Stall until the branch is complete” is too slow.
- Alternative is to assume that the branch will not be taken
- If the branch is taken, the instructions that are being fetched and decoded must be discarded.
- If half the time the branches are taken, this optimization halves the cost of control hazards.
- To discard instructions, we merely change the original control values to 0s.
- Discarding instructions means we must be able to flush instructions in the IF, ID, and EX stages.

# Reducing Branch Penalty

- Moving branch execution earlier in the pipeline
- If the next PC for a branch is selected at the end of EX stage, then the branch penalty becomes only two clock cycles.
- Many MIPS implementation move the branch execution to the ID stage.
- Address calculation can easily be moved to ID stage since the PC and the immediate field are in the IF/ID register.
- Comparing can be done faster using XOR and OR gates than using subtraction

# Flushing @ Branch Misprediction



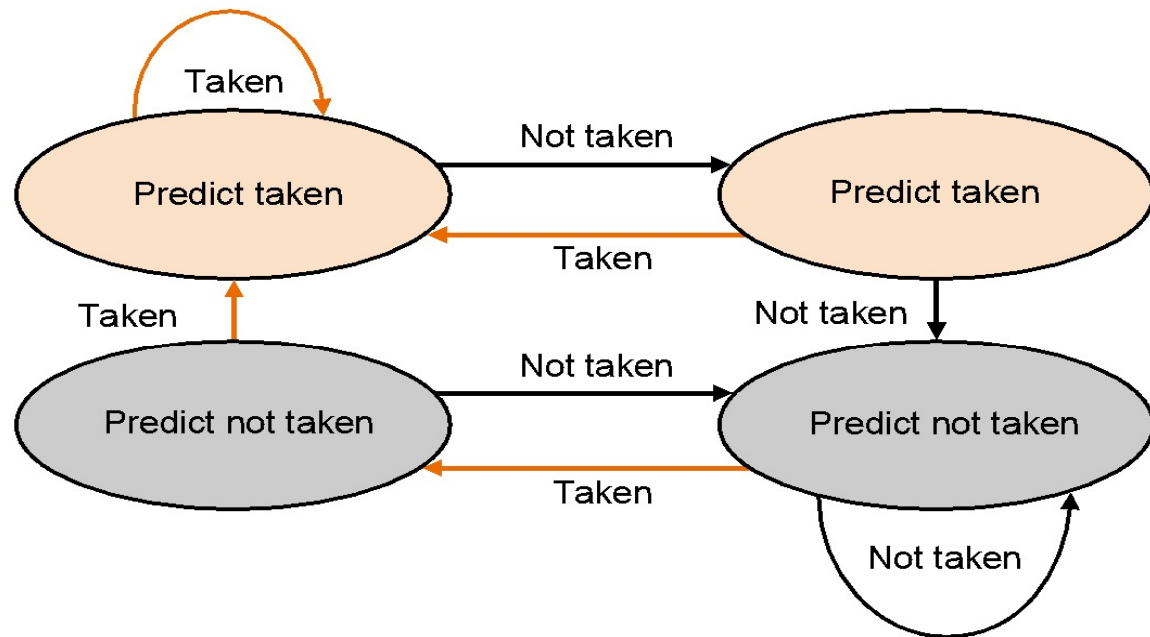


# Dynamic Branch Prediction

- Branch prediction buffer or branch history table contains a bit for each branch about its past behavior.
  - The buffer is indexed by the lower portion of the address of the instruction.
  - branch target buffer
- 1-bit prediction scheme has performance shortcomings.
  - Consider a loop branch that branches nine times in a row, then it is not taken once.
  - The accuracy of this prediction scheme is?

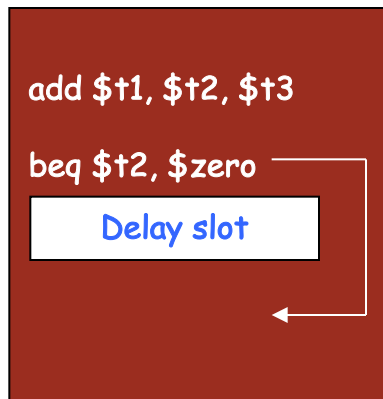
# 2-bit Prediction Scheme

- In 2-bit prediction scheme, a prediction must be wrong twice before it is changed.

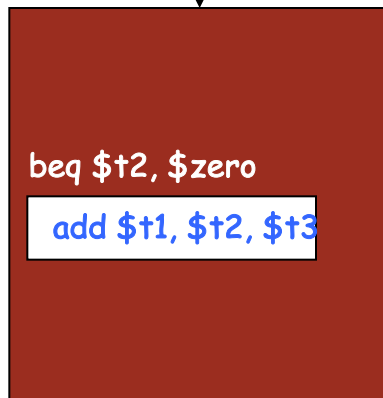


# Delayed-Branch Scheduling

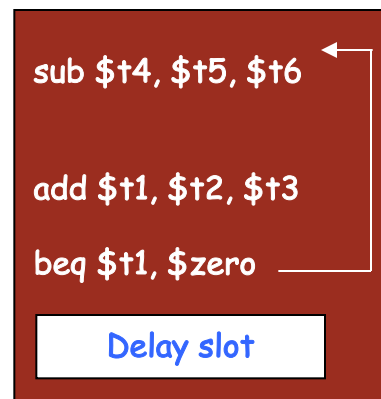
(a) From before



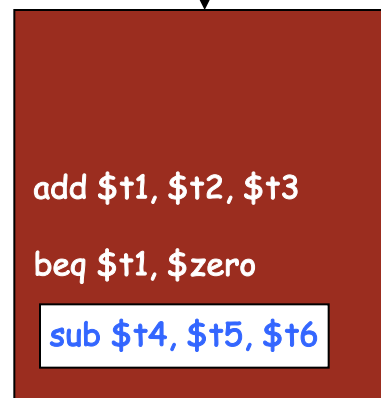
becomes



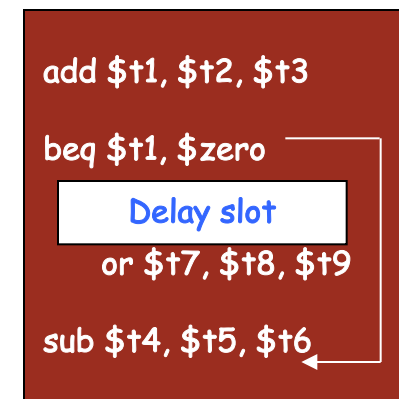
(b) From target



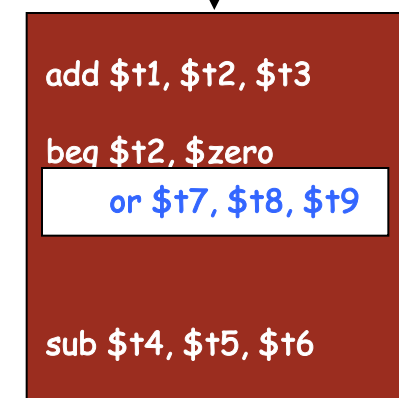
becomes



(c) From fall through



becomes



# Visiting the Old Example 1/2

- Delays of functional units
  - Memory access: 200 ps,
  - ALU op: 100 ps,
  - Register access: 50 ps.
- Single cycle datapath
  - $200 + 50 + 100 + 200 + 50 = 600$  ps
- SPECInt2000
  - Loads: 25%; Stores: 10%; Branch: 11%; Jumps: 2%; ALU Operations: 52%
- Multi cycle data path
  - $\text{CPI} = 5 \times 25\% + 4 \times 10\% + 3 \times 11\% + 3 \times 2\% + 4 \times 52\% = 4.12.$
  - execution time per instruction:  $? \times ? = ?$  ps

# Visiting the Old Example 2/2

- Assumptions:
  - 50% of loads are immediately followed by an instruction that uses the result of load
  - Branch penalty on misprediction: one clock cycle.
  - 25% of branches are mispredicted.
  - Jumps: 2 clock cycles
- Pipelined datapath
  - $\text{CPI} = 1.5 \times 25\% + 1 \times 10\% + 1.25 \times 11\% + 2 \times 2\% + 1 \times 52\% = 1.1725$
  - Average time per instruction:
    - $1.1725 \times 200 \text{ ps} = 234.5 \text{ ps}$

# Exception in the Pipelined Datapath

- |      |     |       |          |     |
|------|-----|-------|----------|-----|
| 0x40 | sub | \$11, | \$2,     | \$4 |
| 0x44 | and | \$12, | \$2,     | \$5 |
| 0x48 | or  | \$13, | \$2,     | \$6 |
| 0x4C | add | \$1,  | \$2,     | \$1 |
| 0x50 | slt | \$15, | \$6,     | \$7 |
| 0x54 | lw  | \$16, | 50 (\$7) |     |
- Suppose an overflow exception occurs in the add instruction.
- Exception handling routine

0x40000040	sw	\$25,	1000 (\$0)
0x40000044	sw	\$26,	1004 (\$0)
...			

# Example: clock 6

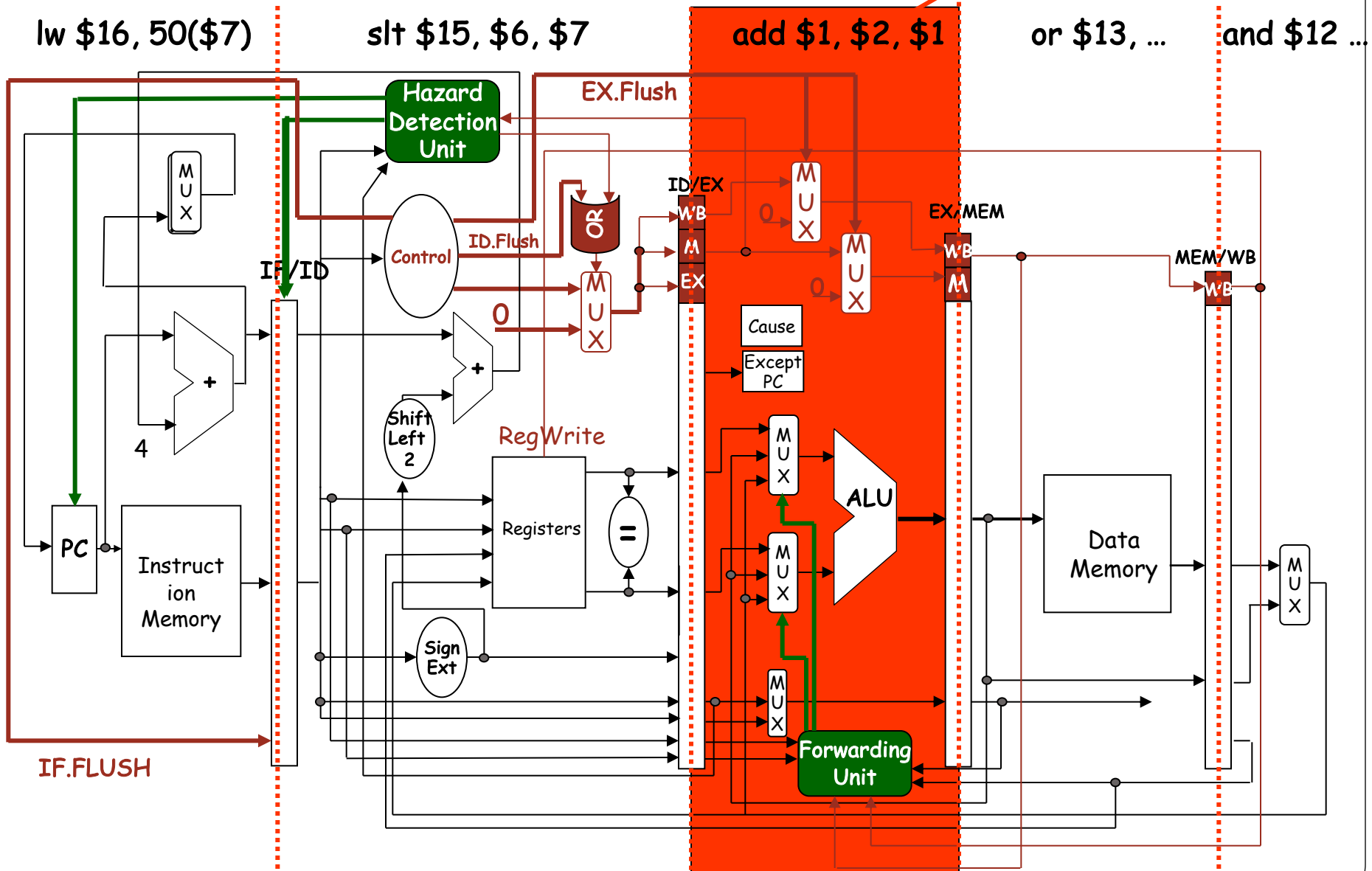
lw \$16, 50(\$7)

slt \$15, \$6, \$7

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

or \$13, ...

and \$12 ...



# Example: clock 7

