Computer Organization
計算機組織

Pipelining

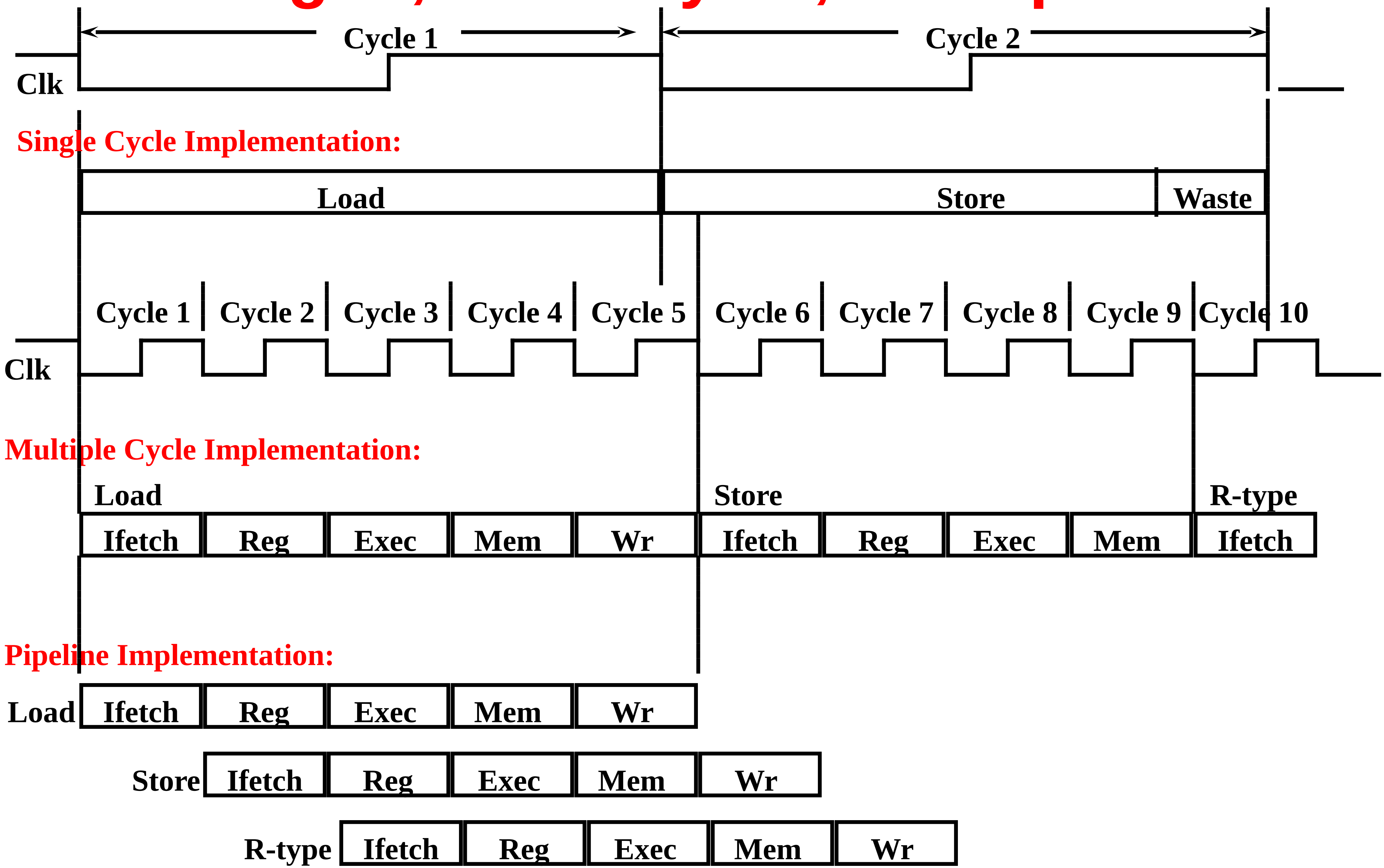
國立成功大學資訊工程學系
105 年度第二學期

Outline

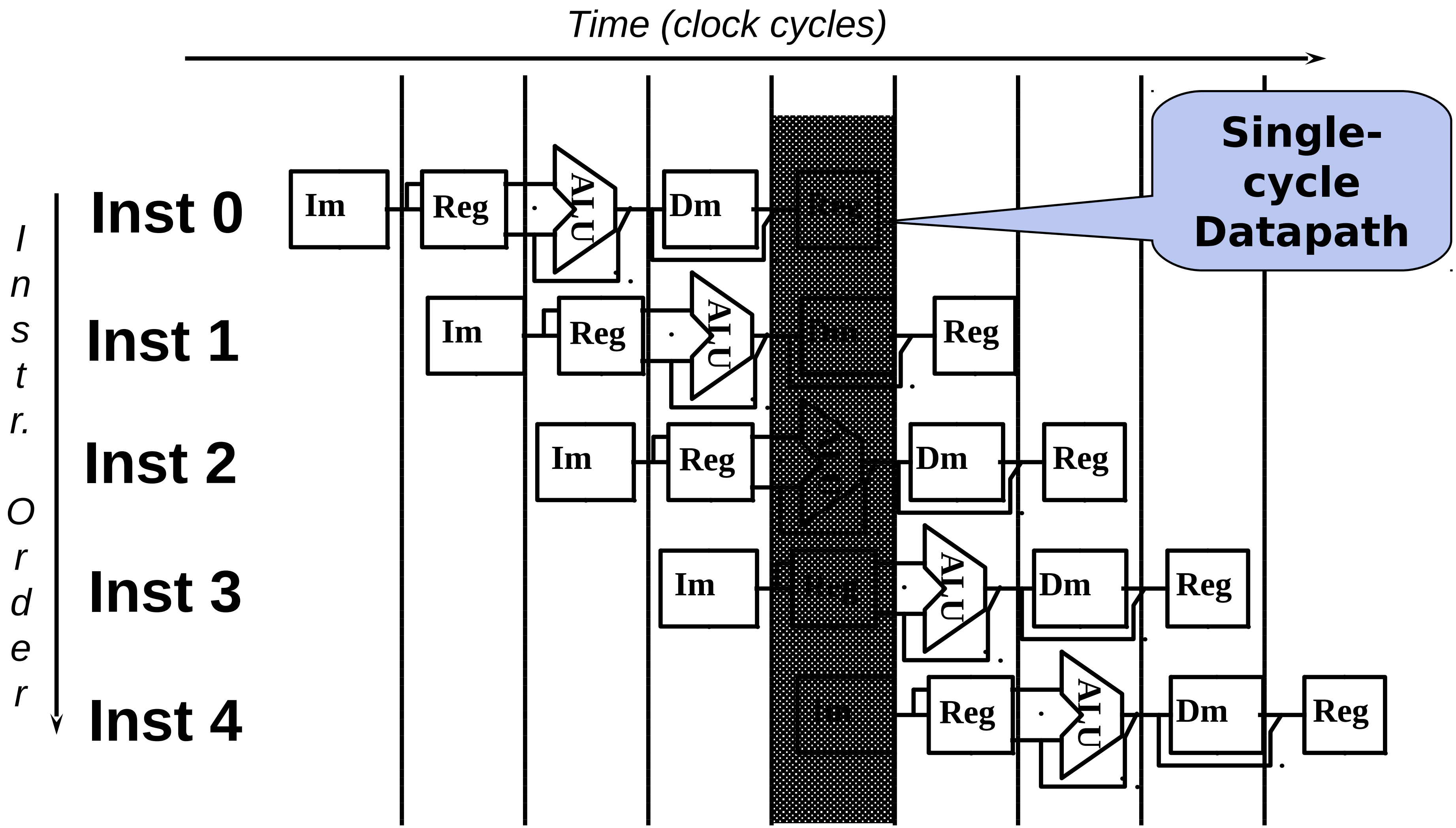
- **Pipelined processor design**
  - **An overview of pipelining**
  - **A pipelined datapath**
  - **Pipelined control**

- **Problems with pipelined processor**
  - **Data hazards and forwarding**
  - **Data hazards and stalls**
  - **Branch hazards**

Single-, Multi-Cycle, vs. Pipeline



Why Pipeline? Because the Resources are There!



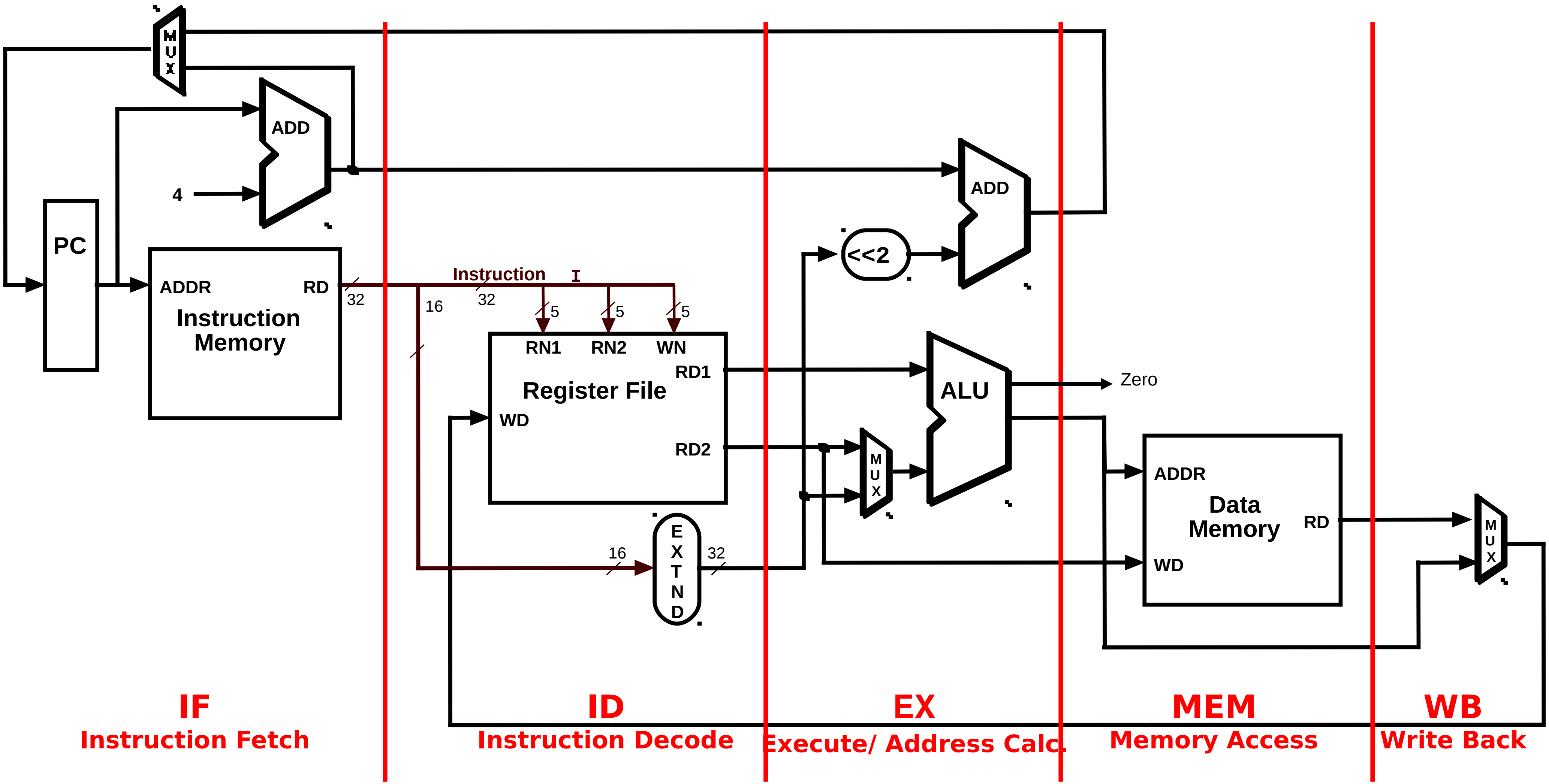
Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

Recall the 5 steps in Instruction Execution

1. **Instruction Fetch & PC Increment (IF)**
2. **Instruction Decode and Register Read (ID)**
3. **Execution or calculate address (EX)**
4. **Memory access (MEM)**
5. **Write result into register (WB)**

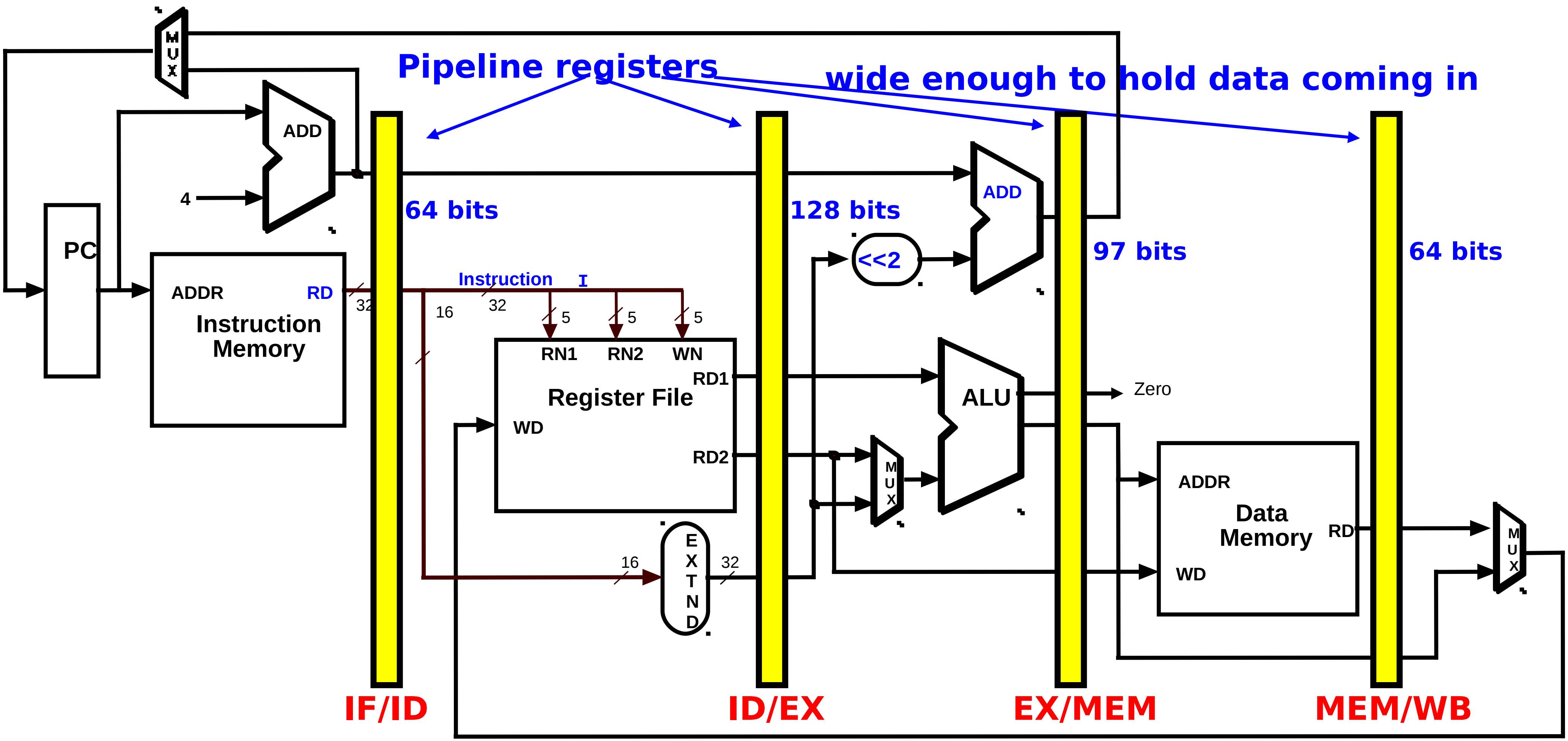
Recall “Single-Cycle” Datapath



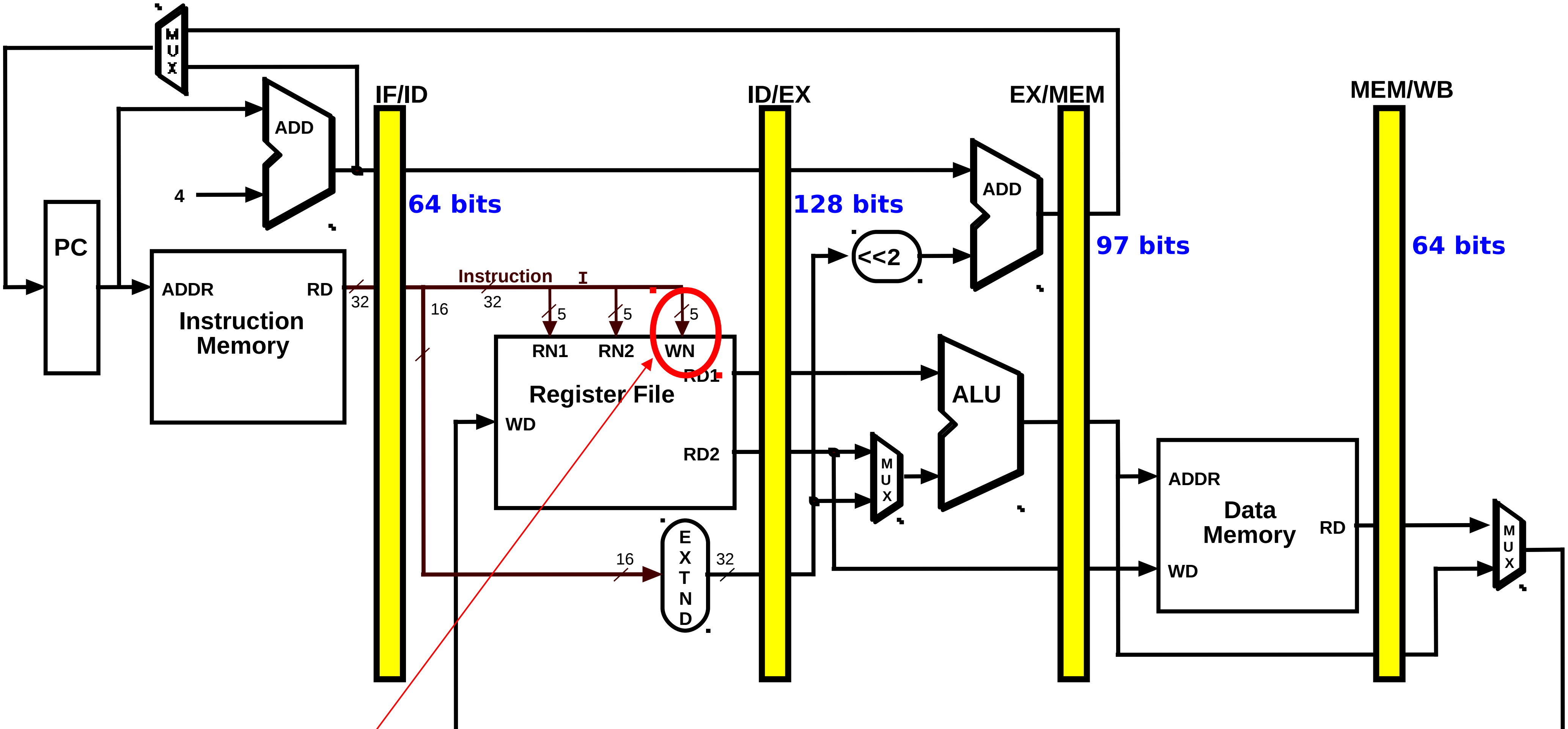
Pipelined Datapath – Key Idea

- **We will have several instructions that execute simultaneously in the processor**
- **Similar to multicycle design**
  - **Revising the single-cycle design by introducing *extra* registers---*pipeline registers*---to hold "relevant state" between cycles**

Pipelined Datapath

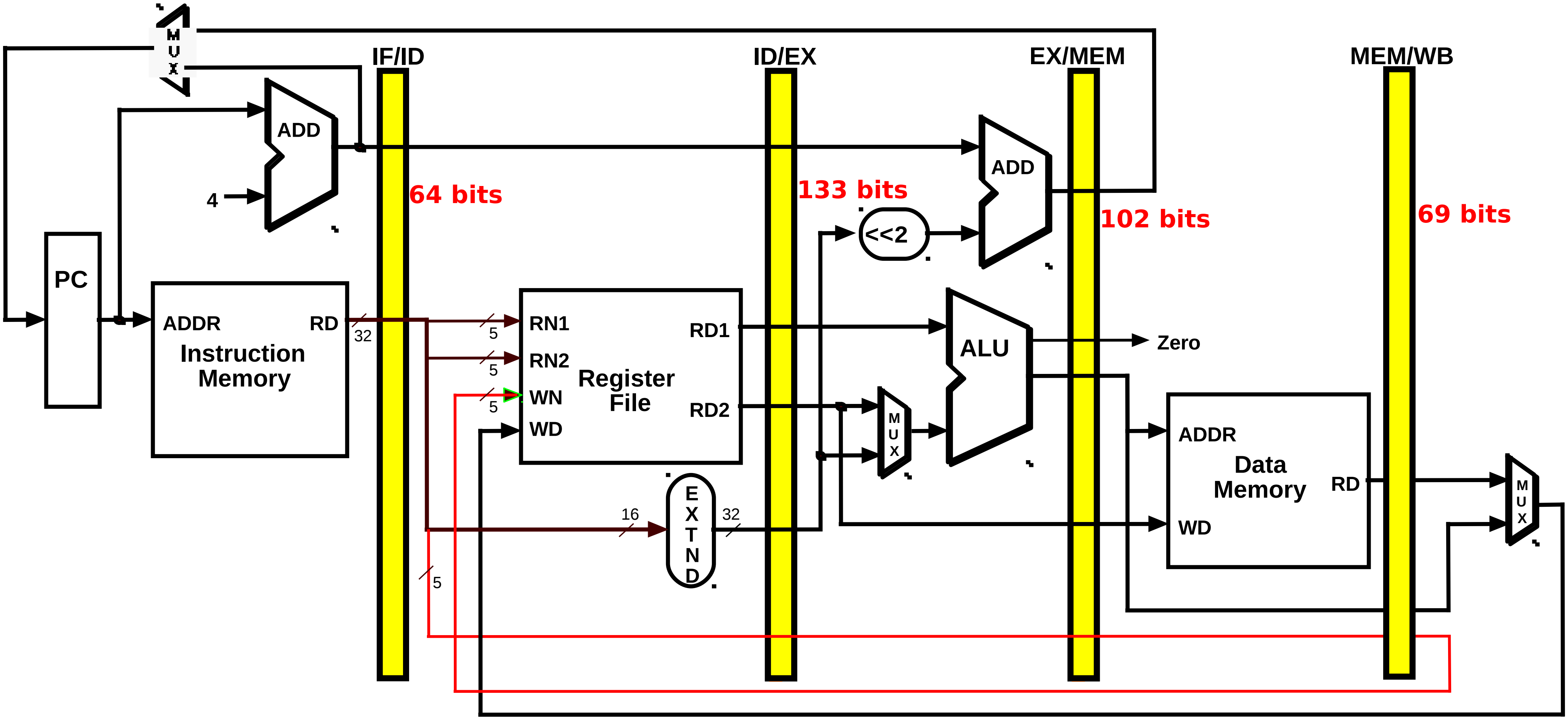


Bug in the Datapath



Write register number comes from another instruction in the latter stage!

Corrected Datapath



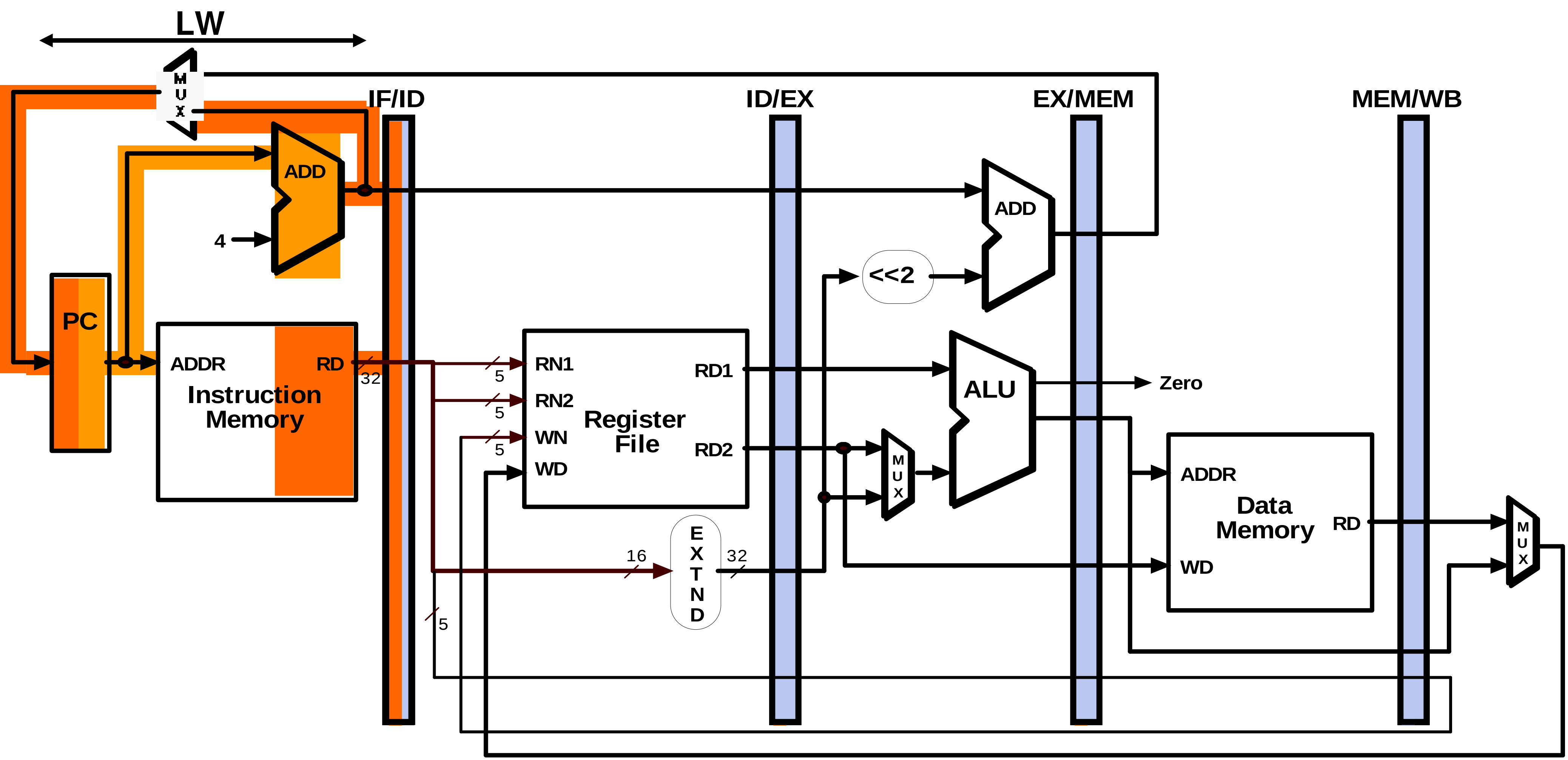
Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

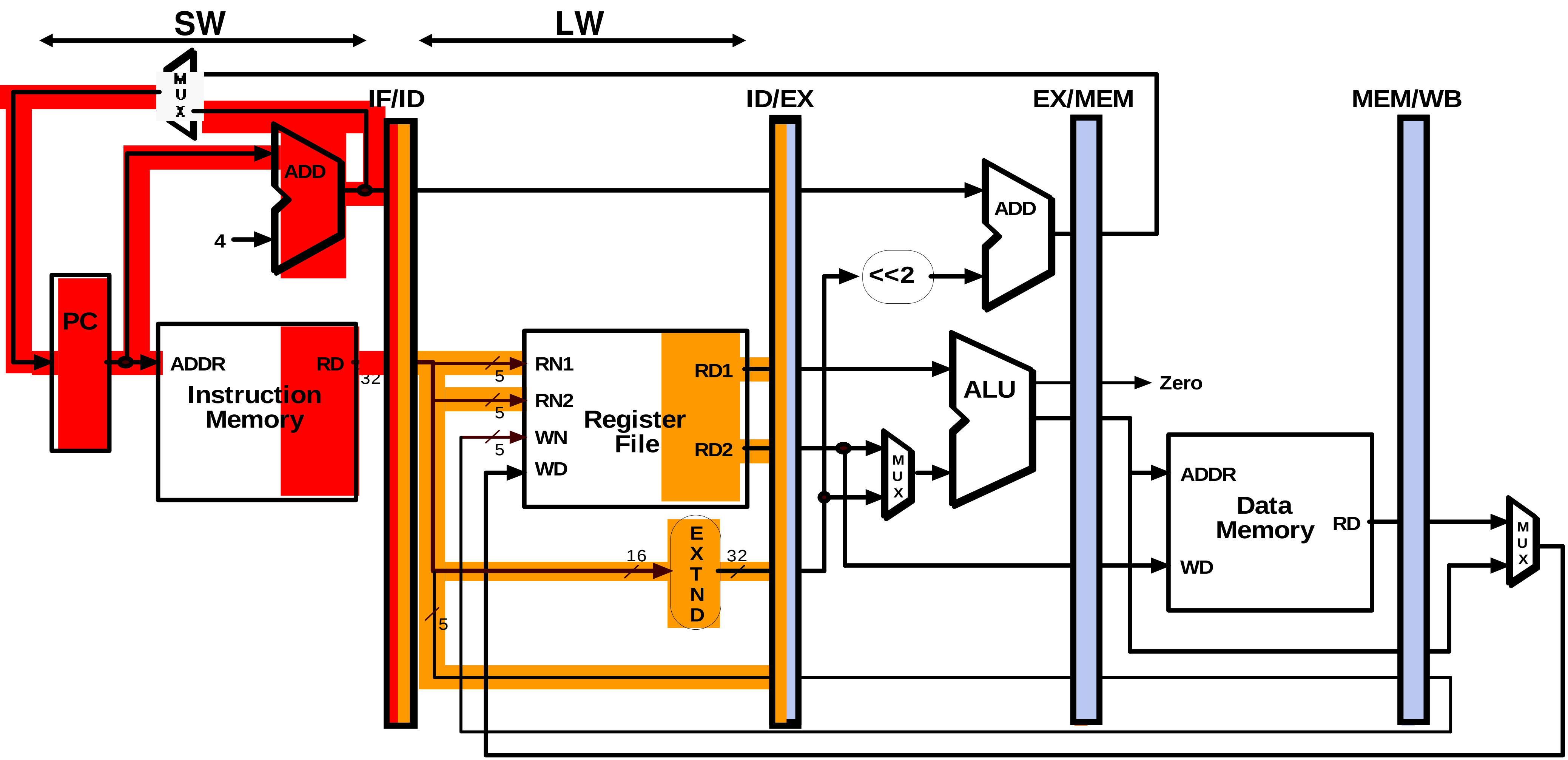
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)

sw  $t3, 20($t4)

add $t5,  $t6,  $t7

sub $t8,  $t9,  $t10
```

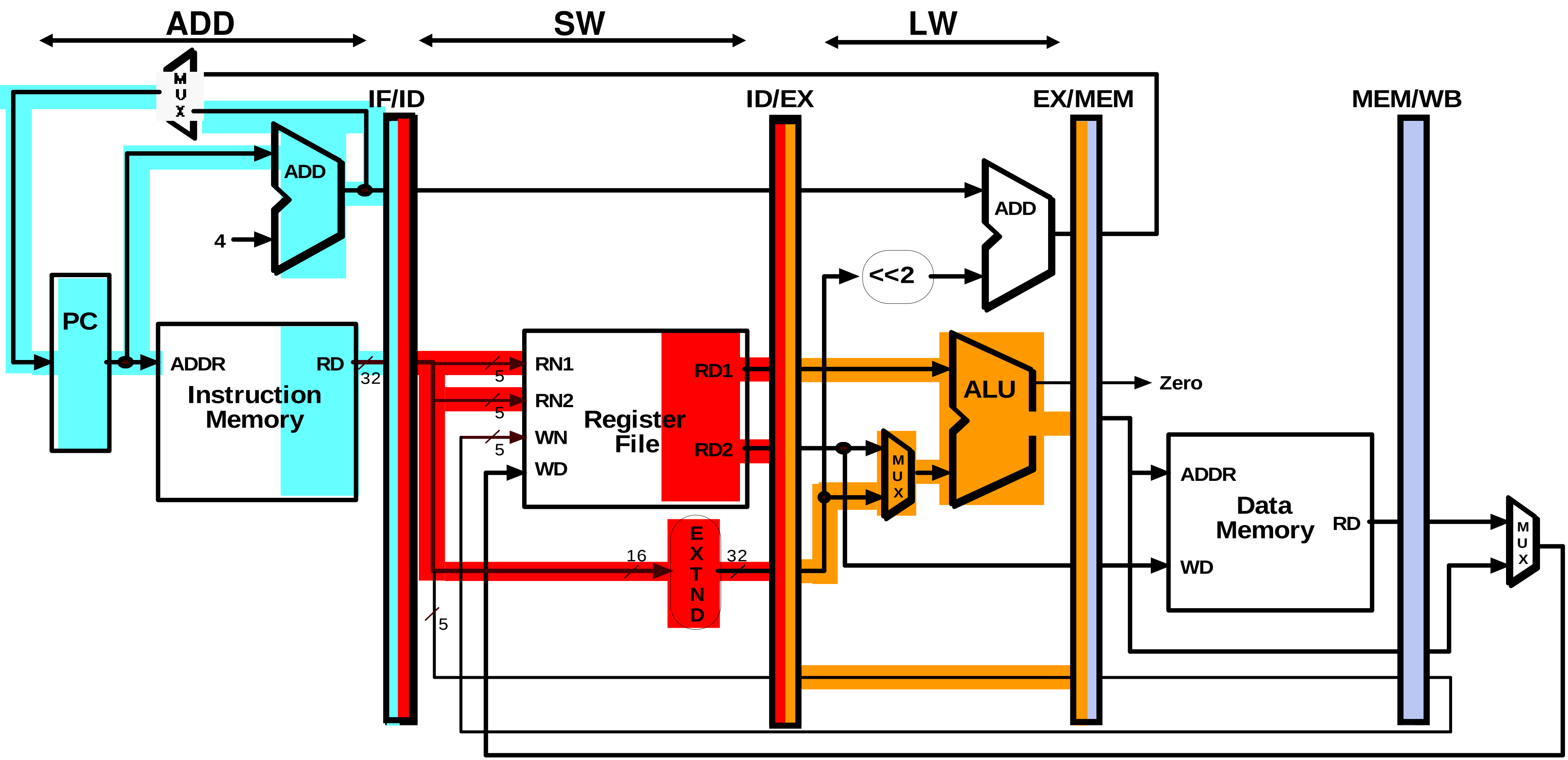
Single-Clock-Cycle Diagram: Clock Cycle 1



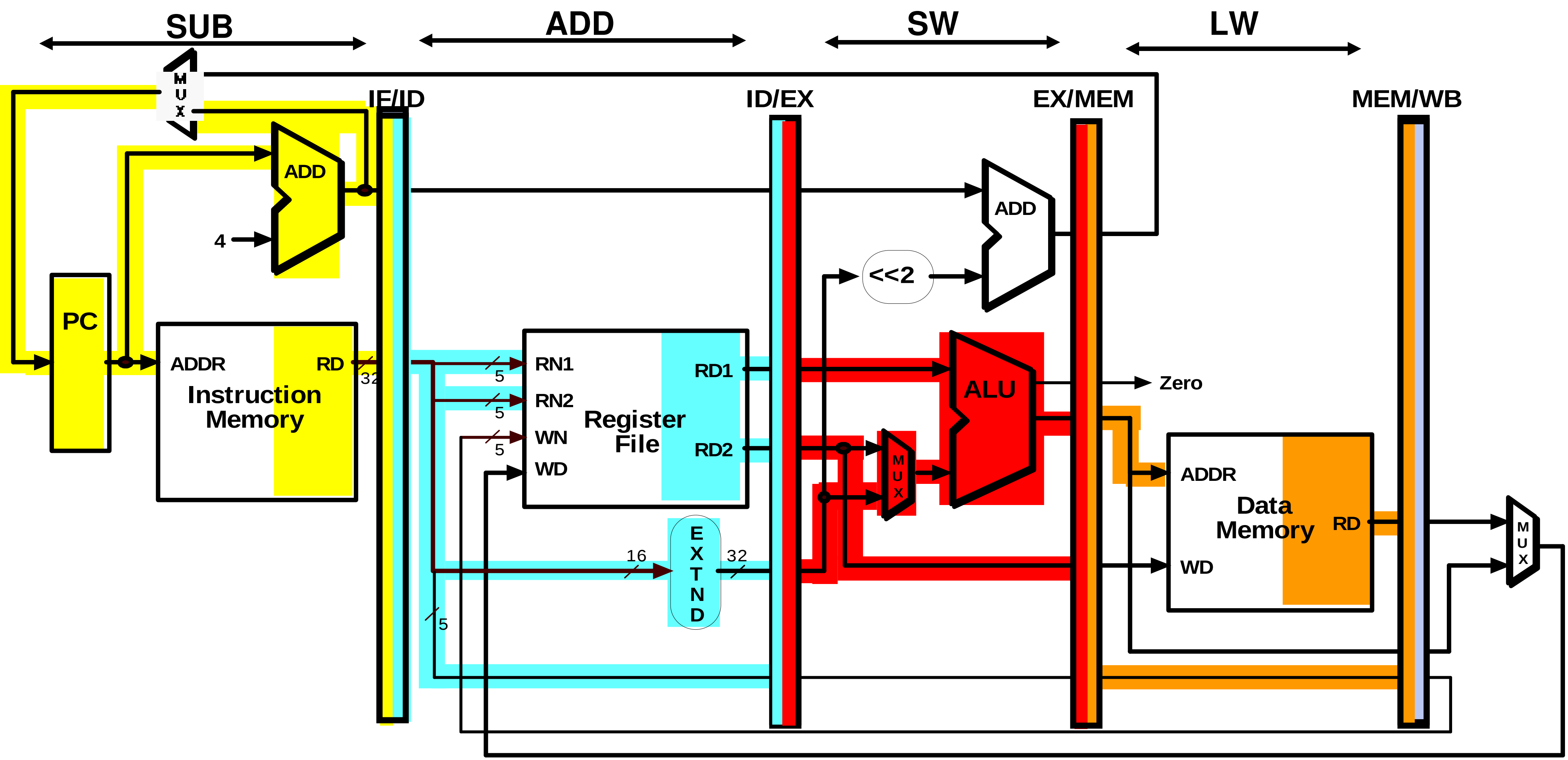
Single-Clock-Cycle Diagram: Clock Cycle 2



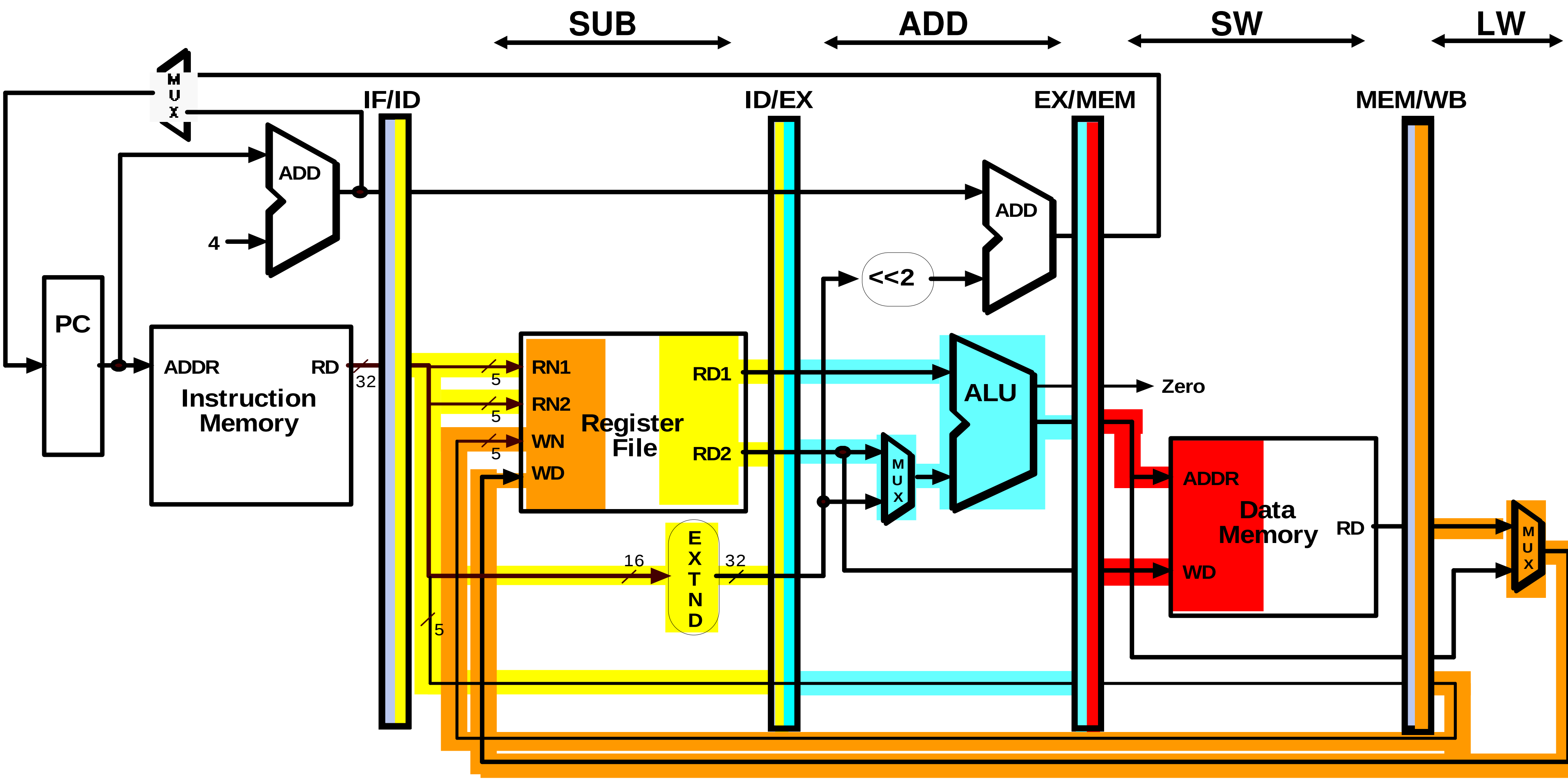
Single-Clock-Cycle Diagram: Clock Cycle 3



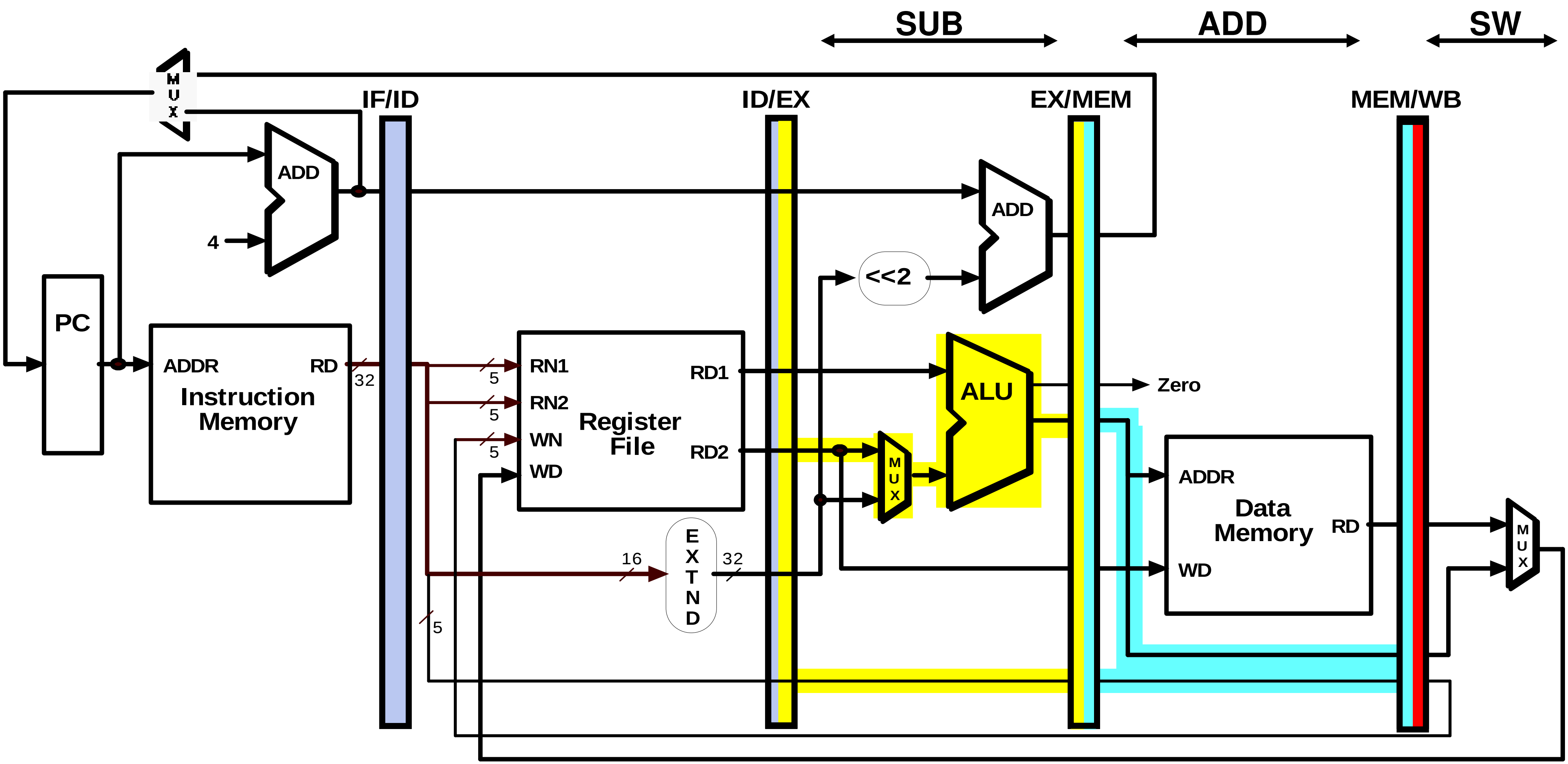
Single-Clock-Cycle Diagram: Clock Cycle 4



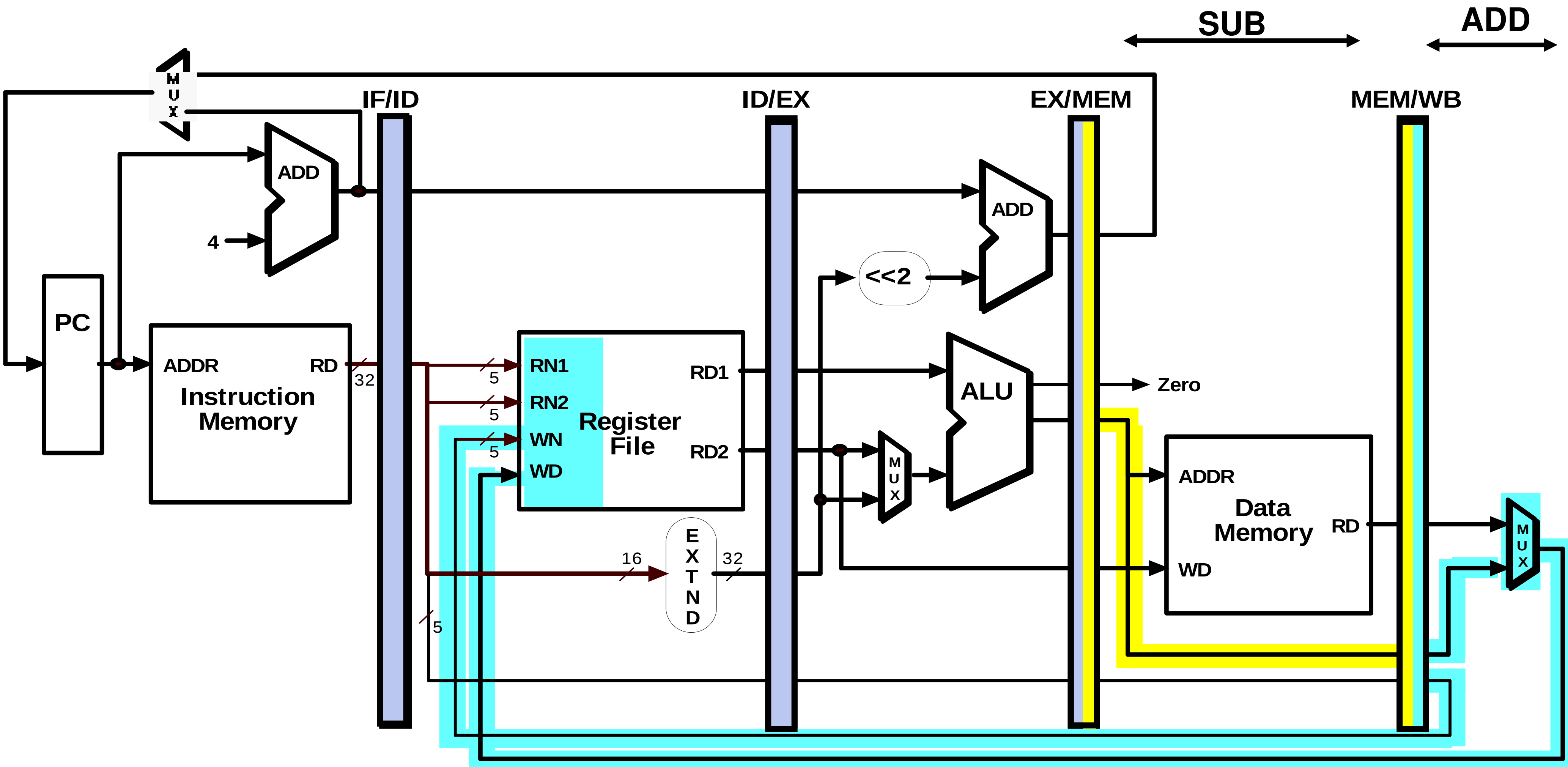
Single-Clock-Cycle Diagram: Clock Cycle 5



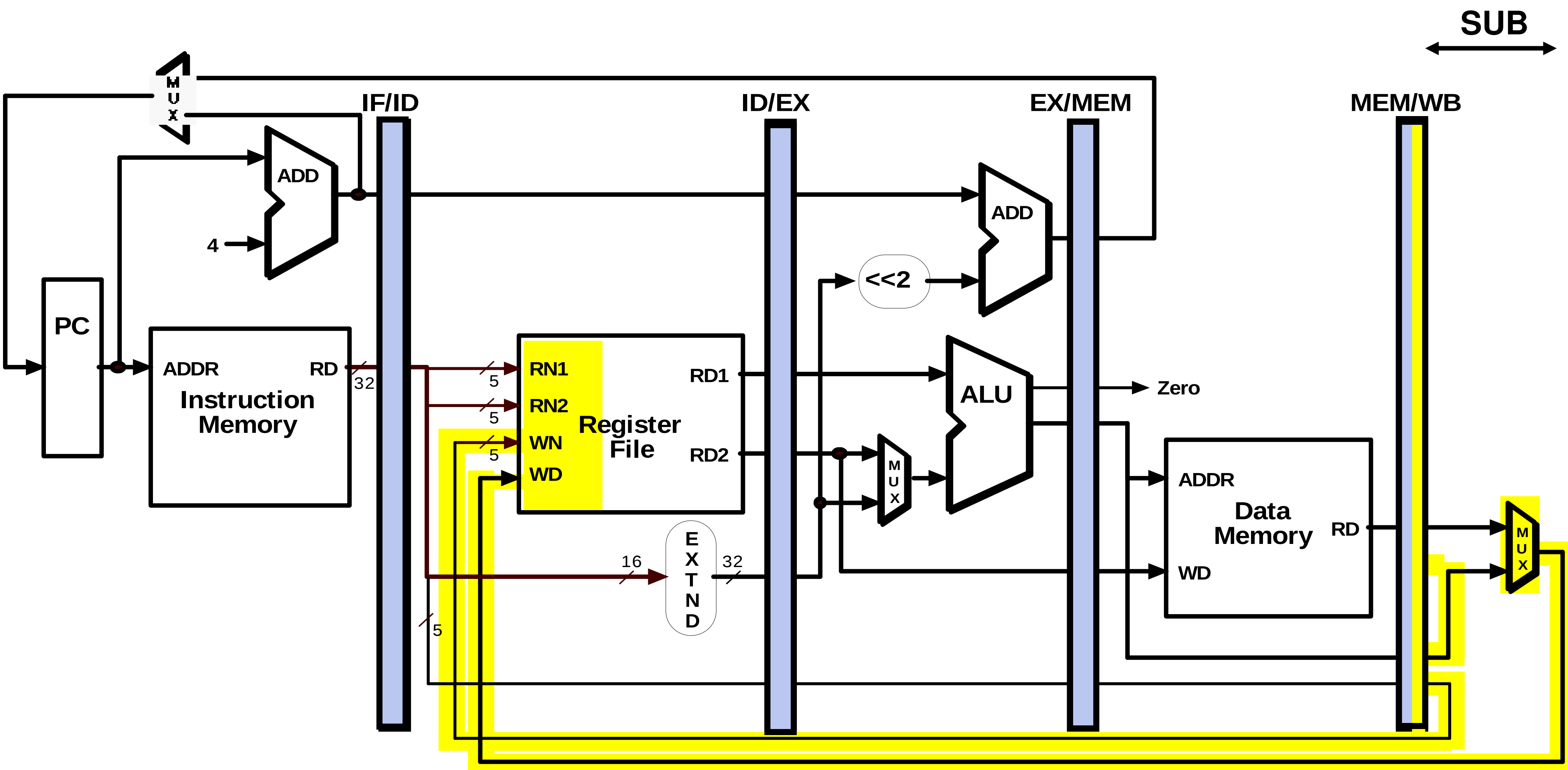
Single-Clock-Cycle Diagram: Clock Cycle 6



Single-Clock-Cycle Diagram: Clock Cycle 7



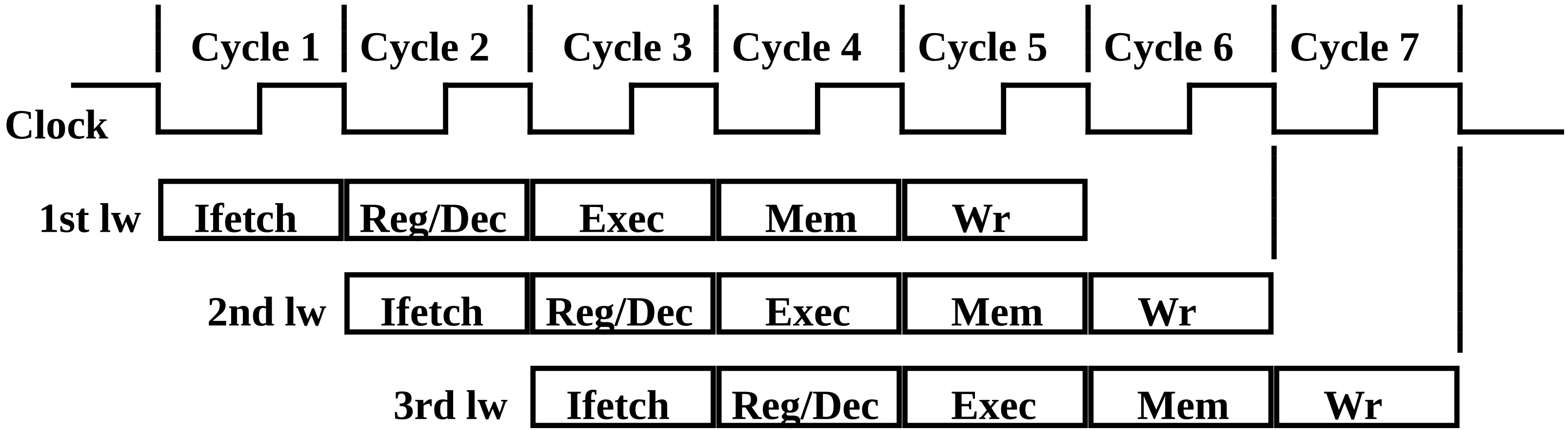
Single-Clock-Cycle Diagram: Clock Cycle 8



Observation

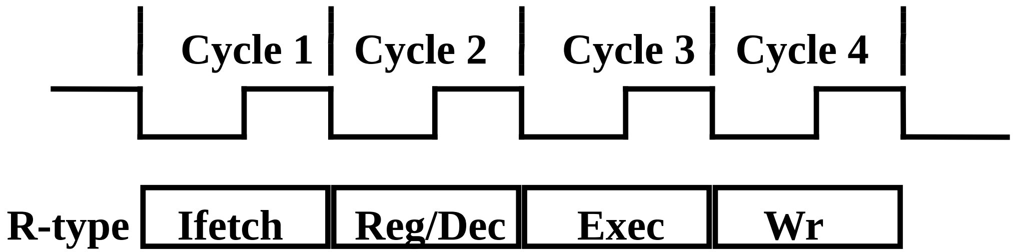
- The pipeline implementation inserts pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1.
  - *What is the CPI value of multicycle design?*
    - *(discussed before)*
- One significant difference in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5th (the last write-back) pipeline stage vs. the 4th stage for the multicycle implementation. *Why?* (see the below)

Pipelining `load`



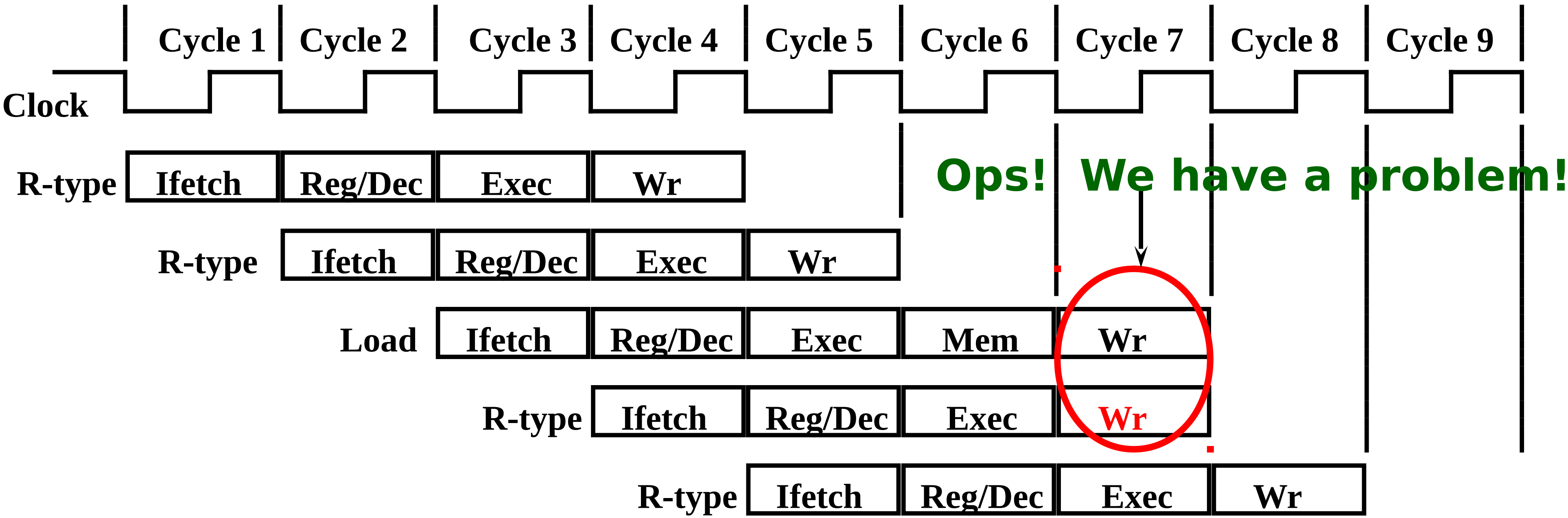
- **5 functional units in the pipeline datapath are:**
  - **Instruction Memory for the Fetch stage**
  - **Register File's Read ports (busA and busB) for the Reg/Dec stage**
  - **ALU for the Exec stage**
  - **Data Memory for the MEM stage**
  - **Register File's Write port (busW) for the WB stage**

Pipelinging R-type Instructions



- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

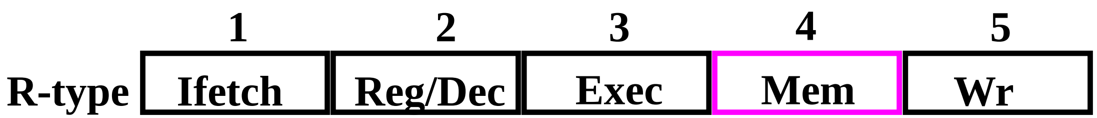
Pipelining R-type and `load`

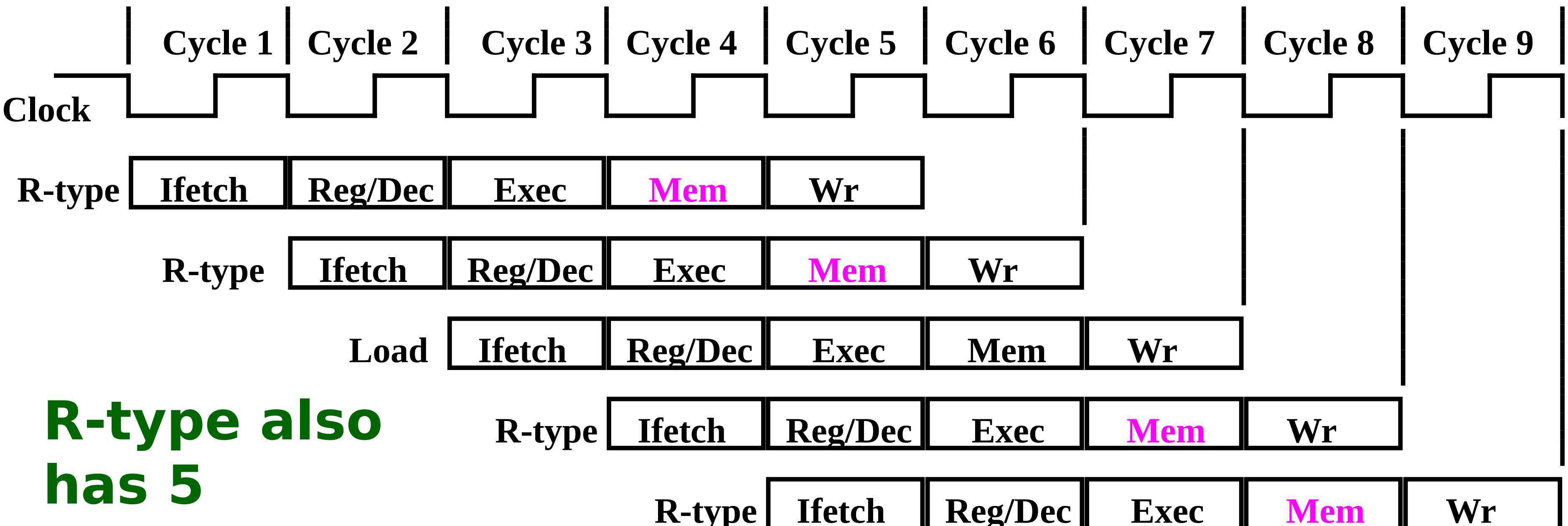


- We have a ***structural hazard***:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Solution: Delay R-type's Write

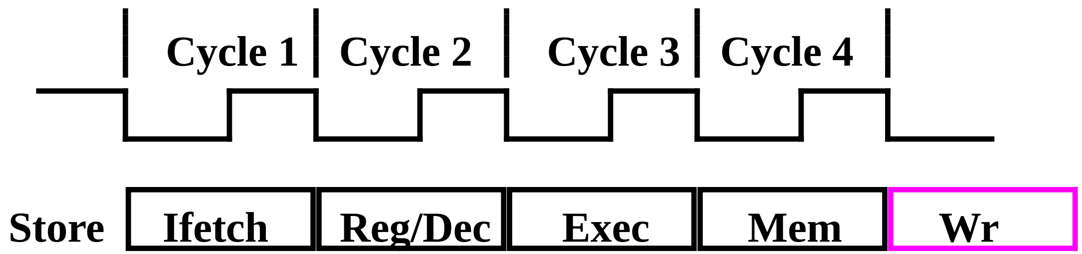
- Delay R-type's register write by one cycle:
  - R-type also use Reg File's write port at Stage 5
  - MEM is a NOP stage: nothing is being done.





R-type also has 5 stages

SW

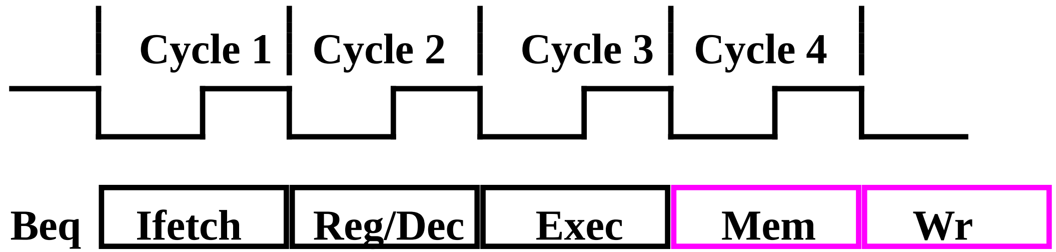


- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: calculate the memory address
- MEM: write the data into the Data Memory

Add an extra stage:

- WB: NOP

BEQ



- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX:
  - compare the two register operand
  - select correct branch target address
  - latch into PC

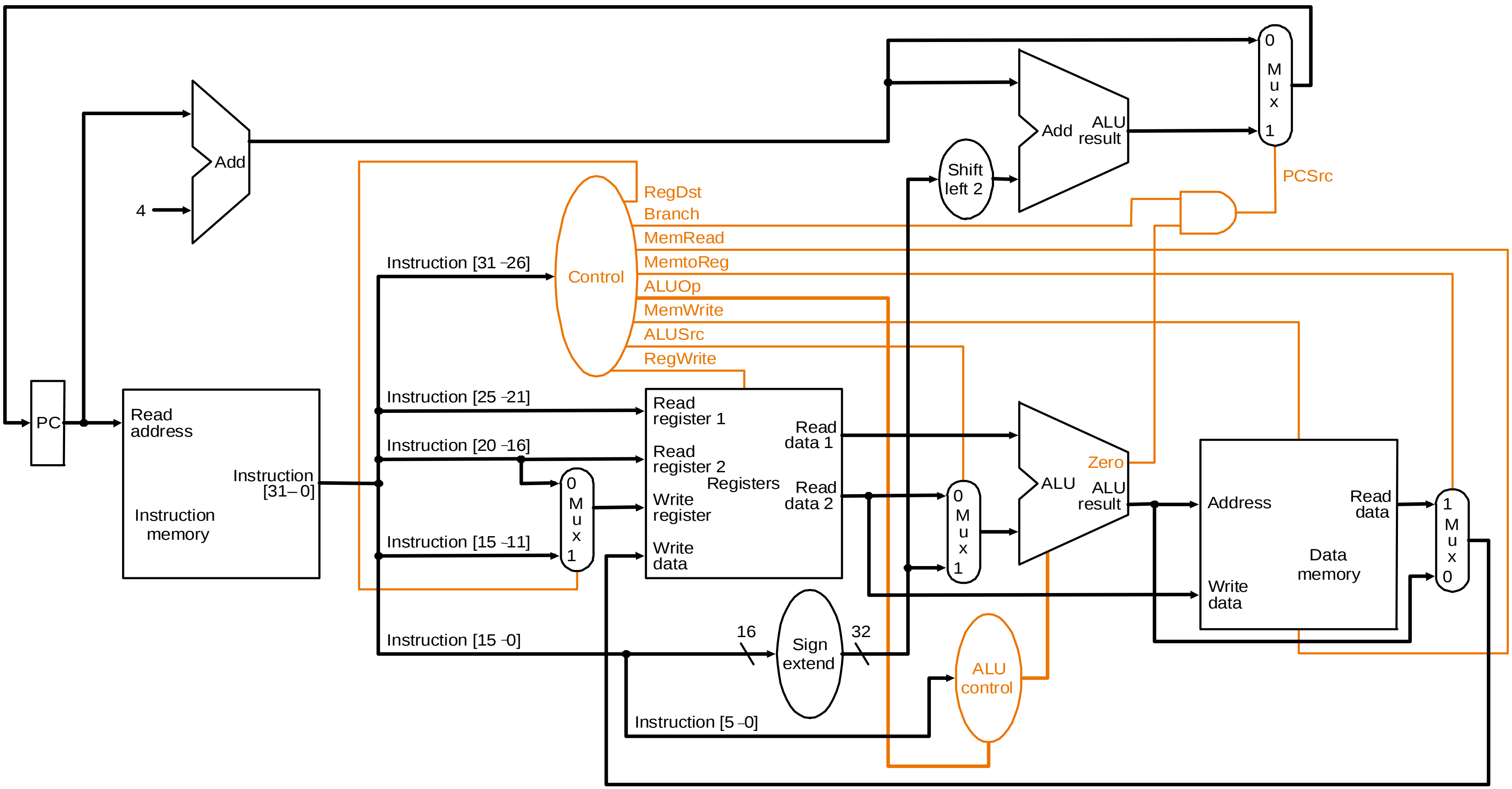
Add two extra stages:

- MEM: NOP
- WB: NOP

Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

Recall Single-Cycle Control – the Datapath



Recall Control for Single-Cycle Design

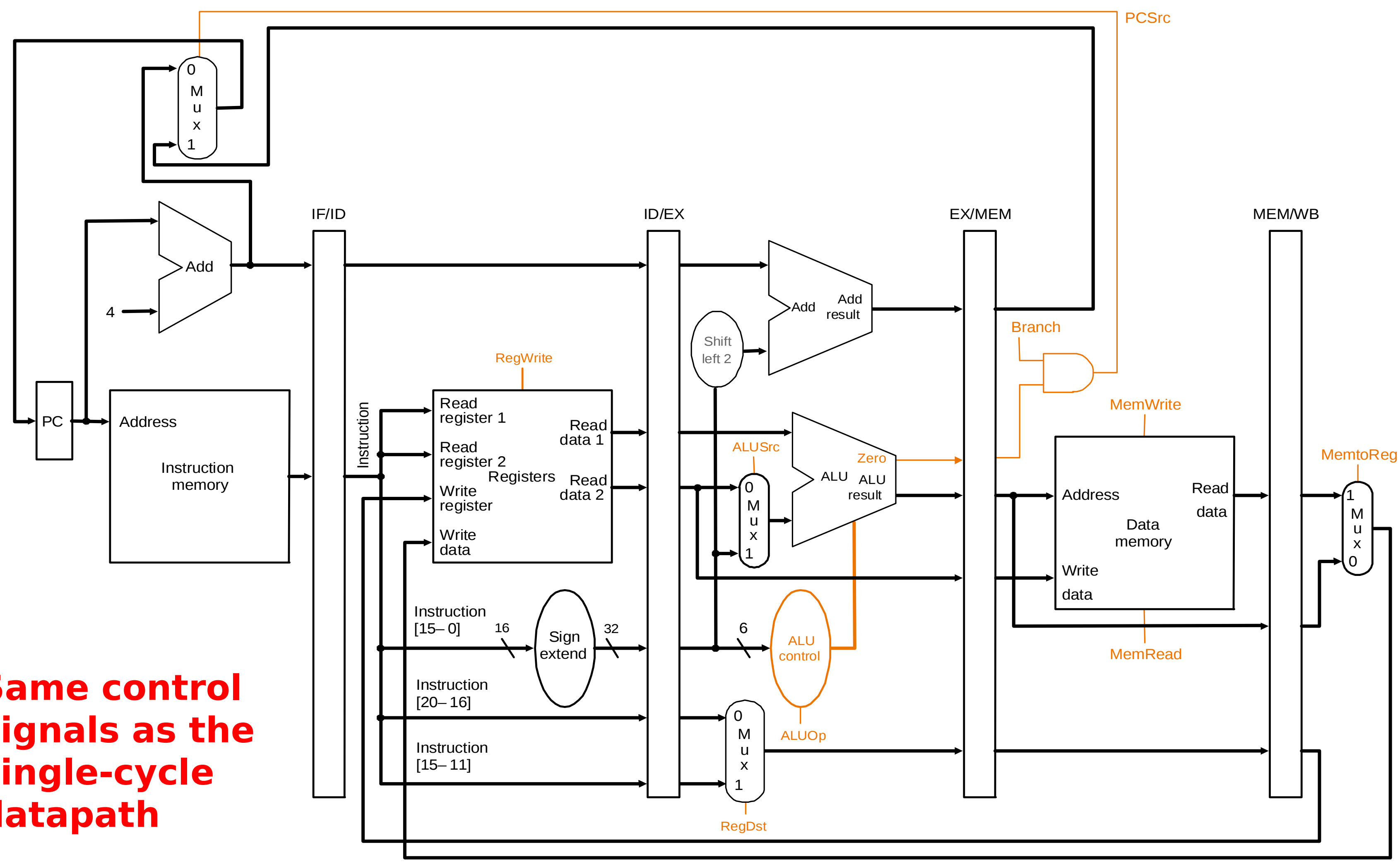
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
`lw`	0	1	1	1	1	0	0	0	0
`sw`	X	1	X	0	0	1	0	0	0
`beq`	X	0	X	0	0	0	1	0	1

Pipeline Control

- **Initial design – motivated by single-cycle datapath control – use the same control signals**
- **Since control signals are associated with components active during a single pipeline stage, can group control lines into five groups according to pipeline stage**
- **Need to set control signals during each pipeline stage**

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

- instruction fetch / PC increment
- instruction decode / register fetch
- execution / address calculation (EX)
- memory access (M)
- write back (WB)

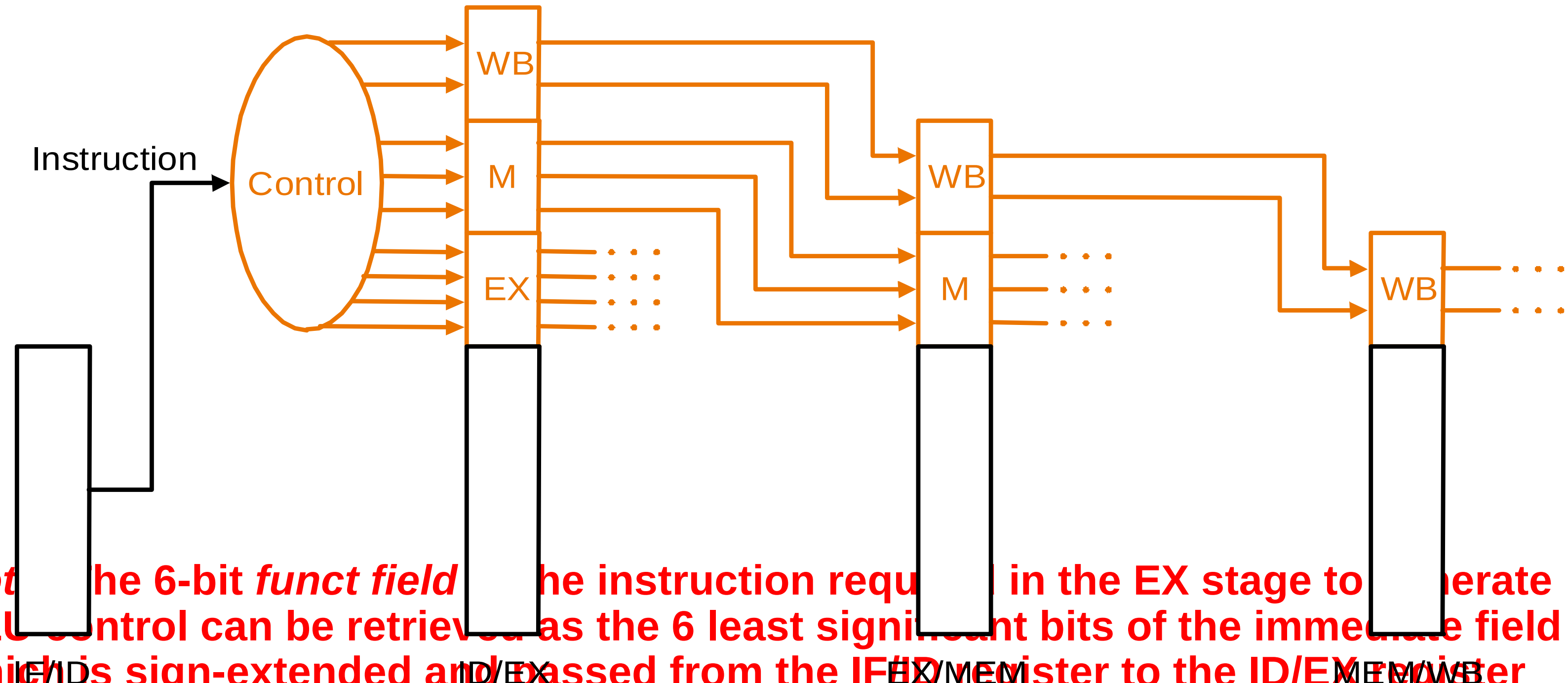
Nothing to control as instruction memory read and PC write are always enabled

(the following table)

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

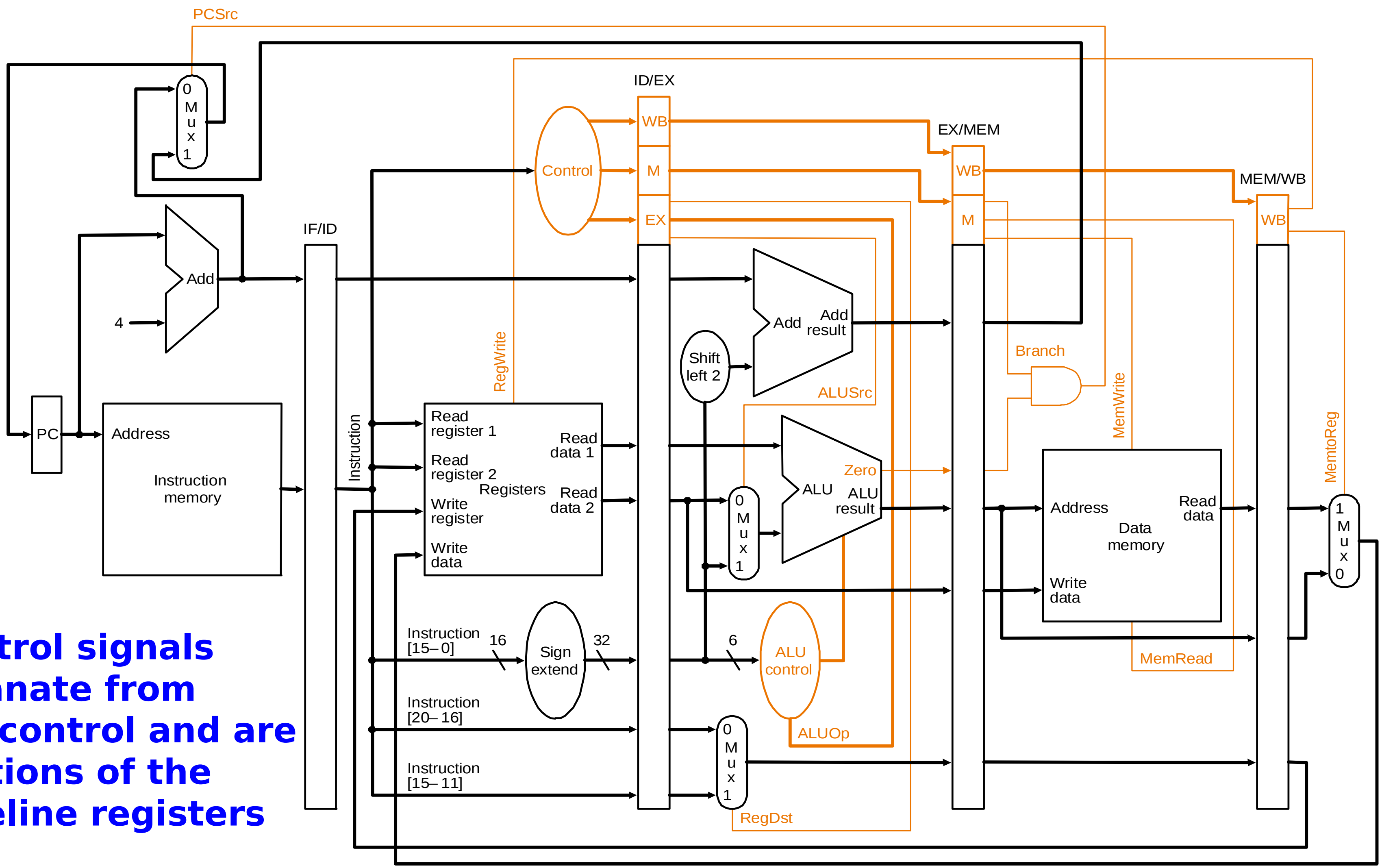
Pipeline Control Implementation

- ***Pass control signals along just like the data*** **– extend each pipeline register to hold needed control bits for succeeding stages**



- ***Note*** **that the 6-bit** ***funct field*** **of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register**

Pipelined Datapath with Control II



Control signals emanate from the control and are portions of the pipeline registers

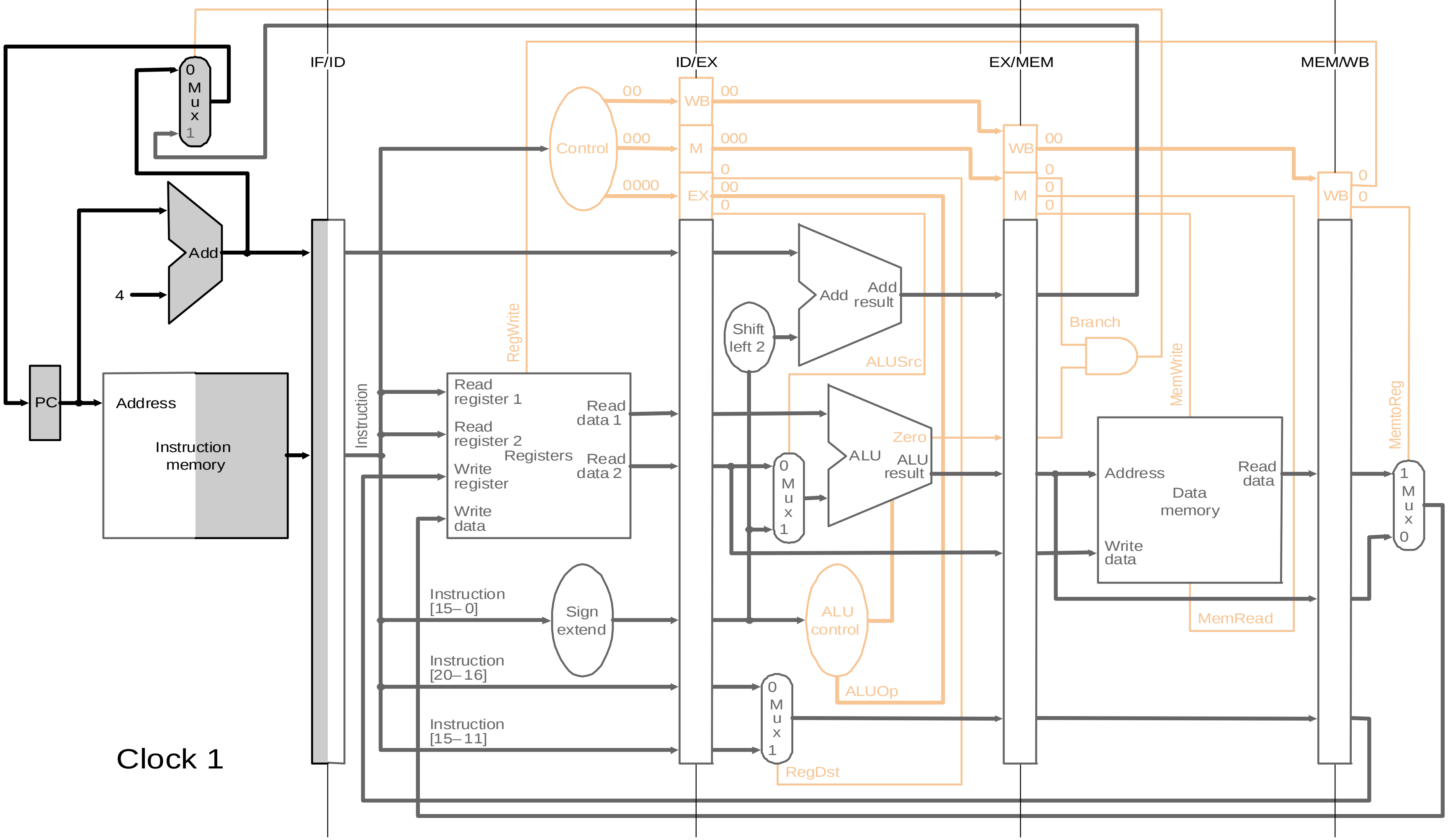
An Example

```
lw    $10, 20($1)
sub   $11, $2, $3
and   $12, $4, $5
or    $13, $6, $7
add   $14, $8, $9
```

Cycle 1

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

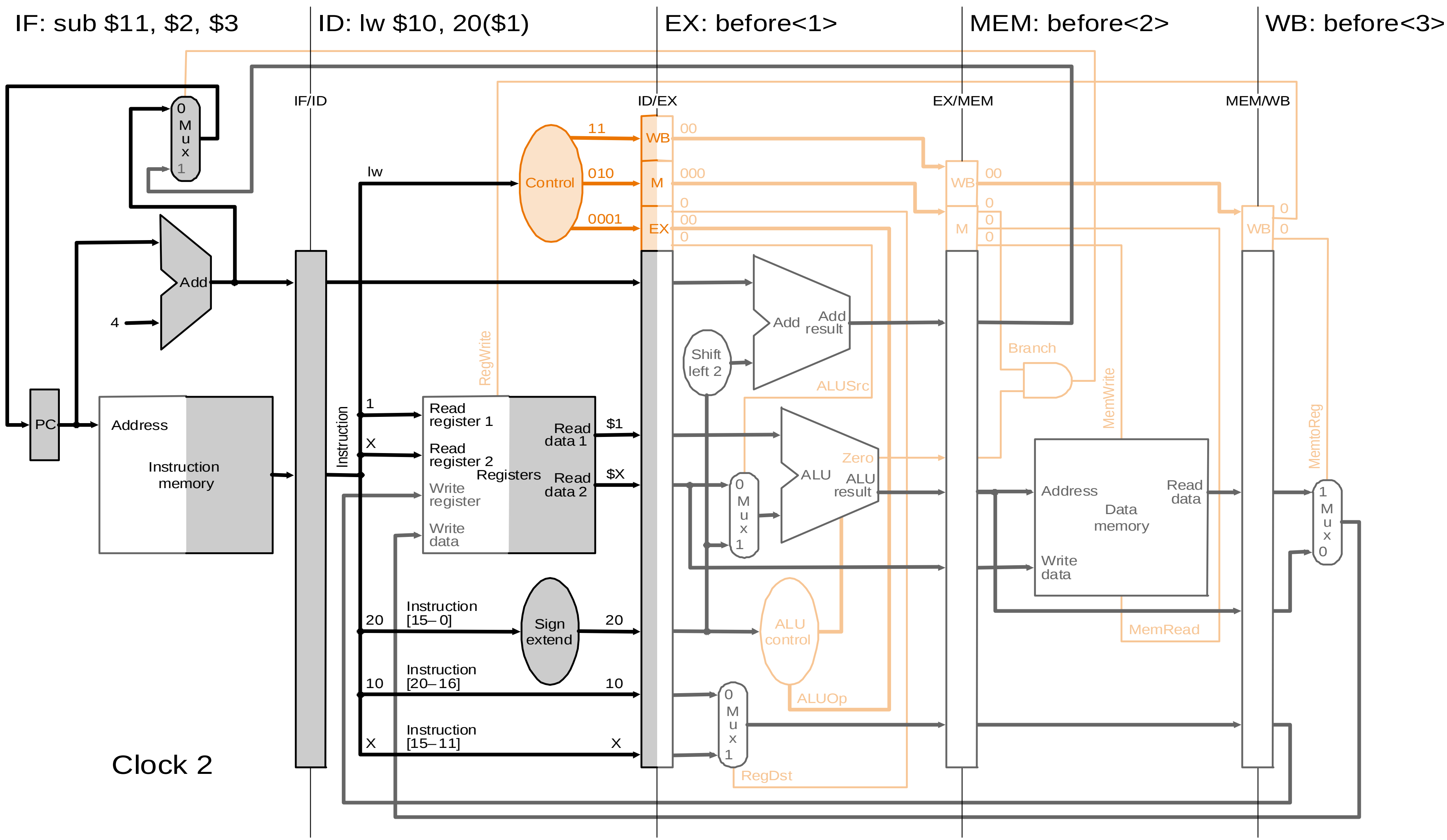
IF: lw $10, 20($1) | ID: before<1> | EX: before<2> | MEM: before<3> | WB: before<4>



Clock 1

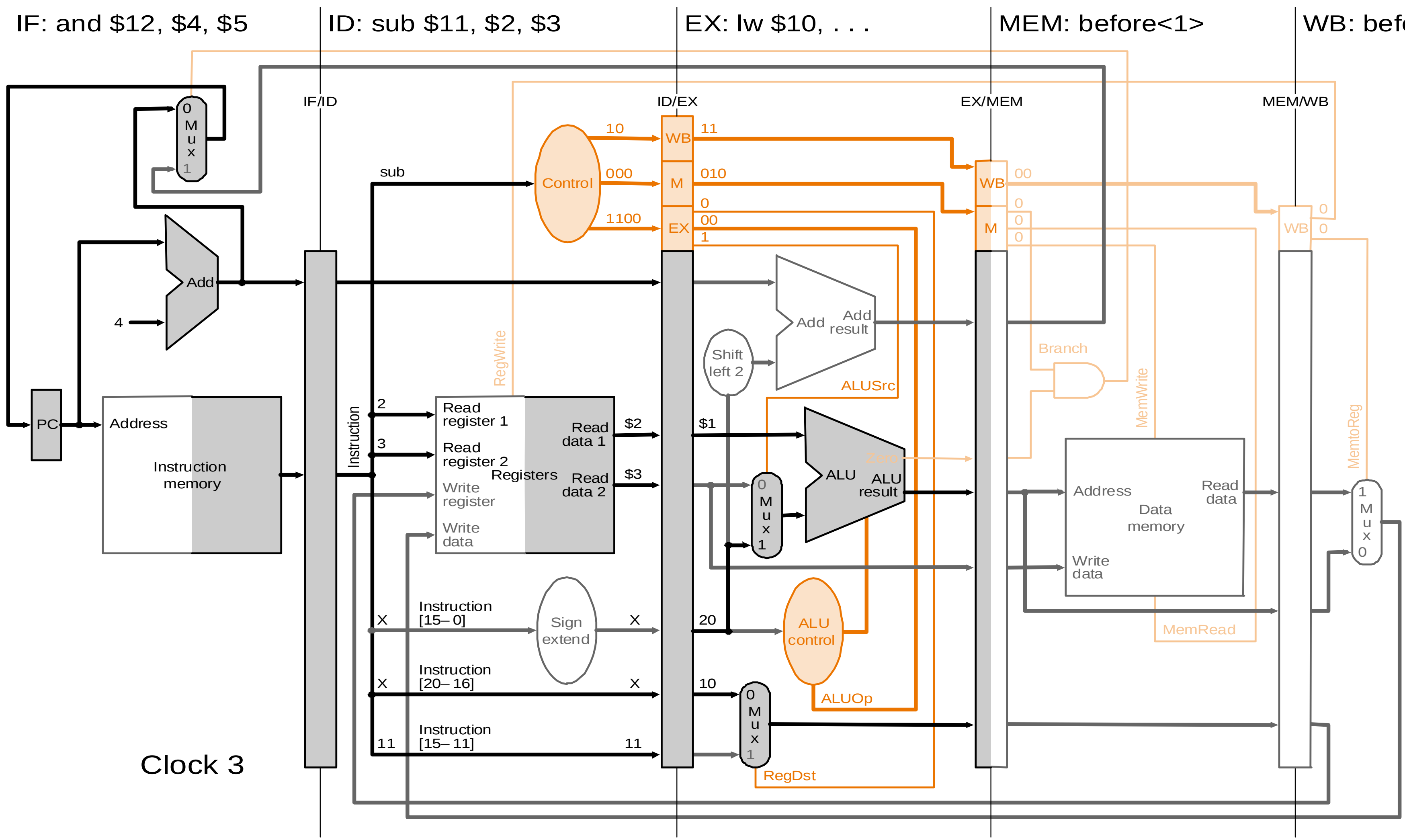
Cycle 2

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



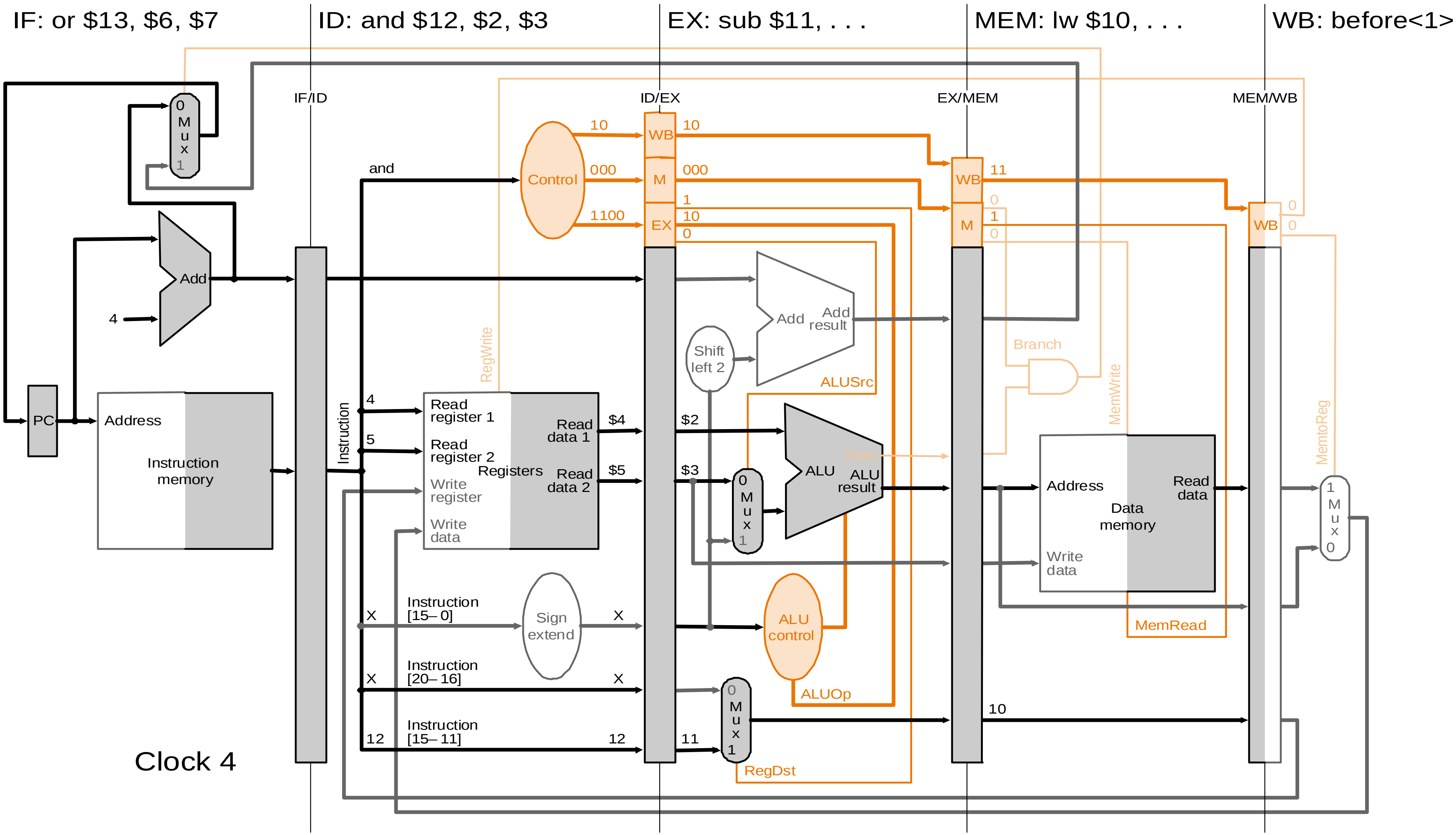
Cycle 3

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



Cycle 4

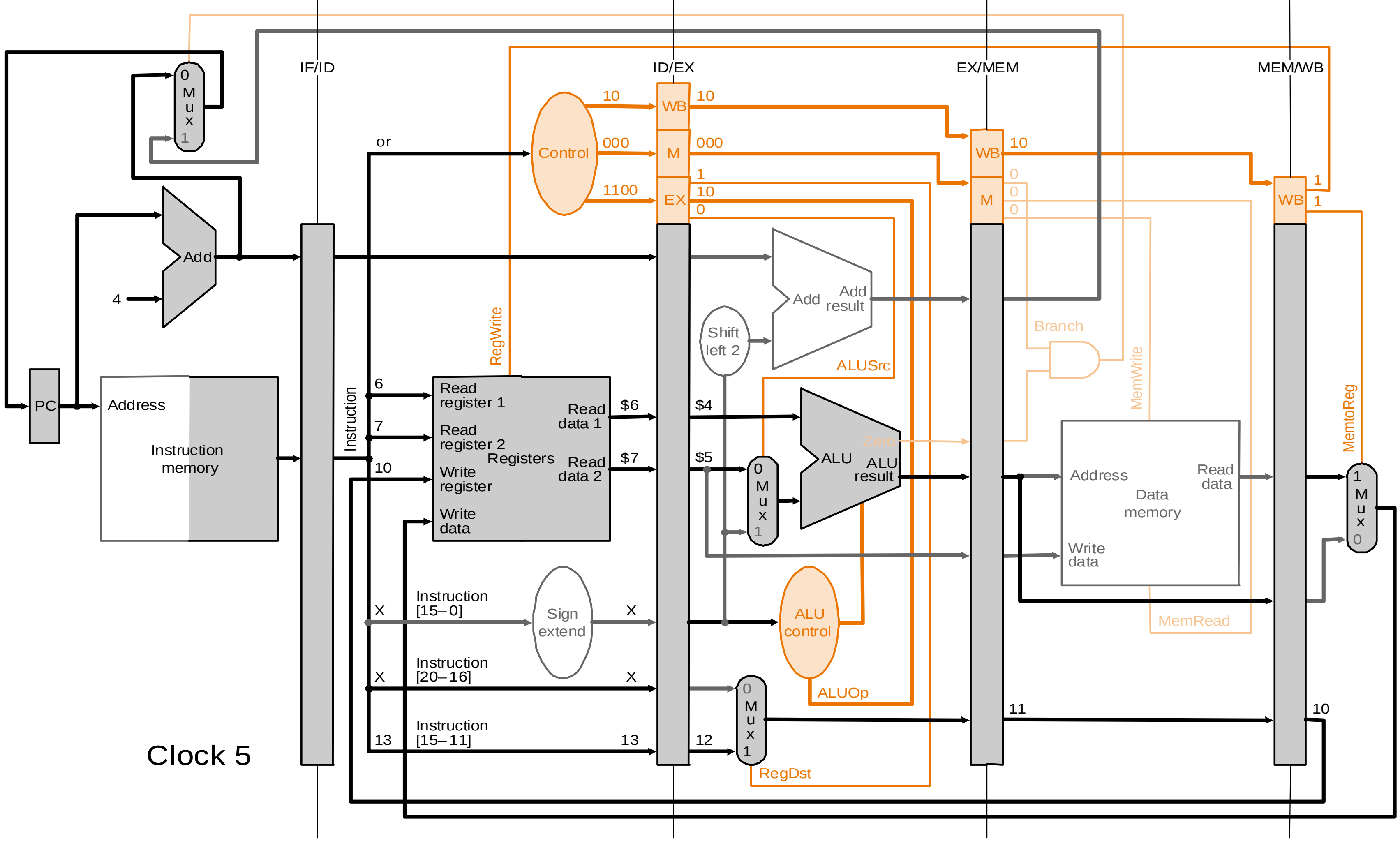
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



Cycle 5

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

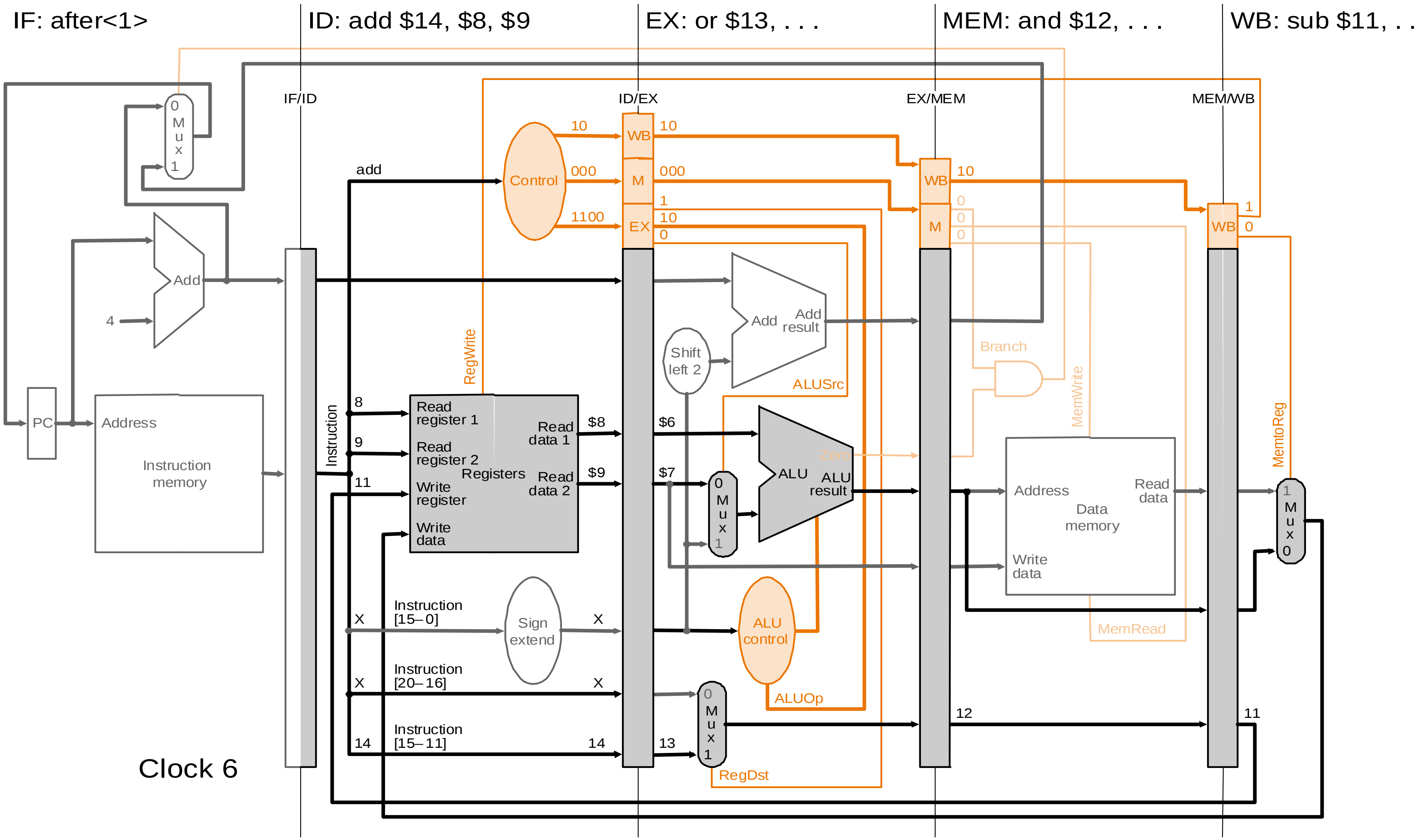
IF: add $14, $8, $9 | ID: or $13, $6, $7 | EX: and $12, . . . | MEM: sub $11, . . . | WB: lw $10, . . .



Clock 5

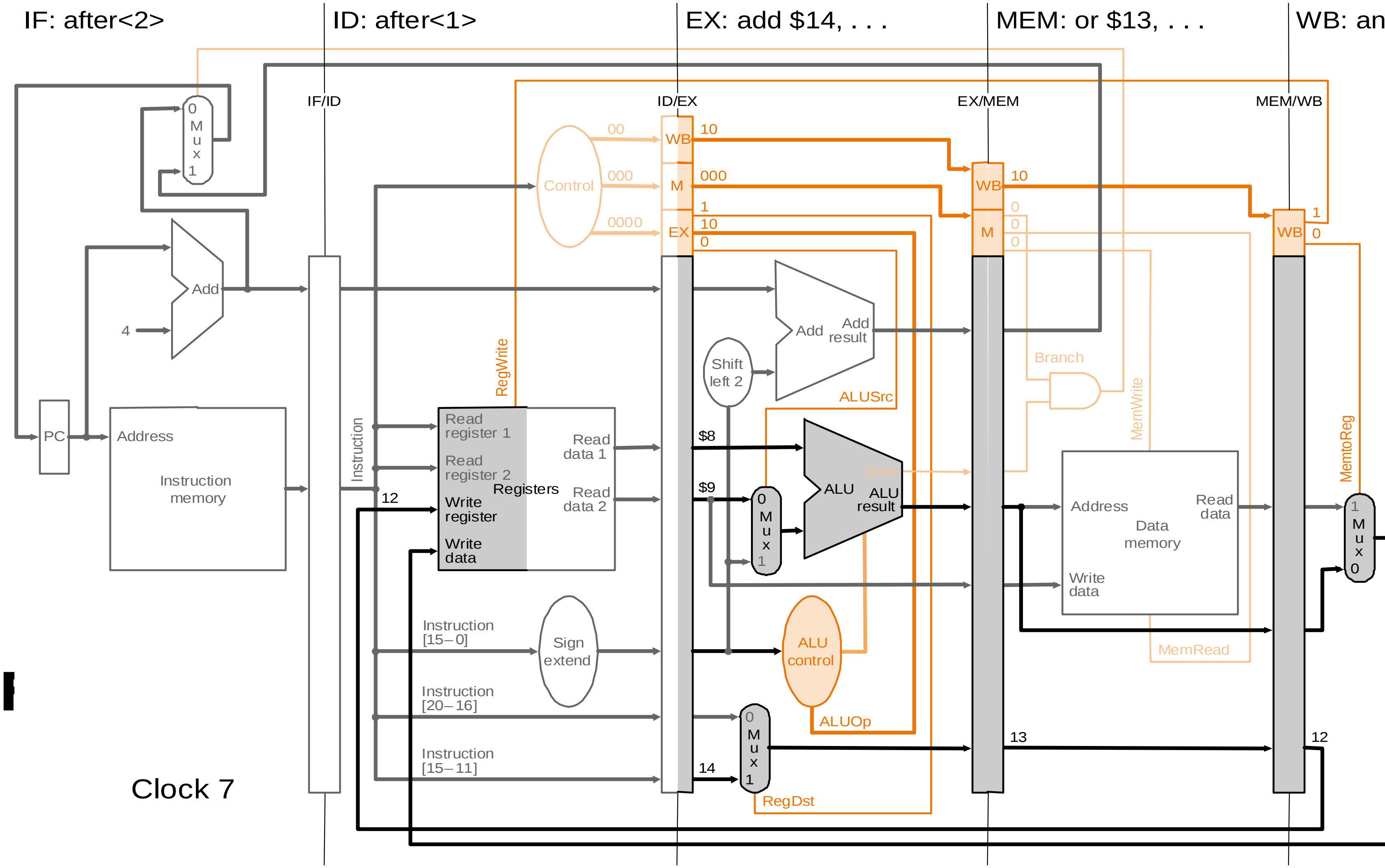
Cycle 6

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



Cycle 7

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

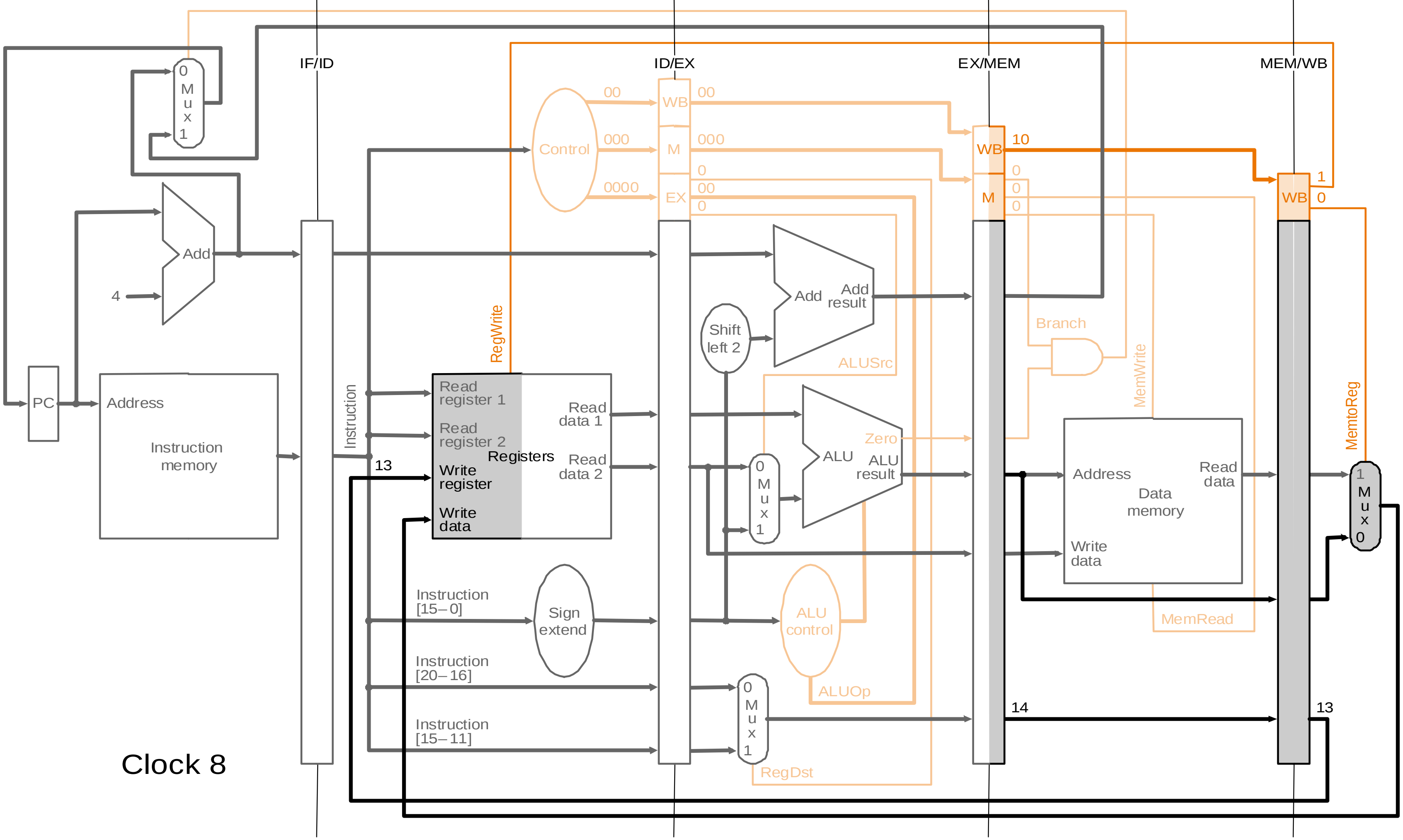


Clock 7

Cycle 8

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

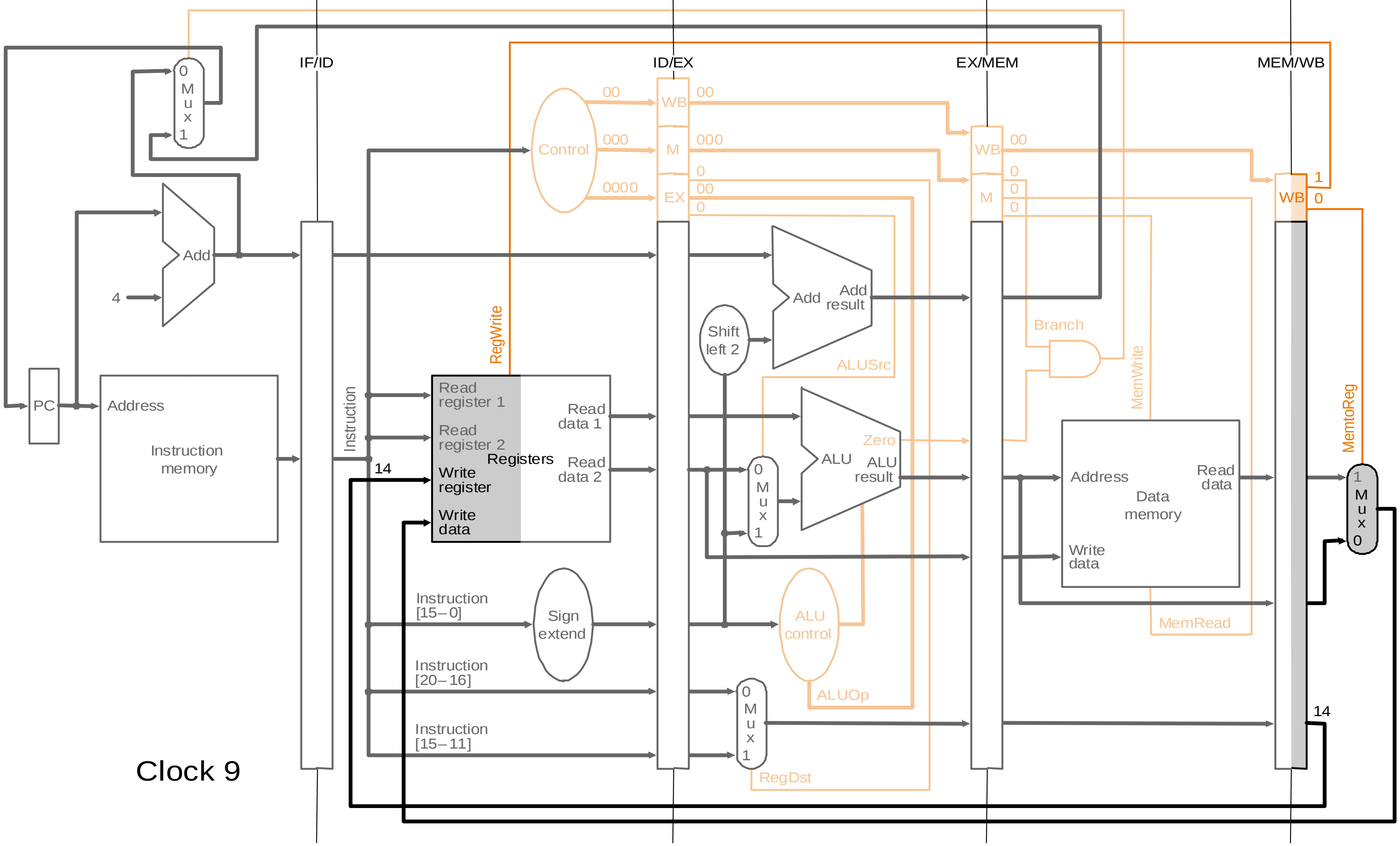
IF: after<3> | ID: after<2> | EX: after<1> | MEM: add $14, . . . | WB: or $13, . . .



Cycle 9

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

IF: after<4> ID: after<3> EX: after<2> MEM: after<1> WB: add $14, . . .



Clock 9

Outline

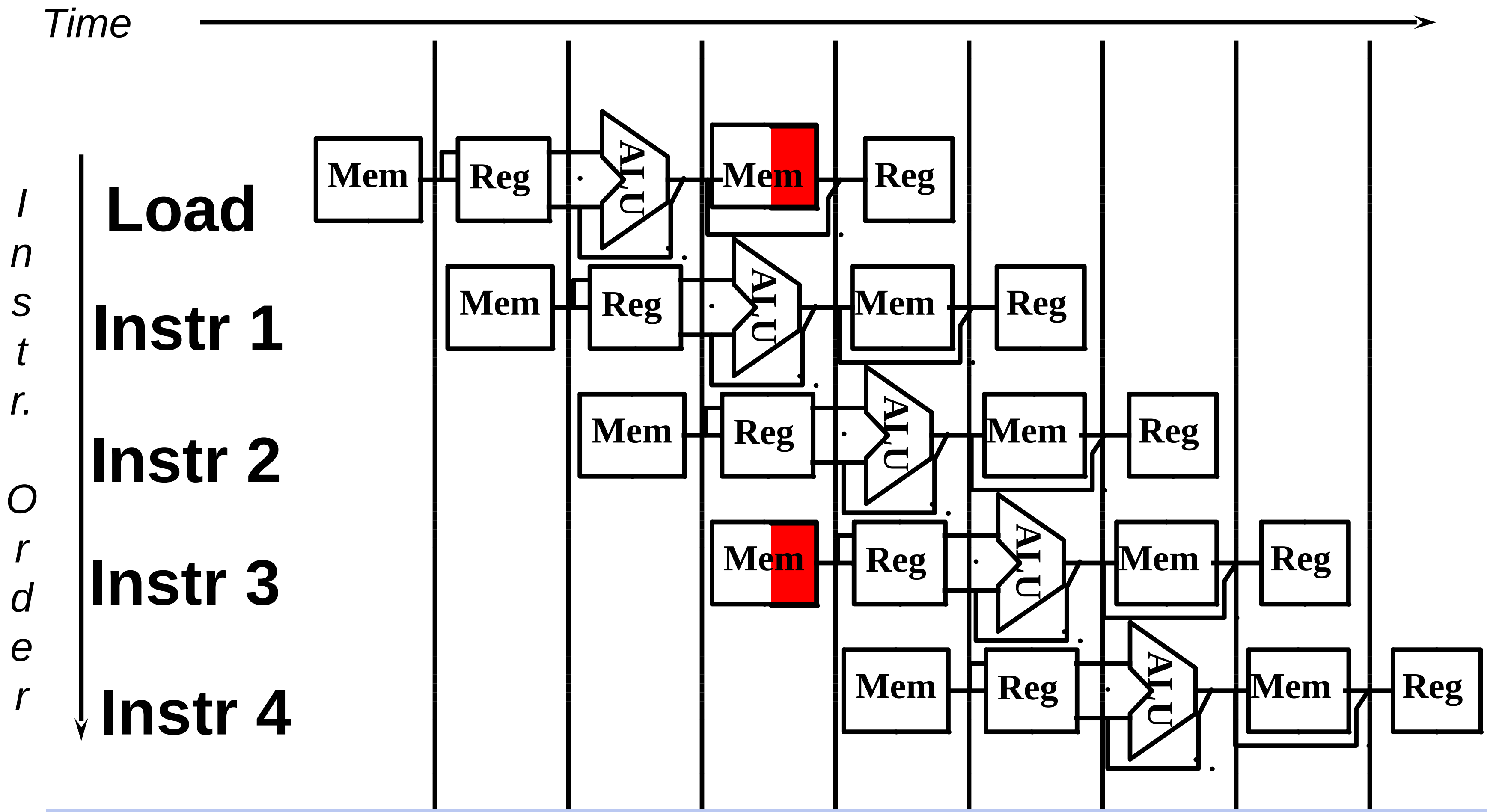
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

Pipeline Hazards

- Pipeline Hazards:
  - Structural hazards: attempt to use the same resource in two different ways at the same time
  - Data hazards: attempt to use data item before ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: attempt to make decision before condition is evaluated
    - Branch instructions

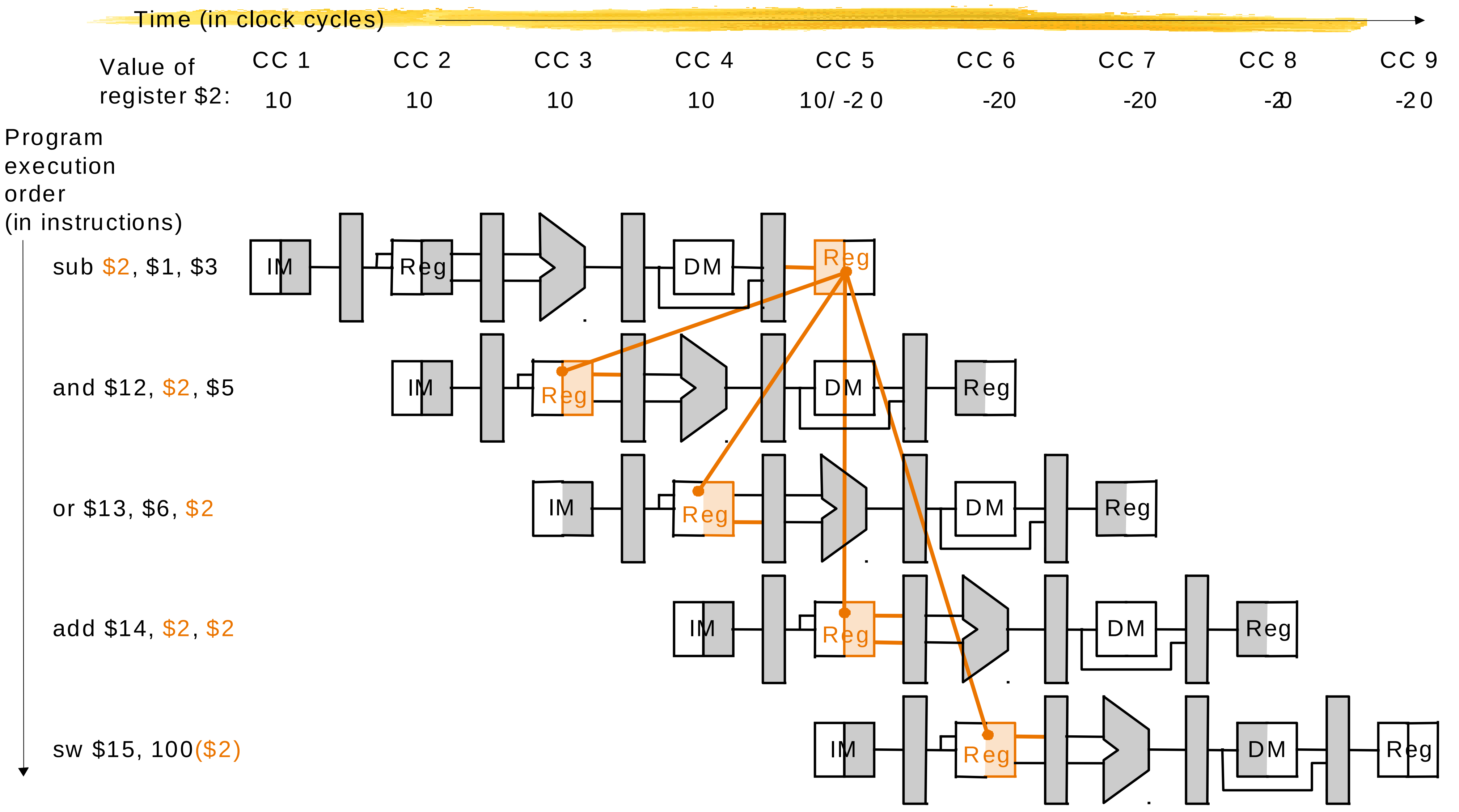
- Can always resolve hazards by waiting
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Structural Hazard: Single Memory



Use 2 memory: data memory and instruction memory

Data Hazards



Types of Data Hazards

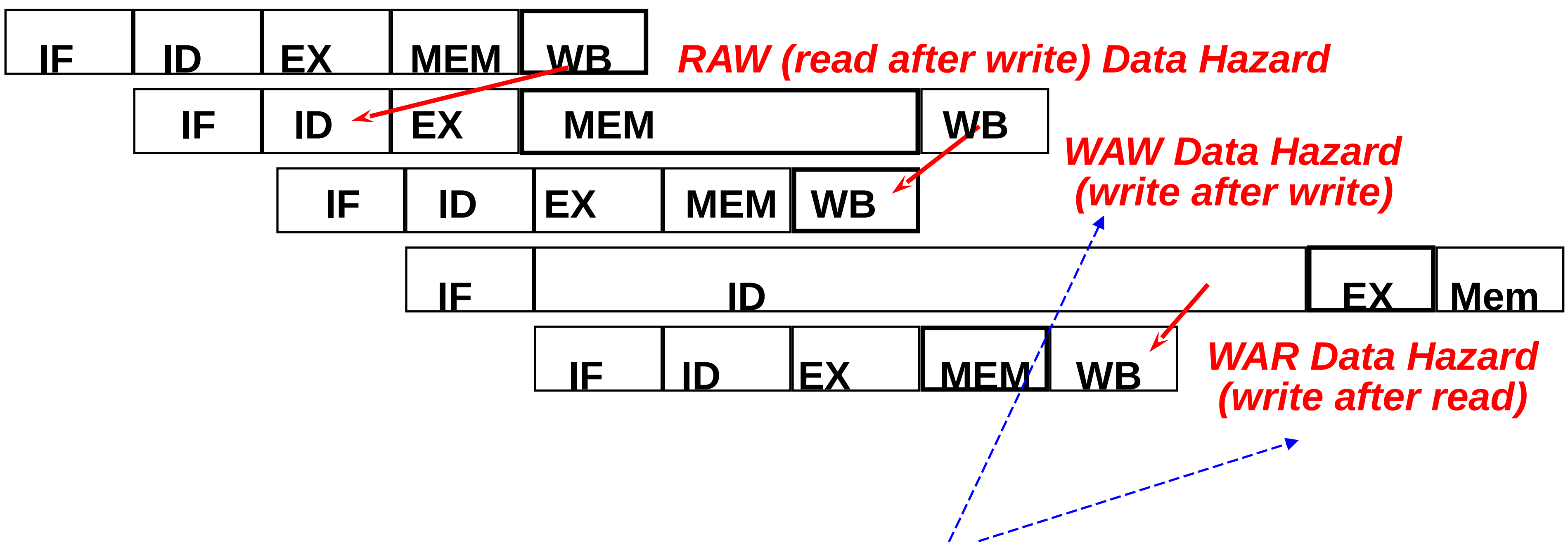
Three types: (inst. i1 followed by inst. i2)

- **RAW (read after write):**
  i2 tries to read operand before i1 writes it

- **WAR (write after read):**
  i2 tries to write operand before i1 reads it
  - **Can't happen in MIPS 5-stage pipeline because all instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5**

- **WAW (write after write):**
  i2 tries to write operand before i1 writes it
  - **Can't happen in MIPS 5-stage pipeline because all instructions take 5 stages, and writes are always in stage 5**

Pipeline Hazards Illustrated



Cannot appear in the MIPS discussed in Chap. 4

Hardware Solution: Forwarding

- Idea: fetch “fresh” data as early as possible

- Two steps:

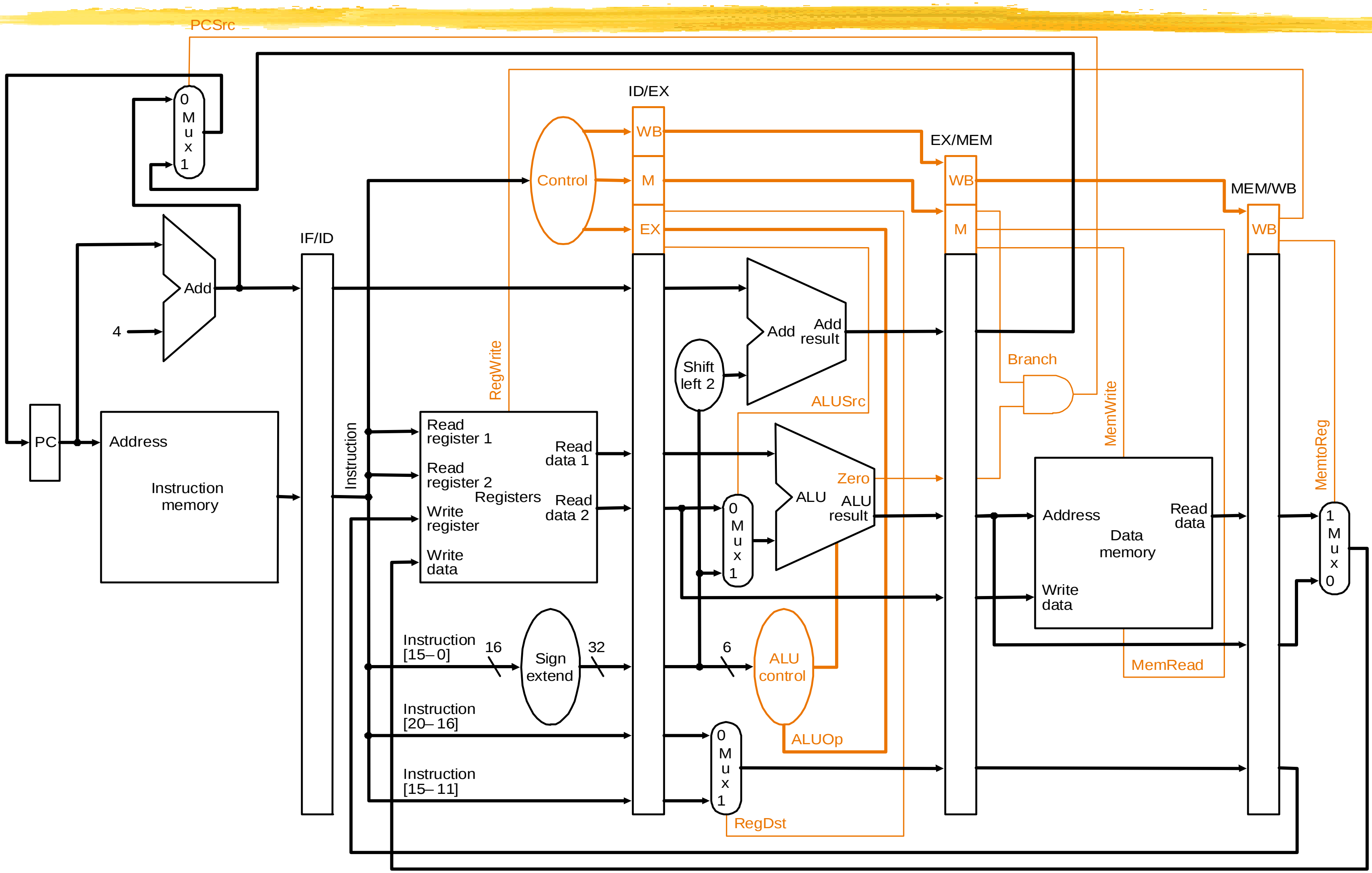
Step 1: Detect data hazard:

- Is the datum just produced required by the following inst.?

Step 2: Forward intermediate data to resolve hazard

- If yes, then forward the requested datum to the requesting inst. immediately.

Pipelined Datapath with Control II (as before)



Hazard Detection

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

- **Hazard conditions:**
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
  - E.g., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the `and` is in EX stage and the `sub` is in MEM stage because
    - `EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)`
- **Whether to forward also depends on:**
  - if the later instruction is going to write the same register – no need to forward, even if there is register number match as in conditions above
  - if the destination register of the later instruction is $0 – in which case there is no need to forward value ($0 is always 0 and never overwritten)

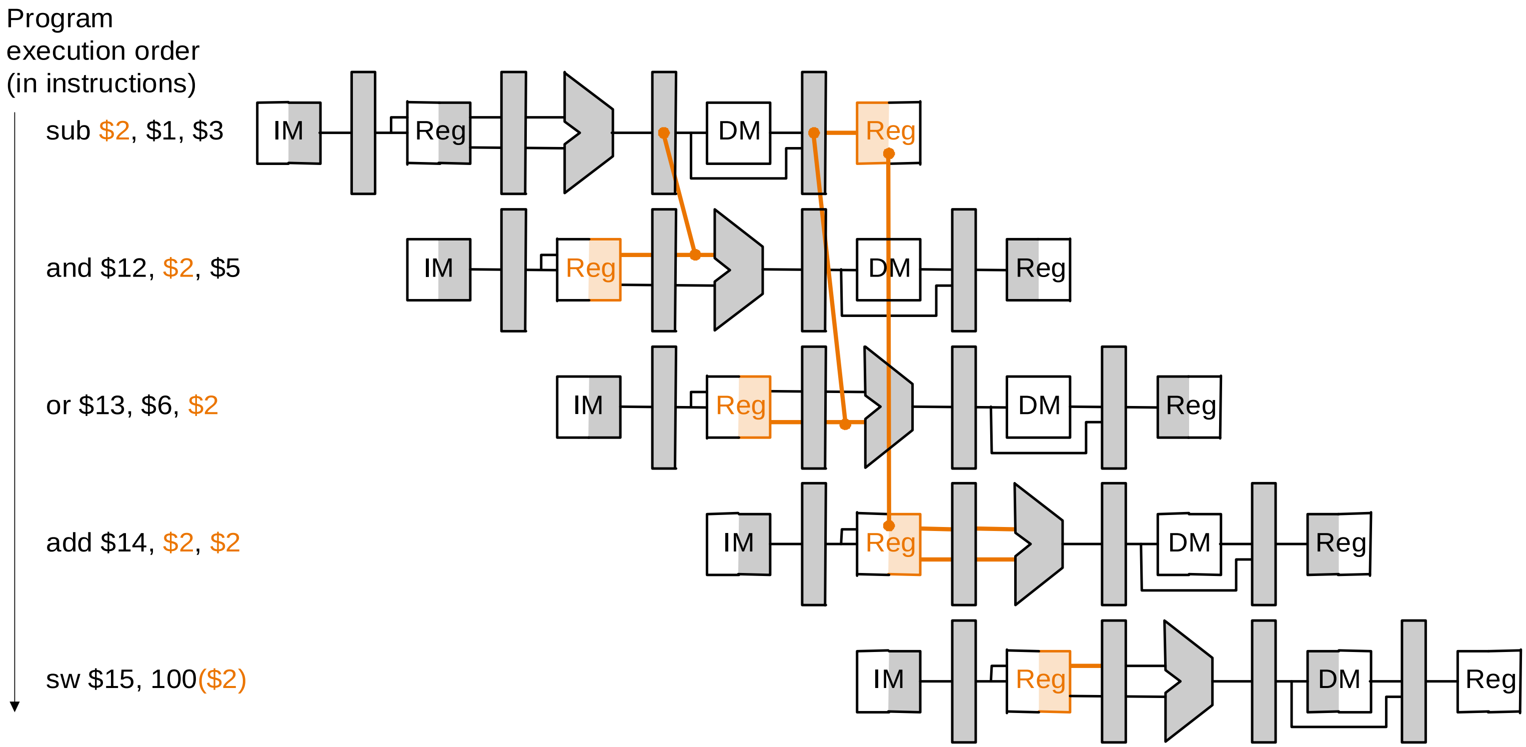
Data Forwarding

- Allow inputs to ALU not just from ID/EX, but also later pipeline registers
- Use multiplexors and control signals to choose appropriate inputs to ALU

Time (in clock cycles)

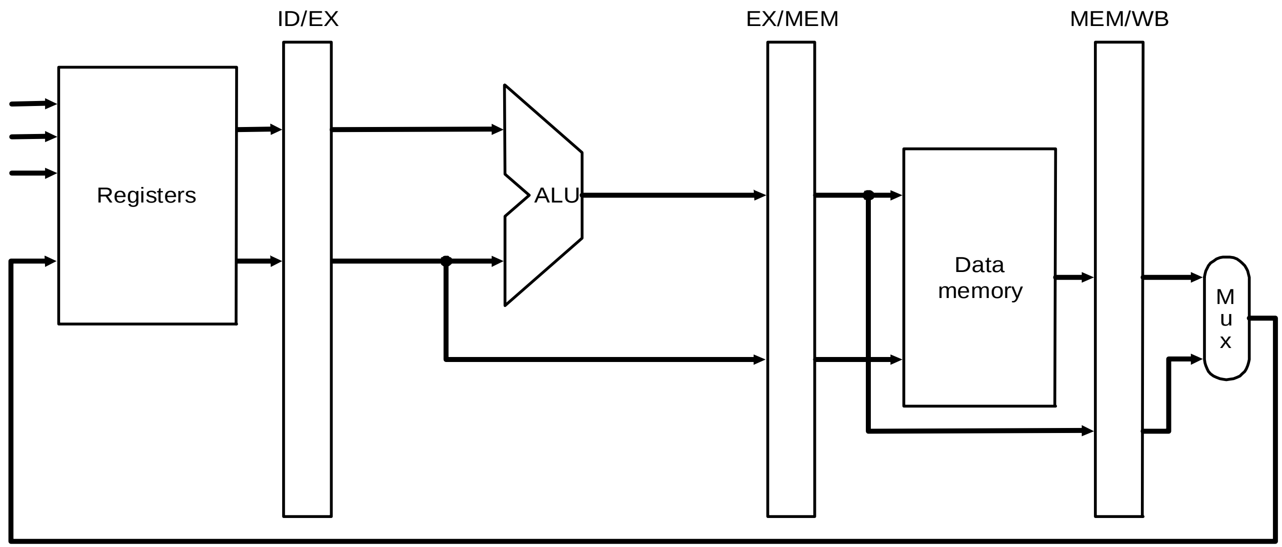
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2 :	10	10	10	10	10/– 20	– 20	– 20	– 20	– 20
Value of EX/MEM :	X	X	X	– 20	X	X	X	X	X
Value of MEM/WB :	X	X	X	X	– 20	X	X	X	X

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

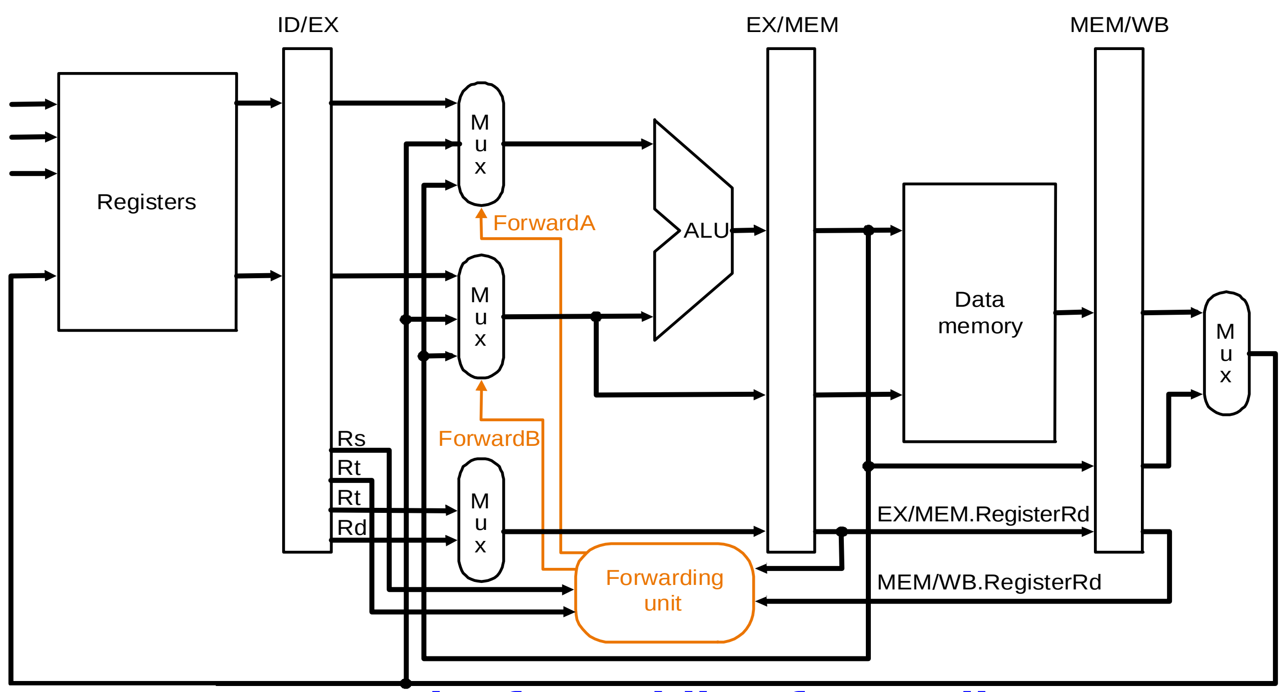


Dependencies between pipelines move forward in time

Forwarding Hardware



Datapath before adding forwarding hardware



Datapath after adding forwarding hardware

Forwarding Hardware: Multiplexor Control

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 prior ALU result
ForwardA = 01	MEM/WB	*The first ALU operand is forwarded from 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 prior ALU result
ForwardB = 01	MEM/WB	*The second ALU operand is forwarded from data memory or an earlier ALU result

* Depending on the selection in the rightmost multiplexor (see datapath with control diagram)

Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. *EX hazard*

```
if (        EX/MEM.RegWrite                                  // if there is a write…
    and ( EX/MEM.RegisterRd ≠ 0 )                            // to a non-$0 register…
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRs ) )  // which matches, then…
            ForwardA = 10

if (        EX/MEM.RegWrite                                  // if there is a write…
    and ( EX/MEM.RegisterRd ≠ 0 )                            // to a non-$0 register…
    and ( EX/MEM.RegisterRd = ID/EX.RegisterRt ) )  // which matches, then…
            ForwardB = 10
```

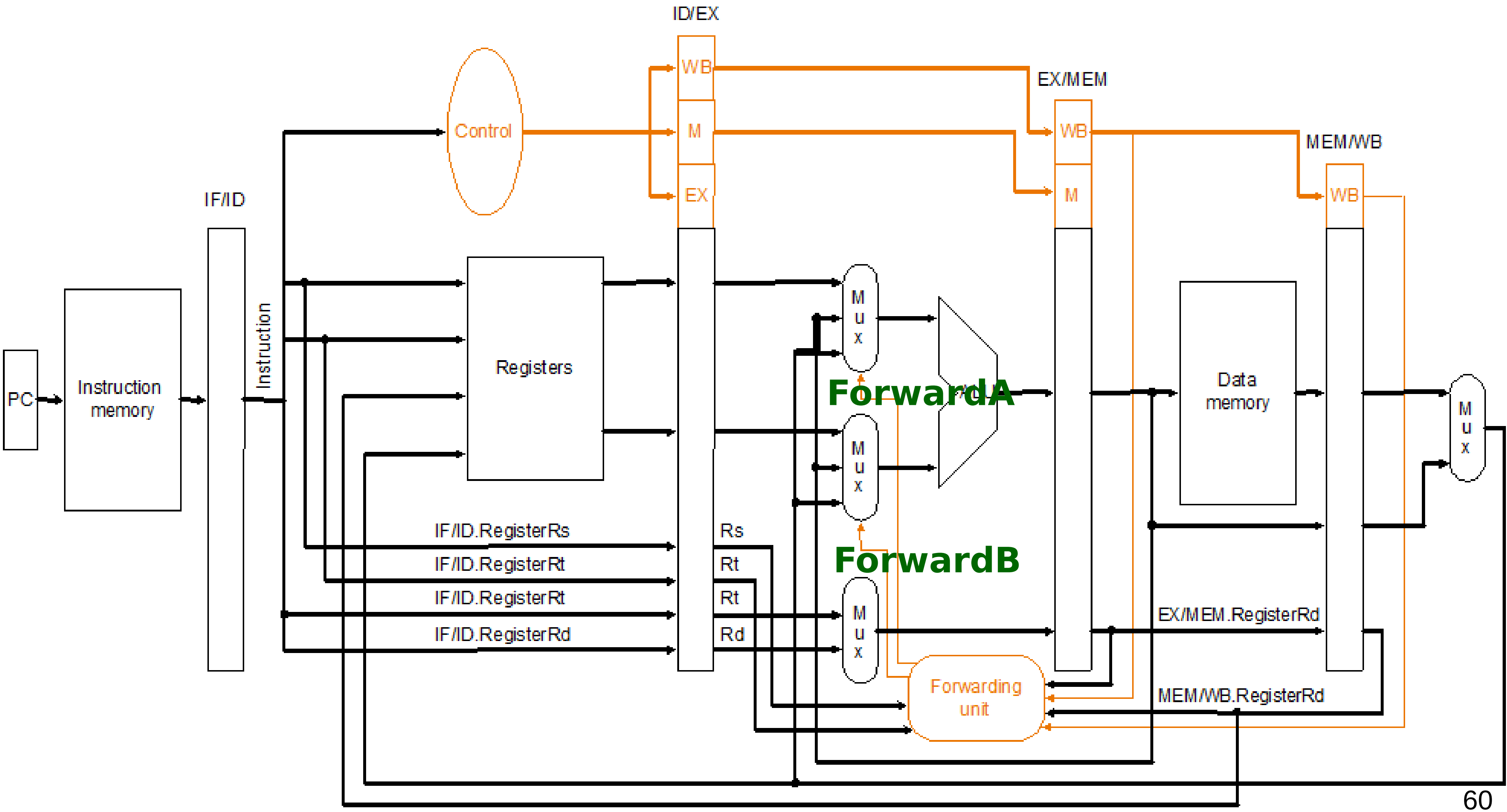
Data Hazard: Detection and Forwarding

2. *MEM hazard*

```
if (      MEM/WB.RegWrite                                   // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                            // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs * )           // and not already a register match
                                                            // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )           // but match with later pipeline register,
then…
          ForwardA = 01

if (      MEM/WB.RegWrite                                   // if there is a write…
   and ( MEM/WB.RegisterRd ≠ 0 )                            // to a non-$0 register…
   and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt * )           // and not already a register match
                                                            // with earlier pipeline register…
   and ( MEM/WB.RegisterRd = ID/EX.RegisterRt ) )           // but match with later pipeline register,
then…
          ForwardB = 01
```

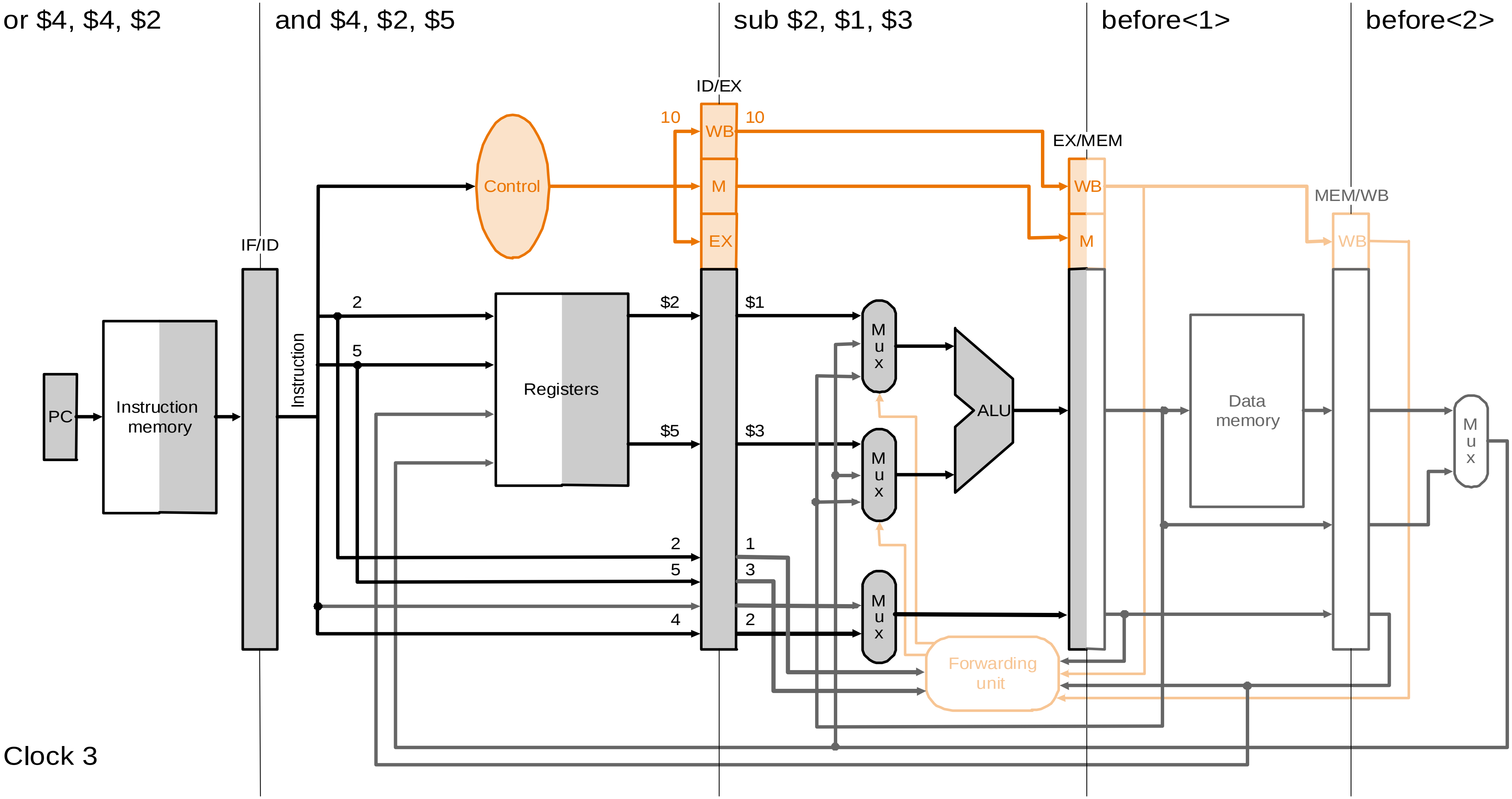
*** This check is necessary, e.g., for sequences such as add $1, $1, $2; add $1, $1, $3; add $1, $1, $4, where an earlier pipeline (EX/MEM) register has more recent data**

Pipeline with Forwarding



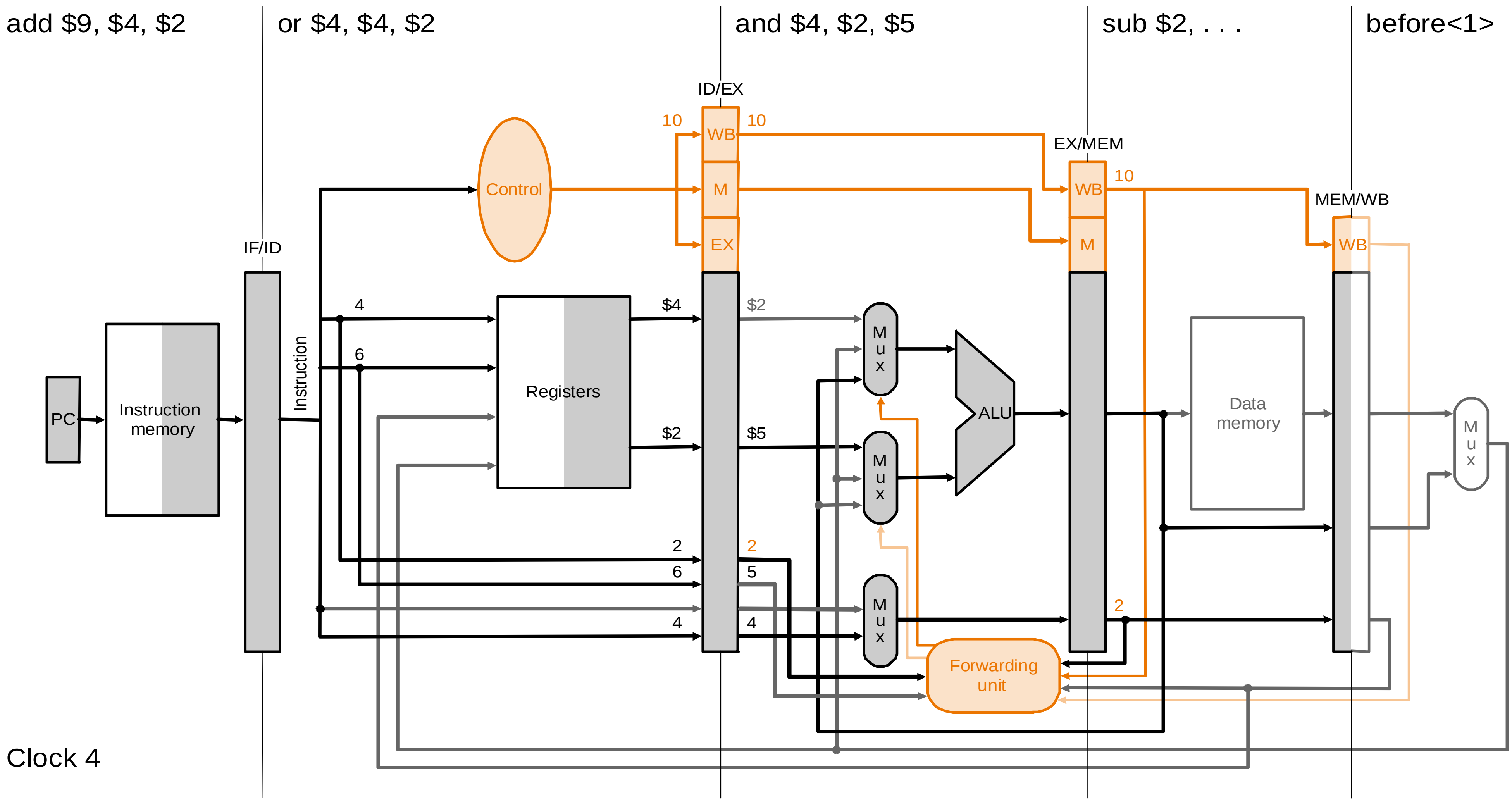
Cycle 3

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



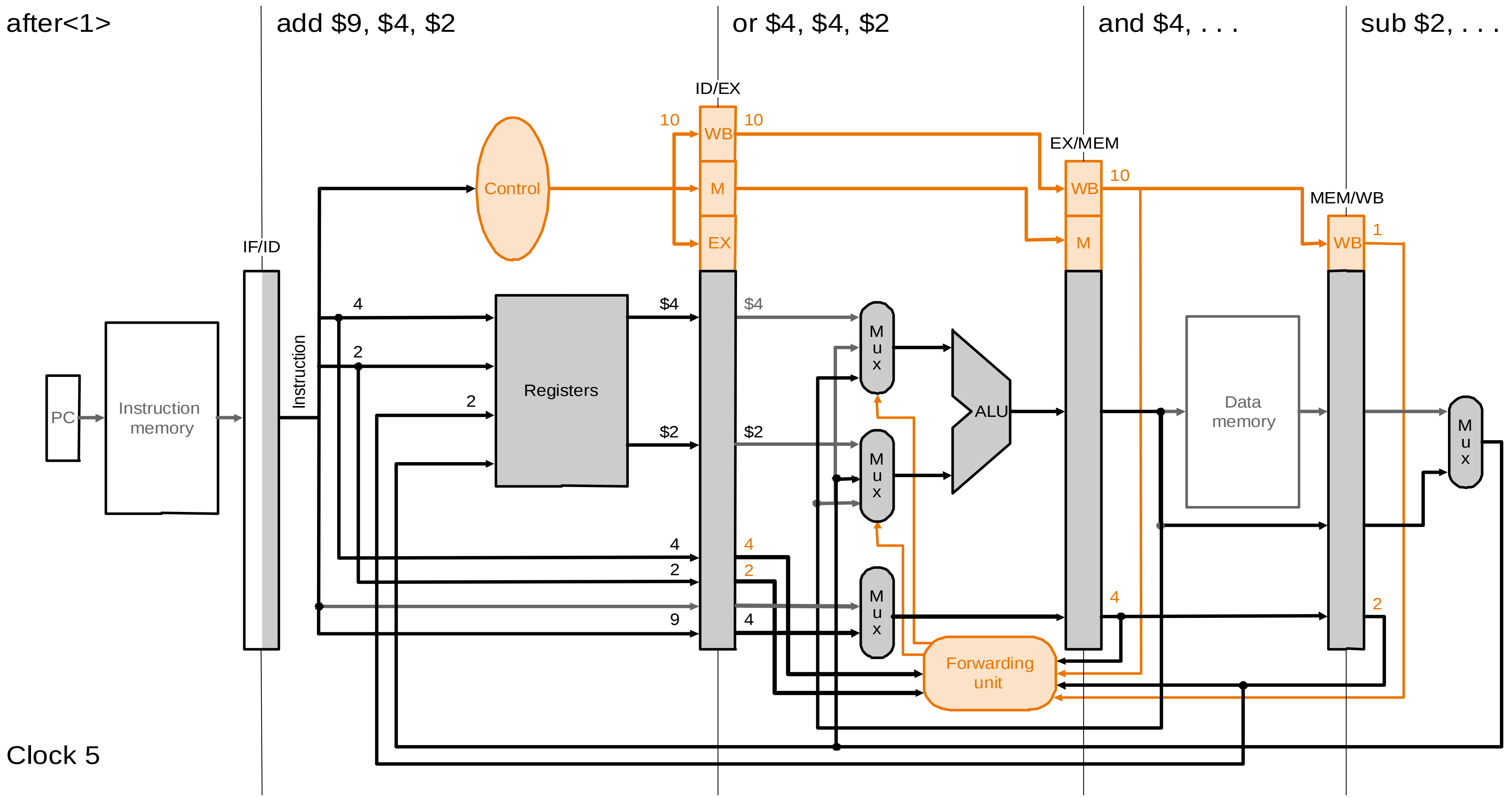
Cycle 4

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



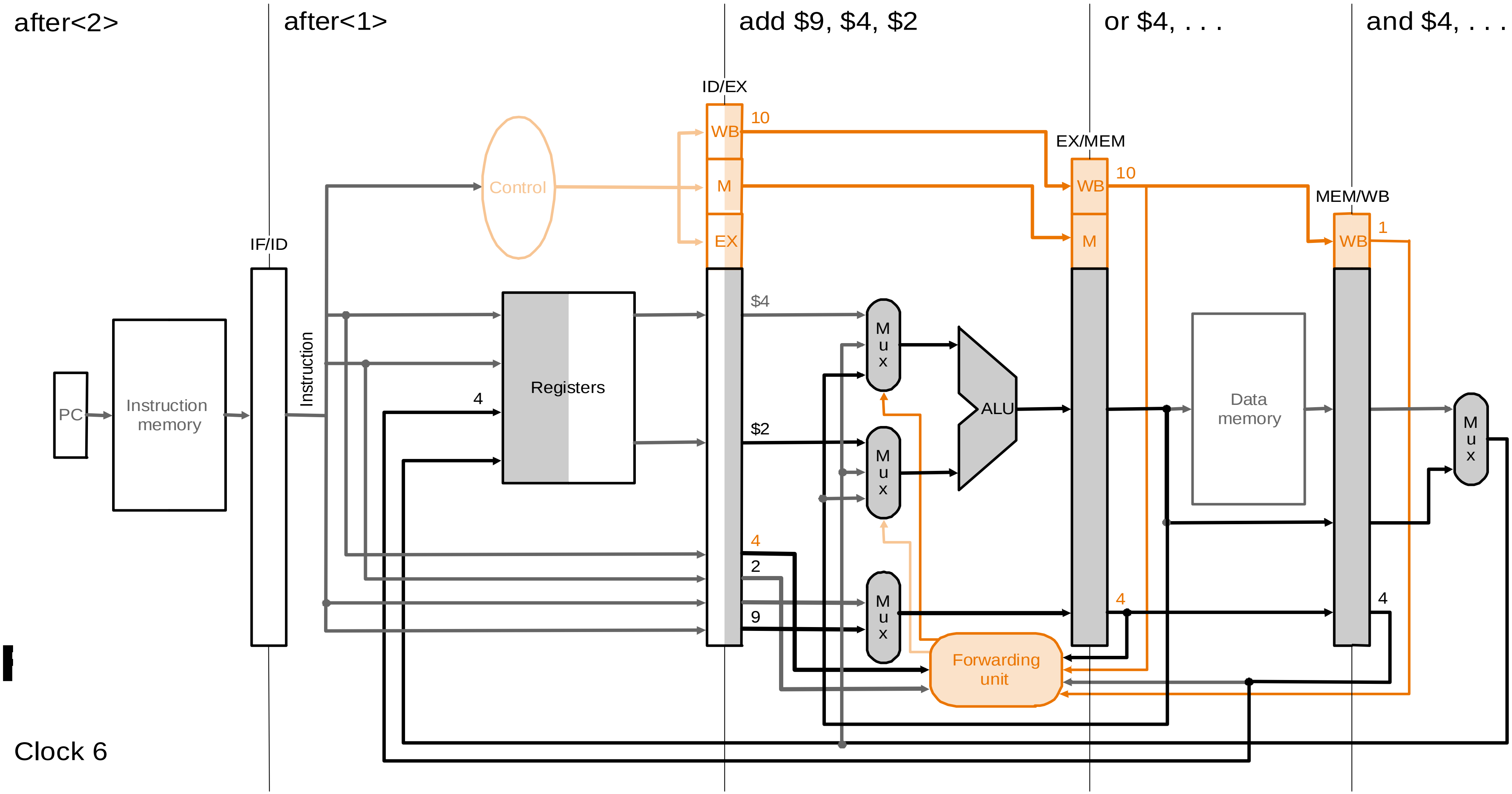
Cycle 5

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Cycle 6

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Questions?

- **The data path is incomplete (in Slide 59) as we cannot compute the target address for `lw` and `sw`.**

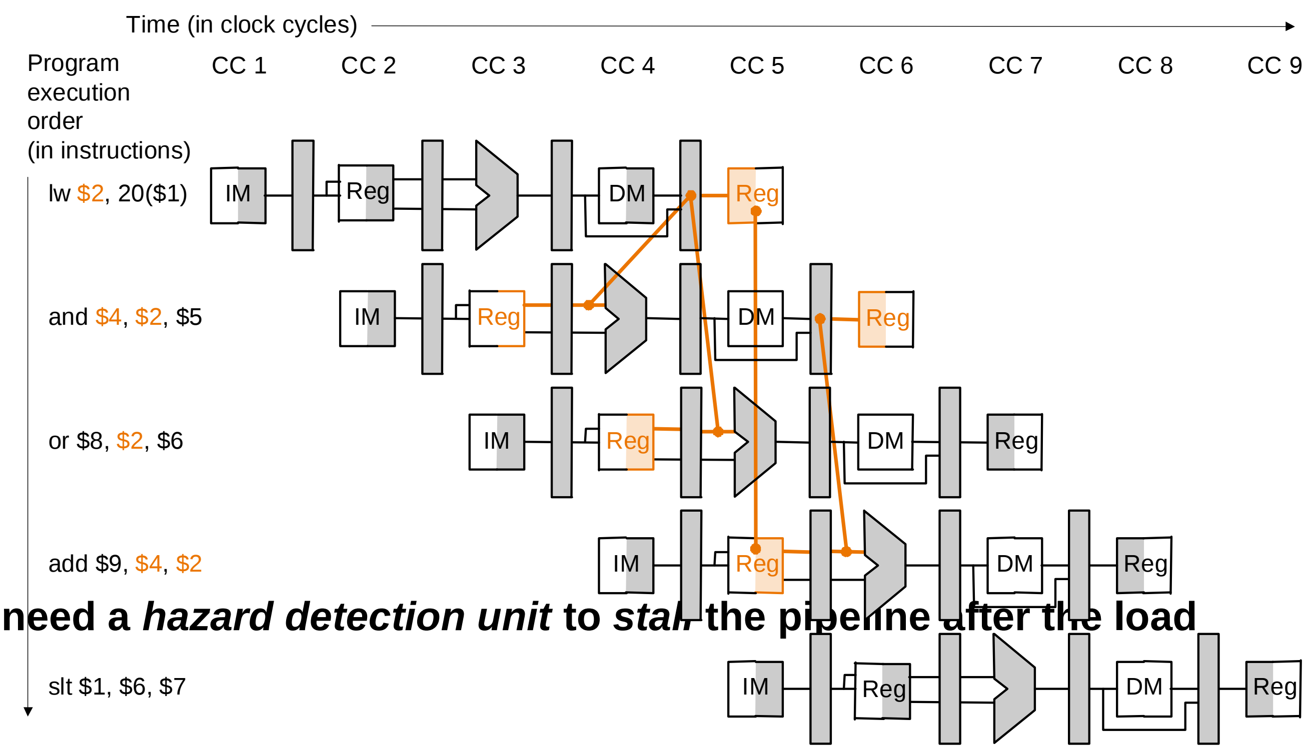
- **How about `lw` followed by `sw`?**

Data Hazards and Stalls

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register
  - therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

```
lw   $2, 20($1)
and  $4, $2, $5
or   $8, $2, $6
add  $9, $4, $2
Slt  $1, $6, $7
```

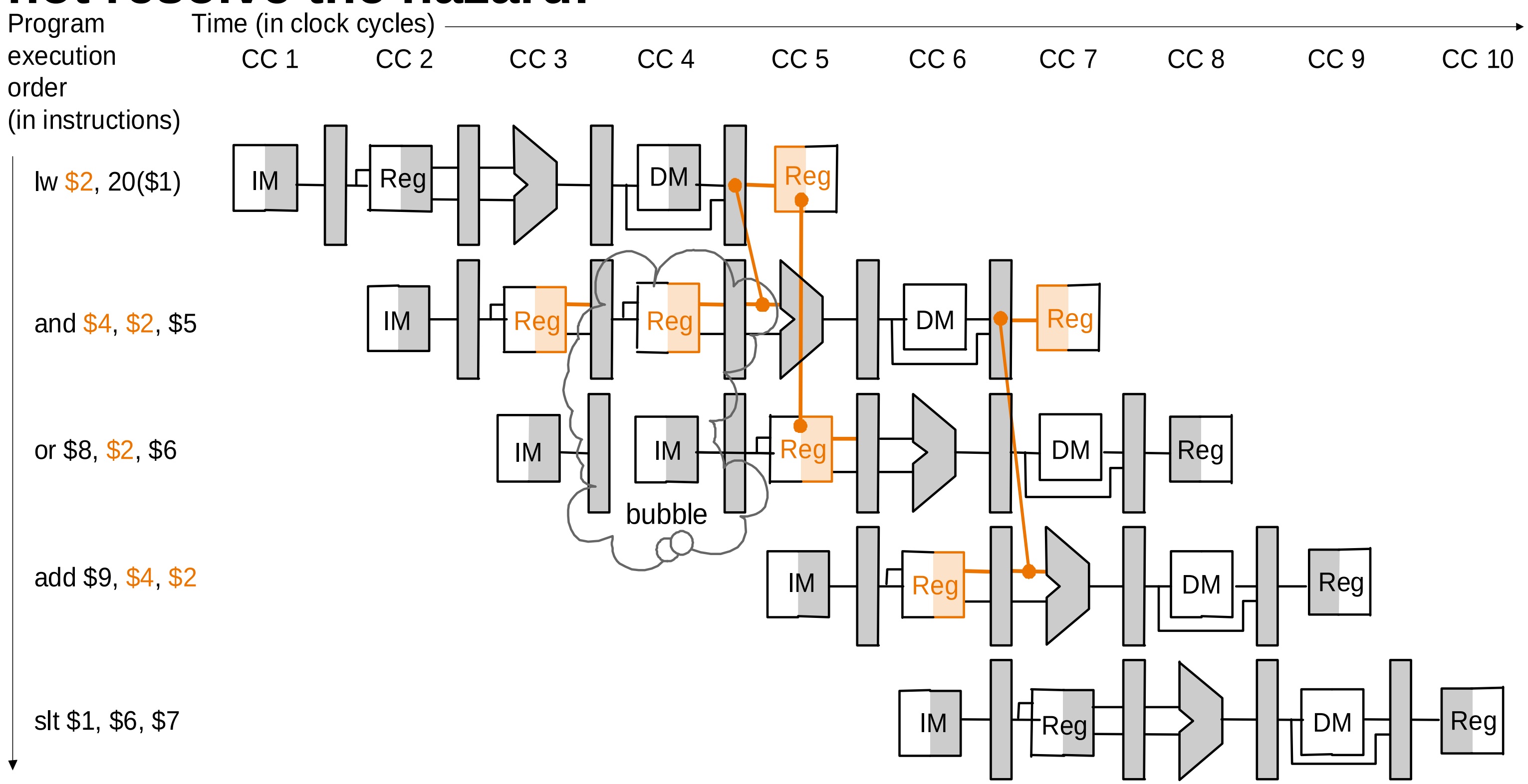
As even a pipeline dependency goes backward in time forwarding will not solve the hazard



Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```



Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

Hazard Detection Logic to Stall

- **Hazard detection unit implements the following check if to stall**

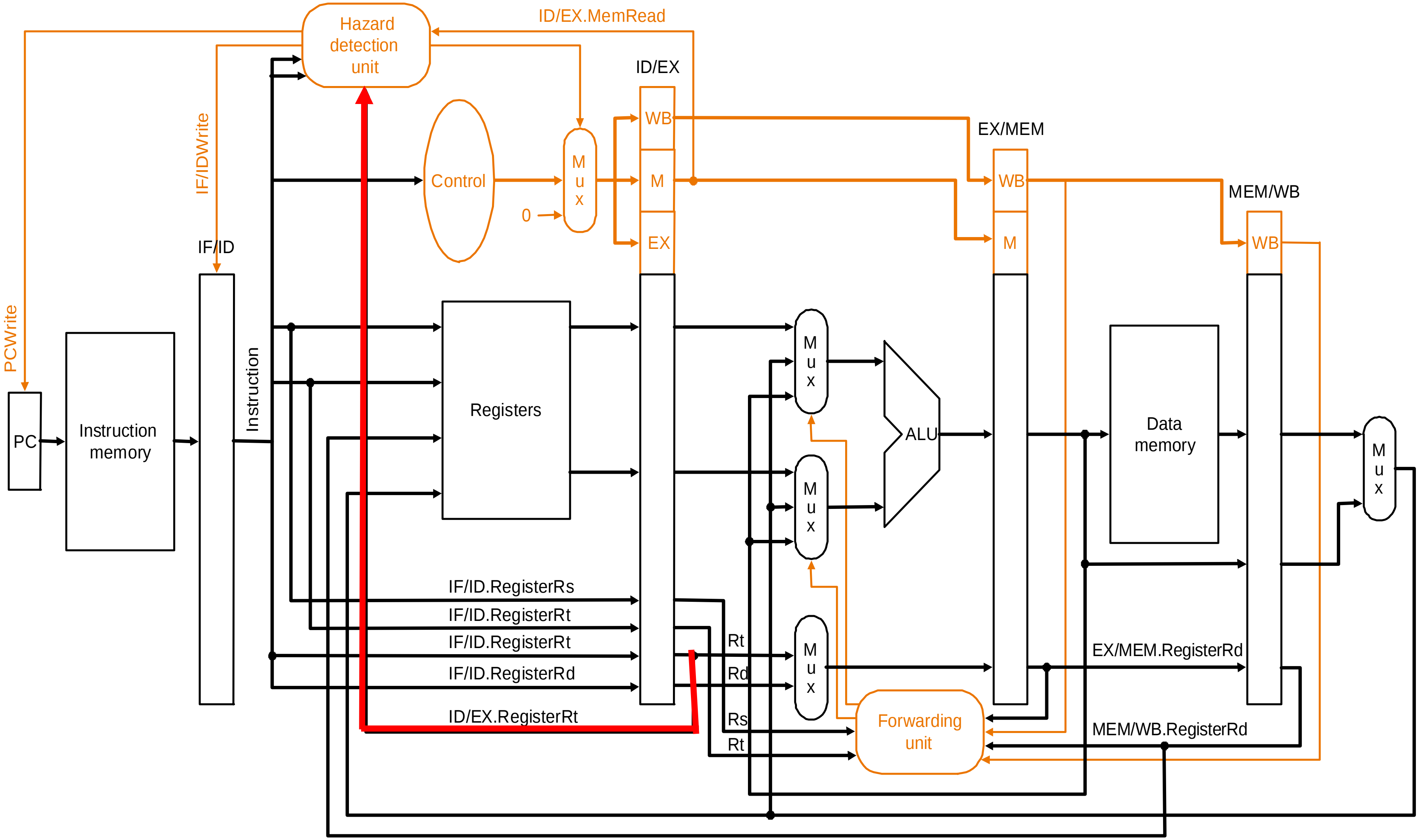
```
if ( ID/EX.MemRead                      // if the instruction in the EX stage is a load…

   and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )      // and the destination register
     or  ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )  // matches either source register
                                        // of the instruction in the ID stage, then…
stall the pipeline
```

Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency

- How the hardware stalls the pipeline 1 cycle:
  - does NOT let the IF/ID register change (disable write!) – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware does NOT let the PC change (disable write!) – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0, so effectively the instruction just behind the load becomes a nop – a bubble is said to have been inserted into the pipeline

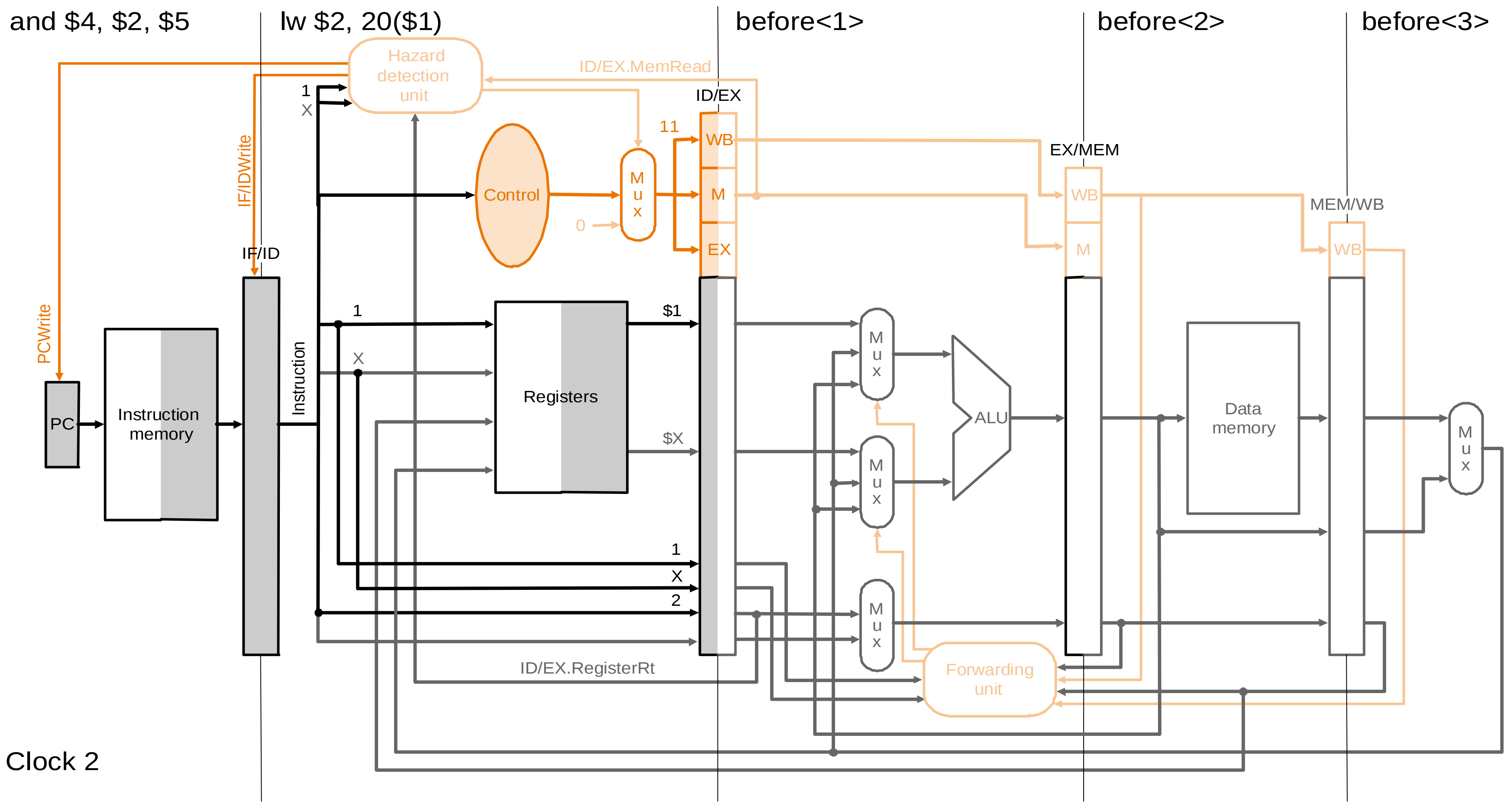
Adding Hazard Detection Unit



Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing

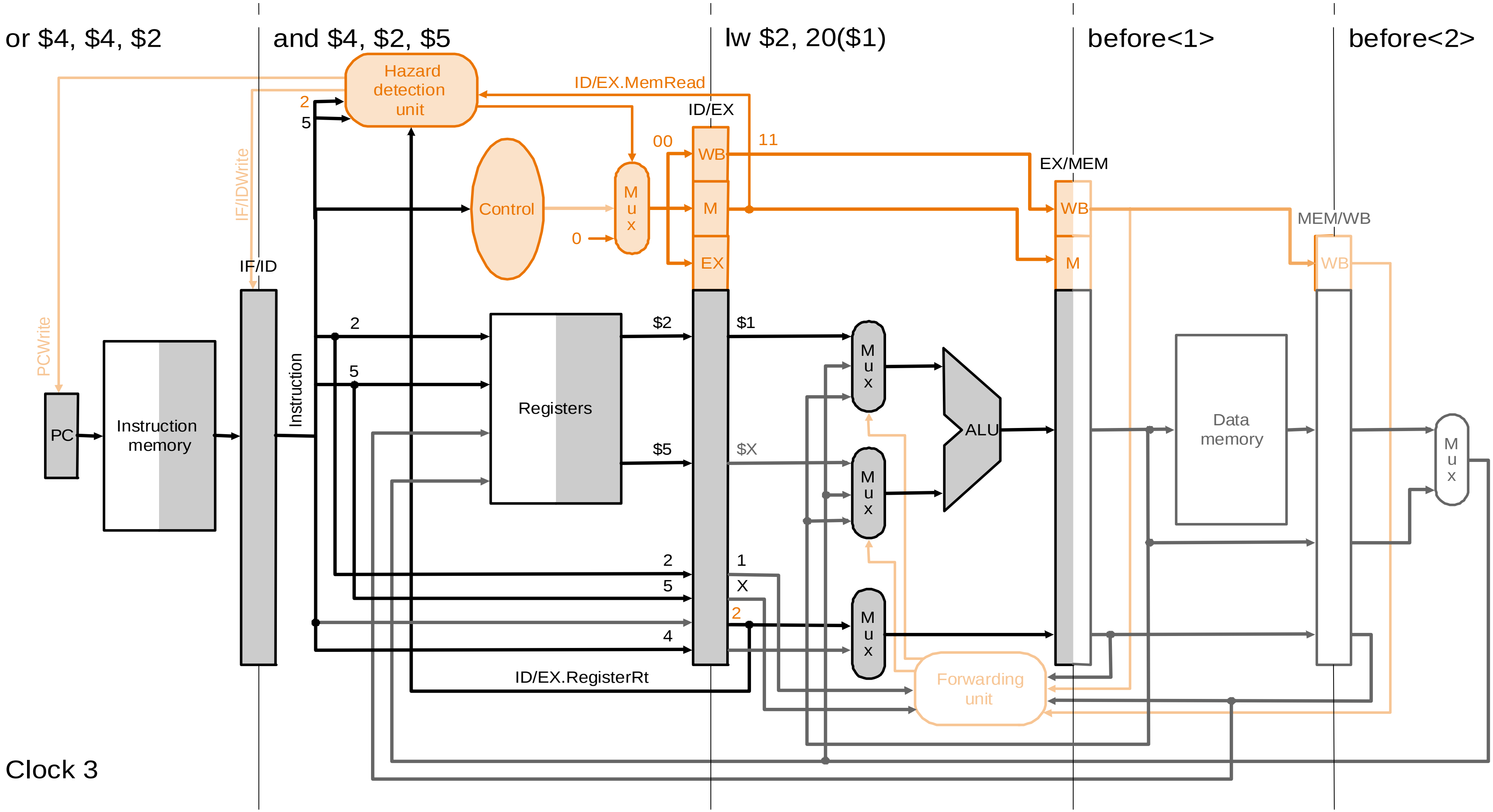
Cycle 2

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



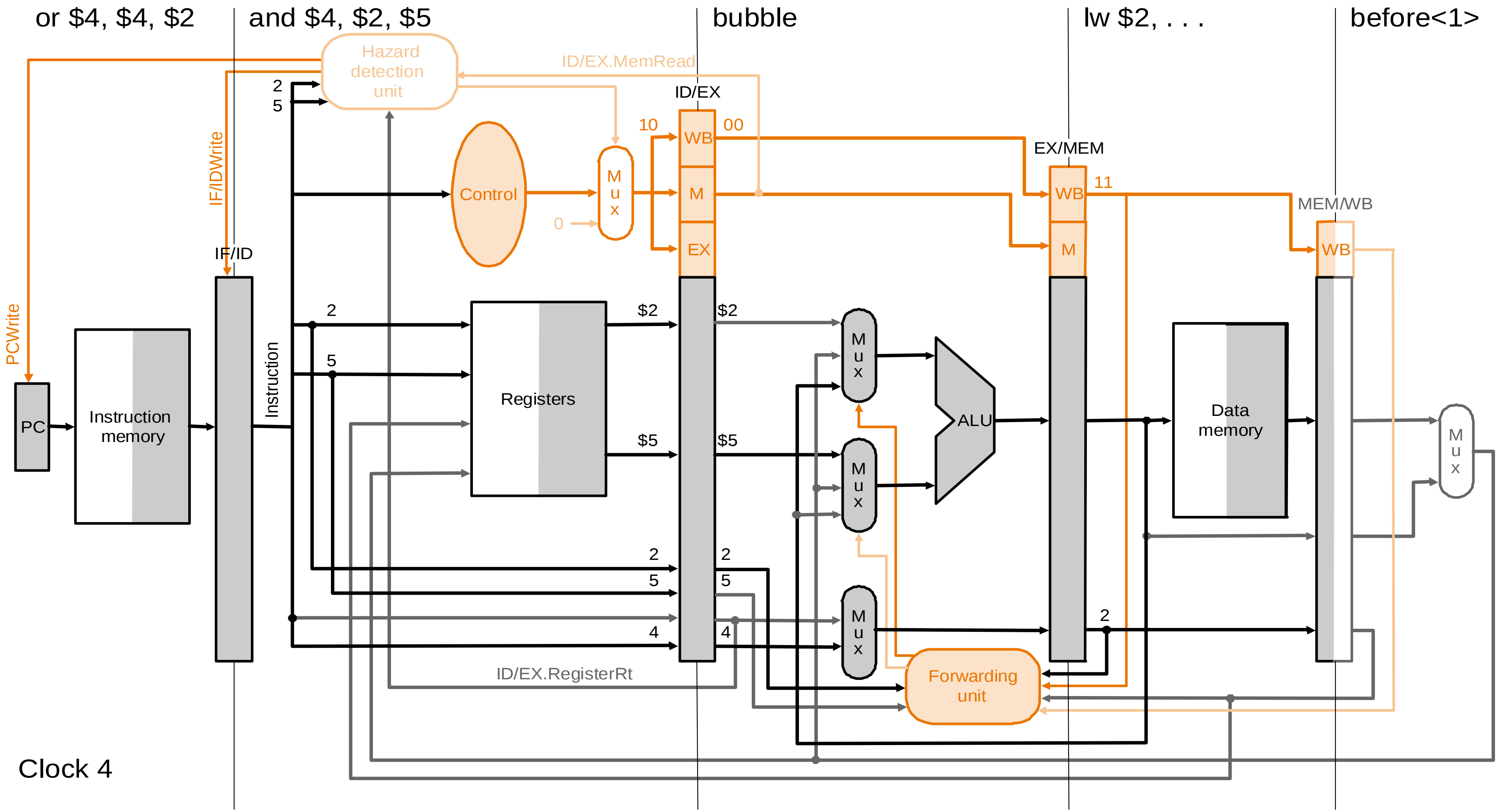
Cycle 3

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



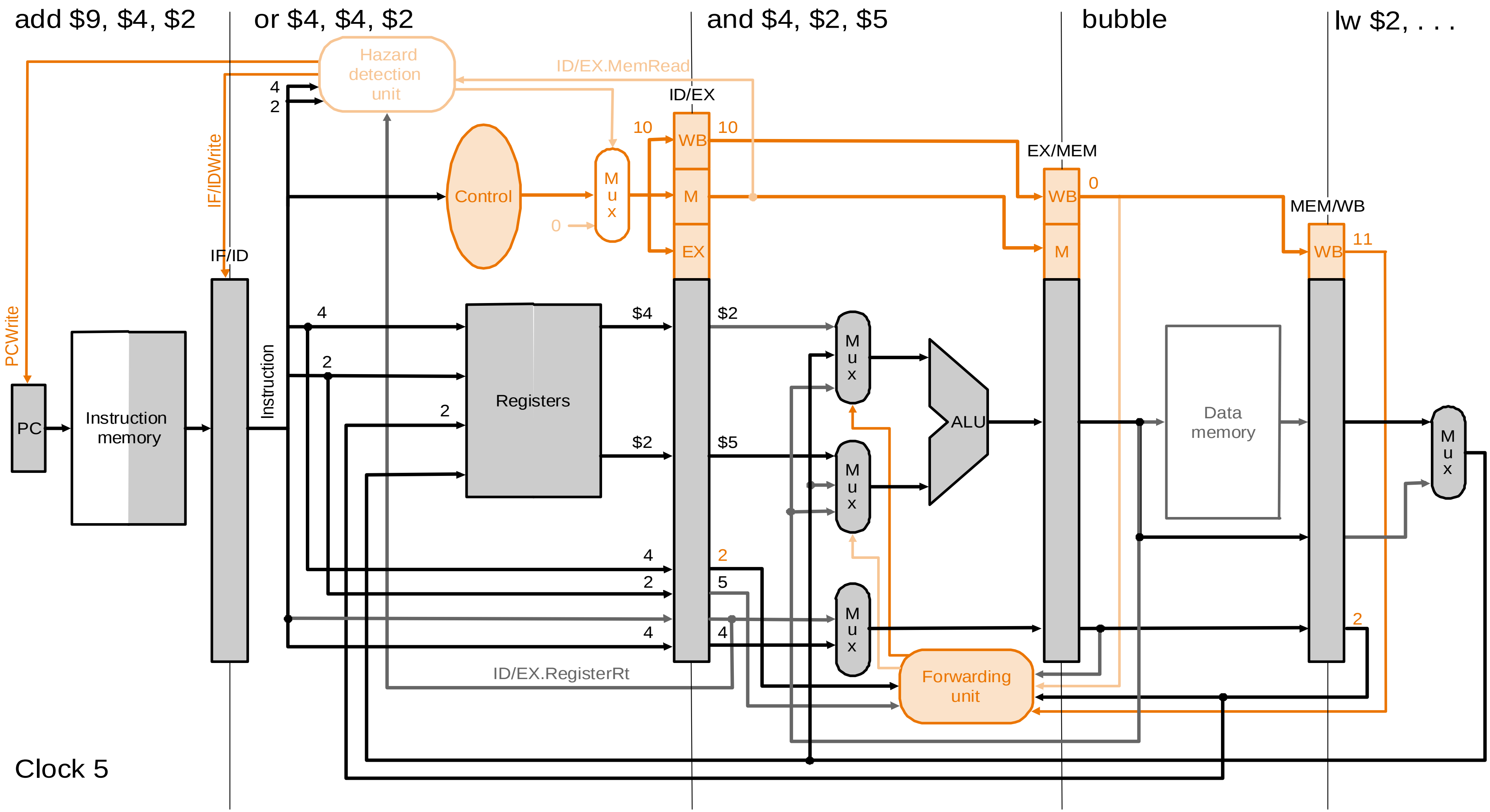
Cycle 4

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



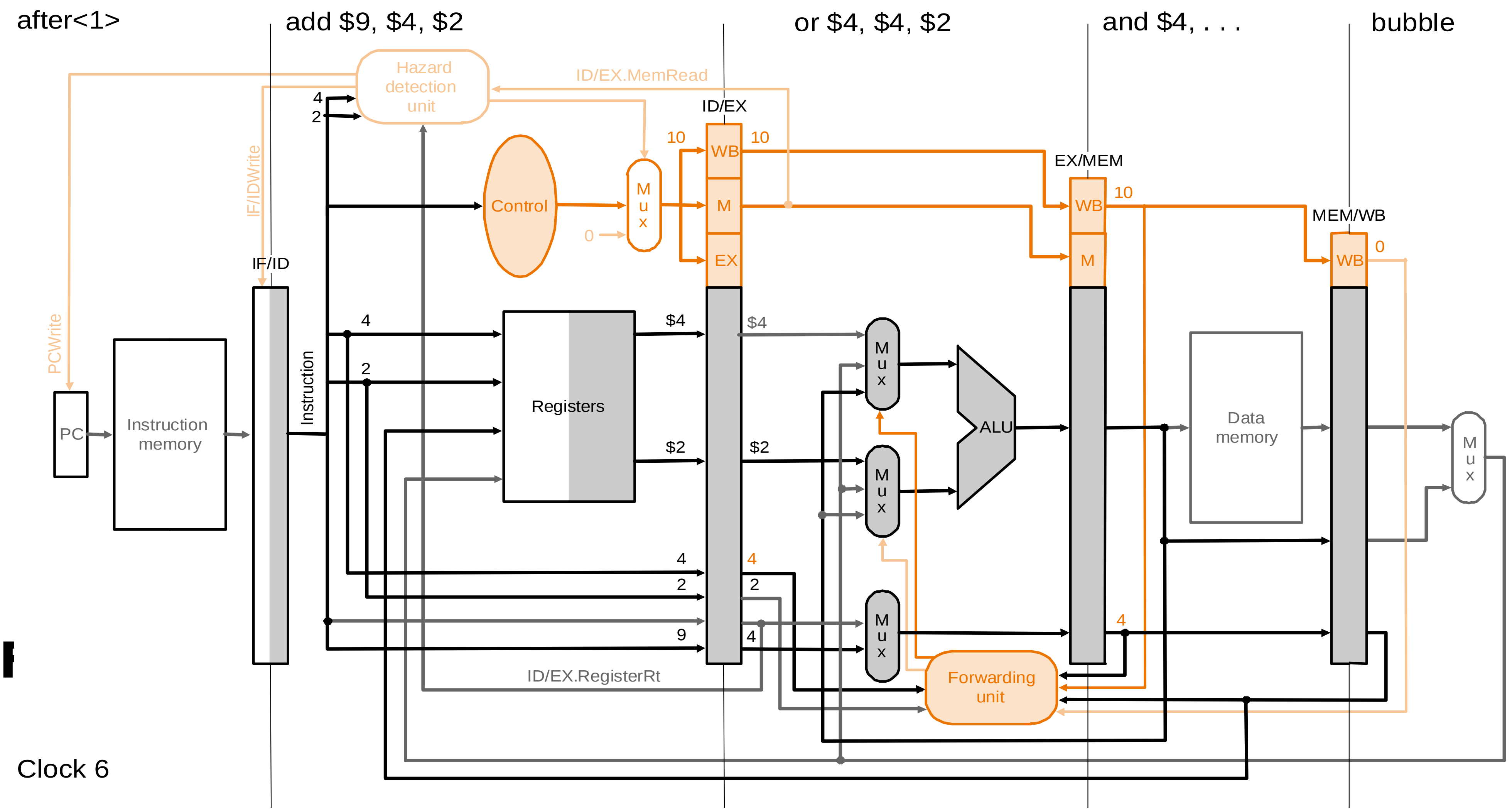
Cycle 5

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Cycle 6

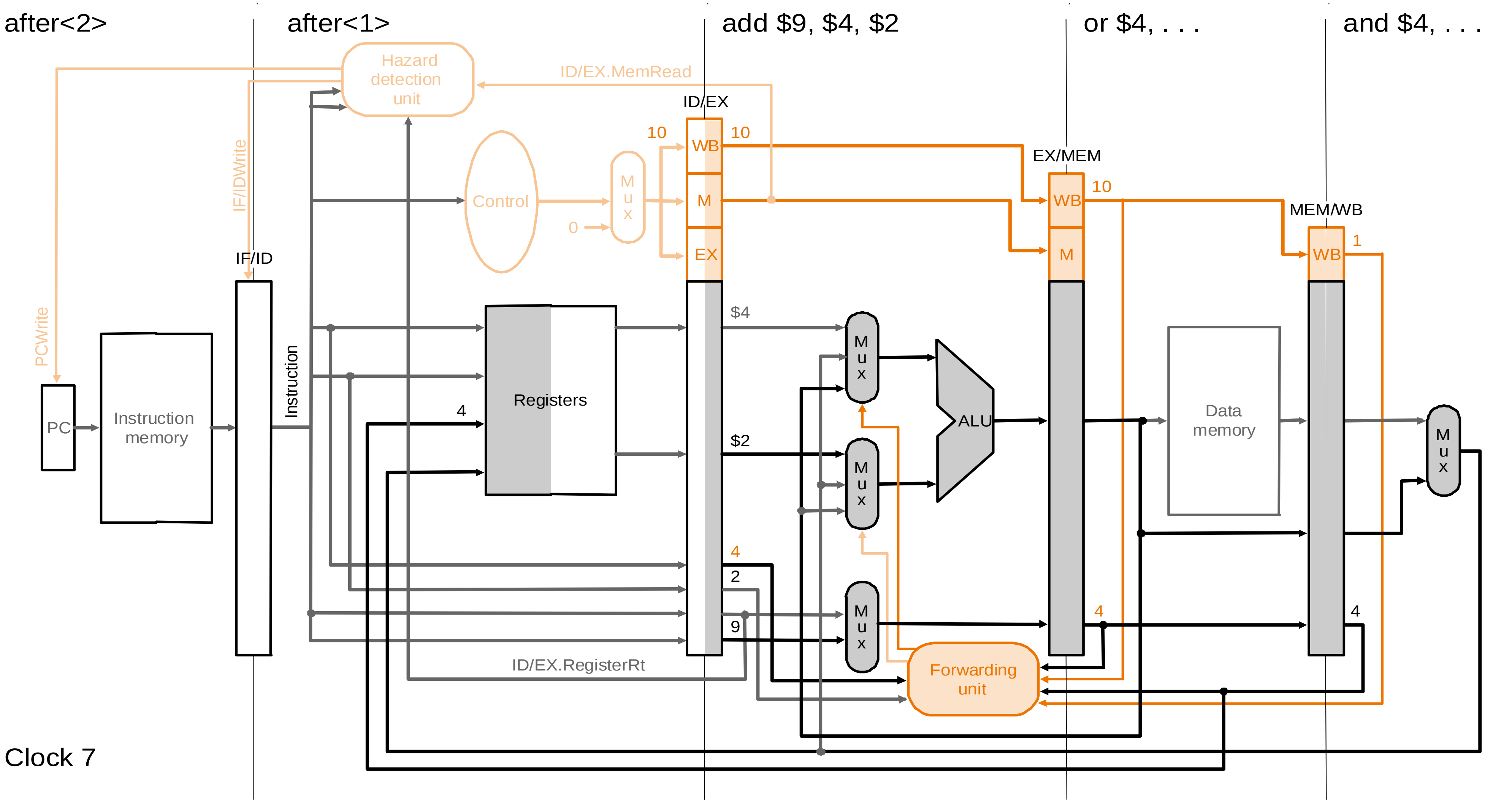
```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock 6

Cycle 7

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

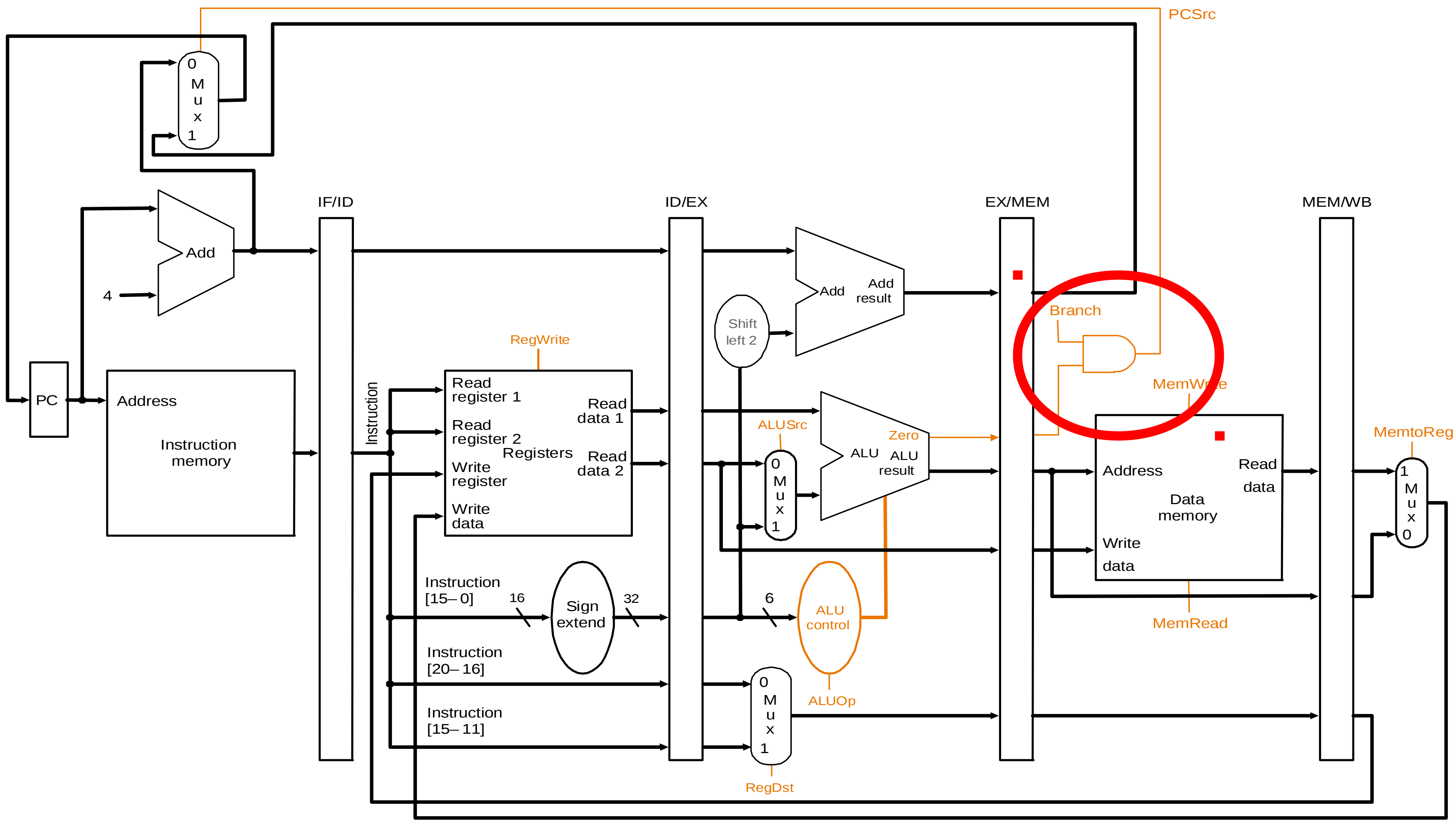


Clock 7

Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

Pipeline Datapath with Control Signals



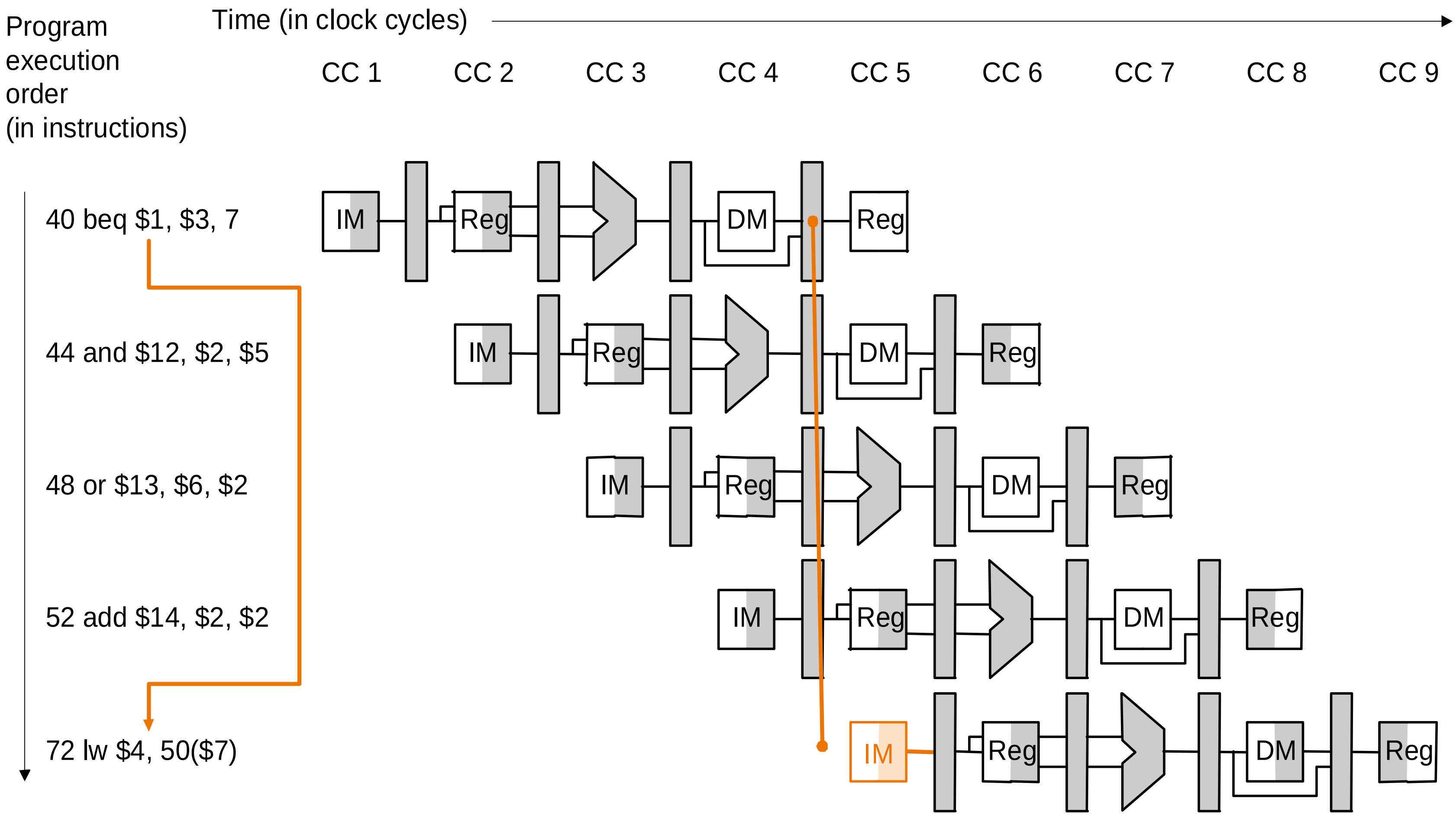
Control (or Branch) Hazards

- Problem with branches in the pipeline we have so far is that the branch decision is not made till the MEM stage – so what instructions, if at all, should we insert into the pipeline following the branch instructions?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong, have to flush the pipeline behind the branch
  - Need to flush 3 instructions (too many to degrade the pipeline performance)

Predicting Branch-not-taken



The outcome of branch taken (prediction wrong) is decided only when `beq` is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at `lw`

Flushing on Misprediction

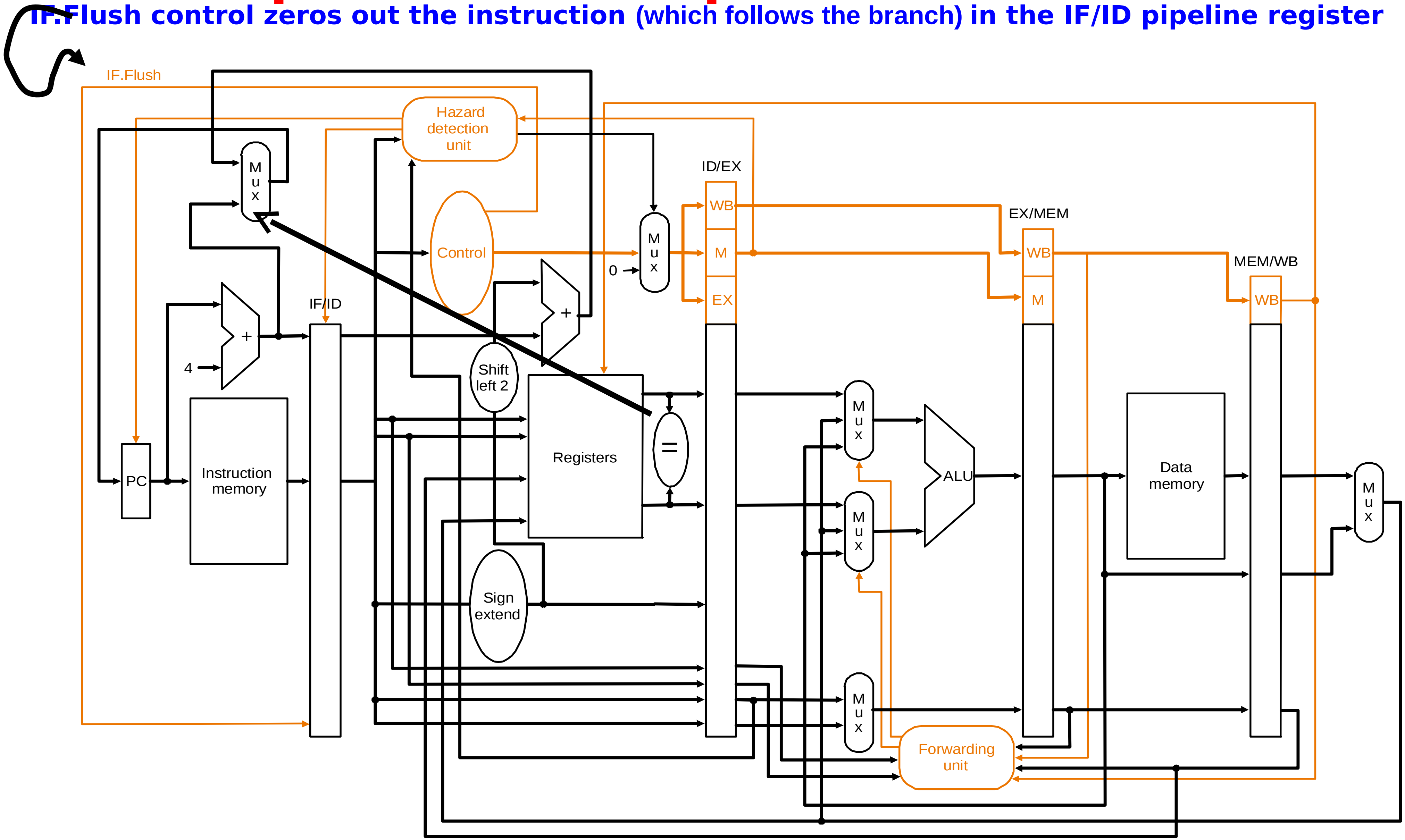
- Similar to the strategy as for stalling on load-use data hazard (RAW) ...
- Flush: Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline
- If branch decision made in the ID stage (discussed later), we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimizing the Pipeline to Reduce Branch Delay

- ***Move the branch decision*** **from the MEM stage (as in our current pipeline)** ***earlier to the ID stage***
  - **calculating the branch target address involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register**
  - **calculating the branch decision is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)**
  - **we must correspondingly make additions to the forwarding and hazard detection units to forward to or stall the branch at the ID stage in case the branch decision depends on an earlier result**

Optimized Datapath for Branch

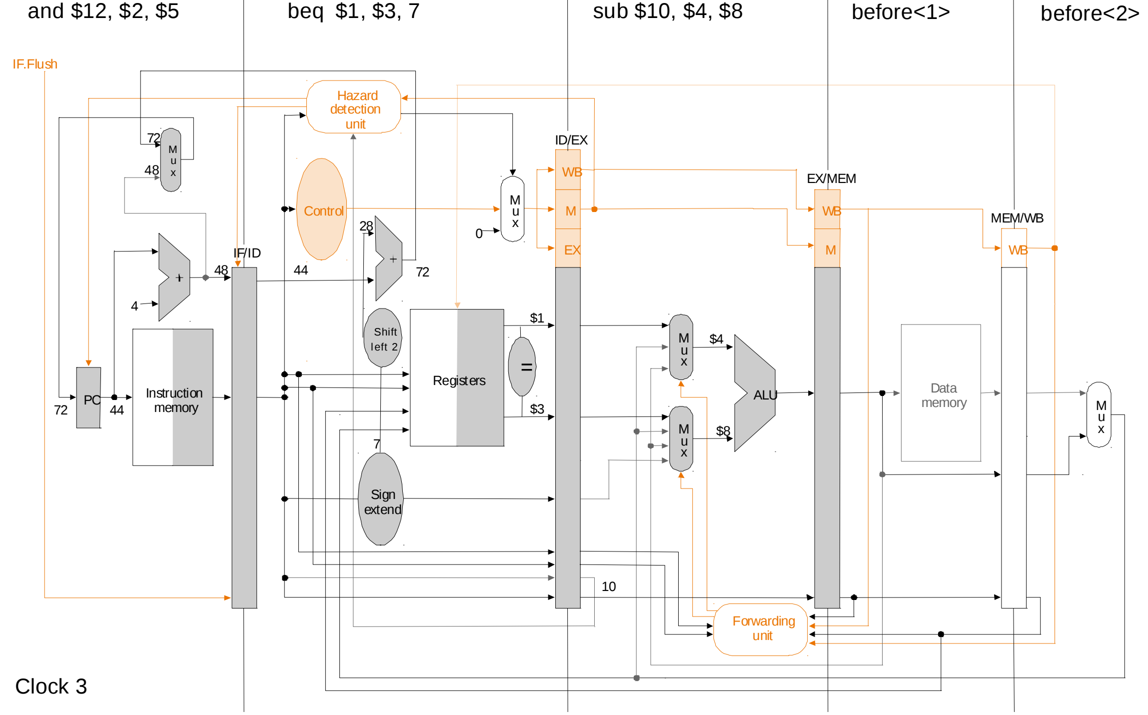
IF.Flush control zeros out the instruction (which follows the branch) **in the IF/ID pipeline register**

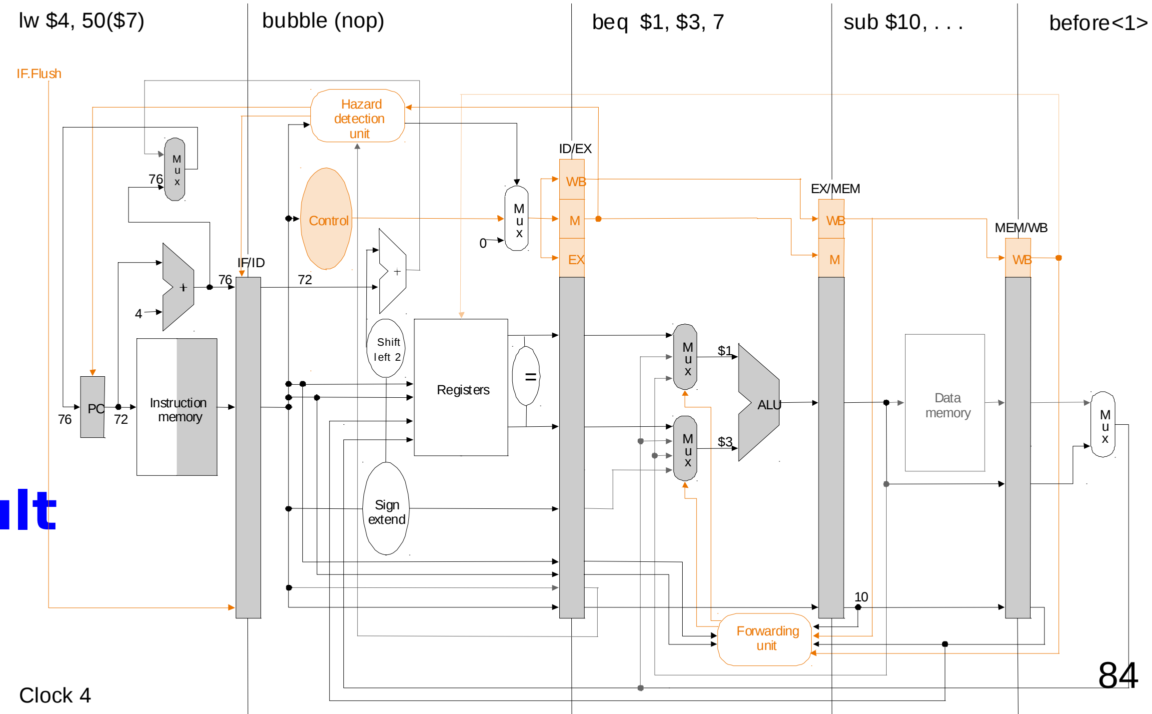


Branch decision is moved from the MEM stage to the ID stage – simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch (predict not taken)

```
36 sub $10, $4, $8
40 beq $1,  $3,  7
44 and $12  $2, $5
48 or  $13  $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7

…
72 lw  $4,  50($7)
```





Optimized pipeline with only one bubble as a result of the taken branch

Summary

- **Pipelined processor design**
  - **An overview of pipelining**
  - **A pipelined datapath**
  - **Pipelined control**

- **Problems with pipelined processor**
  - **Data hazards and forwarding**
  - **Data hazards and stalls**
  - **Branch hazards**