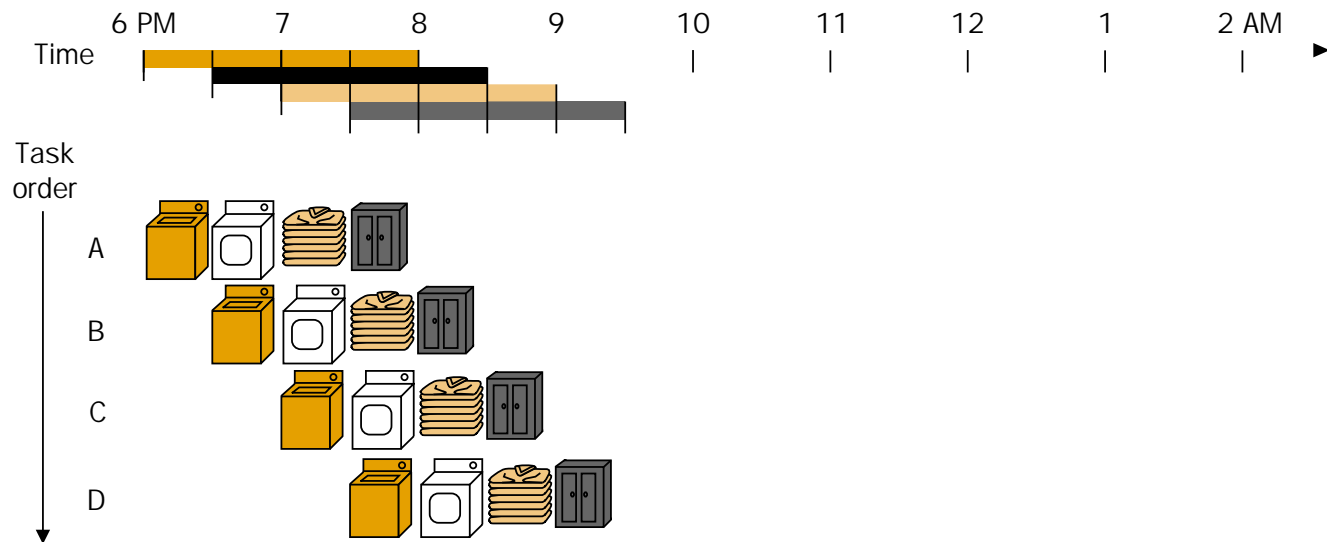
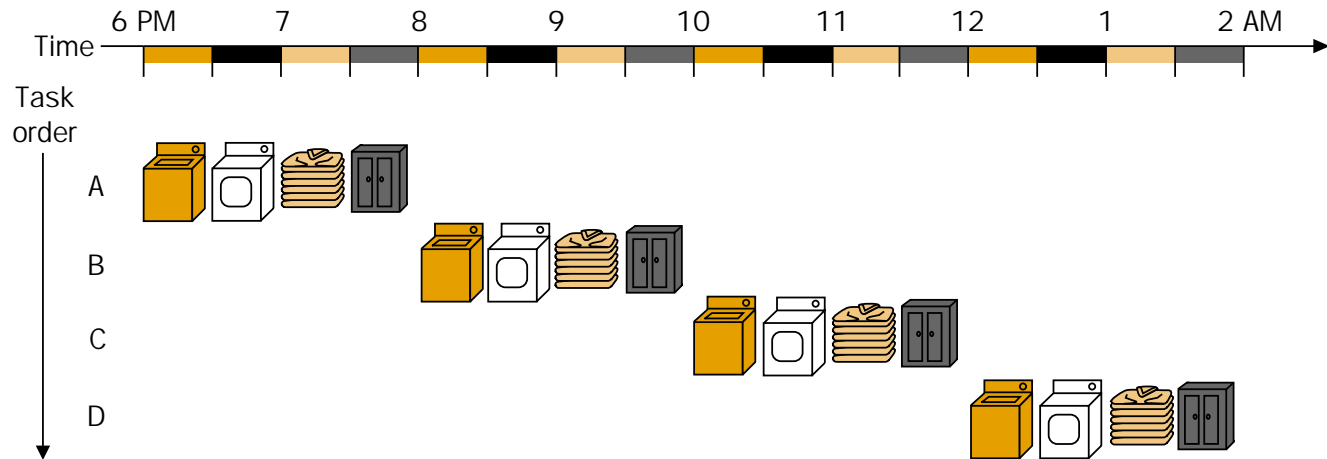


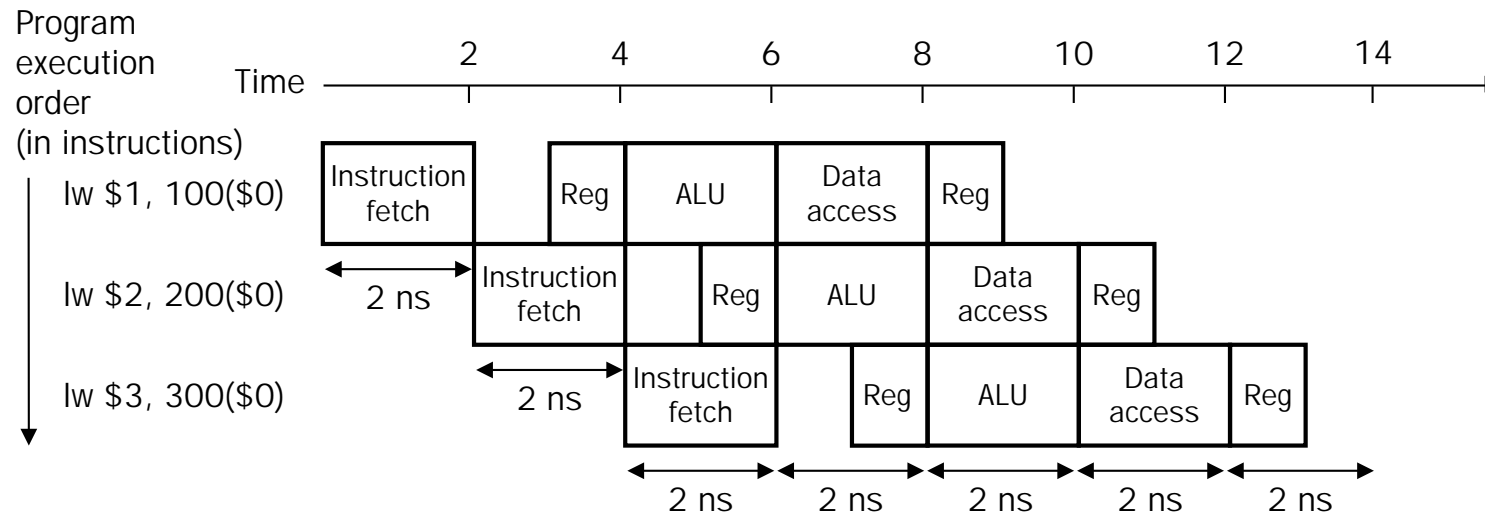
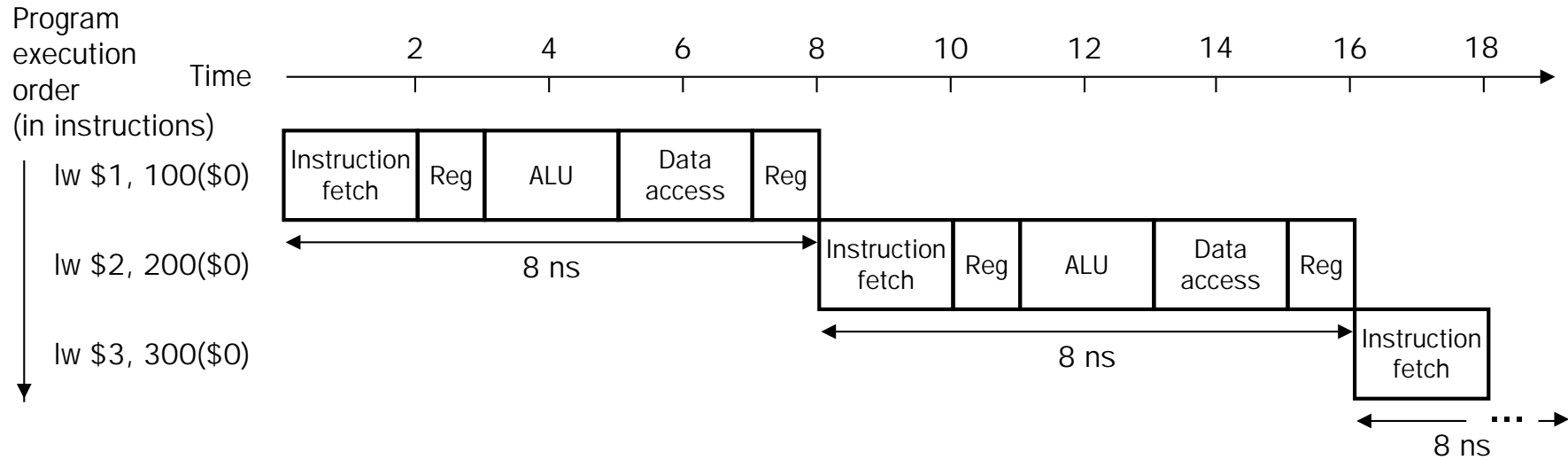
LAUNDRY: SEQUENTIAL vs. PIPELINED EXECUTION



PIPELINING

- In a pipelined computer architecture, a single processor can execute several instructions concurrently
 - ▷ Execution of one instruction uses several hardware functional units (instruction memory, register file, ALU, data memory, etc.)
 - ▷ Most functional units are used once per instruction
 - The register file may be read and written by the same instruction
 - ▷ The hardware functional units are organized into **stages**
 - Execution at each stage takes 1 clock period
 - Stages are separated by clock-controlled **pipeline registers** that preserve the state of execution for the duration of a clock period
 - ▷ The pipeline is subject to **hazards**
 - Data hazards: Write/read conflicts or timing problems
 - Control hazards: Exceptions and branches
- The MIPS R2000 pipeline design strongly influenced the design of all subsequent processors

COMPUTER: SEQUENTIAL vs. PIPELINED EXECUTION



PIPELINING (2)

● Pipeline speedup:

▷ A pipelined processor with s stages can execute n instructions in

$$ET_P = s + (n - 1) \text{ clock periods}$$

(assuming no hazards)

▷ A serial processor executes the same n instructions in

$$ET_S = ns \text{ clock periods}$$

▷ The ideal pipeline speedup equals the number of stages:

$$S_P = \frac{ET_S}{ET_P} = \frac{ns}{s + (n - 1)} \xrightarrow{n \gg s} s$$

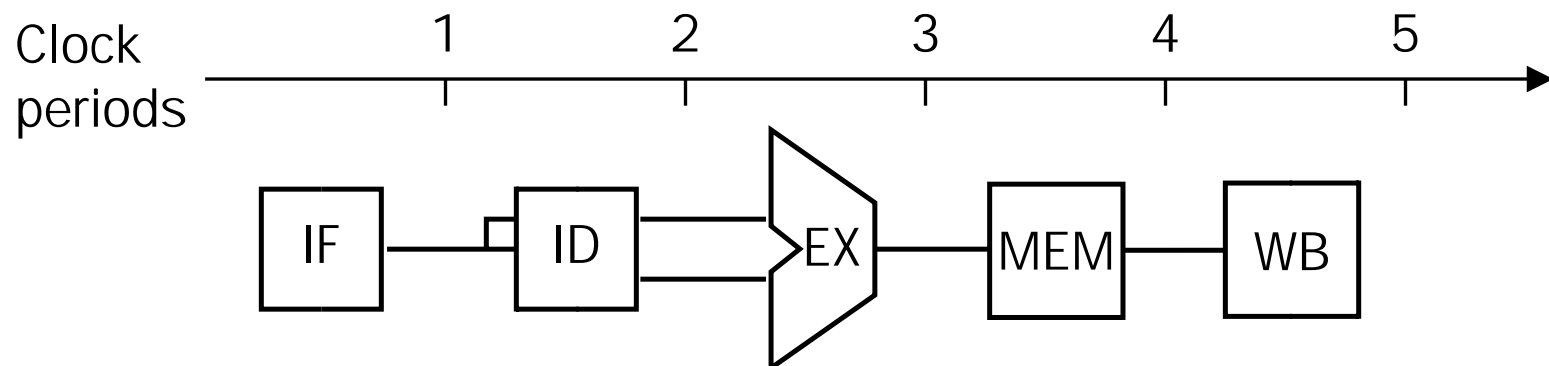
● Amdahl's law applies to pipelining

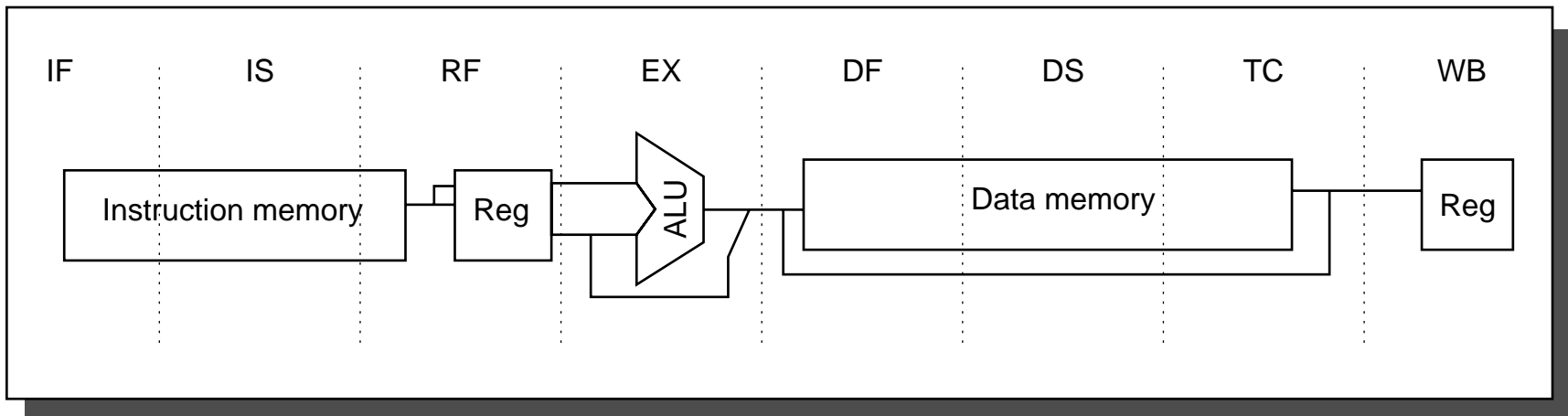
PROGRAMMING IMPLICATIONS OF PIPELINING

- Avoid function or subprogram calls in an inner loop
 - ▷ Jumps force the pipeline to be flushed
- Avoid recursion in an inner loop
 - ▷ Recursion on the elements of an array generally causes data hazards because the value of $v[n]$ has not been written before it is needed for the computation of $v[n+1]$
- Avoid scalar temporary variables in an inner loop
 - ▷ Reading a memory-resident scalar variable may cause a data hazard
- Avoid **case** and **switch** statements in an inner loop
 - ▷ Conditional branches cause control hazards, and the use of a jump table may cause data hazards

MIPS PIPELINES (1)

- MIPS R2000 integer unit pipeline stages
(Patterson & Hennessy, Chapter 6)
 1. Instruction Fetch (IF)
 2. Instruction Decode (ID) and Register Fetch
 3. Execute (EX or ALU)
 - ▷ ALU operations, condition evaluation, address computation
 4. Memory access (MEM)
 5. Write back (WB) to register file

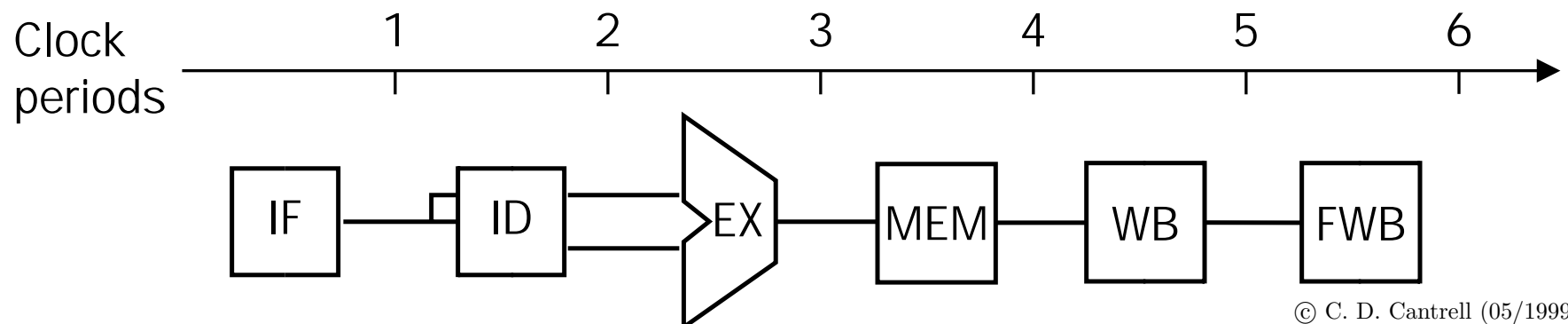




The eight-stage pipeline of the R4000

MIPS PIPELINES (2)

- MIPS R2000 floating-point unit pipeline stages
 1. Instruction Fetch (IF)
 2. Register Fetch and Instruction Decode (RD)
 - ▷ FPU decodes instruction on bus to see if it's floating-point
 - ▷ FPU reads data from its registers
 3. Execute (EX or ALU)
 4. Memory access (MEM)
 5. Exception processing (stage called WB for correspondence with integer pipeline)
 6. Write back (FWB)



DATA HAZARDS (1)

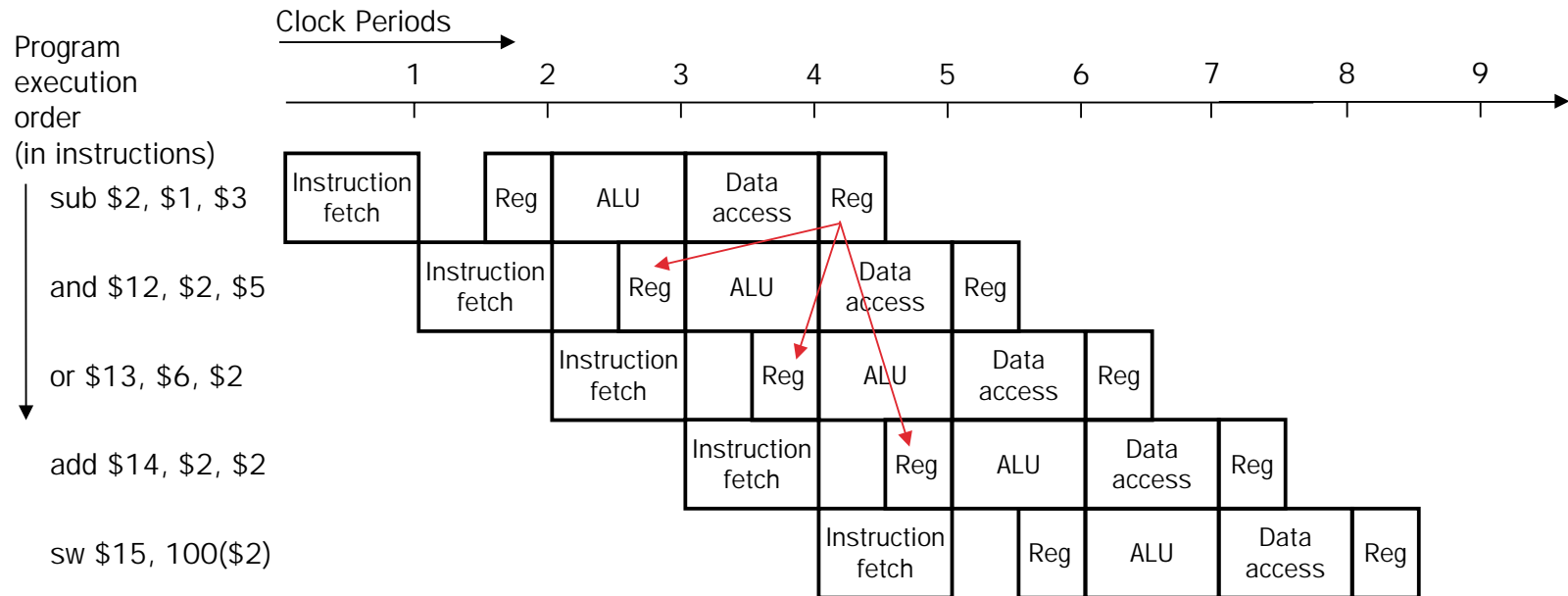
- Data hazards occur when the order of read and write actions is not the order in strictly sequential execution
 - ▷ Hazards are named by the ordering in the program that must be preserved in the course of pipelined execution
 - In the following, the order of execution should be i , then j
 - ▷ RAW (read after write) — j reads a source before i has written it
 - j incorrectly gets the old value
 - Most common kind of data hazard
 - ▷ WAR (write after read) — j writes a destination before i reads it
 - i incorrectly gets the new value
 - ▷ WAW (write after write) — j should write an operand after i writes it, but the writes are performed in the wrong order, incorrectly leaving the value written by i
- Hazards limit pipeline speedup and complicate design

DATA HAZARDS (2)

- RAW hazards are generated by the instructions

```

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



DATA HAZARDS (3)

- RAW hazards with $i = \text{sub } \$2, \$1, \$3$ and source = $\$2$ are generated by the instructions

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

- ▷ The instruction $j = \text{and } \$12, \$2, \$5$ reads $\$2$ before i writes it
- ▷ The instruction $j = \text{or } \$13, \$6, \$2$ reads $\$2$ before i writes it
- ▷ The instruction $j = \text{add } \$14, \$2, \$2$ reads $\$2$ in the same clock period in which i writes it
 - Generates a hazard if the register file's outputs change only on the edge of the main processor clock

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (1)

- Follows ideas of K. Kennedy et al.
- Definition: If control flow within a program can reach statement T after passing through statement S , then **T depends on S**
 - ▷ Dependence is always defined by reference to the results of *serial* execution
- Assume a loop
 - ▷ Dependence analysis outside a loop is trivial
- Assume an array
 - ▷ Dependences that affect pipelining arise from references to the same memory location $M[n]$ in an array

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (2)

- Distinguish between statements (in a program) and instances of those statements (in loop instances or threads)
 - ▷ Let i = loop induction variable
 - Example: A `for` loop in C
 - ◊ Syntax: `for (i=0; i<n; i++) ...`
 - ◊ The induction variable is `i`
 - ▷ Let S_i = instance of statement S that occurs on the value i of the induction variable
- **Flow dependence (RAW):** S_{i_S} writes $M[n]$ and T_{i_T} reads $M[n]$

$$\begin{array}{l}
 S: \quad X[f_S[i]] = \dots \\
 T: \quad \dots = F[X[f_T[i]]]
 \end{array}$$

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (3)

- **Anti-dependence (WAR):** S_{i_S} reads $M[n]$ and T_{i_T} writes $M[n]$

S: $\dots = F[X[f_S[i]]]$

T: $X[f_T[i]] = \dots$

- **Output dependence (WAW):** S_{i_S} and T_{i_T} both write $M[n]$

S: $X[f_S[i]] = \dots$

T: $X[f_T[i]] = \dots$

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (4)

- Requirement for the existence of an instance of dependence:
A real memory location $M[\mathbf{n}]$ exists such that

$$M[\mathbf{n}] = \mathbf{f}_S[i_S] = \mathbf{f}_T[i_T]$$

where S is executed before T and both values of the induction variable are in the range of the loop:

$$p \leq i_S \leq i_T \leq q$$

- Example: f_S and f_T are linear functions

$$f_S[i] = a_S i + b_S, \quad f_T[i] = a_T i + b_T$$

The requirement for dependence implies that

$$a_S i_S + b_S = a_T i_T + b_T$$

$$\Rightarrow a_S i_S - a_T i_T + (b_S - b_T) = 0$$

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (5)

● Example:

```
do 100 i=2,100
T:   b(i)=a(i-1)
S:   a(i)=c(i)
```

▷ Here, $f_S(i) = i$, $f_T(i) = i - 1$

▷ Condition $f_S(i_S) = f_T(i_T)$ is $i_S = i_T - 1 \Rightarrow$ $i_S < i_T$
(hence T depends on S)

▷ The equation $i_S = i_T - 1$ has lots of solutions such that
 $2 \leq i_S < i_T \leq 100$

DEPENDENCE ANALYSIS FOR PIPELINED LOOPS (6)

- In *serial* execution, T_3 reads from $a(2)$ after S_2 writes to $a(2)$:

$$i = 2: \quad T_2: b(2)=a(1)$$
$$S_2: a(2)=c(2)$$

$$i = 3: \quad T_3: b(3)=a(2)$$
$$S_3: a(3)=c(3)$$

- In *vector* execution, T_3 reads from $a(2)$ before S_2 writes to it:

$$b(2)=a(1)$$

$$b(3)=a(2)$$

$$\vdots$$

$$a(2)=c(2)$$

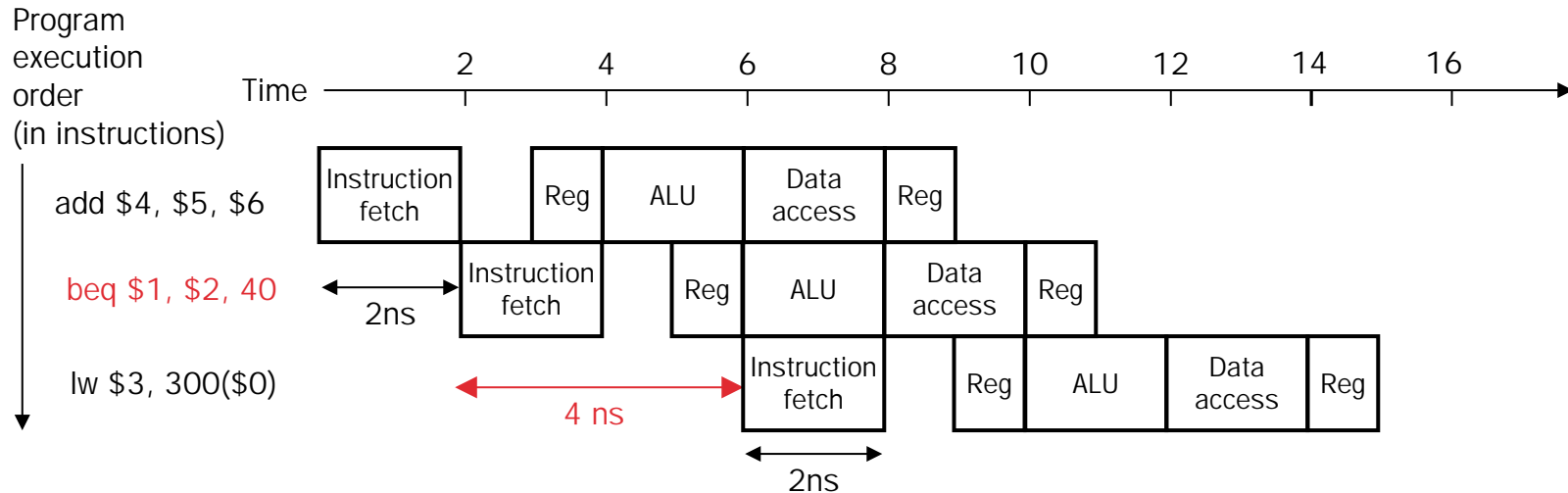
$$a(3)=c(3)$$

CONTROL HAZARDS

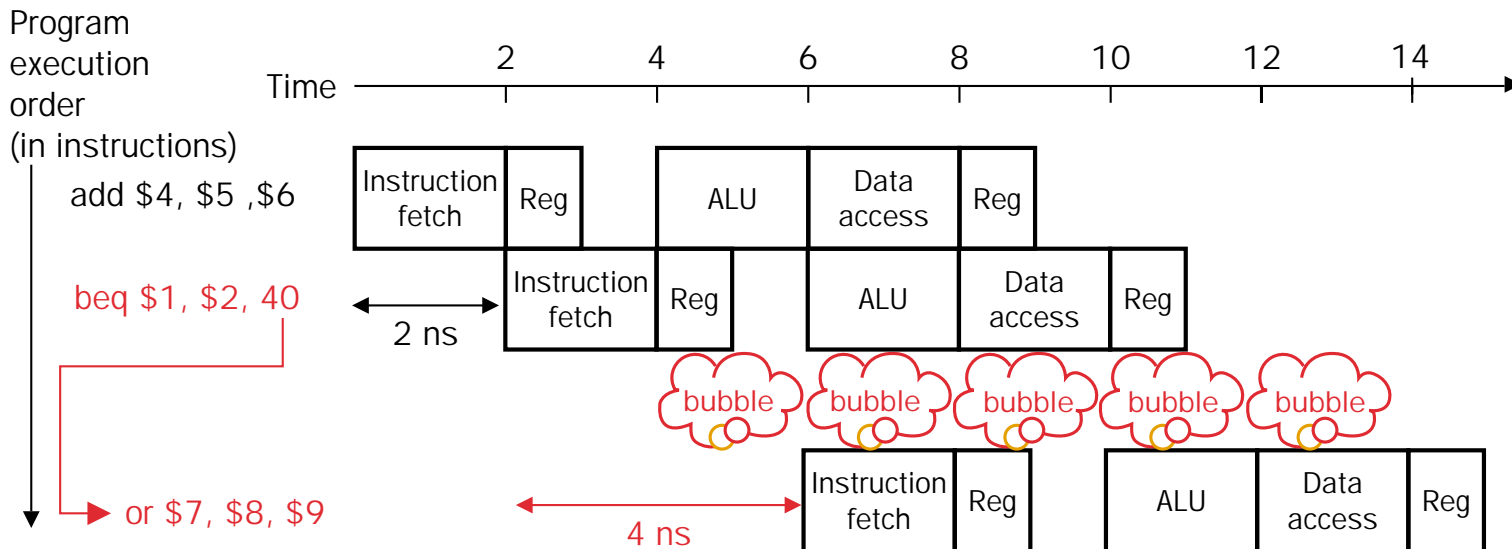
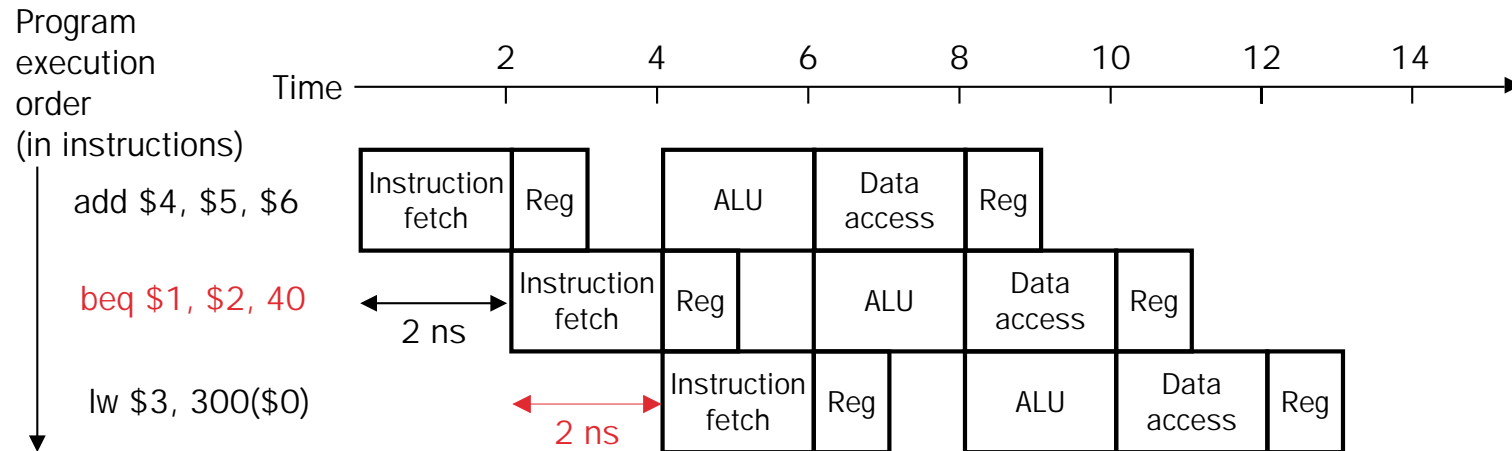
- A control hazard occurs because a branch or jump instruction needs to make a decision based on the results of operations or instructions that are still pending
 - ▷ A **beq** instruction cannot update the PC before the test for equality has completed
 - We'll see later that it's possible to make the decision at the Instruction Decode/Register Fetch stage instead of the ALU stage
 - ▷ In the MIPS ISA, only the instruction immediately following the branch is executed or not executed, depending on the branch decision
 - This isn't true in longer pipelines
 - ▷ Methods for dealing with control hazards
 - Stall
 - Predict the branch
 - Always execute the instruction following the branch

STALLING AS A SOLUTION FOR CONTROL HAZARDS

- After a conditional branch (**beq**), there is a one-stage pipeline stall (bubble)

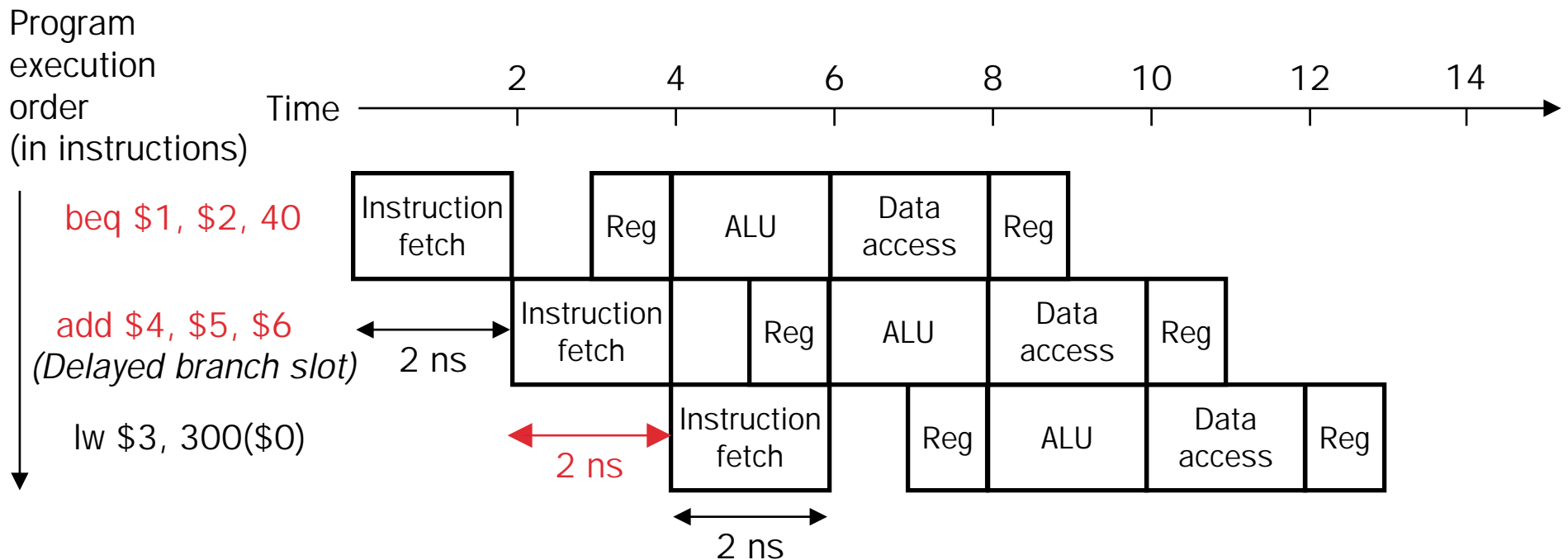


PREDICTING BRANCHES NOT TAKEN AS A SOLUTION

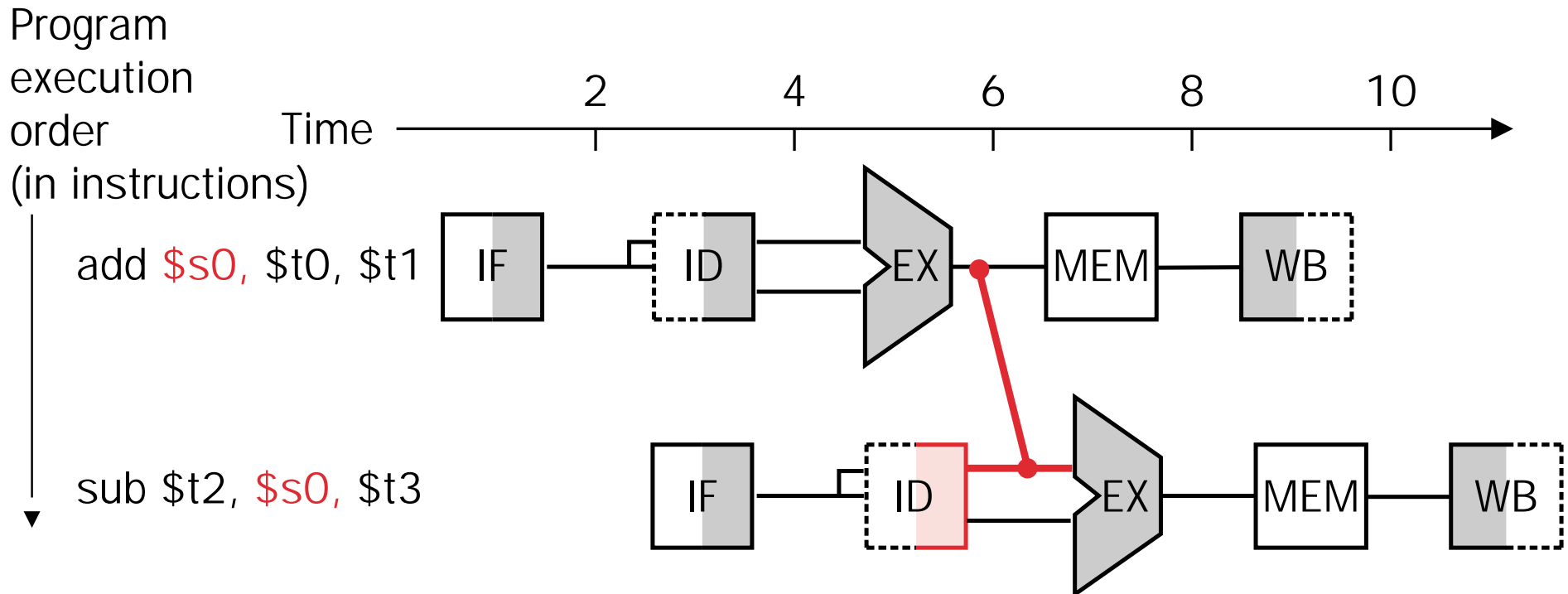


PIPELINE DELAYED BRANCH AS A SOLUTION

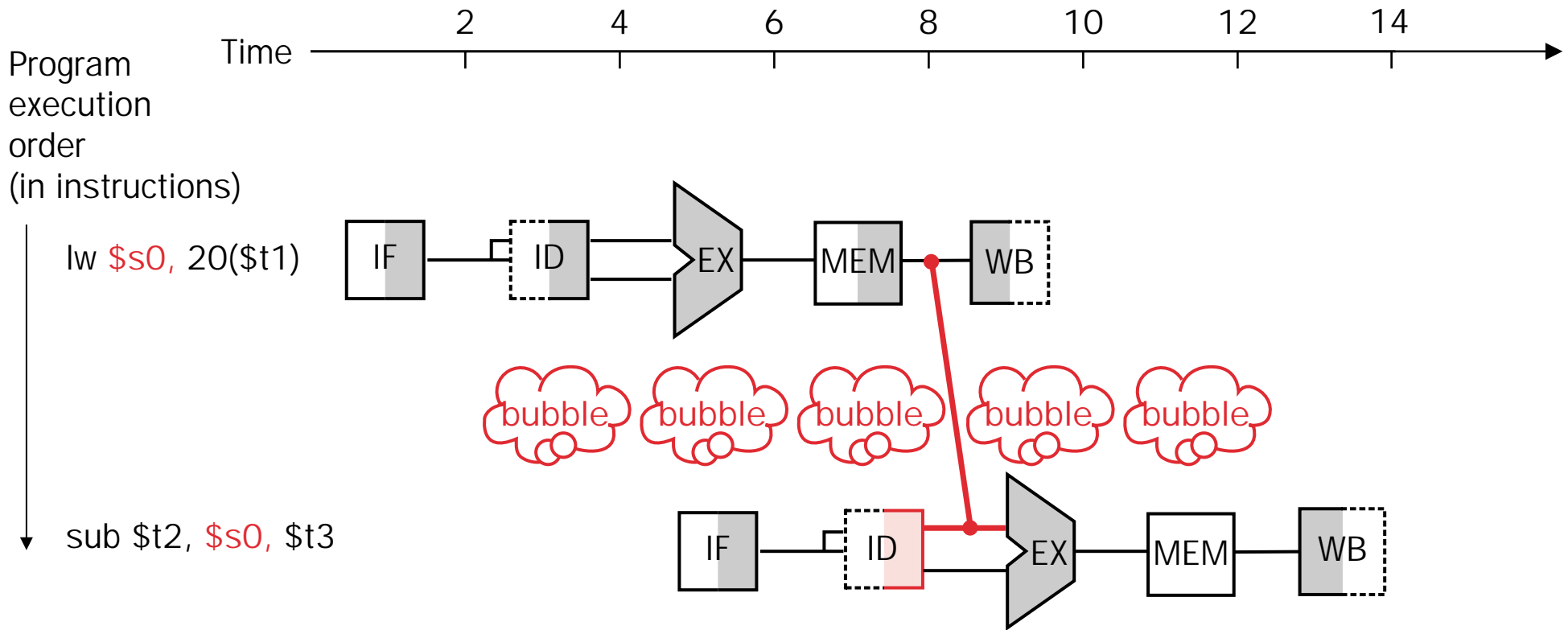
- After the conditional branch (**beq**), there is an **add** instruction (which can do useful work) instead of a stall (bubble), which does nothing



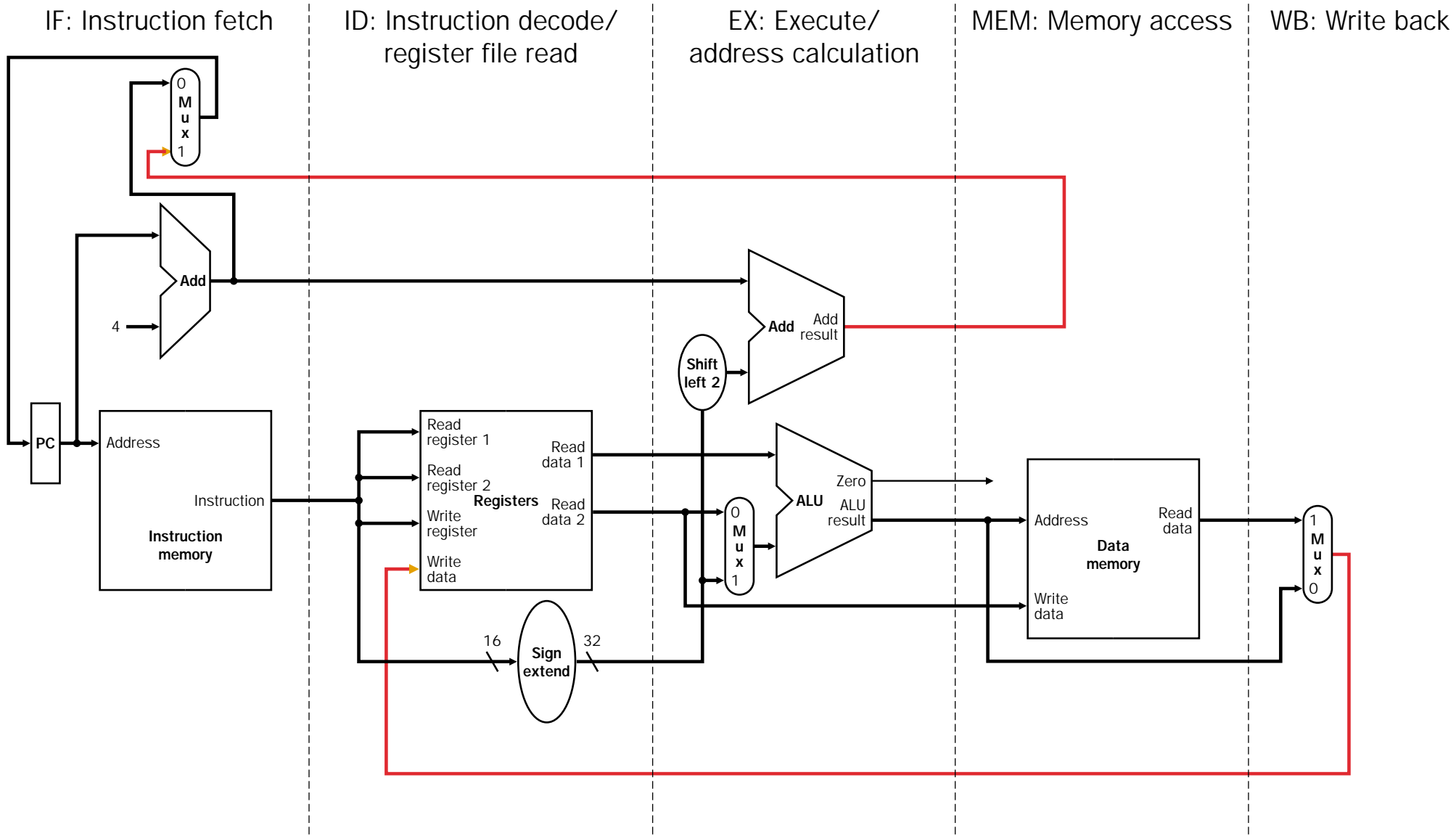
FORWARDING AS A SOLUTION FOR DATA HAZARDS (1)



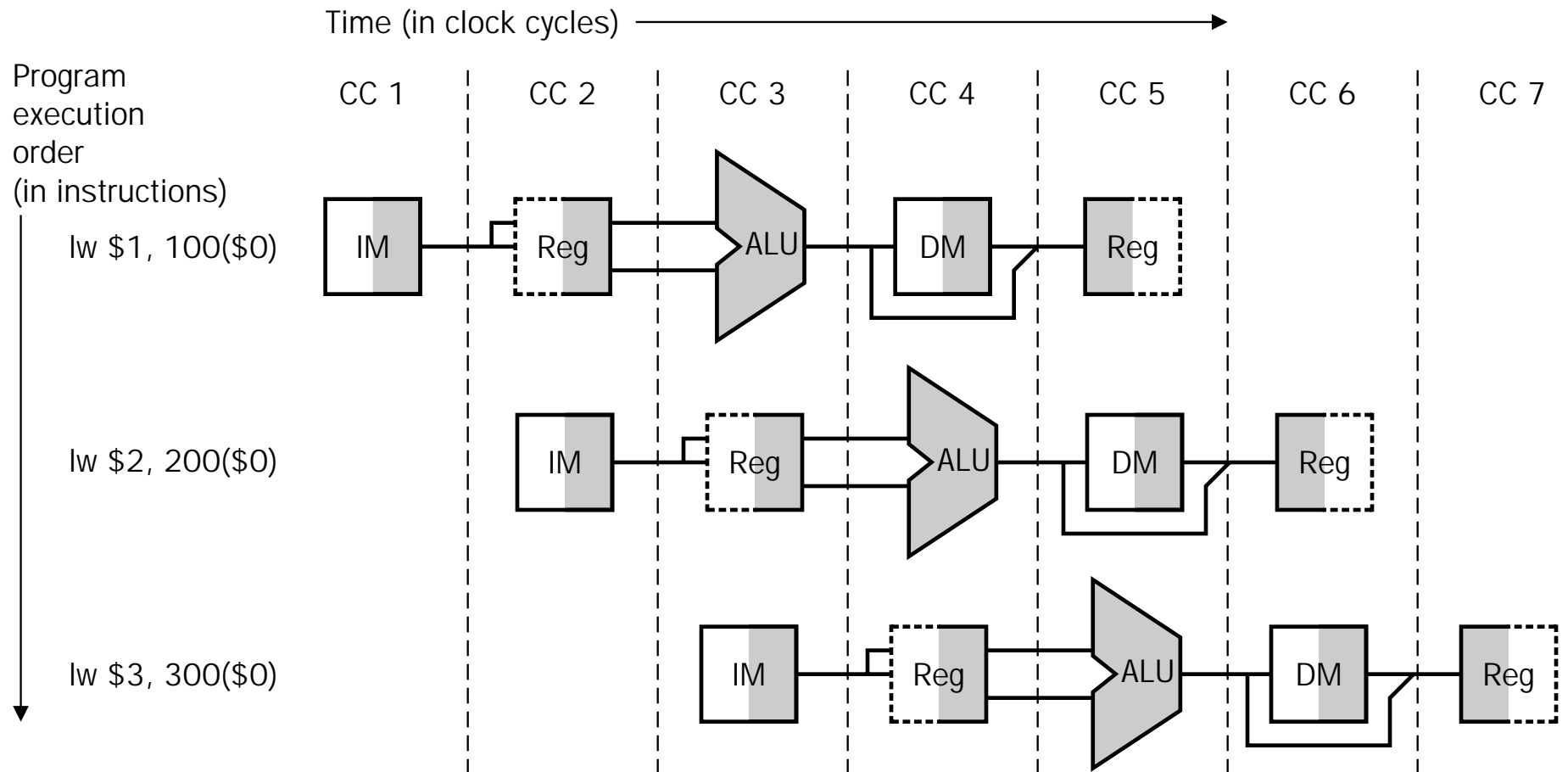
FORWARDING AS A SOLUTION FOR DATA HAZARDS (2)



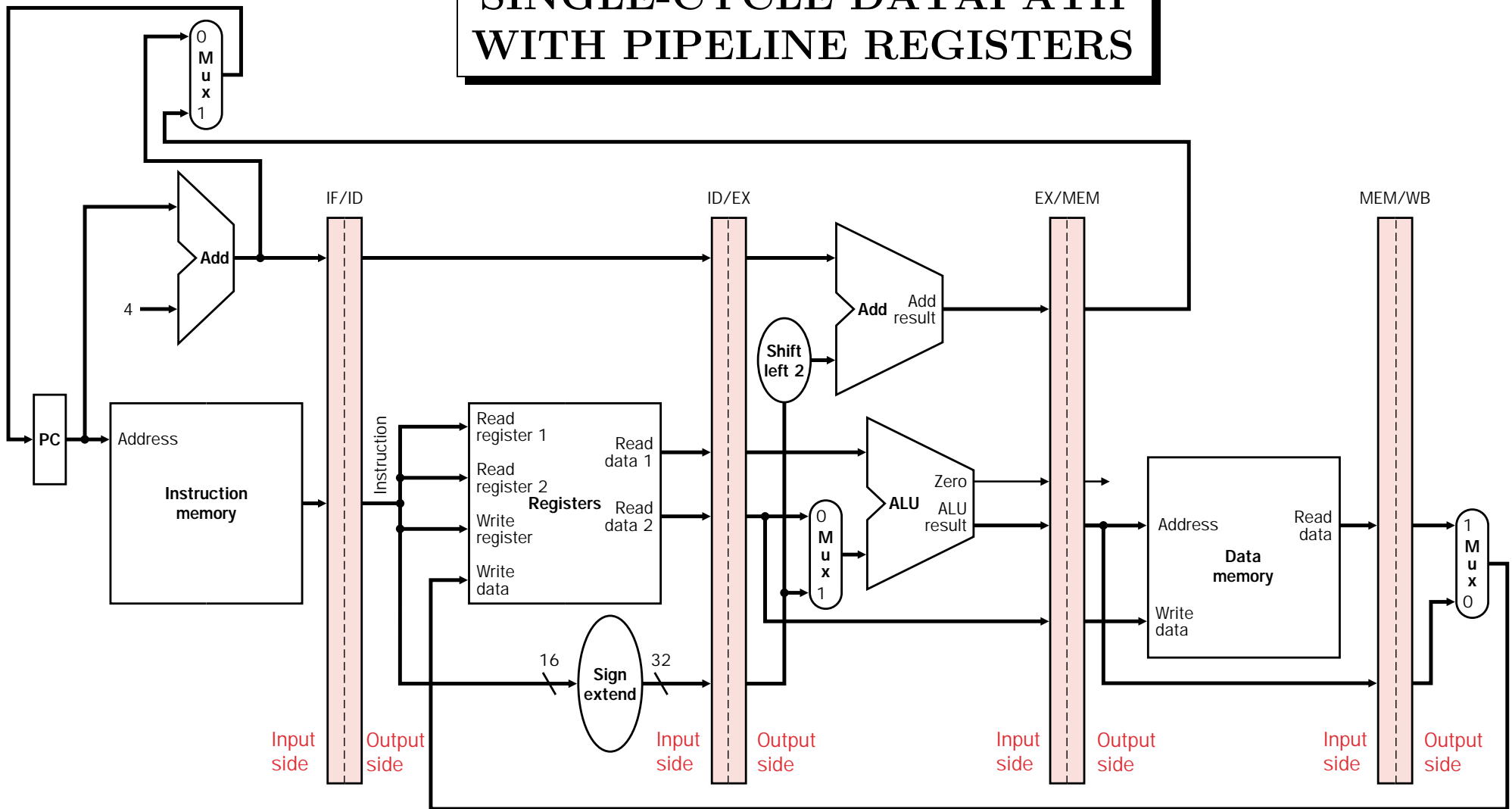
SINGLE-CYCLE DATAPATH



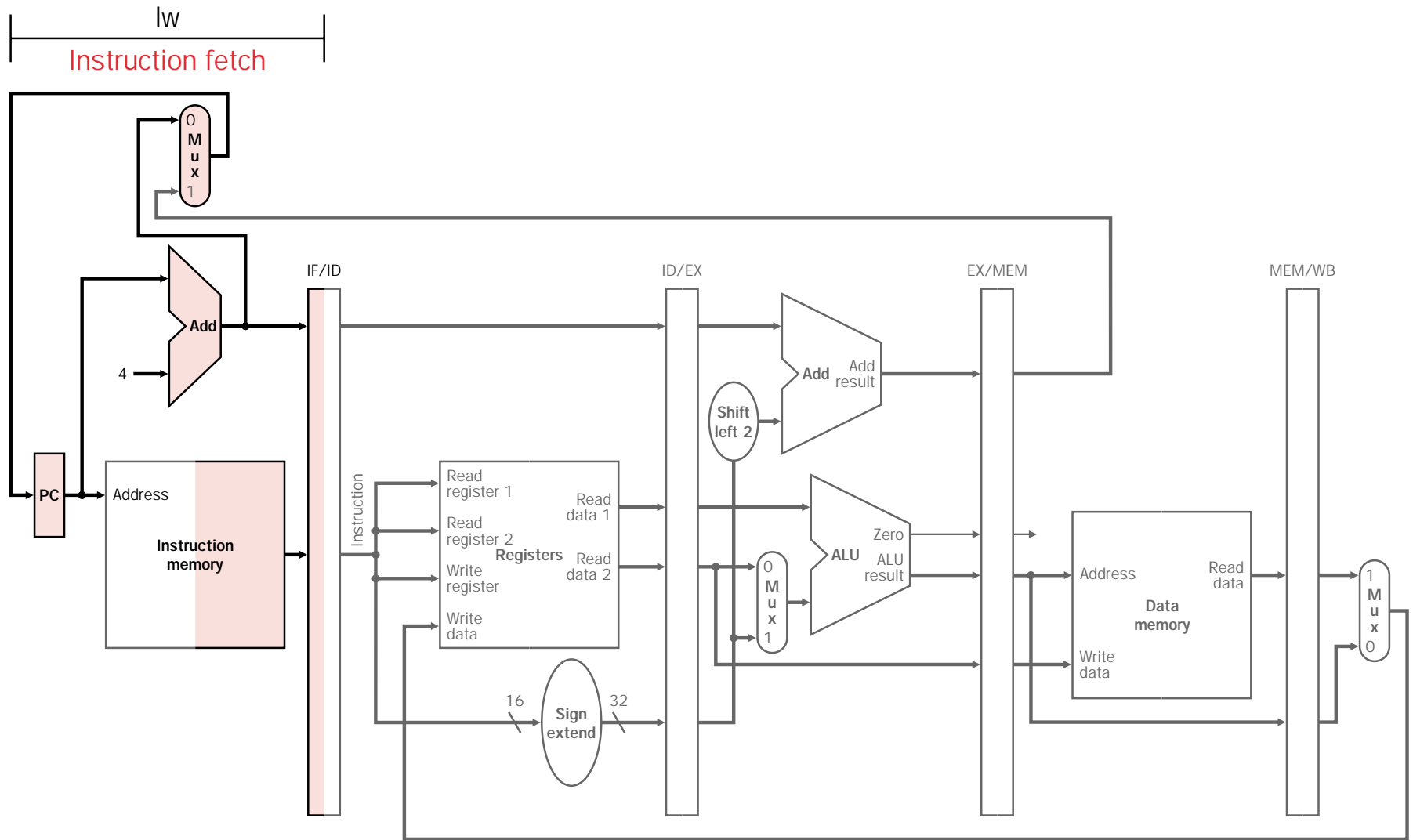
PIPELINED EXECUTION IN SINGLE-CYCLE DATAPATH

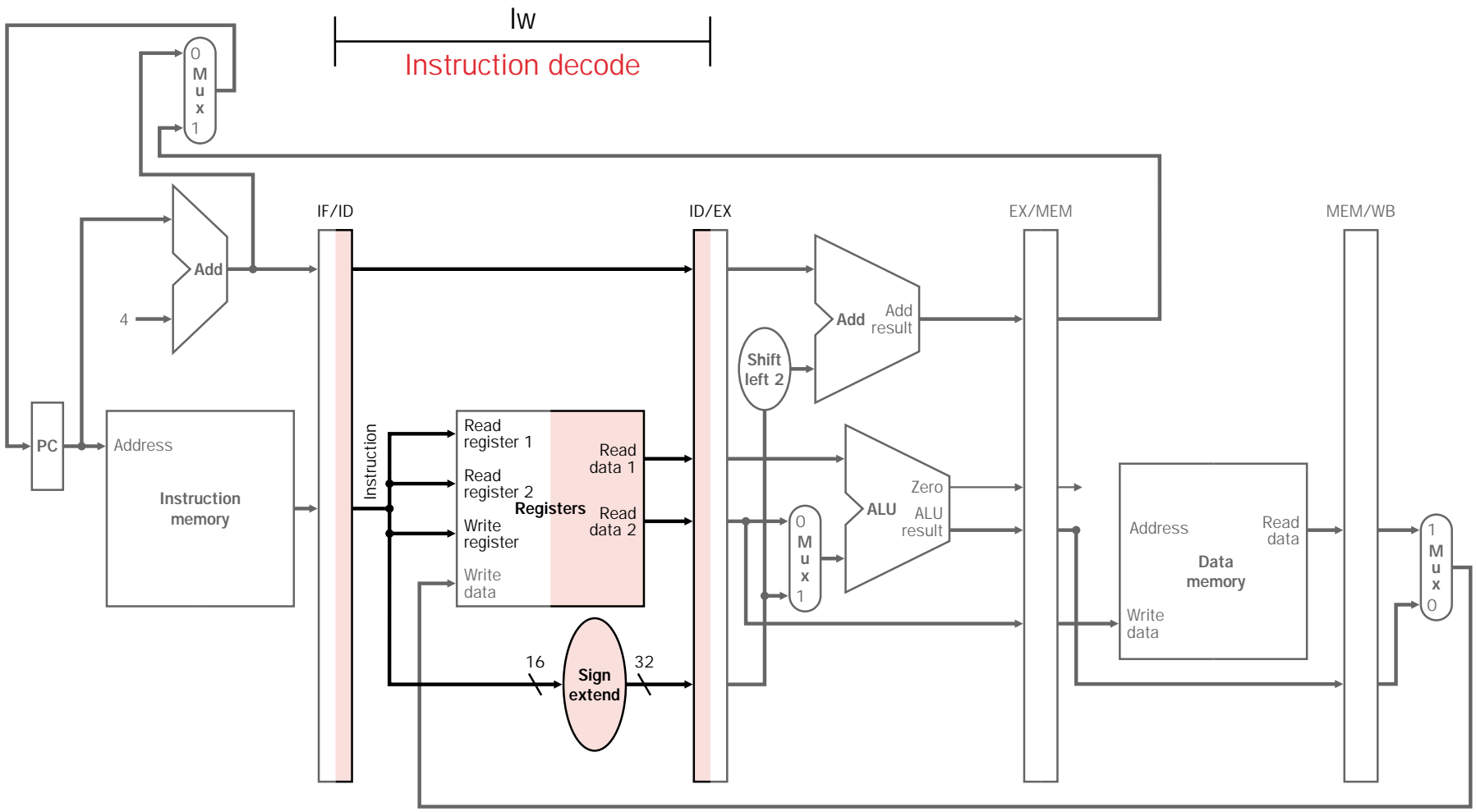


SINGLE-CYCLE DATAPATH WITH PIPELINE REGISTERS



Because the state of a D flip-flop changes only on clock edges, new data can be asserted on the inputs of the pipeline registers while the data written in the previous clock period is still valid on the outputs





lw
Instruction decode

Mux
1

Add
4

IF/ID

ID/EX

EX/MEM

MEM/WB

PC
Address
Instruction memory

Instruction
Read register 1
Read data 1
Read register 2
Read data 2
Registers
Write register
Write data

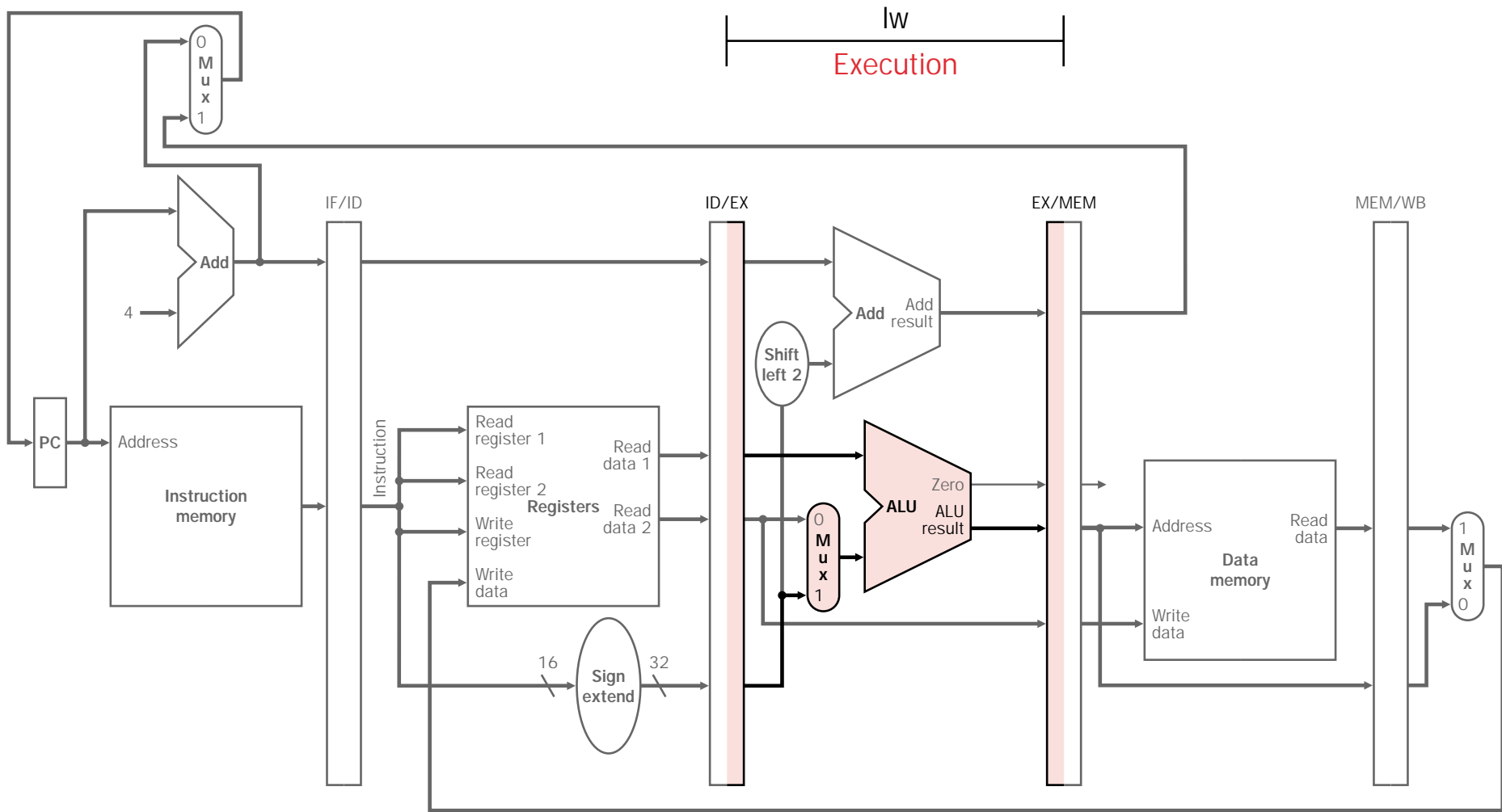
16
Sign extend
32

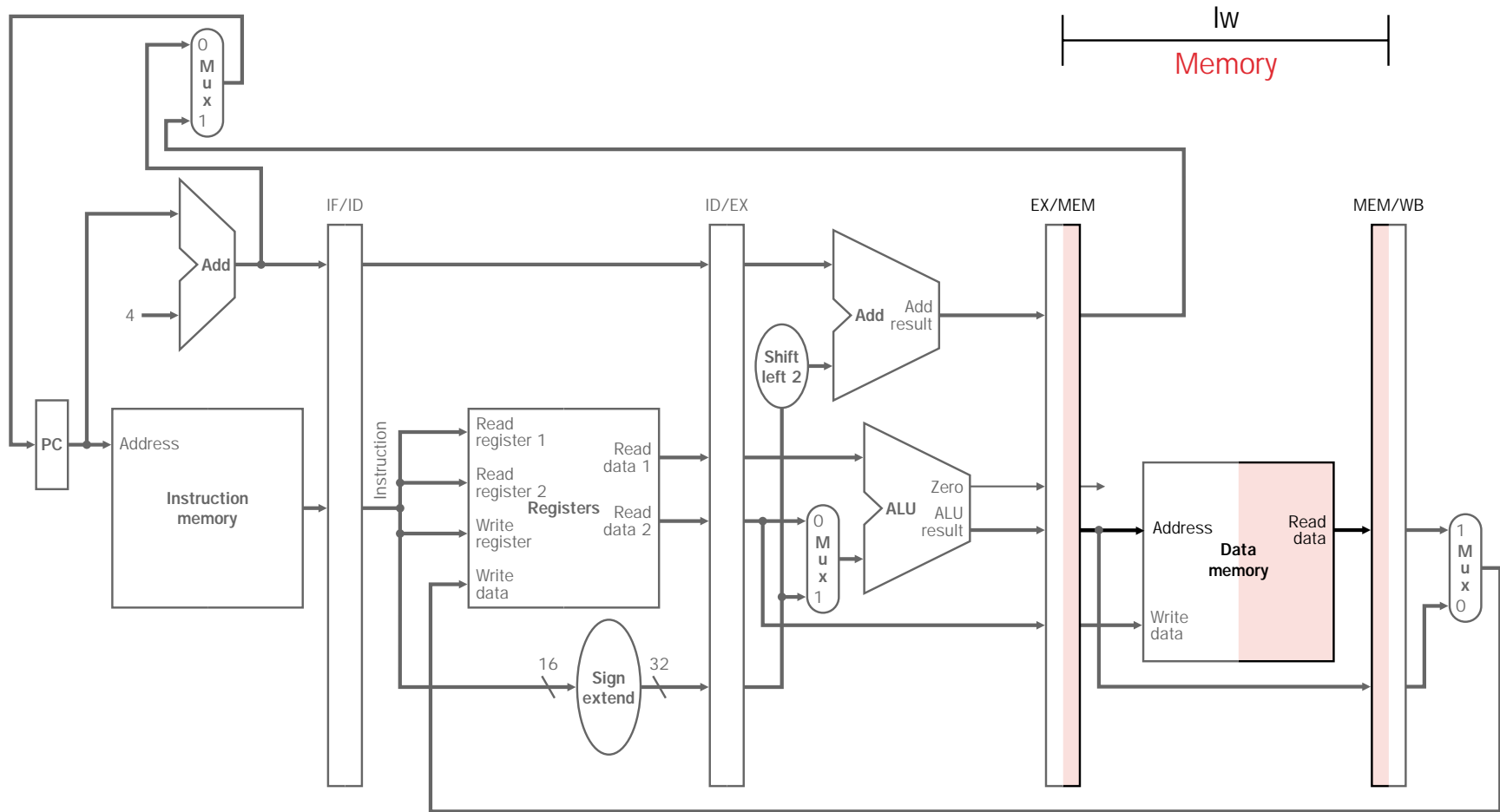
Shift left 2
Add
Add result

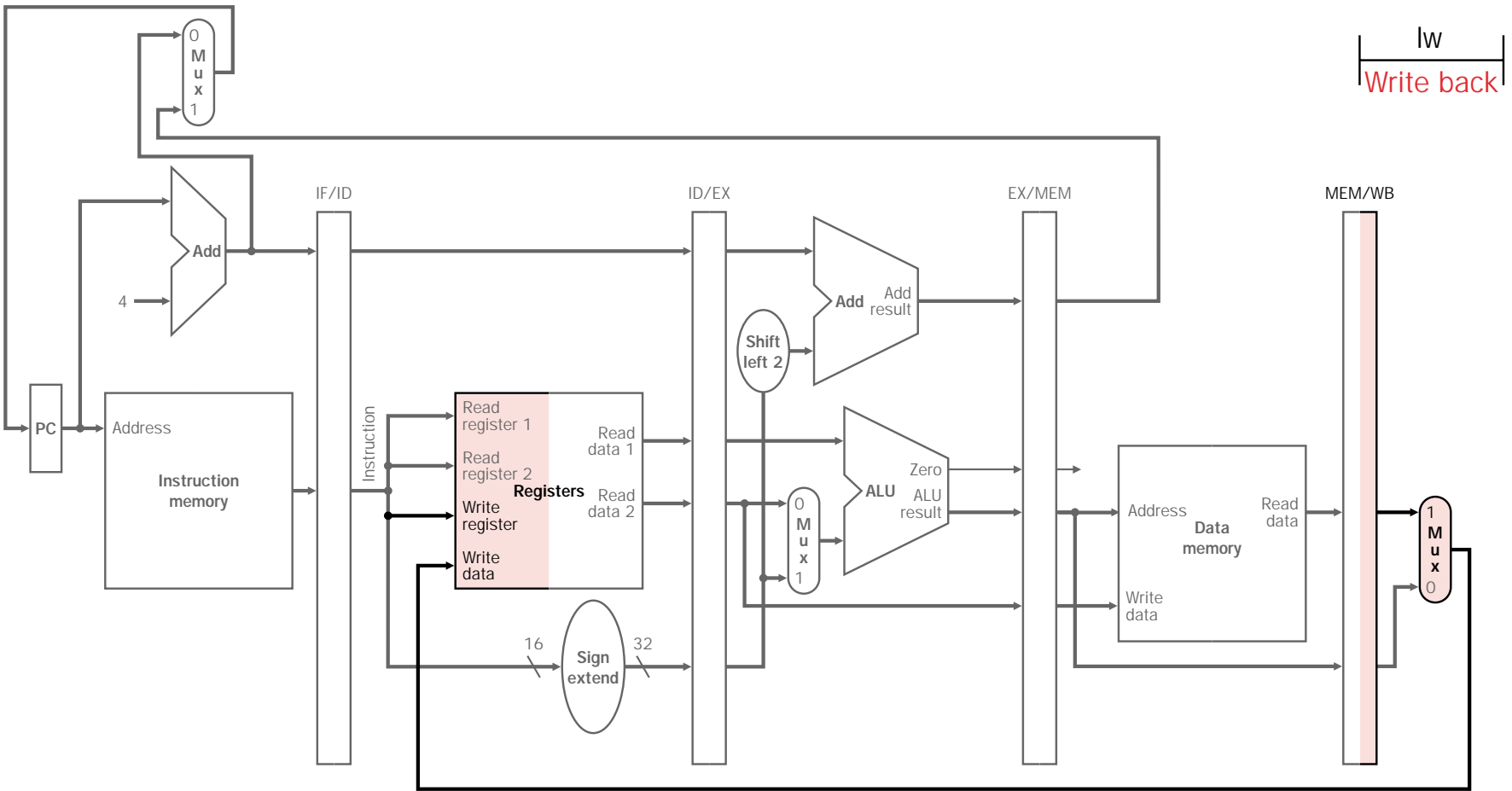
Mux
1
ALU
Zero
ALU result

Address
Data memory
Read data
Write data

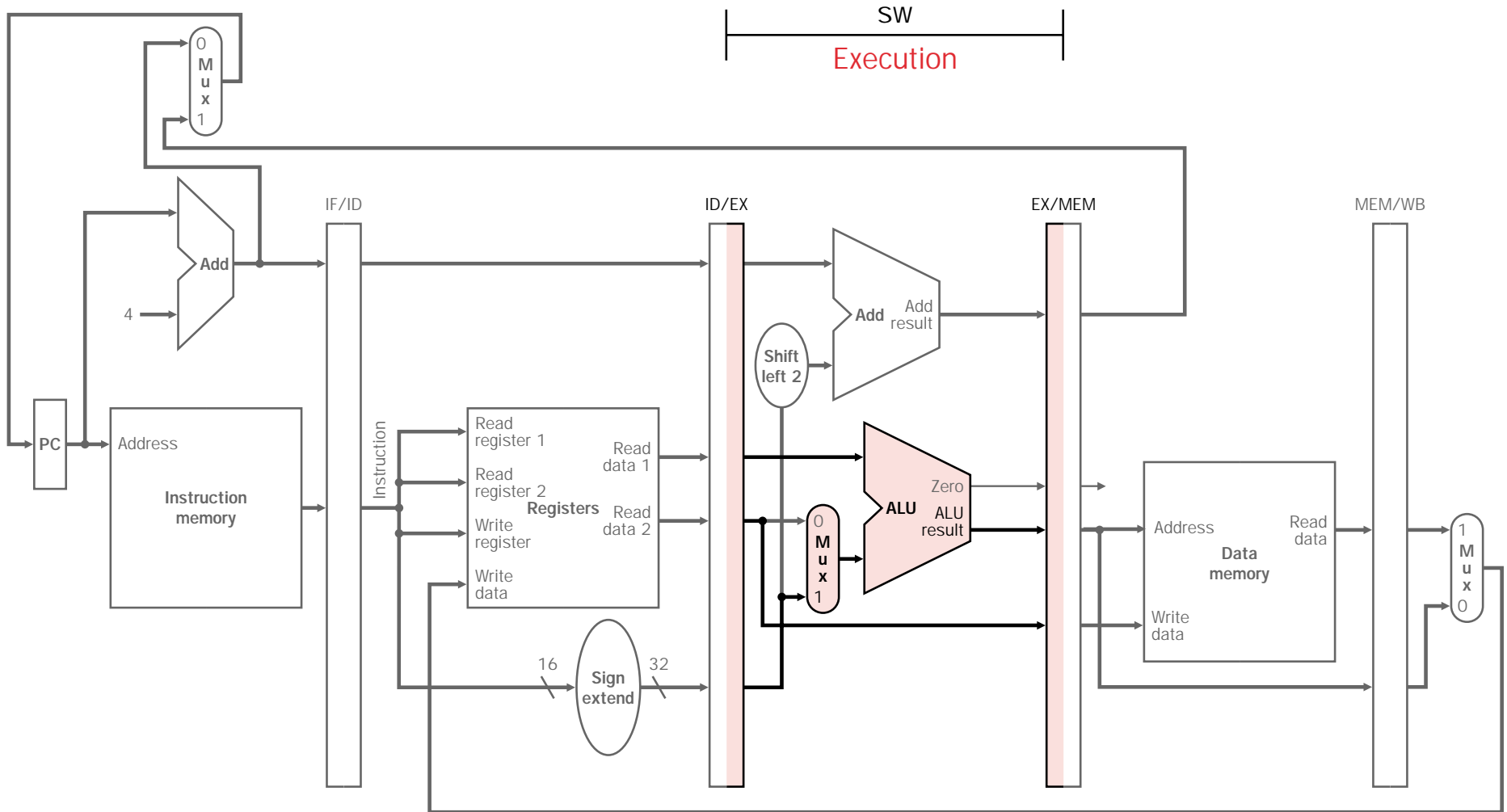
Mux
1
0

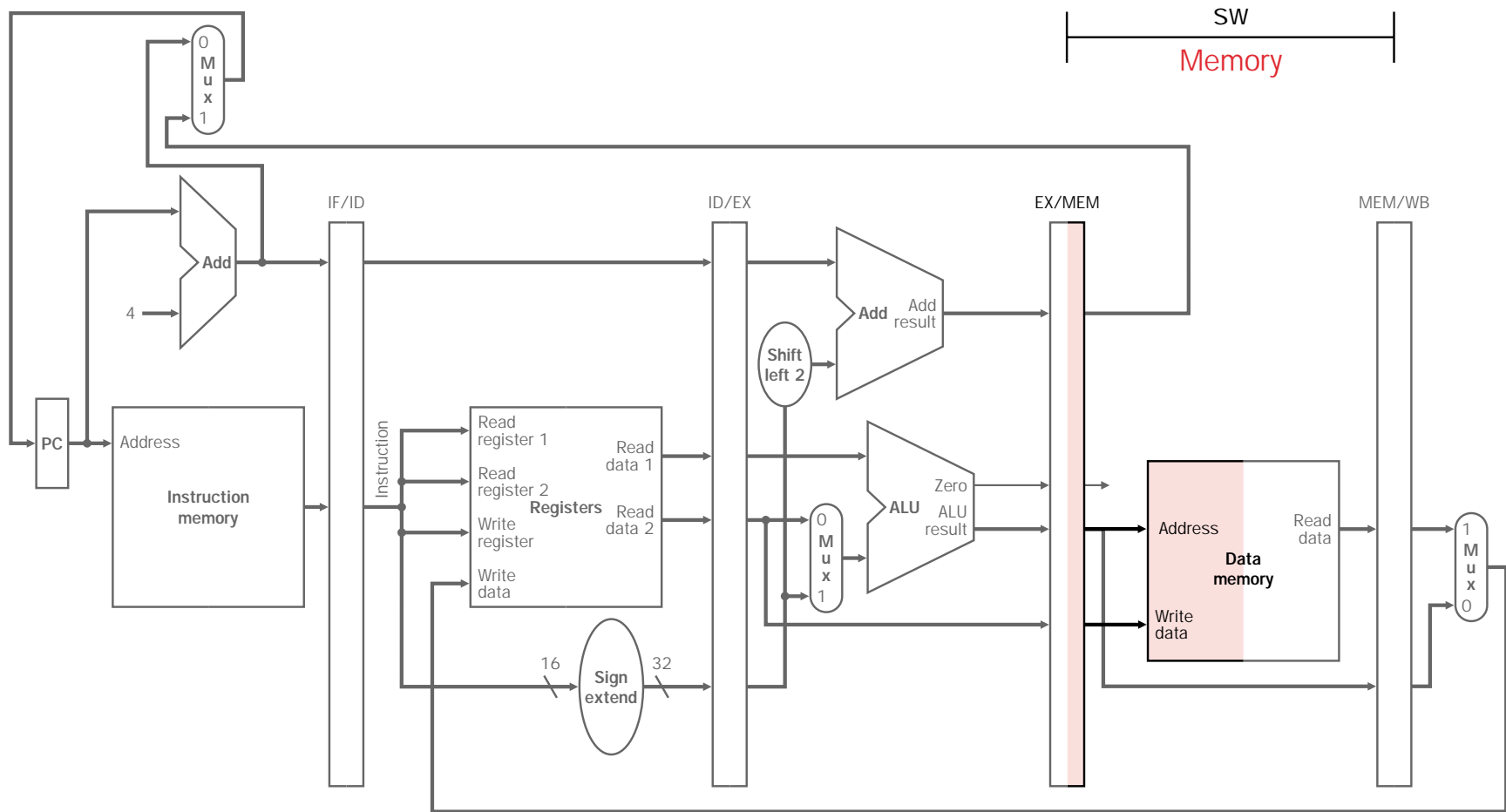


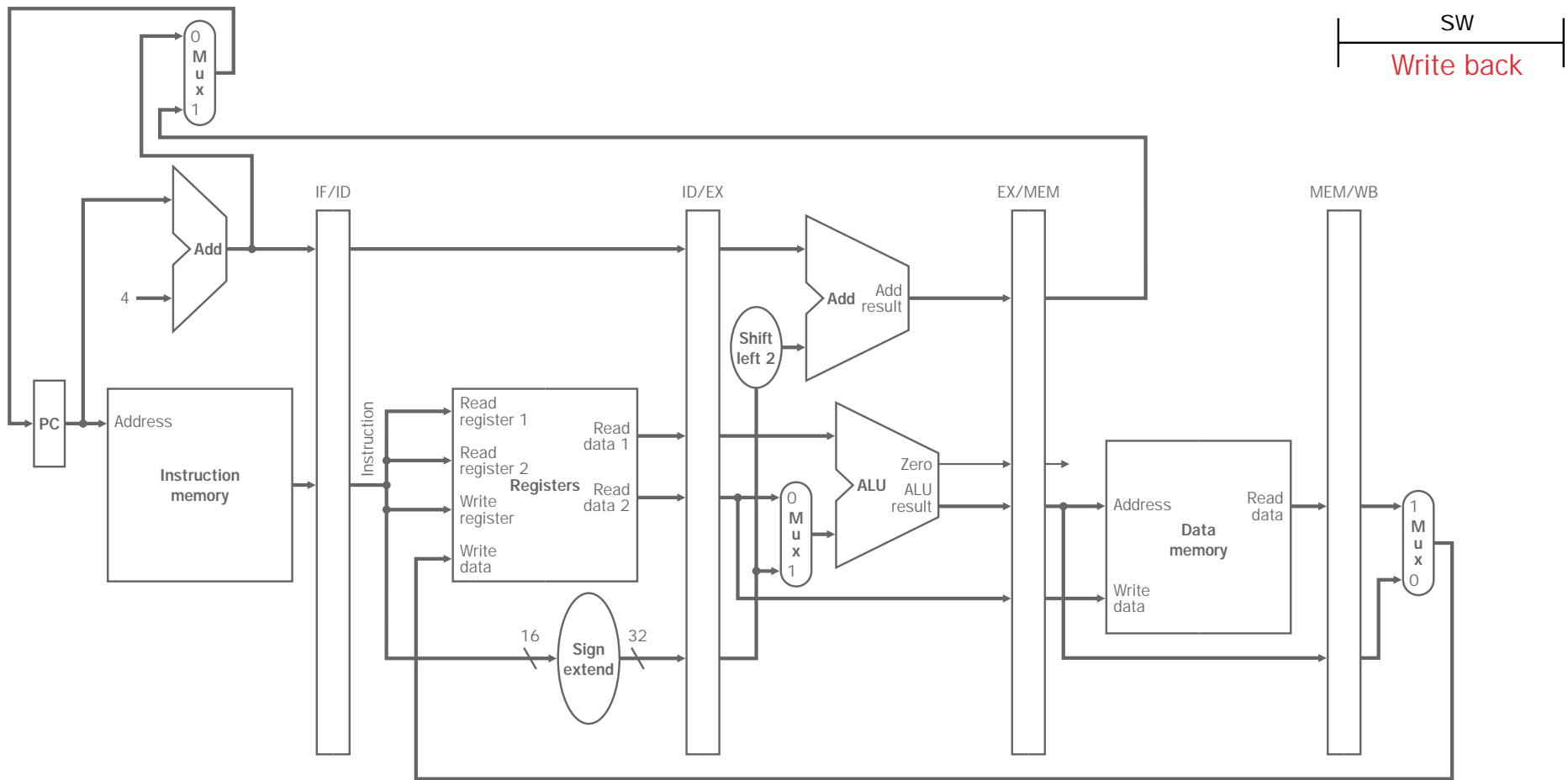




lw
Write back







DATAPATH MODIFICATIONS FOR PIPELINING

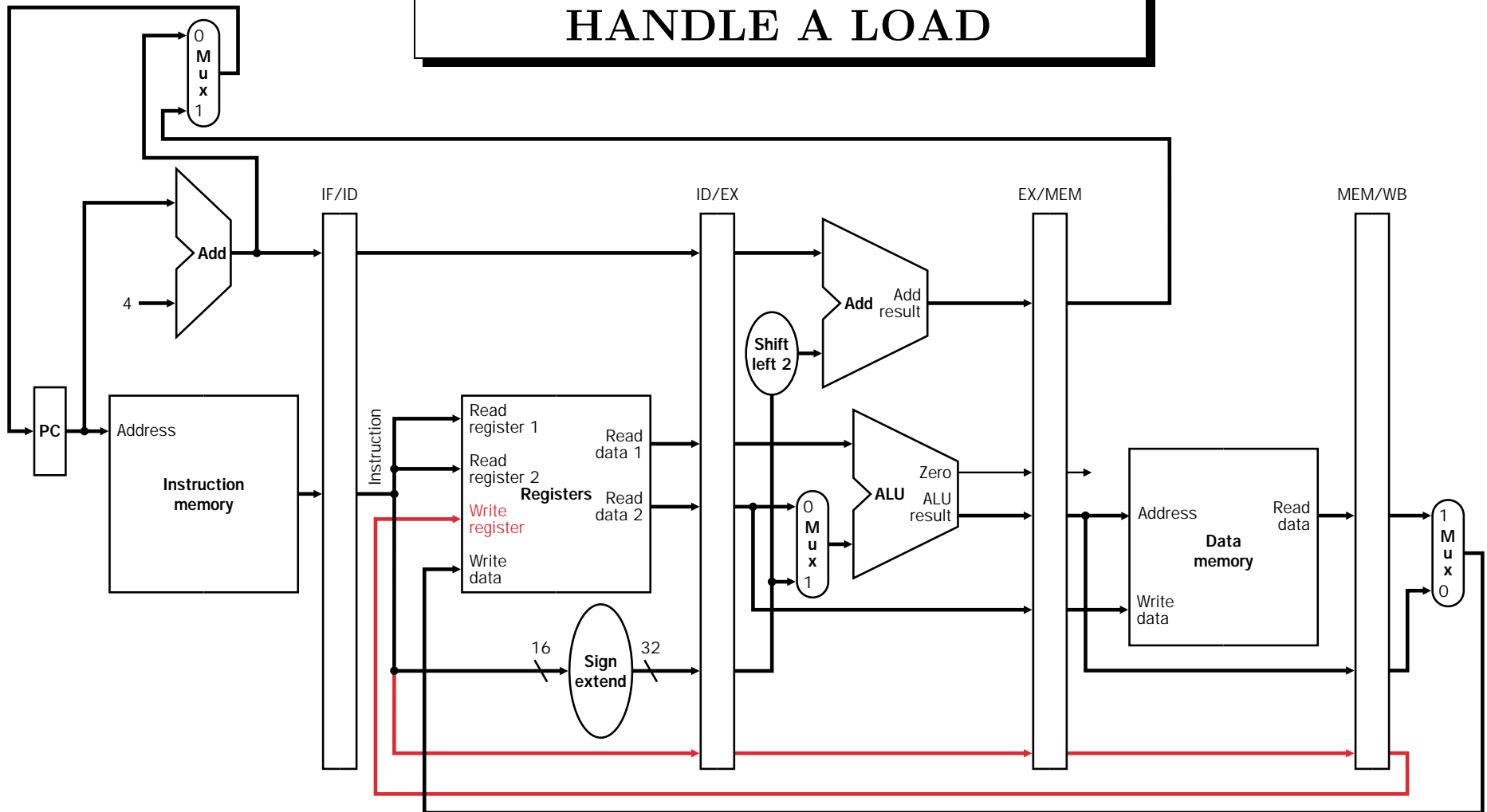
- The number of the register that an instruction must write to is read in the ID stage
- Consider two instructions:

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

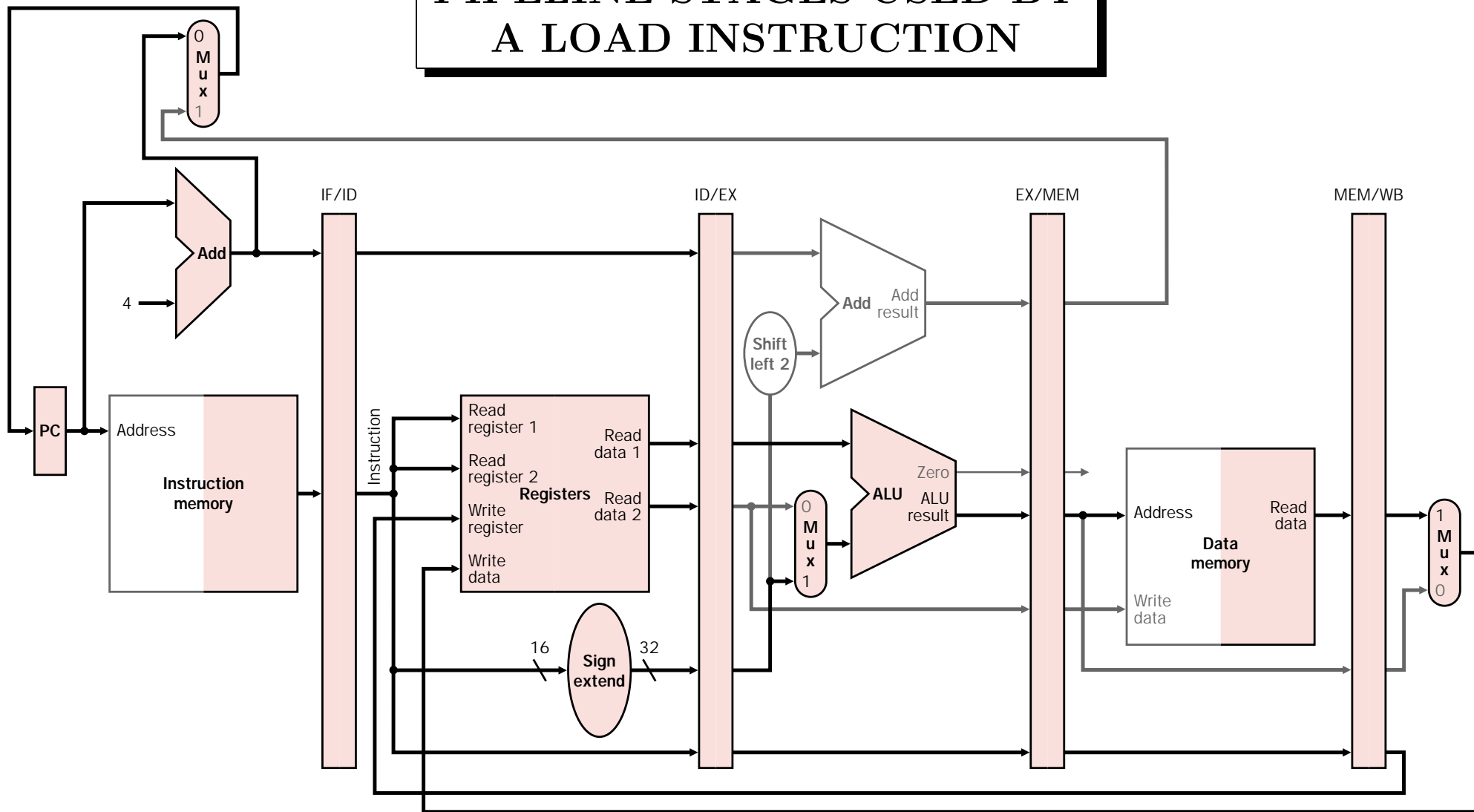
```
sub $11, $2, $3
```

- ▷ WriteRegister signal values are 10 (for **lw**) and 11 (for **sub**)
- ▷ The WriteRegister signal is read only in the WB stage
- ▷ The **sub**'s ID stage modifies WriteRegister before **lw** can read it
- ▷ Therefore the value of the WriteRegister signal is part of the instruction's state, and must be passed along in pipeline registers as the instruction executes

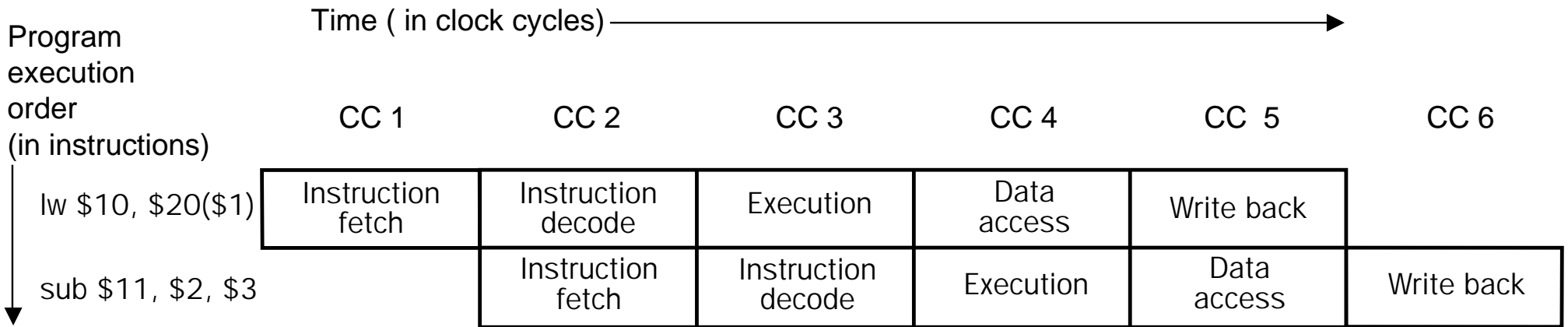
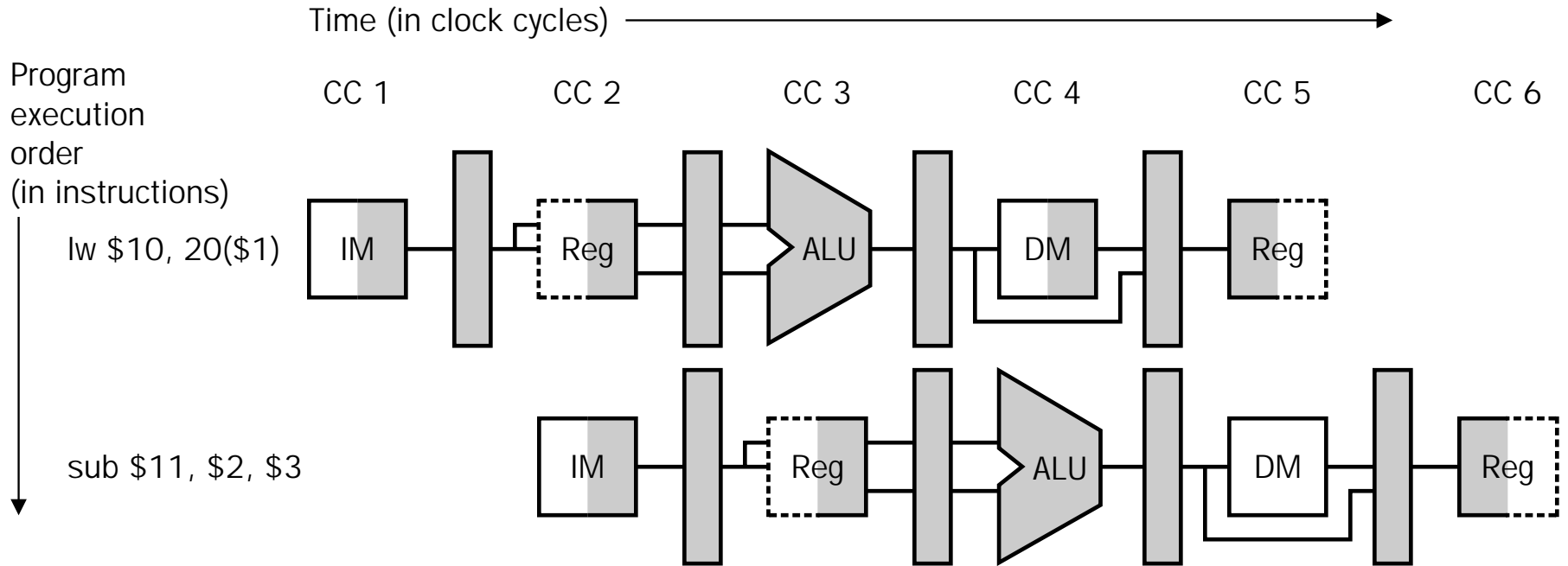
DATAPATH MODIFIED TO HANDLE A LOAD



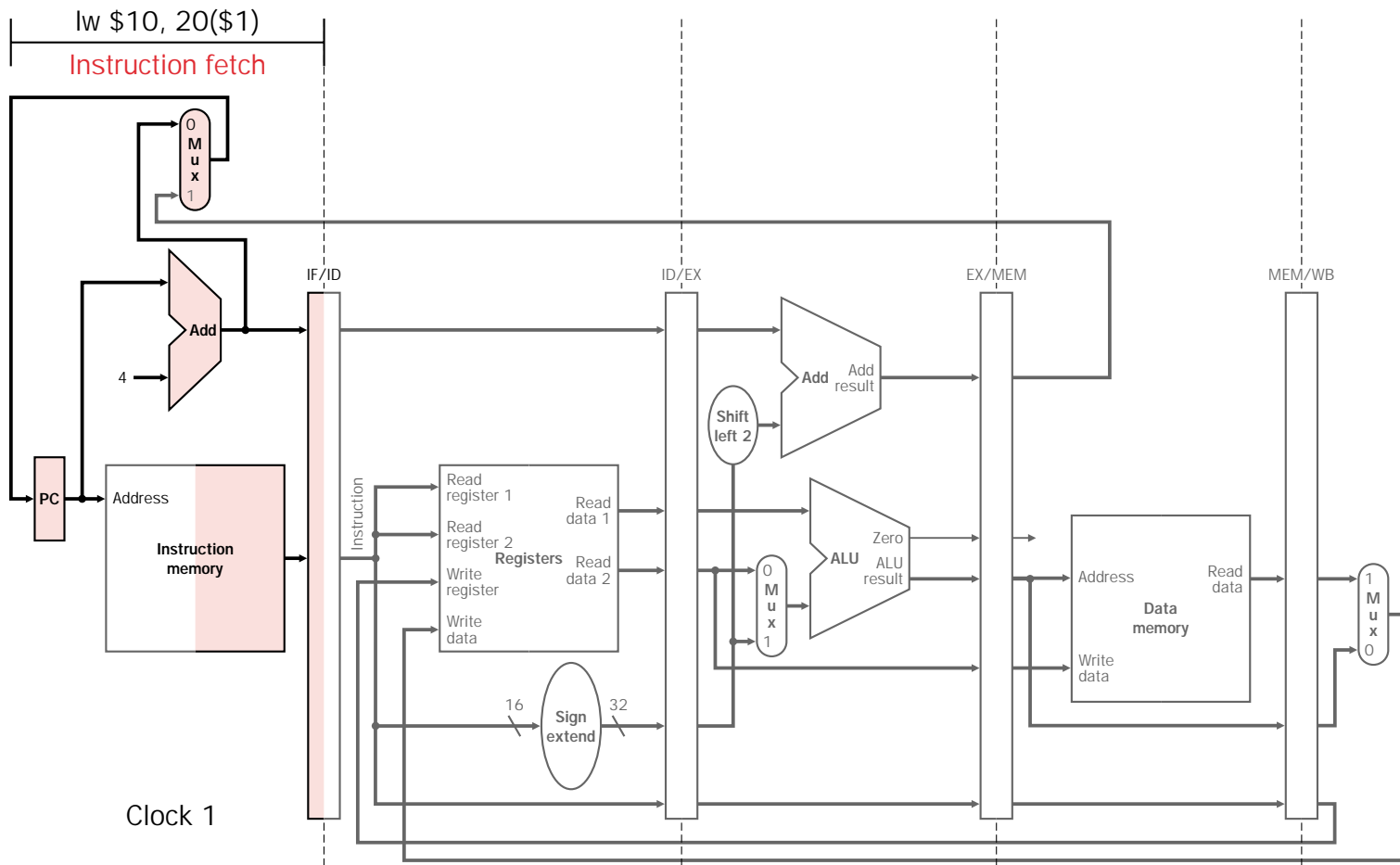
PIPELINE STAGES USED BY A LOAD INSTRUCTION



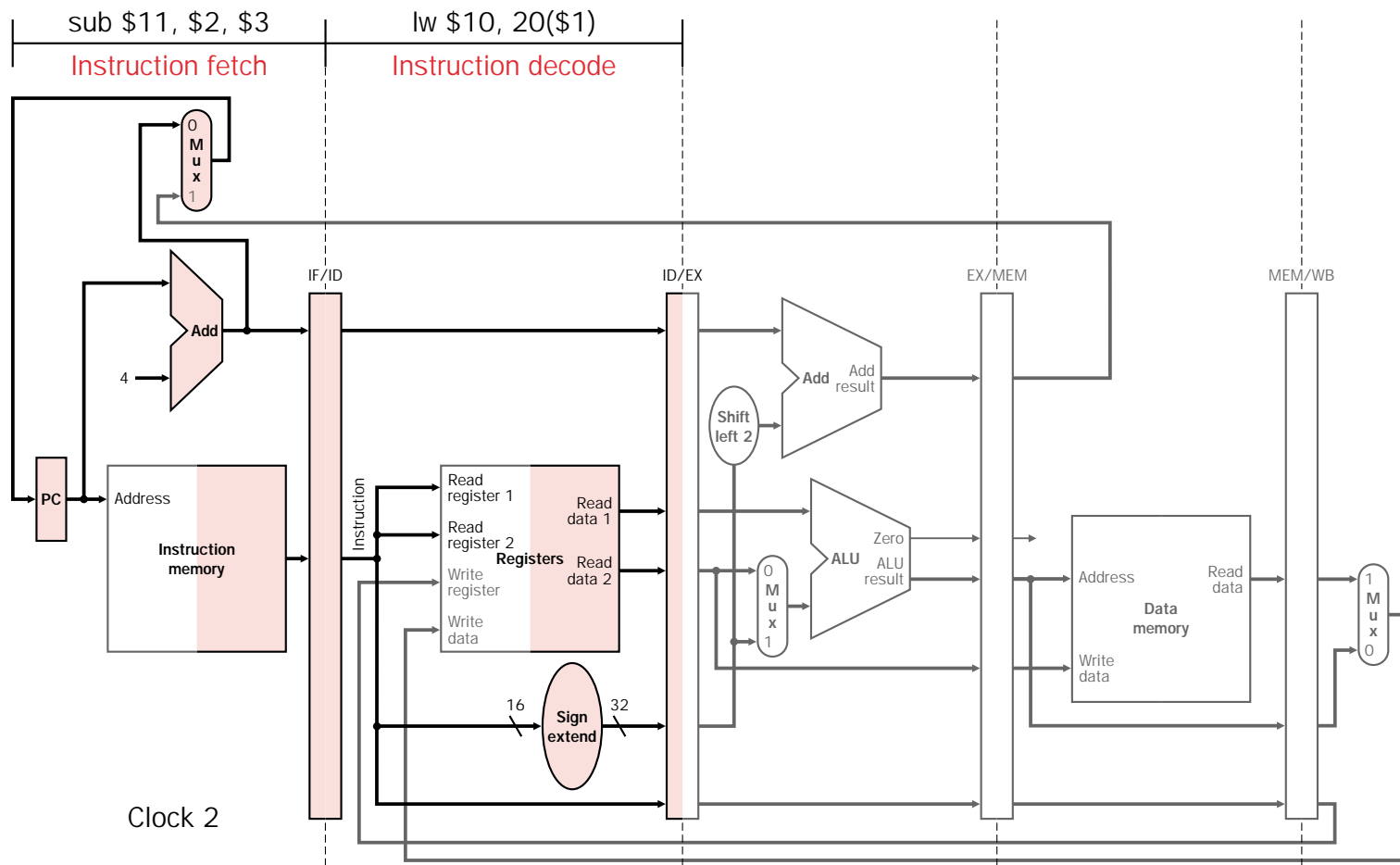
TWO REPRESENTATIONS OF PIPELINED EXECUTION



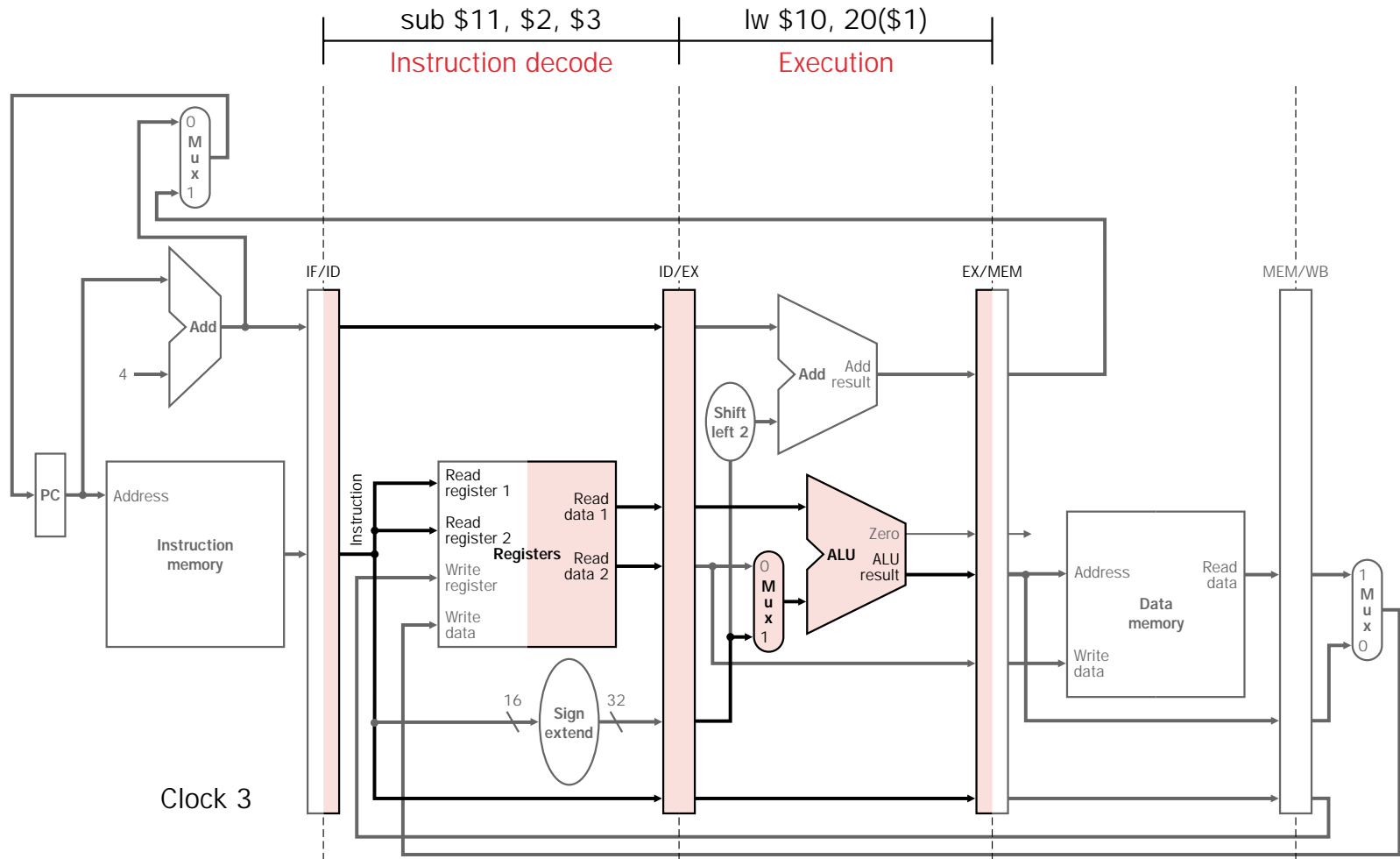
CLOCK PERIOD 1



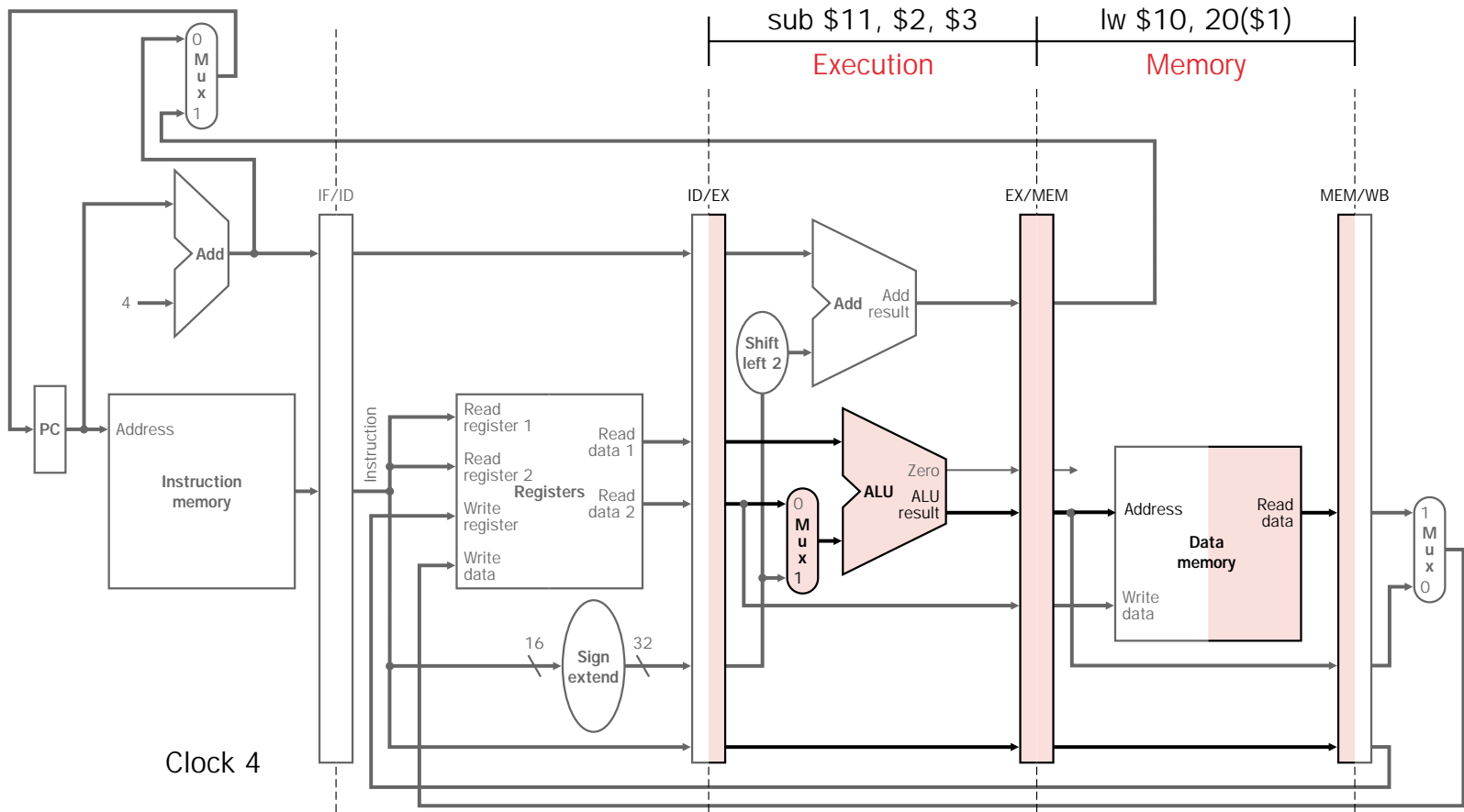
CLOCK PERIOD 2



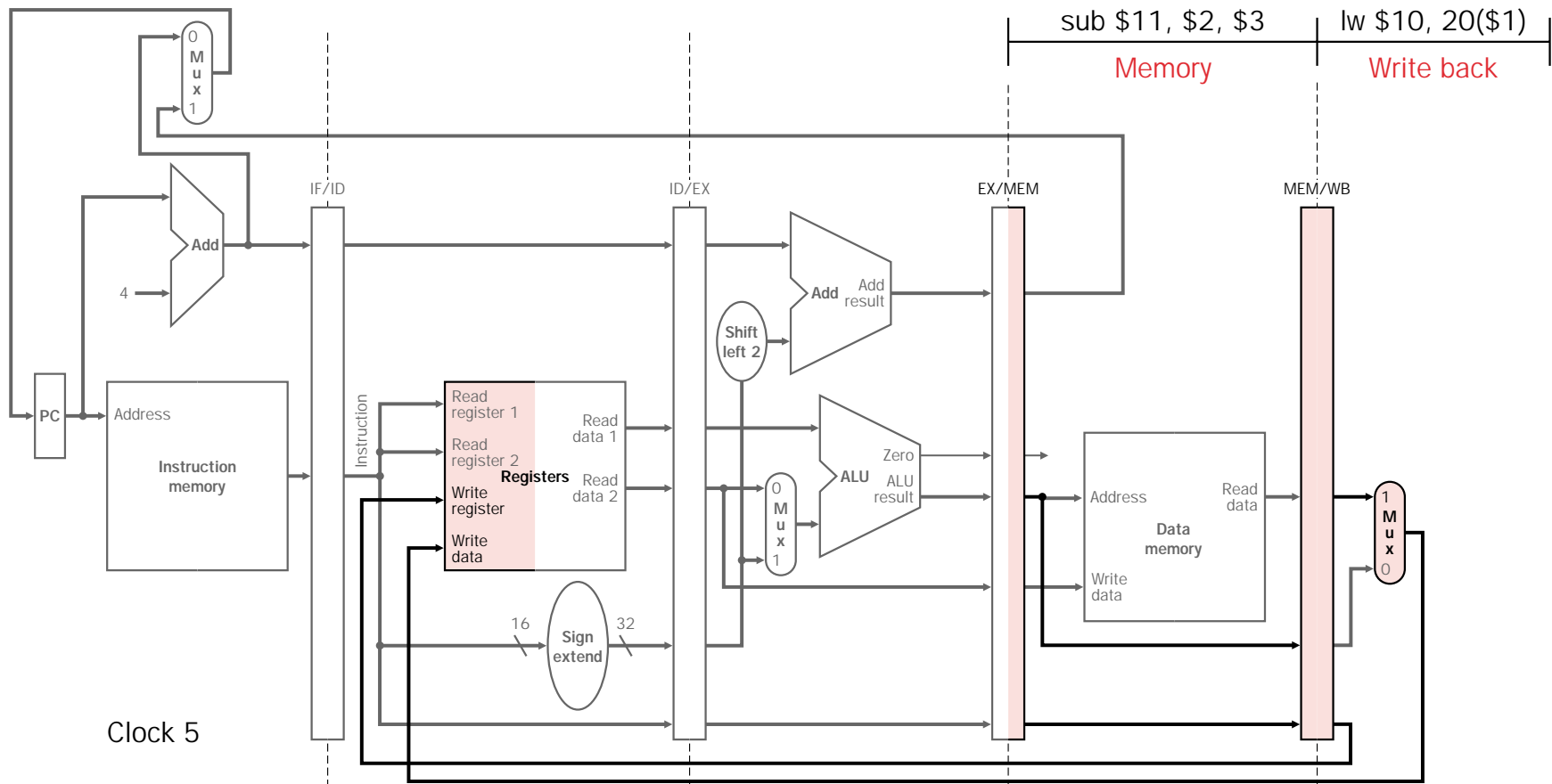
CLOCK PERIOD 3



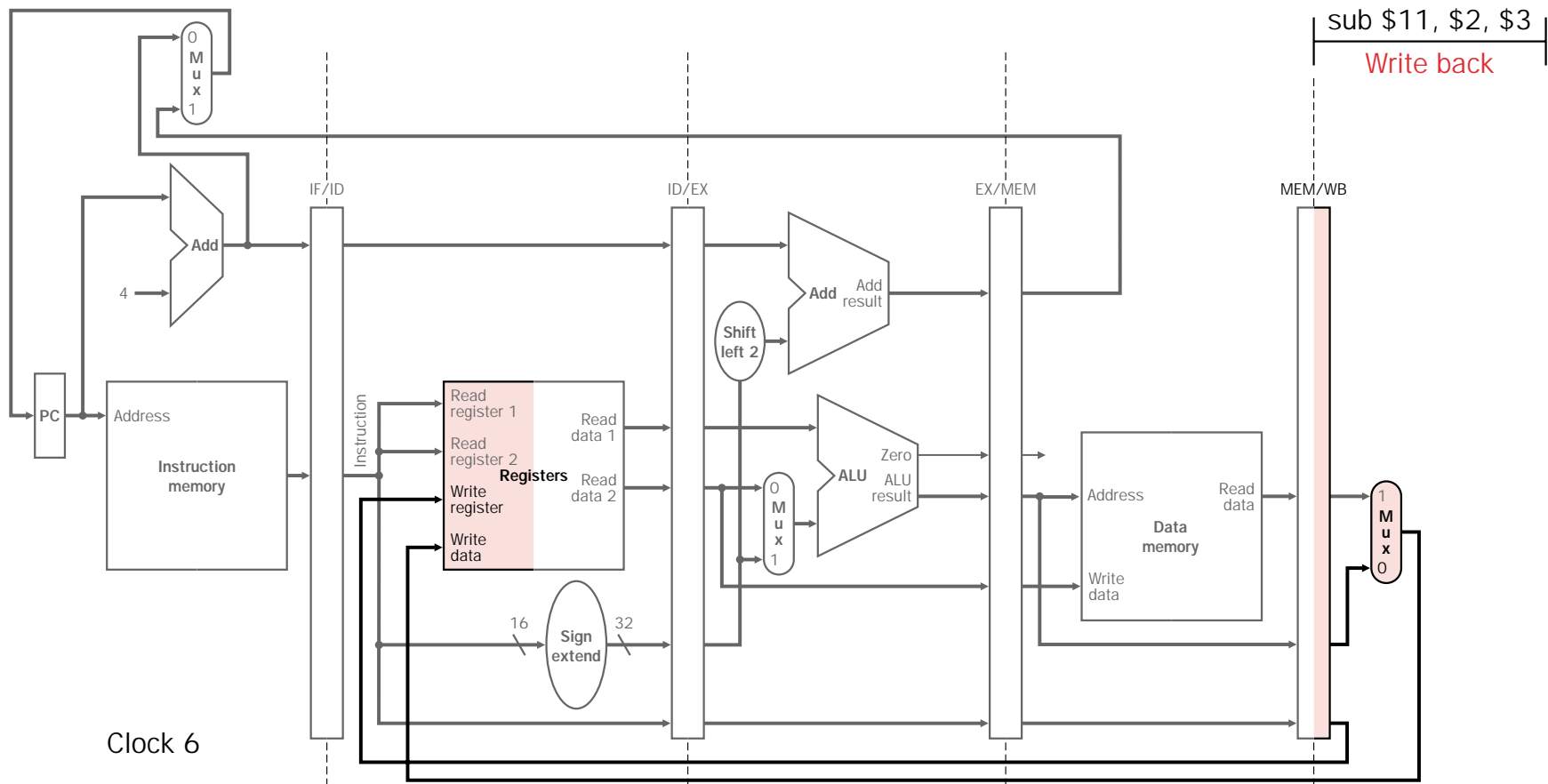
CLOCK PERIOD 4



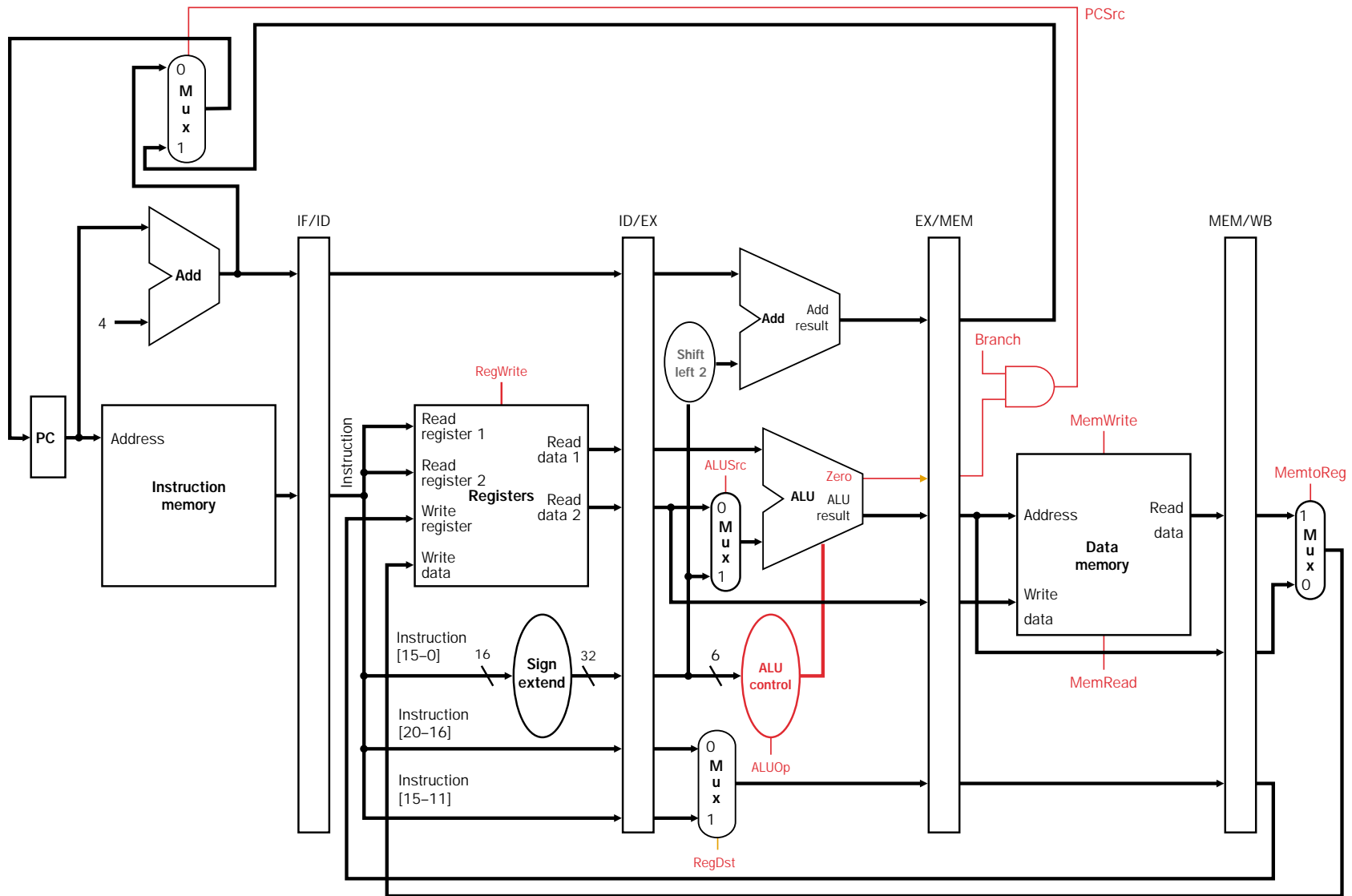
CLOCK PERIOD 5



CLOCK PERIOD 6



PIPELINED DATAPATH WITH CONTROL SIGNALS



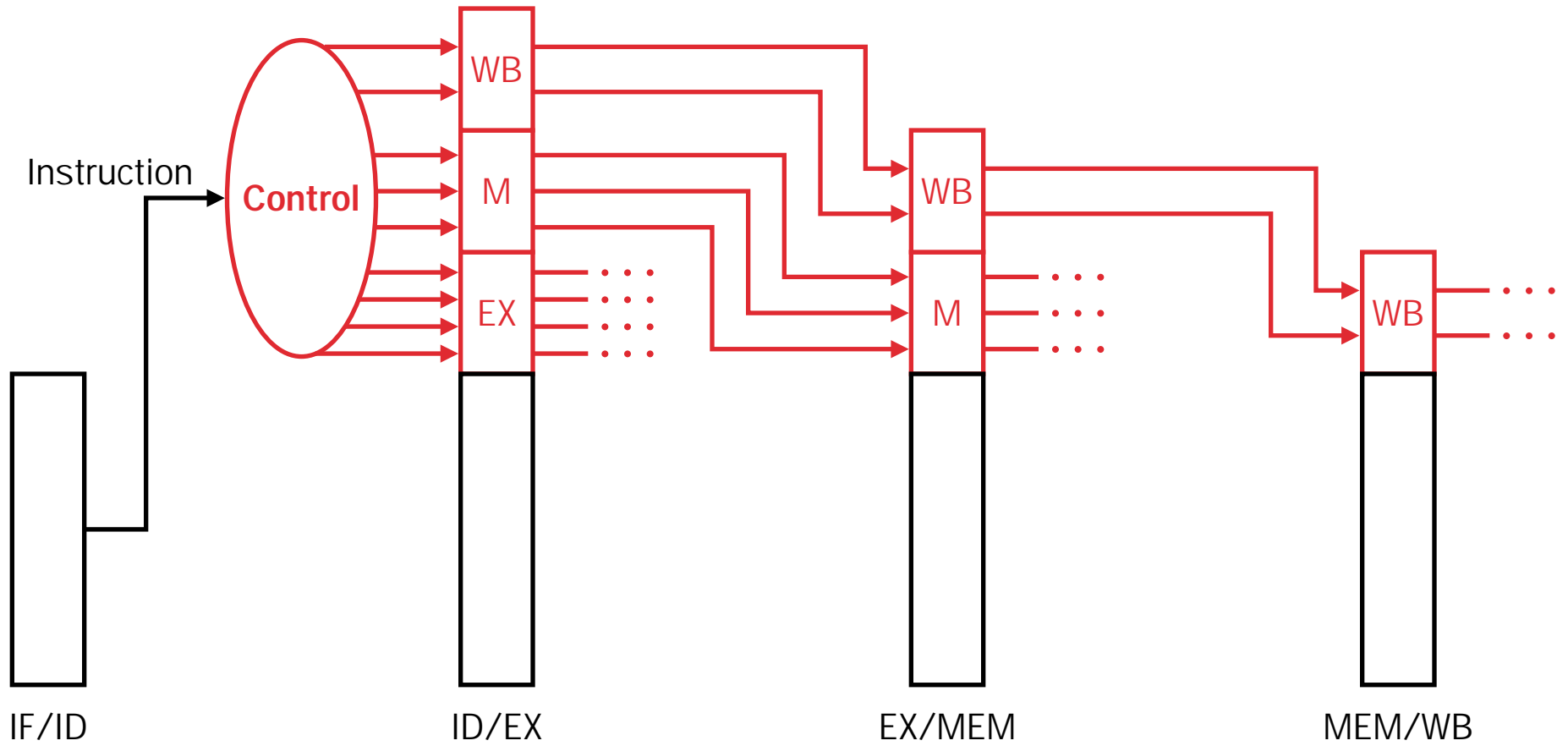
SETTINGS OF CONTROL LINES

Inst. type	Ex/Address Calc.				Mem. Access			WriteBack	
	Reg Dst	ALUOp bit 1	ALUOp bit 0	ALU Src	Br	Mem Read	Mem Write	Reg Write	Mem- to-reg.
R-type	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	d	0	0	1	0	0	1	0	d
beq	d	0	1	0	1	0	0	0	d

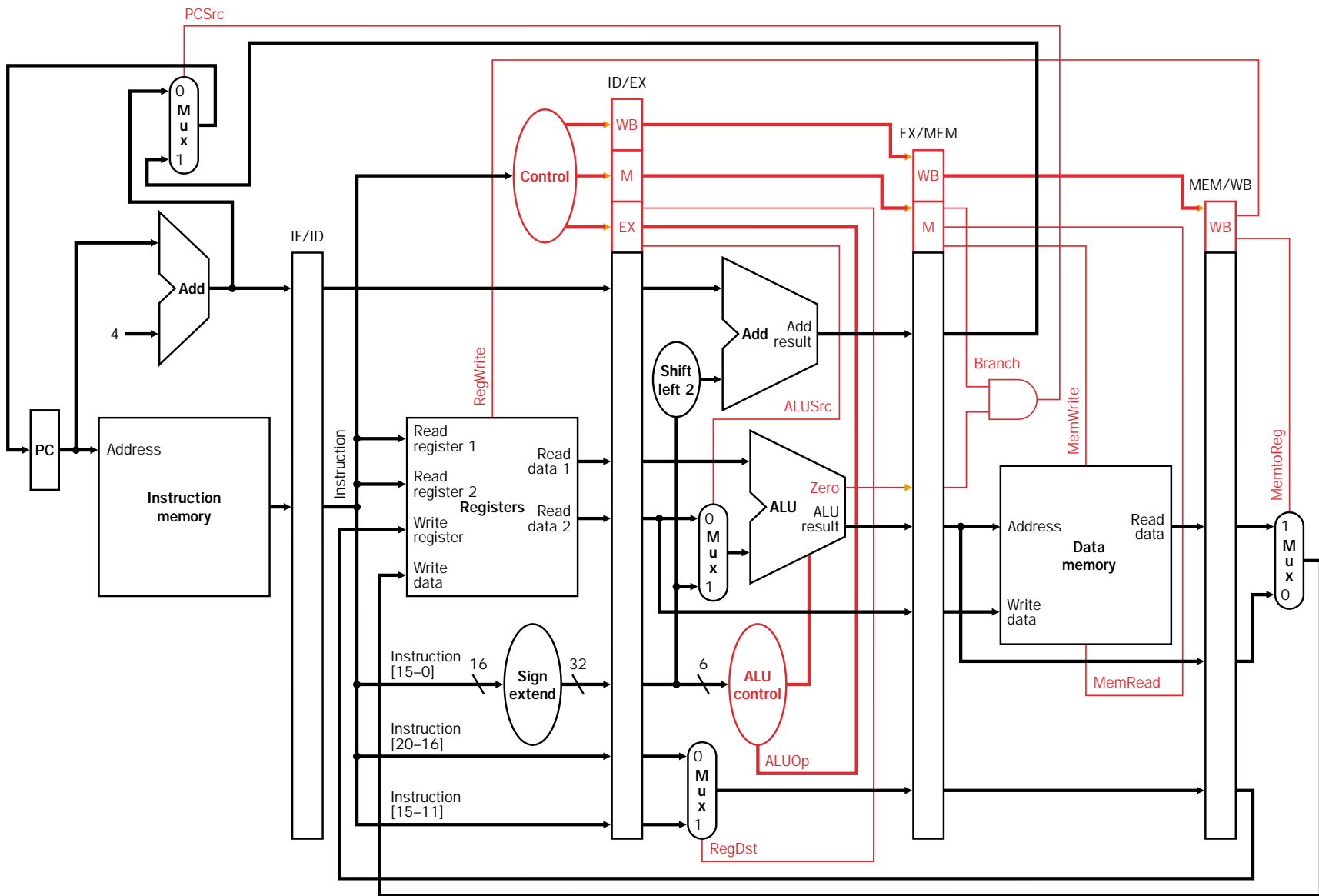
PIPELINED CONTROL

- The control signals for an instruction are determined in the ID stage
 - ▷ The next instruction's ID stage asserts new values of the control signals
 - ▷ Control signals are part of the state of the instruction, and therefore must be passed along from stage to stage in pipeline registers, just like data

CONTROL LINES FOR THE THREE FINAL STAGES



PIPELINED DATAPATH WITH CONTROL LOGIC AND SIGNALS



PIPELINED EXECUTION OF FIVE INSTRUCTIONS

- We'll follow what happens in the instruction sequence

[40000024] lw \$10, 20(\$1)

[40000028] sub \$11, \$2, \$3

[4000002c] and \$12, \$4, \$5

[40000030] or \$13, \$6, \$7

[40000034] add \$14, \$8, \$9

- ▷ For each clock period, note the values of the RegisterWrite signal in the ID/EX, EX/MEM, and MEM/WB pipeline registers

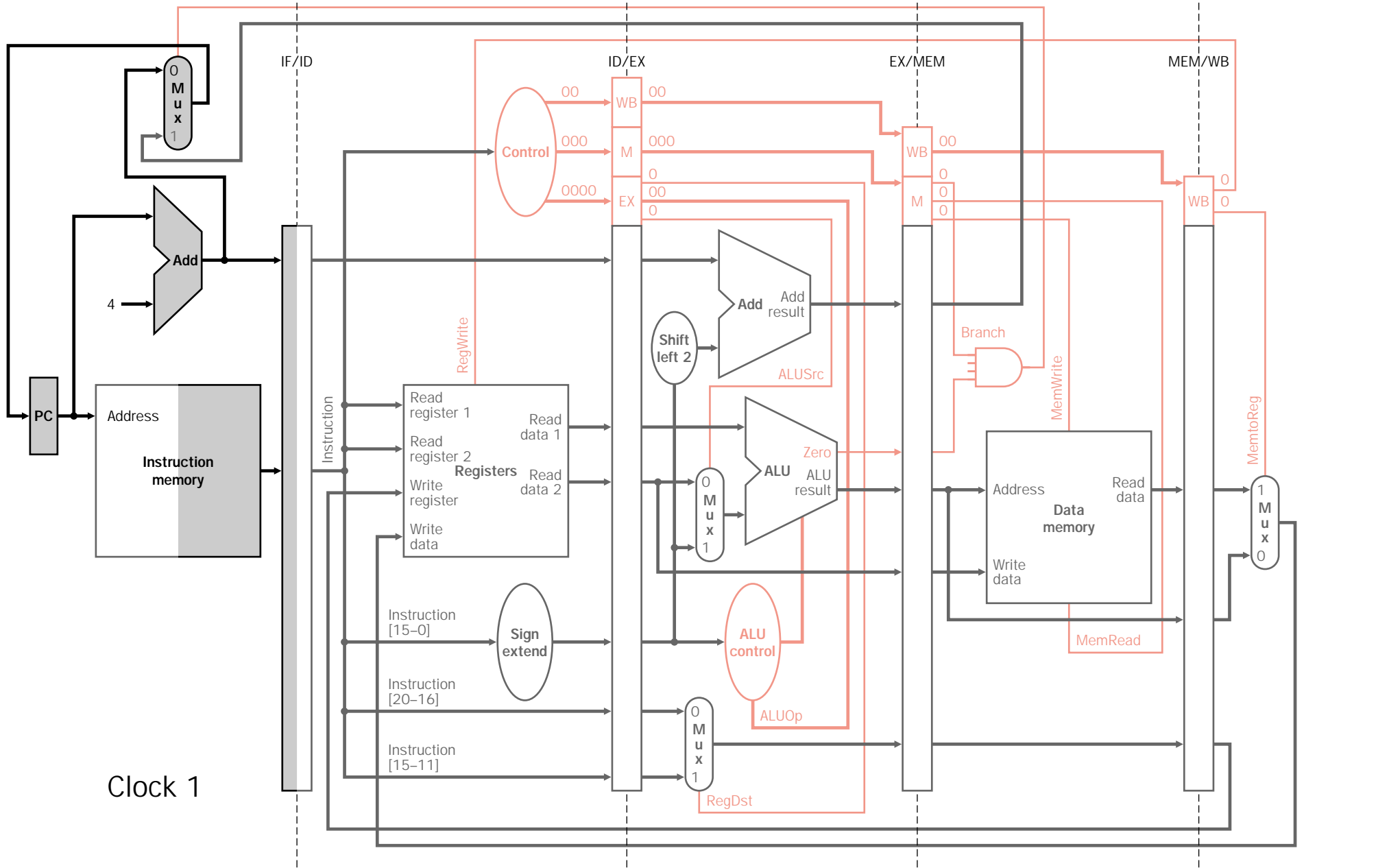
IF: lw \$10, 20(\$1)

ID: before<1>

EX: before<2>

MEM: before<3>

WB: before<4>



Clock 1

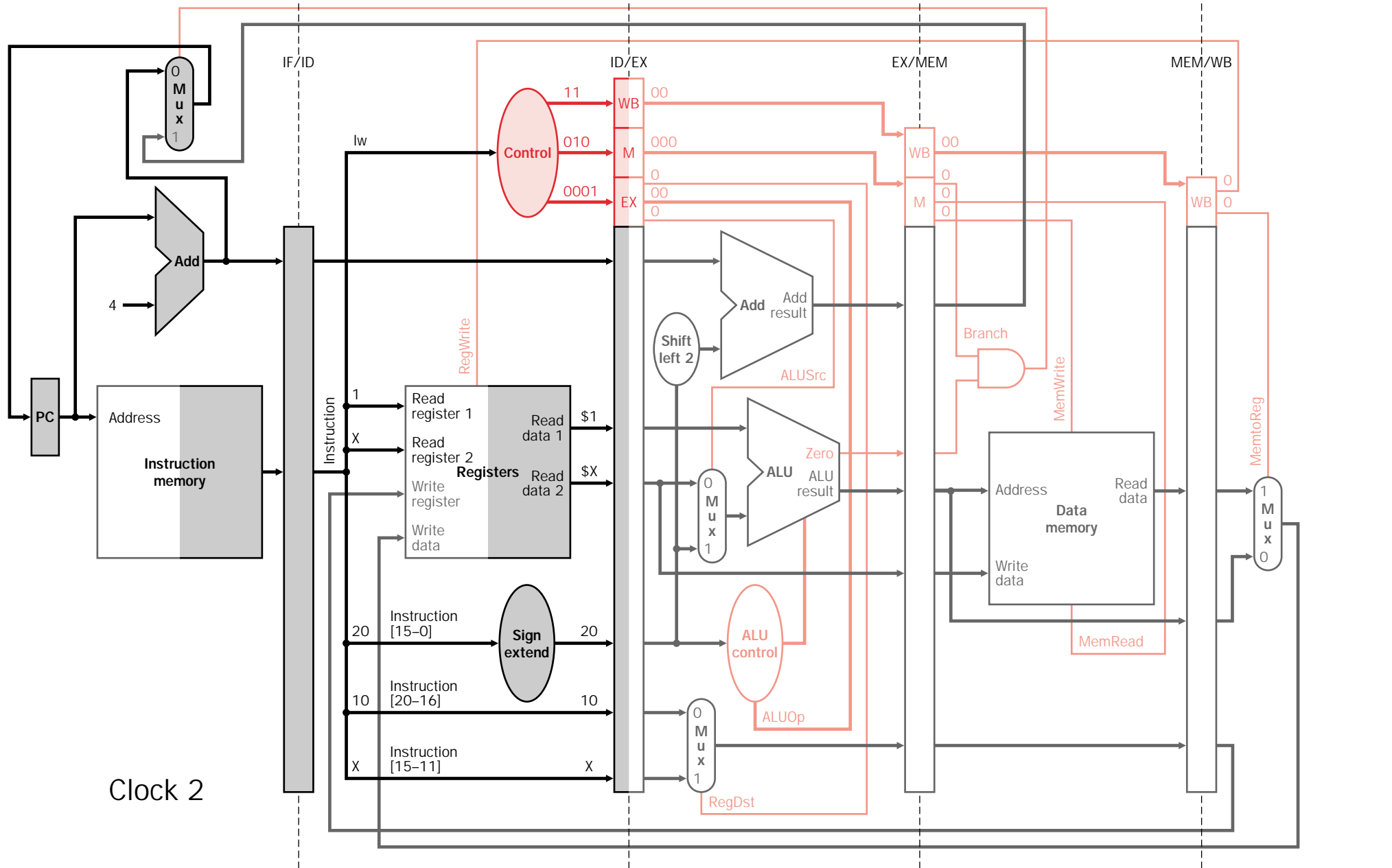
IF: sub \$11, \$2, \$3

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

EX: before<1>

MEM: before<2>

WB: before<3>



Clock 2

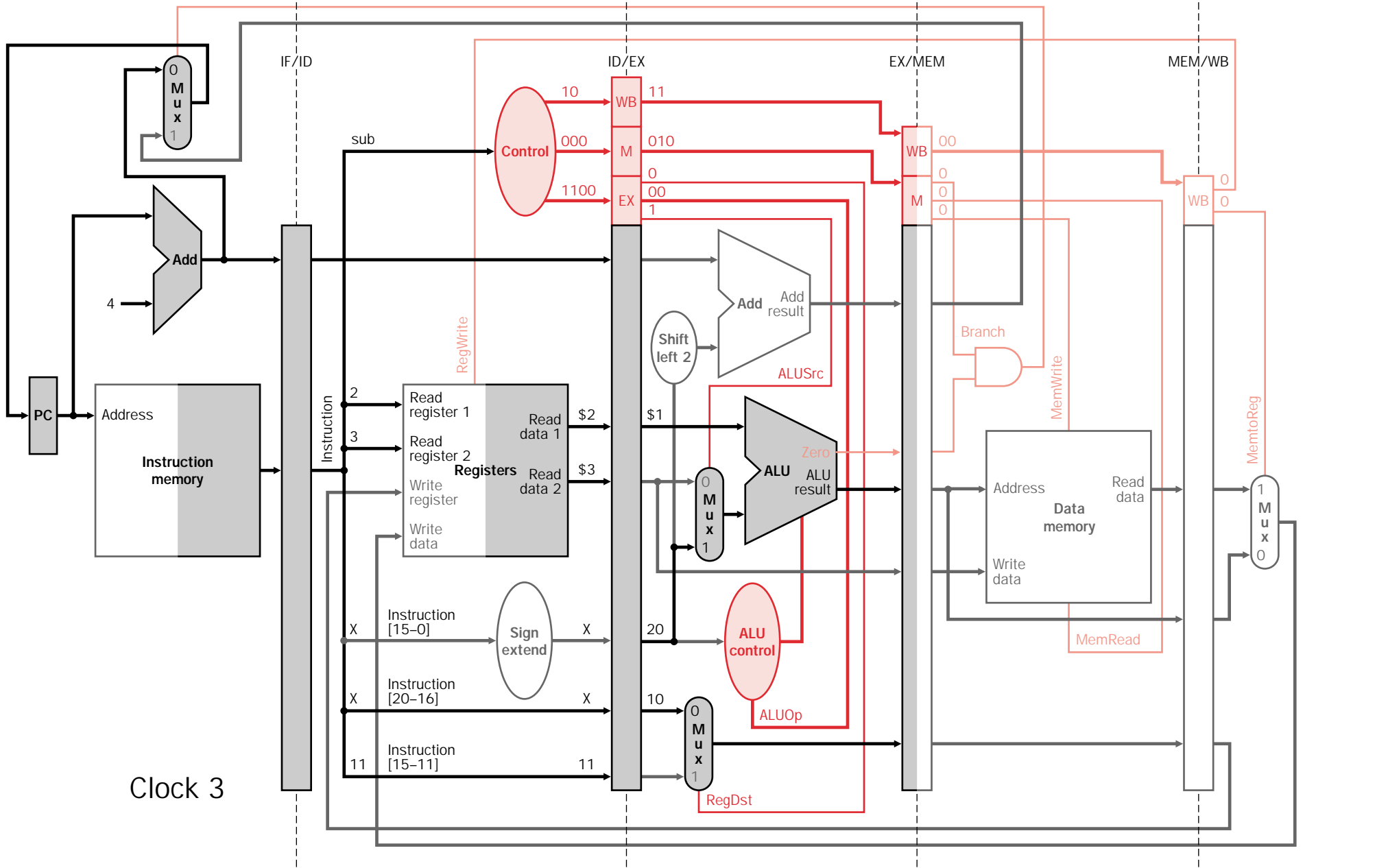
IF: and \$12, \$4, \$5

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

EX: lw \$10, ...

MEM: before<1>

WB: before<2>



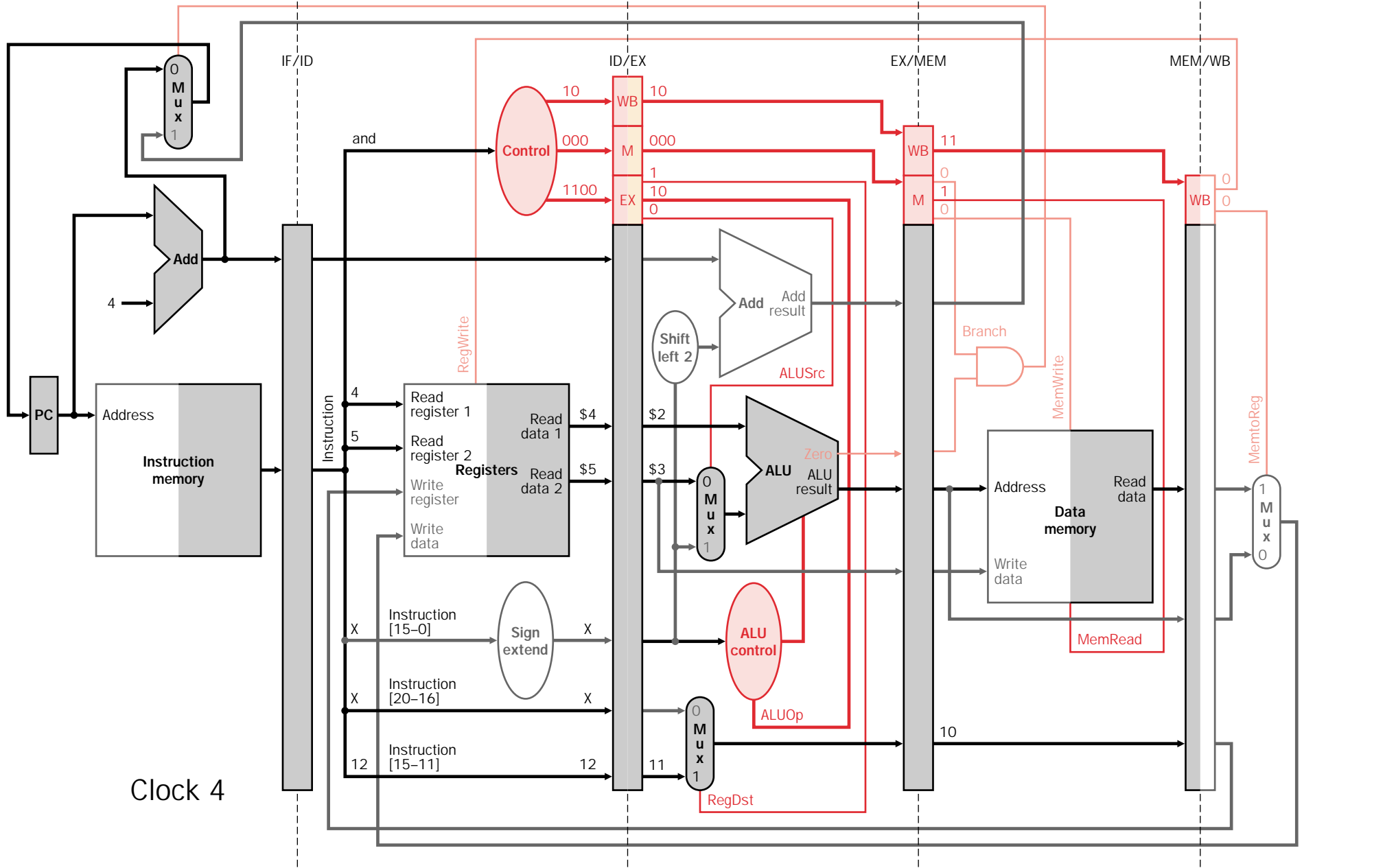
IF: or \$13, \$6, \$7

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

EX: sub \$11, ...

MEM: lw \$10, ...

WB: before <1>



Clock 4

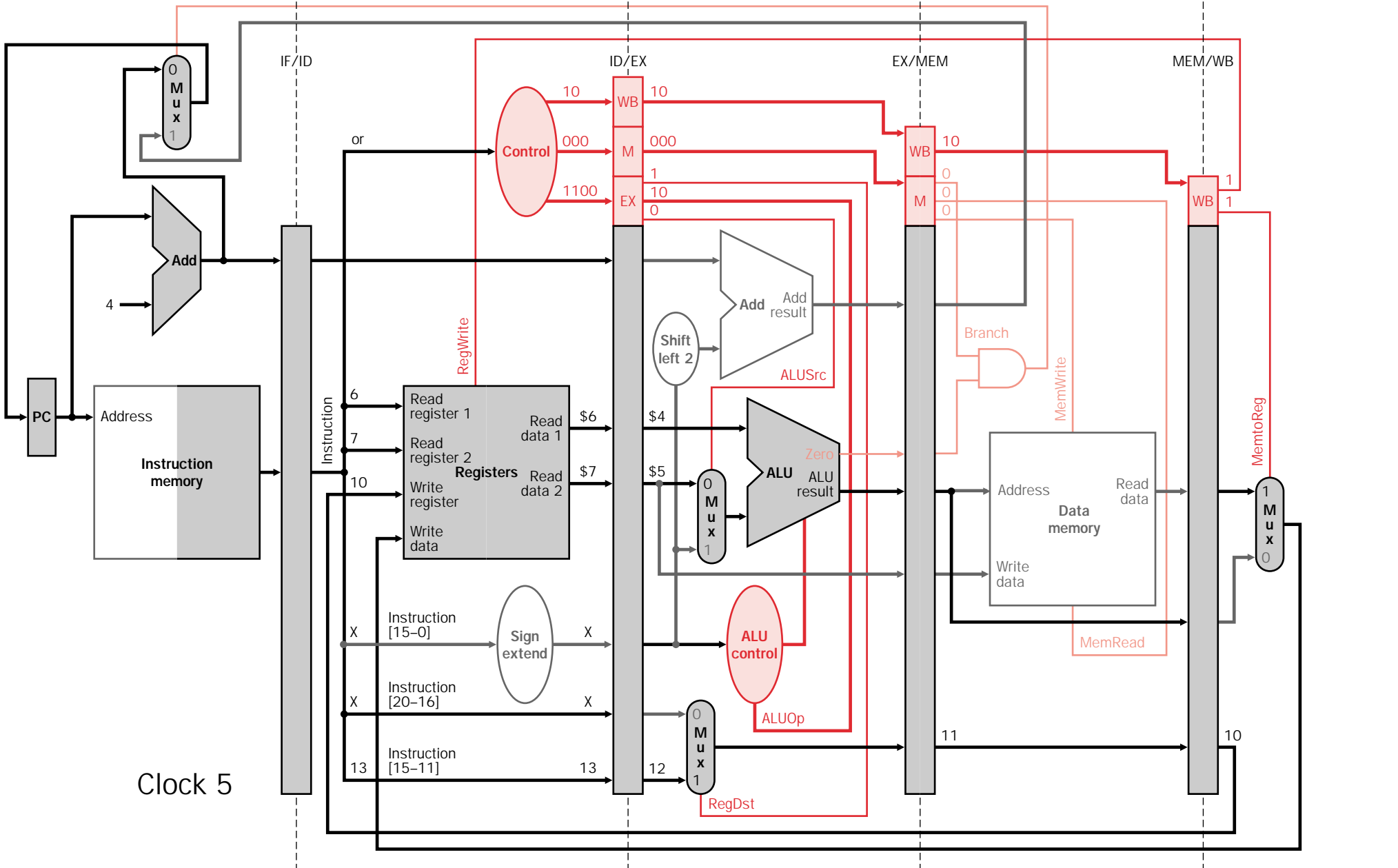
IF: add \$14, \$8, \$9

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

EX: and \$12, ...

MEM: sub \$11, ...

WB: lw \$10, ...



Clock 5

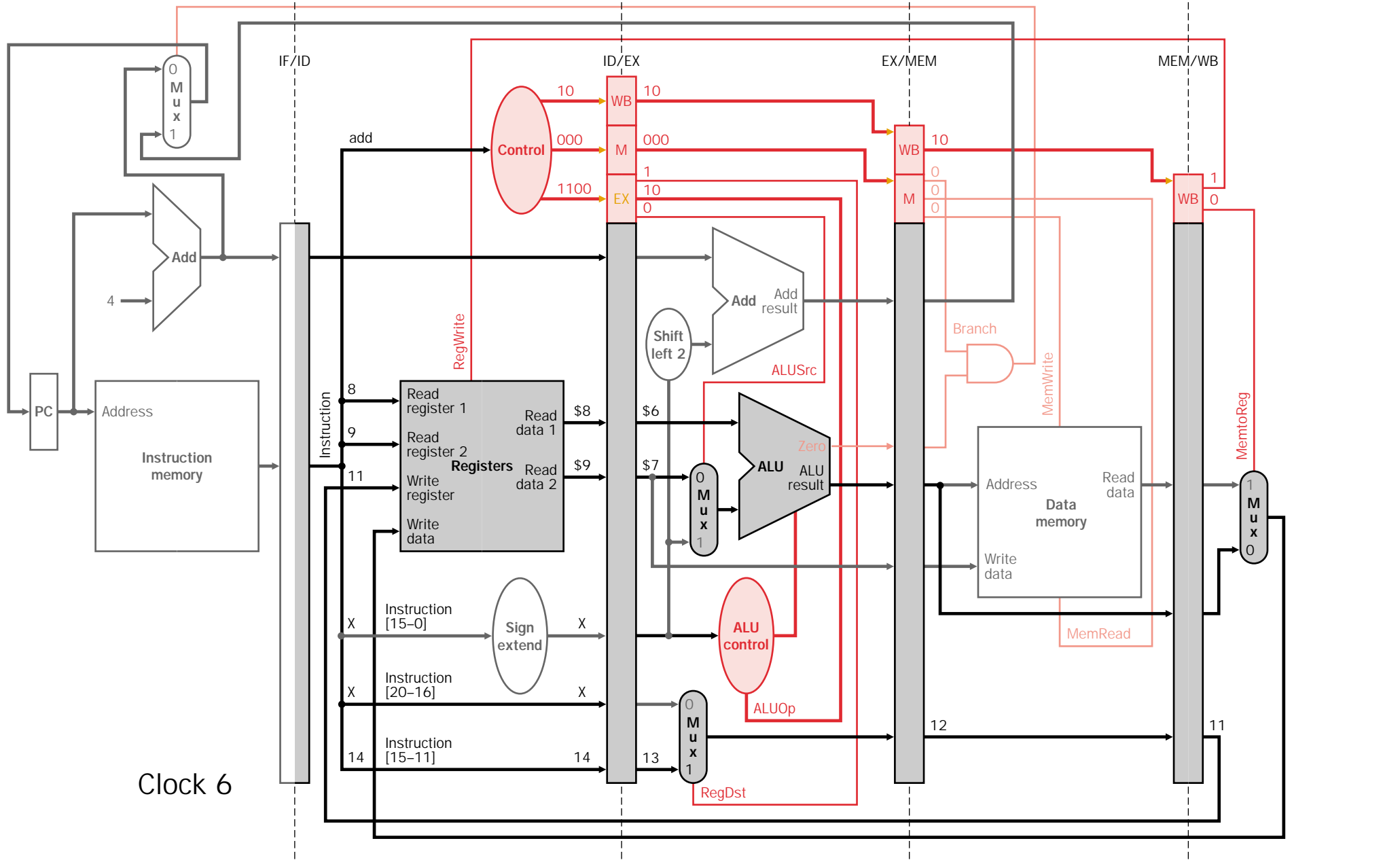
IF: after<1>

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

EX: or \$13, ...

MEM: and \$12, ...

WB: sub \$11, ...



Clock 6

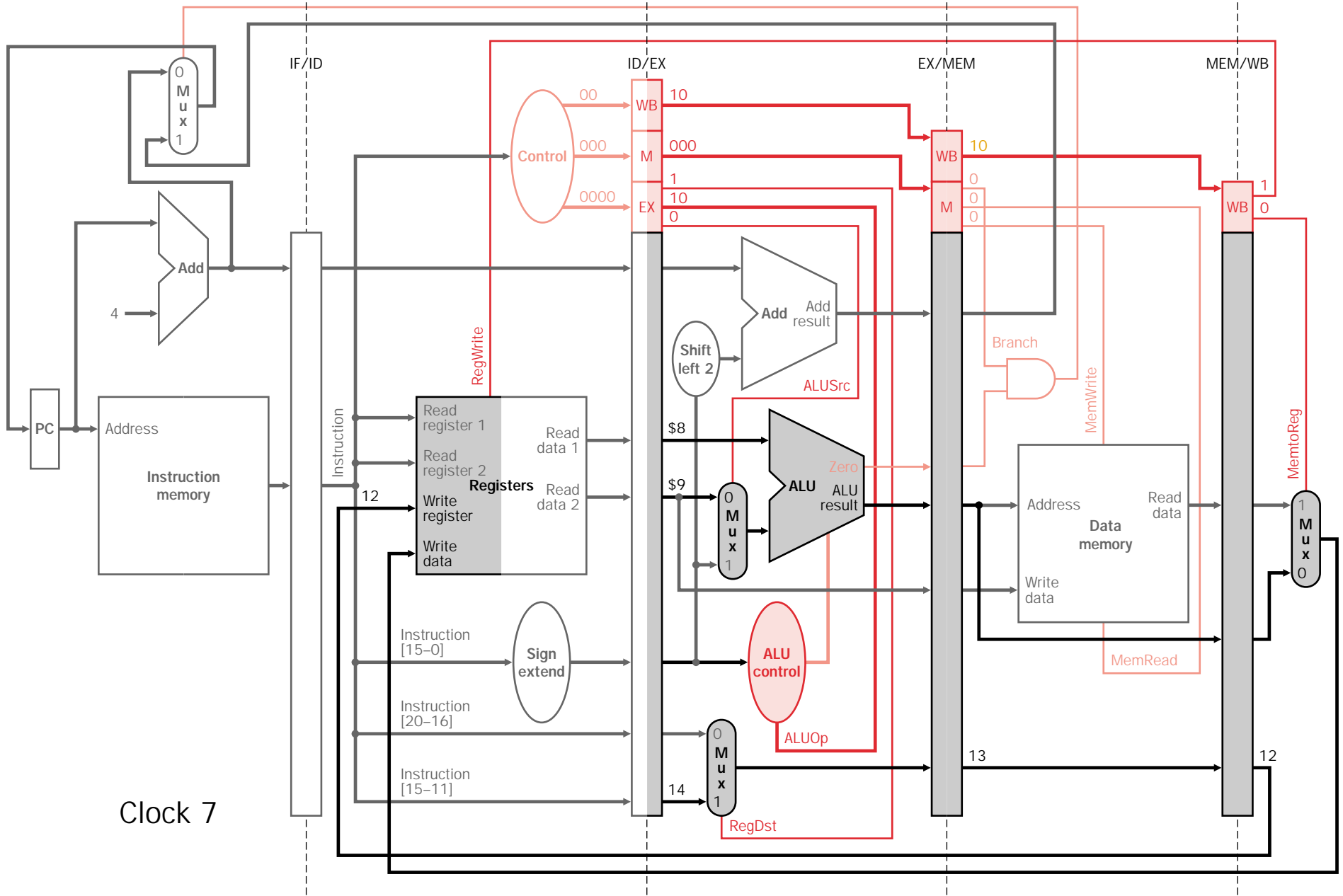
IF: after<2>

ID: after<1>

EX: add \$14, ...

MEM: or \$13, ...

WB: and \$12, ...



Clock 7

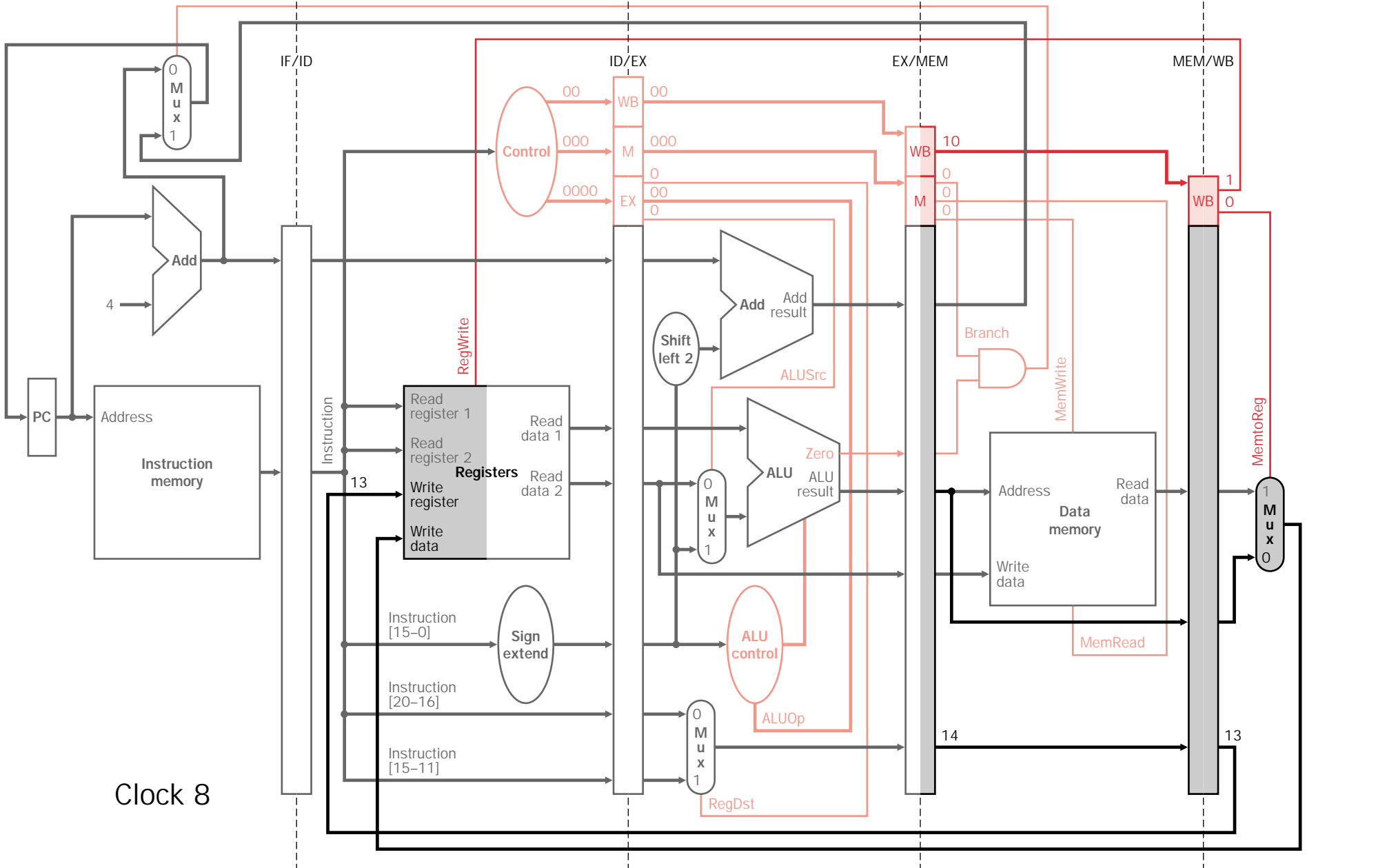
IF: after<3>

ID: after<2>

EX: after<1>

MEM: add \$14, ...

WB: or \$13, ...



Clock 8

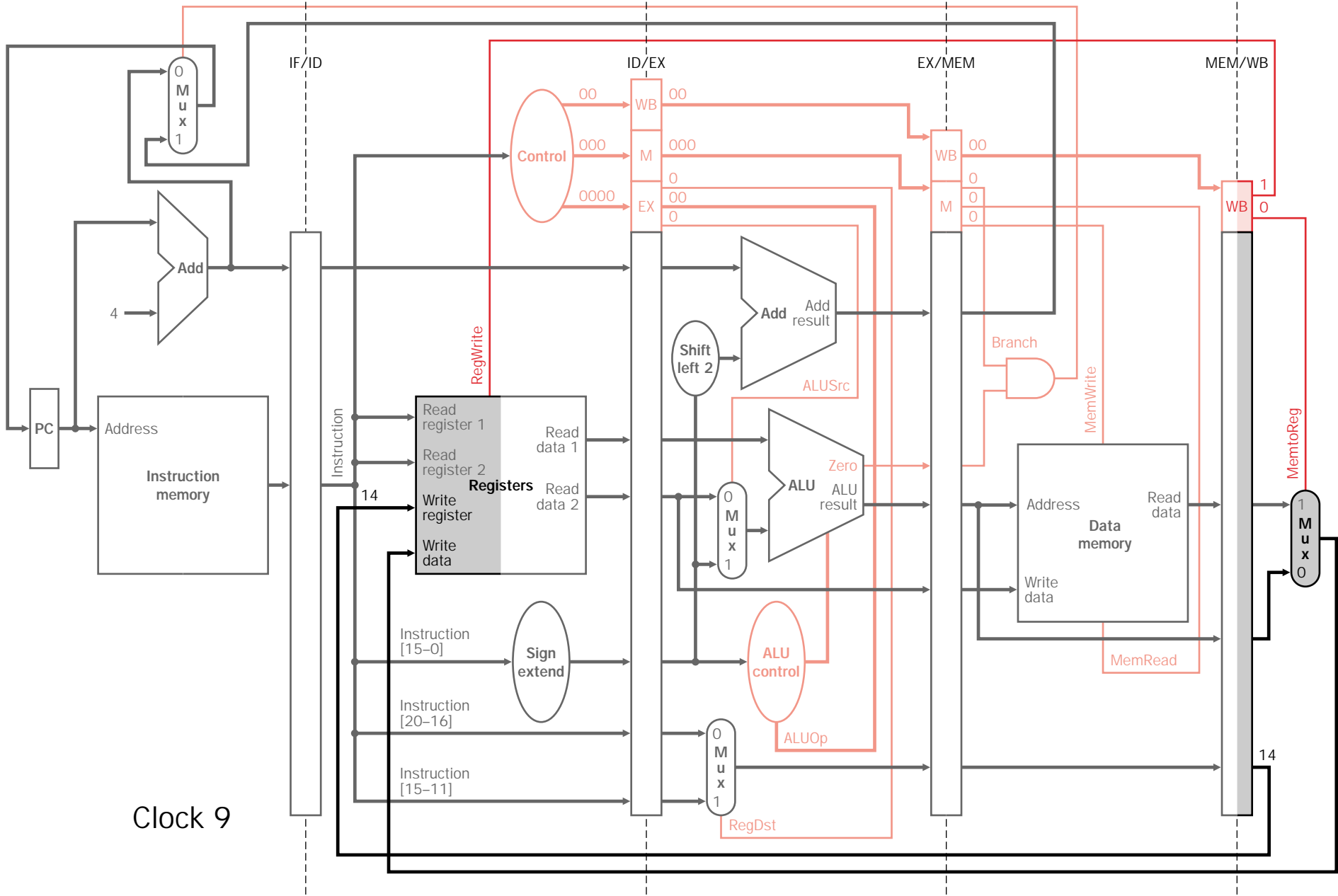
IF: after<4>

ID: after<3>

EX: after<2>

MEM: after<1>

WB: add \$14, ...



Clock 9

MIPS PIPELINES (3)

- MIPS R2000 integer unit pipeline hazards, named for pipeline registers:

$$\underbrace{\text{IF/ID}}_{\text{register}} . \underbrace{\text{ReadRegister}j}_{\text{name of register field}}$$

1. ID/EX.WriteRegister = IF/ID.ReadRegister j ($j = 1, 2$)
2. EX/MEM.WriteRegister = IF/ID.ReadRegister j ($j = 1, 2$)
3. MEM/WB.WriteRegister = IF/ID.ReadRegister j ($j = 1, 2$)

MIPS PIPELINES (4)

- MIPS R2000 integer pipeline hazards generated by the instructions

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

- ▷ The instruction `and $12,$2,$5` results in hazard 1a,
ID/EX.WriteRegister = IF/ID.ReadRegister1 = 2
- ▷ The instruction `or $13,$6,$2` results in hazard 2b,
EX/MEM.WriteRegister = IF/ID.ReadRegister2 = 2
- ▷ The instruction `add $14,$2,$2` results in hazards 3a and 3b,
MEM/WB.WriteRegister = IF/ID.ReadRegister1 = 2
MEM/WB.WriteRegister = IF/ID.ReadRegister2 = 2

Time (in clock cycles)

Value of register \$2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)

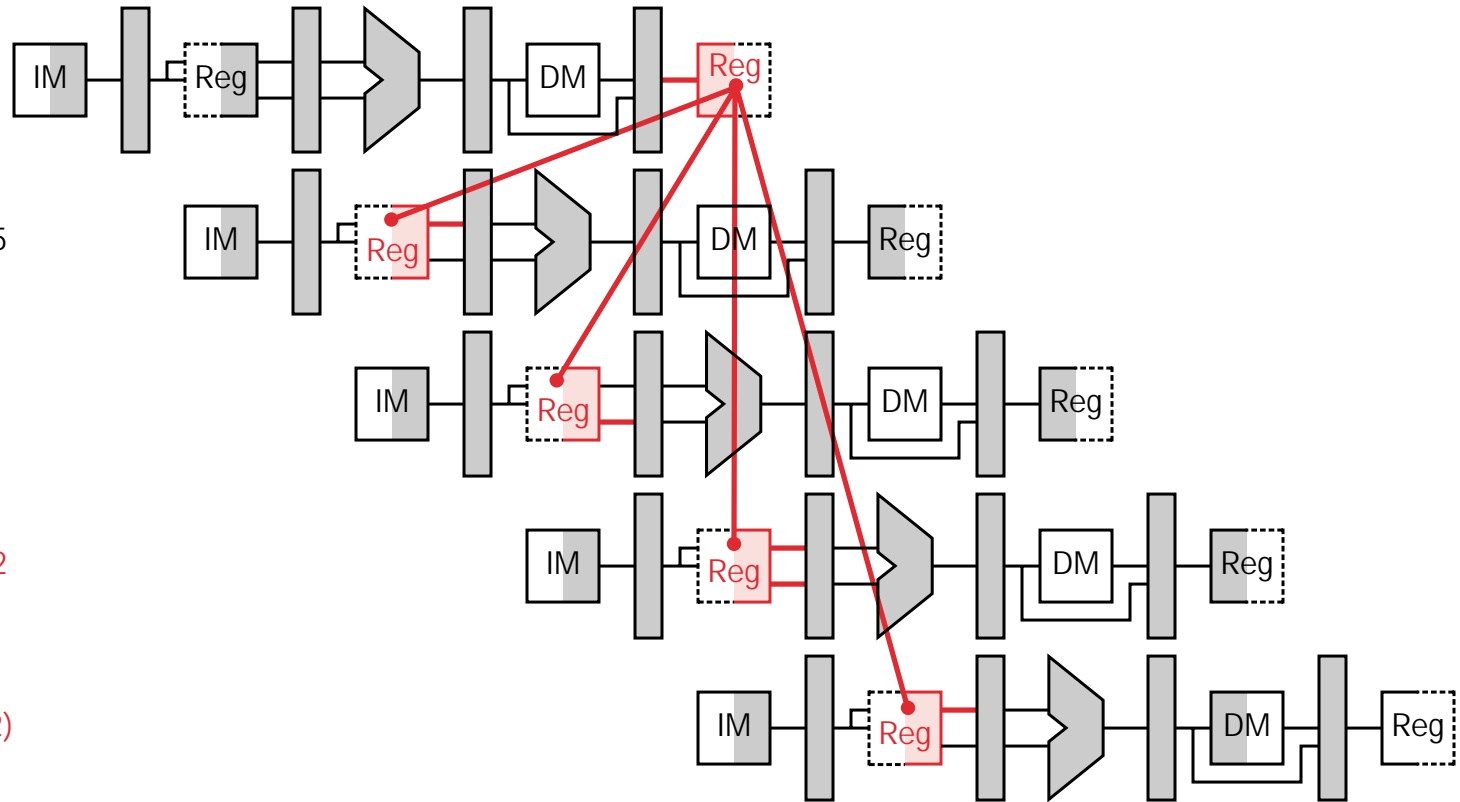
sub \$2, \$1, \$3

and \$12, \$2, \$5

or \$13, \$6, \$2

add \$14, \$2, \$2

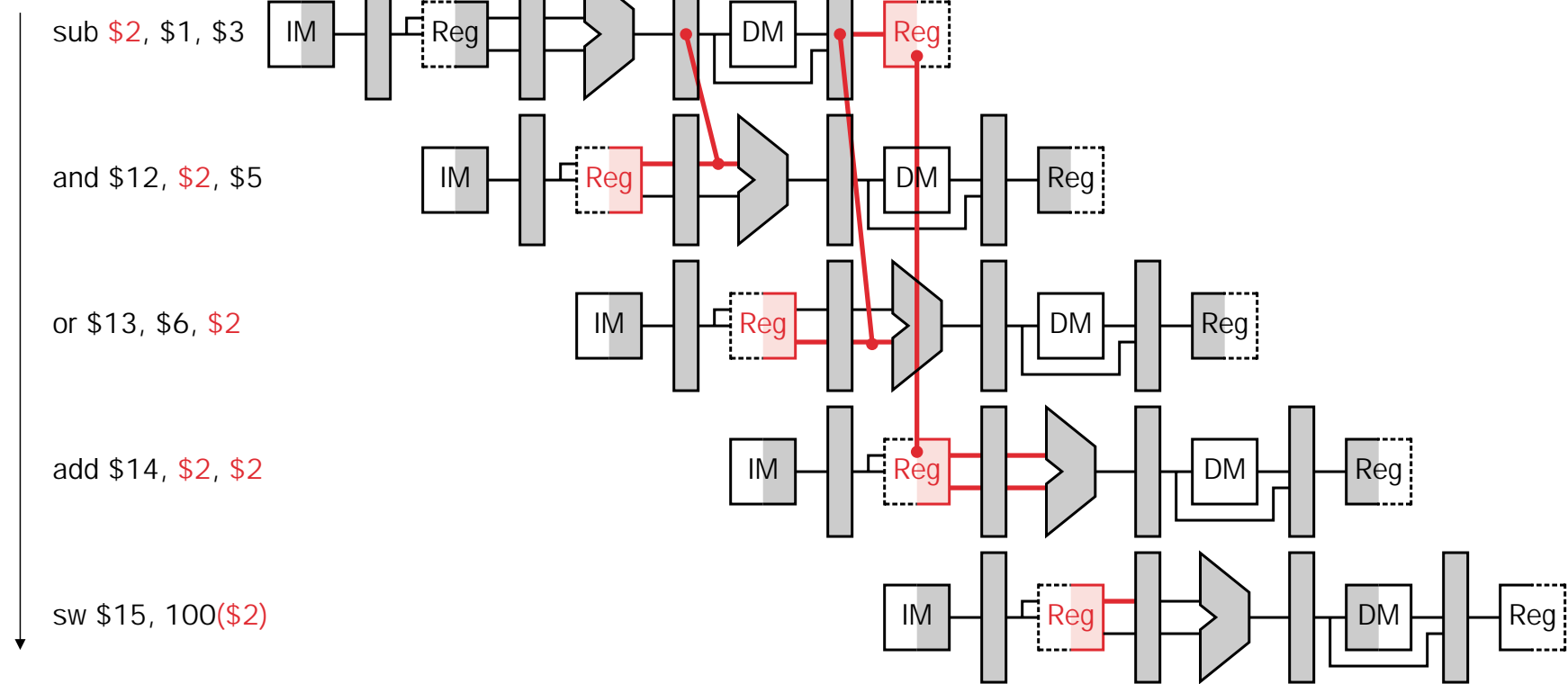
sw \$15, 100(\$2)



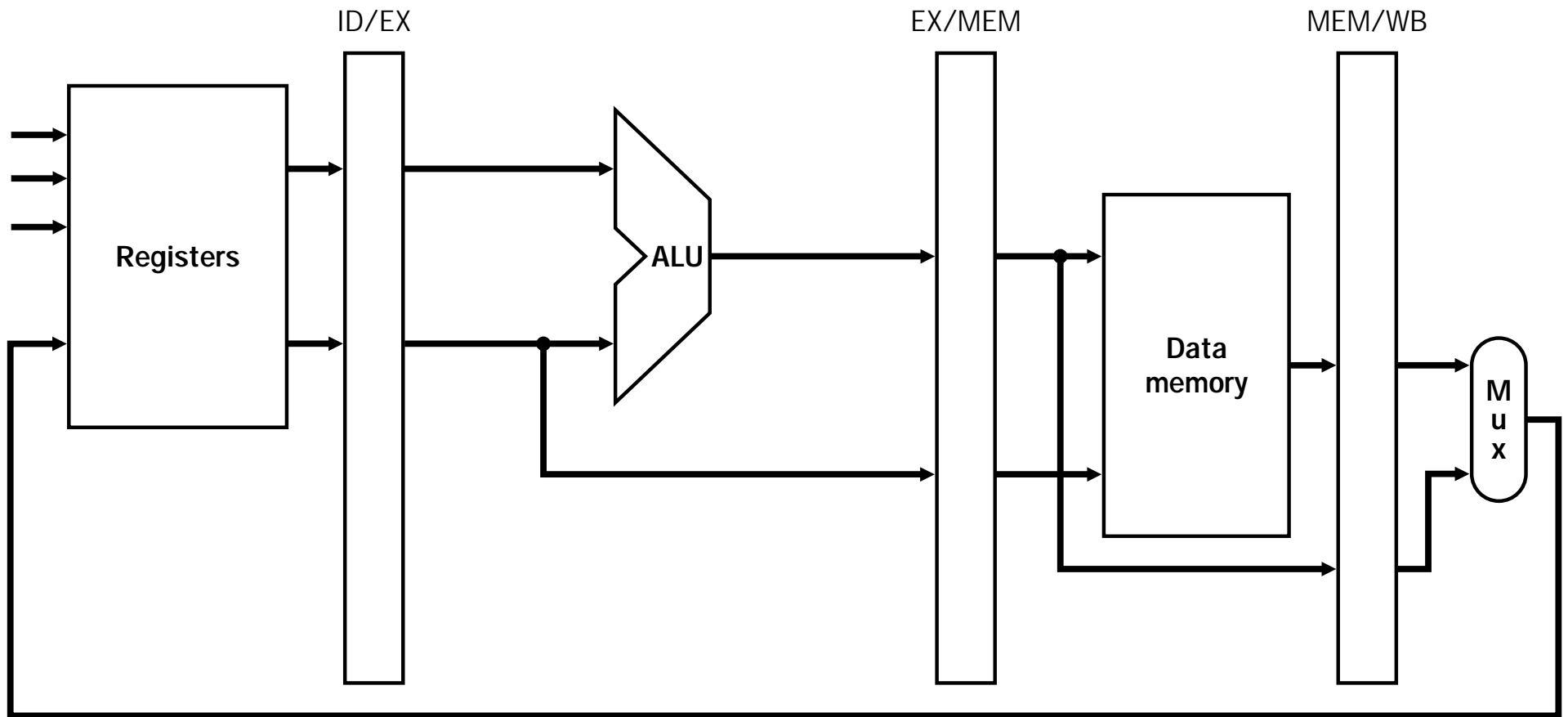
Time (in clock cycles) →

	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

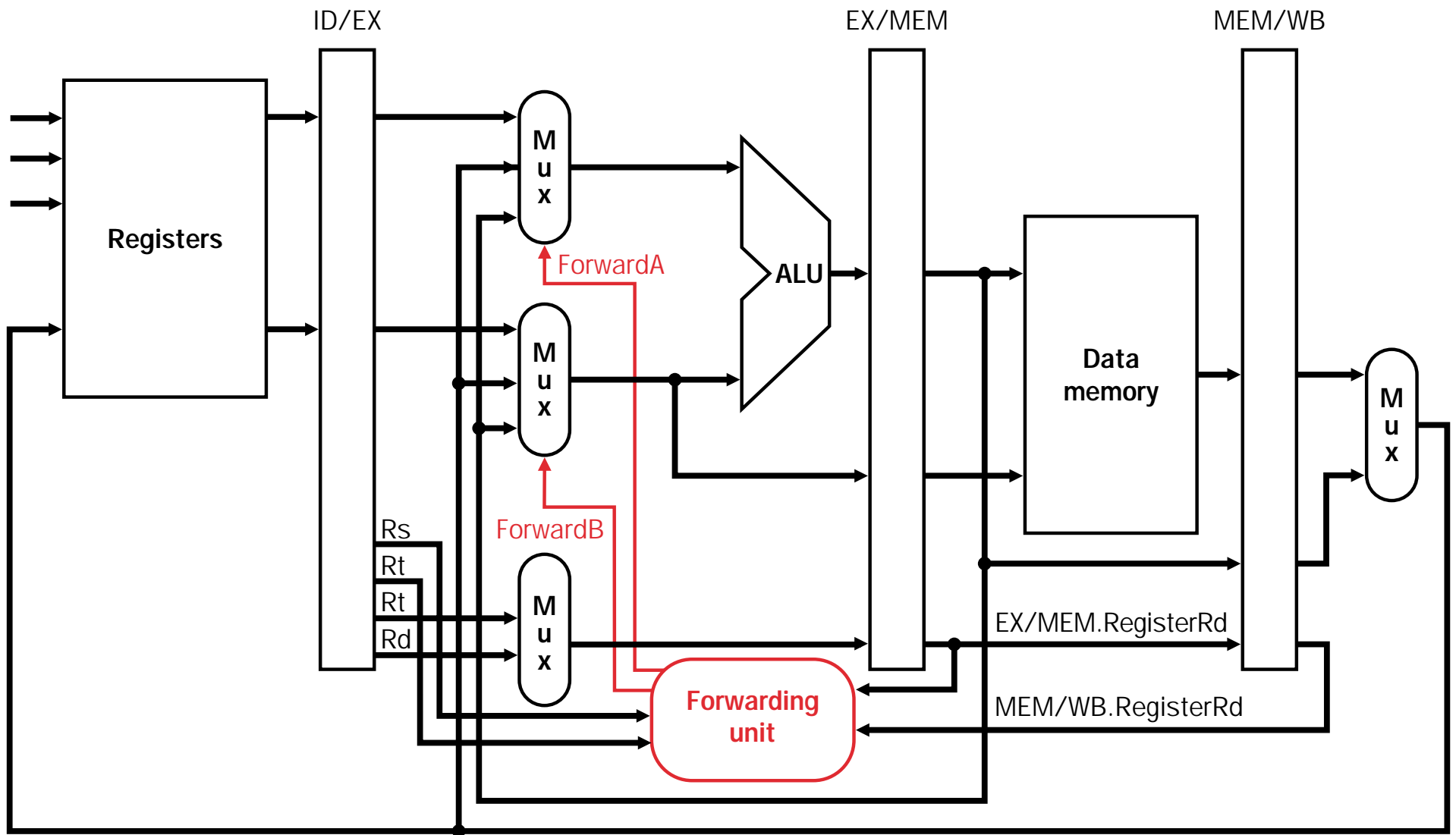
Program execution order (in instructions)



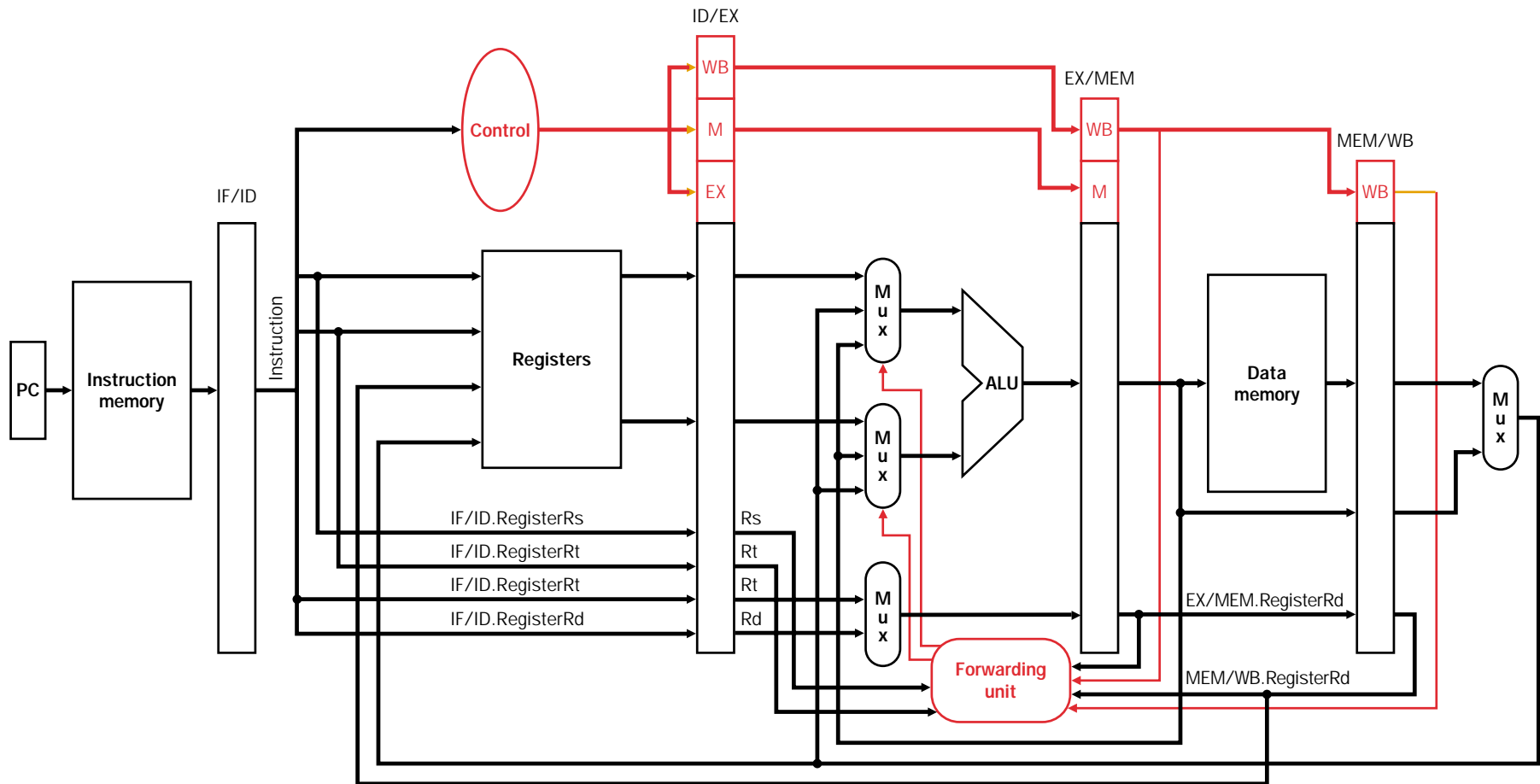
PIPELINED DATAPATH WITHOUT FORWARDING



PIPELINED DATAPATH WITH FORWARDING



DATAPATH MODIFIED TO RESOLVE HAZARDS BY FORWARDING



PIPELINED EXECUTION WITH FORWARDING

- We'll follow what happens in the instruction sequence

[40000028] sub \$2, \$1, \$3

[4000002c] and \$4, \$2, \$5

[40000030] or \$4, \$4, \$2

[40000034] add \$9, \$4, \$2

- ▷ Without forwarding, there would be RAW hazards on register \$2 in the **and** instruction and on register \$4 in the **or** and **add** instructions

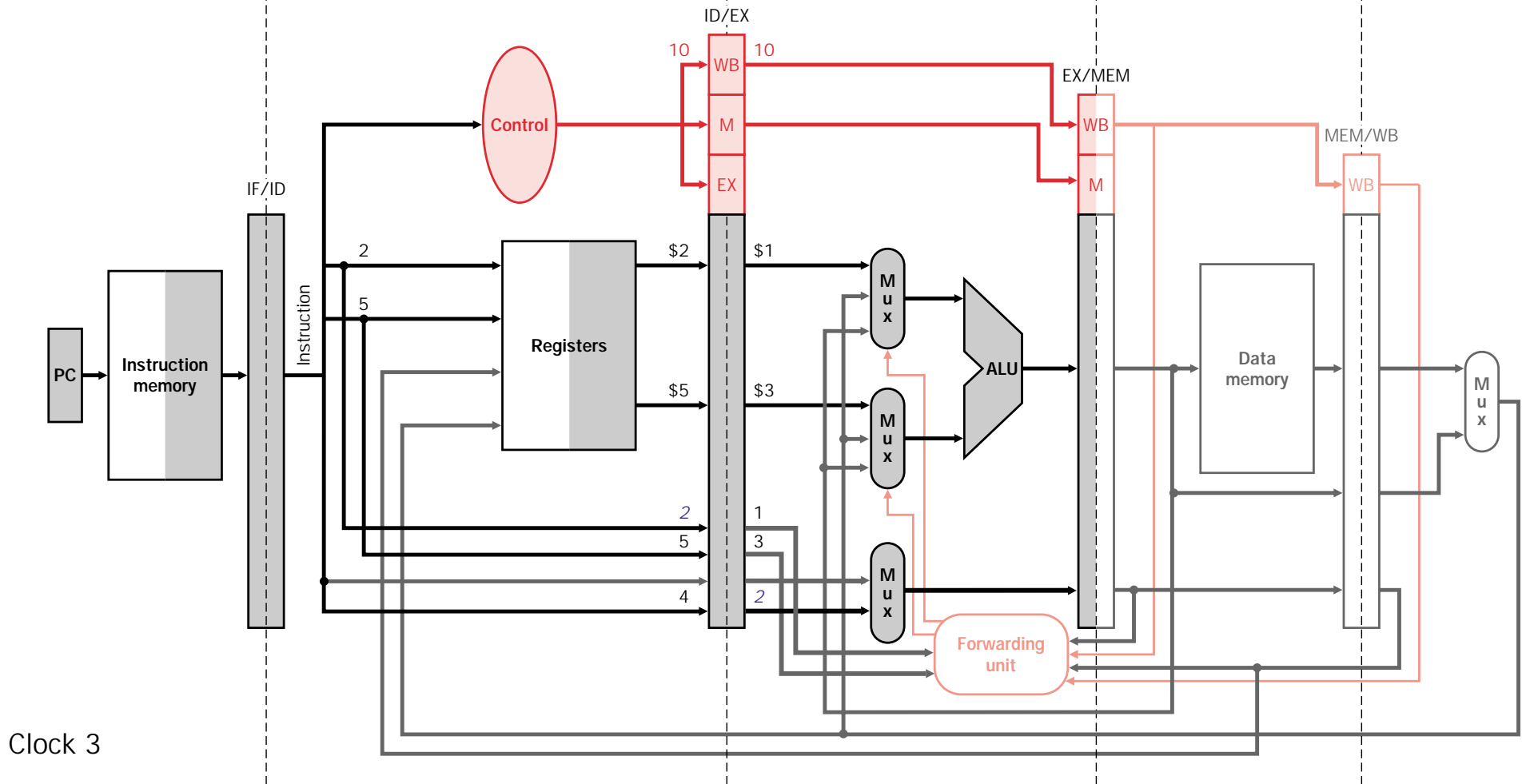
or \$4, \$4, \$2

and \$4, \$2, \$5

sub \$2, \$1, \$3

before<1>

before<2>



ID/EX.WriteRegister
 = IF/ID.ReadRegister1
 = 2
 (RAW data hazard)

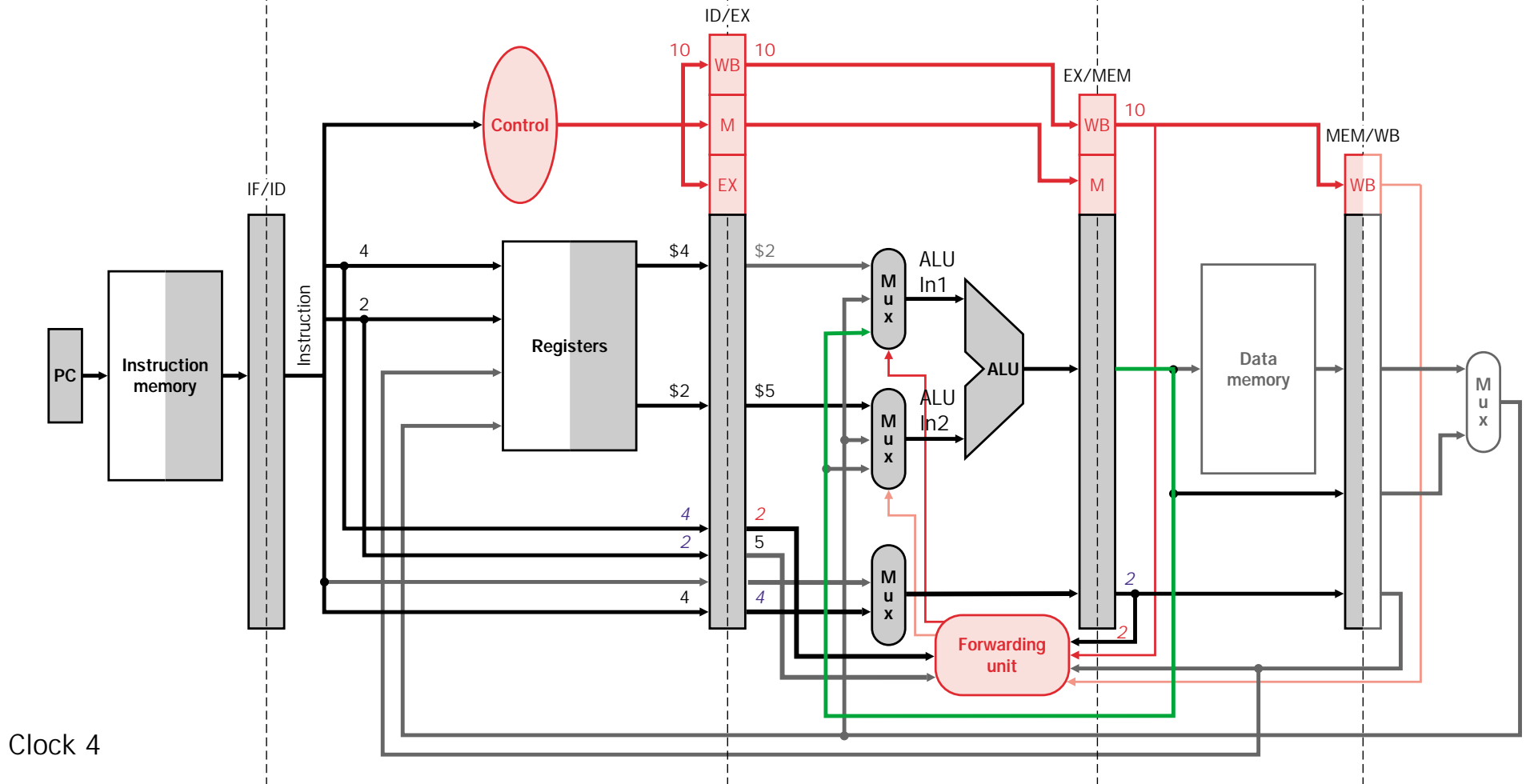
add \$9, \$4, \$2

or \$4, \$4, \$2

and \$4, \$2, \$5

sub \$2, ...

before <1>



Clock 4

ID/EX.WriteRegister
 = IF/ID.ReadRegister1
 = 4 and
 EX/MEM.WriteRegister
 = IF/ID.ReadRegister2
 = 2
 (RAW data hazards)

EX/MEM.WriteRegister
 = ID/EX.ReadRegister1
 = 2
 (test by which the need
 for forwarding to ALUIn1
 is actually detected)

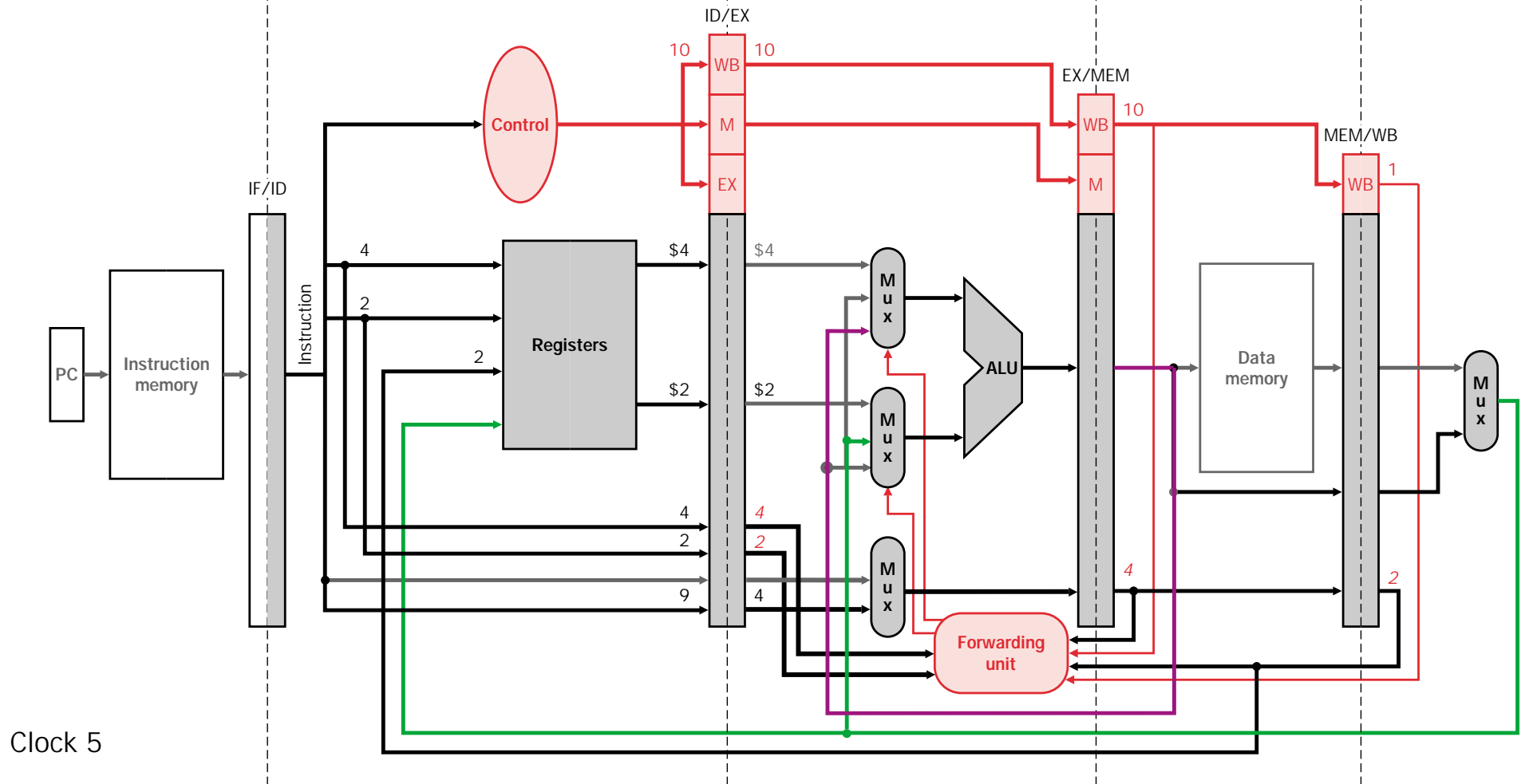
after<1>

add \$9, \$4, \$2

or \$4, \$4, \$2

and \$4, ...

sub \$2, ...



Clock 5

EX/MEM.WriteRegister
 = IF/ID.ReadRegister1
 = 4 and
 MEM/WB.WriteRegister
 = IF/ID.ReadRegister2
 = 2

EX/MEM.WriteRegister
 = ID/EX.ReadRegister1
 = 4 and
 MEM/WB.WriteRegister
 = ID/EX.ReadRegister2
 = 2

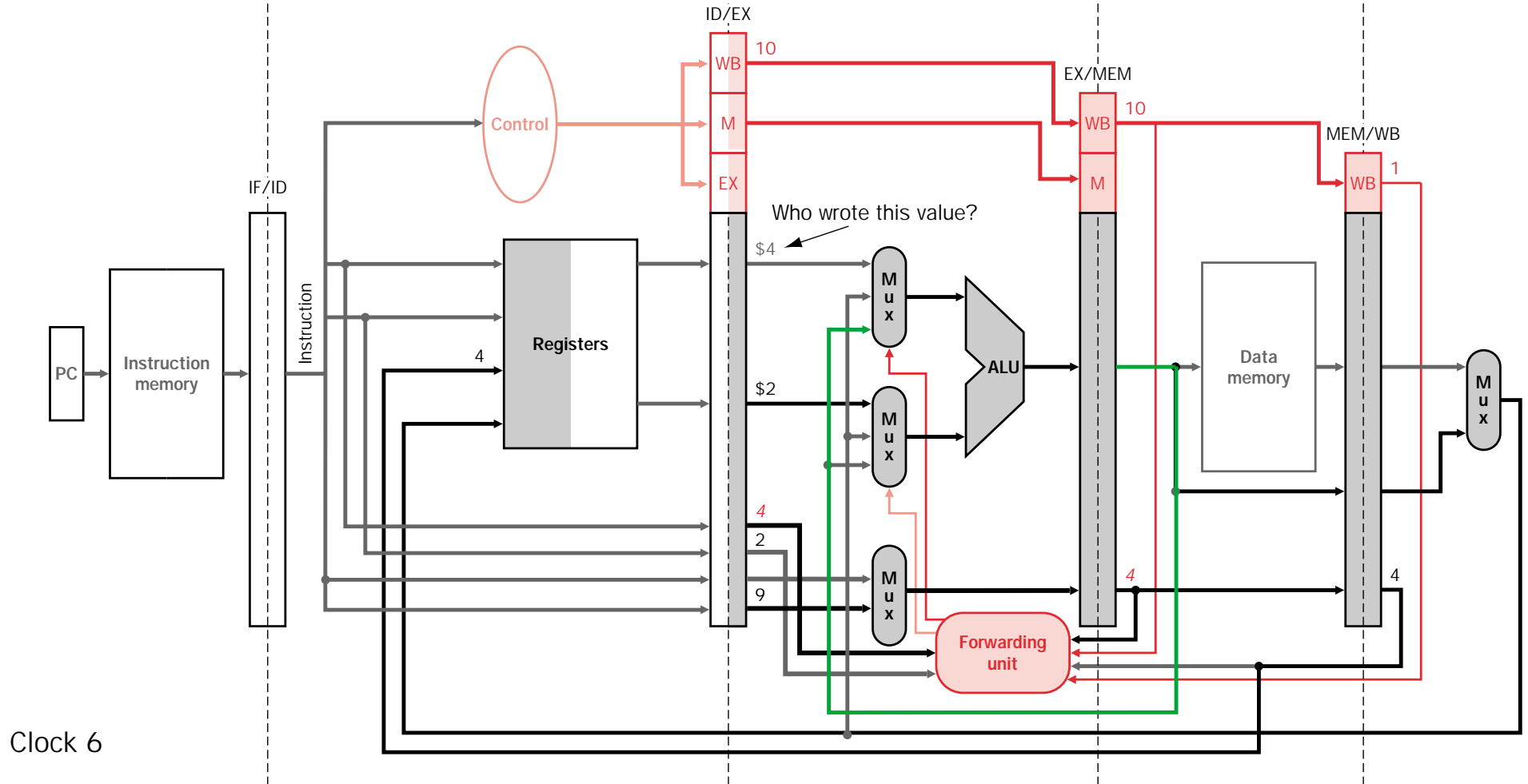
after<2>

after<1>

add \$9, \$4, \$2

or \$4, ...

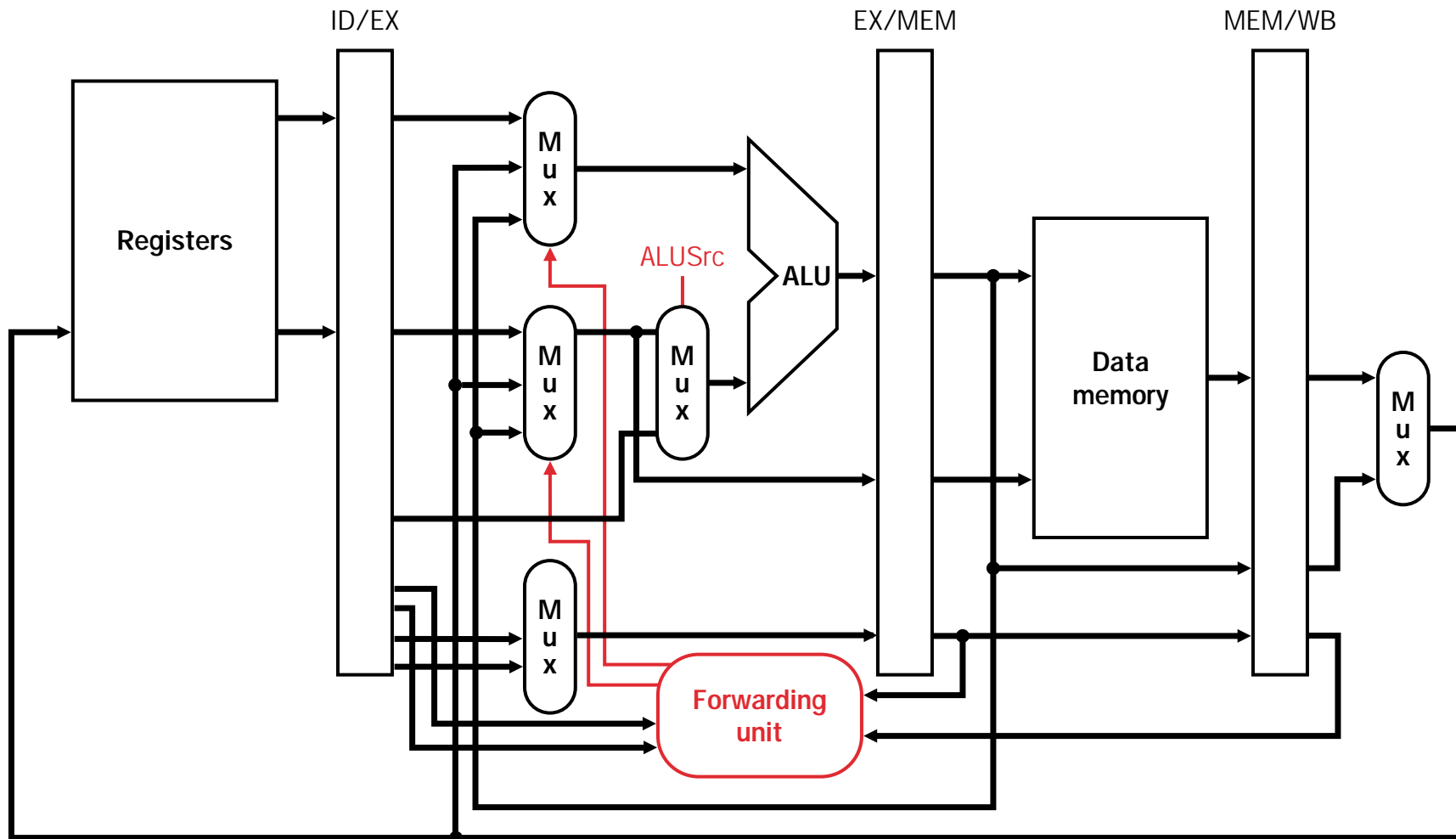
and \$4, ...



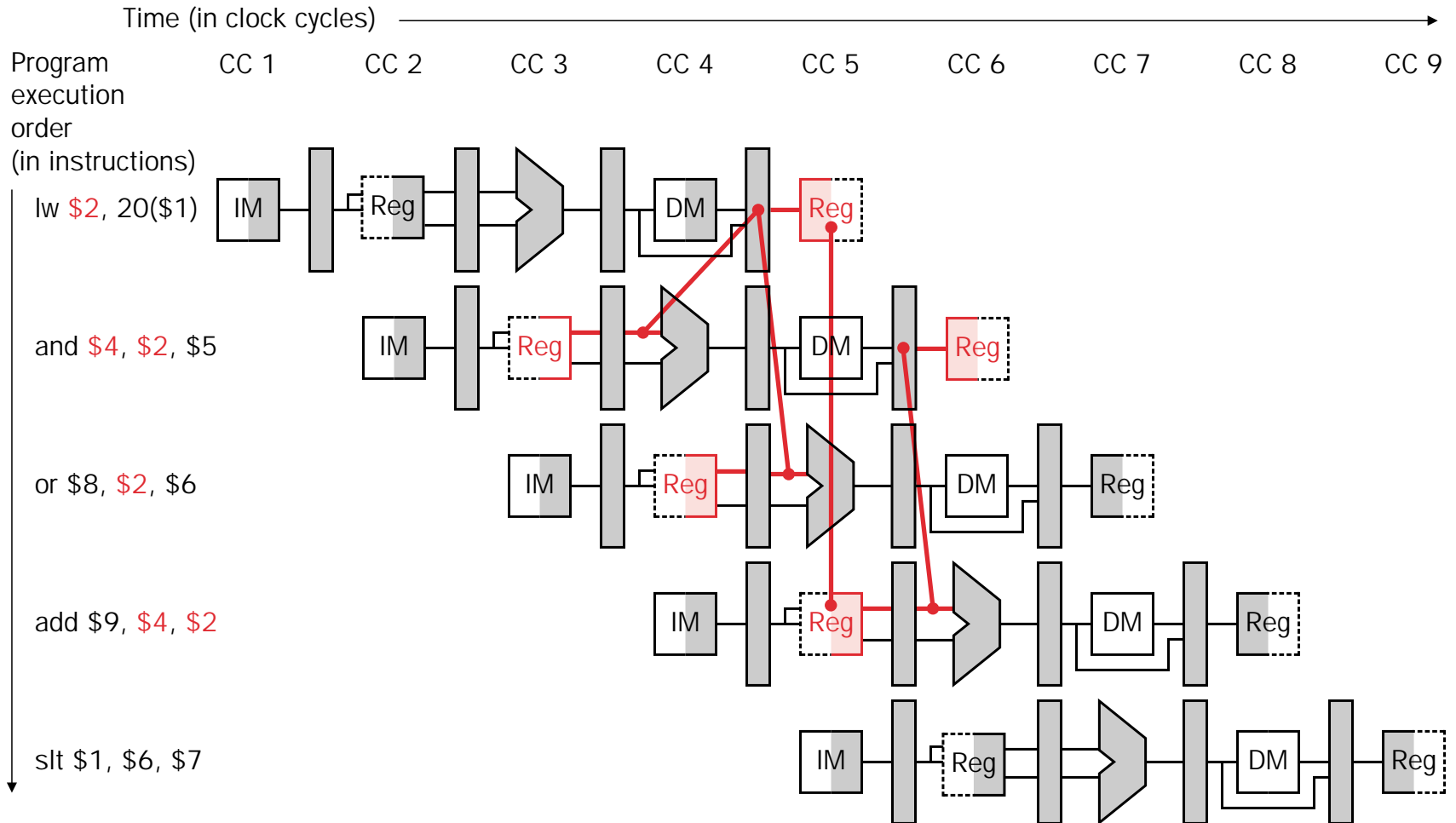
Clock 6

$$\begin{aligned} \text{EX/MEM.WriteRegister} &= \text{ID/EX.ReadRegister2} \\ &= 4 \end{aligned}$$

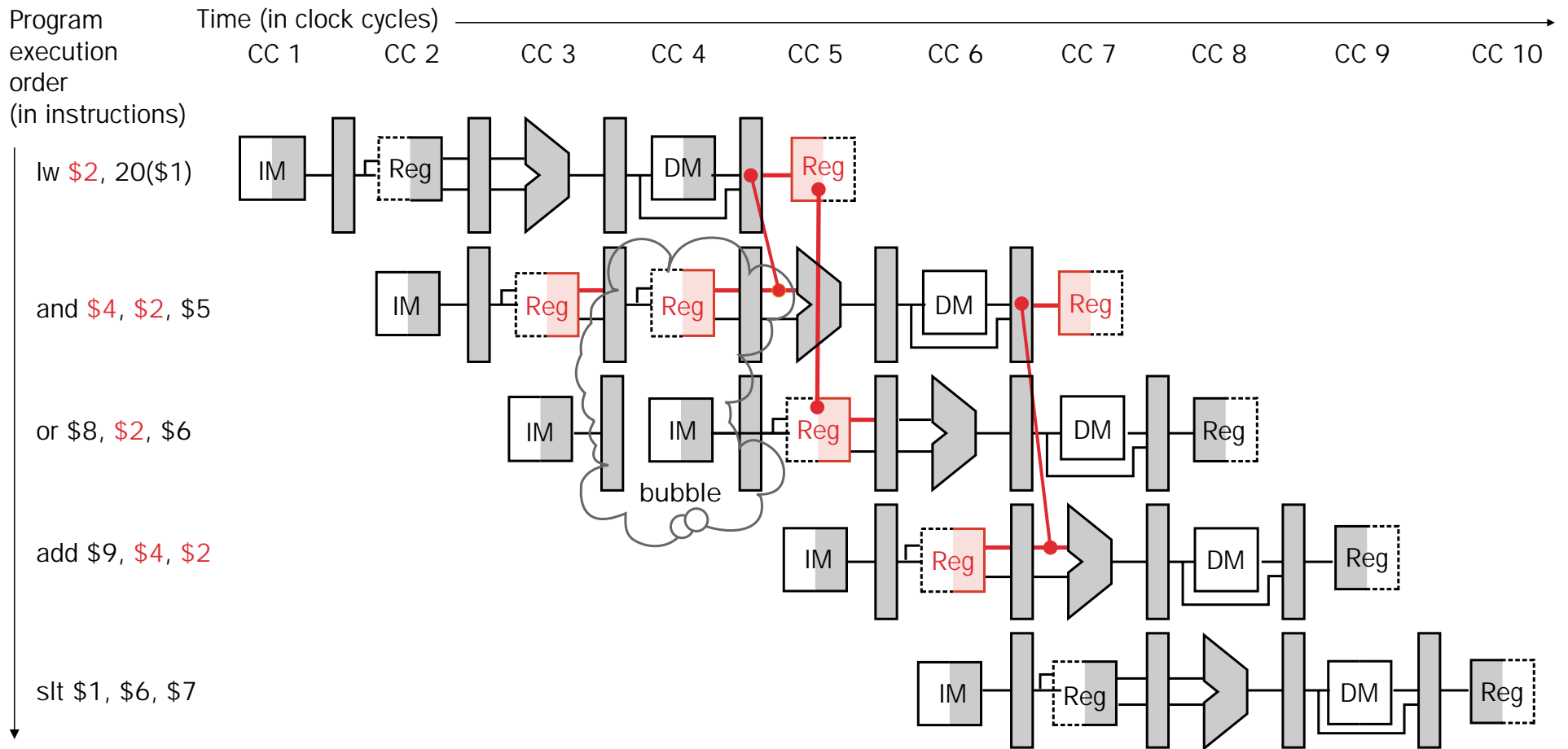
ADDITION OF A MULTIPLEXOR TO CHOOSE THE IMMEDIATE VALUE



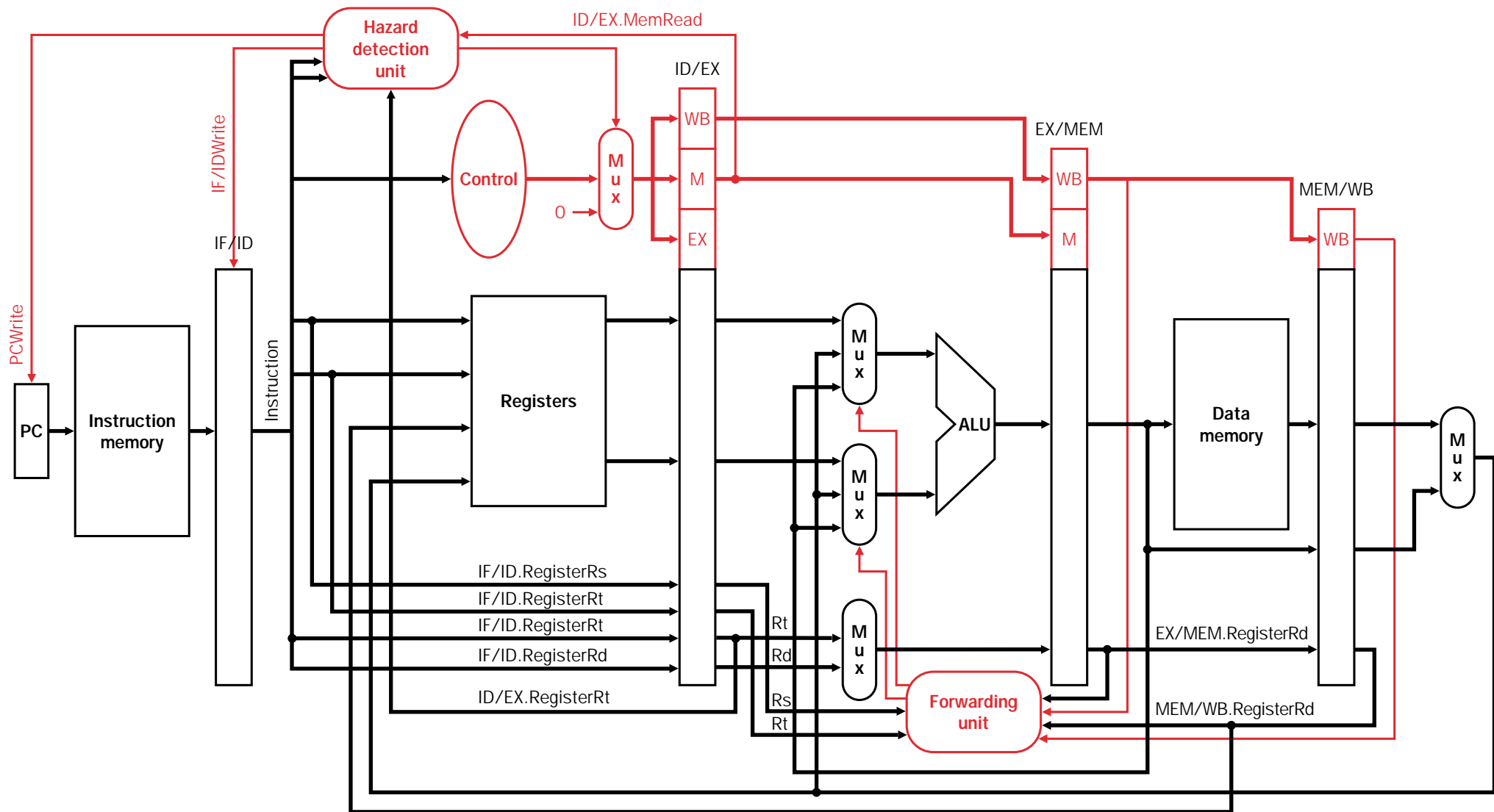
A DATA HAZARD THAT CANNOT BE RESOLVED BY FORWARDING



HOW STALLS ARE INSERTED INTO A PIPELINE



OVERVIEW OF PIPELINED CONTROL



PIPELINED EXECUTION WITH A STALL

- We'll follow what happens in the instruction sequence

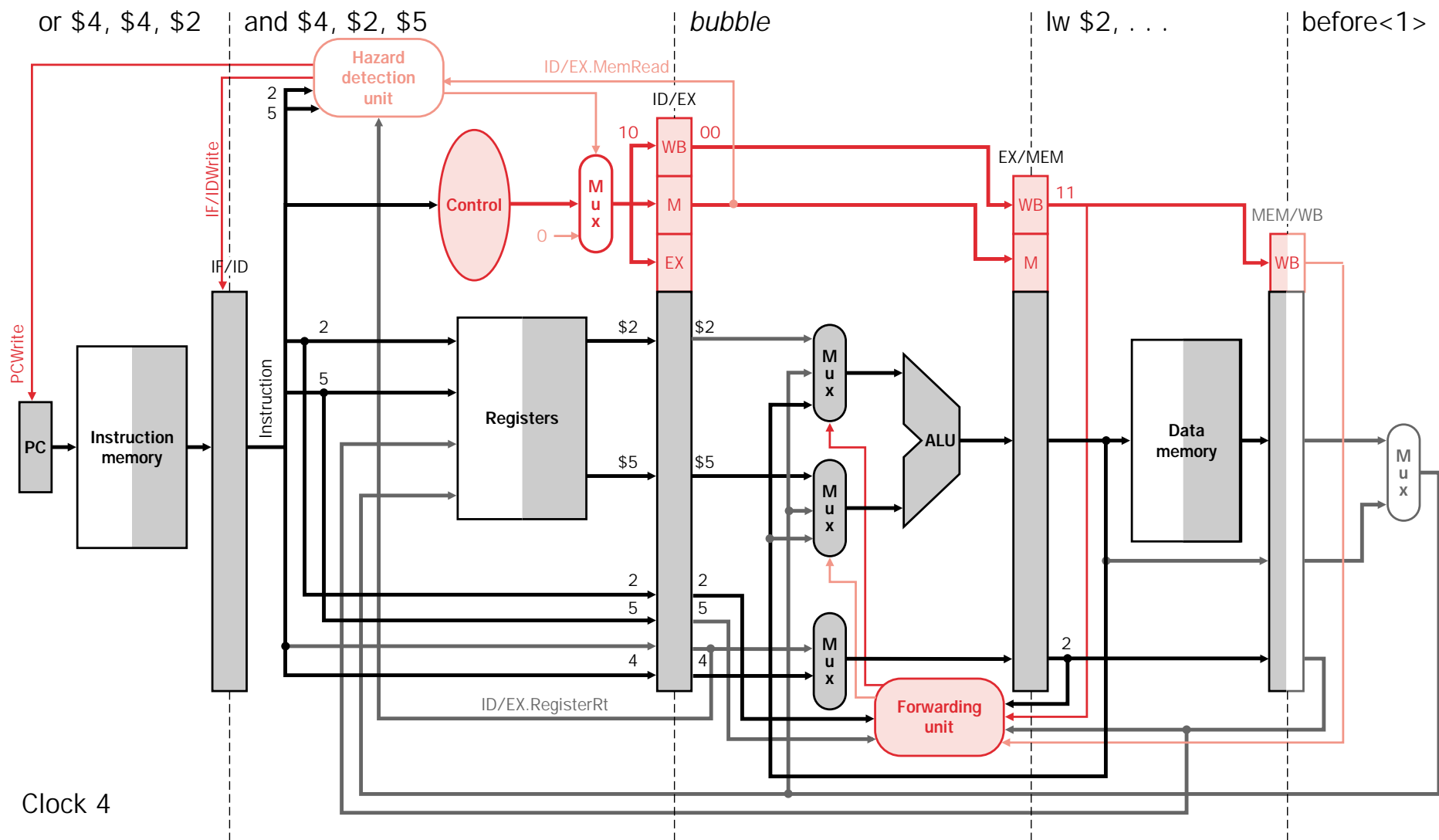
[40000028] lw \$2, 20(\$1)

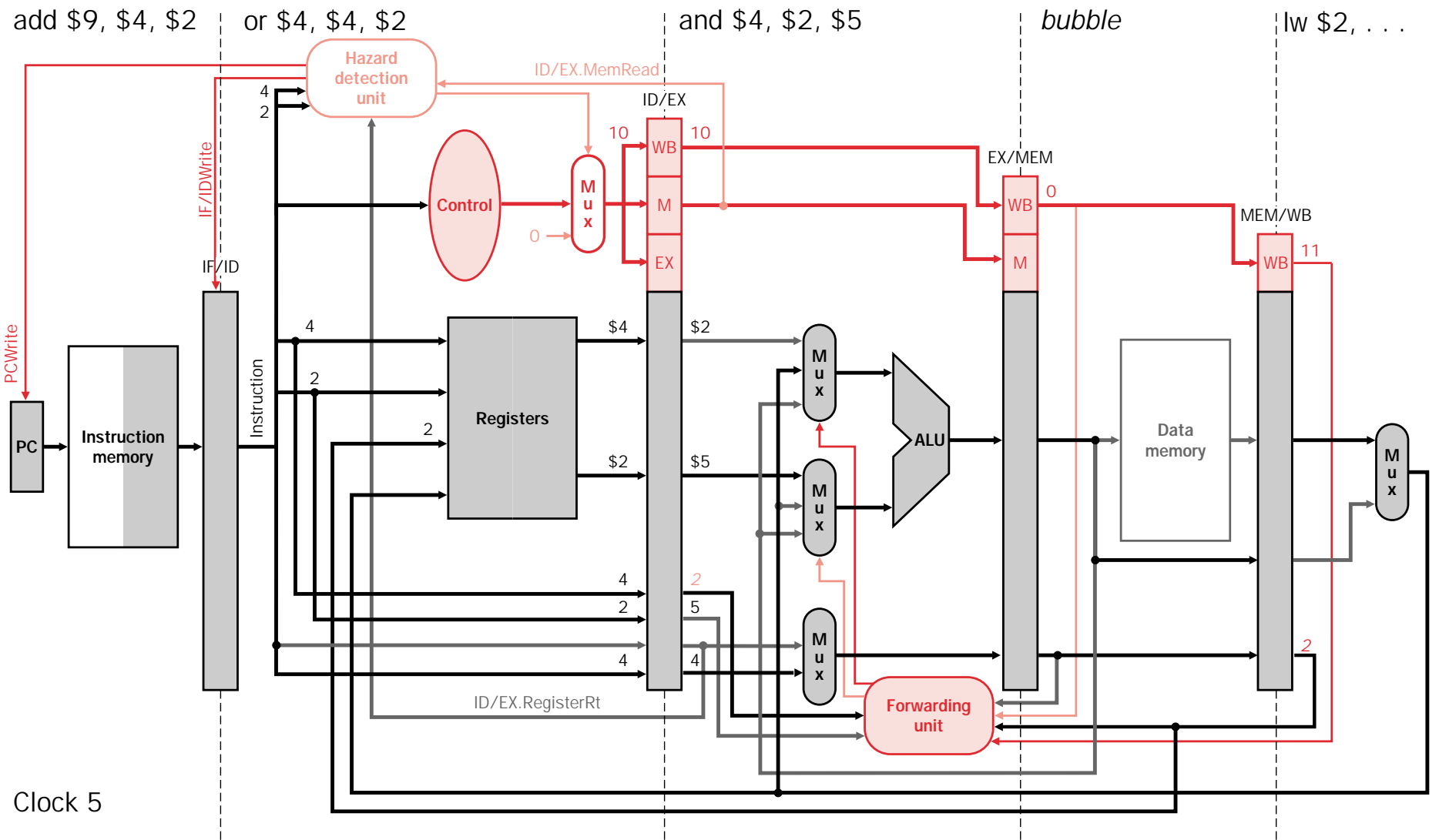
[4000002c] and \$4, \$2, \$5

[40000030] or \$4, \$4, \$2

[40000034] add \$9, \$4, \$2

- ▷ Note that the hardware inserts a stall after the **lw** instruction
- ▷ Forwarding resolves the RAW hazards on register **\$2** in the **and** instruction and on register **\$4** in the **or** and **add** instructions





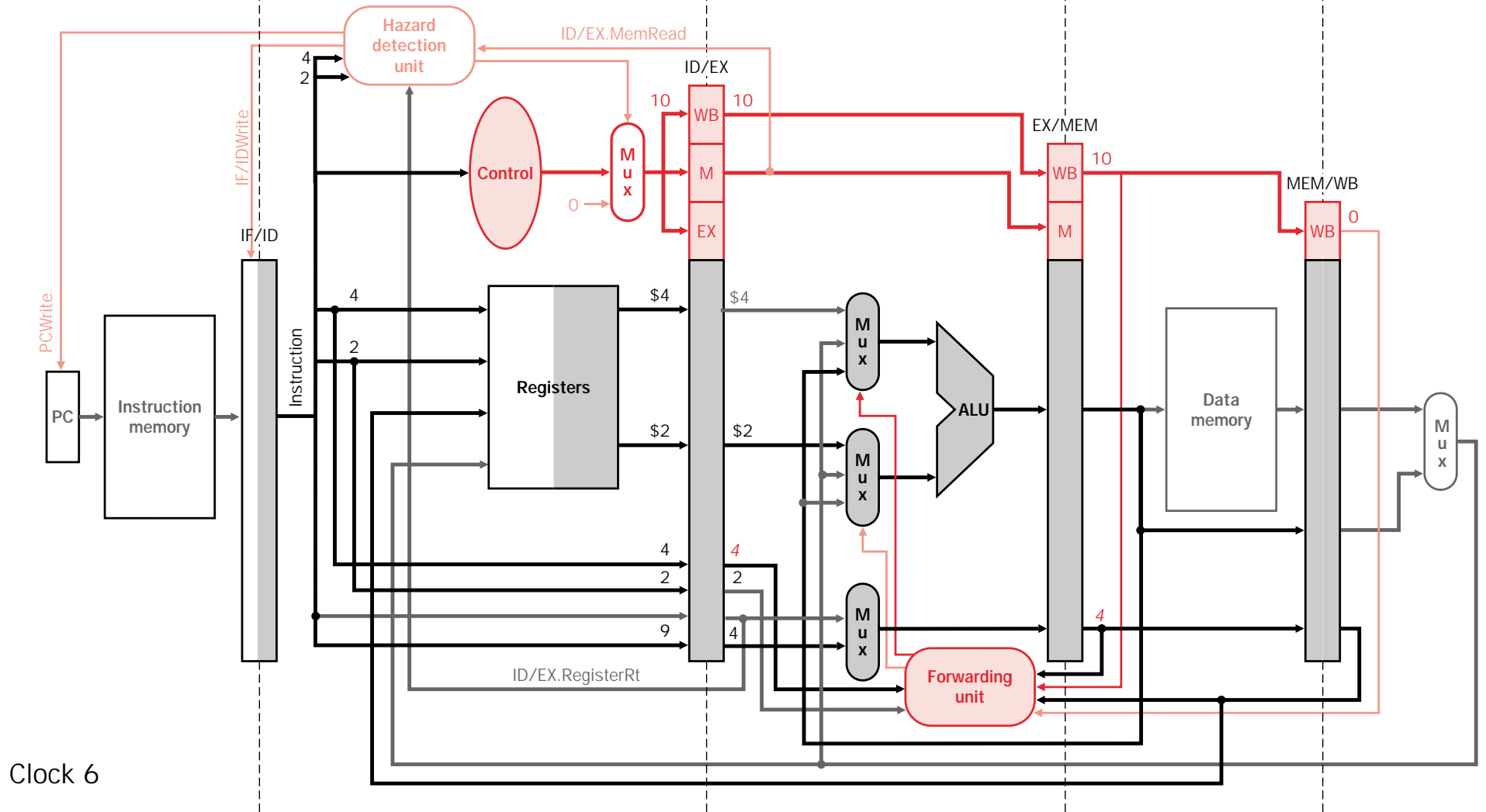
after<1>

add \$9, \$4, \$2

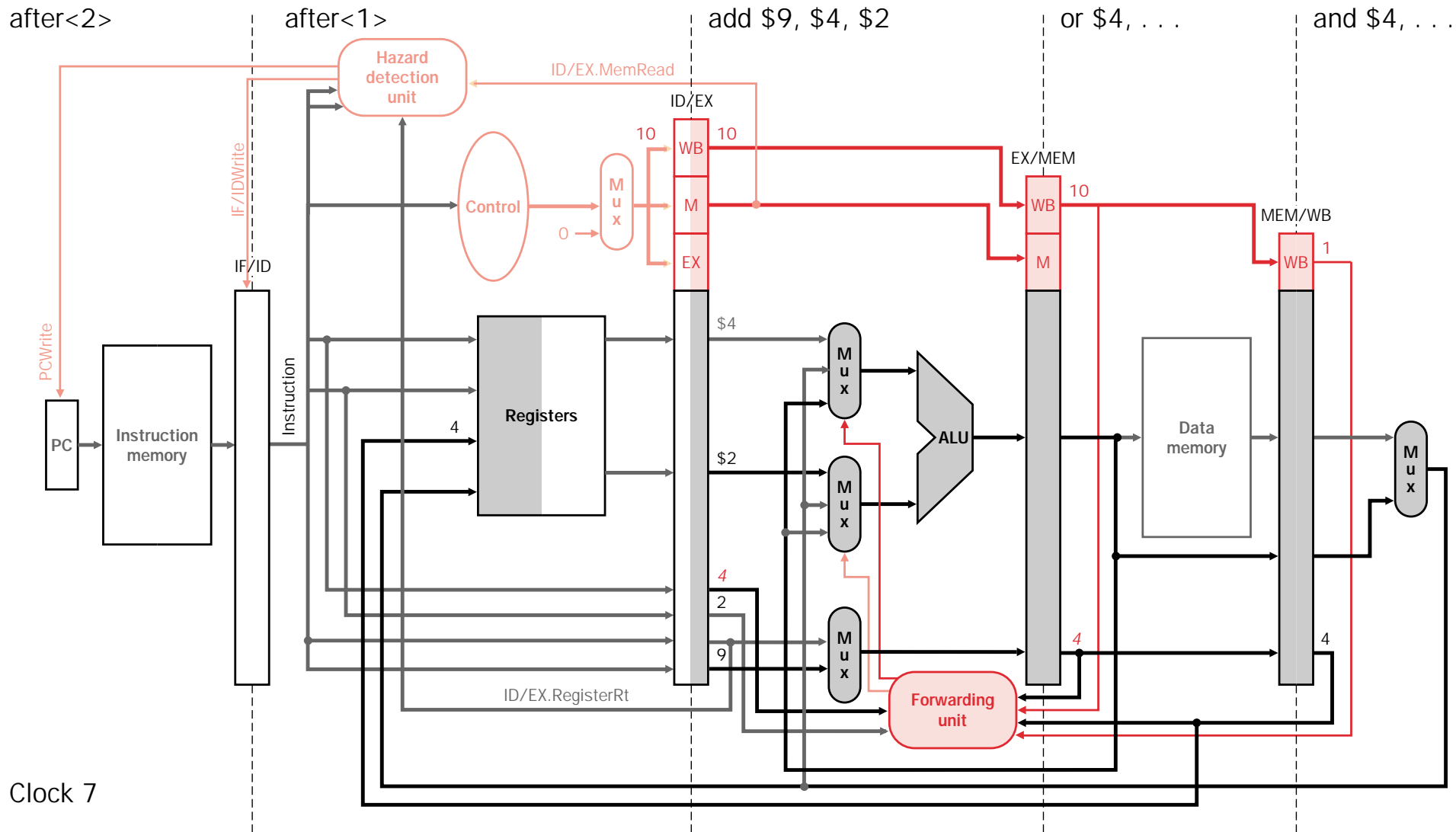
or \$4, \$4, \$2

and \$4, ...

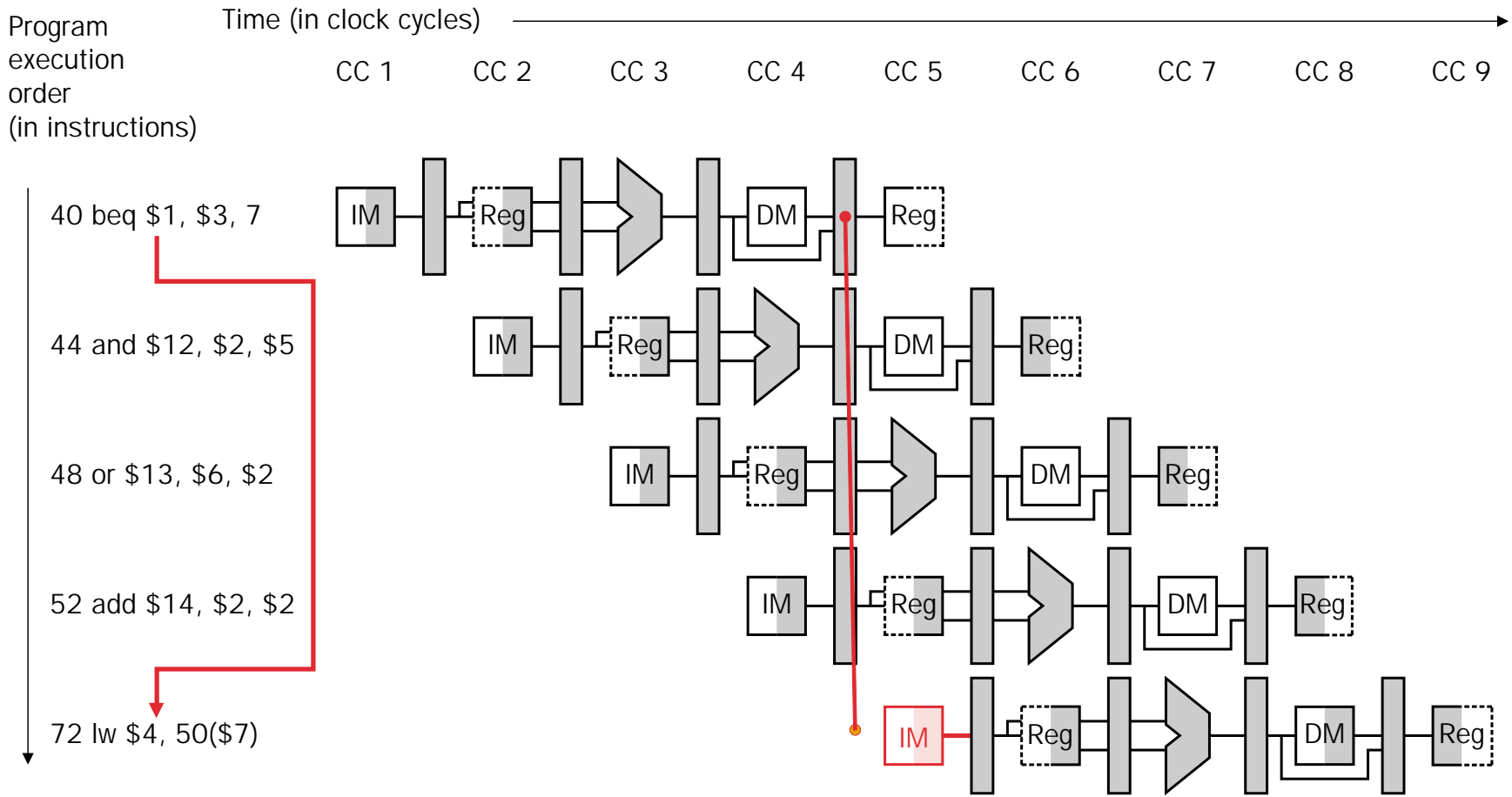
bubble



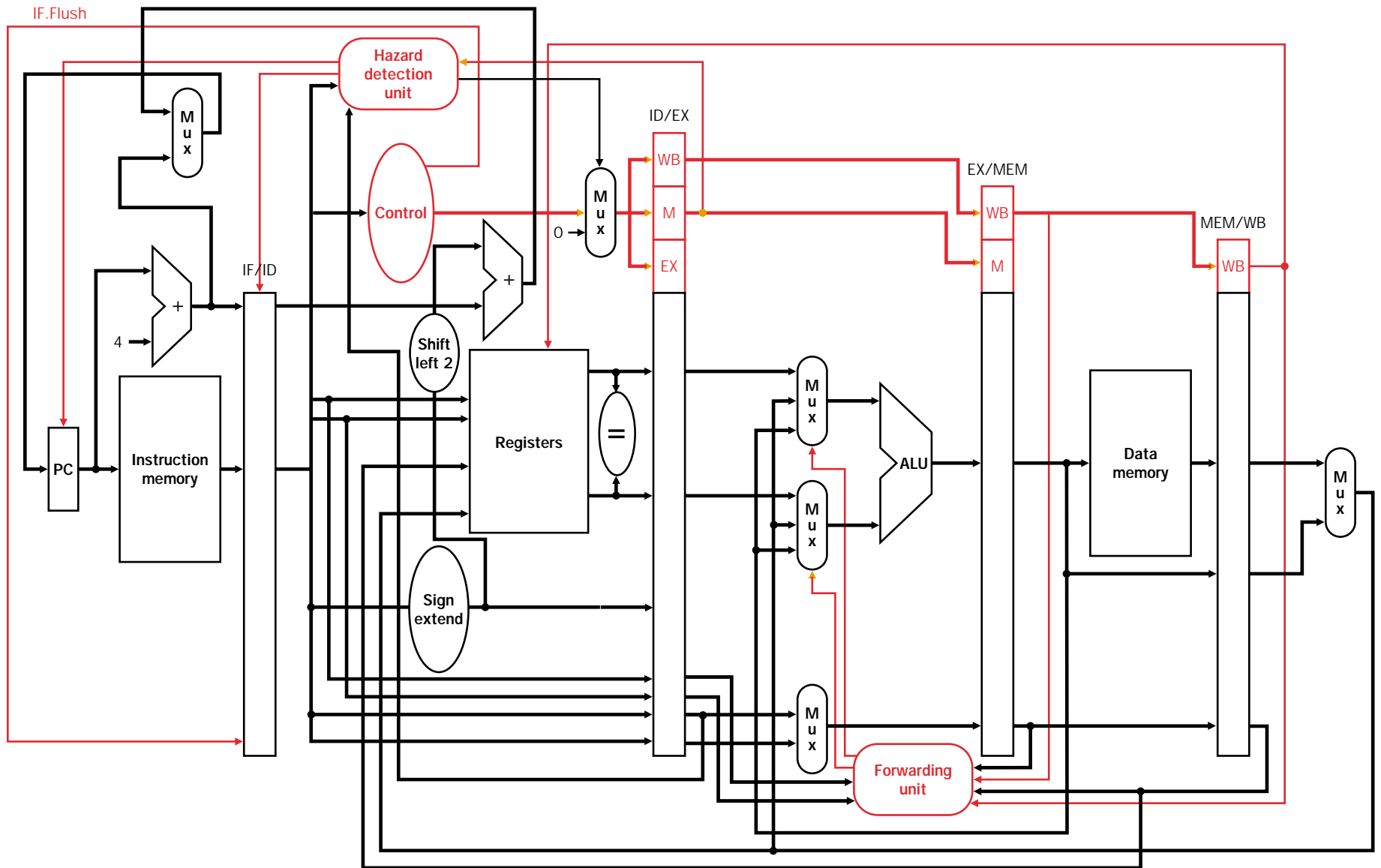
Clock 6



EFFECT OF A PIPELINE ON A BRANCH INSTRUCTION



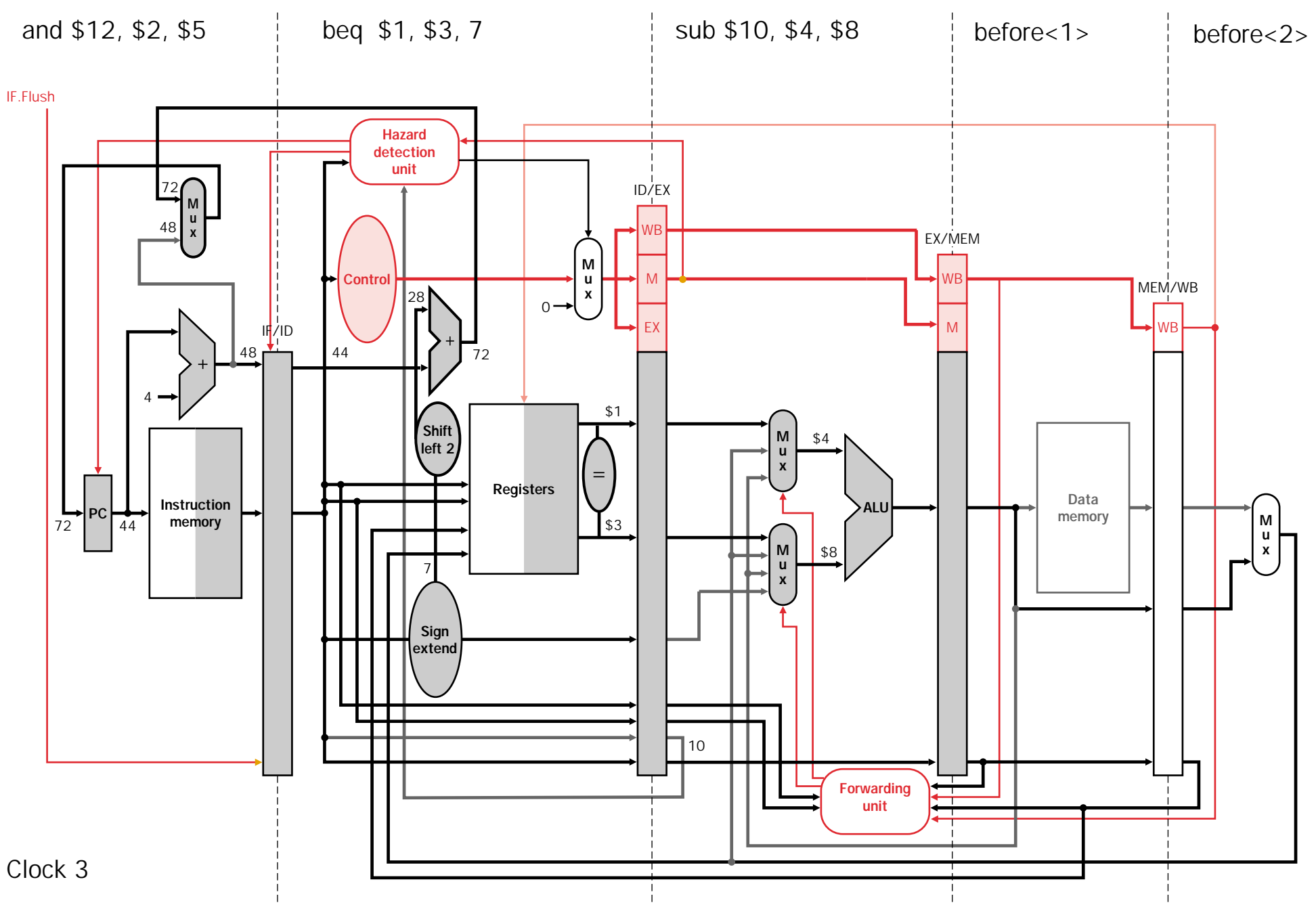
PIPELINED DATAPATH INCLUDING SUPPORT FOR BRANCHES



A PIPELINED BRANCH

- We'll follow what happens in the instruction sequence

```
[40000024] sub $10, $4, $8
[40000028] beq $1, $3, $7 # PC-relative branch to offset
[4000002c] and $12, $2, $5 # 40 + 4 + 7*4 = 72 = 0x48
[40000030] or $14, $2, $6
[40000034] add $14, $4, $2
[4000003c] slt $15, $6, $7
. . .
[40000048] lw $14, 50($7)
```



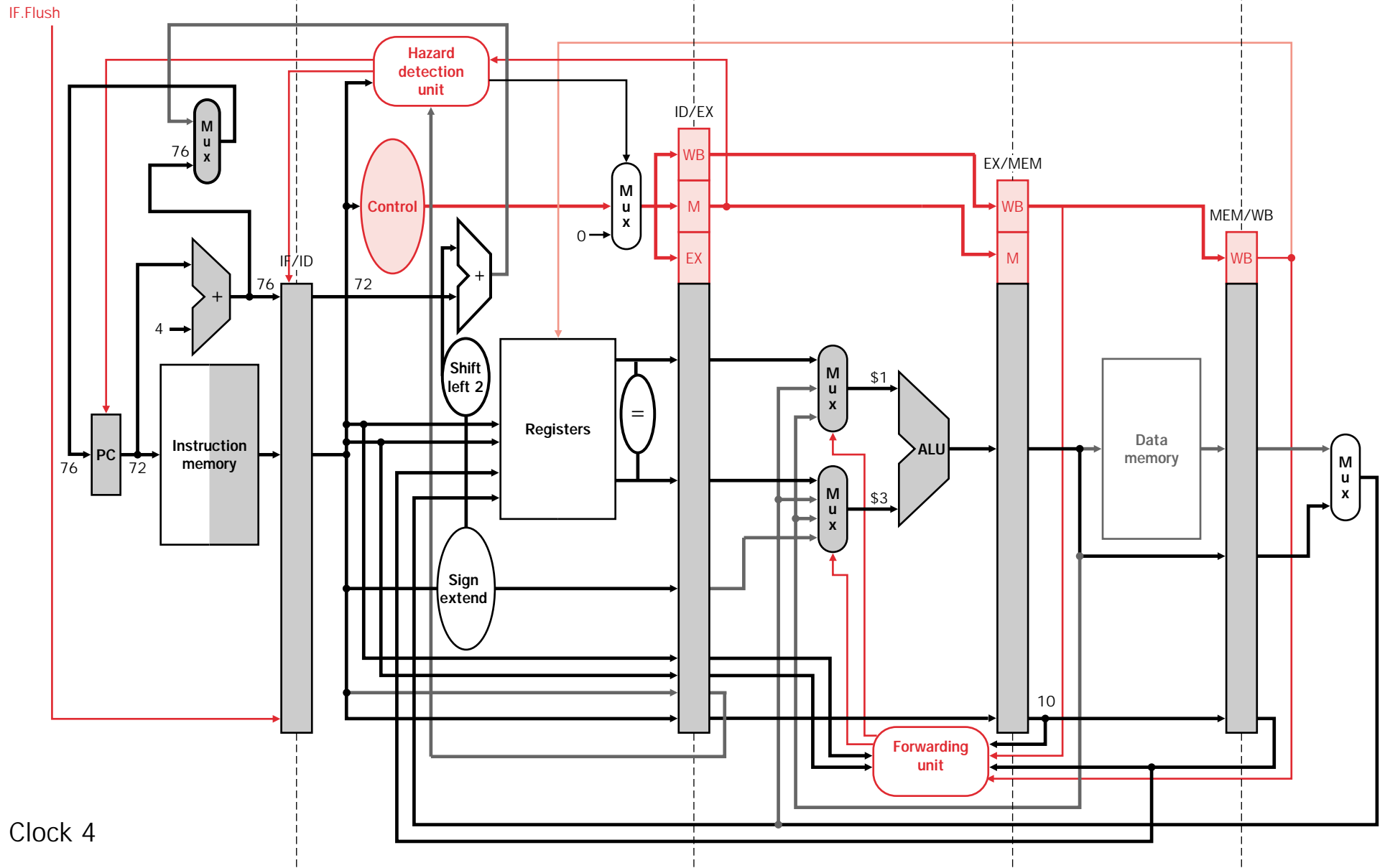
lw \$4, 50(\$7)

bubble (nop)

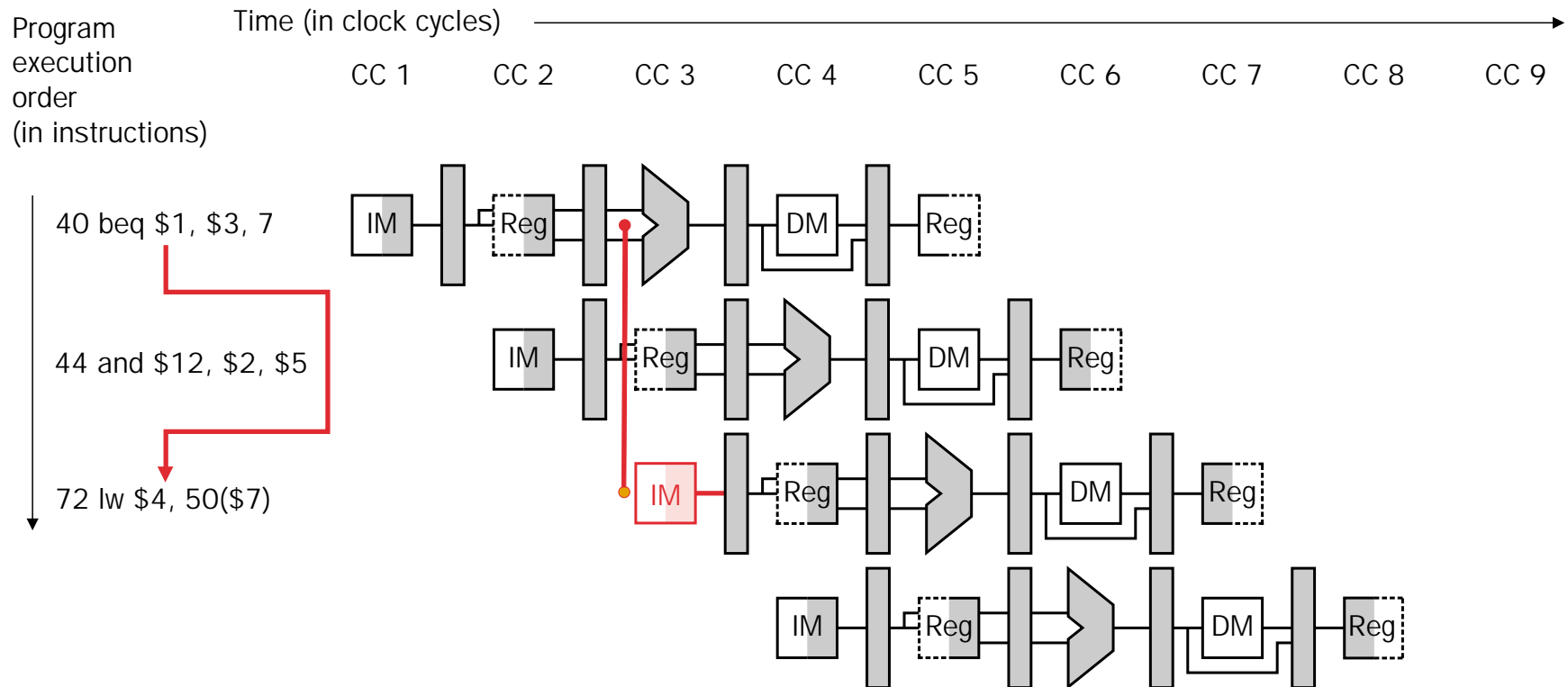
beq \$1, \$3, 7

sub \$10, ...

before<1>

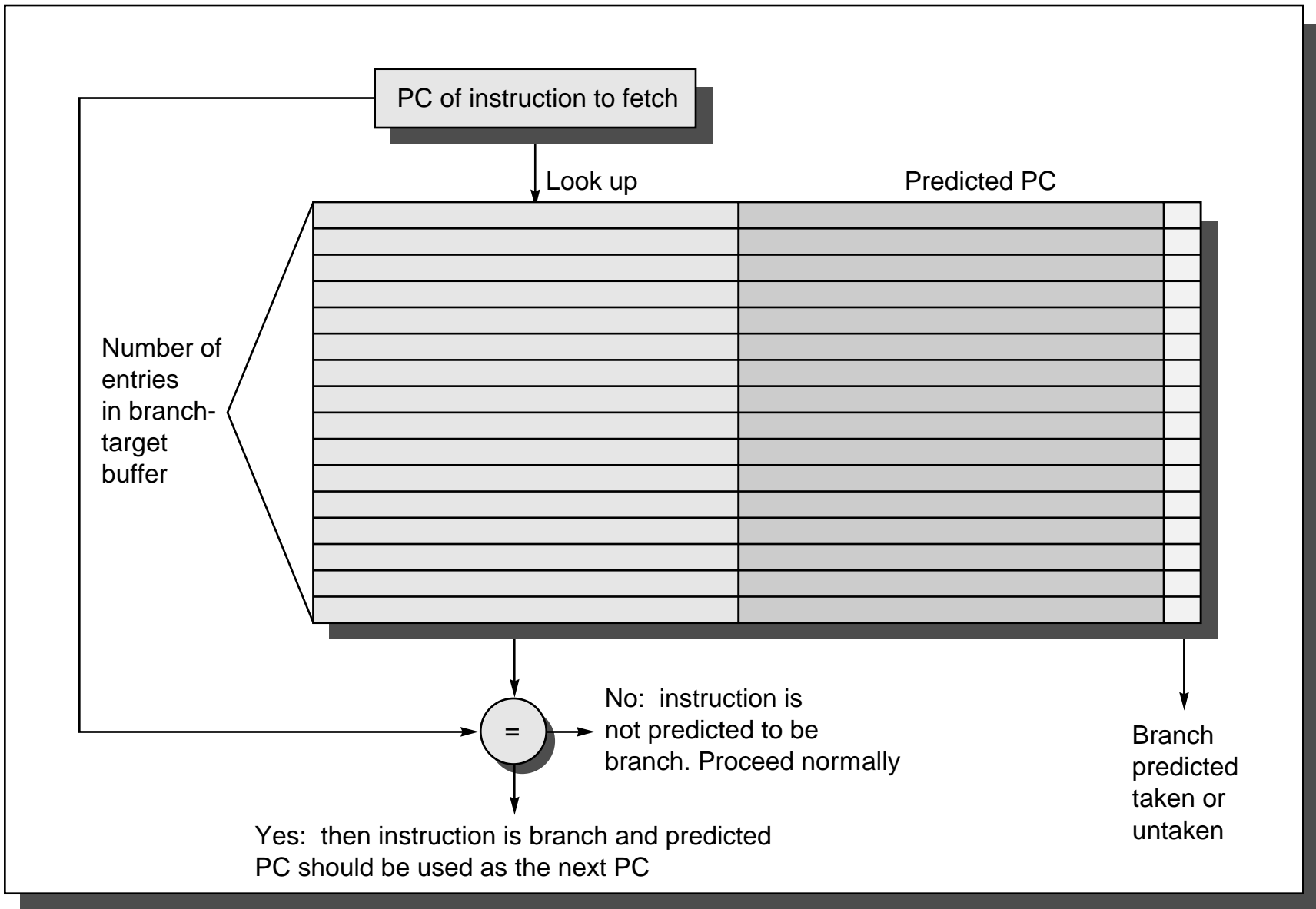


EFFECT OF AN OPTIMIZED PIPELINE ON A BRANCH INSTRUCTION



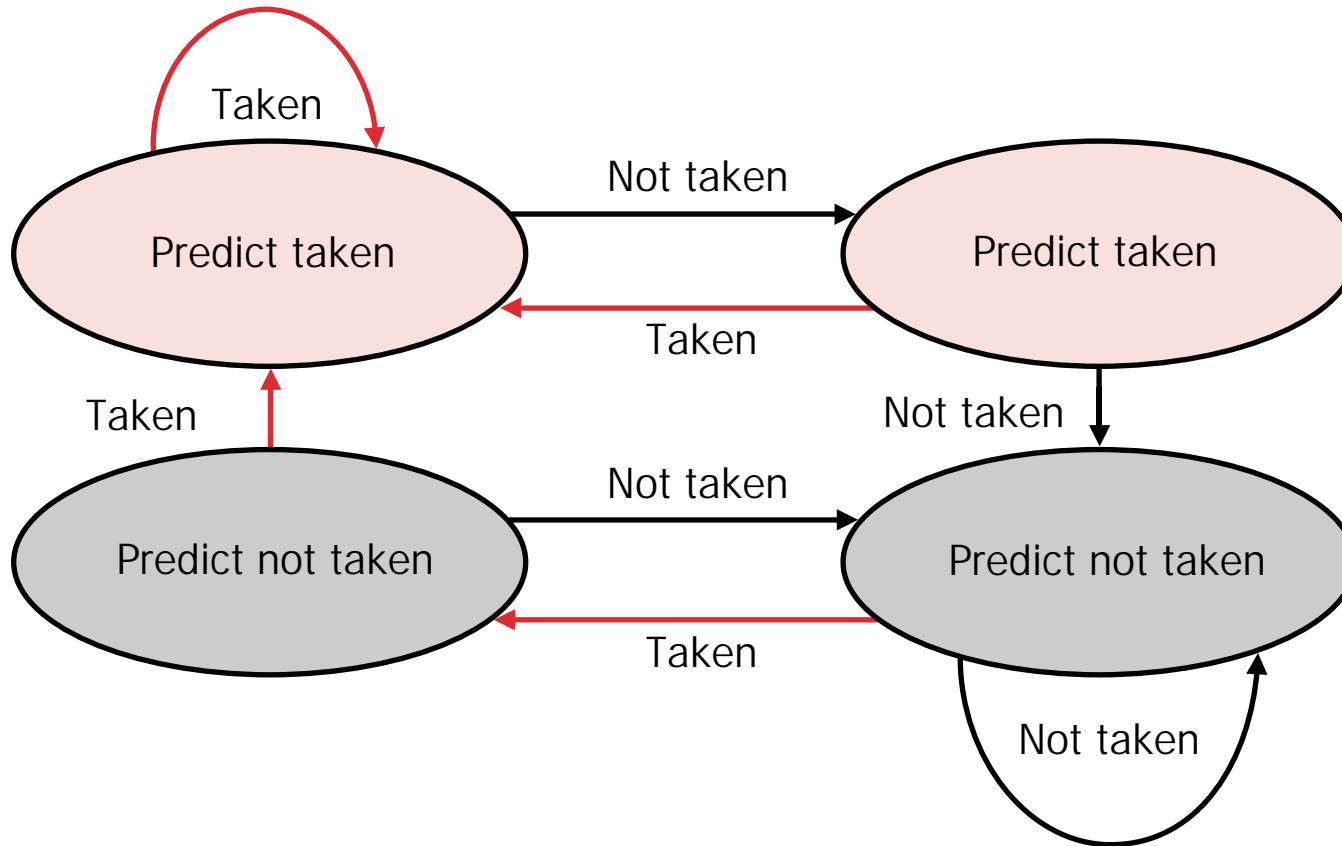
BRANCH PREDICTION SCHEMES

- 0-bit branch prediction: Assume that the branch is not taken
- A common branch prediction scheme uses a branch history table
 - ▷ Each entry in the memory is indexed by the lower 16 bits of the address of the branch instruction
 - ▷ Each entry consists of a bit that is set if the branch was recently taken
 - ▷ If the branch is not taken, the bit is toggled
 - ▷ Performance shortcoming: If a branch is almost always taken (or not taken), then the bit gets toggled on a wrong prediction, and the next branch is likely to be mispredicted
- A 2-bit branch prediction scheme uses a branch history table in which each entry contains 2 bits to indicate the state of a branch prediction FSM (next slide)
 - ▷ This scheme mispredicts only once if a branch almost always goes one way



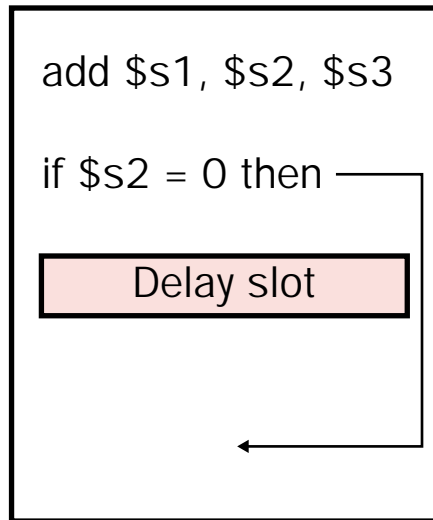
A branch-target buffer

STATES IN A 2-BIT BRANCH PREDICTION SCHEME

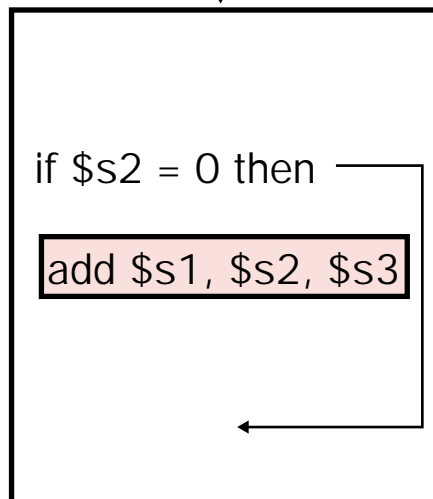


SCHEDULING THE BRANCH DELAY SLOT

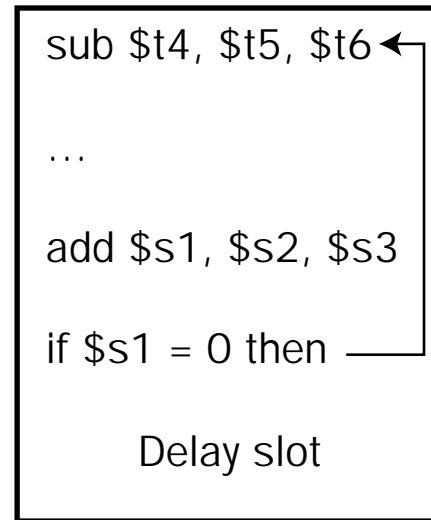
a. From before



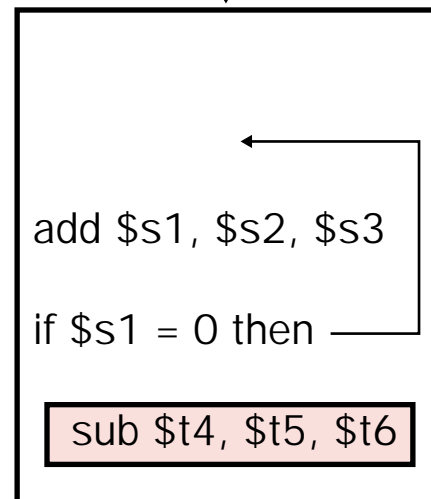
Becomes



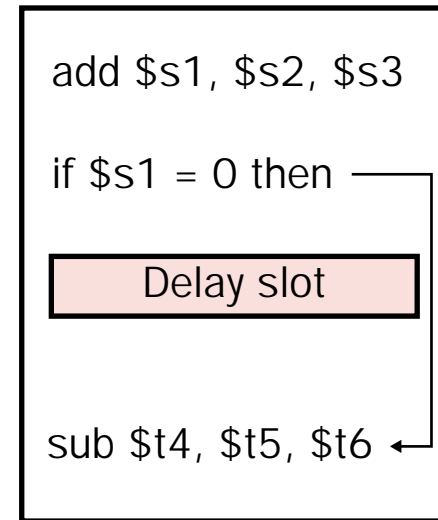
b. From target



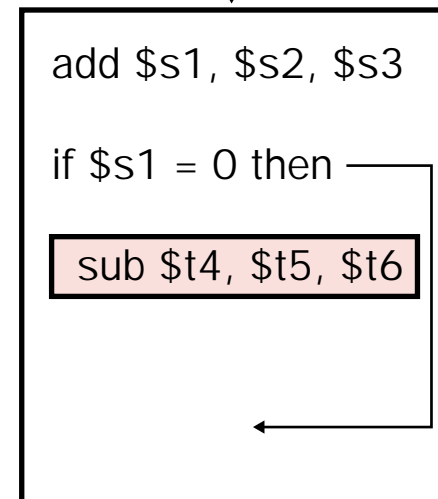
Becomes



c. From fall through



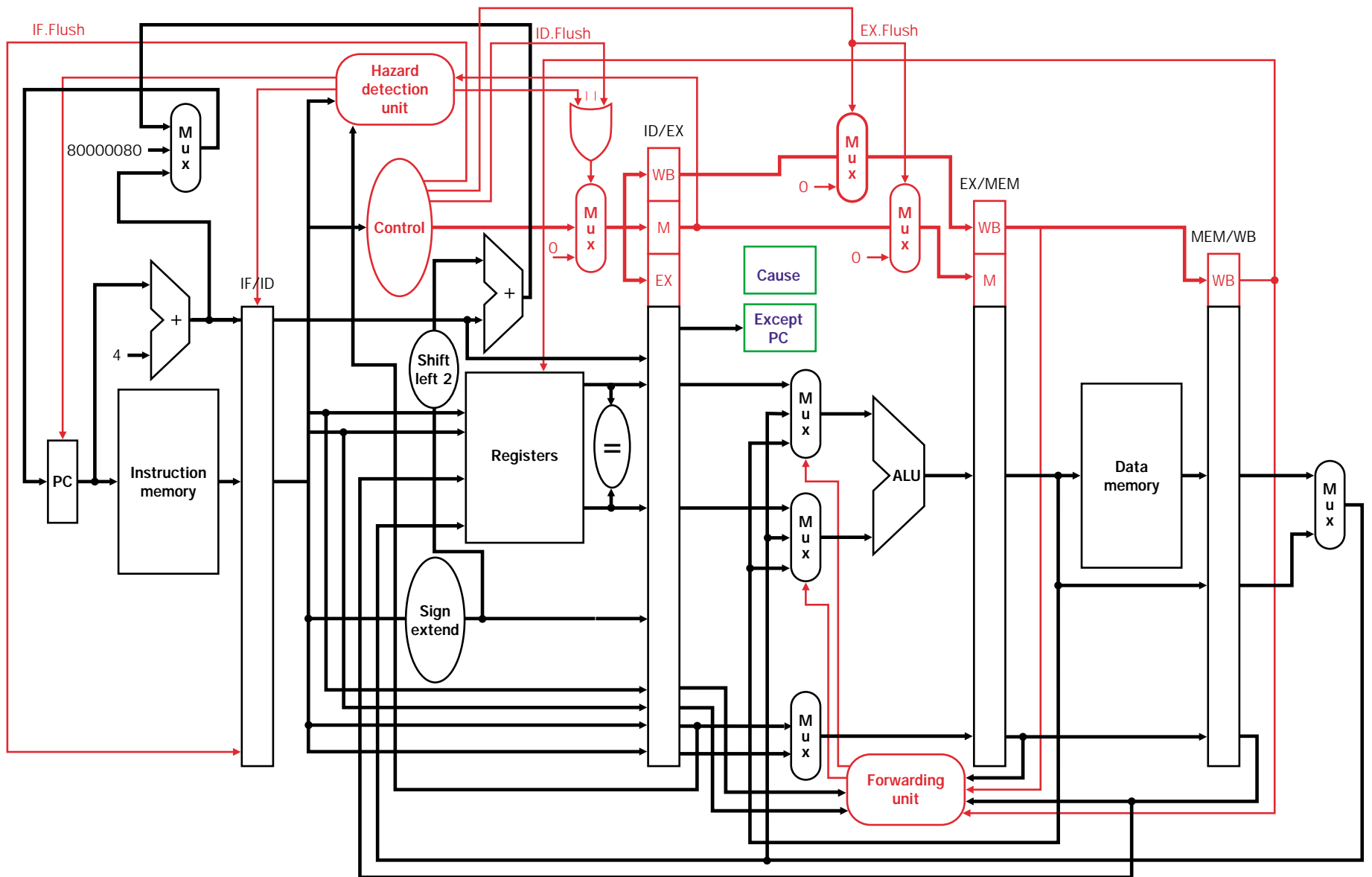
Becomes



EXCEPTIONS IN A PIPELINED PROCESSOR

- Five instructions are active in any given clock period
 - ▷ Multiple exceptions can occur simultaneously
 - ▷ If execution is not stopped soon enough, the value in the register that helped cause the exception may be overwritten in the WB stage
 - ▷ If the branch is not taken, the bit is toggled
 - ▷ To flush the instructions that follow the instruction that caused the exception, we add two new signals, ID.Flush and EX.Flush
 - ▷ ID.Flush is ORed with the stall signal from the hazard detection unit to flush an instruction during its ID stage
 - ▷ To flush an instruction in its EX stage, we add an input to the PC multiplexor that sends `0x80000080` to the PC

DATAPATH WITH CONTROLS TO HANDLE EXCEPTIONS



A PIPELINED EXCEPTION

- We'll follow what happens in the instruction sequence

```
[40000040] sub $11, $2, $4
[40000044] and $12, $2, $5
[40000048] or  $13, $2, $6
[4000004c] add $1,  $2, $1 # overflow exception occurs here
[40000050] slt $15, $6, $7
[40000054] lw  $16, 50($7)
. . .
```

given that the instructions to execute when an exception occurs are

```
[80000080] lui $1, -28672
[80000084] sw  $2, 592($1) # sw $v0 s1
. . .
```

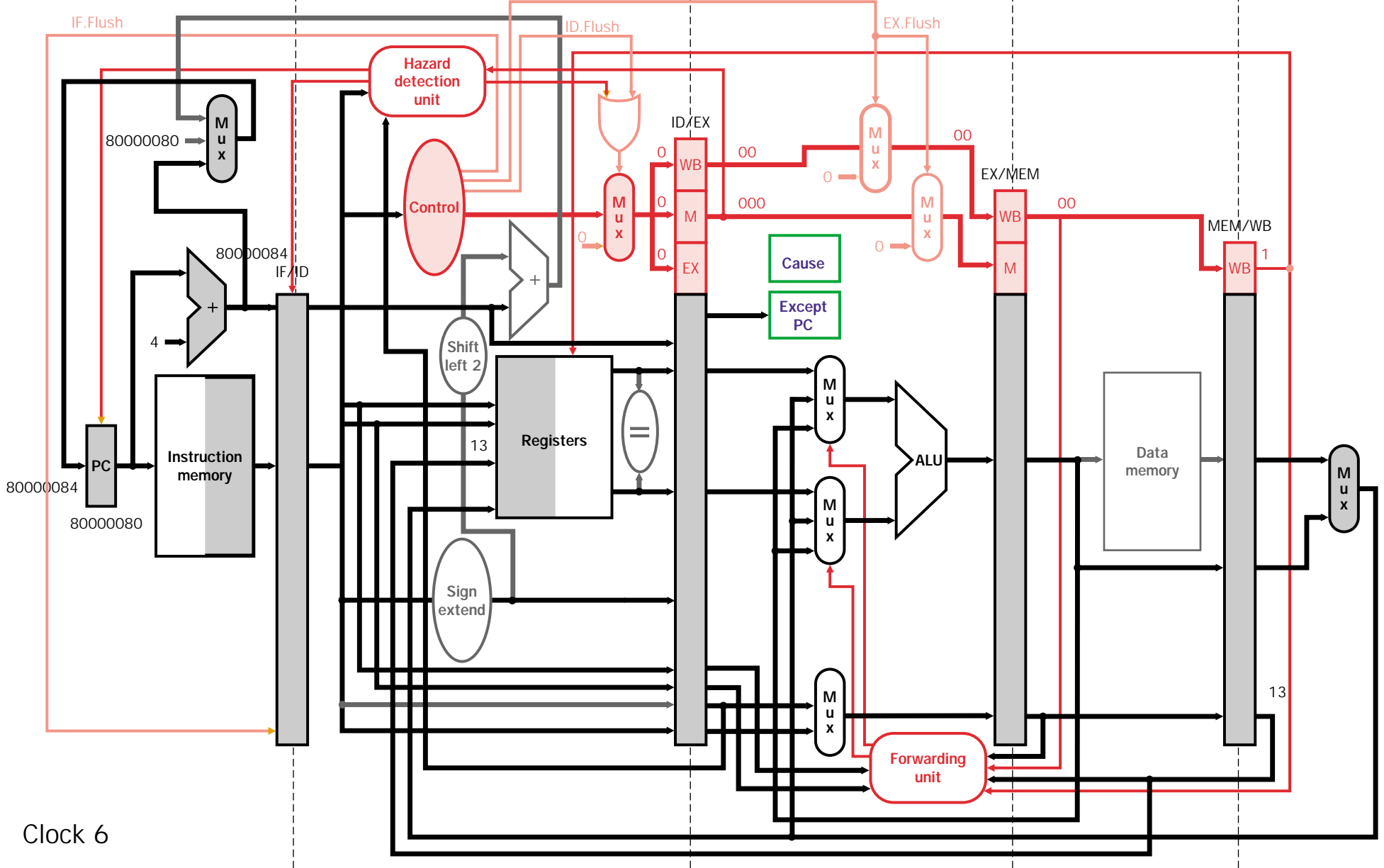

lui \$1, -28672

bubble (nop)

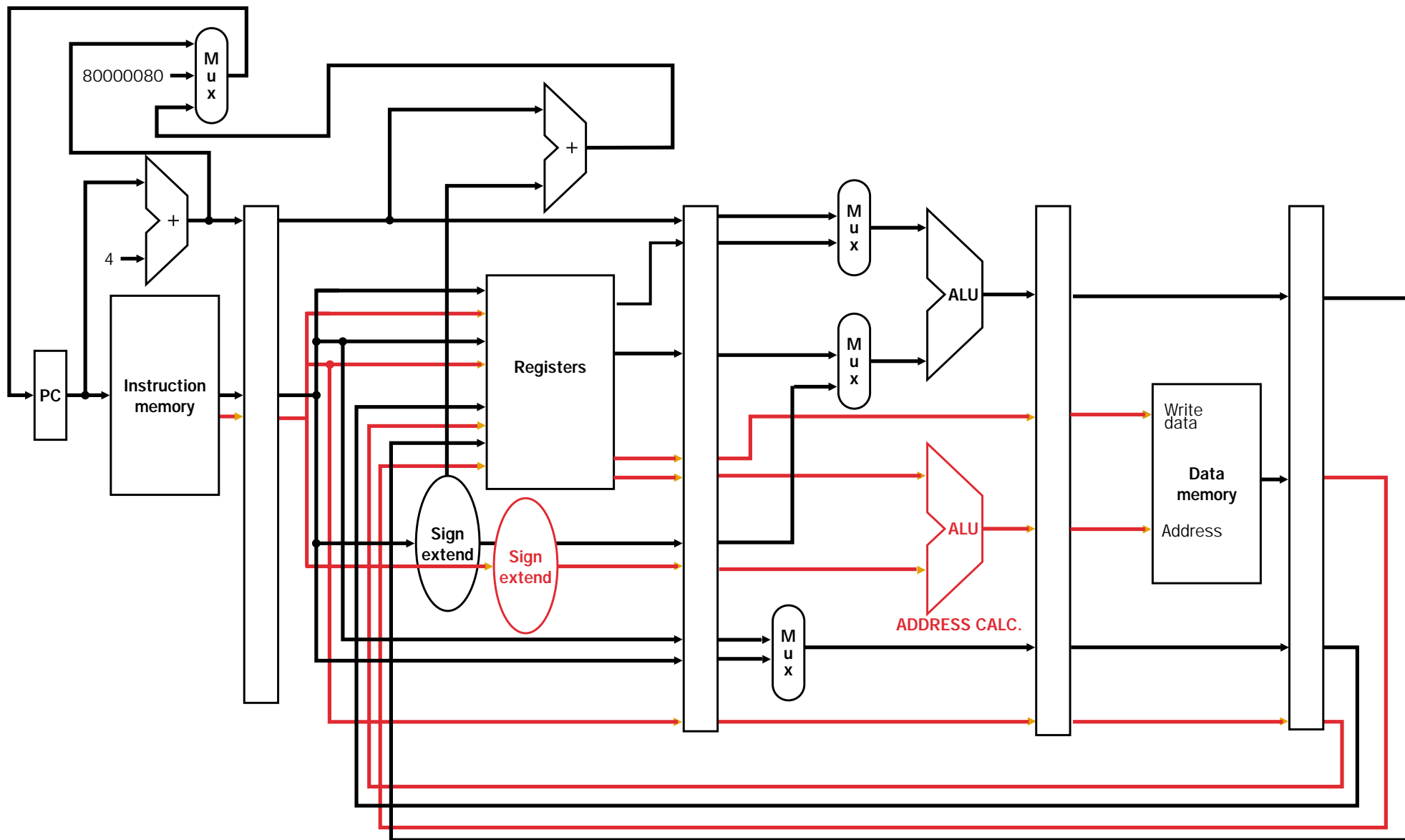
bubble

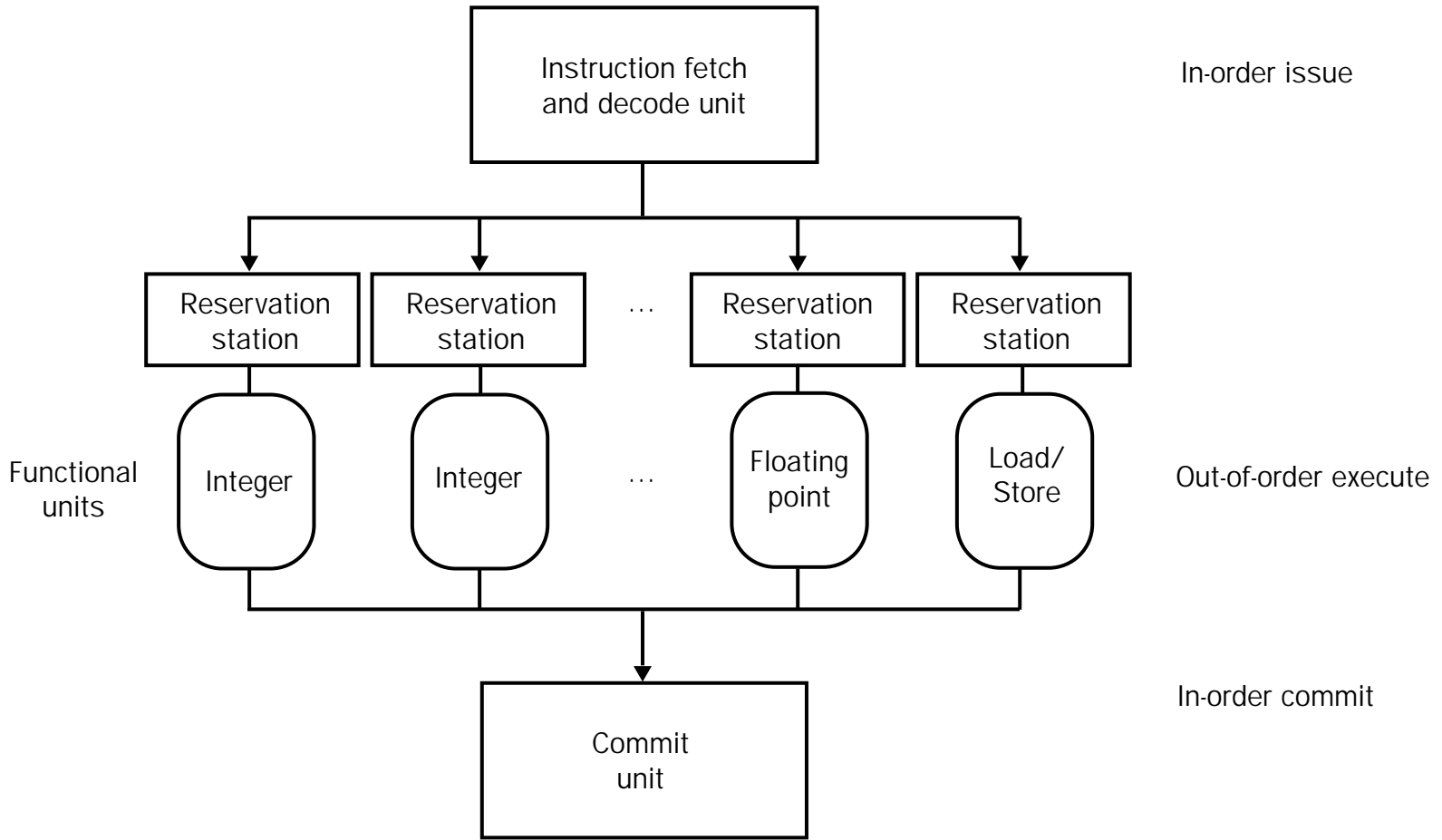
bubble

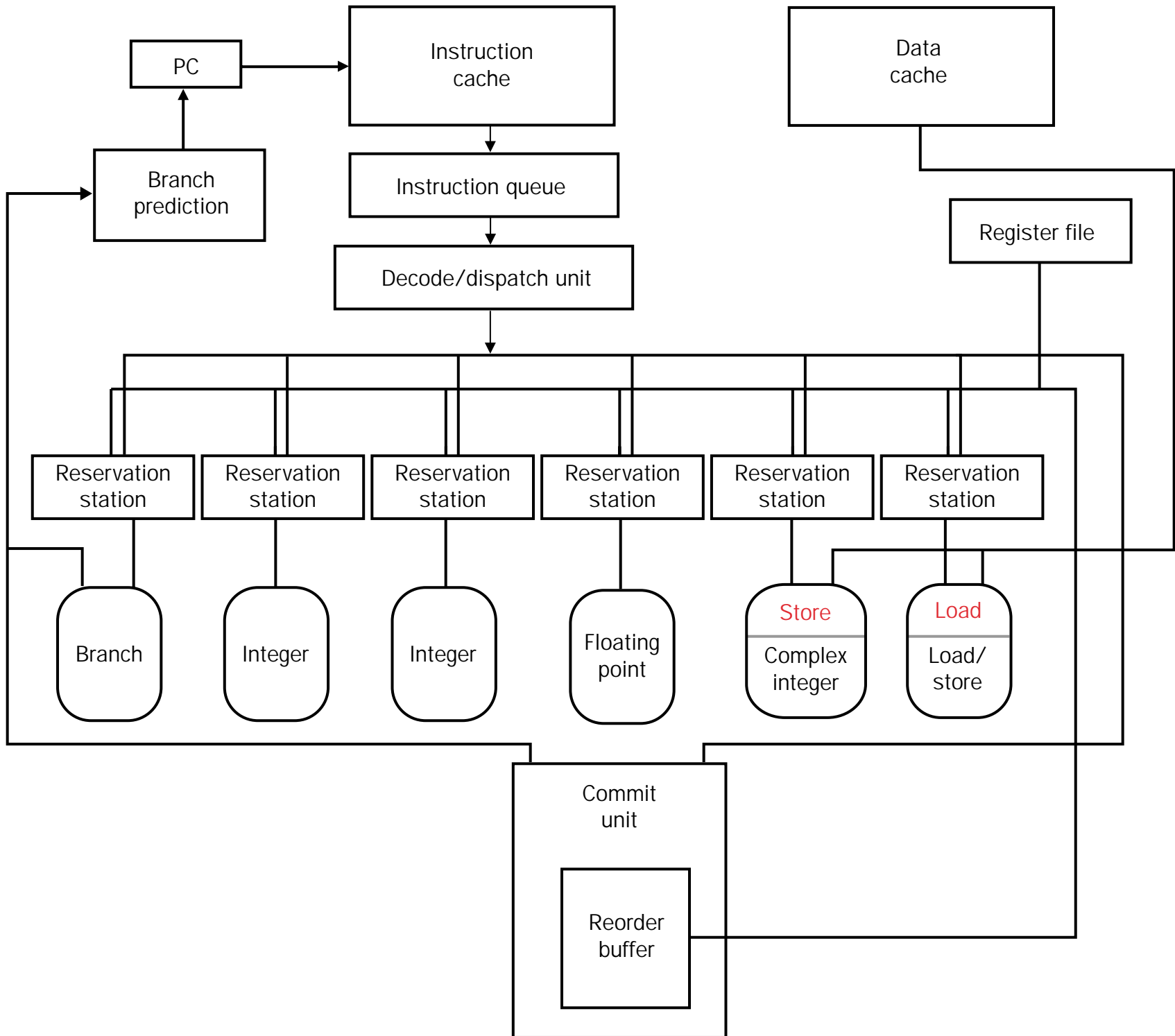
or \$13, ...



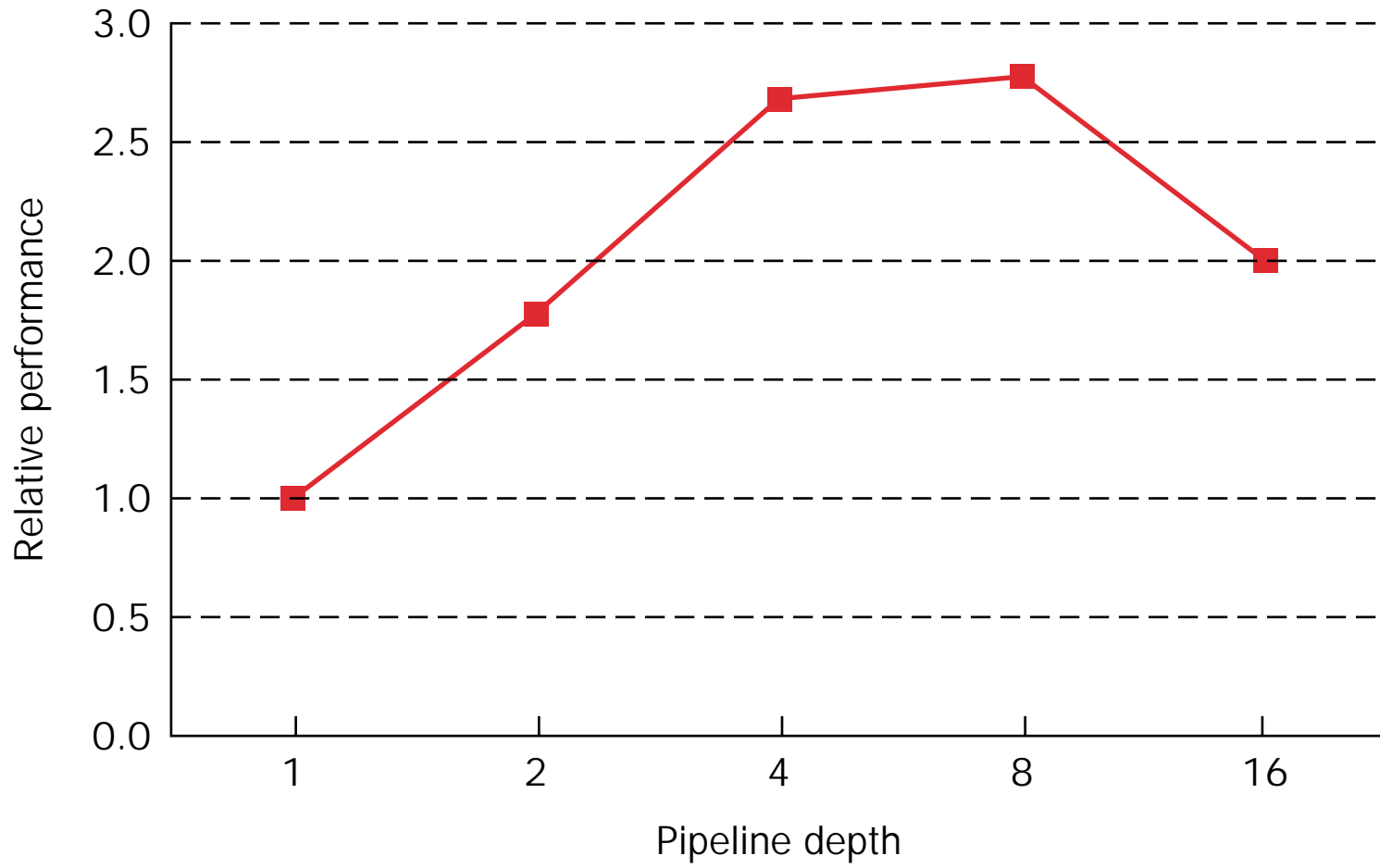
SUPERSCALAR DATAPATH







PIPELINE DEPTH vs. SPEEDUP



PIPELINED DATAPATH WITH CONTROL

