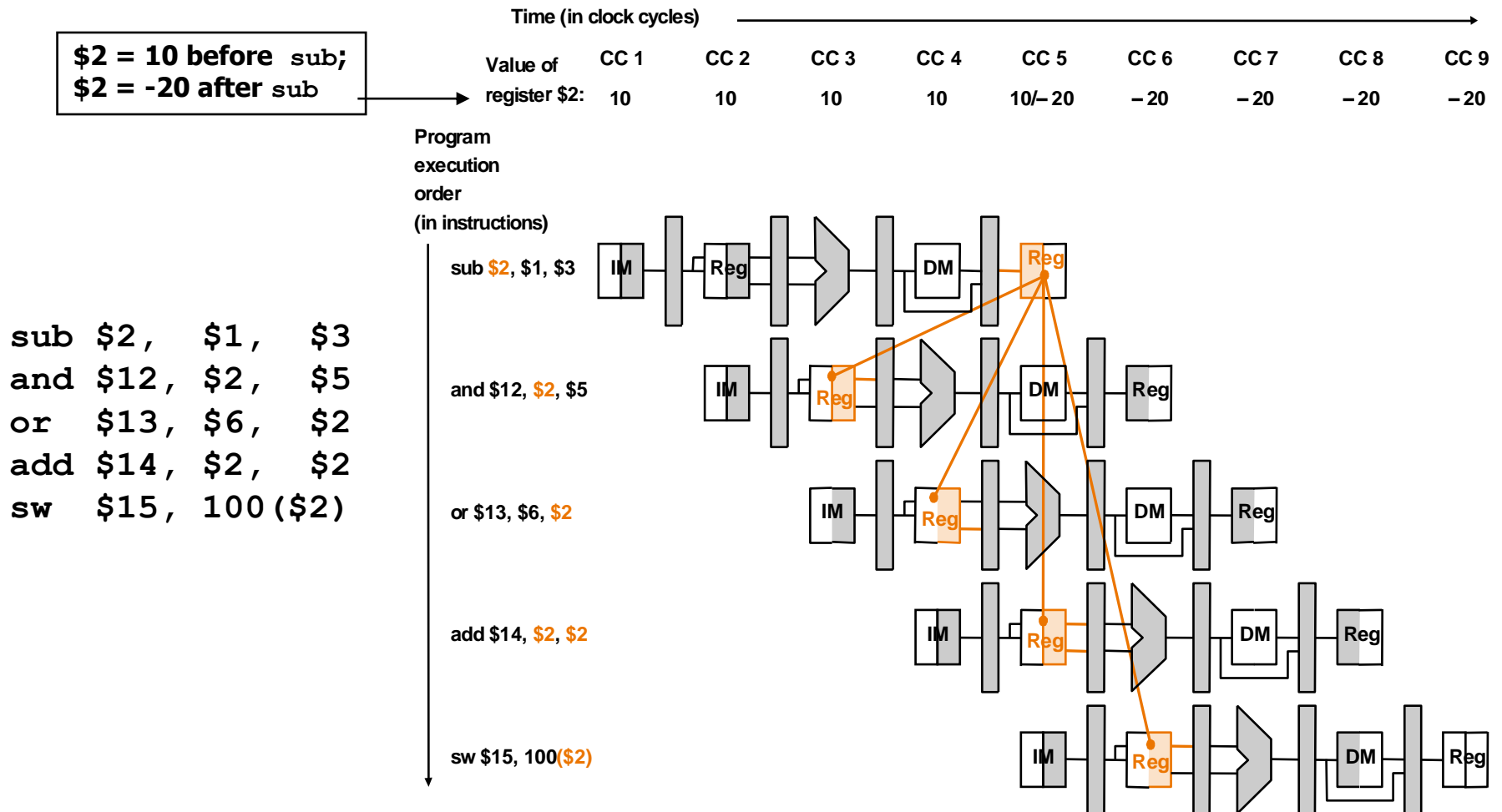


Revisiting Hazards

- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware*...

Data Hazards and Forwarding

- Problem with starting an instruction before previous are finished:
 - data dependencies that go backward in time – called *data hazards*



Software Solution

- Have compiler guarantee *never* any data hazards!
 - by *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,
 - or, if such rearrangement is not possible, *insert nops*

sub	\$2, \$1, \$3		sub	\$2, \$1, \$3
lw	\$10, 40(\$3)		nop	
slt	\$5, \$6, \$7		nop	
and	\$12, \$2, \$5	or	and	\$12, \$2, \$5
or	\$13, \$6, \$2		or	\$13, \$6, \$2
add	\$14, \$2, \$2		add	\$14, \$2, \$2
sw	\$15, 100(\$2)		sw	\$15, 100(\$2)

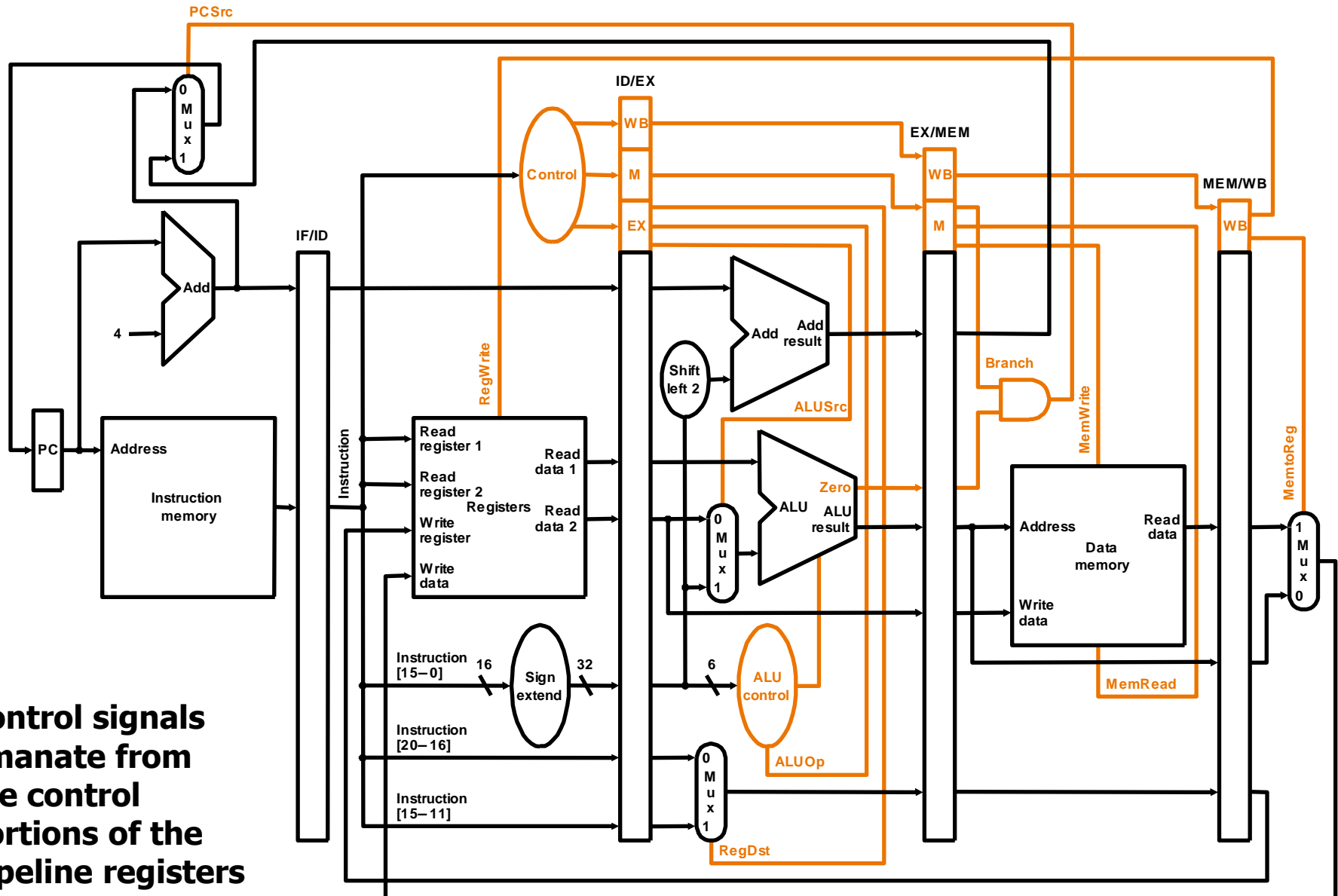
- Such compiler solutions may not always be possible, and nops slow the machine down

MIPS: nop = "no operation" = 00...0 (32bits) = sll \$0, \$0, 0

Hardware Solution: Forwarding

- Idea: *use intermediate data*, do not wait for result to be finally written to the destination register. Two steps:
 1. *Detect* data hazard
 2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)



**Control signals
emanate from
the control
portions of the
pipeline registers**

Hazard Detection

- Hazard conditions:

- 1a. $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs}$

- 1b. $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRt}$

- 2a. $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRs}$

- 2b. $\text{MEM/WB.RegisterRd} = \text{ID/EX.RegisterRt}$

- Eg., 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

- $\text{EX/MEM.RegisterRd} = \text{ID/EX.RegisterRs} = \2 (1a)

- Whether to forward also depends on:

- *if the later instruction is going to write a register – if not, 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

Plan:

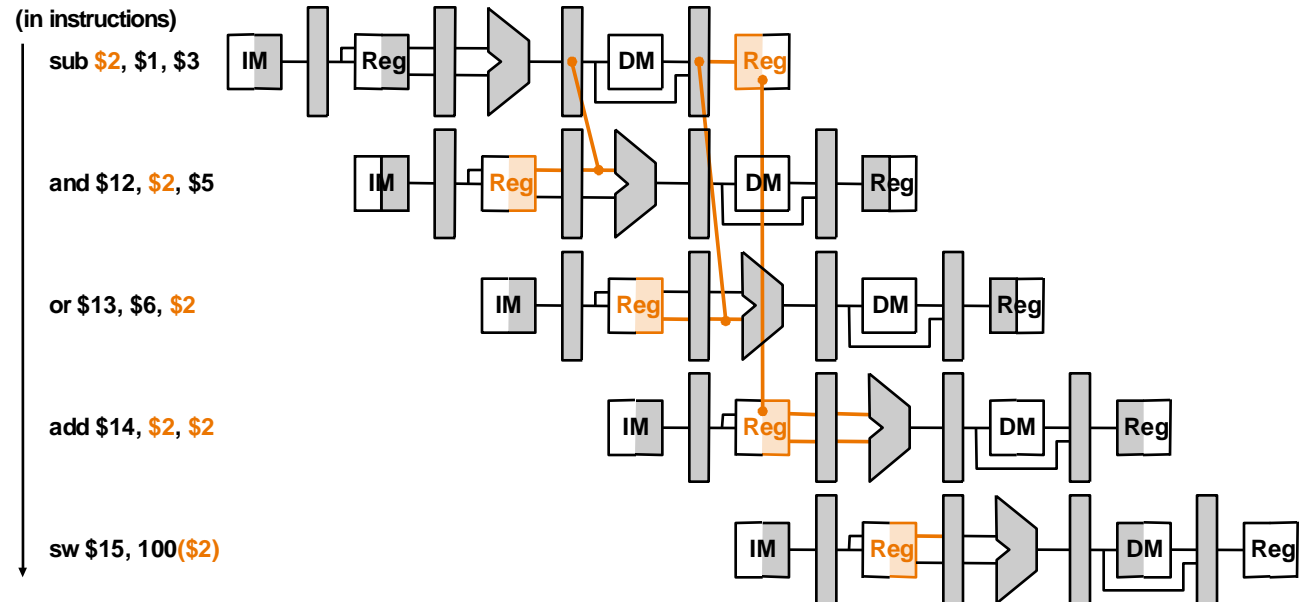
- allow inputs to the ALU not just from ID/EX, but also later pipeline registers, and
- use multiplexors and control signals to choose appropriate inputs to ALU

	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)
  
```

Program
execution order
(in instructions)

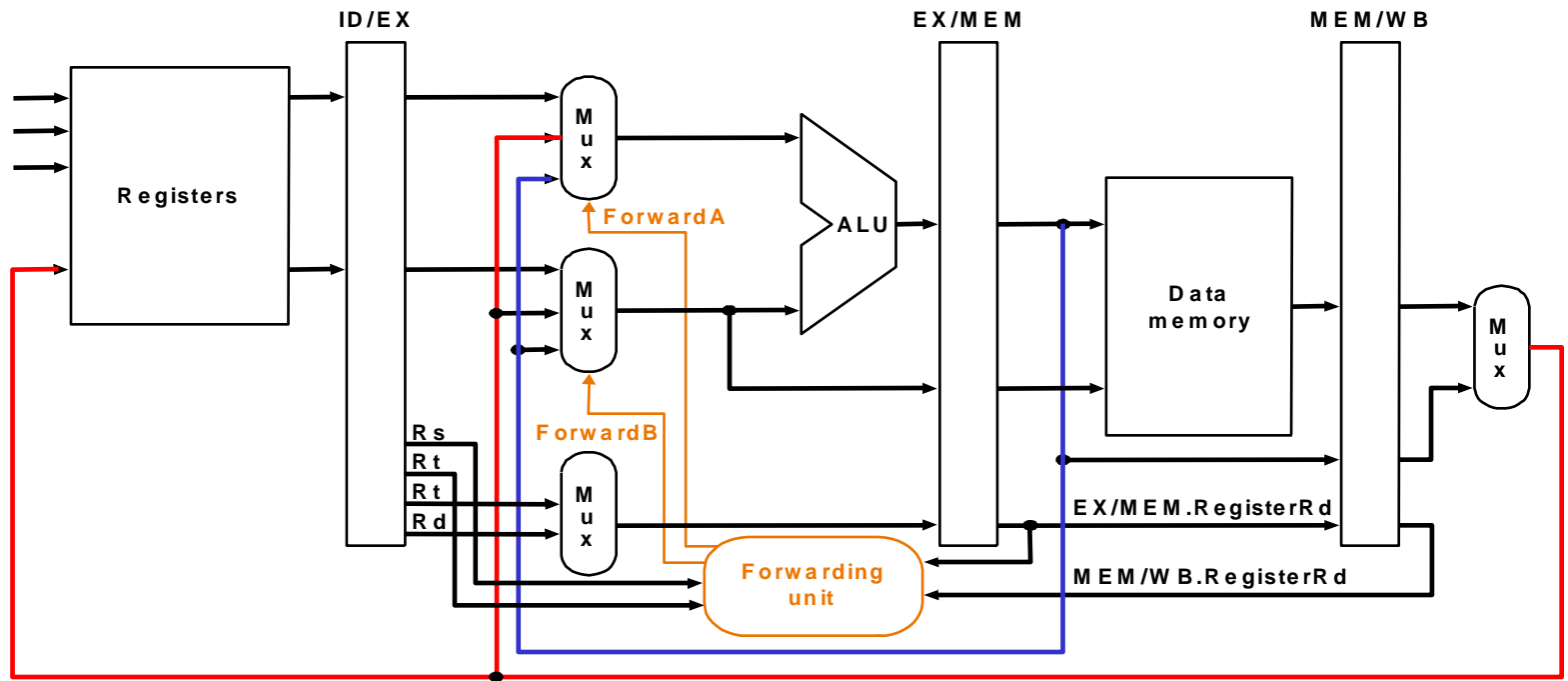


Dependencies between pipelines move forward in time

```

graph LR
    PC[PC] --> Mux1((Mux))
    Mux1 --> ID_EX[ID/EX]
    ID_EX --> EX_MEM[EX/MEM]
    EX_MEM --> MEM_WB[MEM/WB]
    MEM_WB --> Mux2((Mux))
    Mux2 --> PC
    ID_EX --> ALU[ALU]
    EX_MEM --> ALU
    ALU --> EX_MEM
    EX_MEM --> DM[Data memory]
    DM --> MEM_WB
    MEM_WB --> ALU
    MEM_WB --> Mux2
    
```

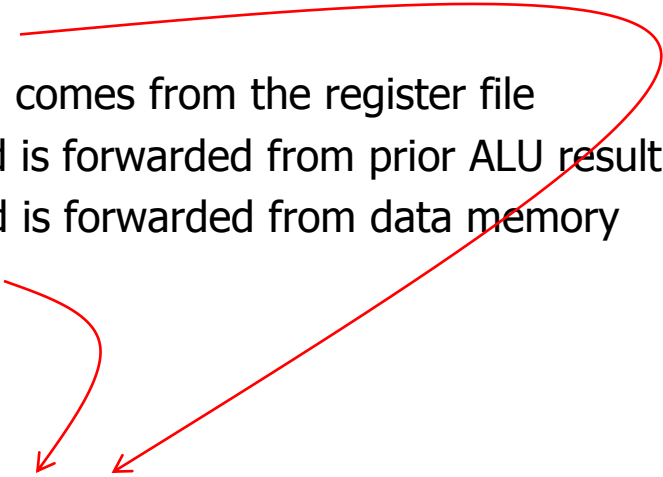
a. No forwarding **Datapath before adding forwarding hardware**



b. With forwarding 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  $\neq$  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  $\neq$  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  $\neq$  0 ) // to a non-$0 register...
    and ( EX/MEM.RegisterRd  $\neq$  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...
```

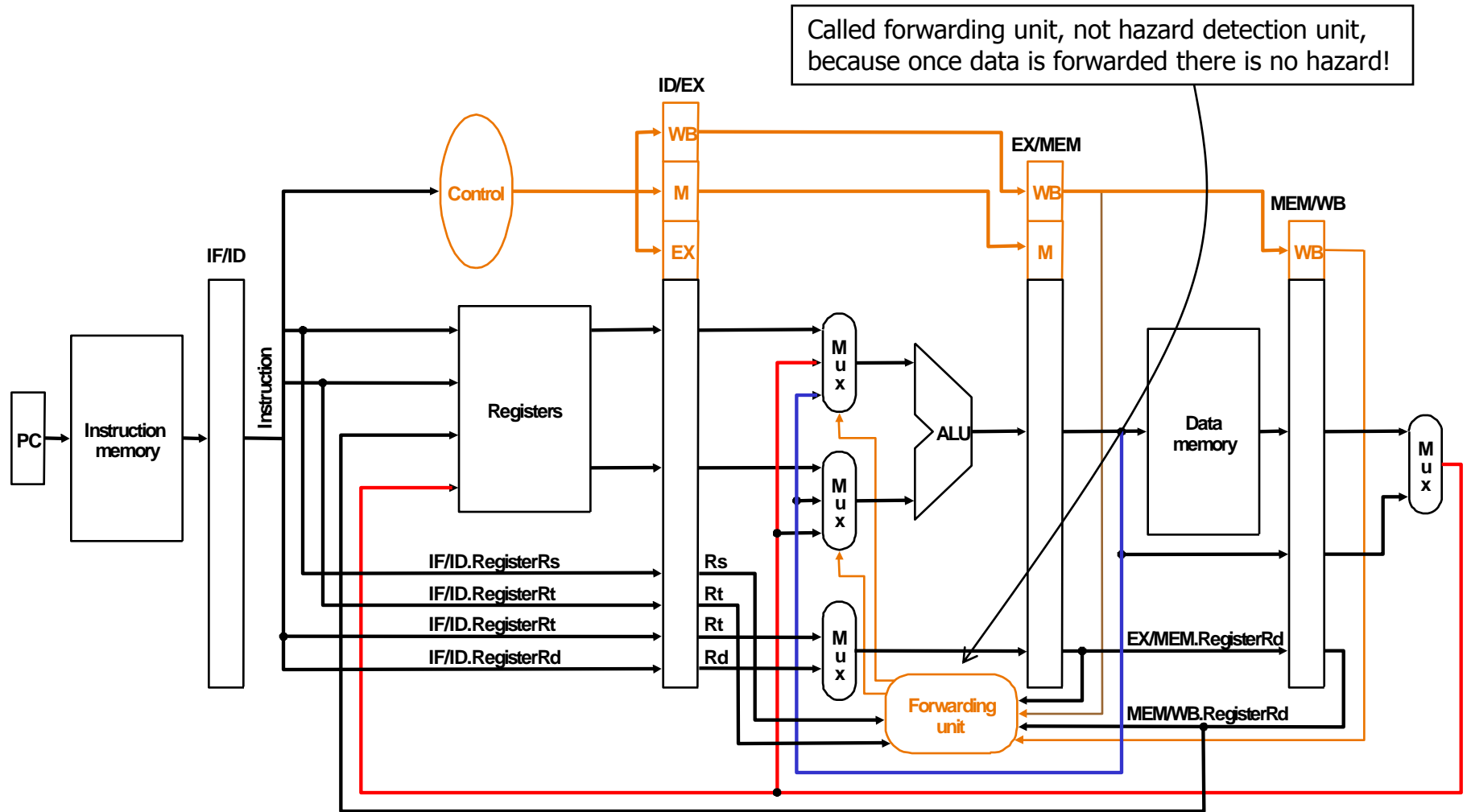
ForwardA = 01

```
if (      MEM/WB.RegWrite           // if there is a write...
    and ( MEM/WB.RegisterRd  $\neq$  0 ) // to a non-$0 register...
    and ( EX/MEM.RegisterRd  $\neq$  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; (array summing?), where an earlier pipeline (EX/MEM) register has more recent data

Forwarding Hardware with Control



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

Note: so far we have only handled forwarding to R-type instructions...!

Forwarding

- Execution example:

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

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

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

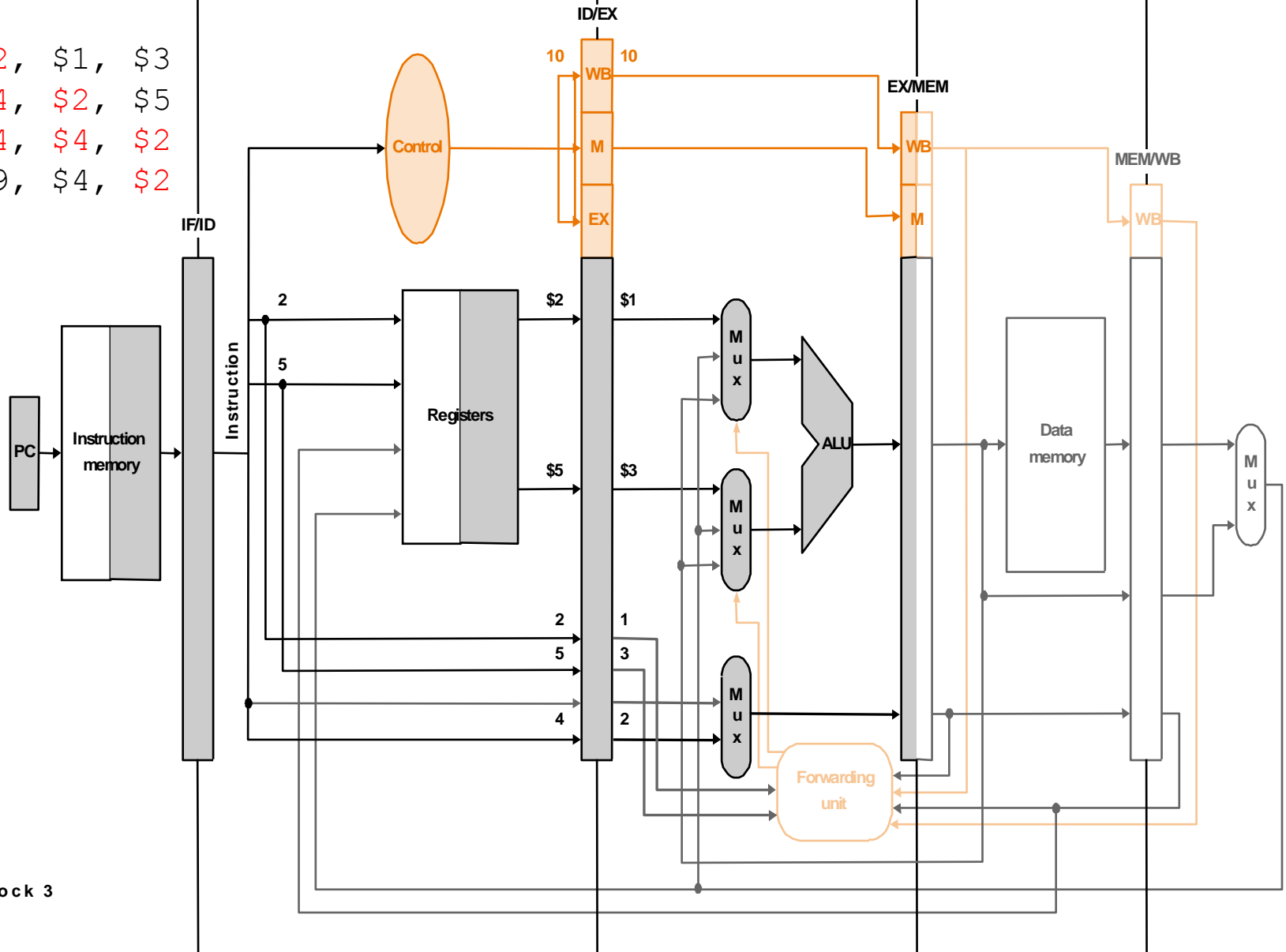
sub \$2, \$1, \$3

before <1>


before <2>

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

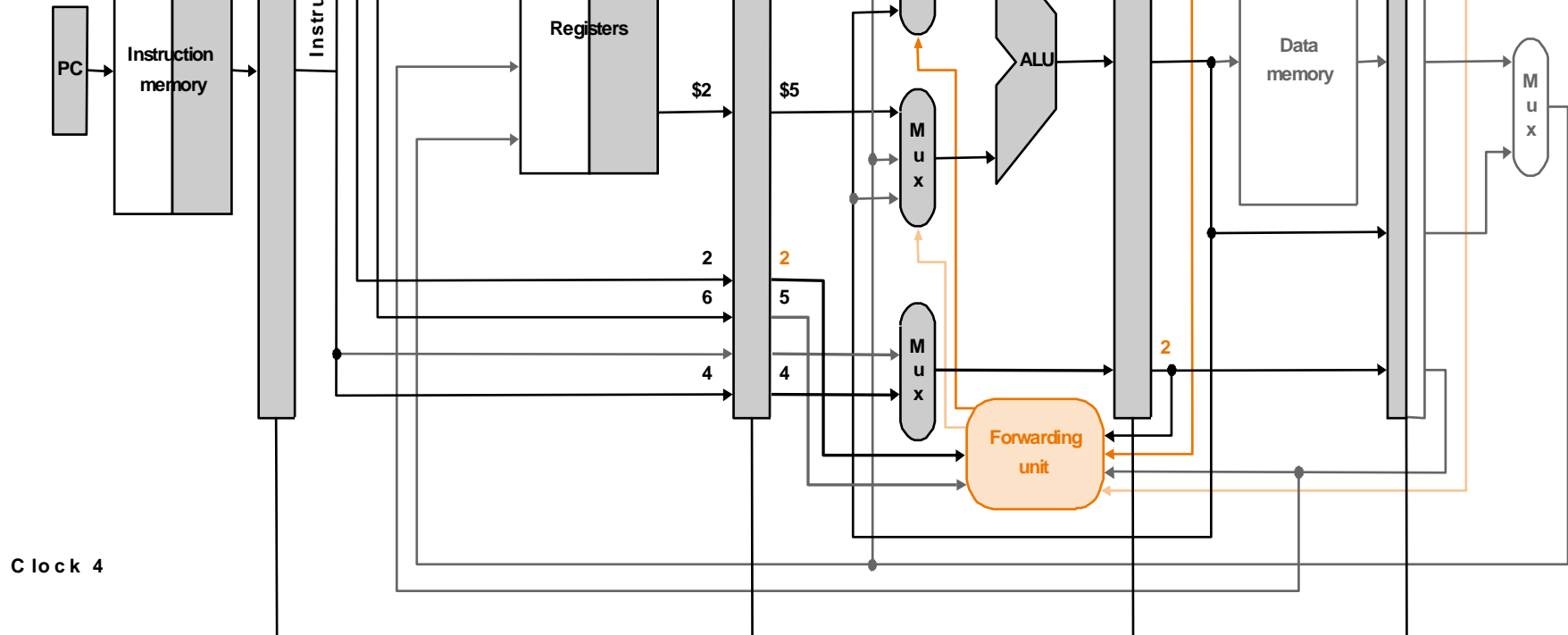
Clock 3



before < 1 >

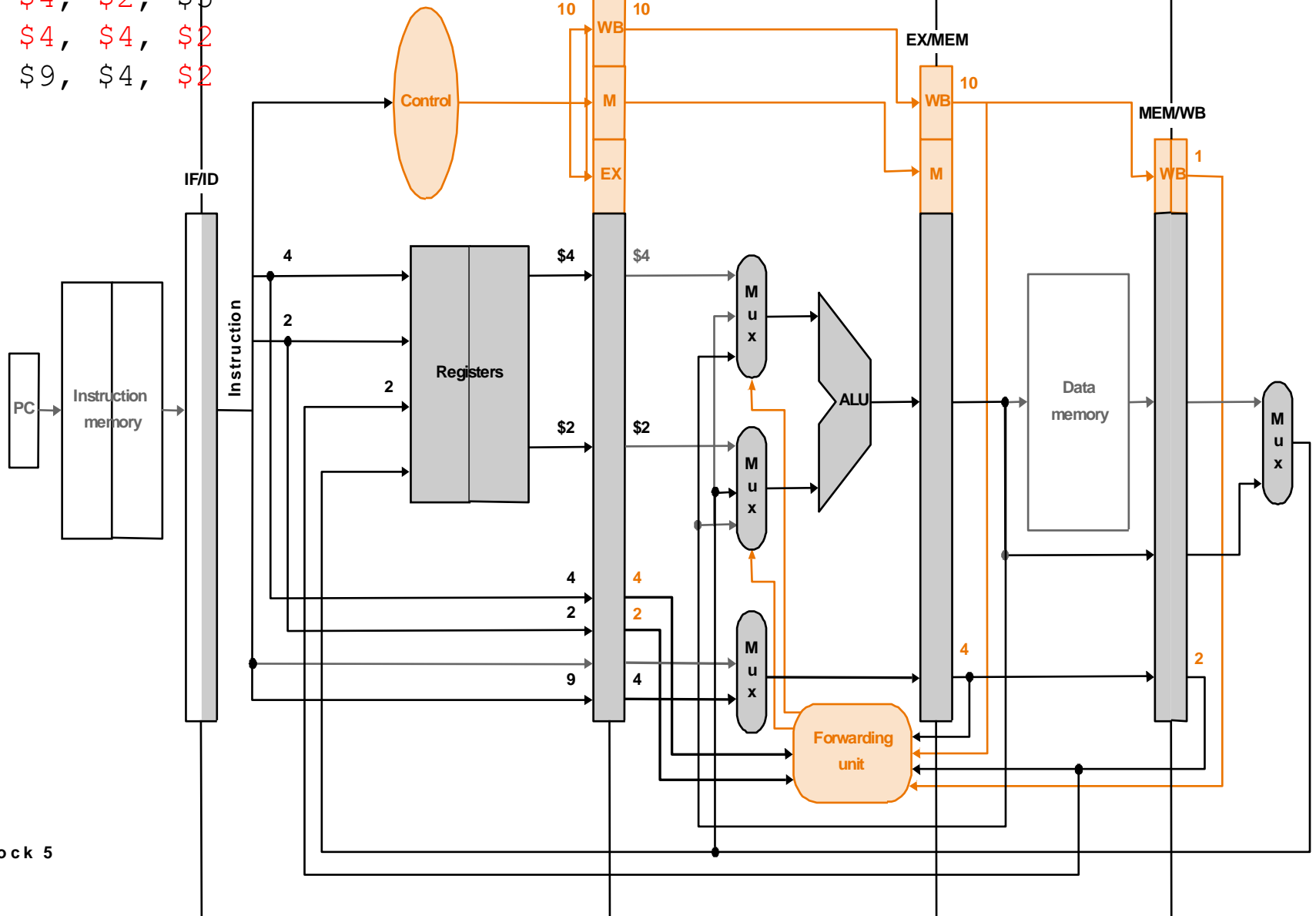


A diagram of a control block, represented as a vertical oval with an orange border and light orange fill. The word "Control" is written in orange text in the center. A black arrow points into the left side of the oval, and another black arrow points out of the right side.



sub \$2, . . .

ID/EX



after r<2>

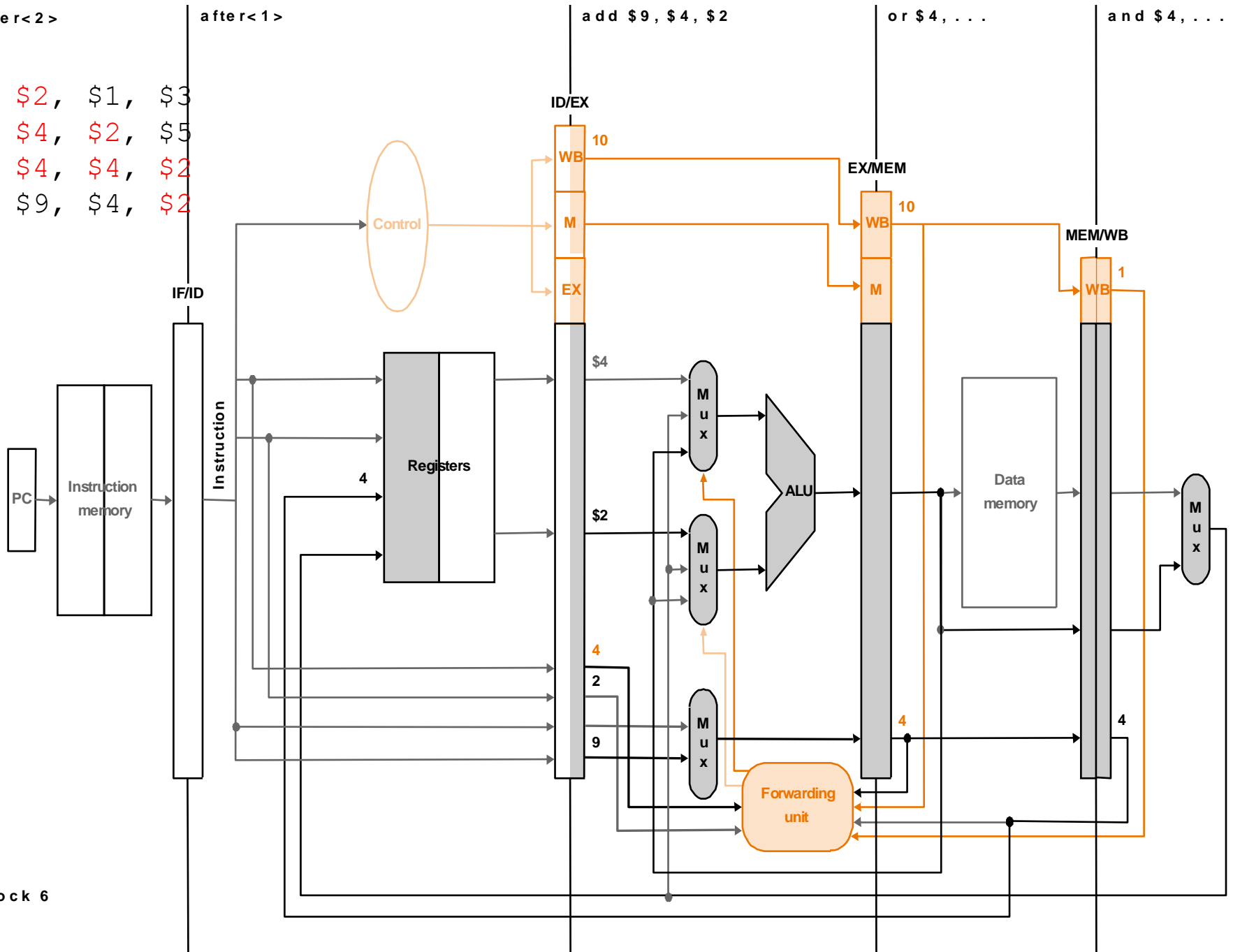
after r<1>

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

or \$4, ...

and \$4, ...

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



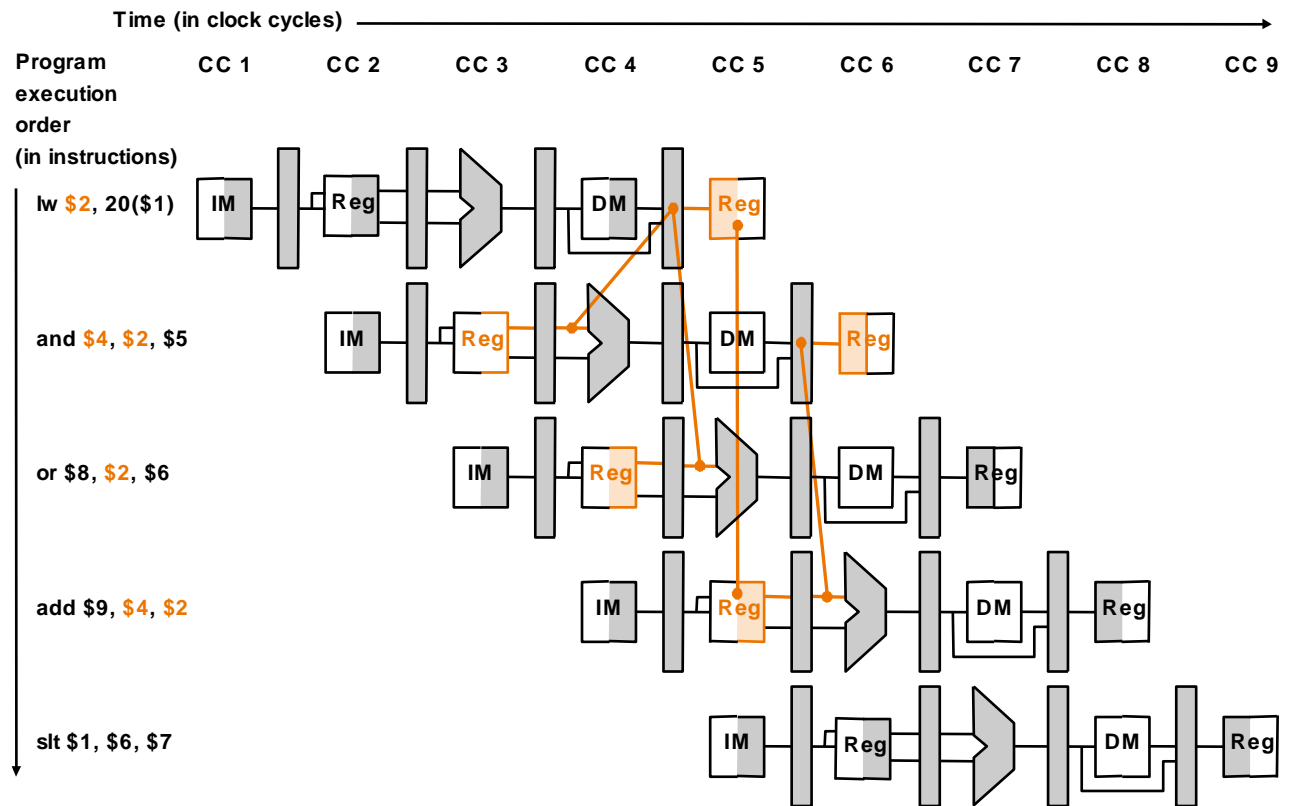
Clock 6

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

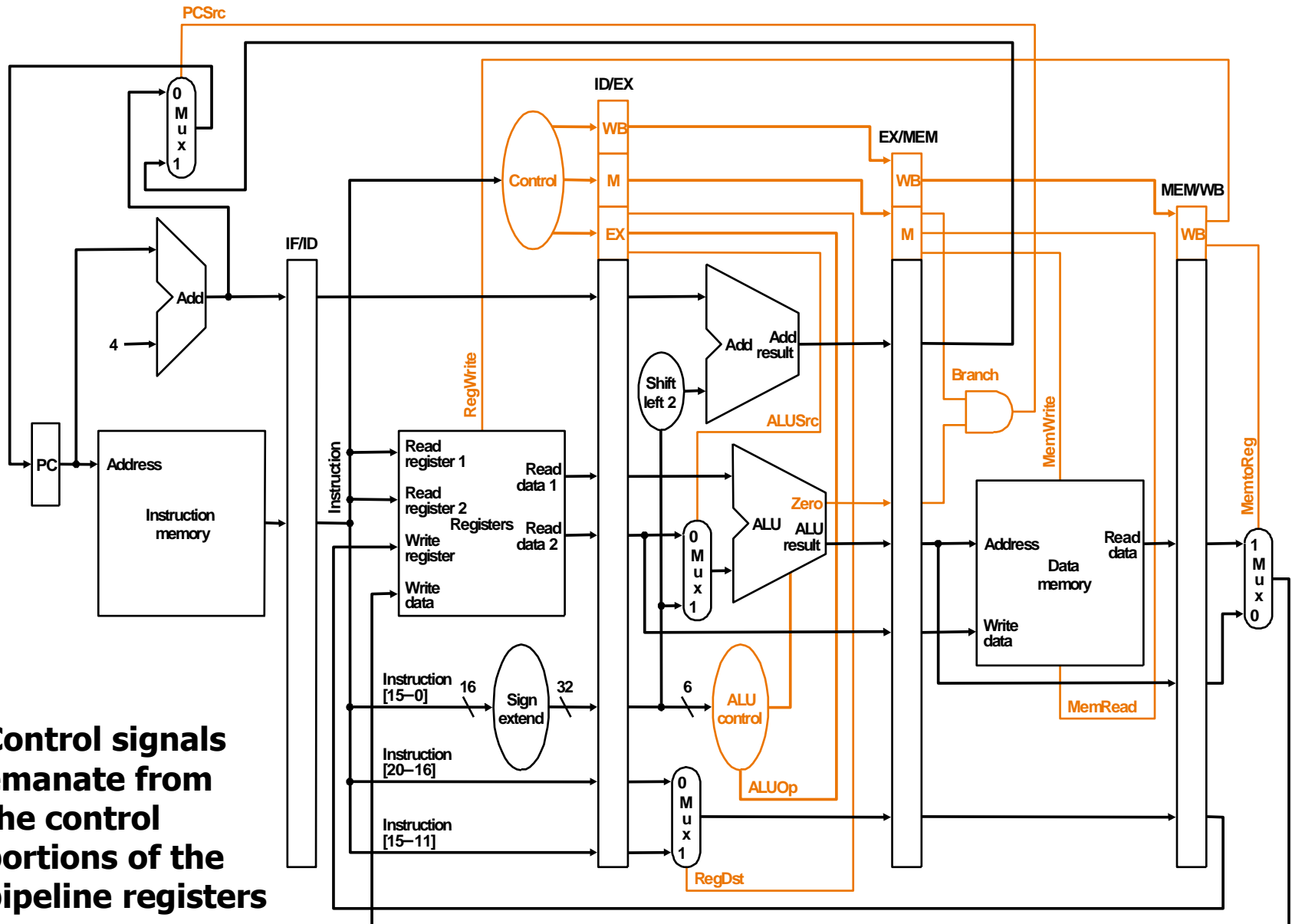
```
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
```

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



- therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection Logic to Stall

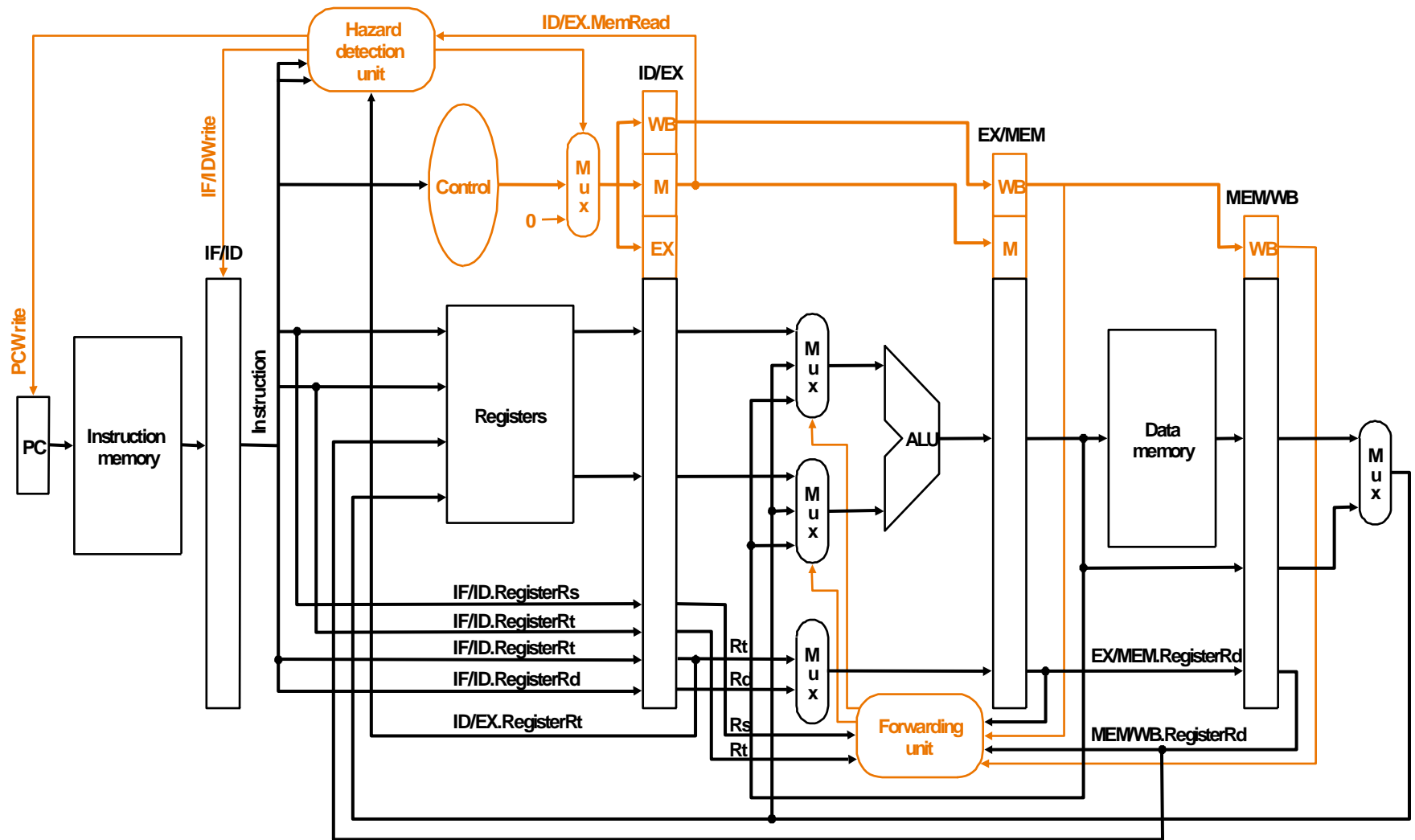
- Hazard detection unit implements the following check if to stall

```
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
- What the hardware does to stall 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
 - note that we cannot turn that instruction into an `nop` by 0ing all the bits in the instruction itself – recall `nop` = 00...0 (32 bits) – because it has already been decoded and control signals generated

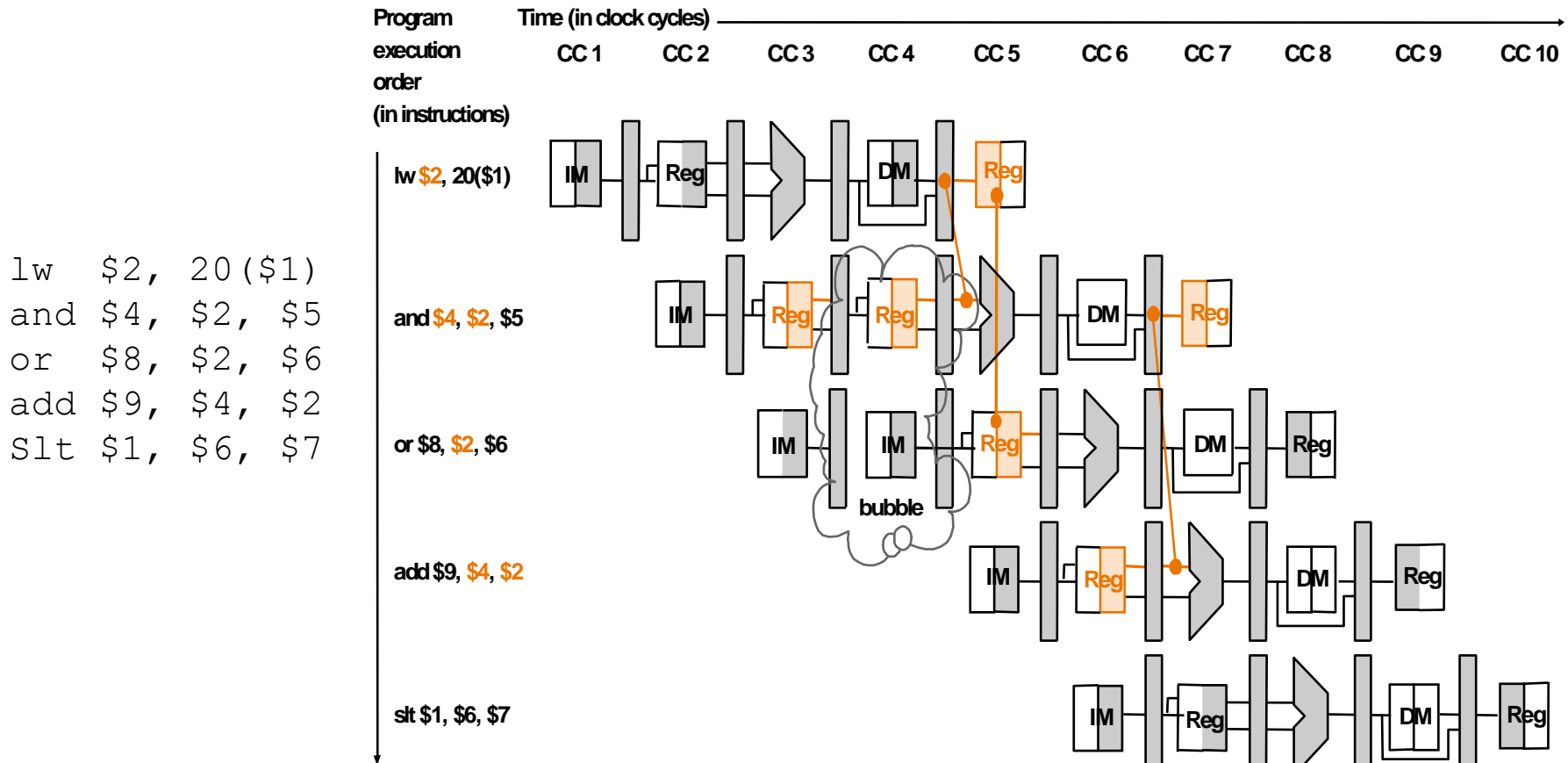
Hazard Detection Unit



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

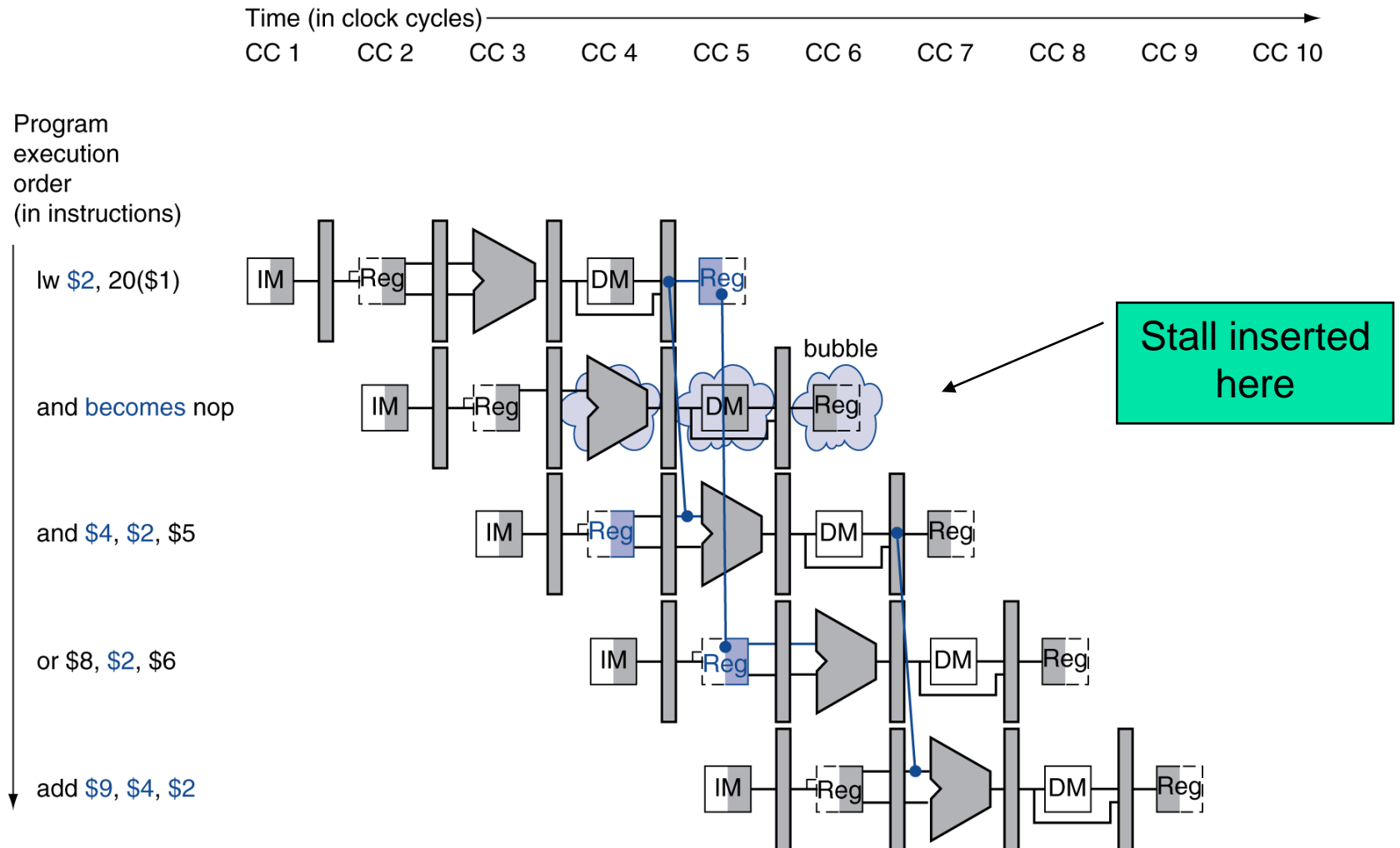
Stalling Resolves a Hazard

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

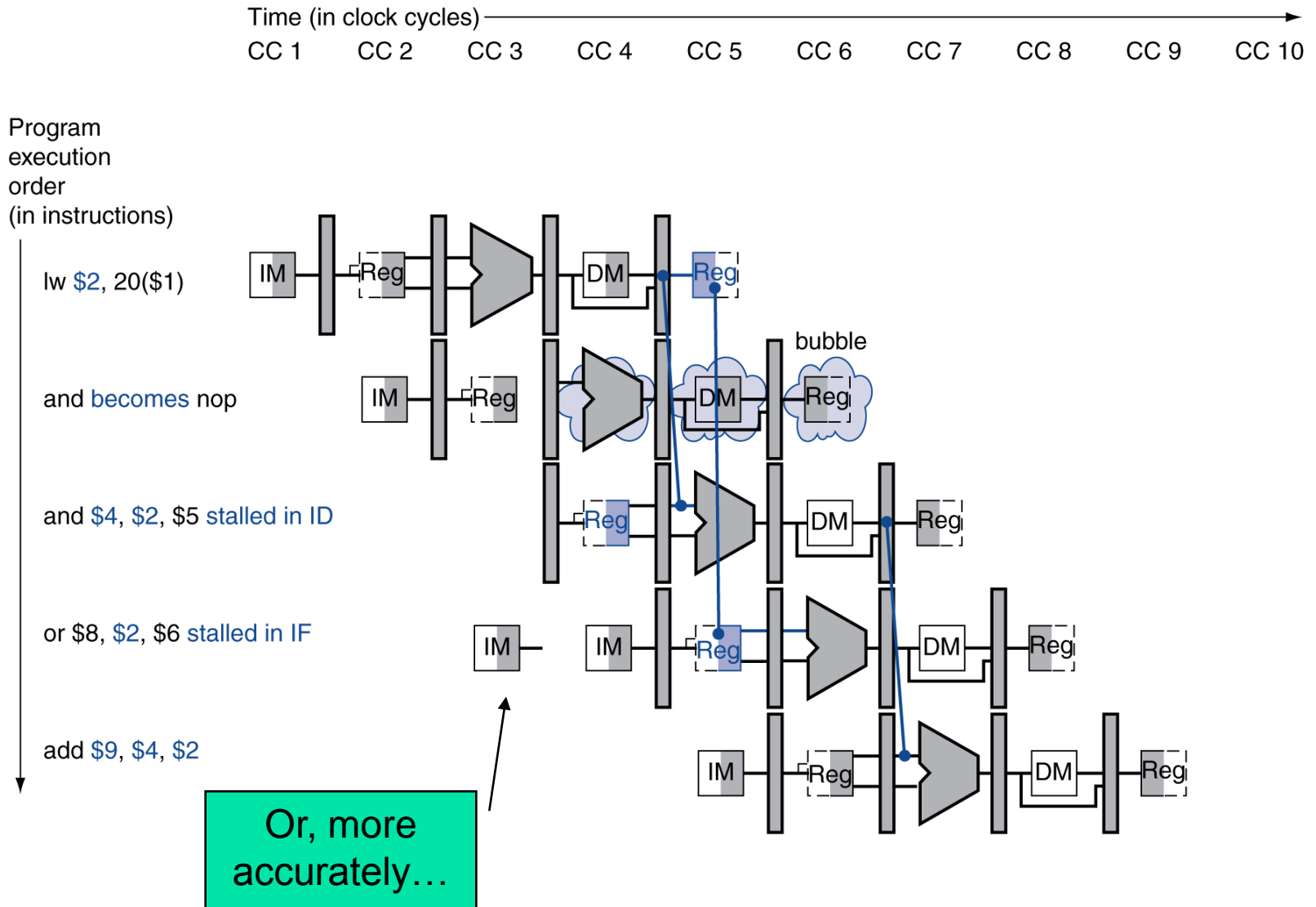


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

Stall/Bubble in the Pipeline



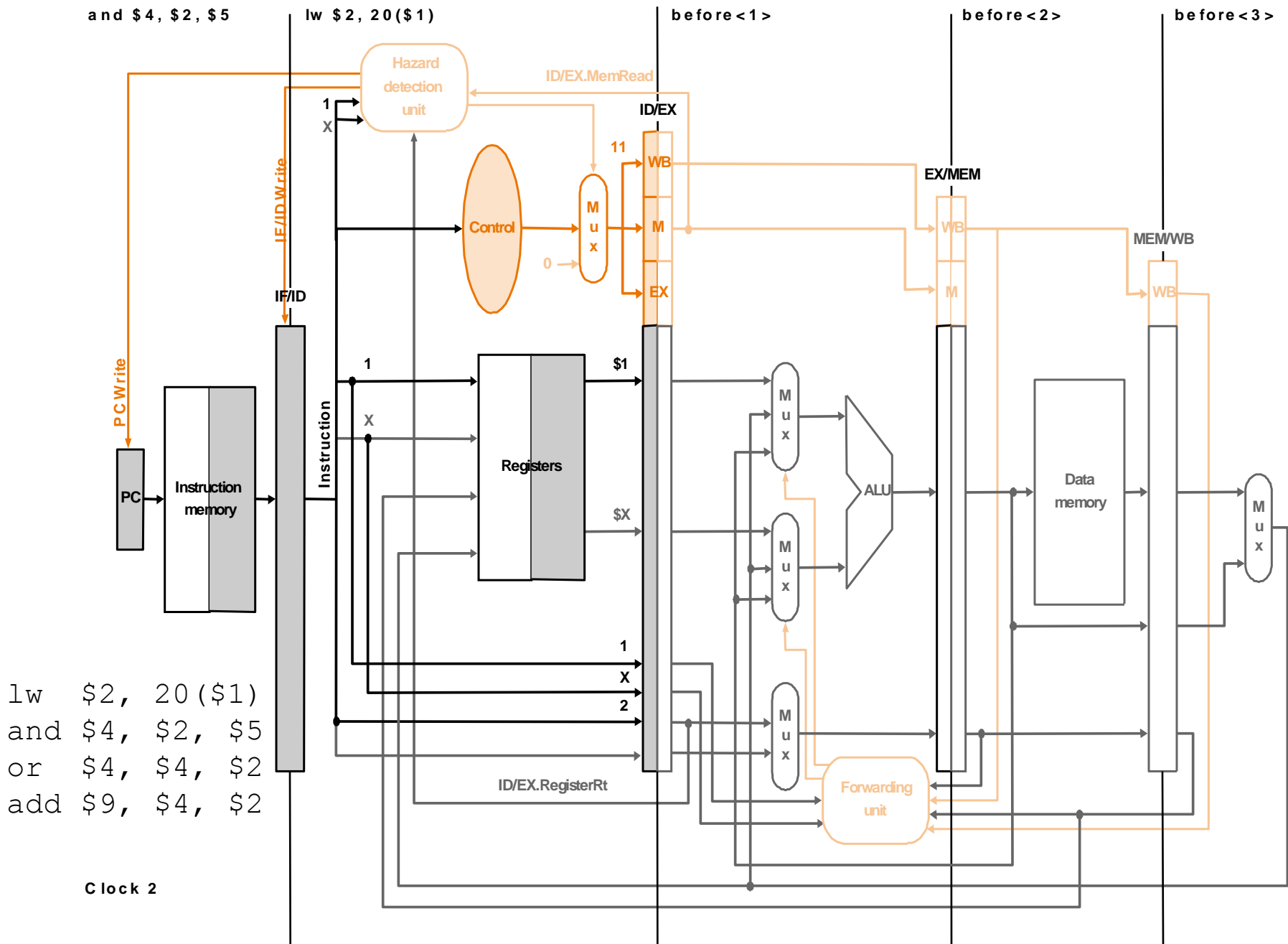
Stall/Bubble in the Pipeline

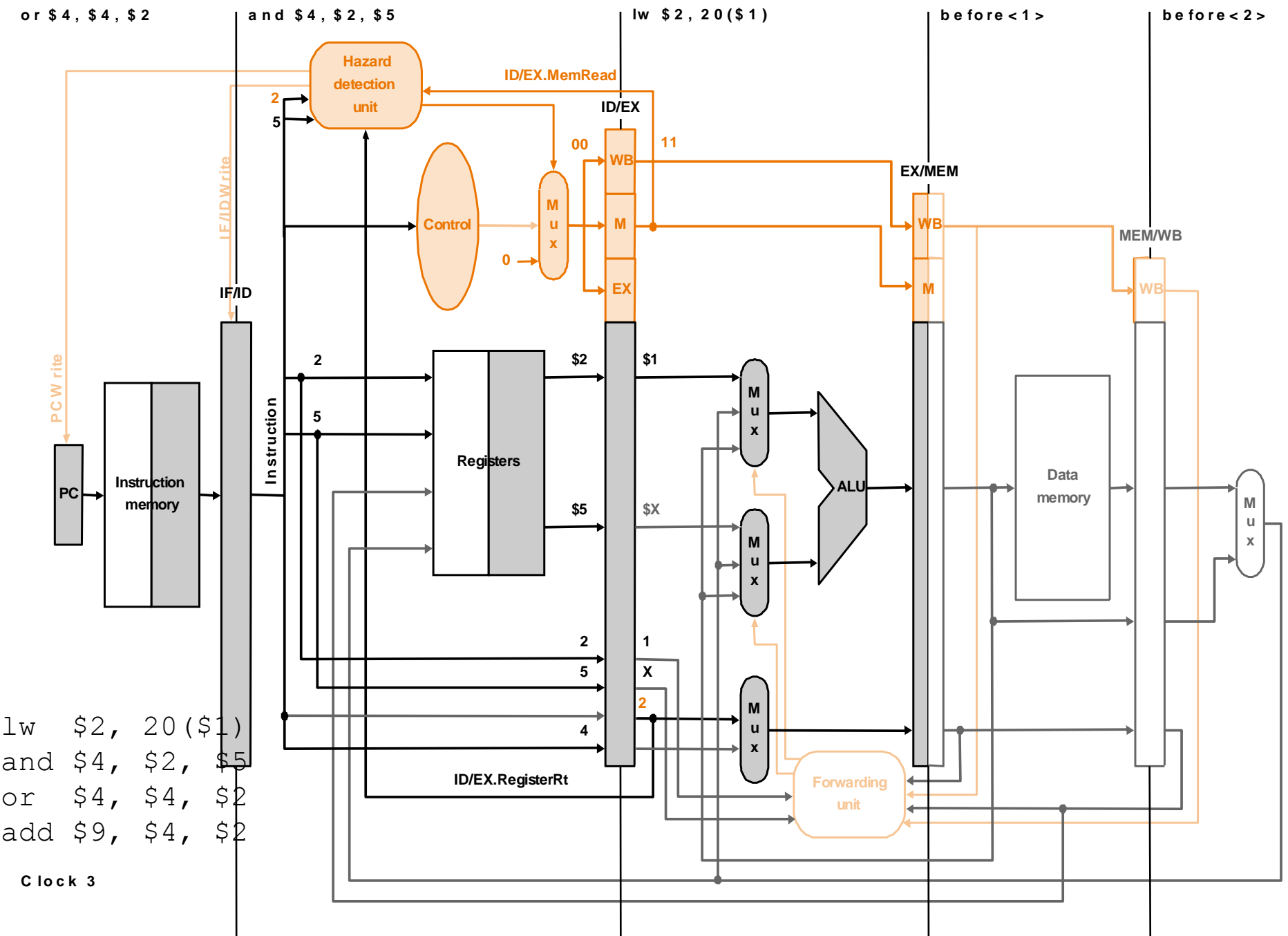


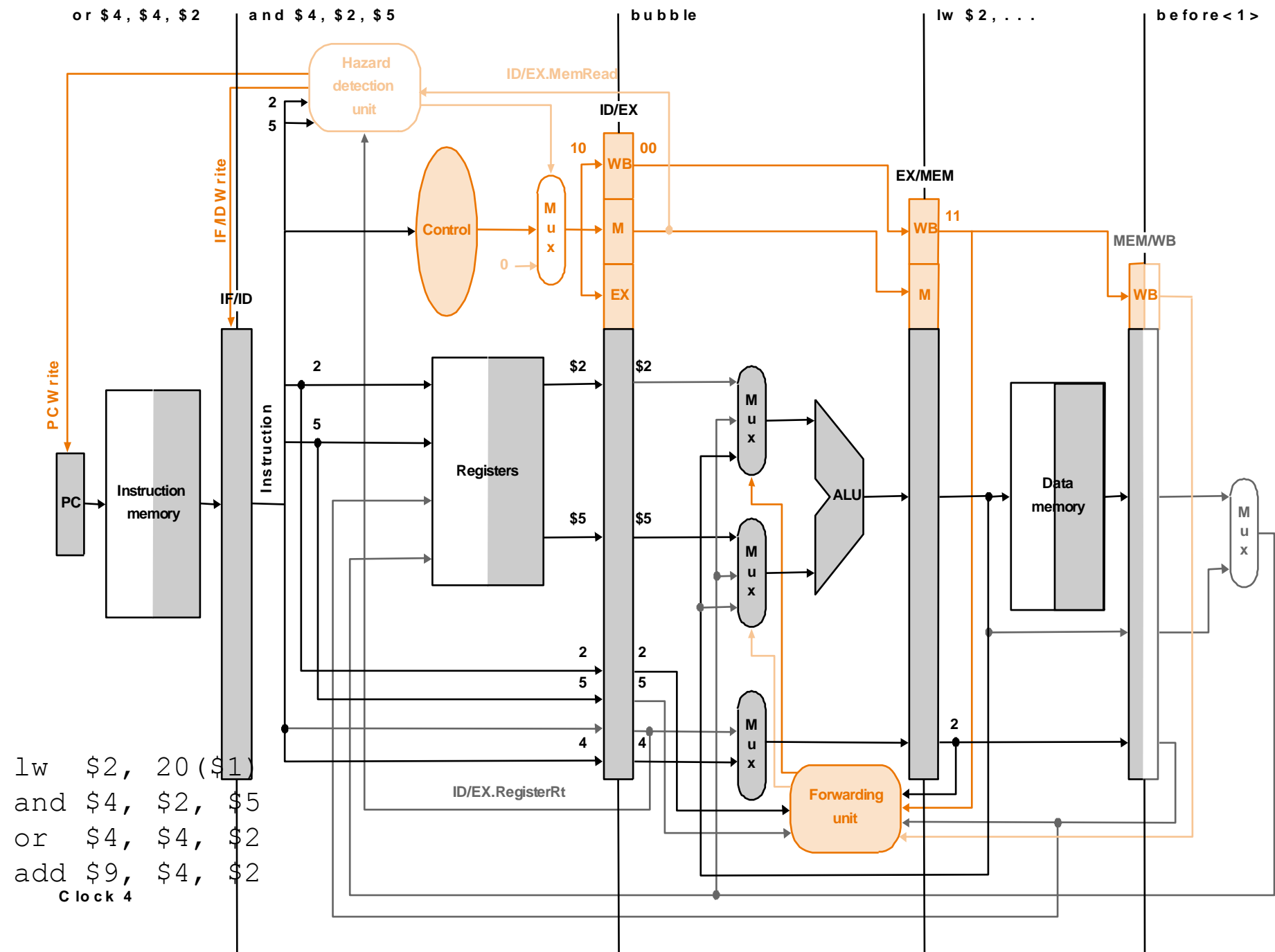
Stalling

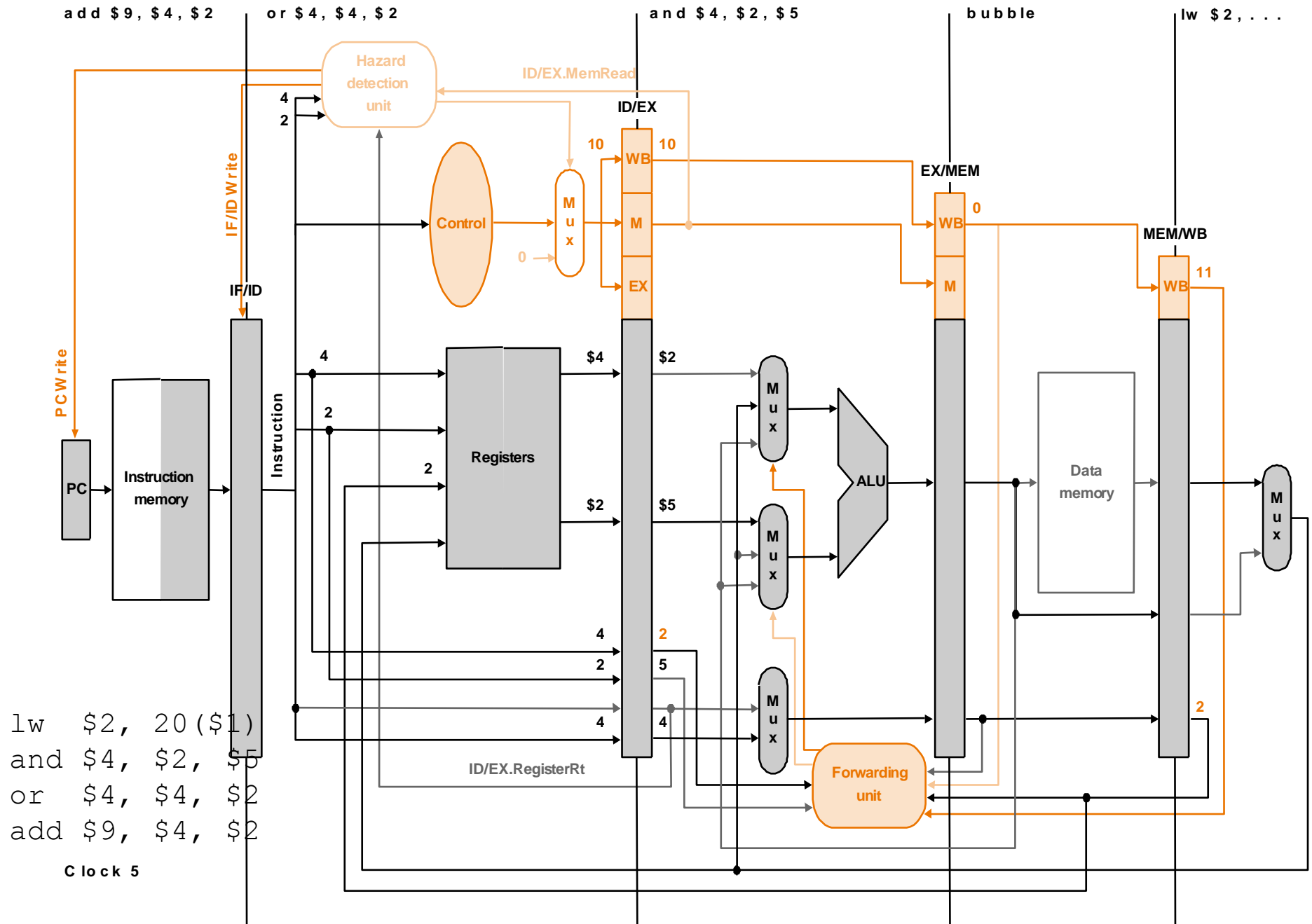
- Execution example:

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









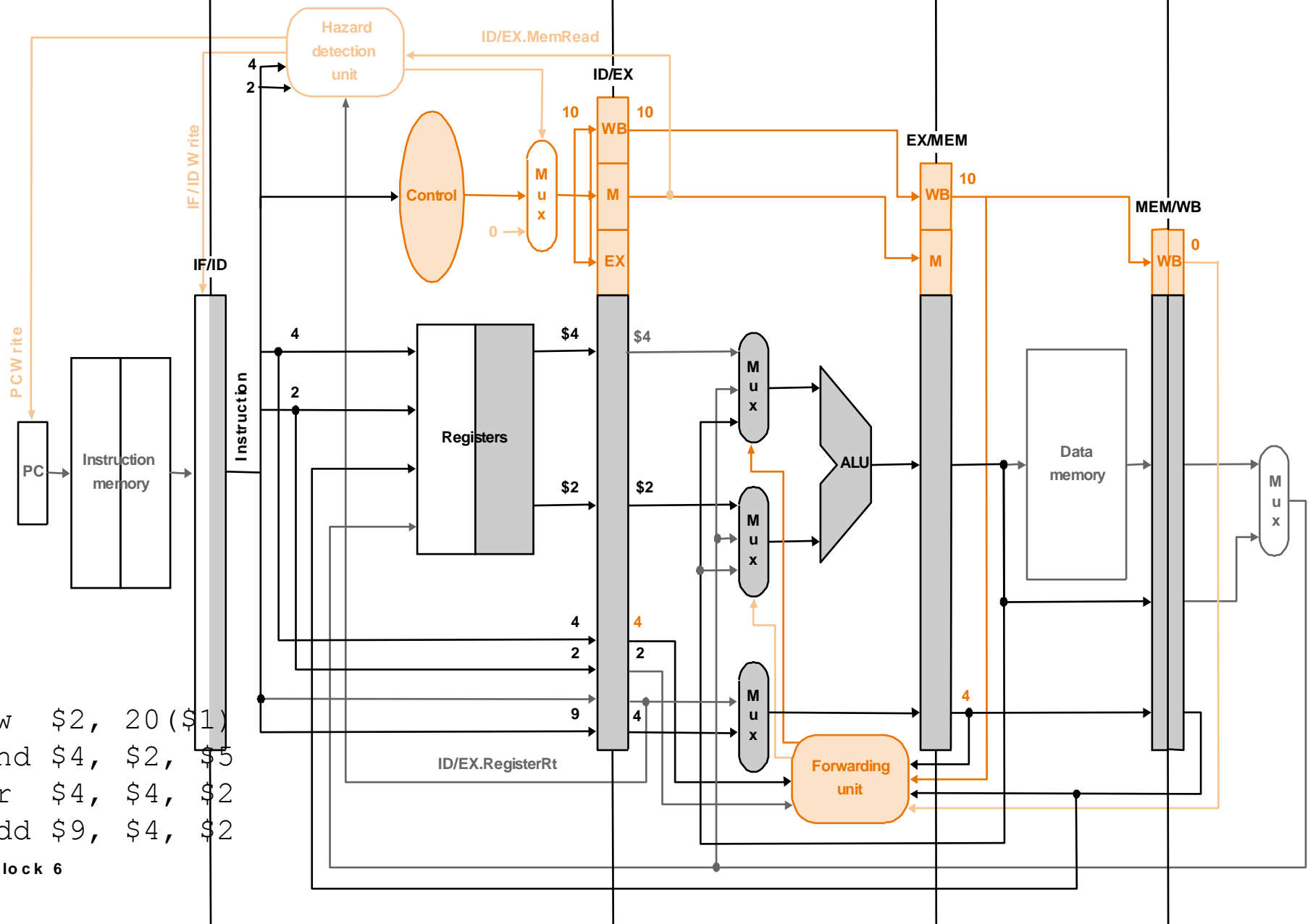
after <1>

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

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

and \$4, ...

bubble



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

Clock 6

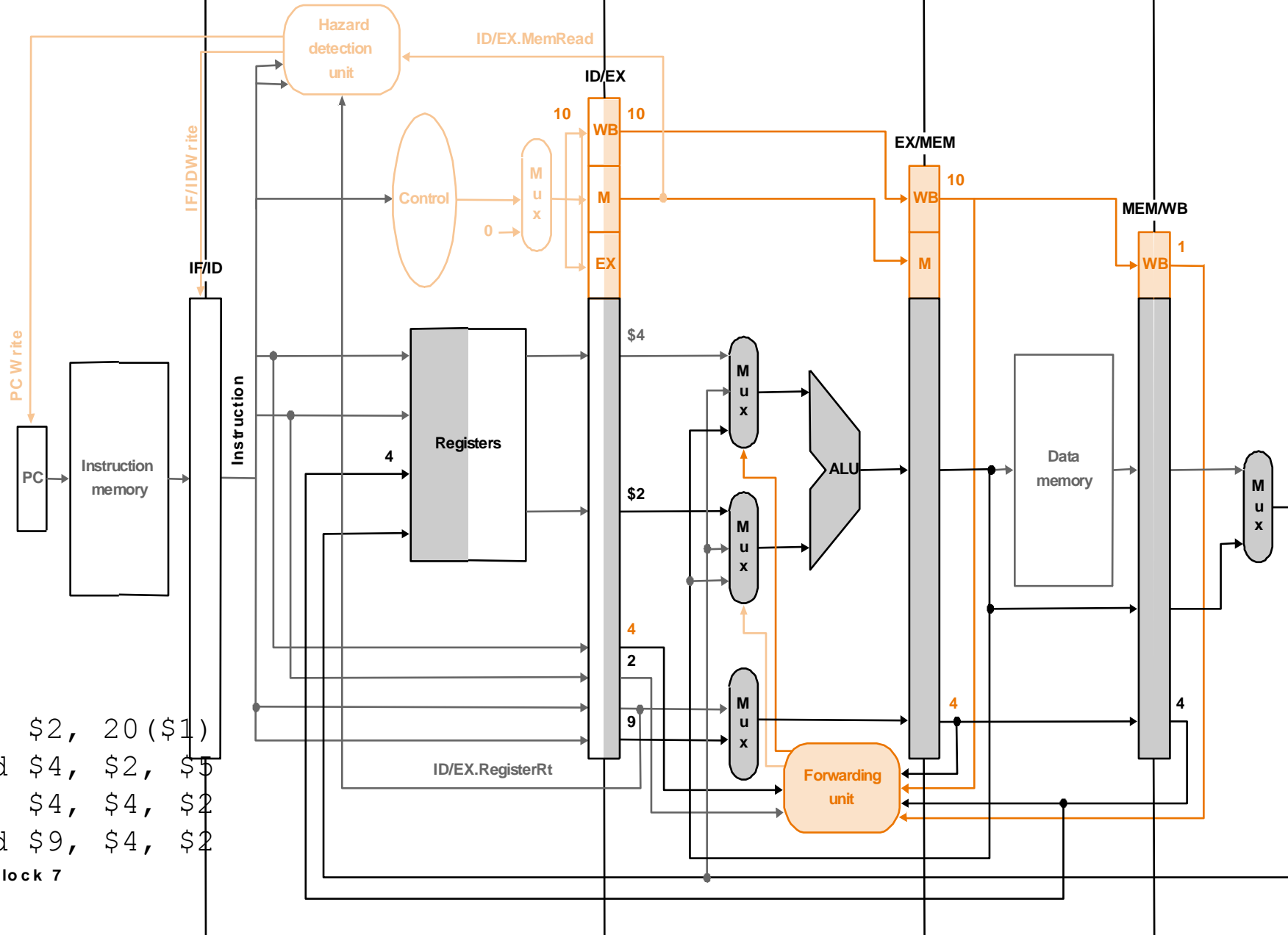
after<2>

after<1>

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

or \$4, ...

and \$4, ...

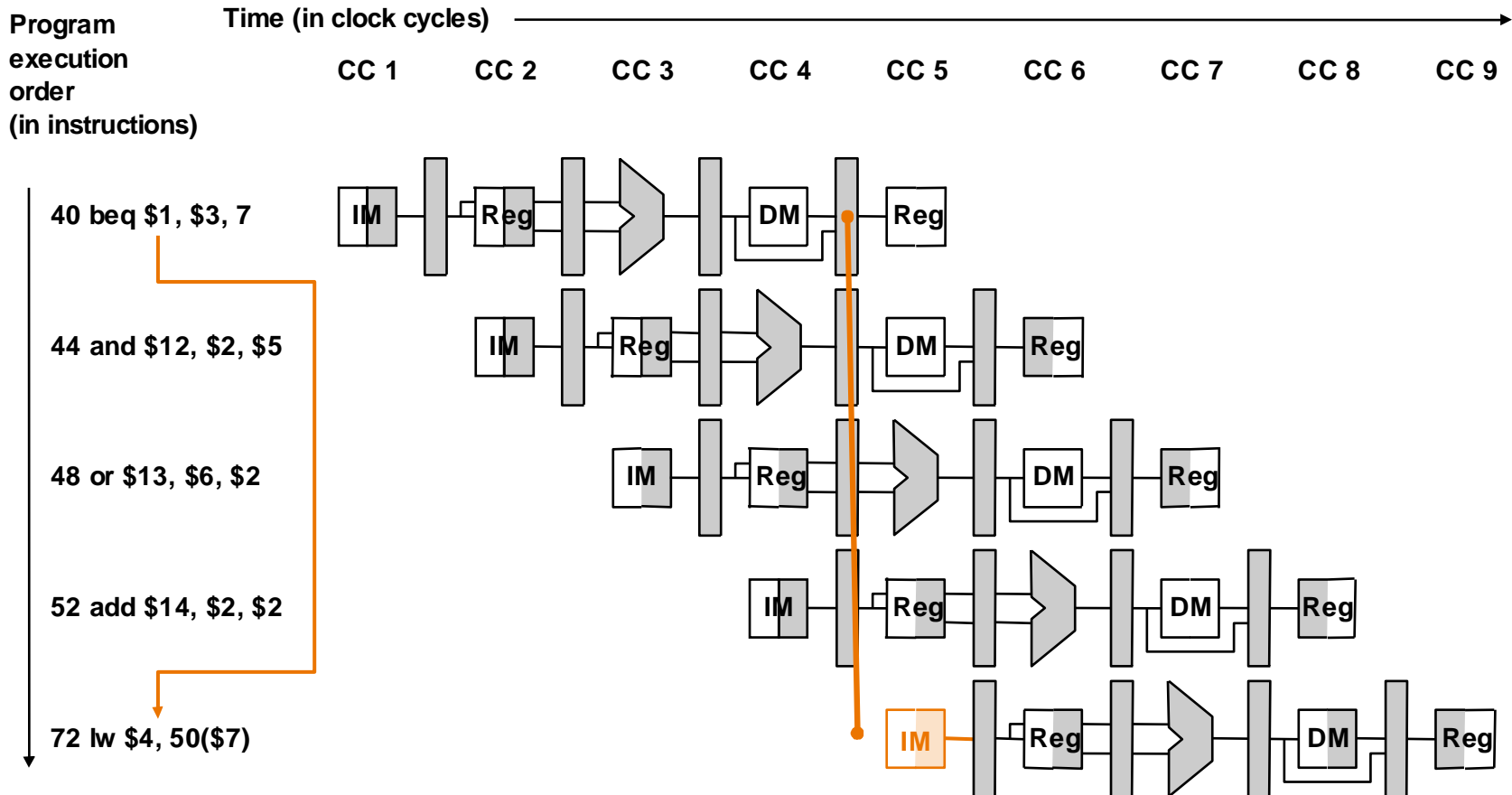


C lock 7

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 – discard instructions already fetched or decoded – and *continue execution at the branch target*

Predicting Branch-not-taken: Misprediction delay



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

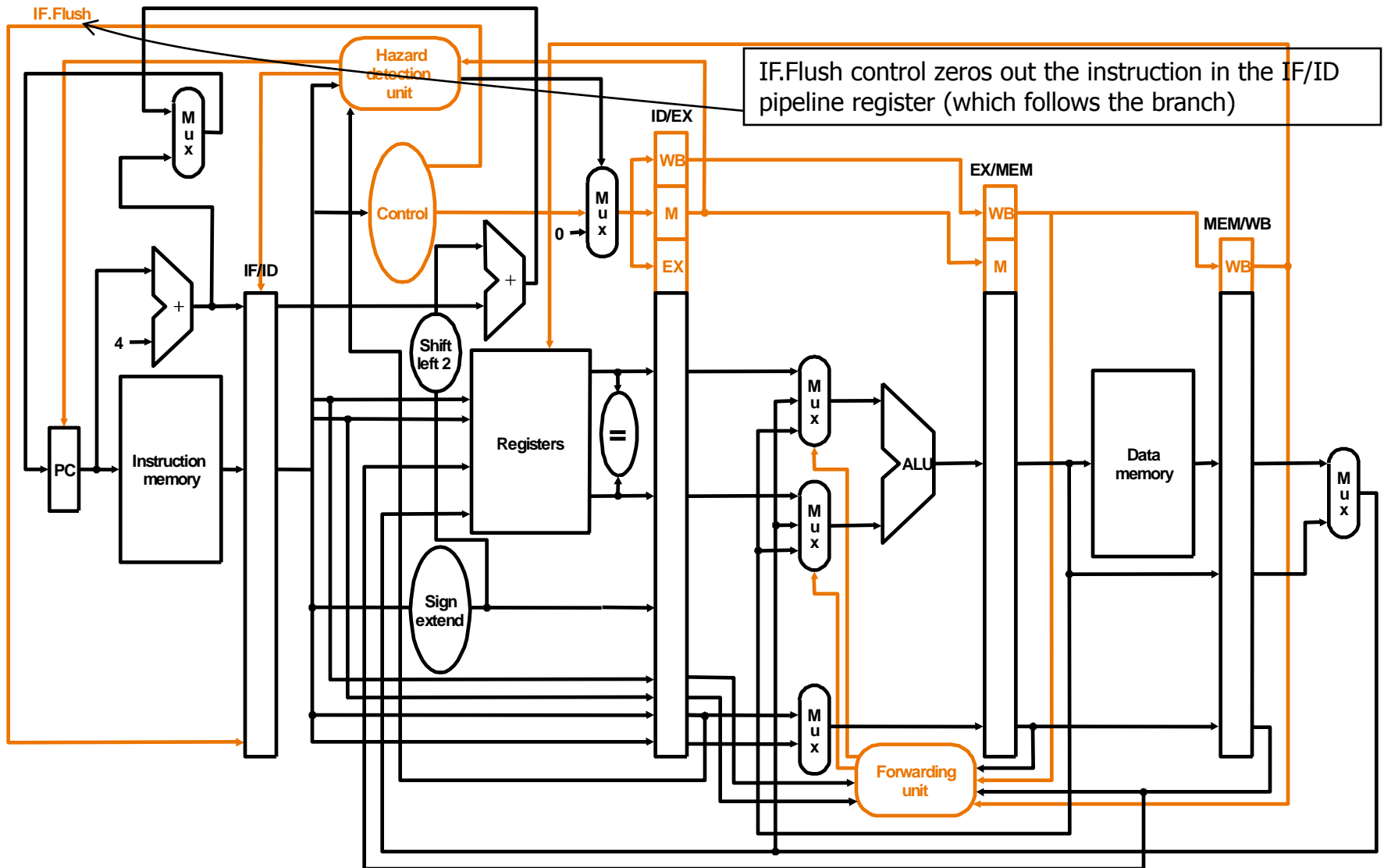
Optimizing the Pipeline to Reduce Branch Delay

- *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)
 - with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember an objective of pipeline design is to keep pipeline stages balanced
 - *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

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard...
- 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 – effectively turning them into `nops` – so they are flushed
 - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimized Datapath for Branch



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

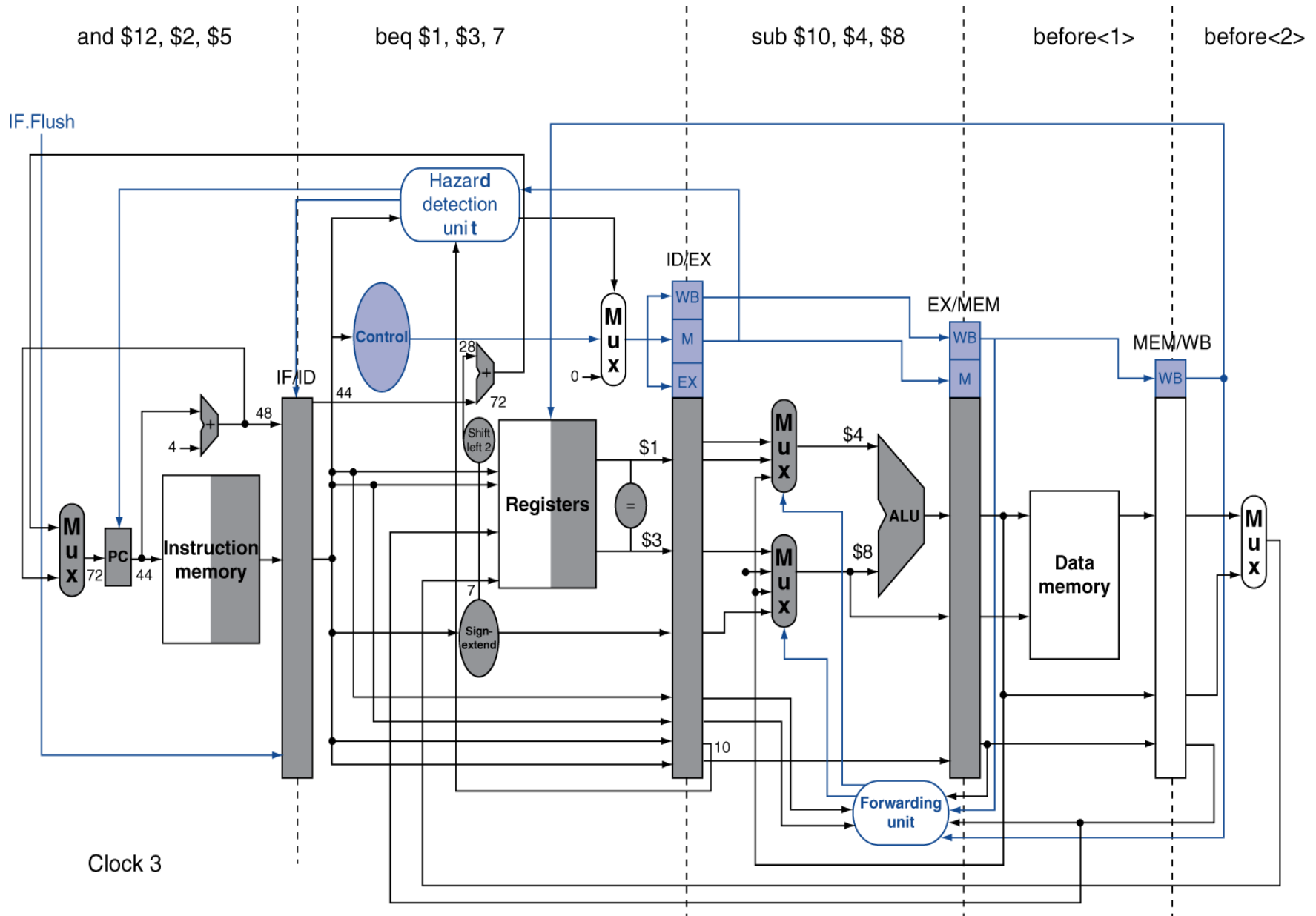
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
 - Target address adder
 - Register comparator
- Example: branch taken

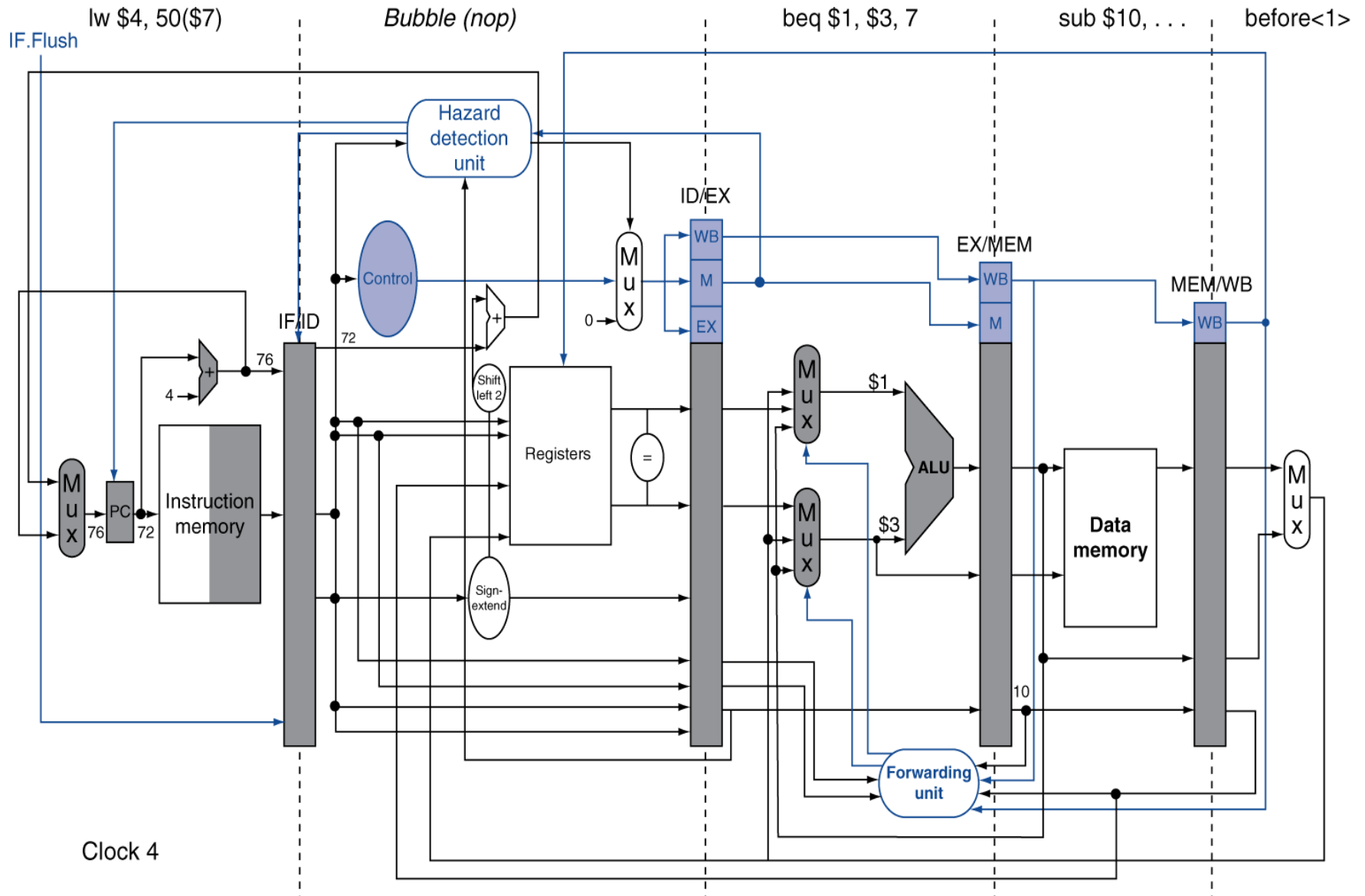
```
36:  sub    $10, $4, $8
40:  beq    $1,  $3, 7
44:  and    $12, $2, $5
48:  or     $13, $2, $6
52:  add    $14, $4, $2
56:  slt    $15, $6, $7

    . . .
72:  lw     $4, 50($7)
```

Example: Branch Taken

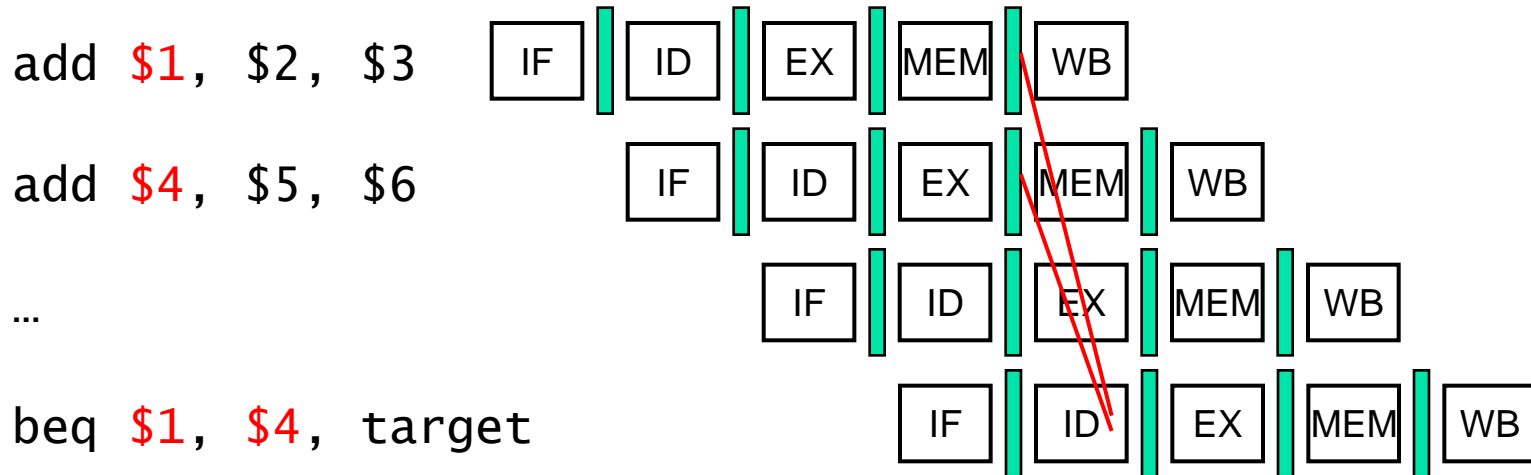


Example: Branch Taken



Data Hazards for Branches

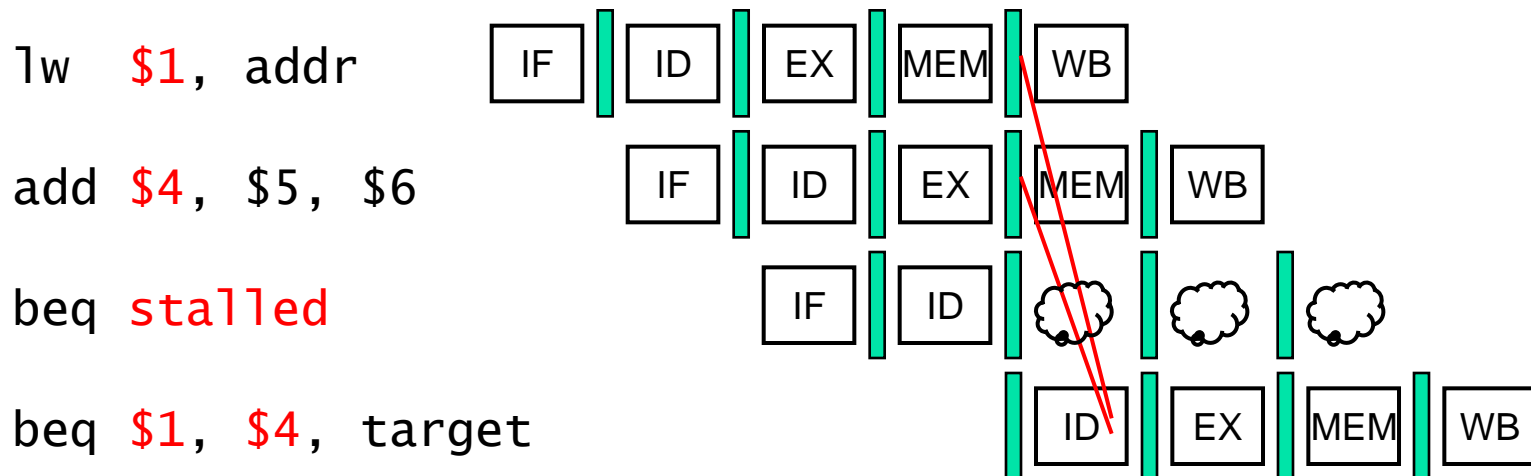
- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction



- Can resolve using forwarding

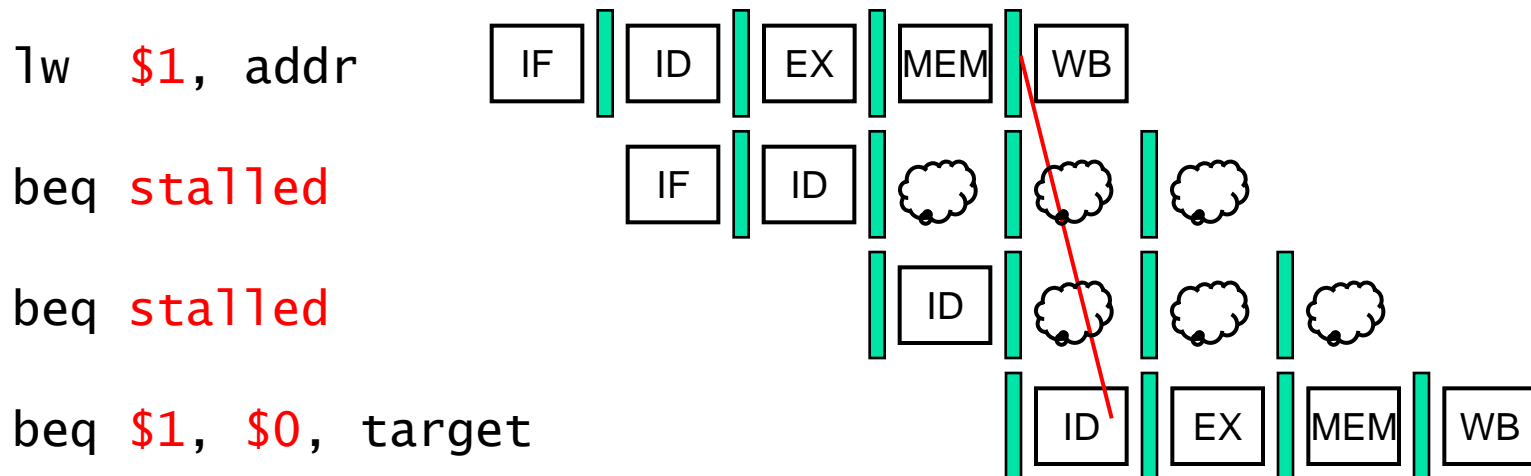
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
 - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
 - Need 2 stall cycles



Simple Example: Comparing Performance

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
 - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
 - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
 - for pipelined execution assume
 - 50% of the loads are followed immediately by an instruction that uses the result of the load
 - 25% of branches are mispredicted
 - branch delay on misprediction is 1 clock cycle
 - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Simple Example: Comparing Performance

- *Single-cycle* : average instruction time 8 ns
- *Multicycle* : average instruction time 8.04 ns
- *Pipelined*:
 - loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
 - stores use 1 cc each
 - branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
 - jumps use 2 cc each
 - ALU instructions use 1 cc each
 - therefore, average CPI is
$$1.5 \times 23\% + 1 \times 13\% + 1.25 \times 19\% + 2 \times 2\% + 1 \times 43\% = 1.18$$
 - therefore, average instruction time is $1.18 \times 2 = 2.36$ ns