Data hazard in pipeline and different solution technique

Presented By

Dr. Banchhanidhi Dash

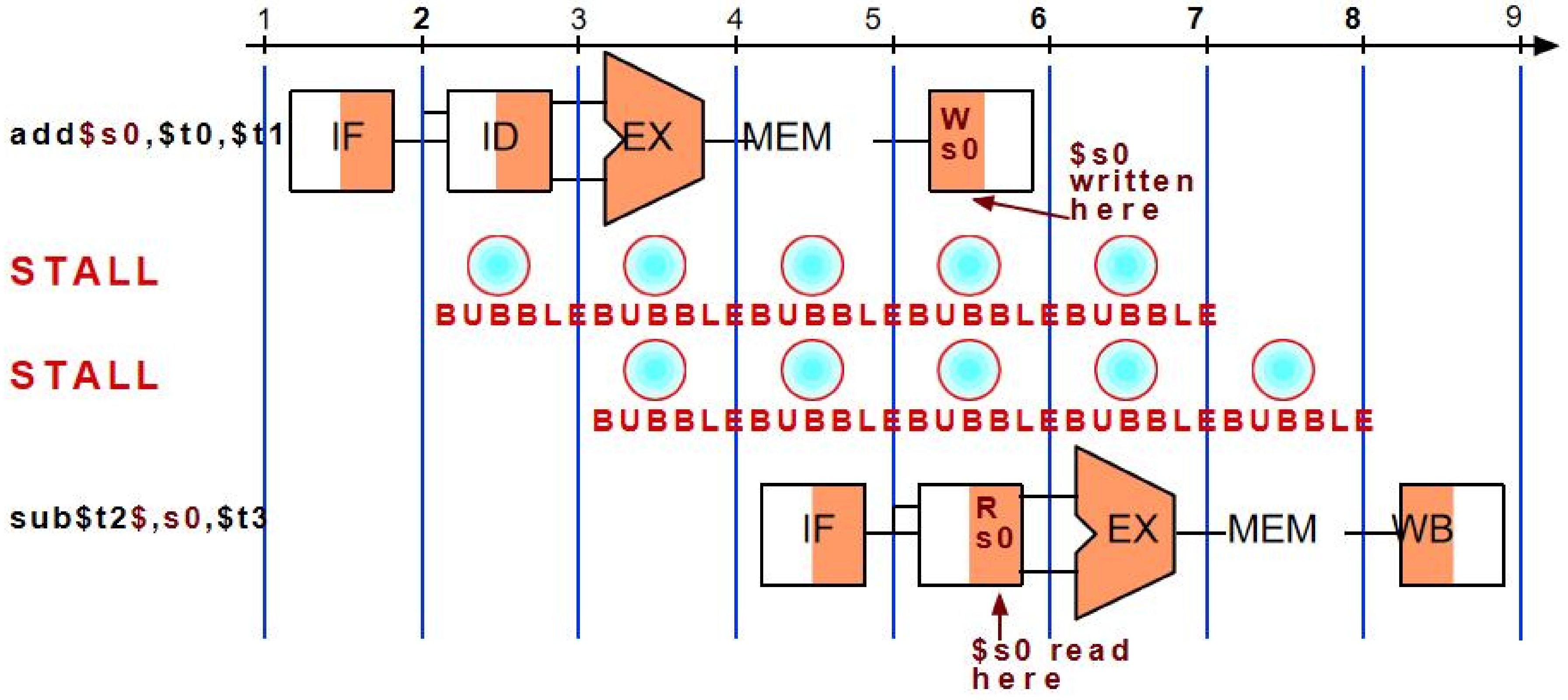
School of Computer Engineering
KIIT University

11/18/2023

Solution for Data Hazards

- Solutions for Data Hazards
  - Stalling
  - Forwarding:
    - » connect new value directly to next stage
  - Reordering

Stalling


1
2
3
4
5
6
7
8
9
add$s0,$t0,$t1
IF
ID
EX
MEM
W s0
$s0 written here
STALL
BUBBLE BUBBLE BUBBLE BUBBLE BUBBLE
STALL
BUBBLE BUBBLE BUBBLE BUBBLE BUBBLE
sub$t2$,s0,$t3
IF
R s0
EX
MEM
WB
$s0 read here

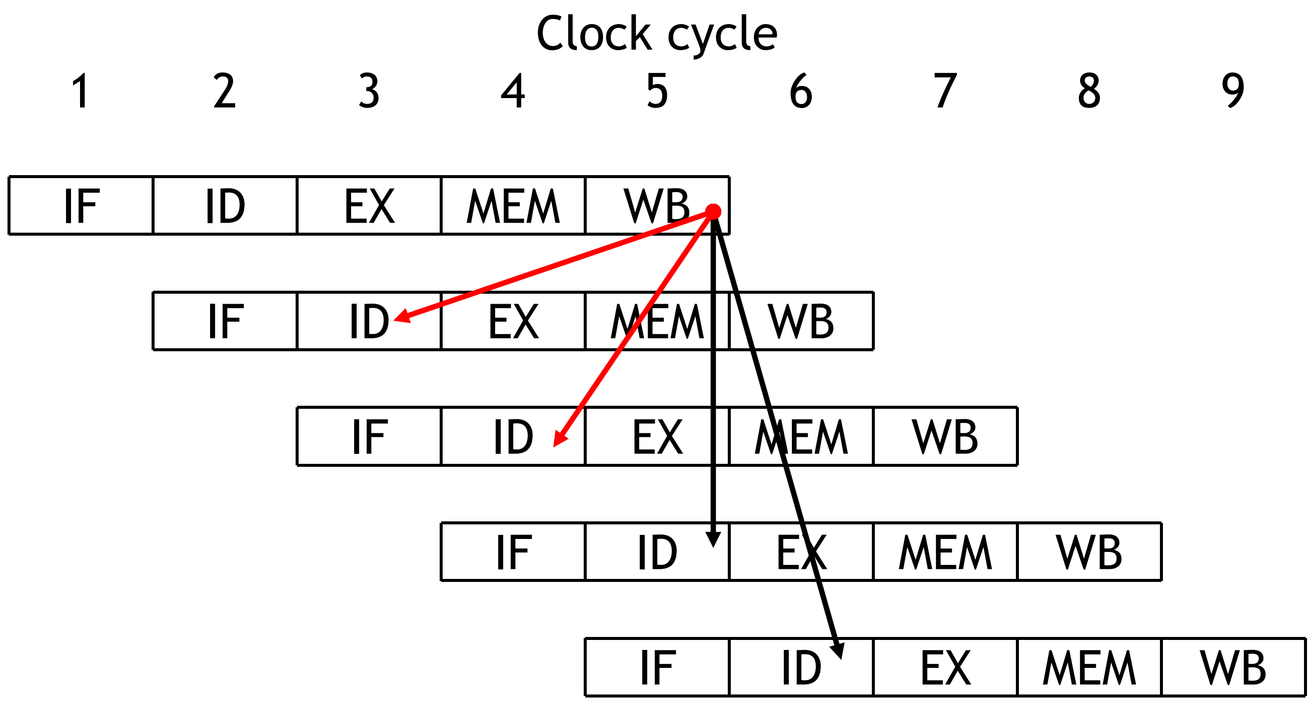
An example with dependencies

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

- There are several dependencies in this new code fragment.
  - The first instruction, SUB, stores a value into $2.
  - That register is used as a source in the rest of the instructions.
- How would this code sequence fare in our pipelined datapath?

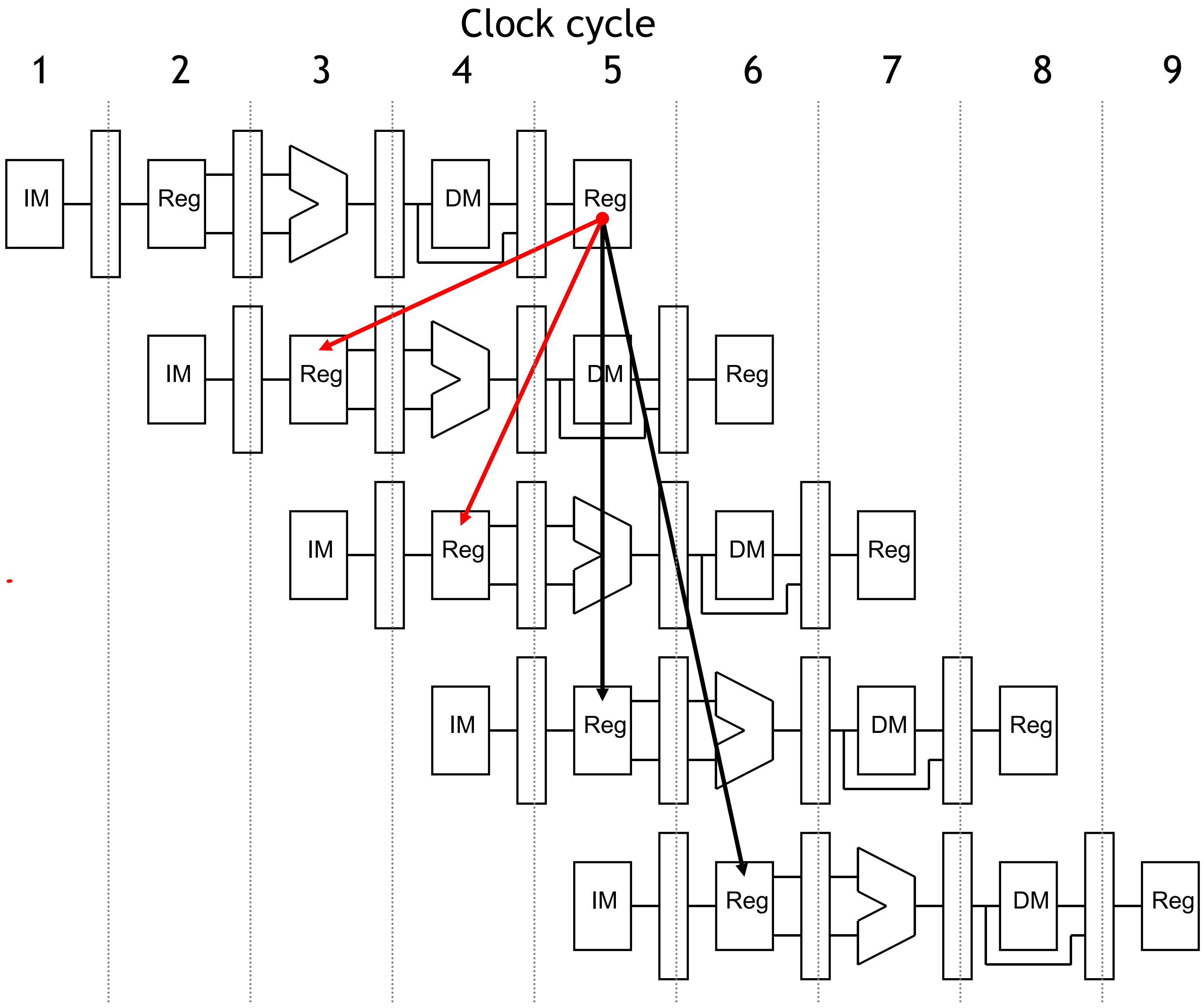
Dependency arrows

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



- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.

A fancier pipeline diagram



sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

Forwarding

- Now, we'll introduce some problems that data hazards can cause for our pipelined processor, and show how to handle them with forwarding.

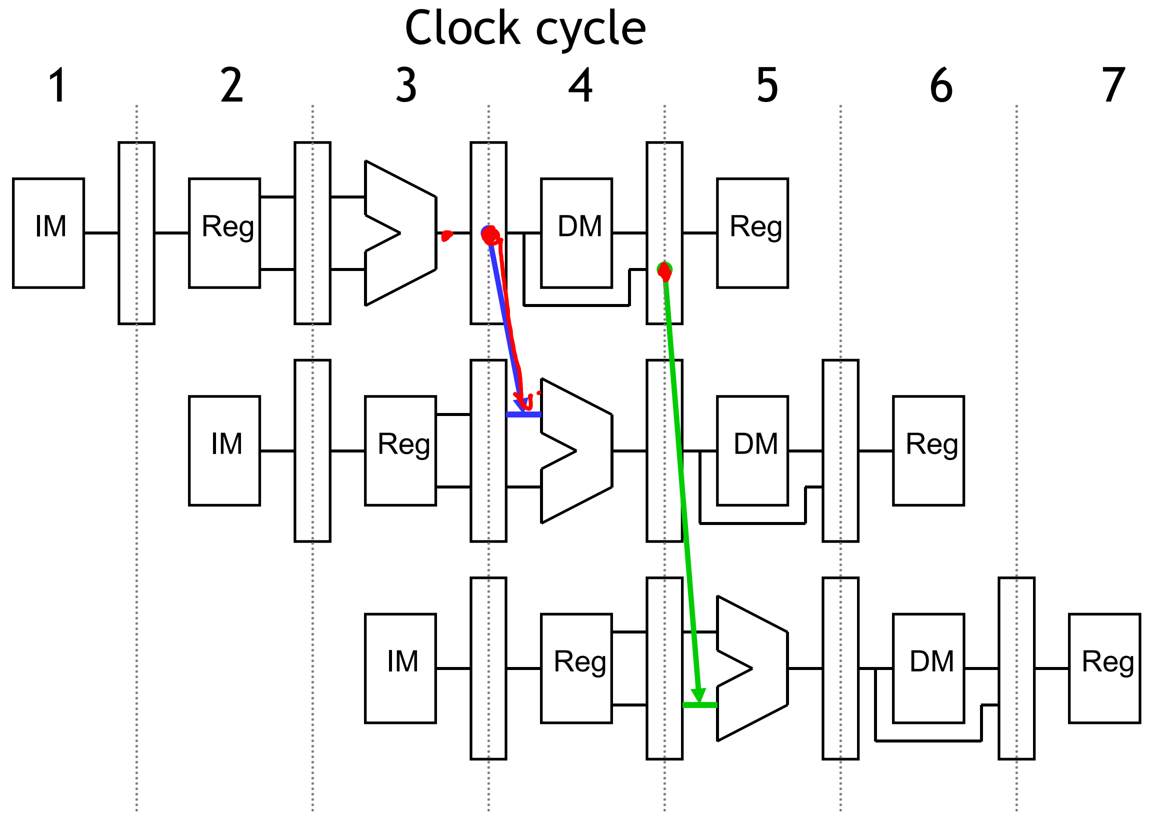
Forwarding

- Since the pipeline registers already contain the ALU result, we could just forward that value to subsequent instructions, to prevent data hazards.
  - In clock cycle 4, the AND instruction can get the value \$1 – \$3 from the EX/MEM pipeline register used by sub.
  - Then in cycle 5, the OR can get that same result from the MEM/WB pipeline register being used by SUB.

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

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

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



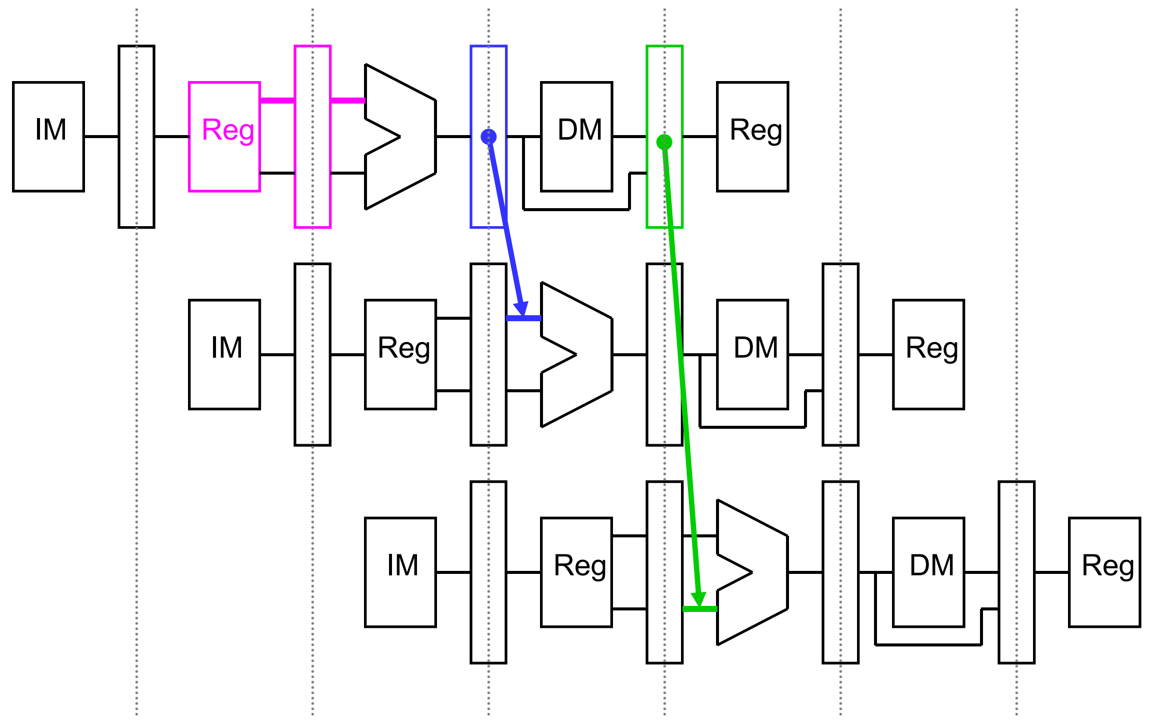
Outline of forwarding hardware

- A forwarding unit selects the correct ALU inputs for the EX stage.
  - If there is no hazard, the ALU’s operands will come from the register file, just like before.
  - If there is a hazard, the operands will come from either the EX/MEM or MEM/WB pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named ForwardA and ForwardB.

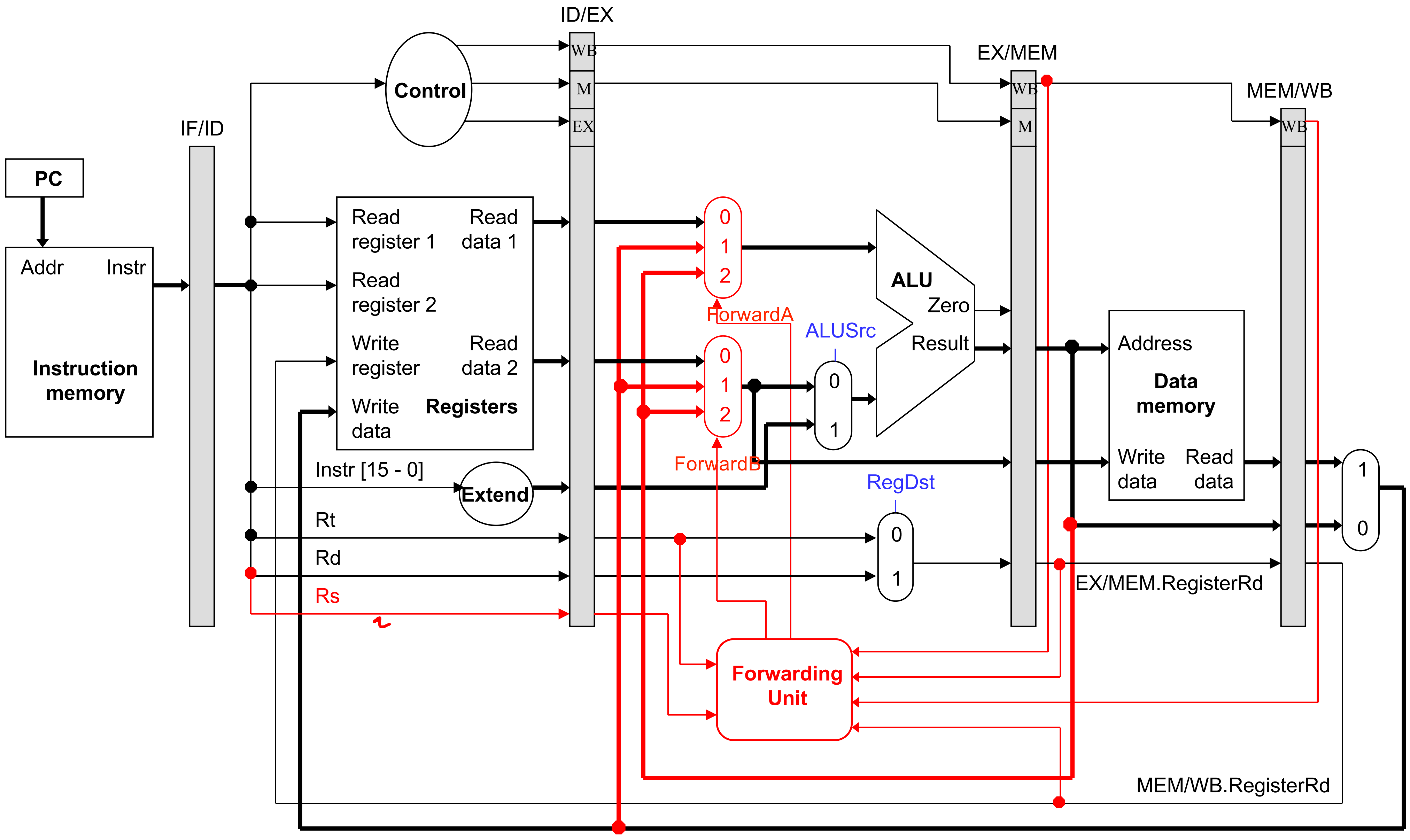
sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2



Complete pipelined datapath



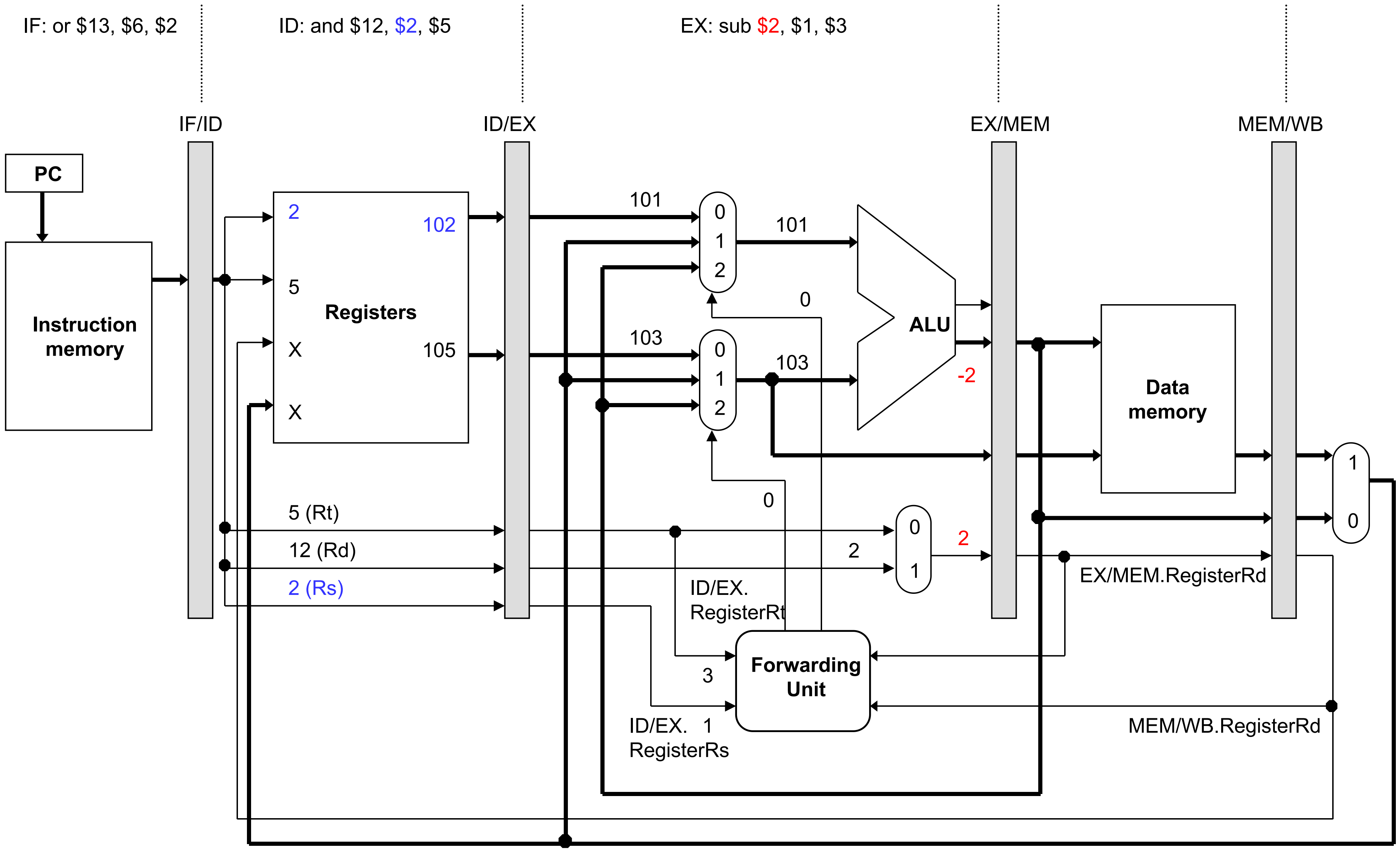
Example

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

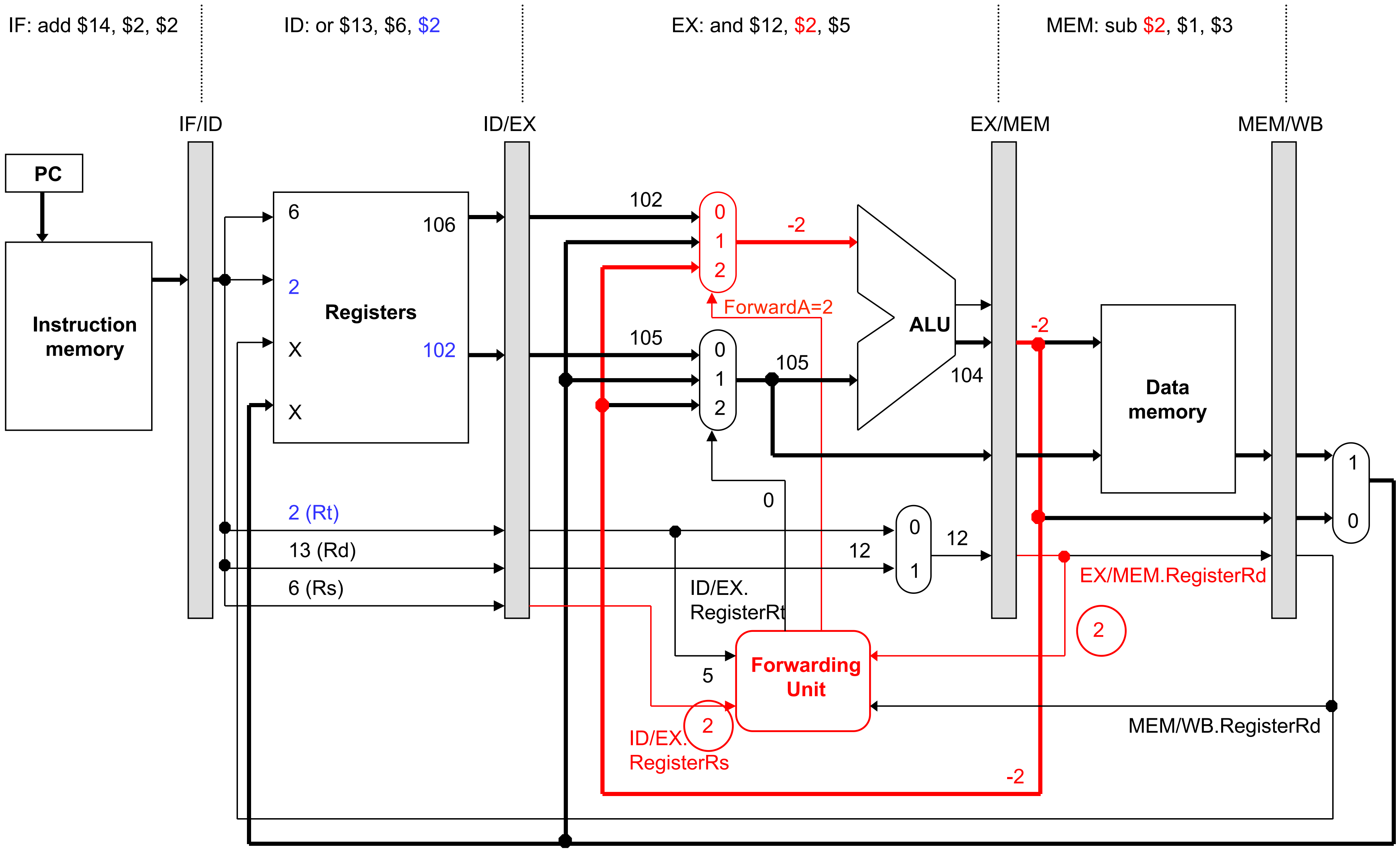
- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain −2 (101 − 103).
  - The other instructions should all use −2 as one of their operands.

- We’ll try to keep the example short.
  - Assume no forwarding is needed except for register $2.
  - We’ll skip the first two cycles, since they’re the same as before.

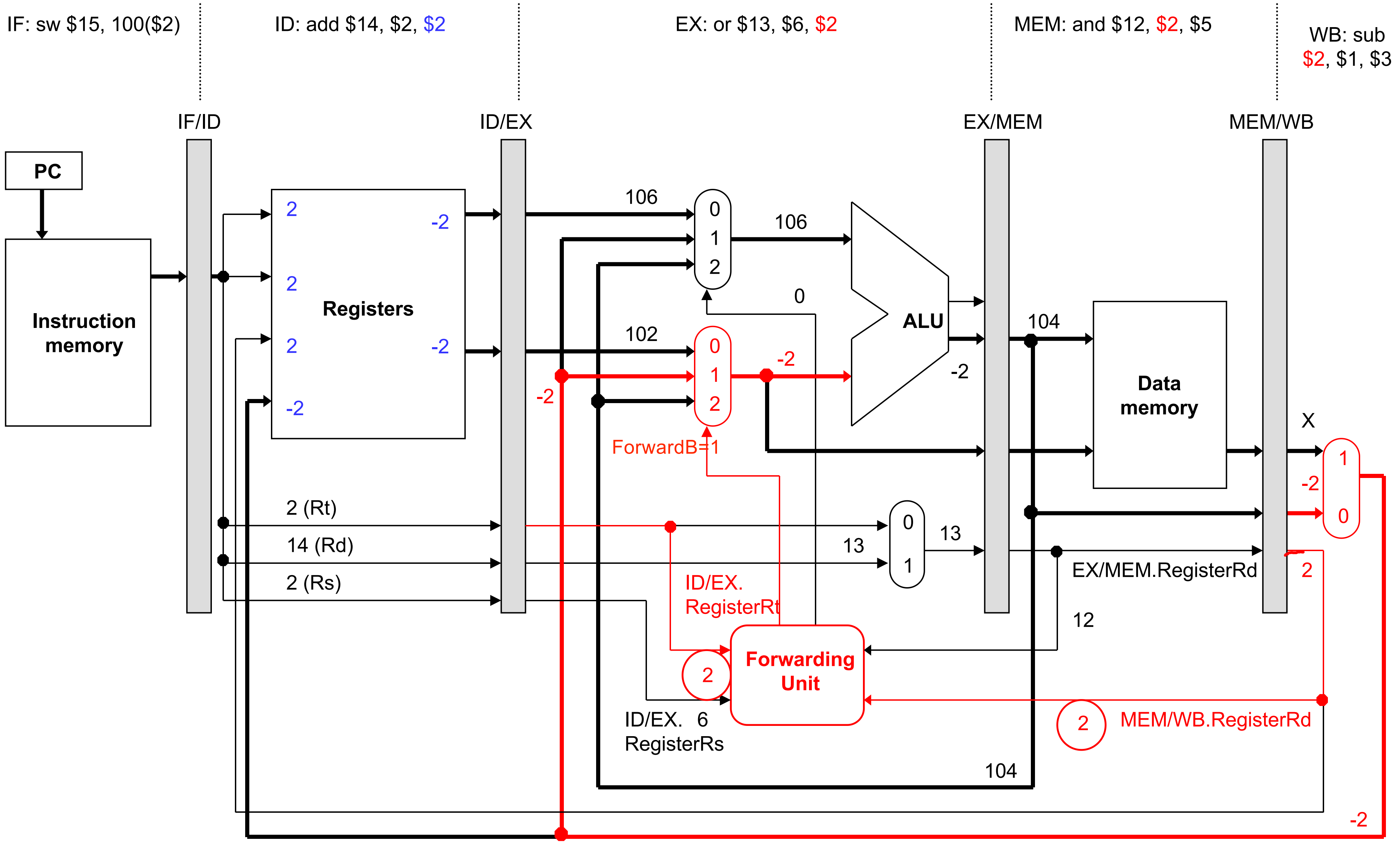
Clock cycle 3



Clock cycle 4: forwarding $2 from EX/MEM



Clock cycle 5: forwarding $2 from MEM/WB



The forwarding unit

- The forwarding unit has several control signals as inputs.

ID/EX.RegisterRs	EX/MEM.RegisterRd	MEM/WB.RegisterRd
ID/EX.RegisterRt	EX/MEM.RegWrite	MEM/WB.RegWrite

  The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU.
- Some new buses route data from pipeline registers to the new muxes.

EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2
```

- The second ALU source is similar.

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRt)
then ForwardB = 2
```

MEM/WB hazard equations

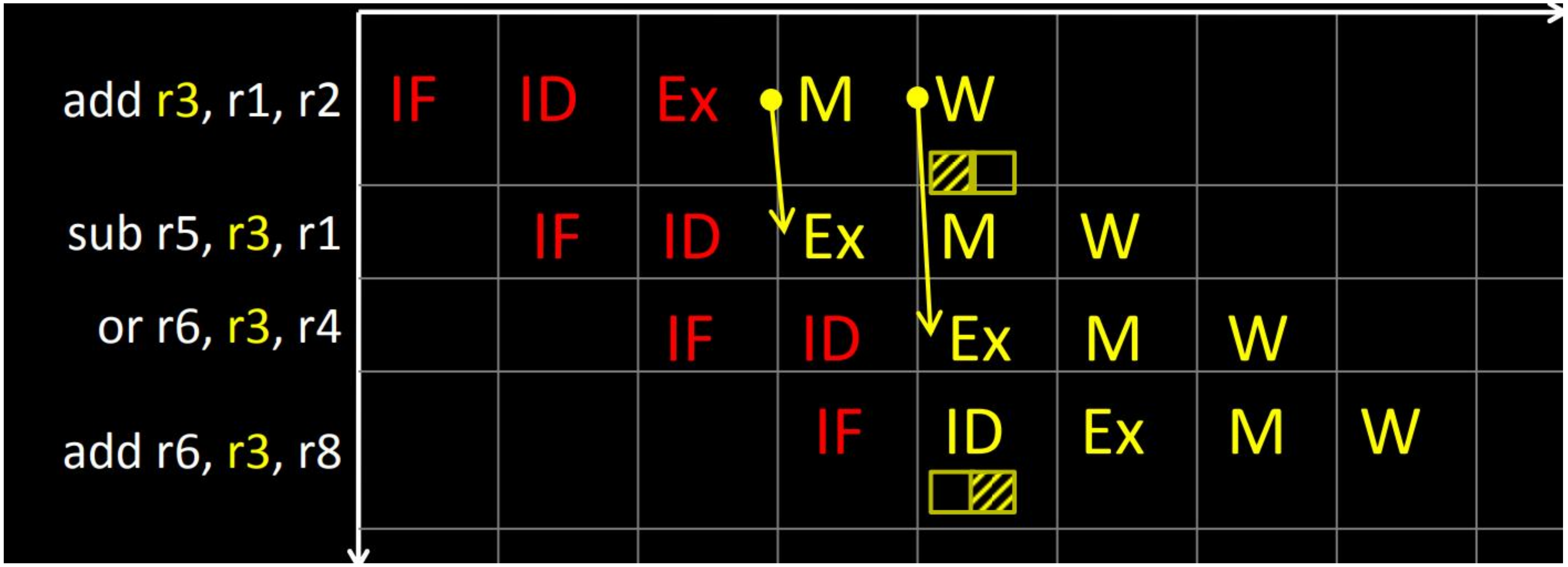
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

```
if (MEM/WB.RegWrite = 1
    and MEM/WB.RegisterRd = ID/EX.RegisterRs
    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
then ForwardA = 1
```

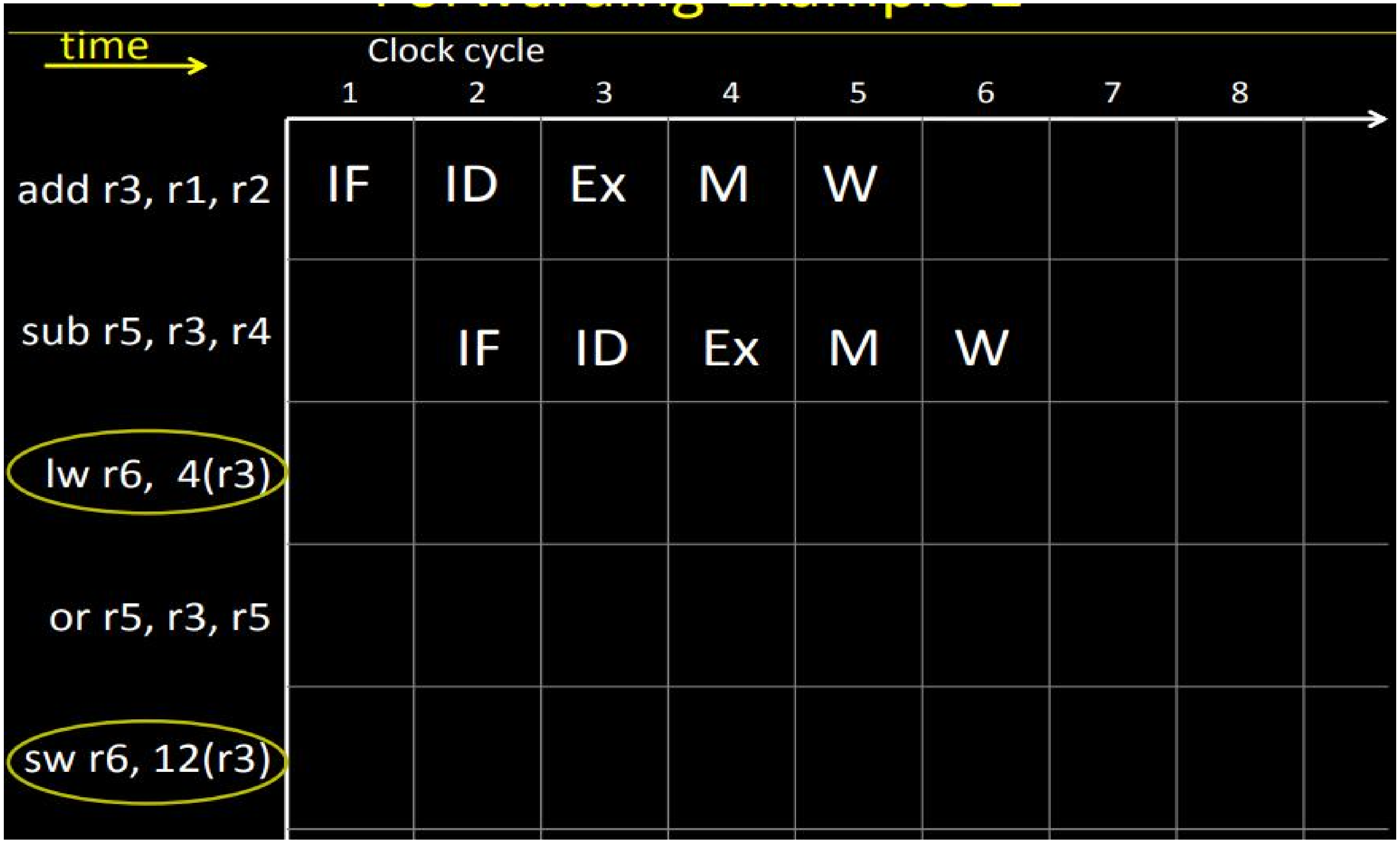
- The second ALU operand is handled similarly.

```
if (MEM/WB.RegWrite = 1
    and MEM/WB.RegisterRd = ID/EX.RegisterRt
    and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
then ForwardB = 1
```

forwarding example



forwarding example :
clculate total,number of clock cycle to execute the following 5 instruction

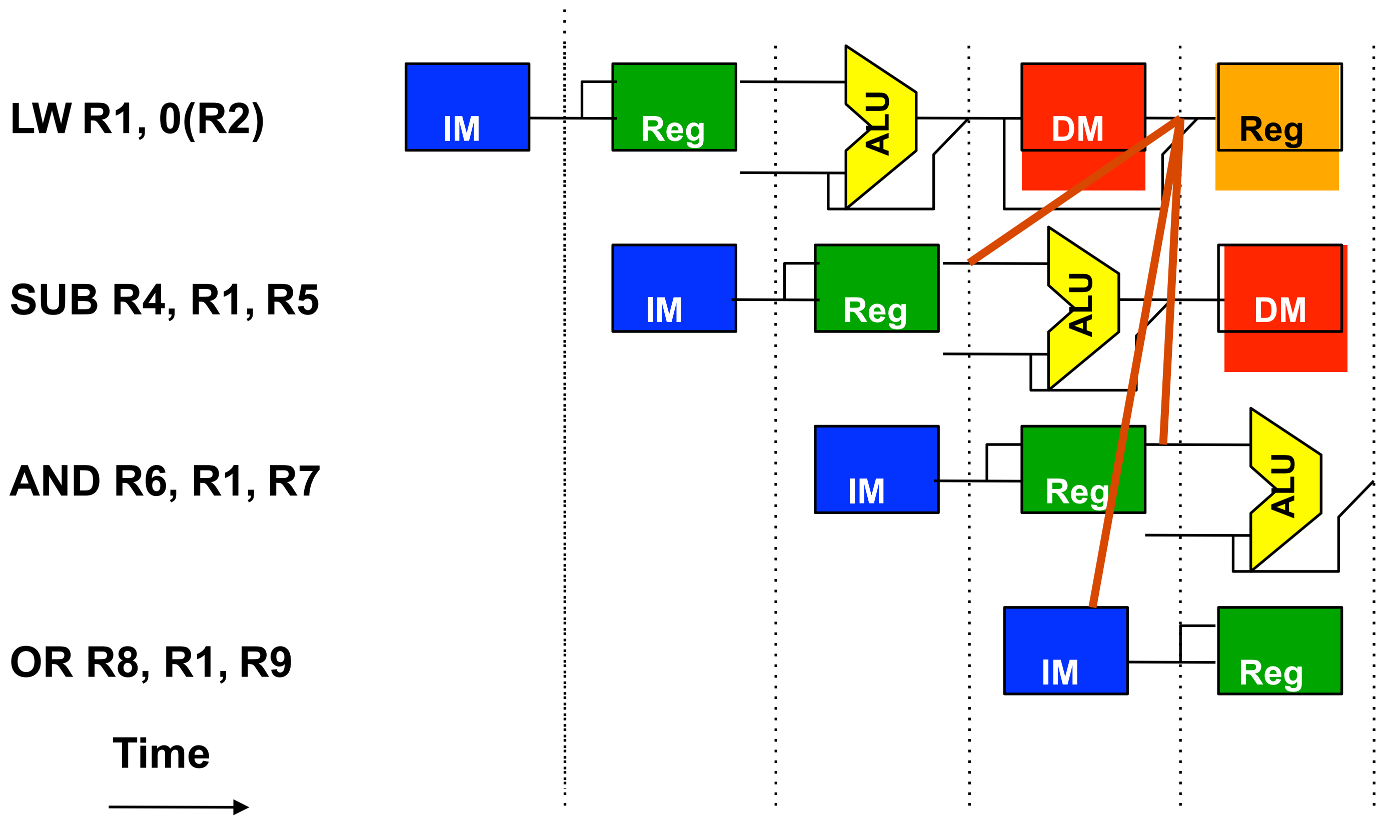

time

Clock cycle

	1	2	3	4	5	6	7	8	
add r3, r1, r2	IF	ID	Ex	M	W				
sub r5, r3, r4		IF	ID	Ex	M	W			
lw r6, 4(r3)									
or r5, r3, r5									
sw r6, 12(r3)									

- So far, we have discussed data hazards that can occur in pipelined CPUs if some instructions depend upon others that are still executing.
  - Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage.
  - The pipeline continues running at full speed, with one instruction beginning on every clock cycle.
- Now, we'll see some real limitations of pipelining.
  - Forwarding may not work for data hazards from load instructions.
  - Branches affect the instruction fetch for the next clock cycle.
- In both of these cases we may need to slow down, or stall, the pipeline.

Forwarding doesn’t always work

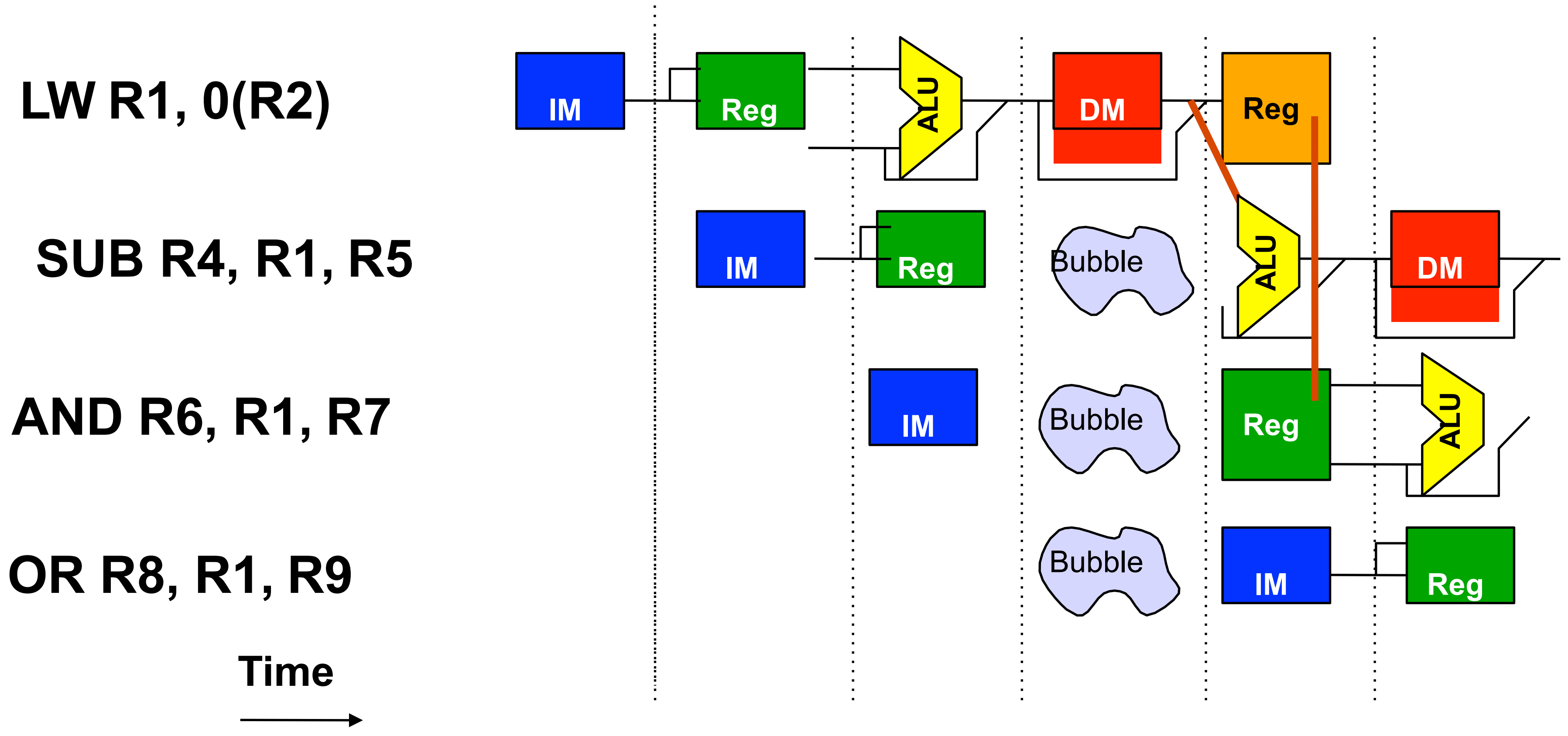


Load has a latency that forwarding can’t solve.

Pipeline must stall until hazard cleared (starting with instruction that wants to use data until source produces it).

Can’t get data to subtract b/c result needed at beginning of CC #4, but not produced until end of CC #4.

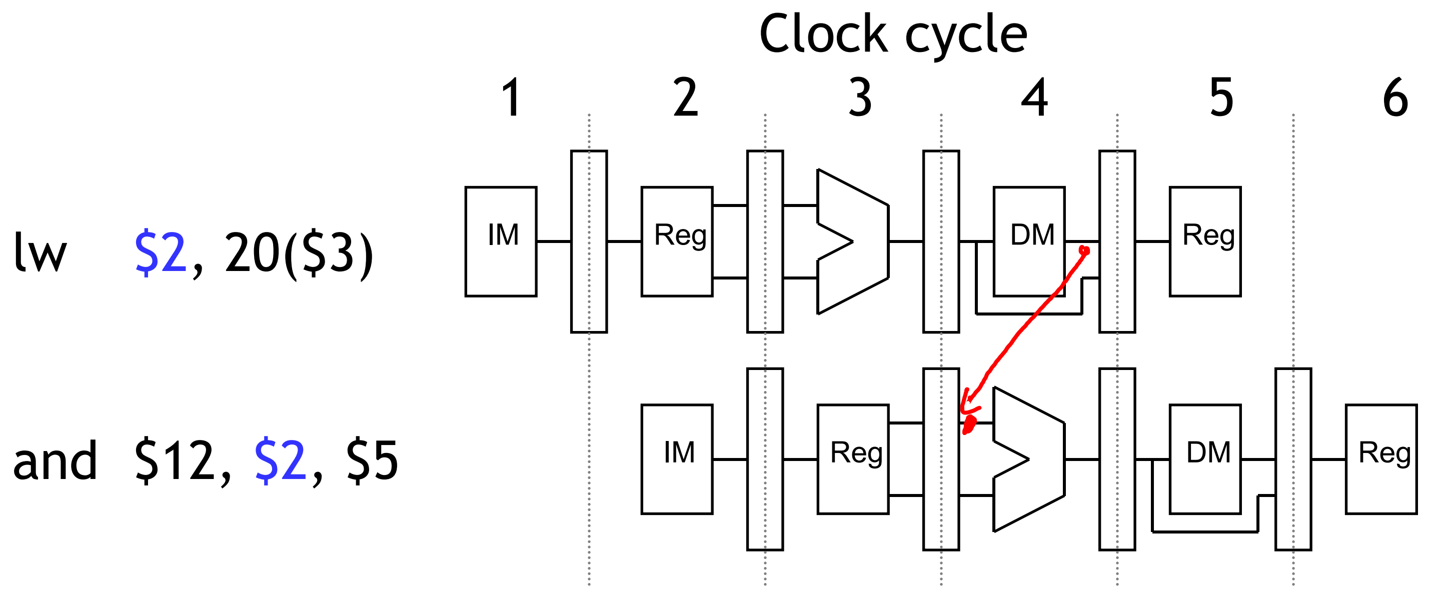
The solution pictorially



Insertion of bubble causes # of cycles to complete this sequence to grow by 1

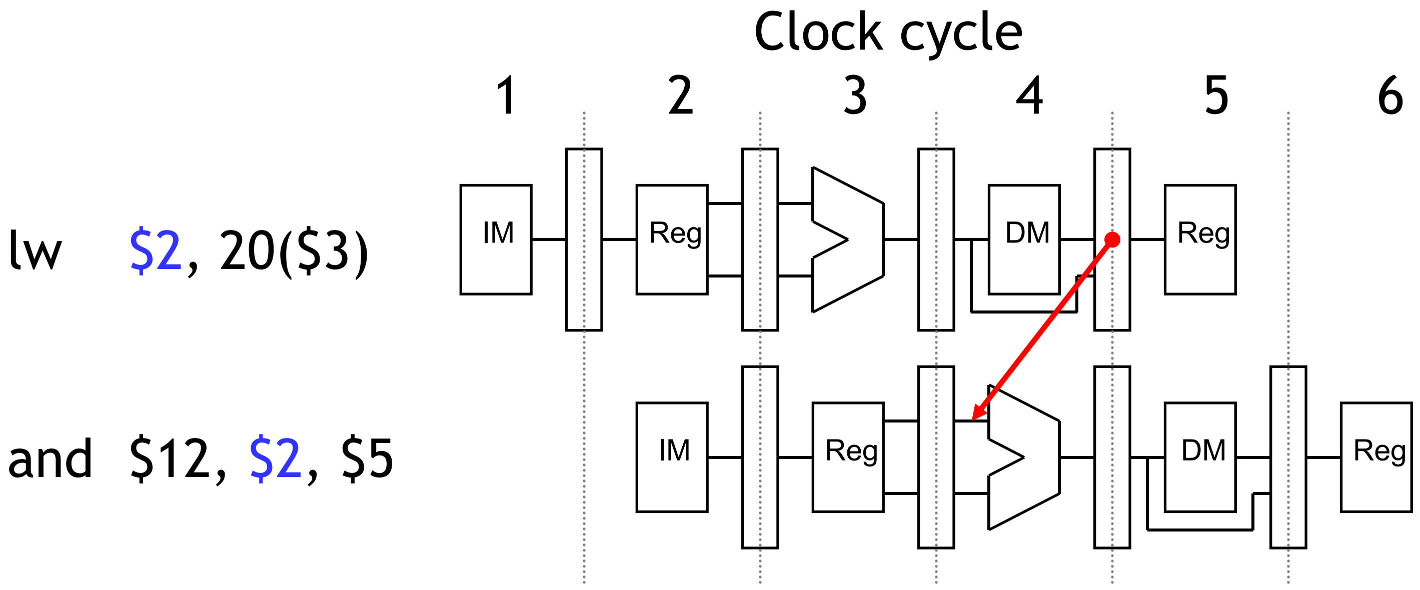
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - How does this change the data hazard?



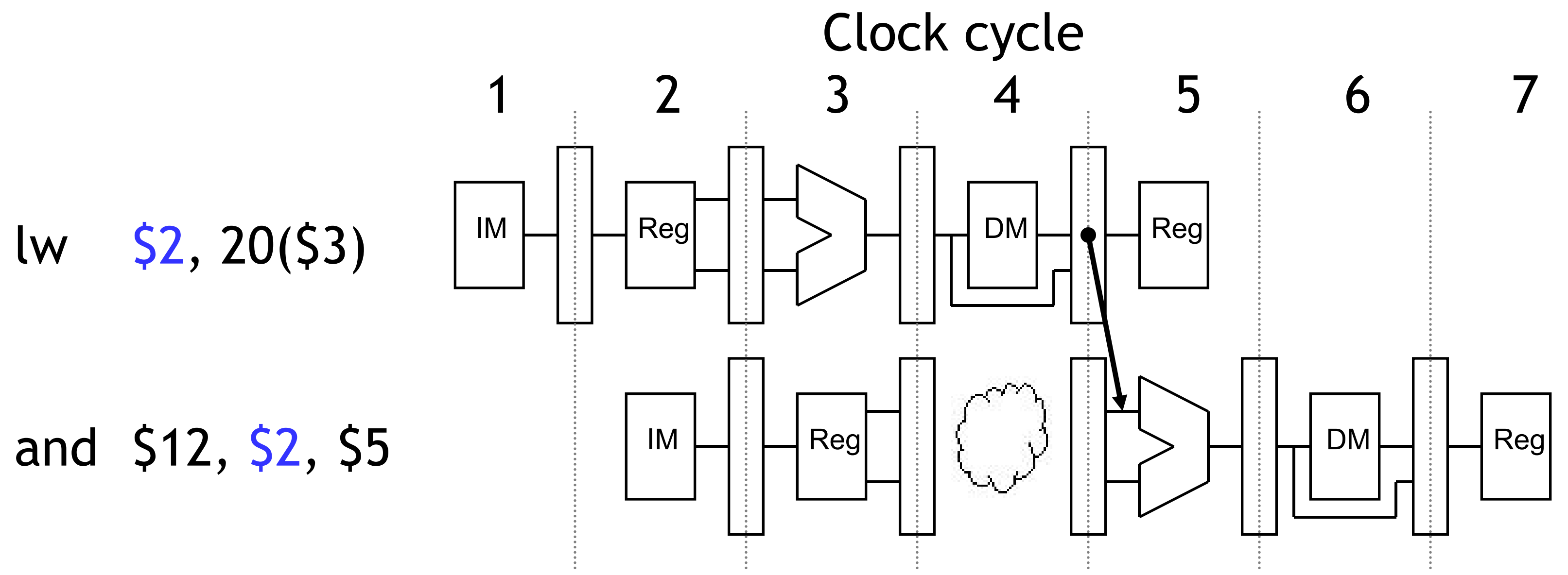
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - The load data doesn't come from memory until the *end* of cycle 4.
  - But the AND needs that value at the *beginning* of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.



Stalling

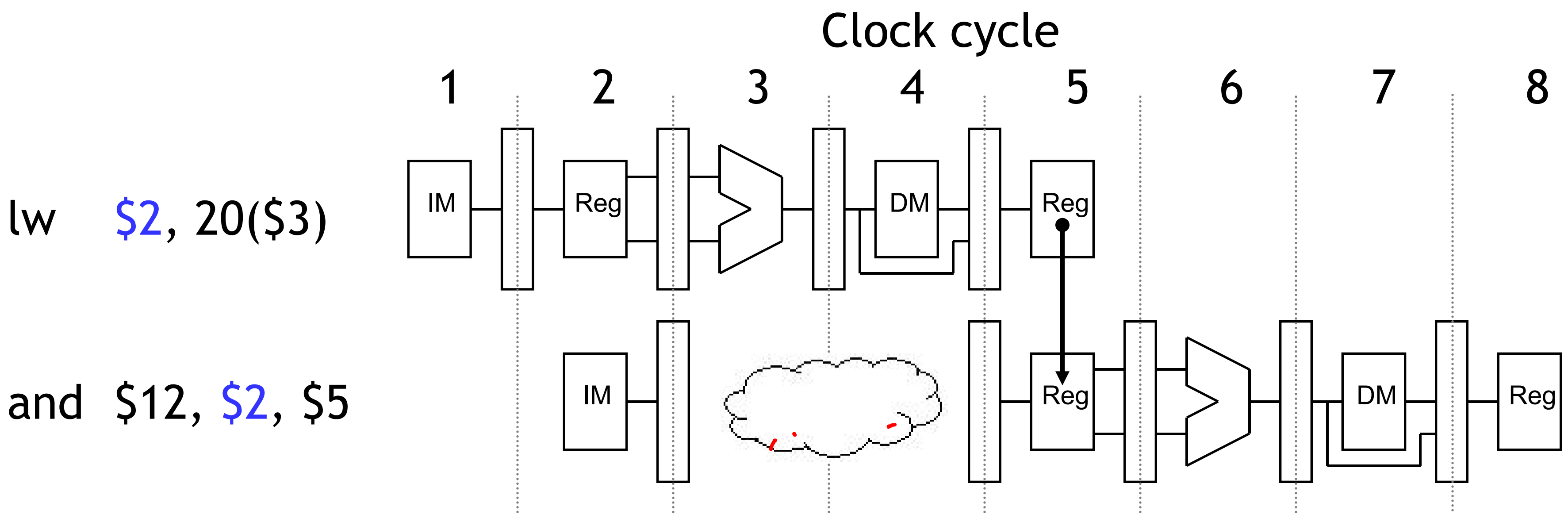
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

Stalling and forwarding

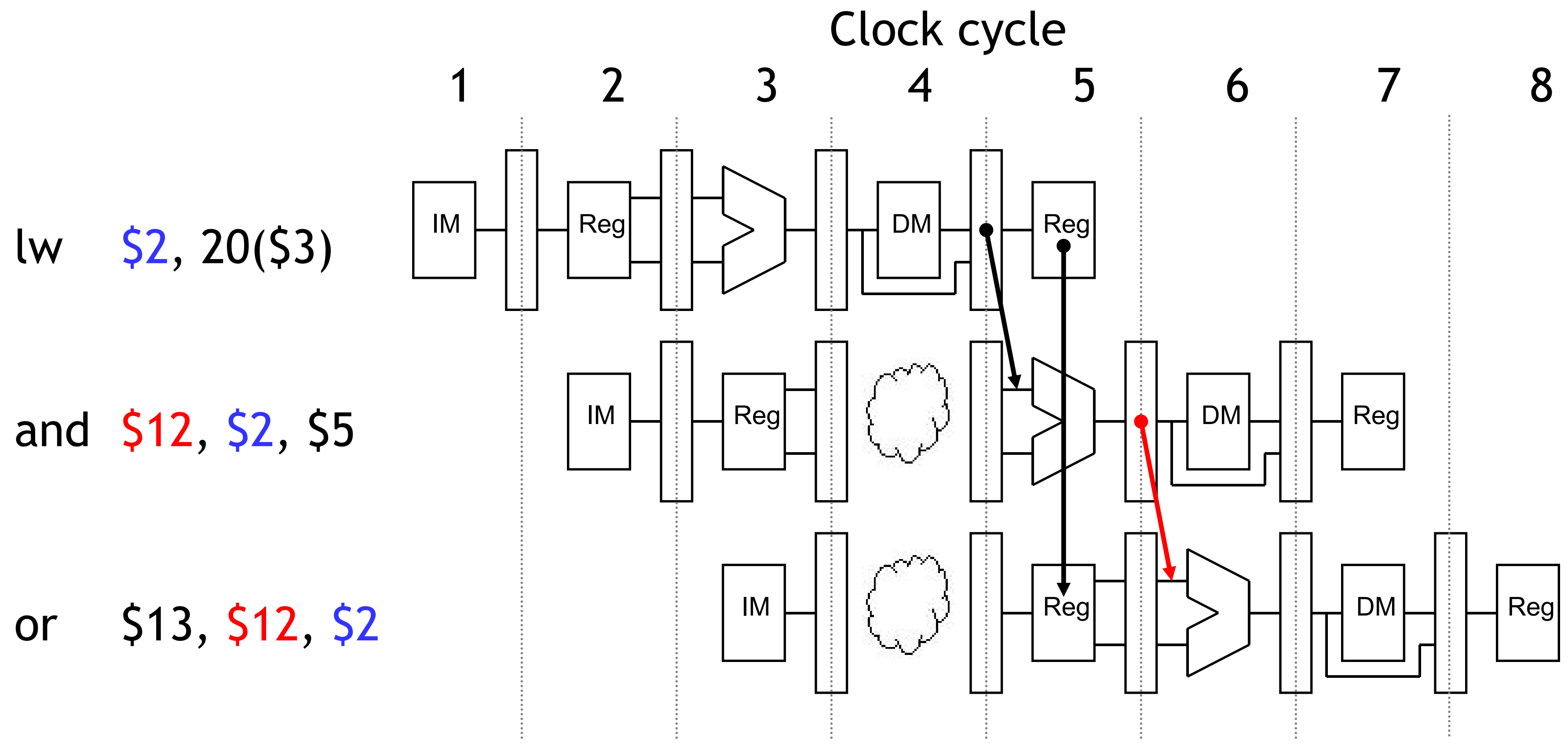
- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.

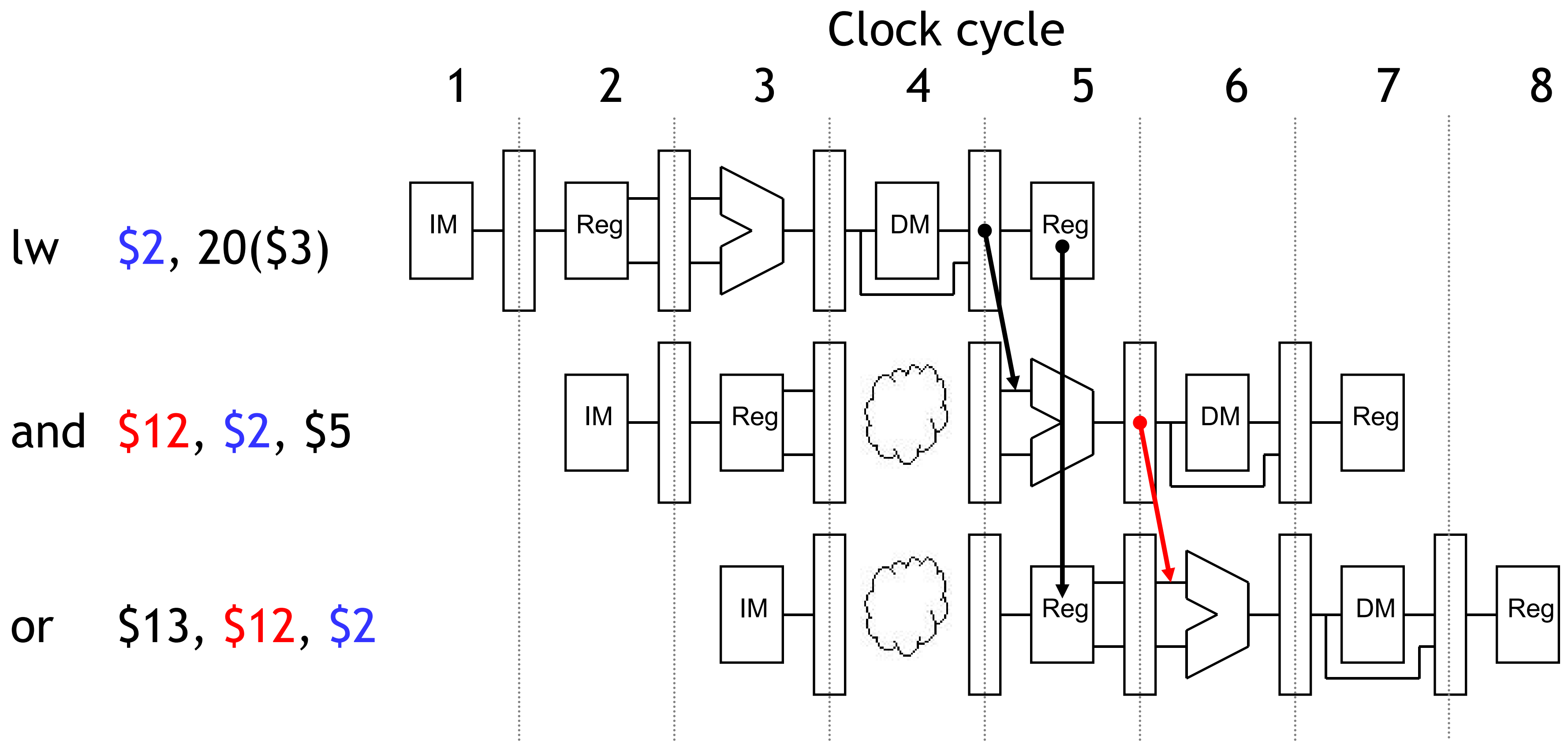
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - Why?



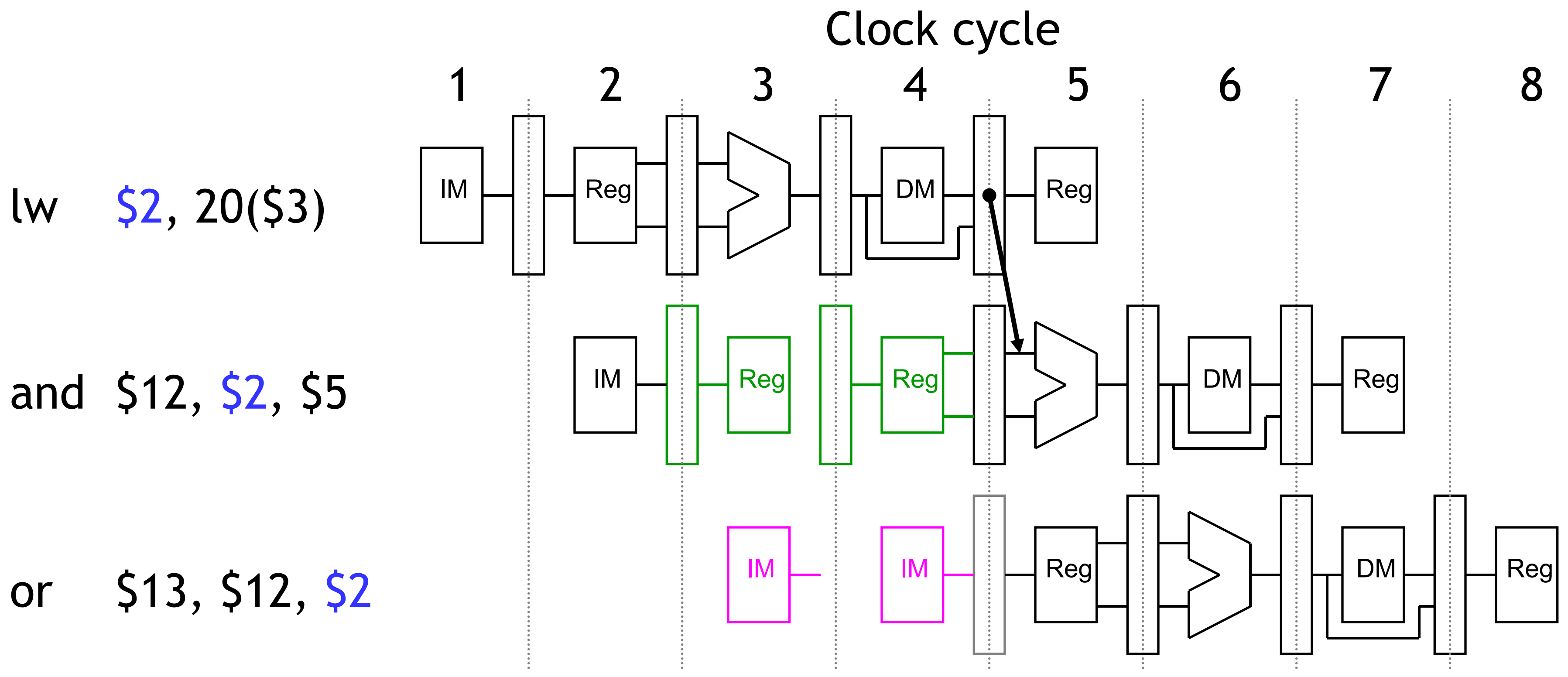
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - This is necessary to make forwarding work between AND and OR.
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle.



Implementing stalls

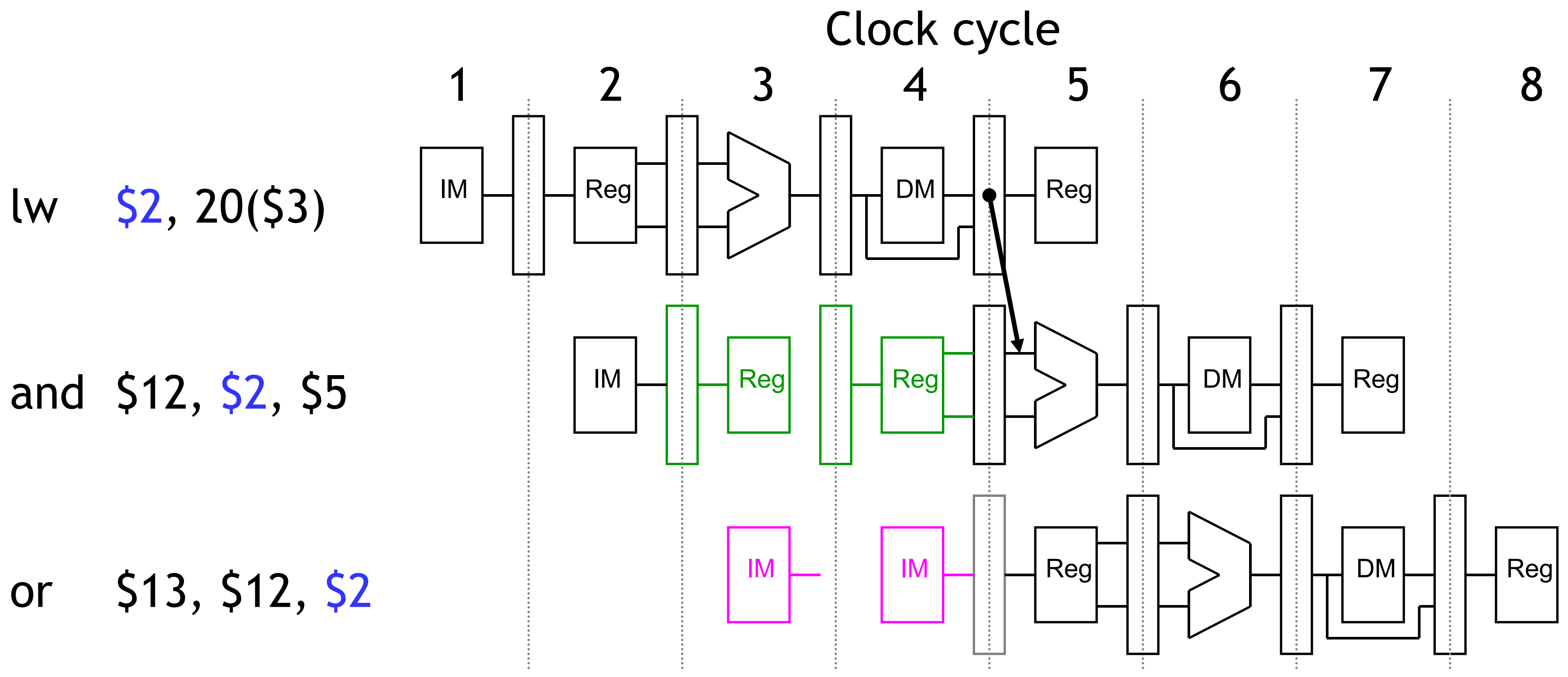
- One way to implement a stall is to force the two instructions after LW to pause and remain in their ID and IF stages for one extra cycle.



- This is easily accomplished.
  - Don't update the PC, so the current IF stage is repeated.
  - Don't update the IF/ID register, so the ID stage is also repeated.

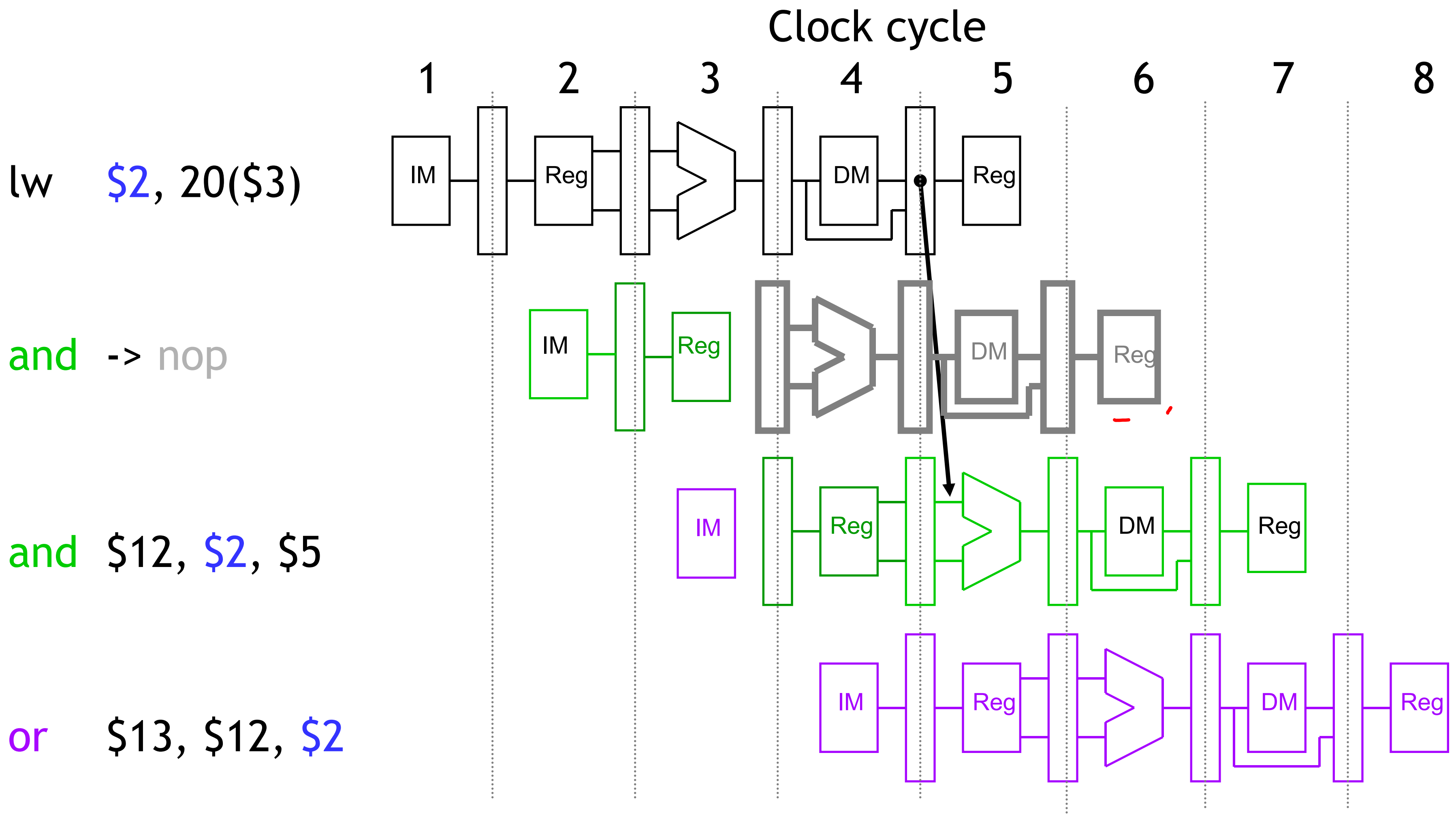
What about EXE, MEM, WB

- But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?



- Those units aren’t used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.

Stall = Nop conversion



- The effect of a load stall is to insert an empty or nop instruction into the pipeline

Detecting stalls

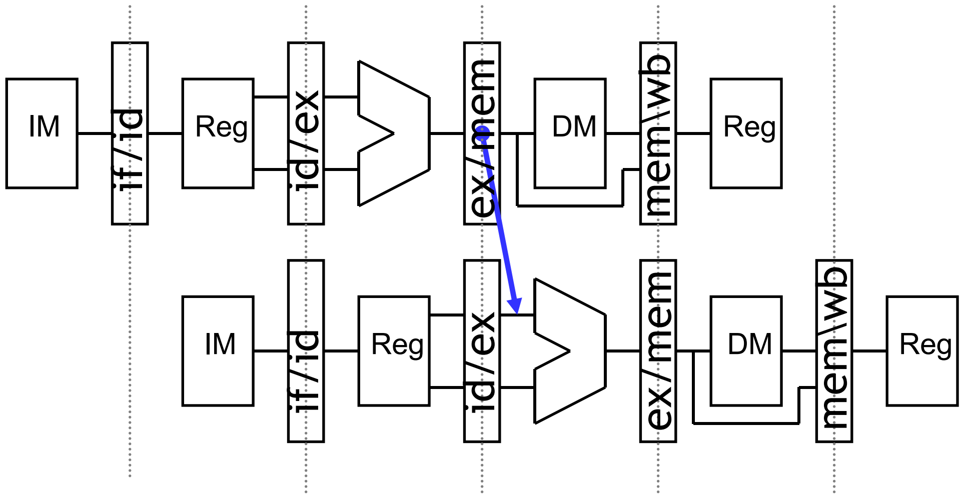
- Detecting stall is much like detecting data hazards.

- Recall the format of hazard detection equations:

  if (EX/MEM.RegWrite = 1
      and EX/MEM.RegisterRd = ID/EX.RegisterRs)
  then Bypass Rs from EX/MEM pipeline register

sub $2, $1, $3

and $12, $2, $5

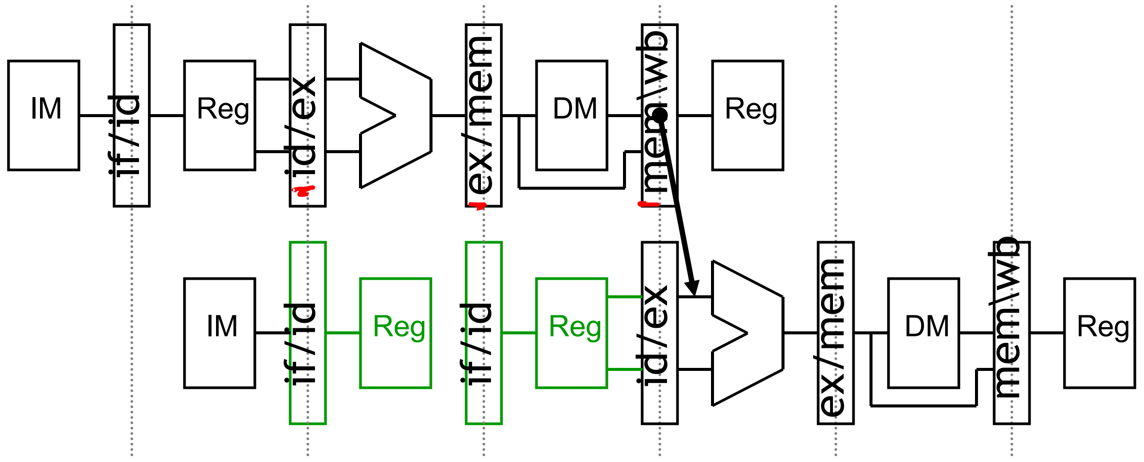


Detecting Stalls, cont.

- When should **stalls** be detected?

lw $2, 20($3)

and $12, $2, $5



- What is the stall condition?

if (condn)
then stall

Detecting stalls

- We can detect a load hazard between the current instruction in its ID stage and the previous instruction in the EX stage just like we detected data hazards.
- A hazard occurs if the previous instruction was LW...

ID/EX.MemRead = 1

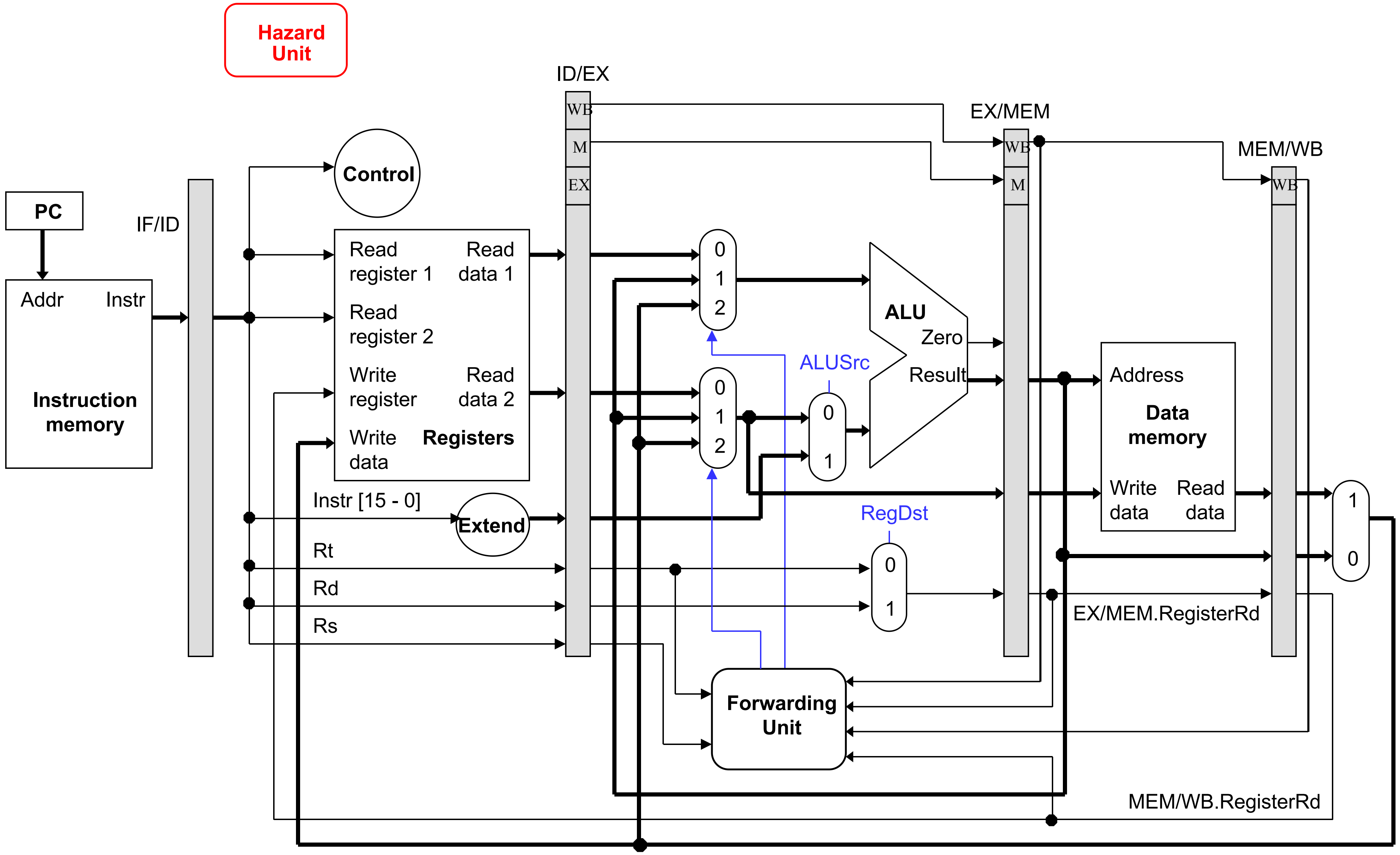
...and the LW destination is one of the current source registers.

ID/EX.RegisterRt = IF/ID.RegisterRs
or
ID/EX.RegisterRt = IF/ID.RegisterRt

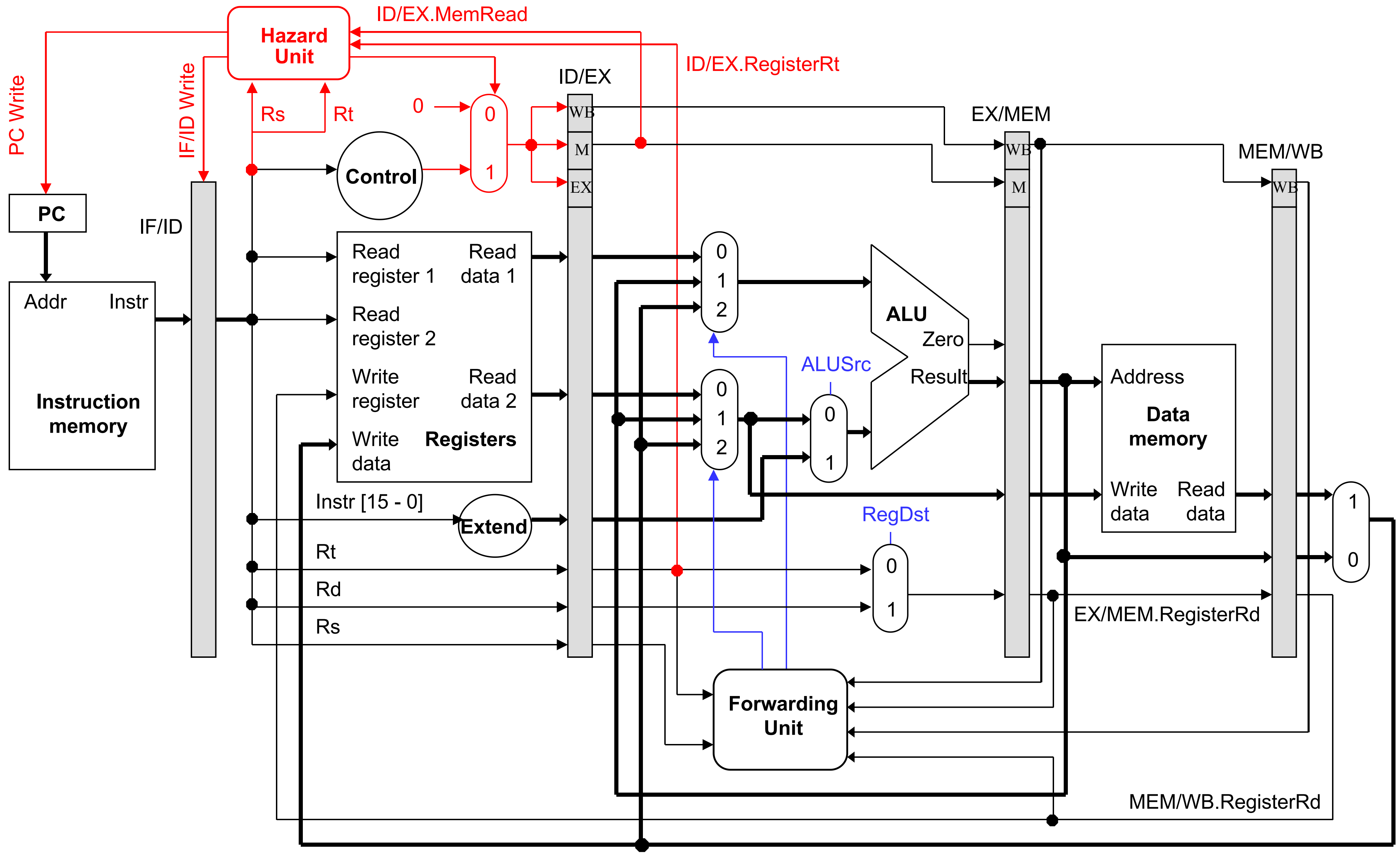
- The complete test for stalling is the conjunction of these two conditions.

```
if (ID/EX.MemRead = 1 and
    (ID/EX.RegisterRt = IF/ID.RegisterRs or
     ID/EX.RegisterRt = IF/ID.RegisterRt))
then stall
```

Adding hazard detection to the CPU



Adding hazard detection to the CPU



The hazard detection unit

- The hazard detection unit's inputs are as follows.
  - IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers for the current instruction.
  - ID/EX.MemRead and ID/EX.RegisterRt, to determine if the previous instruction is LW and, if so, which register it will write to.
- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues.
  - A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall.

Instruction Reordering

- ADD R1 , R2 , R3
- SUB R4 , R1 , R5
- XOR R8 , R6 , R7
- AND R9 , R10 , R11

Before

- ADD R1 , R2 , R3
- XOR R8 , R6 , R7
- AND R9 , R10 , R11
- SUB R4 , R1 , R5

After

Instruction Reordering

Before

```
                        1   2  3    4    5      6      7    8       9
▪ LW $2,4($10)     IF ID EX MEM WB
▪ LW $3,8($11)        IF ID  EX   MEM  WB
▪ add $4,$2,$3          IF   ID     S       EX MEM  WB
▪ LW $5,12($1)              IF     S       ID  EX    MEM  WB
```

After

```
                          1 2   3    4      5     6      7     8
▪ LW $2,4($10)      IF ID EX MEM   WB
▪ LW $3,8($11)        IF ID   EX    MEM  WB
▪ LW $5,12($1)          IF   ID      EX    MEM   WB
▪ add $4,$2,$3              IF     ID      EX    MEM WB
```