A more detailed look at the pipeline

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

	Clock cycle						
	1	2	3	4	5	6	7
sub $2, $1, $3	IF	ID	EX	MEM	WB		
and $12, $2, $5		IF	ID	EX	MEM	WB	
or $13, $6, $2			IF	ID	EX	MEM	WB

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $2.
- When is the data is actually produced and consumed?
- What can we do?

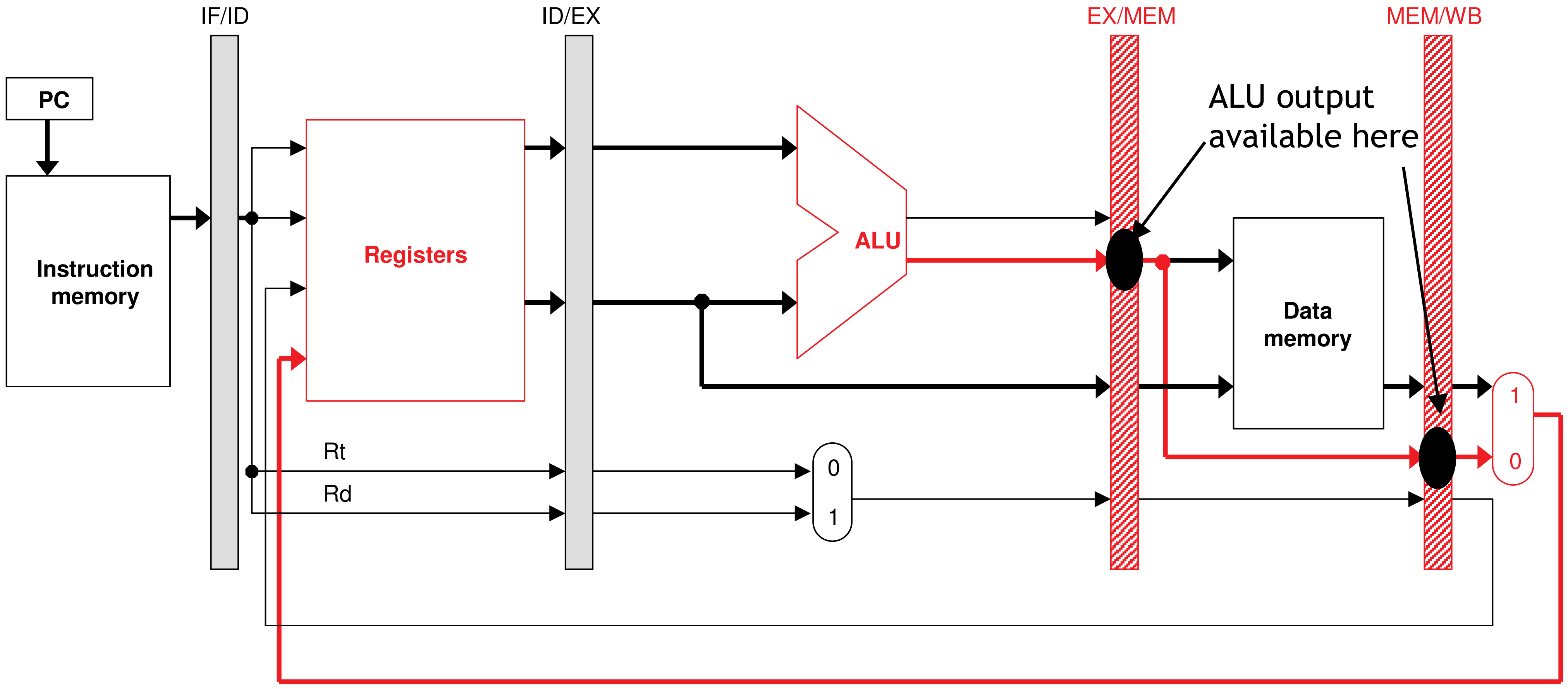
Bypassing the register file

- The actual result $1 – $3 is computed in clock cycle 3, *before* it’s needed in cycles 4 and 5.
- If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
  - Today we’ll focus on hazards involving arithmetic instructions.
  - We’ll also examine a complication with the lw instruction.
- Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.

	Clock cycle						
	1	2	3	4	5	6	7
sub $2, $1, $3	IF	ID	EX	MEM	WB		
and $12, $2, $5		IF	ID	EX	MEM	WB	
or $13, $6, $2			IF	ID	EX	MEM	WB

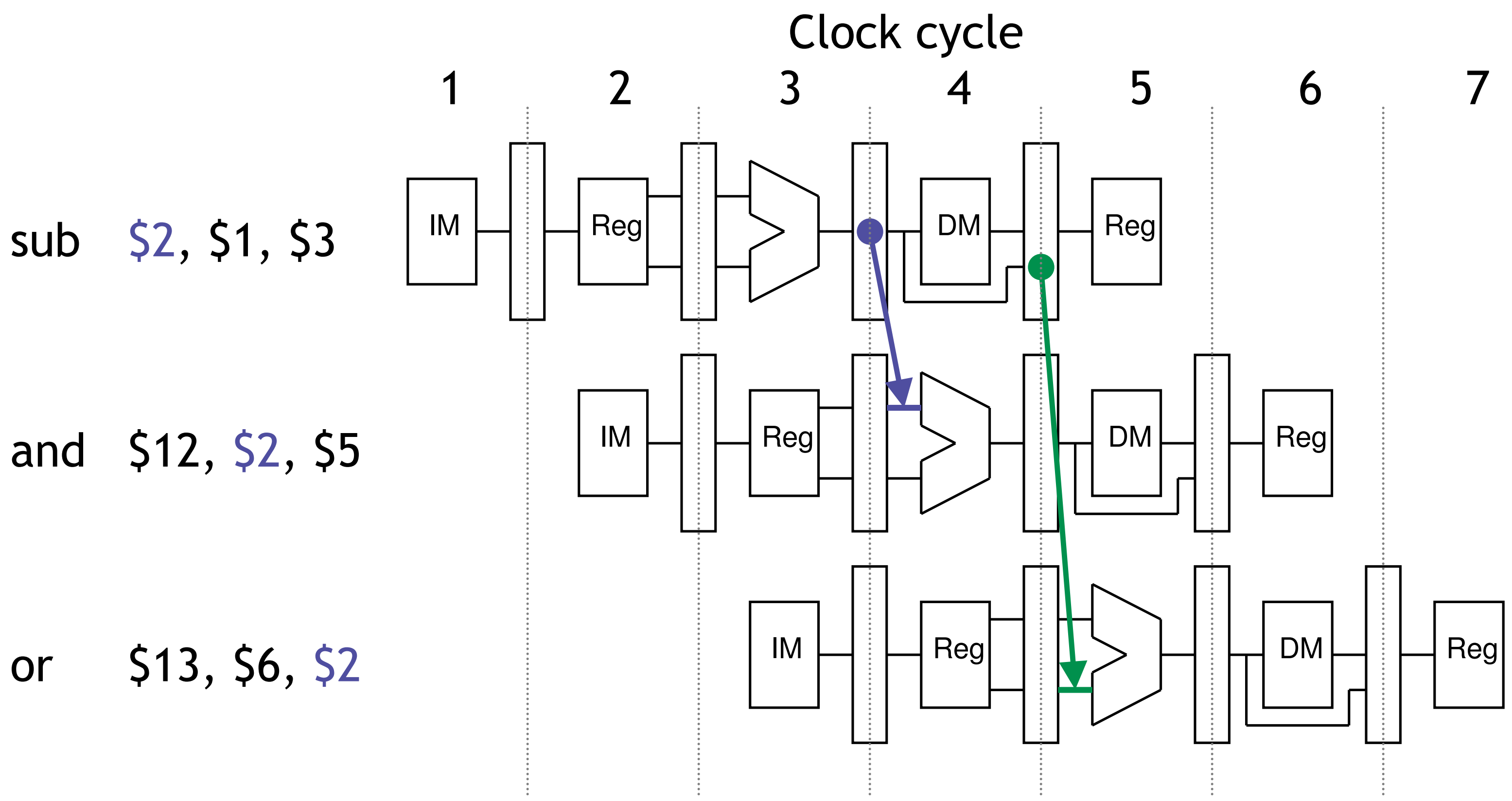
Where to find the ALU result

- The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.
- This is an abridged diagram of our pipelined datapath.



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.



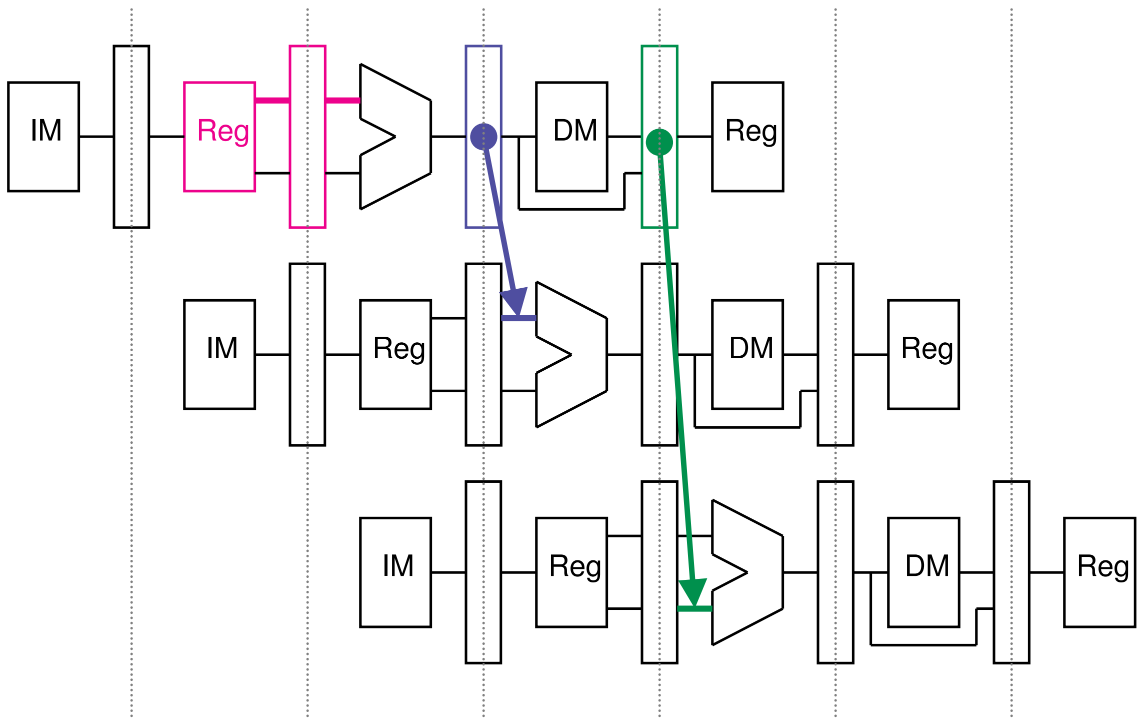
Outline of forwarding hardware

- 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.

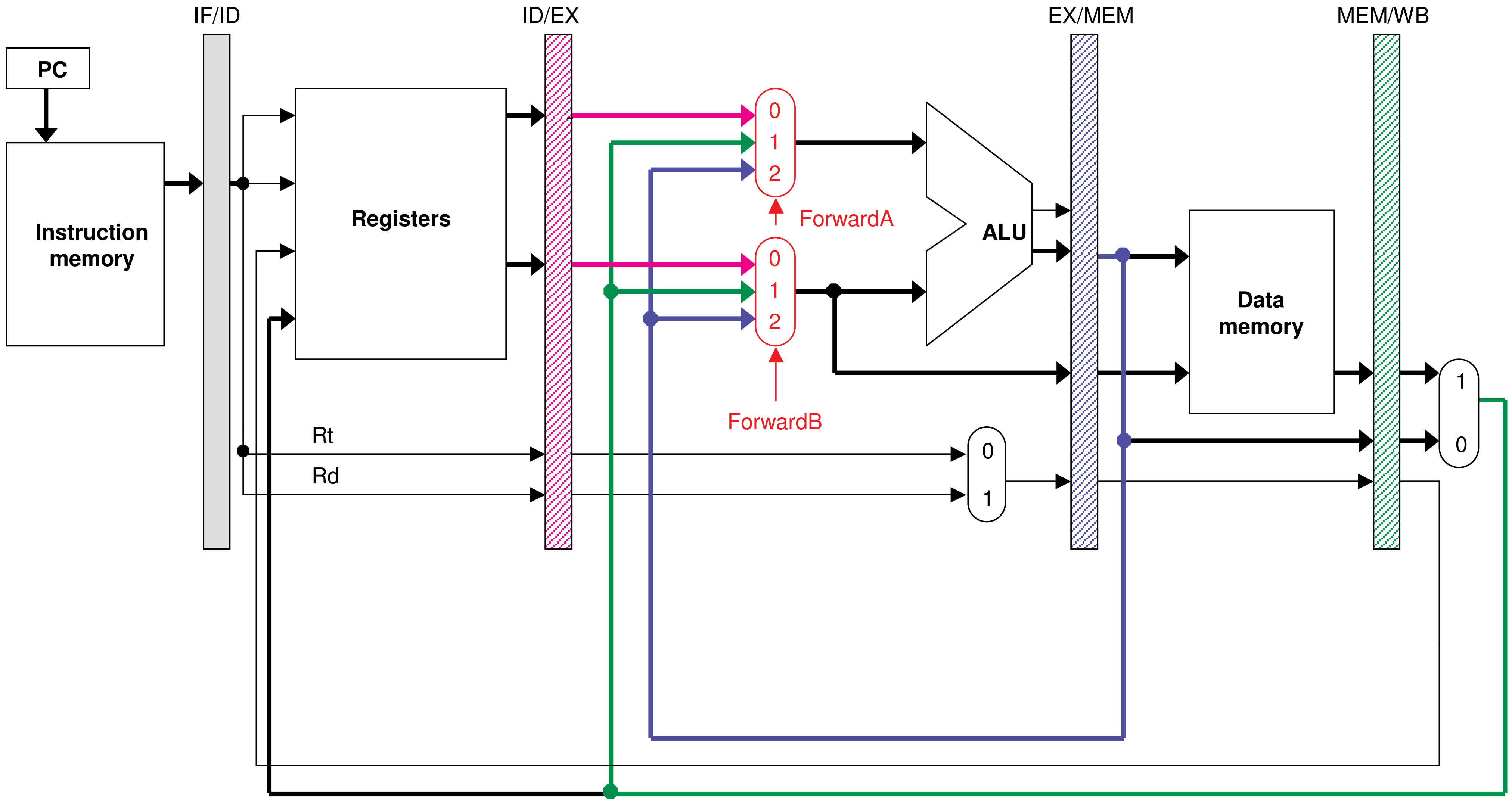
sub $2, $1, $3

and $12, $2, $5

or $13, $6, $2

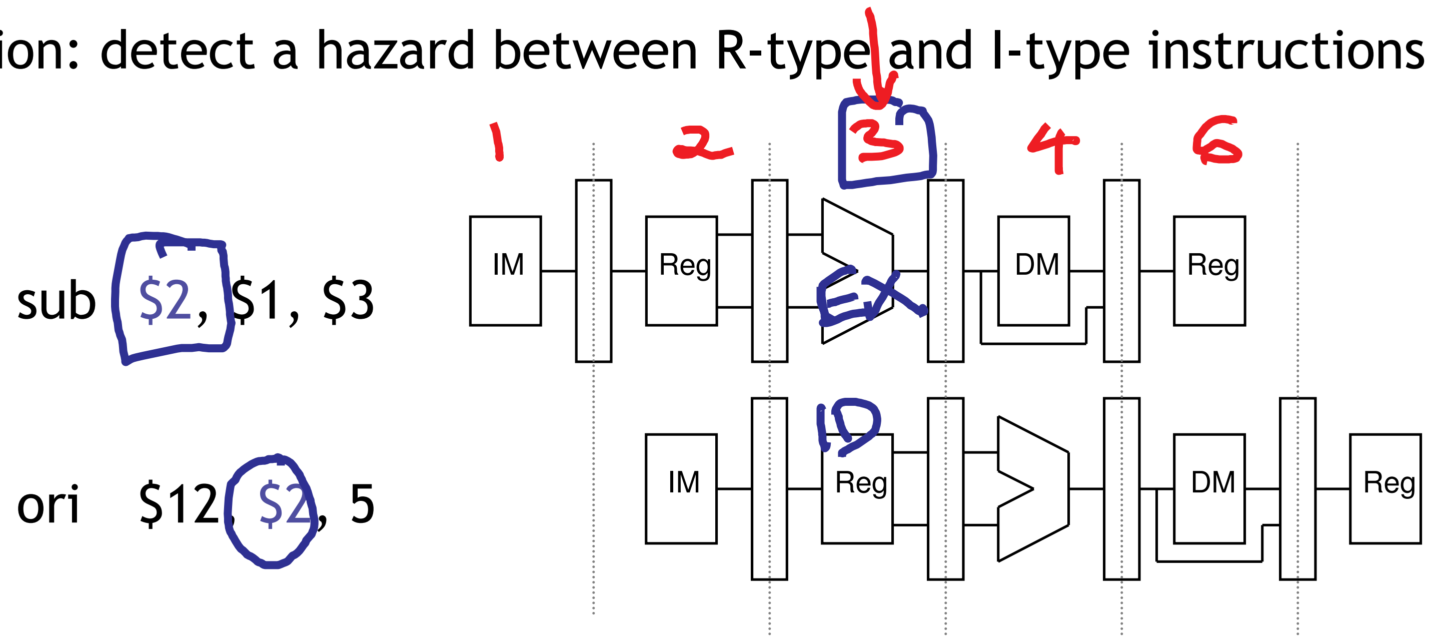


Simplified datapath with forwarding muxes



Detecting EX/MEM Hazards

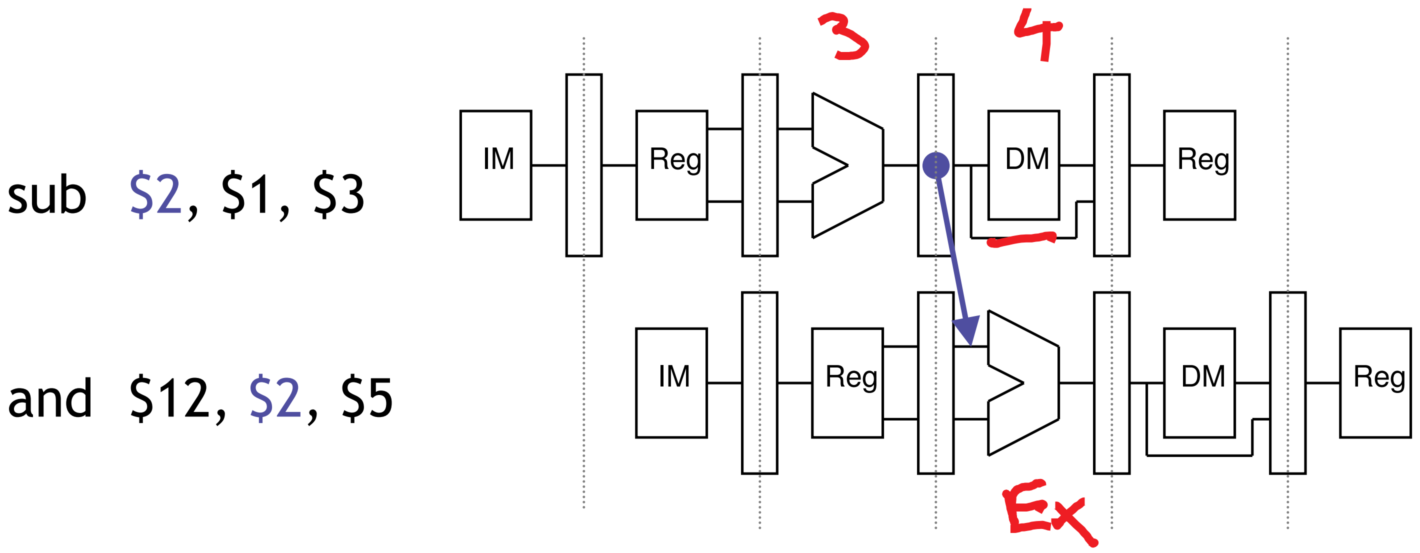
- Section: detect a hazard between R-type and I-type instructions



- Can't detect the hazard until cycle 3: `sub` is in EX, `ori` is in ID
- Hazard because: ID/EX.RegisterRd = IF/ID.RegisterRs
- In MP 5 (Java) terminology: `instr[EX].rd() == instr[ID].rs()`

Eliminating EX/MEM data hazards

- When do we need to know that a hazard exists?
- So how can the hardware determine if a hazard exists?
- An EX/MEM hazard occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, *and*
  2. The destination is one of the ALU source registers in the EX stage.



- Data in a pipeline register can be referenced using a class-like syntax. For example, ID/EX.RegisterRt refers to the rt field stored in the ID/EX pipeline.

EX/MEM data hazard equations

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

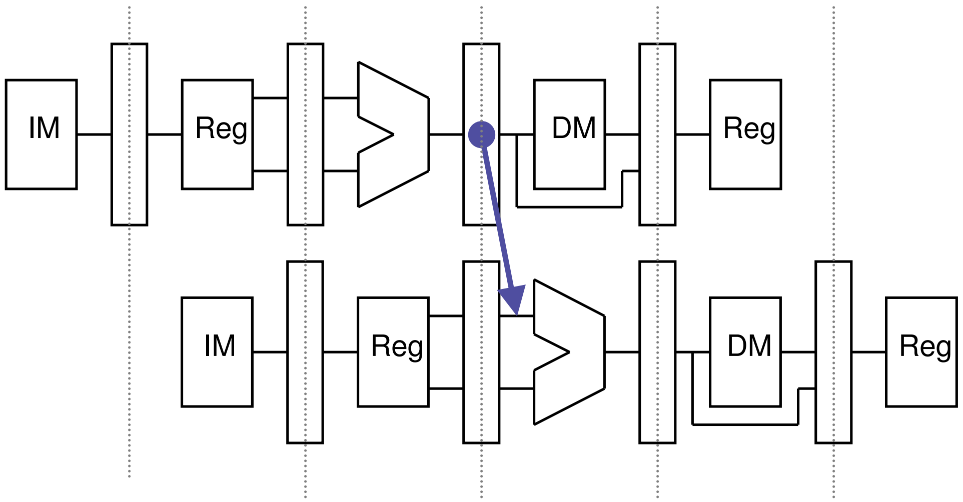
```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then ForwardA = 2
```

- The second ALU source is similar.

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

sub $2, $1, $3

and $12, $2, $5



MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

```
add  $1, $2, $3
add  $1, $1, $4
sub  $5, $5, $1
```

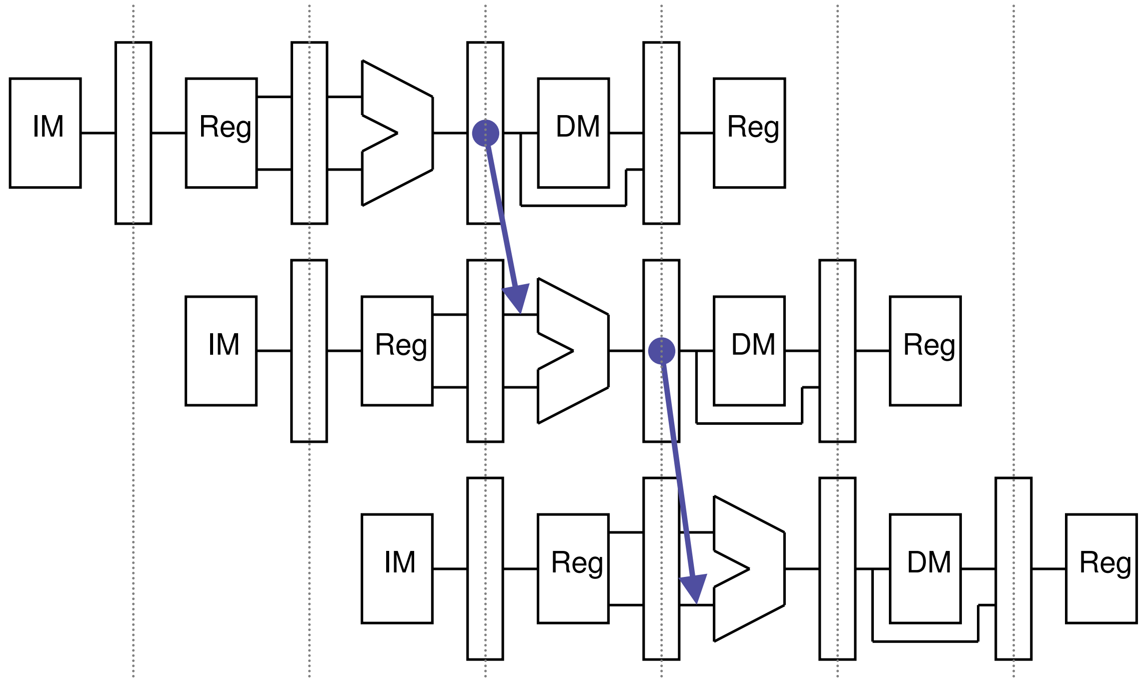
NOT data hazard

- Register $1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



MEM/WB hazard equations

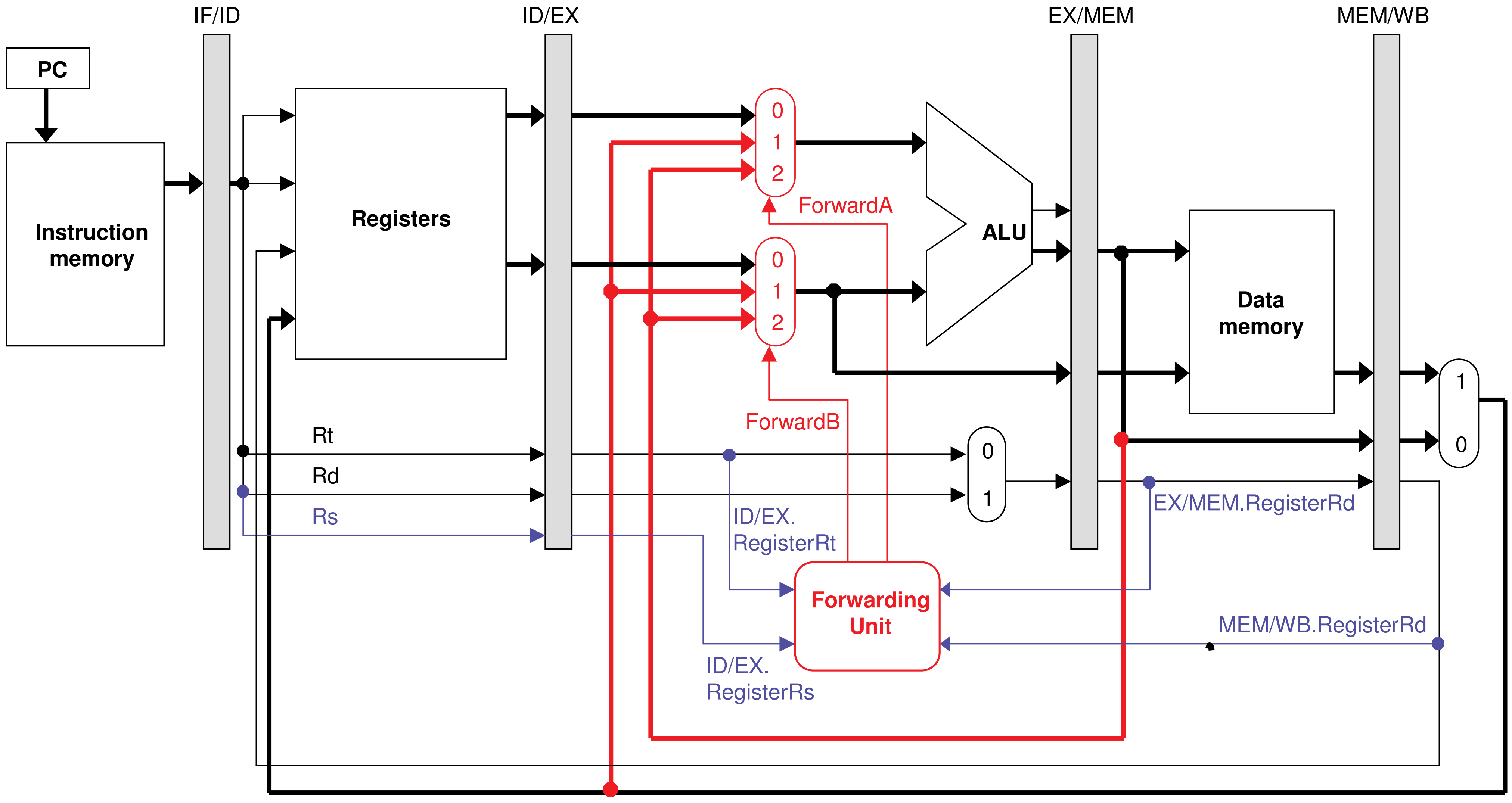
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

```
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
```

Simplified datapath with forwarding



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 two RegWrite signals are not shown in the diagram, but they come from the control unit.)
- The fowarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

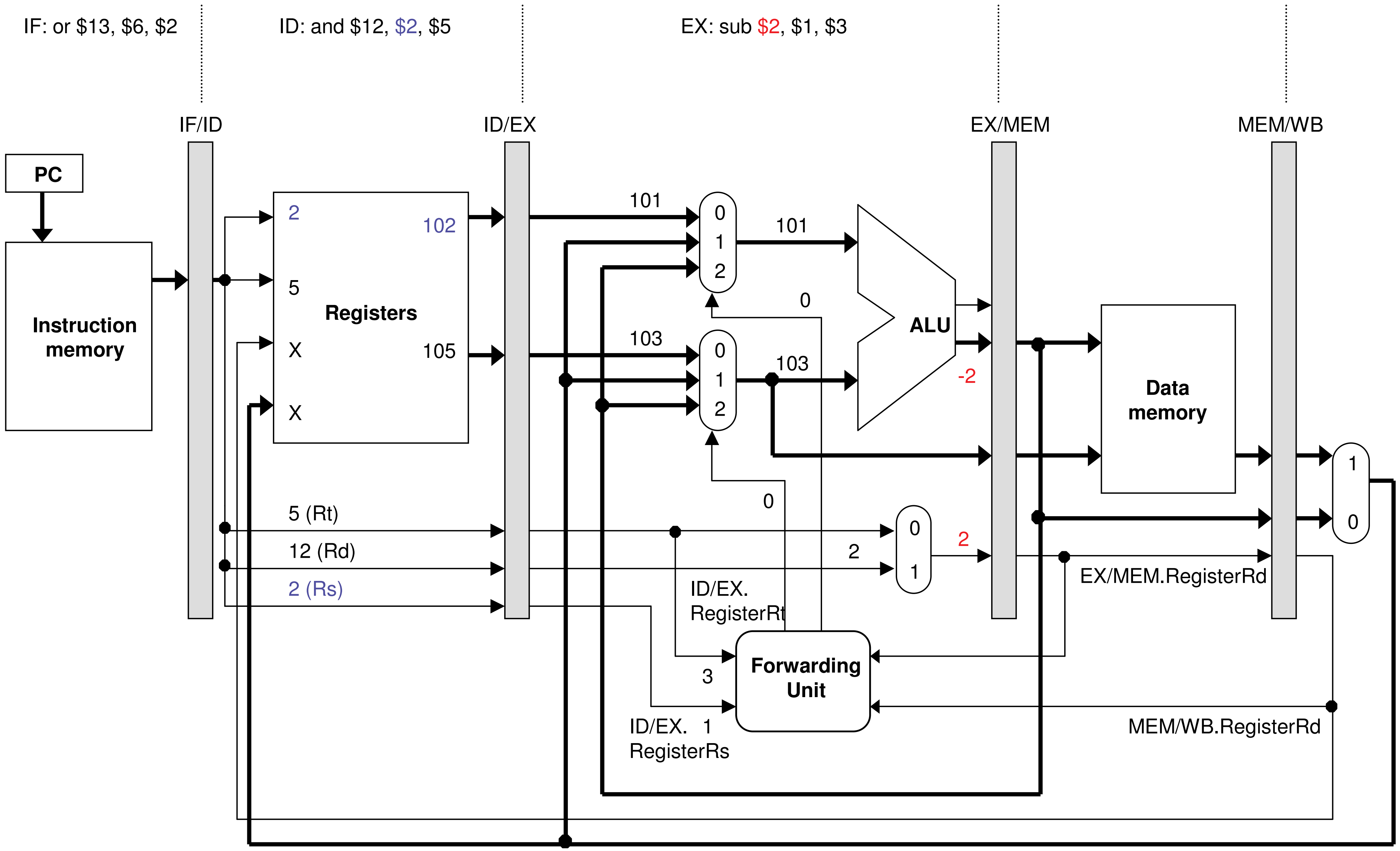
Example

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

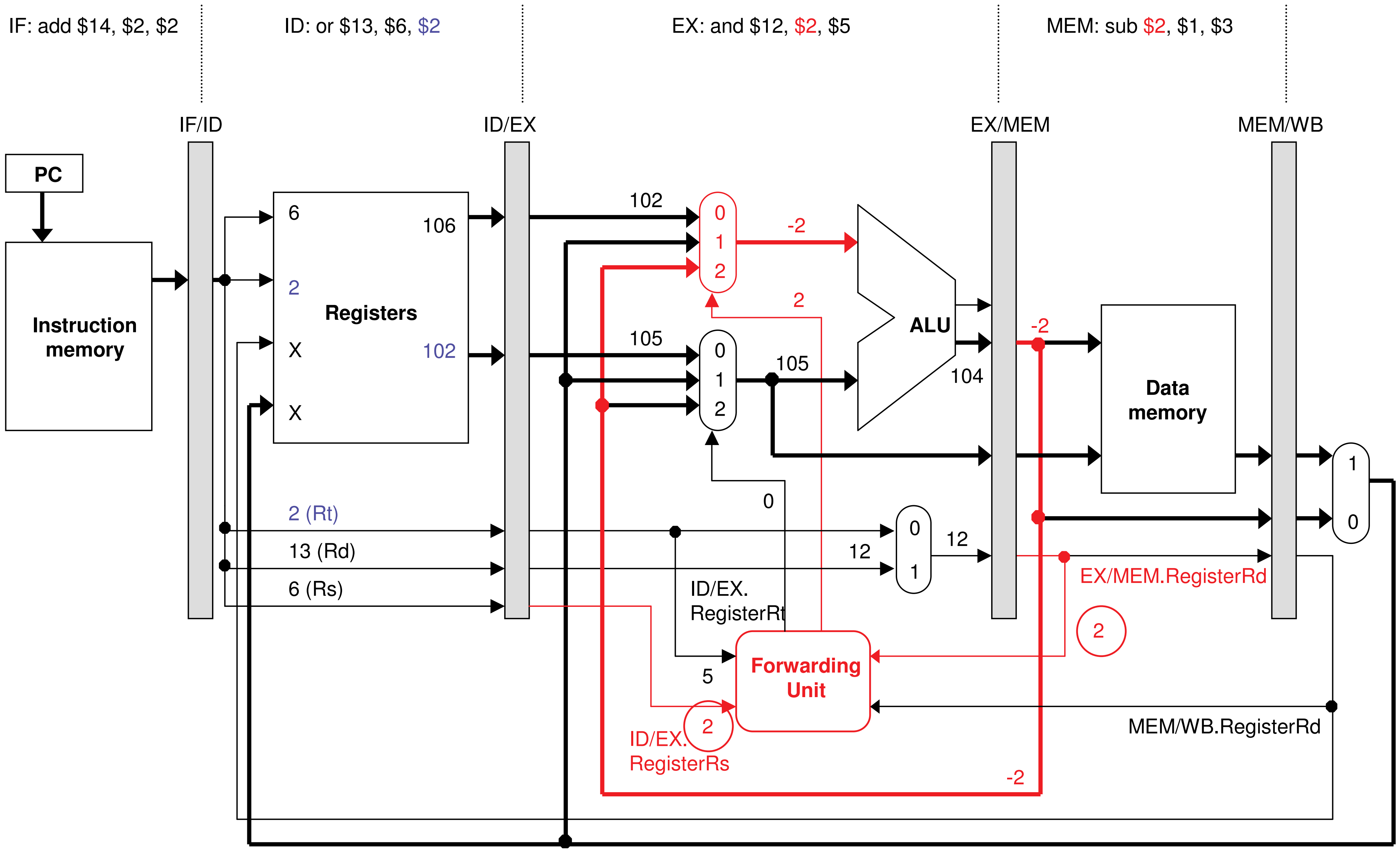
- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain −2 (101 − 103).
  - The other instructions should all use −2 as one of their operands.

- 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.

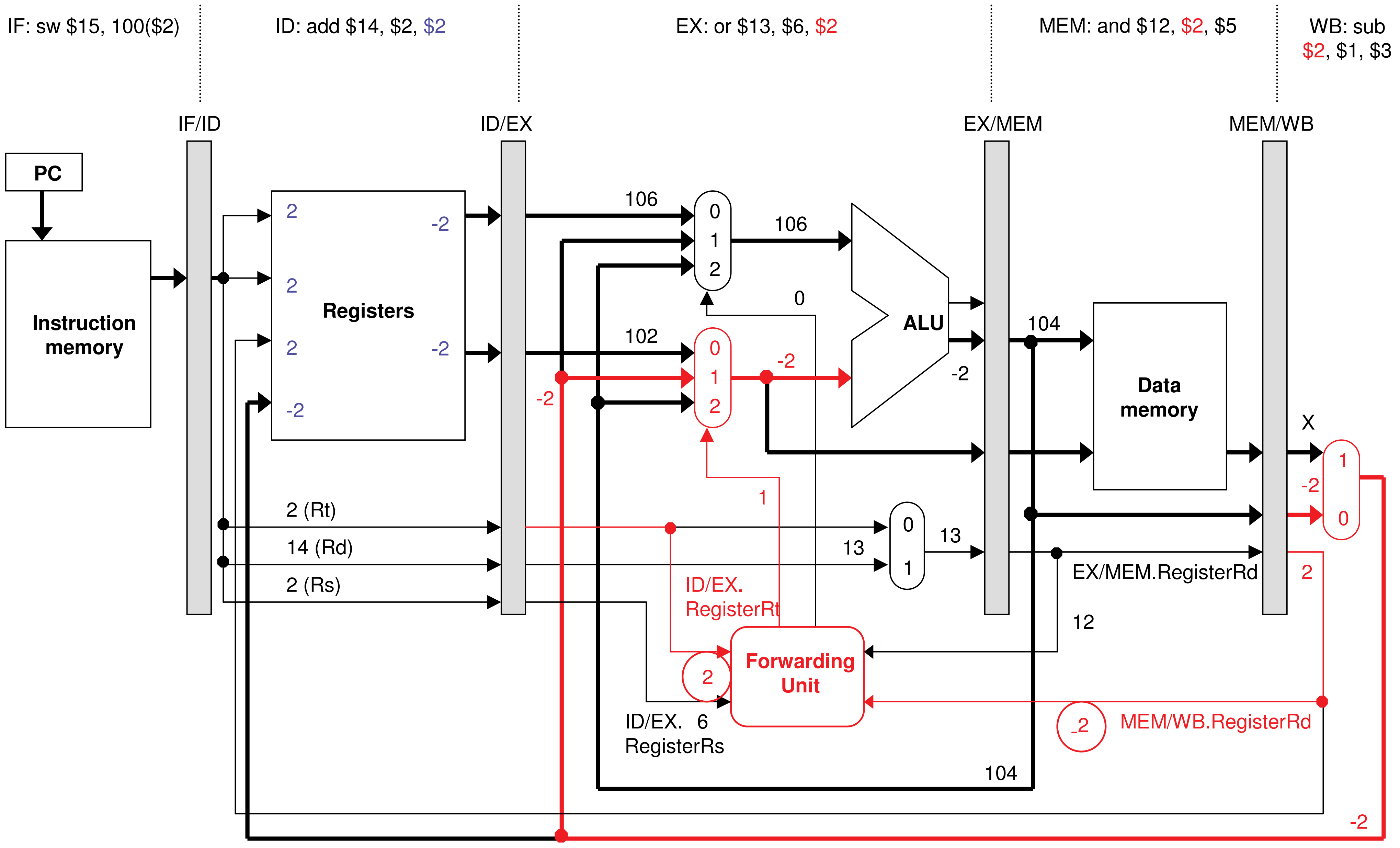
Clock cycle 3



Clock cycle 4: forwarding $2 from EX/MEM



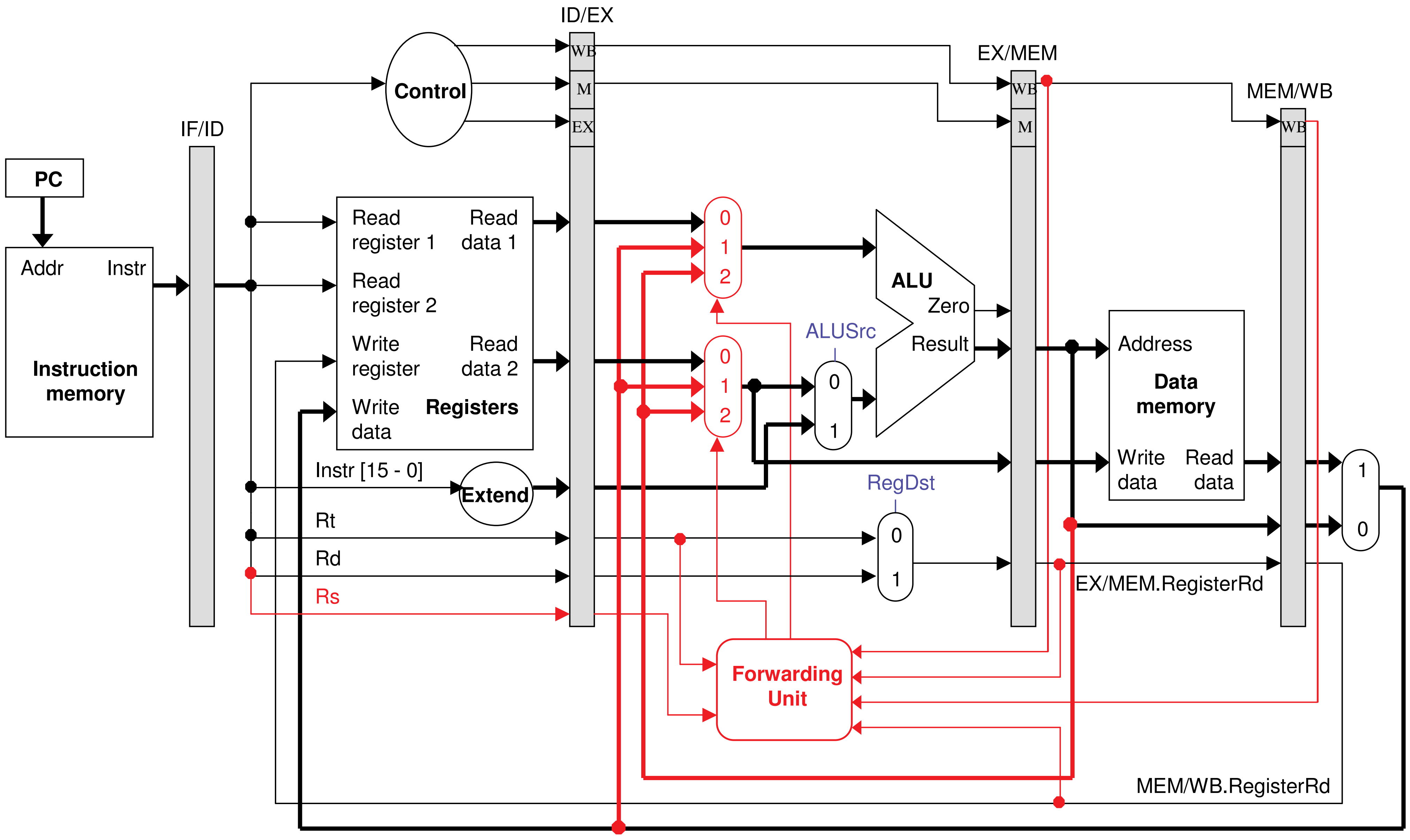
Clock cycle 5: forwarding $2 from MEM/WB



Lots of data hazards

- The first data hazard occurs during cycle 4.
  - The forwarding unit notices that the ALU's first source register for the AND is also the destination of the SUB instruction.
  - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
- A second hazard occurs during clock cycle 5.
  - The ALU's second source (for OR) is the SUB destination again.
  - This time, the value has to be forwarded from the MEM/WB pipeline register instead.
- There are no other hazards involving the SUB instruction.
  - During cycle 5, SUB writes its result back into register $2.
  - The ADD instruction can read this new value from the register file in the same cycle.

Complete pipelined datapath...so far

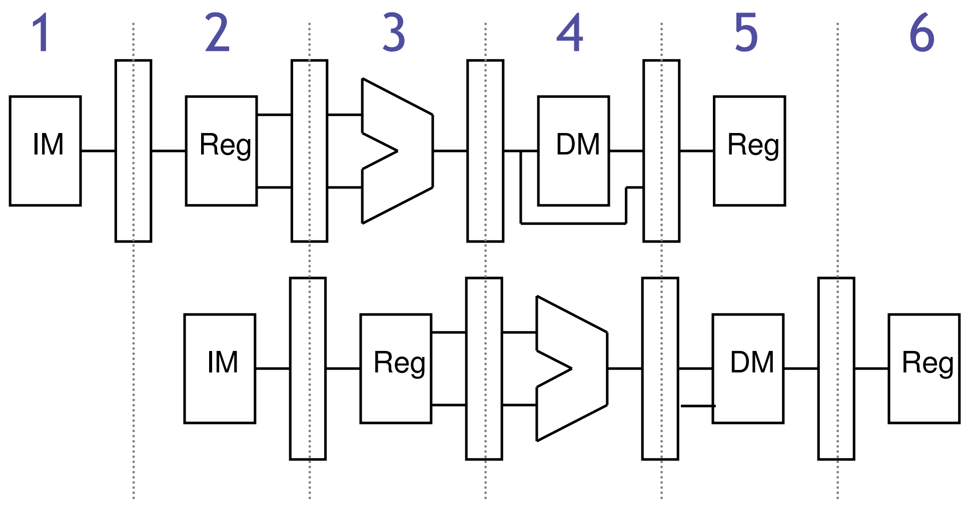


What about stores?

- Two “easy” cases:

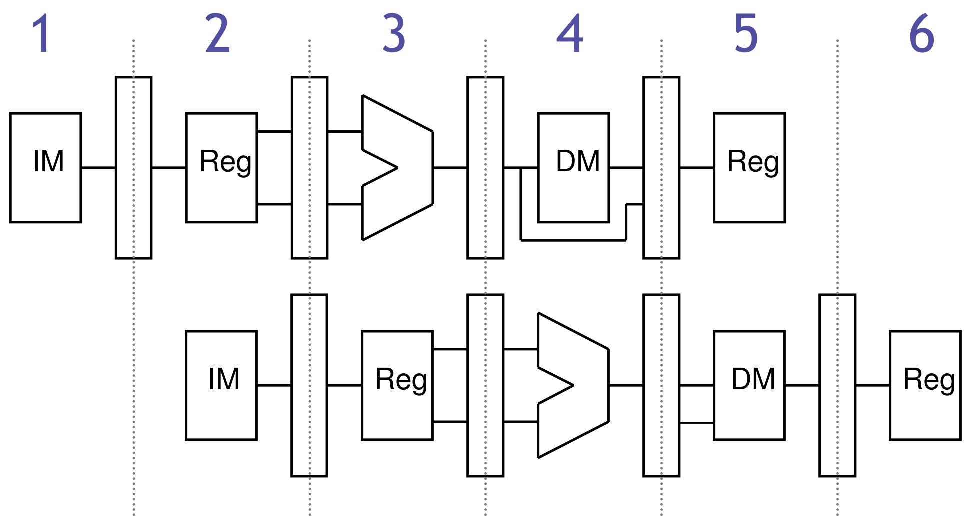
add $1, $2, $3

sw $4, 0($1)

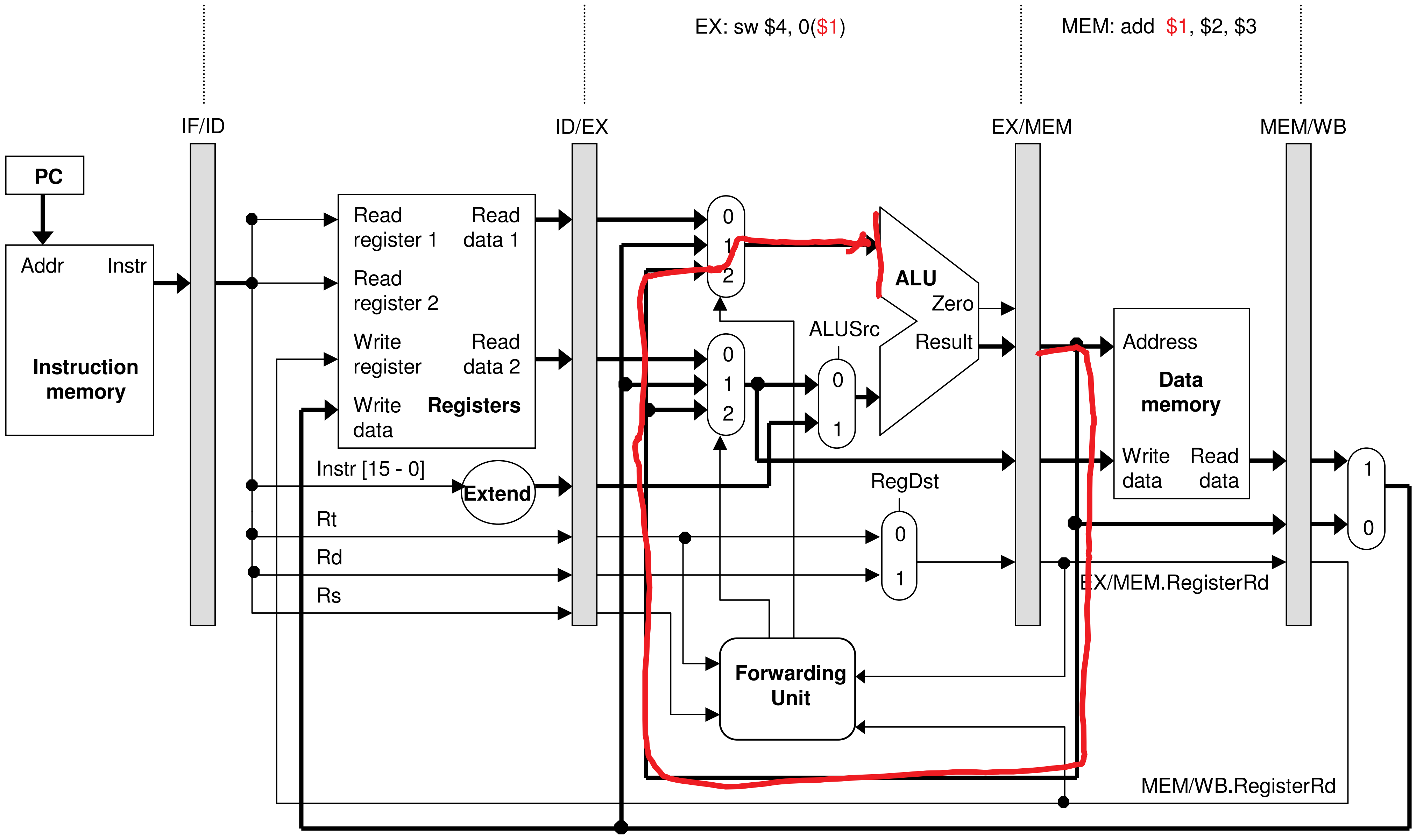


add $1, $2, $3

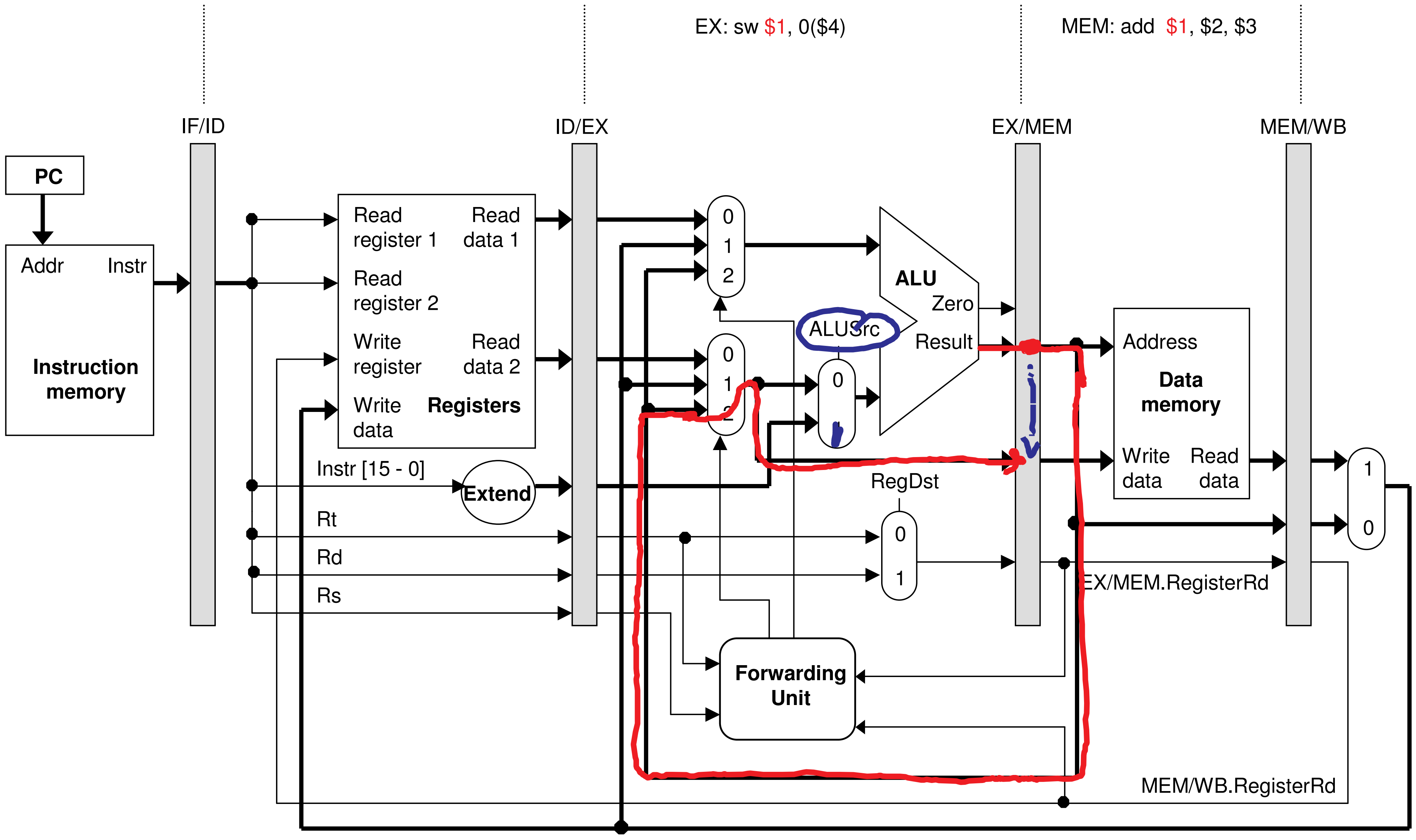
sw $1, 0($4)



Store Bypassing: Version 1



Store Bypassing: Version 2

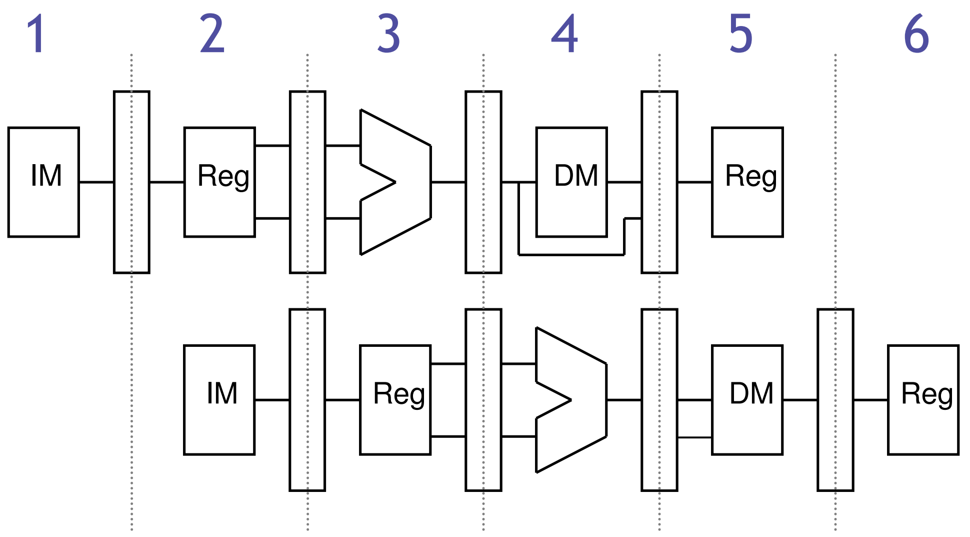


What about stores?

- A harder case:

lw $1, 0($2)

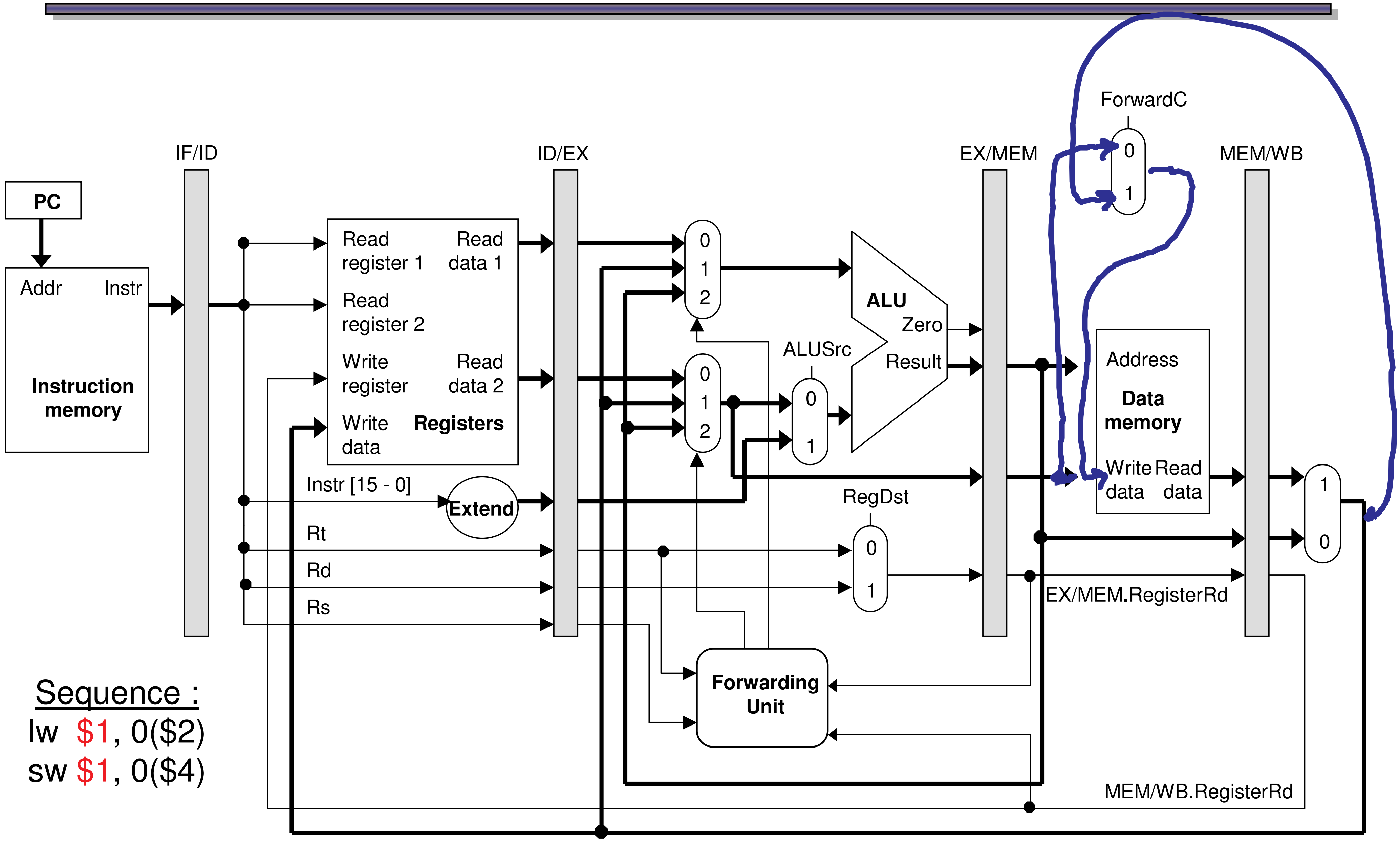
sw $1, 0($4)



- In what cycle is:
  - The load value available? End of cycle 4
  - The store value needed? Start of cycle 5

- What do we have to add to the datapath?

Load/Store Bypassing: Extend the Datapath



Sequence :

lw $1, 0($2)

sw $1, 0($4)

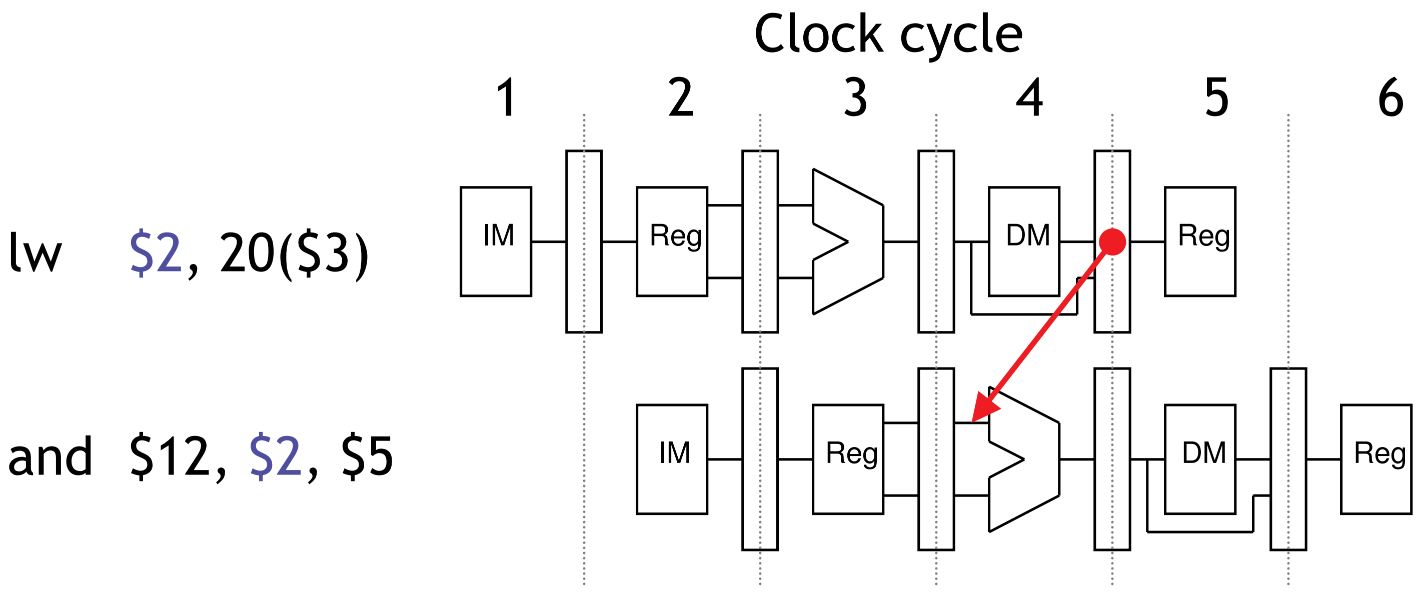
Miscellaneous comments

- Each MIPS instruction writes to at most one register.
  - This makes the forwarding hardware easier to design, since there is only one destination register that ever needs to be forwarded.
- Forwarding is especially important with deep pipelines like the ones in all current PC processors.
- Section 6.4 of the textbook has some additional material not shown here.
  - Their hazard detection equations also ensure that the source register is not $0, which can never be modified.
  - There is a more complex example of forwarding, with several cases covered. Take a look at it!

- For MP 5:
  - If `hasForwarding`, “hazards” that can be eliminated by forwarding should not be identified

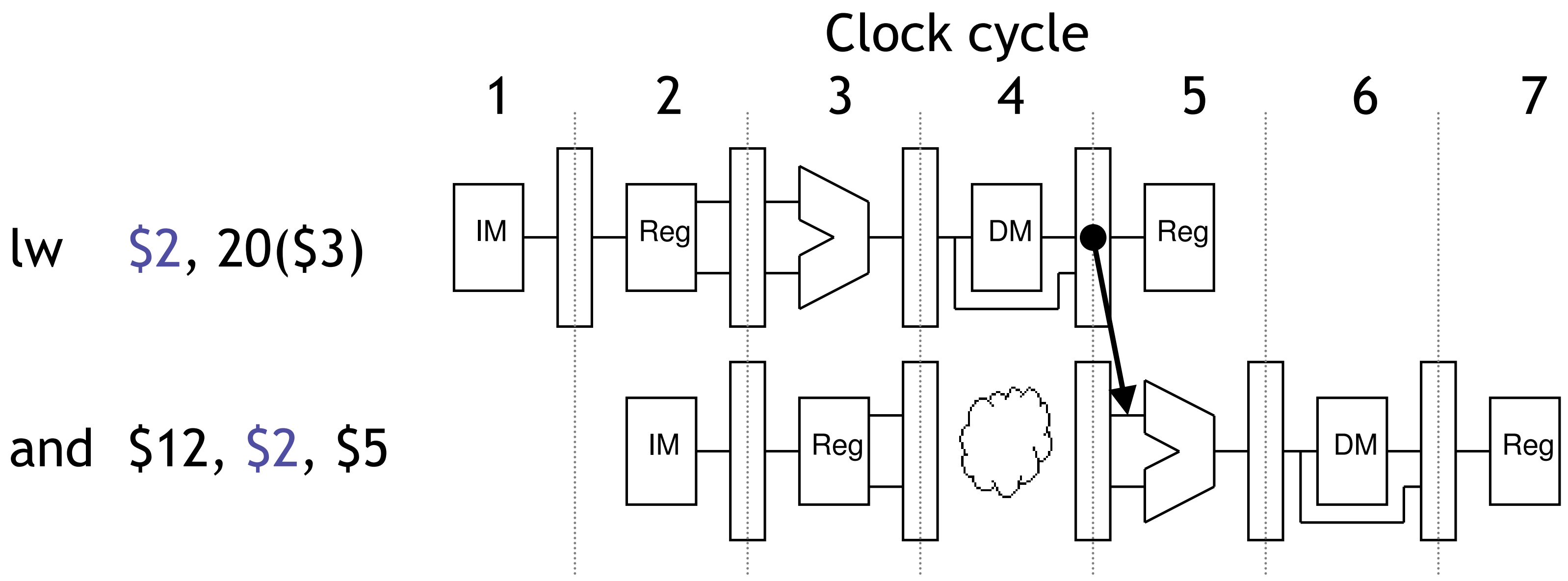
What about loads?

- Consider the instruction sequence shown below.
  - 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

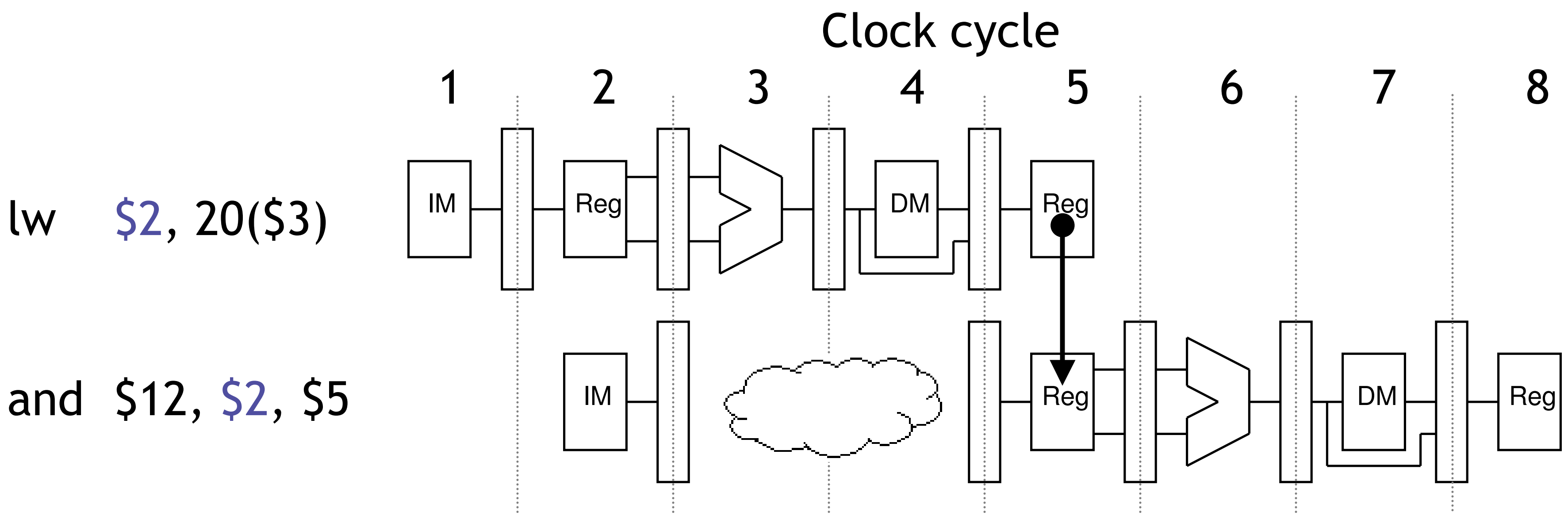
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- 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.