Forwarding

233 in one slide!

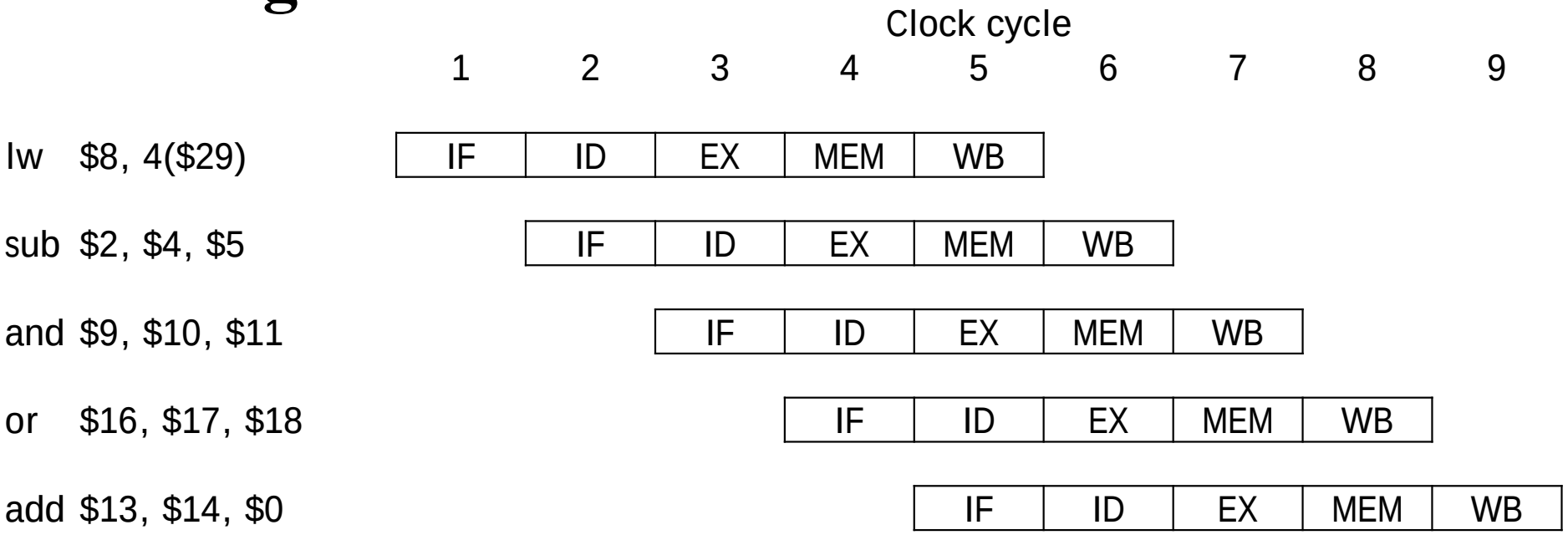
- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
  1. You will build a simple computer processor
     Build and create **state** machines with **data**, **control**, and **indirection**
  2. You will learn how high-level language code executes on a processor
     Time limitations create **dependencies** in the **state** of the processor
  3. You will learn why computers perform the way they do
     Physical limitations require **locality** and **indirection** in how we access **state**
  4. You will learn about hardware mechanisms for parallelism
     Locality, dependencies, and **indirection** on performance enhancing drugs

- We will have a SPIMbot contest!

Today's Lecture

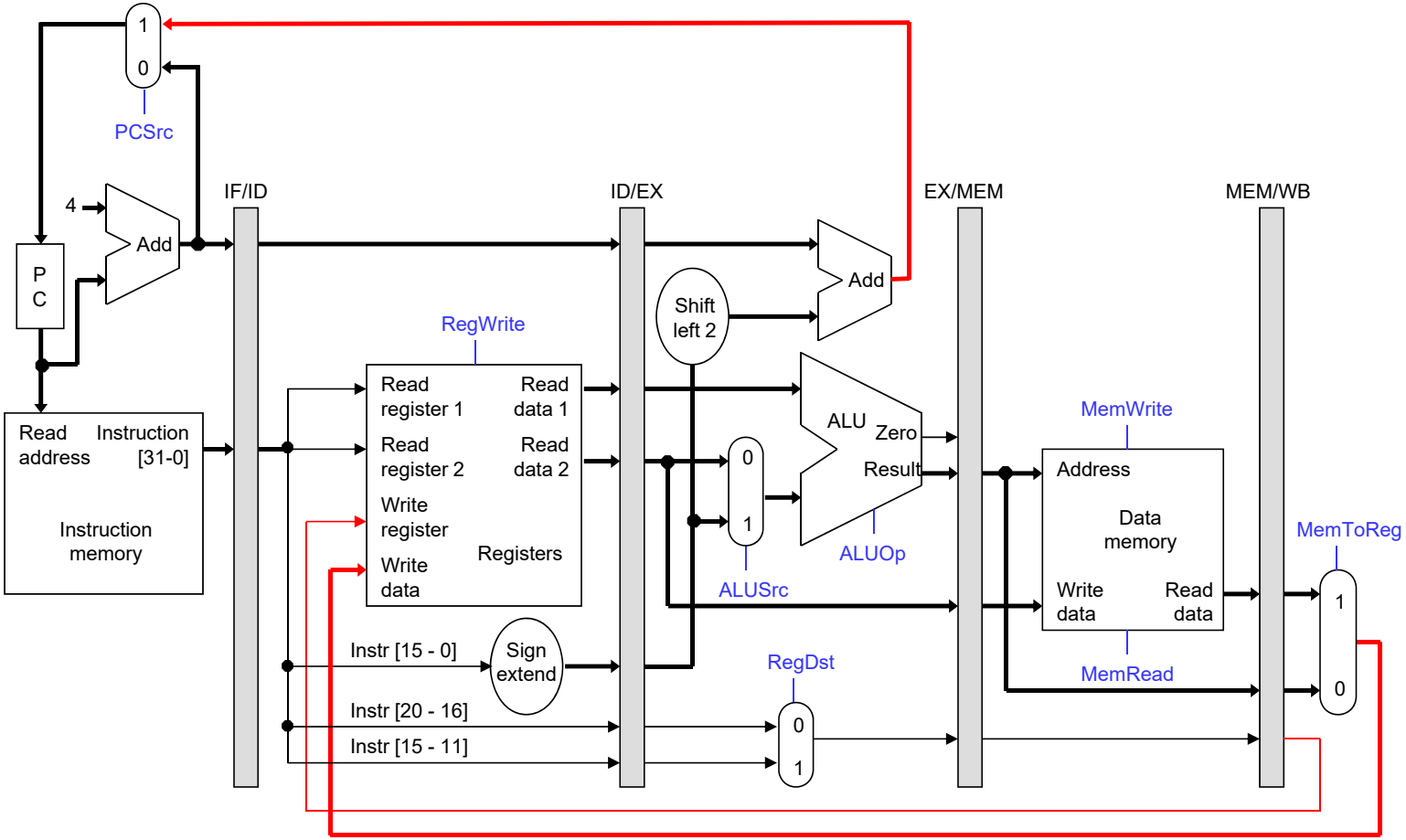
- Why dependencies and delayed feedback necessitate forwarding
- How to implement forwarding

This diagram shows the execution of an ideal code fragment.



- Each instruction needs a total of five cycles for execution.
- One instruction begins on every clock cycle for the first five cycles.
- One instruction completes on each cycle from that time on.

Note how everything goes left to right, except …



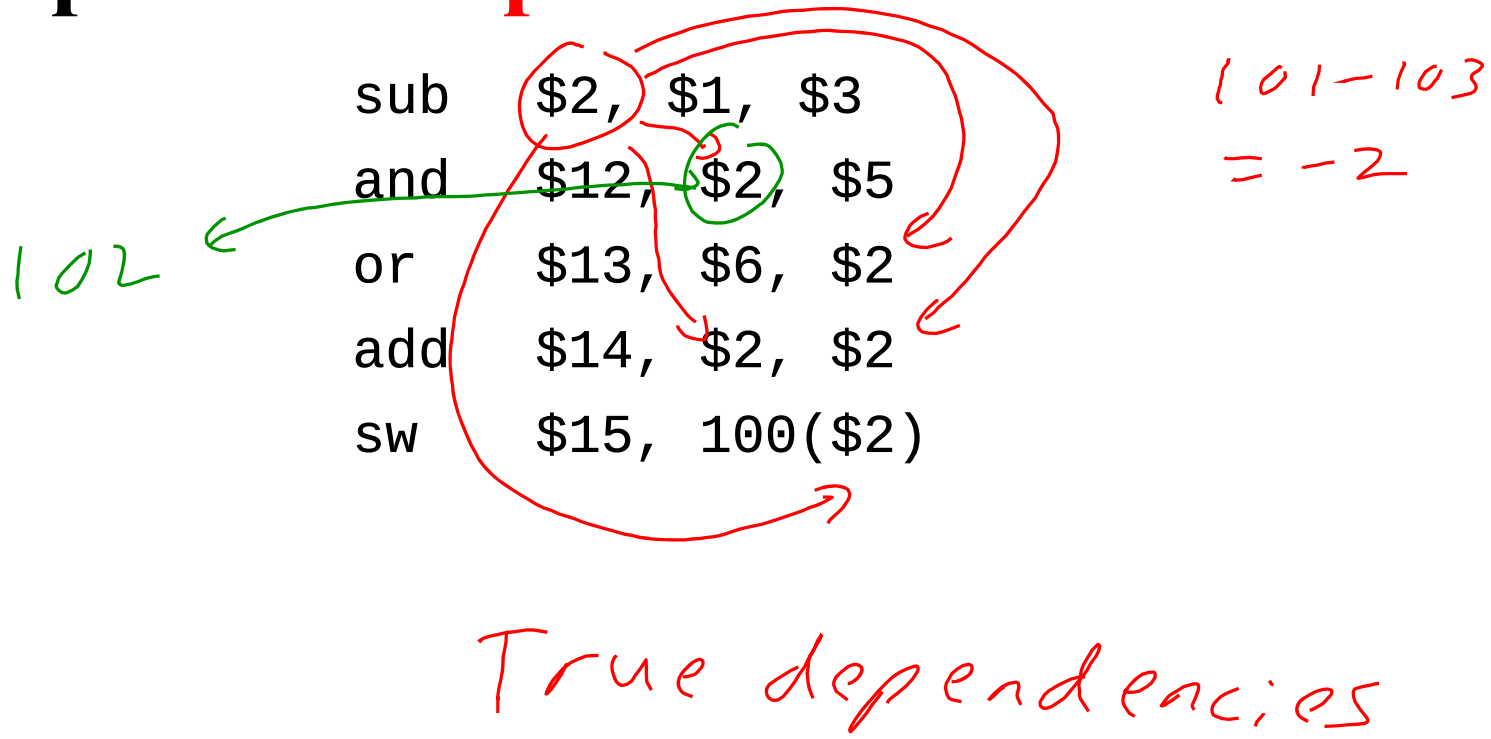
Writing back to registers is delayed creating data hazards

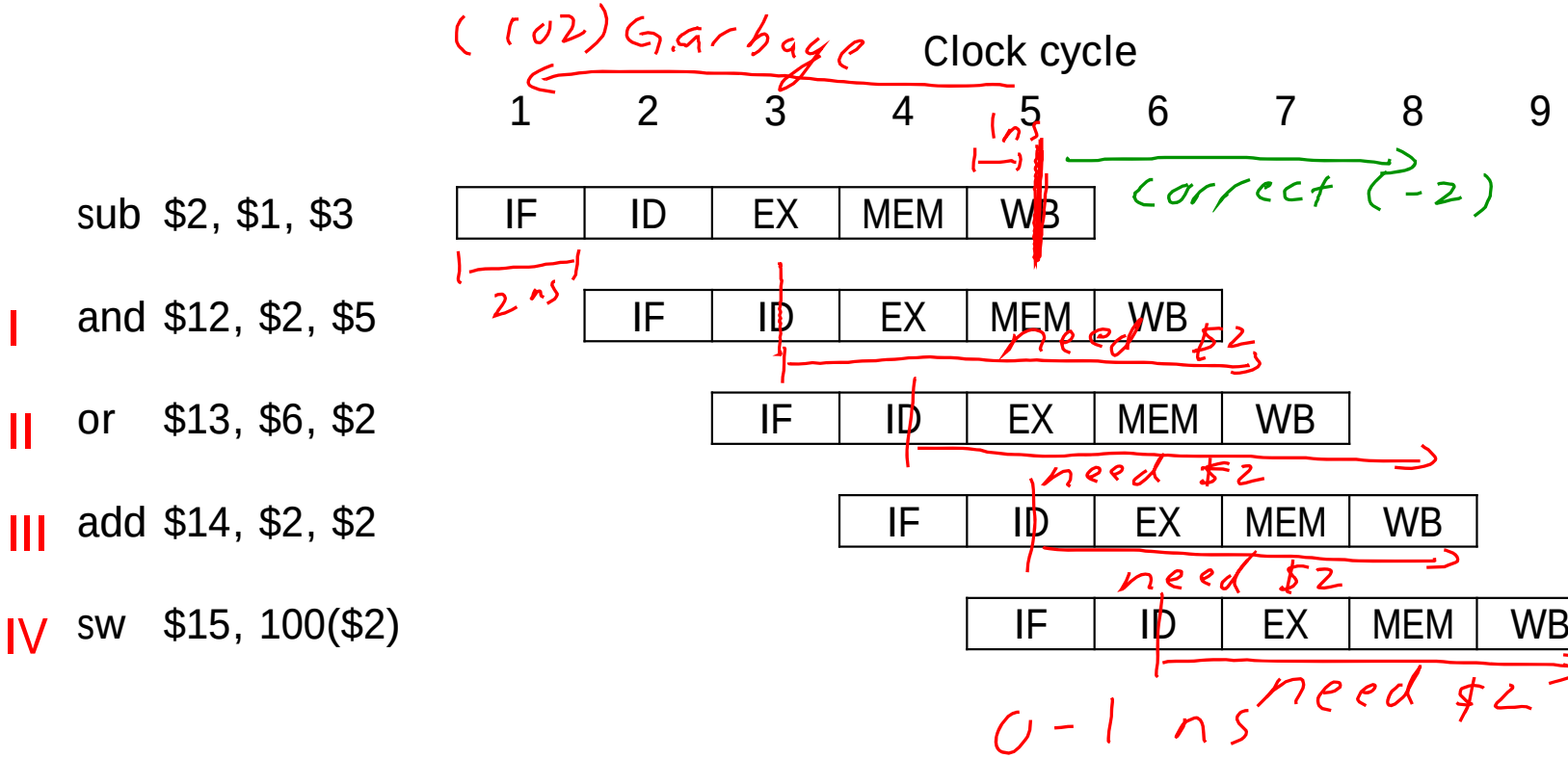
Our examples were too simple because they lacked dependencies between instructions

```
lw    $8, 4($29)
sub   $2, $4, $5
and   $9, $10, $11
or    $16, $17, $18
add   $13, $14, $0
```

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our datapath handles this sequence easily, as we saw last time.
- But most sequences of instructions are *not* independent!

An example with dependencies

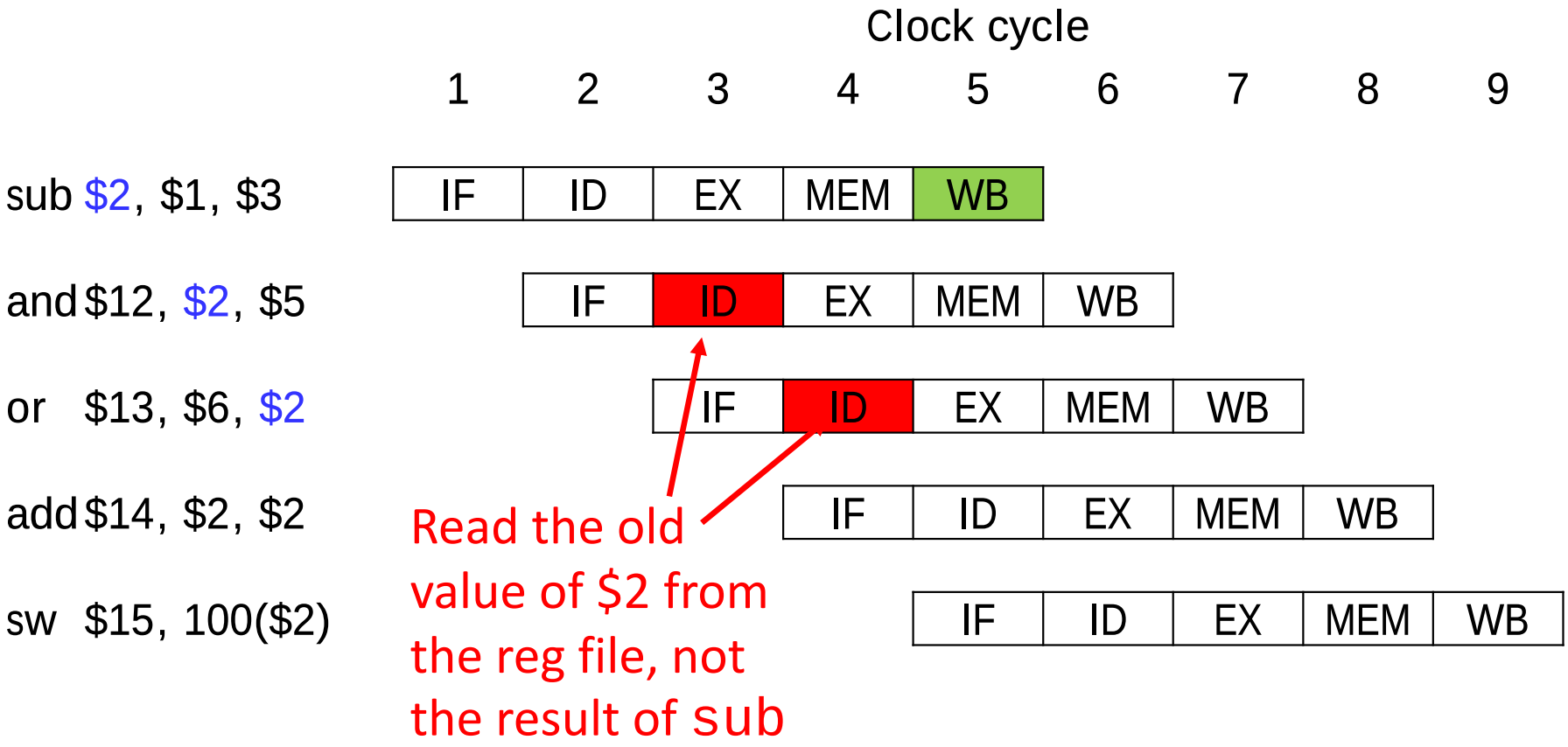




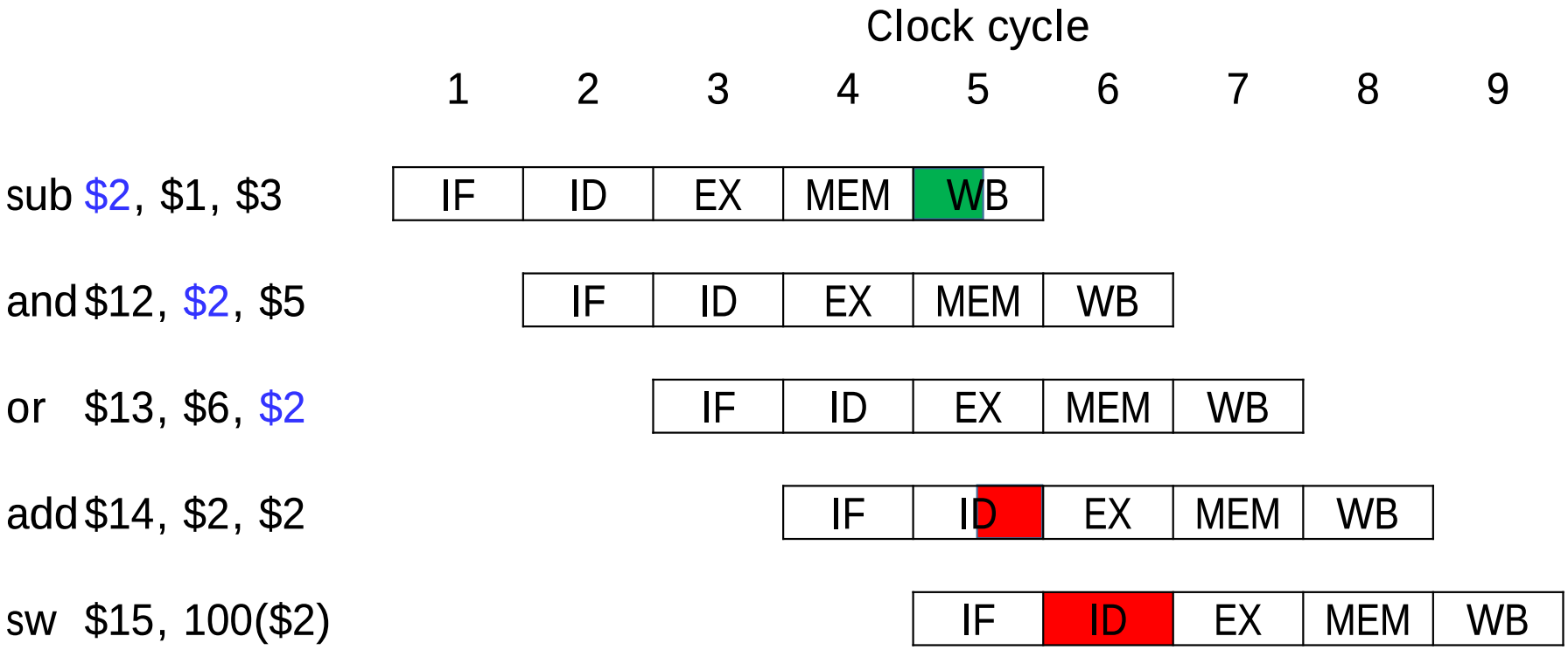
Which instructions might not execute correctly on the pipelined processor?

a) Just I
b) I and II
c) I, II, and III
d) I, II, III, and IV

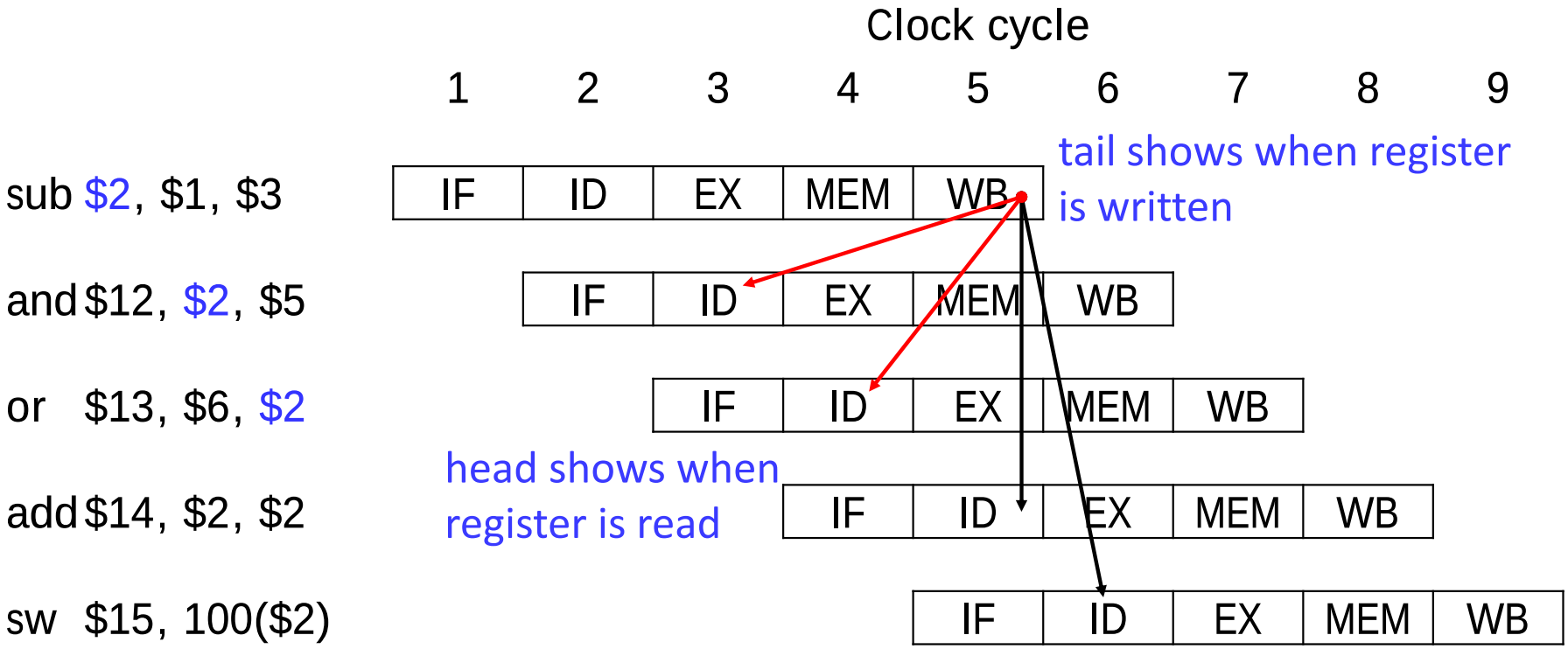
sub does not write to the reg file until cycle 5, this creates two data hazards



sub finishes writing to the reg file after half a clock cycle, reg file reads take half a cycle



Use arrows to show dependencies: Arrows that point backwards reveal data hazards



An alternate pipeline diagram showing components

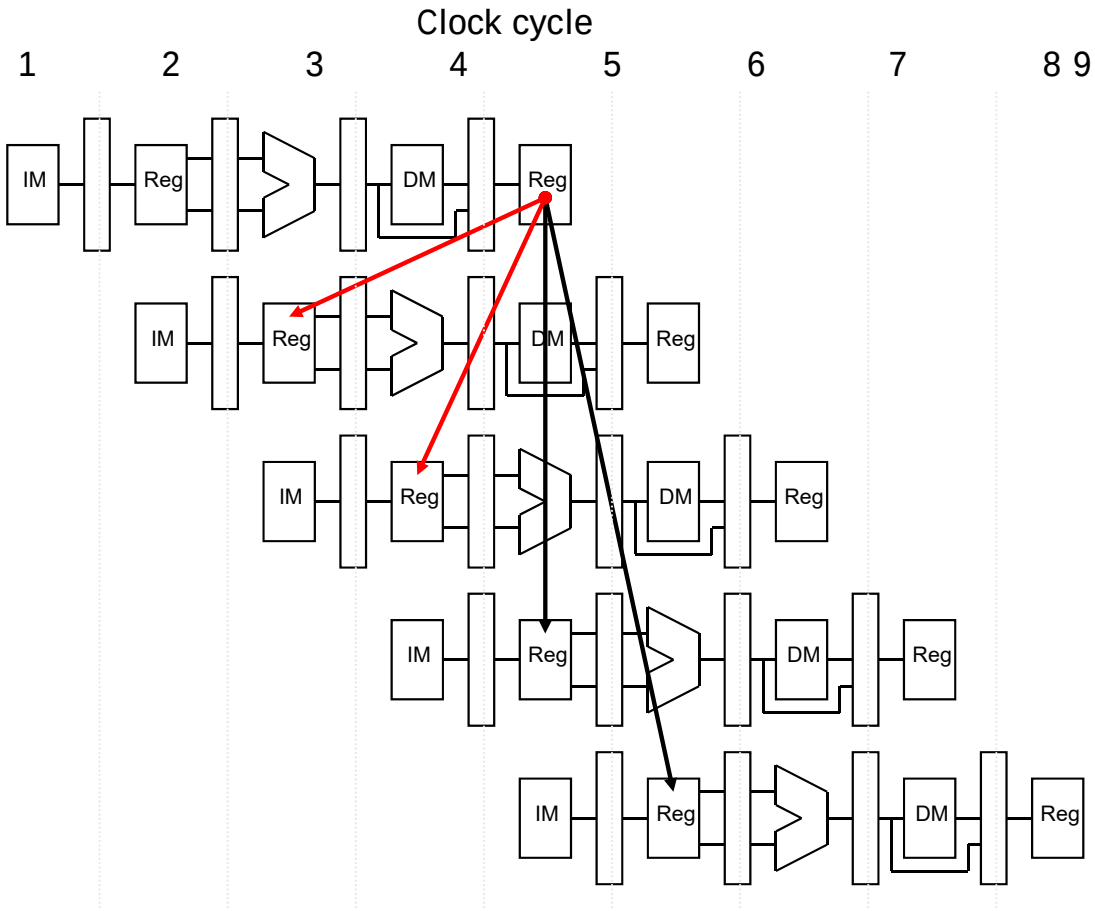
sub $2, $1, $3

and $12, $2, $5

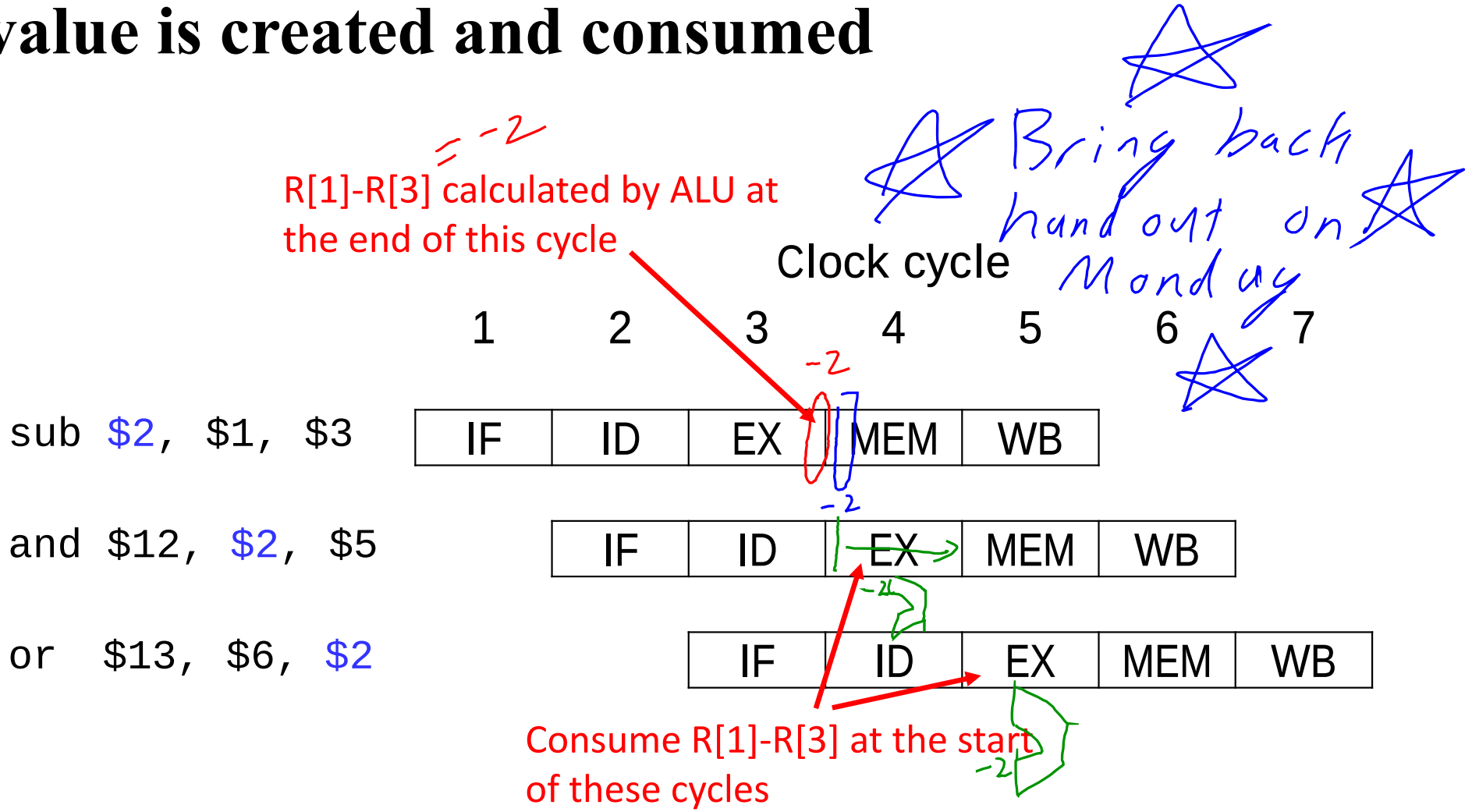
or $13, $6, $2

add $14, $2, $2

sw $15, 100($2)

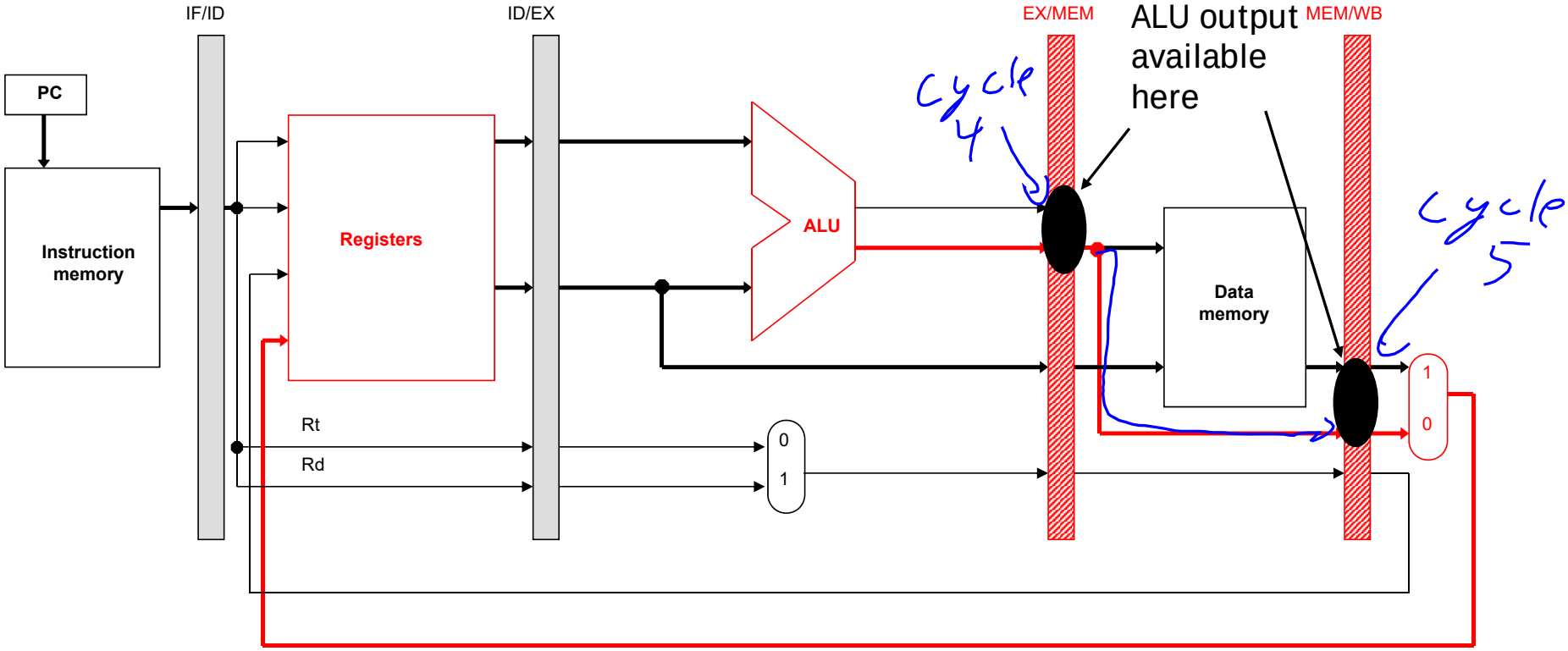


To eliminate hazards, identify when the correct value is created and consumed



Use pipeline registers to access correct values before values are written back to the reg file

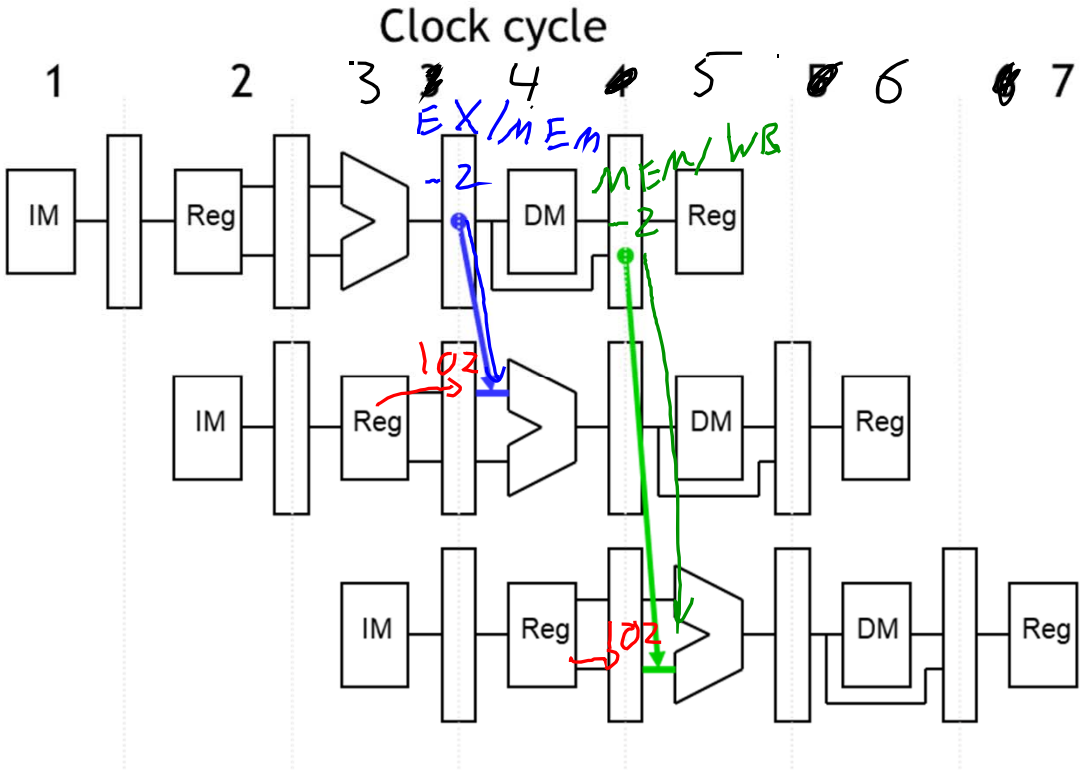
"Forward" data from pipeline registers to later instructions



Forward values from pipeline registers so later instructions can use the correct value

- In clock cycle 4, AND gets R[1]-R[3] from EX/MEM
- In cycle 5, OR gets R[1]-R[3] from MEM/WB

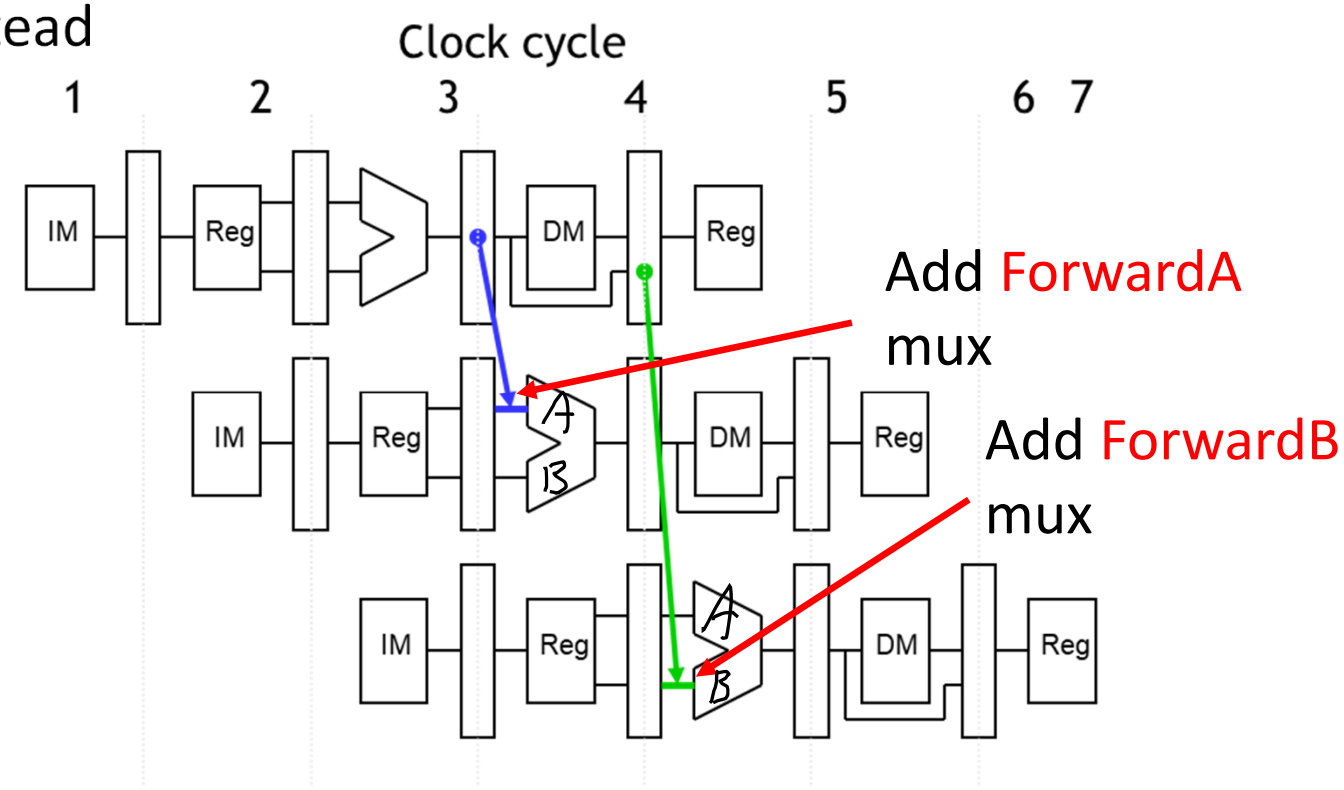
```
sub $2, $1, $3
and $12, $2, $5
Or  $13, $6, $2
```



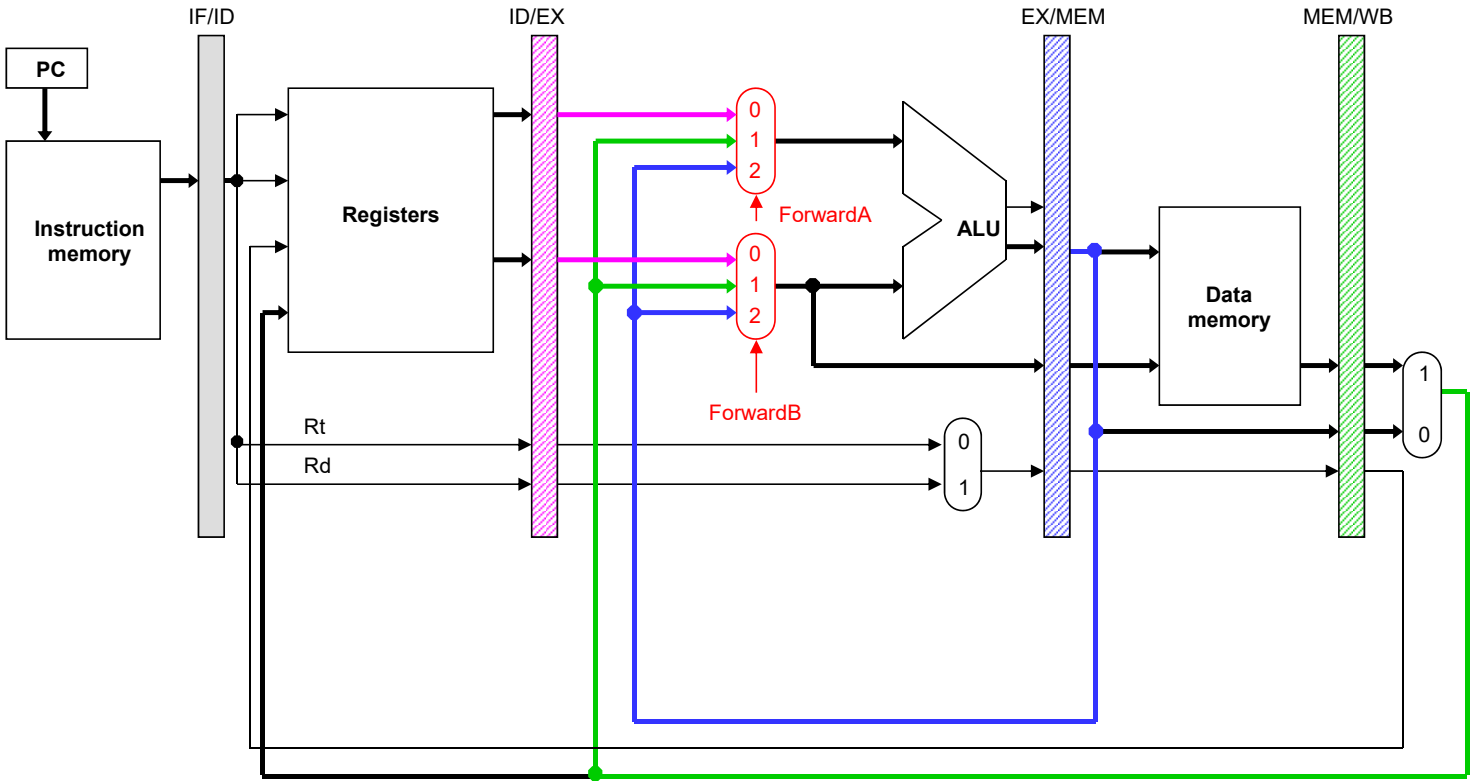
A forwarding unit selects the correct ALU inputs for the EX stage:

- No hazard: ALU's operands come from the register file, like normal
- Data hazard: operands come from either the EX/MEM or MEM/WB pipeline registers instead

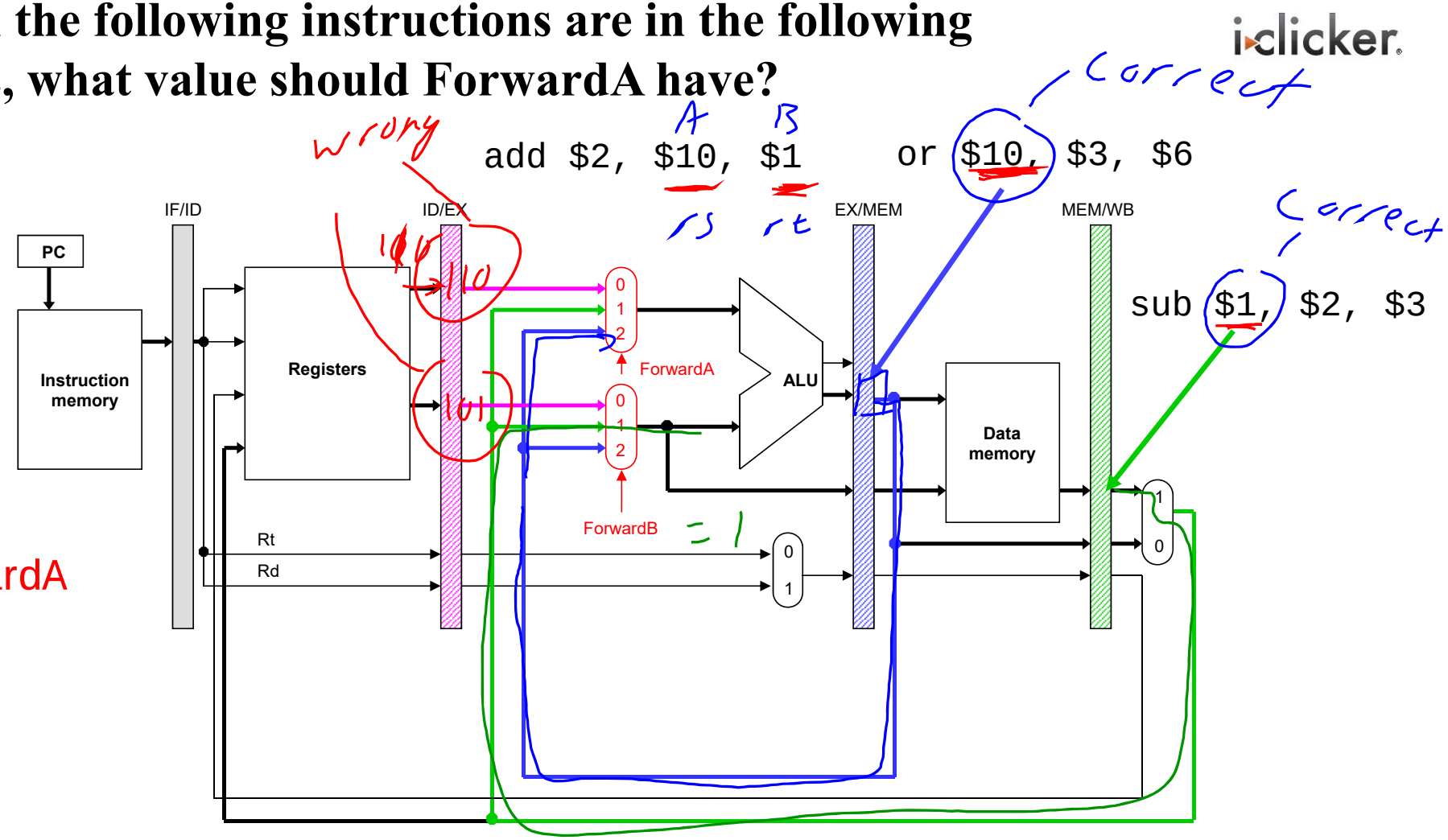
```
sub $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
```



Add forwarding muxes in front of the ALU



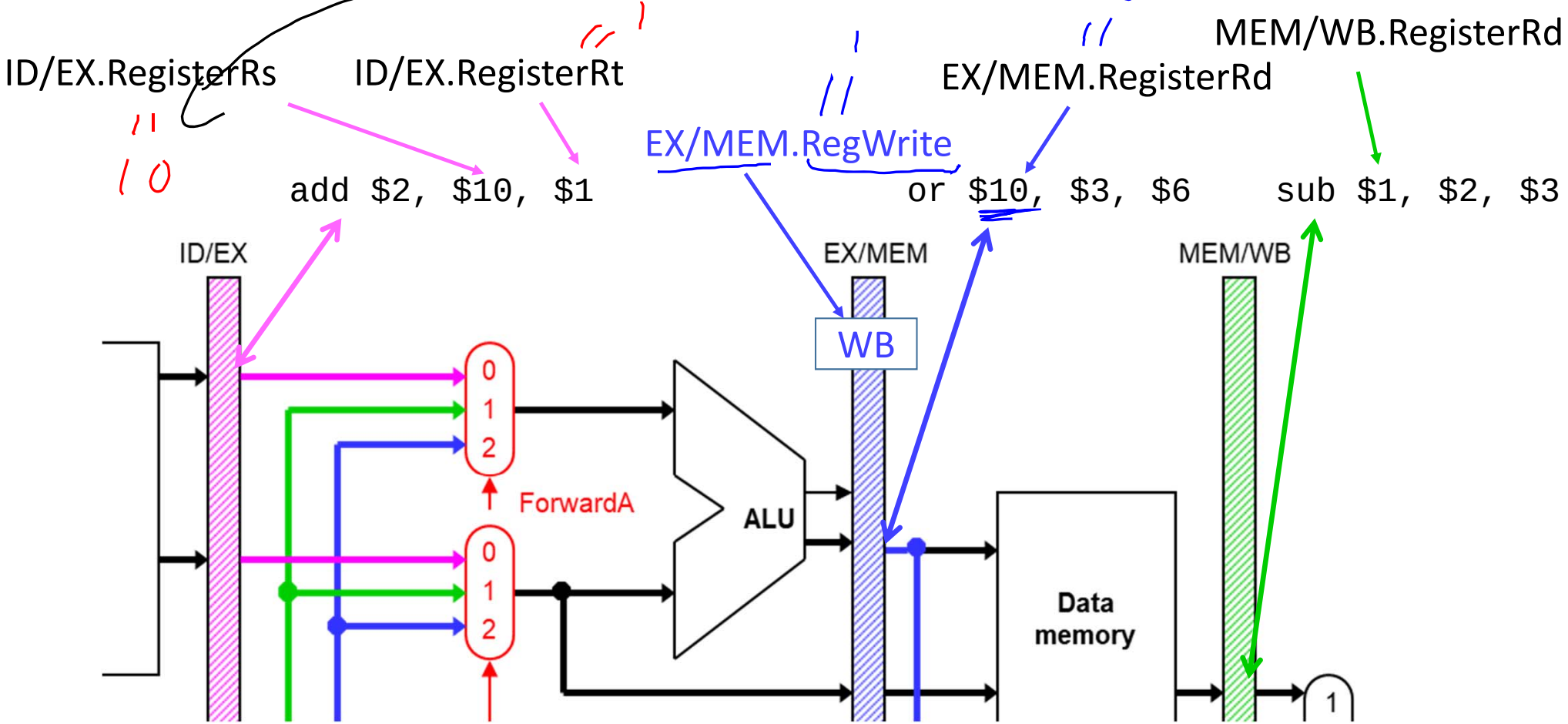
Given the following instructions are in the following stages, what value should ForwardA have?



ForwardA

A) 0
B) 1
C) 2

Use "." notation to indicate contents of pipeline registers

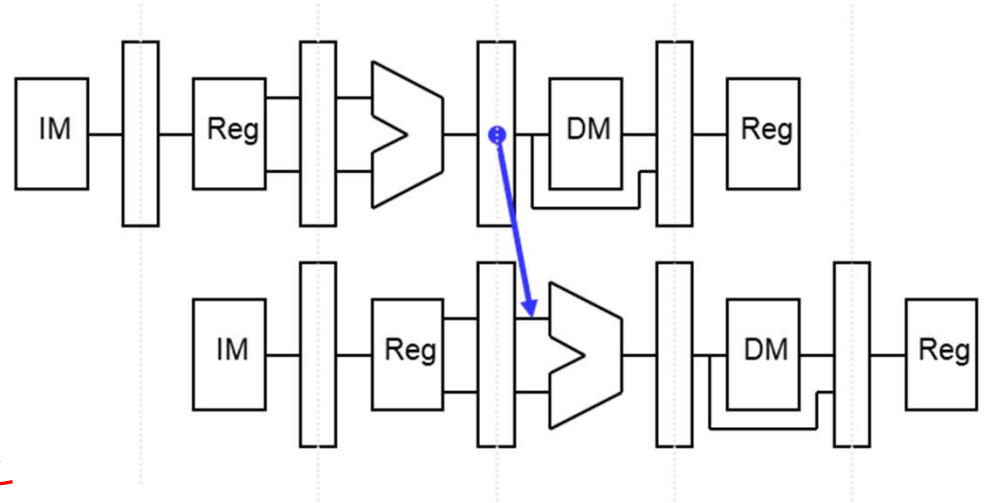


ALU source A comes from the pipeline register when necessary.

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

sub $2, $1, $3

and $12, $2, $5



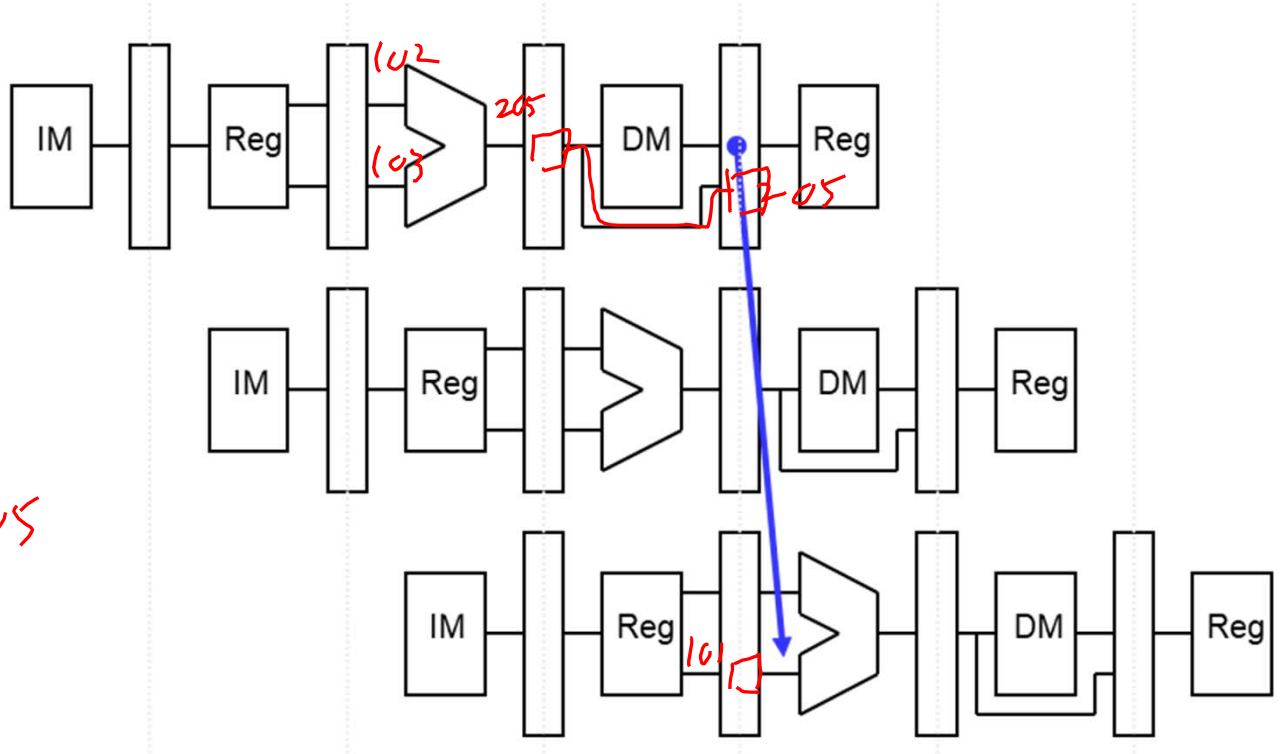
AND $12, $5, $2

- ALU source B is similar.

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

A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago

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



What happens if a register is updated twice in a row?

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

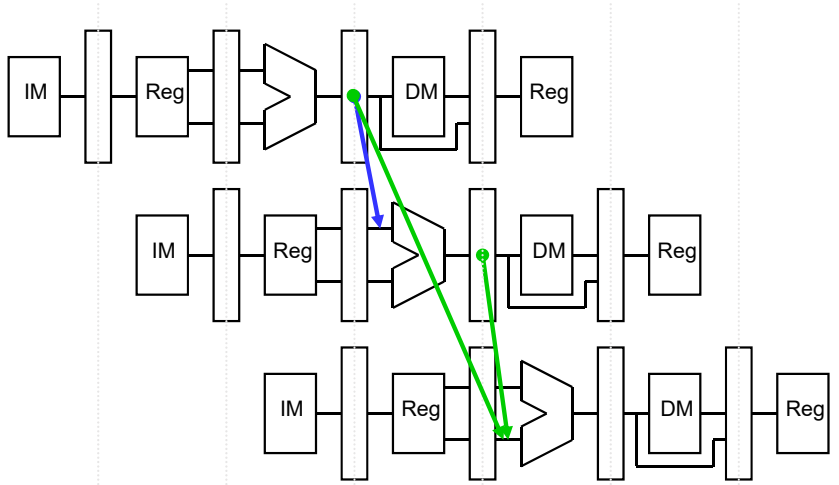
- Register $1 is written by *both* of the previous instructions; from which instruction should it receive its value?

i-clicker

a) add $1, $2, $3

b) add $1, $1, $4

sub $5, $5, $1



MEM/WB hazard equations

- Equation for MEM/WB hazards for ALU source A

  if (MEM/WB.RegWrite = 1
    and MEM/WB.RegisterRd = ID/EX.RegisterRs

  then ForwardA = 1

- Equation for MEM/WB hazards for ALU source B

  if (MEM/WB.RegWrite = 1
    and MEM/WB.RegisterRd = ID/EX.RegisterRt

  then ForwardB = 1

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

MEM/WB hazard equations

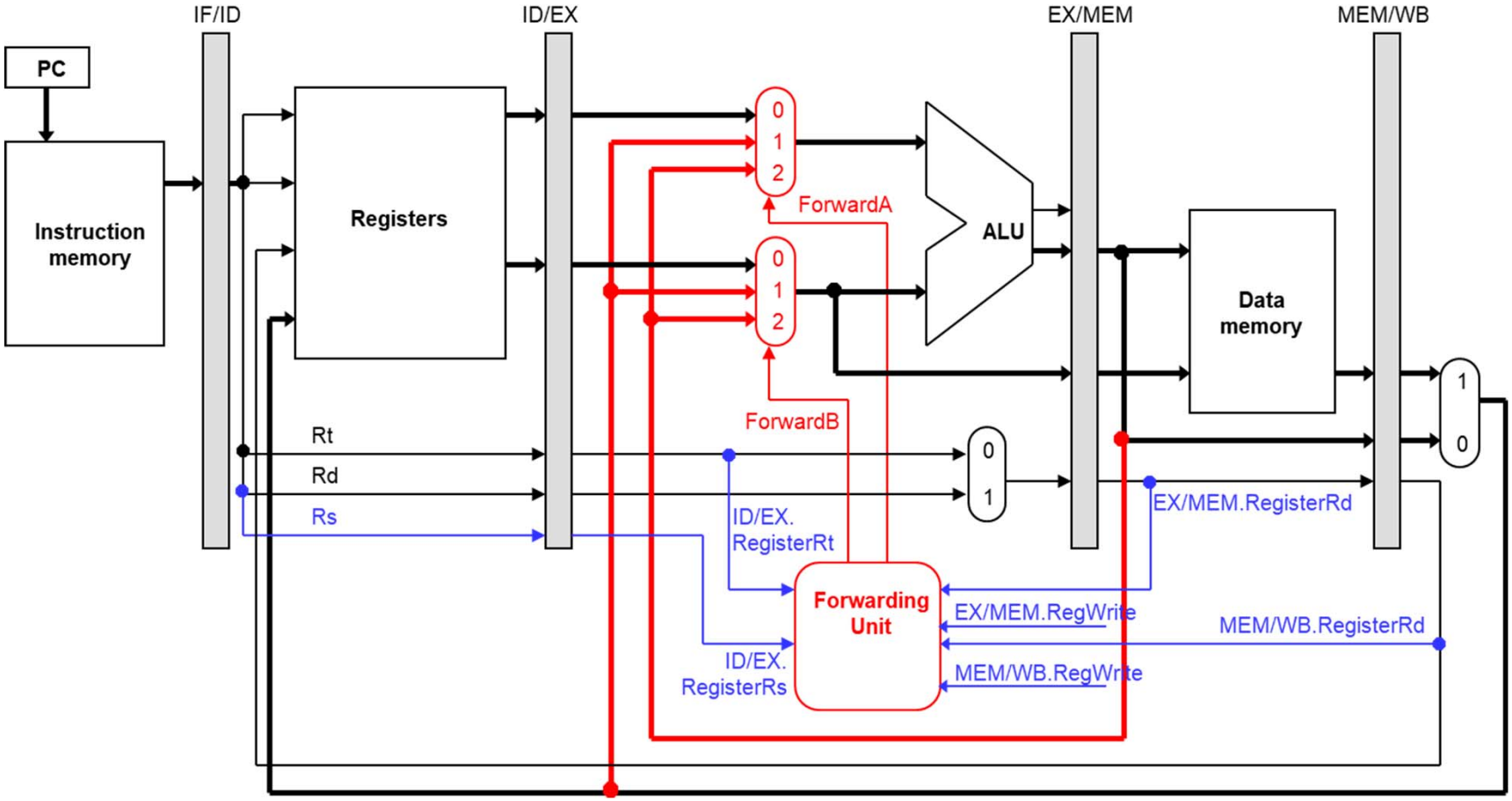
- Equation for MEM/WB hazards for ALU source A

  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

- Equation for MEM/WB hazards for ALU source B

  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

Add forwarding unit control logic



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. 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

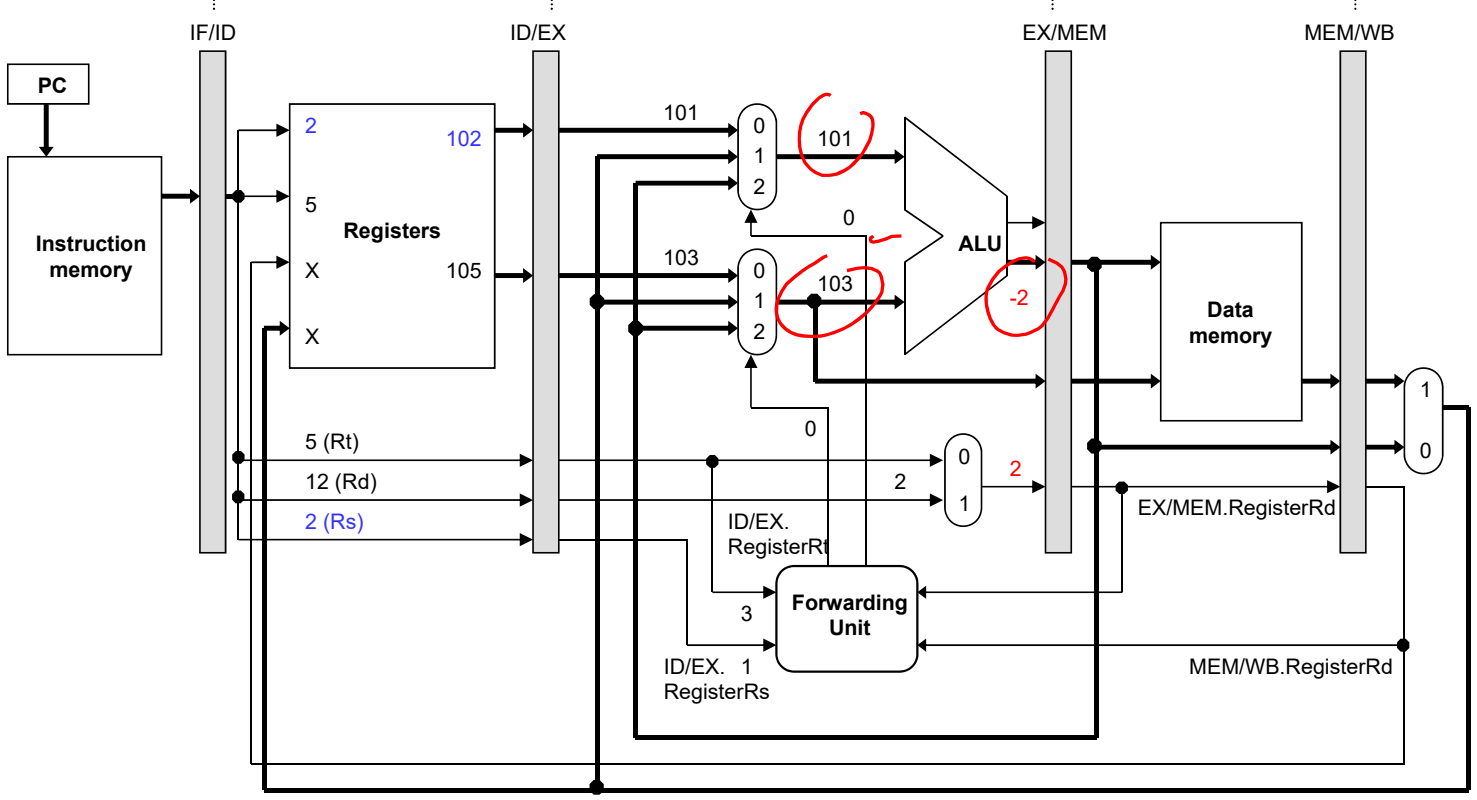
```
sub $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

- 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.

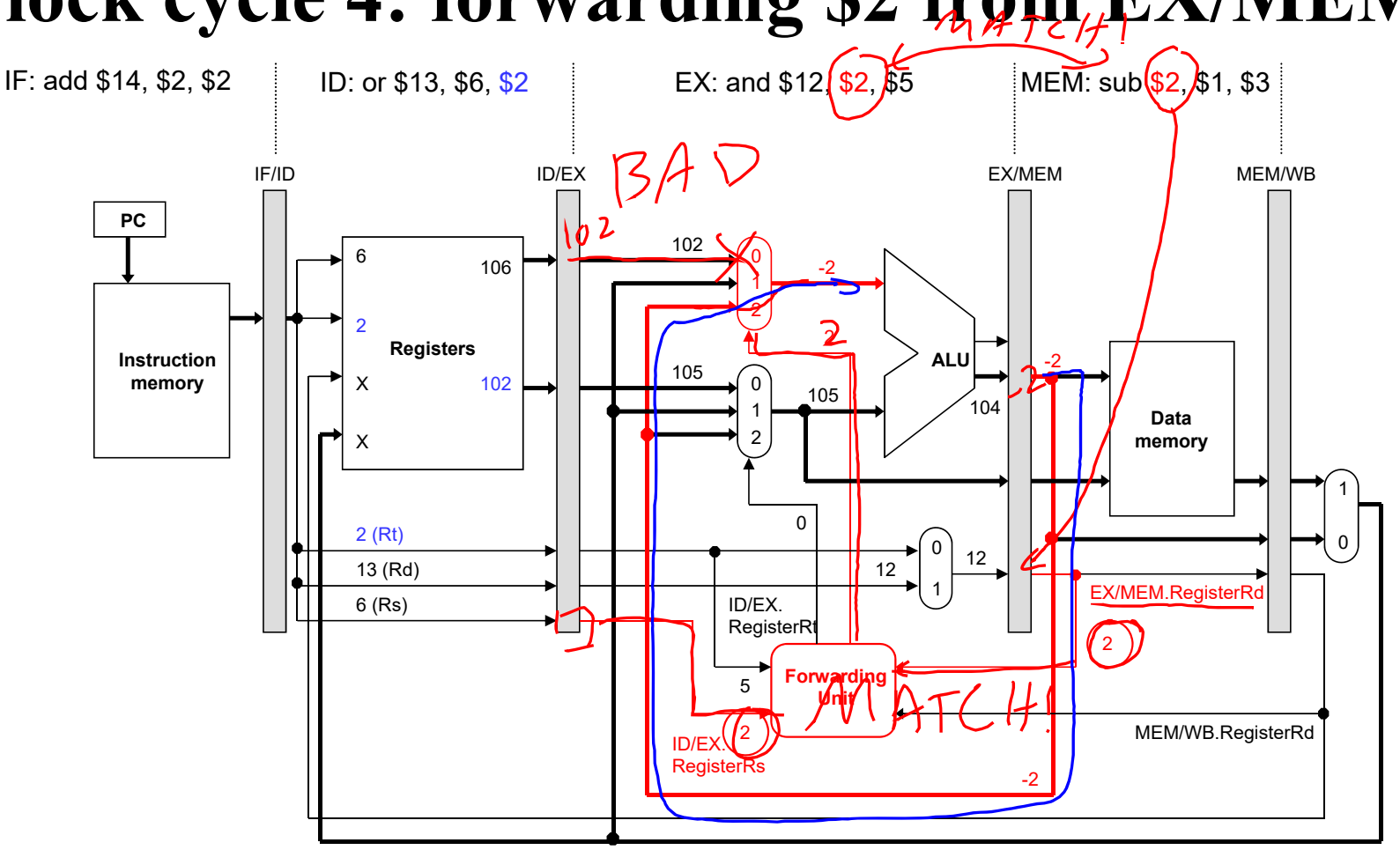
- We'll try to keep the example short.
  - Assume no forwarding is needed except for register $2.
  - We'll skip the first two cycles, since they're the same as before.

Clock cycle 3

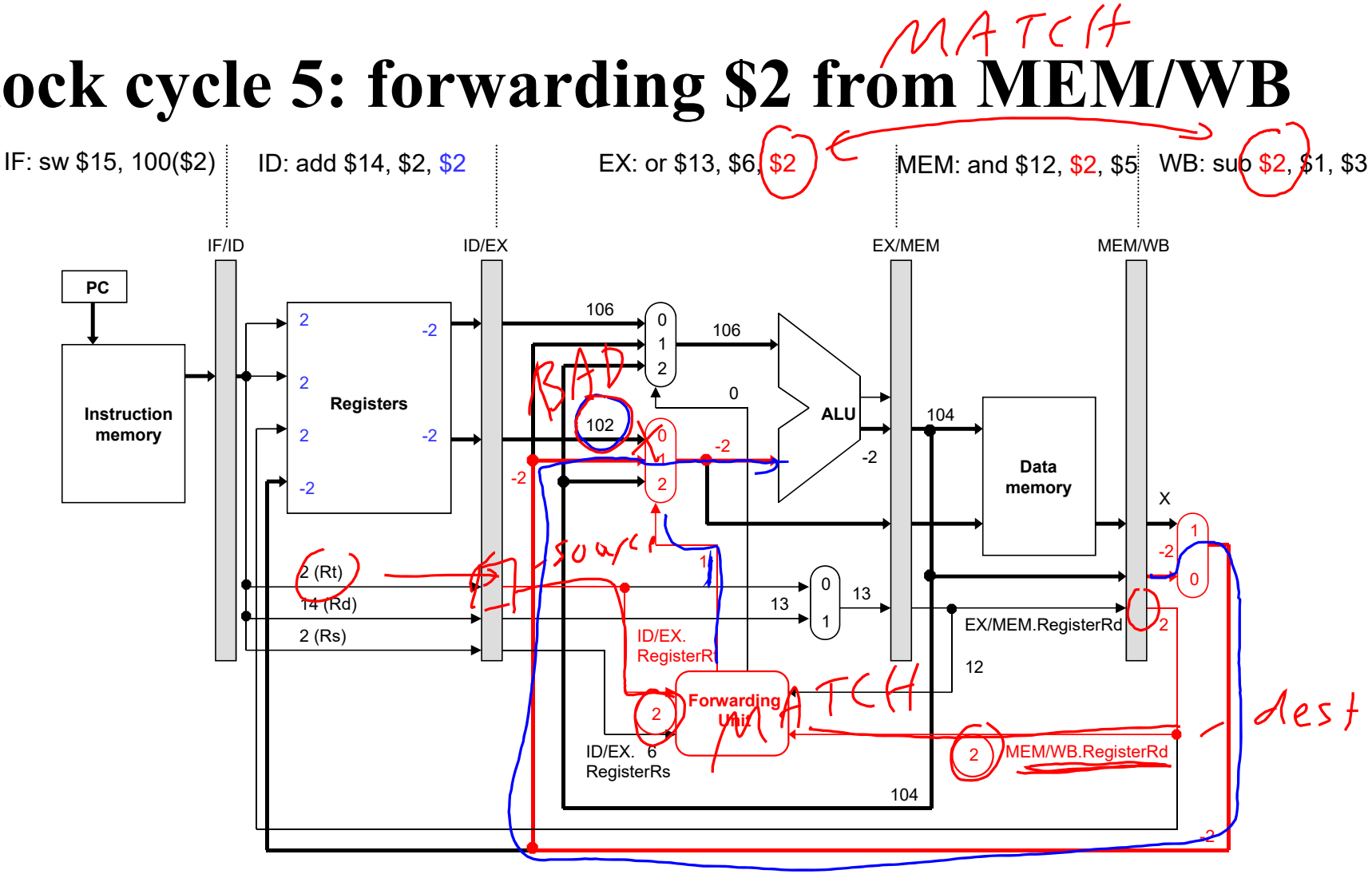
IF: or $13, $6, $2 | ID: and $12, $2, $5 | EX: sub $2, $1, $3



Clock cycle 4: forwarding $2 from EX/MEM



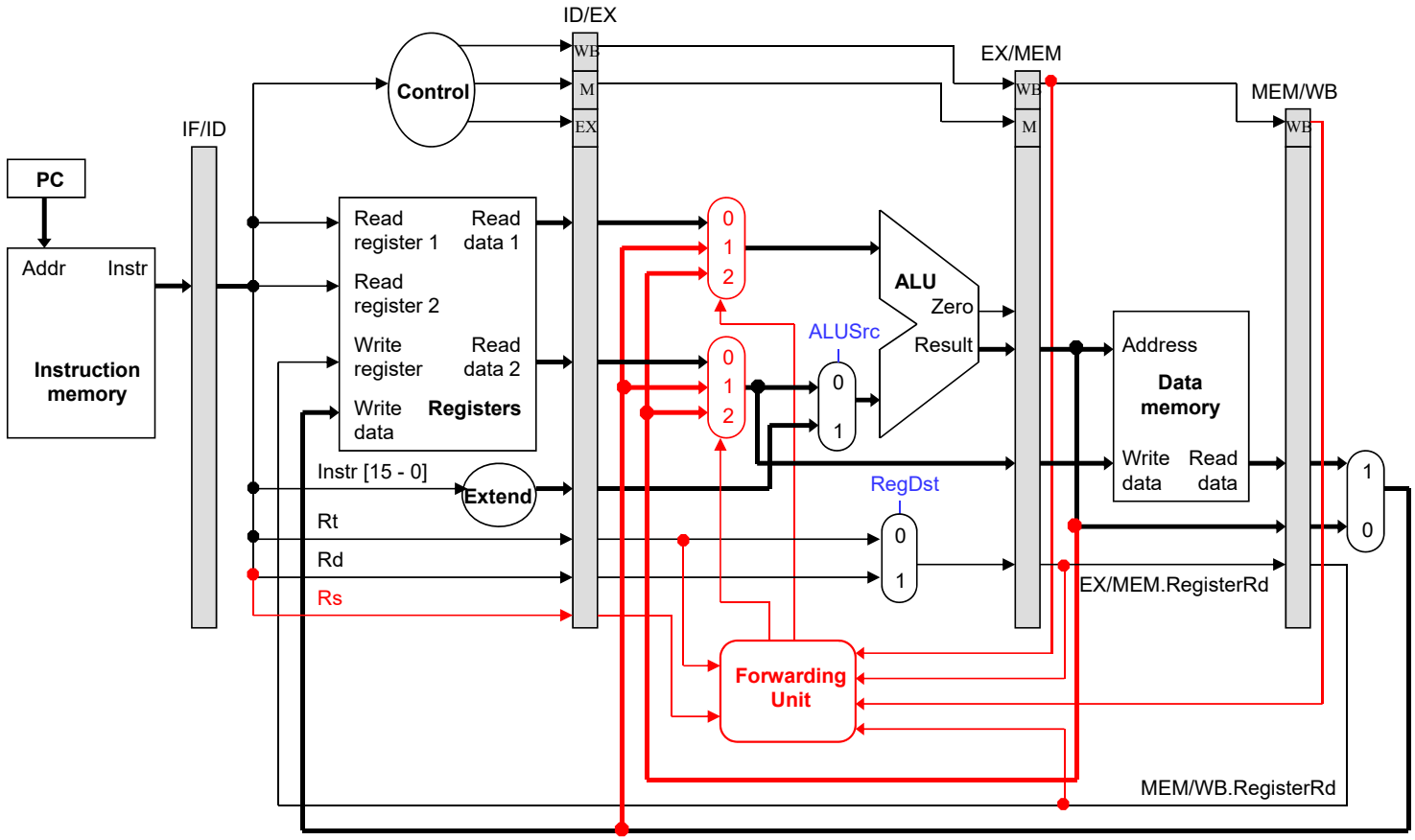
Clock cycle 5: forwarding $2 from MEM/WB



Lots of data hazards

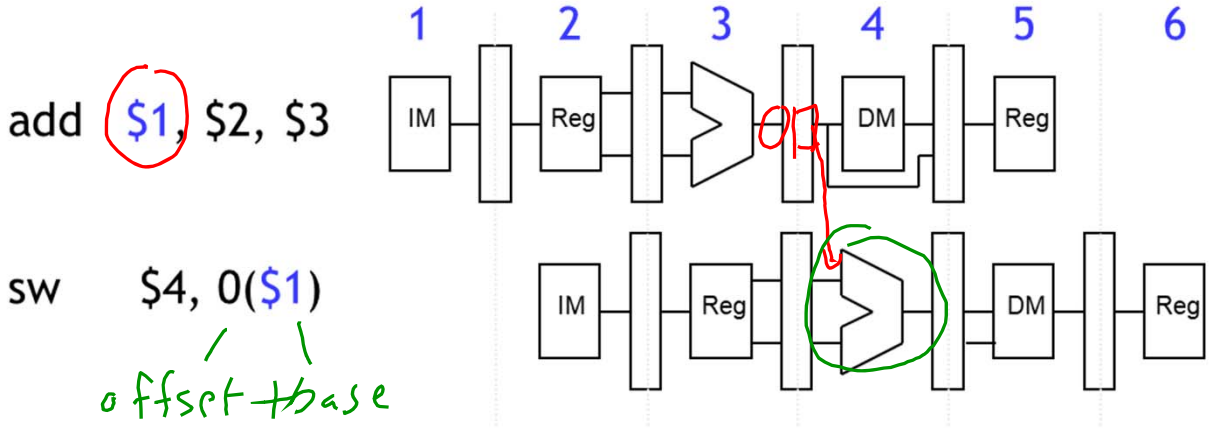
- The first data hazard occurs during cycle 4.
  - The forwarding unit notices that the ALU's first source register for the AND is also the destination of the SUB instruction.
  - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
- 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

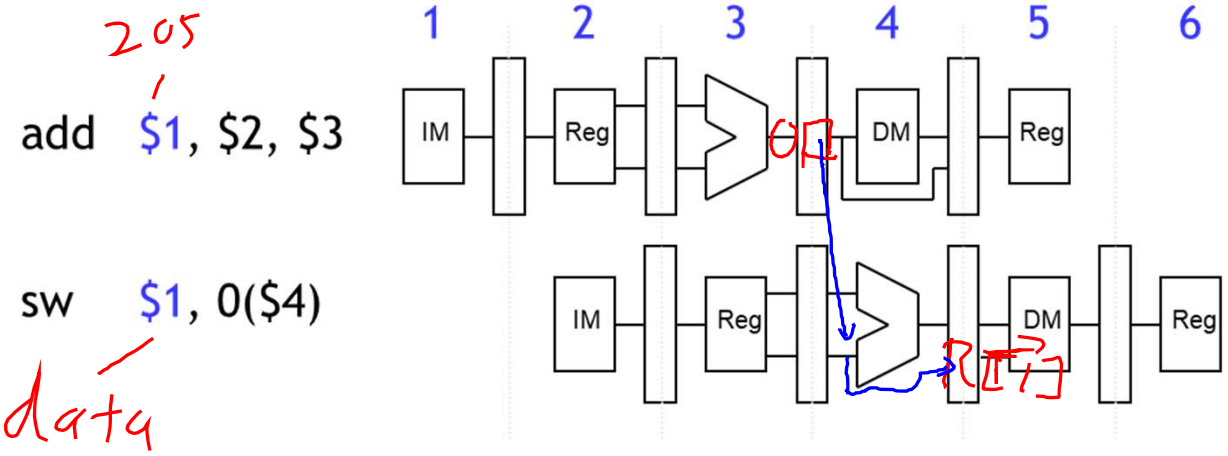


We can forward for store instructions too

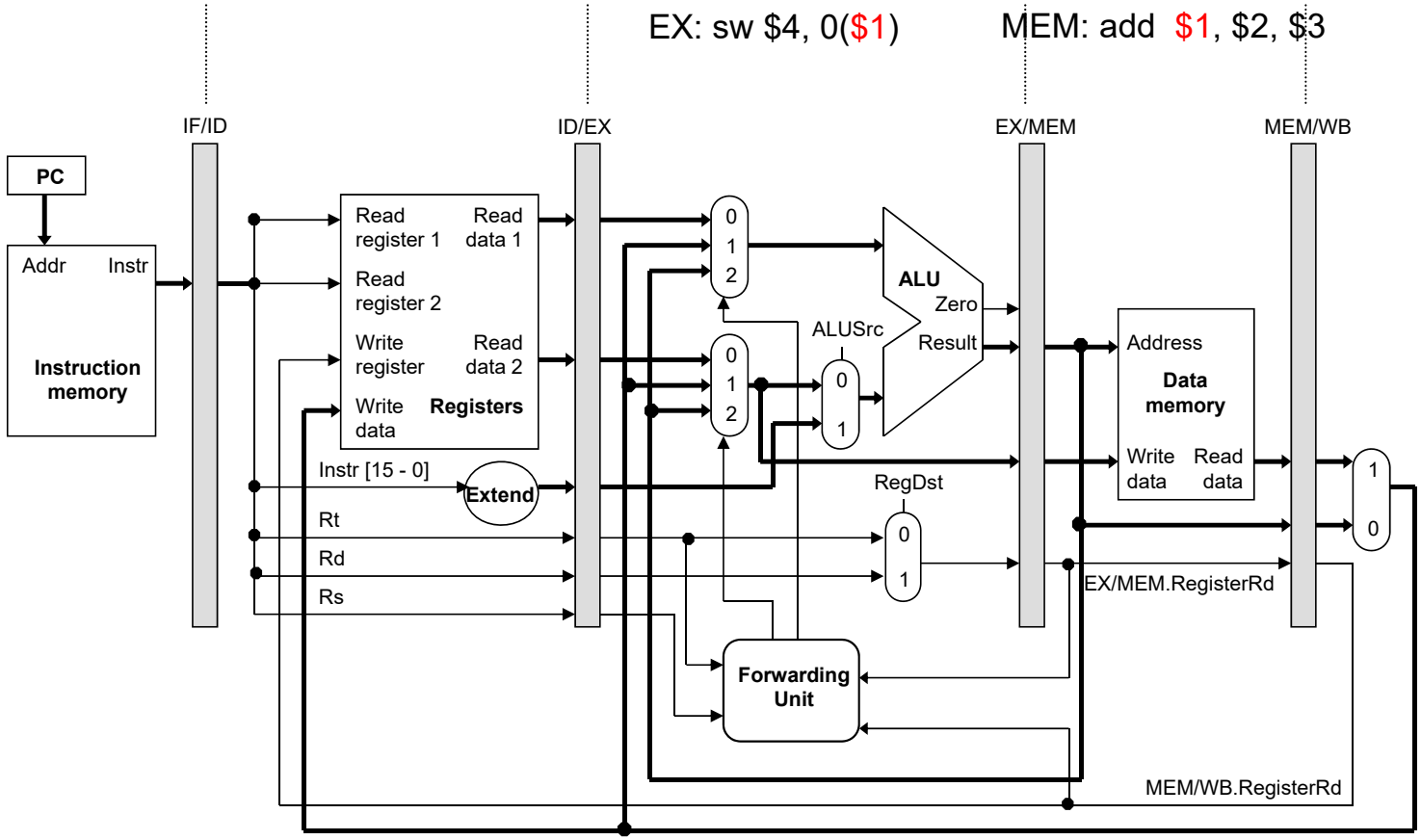
Forward to ~~offset~~ Base



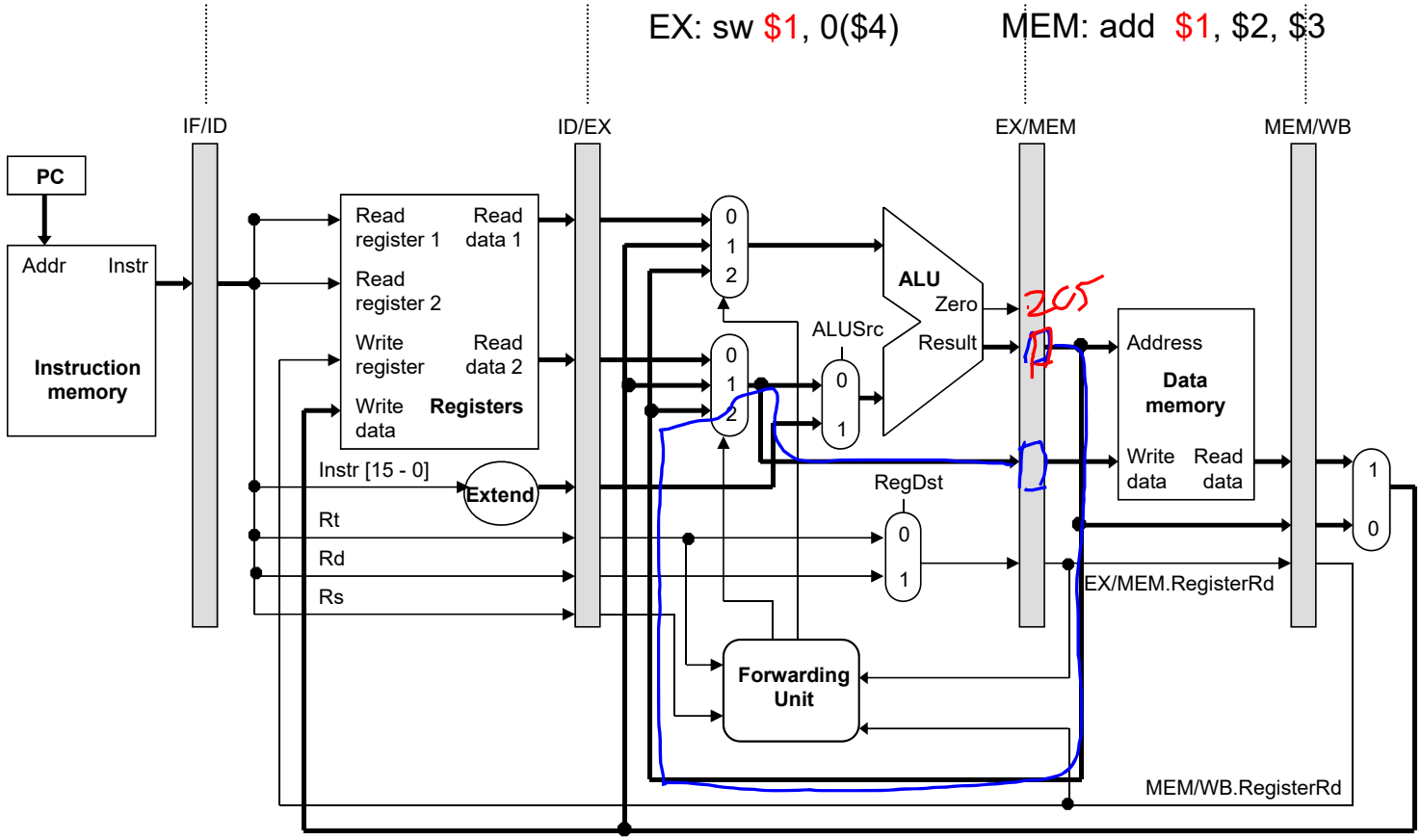
Forward to data



Store Bypassing: Forward to offset

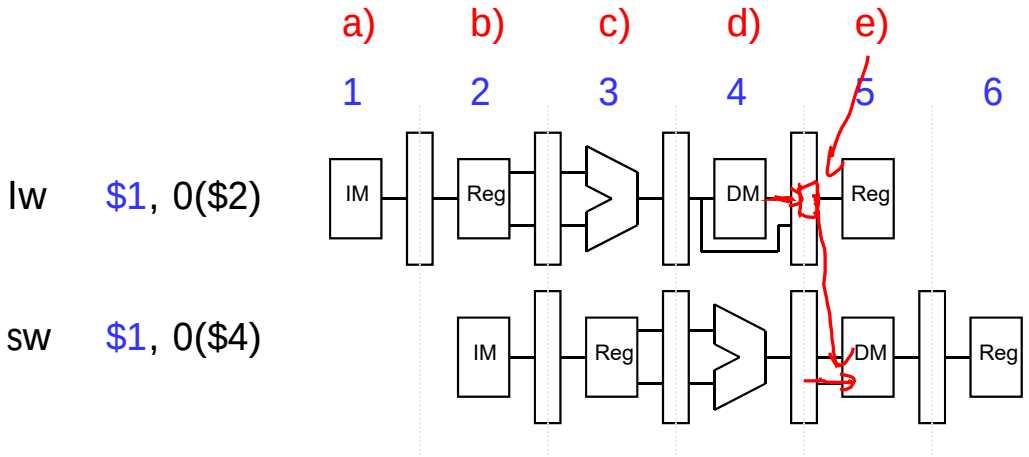


Store Bypassing: Forward to data



What about stores after a load?

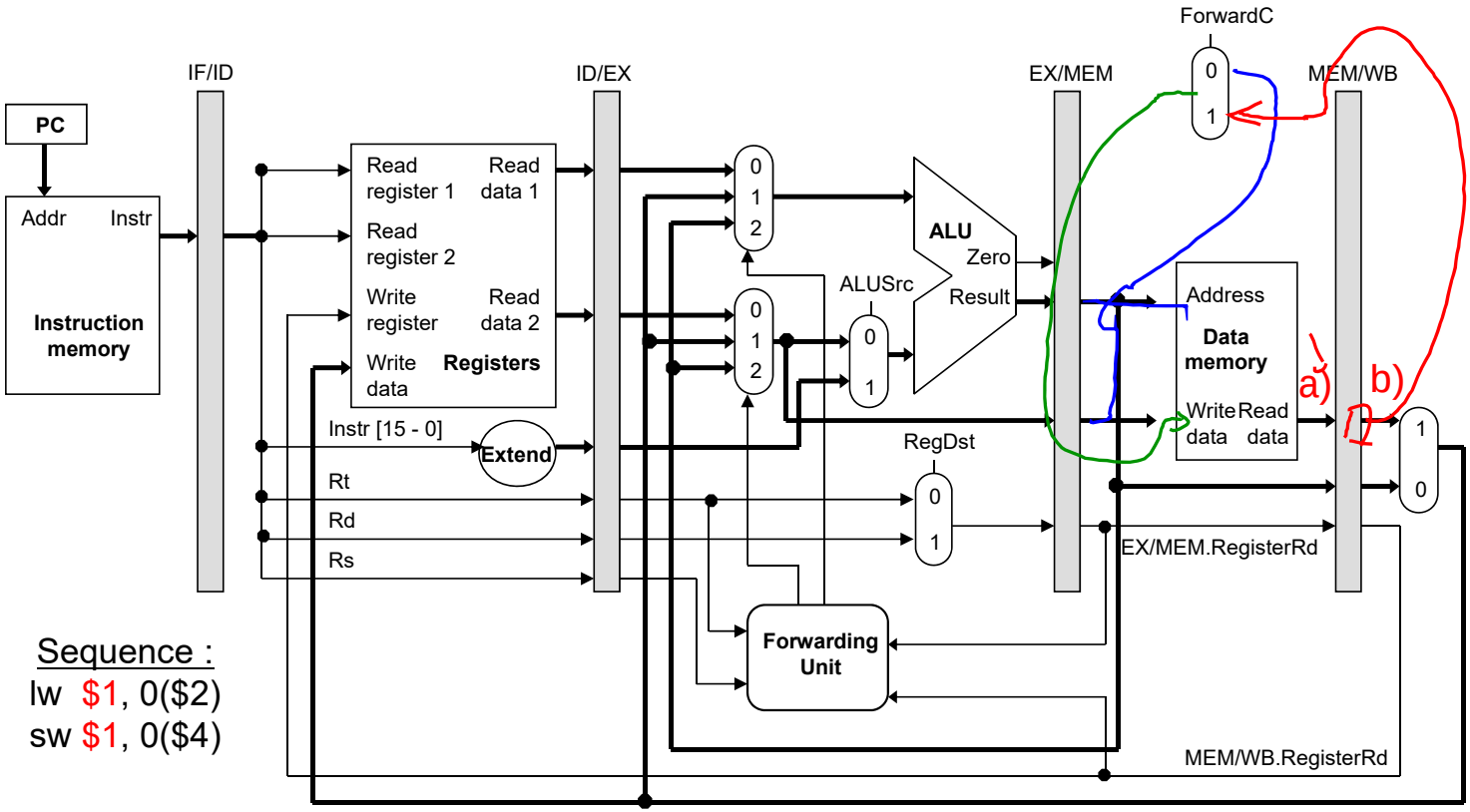
i-clicker



- In what cycle is:
  - The load value available?
  - The store value needed?

- What do we have to add to the datapath?

Load/Store Bypassing: Extend the Datapath



Sequence :

lw $1, 0($2)

sw $1, 0($4)

Where should we forward from?

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!

Summary

- In real code, most instructions are dependent upon other ones.
  - This can lead to data hazards in our original pipelined datapath.
  - Instructions can't write back to the register file soon enough for the next two instructions to read.
- Forwarding eliminates data hazards involving arithmetic instructions.
  - The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction.
  - Hazards are avoided by grabbing results from the pipeline registers *before* they are written back to the register file.
- Next time we'll finish up pipelining.
  - Forwarding can't save us in some cases involving lw.
  - We still haven't talked about branches for the pipelined datapath.