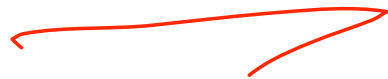


# Forwarding

---

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

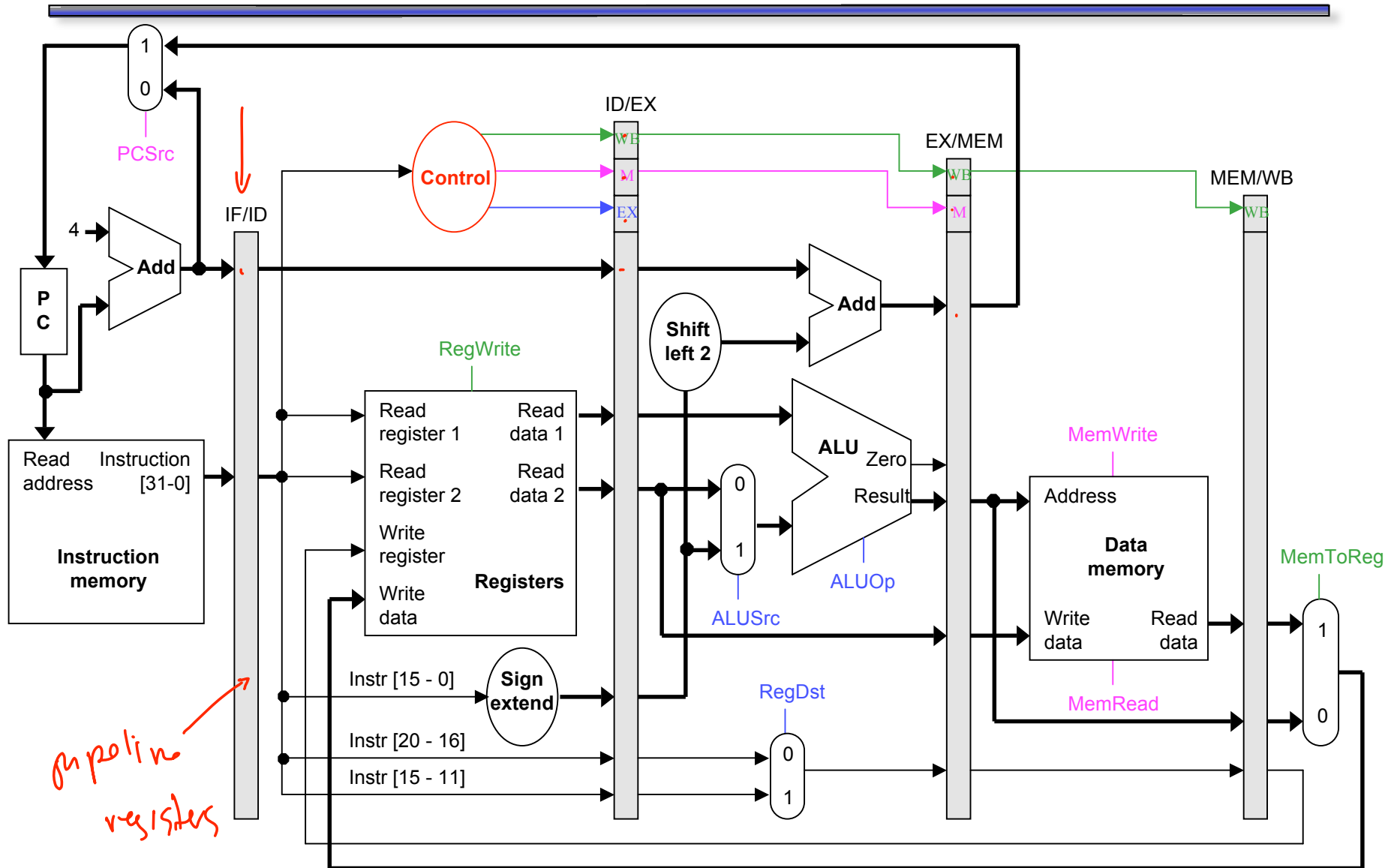
Pick-up hand-out!



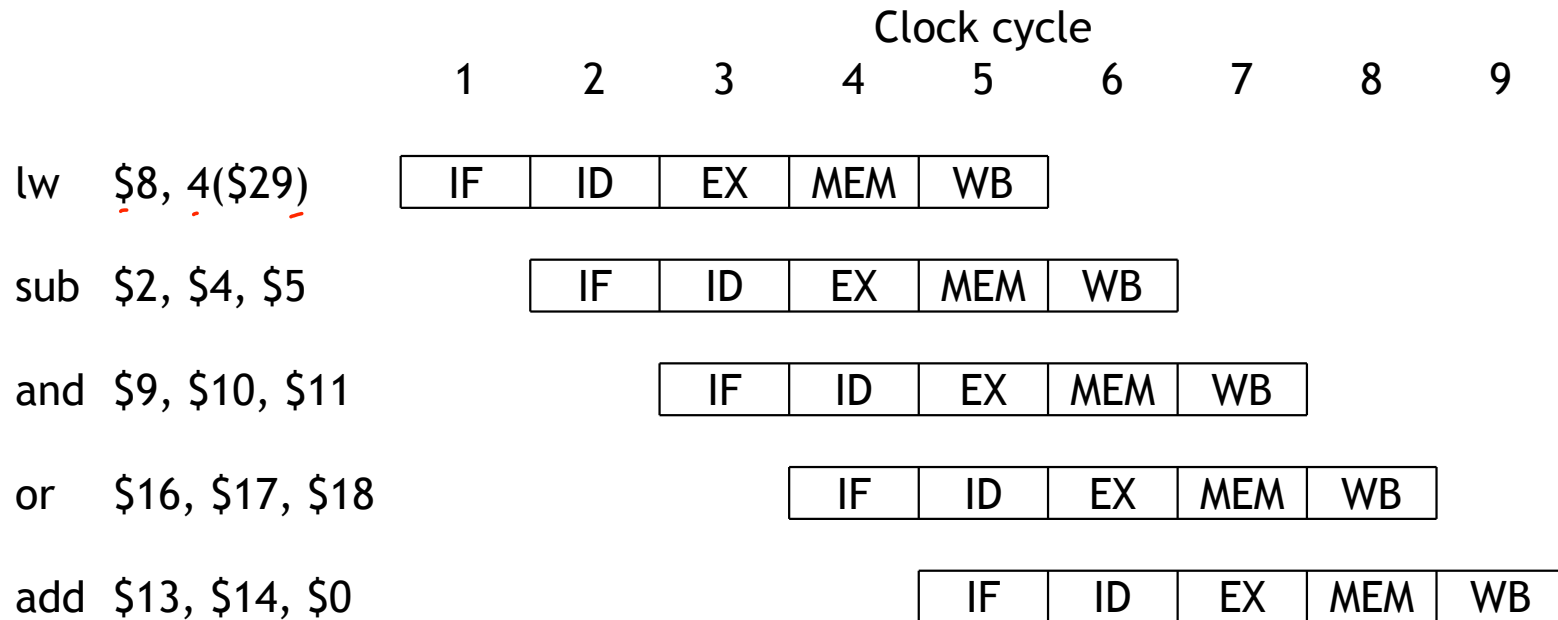
0

h

# The pipelined datapath



# Pipeline diagram review



- 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.

## Our examples are too simple

---

- Here is the example instruction sequence used to illustrate pipelining on the previous page.

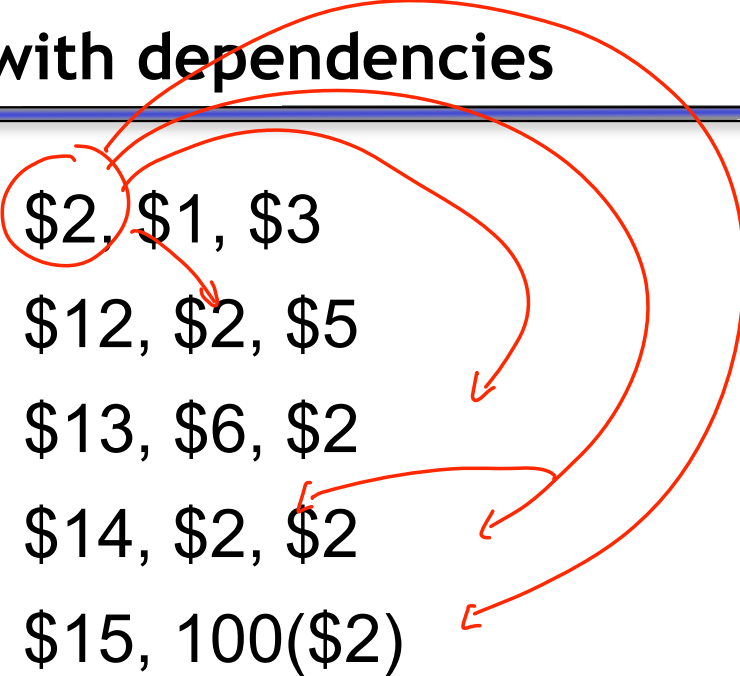
```
lw    $8, 4($29)
sub   $2, $4, $5
and   $9, $10, $11
or    $16, $17, $18
add   $13, $14, $0
```

- 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

---

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



The diagram shows red arrows indicating dependencies between the '\$2' values in the list items. A circle highlights the '\$2' in the 'sub' item. Arrows point from this '\$2' to the '\$2' in 'and', 'or', 'add', and 'sw'. Additionally, an arrow points from the '\$2' in 'and' to the '\$2' in 'or', and another from the '\$2' in 'or' to the '\$2' in 'add'.

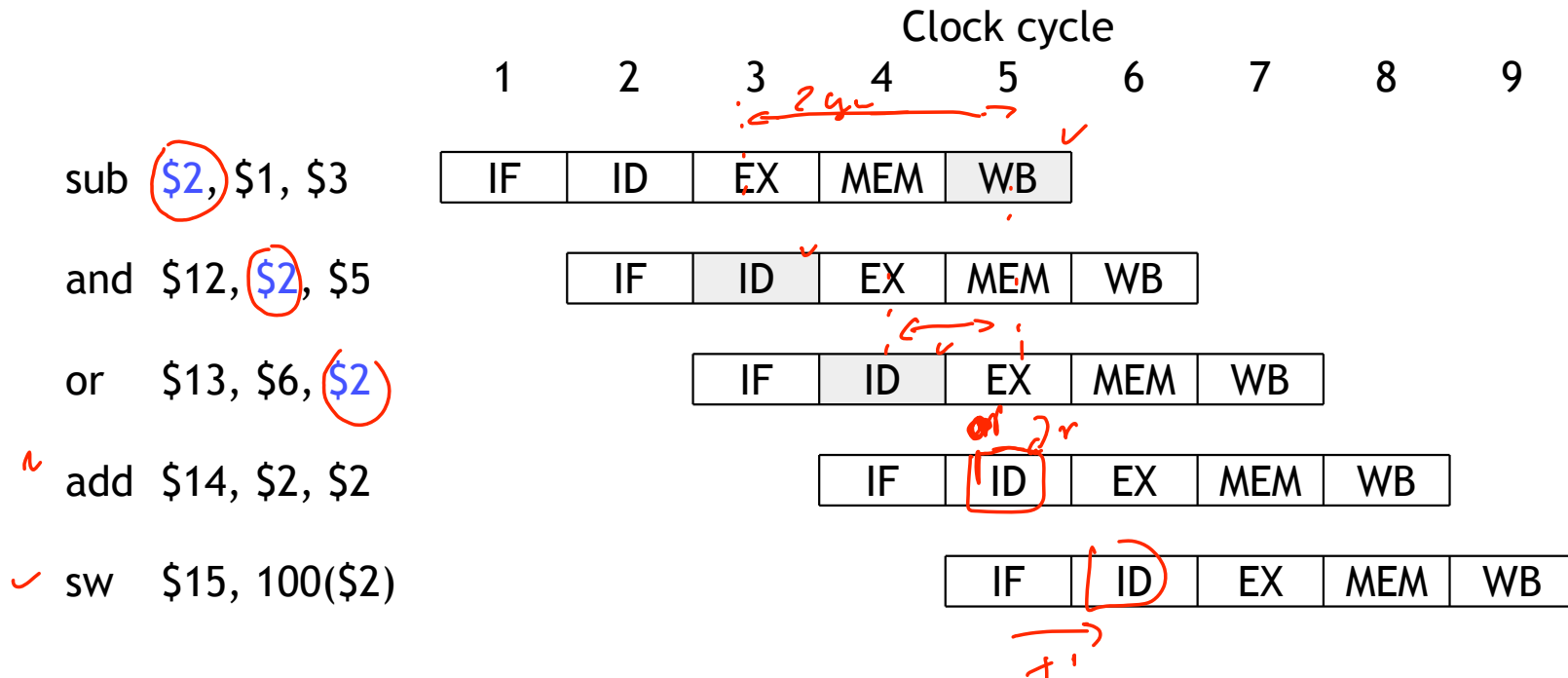
## An example with dependencies

---

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

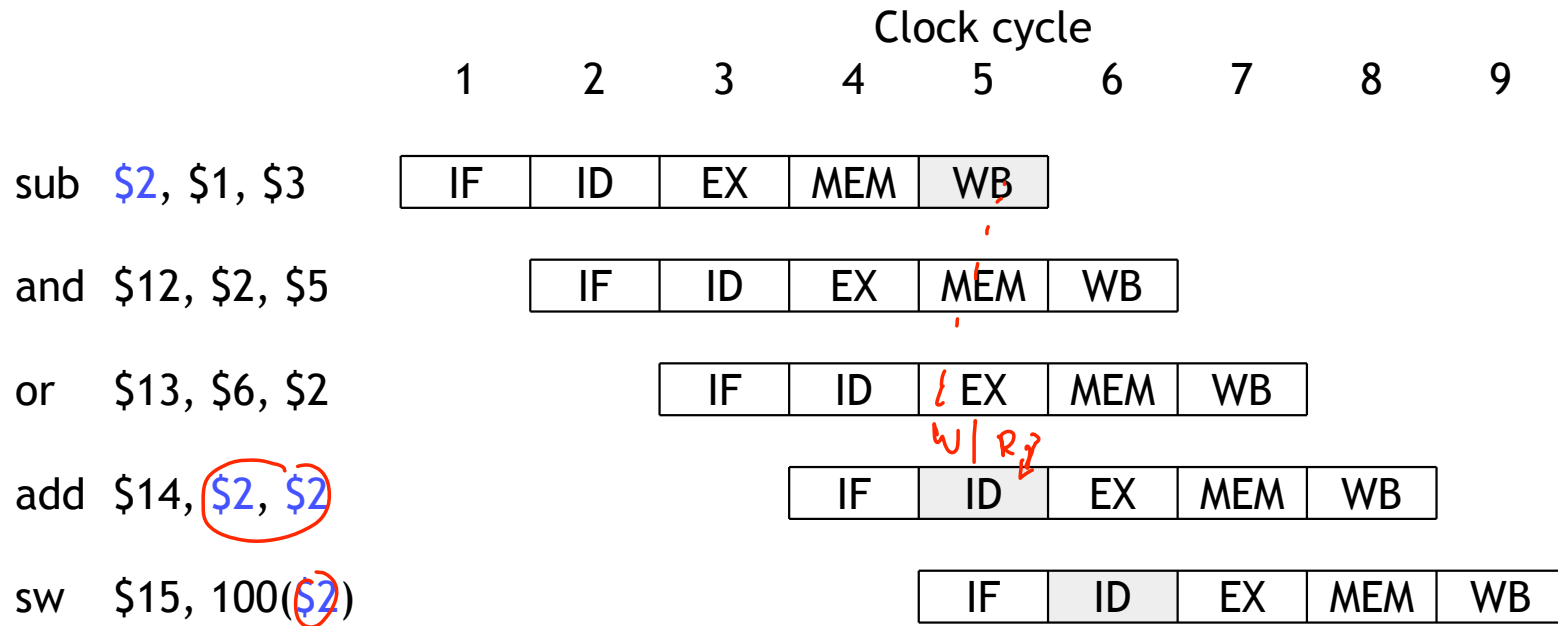
- There are several **dependencies** in this new code fragment.
  - The first instruction, SUB, stores a value into \$2.
  - That register is used as a source in the rest of the instructions.
- This is not a problem for the single-cycle and multicycle datapaths.
  - Each instruction is executed completely before the next one begins.
  - This ensures that instructions 2 through 5 above use the new value of \$2 (the sub result), just as we expect.
- How would this code sequence fare in our pipelined datapath?

# Data hazards in the pipeline diagram



- The SUB instruction does not write to register \$2 until clock cycle 5. This causes two **data hazards** in our current pipelined datapath.
  - The AND reads register \$2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of \$2, not the new one.
  - Similarly, the OR instruction uses register \$2 in cycle 4, again before it's actually updated by SUB.

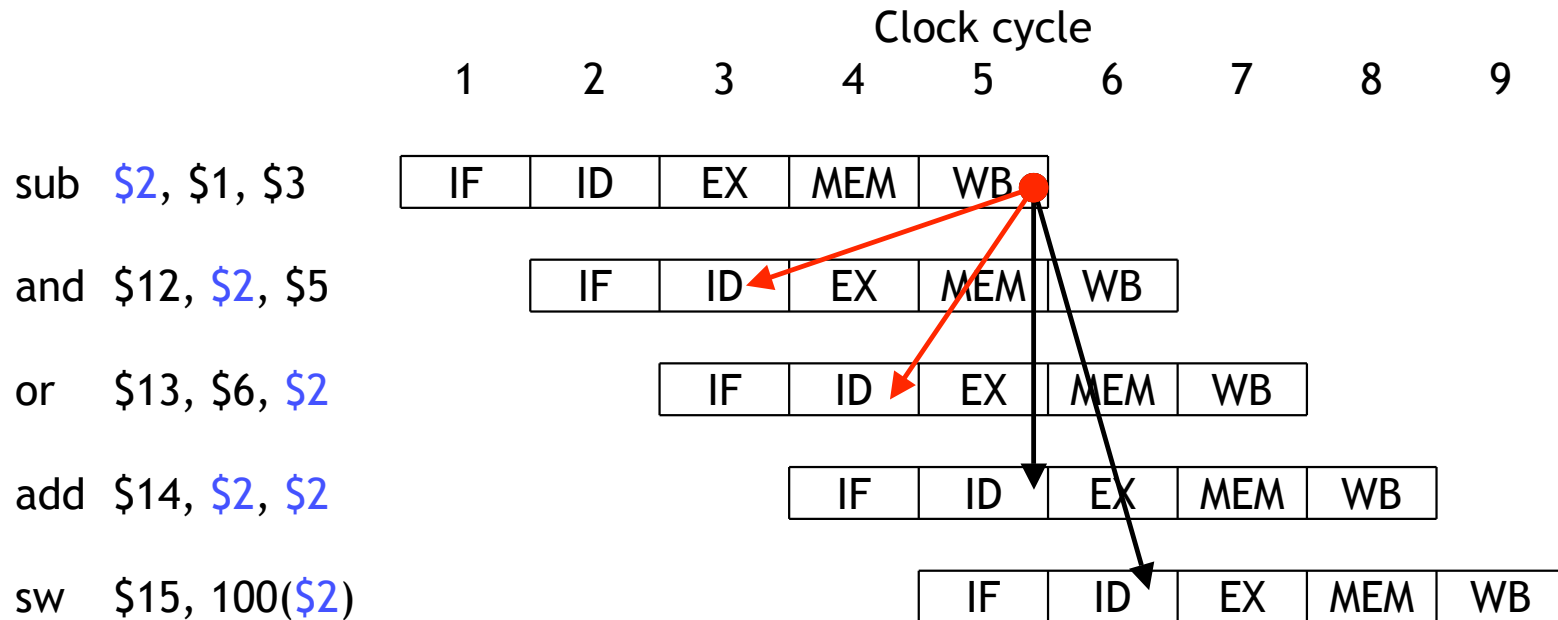
# Things that are okay



- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads \$2 after the SUB finishes.

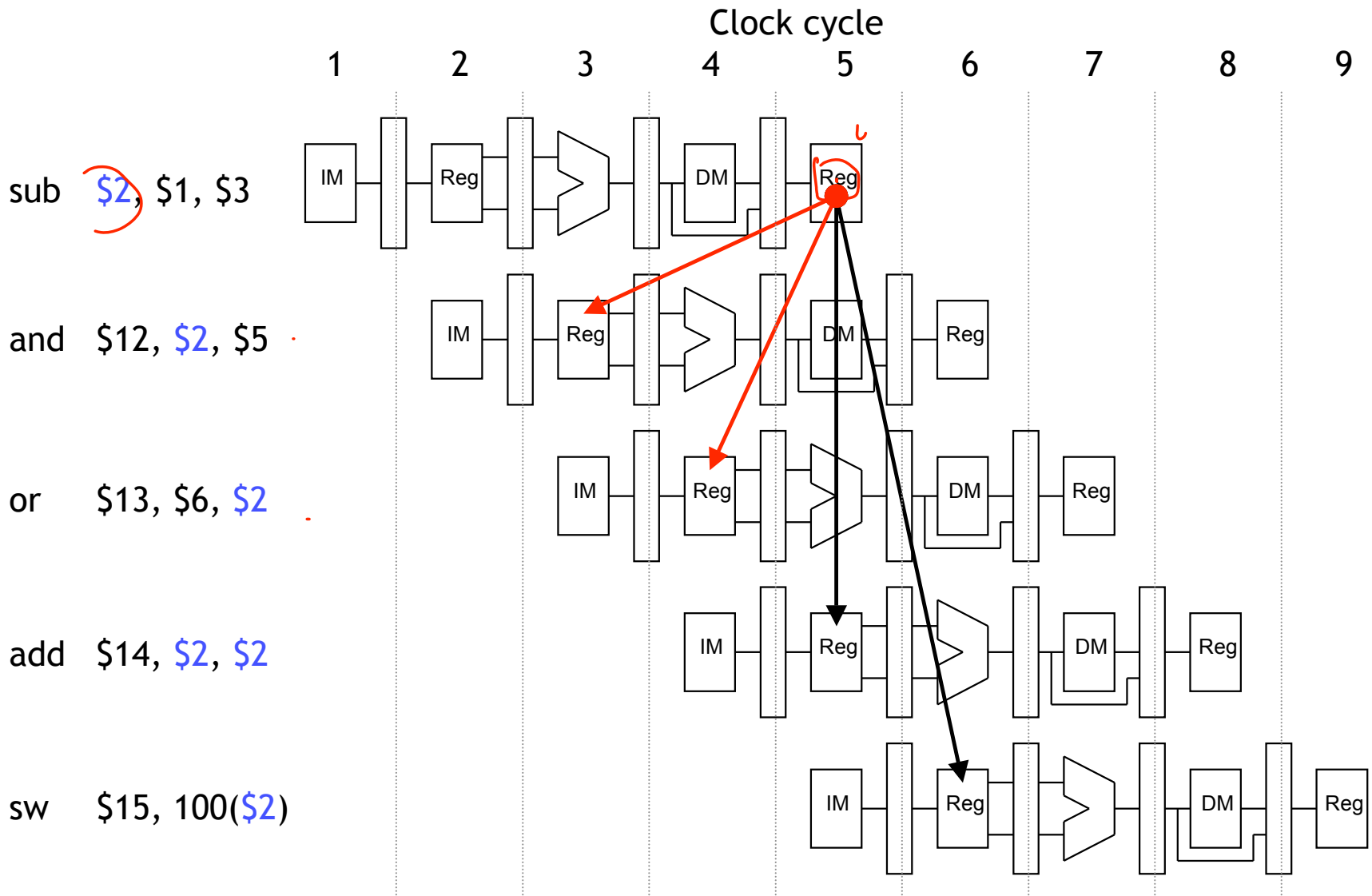


# Dependency arrows



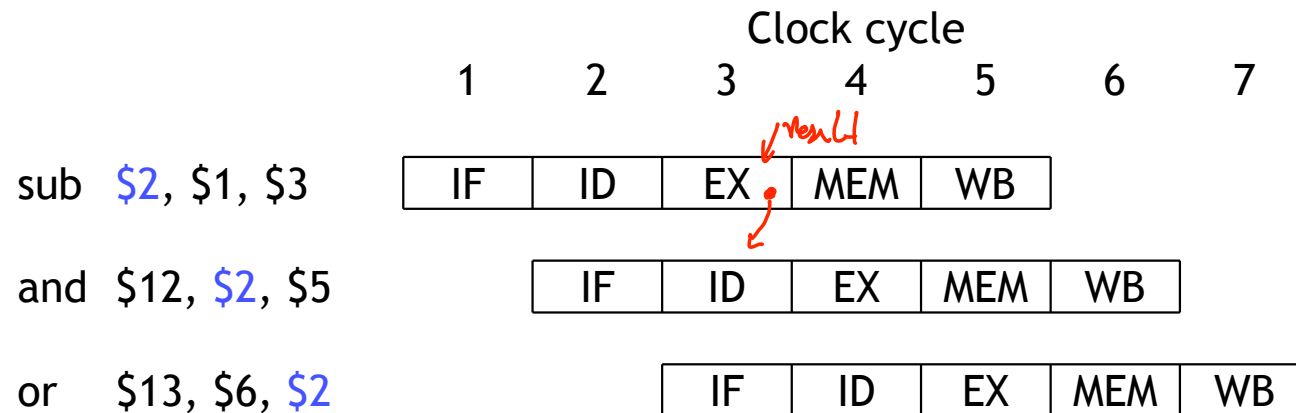
- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register \$2 is written.
  - The heads of the arrows show when \$2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions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

# A fancier pipeline diagram



## A more detailed look at the pipeline

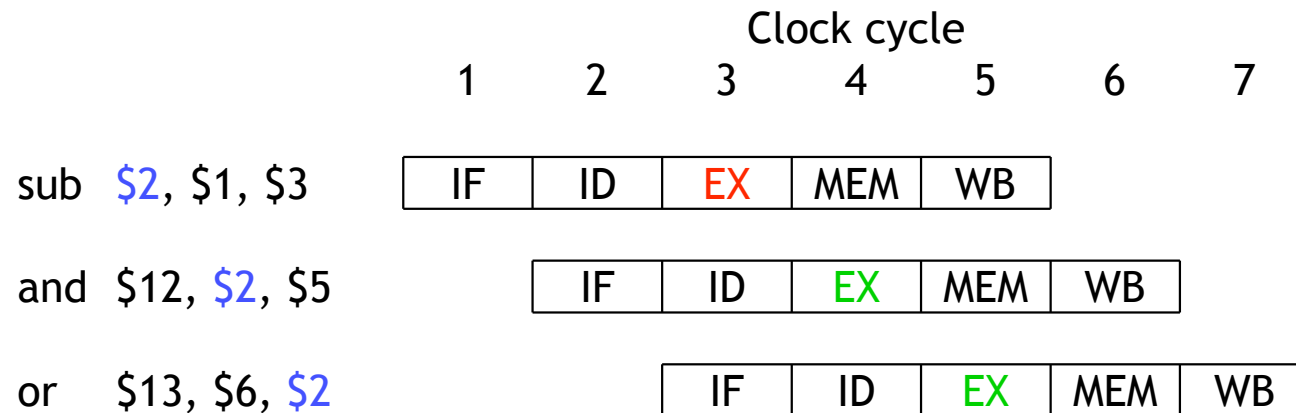
- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register \$2.
- When is the data actually produced and consumed?
- What can we do?



## A more detailed look at the pipeline

---

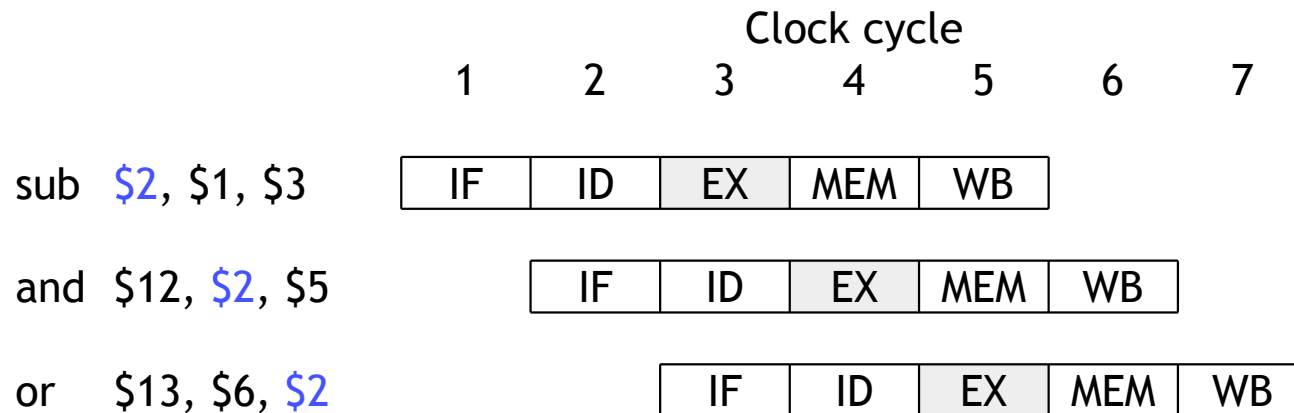
- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register \$2.
- Let's look at when the data is actually produced and consumed.
  - The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
  - The AND and OR need the new value of \$2 in their EX stages, during clock cycles 4-5 here.



## Bypassing the register file

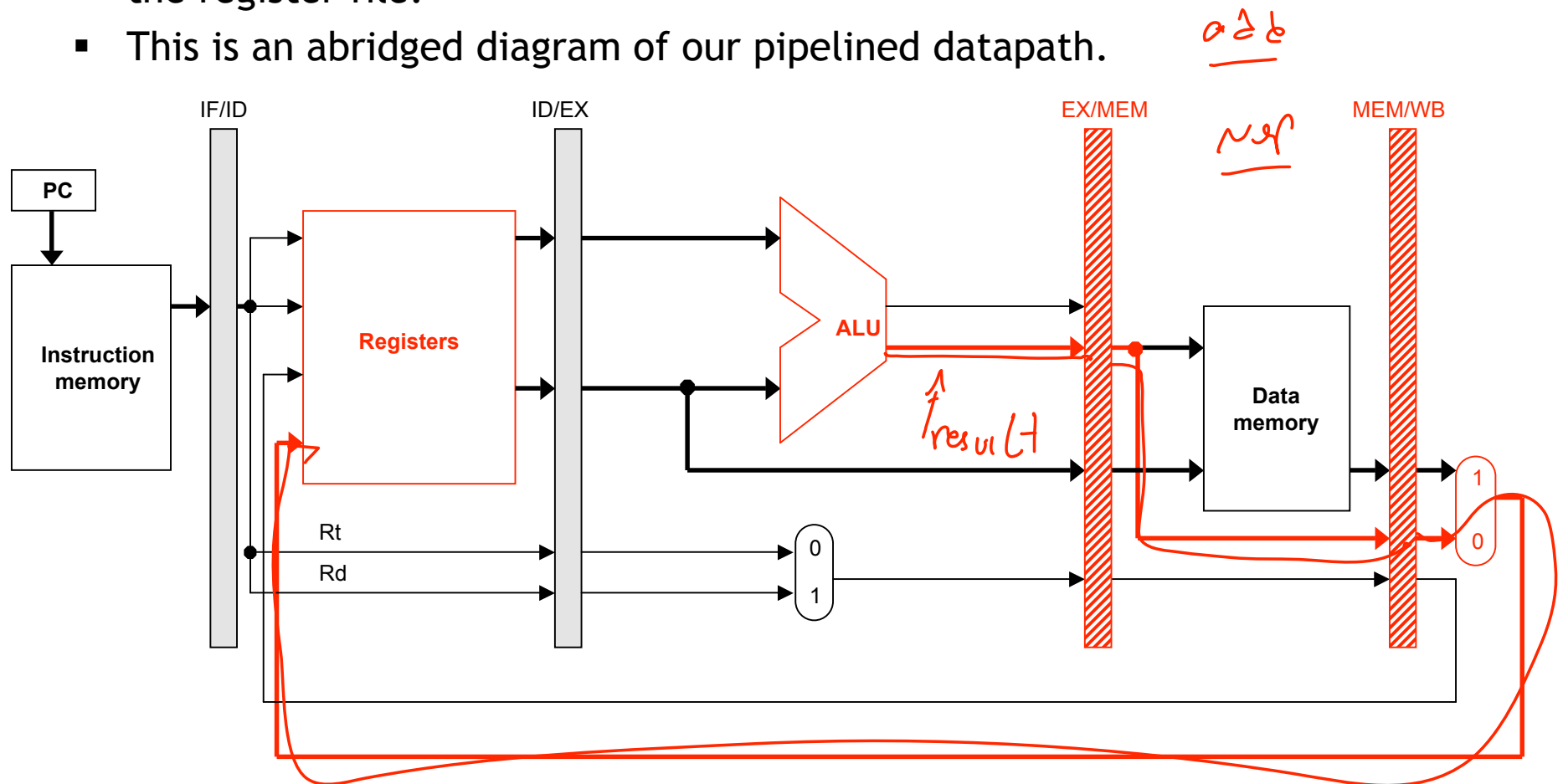
---

- 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.
  - Next time, we'll examine 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.



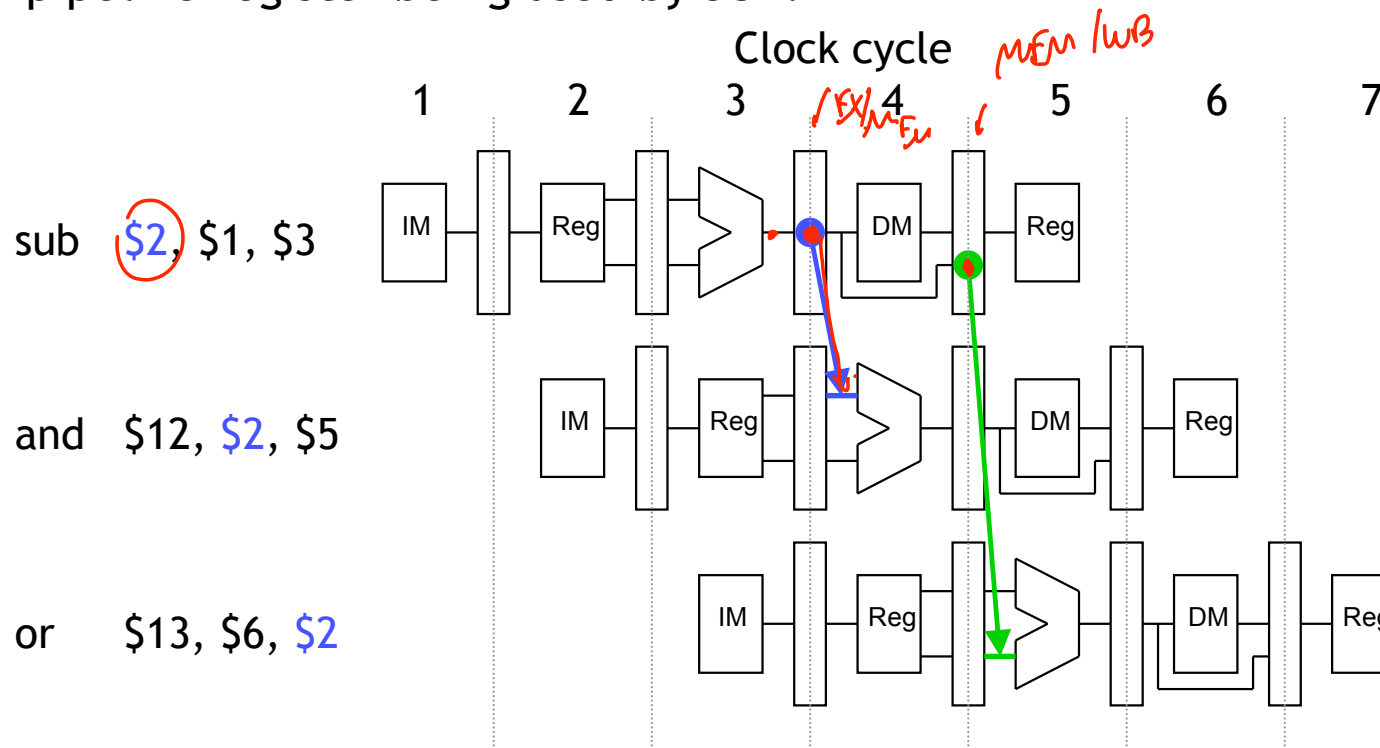
# Where to find the ALU result

- 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.



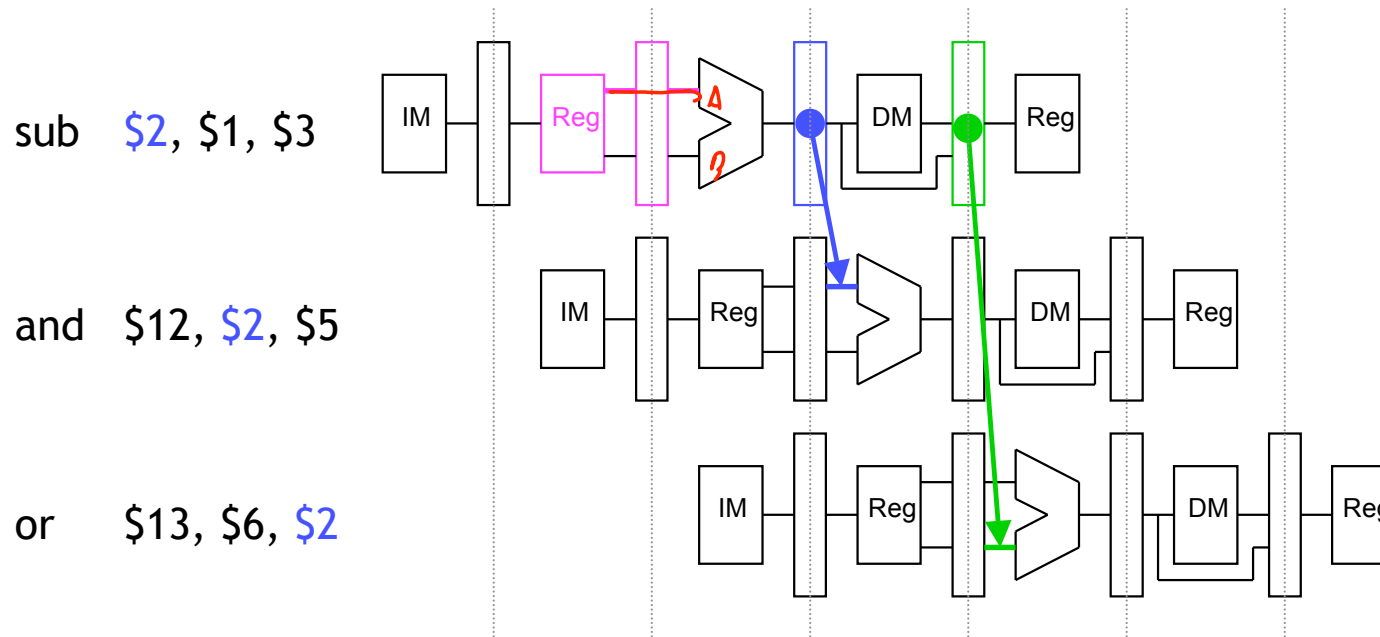
# Forwarding

- 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.



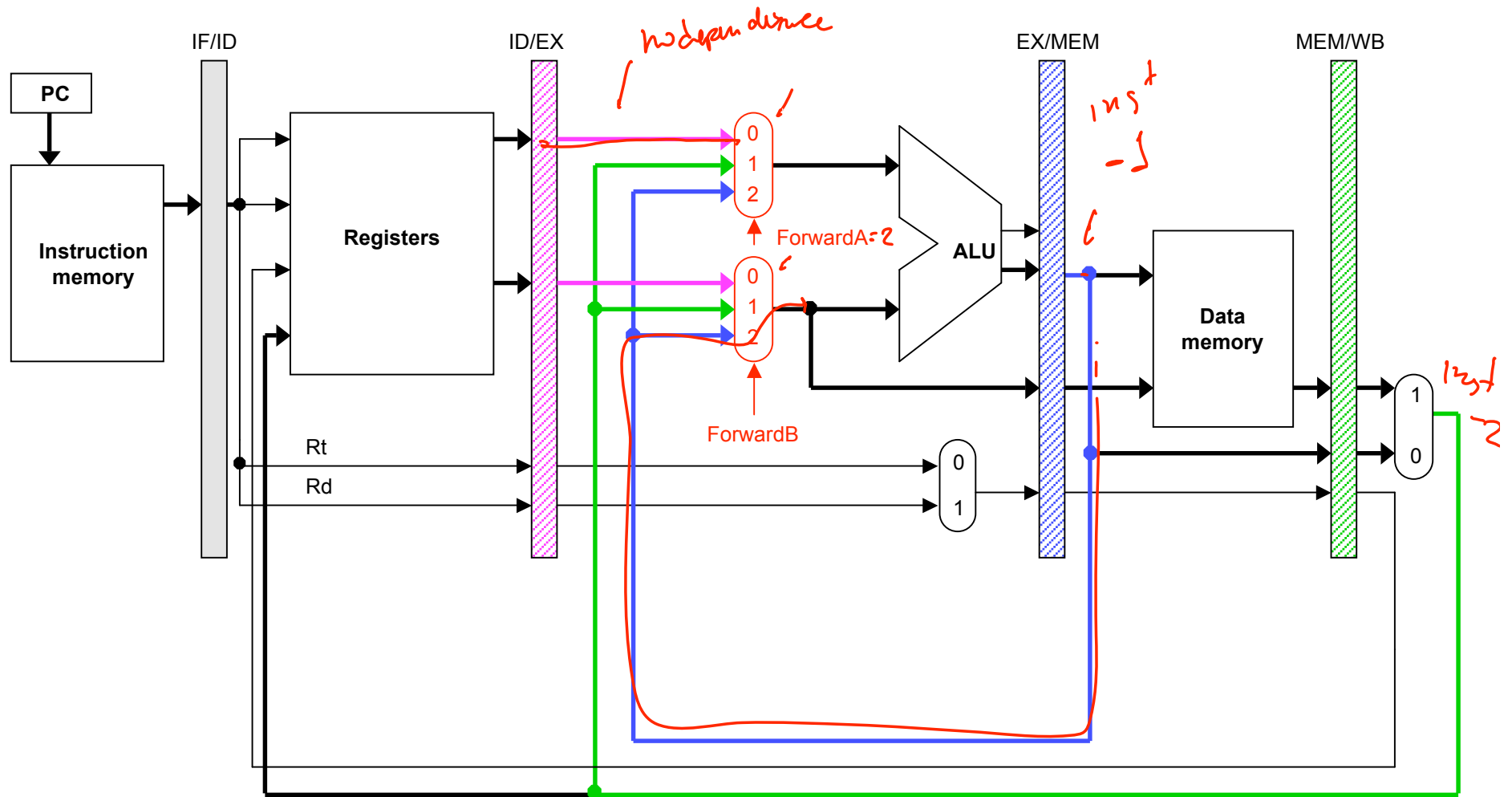
# Outline of forwarding hardware

- 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.



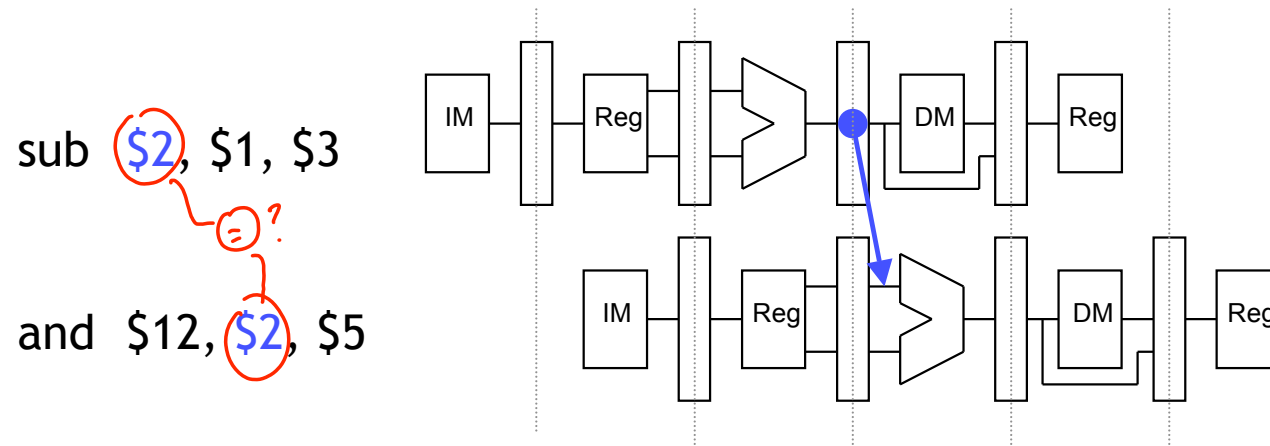


# Simplified datapath with forwarding muxes



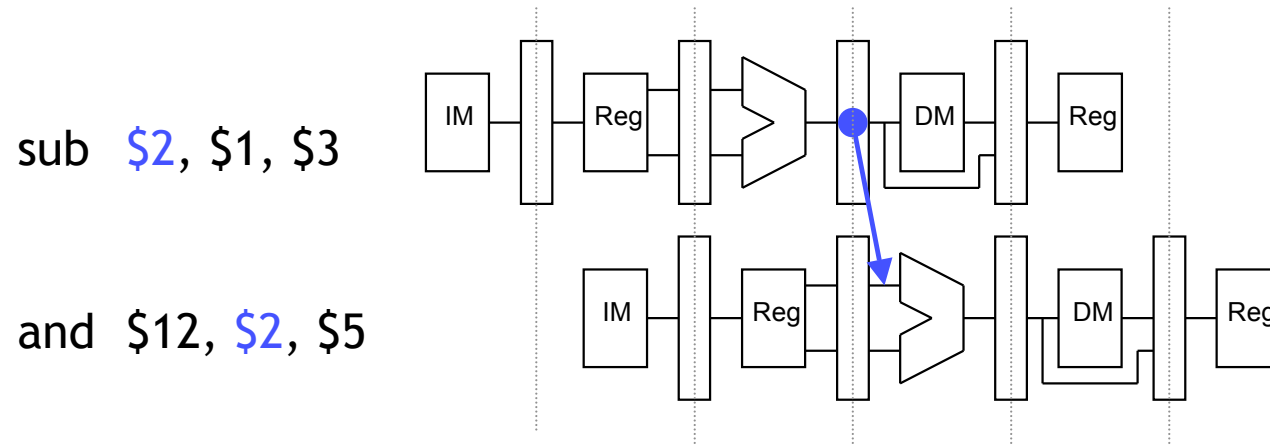
# Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?



## Detecting EX/MEM data hazards

- 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.
- There is an EX/MEM hazard between the two instructions below.



- 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

*EX/MEM RegWrite*

*RegWrite*

- 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

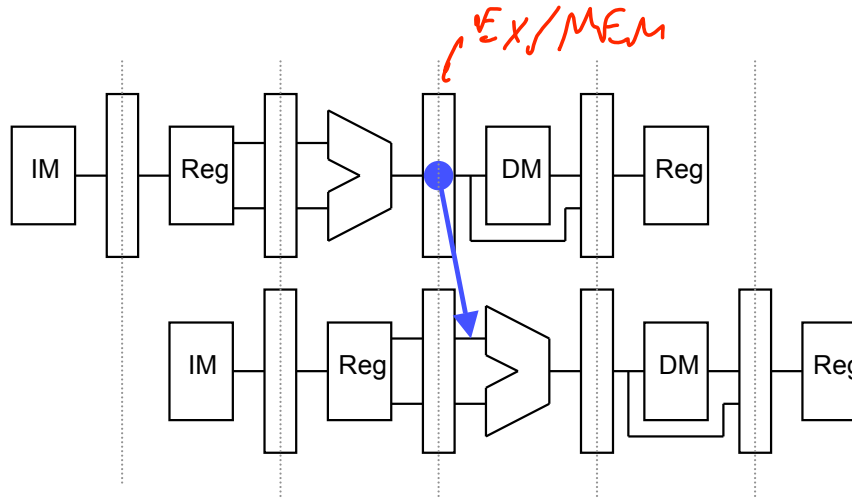
*R-type*

*rd = rs or rt*

*EX/MEM*

→ sub \$2, \$1, \$3

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

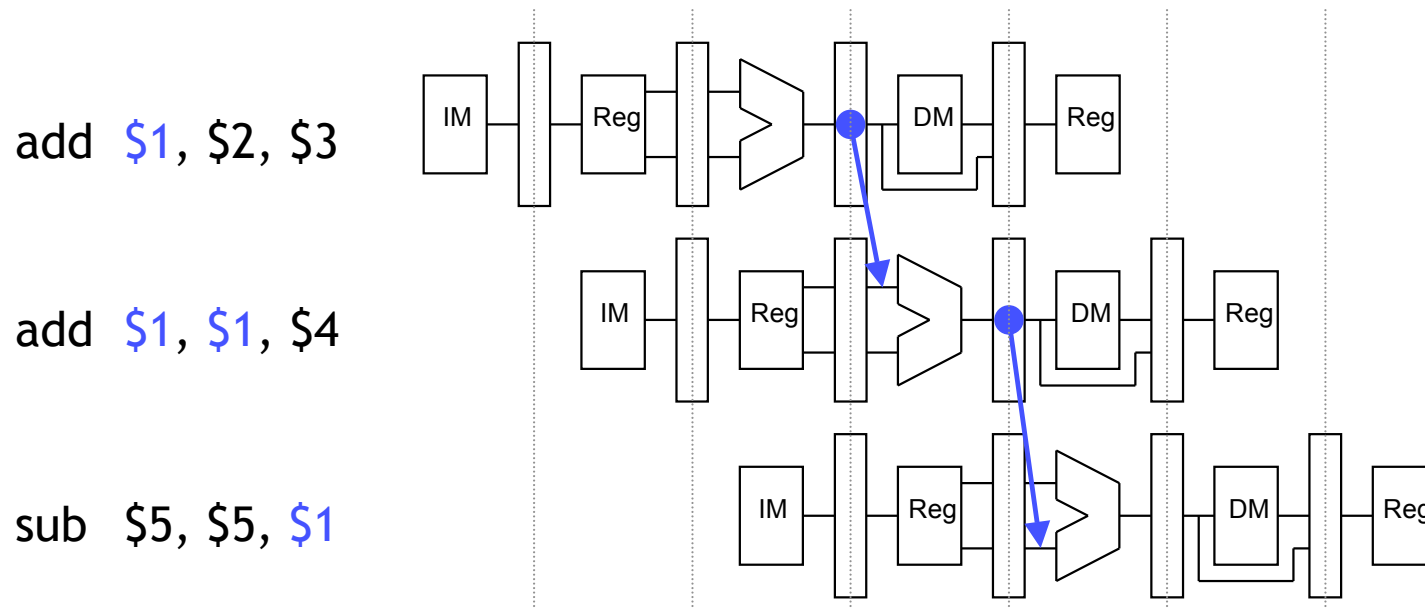


## Detecting MEM/WB data hazards

- 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.

x
v
  
 add \$1, \$2, \$3
   
 add \$1, \$1, \$4
   
 sub \$5, \$5, \$1

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



## MEM/WB hazard equations

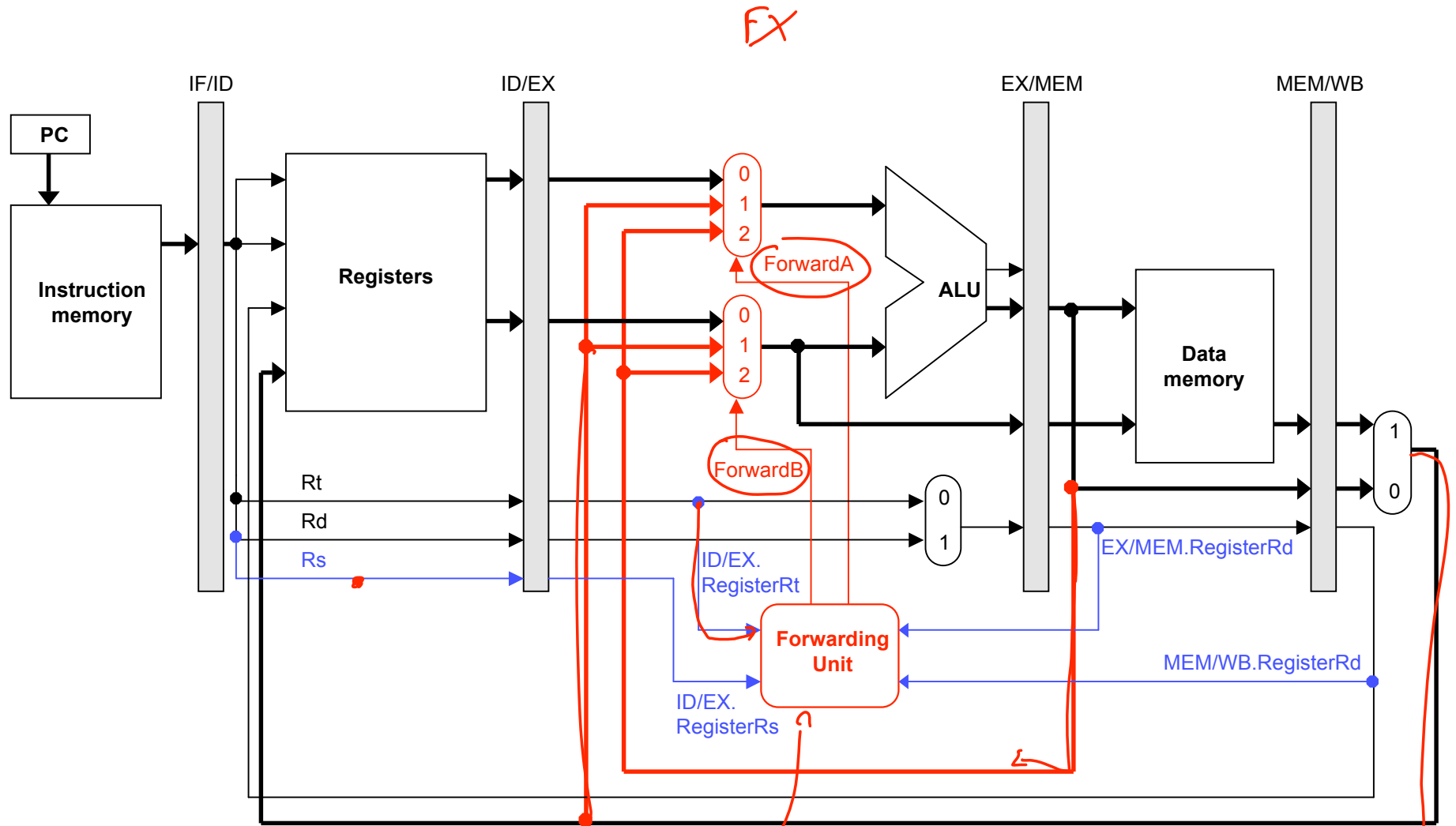
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

$$\begin{array}{cccc}
 \text{IF/ID} & \text{ID/EX} & \text{EX/MEM} & \text{MEM/WB} \\
 & & 0 & -1 \\
 & & & -2 \\
 & & & \uparrow
 \end{array}$$
 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



-1 -2 Reg writers

## 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 forwarding 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.



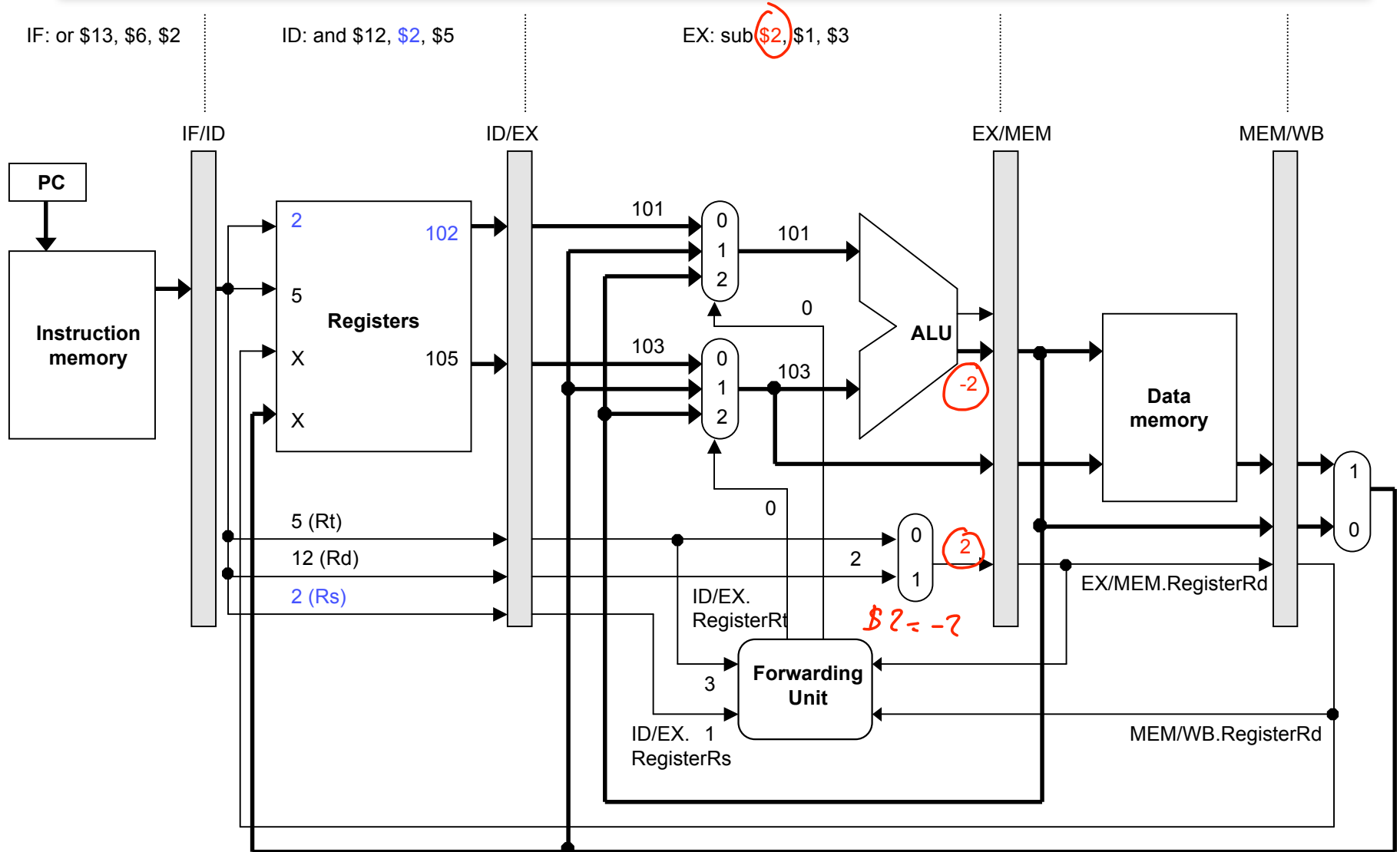
## Example

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

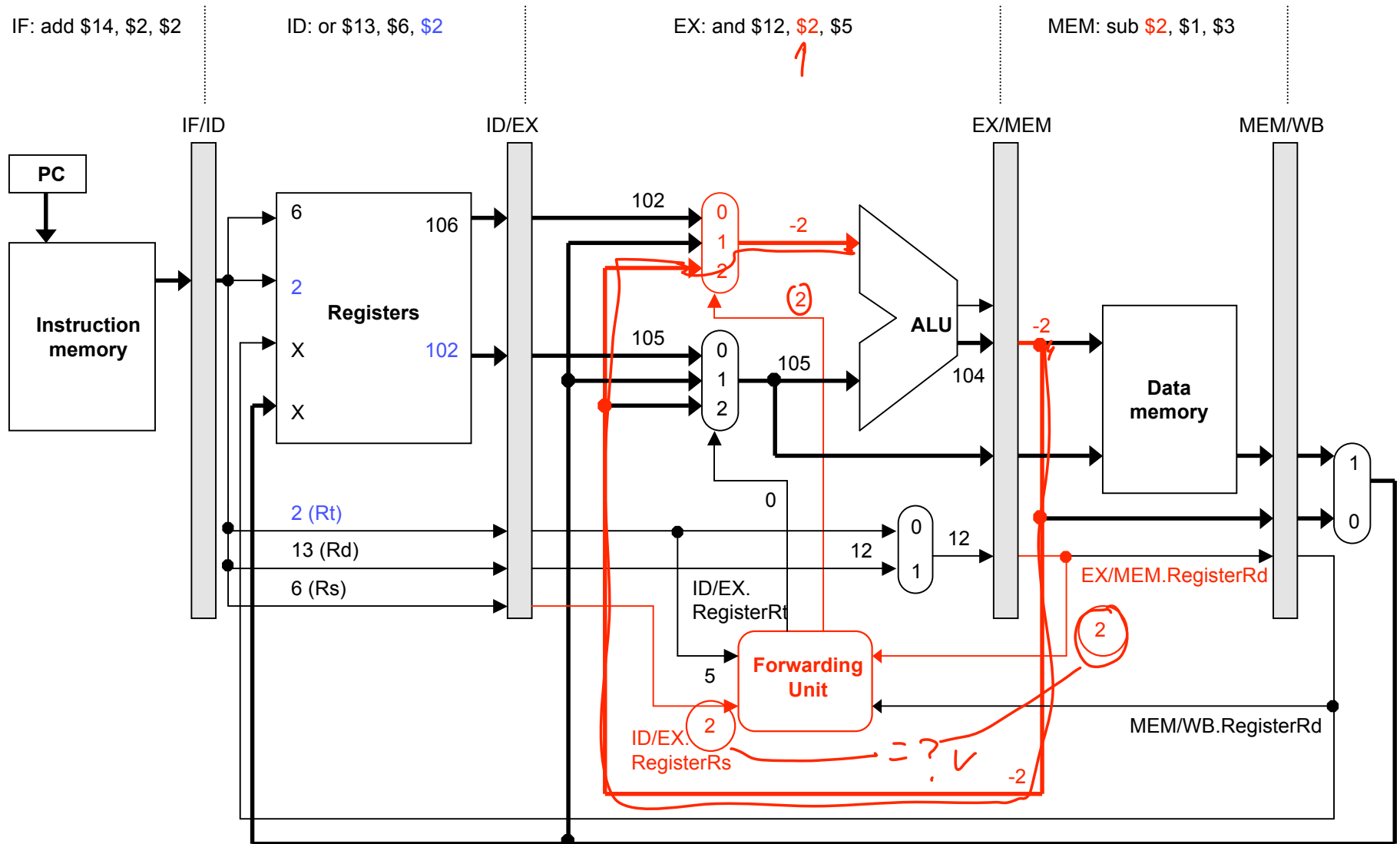
$$\begin{array}{r} \$1 \quad \$3 \\ \rightarrow \$2 = 101 - 103 = -2 \end{array}$$

- 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.

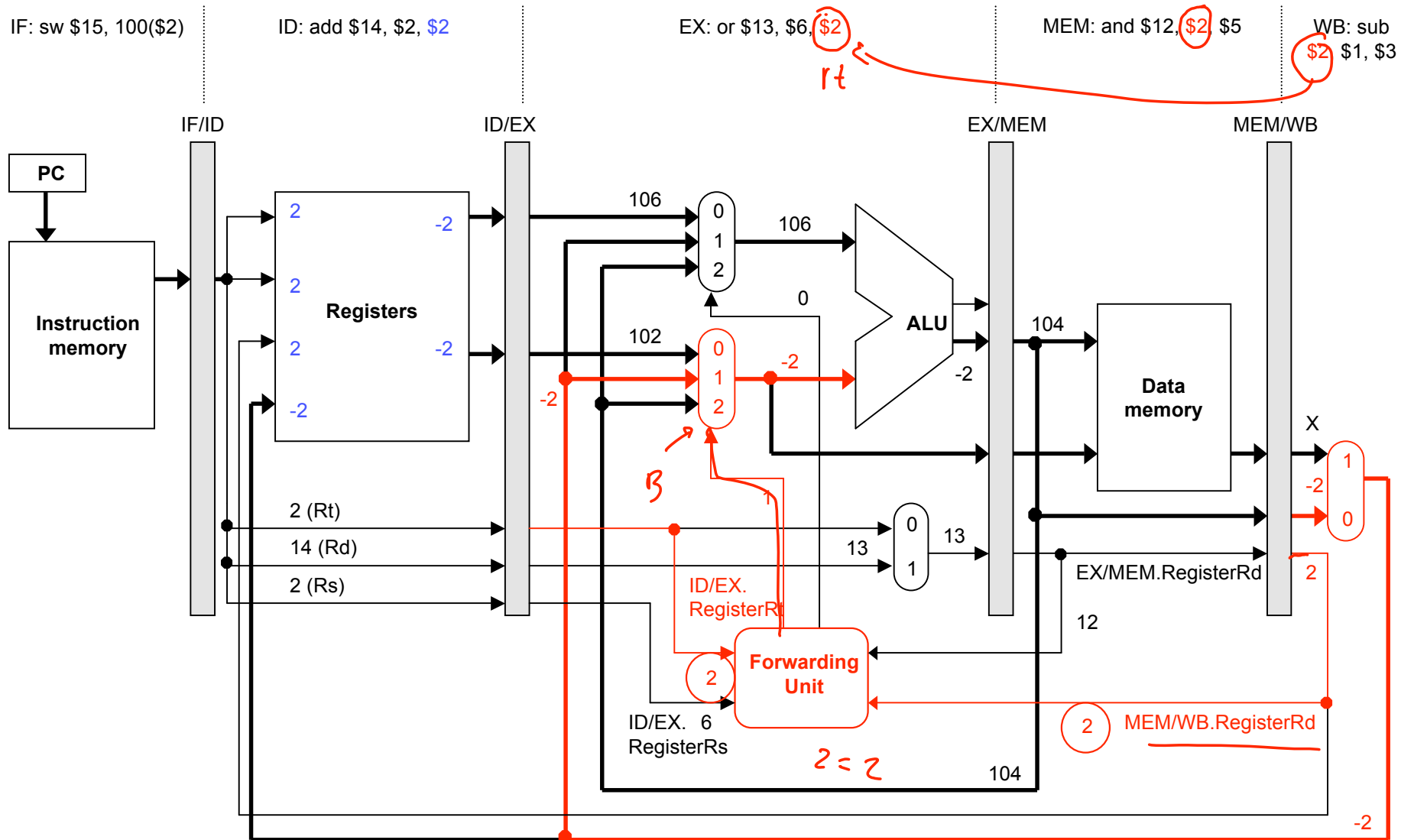
# Clock cycle 3



# Clock cycle 4: forwarding \$2 from EX/MEM



# Clock cycle 5: forwarding \$2 from MEM/WB

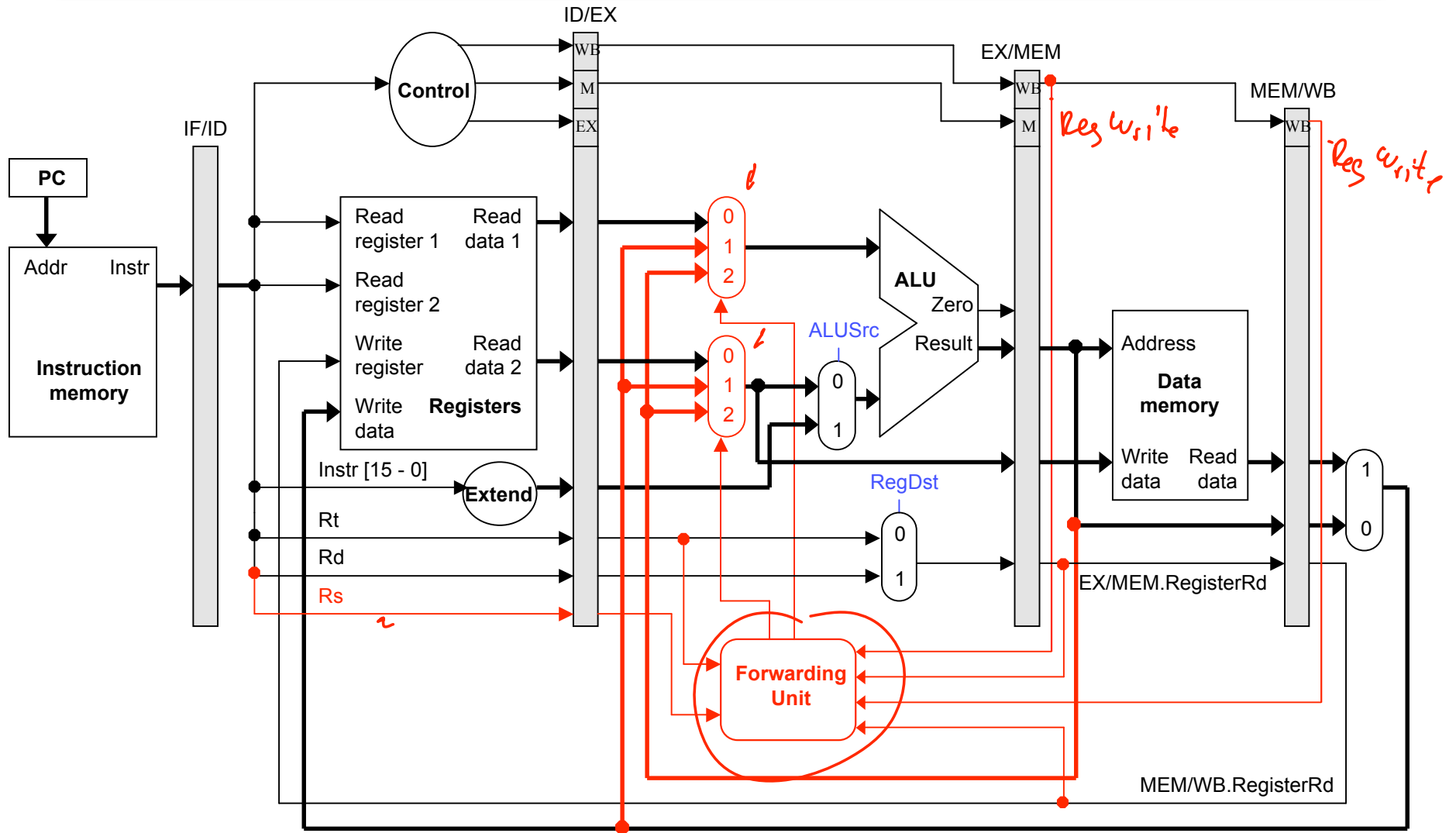


# Lots of data hazards

---

- 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.
  - 1 – 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.
  - 2 – 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

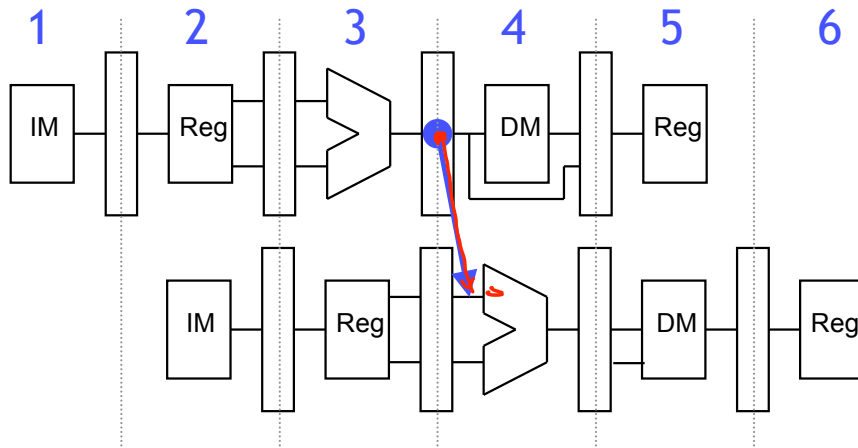


# What about stores?

- Two “easy” cases:

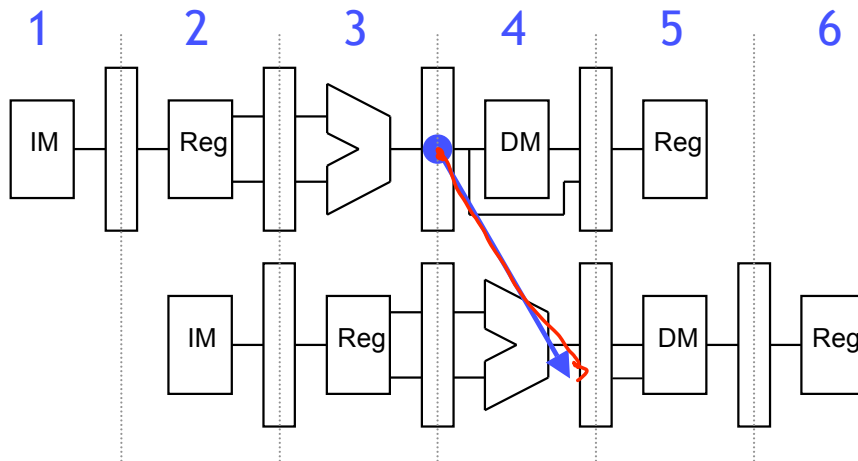
add (\$1, \$2, \$3

sw \$4, 0(\$1)

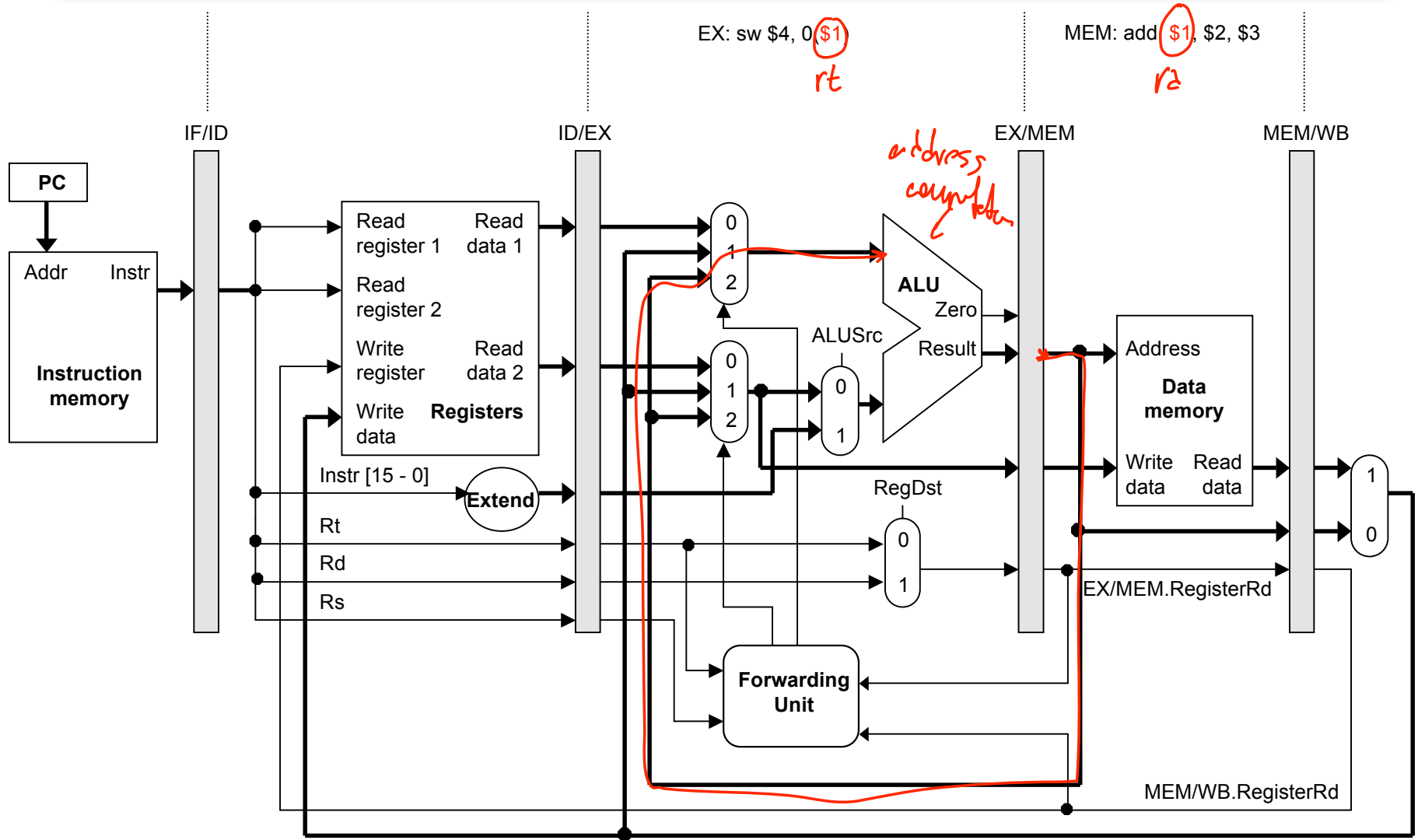


add (\$1, \$2, \$3

sw \$1, 0(\$4)

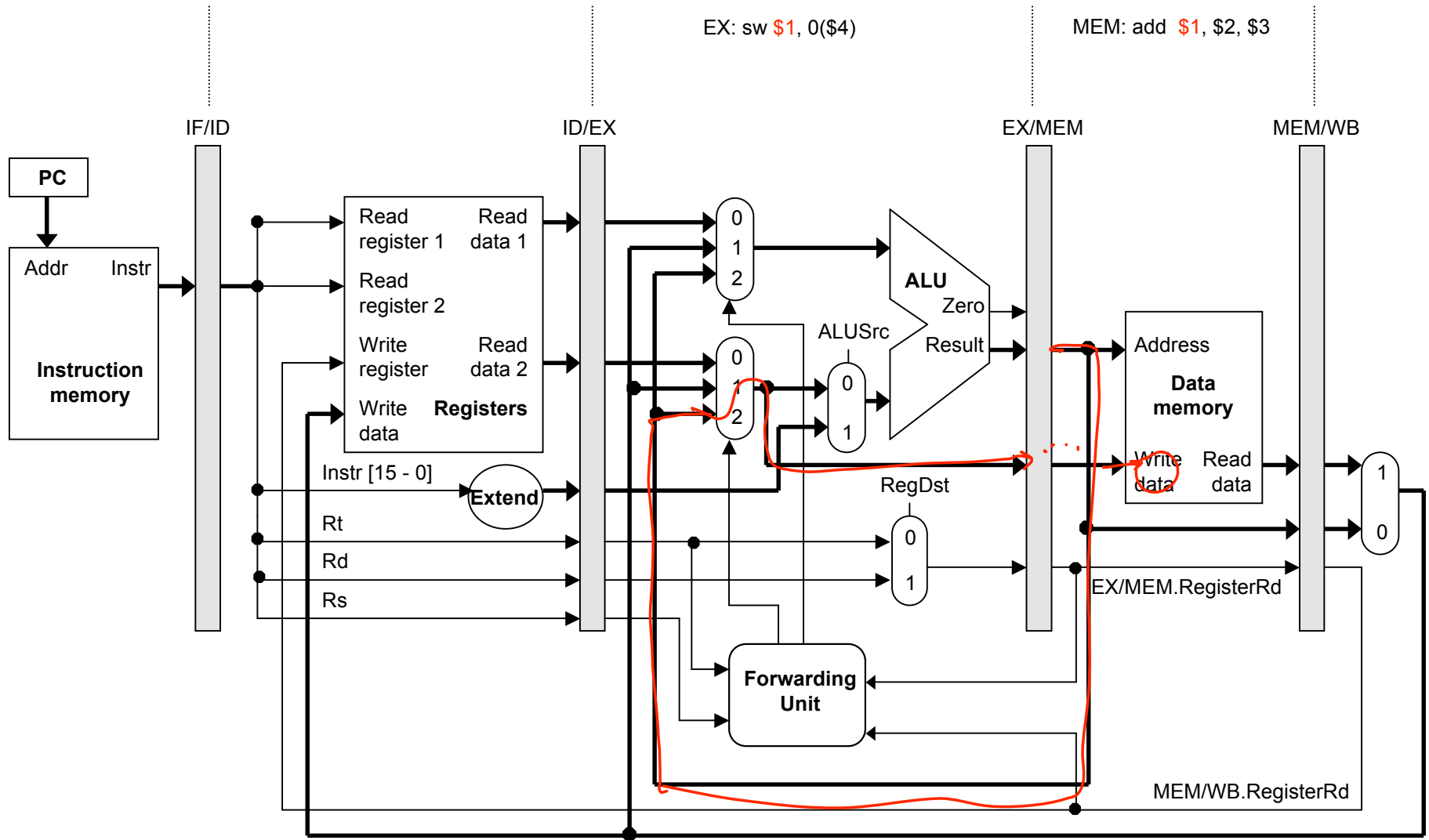


# Store Bypassing: Version 1



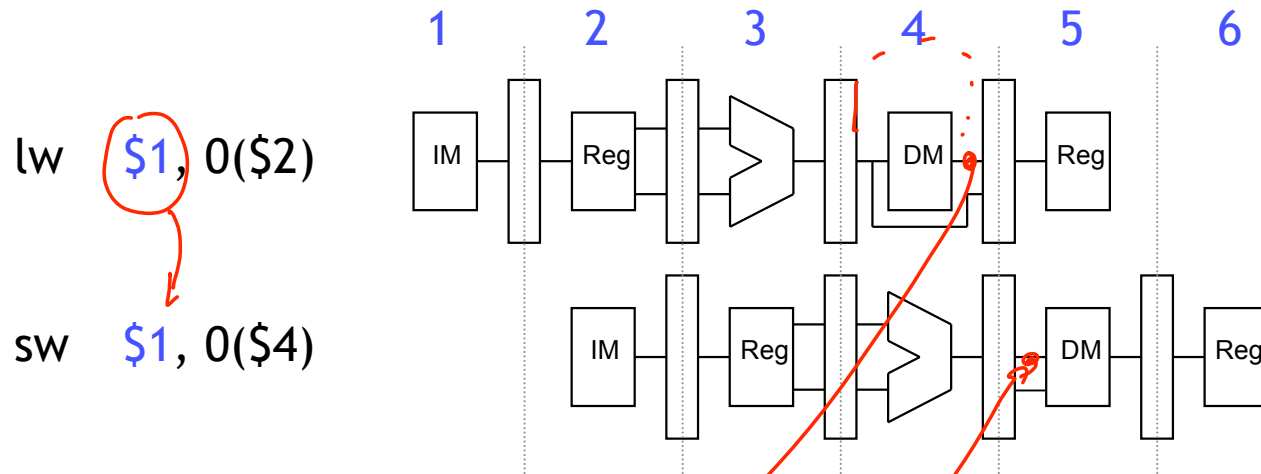


# Store Bypassing: Version 2



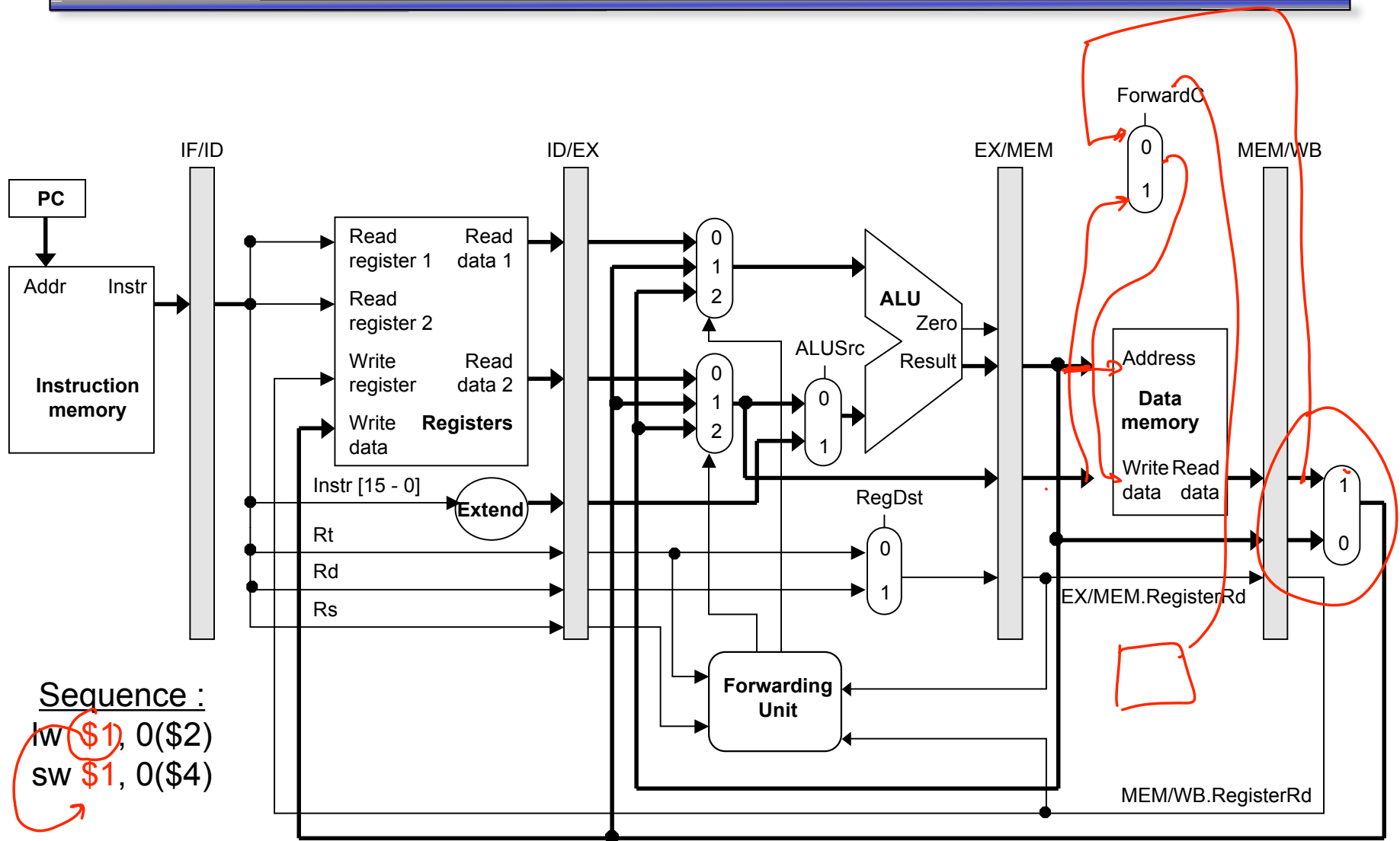
# What about stores?

- A harder case:



- 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



## 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, 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.

