CSEE 3827: Fundamentals of Computer Systems, Spring 2022

Lecture 12

Prof. Dan Rubenstein (danr@cs.columbia.edu)

Agenda (P&H 4.5-4.8)

- Pipelining
- Hazards

Single-Cycle CPU Performance Issues

- Recall: In single cycle, longest latency instruction lower bounds clock period
  - Clock must allow all data to flow through CPU (large sequential circuit) so that flip-flops (registers in register file), latches (in memory) record the right values when written to.
  - Basic flow:
    - instruction memory → register file → ALU → data memory → register file
- Not (easily) feasible to vary clock period for different instructions

Single-Cycle CPU Performance Issues

- We will further improve performance by pipelining
- Pipelining Idea: instruction processing can be broken into stages
  - each stage takes a fraction of time of the instruction
  - have each stage use a clock cycle (instead of entire instruction)
    - so instructions use several clock cycles
    - clock cycles can run faster
    - instructions processed simultaneously, but staggered (simultaneous instructions are at different stages)

Pipelining Laundry Analogy

- Analogy works when you own 1 washer and 1 dryer
- Non-pipeline: perform entire task before starting next
- Pipelined: 4 workers, one handles each job, can work in parallel

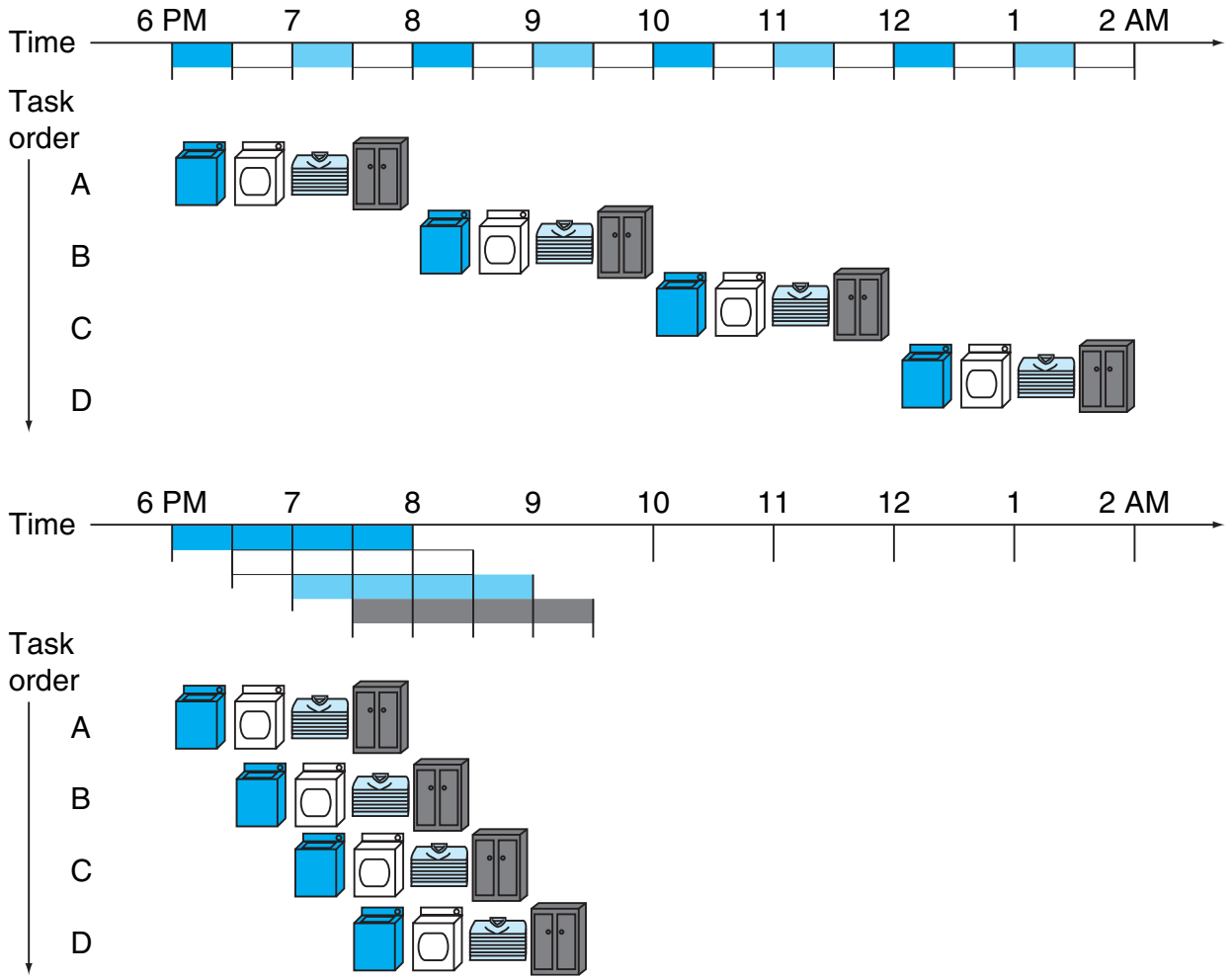
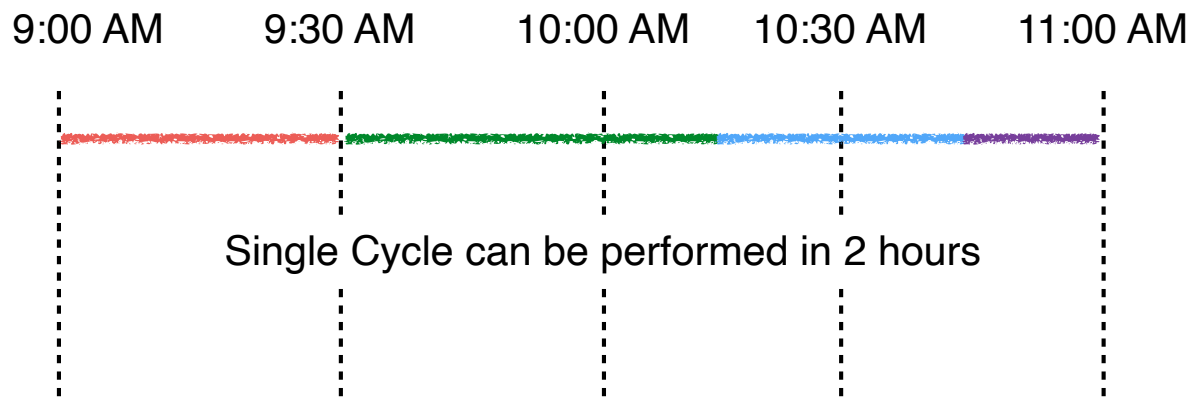


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

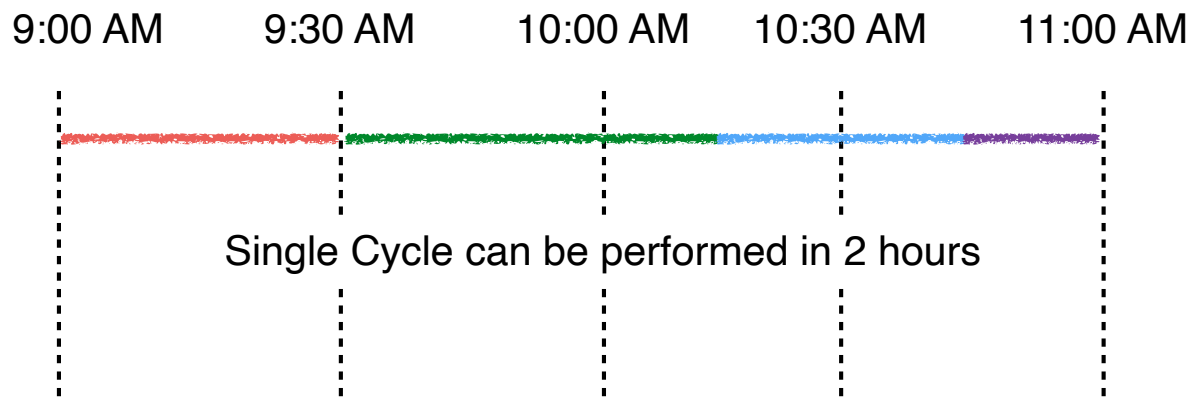
Suppose Stages require different times

- Suppose:
  - Wash: 30 min
  - Dry: 45 min
  - Fold: 30 min
  - Put Away: 15 min

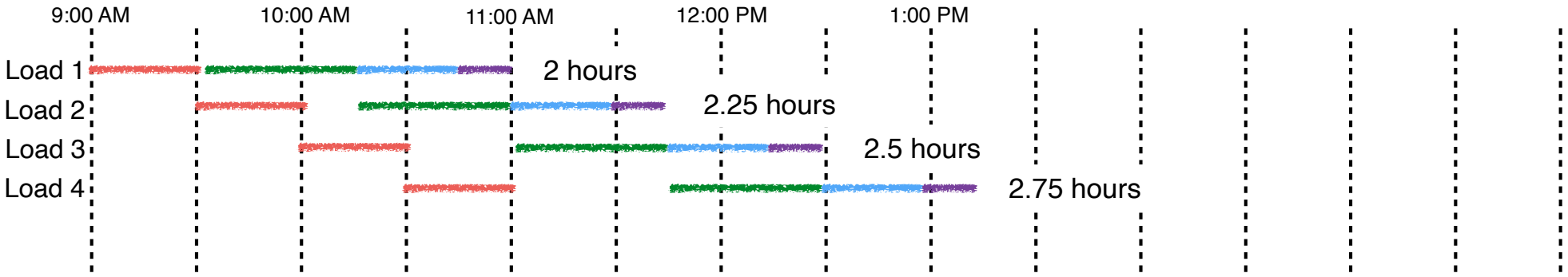


Suppose Stages require different times

- Suppose:
  - Wash: 30 min
  - Dry: 45 min
  - Fold: 30 min
  - Put Away: 15 min

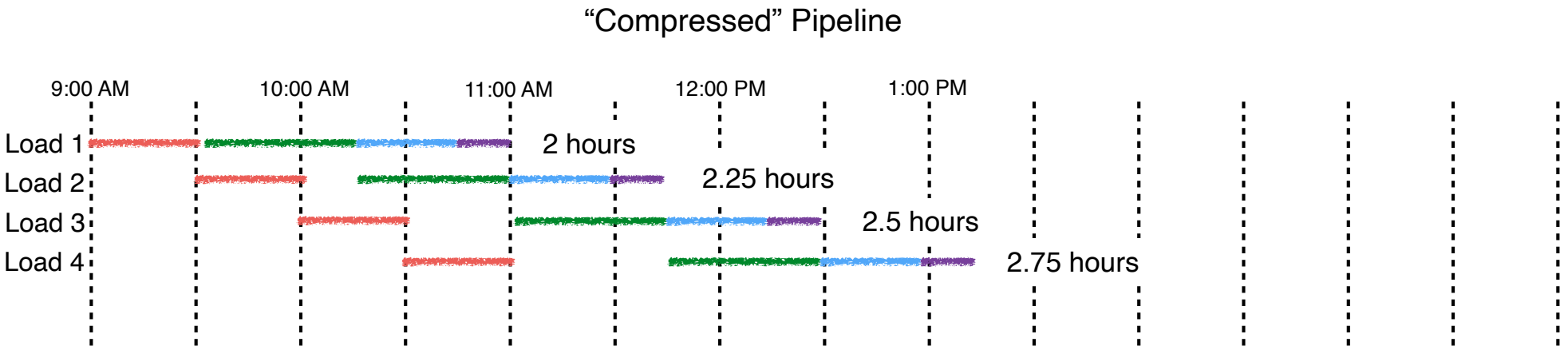


- Pipelining:
  - Eventually constrained by longest stage (must wait for dryer)
  - Individual laundry cycle may take longer to complete

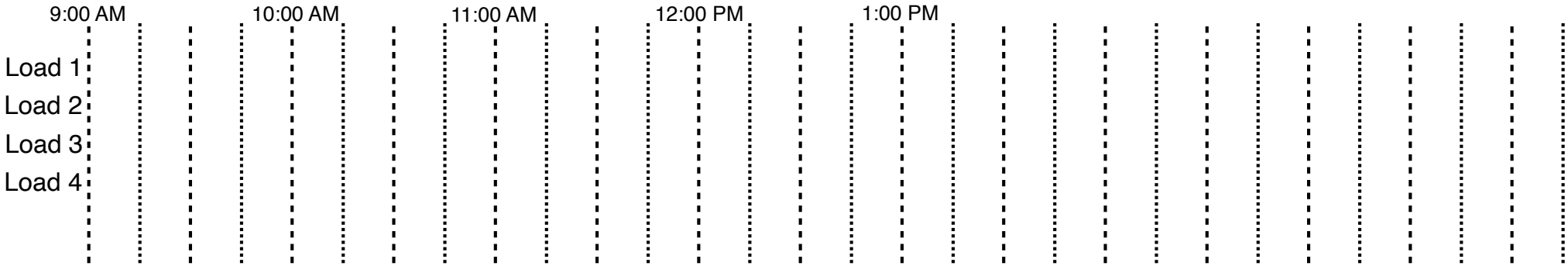


- However, 4 loads take less overall time (4.25 hours vs. 8 hours)
- Pipelining's parallelism makes allows doing many "loads" in less time

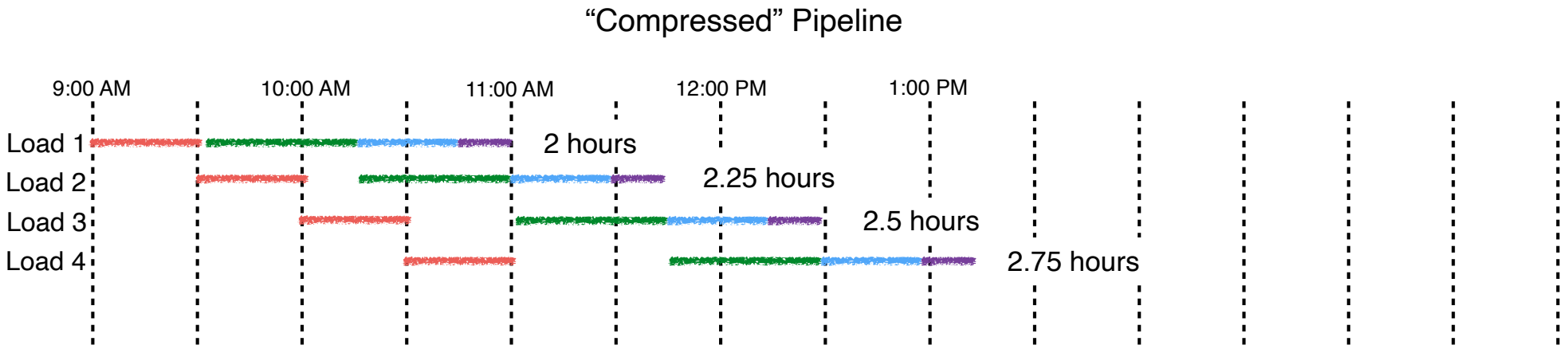
Using a fixed clock cycle



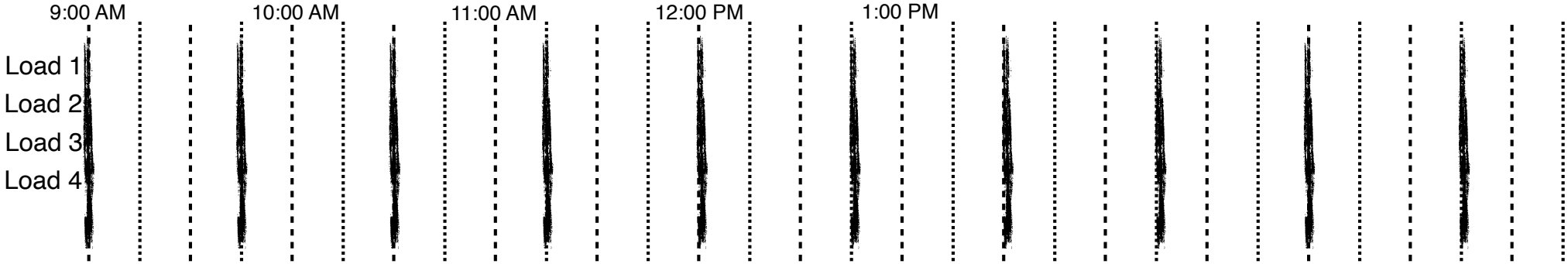
- Assign a fixed clock cycle (45 min) that is long enough to accommodate slowest stage (Dry)
  - Each stage given this amount of time to complete, some stages finish early



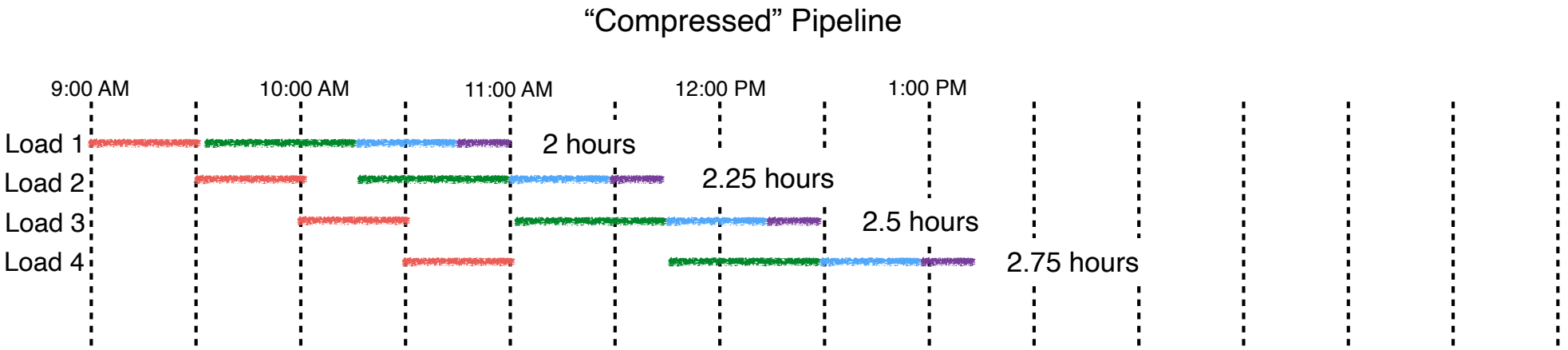
Using a fixed clock cycle



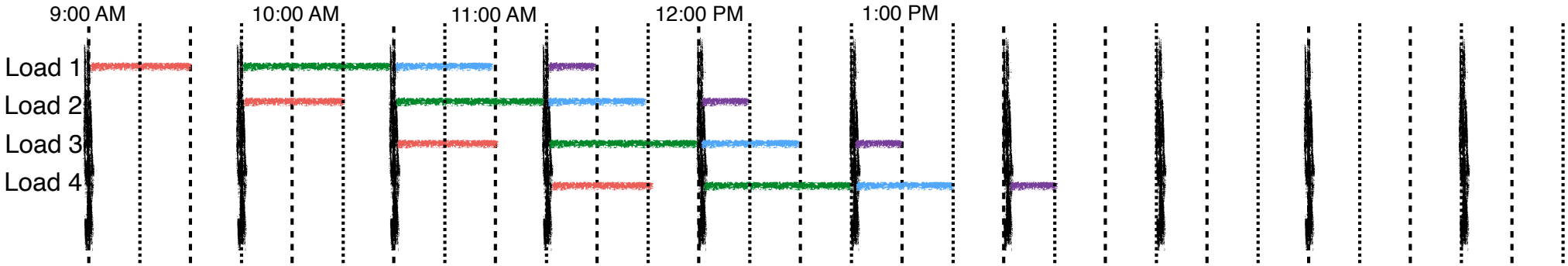
- Assign a fixed clock cycle (45 min) that is long enough to accommodate slowest stage (Dry)
  - Each stage given this amount of time to complete, some stages finish early



Using a fixed clock cycle



- Assign a fixed clock cycle (45 min) that is long enough to accommodate slowest stage (Dry)
  - Each stage given this amount of time to complete, some stages finish early



- Each load requires 4x45 min (3 hours) to complete

Compare these three versions

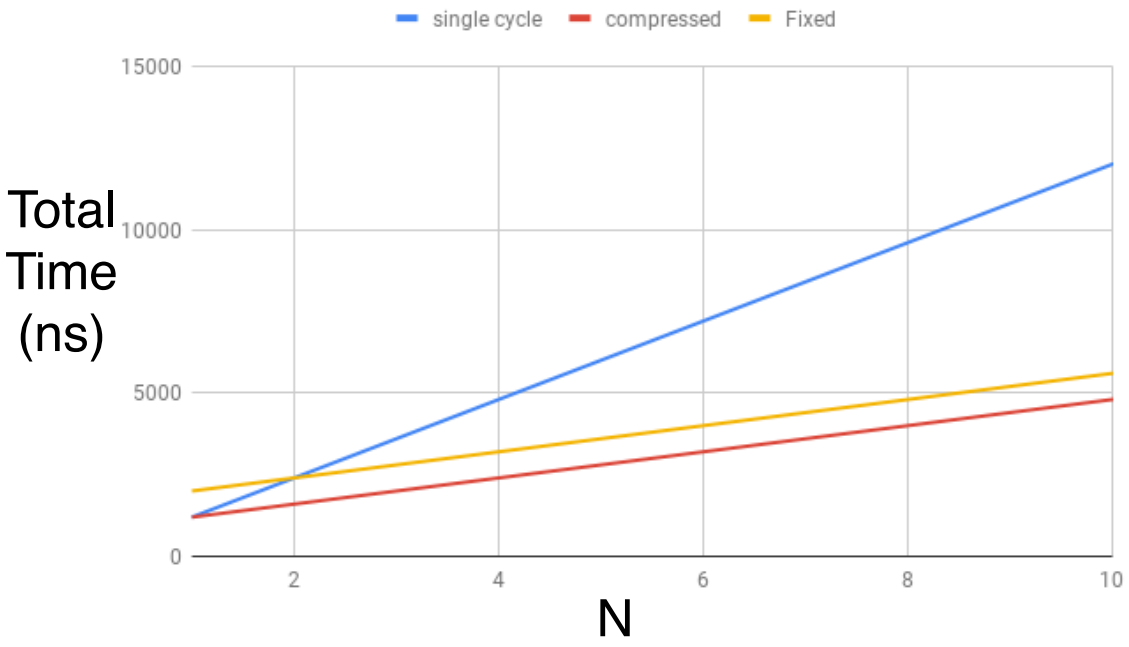
- Assume there are k stages, where the i^{th} stage requires time T_i
- Laundry example:
  - k = 4 (wash, dry, fold, store)
  - T_0=0.5, T_1=0.75, T_2=0.5, T_3=0.25
  - $\sum T_i = 2$, $\max_i\{T_i\} = 0.75$

Arch	Single Cycle	"Compressed" pipeline	Fixed Cycle Pipeline
Time to complete cycle	$\sum T_i$	growing	k $\max_i\{T_i\}$
Time to complete N "instructions"	$N\sum T_i$	$\sum T_i$+(N-1) $\max_i\{T_i\}$	(N+k-1) $\max_i\{T_i\}$

Compare these three versions

Arch	Single Cycle	"Compressed" pipeline	Fixed Cycle Pipeline
Time to complete cycle	$\sum T_i$	growing	$k \max_i\{T_i\}$
Time to complete N "instructions"	$N\sum T_i$	$\sum T_i+(N-1) \max_i\{T_i\}$	$(N+k-1) \max_i\{T_i\}$

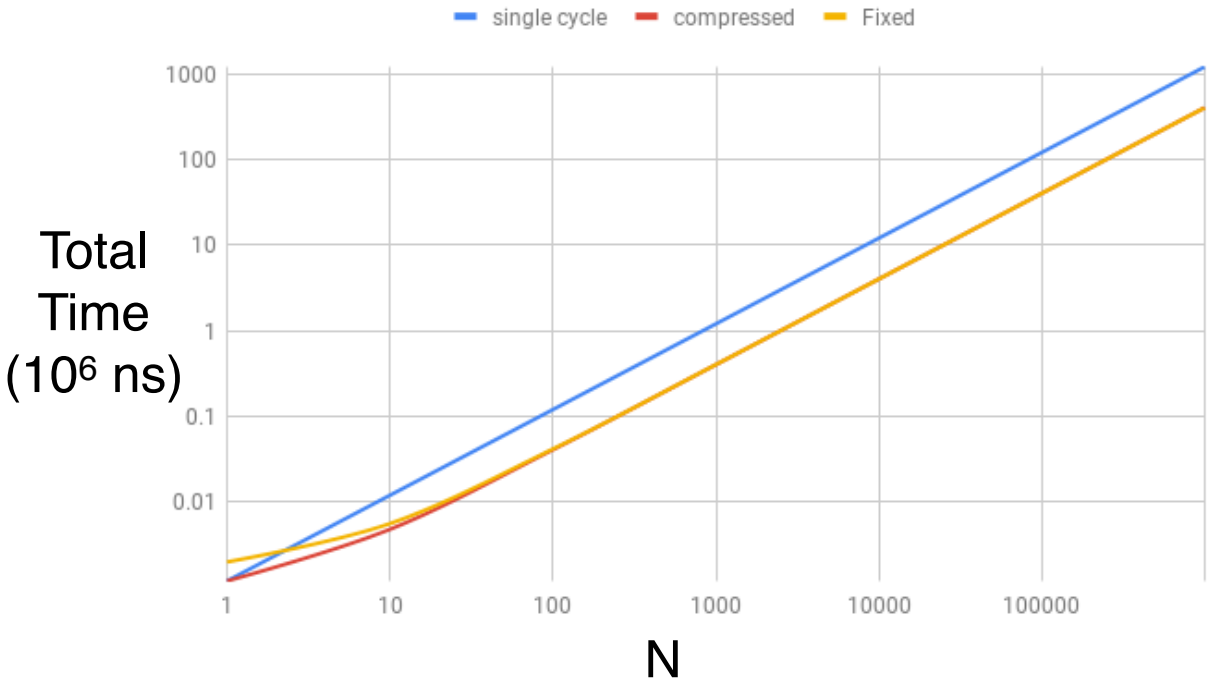
- Example with k=5
  - T_1=100 ns
  - T_2=200 ns
  - T_3=200 ns
  - T_4=400 ns
  - T_5=300 ns



For 1 instruction, single cycle or compressed is best, but single cycle quickly becomes slower as N increases

Compare these three versions

Arch	Single Cycle	"Compressed" pipeline	Fixed Cycle Pipeline
Time to complete cycle	$\sum T_i$	growing	$k \max_i\{T_i\}$
Time to complete N "instructions"	$N\sum T_i$	$\sum T_i+(N-1)\max_i\{T_i\}$	$(N+k-1)\max_i\{T_i\}$

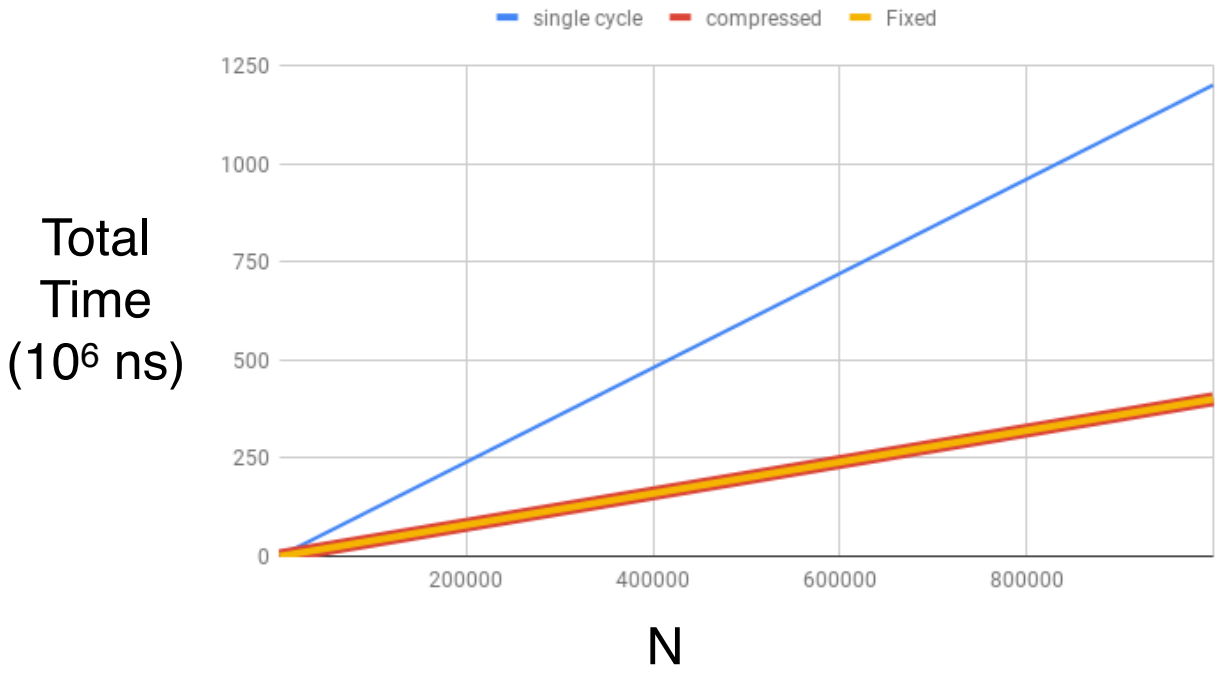


- Example with k=5
  - T_1=100 ns
  - T_2=200 ns
  - T_3=200 ns
  - T_4=400 ns
  - T_5=300 ns

Log-Log scale plot: Fixed and Compressed Pipeline about the same for large N, Single Cycle much worse

Compare these three versions

Arch	Single Cycle	"Compressed" pipeline	Fixed Cycle Pipeline
Time to complete cycle	$\sum T_i$	growing	$k \max_i\{T_i\}$
Time to complete N "instructions"	$N\sum T_i$	$\sum T_i+(N-1)\max_i\{T_i\}$	$(N+k-1)\max_i\{T_i\}$



- Example with k=5
  - T_1=100 ns
  - T_2=200 ns
  - T_3=200 ns
  - T_4=400 ns
  - T_5=300 ns

Not logscale plot: Fixed and Compressed Pipeline about the same for large N, Single Cycle much worse

Pipelining laundry helps slobs too...

- Even if individual instructions might take less time, pipelining still beneficial because:
  - A: can't beat the savings due to parallelism across stages
  - B: usually still have to process instructions at the rate of the slowest

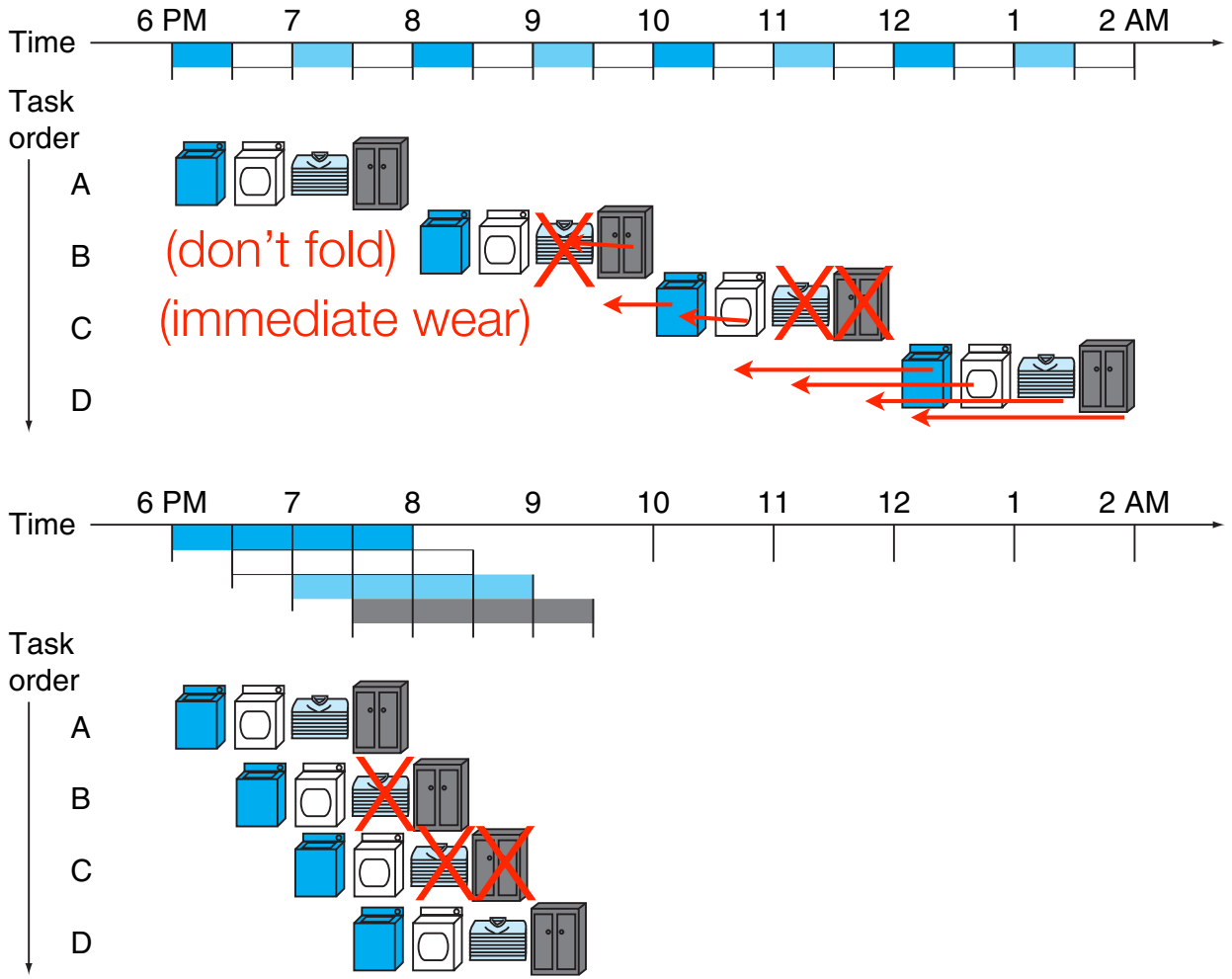


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, "folder," and "storer" each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

The MIPS Pipeline

MIPS Pipelined Datapath: 5 stages

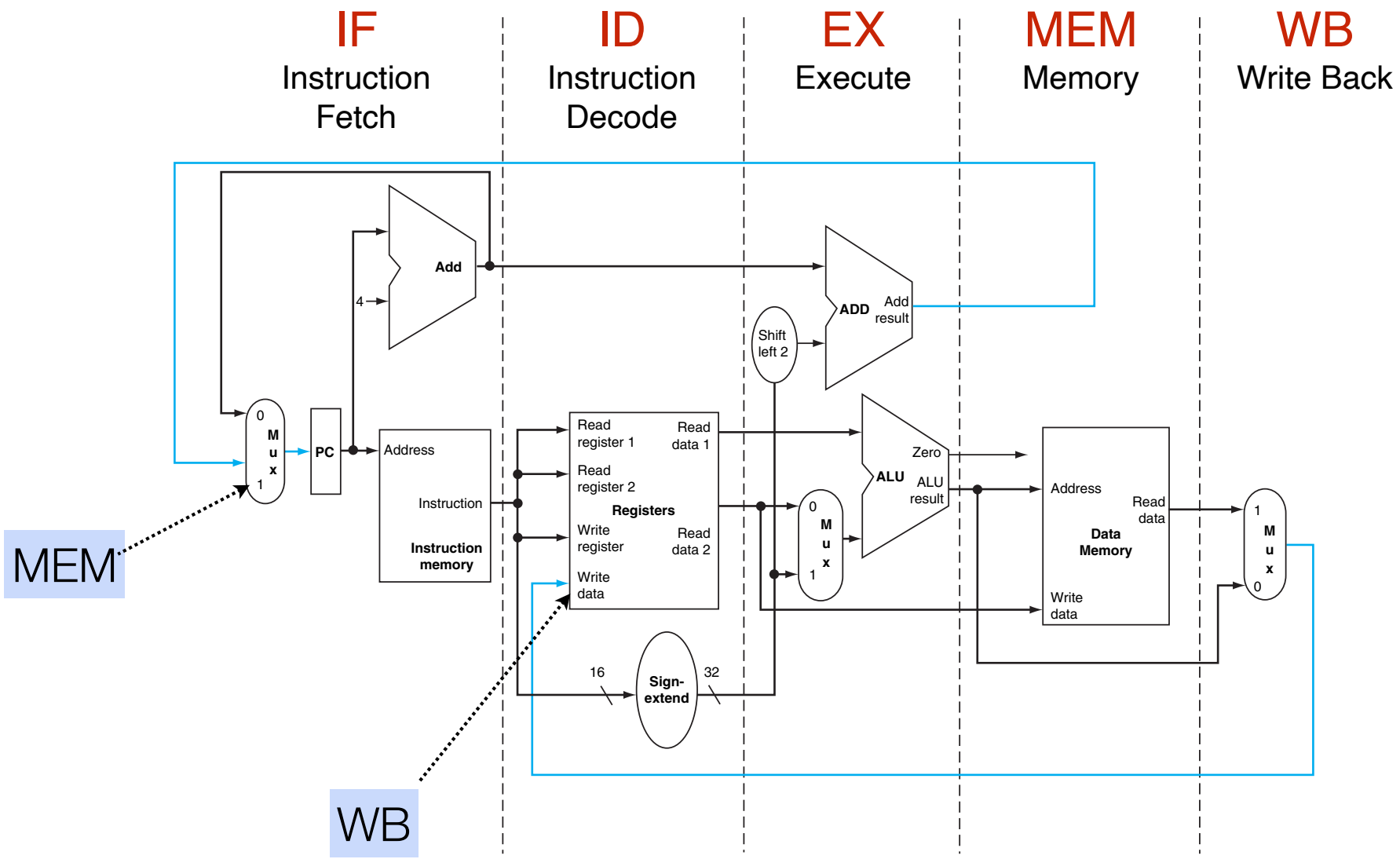


FIGURE 4.33 The single-cycle datapath from Section 4.4 (similar to Figure 4.17). Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the update of the PC and the write-back step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file. (Normally we use color lines for control, but these are data lines.)

How to implement stages

- Need registers (i.e., Flip Flops) between stages, to hold information produced in previous cycle: These are called Pipeline Registers

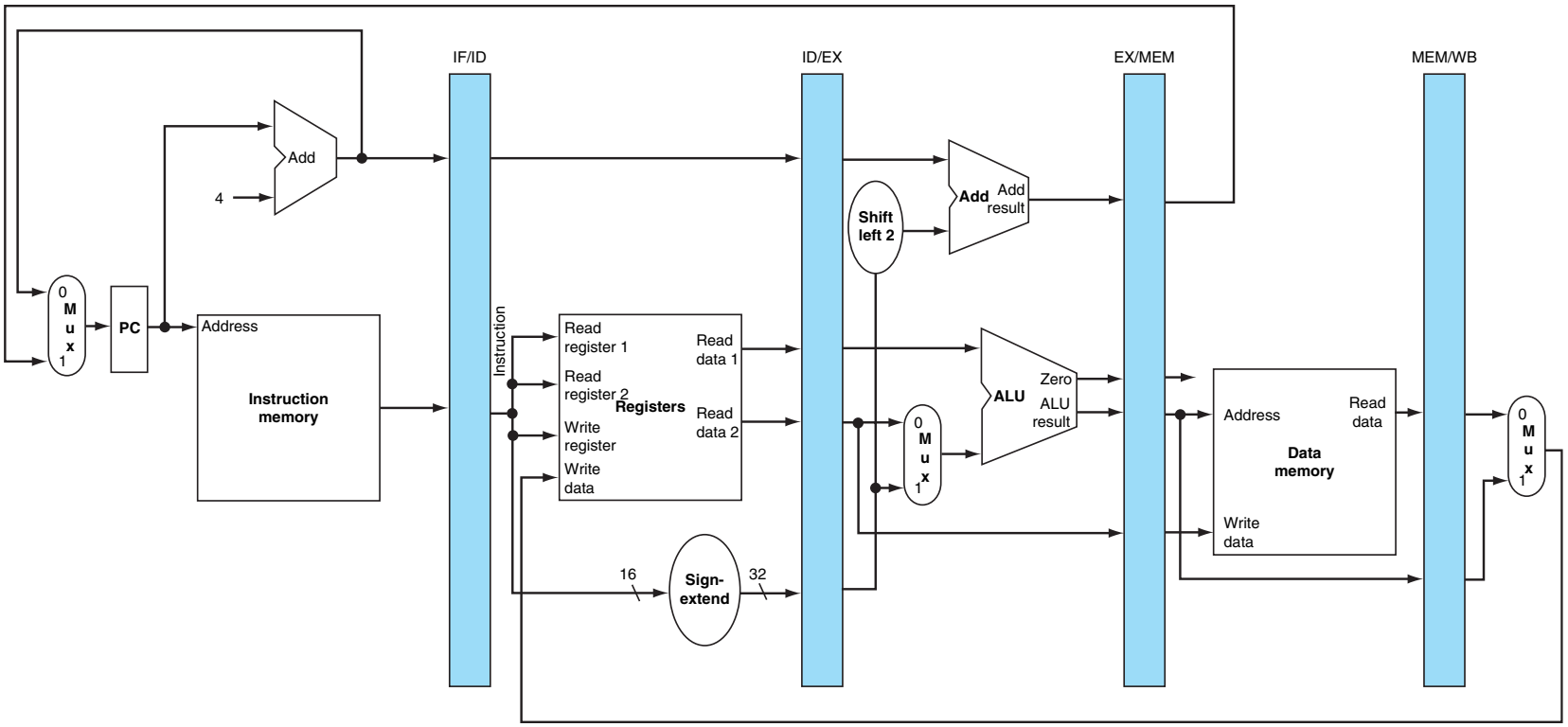
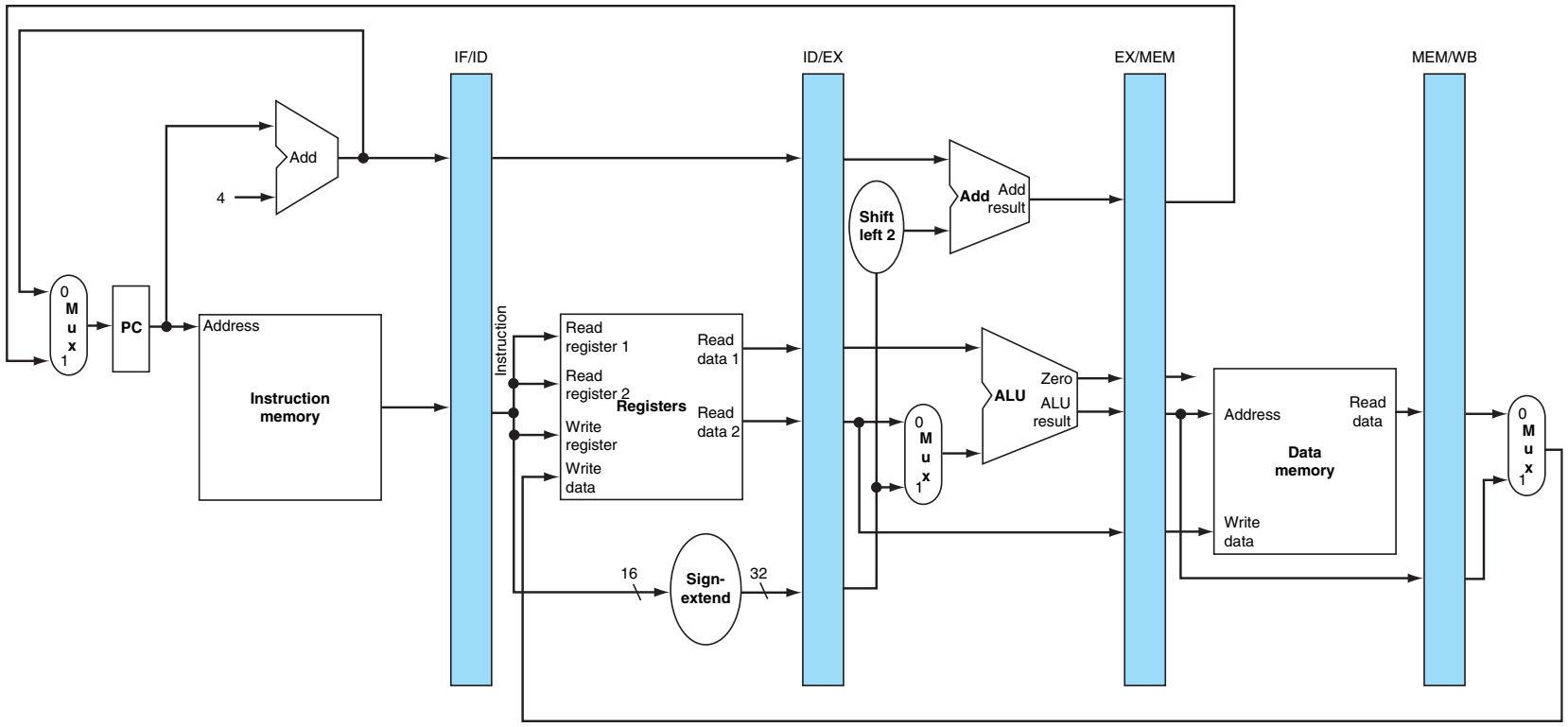


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

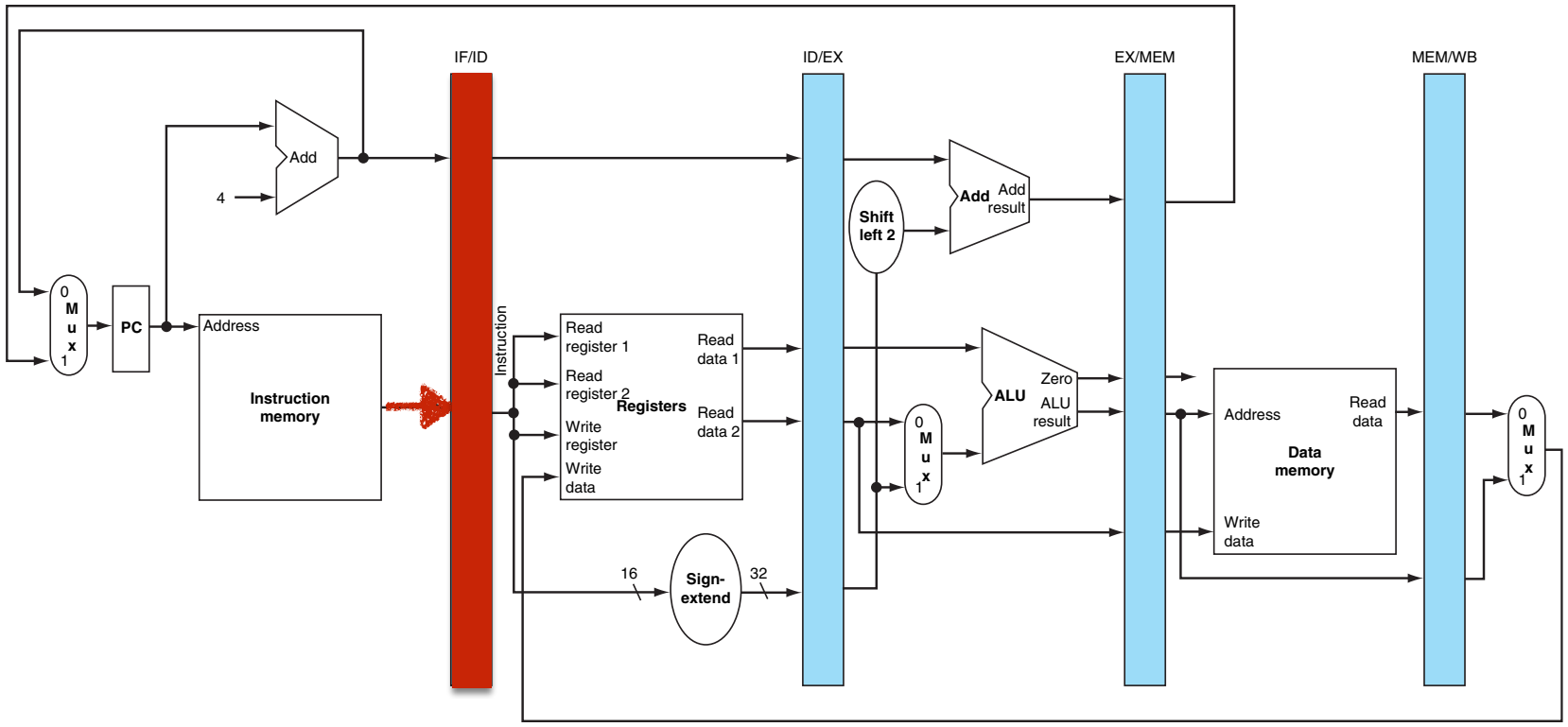
Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:



Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:

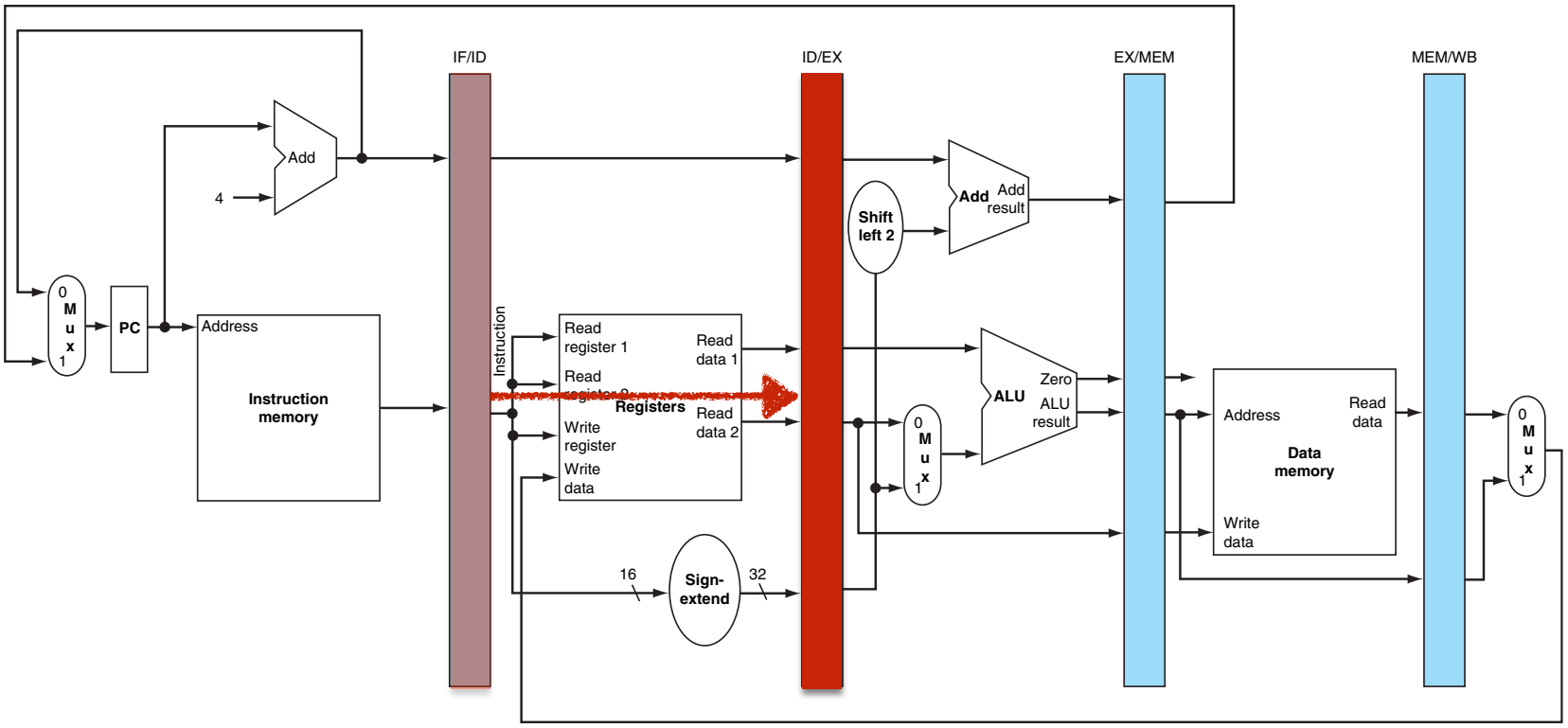


Clock Cycle t

Instruction is read from Instr memory,
Saved into IF/ID pipeline reg

Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:

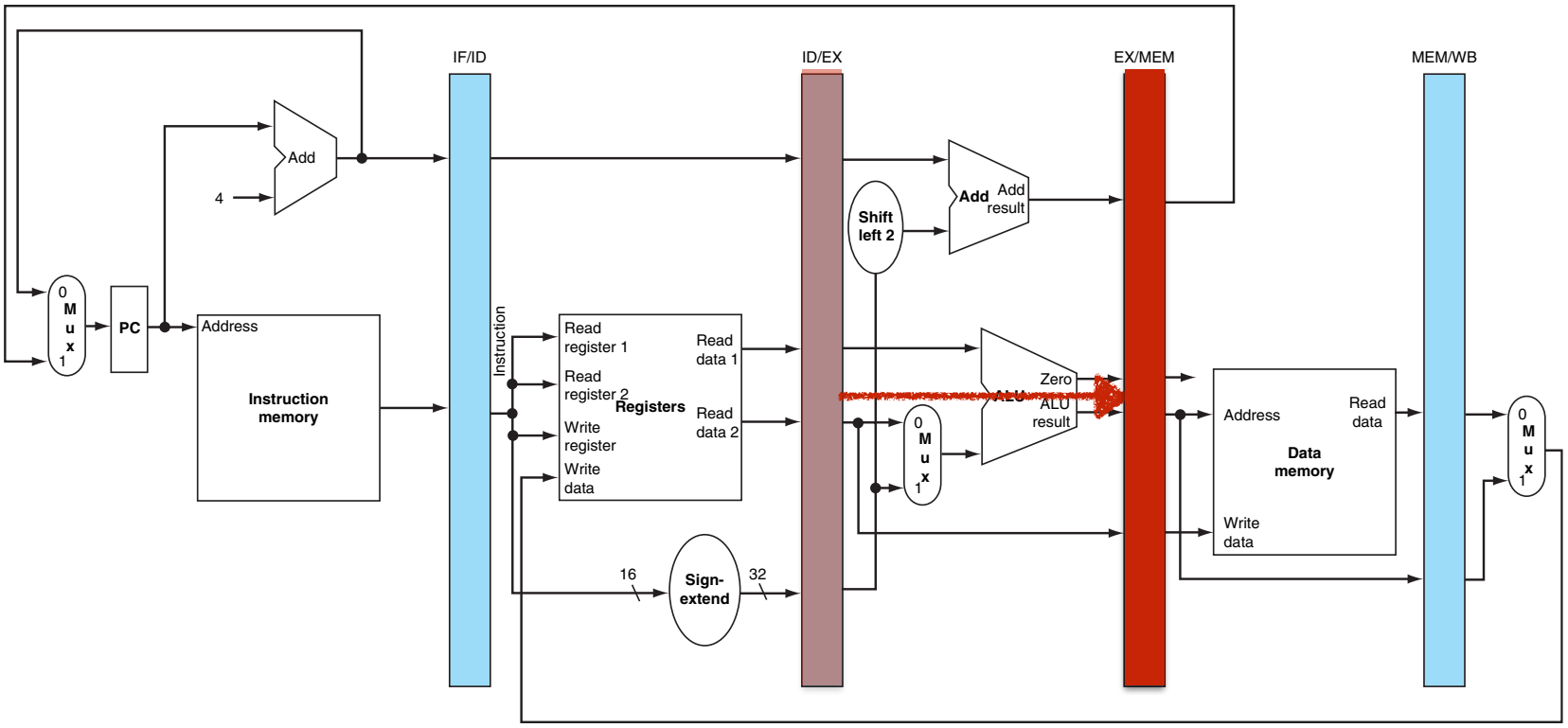


Clock Cycle t+1

Instruction read from IF/ID pipeline reg,
Pulls data from register file, stores results
in ID/EX pipeline reg

Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:

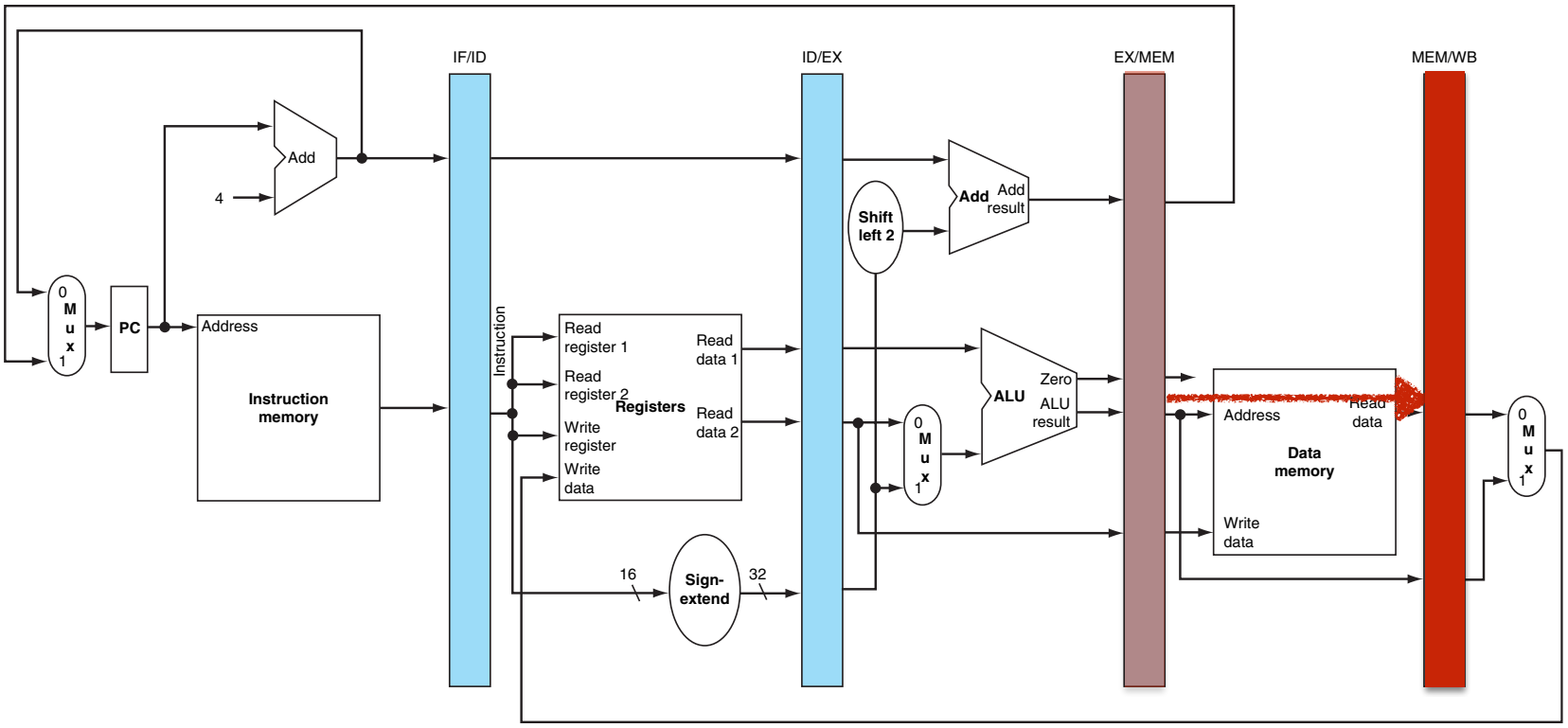


Clock Cycle t+2

Data read from ID/EX pipeline reg,
Runs data through ALU
Store results in EX/MEM pipeline reg

Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:

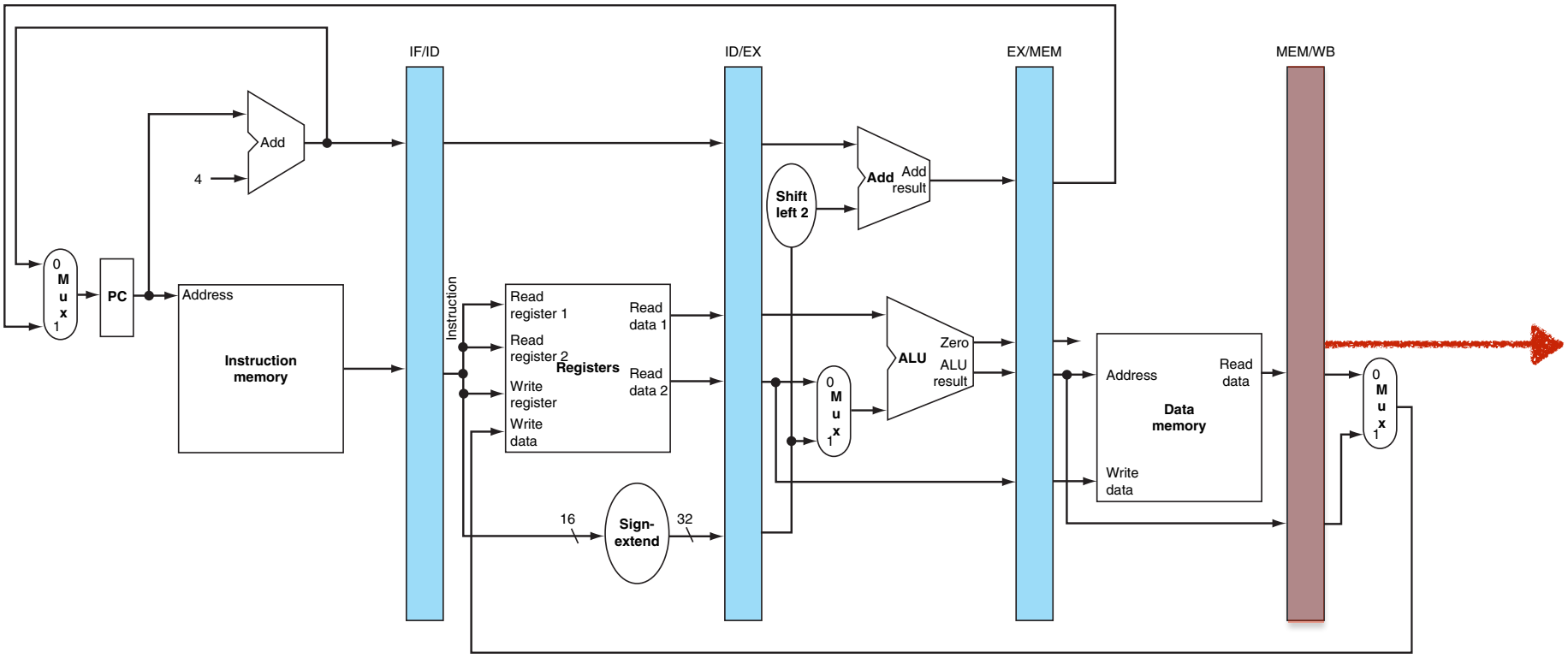


Clock Cycle t+3

Data read from EX/MEM pipeline reg,
Runs data through Mem
Store results in MEM/WB pipeline reg

Pipeline Reg example

- Let's follow an instruction through the pipeline over 5 stages:



Clock Cycle t+4

Data read from MEM/WB pipeline reg,
Writes to reg file
Instruction exits pipeline

Clock for single cycle v. pipelined

- In single cycle, clock must permit flow through all parts of “big” circuit
- In pipeline, clock must permit flow (in parallel) through stage. Constrained by “slowest” stage
  - Thus, clock in pipelined arch runs faster than clock in single cycle

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

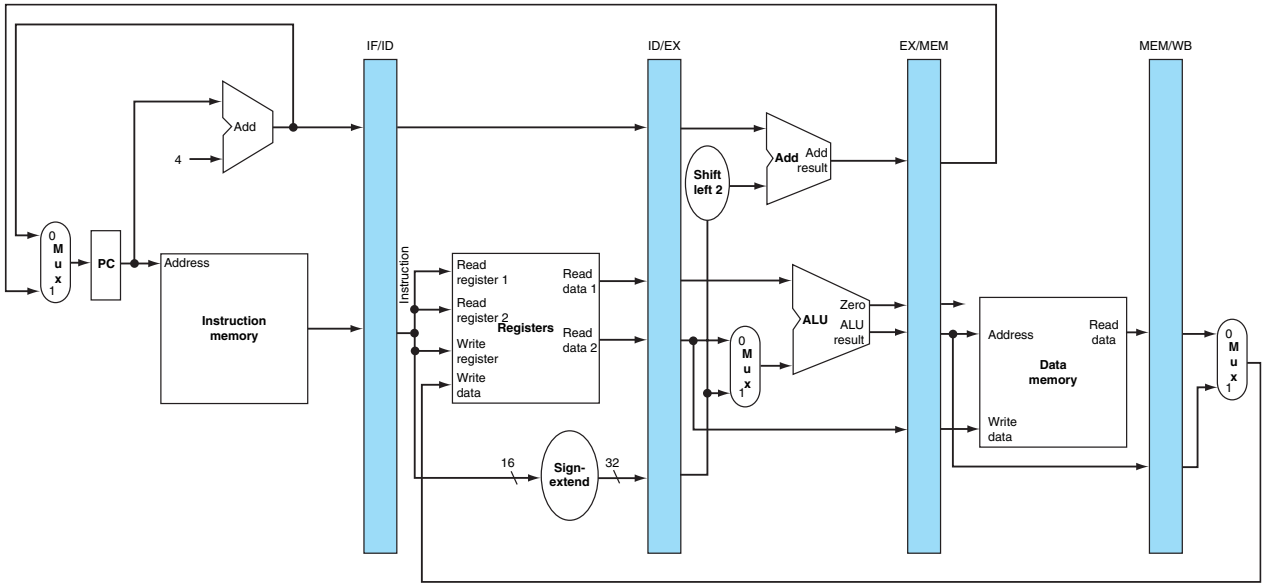


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

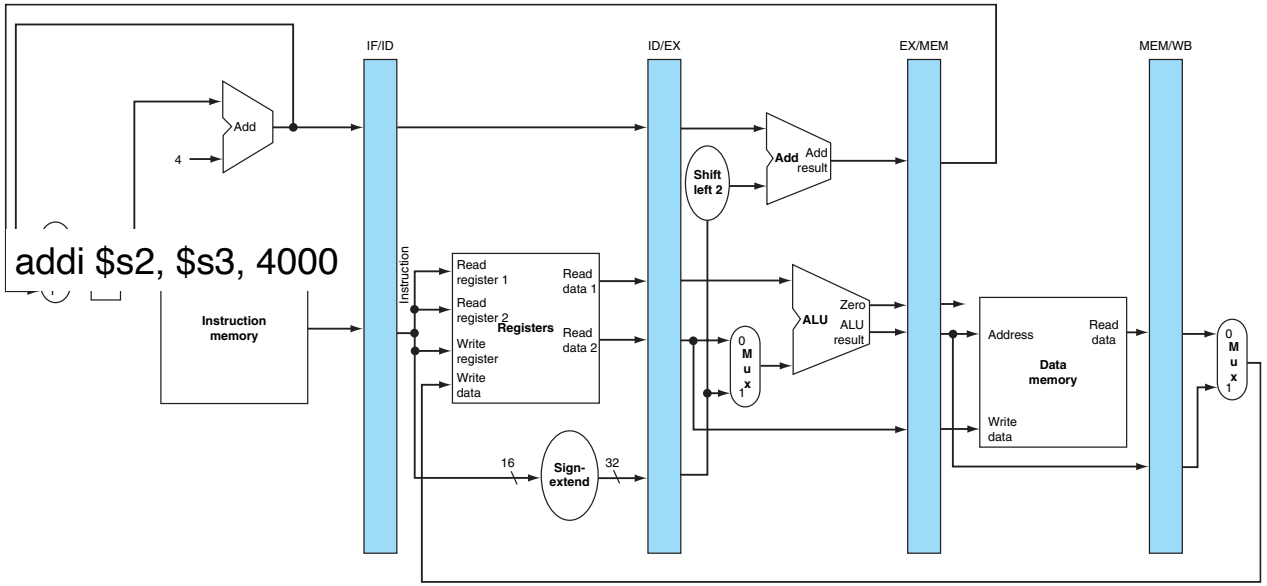


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

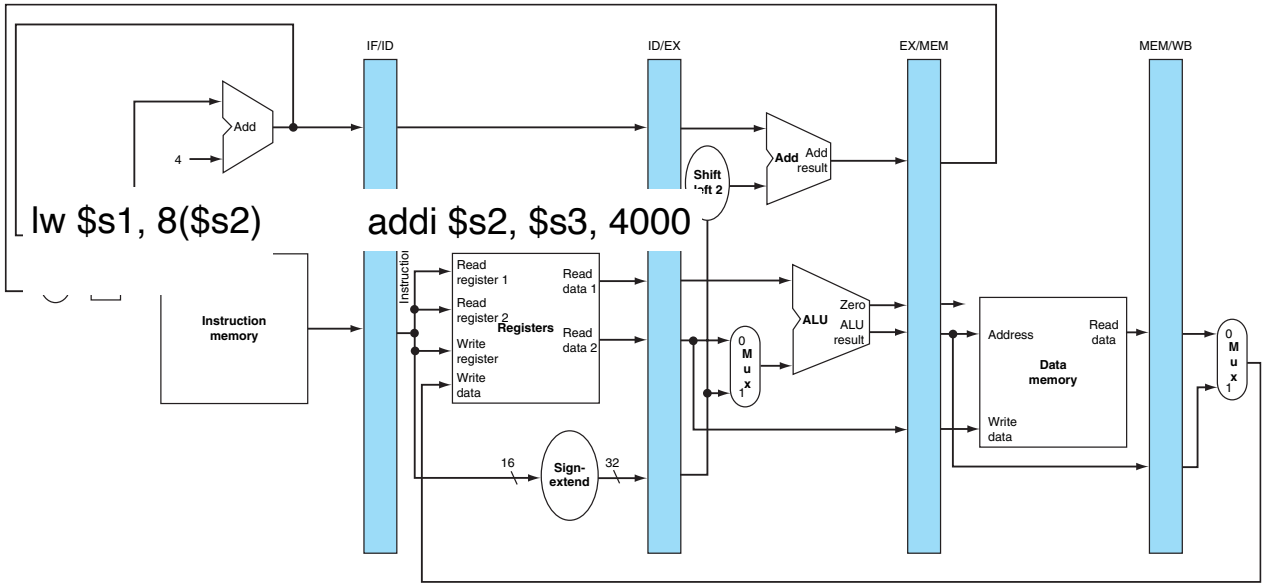


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

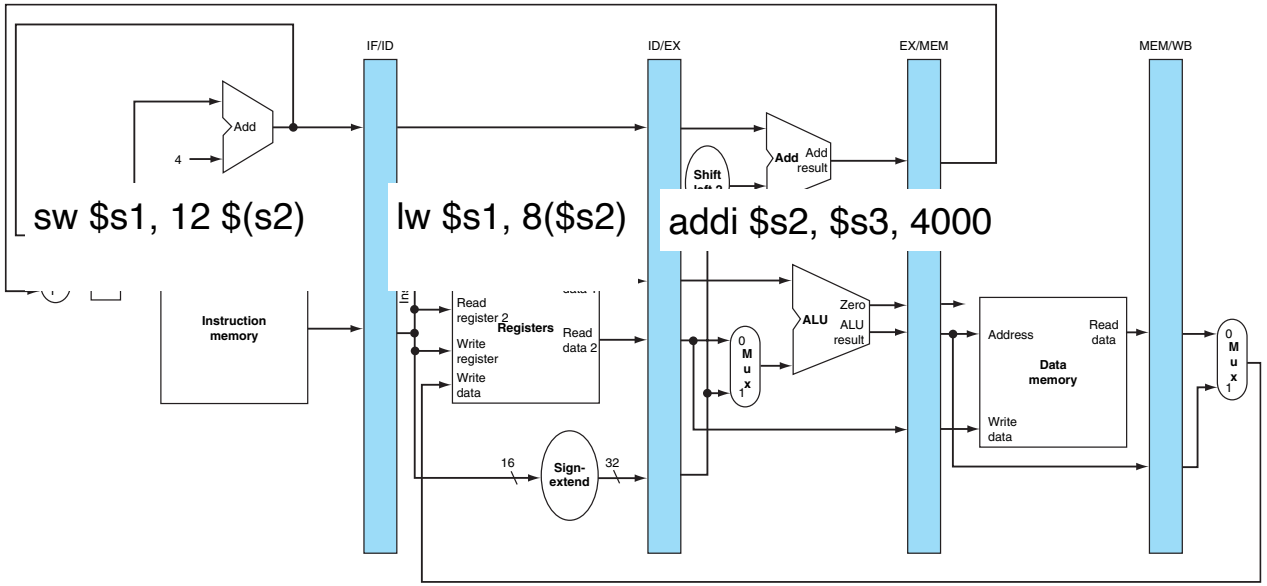


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

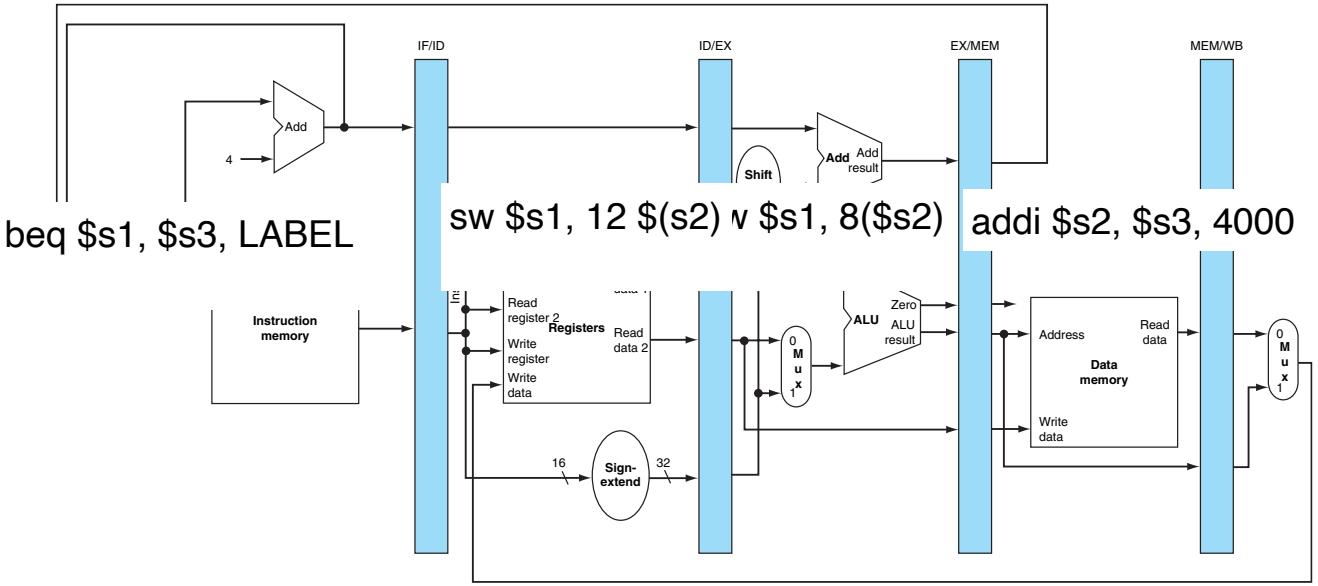


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

How to implement stages

Code in pipeline:

e.g.,

addi $s2, $s3, 4000
lw $s1, 8($s2)
sw $s1, 12 $(s2)
beq $s1, $s3, LABEL
or $s4, $s5, $s6

Assuming conditional evaluated to false in beq

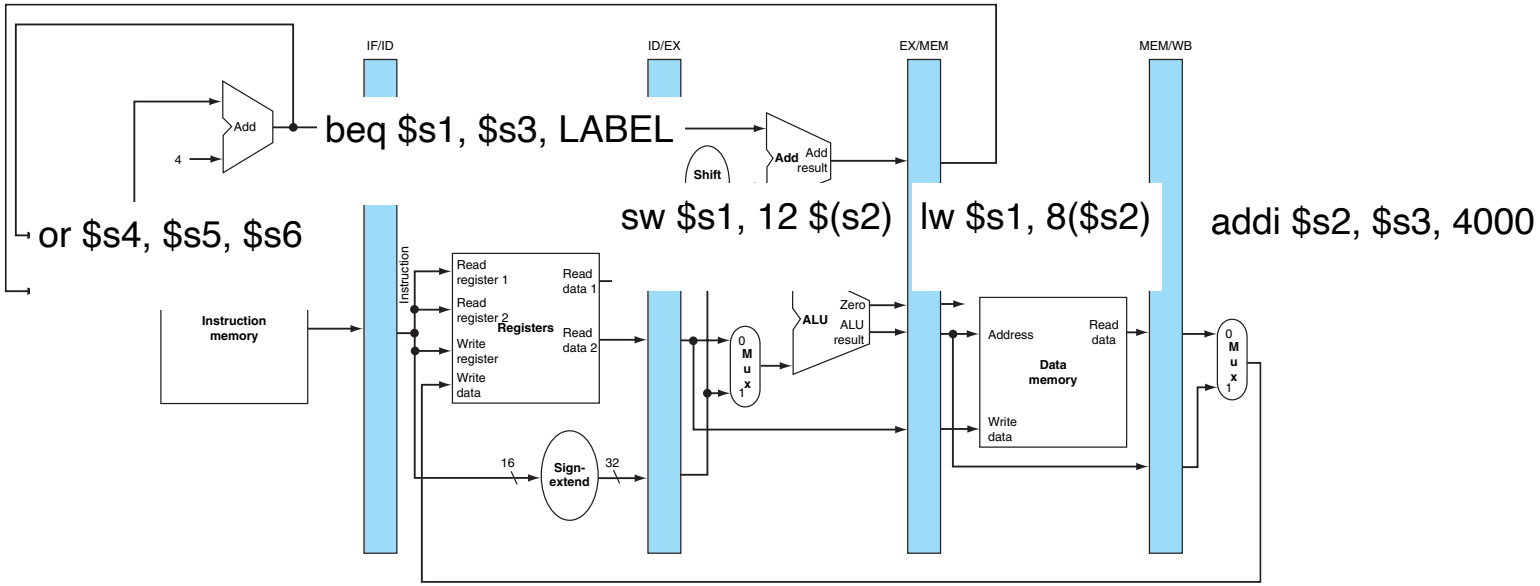
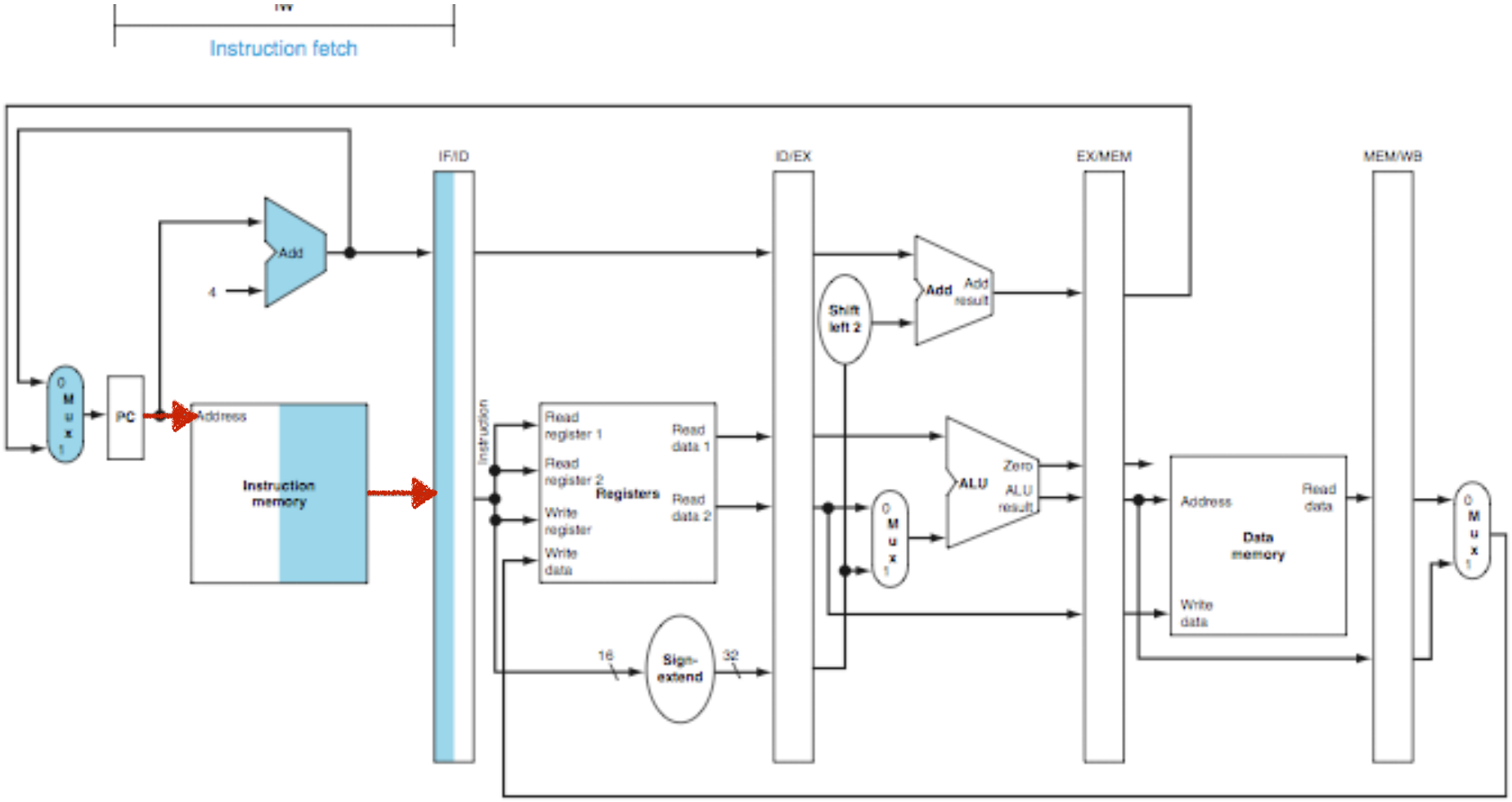


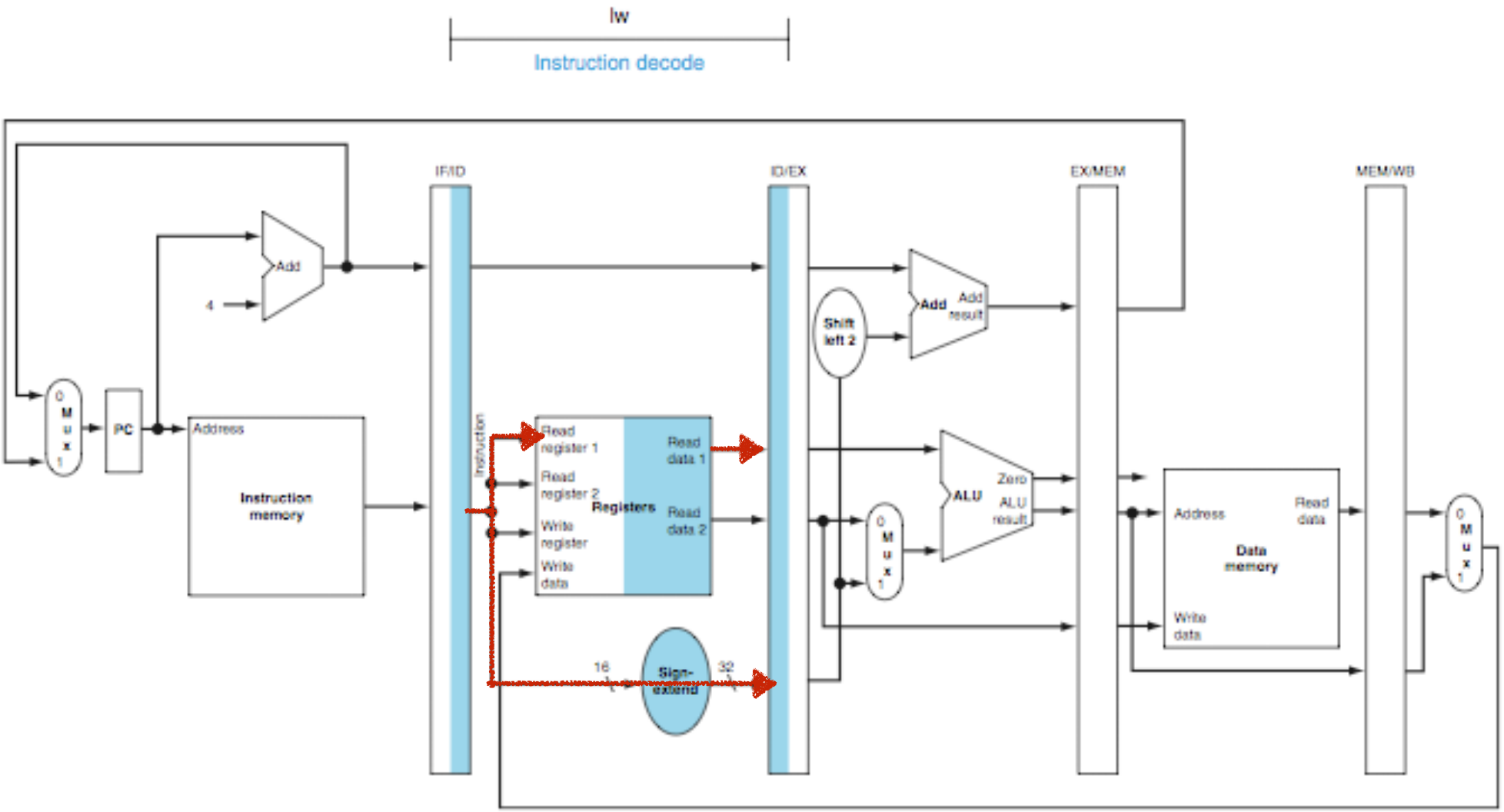
FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

lw through the pipeline…

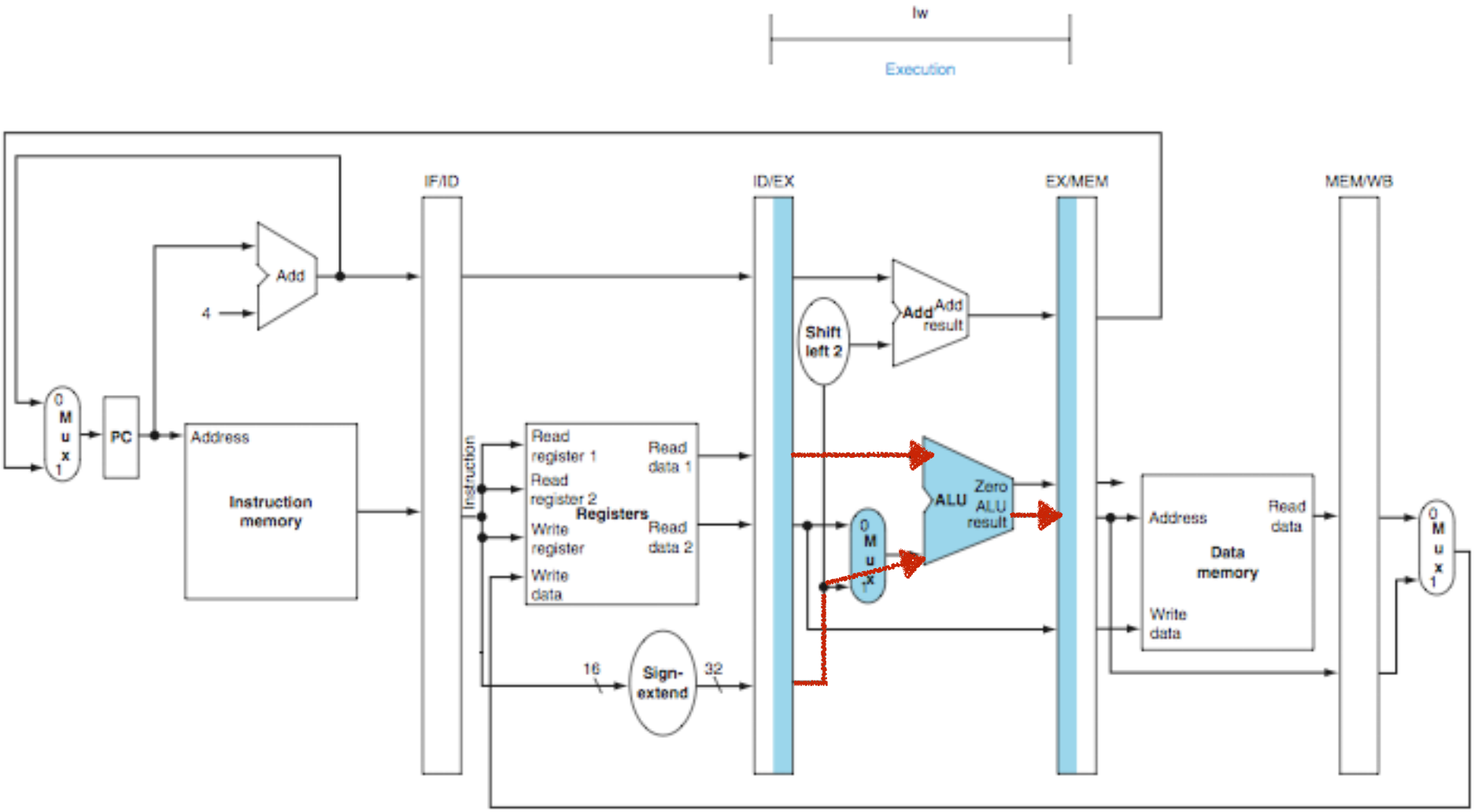
IF Stage for lw



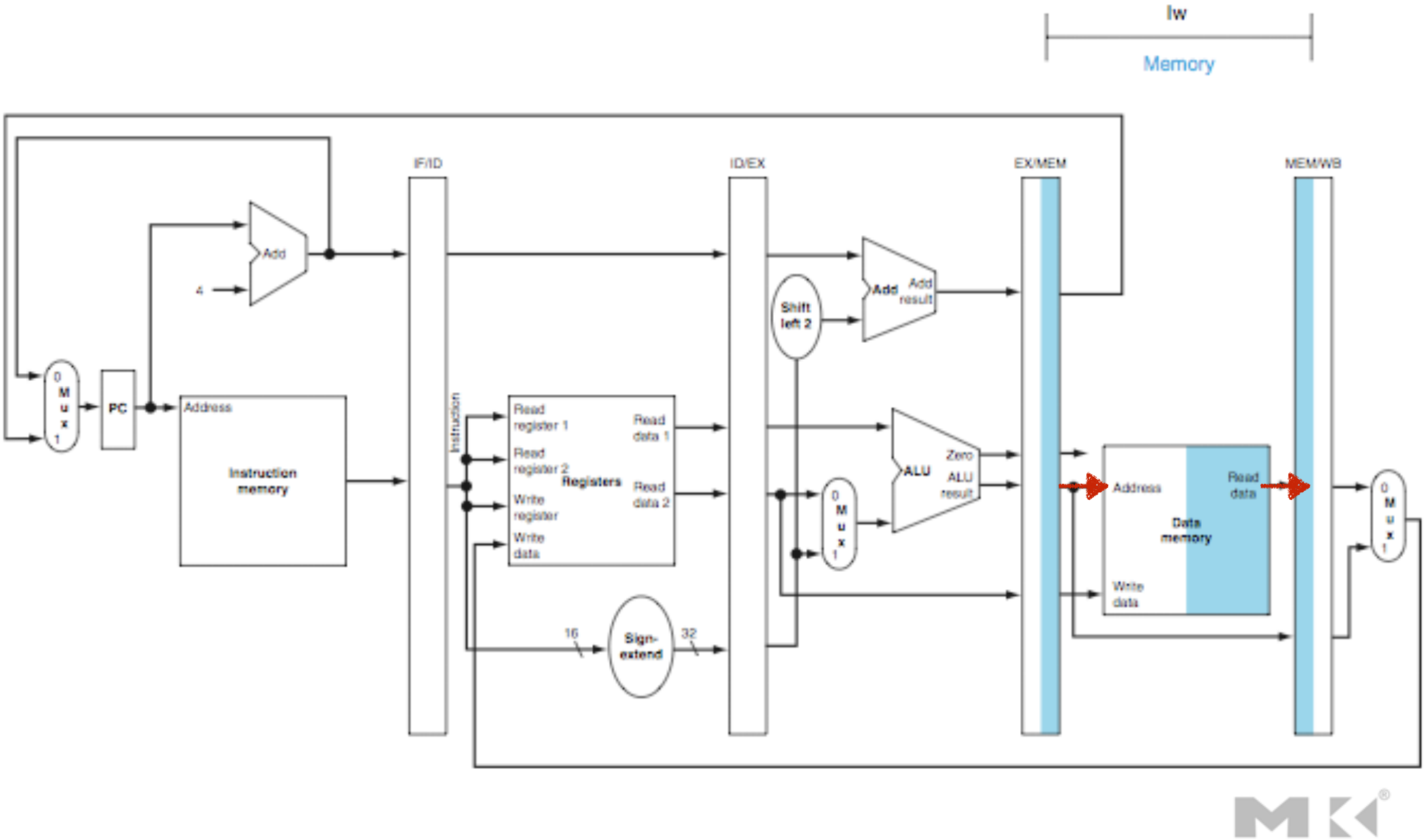
ID Stage for lw



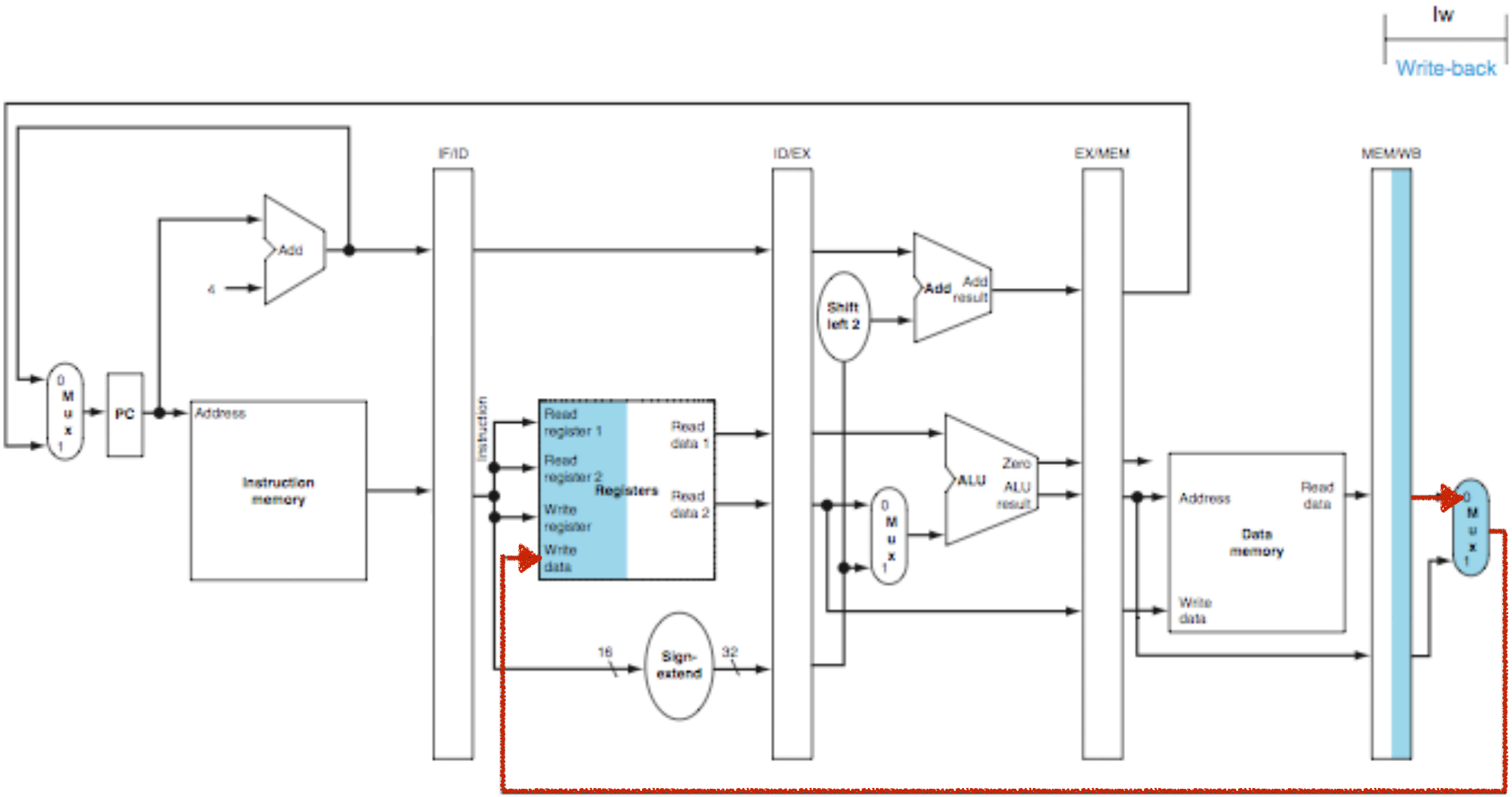
EX Stage for lw



MEM Stage for lw



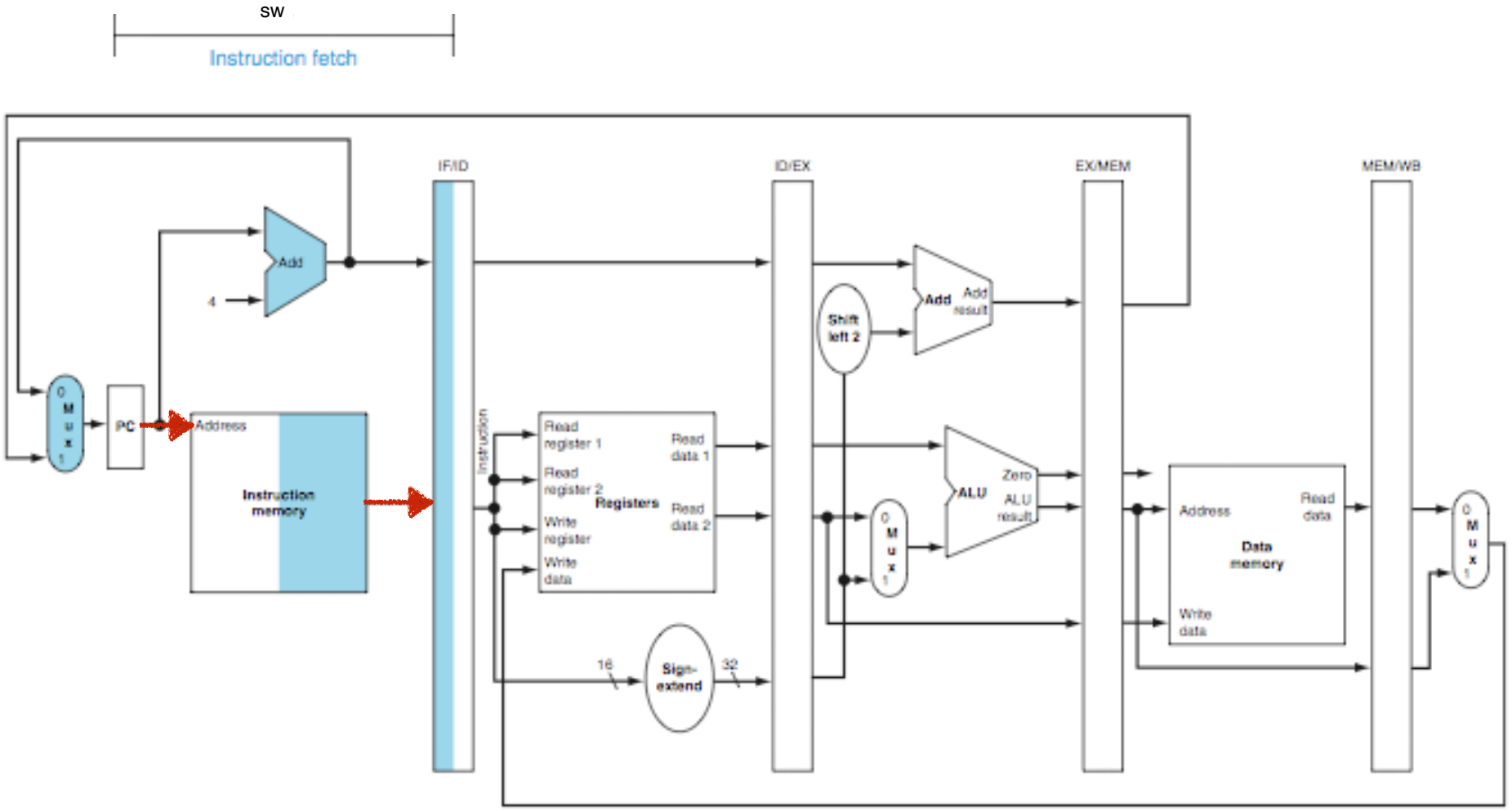
WB Stage for lw



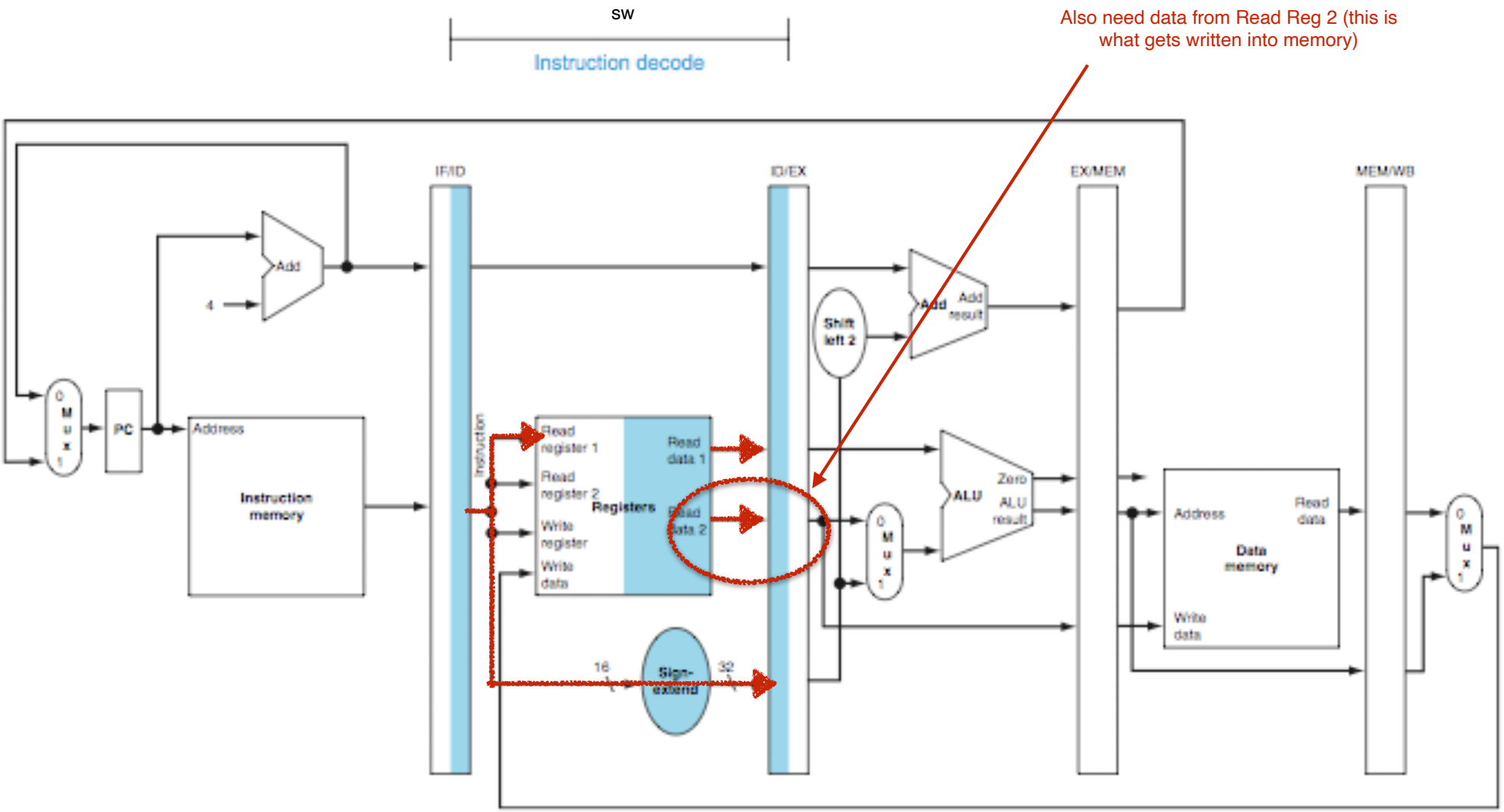
One small error in above - do you see it?

sw through the pipeline…

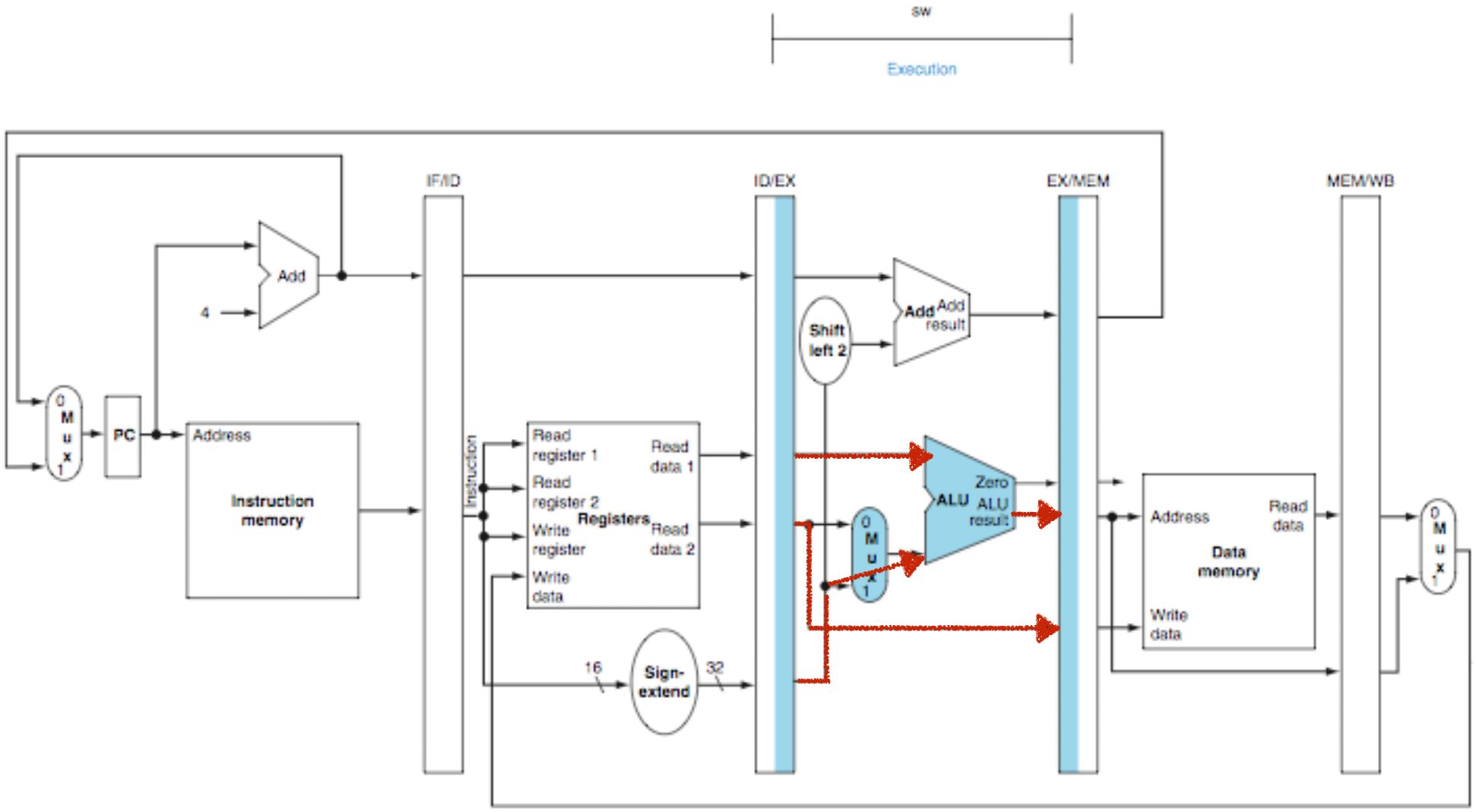
IF for sw (Same as lw)



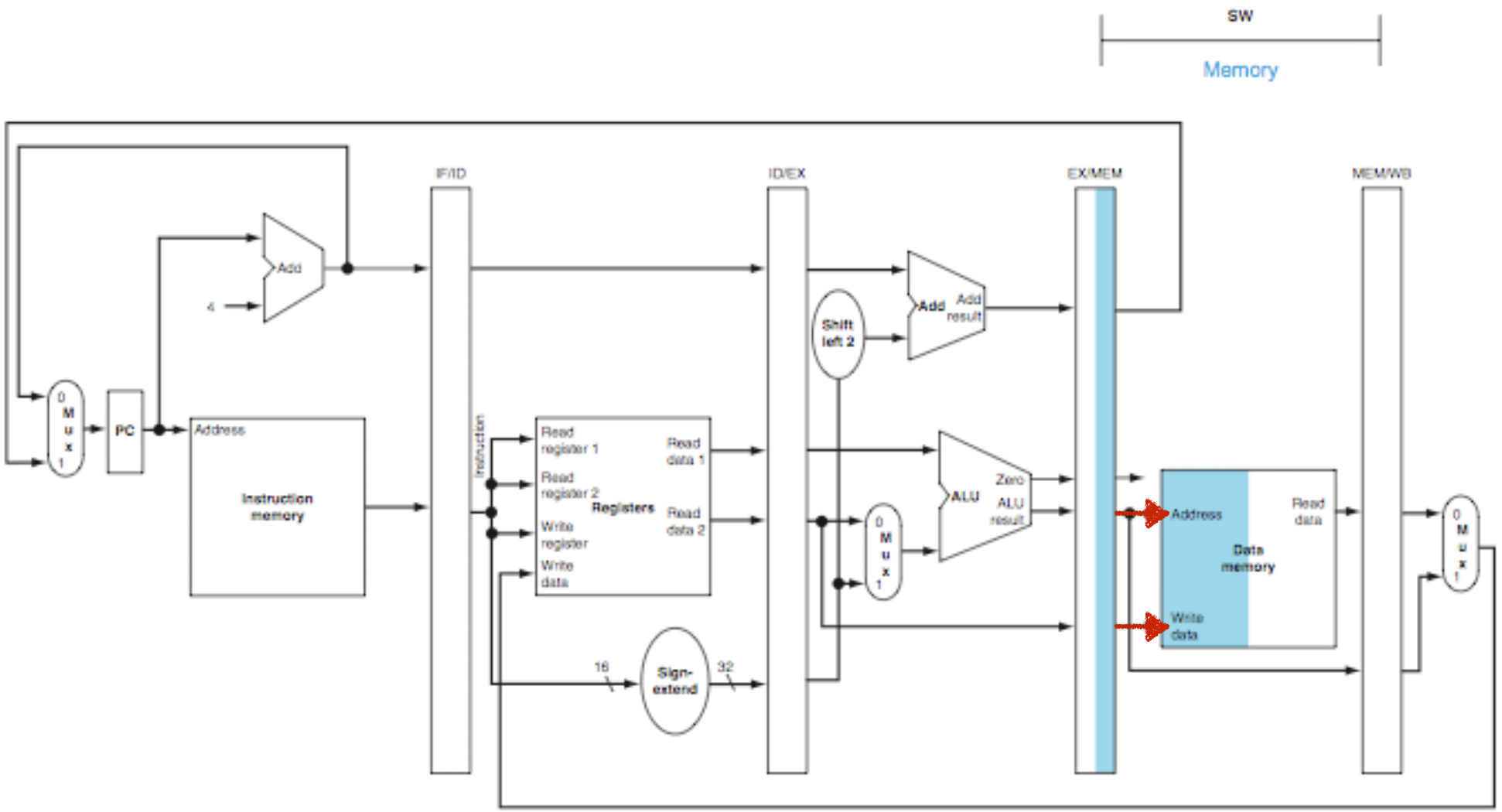
ID for sw (almost same as lw)



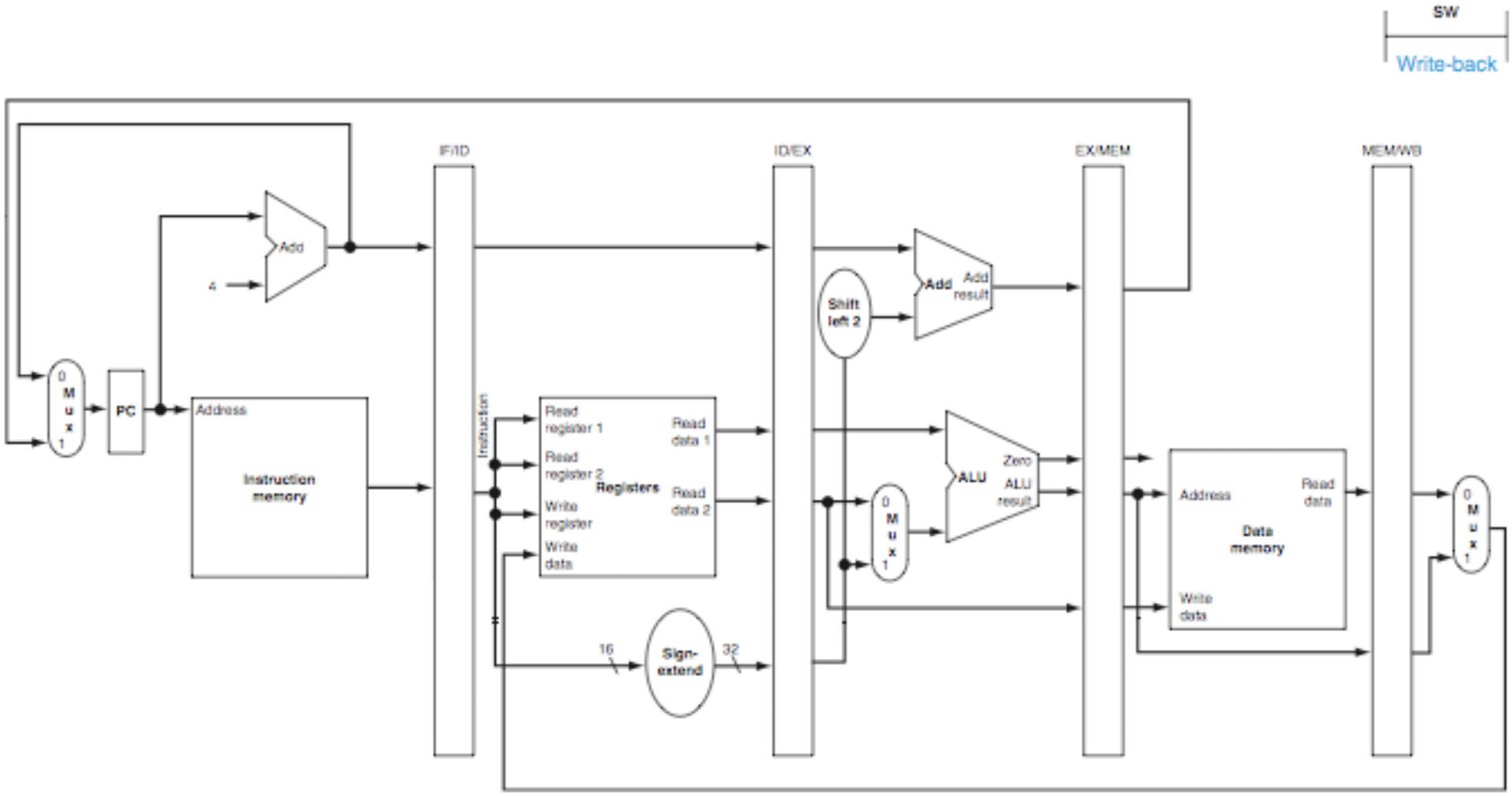
EX for sw (almost as lw)



MEM for sw

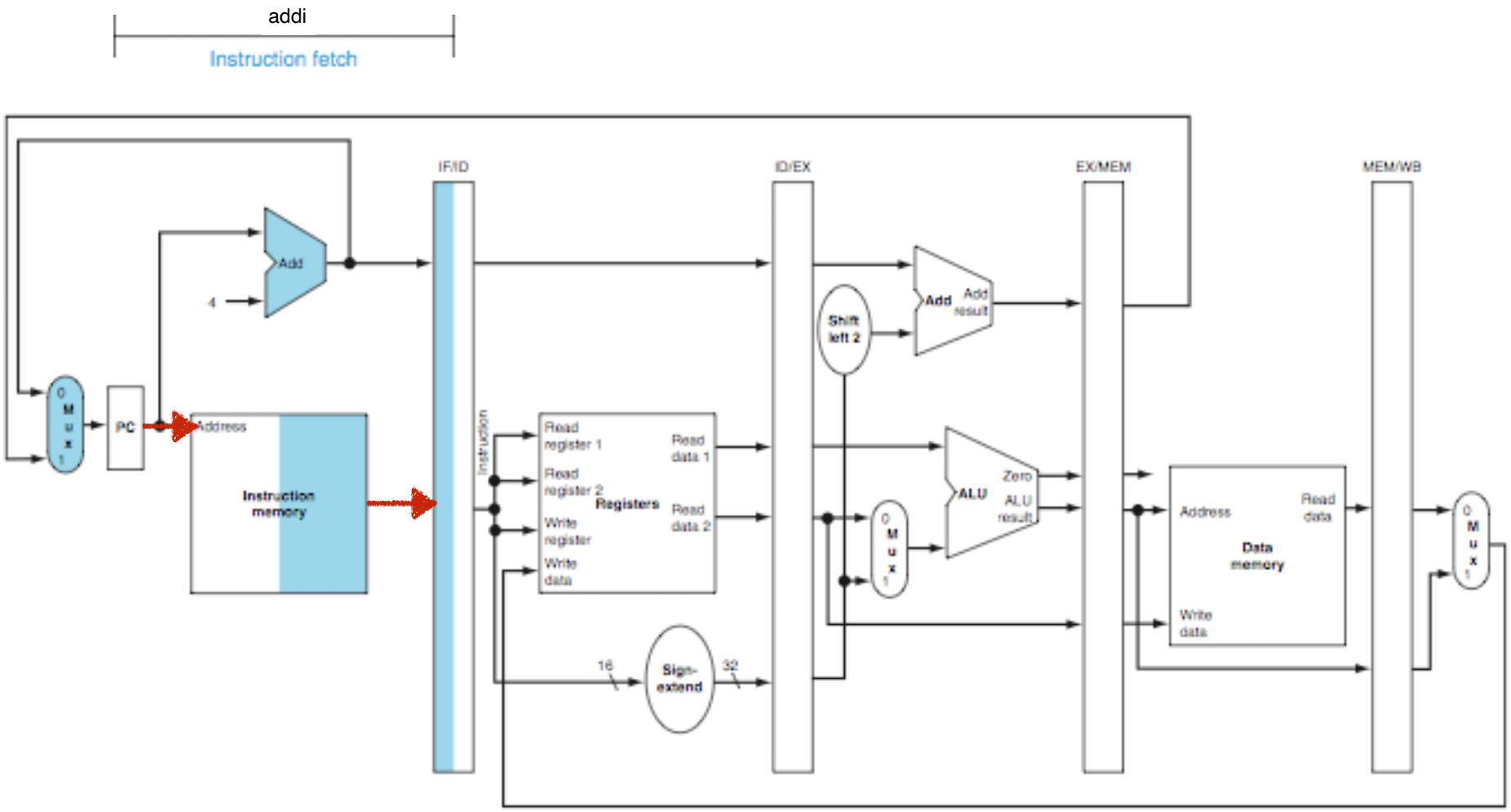

SW
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32

WB for sw (do nothing)

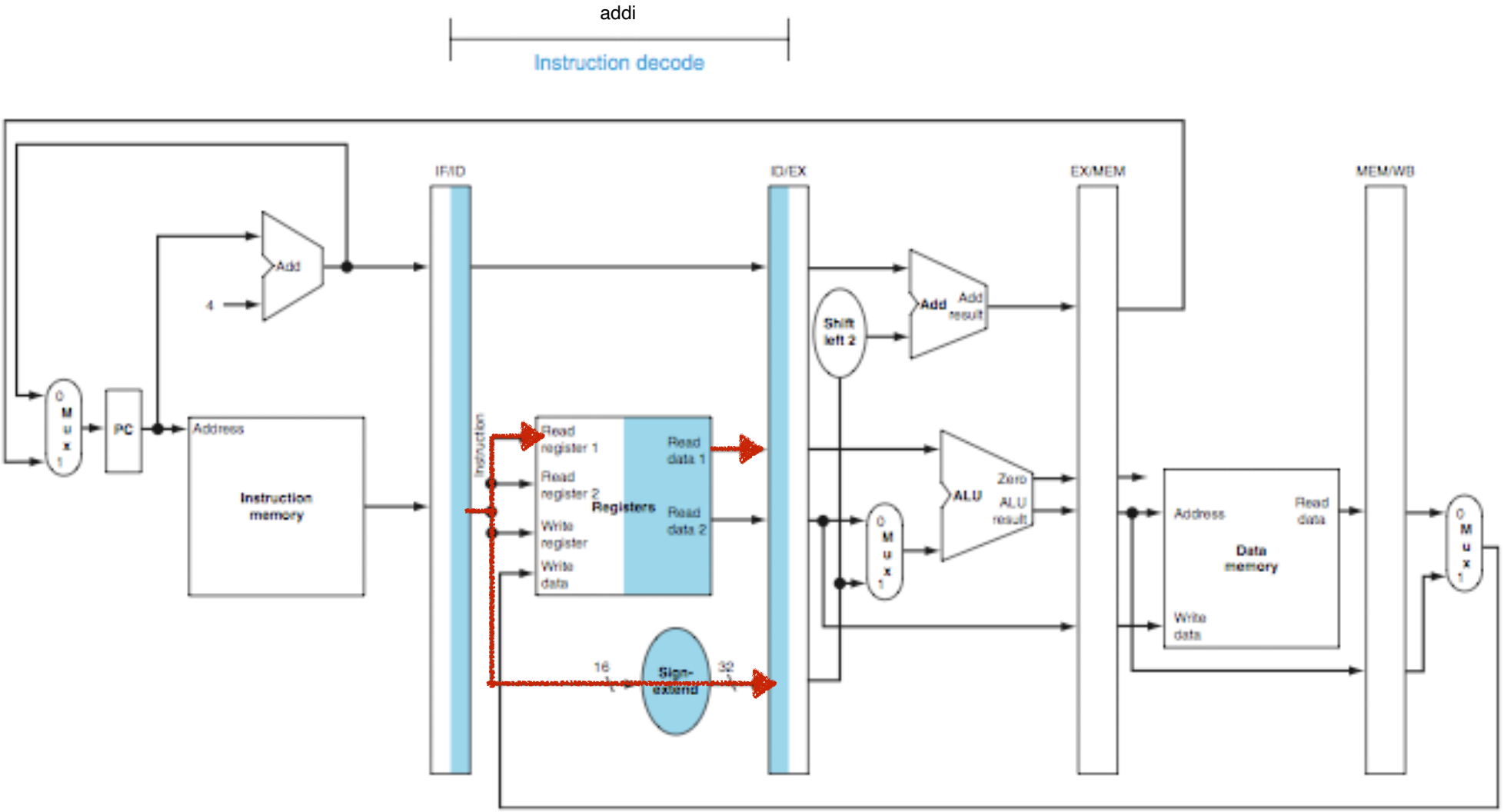


Addi through the pipeline…

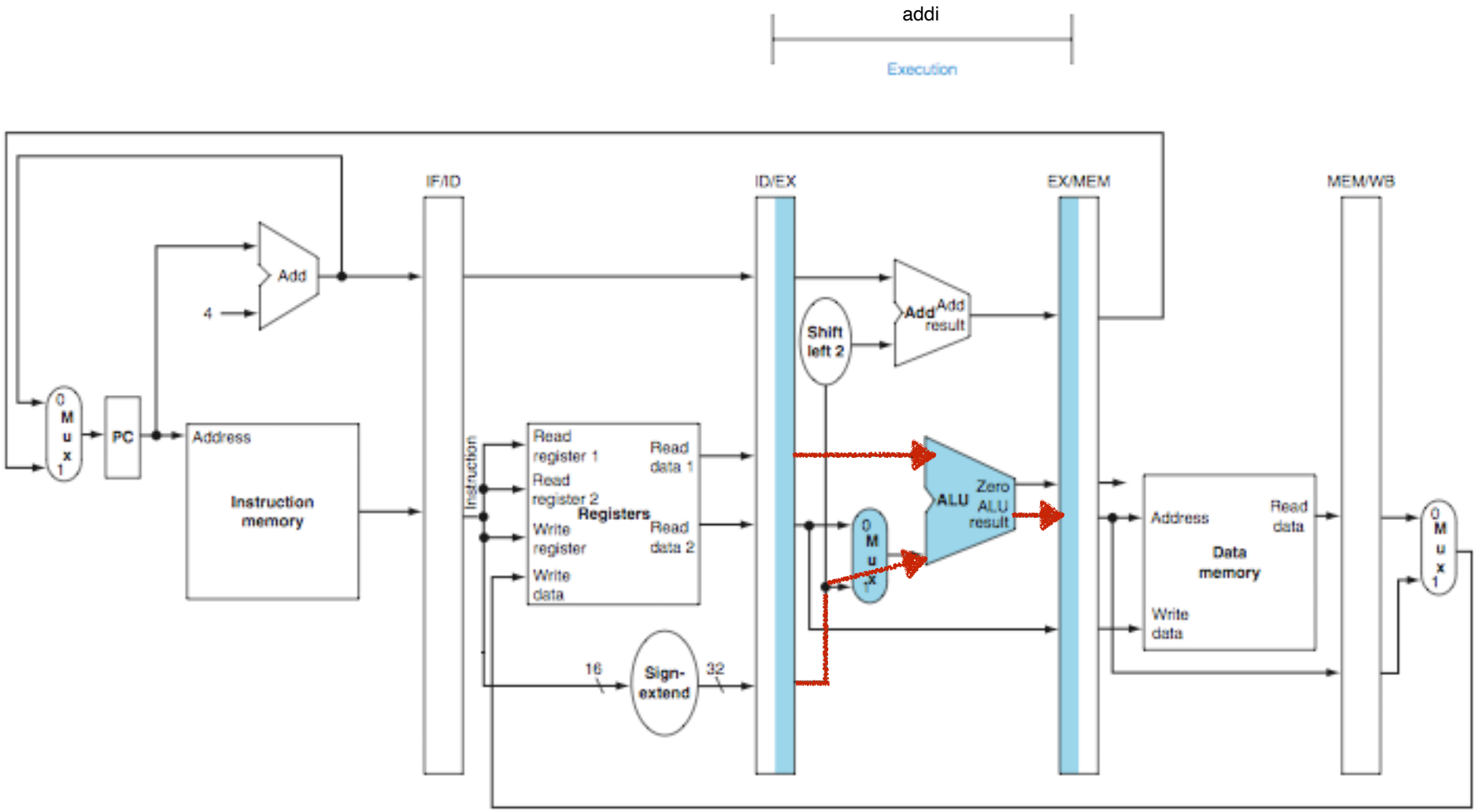
IF Stage for addi (just like lw)



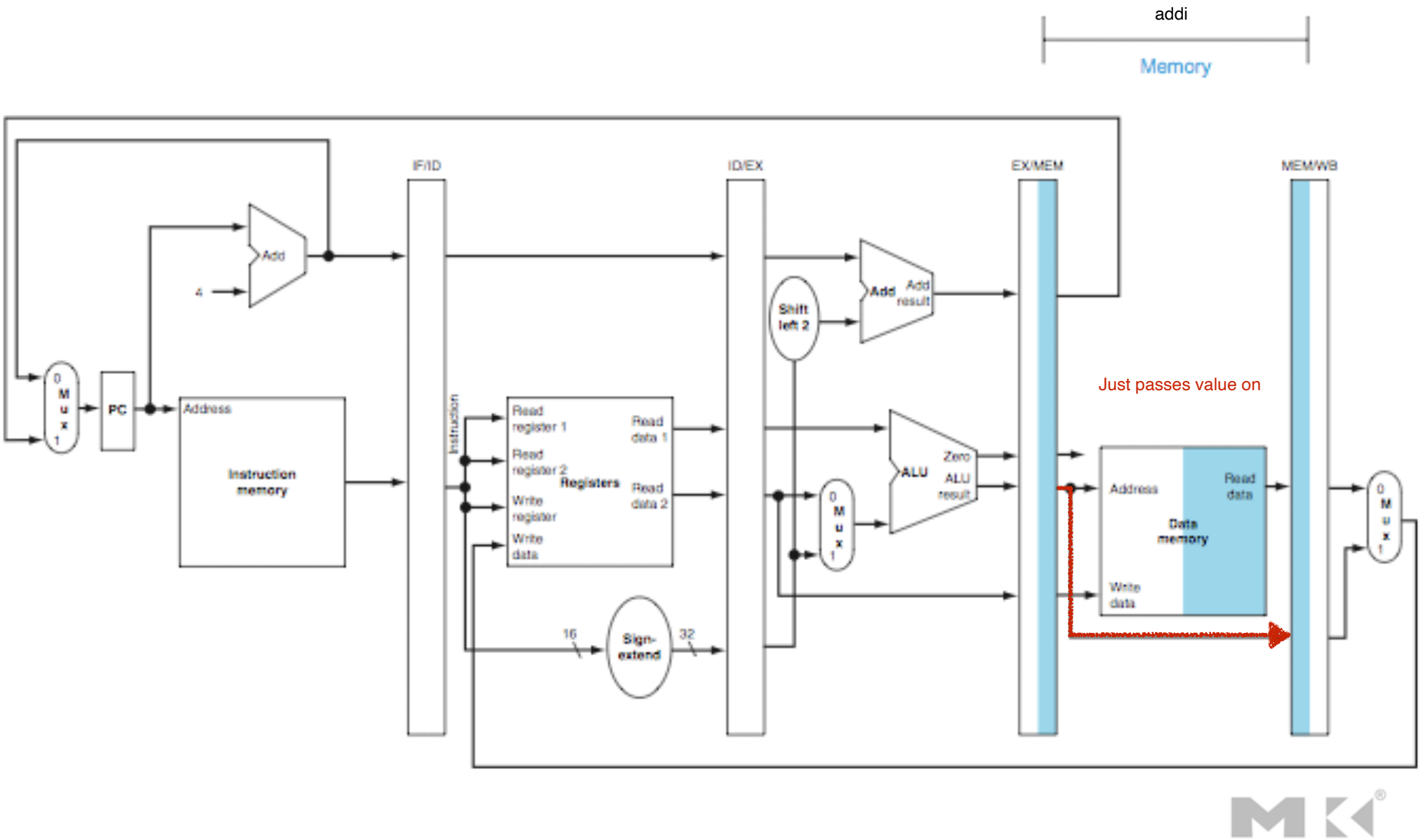
ID Stage for addi (just like lw)



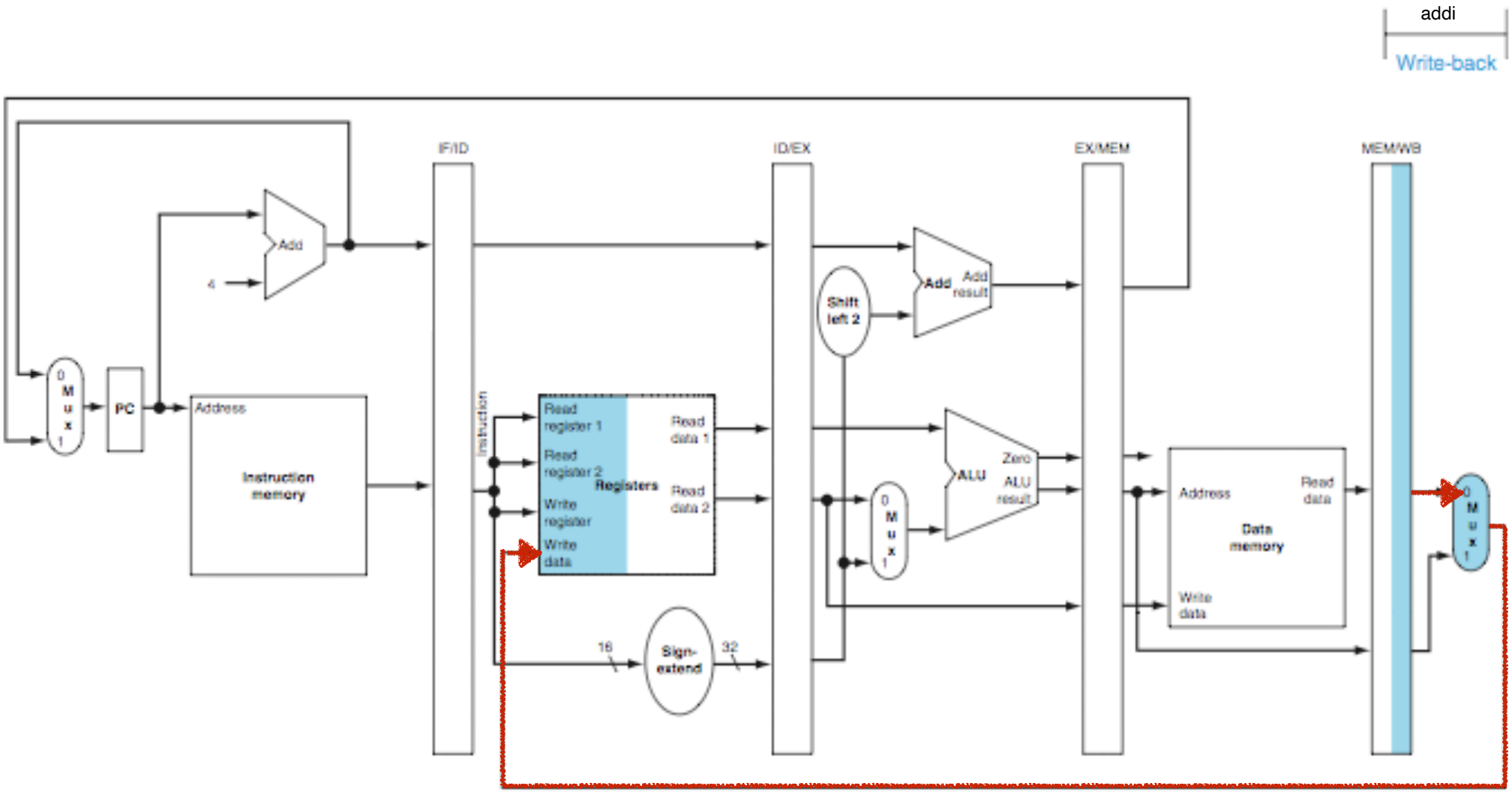
EX Stage for addi (just like lw)



MEM Stage for addi

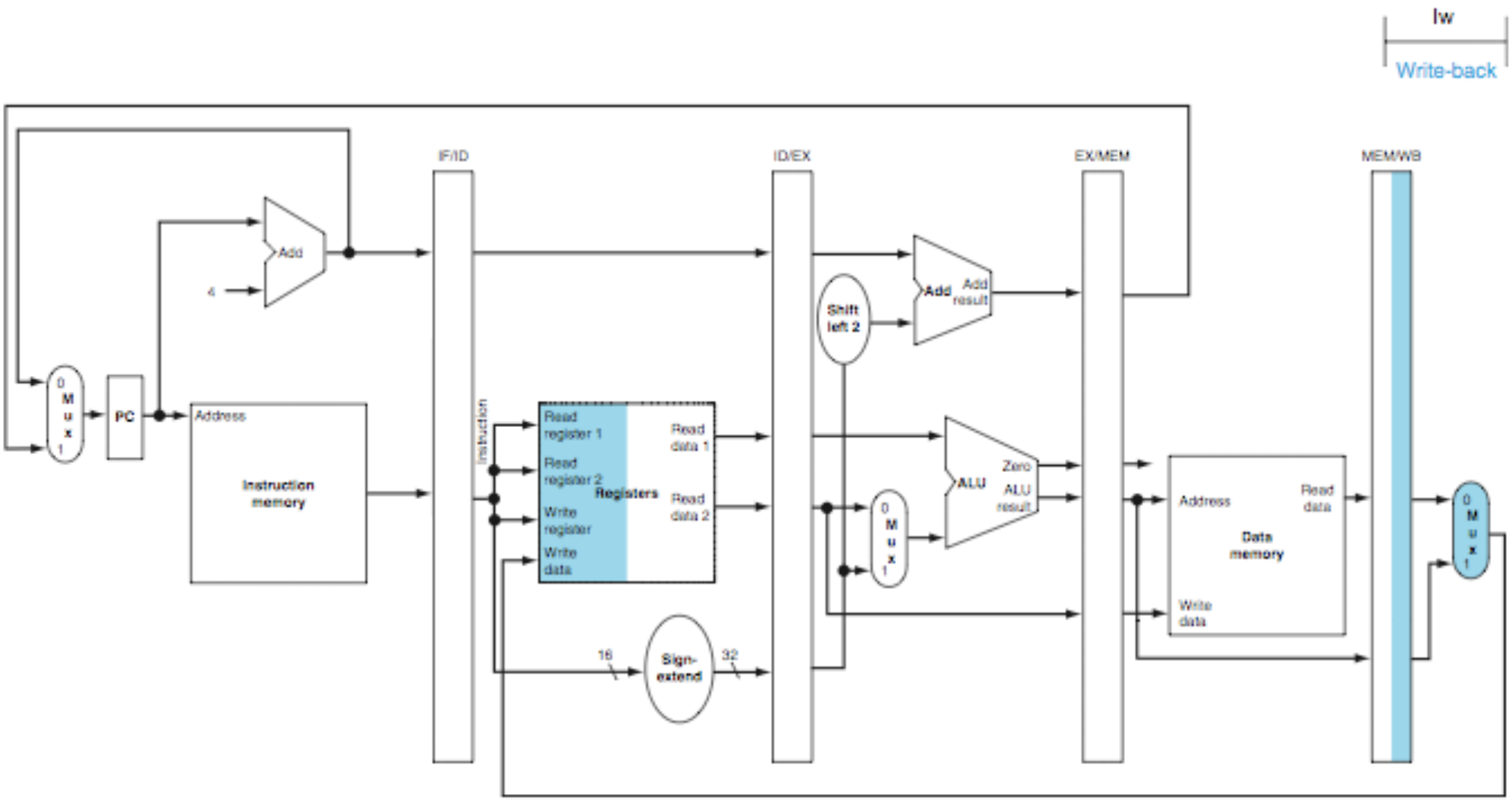


WB Stage for addi



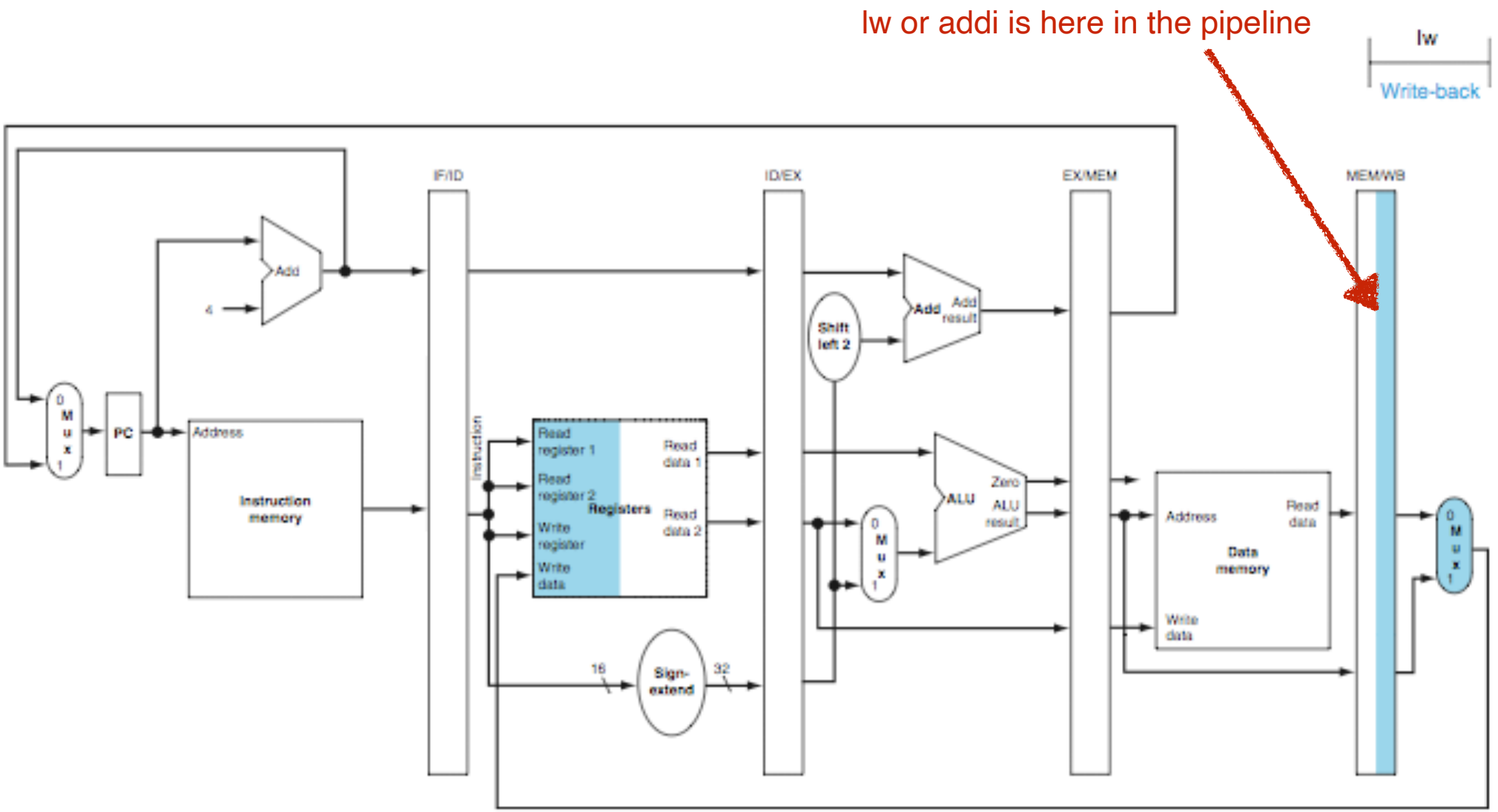
One small error in above - do you see it?

WB Stage for addi & lw



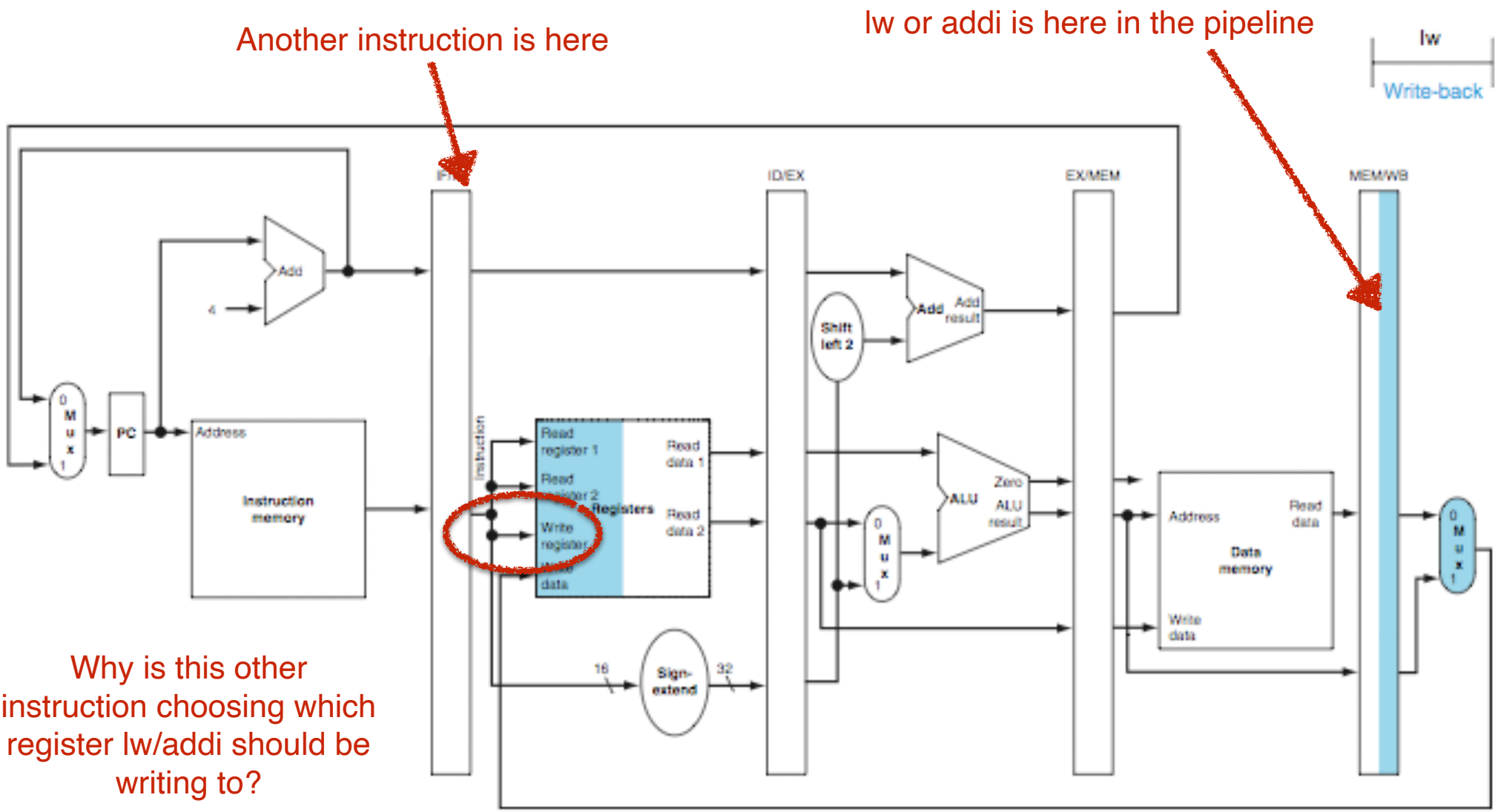
1 small error in above, do you see it?

WB Stage for addi & lw



1 small error in above, do you see it?

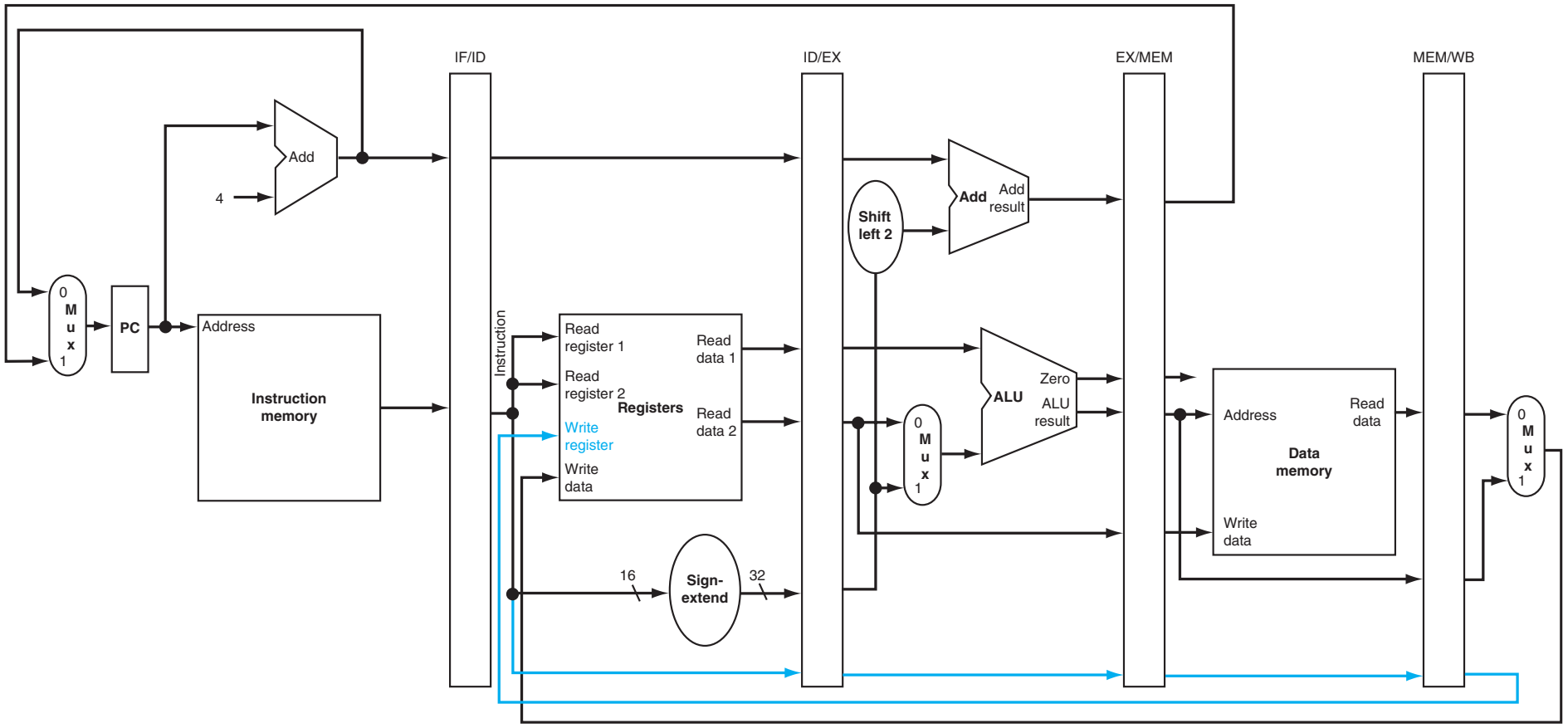
WB Stage for lw



Why is this other instruction choosing which register lw/addi should be writing to?

1 small error in above, do you see it?

Corrected Datapath

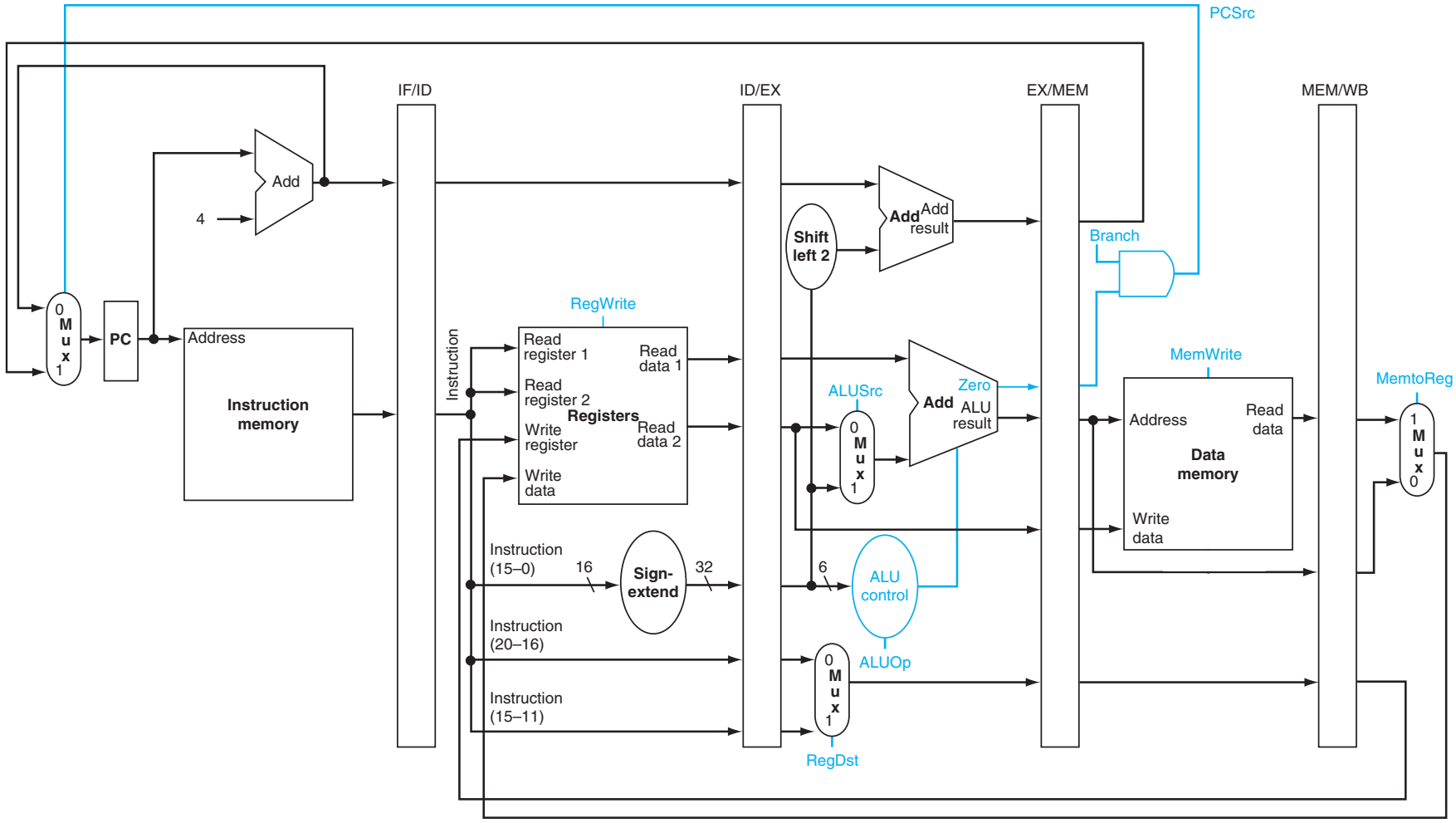


Pass information needed in later stages through the pipeline!

Pipeline Data “Control”

(How selector info is passed through pipeline)

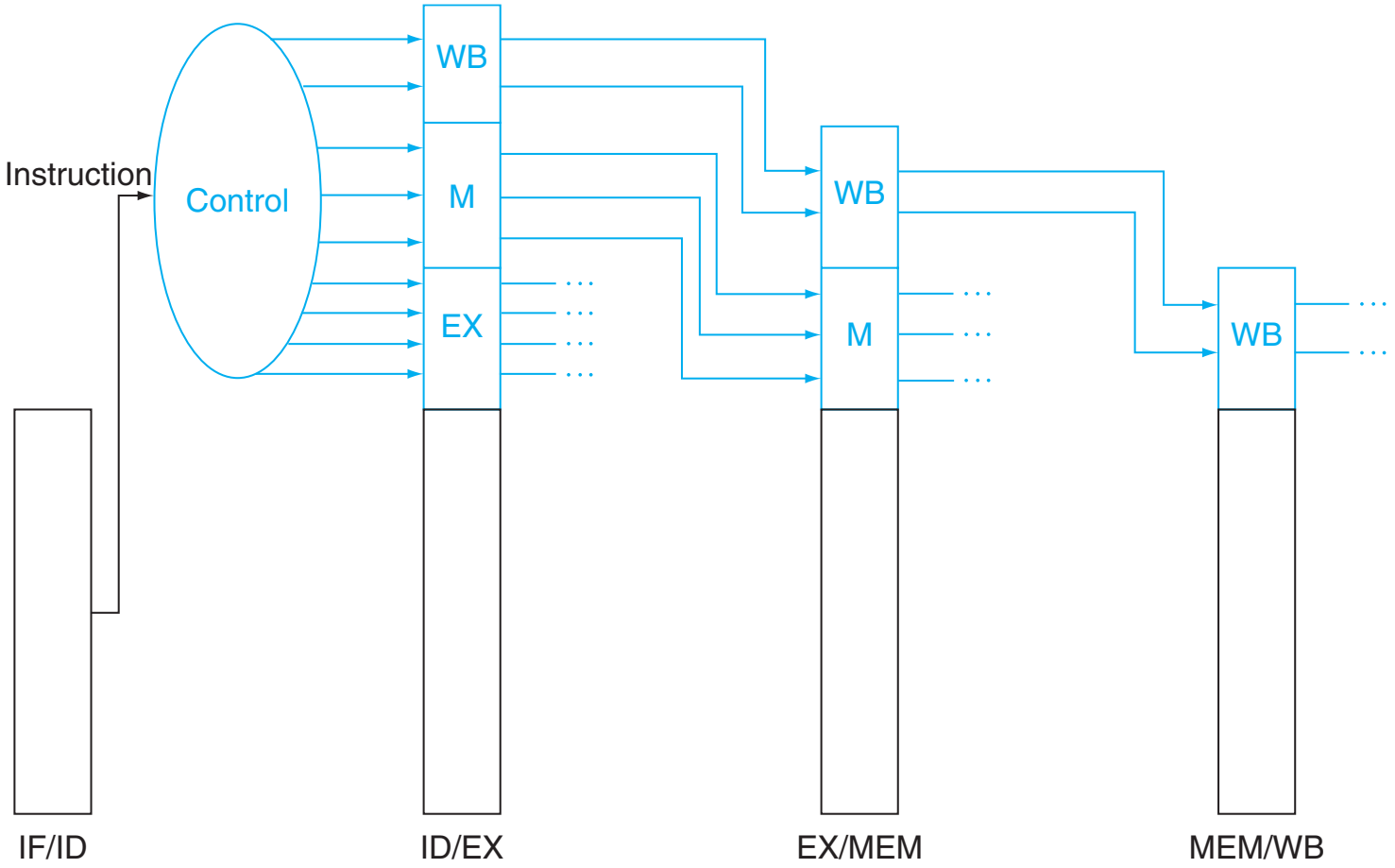
Pipelined Control (Simplified) - similar to non-PL



Same selectors out of Data “Control”, just spread out over various stages

Pipelined Control Scheme

- As in single-cycle implementation, control signals derived from instruction



Pass the signals from stage to stage until they are used, then discard

FIGURE 4.50 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

Pipeline Control Values

- Control signals are conceptually the same as they were in the single cycle CPU.
- "ALU Control" circuit same as before.
- Main control also unchanged. Table below shows same control signals grouped by pipeline stage

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

FIGURE 4.49 The values of the control lines are the same as in Figure 4.18, but they have been shuffled into three groups corresponding to the last three pipeline stages.

Controlled Pipelined CPU

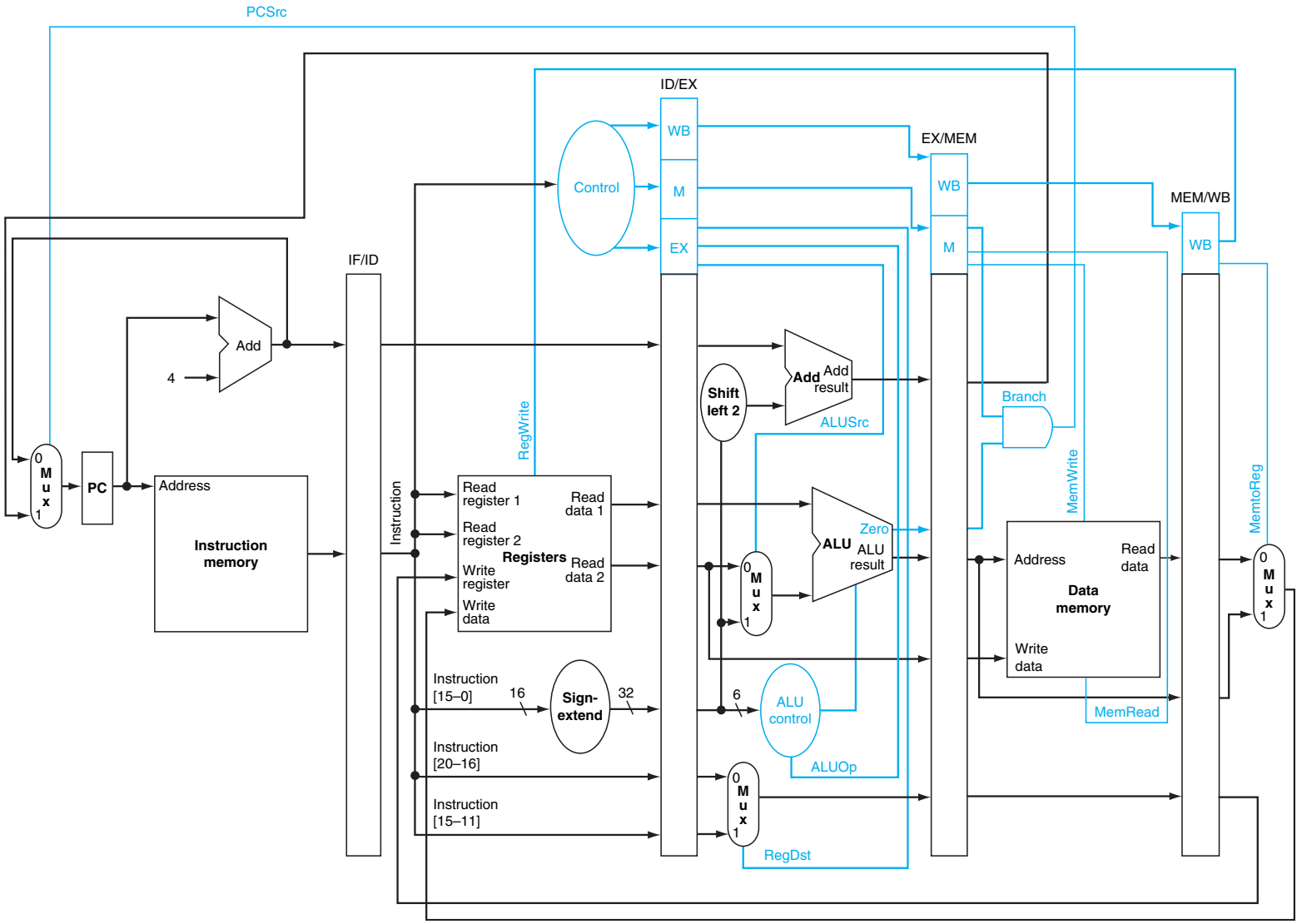


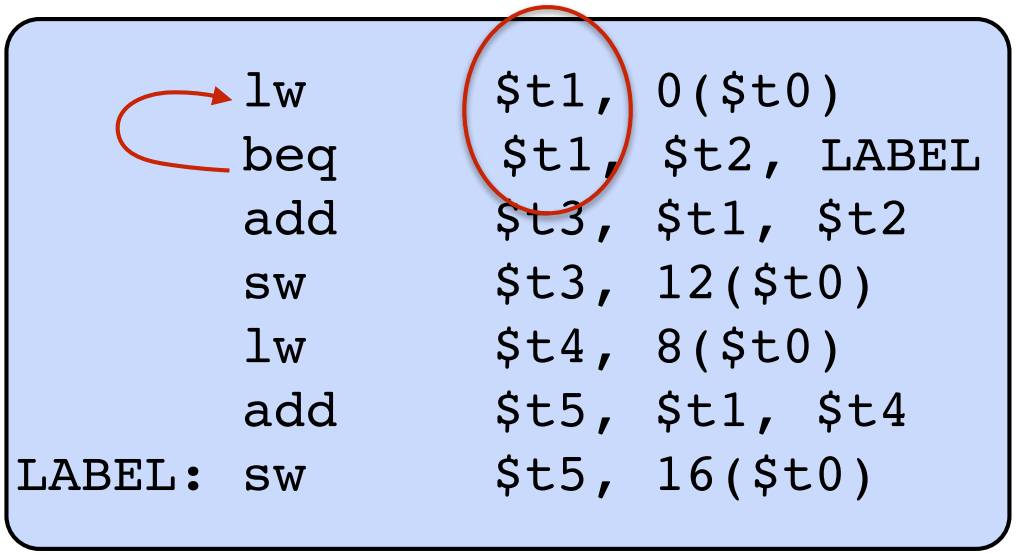
FIGURE 4.51 The pipelined datapath of Figure 4.46, with the control signals connected to the control portions of the pipeline registers. The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

Dependencies in Assembly Code

Types of dependencies

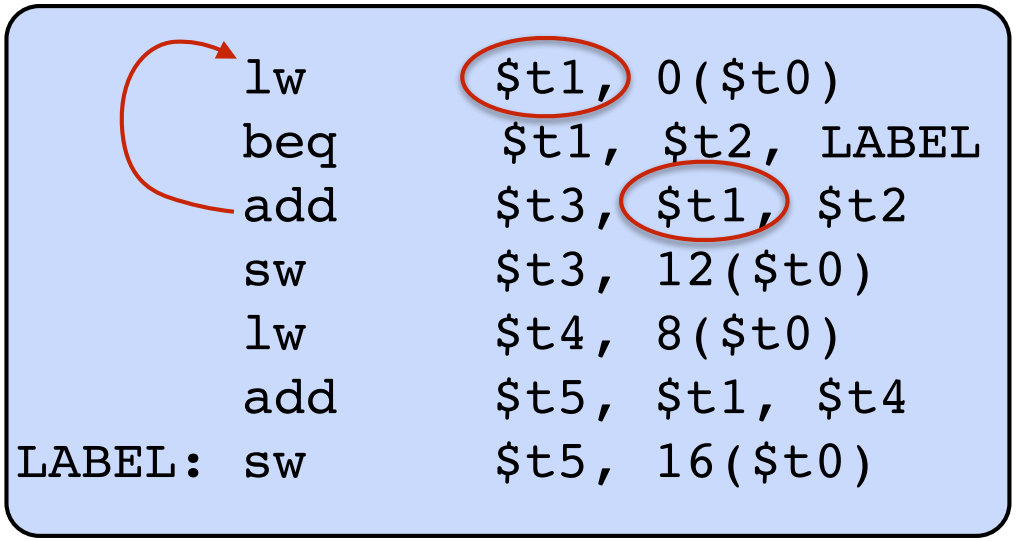
- 2 types of dependencies:
  - **Data dependence:** Instruction B has a data dependence on instruction A when instruction A possibly changes the state (memory or register) of data used by instruction B
  - **Control dependence:** Instruction B has a control dependence on instruction A if instruction A's outcome affects whether instruction B is processed.

Dependency Examples



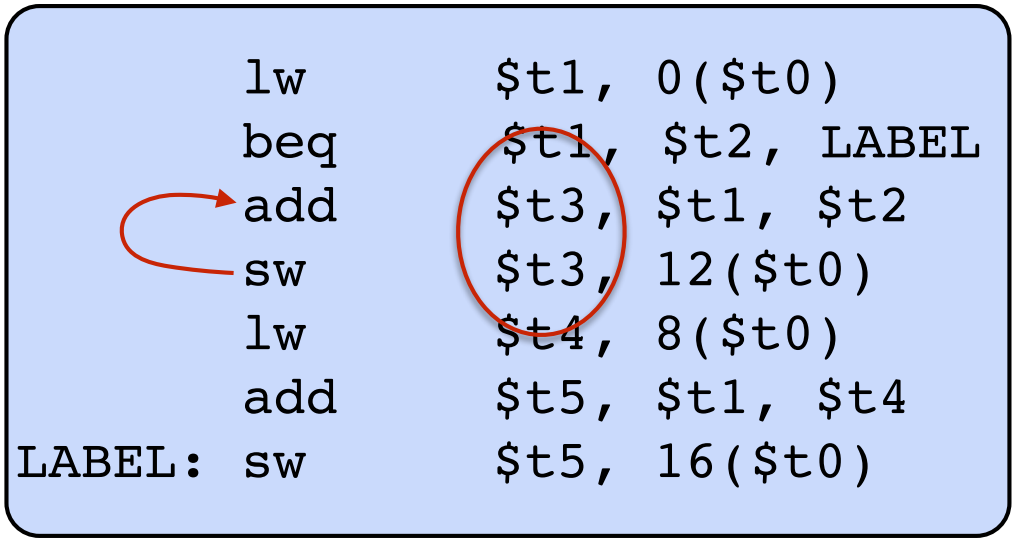
beq $t1, $t2, LABEL data depends on lw $t1, 0($t0)
(lw changes value stored in $t1)

Dependency Examples



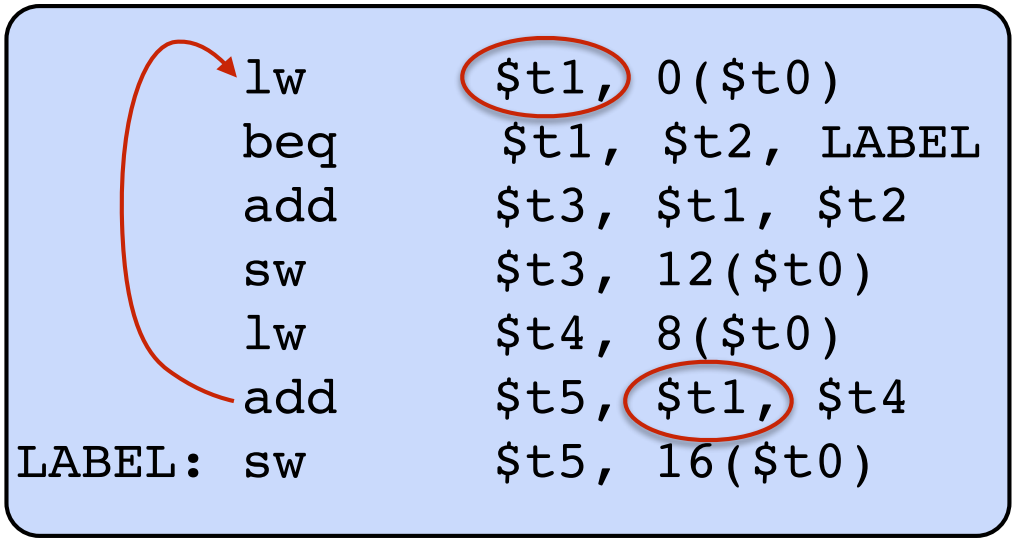
add $t3, $t1, $t2 data depends on lw $t1, 0($t0)

Dependency Examples



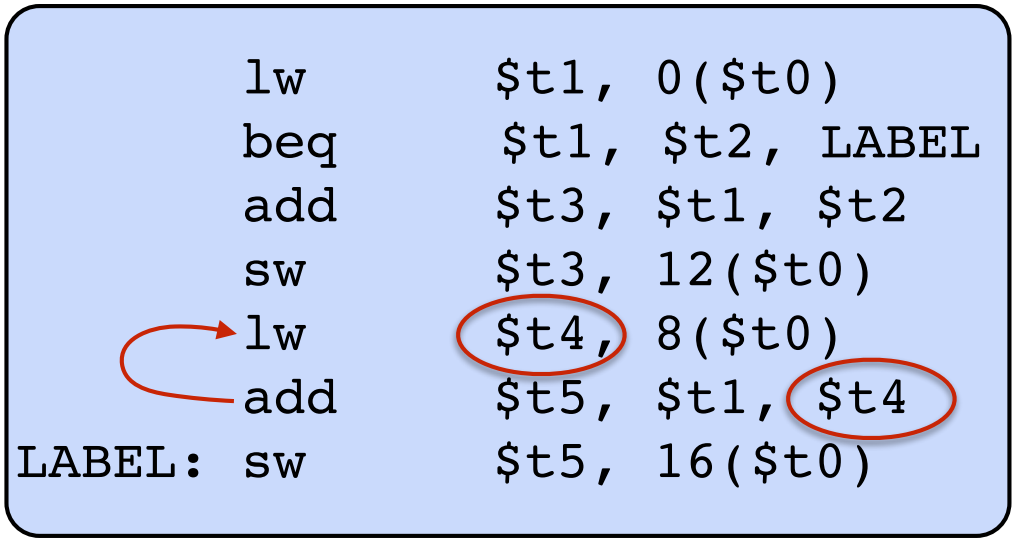
sw $t3, 12($t0) data depends on add $t3, $t1, $t2

Dependency Examples



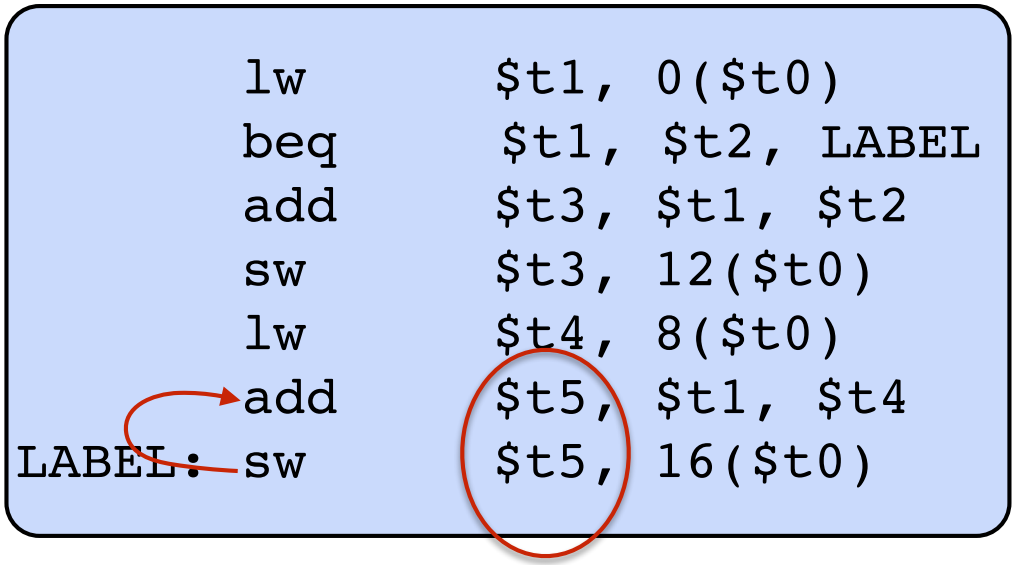
add $t5, $t1, $t4 data depends on lw $t1, 0($t0)

Dependency Examples



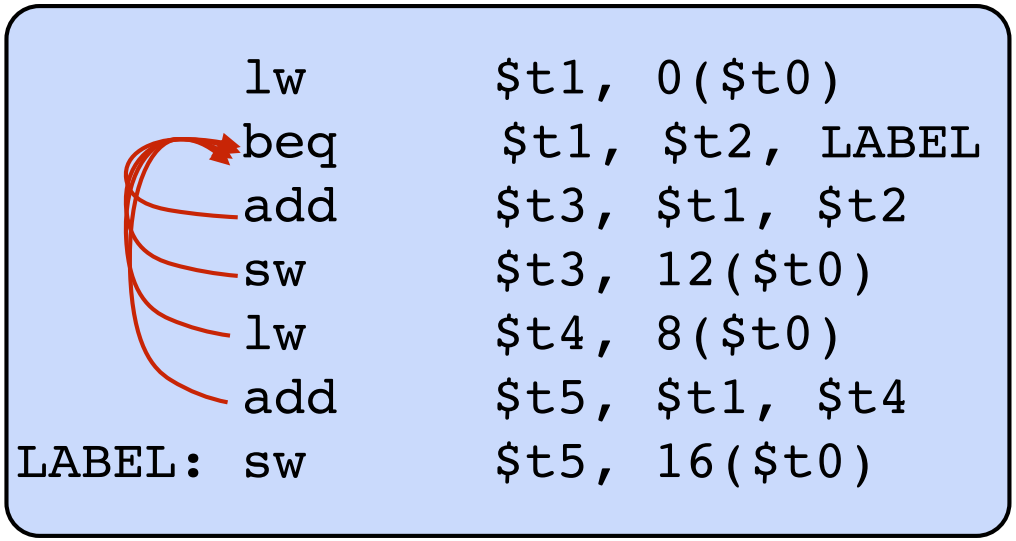
add $t5, $t1, $t4 data depends on lw $t4, 8($t0)

Dependency Examples



sw $t5, 16($t0) data depends on add $t5, $t1, $t4

Dependency Examples



Several instructions following beq control-depend upon it.

Hazards (in the pipeline)

Hazard

- A hazard is a situation where an instruction entering the pipeline will be unable to correctly perform its intended function.

- 3 types of hazards

  - **Structural hazard:** the instruction will need to utilize some circuitry during a clock cycle that will be occupied by another instruction

  - **Data hazard:** the instruction will need access to data that is not available in permanent state (i.e., in register file or memory) during the required clock cycle

  - **Control hazard:** the instruction is not the correct instruction (doesn't belong in the pipeline) or system unsure what instruction enters pipeline next.

Example of Structural Hazard

- Suppose MIPS architecture has 1 memory for both program and data
    - lw instruction needs to read from memory during a clock cycle (i.e., when in MEM stage)
    - In same clock cycle, instruction must be fetched from memory
    - Only 1 word of data readable from memory at a time (in a clock cycle)
- Solution: Replicate hardware, i.e., have separate program and data memories
- In general, structural hazards are resolved by replicated hardware

Example of Control Hazard

Code in pipeline, e.g.,

```
LABEL:
   addi $s2, $s3, 4000
   lw $s1, 8($s4)
   sw $t0, 12 $(s0)
   beq $s1, $t0, LABEL
   or $s4, $s5, $s6
   and $t0, $t1, $t2
```

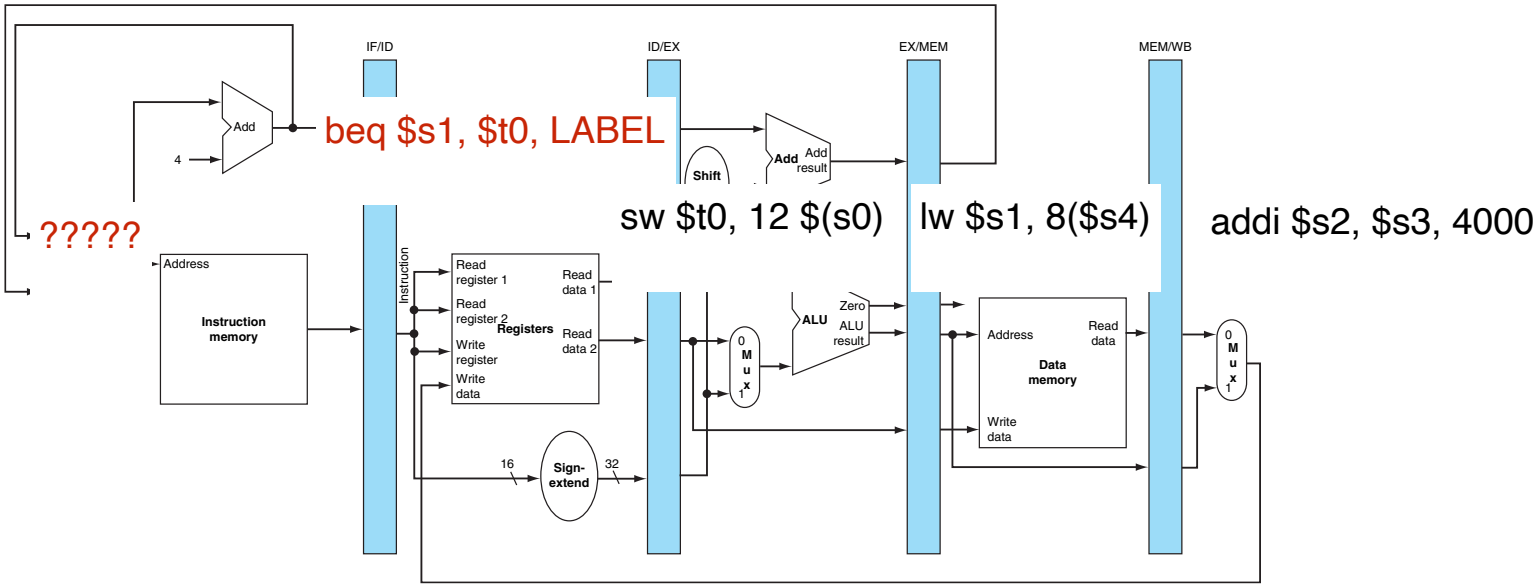
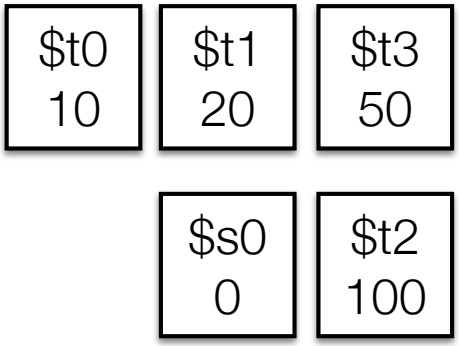


FIGURE 4.35 The pipelined version of the datapath in Figure 4.33. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The registers must be wide enough to store all the data corresponding to the lines that go through them. For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address. We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

- What follows beq into the pipeline? addi $s2,$s3,4000 or or $s4,$s5,$s6?
  - Depends on whether data in $s1 matches that in $t0, won't know answer until beq completes EX stage (where conditional is evaluated)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```



- Let's first look at anticipated behavior (i.e., single cycle)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0	$t1	$t3
10	20	50

$s0	$t2
30	100

- Let's first look at anticipated behavior (i.e., single cycle)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
30	-20

- Let's first look at anticipated behavior (i.e., single cycle)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

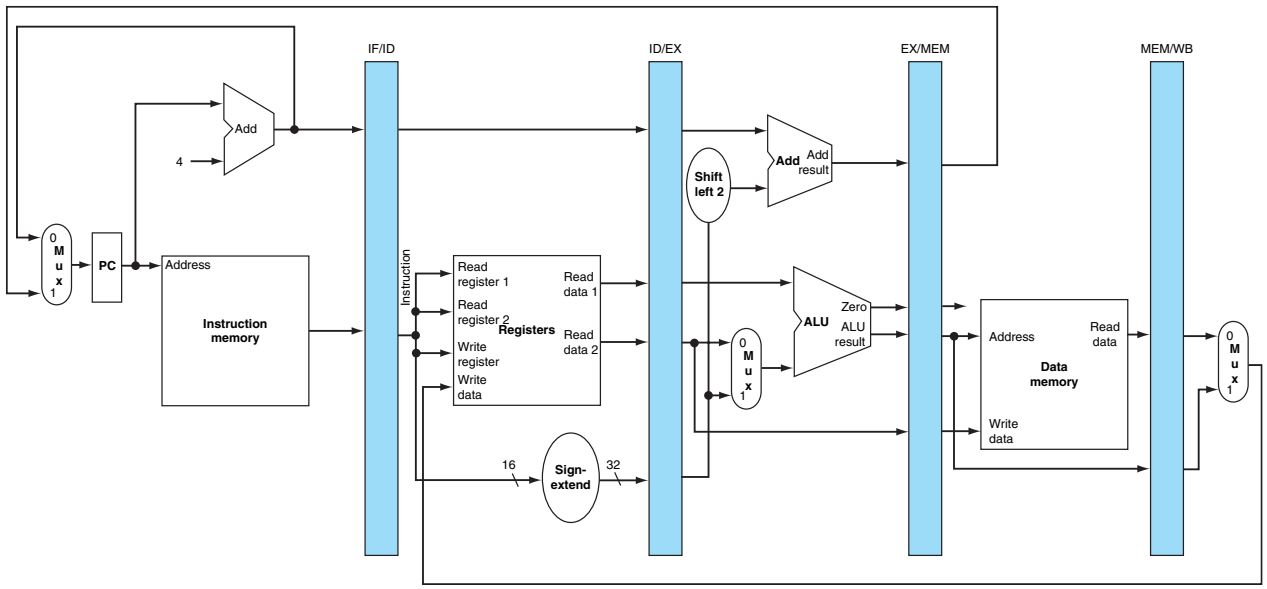
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

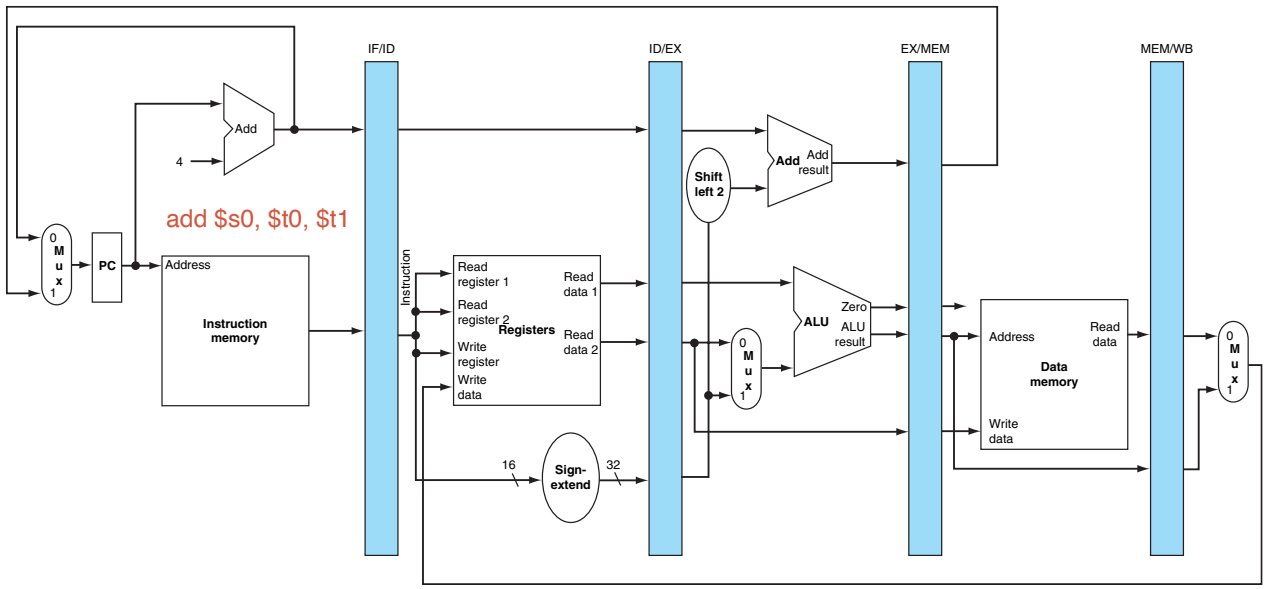
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



- Clock Cycle t: add enters the pipeline

Example of Data Hazard

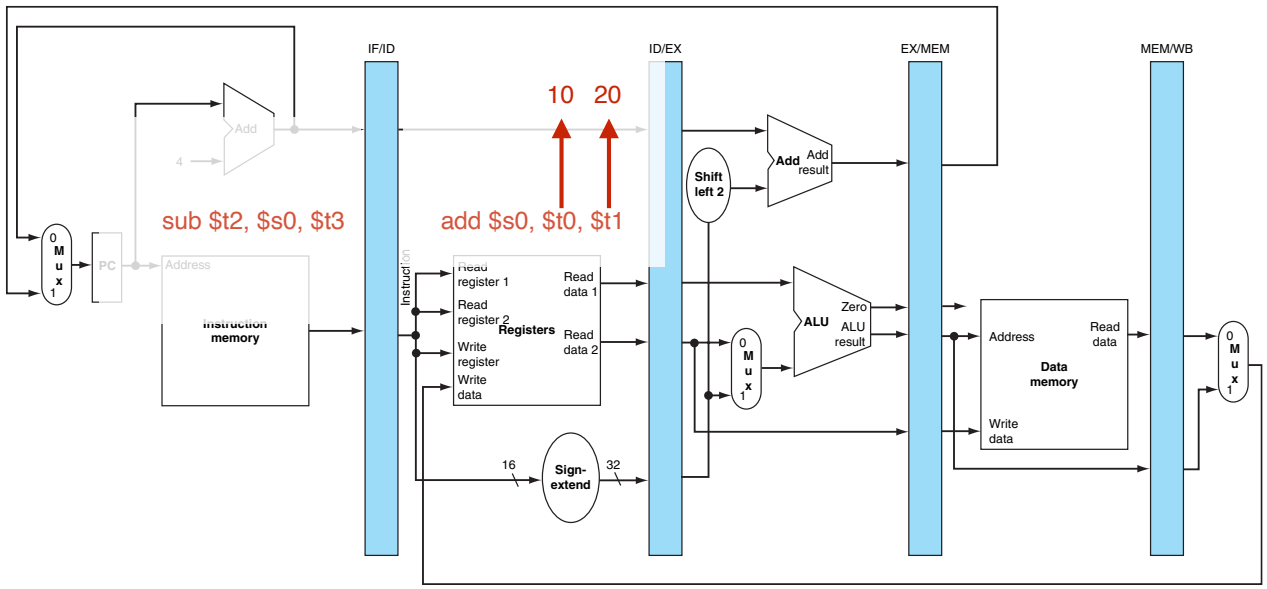
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0 10	$t1 20	$t3 50
$s0 0	$t2 100	



- Clock Cycle t+1: sub enters pipeline, add pulls values for $t0 and $t1 from reg file

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

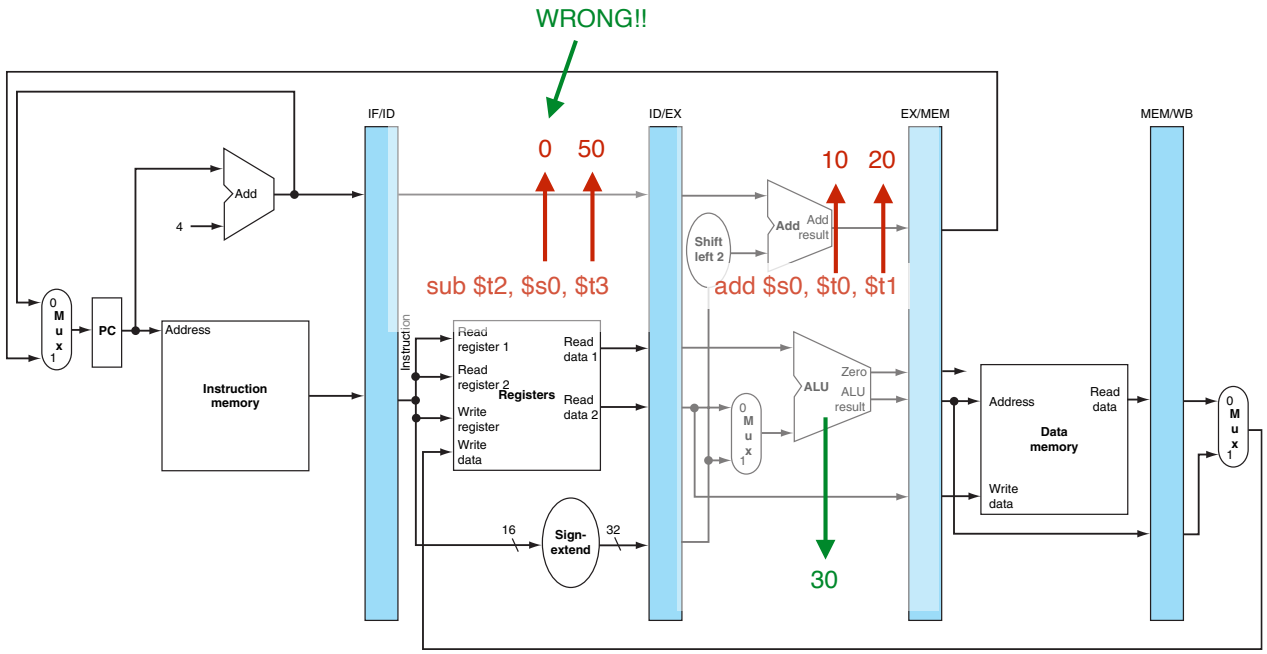
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



- Clock Cycle t+2: sub pulls values out of reg file. Wrong value for $s0!!!
- Problem: 2nd instruction read from reg file before 1st had chance to write

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

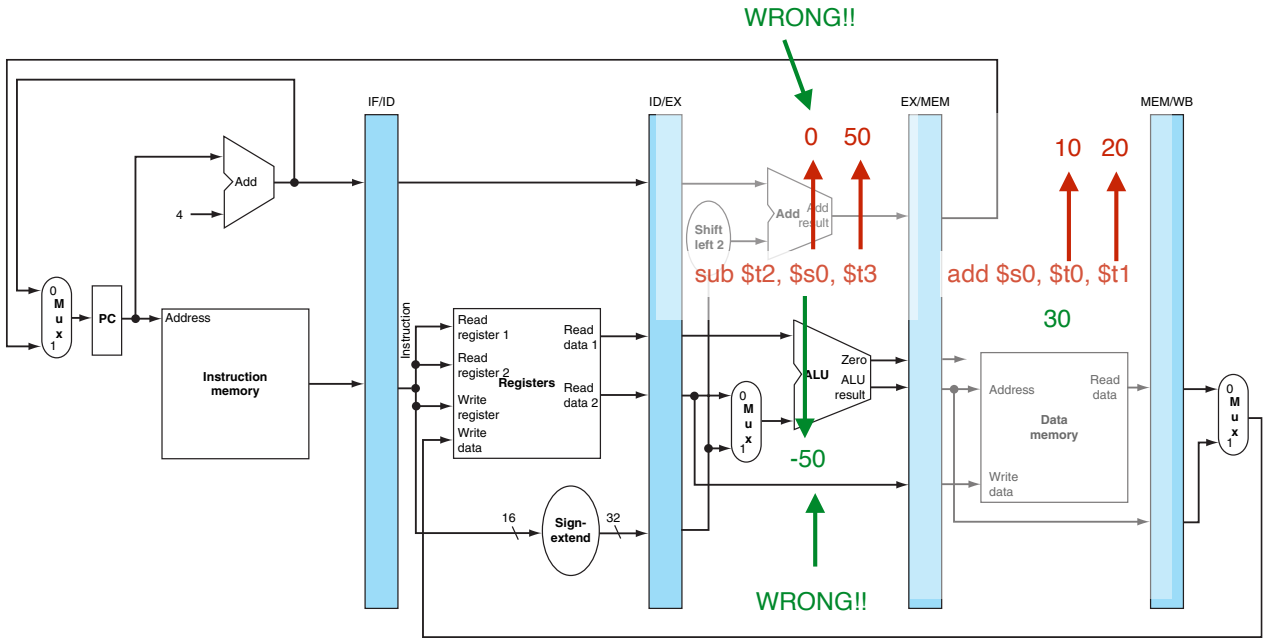
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



- Clock Cycle t+3: sub computes wrong value to go in $s0 (-50 instead of -20)
- add instruction does nothing in MEM stage (doesn't access memory)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

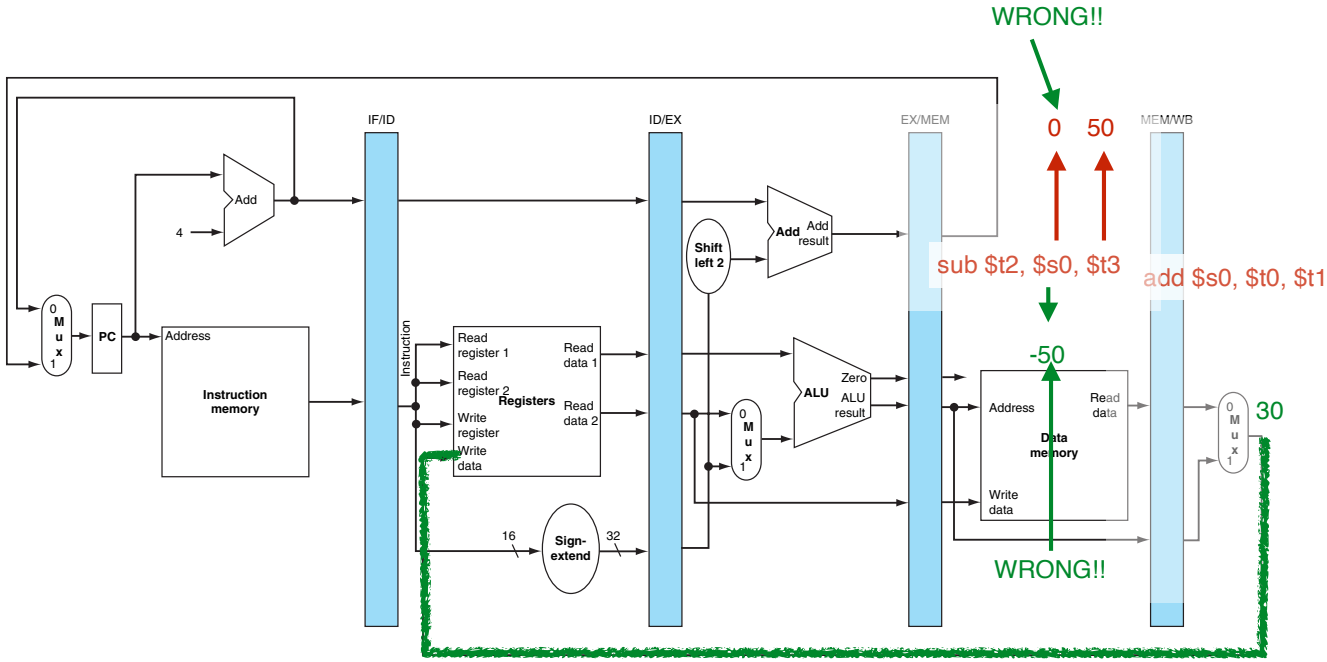
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
30	100



- Clock Cycle t+4: sub computes wrong value to go in $s0 (-50 instead of -20)
- add instruction does nothing in MEM stage (doesn't access memory)

Example of Data Hazard

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

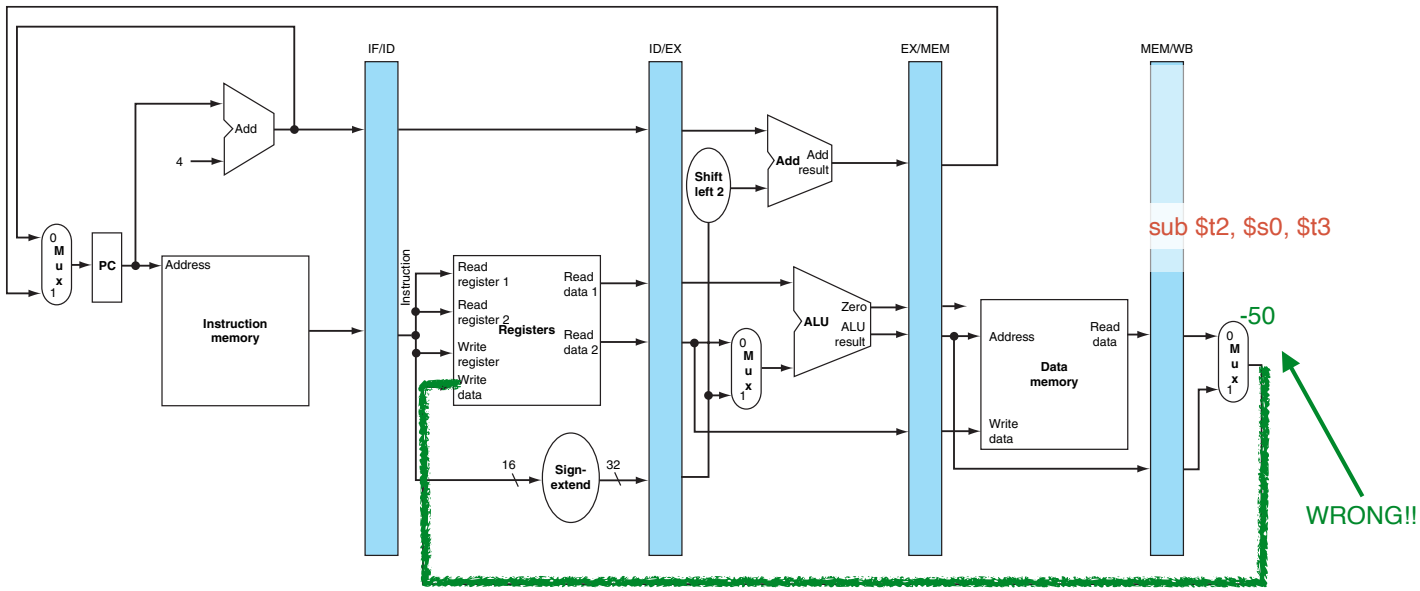
$s0 set during WB phase

$s0 read during ID phase

When snippet ends:
30 should be stored in $s0
-20 stored in $t2

$t0	$t1	$t3
10	20	50

$s0	$t2
30	-50



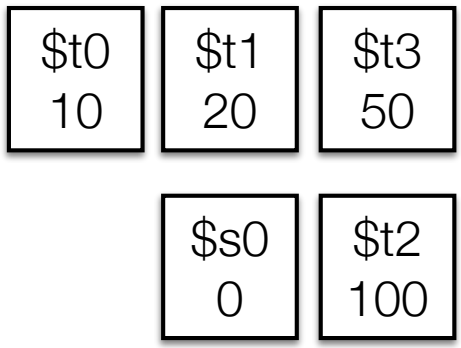
- Clock Cycle t+5: sub writes wrong value into $s0 (-50 instead of -20)

Avoiding Hazards 101: NOP instruction

One way to avoid Data/Control Hazards: NOP

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
nop    # delay 3x
sub   $t2, $s0, $t3
```

NOP is “no operation” instruction:
do nothing (except PC = PC + 4)



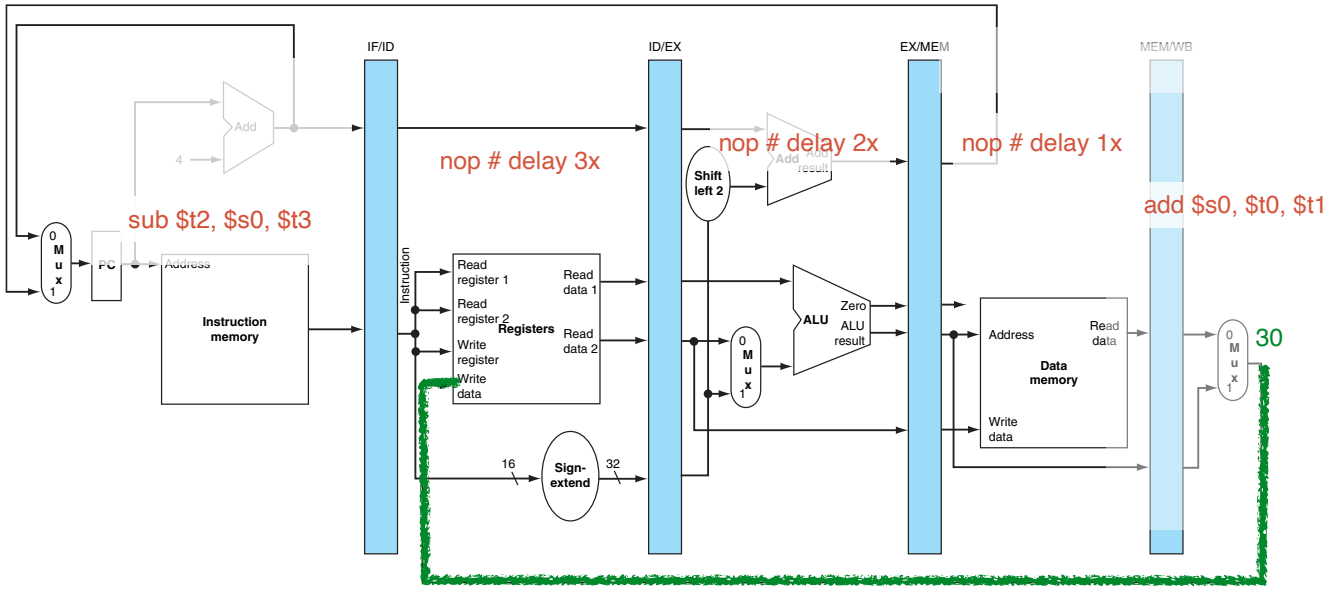
One way to avoid Data/Control Hazards: NOP

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
nop    # delay 3x
sub   $t2, $s0, $t3
```

NOP is an actual MIPS “no operation” instruction: do nothing (except PC = PC + 4)

$t0	$t1	$t3
10	20	50

$s0	$t2
30	100



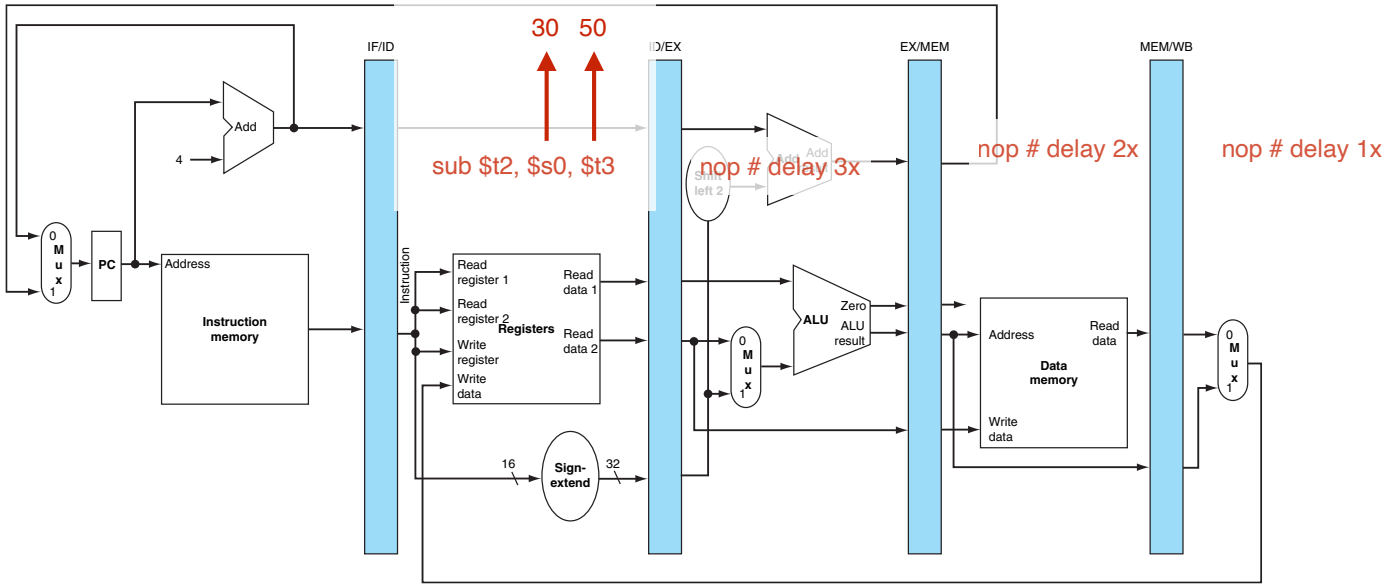
- Delayed sub entering ID stage until after add finished writing to reg file.
- Problem: lose advantage of parallelism from pipelining

One way to avoid Data/Control Hazards: NOP

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
nop    # delay 3x
sub   $t2, $s0, $t3
```

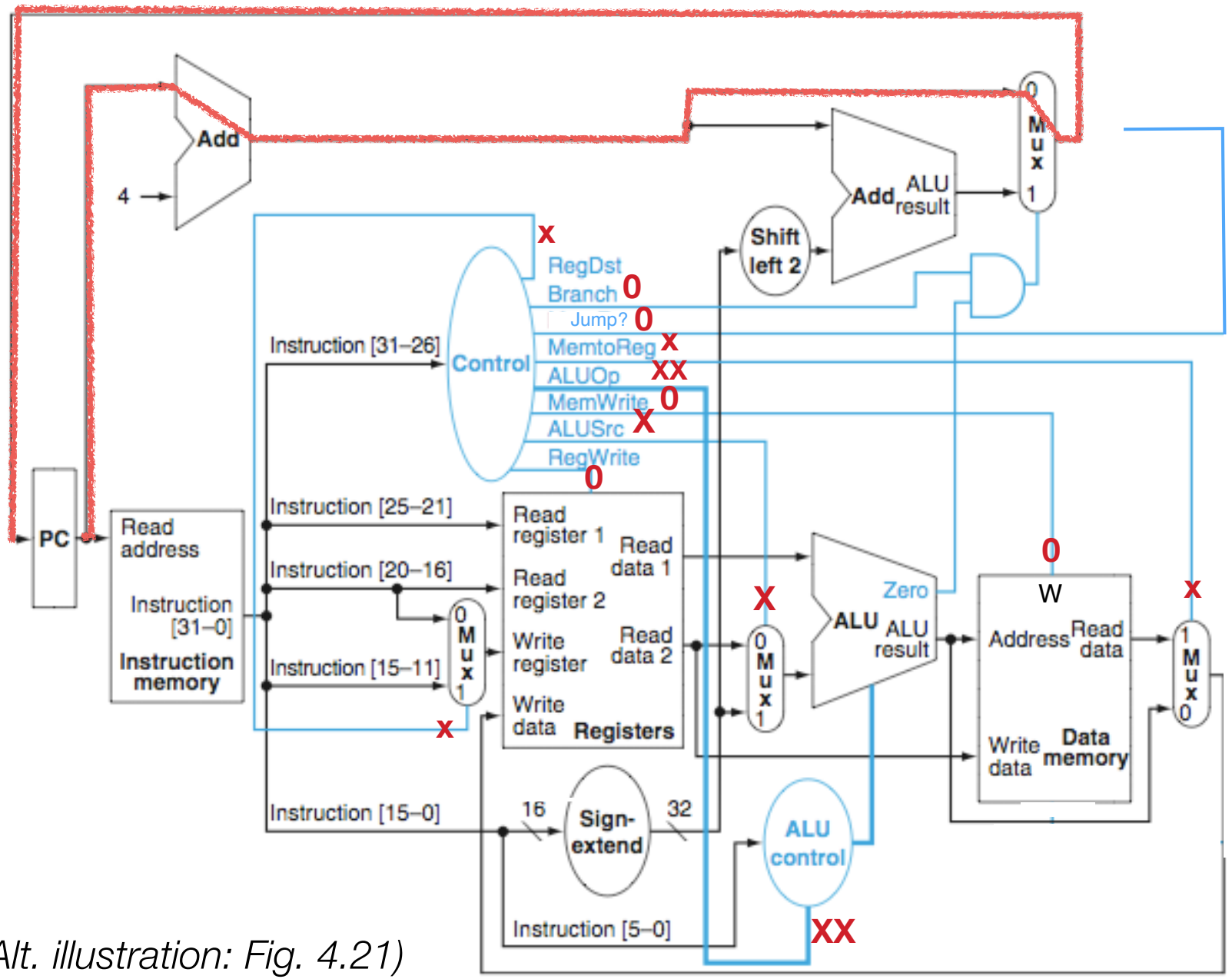
NOP is an actual MIPS “no operation” instruction: do nothing (except PC = PC + 4)

$t0 10	$t1 20	$t3 50
	$s0 30	$t2 100



- Delayed sub entering ID stage until after add finished writing to reg file.
- Problem: lose advantage of parallelism from pipelining

`nop` instruction: how implemented



Control outputs all 0's:

PC = PC + 4
Don't write to reg file
Don't write to MEM

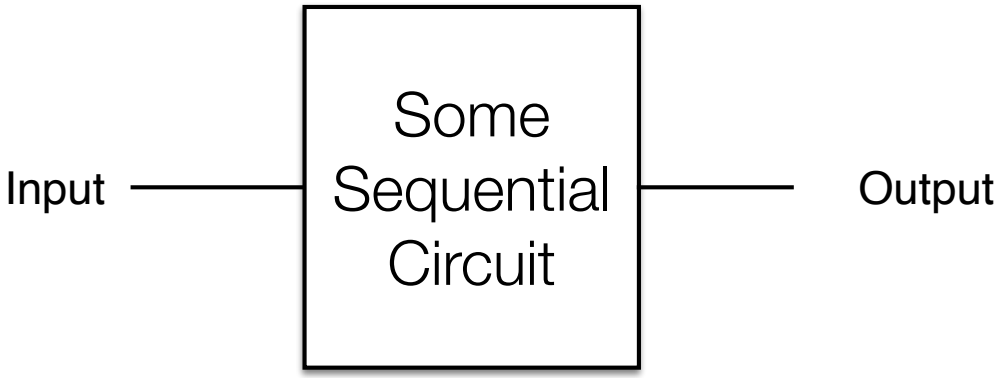
(Alt. illustration: Fig. 4.21)

Instruction is a NOP if doesn’t write to state

- An instruction has no effect as long as it:
  - adjusts PC = PC + 4
  - doesn’t write to memory
  - doesn’t write to register file
- It might still:
  - read from register file
  - use the ALU to do a computation
  - read from memory
- If all the intermediate results get tossed without writing to state, there is no effect

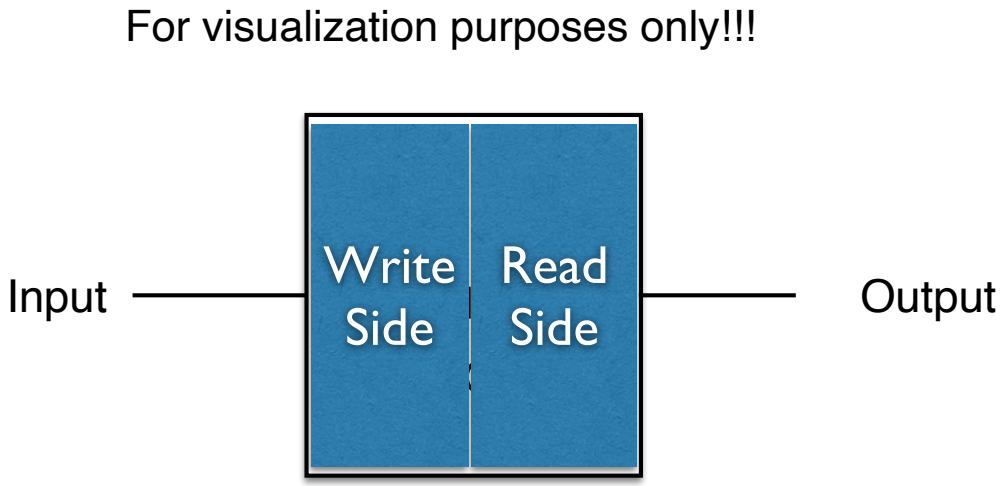
First Pipeline Optimization: Write-before-Read Reg File

General Read-Write format of Circuit



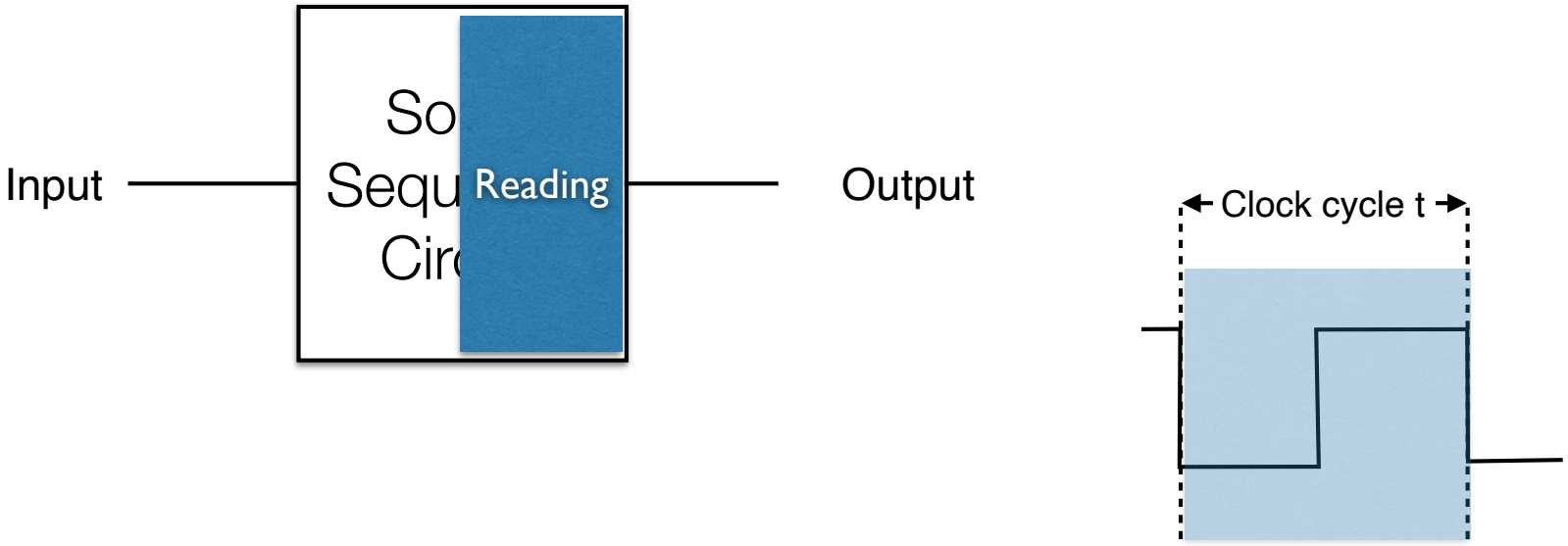
- We've always drawn circuits in this general form (e.g., Reg File)

General Read-Write format of Circuit



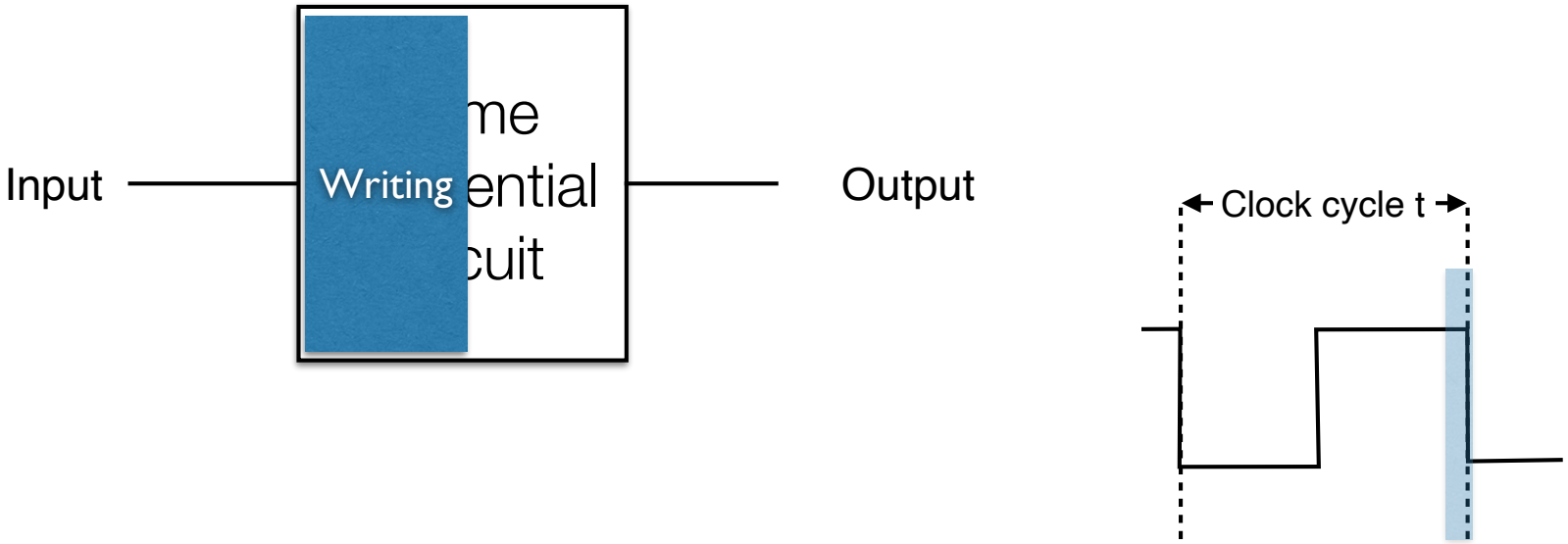
- We've always drawn circuits in this general form (e.g., Reg File)
- **As a visual only**: Artificially delineate a "Read Side" and "Write Side"
- Until now, we assumed the "read side" values changed at the end of the clock cycle when the "write side" got written to
- i.e., assumed Data written in clock cycle t available during clock cycle t+1

Traditional view: Read interval



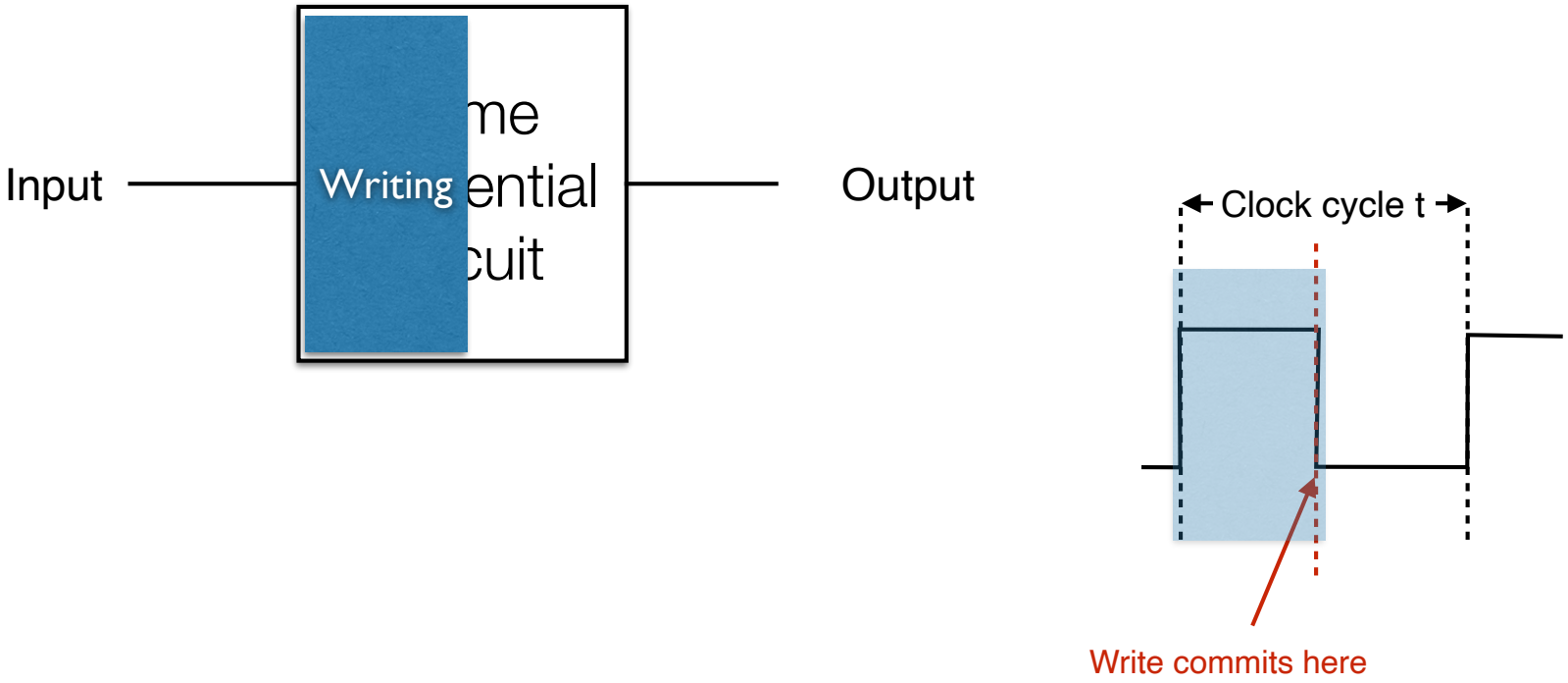
- Traditionally, assume reading can be done during full clock cycle

Traditional view: Read interval



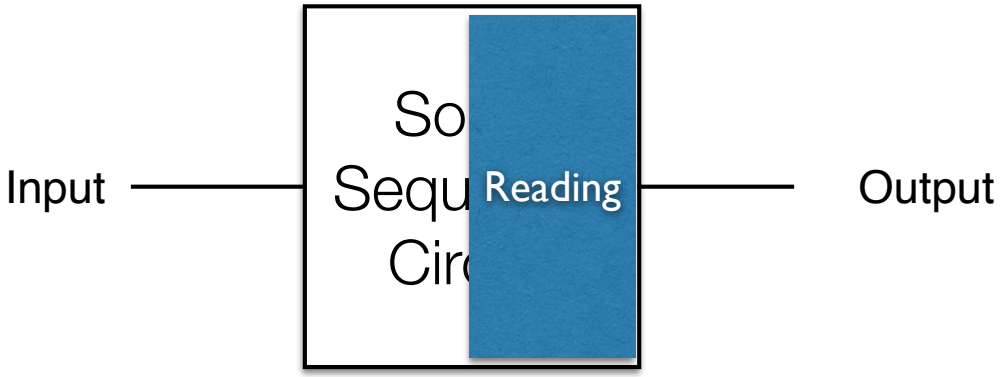
- Writing happens at the very end of the clock cycle

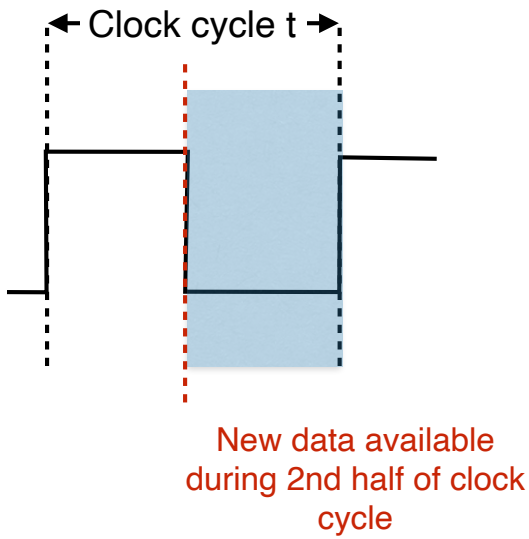
Revised View: Writing happens first, then reading



- At end of first half of clock cycle, reg file gets written to

New Assumption

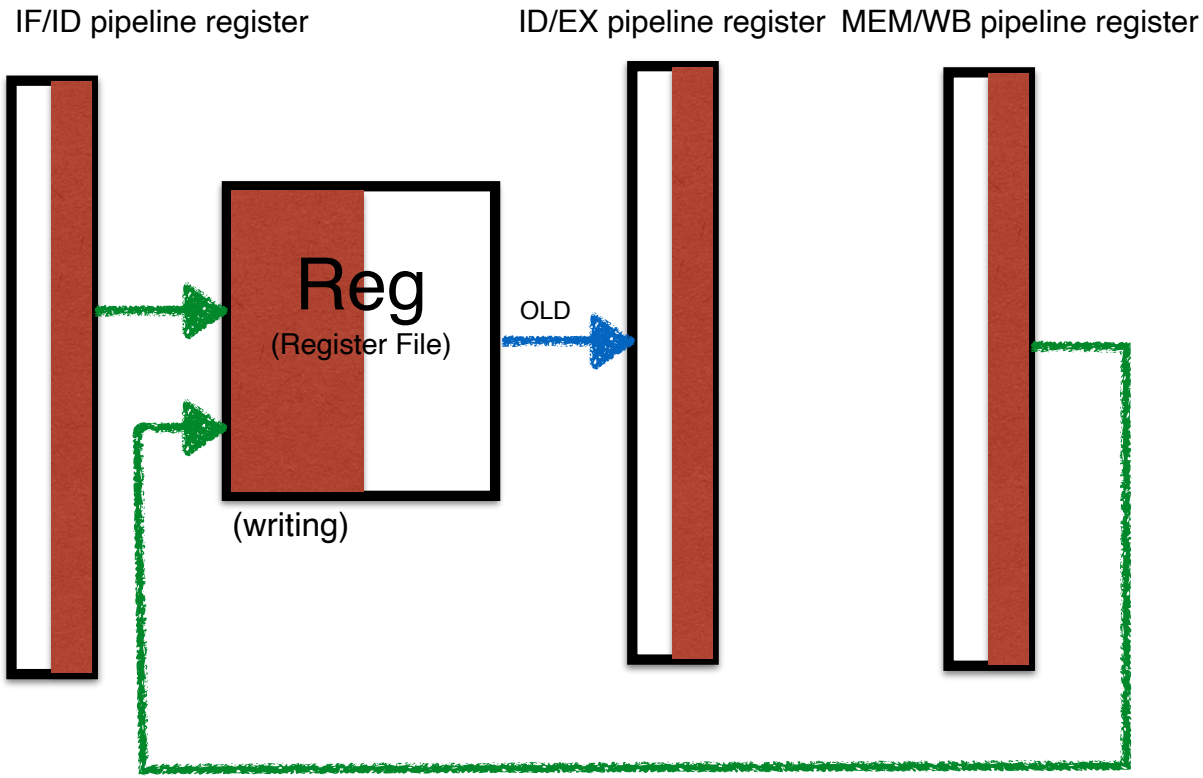


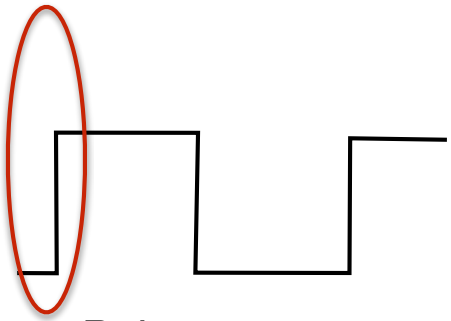


- At end of first half of clock cycle, reg file gets written to
- During second half of clock cycle, reg file is read from: yielding values written in first half

How to Implement Write before Read

- Won't be tested on this, just FYI…
- Make Pipeline registers positive pulse triggered
- Register File stays negative-pulse triggered

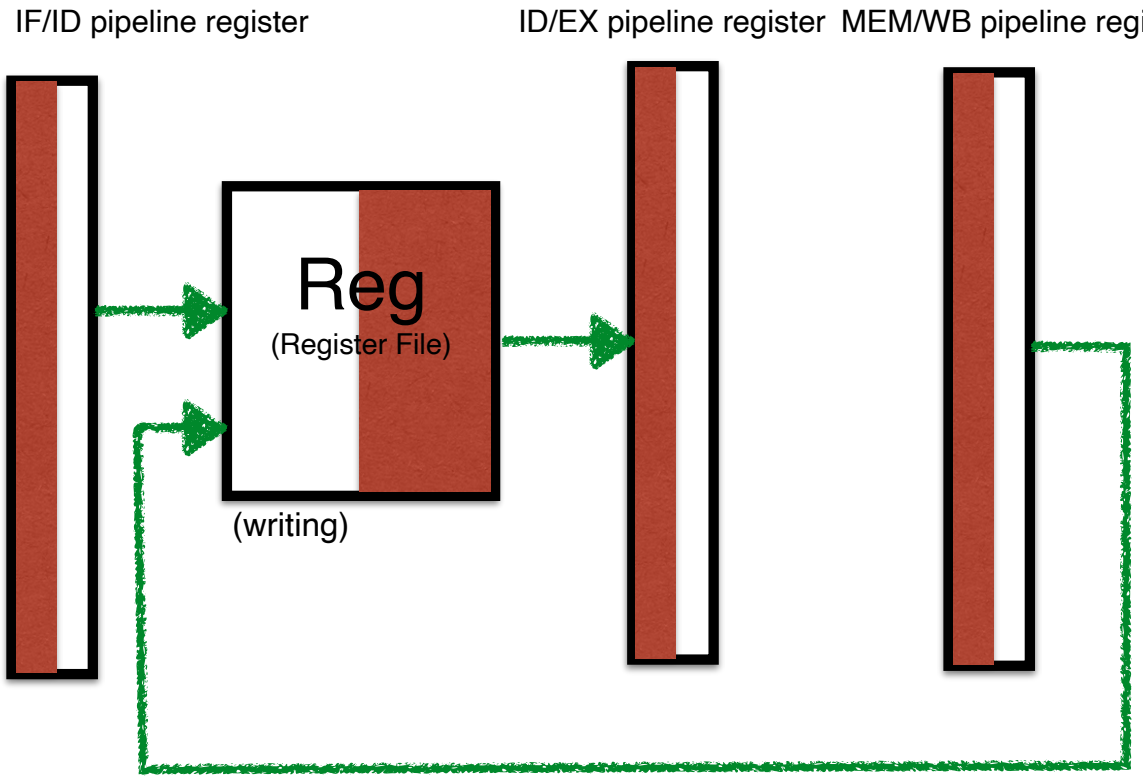


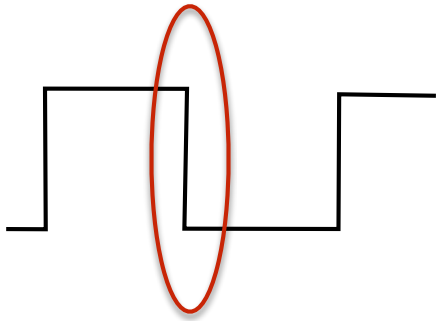


Pulse goes up:
pipeline regs, just written to, read out new (green) values: "old" data still coming out of reg file

How to Implement Write before Read

- Won't be tested on this, just FYI…
- Make Pipeline registers positive pulse triggered
- Register File stays negative-pulse triggered

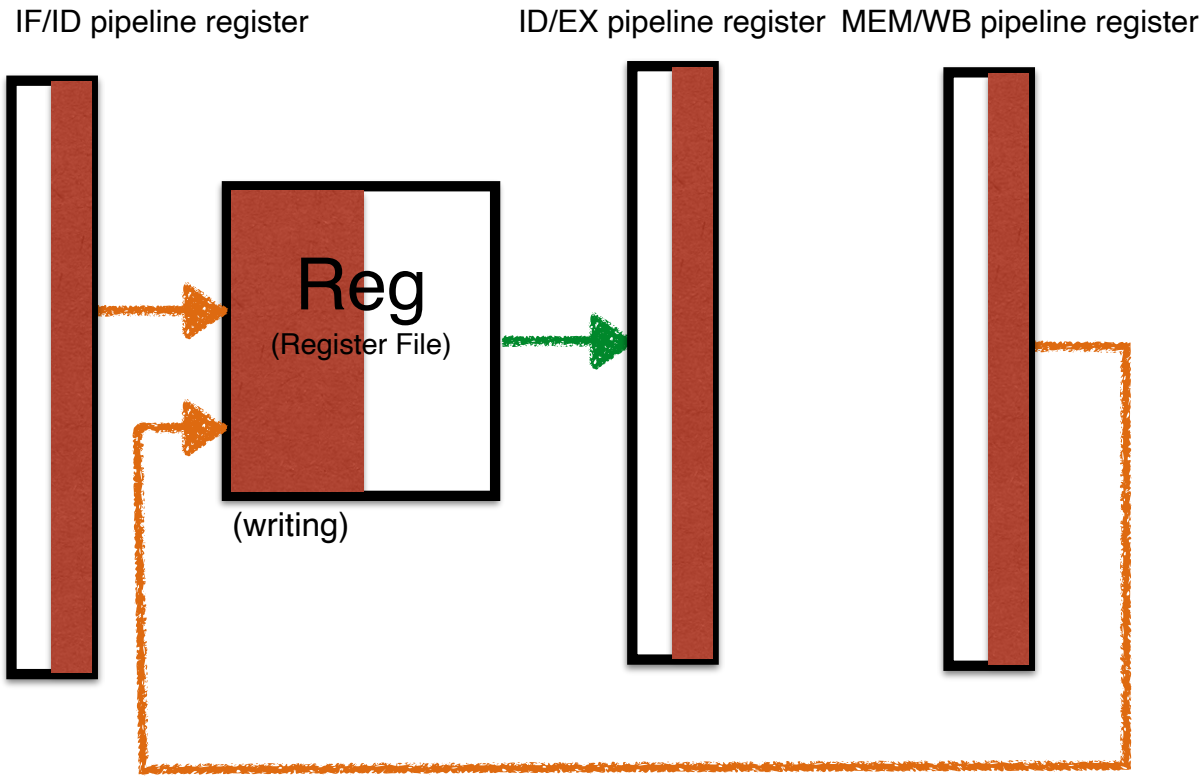


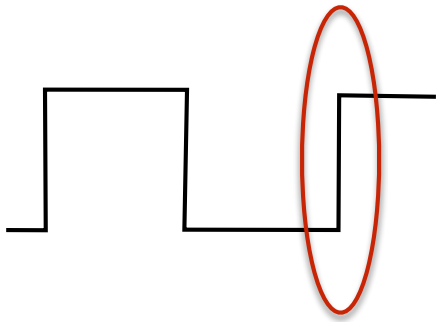


Pulse goes down: new (green) values written into Reg File and are also output

How to Implement Write before Read

- Won't be tested on this, just FYI…
- Make Pipeline registers positive pulse triggered
- Register File stays negative-pulse triggered





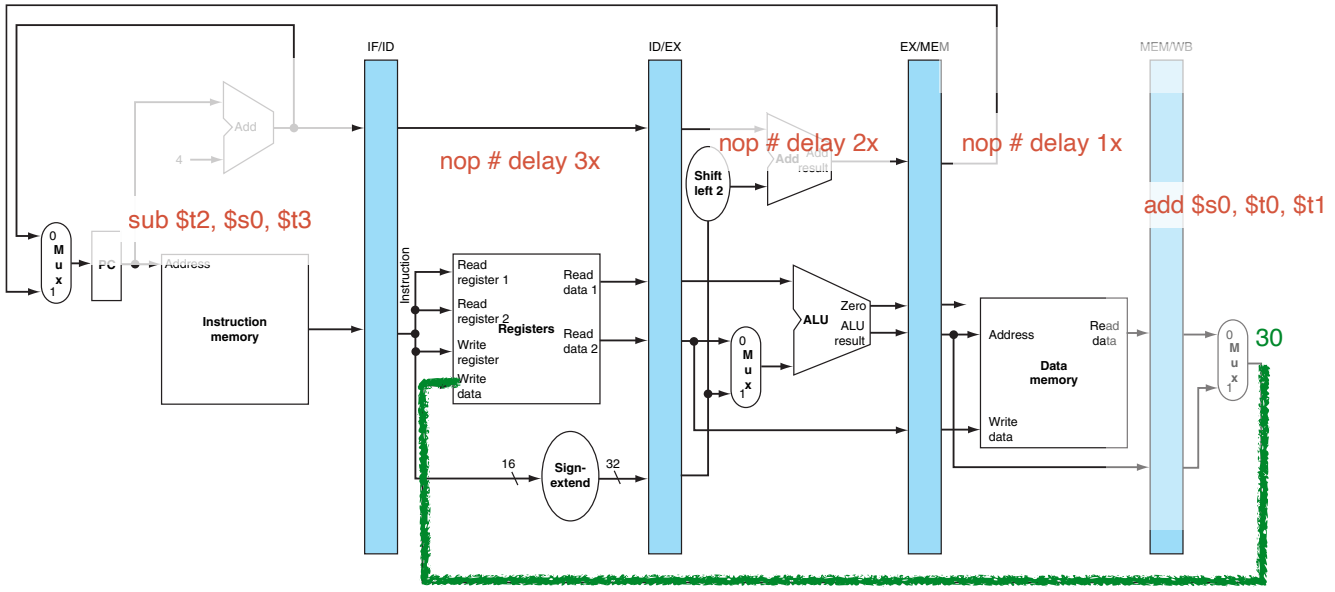
Pulse goes up, values written into pipeline regs: next cycle's data starts

Recall: 3 NOPs

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
nop    # delay 3x
sub   $t2, $s0, $t3
```

- Traditional Read-Write semantics of registers required 3 NOPs
- e.g., add instr. writes to reg file @ end of clock cycle t+4
- sub reads from reg file @ start of clock cycle t+5

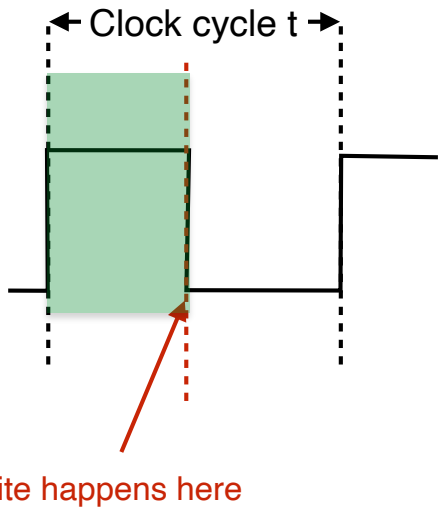
$t0 10	$t1 20	$t3 50
$s0 30	$t2 100	

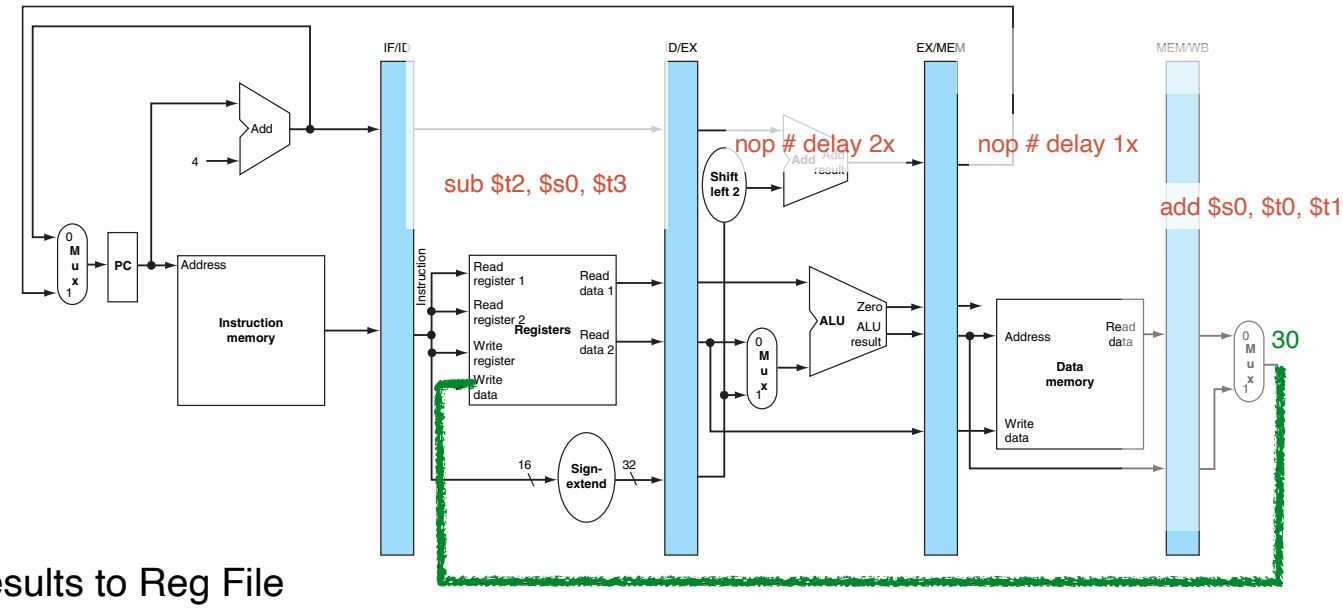


Optimization 1: Reg file does write before read

- Instead of writing happening at end of clock cycle:
  - let write happen in middle of clock cycle for Reg File
  - these written values can then be read @ end of clock cycle
  - Pipeline registers still written to at end of clock cycle

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
sub   $t2, $s0, $t3
```



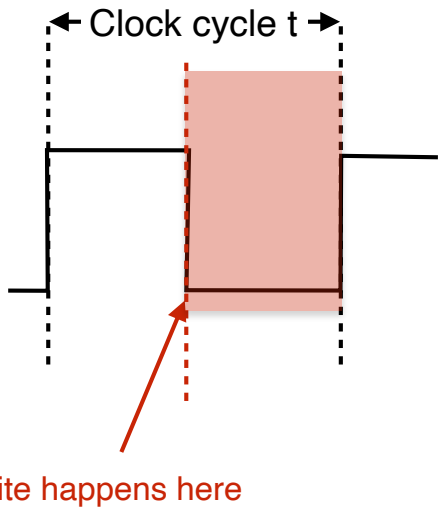


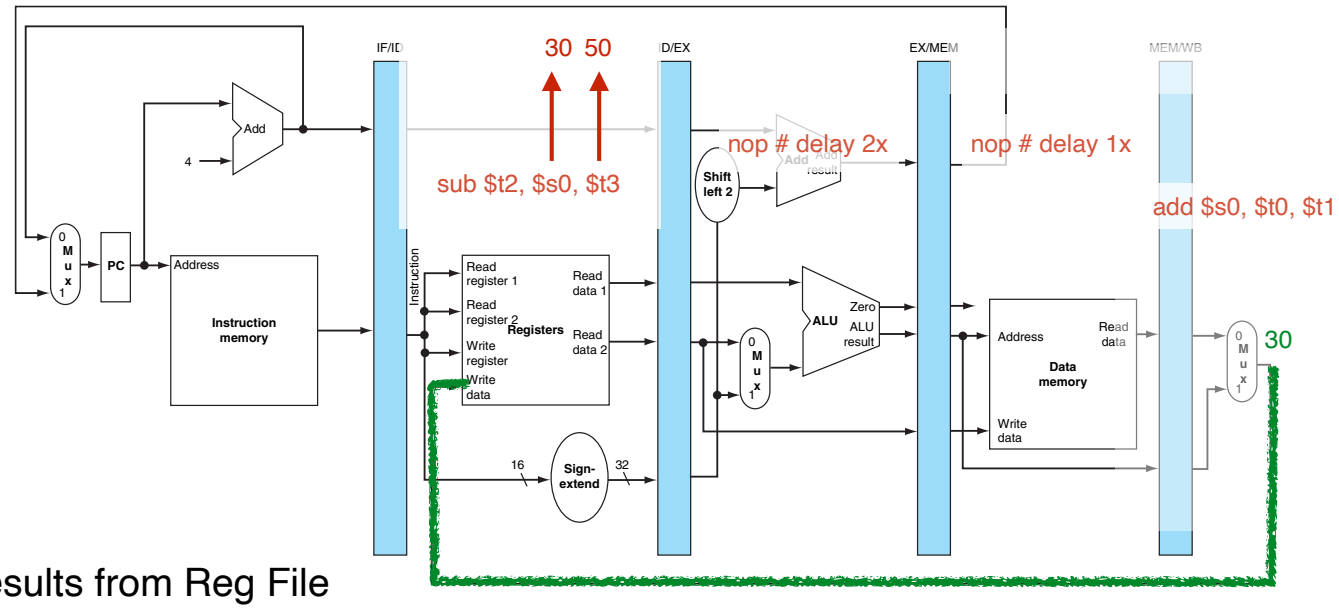
1st half cycle t: add instr. writes results to Reg File

Optimization 1: Reg file: write before read

- Instead of writing happening at end of clock cycle:
  - let write happen in middle of clock cycle for Reg File
  - these written values can then be read @ end of clock cycle
  - Pipeline registers still written to at end of clock cycle

```
add   $s0, $t0, $t1
nop    # delay 1x
nop    # delay 2x
sub   $t2, $s0, $t3
```





2nd half cycle t: sub instr. reads results from Reg File

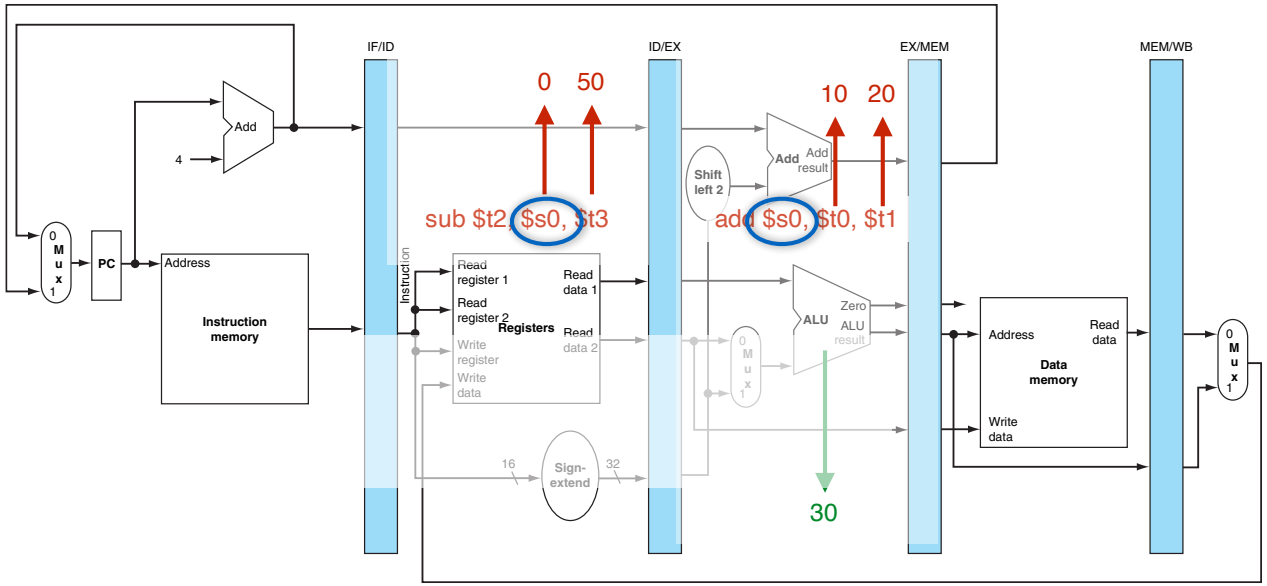
Timing for components of Pipeline Architecture

Assumed Timing for Components

- PC: changes at end of clock cycle (clocked like pipeline registers)
- Reg File: write, then read
- ALU: requires full clock cycle to compute result
- Memory: requires full clock cycle to read or write

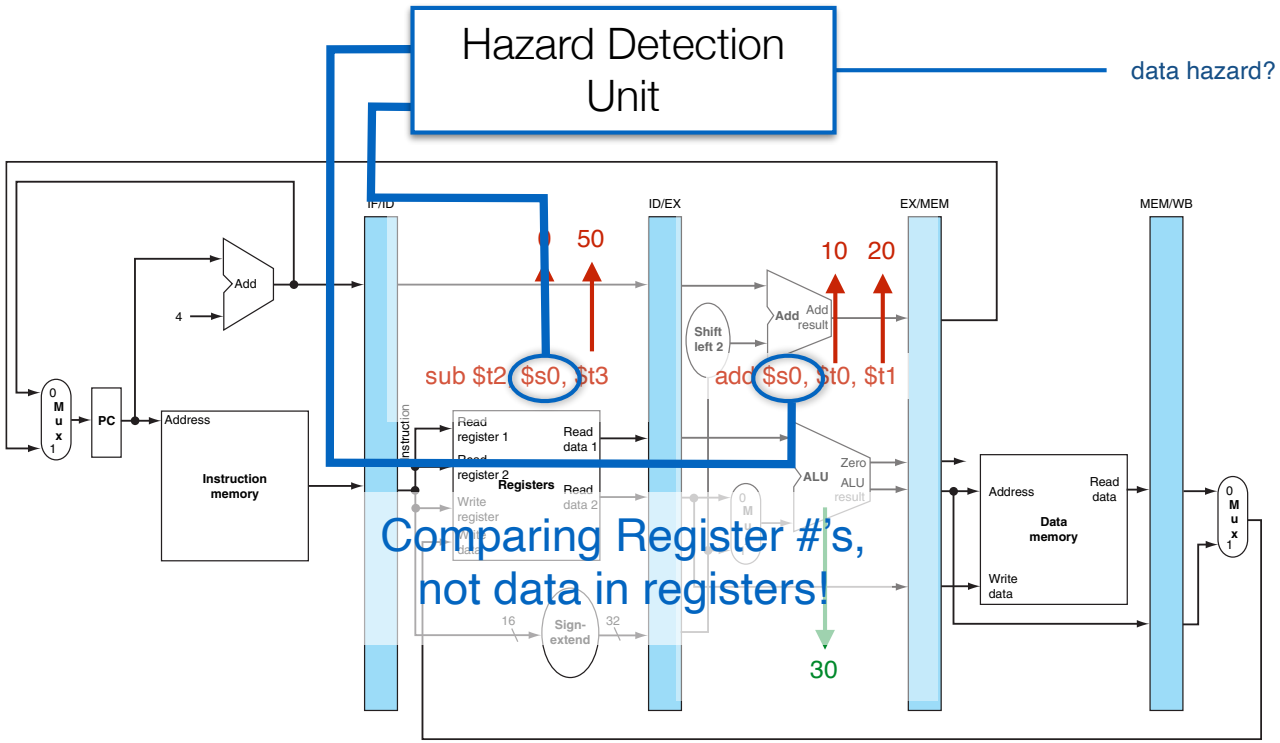
Data Hazard Detection

Add'l Circuitry could detect hazard



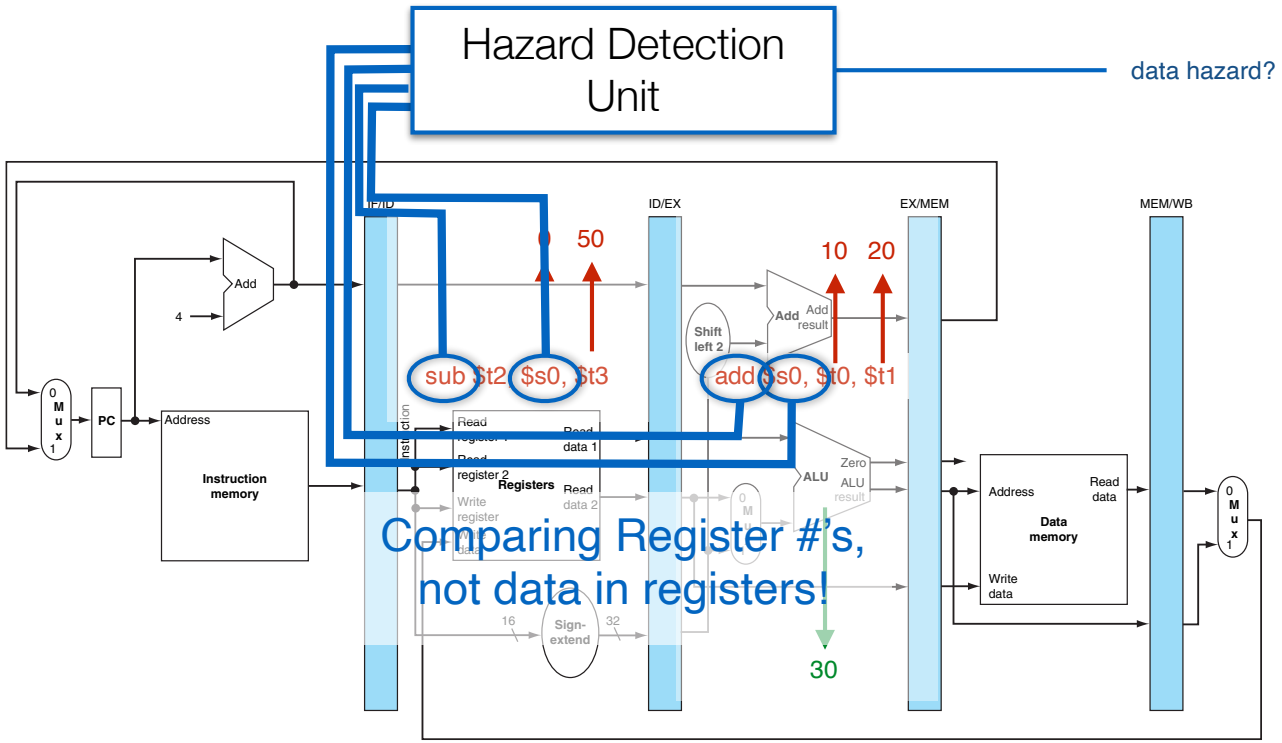
- In this clock cycle, both instructions are in the pipeline
- Could add additional (combinatorial) circuitry to identify:
  - Instruction in stage 2 is reading from a register being written to by instruction in stage 3

Add'l Circuitry could detect hazard



- In this clock cycle, both instructions are in the pipeline
- Could add additional (combinatorial) circuitry to identify:
  - Instruction in stage 2 is reading from a register being written to by instruction in stage 3

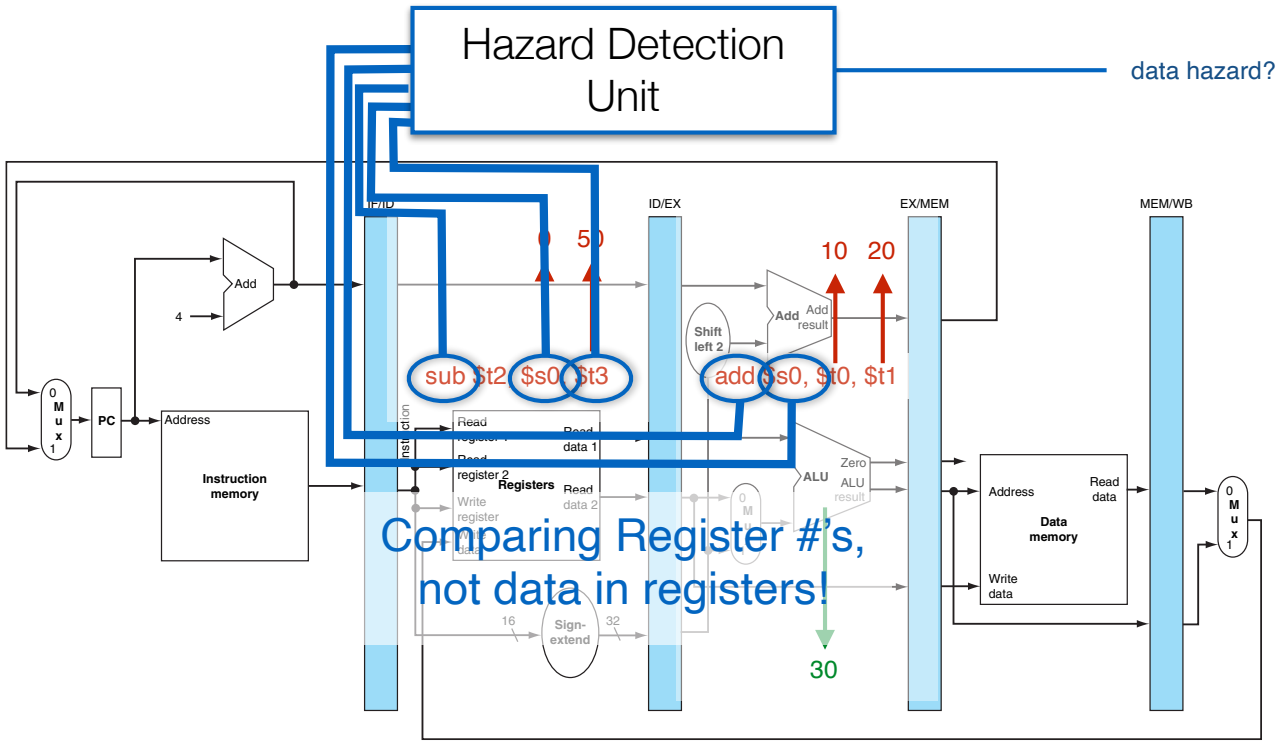
Add'l Circuitry could detect hazard



Also verify (through OpCodes) that instructions are reading and writing to reg file

- In this clock cycle, both instructions are in the pipeline
- Could add additional (combinatorial) circuitry to identify:
  - Instruction in stage 2 is reading from a register being written to by instruction in stage 3

Add'l Circuitry could detect hazard



Also check the other read parameter (if R-type instruction)

- In this clock cycle, both instructions are in the pipeline
- Could add additional (combinatorial) circuitry to identify:
  - Instruction in stage 2 is reading from a register being written to by instruction in stage 3

Data Hazard Handling

Handling Data Hazards

- 2 approaches to dealing with Data Hazards
  - Stalling: delay the 2nd instruction in the pipeline
  - Data Forwarding: route data in pipeline between instructions

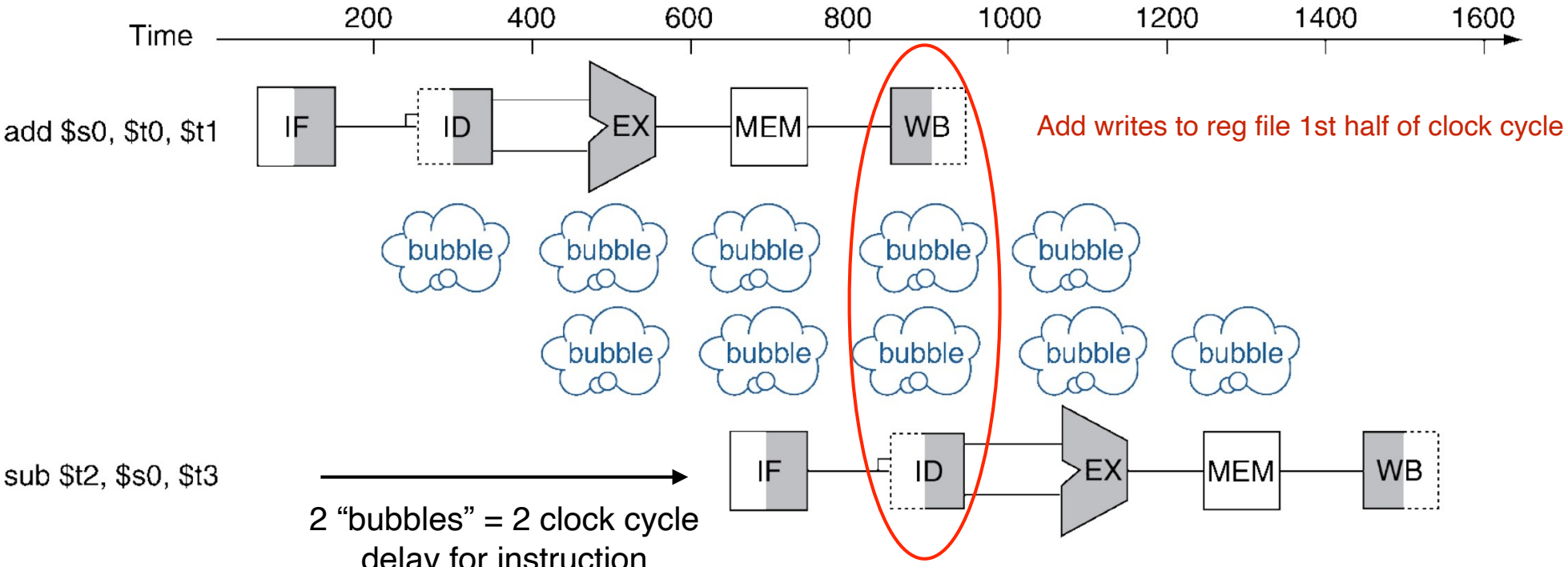
Stalling Conceptually: Bubbles in pipeline

- Delay the 2nd instruction entering the pipeline (have bubbles in the pipeline)

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$s0 set during WB phase

$s0 read during ID phase



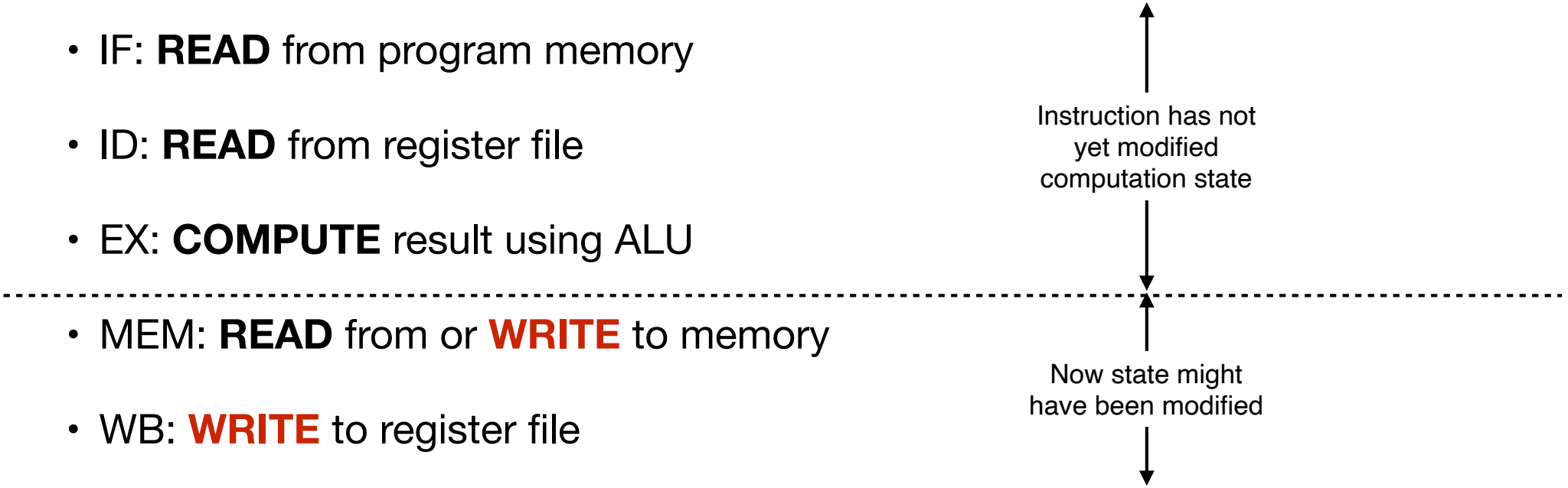
Bubbles are NOPs (no-operation instructions)

```
add   $s0, $t0, $t1
nop # do nothing
nop # do nothing
sub   $t2, $s0, $t3
```

- Option 1: Require user to know about pipelining and place NOPs in their code?
- Option 2: Have assembler put NOPs in?
- Option 3: The hardware puts in the NOP

Abortability of an Instruction

- IF: **READ** from program memory
- ID: **READ** from register file
- EX: **COMPUTE** result using ALU

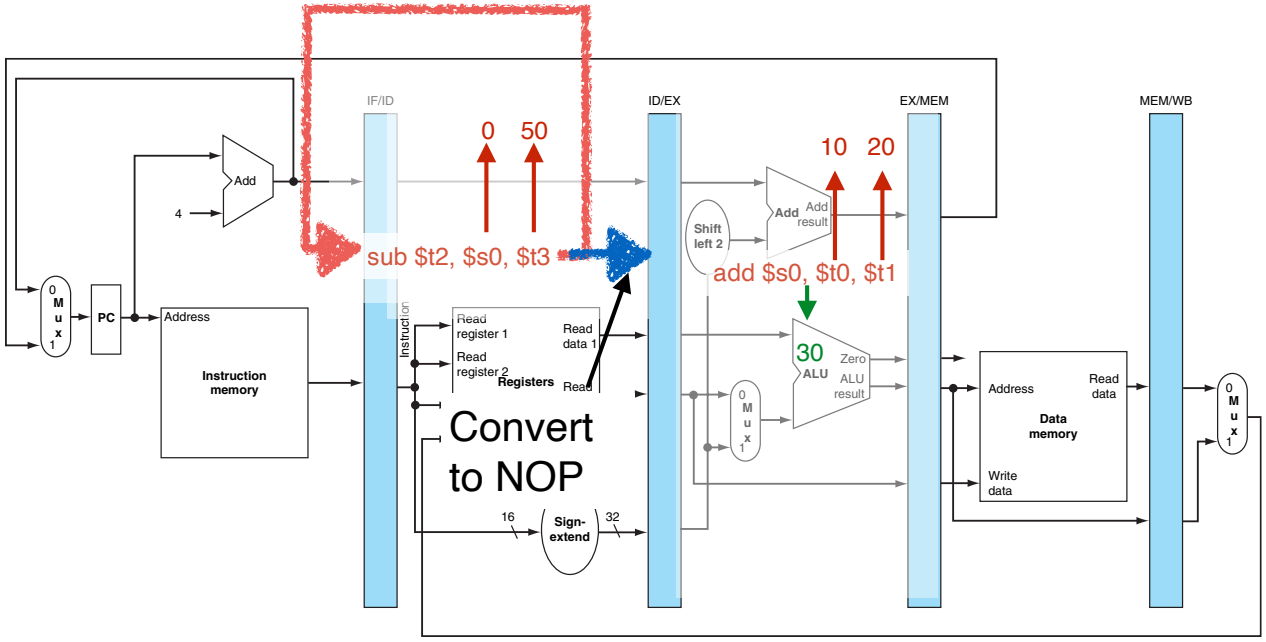


- MEM: **READ** from or **WRITE** to memory
- WB: **WRITE** to register file

- The point:
  - Instruction has no effect on program state prior to MEM and WB stages
  - If instruction were aborted prior to MEM & WB stages, it's as if it never happened

Option A: Arch Inserts “Bubbles”

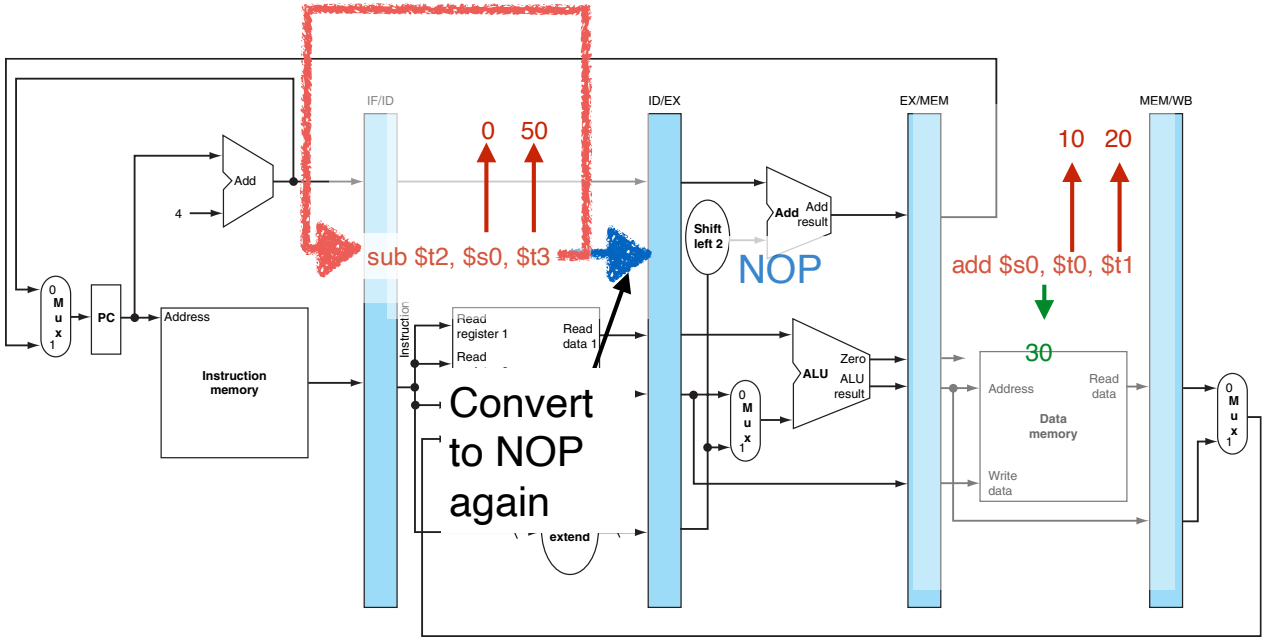
- duplicate the “reading” instruction and rerun through same stage
- convert original copy of “reading” instruction to NOP: don’t let it write to memory or reg file (recall: it then has no effect on state)



Hazard Detector identifies hazard: instructs arch to stall

Option A: Arch Inserts “Bubbles”

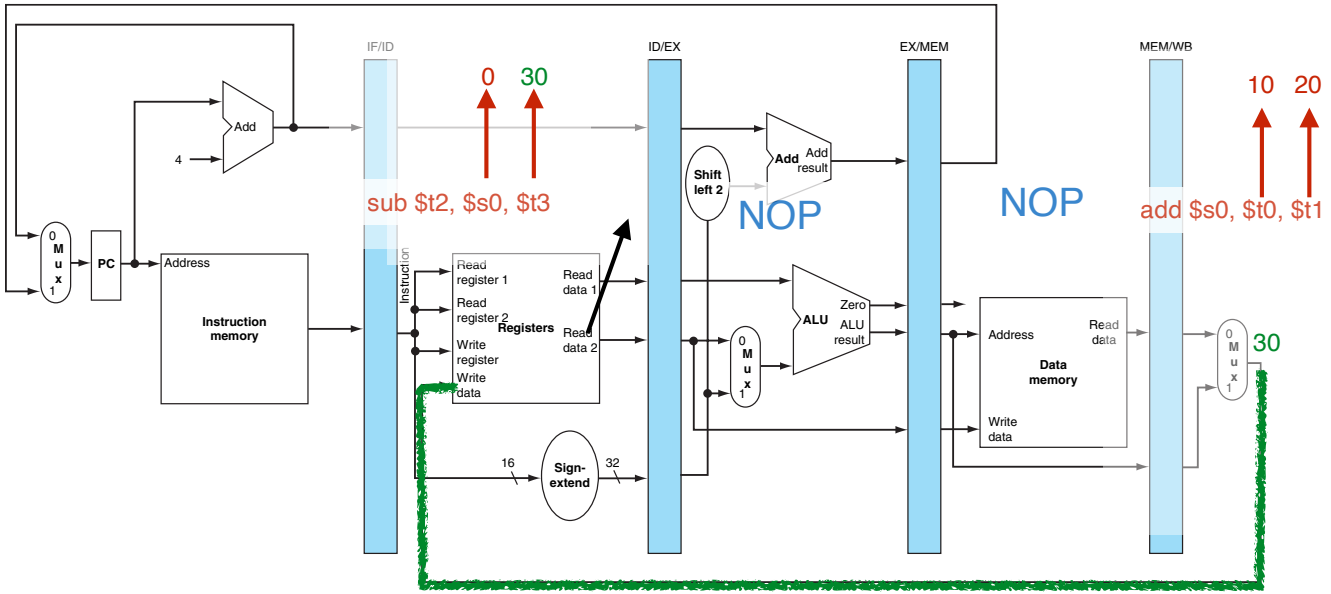
- duplicate the “reading” instruction and rerun through same stage
- convert original copy of “reading” instruction to NOP: don’t let it write to memory or reg file (recall: it then has no effect on state)



Hazard Detector identifies hazard again: instructs arch to stall (again)

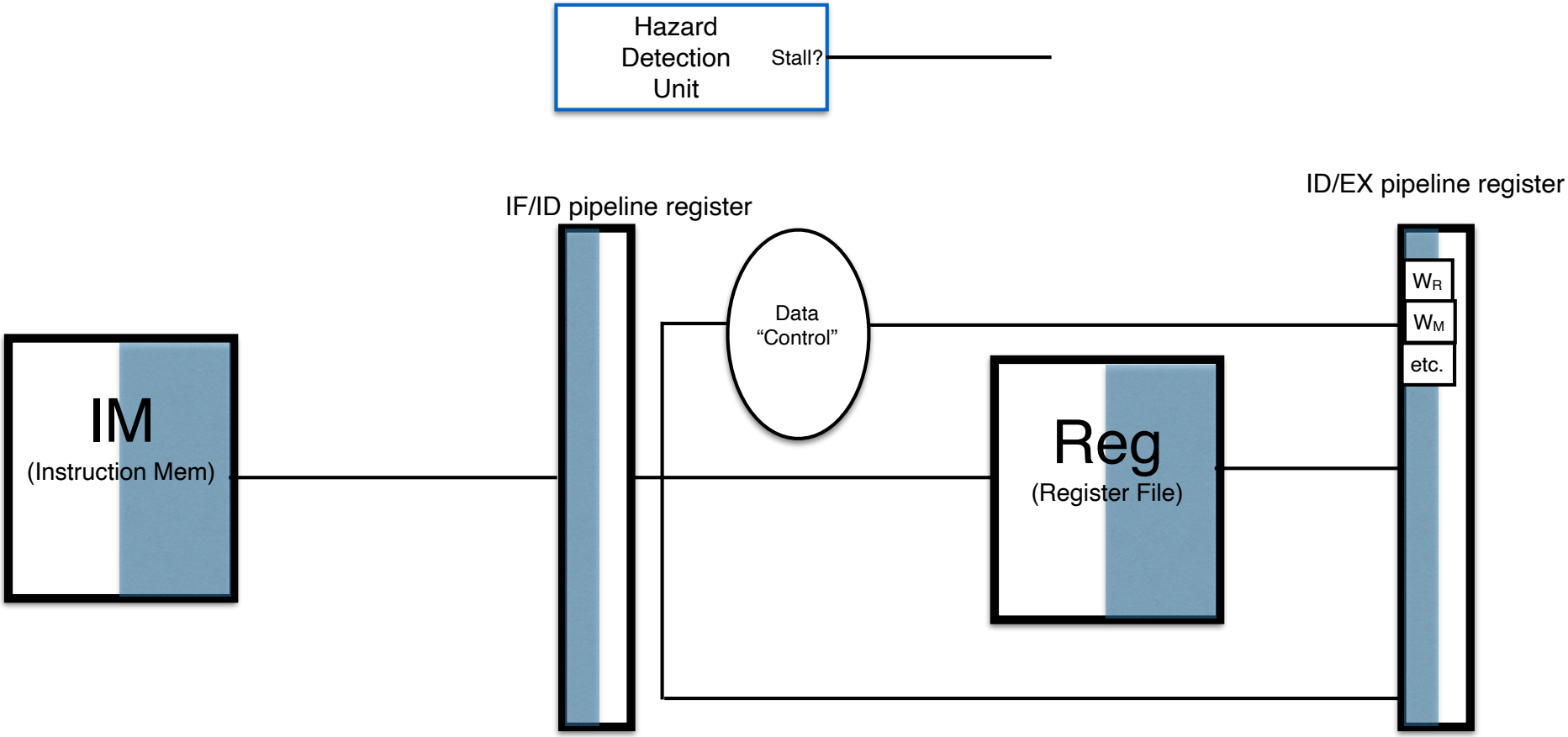
Option A: Arch Inserts “Bubbles”

- duplicate the “reading” instruction and rerun through same stage
- convert original copy of “reading” instruction to NOP: don’t let it write to memory or reg file (recall: it then has no effect on state)



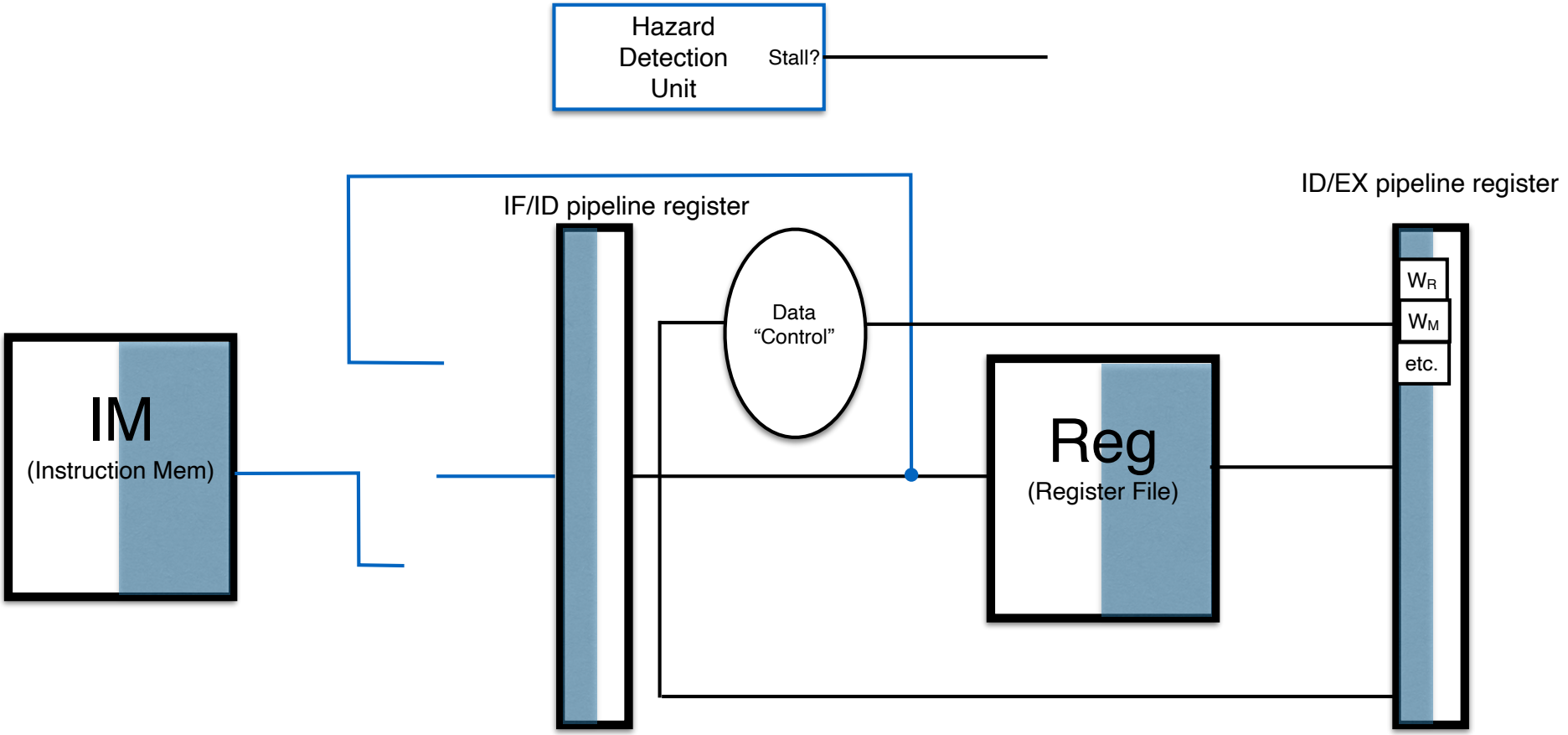
Hazard Detector sees no hazard: allow pipeline to proceed

Implementing the NOP “Bubble”



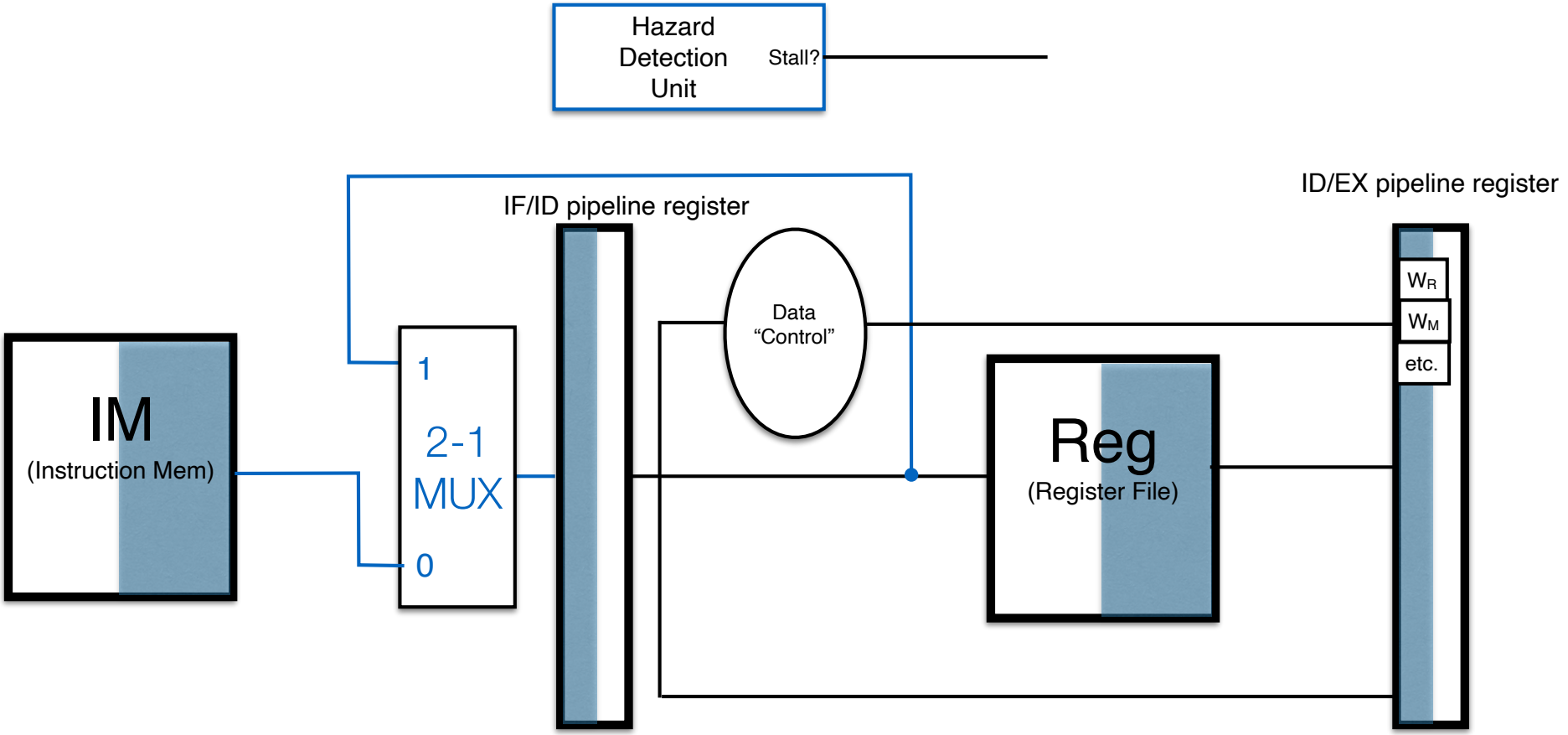
- When Hazard Detection Unit Indicates Stall, then

Implementing the NOP “Bubble”



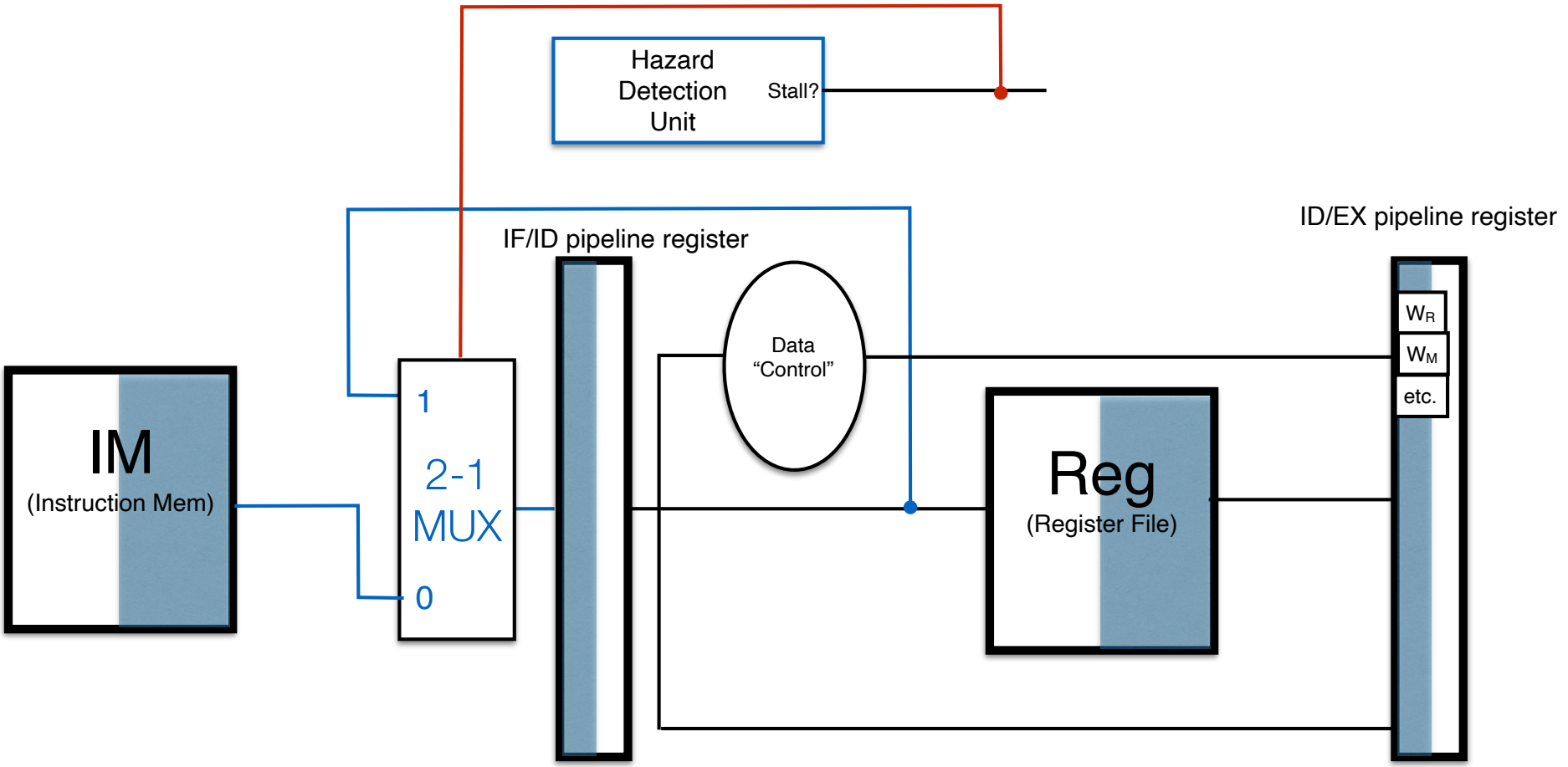
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage

Implementing the NOP “Bubble”



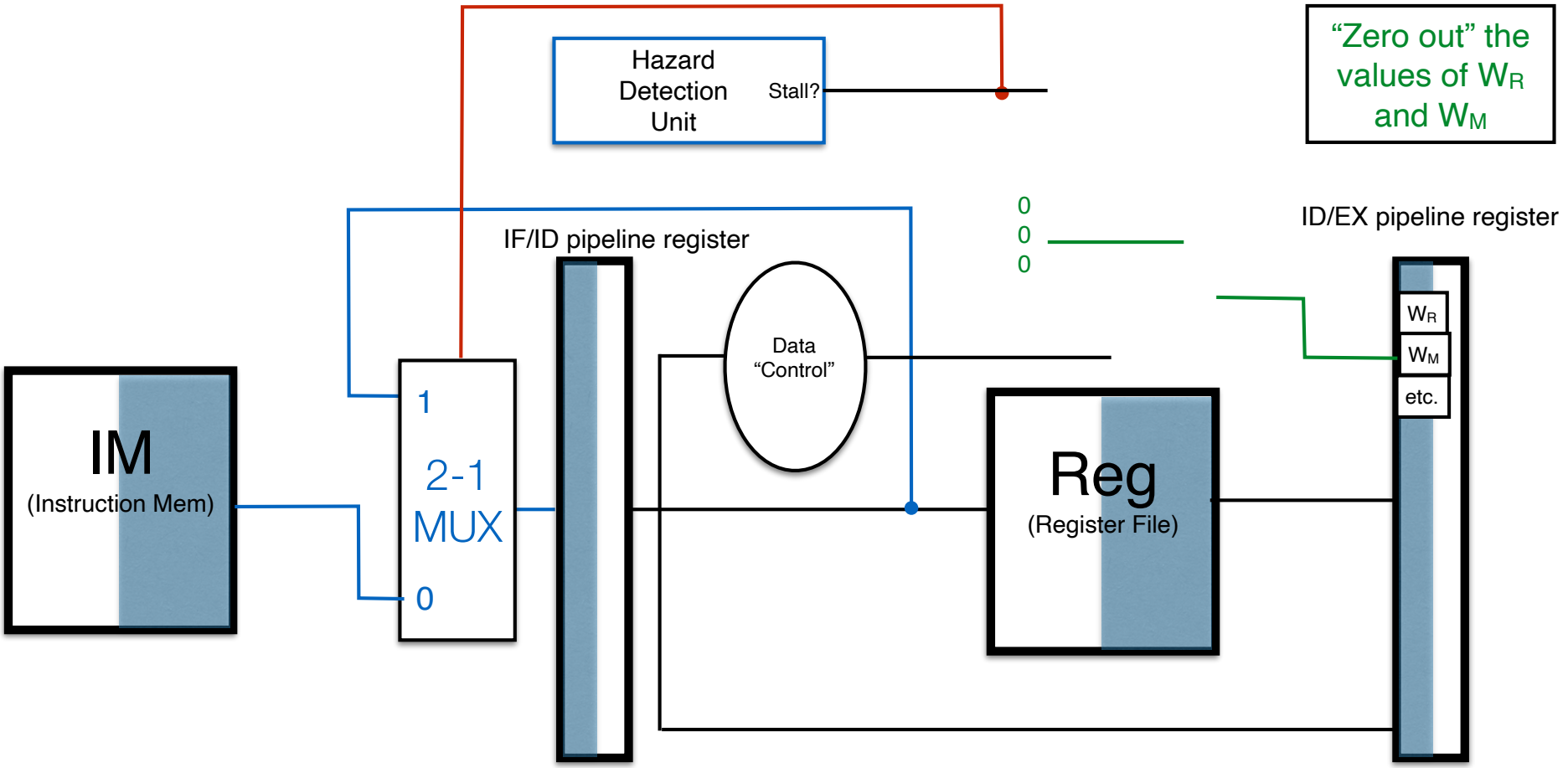
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage

Implementing the NOP “Bubble”



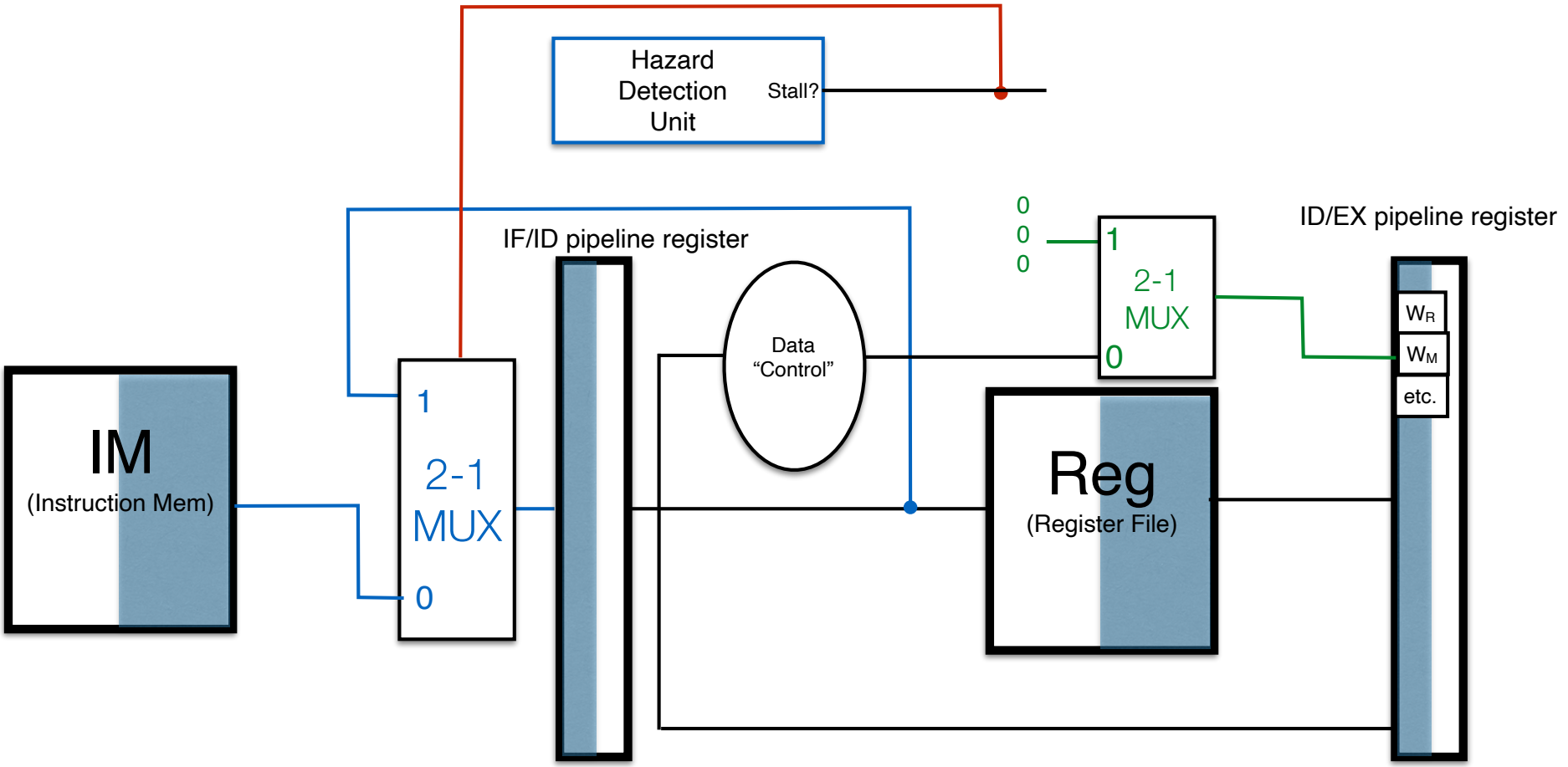
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage

Implementing the NOP “Bubble”



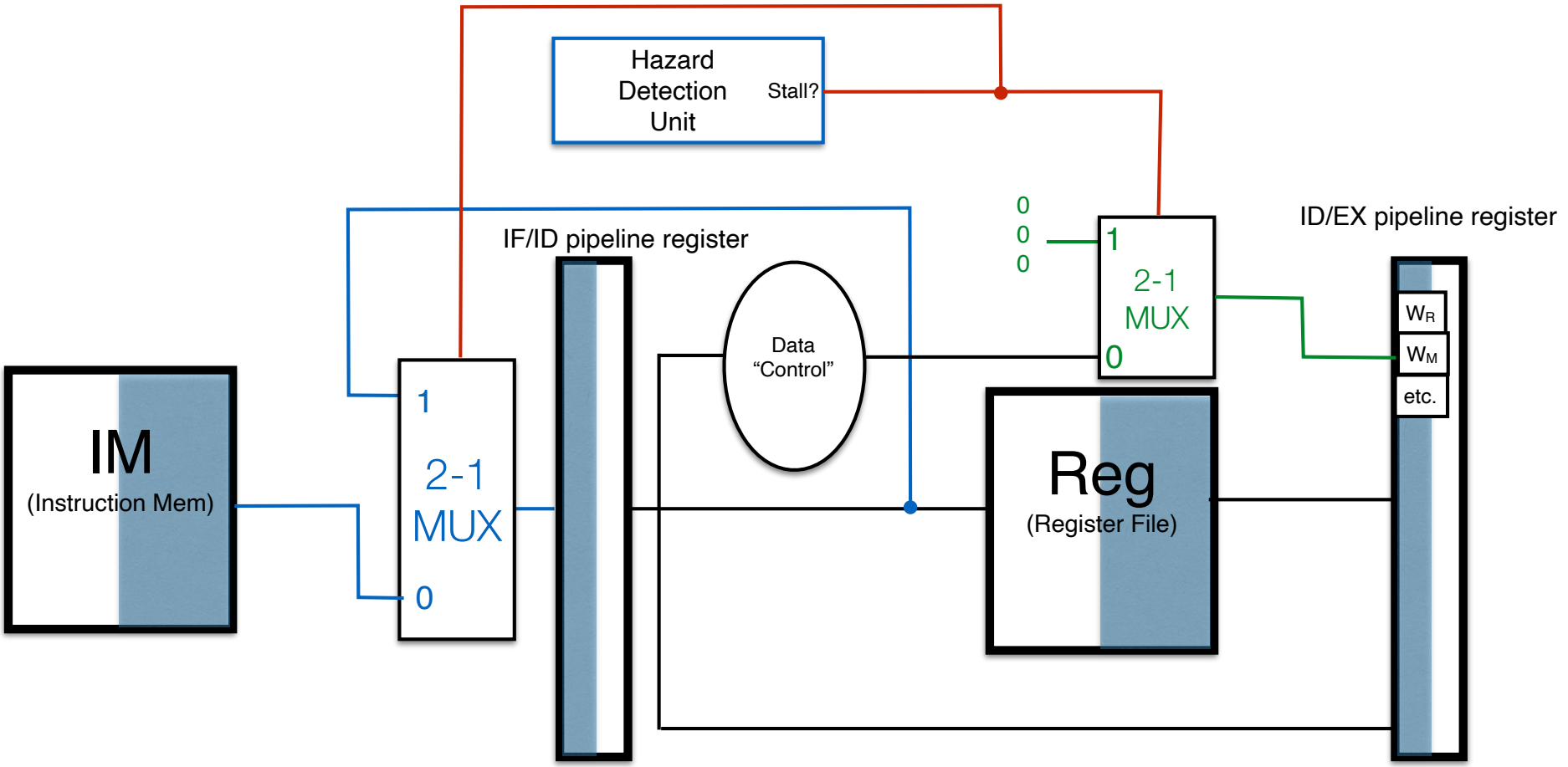
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage
  - prevent current instruction in ID stage from writing to state

Implementing the NOP “Bubble”



- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage
  - prevent current instruction in ID stage from writing to state

Implementing the NOP “Bubble”



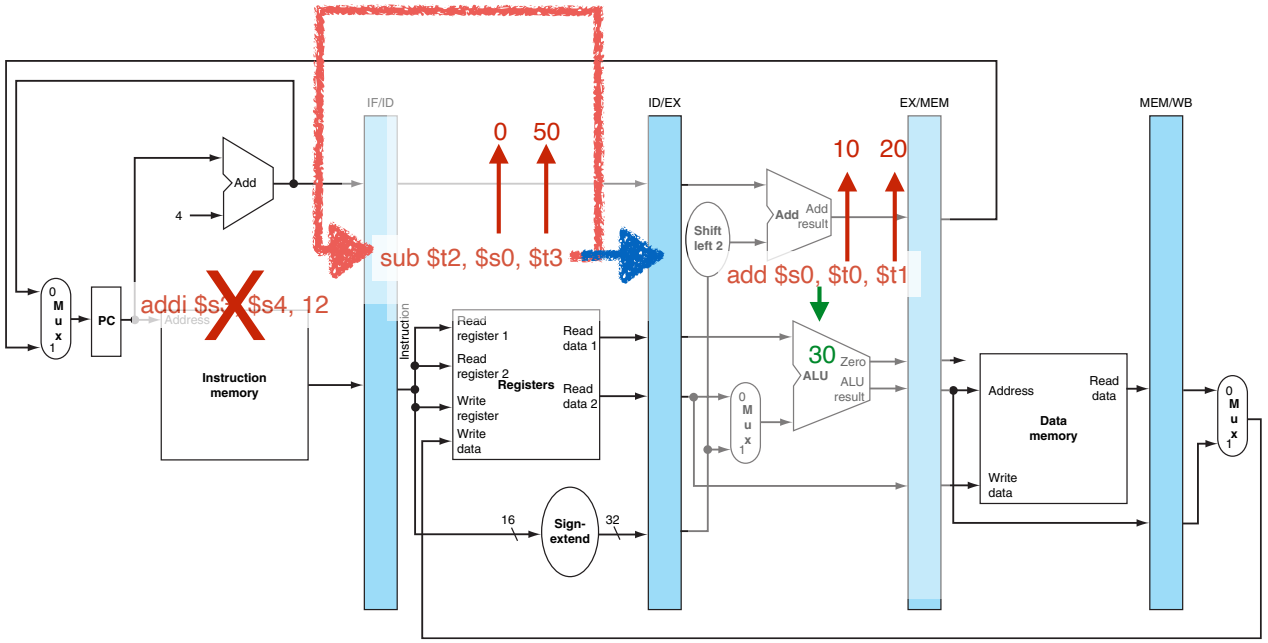
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage
  - prevent current instruction in ID stage from writing to state

What about the PC?

- Still one problem to resolve:

- PC keeps advancing

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
addi $s3, $s4, 12
ori   $t4, $t5, 19
```

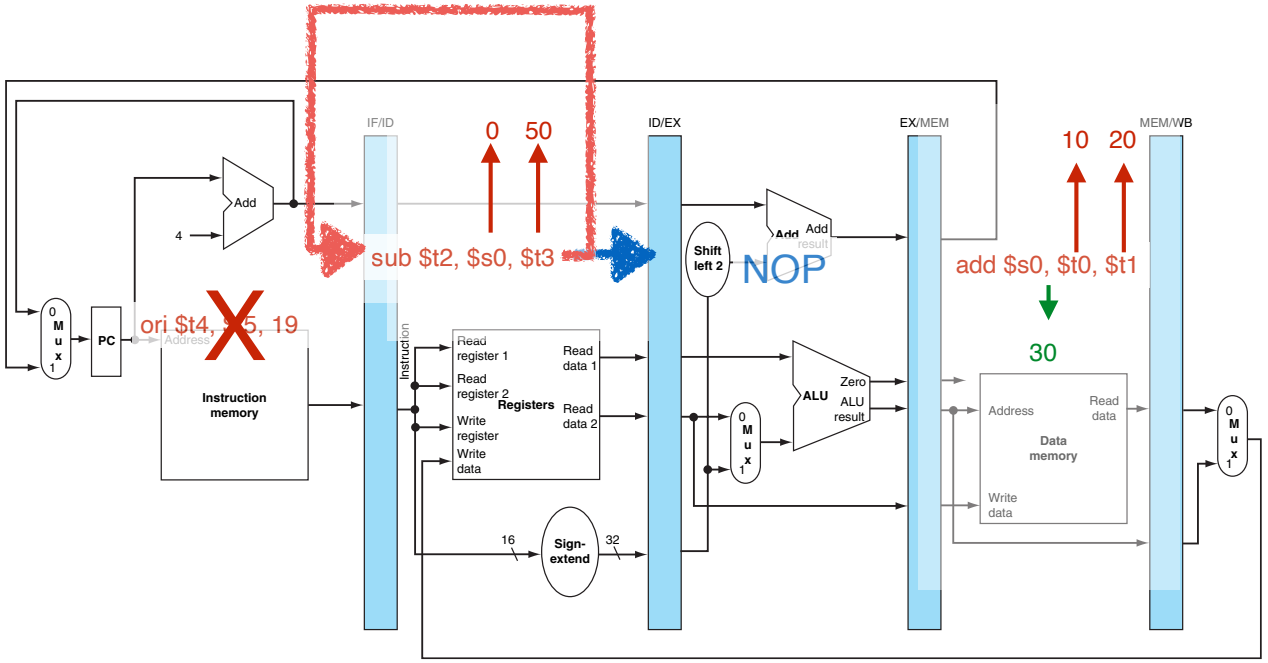


What about the PC?

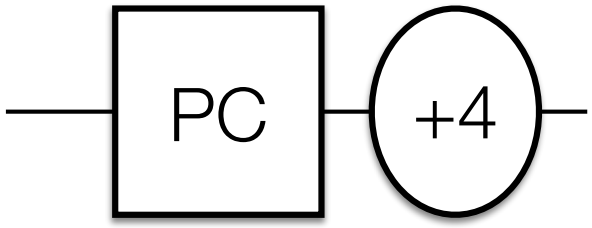
- Still one problem to resolve:

- PC keeps advancing

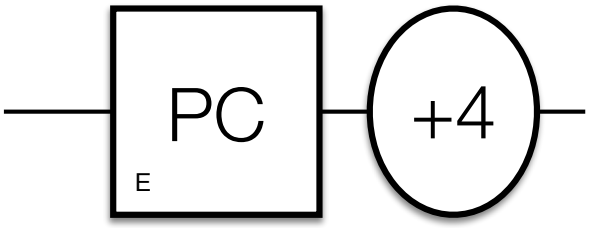
```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
addi $s3, $s4, 12
ori   $t4, $t5, 19
```



Make PC a register with Enable

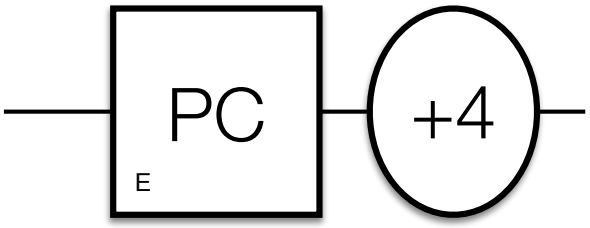


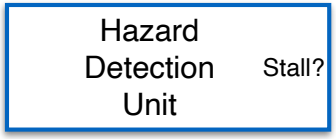
Make PC a register with Enable



- When Enabled (E=1), PC value can be changed, otherwise stays same

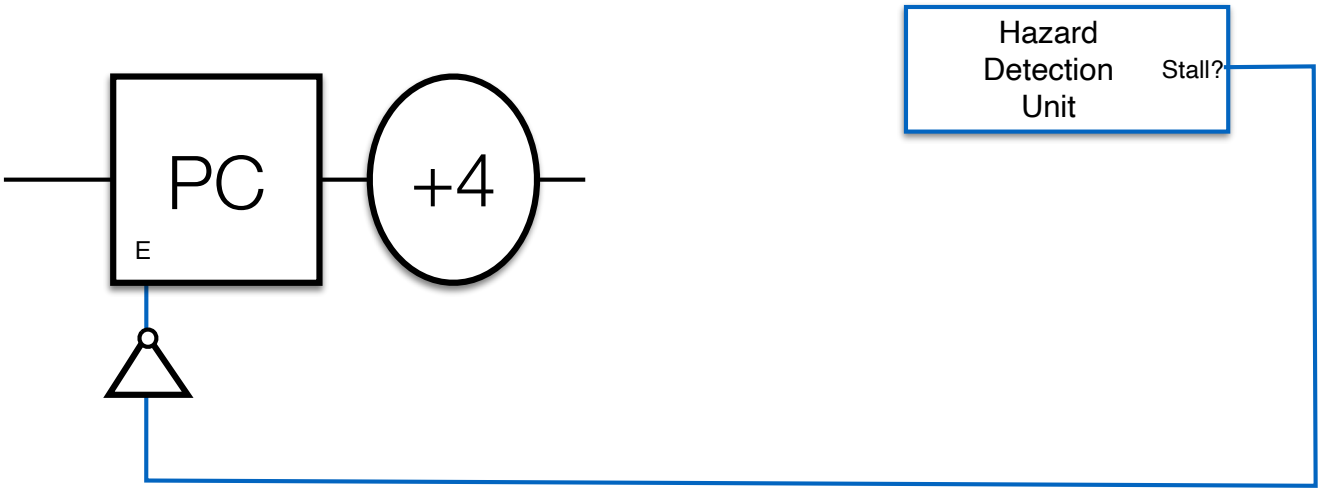
Make PC a register with Enable





- When Enabled (E=1), PC value can be changed, otherwise stays same
- When HDU Stall output is 0, PC is enabled, does +=4 (or branch or jump)
- When HDU Stall output is 1, PC is not enabled, holds current value

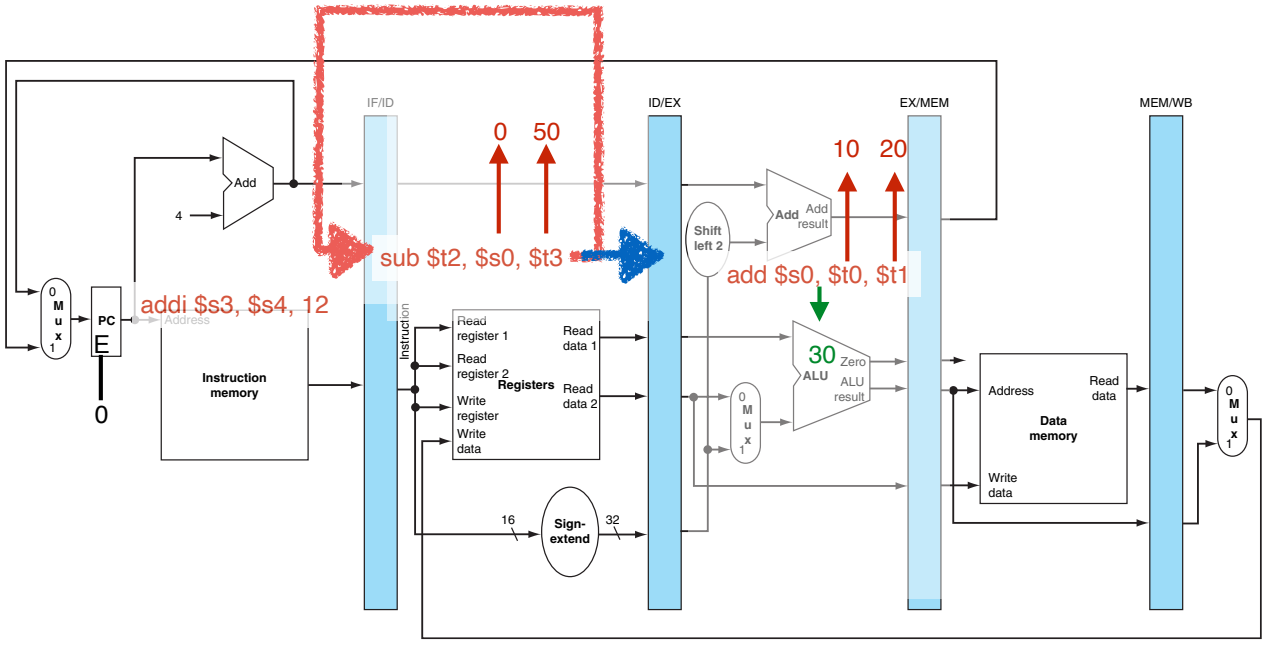
Make PC a register with Enable



- When Enabled (E=1), PC value can be changed, otherwise stays same
- When HDU Stall output is 0, PC is enabled, does +=4 (or branch or jump)
- When HDU Stall output is 1, PC is not enabled, holds current value

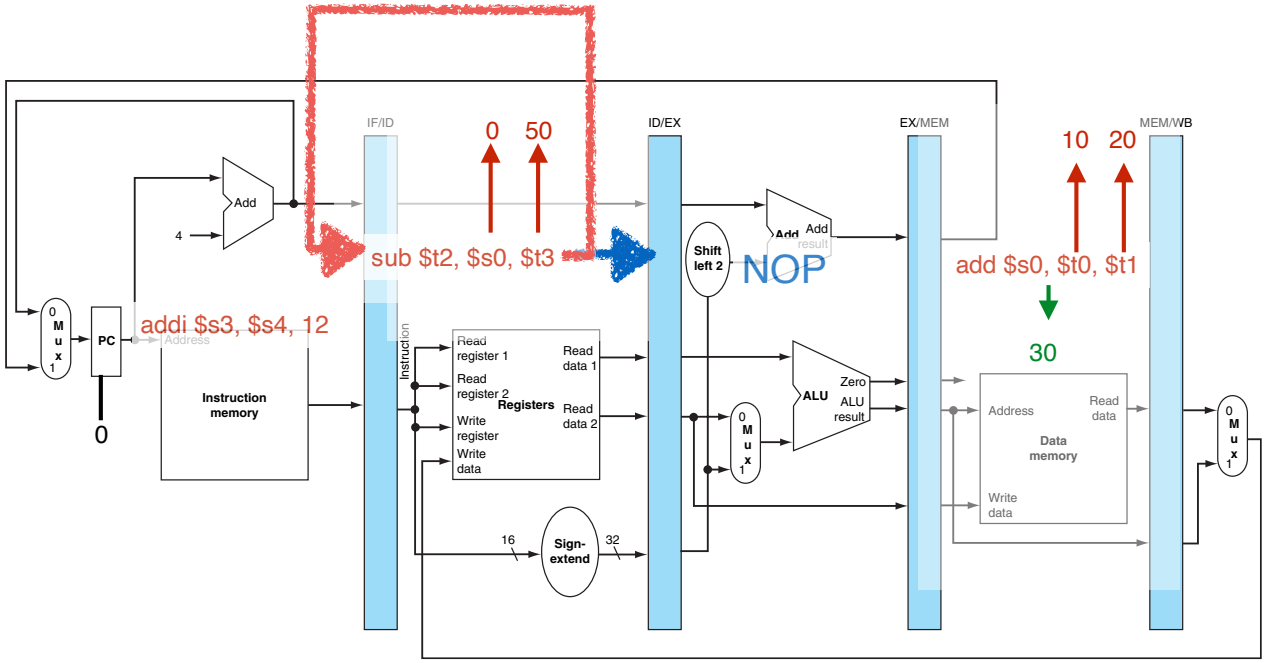
Stalling PC holds the instruction in place

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
addi $s3, $s4, 12
ori   $t4, $t5, 19
```

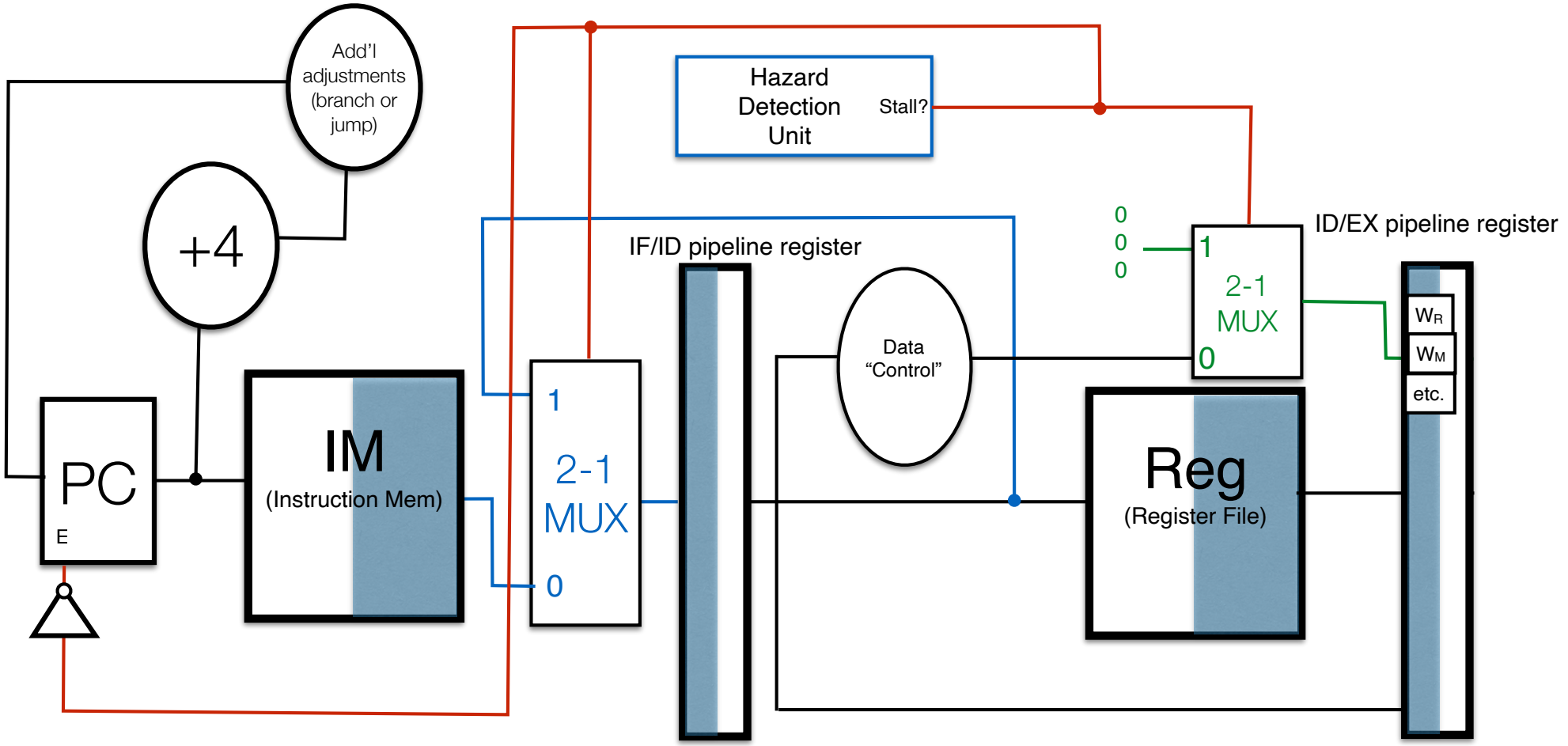


Stalling PC holds the instruction in place

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
addi $s3, $s4, 12
ori   $t4, $t5, 19
```



Stalling Arch (inputs to HDU not shown)



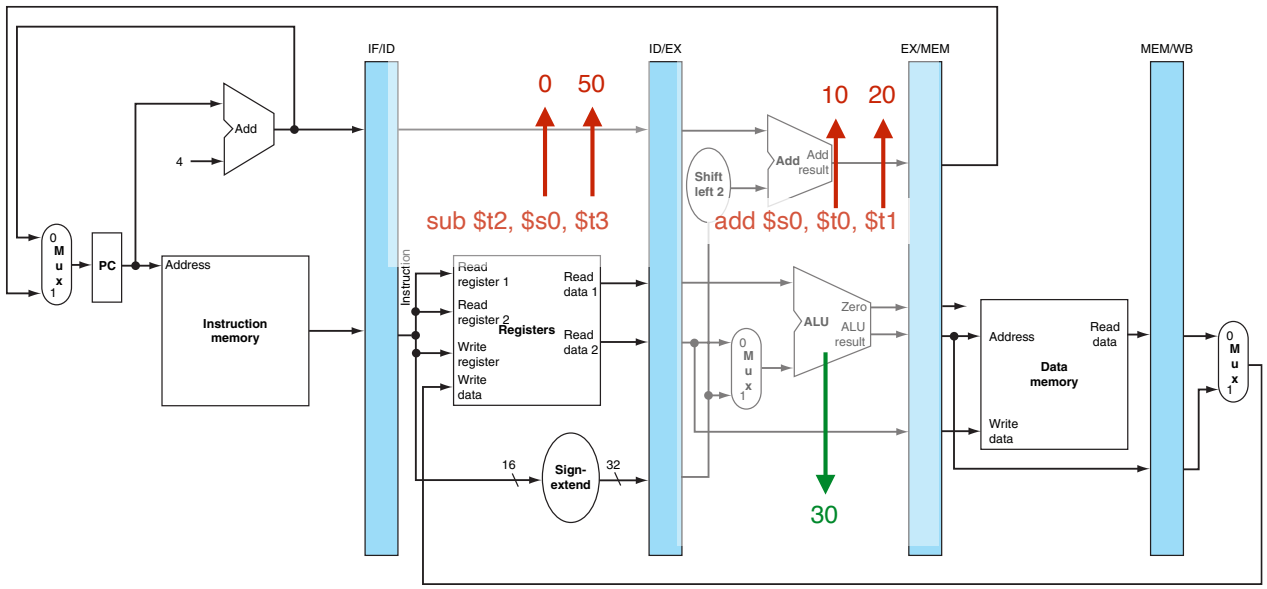
- When Hazard Detection Unit Indicates Stall, then
  - repeat instruction in IF stage
  - prevent current instruction in ID stage from writing to state

Data Forwarding

Data Forwarding

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0 10	$t1 20	$t3 50
$s0 0	$t2 100	



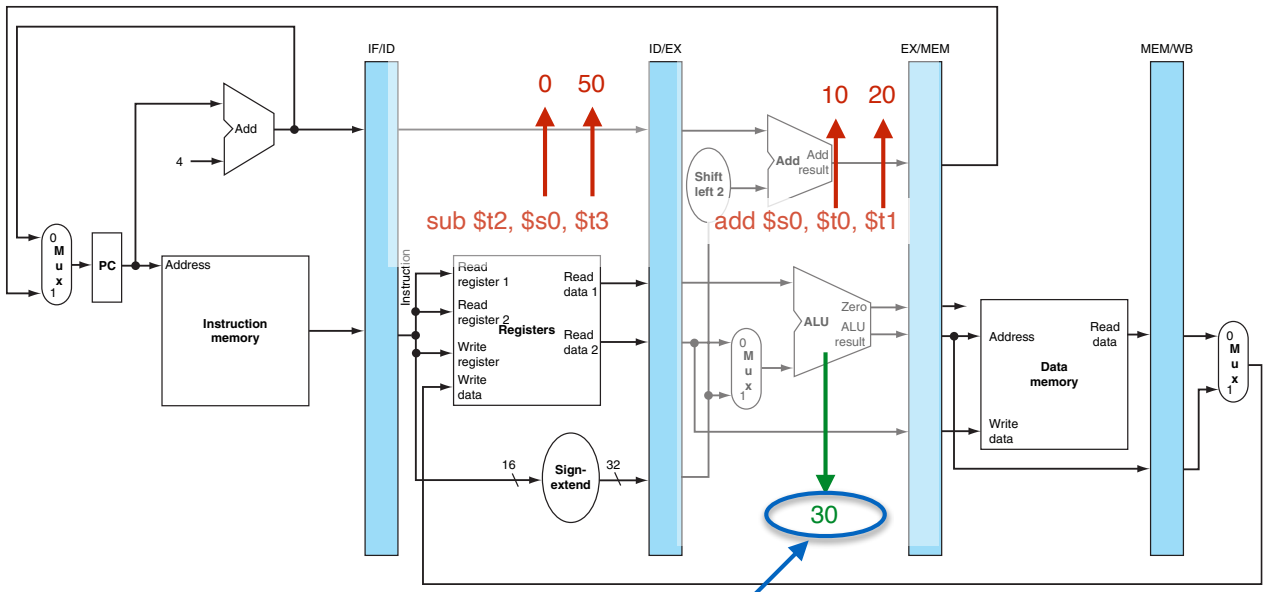
- Stalling works, but undesirable (each “bubble” potentially wastes a clock cycle)
- Observation: Sometimes the needed data is not in (memory or register) state, but already exists in the pipeline (e.g., in pipeline register)

Data Forwarding

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



This is the value not yet written into $s0 that we want to "forward" to the sub instruction

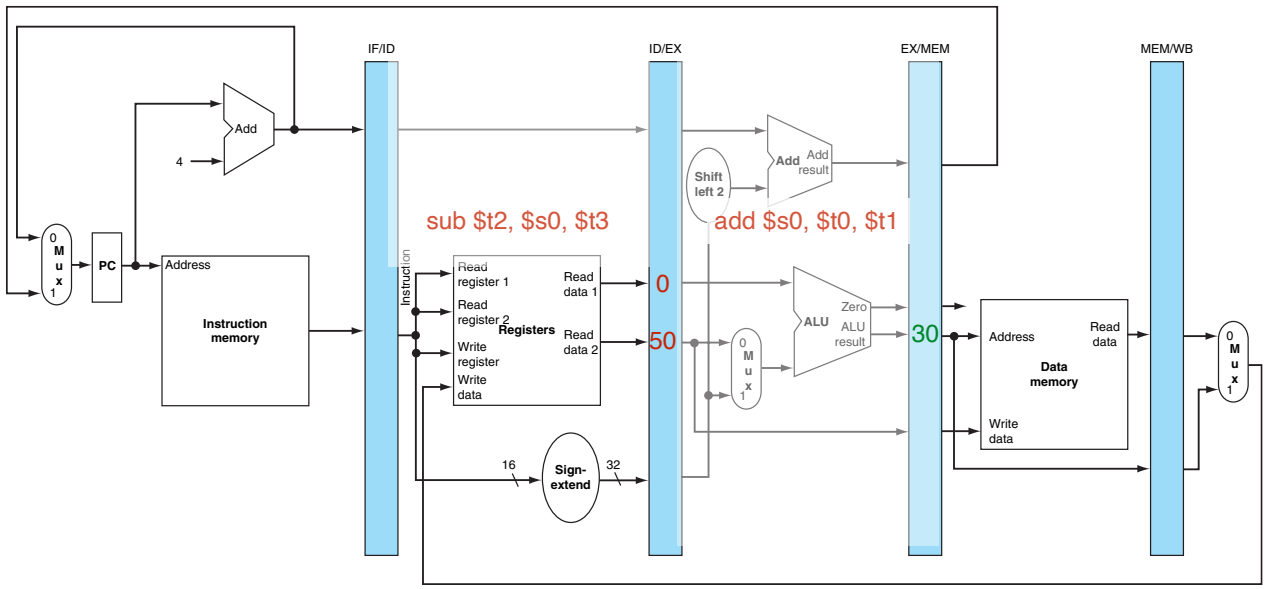
- Stalling works, but undesirable (each "bubble" potentially wastes a clock cycle)
- Observation: Sometimes the needed data is not in (memory or register) state, but already exists in the pipeline (e.g., in pipeline register)

Data Forwarding

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



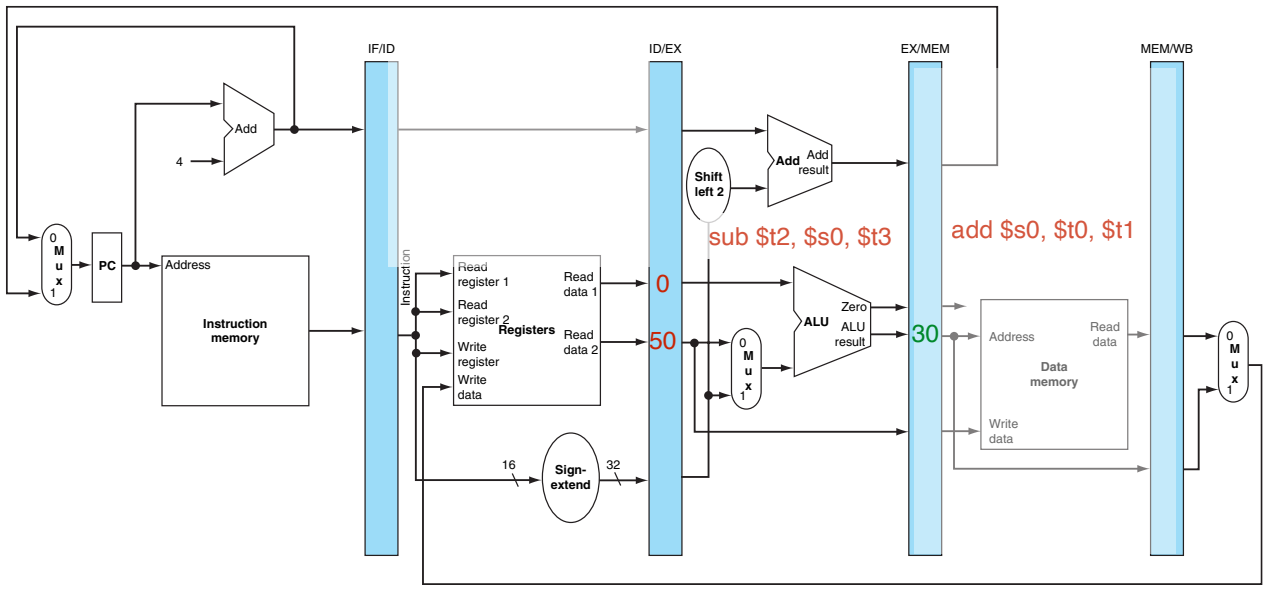
- The values actually get written into the pipeline registers to be forwarded to the next stage

Data Forwarding

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



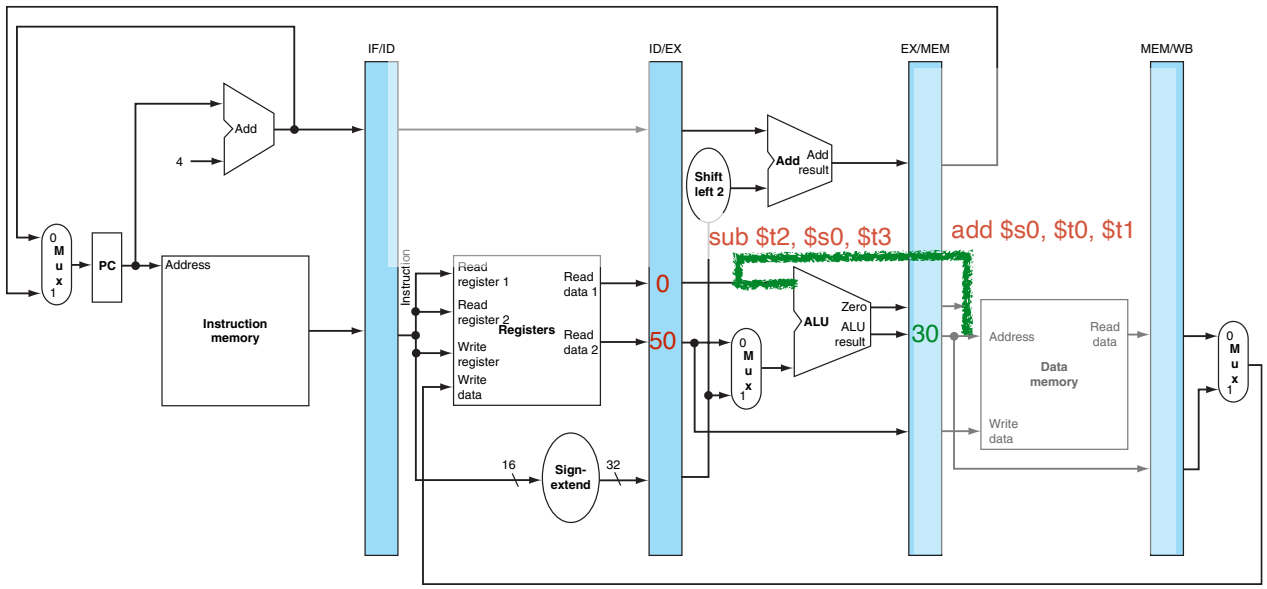
- Next clock cycle: sub is ready to send values into the ALU to be computed upon

Data Forwarding

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

$t0	$t1	$t3
10	20	50

$s0	$t2
0	100



- Next clock cycle: sub is ready to send values into the ALU to be computed upon
- Need to “forward” the value in the pipeline back to the ALU

Forwarding: Seeing it "High Level"

- Another way to view forwarding visually:

- Plot instructions on timeline. If info to be forwarded can move down and forward in time, then can be forwarded

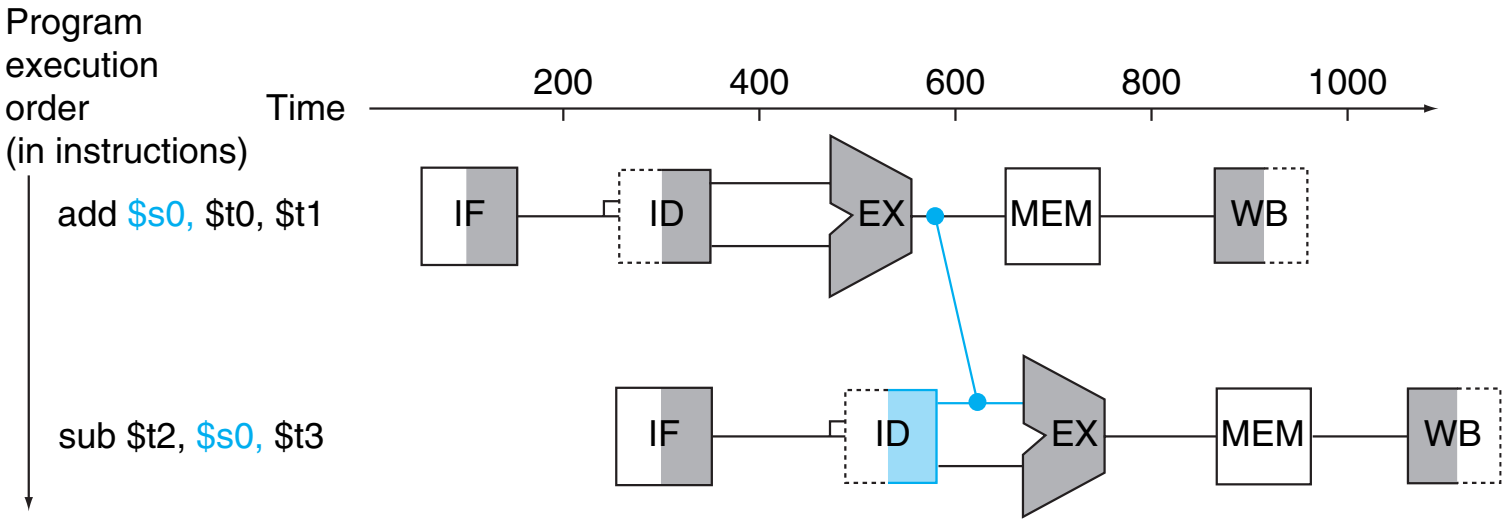
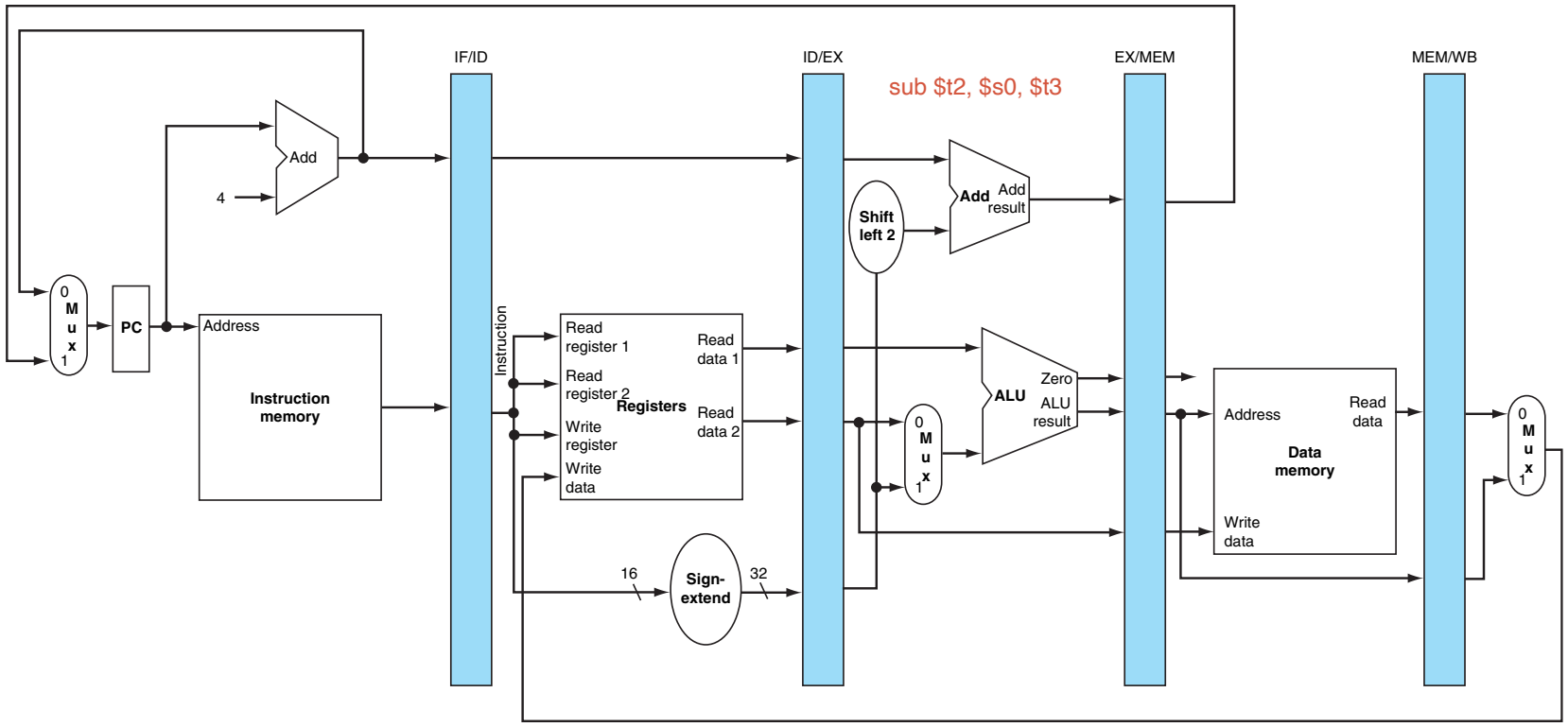


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

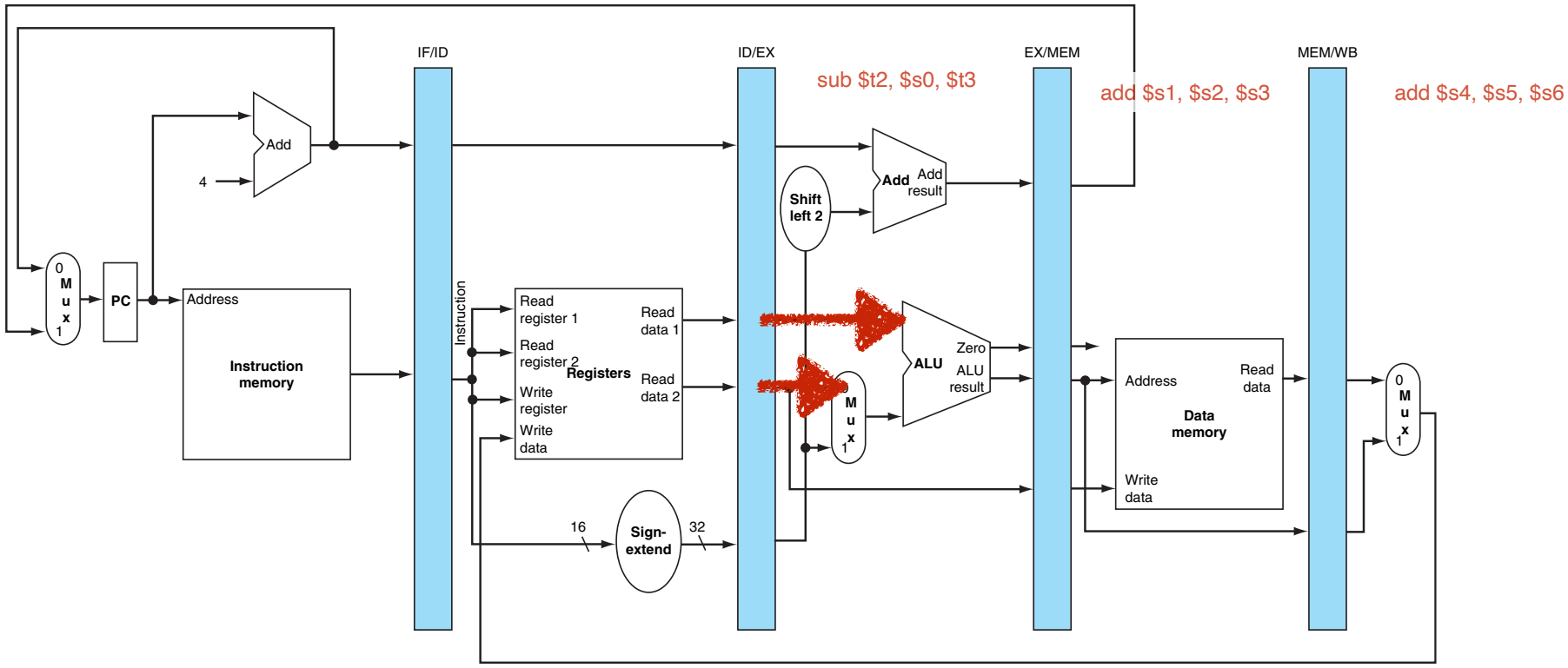
General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



General Forwarding Idea

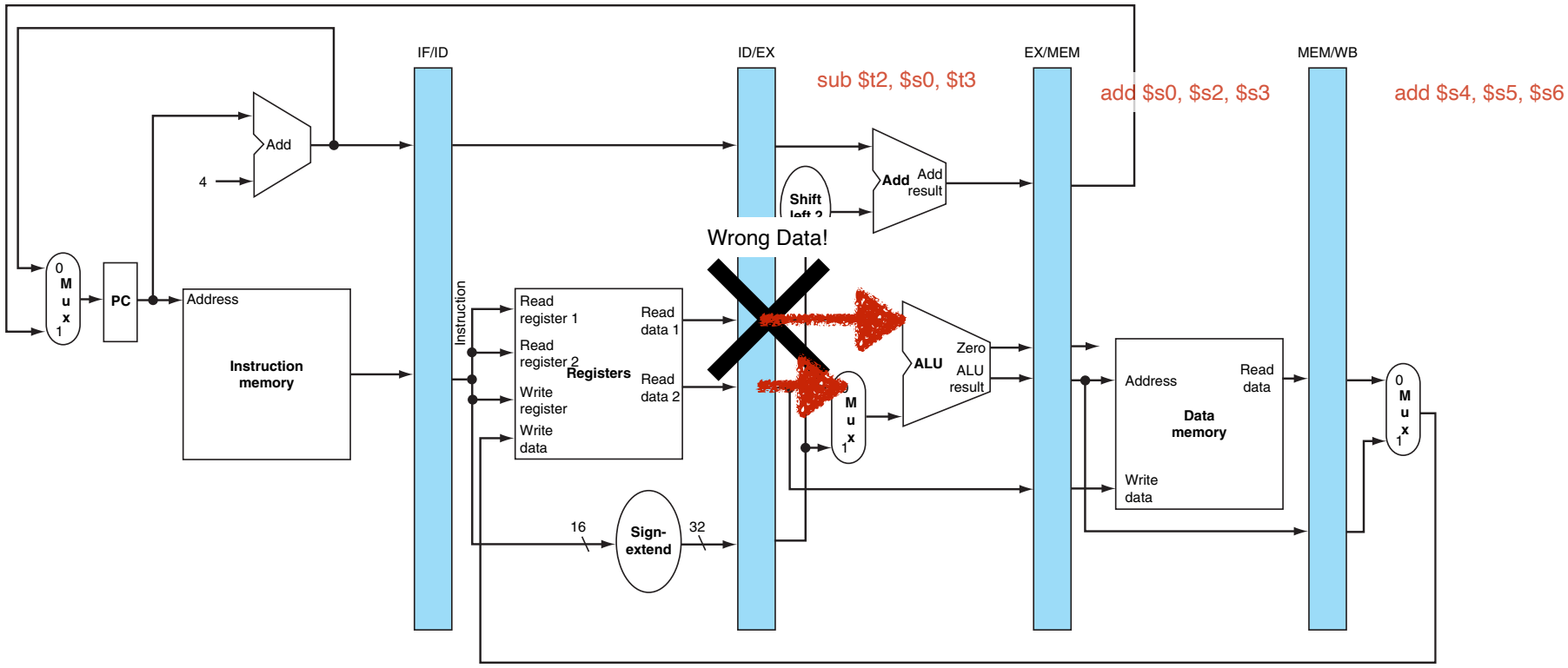
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



"Normal" forwarding: the non-hazard path through the pipeline

General Forwarding Idea

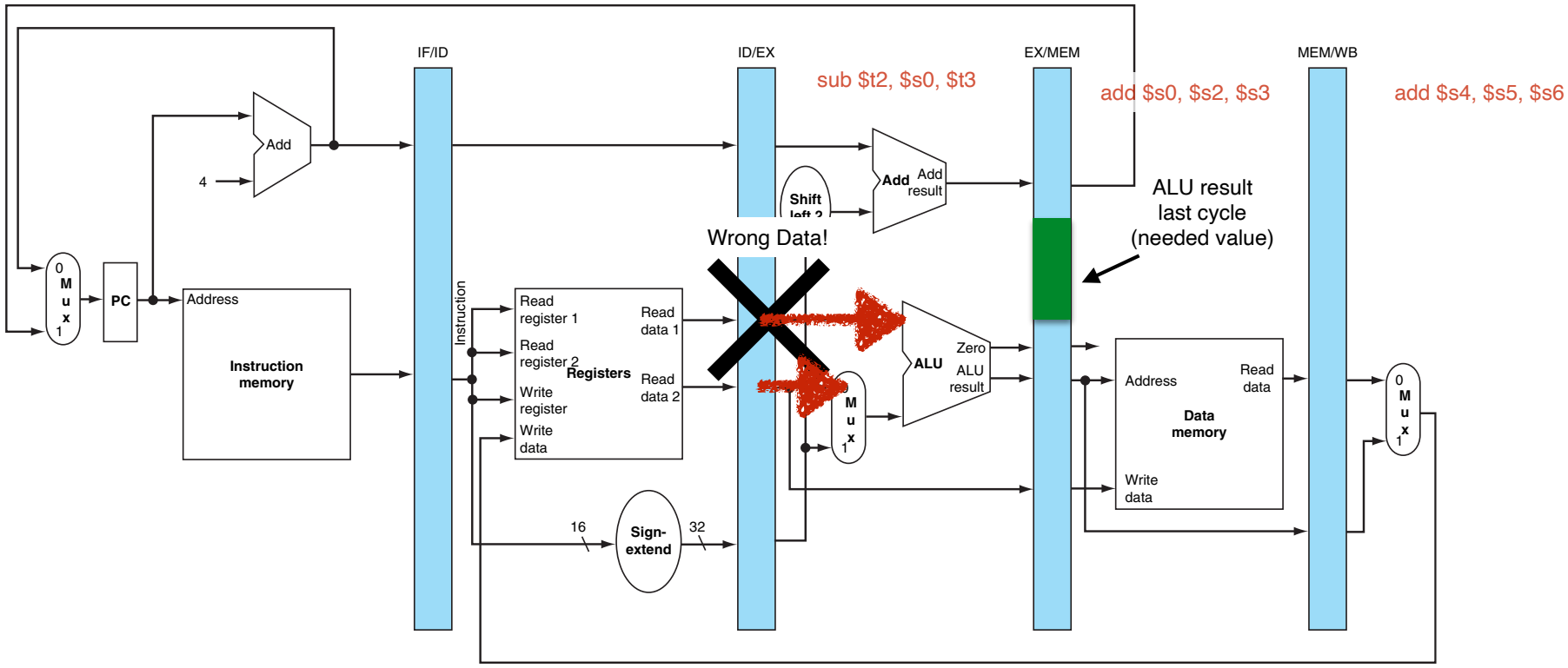
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



But what if instruction currently in MEM stage altered $s0?

General Forwarding Idea

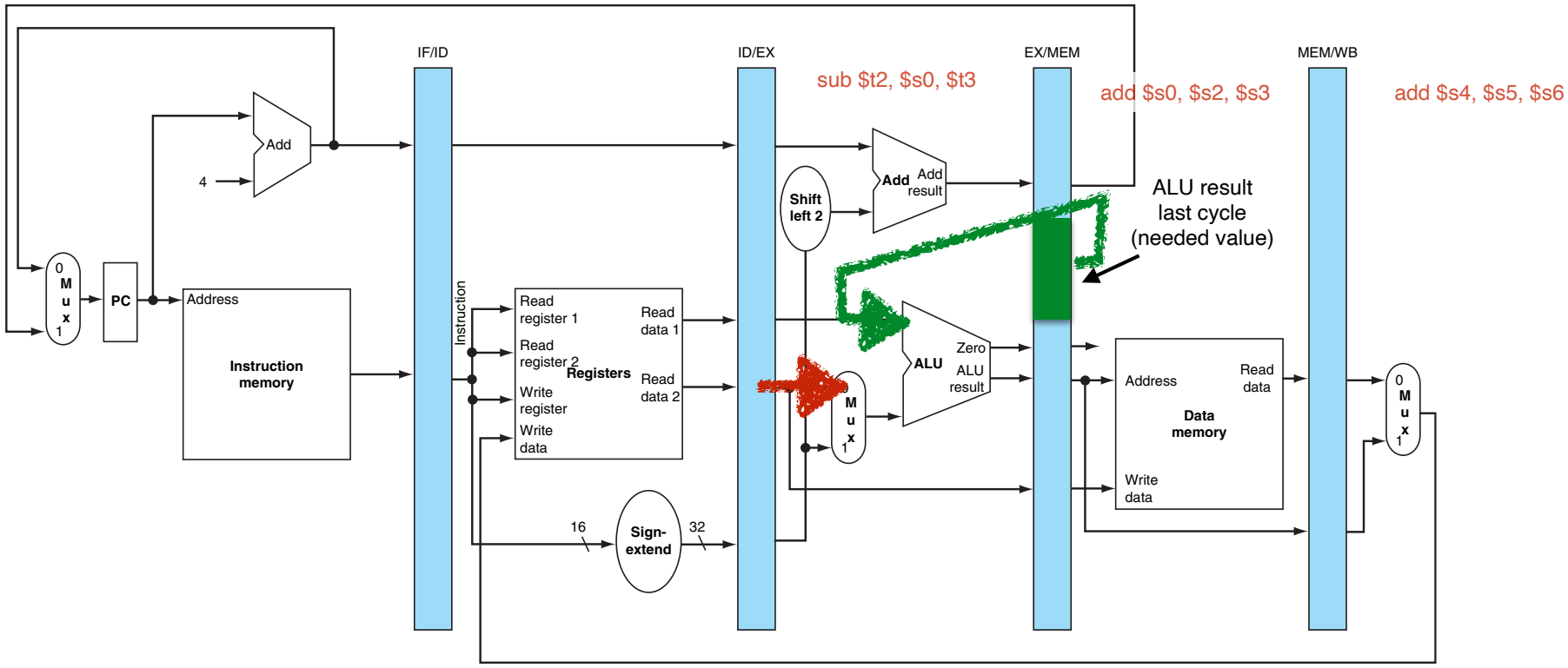
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



But what if instruction currently in MEM stage altered $s0?

General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?

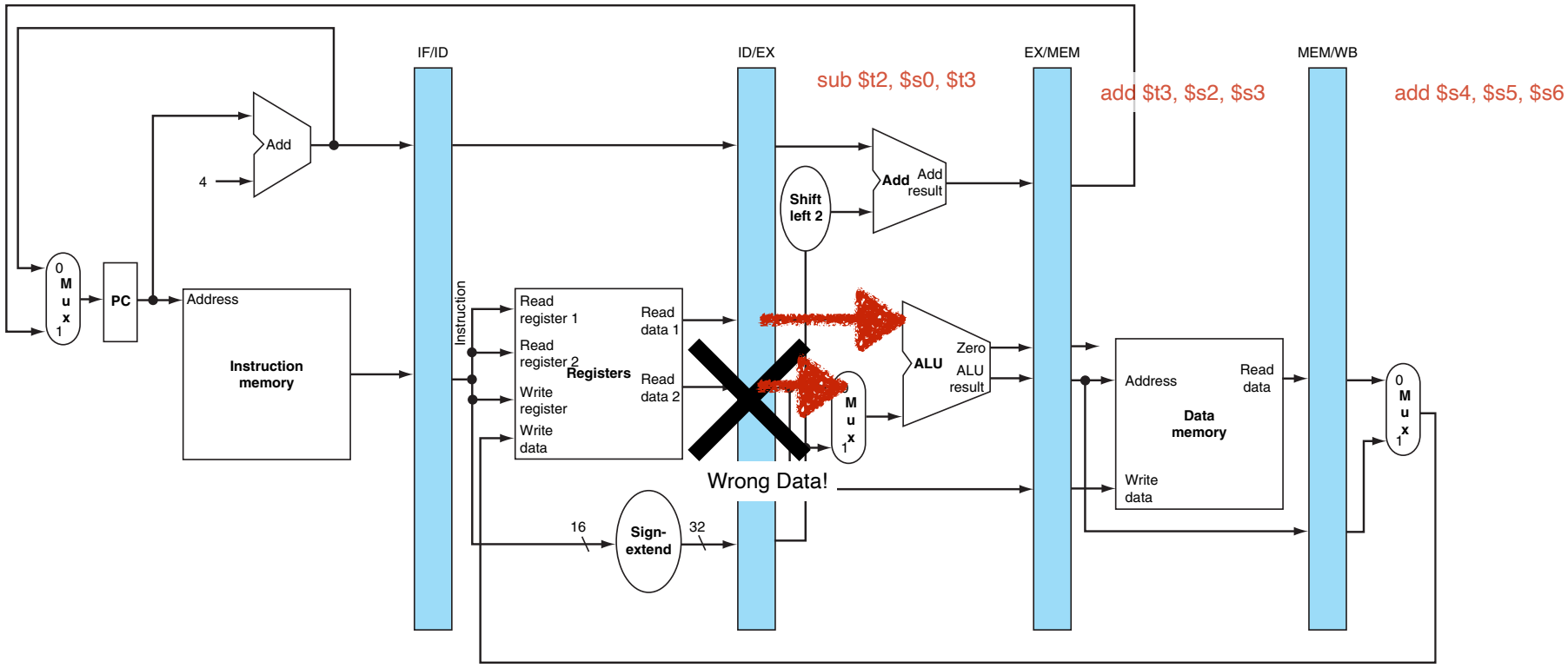


But what if instruction currently in MEM stage altered $s0?

Solution: route correct result inside pipeline reg.

General Forwarding Idea

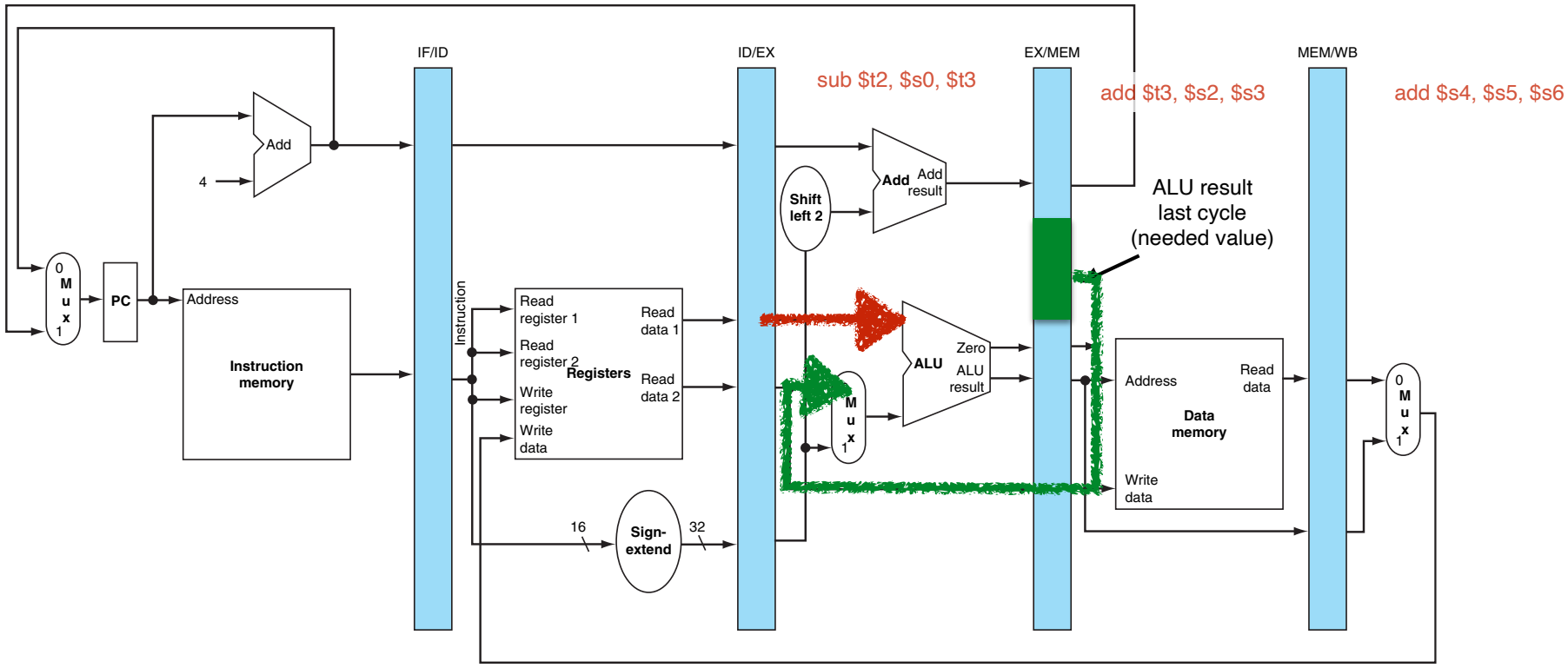
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



But what if instruction currently in MEM stage altered $t3?

General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?

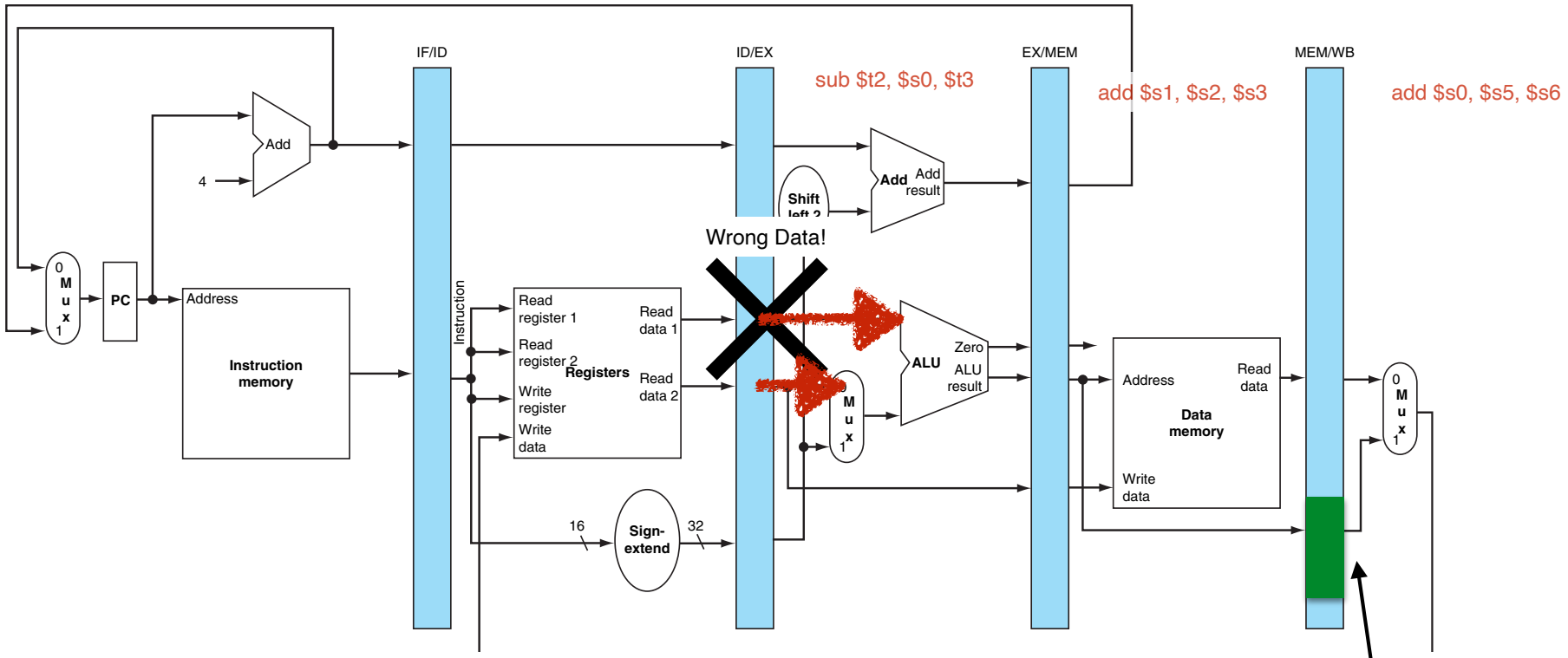


But what if instruction currently in MEM stage altered $t3?

Solution: similar solution

General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?

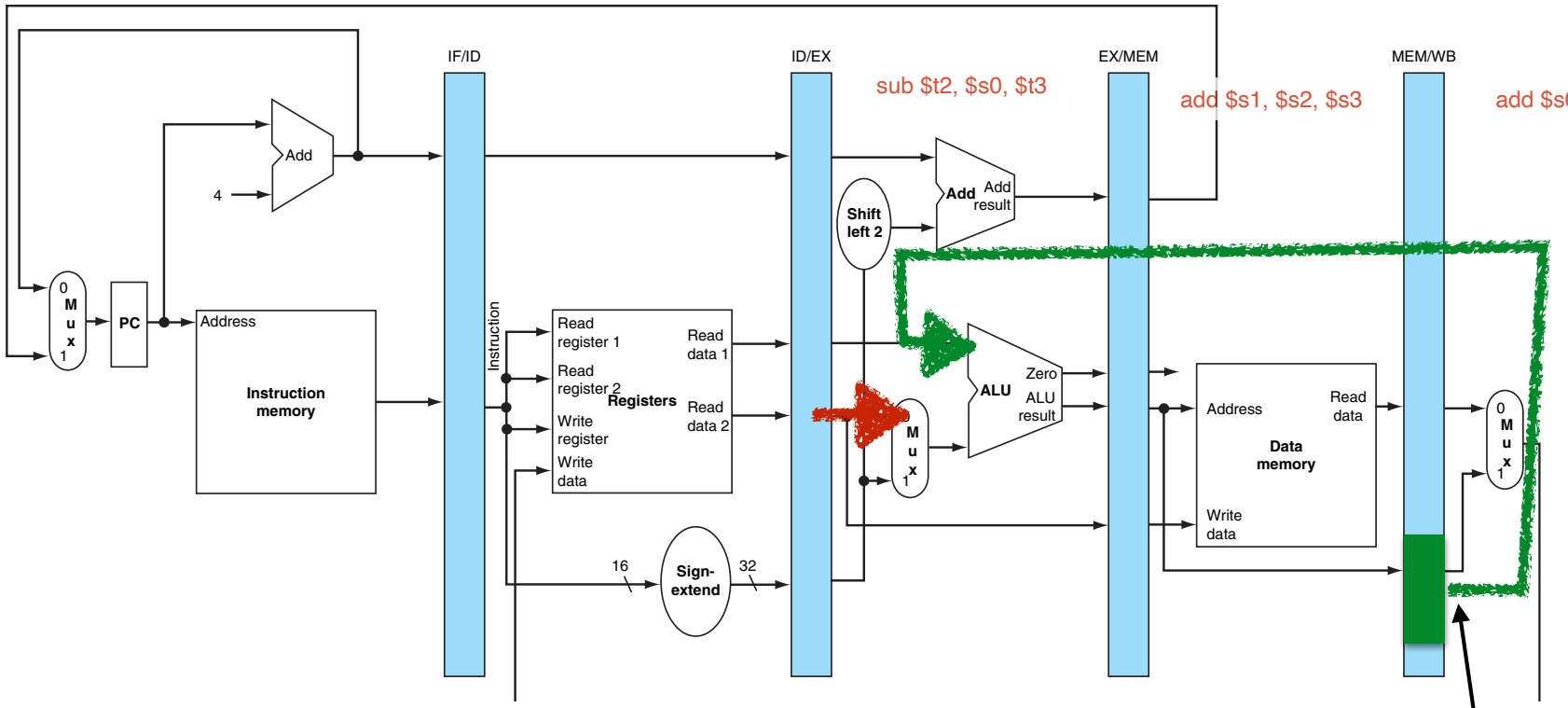


But what if instruction currently in WB stage altered $s0?
Write before read in reg file, but ALU already pulling "stale" value from pipeline reg

ALU result
2 cycles ago
(needed value)

General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?

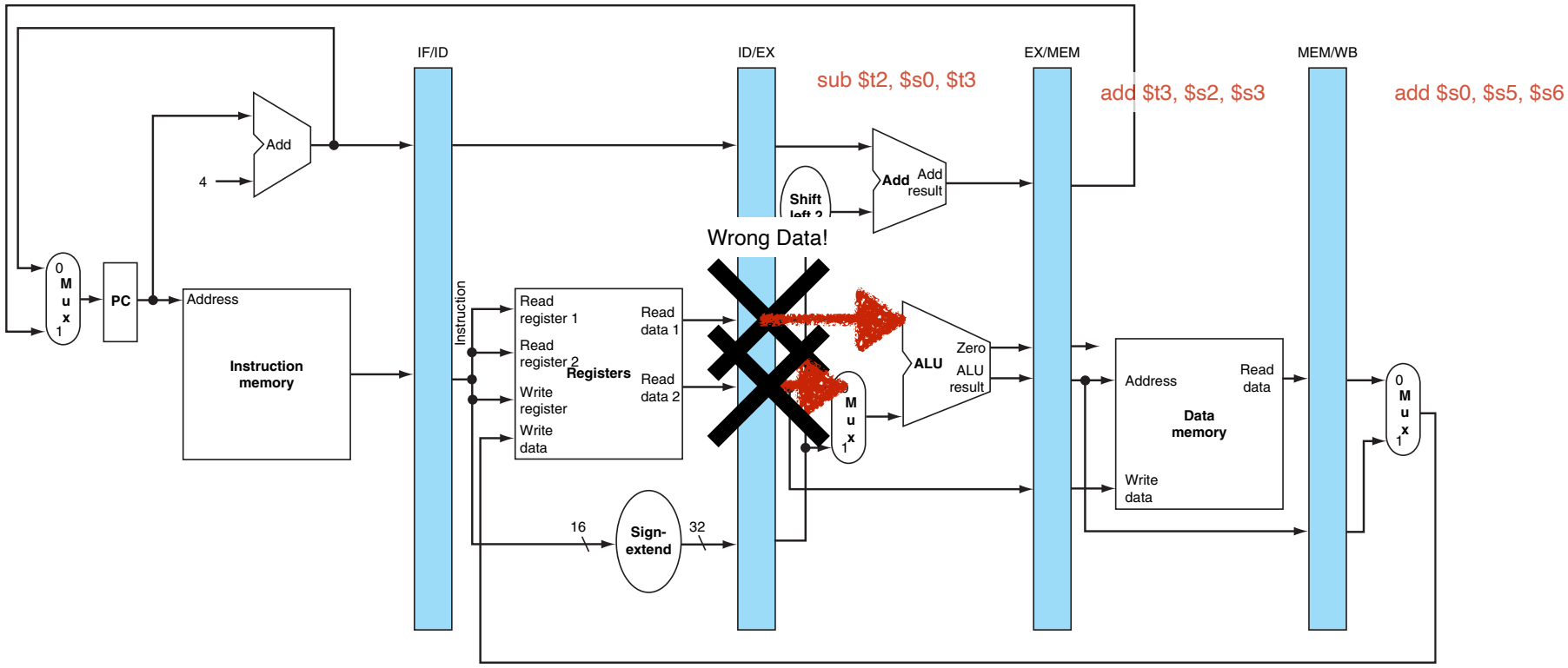


But what if instruction currently in WB stage altered $s0?
Write before read in reg file, but ALU already pulling "stale" value from pipeline reg

ALU result
2 cycles ago
(needed value)

General Forwarding Idea

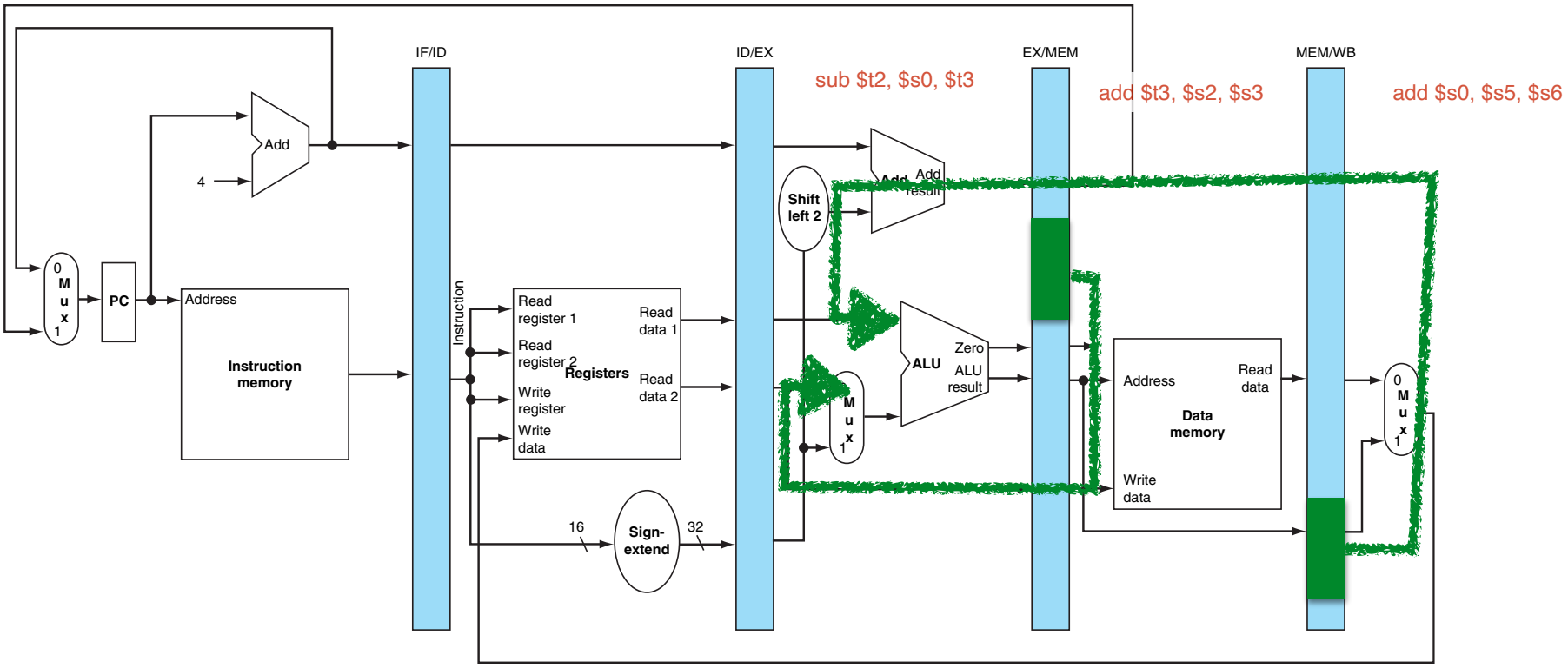
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



YIKES!!!!

General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?

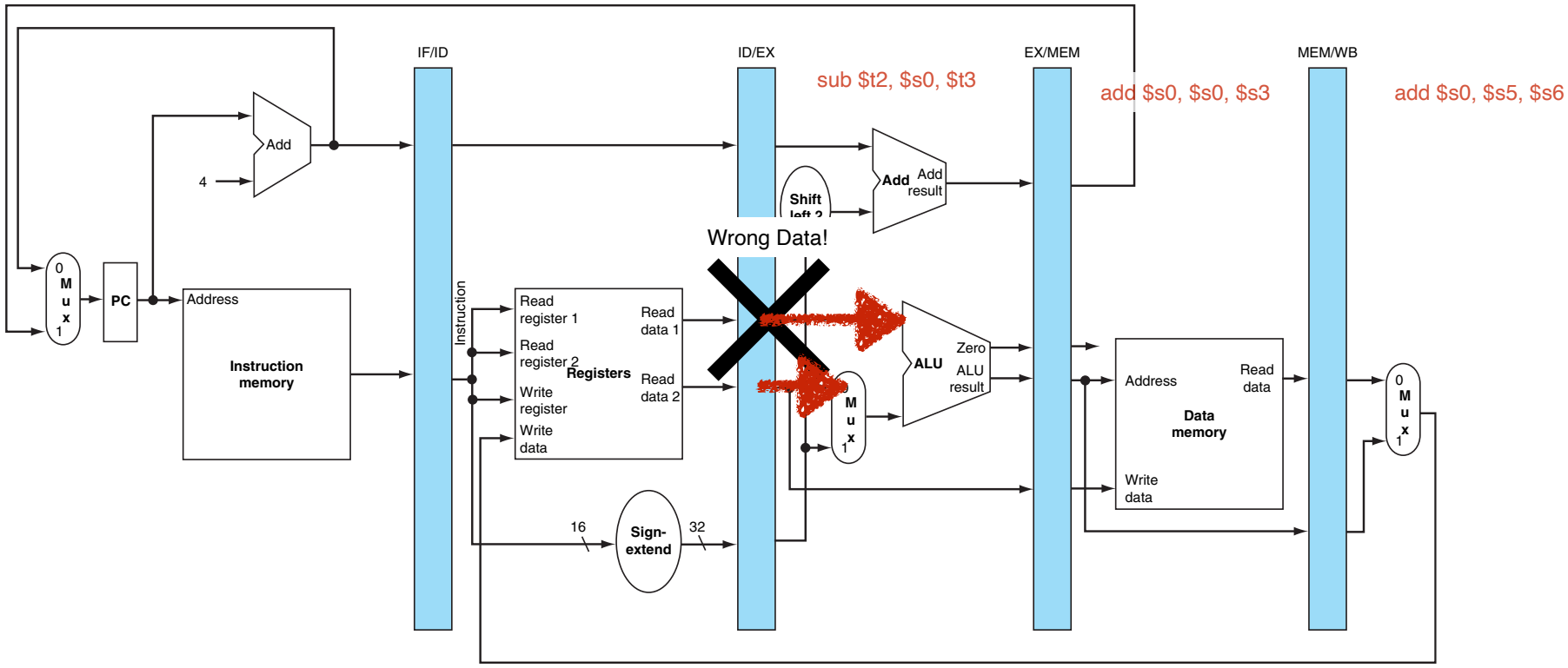


YIKES!!!!

Resolved!

General Forwarding Idea

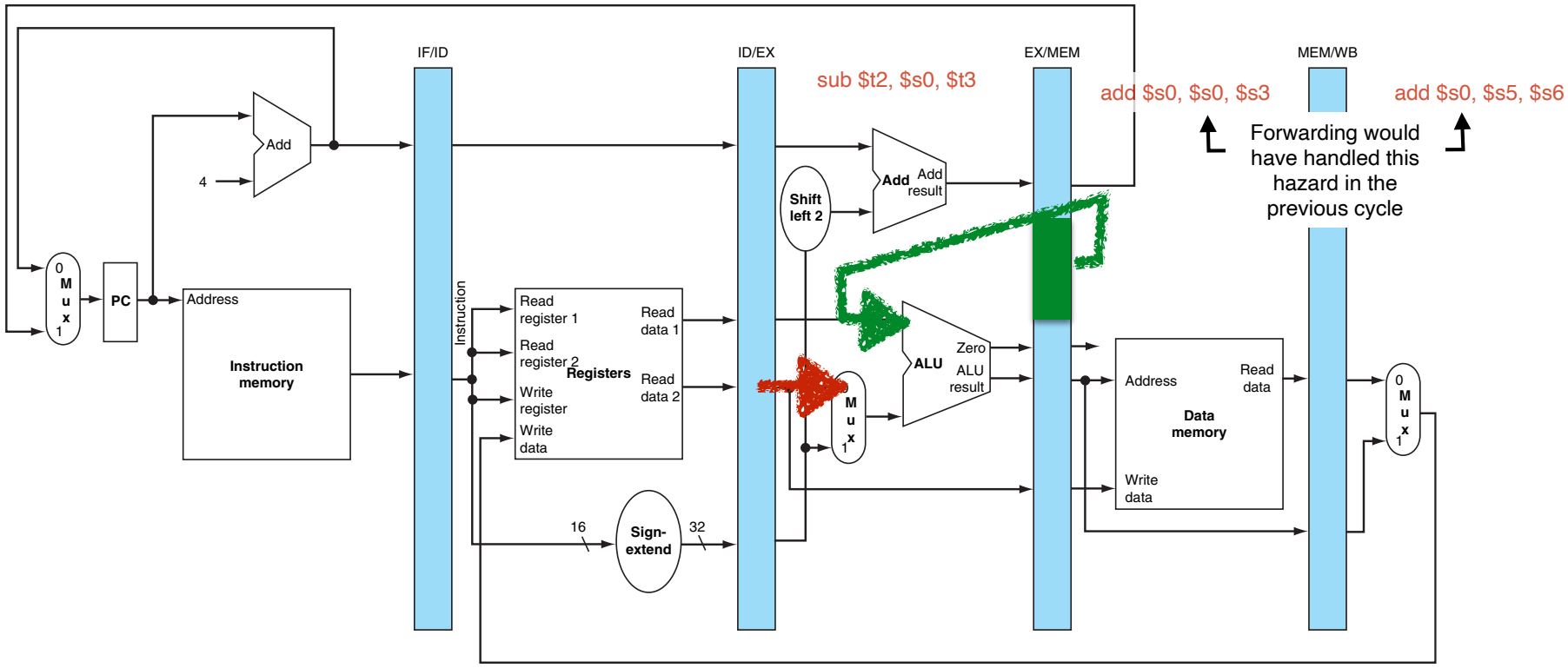
- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



YIKES!!!! Which $s0 prior state is the right one?

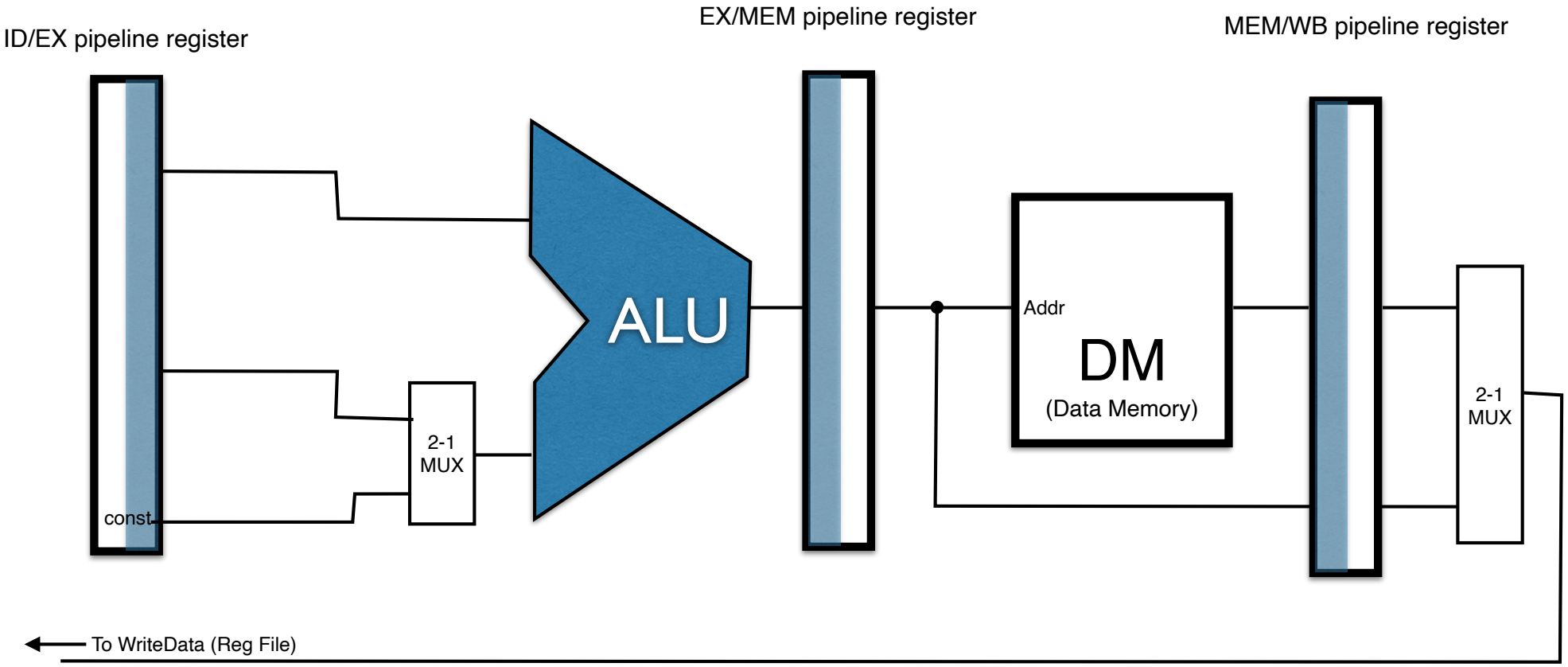
General Forwarding Idea

- Forwarding will always be implemented INTO the EX stage, i.e., what goes into the ALU?



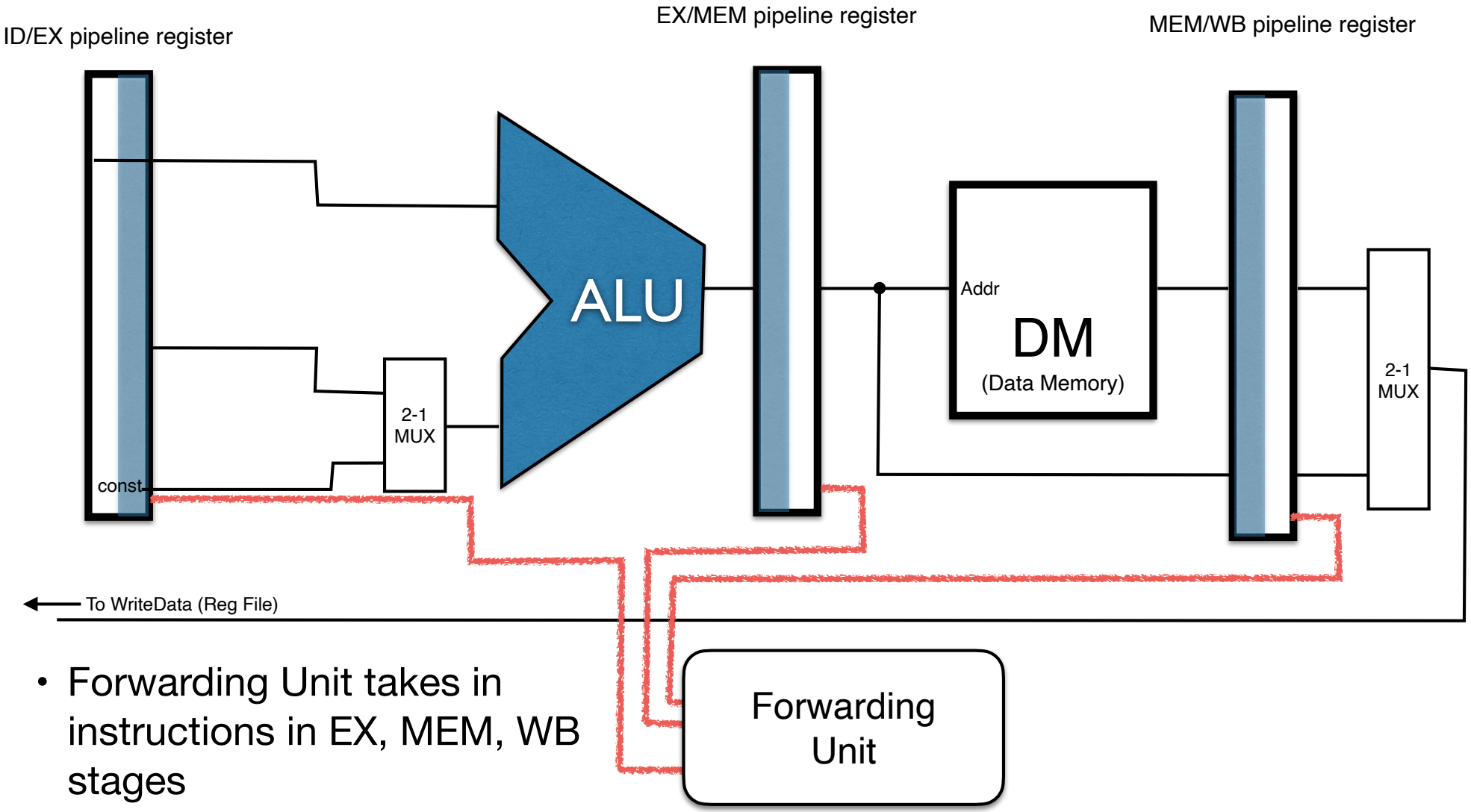
YIKES!!!! Which $s0 prior state is the right one?
Think about it from a coding perspective: it's the immediately preceding instruction.

Data Forwarding Design



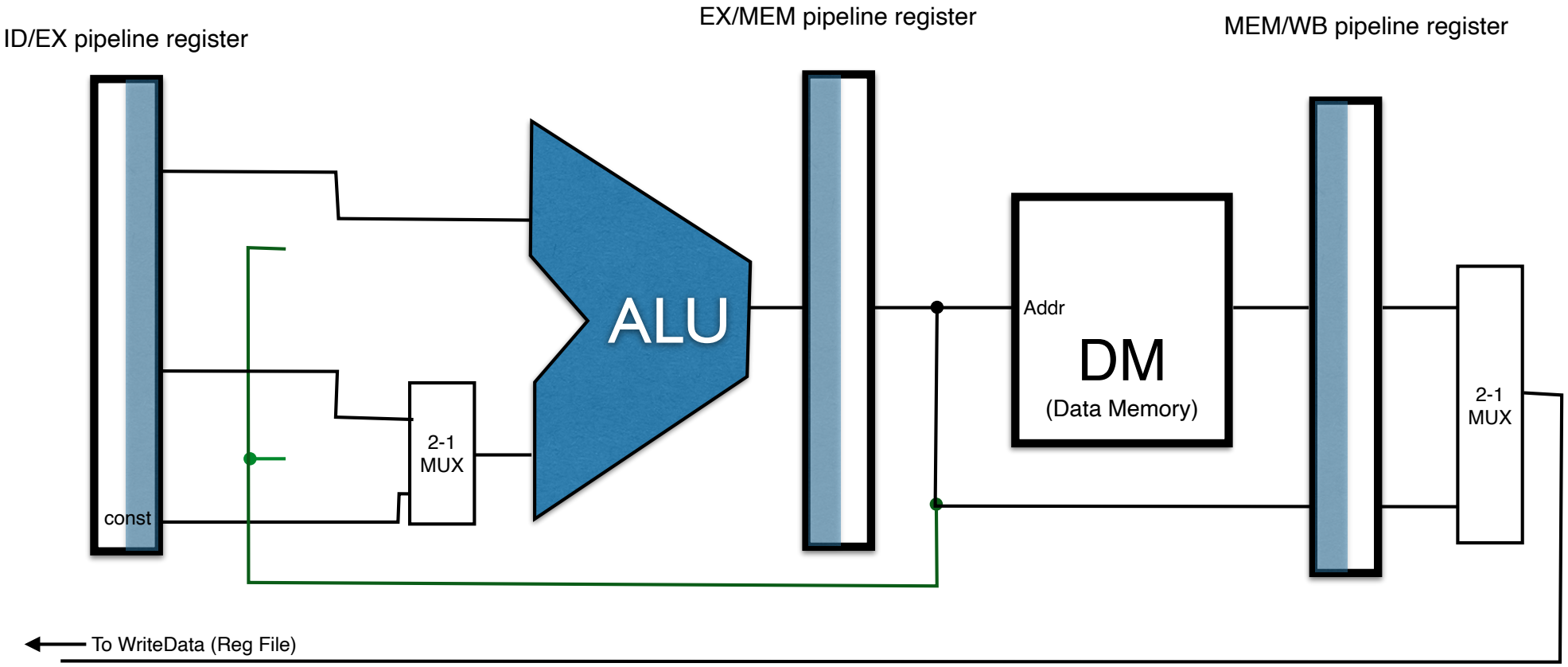
- non-data-forwarded pipeline arch

Data Forwarding Design



- Forwarding Unit takes in instructions in EX, MEM, WB stages

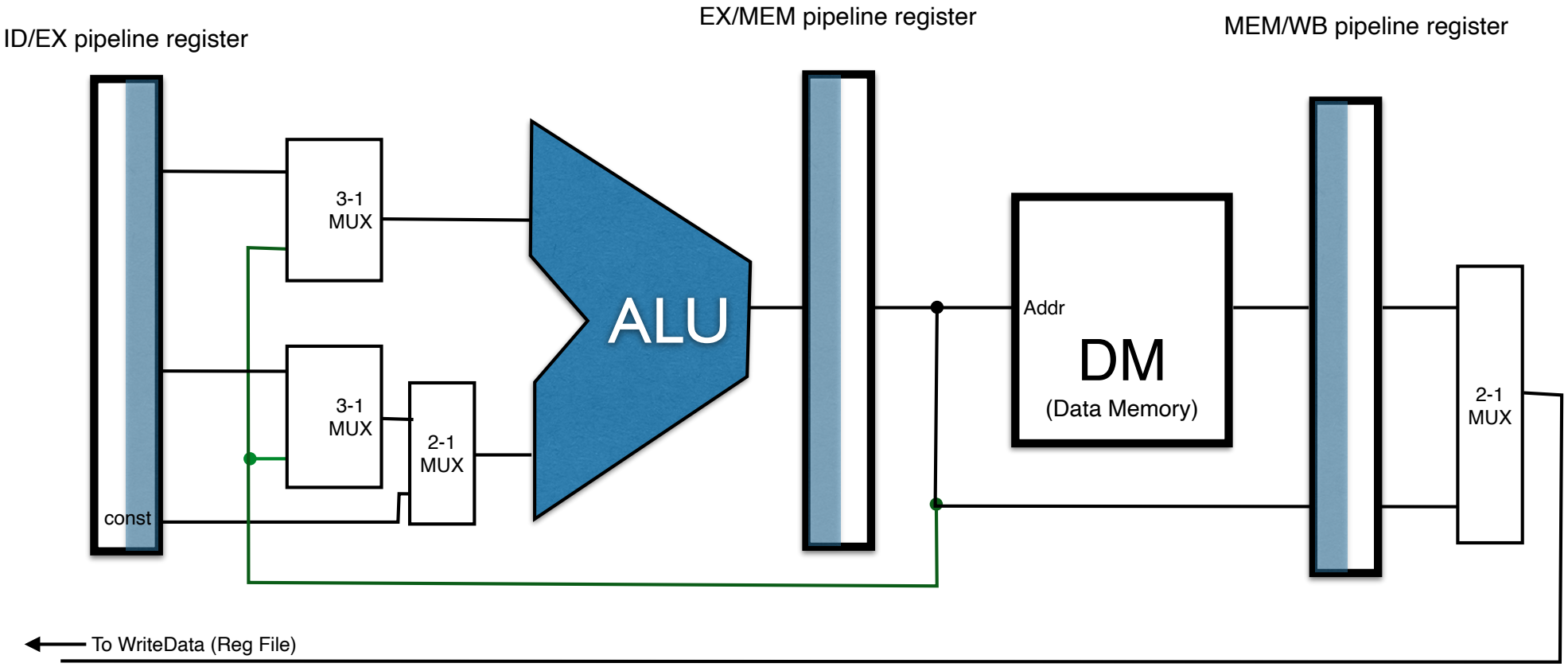
Data Forwarding Design



- Forwarding can come from MEM stage

Forwarding Unit

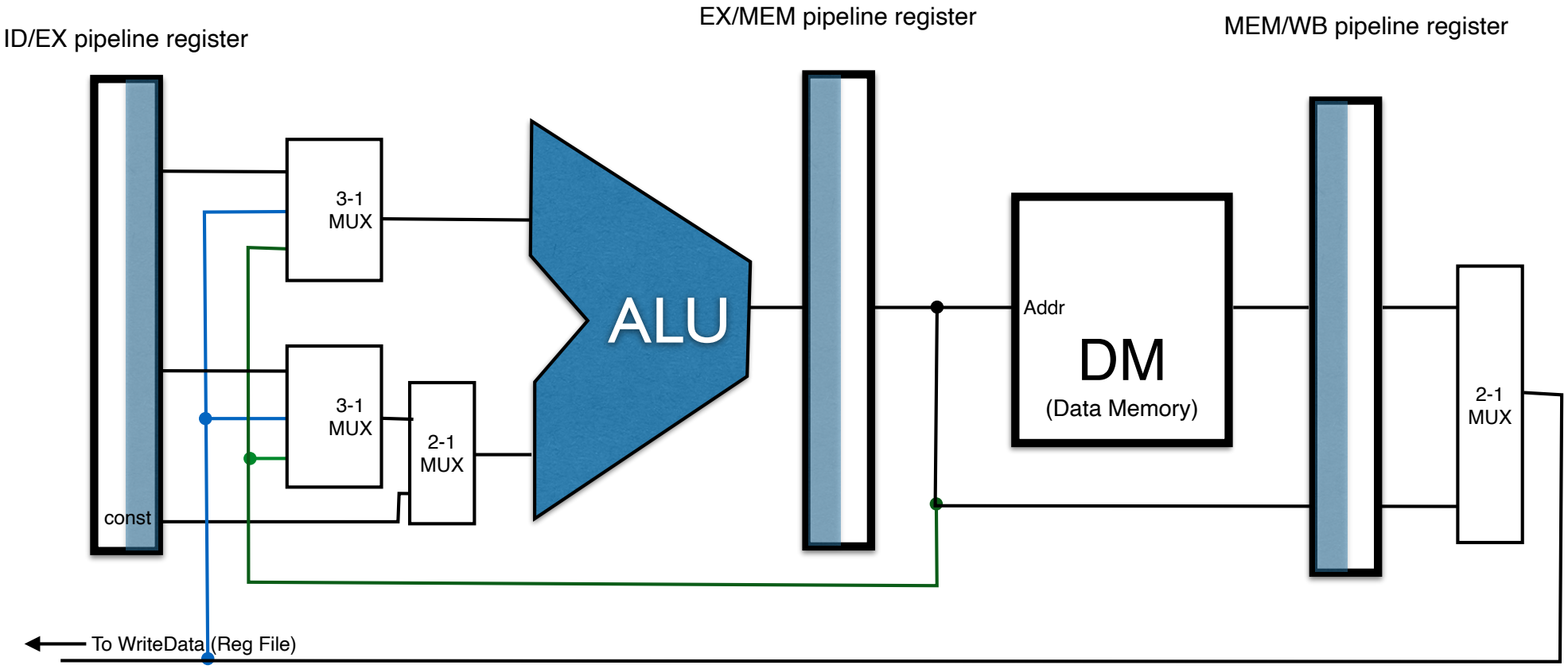
Data Forwarding Design



- Forwarding can come from MEM stage

Forwarding Unit

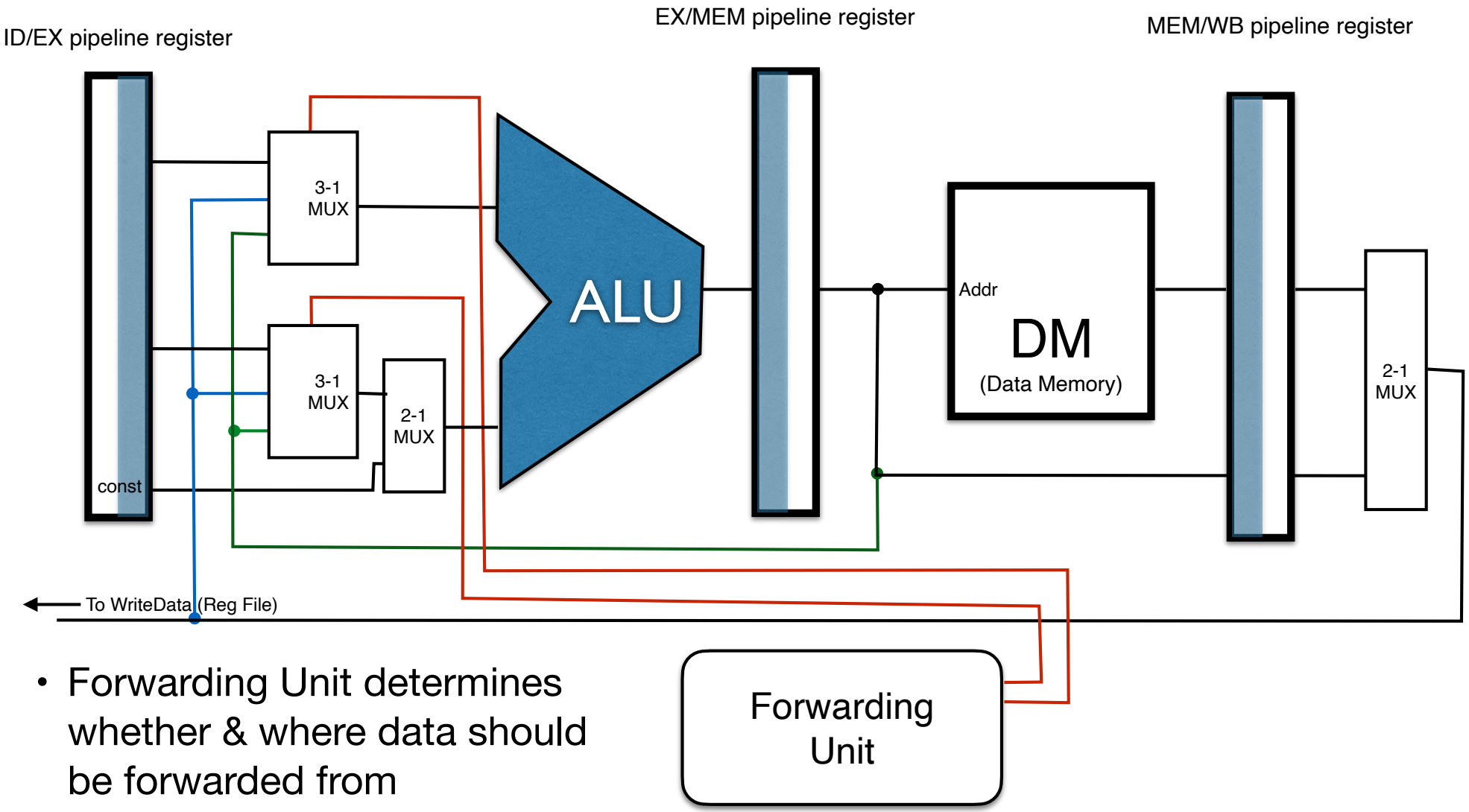
Data Forwarding Design



- Forwarding can also come from WB stage

Forwarding Unit

Data Forwarding Design

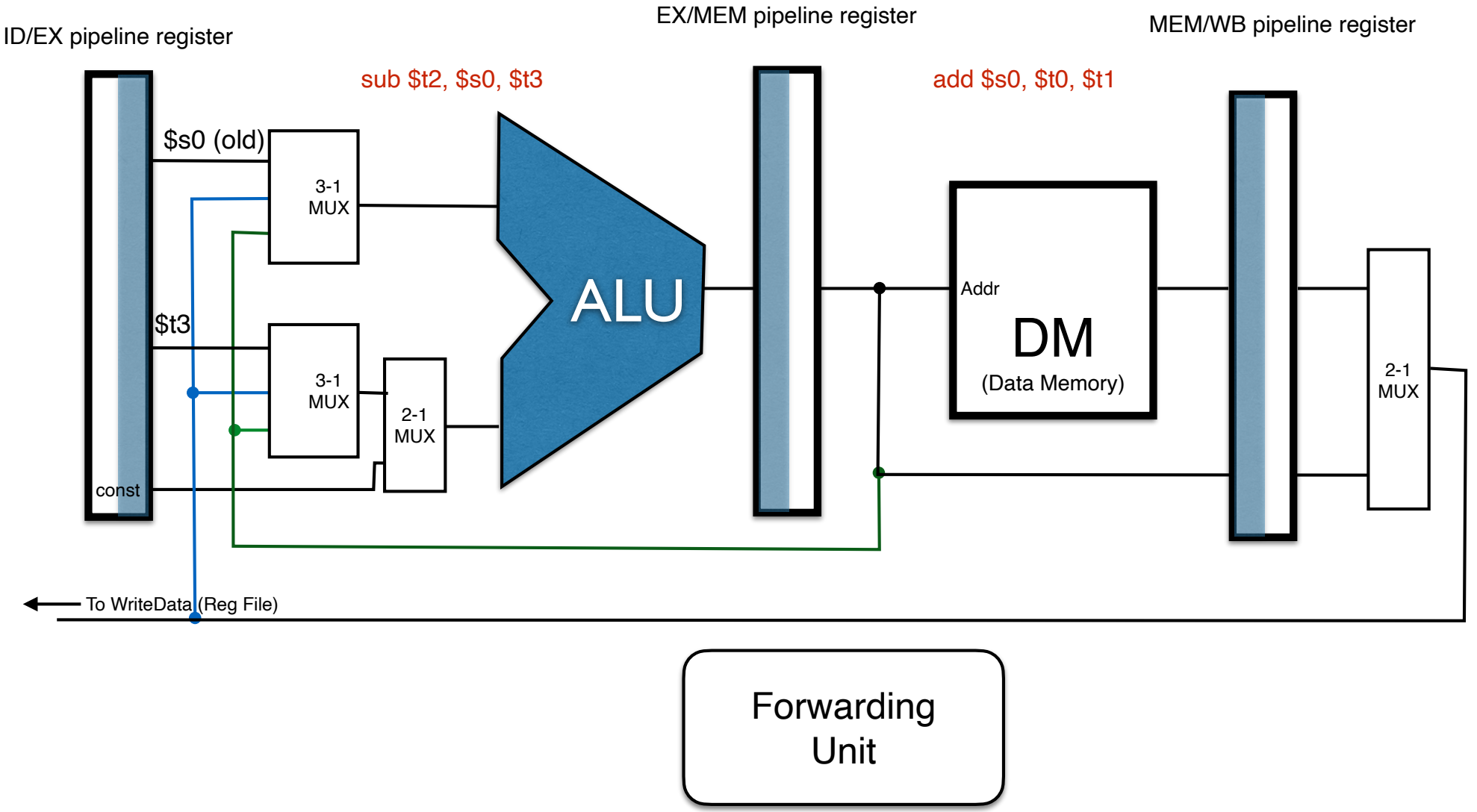


- Forwarding Unit determines whether & where data should be forwarded from

Example 1

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

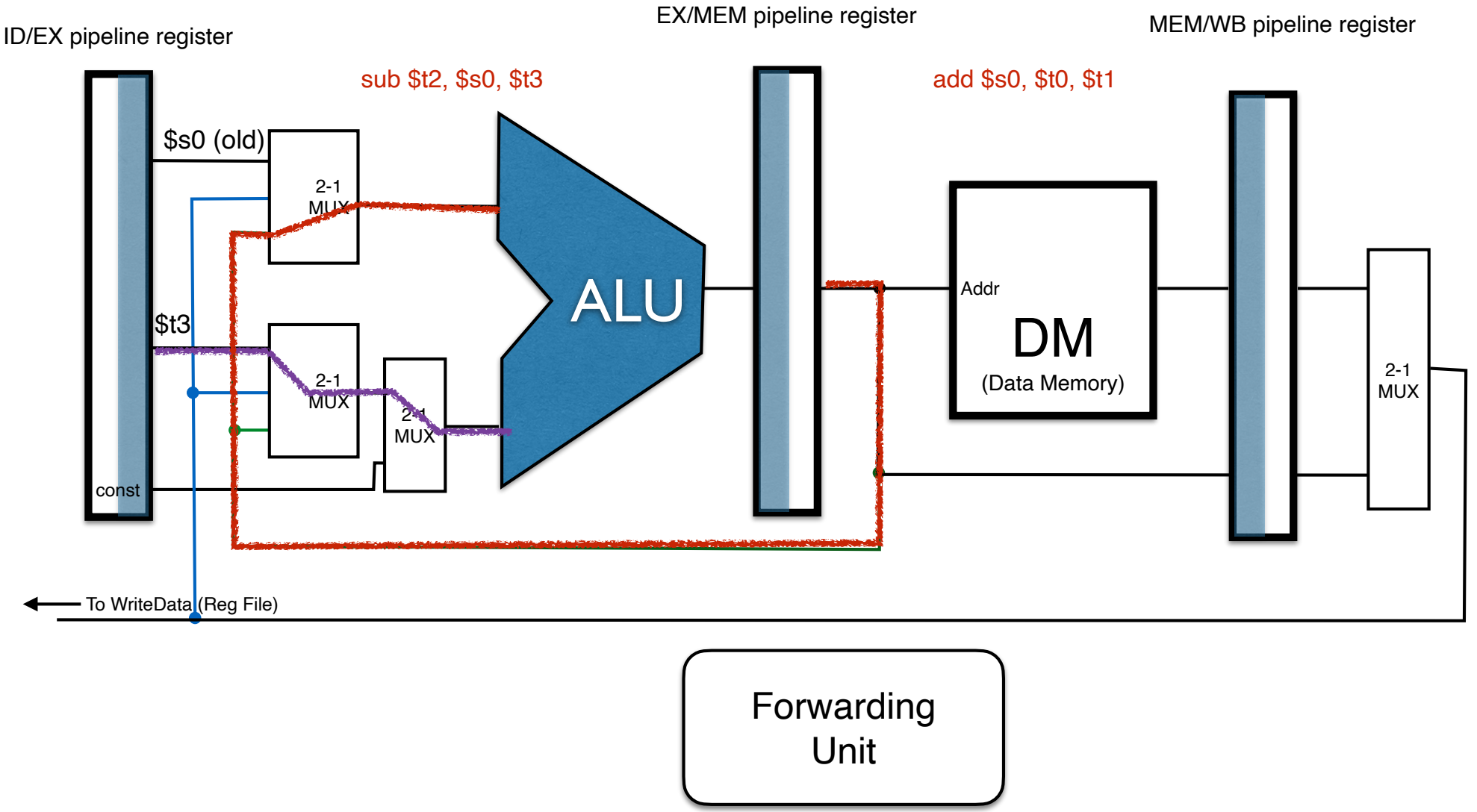
Forwarding to “add” to “sub”



Example 1

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```

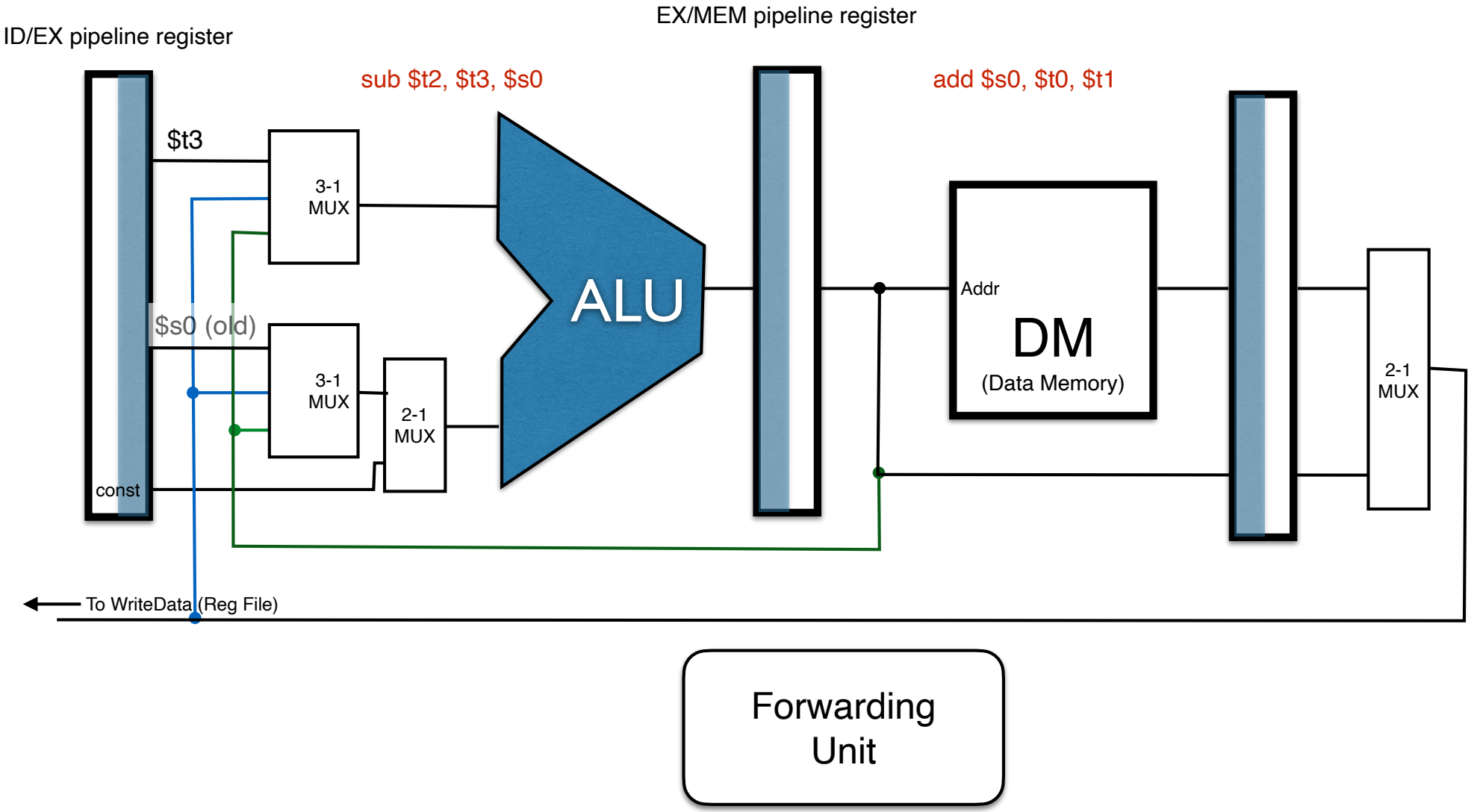
Forwarding to “add” to “sub”



Example 2: clock cycle t

```
add   $s0, $t0, $t1
sub   $t2, $t3, $s0
or    $t4, $s0, $t2
```

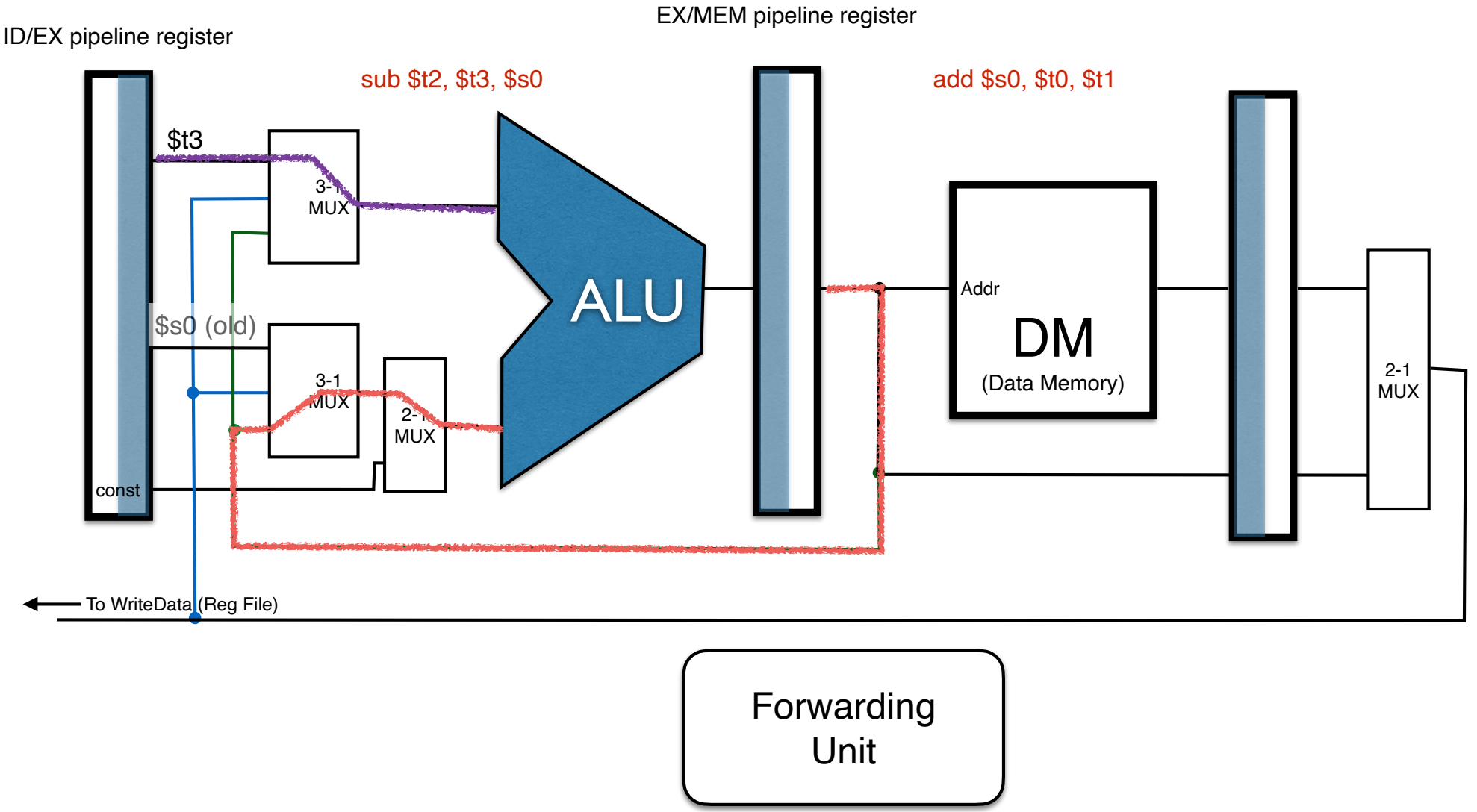
Forwarding to “add” to “sub”



Example 2: clock cycle t

```
add   $s0, $t0, $t1
sub   $t2, $t3, $s0
or    $t4, $s0, $t2
```

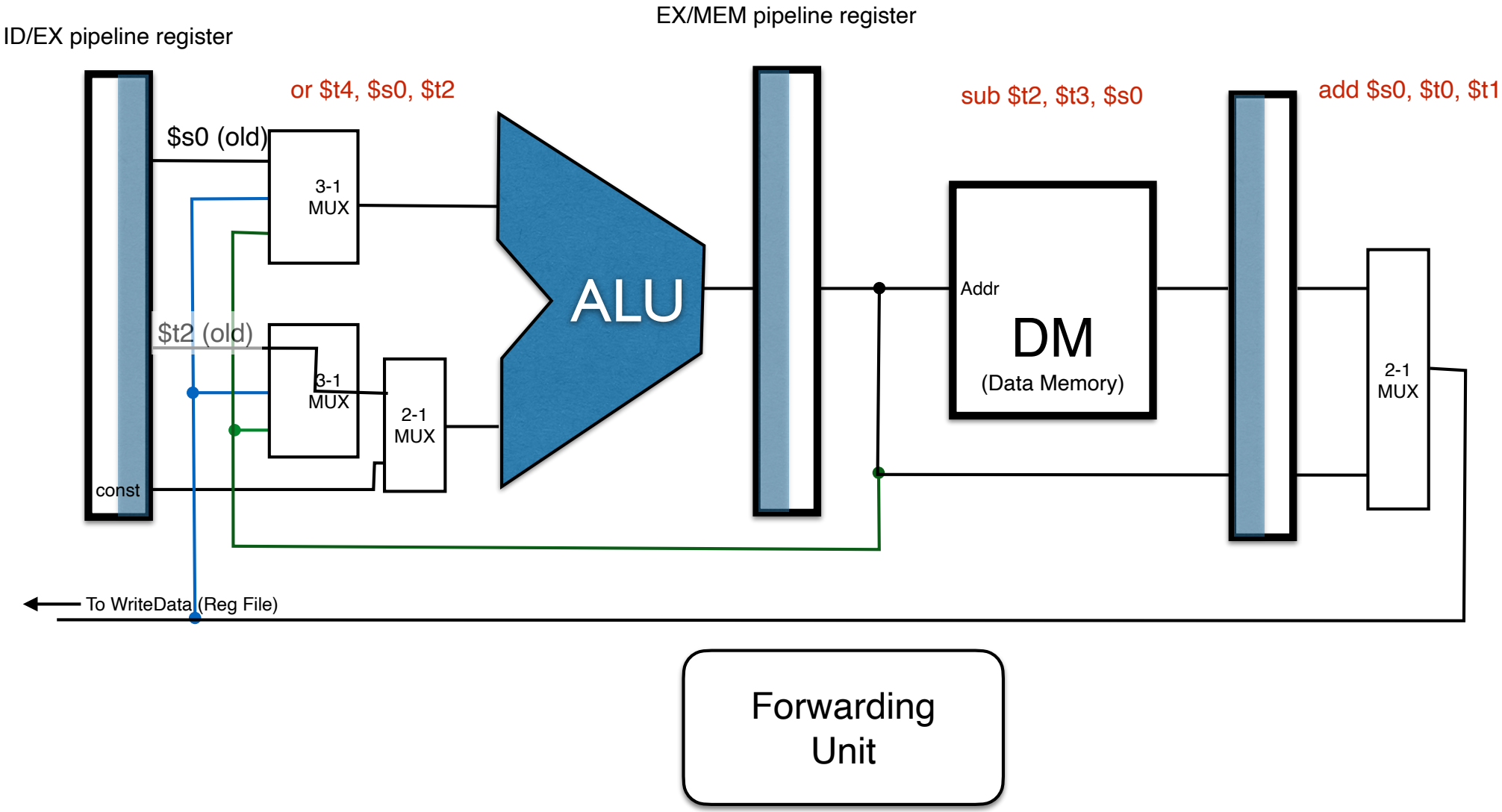
Forwarding to “add” to “sub”



Example 2: clock cycle t+1

```
add   $s0, $t0, $t1
sub   $t2, $t3, $s0
or    $t4, $s0, $t2
```

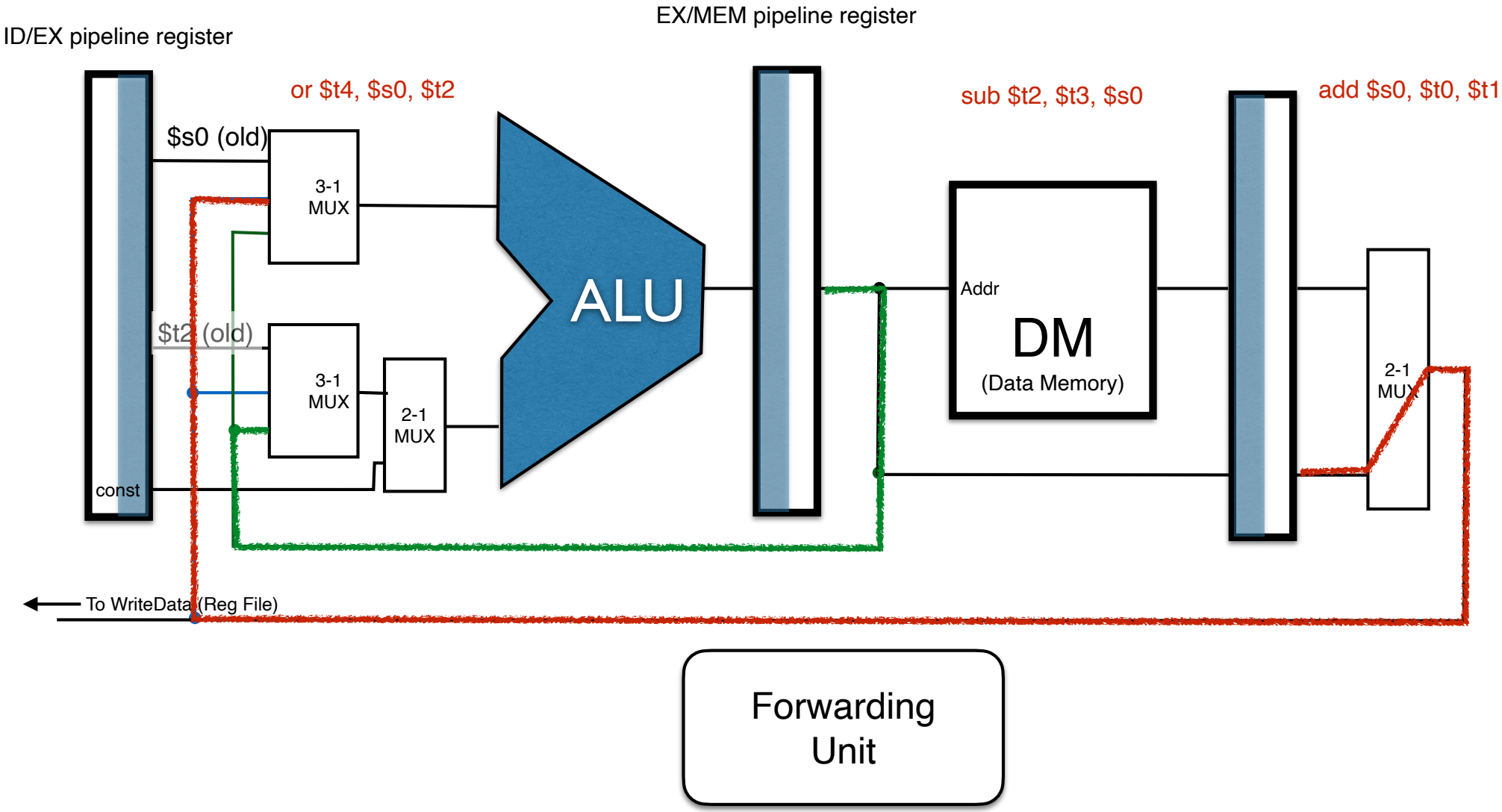
Forwarding to “sub” to “or”
Forwarding “add” to “or”



Example 2: clock cycle t+1

```
add   $s0, $t0, $t1
sub   $t2, $t3, $s0
or    $t4, $s0, $t2
```

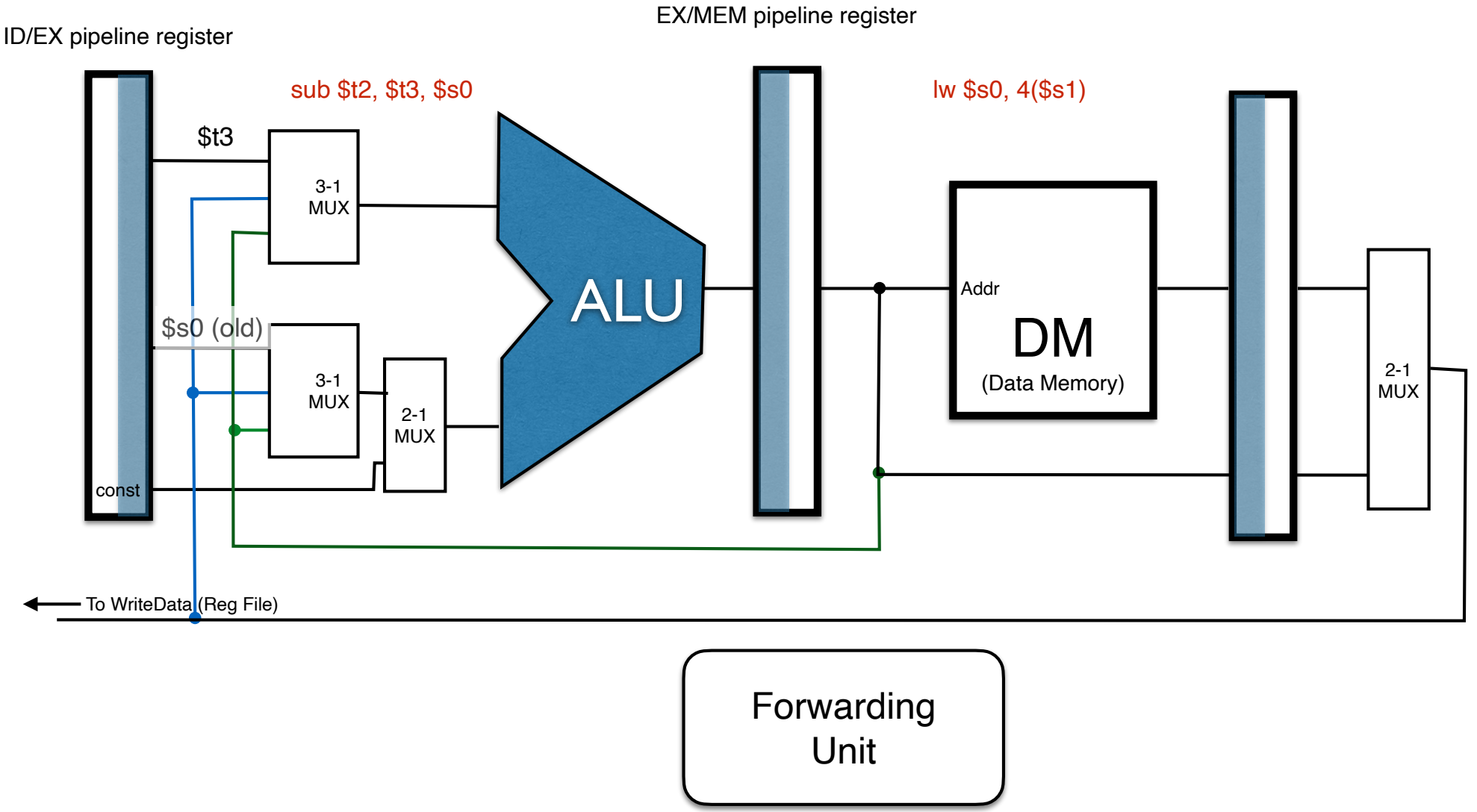
Forwarding to “sub” to “or”
Forwarding “add” to “or”



Example 3: May Still need stalls

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```

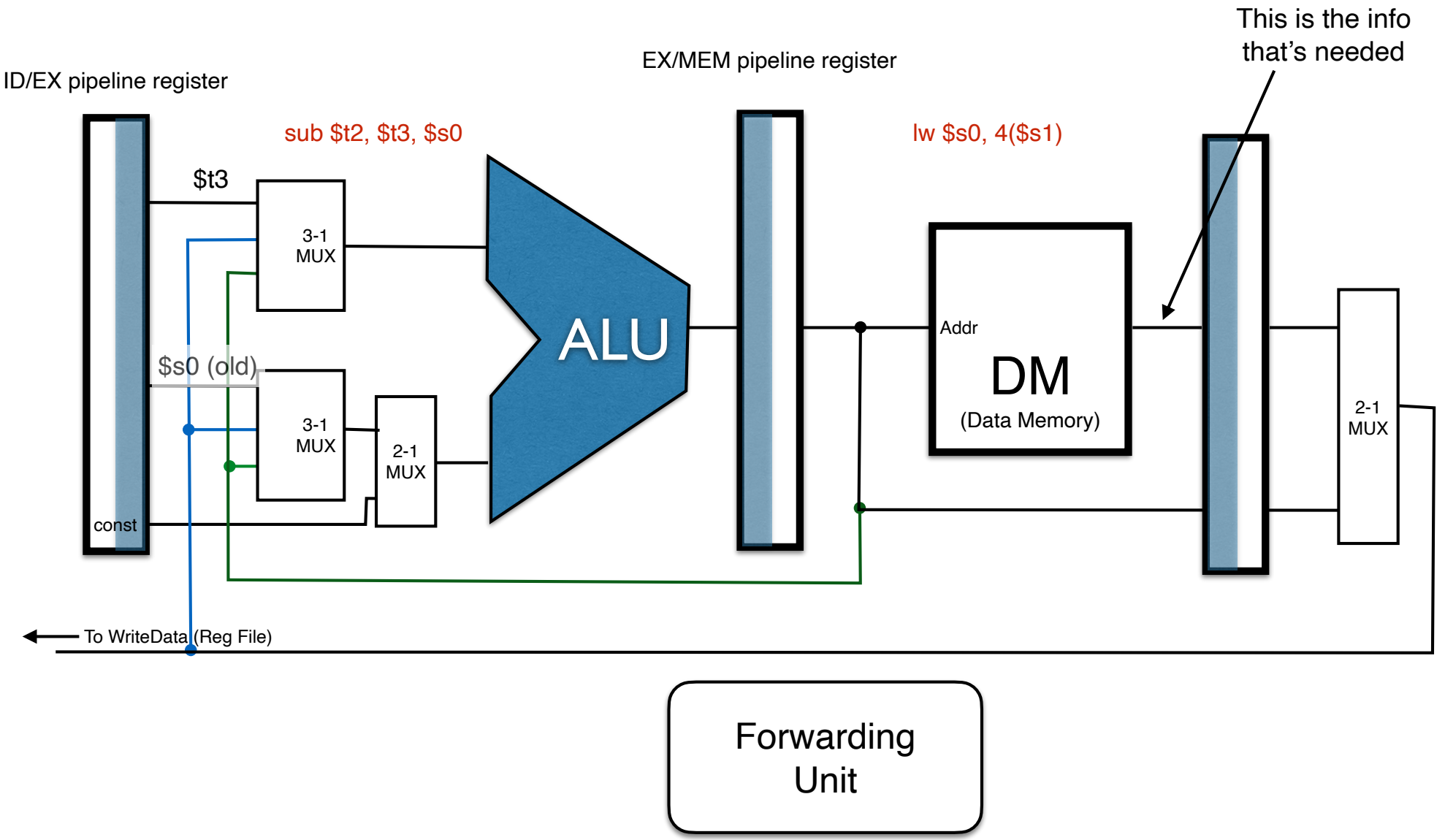
Need info coming out of memory: available too late in clock cycle



Example 3: May Still need stalls

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```

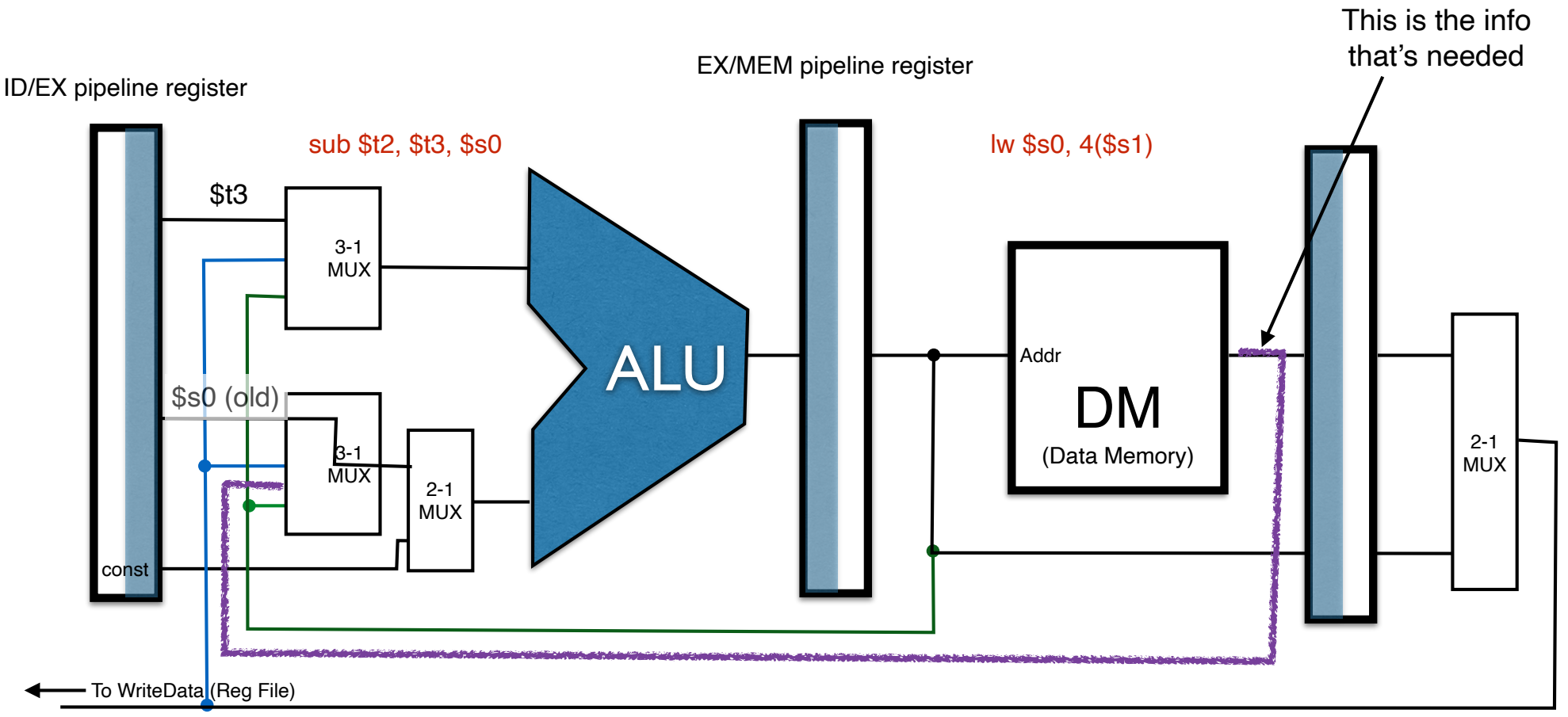
Need info coming out of memory: available too late in clock cycle



Example 3: May Still need stalls

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```

Need info coming out of memory: available too late in clock cycle

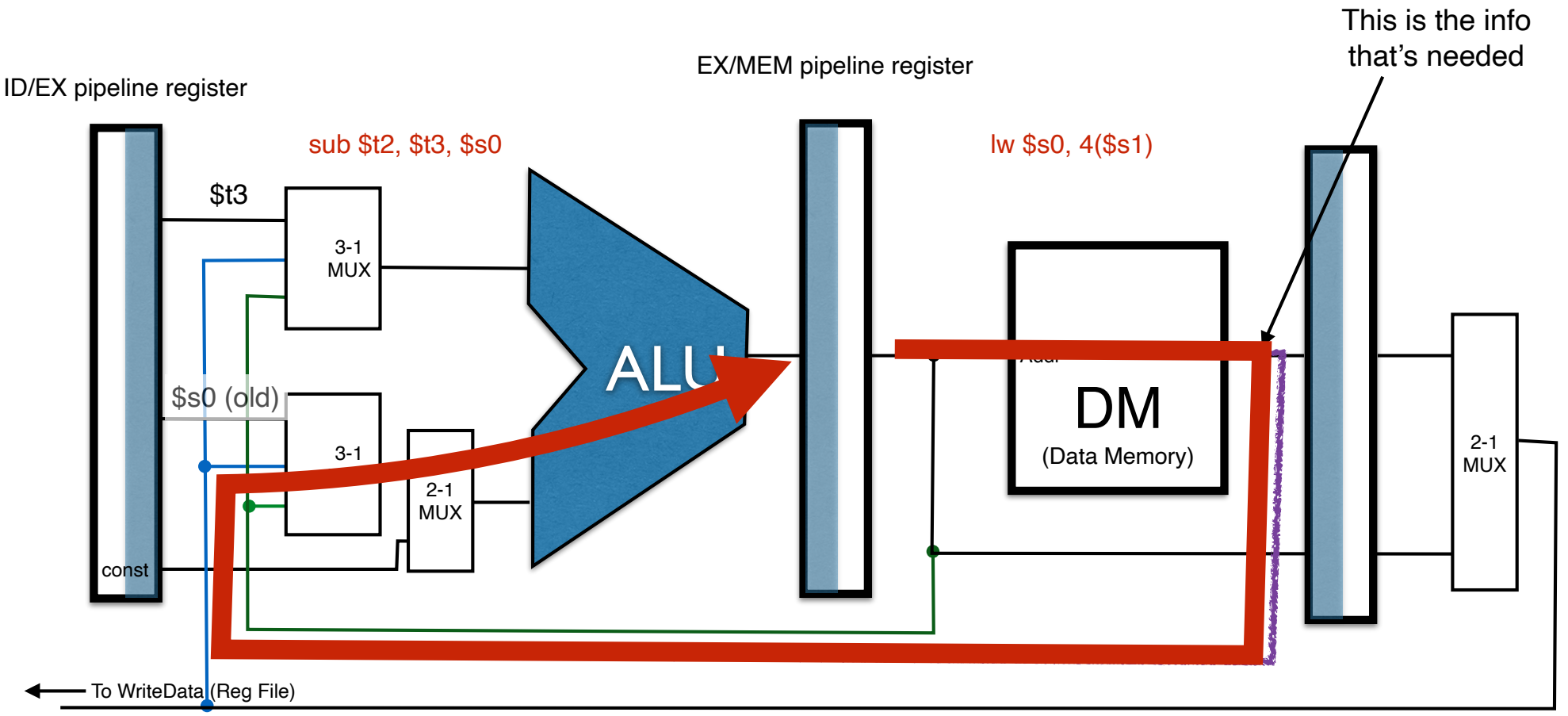


Q: Why not add wires like this to do forwarding?

Example 3: May Still need stalls

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```

Need info coming out of memory: available too late in clock cycle



Q: Why not add wires like this to do forwarding?

A: Clock cycle would have to be slowed too much (doubled?) to permit both a memory read and ALU computation in a single clock cycle: would slow every instruction to avoid occasional lw stall

Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed, can't forward backward in time!

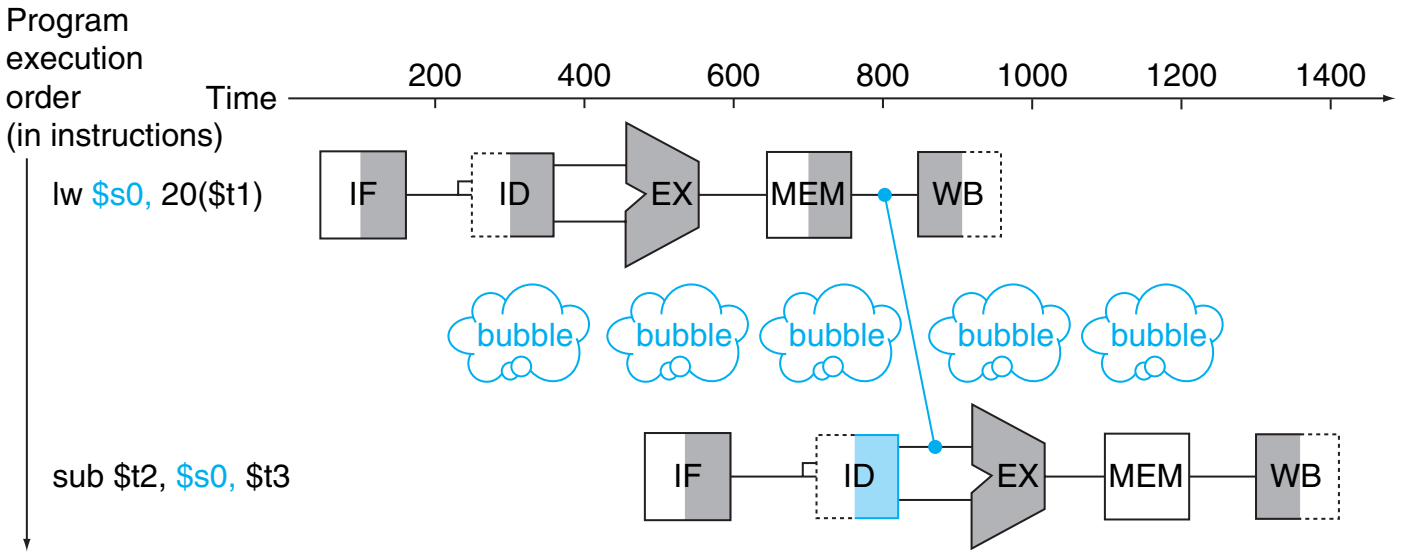
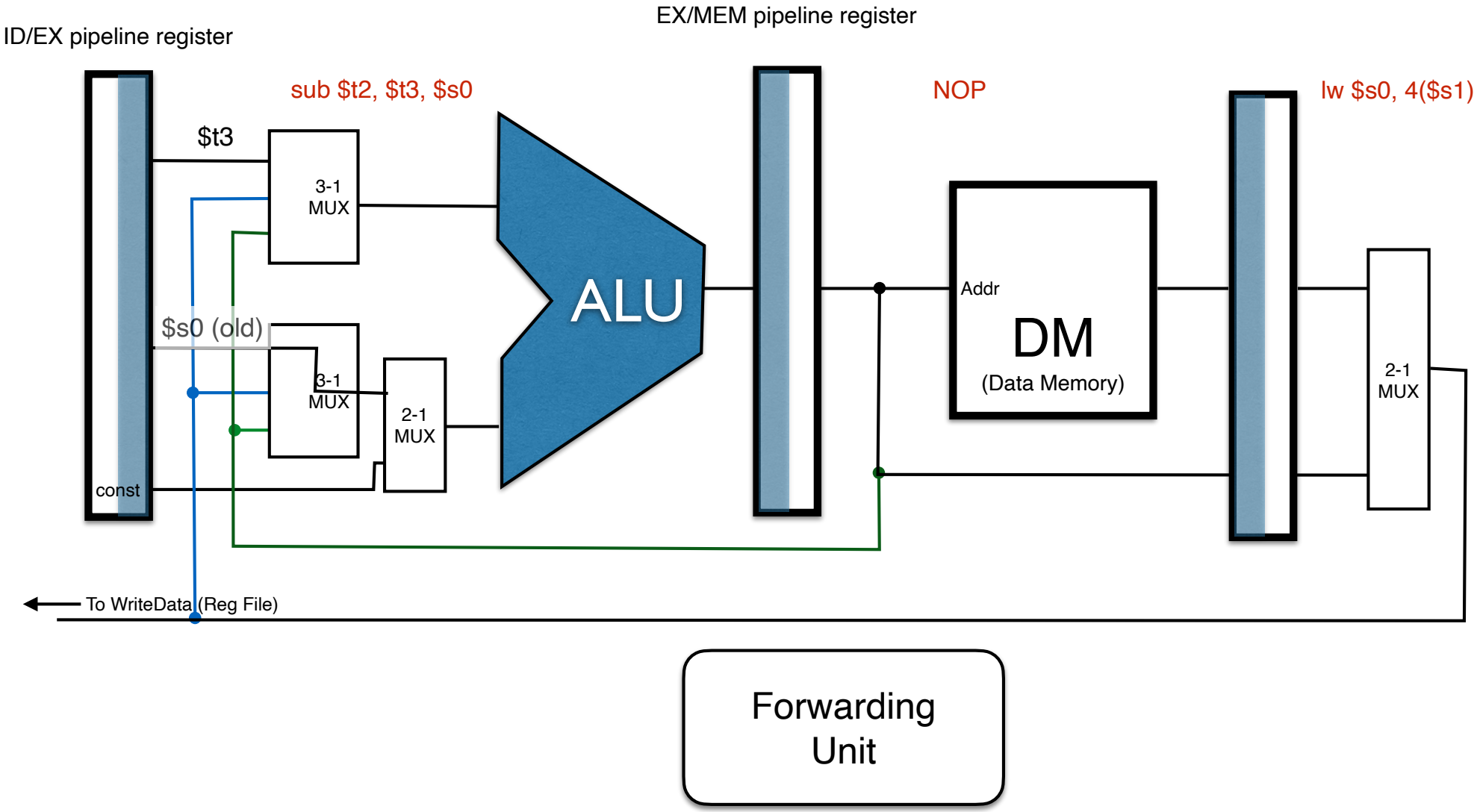


FIGURE 4.30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data. Without the stall, the path from memory access stage output to execution stage input would be going backward in time, which is impossible. This figure is actually a simplification, since we cannot know until after the subtract instruction is fetched and decoded whether or not a stall will be necessary. Section 4.7 shows the details of what really happens in the case of a hazard.

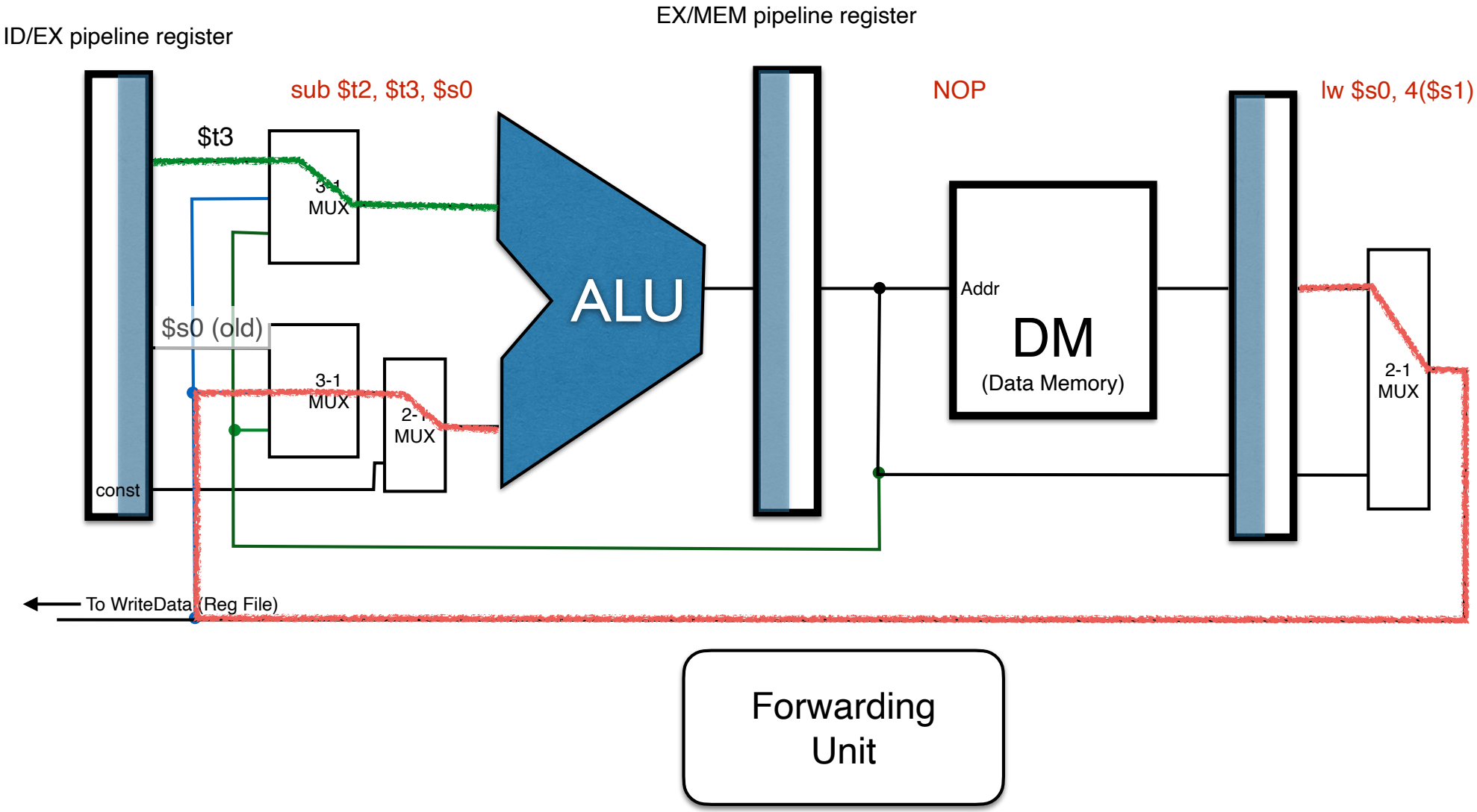
Solution: stall & data fwd

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```



Solution: stall & data fwd

```
lw    $s0, 4($s1)
sub   $t2, $t3, $s0
```



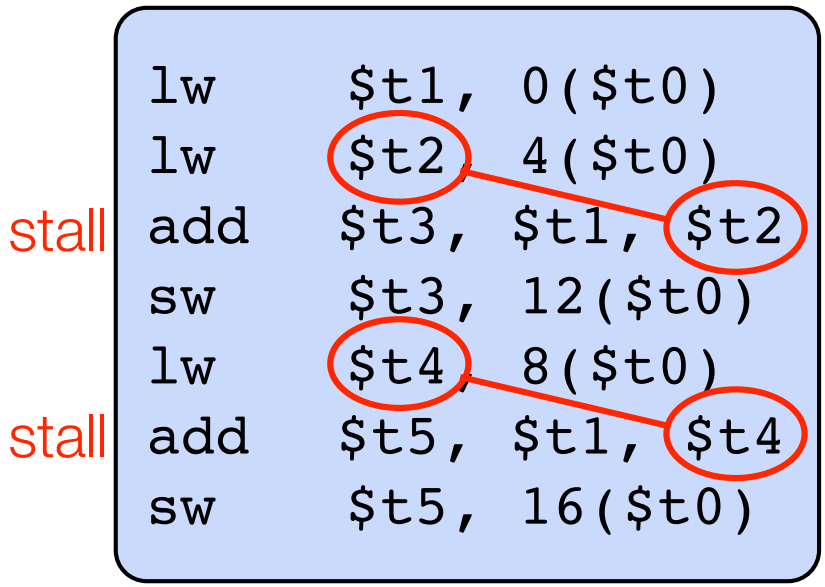
Final Thoughts on Data Fwd Design

- Data Forwarding Preferred to Stalling since does not introduce “bubble”, does not waste a clock cycle
- For data hazard, when to stall depends on whether or not data forwarding is implemented (& how)
- Forwarding Unit and Hazard Detection (that decides when to stall) operate independently, but “aware” of existence of the other
  - Both independently read in details of instructions (opcodes + reg #s)
  - Hazard Detection Unit stalls instructions in ID stage. Will “know” when hazards are being resolved by data forwarding
  - Forwarding Unit does forwarding in stages after ID

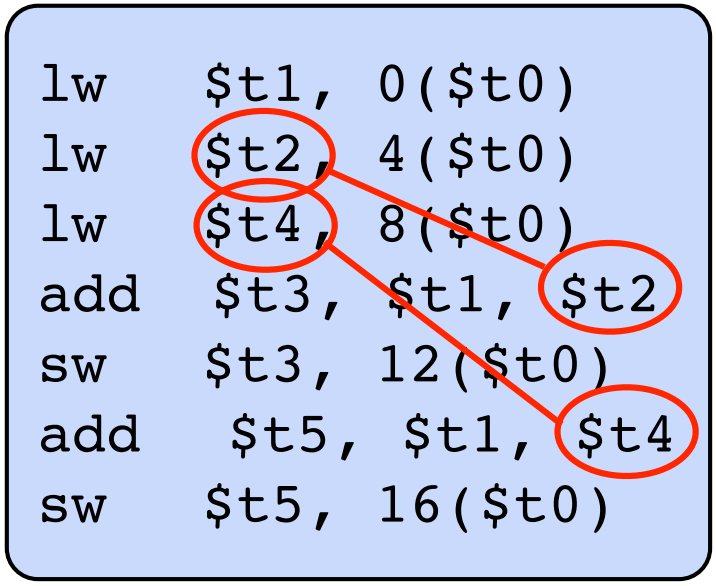
Assembly Tricks to Avoid Data Stalls

Code Scheduling to Avoid Stalls

- Sometimes can reorder code to avoid use of load result in the next instruction



13 cycles



11 cycles

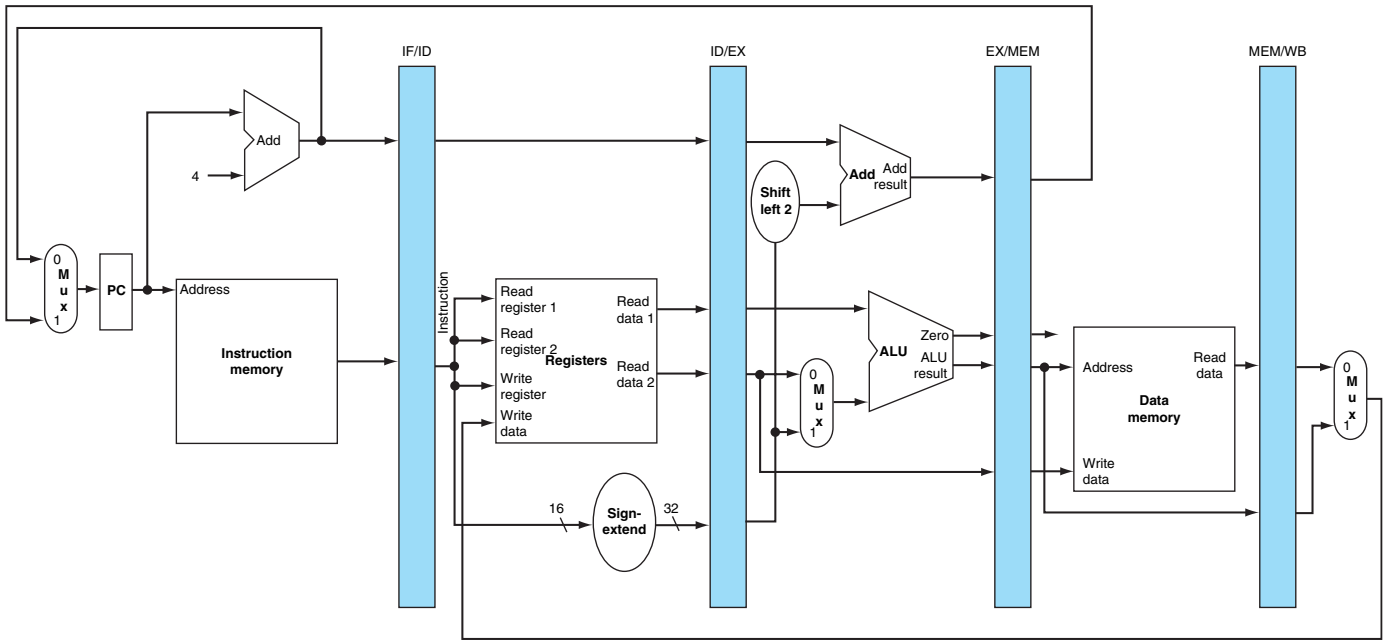
- Just make sure not to introduce other dependencies that might cause stalls!

Control Hazard Resolution
(look @ jump, then branch)

Jump Example

PC

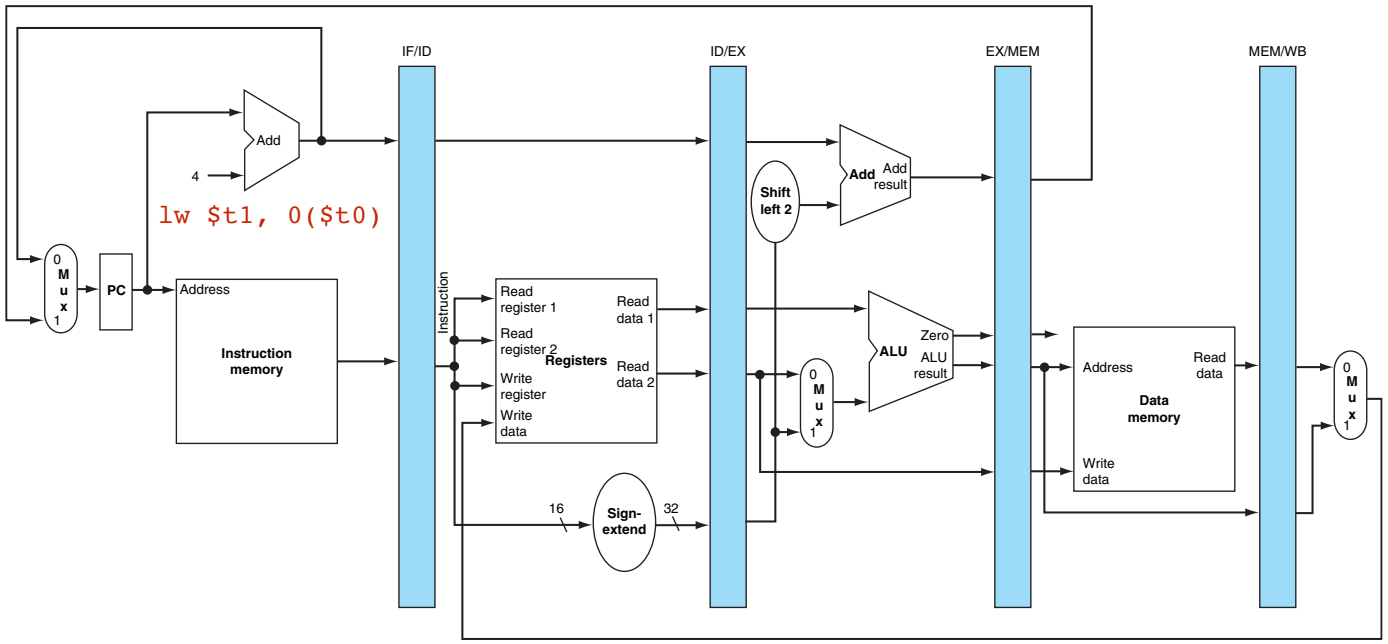
```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



Jump Example

```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

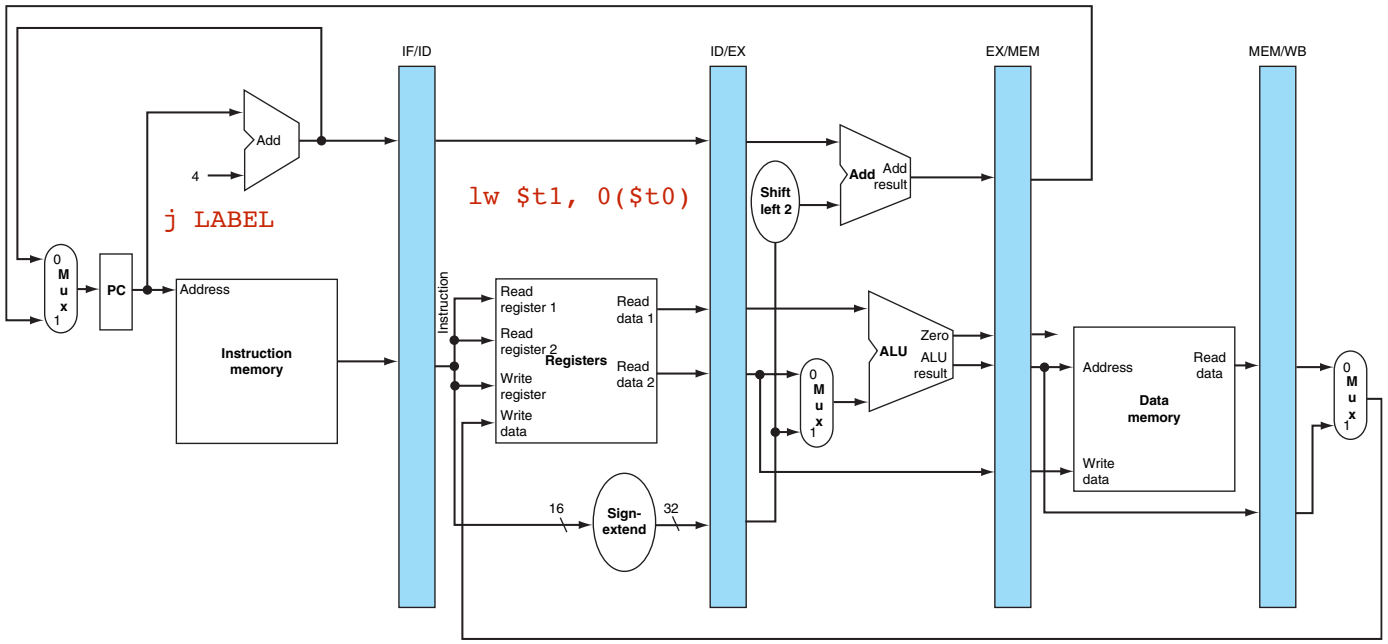
PC



Jump Example

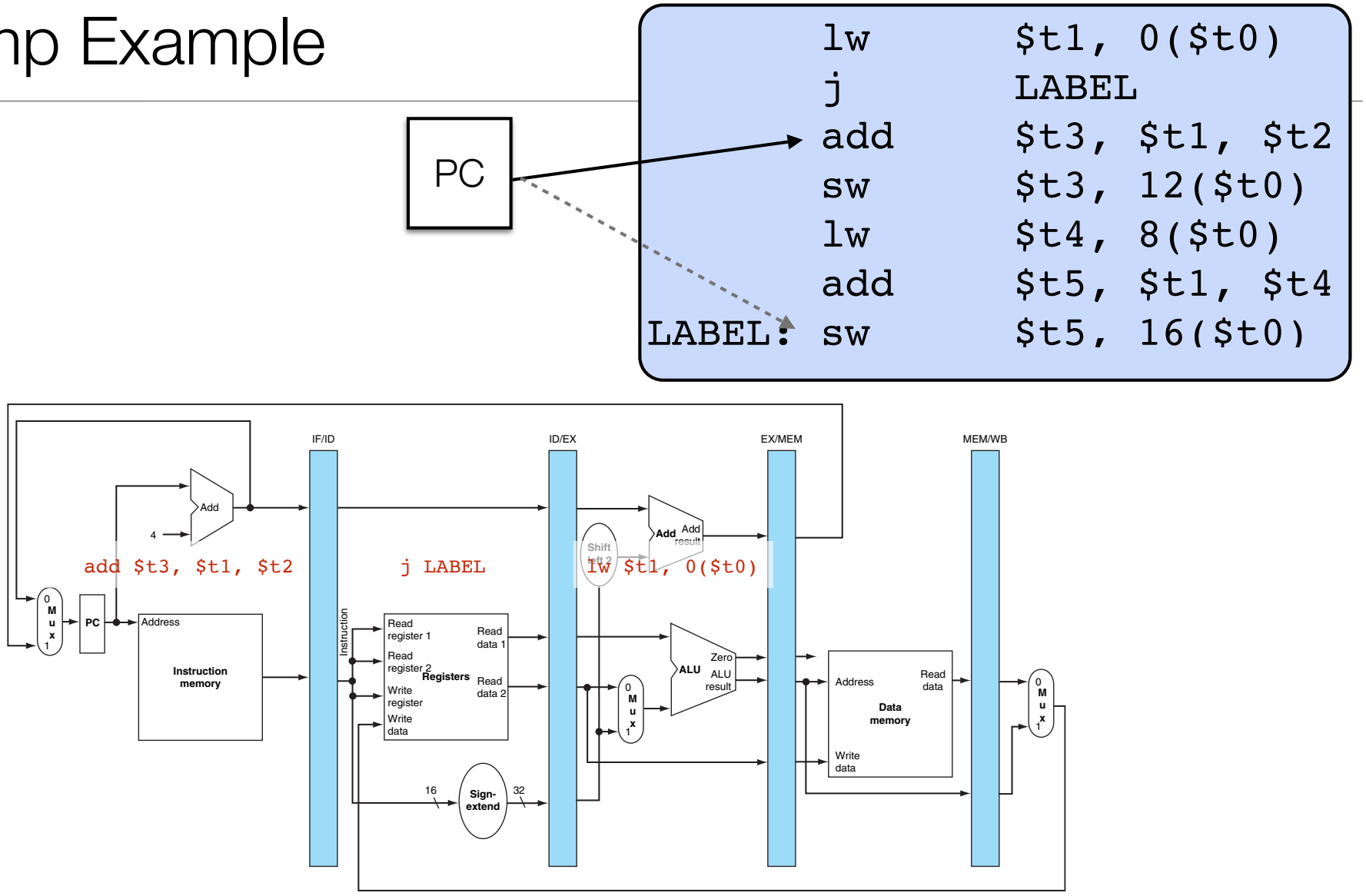
```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC



- Even though j now in pipeline, it hasn't been evaluated yet. PC+4 instruction will be brought into pipeline next clock cycle

Jump Example

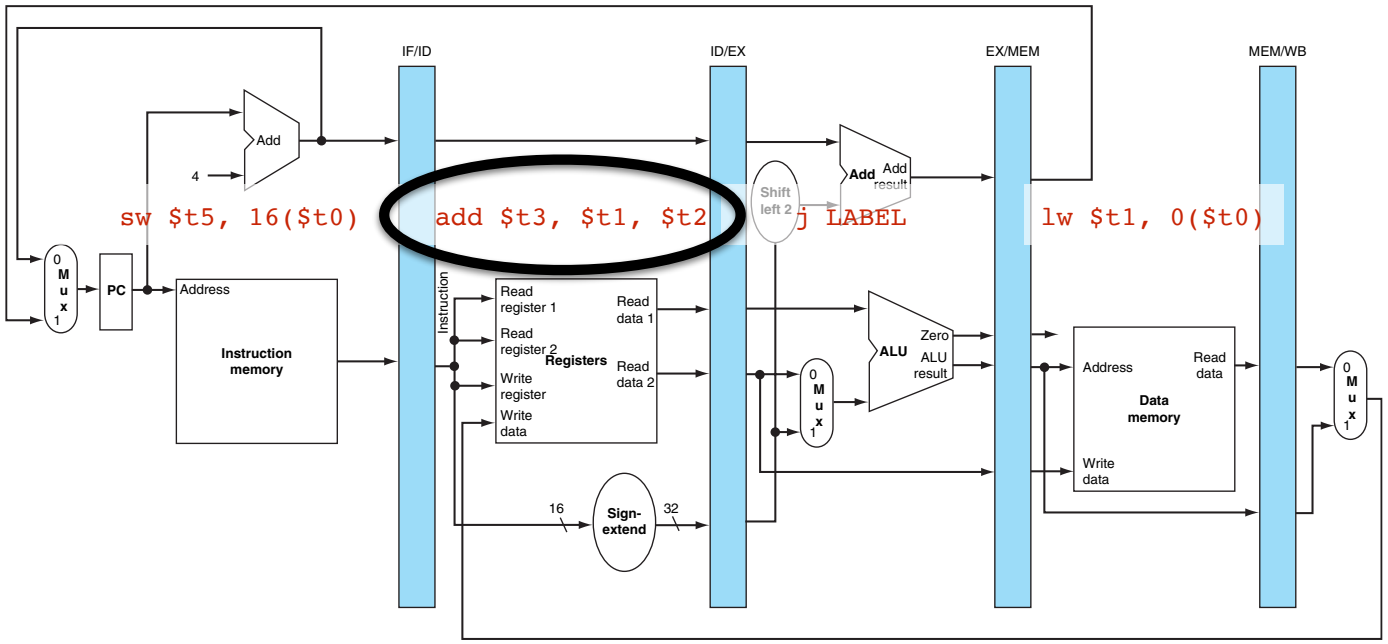


- Now j is being evaluated and PC set to address of LABEL, but wrong instruction already pulled in

Jump Example

```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

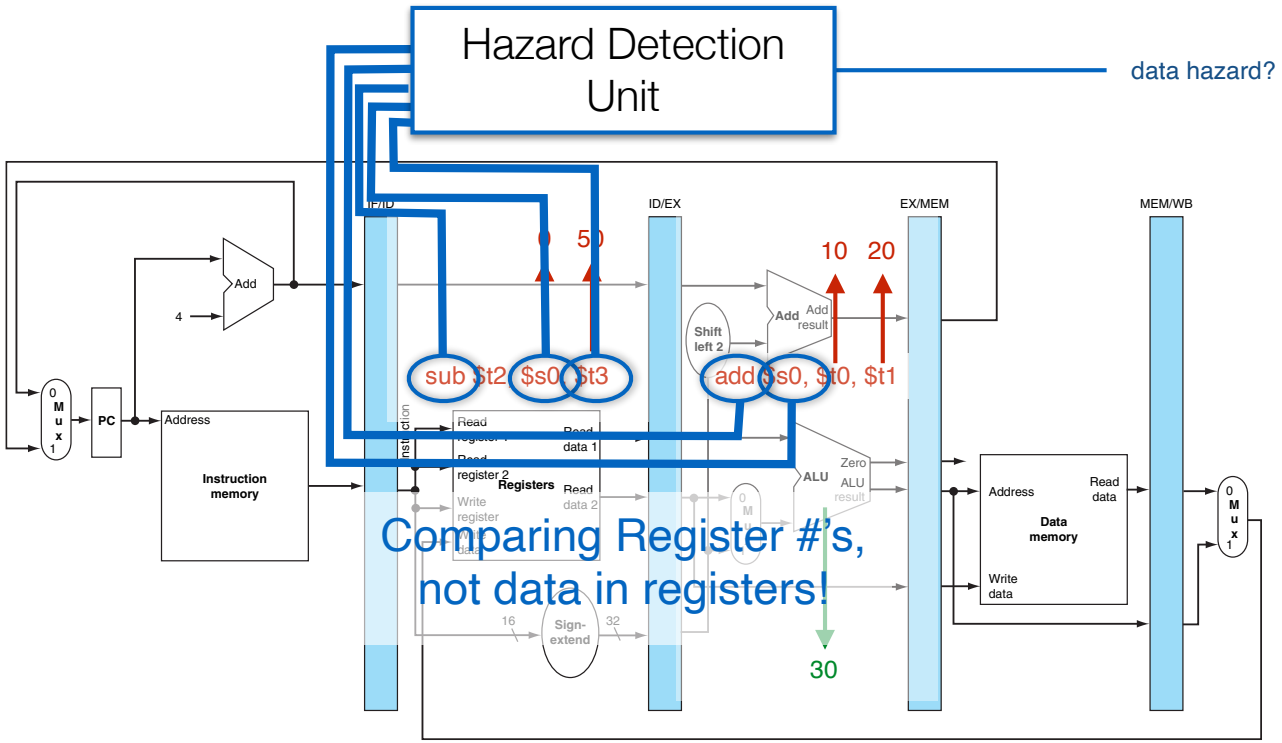
PC



- j instruction done adjusting PC. Will be NOP in 3 remaining stages.

- Still have add instruction that doesn't belong in pipeline

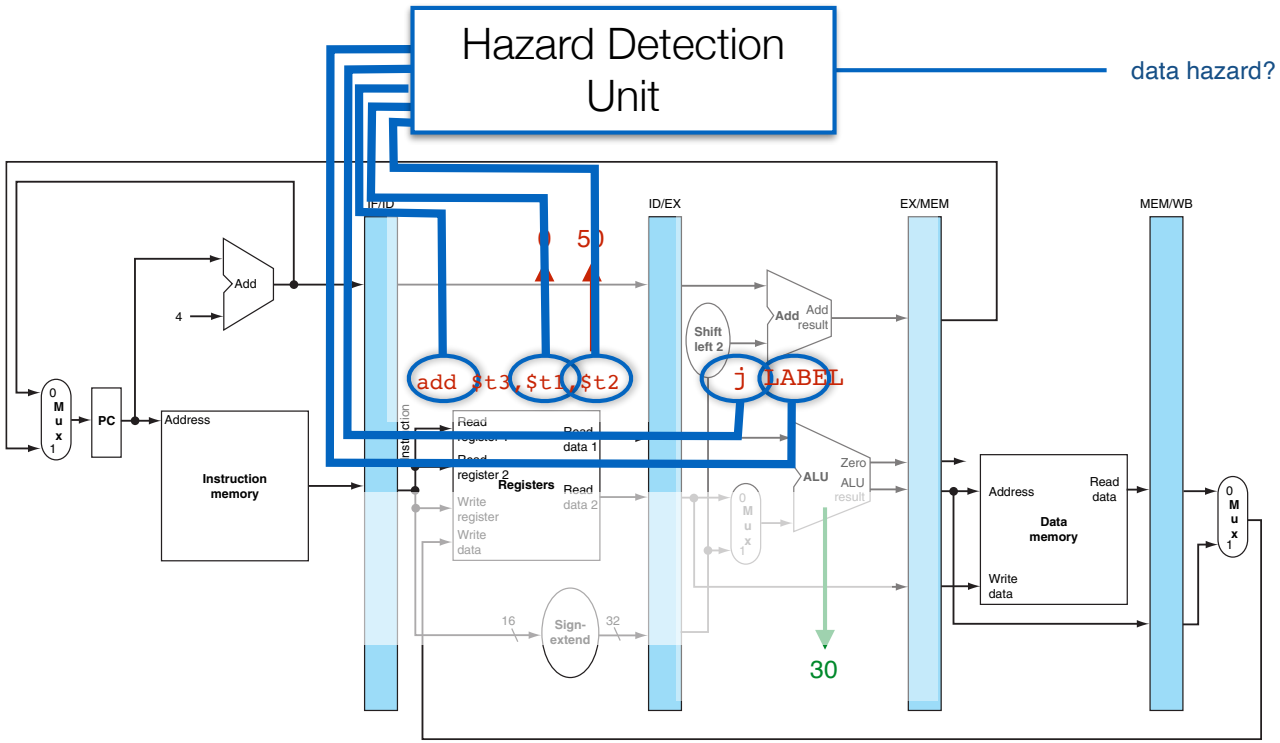
Recall: Add'l Circuitry could detect hazard



Also check the other read parameter (if R-type instruction)

- In this clock cycle, both instructions are in the pipeline
- Could add additional (combinatorial) circuitry to identify:
  - Instruction in stage 2 is reading from a register being written to by instruction in stage 3

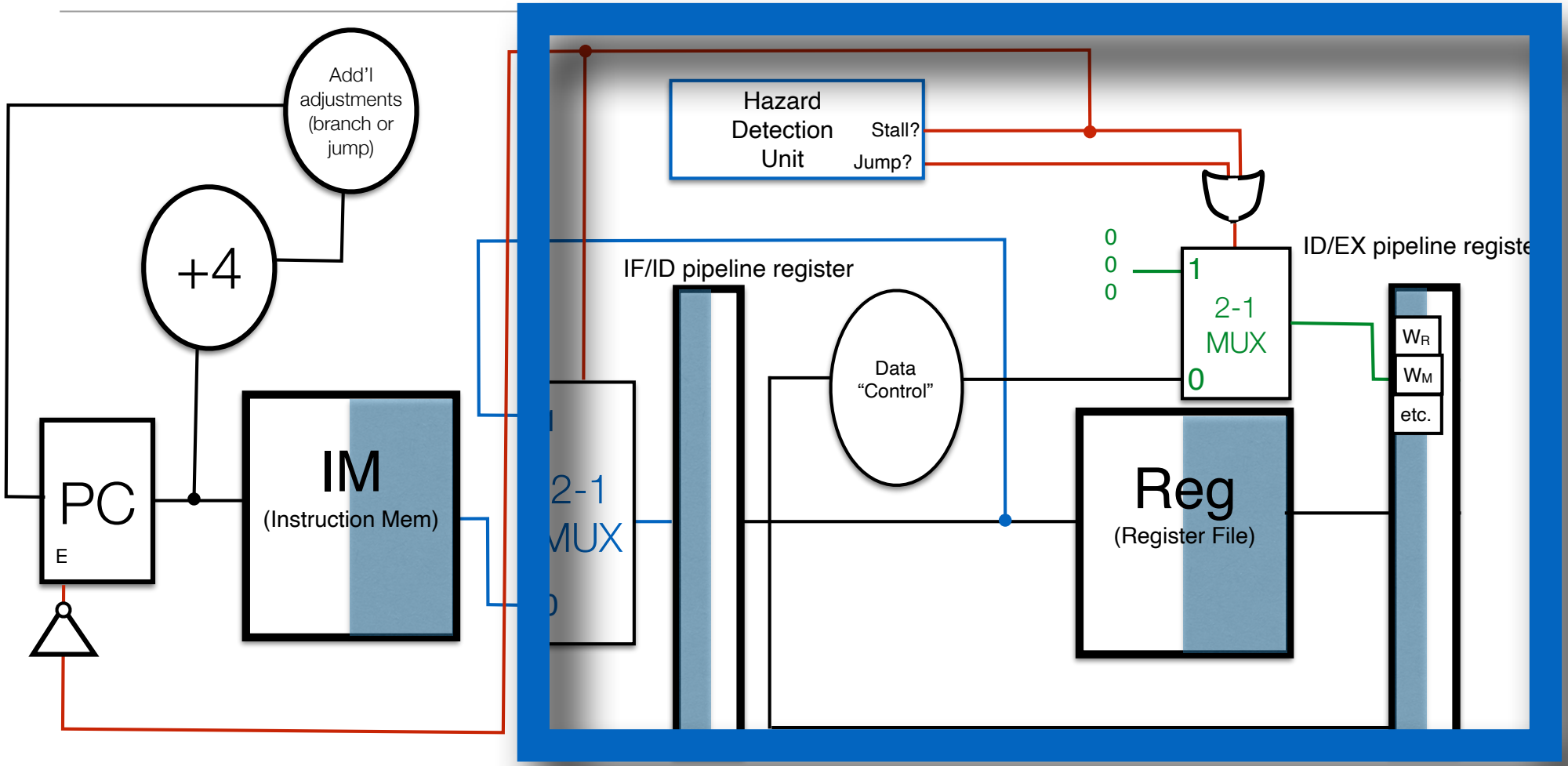
Recall: Add'l Circuitry could detect hazard



Also check the other read parameter (if R-type instruction)

- HDU already looks at op-code of instruction in EX stage
- Could have it detect when it's a jump instruction

Also recall: turn into NOP

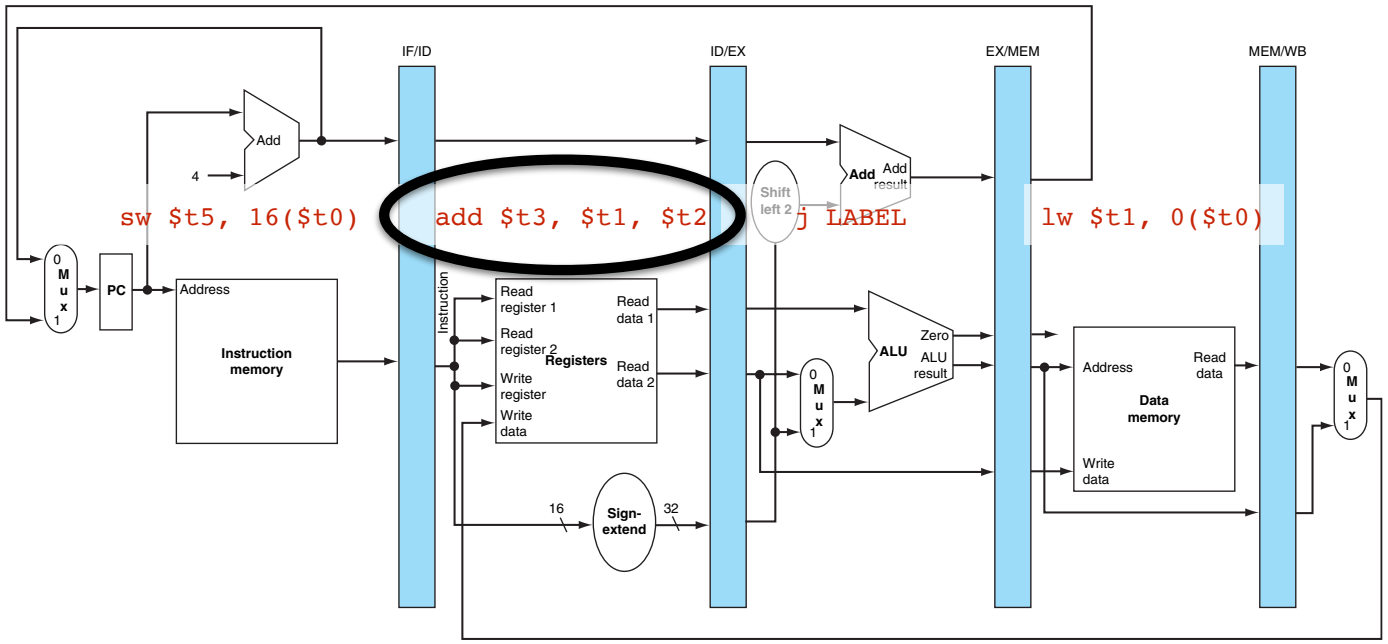


•

Jump Example

```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC



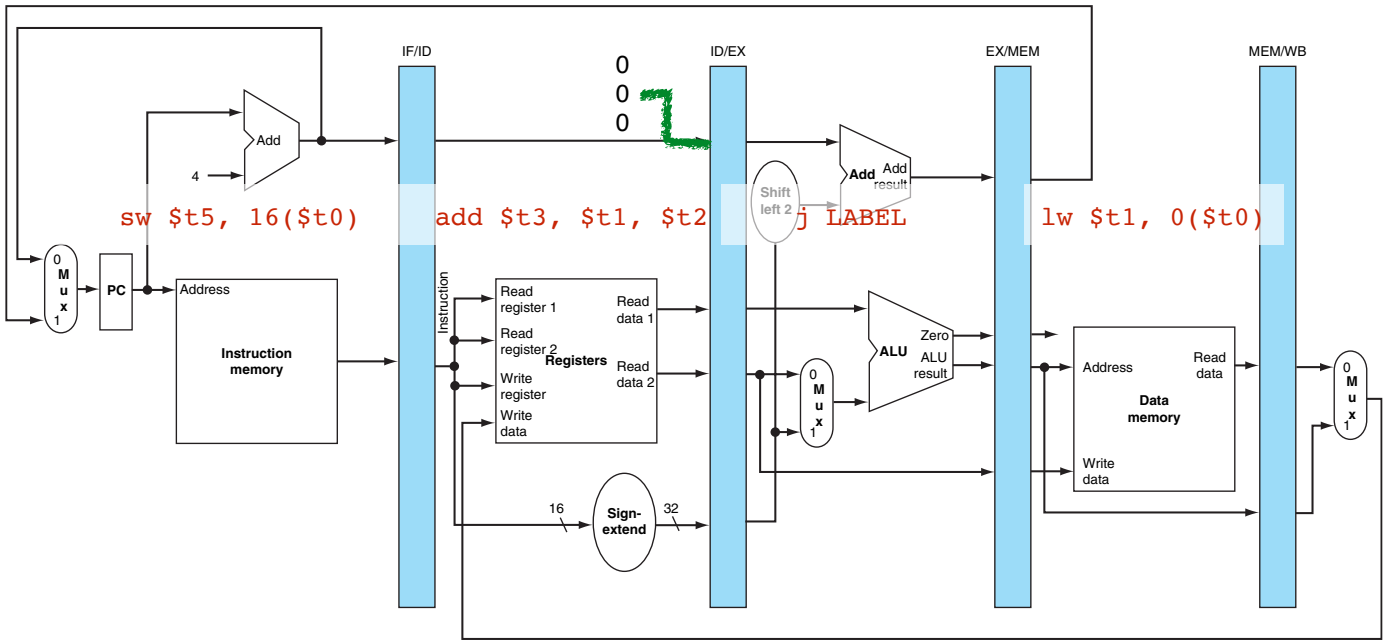
- j instruction done adjusting PC. Will be NOP in 3 remaining stages.

- Still have add instruction that doesn't belong in pipeline

Jump Example

```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC



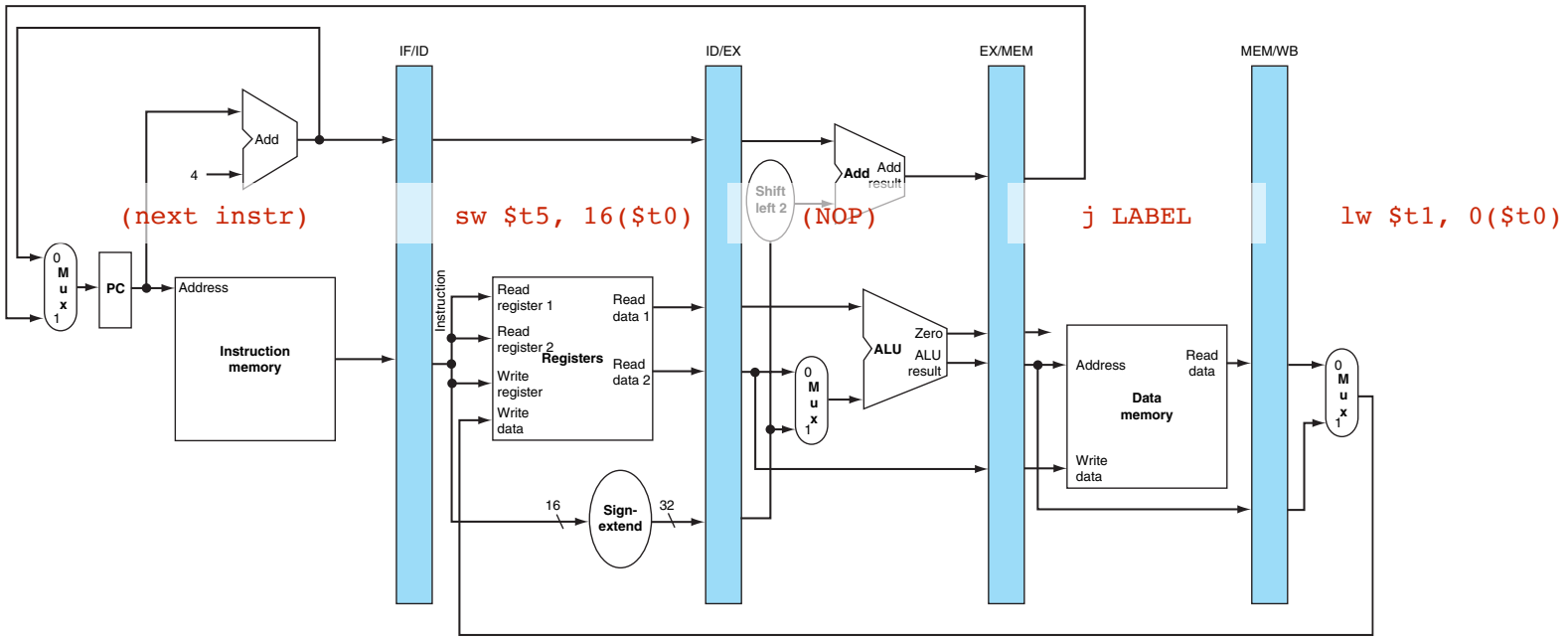
- j instruction done adjusting PC. Will be NOP in 3 remaining stages.

- “NOP” the unwanted instruction, no stalling necessary (just have a bubble)

Jump Example

```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC

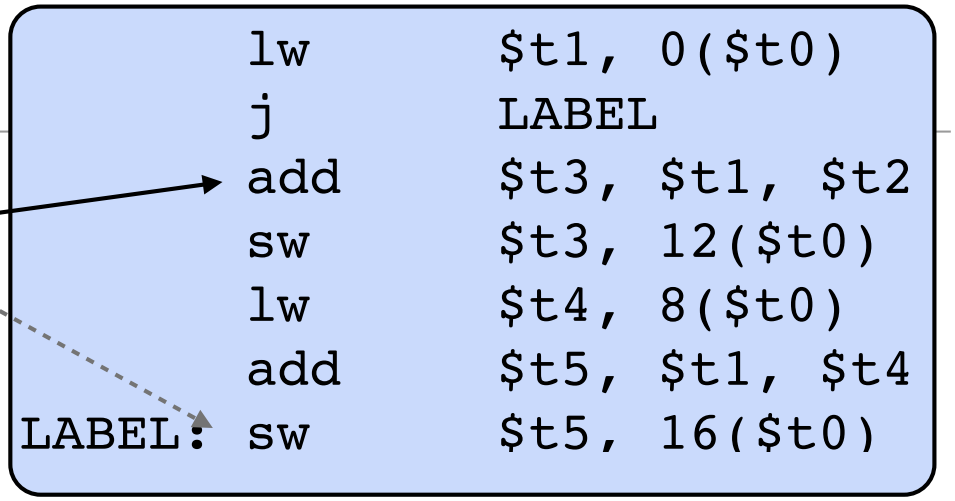


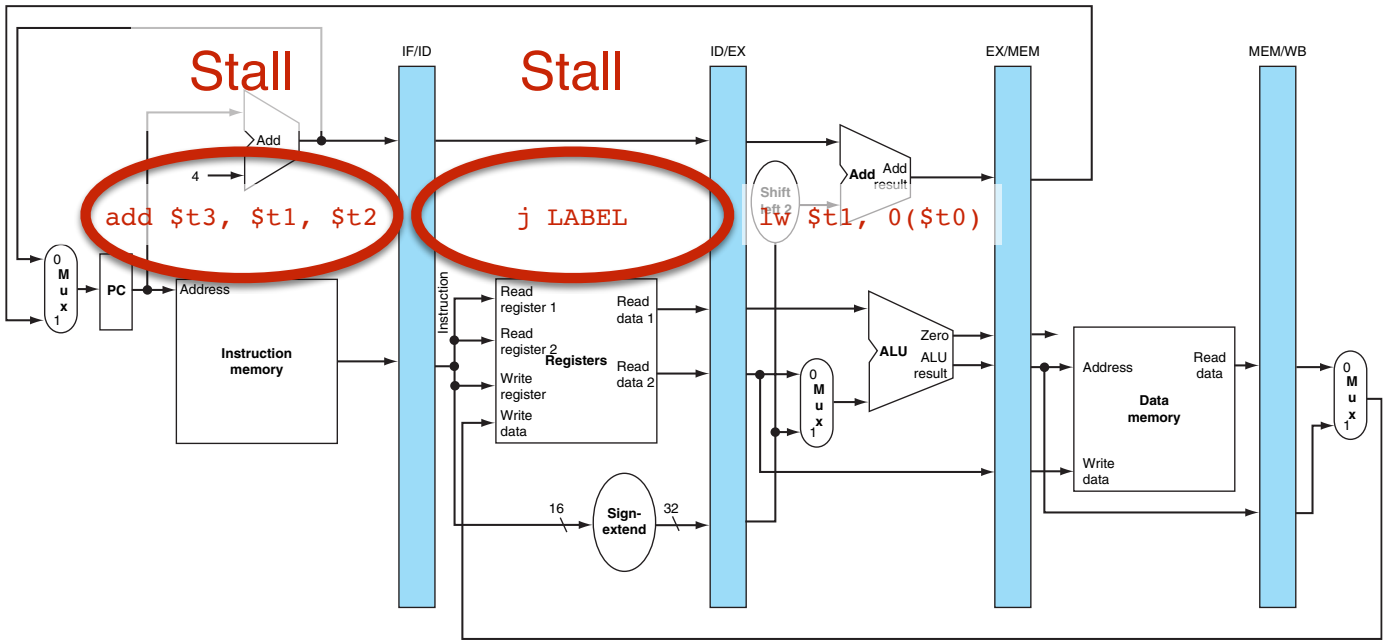
- j instruction done adjusting PC. Will be NOP in 3 remaining stages.

- “NOP” the unwanted instruction, no stalling necessary (just have a bubble)

Alternate Approach

Stall Pipeline

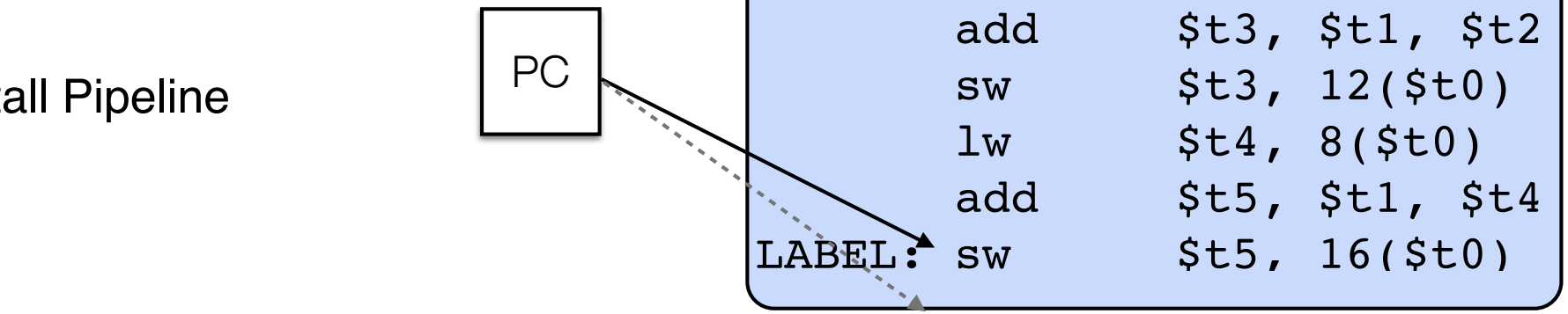


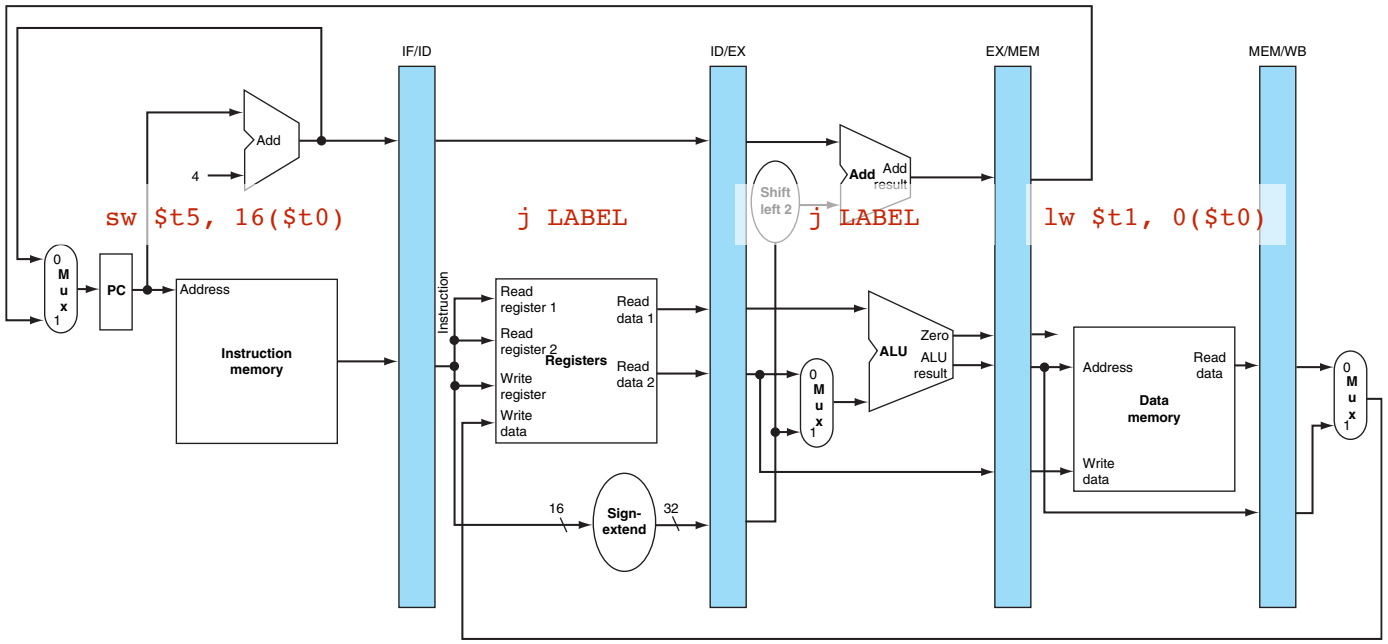


- Now j is being evaluated and PC set to address of LABEL, but wrong instruction already pulled in

Alternate Approach

Stall Pipeline





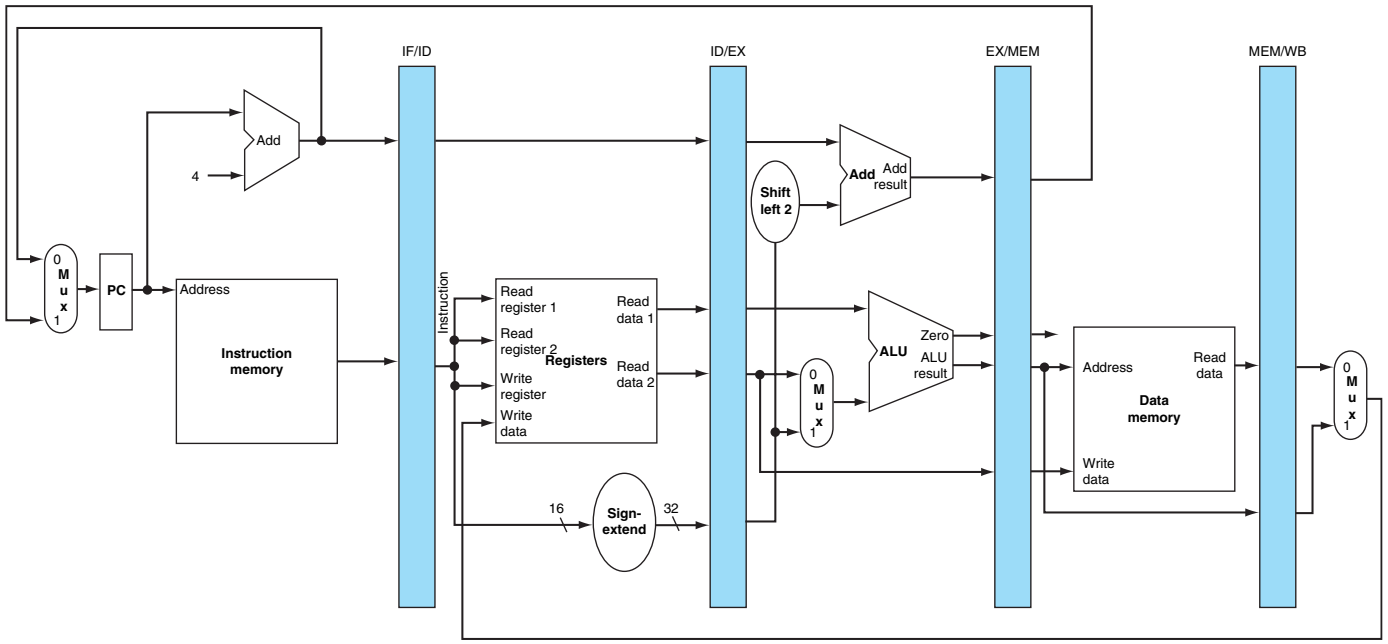
- Stalling the first two stages: j replicated moves to 3rd stage. When see replicated instruction, don't stall anymore. wrong instruction (add) replaced in pipeline. This approach offers no benefit over previous (still one wasted cycle)

Ctrl Hazard Resolution for Branches

Jump Example becomes…

PC

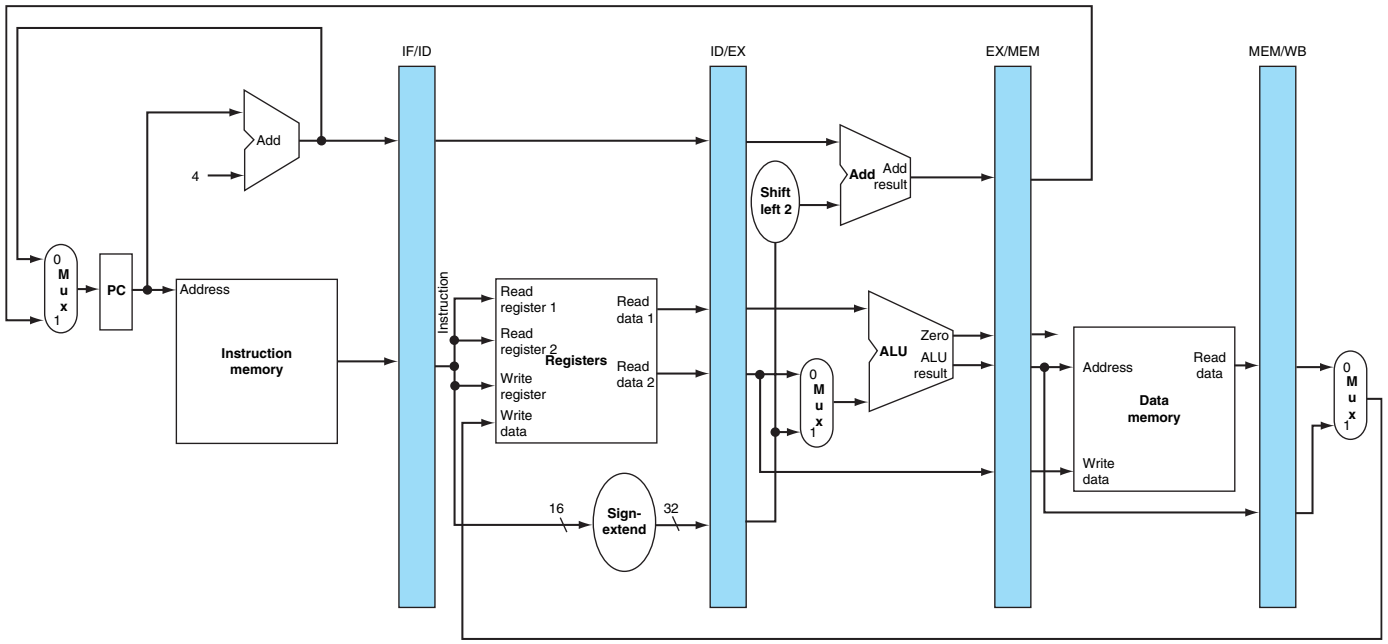
```
        lw      $t1, 0($t0)
        j       LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



Branch Example

PC

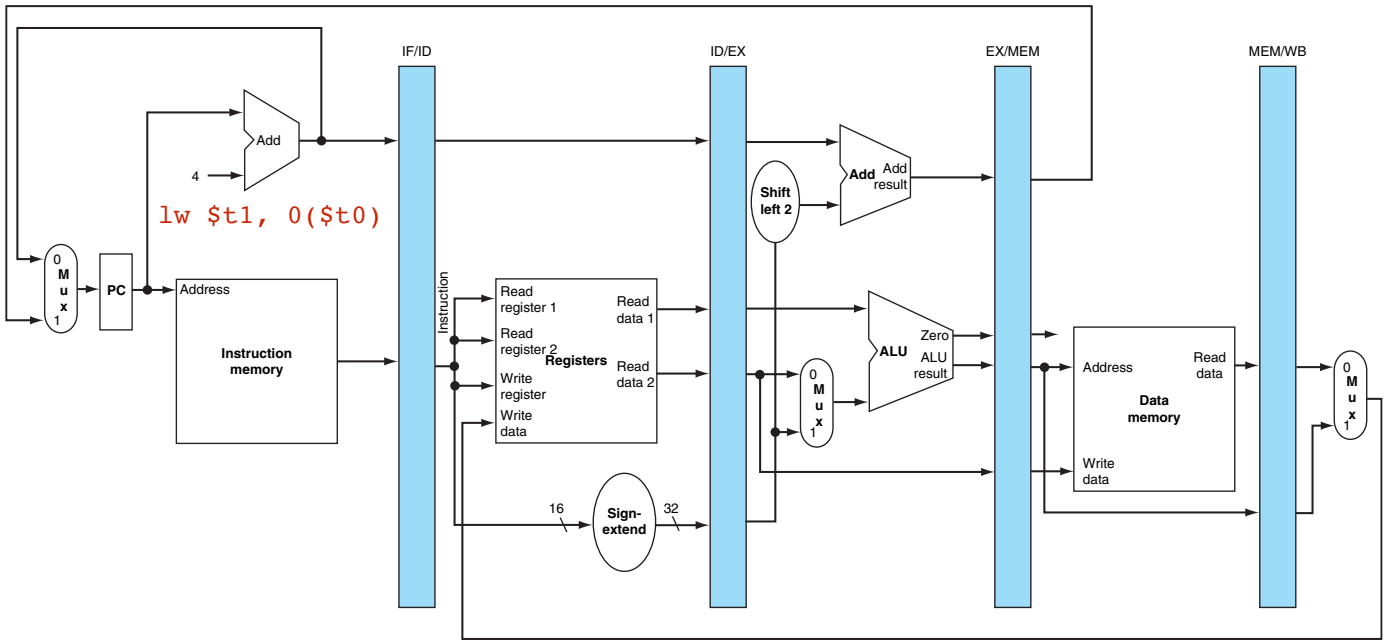
```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



Branch Example

PC

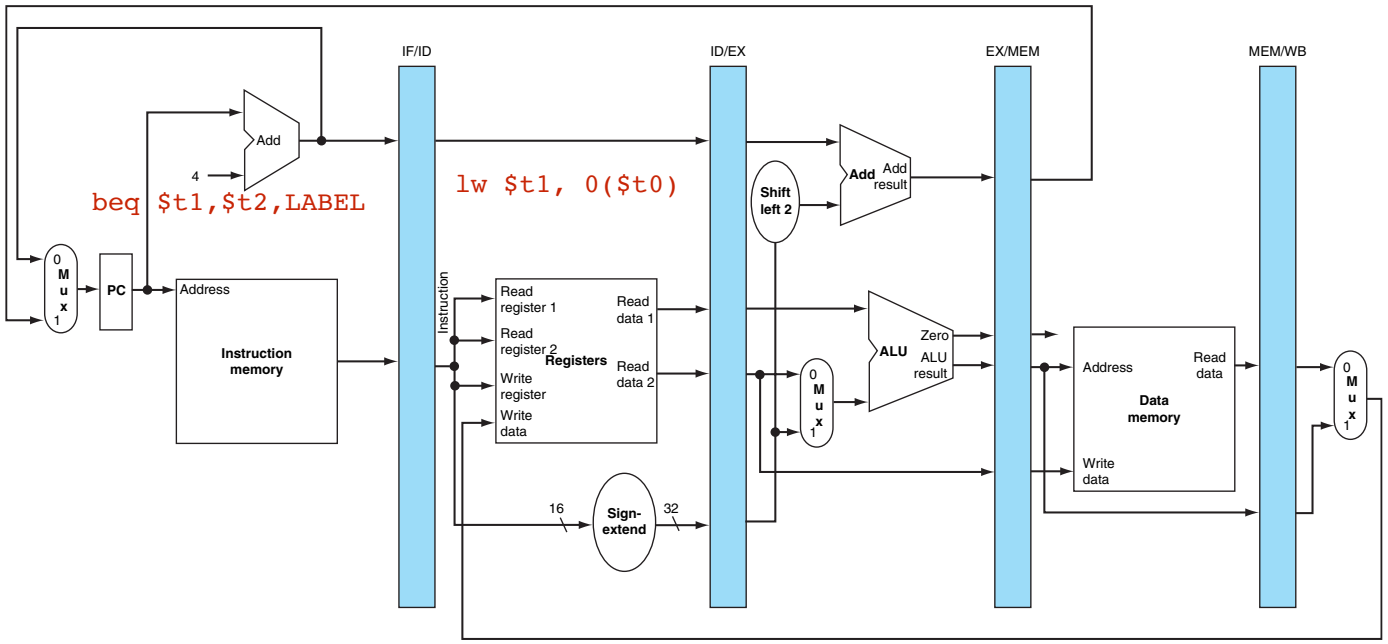
```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



Branch Example

```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC

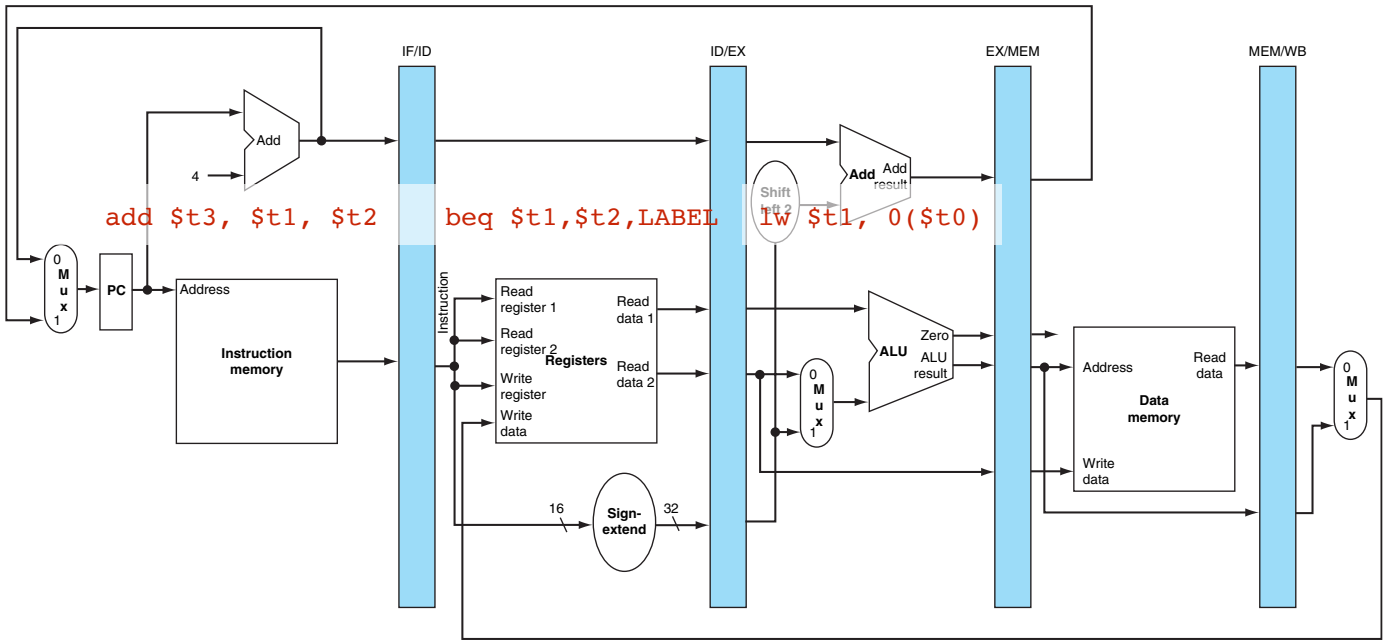


- Even though beq now in pipeline, it hasn't been evaluated yet. PC+4 instruction will be brought into pipeline next clock cycle

Branch Example

```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```

PC



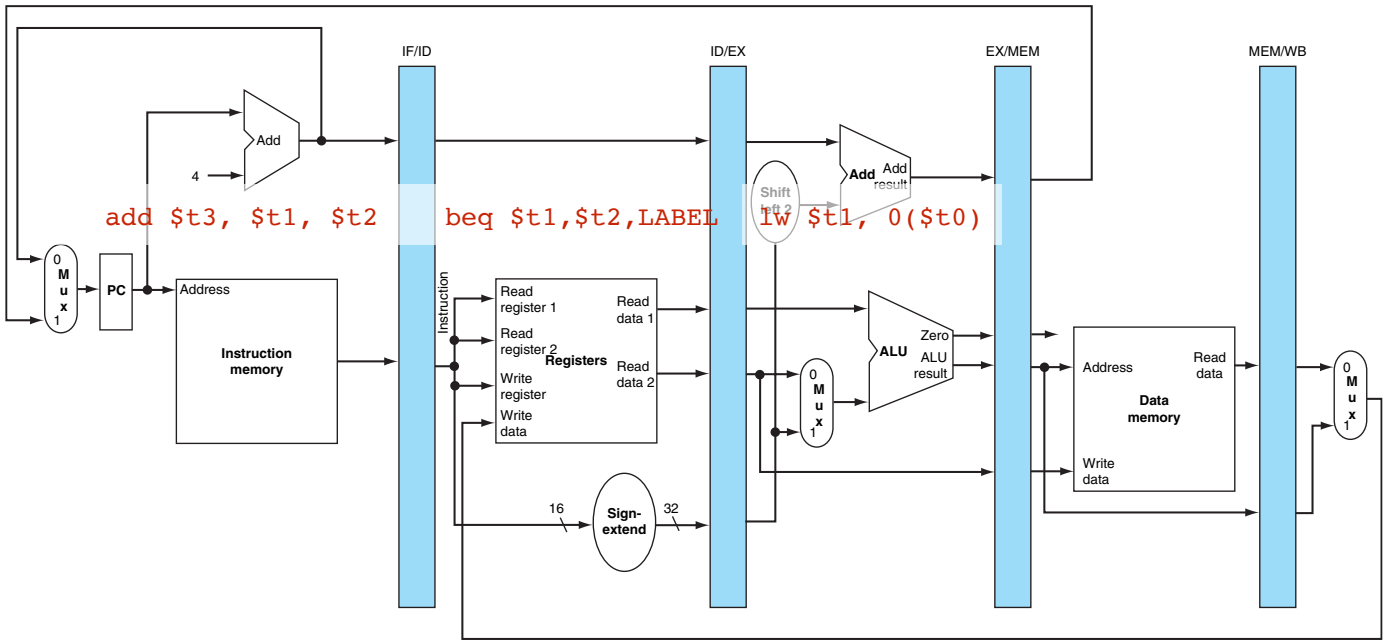
- We now know it's a branch instruction, but conditional still not evaluated. What to do?

Option A

Stall pipeline until conditional resolved

PC

```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



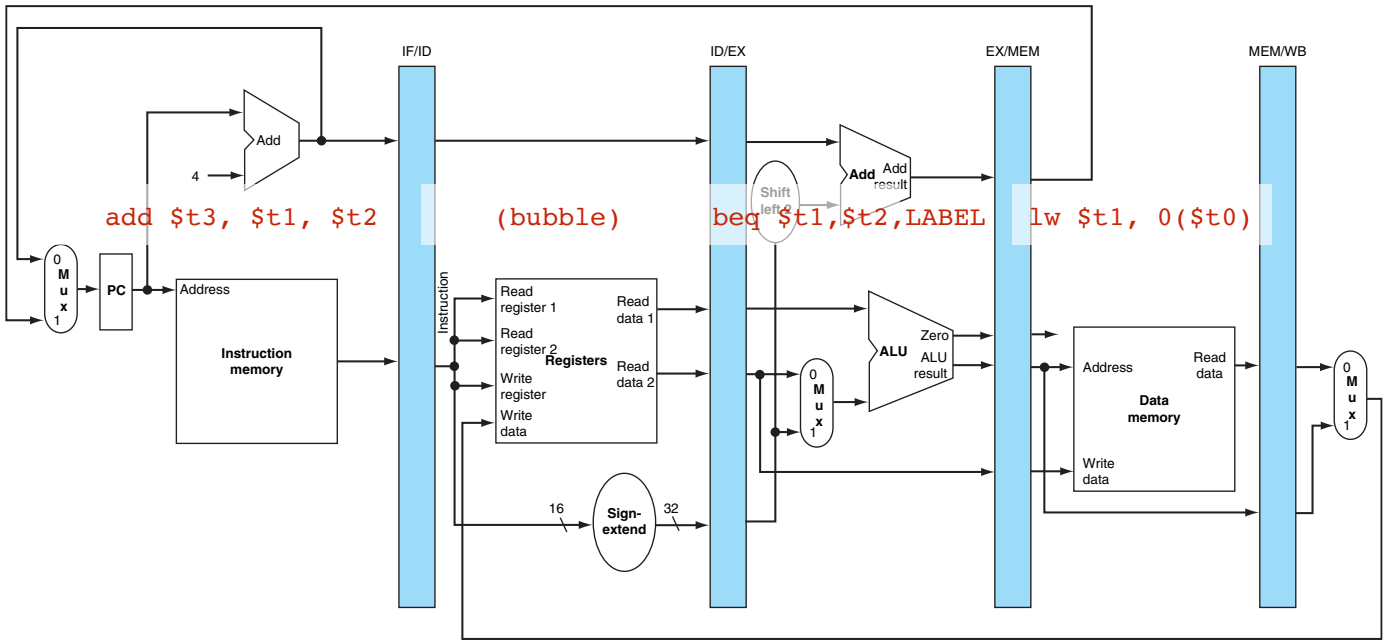
- We now know it's a branch instruction, but conditional still not evaluated. What to do?

Option A

Stall pipeline until conditional resolved

PC

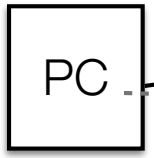
```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



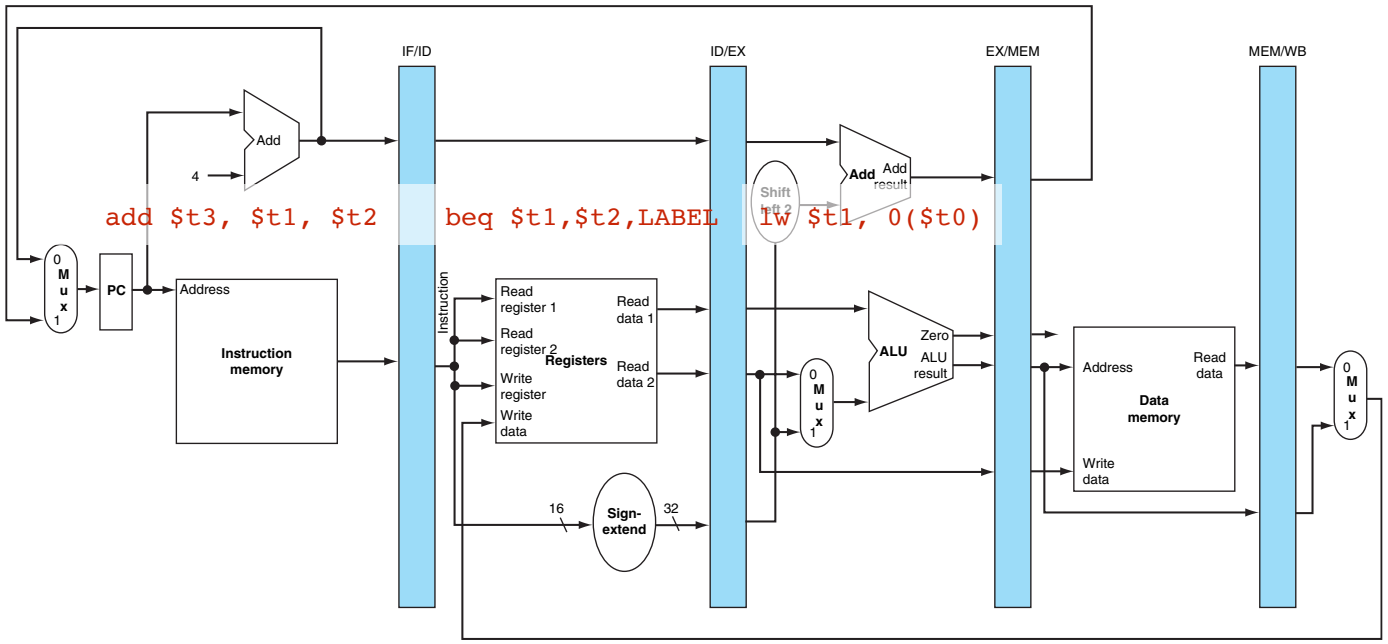
- Conditional True: adjust PC, “flush” pipeline and resume
- Conditional False: resume pipeline

Option B

Take a guess:
Branch Prediction




```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



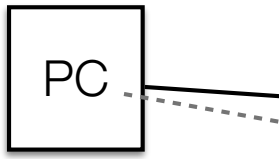
- Guess result of conditional (e.g., guess false).

Option B

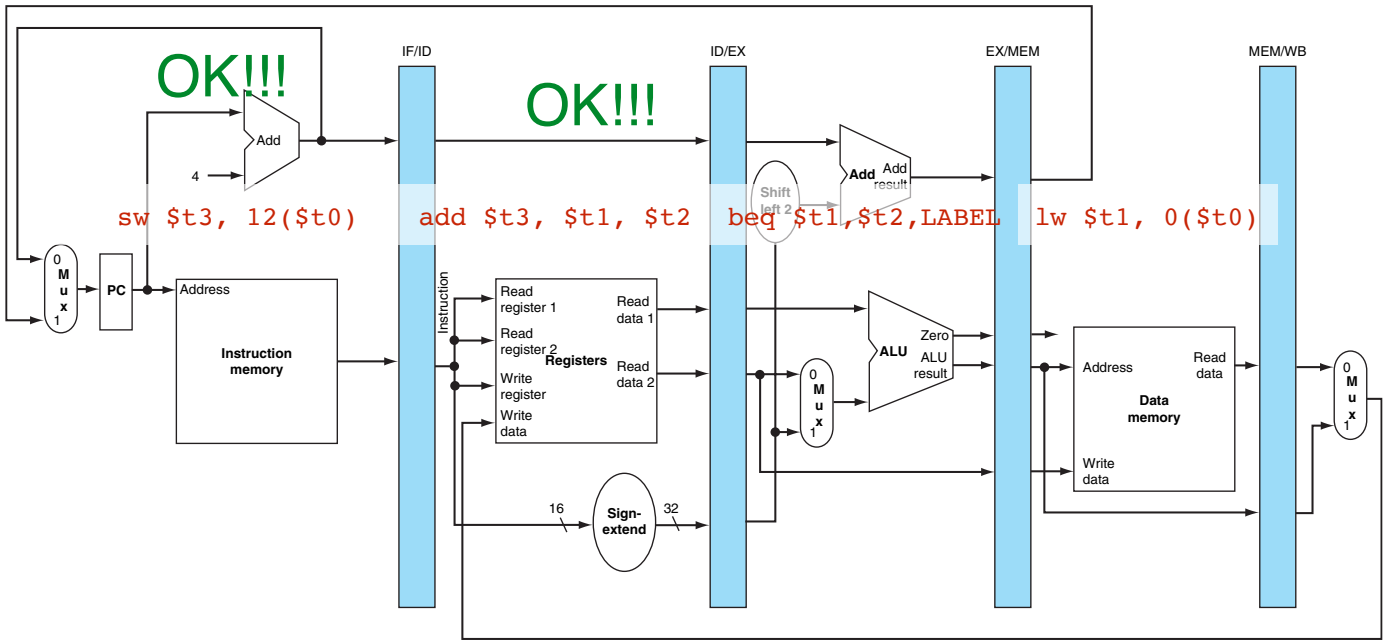
Take a guess:

Branch Prediction

Case 1: guessed right




```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



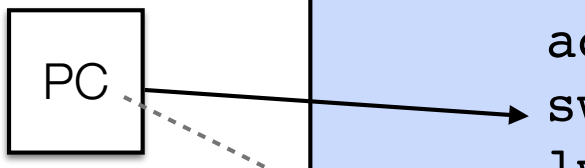
- Guess result of conditional (e.g., guess false).
- Guessed right, no stall at all

Option B

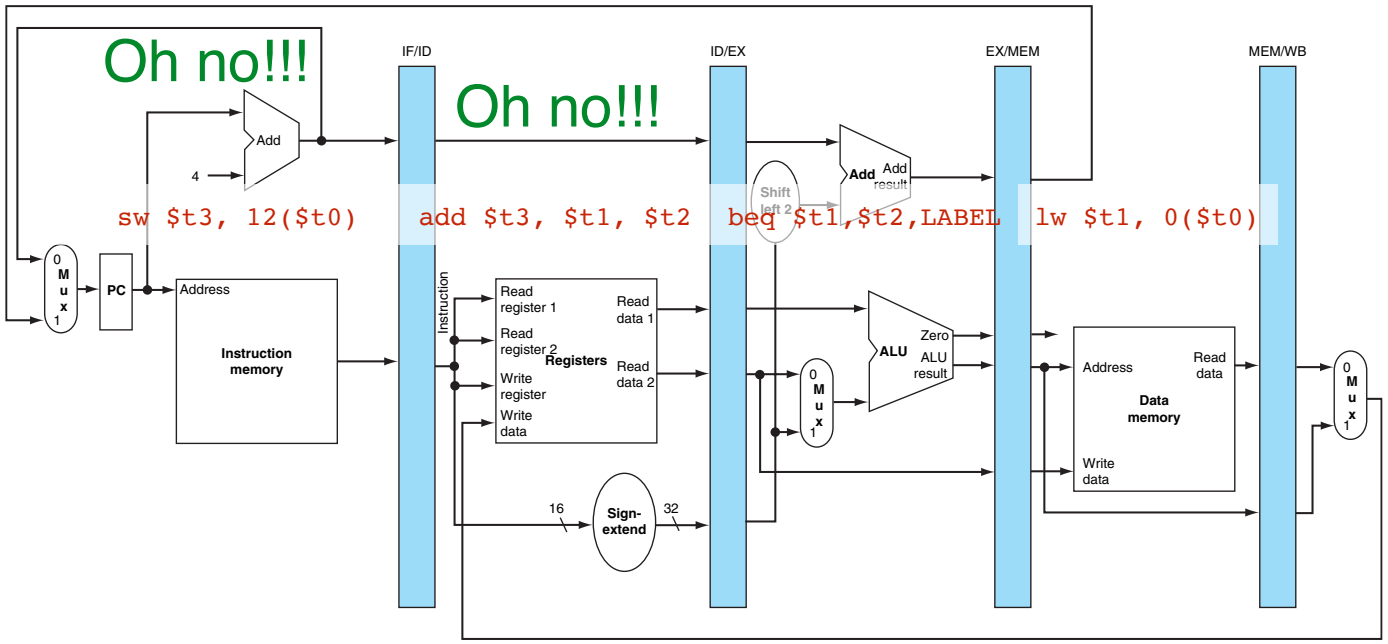
Take a guess:

Branch Prediction

Case 2: guessed wrong



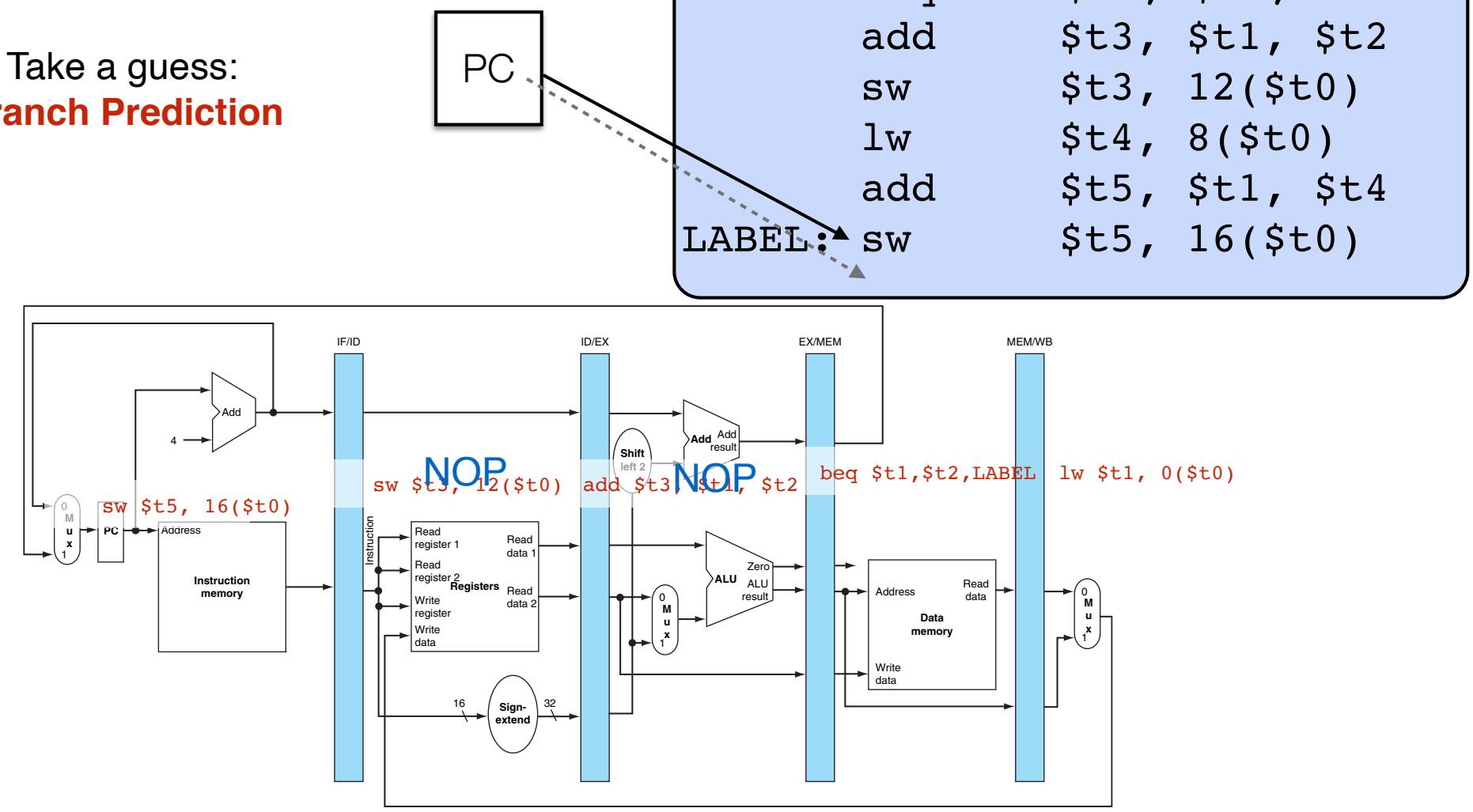

```
        lw      $t1, 0($t0)
        beq     $t1, $t2, LABEL
        add     $t3, $t1, $t2
        sw      $t3, 12($t0)
        lw      $t4, 8($t0)
        add     $t5, $t1, $t4
LABEL:  sw      $t5, 16($t0)
```



- Guess result of conditional (e.g., guess false).
- Guessed wrong, then set PC to LABEL address, 2 instructions don't belong in pipeline

Option B

Take a guess:
Branch Prediction



- Guess result of conditional (e.g., guess false).
- Guessed wrong, NOP two instructions erroneously put in pipeline: no damage

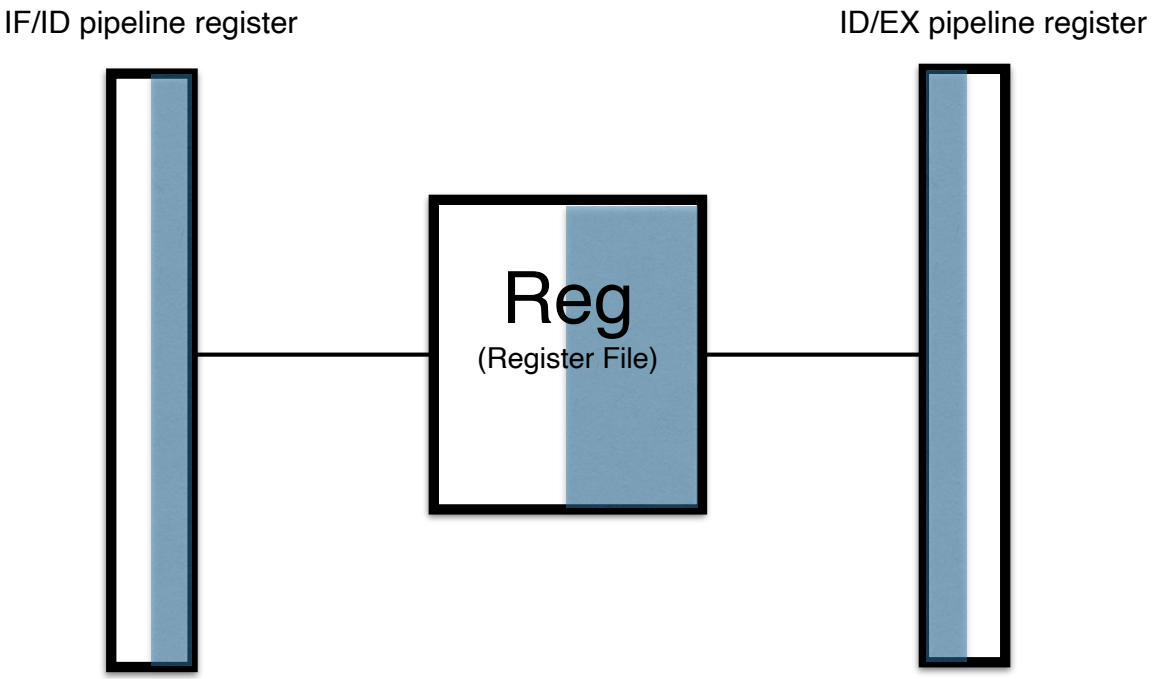
Move Conditional Check Forward

Moving Conditional Check Forward

- In Single Cycle Arch, used ALU to do branch comparison
  - Perform subtraction, check if equalled 0
- Mapped Single Cycle Arch to Pipeline Arch
  - Conditional Check in EX stage using ALU
- Consequence: Conditional evaluated in third stage
- Idea:
  - don’t use the ALU to perform conditional check, but include hardware to perform check at end of ID stage
  - on branch prediction failure, only stall once instead of twice

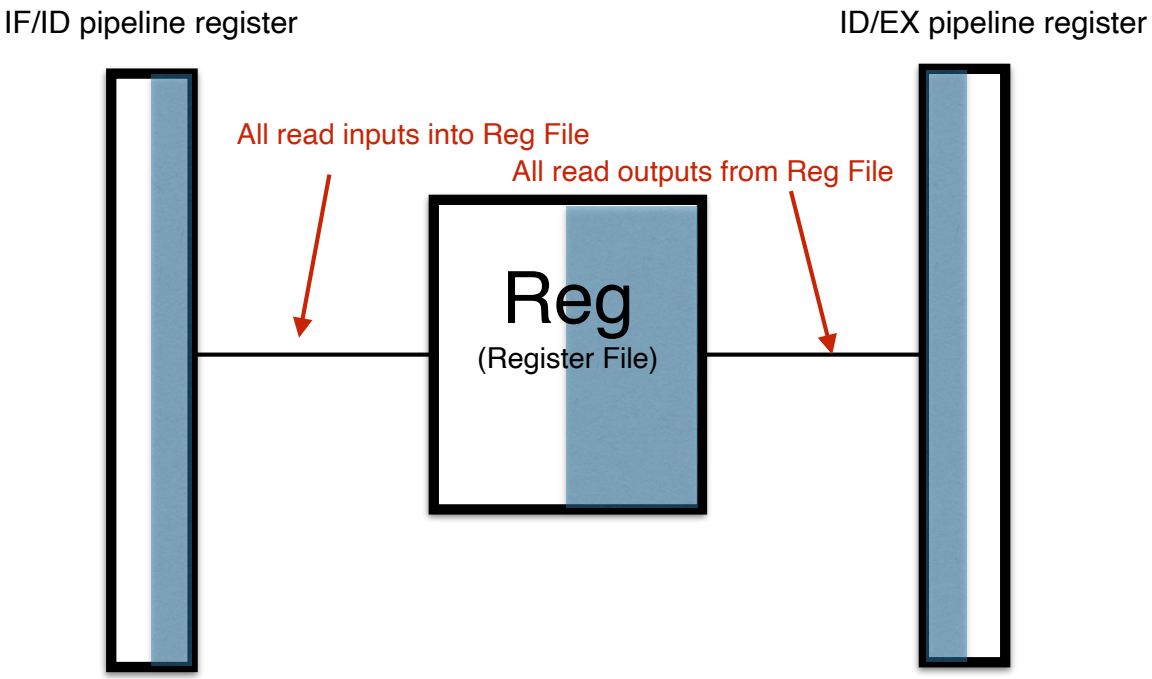
Reg File: Read 2nd half clock cycle

Instruction Decode (ID) Stage



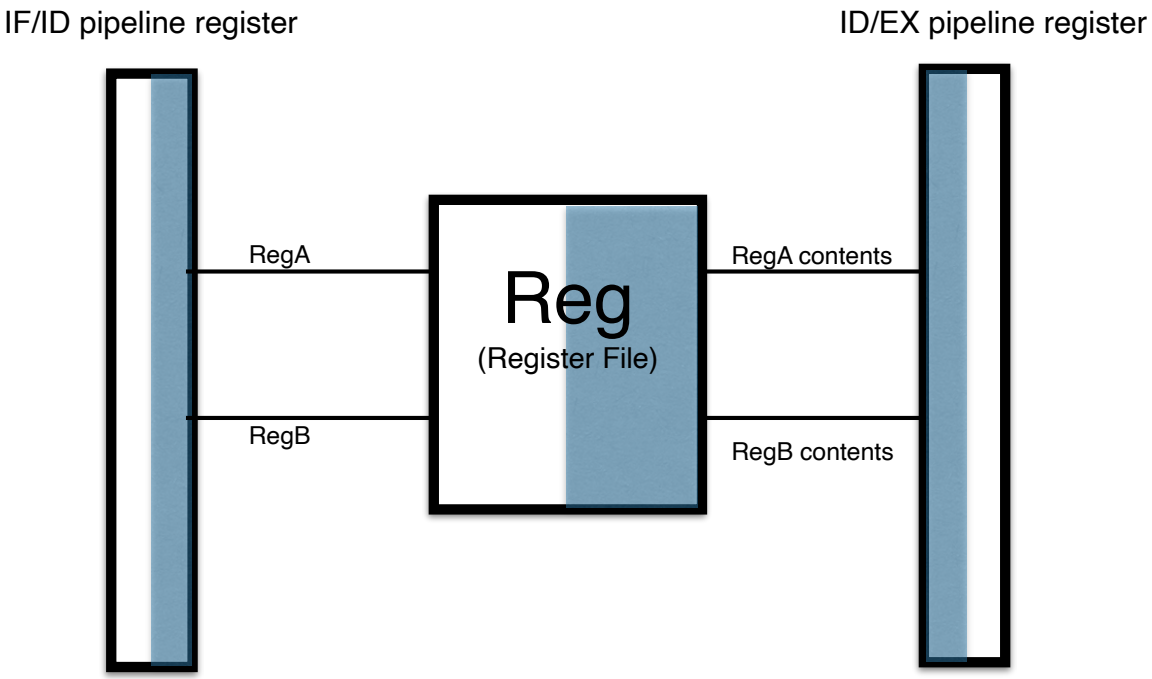
Reg File: Read 2nd half clock cycle

Instruction Decode (ID) Stage



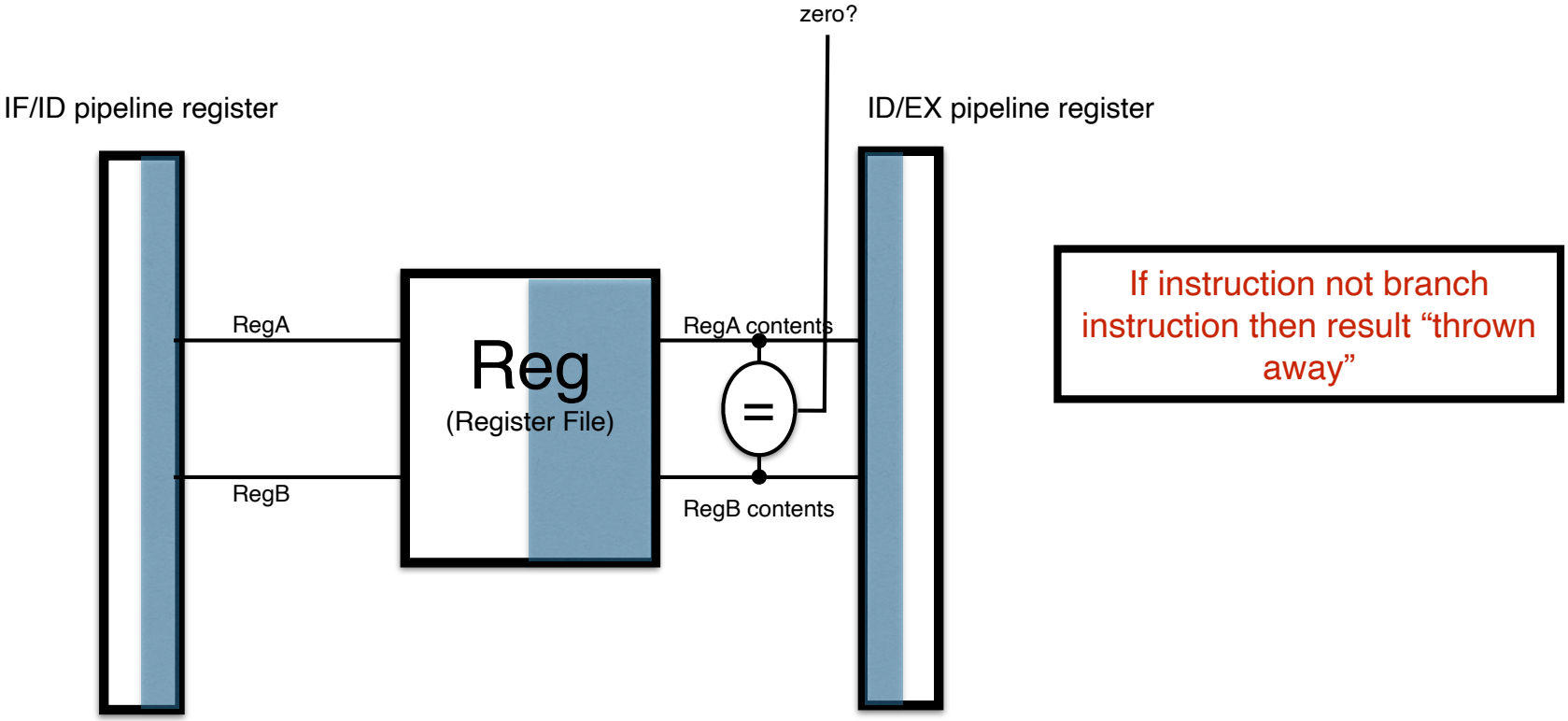
Reg File: Read 2nd half clock cycle

Instruction Decode (ID) Stage



Reg File: Read 2nd half clock cycle

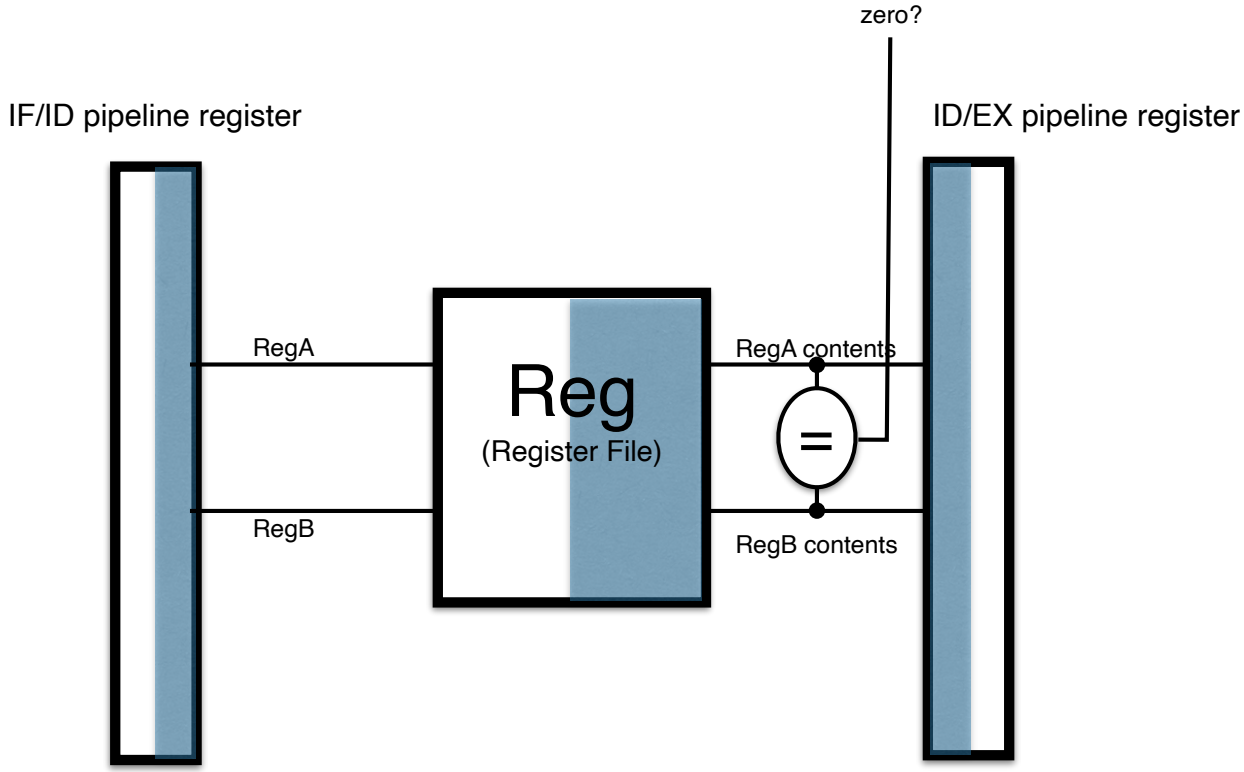
Instruction Decode (ID) Stage



- Equality Circuit: XOR and verify all 0's sufficient for checking conditional for beq, one
- Checking equality in ID stage reduces stall for branch by 1 clock cycle
- Only "downside": cannot use data forwarding if some instruction further in pipeline modifying RegA or RegB

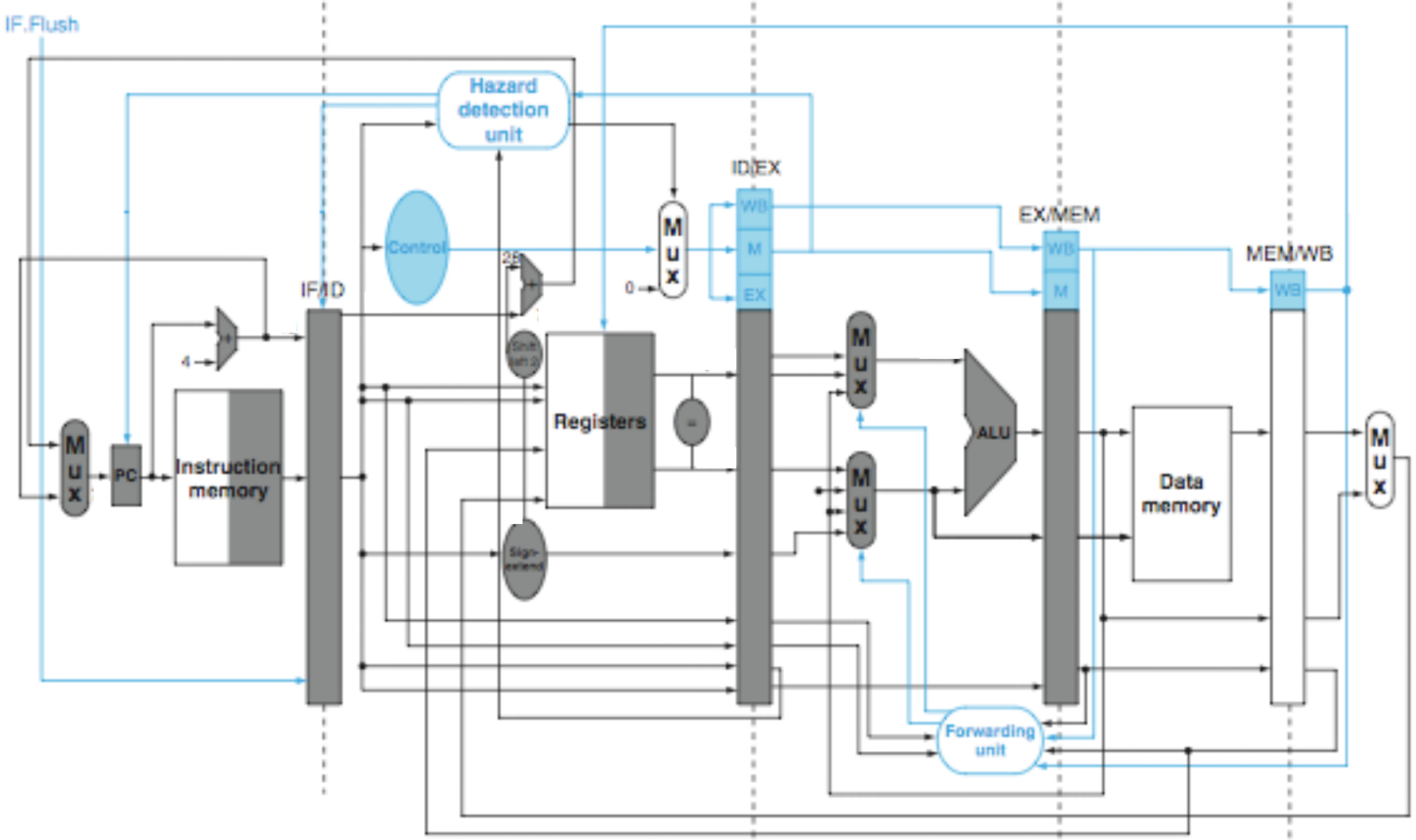
Remember: No blt, blte (only pseudo-instructions)

Instruction Decode (ID) Stage

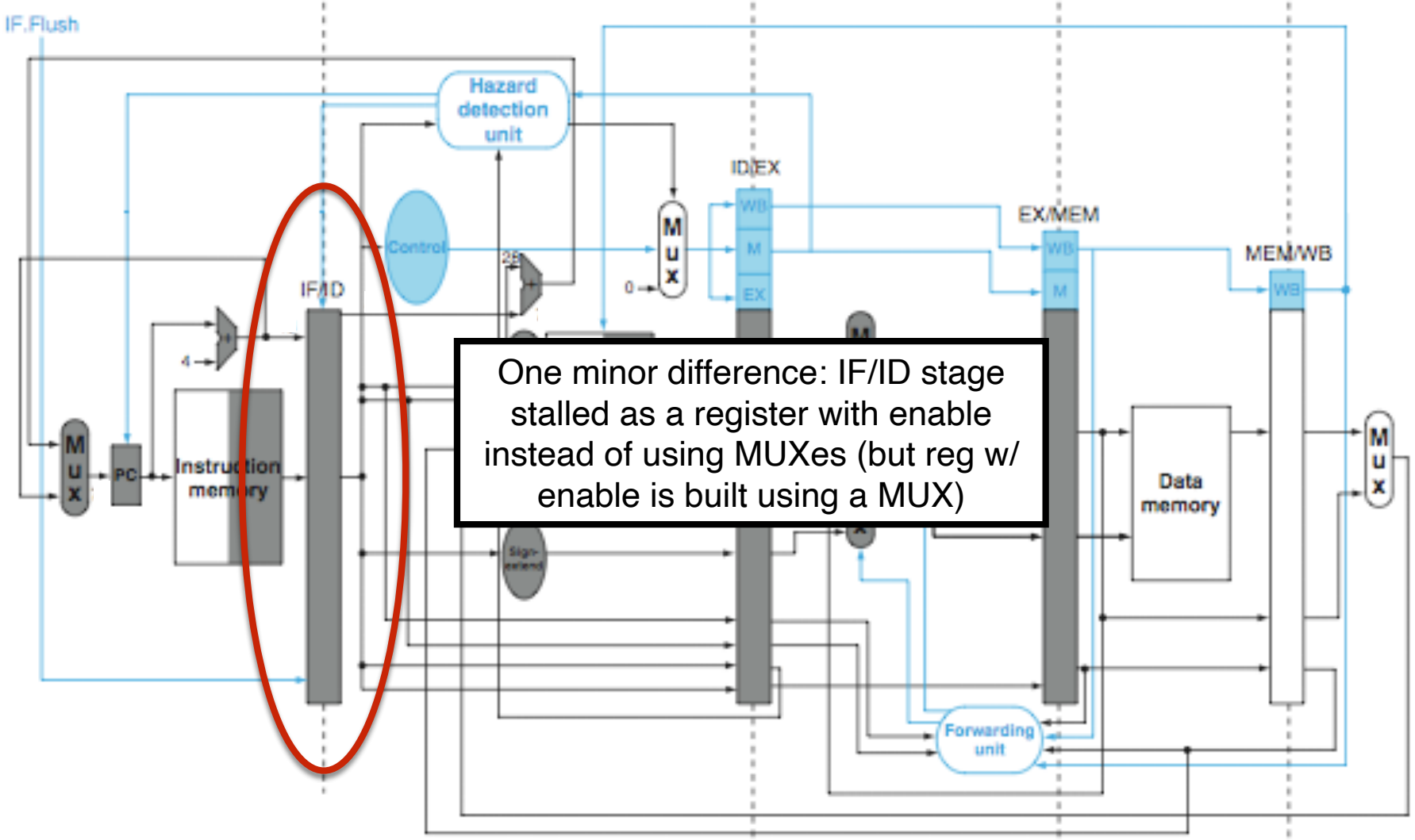


- Remember that we said that “blte” took too much time - this branch optimization is why…
- Just want a very quick comparator here to test for equality (XOR): circuit to check less than has logarithmic depth

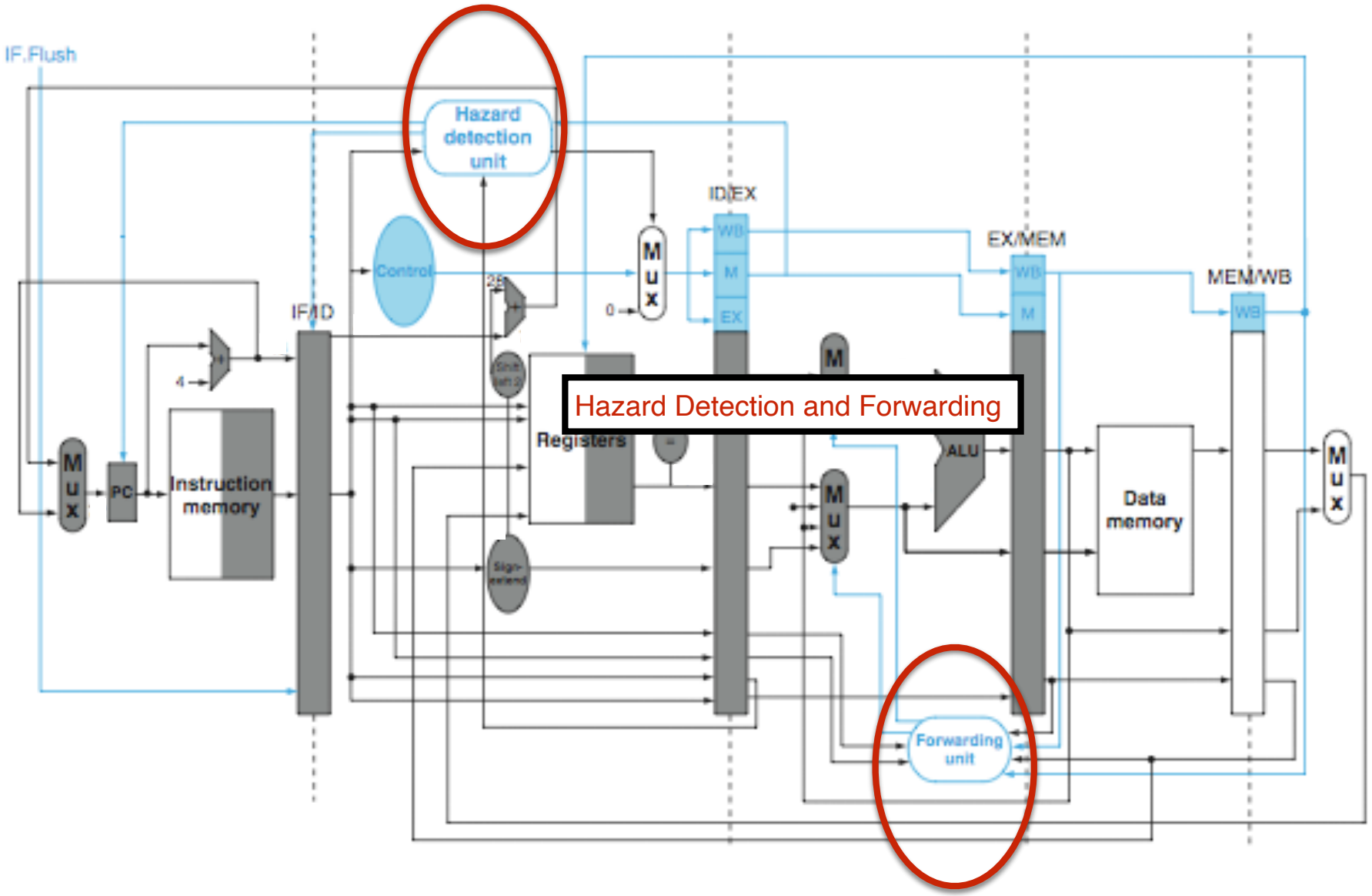
The Book's putting it together (no jump instr.)



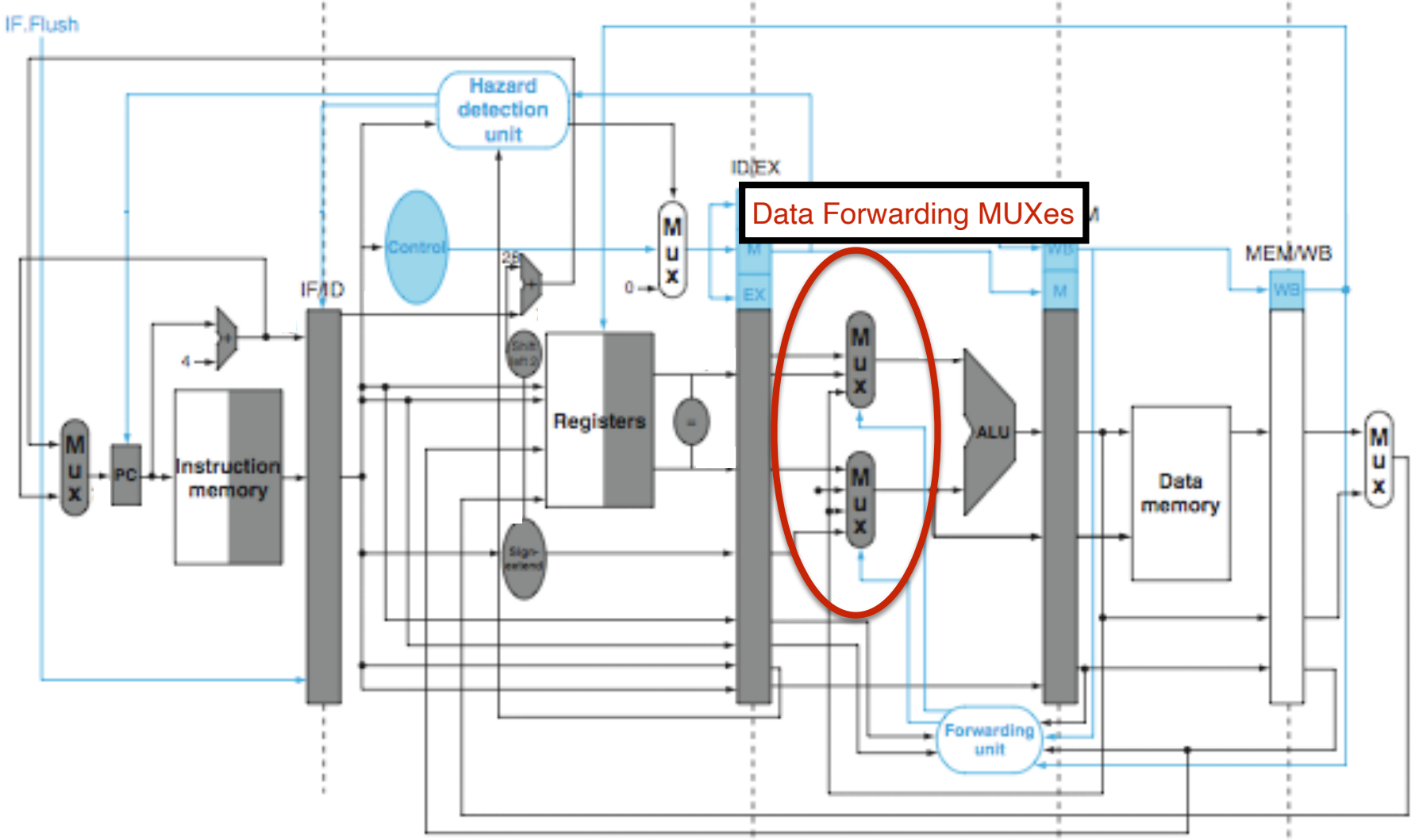
The Book's putting it together (no jump instr.)



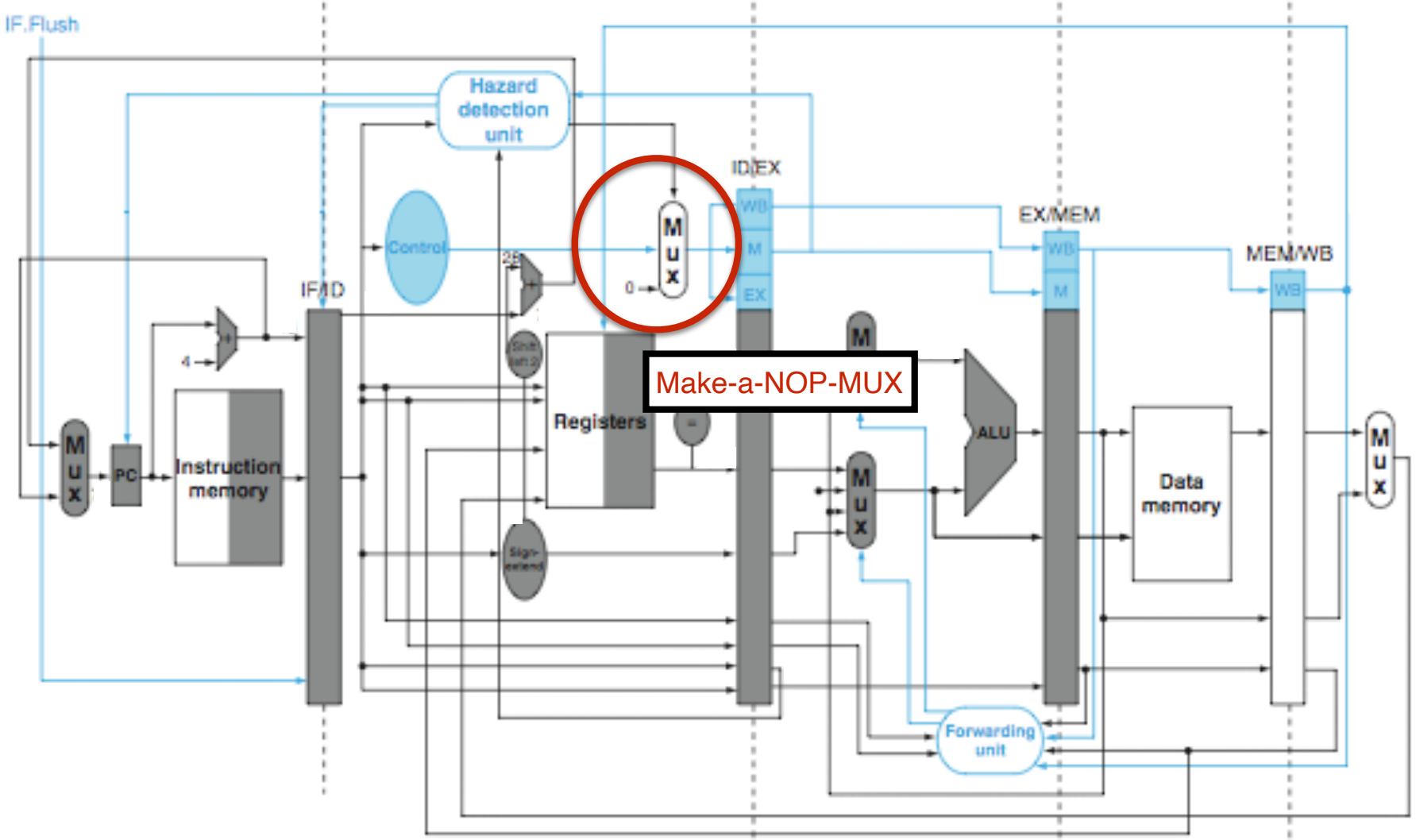
The Book's putting it together (no jump instr.)



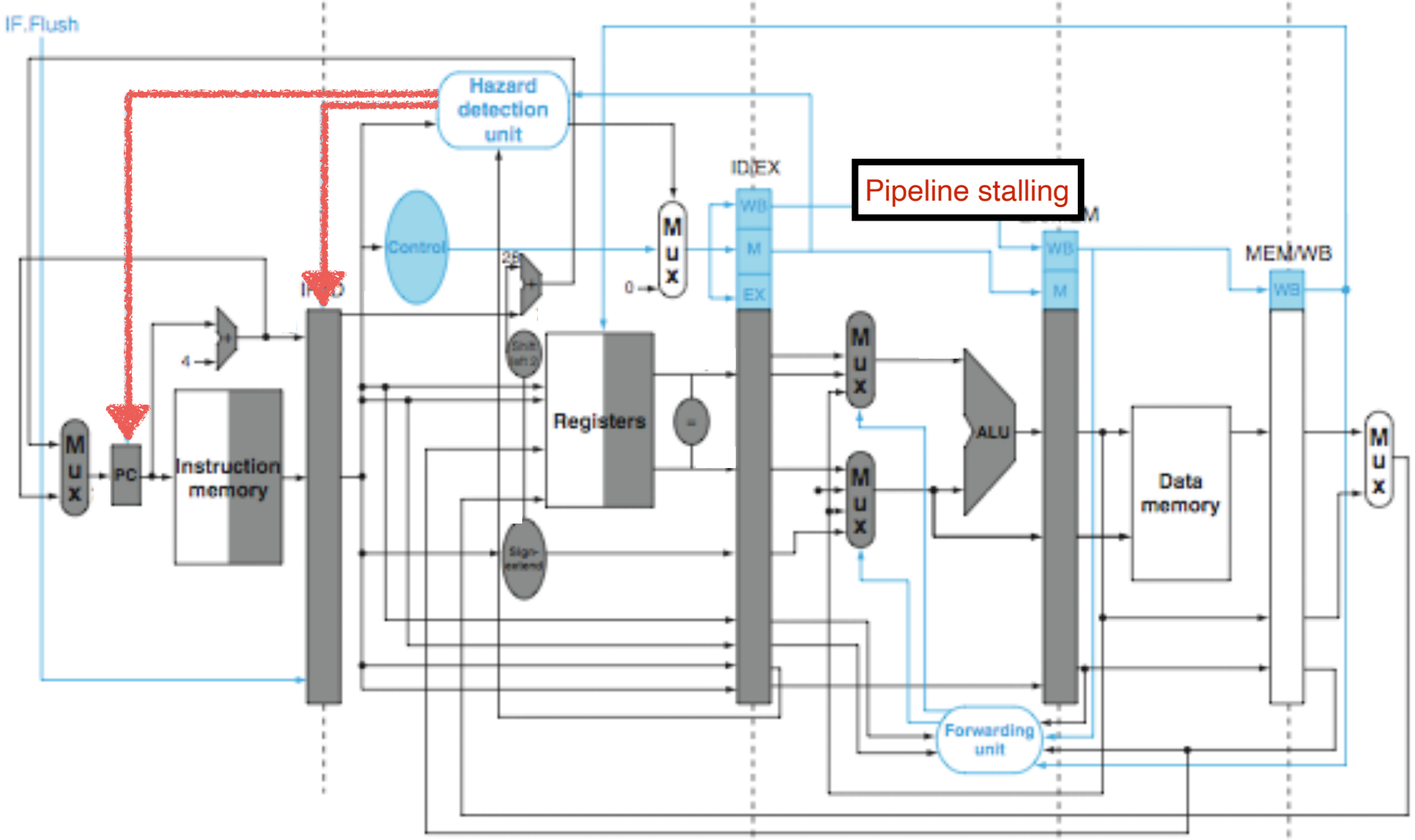
The Book's putting it together (no jump instr.)



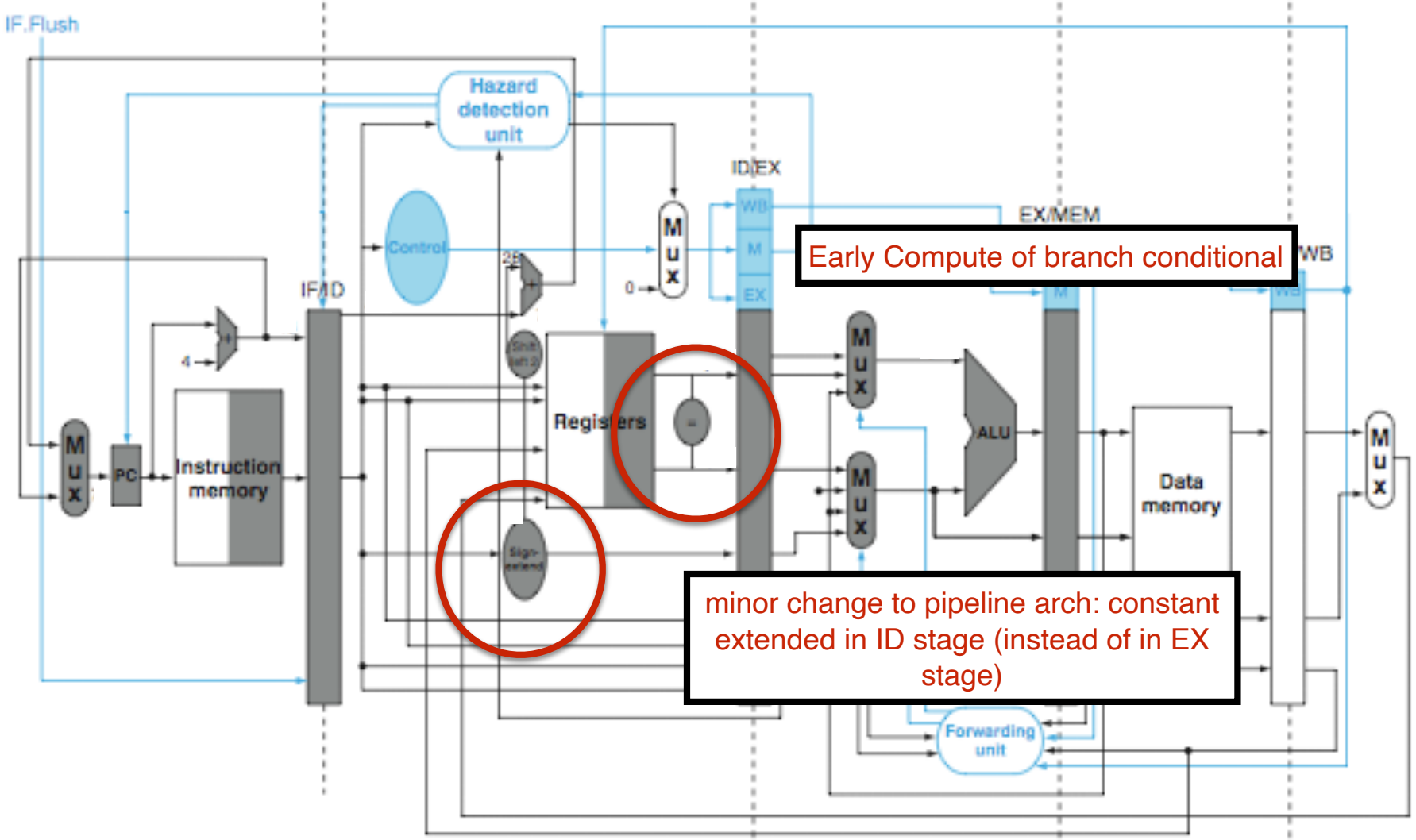
The Book's putting it together (no jump instr.)



The Book's putting it together (no jump instr.)

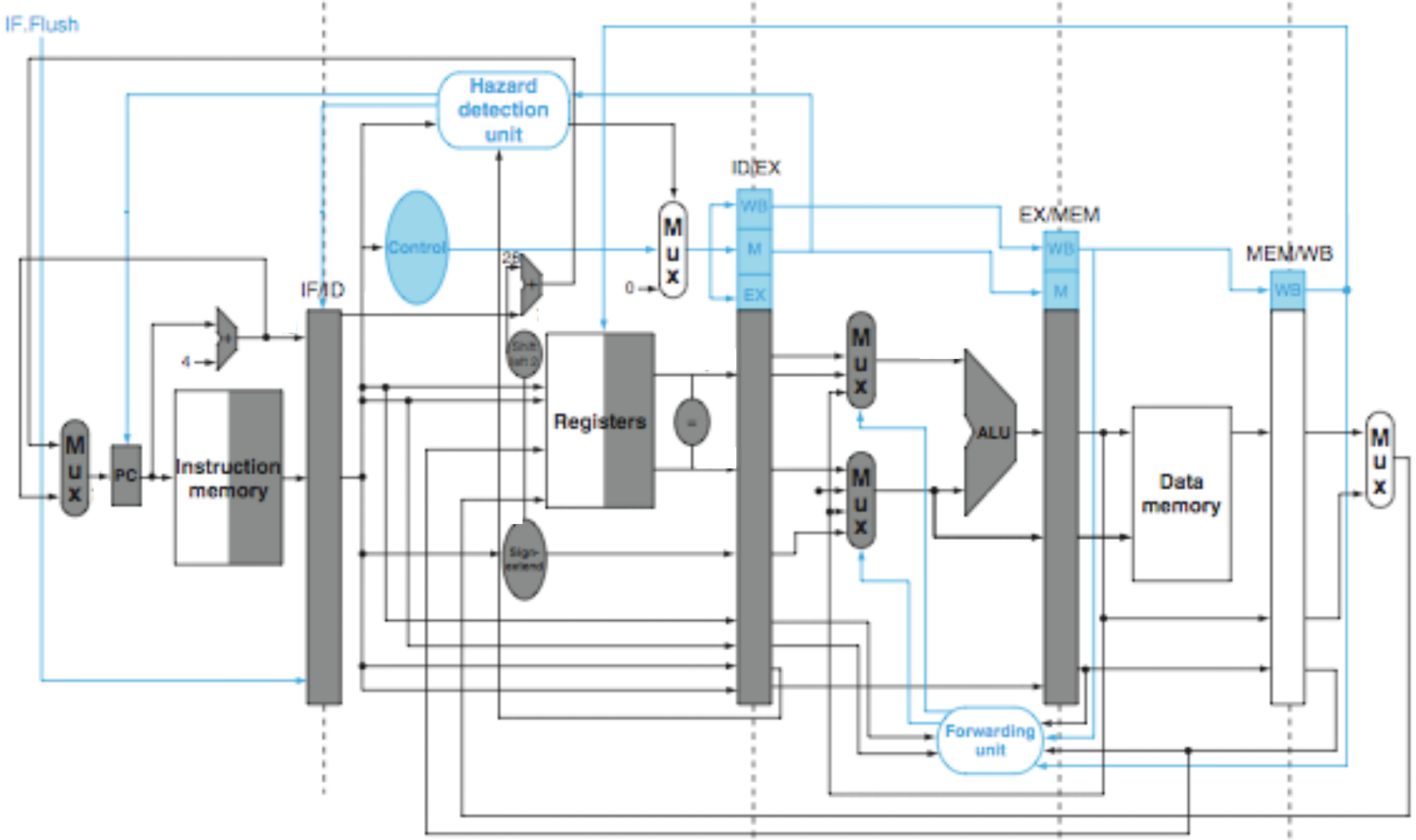


The Book's putting it together (no jump instr.)

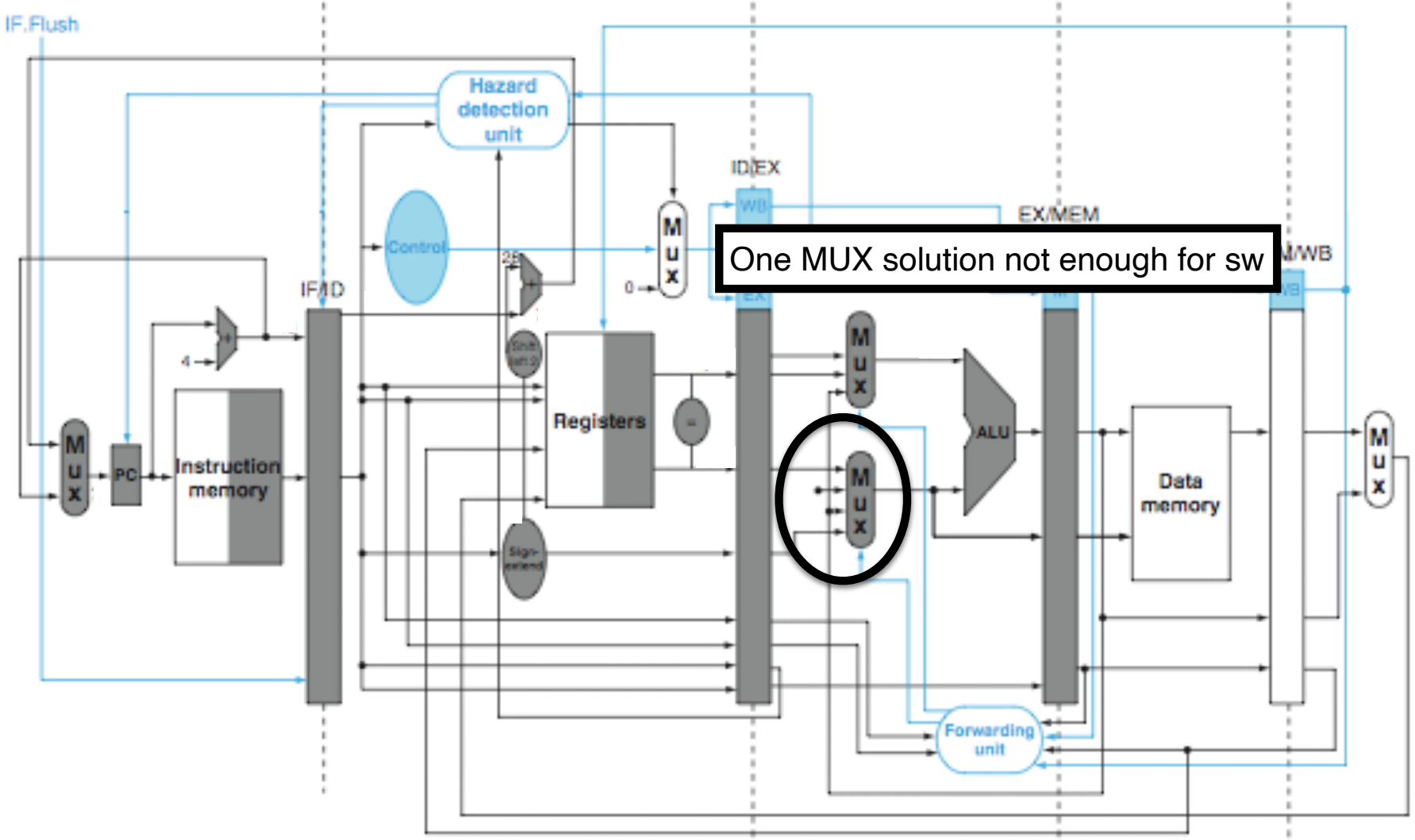


Book Oops!

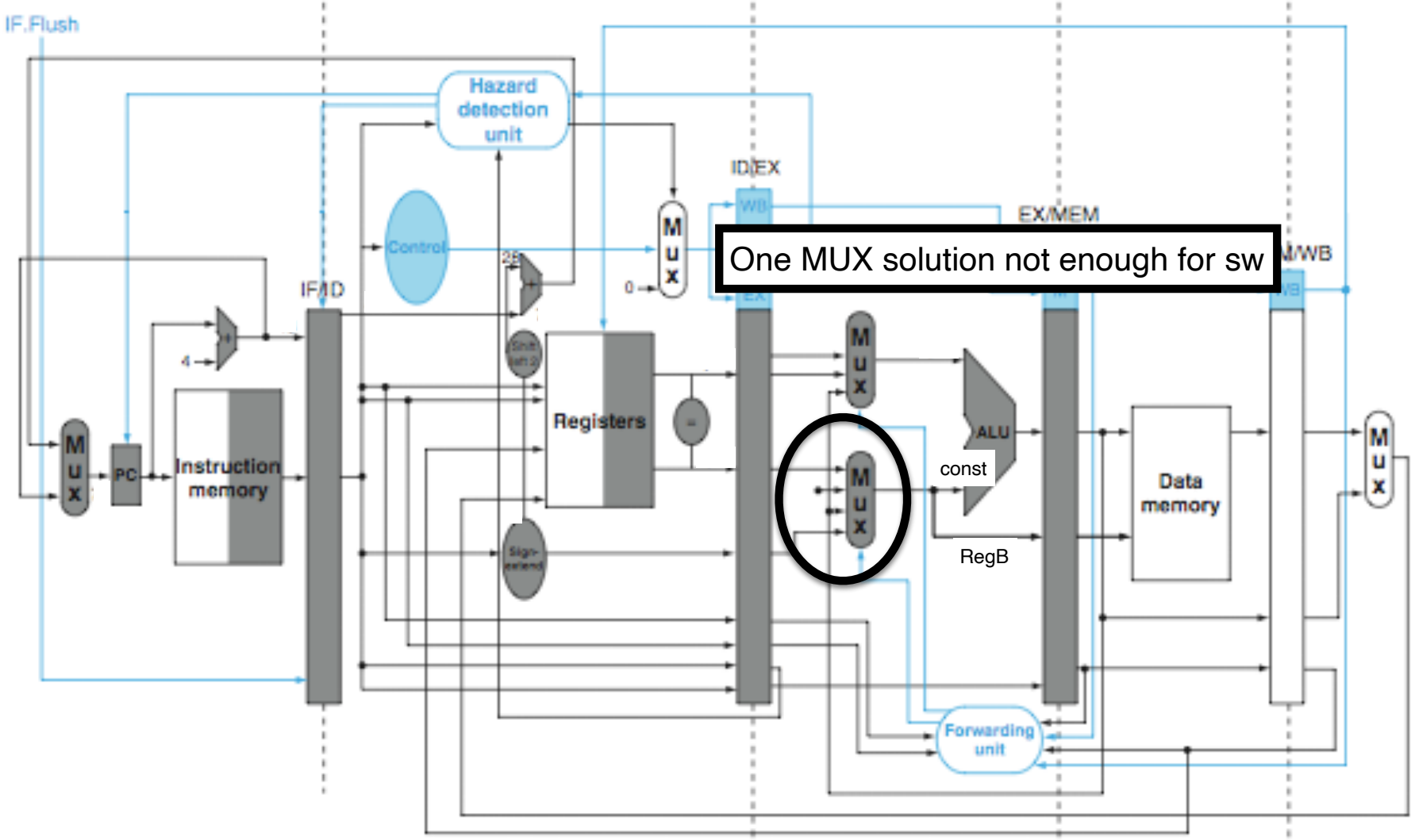
The Book's putting it together (no jump instr.)



The Book's putting it together (no jump instr.)

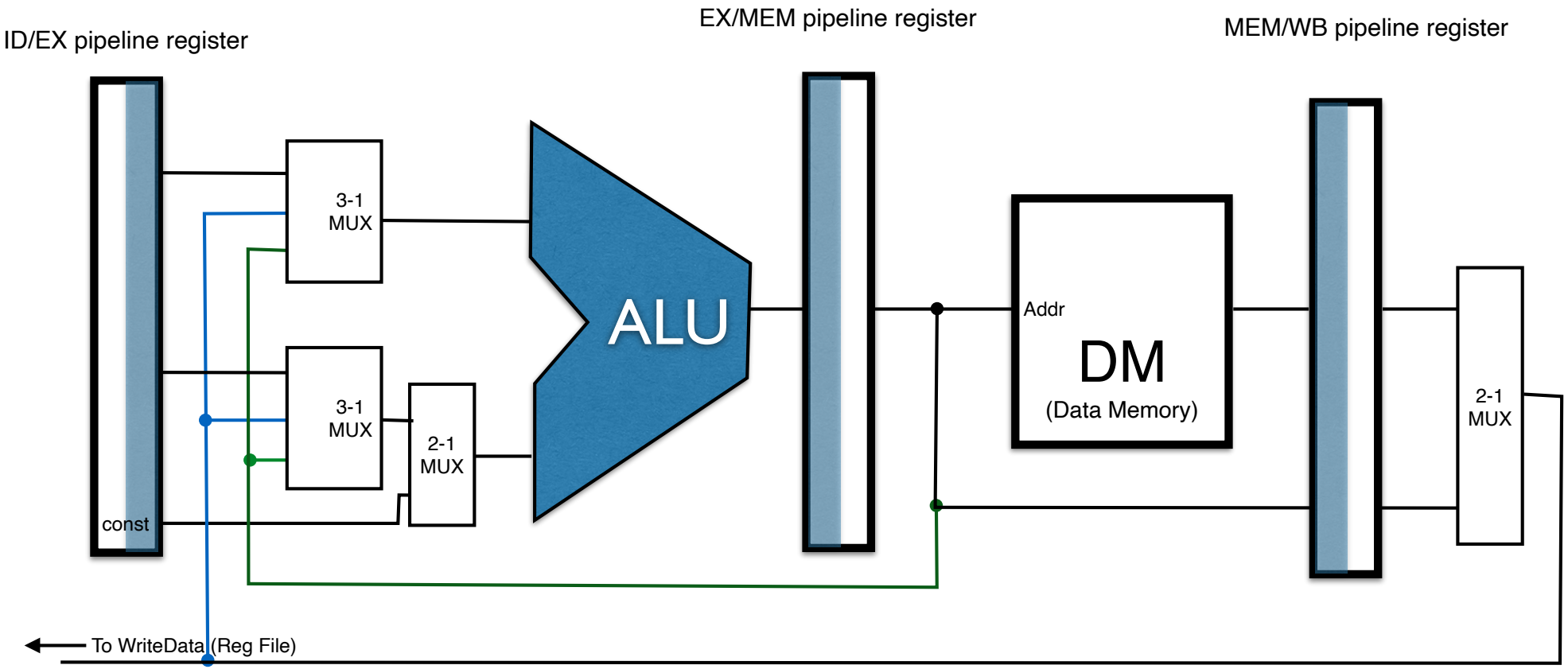


The Book's putting it together (no jump instr.)



Need both const and RegB: can't both come out of the "Data Forward" MUX

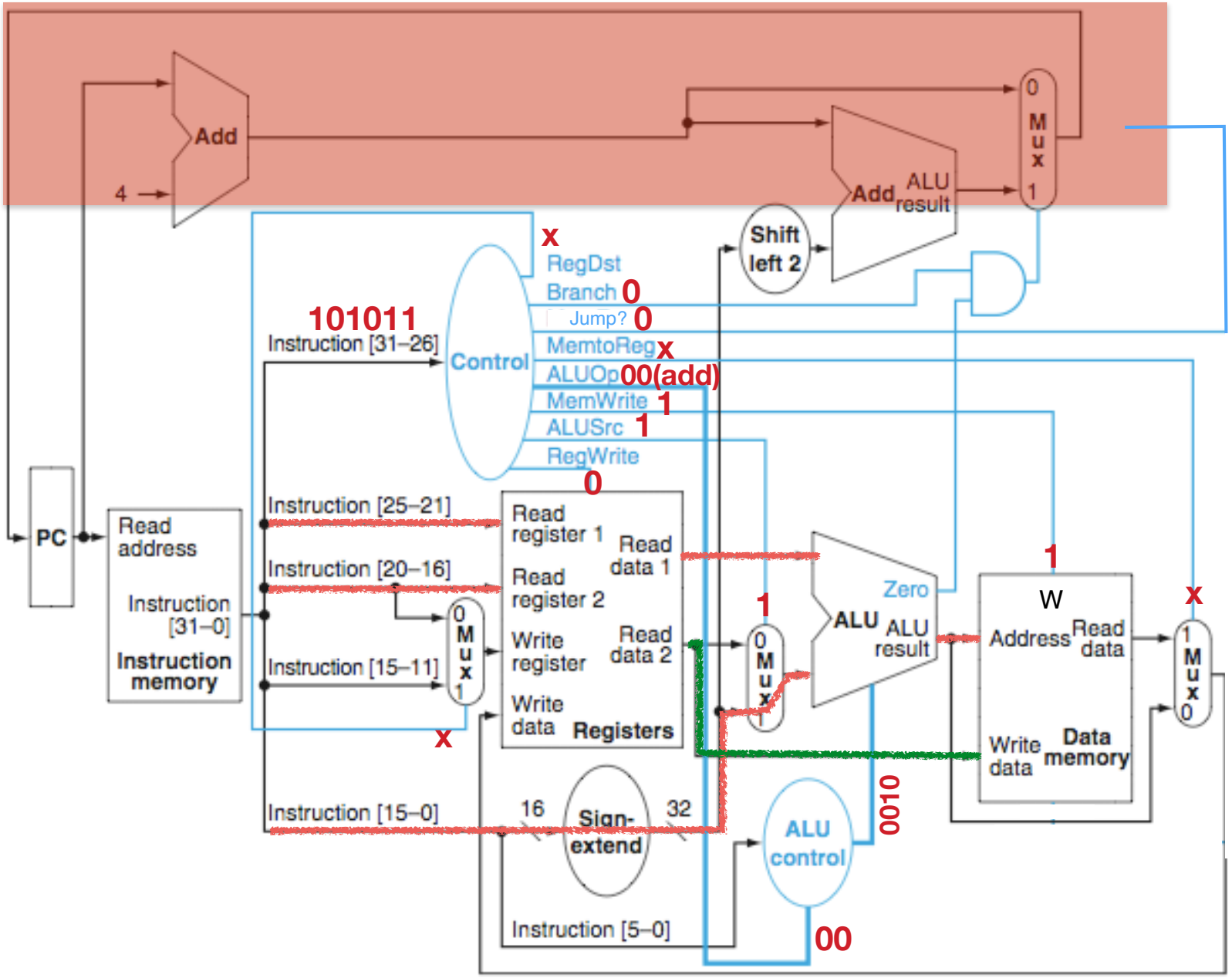
The Fix: as we drew it here…



- Forwarding can come from MEM stage

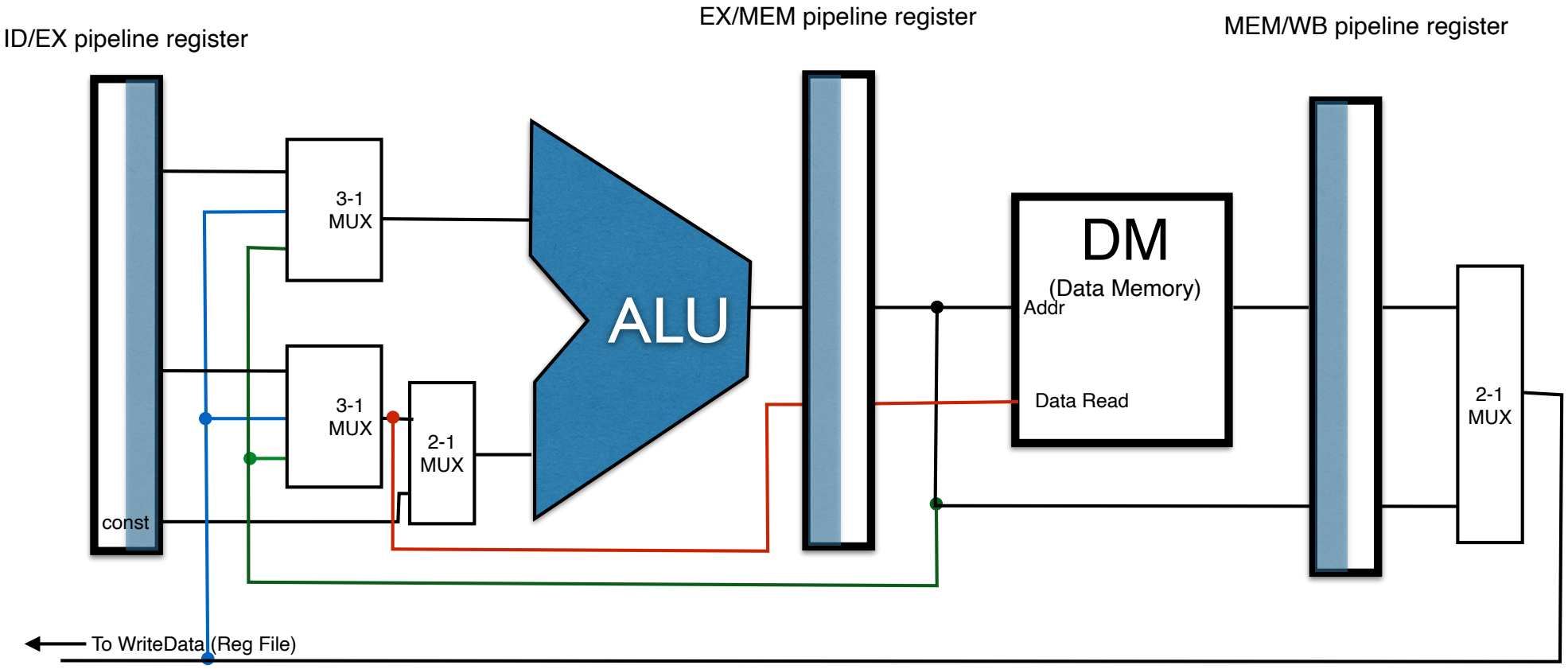
Forwarding Unit

`sw` instruction (write to memory): Used Data Flow



Red flow: reads
Green flow: writes

The Fix: as we drew it here…



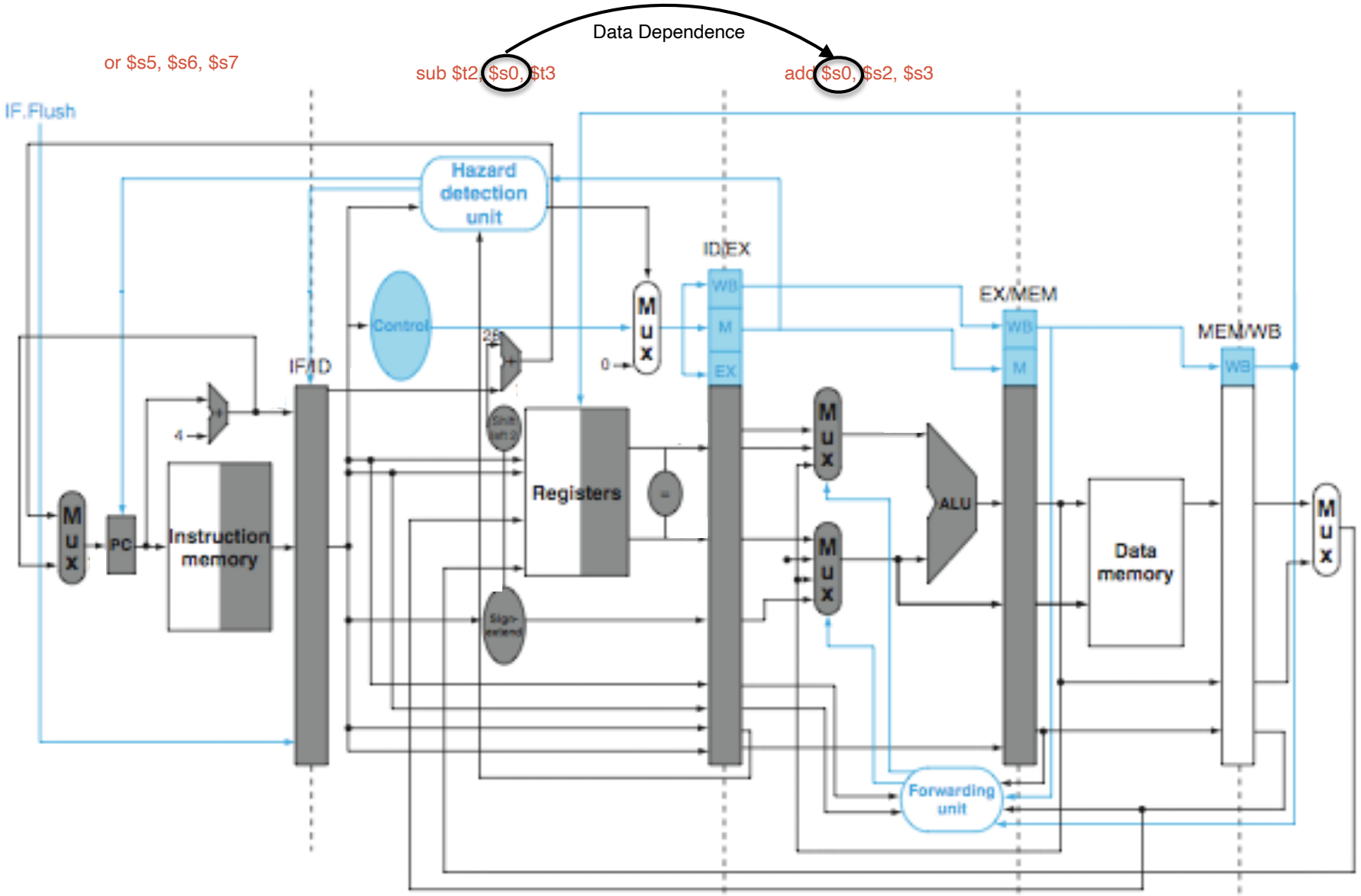
- Forwarding can come from MEM stage

Forwarding Unit

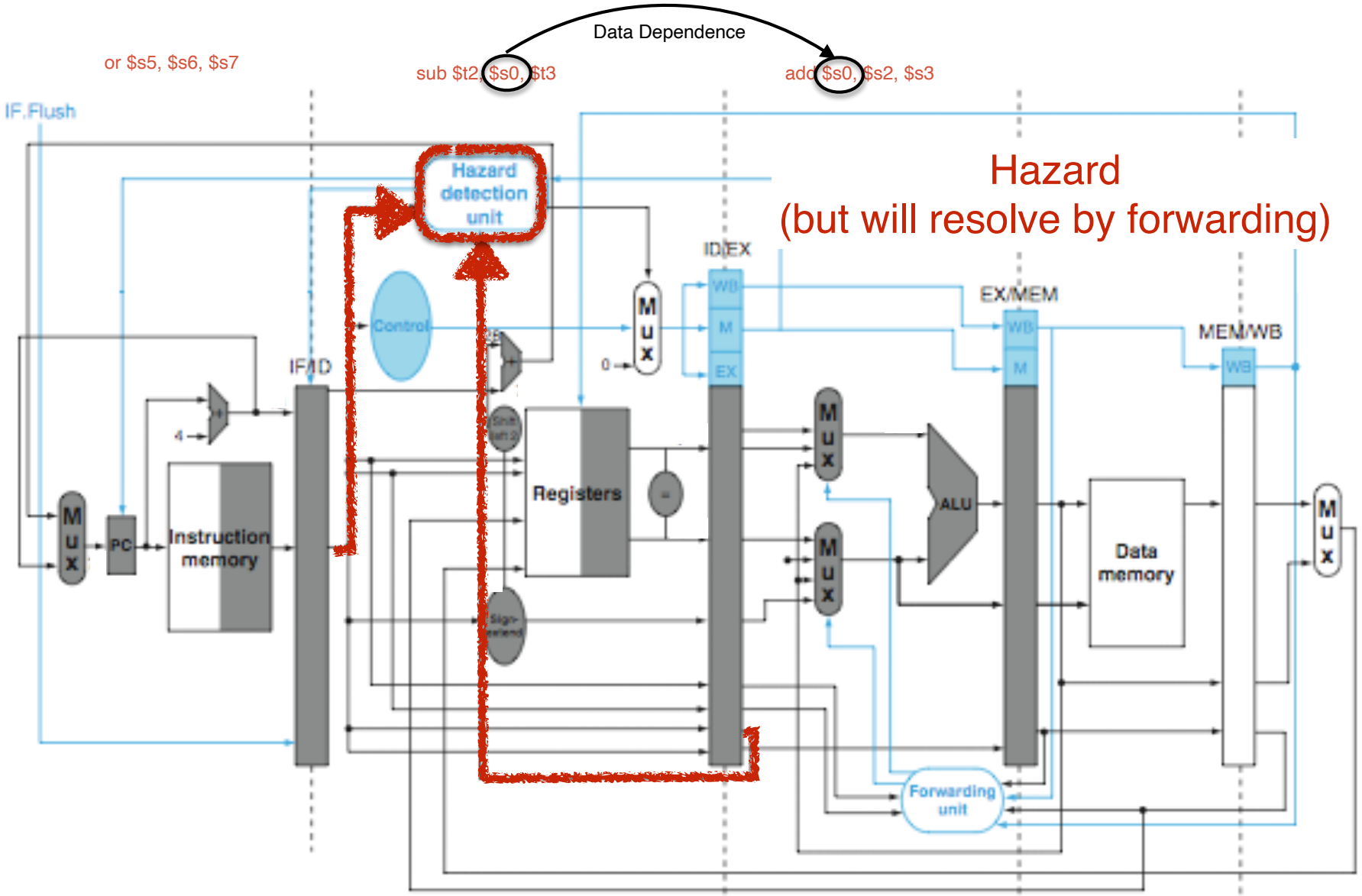
Examples of Pipelining with Hazard Resolution (the final arch)

Data Forward Example

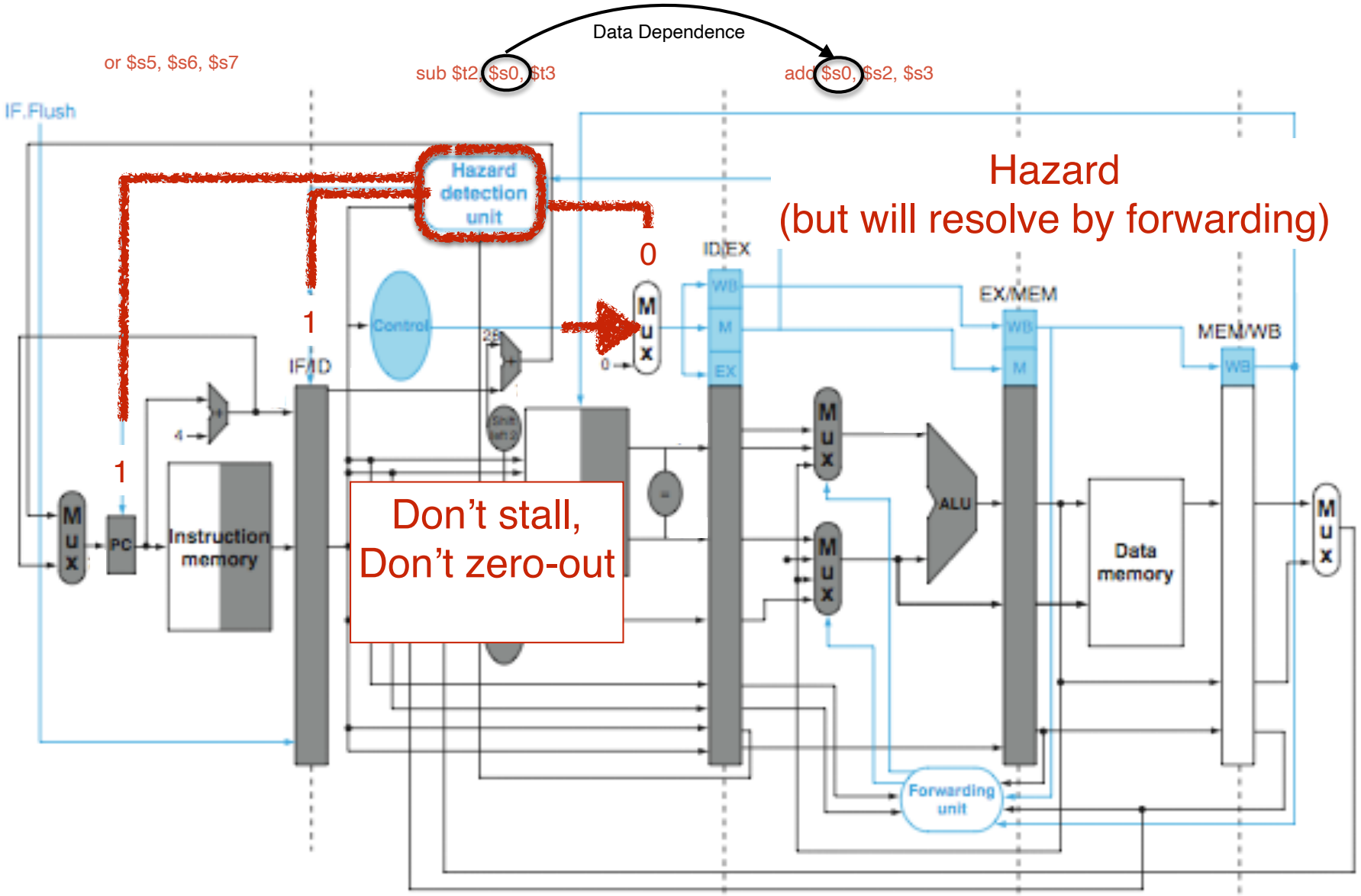
Data Forwarding (cycle t)



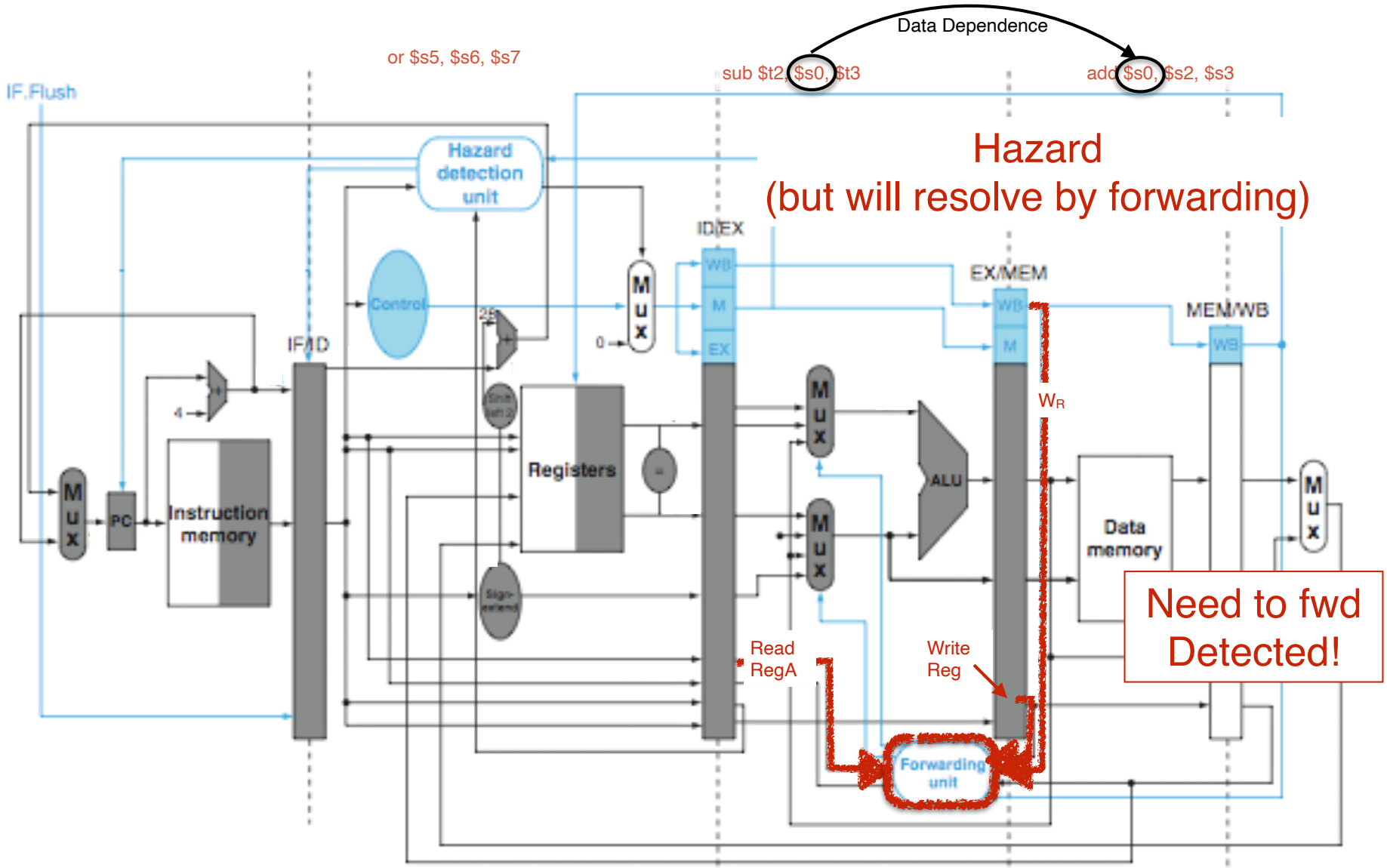
Data Forwarding (cycle t)



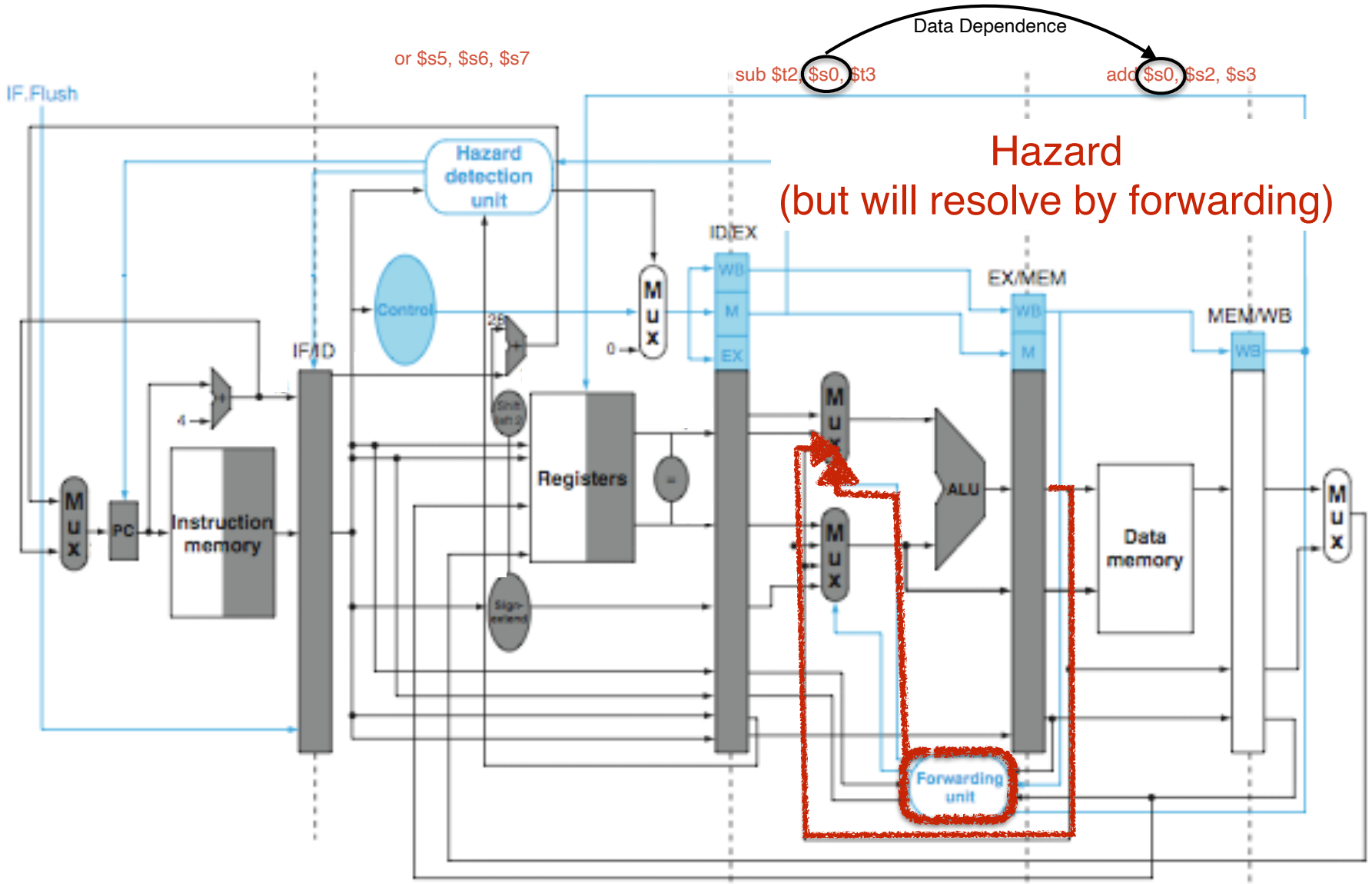
Data Forwarding (cycle t)



Data Forwarding (cycle t+1)

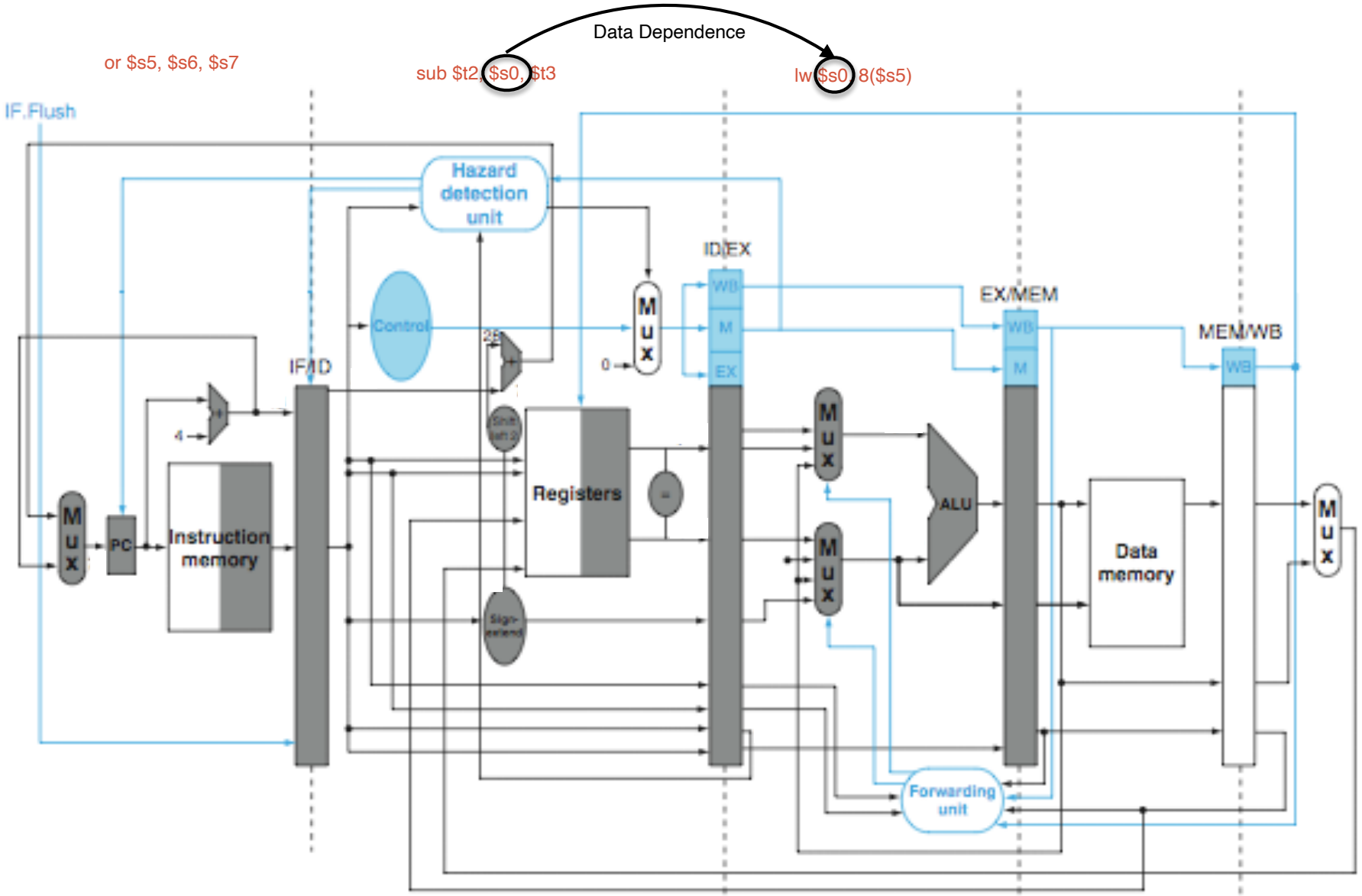


Data Forwarding (cycle t+1)

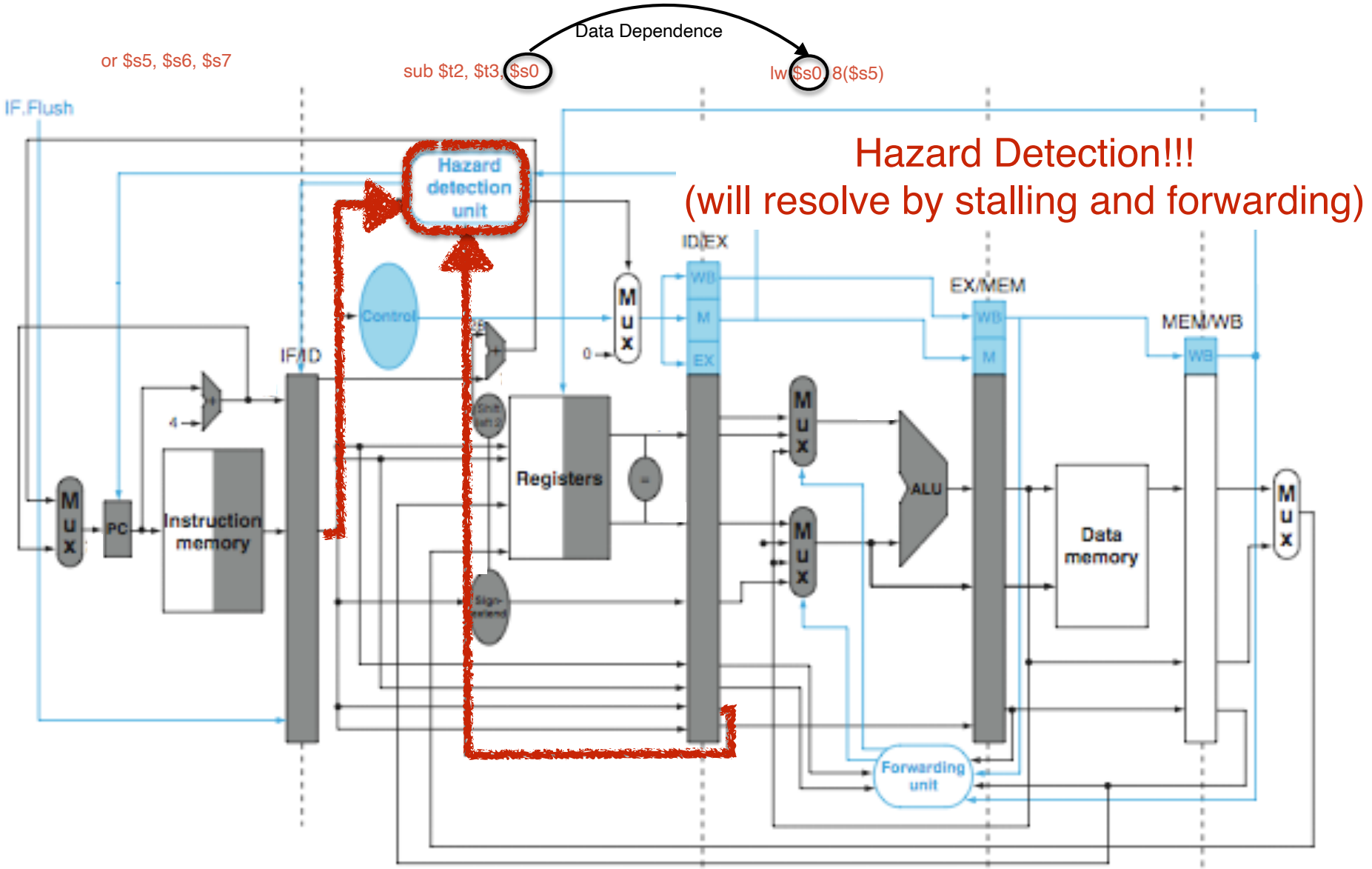


Stall and Data Forward Example

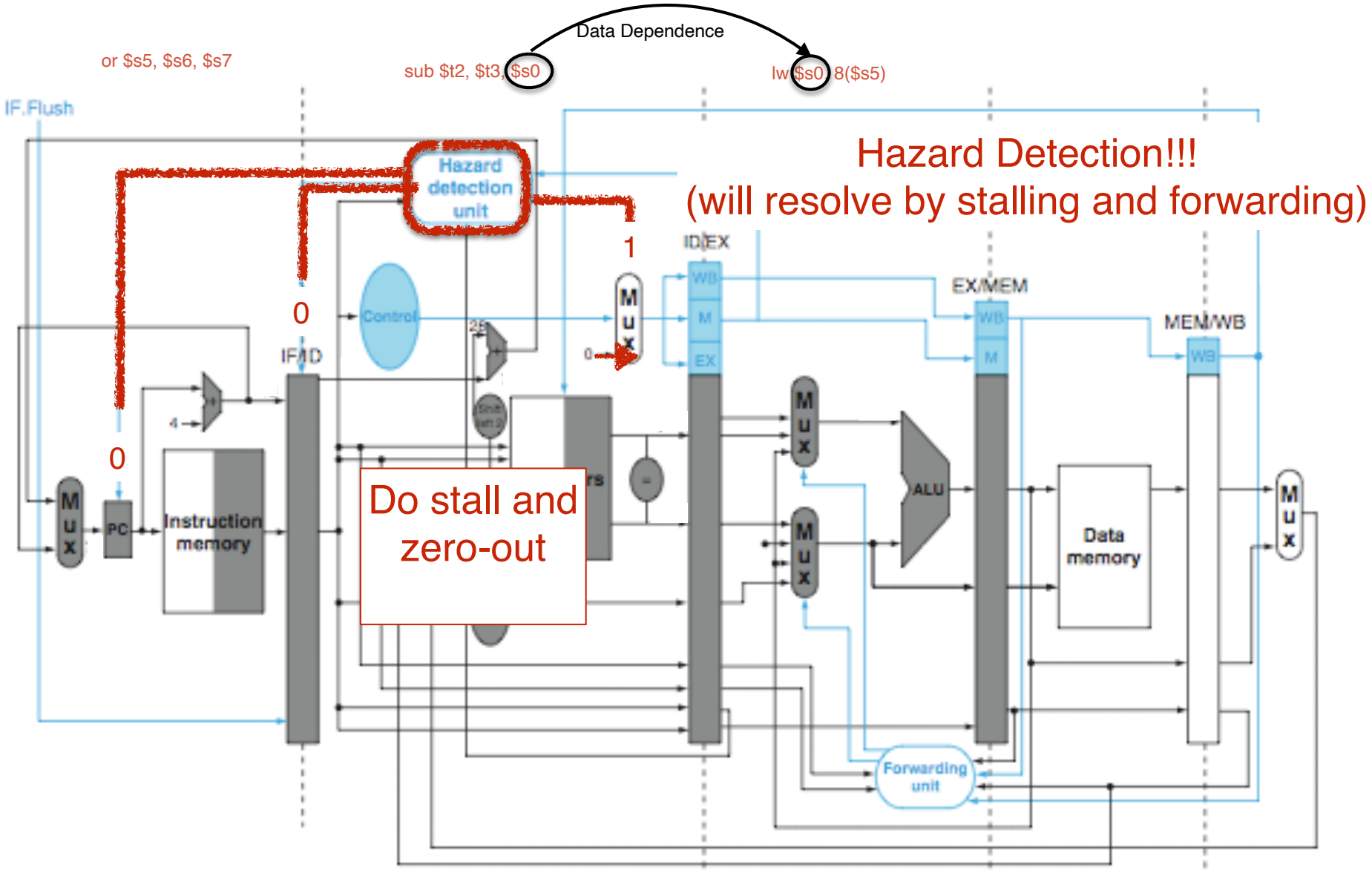
lw: Stall and Fwd (cycle t)



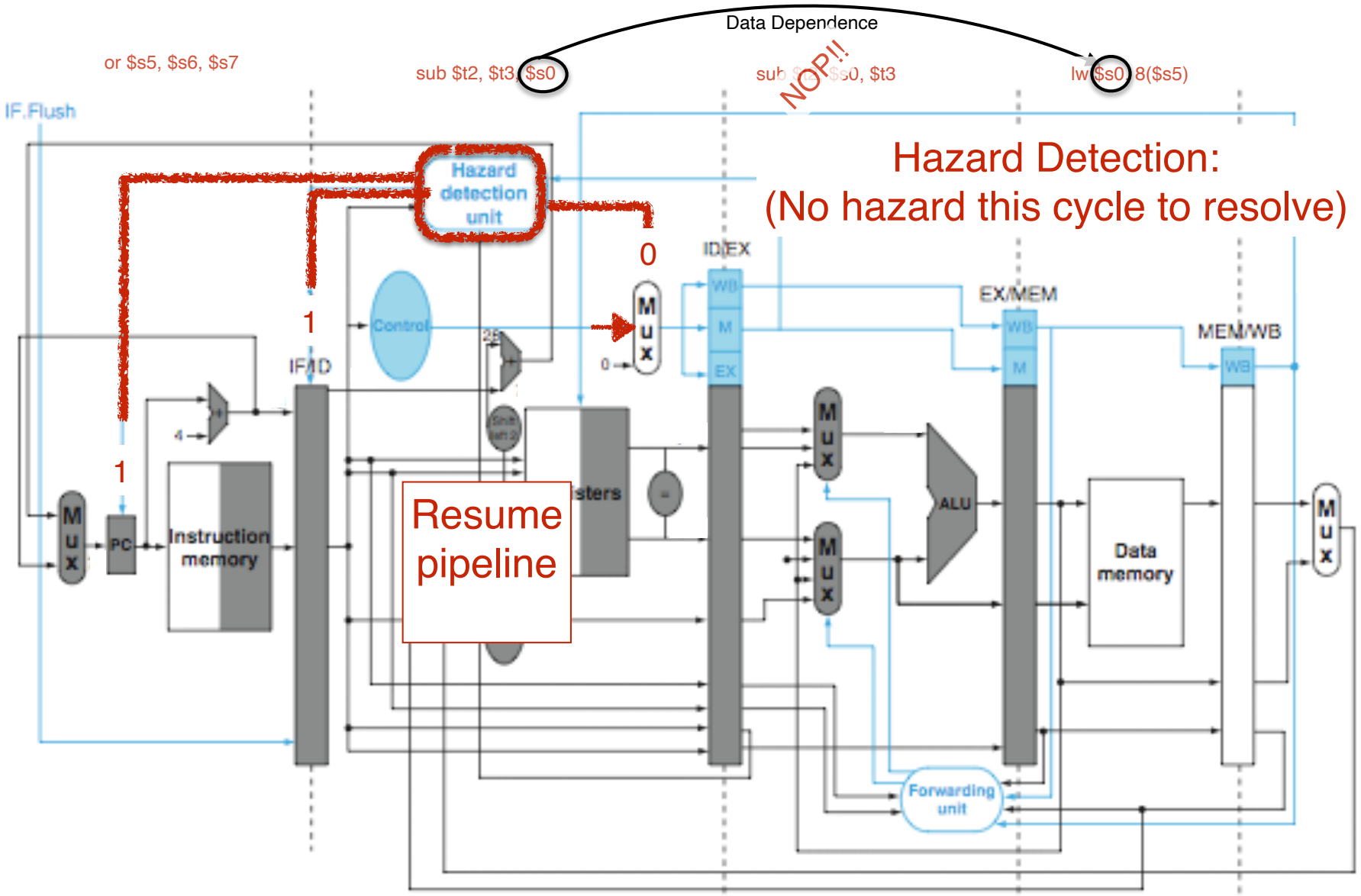
lw: Stall and Fwd (cycle t)



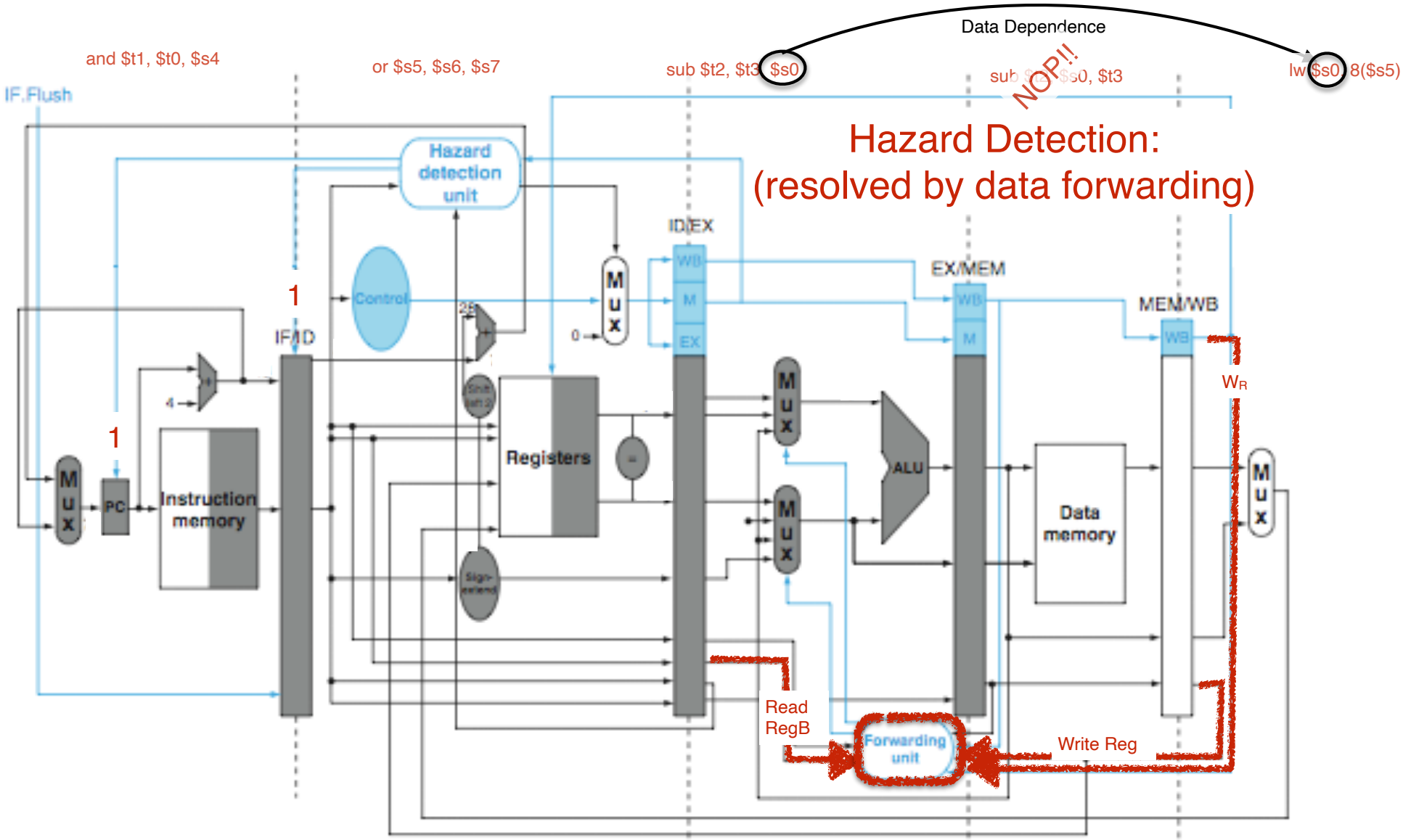
lw: Stall and Fwd (cycle t)



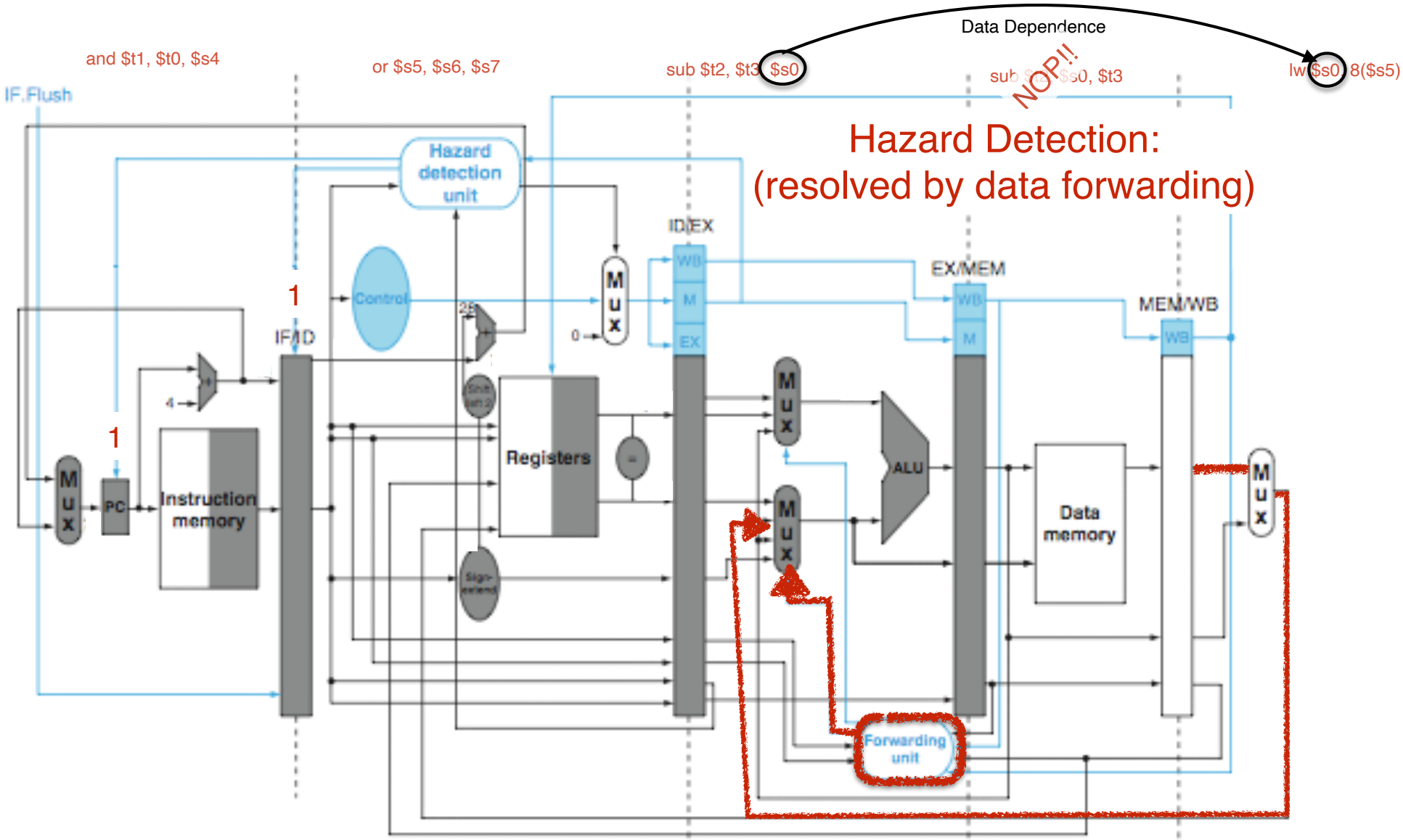
lw: Stall and Fwd (cycle t+1)



lw: Stall and Fwd (cycle t+2)

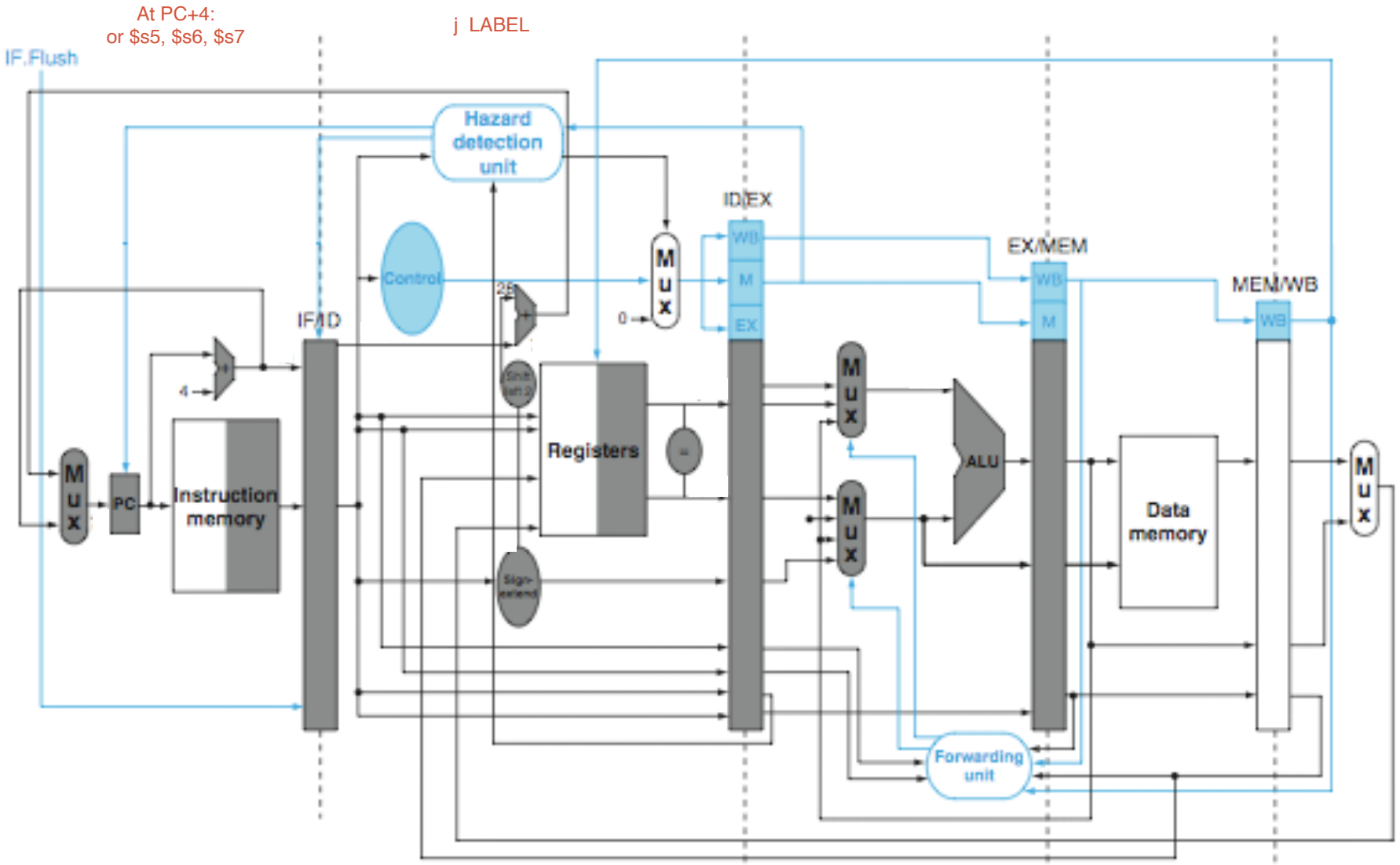


lw: Stall and Fwd (cycle t+2)

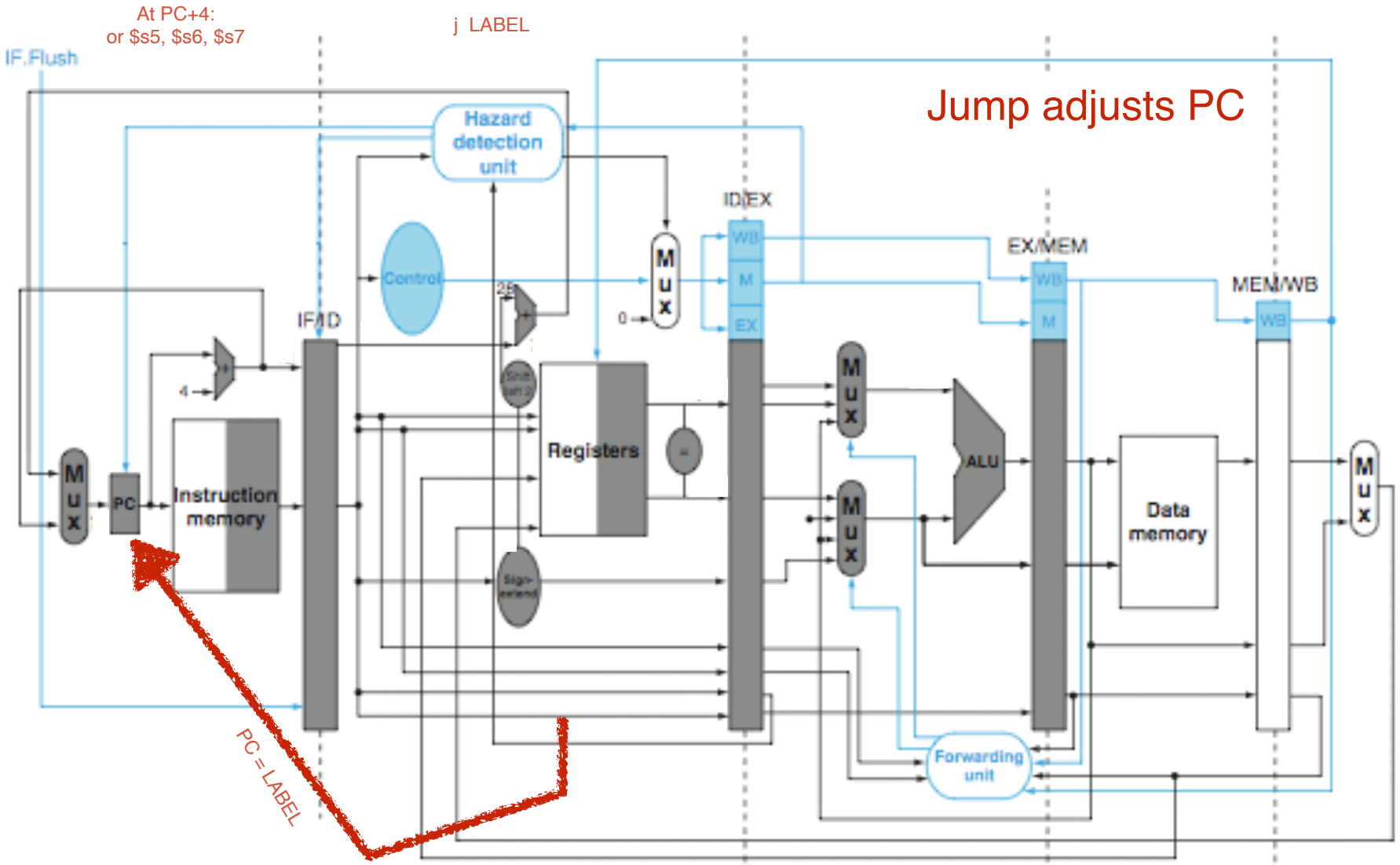


Jump Example

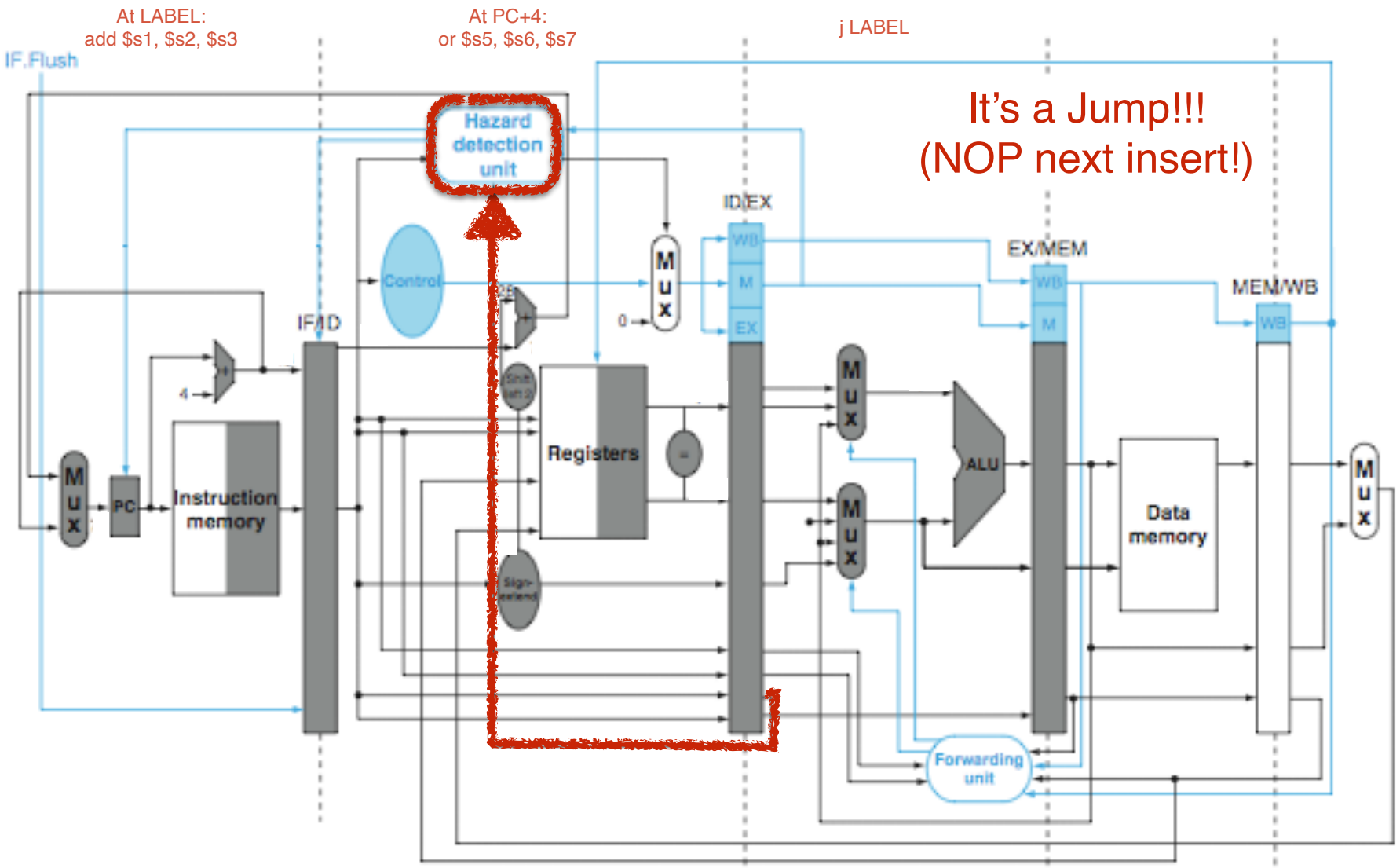
Jump (cycle t)


At PC+4:
or $s5, $s6, $s7
j LABEL
IF.Flush
Hazard detection unit
Control
IF/ID
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Mux
PC
Instruction memory
Registers
ALU
Data memory
Forwarding unit
4
0

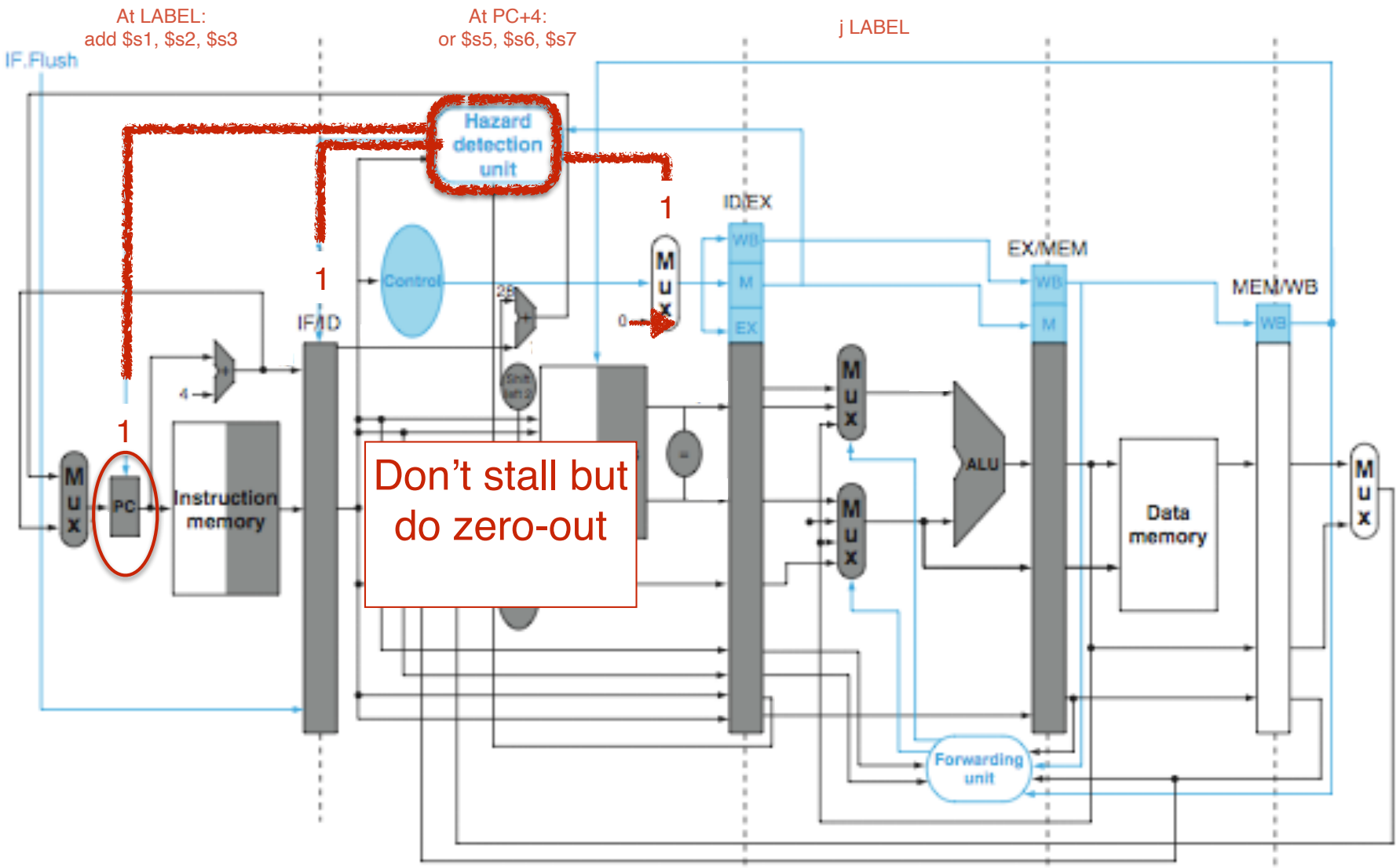
Jump (cycle t)


At PC+4:
or $s5, $s6, $s7
j LABEL
IF.Flush
Jump adjusts PC
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/D
WB
M
EX
Registers
ALU
Instruction memory
PC
Data memory
Forwarding unit
PC = LABEL

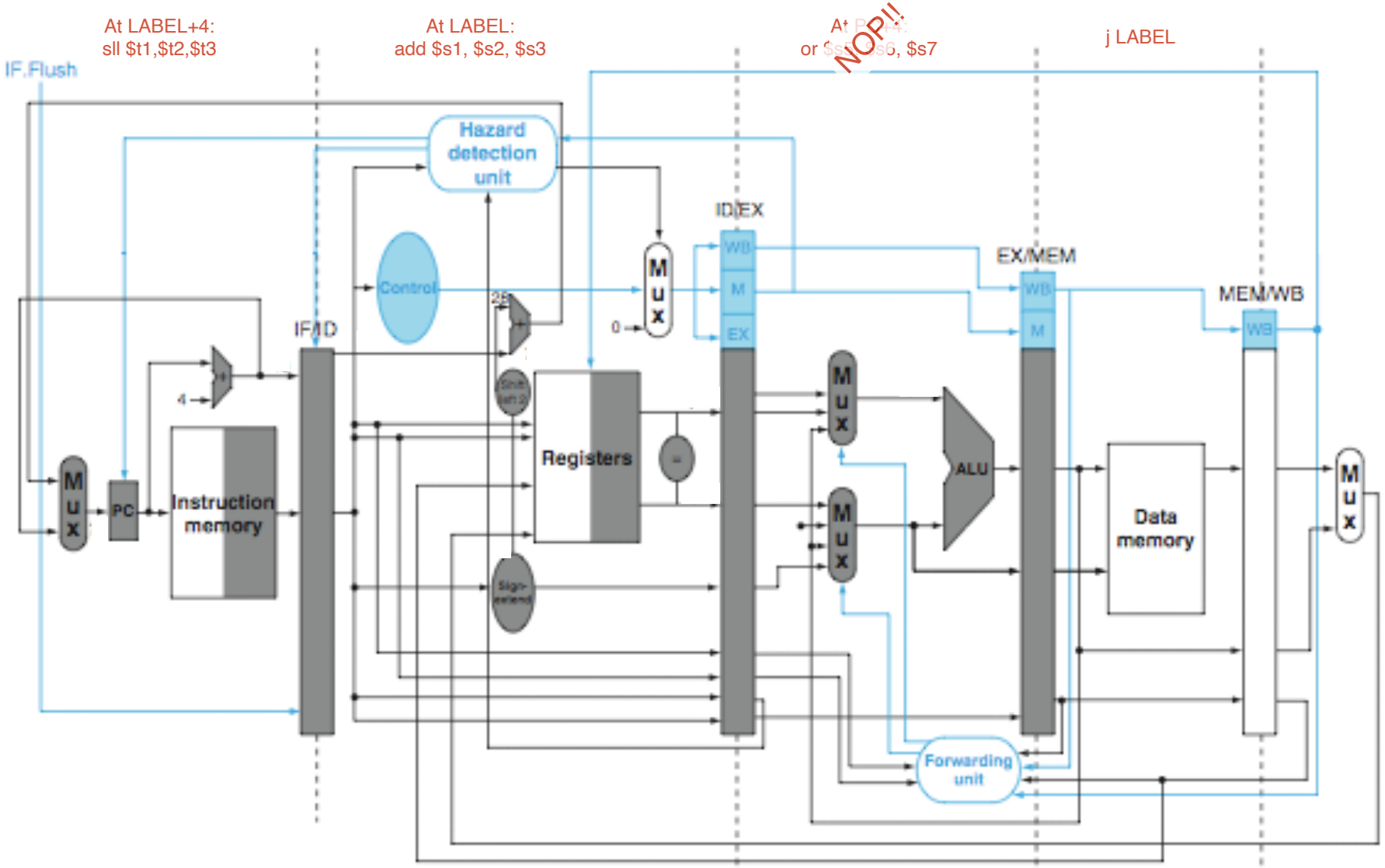
Jump (cycle t+1)



Jump (cycle t+1)



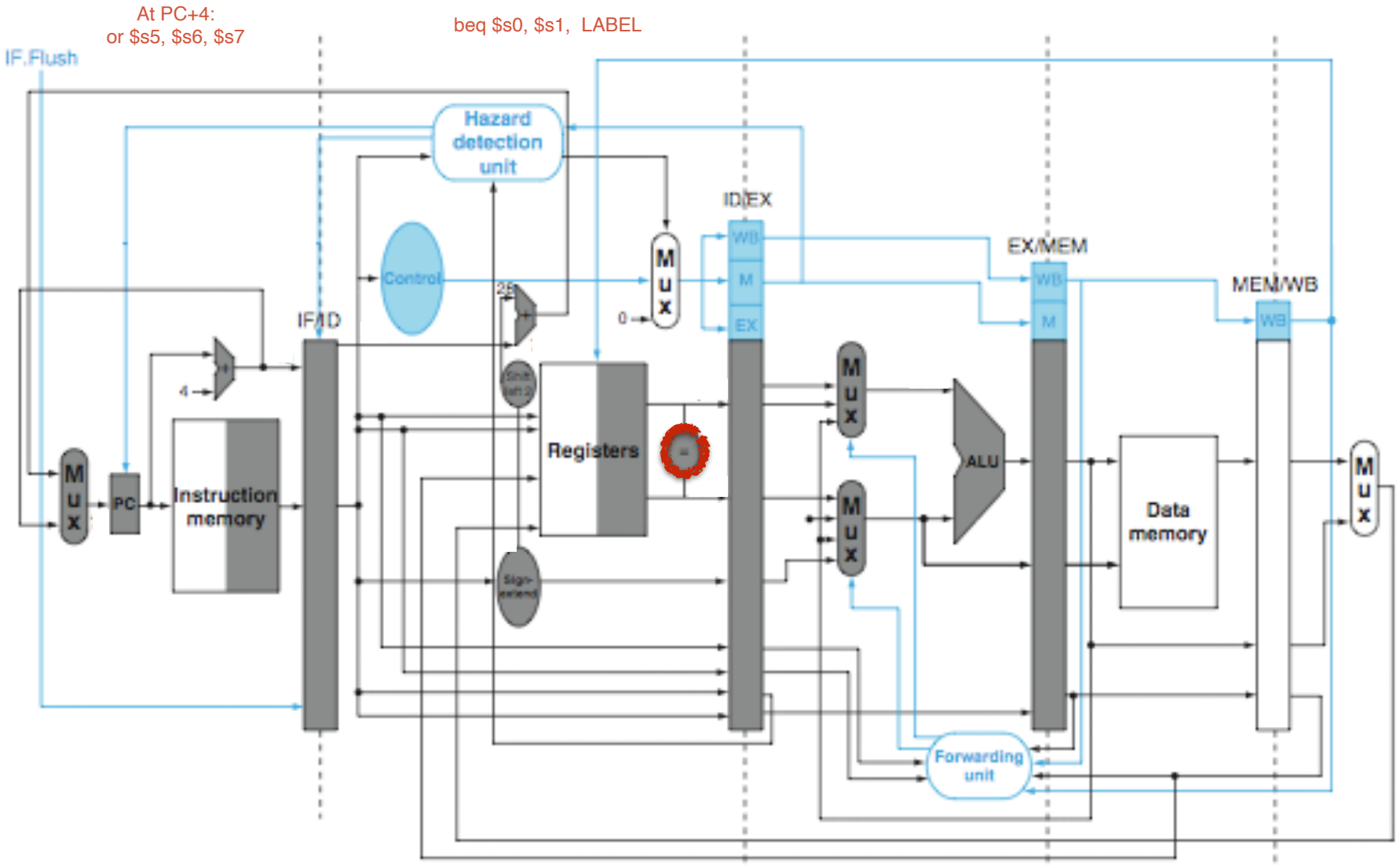
Jump (cycle t+2)



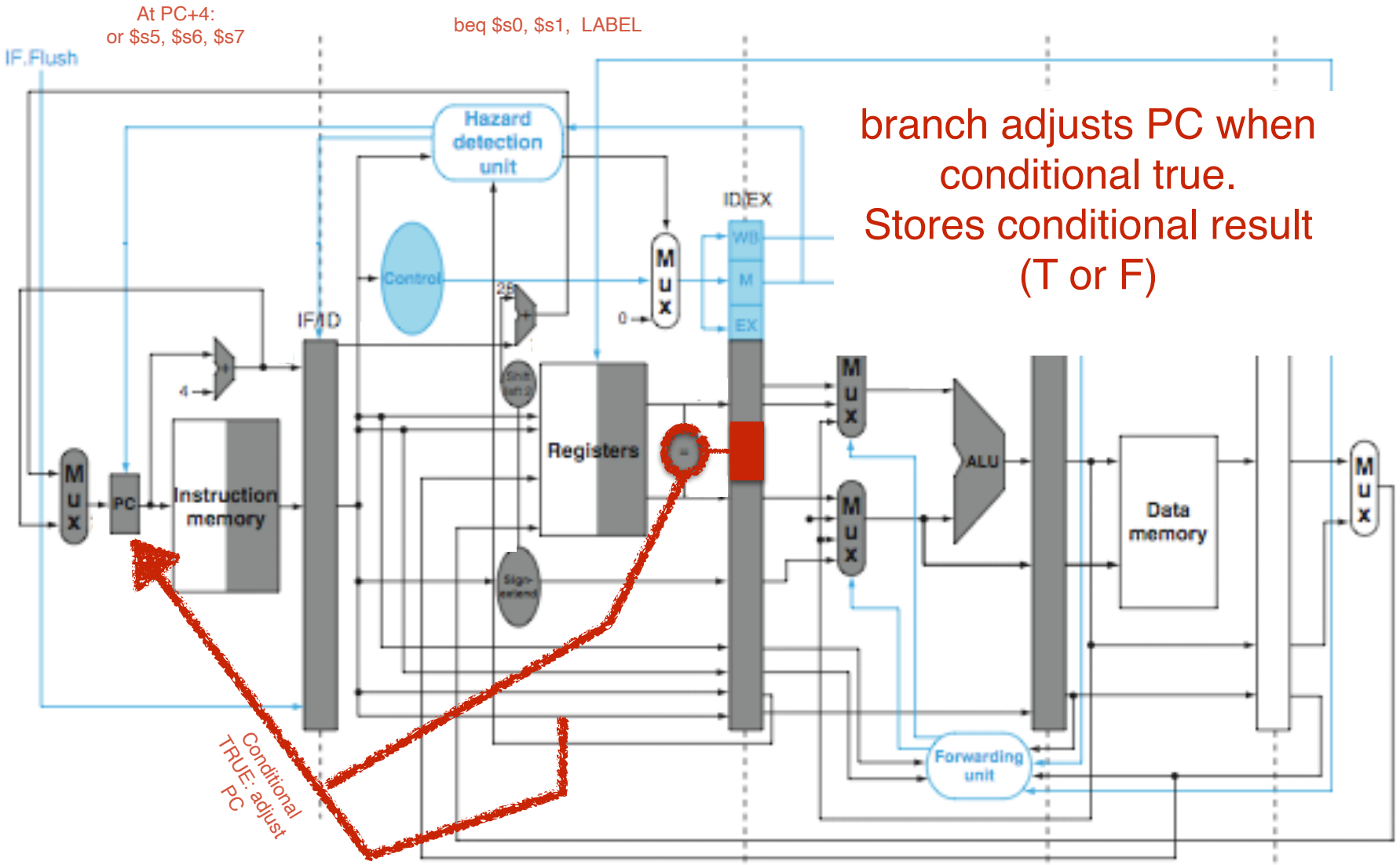
BEQ example with branch prediction:

predicts conditional false

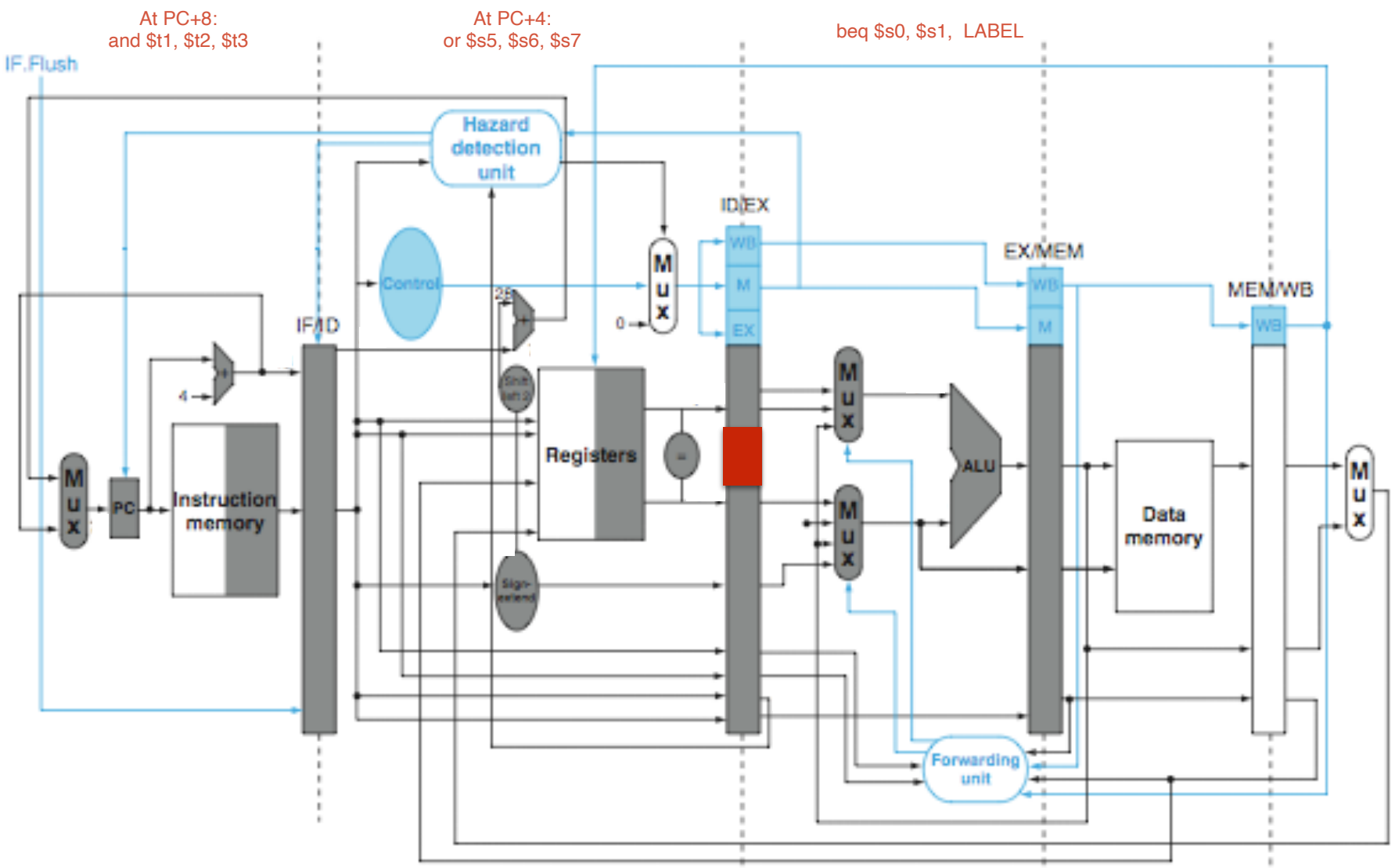
BEQ (cycle t): assume branch prediction



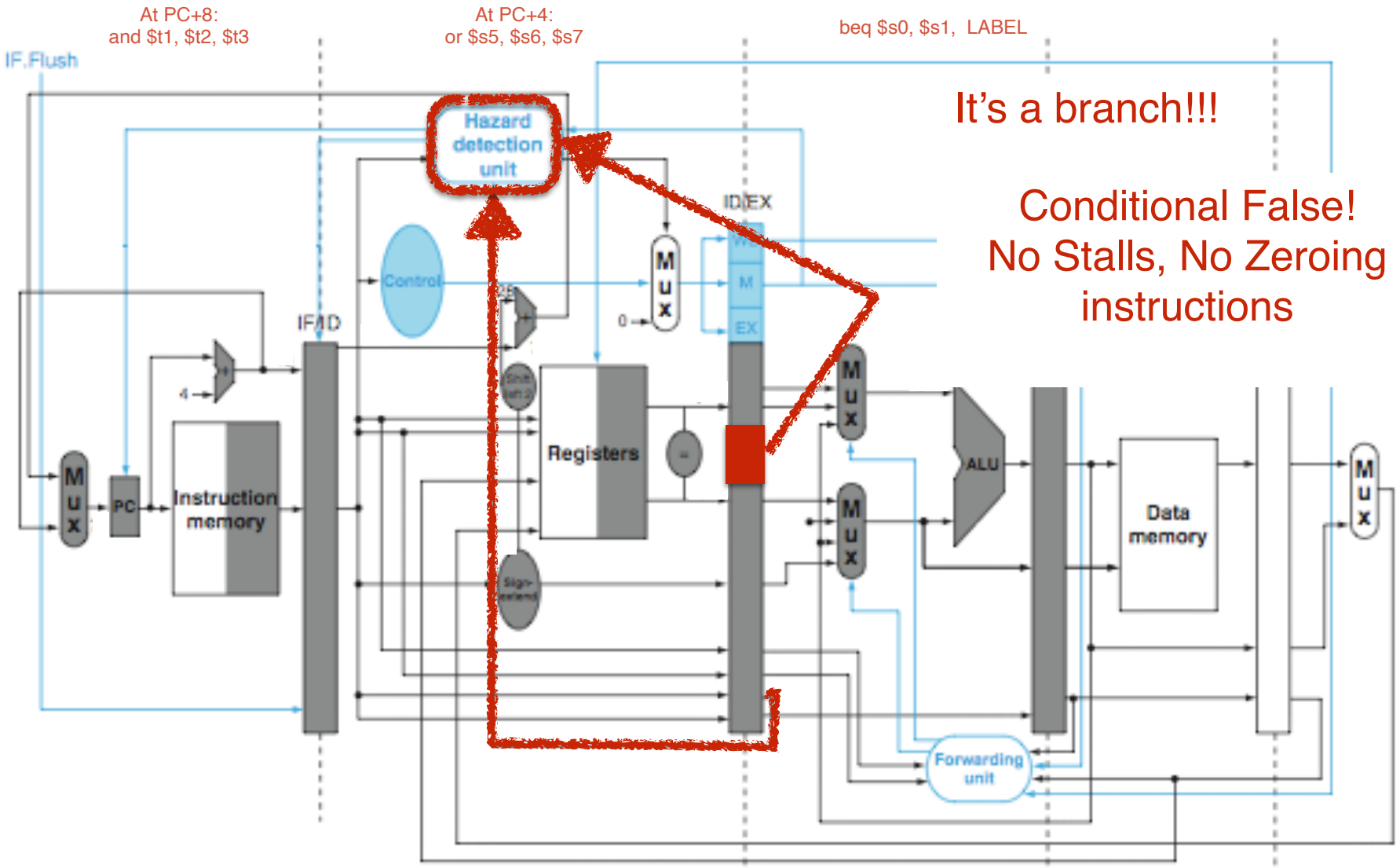
BEQ (cycle t): assume branch prediction



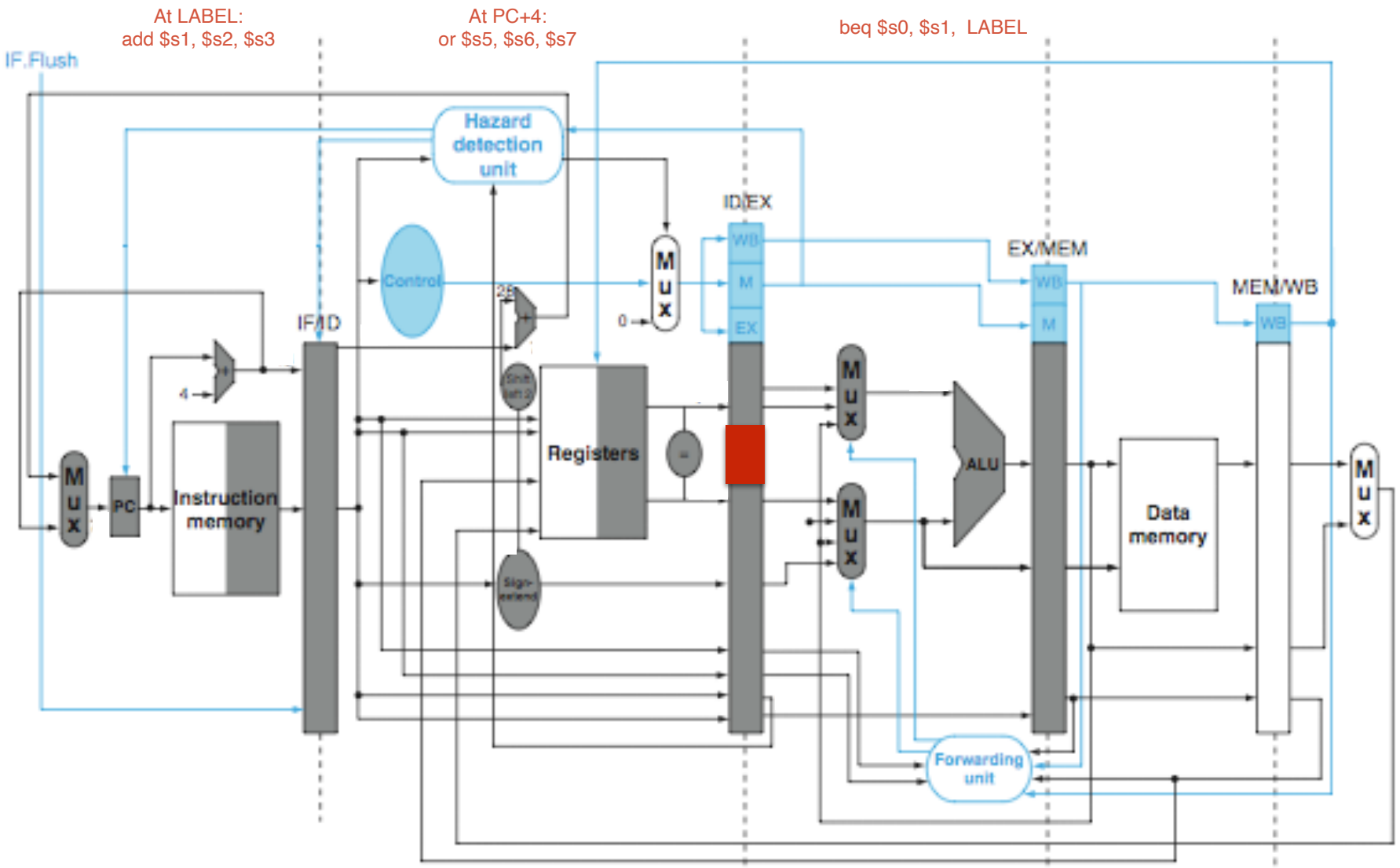
BEQ (cycle t+1): Conditional False Case



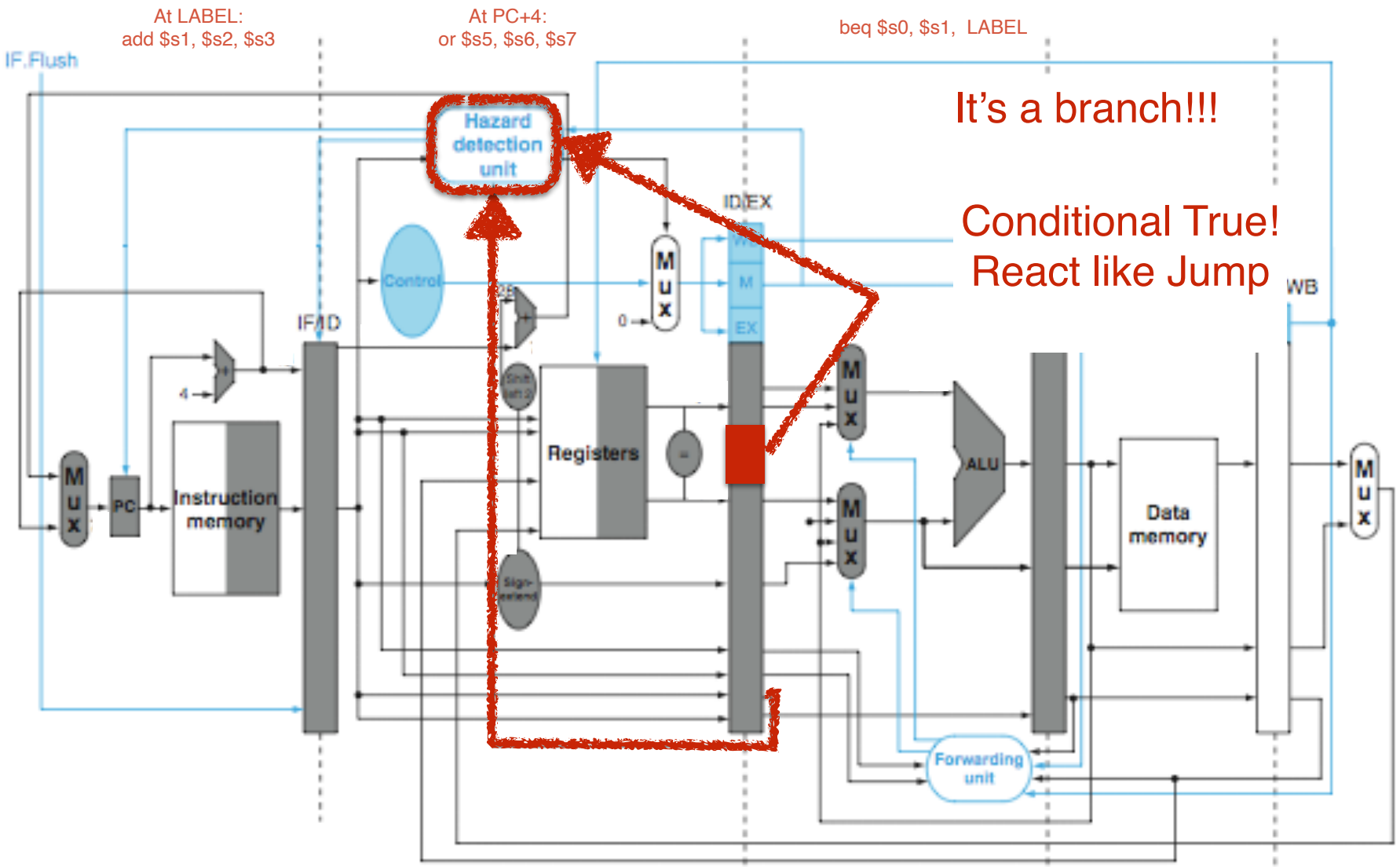
BEQ (cycle t+1): Conditional False Case



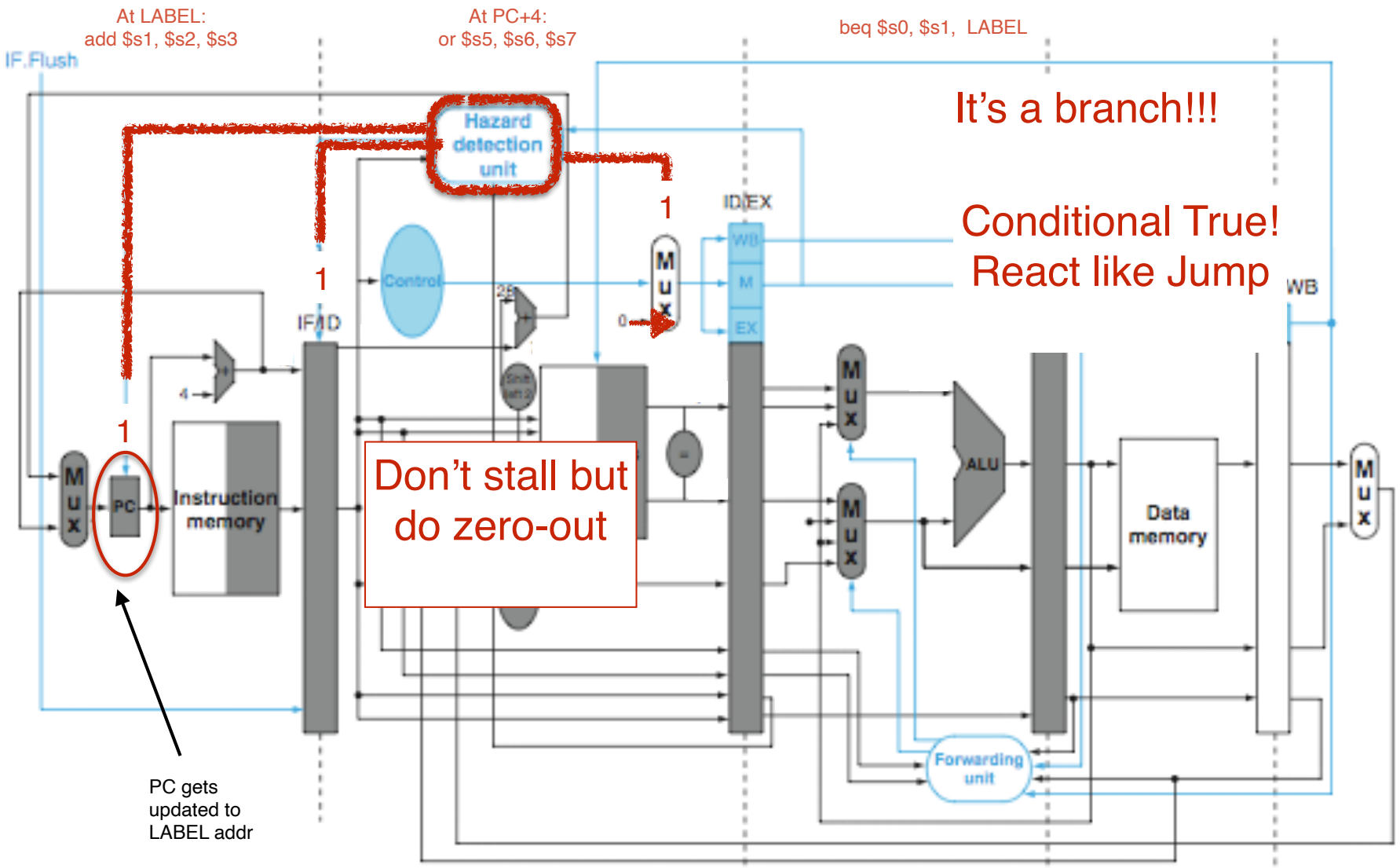
BEQ (cycle t+1): Conditional True case



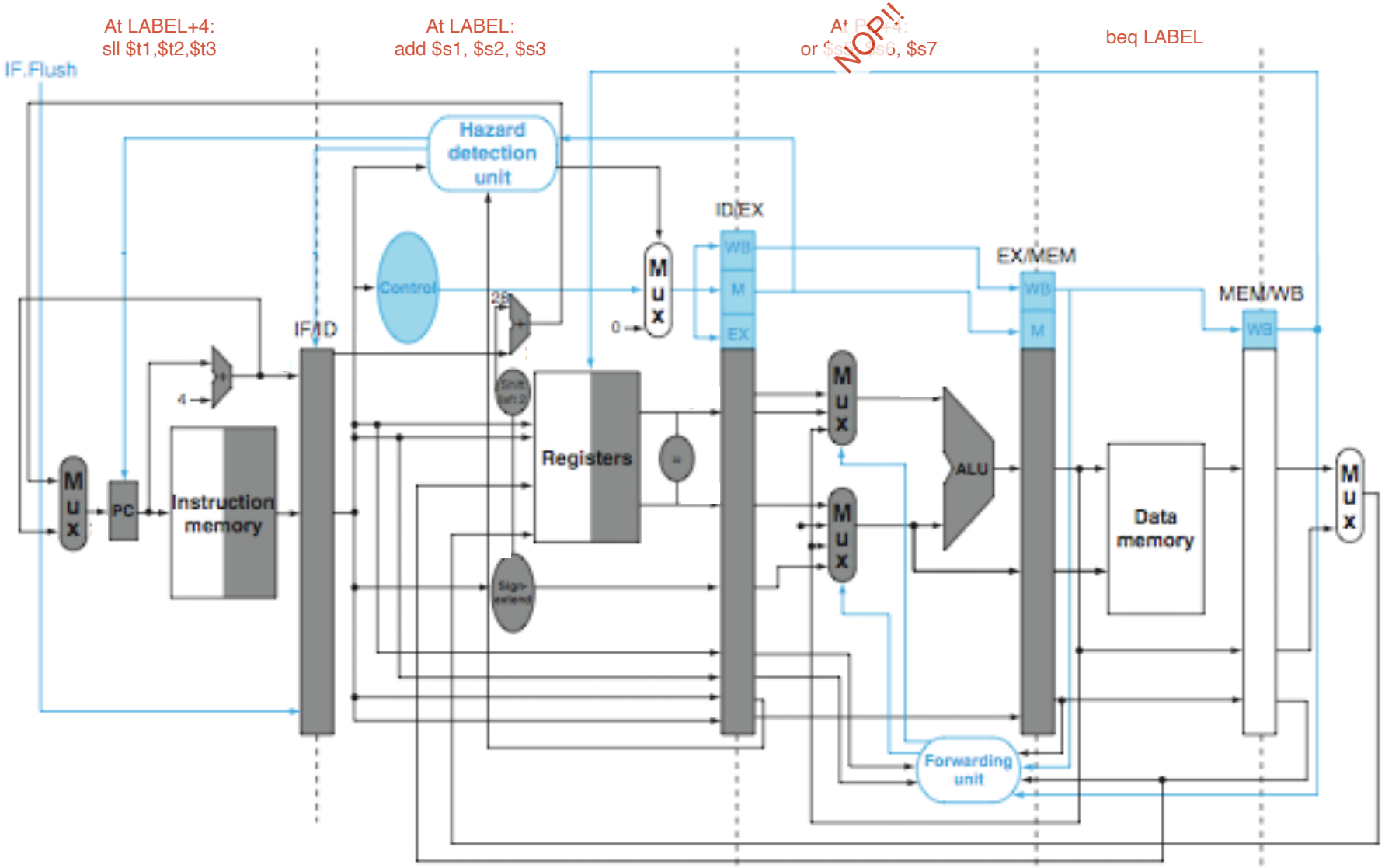
BEQ (cycle t+1): Conditional True case



BEQ (cycle t+1): conditional True



BEQ (cycle t+2)


At LABEL+4:
sll $t1,$t2,$t3
At LABEL:
add $s1, $s2, $s3
beq LABEL
NOP!!
IF.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Registers
ALU
Data memory
Instruction memory
PC
Forwarding unit
Mux