CO101
Principle of Computer Organization

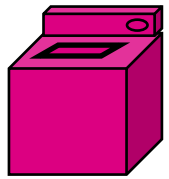
Lecture 06: Pipelined MIPS Processor 1

Liang Yanyan

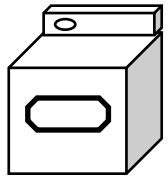
澳門科技大學
Macau of University of Science and Technology

Pipelining: Laundry example

- Four persons: A, B, C, and D.
  - Each has one load of clothes to wash, dry, fold, and stash.
- Assume
  - “Washer” takes 30 minutes.
  - “Dryer” takes 30 minutes.
  - “Folder” takes 30 minutes.
  - “Stasher” takes 30 minutes to put clothes into drawers.



Wash: 30 mins

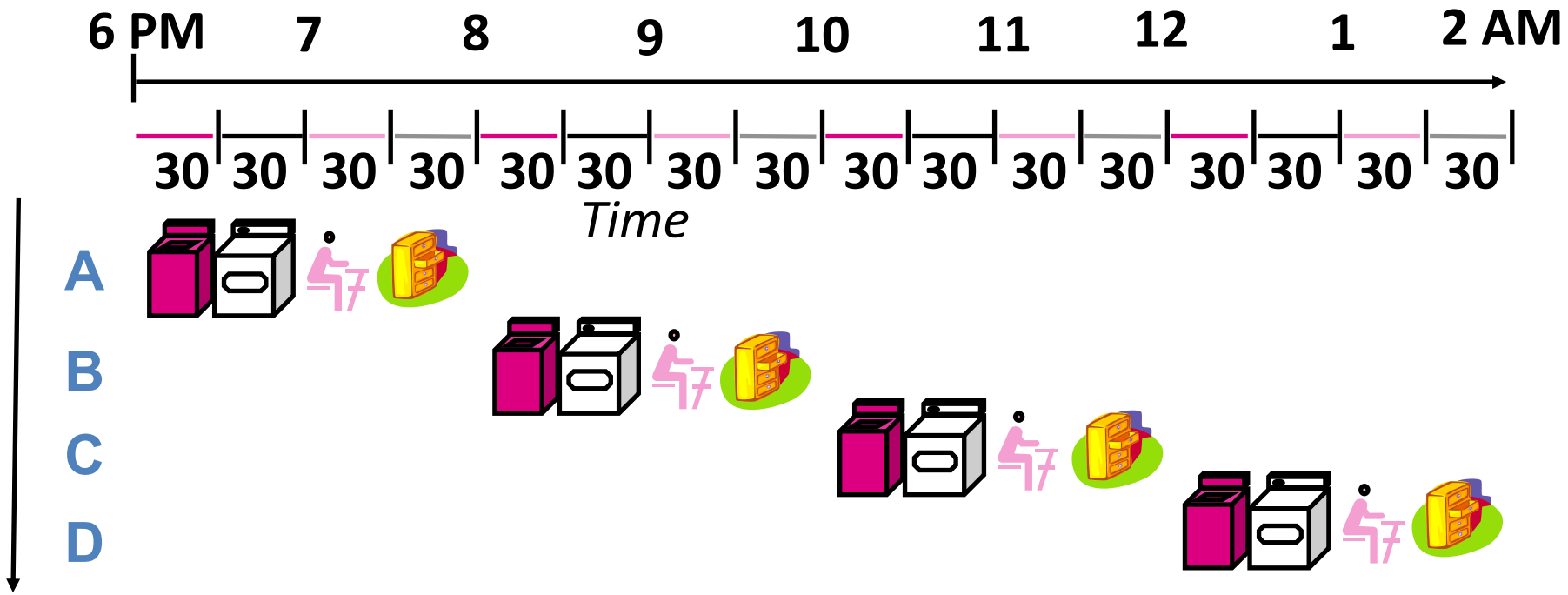


Dry: 30 mins

Fold: 30 mins

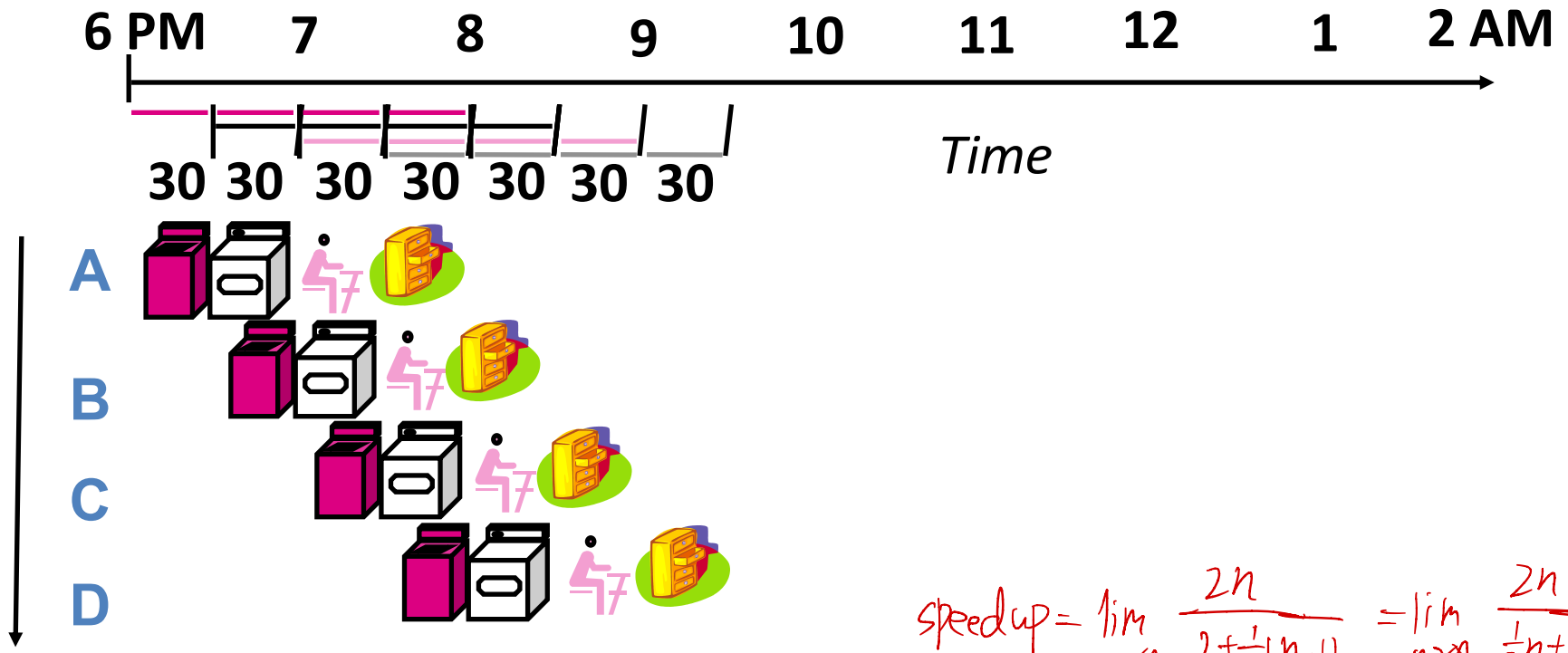
Stash: 30 mins

Sequential Laundry



- If each load is done sequentially: 8 hours for 4 loads.
- Question: can B washes after A has finished washing?

Pipelined Laundry



$$\text{speedup} = \lim_{n\to\infty} \frac{2n}{2+\frac{1}{2}(n-1)} = \lim_{n\to\infty} \frac{2n}{\frac{1}{2}n+\frac{3}{2}}$$

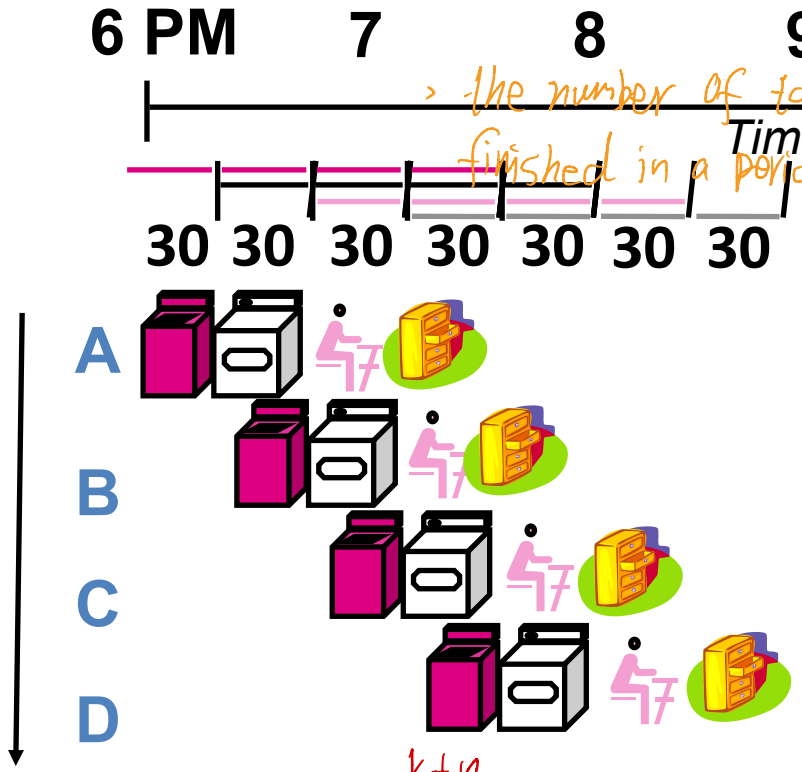
$$= \lim_{n\to\infty} \frac{4n}{n+3}$$

$$= \lim_{n\to\infty} \frac{4}{1+\frac{3}{n}} = 4$$

- Start work as soon as possible.
- It takes 3.5 hours for 4 loads!
  - Speedup = 8/3.5 = 2.3
  - What is the speedup if there are 10000 loads?

$$\text{speedup} = \frac{2\times 100}{2+0.5\times 99} = \frac{200}{51.5} \approx 3.9$$

Pipelining Lessons



- Multiple tasks operate simultaneously using distinct resources.
- Pipelining doesn't improve latency of individual task, it improves throughput of entire workload.
- Potential speedup = Number of pipeline stages.
- Time to "fill" pipeline and time to "drain" it reduces speedup.
- Pipeline rate (throughput) limited by slowest pipeline stage (i.e. slowest task)
  - Wash takes 300 mins?
  - Unbalanced lengths of pipeline stages reduces throughput.

$$\text{Speedup} = \lim_{n\to\infty} \frac{kt+tn}{kt+t(n-1)} = \lim_{n\to\infty} \frac{kn}{k+n-1}$$

$$= \lim_{n\to\infty} \frac{k}{\frac{k}{n}+1-\frac{1}{n}} = k$$

k -> number of stages

t -> time needed for each task of the stage

n -> the number of loads

Use pipelining to improve speedup by increasing throughput.

Question:

Complete execution time which can be reduced.

- Can we improve the performance of processor using pipelining? YES
- Increase the speedup (throughput) by pipelining? YES

Review: Instruction Critical Paths

- Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires) except:
  - Instruction and Data Memory (4 ns)
  - ALU and adders (2 ns)
  - Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	4	1	2		1	8
load	4	1	2	4	1	12
store	4	1	2	4		11
beq	4	1	2			7
jump	4					4

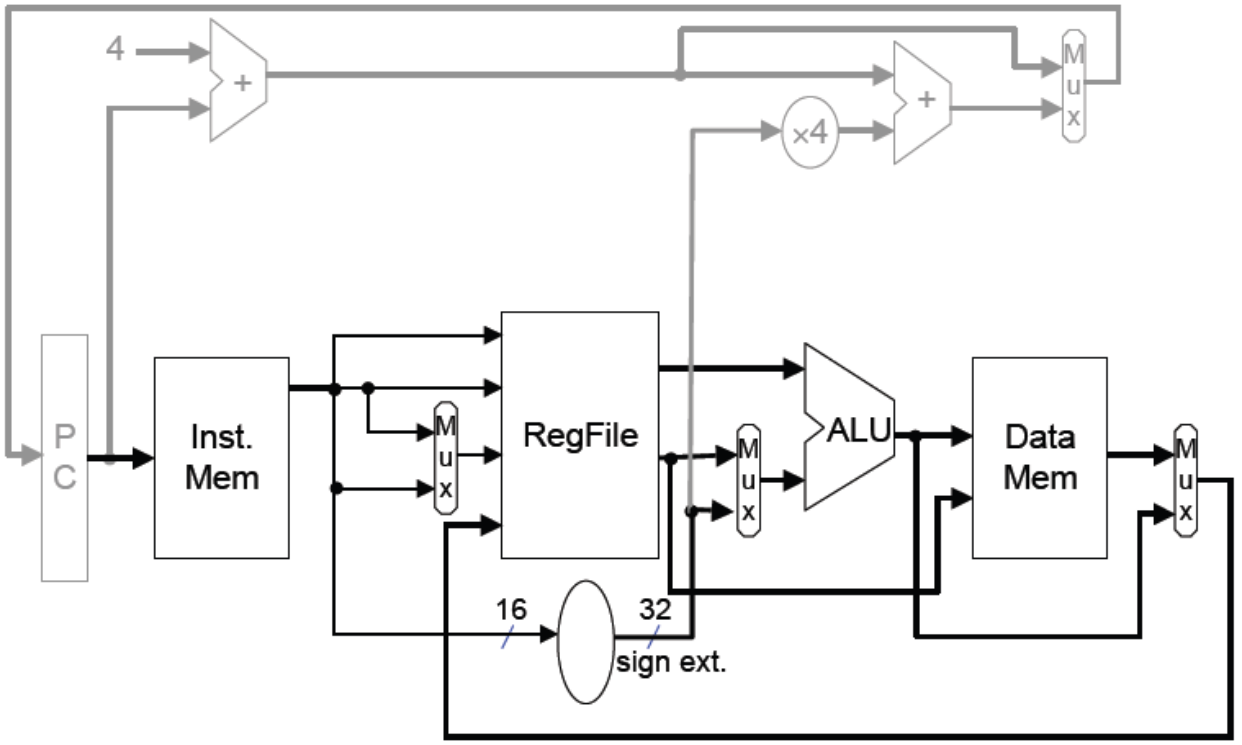
LW instruction: contains 5 steps

5-stage pipeline

- IF: Instruction fetch from instruction memory.
- ID: Instruction decode and register fetch.
- EX: Calculate the memory address.
- MEM: Read the data from the data memory.
- WB: Write the data back to the register file.

Observation of single cycle processor

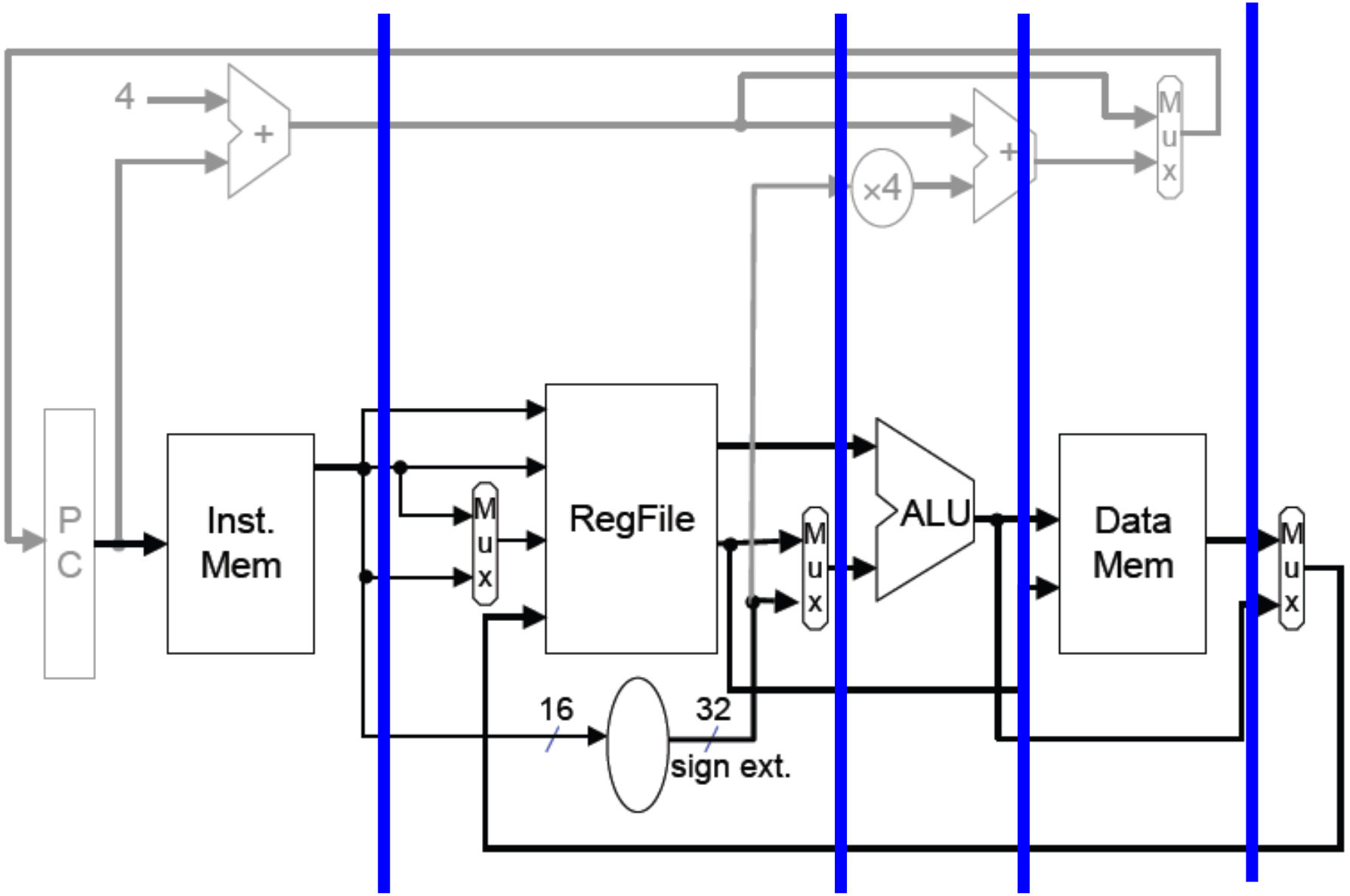
- Each step of LW uses distinct functional unit.
- Most of functional units are idle during the execution.
  - E.g. ALU is idle during IF, ID, etc.
- Put slackers back to work.



How Can We Make It Faster?

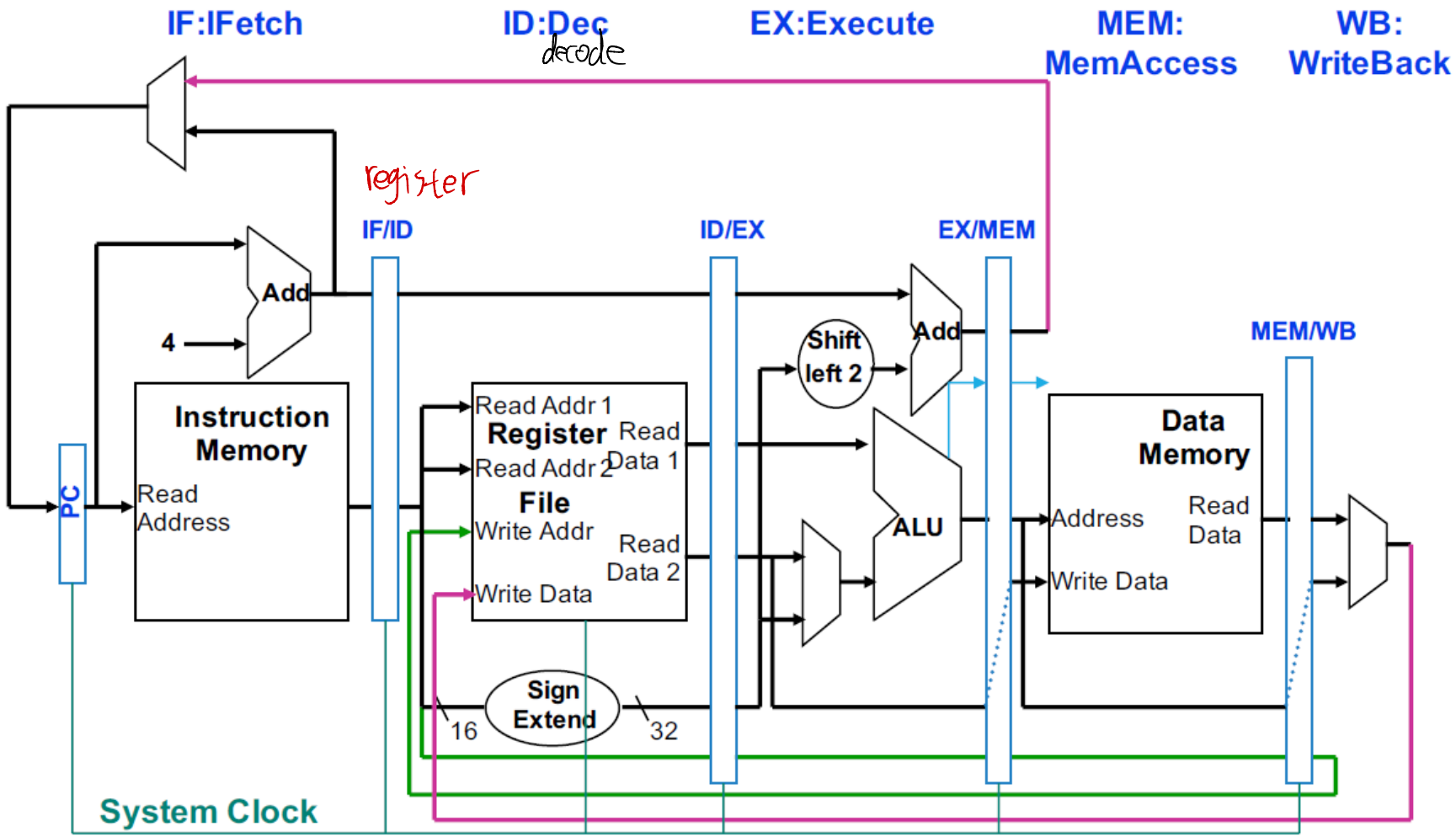
- Start fetching and executing the next instruction before the current one has completed.
  - Pipelining – modern processors are almost pipelined for performance.
  - Remember the performance equation:
    - CPU time = IC * CPI * Clock Cycle time
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages.

Single cycle processor: partitioning



MIPS Pipeline Datapath Additions

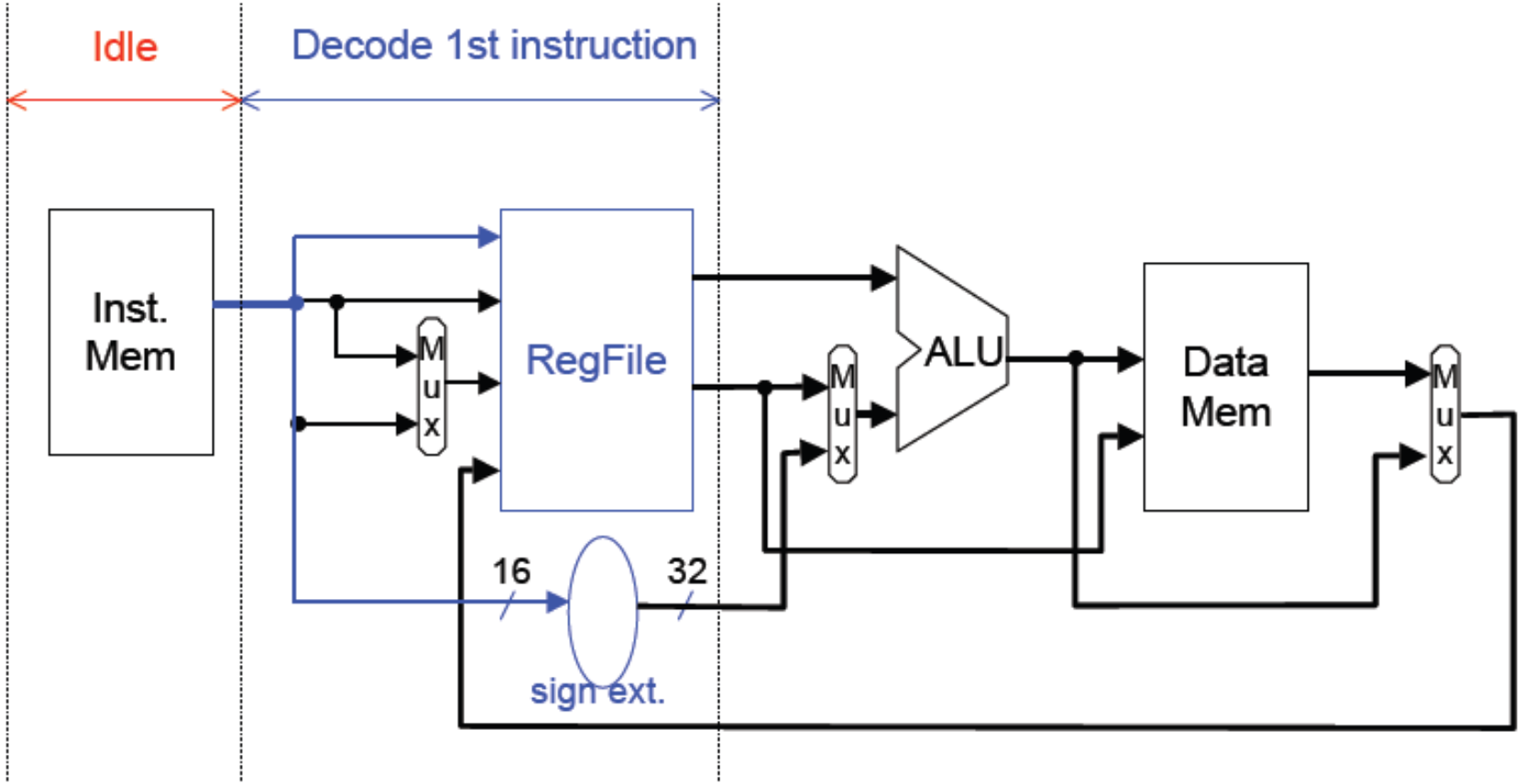
- State registers between each pipeline stage to isolate them.



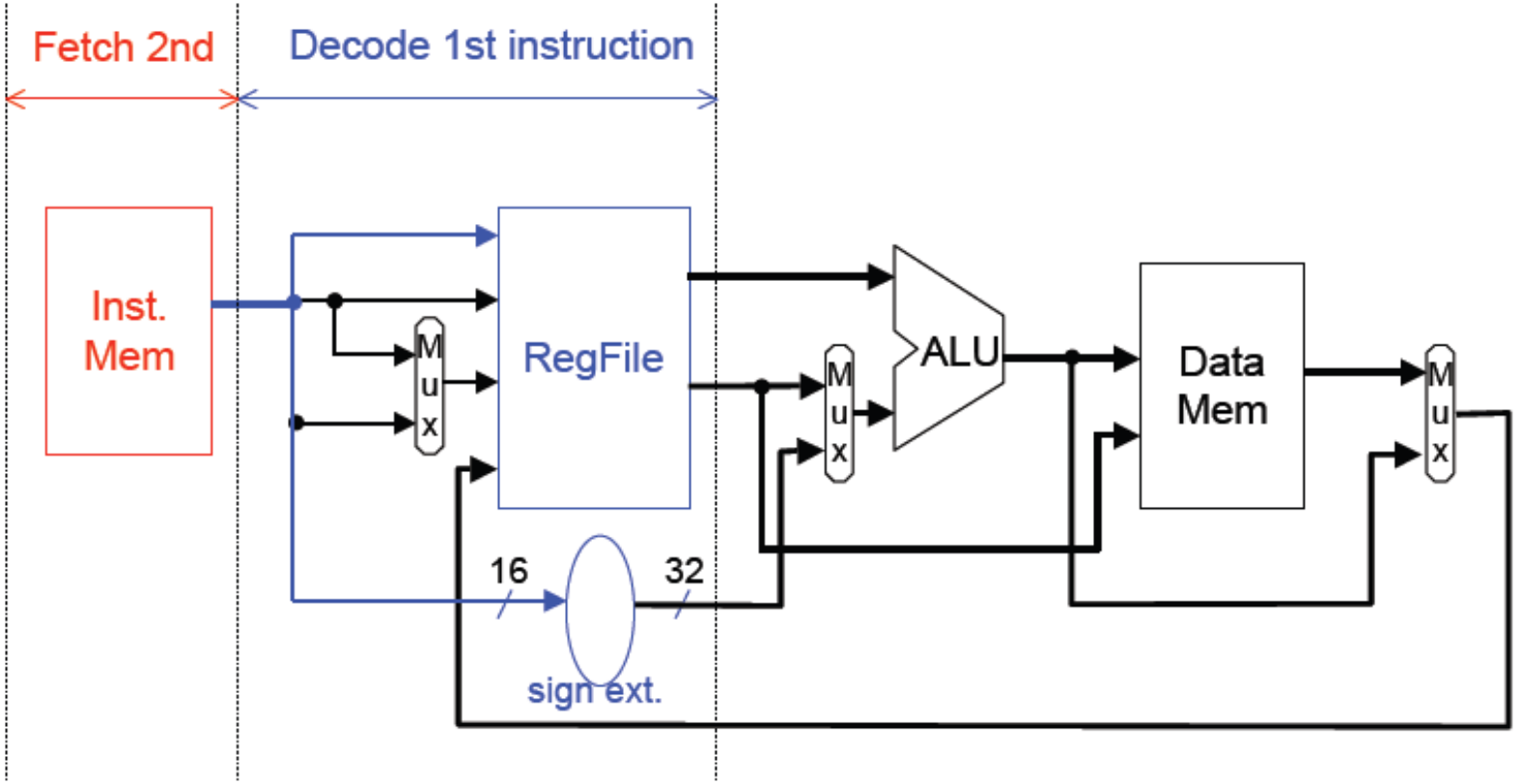
Pipeline Design

- Can we fetch the 2nd LW instruction when decoding the 1st LW instruction?

Yes, we can

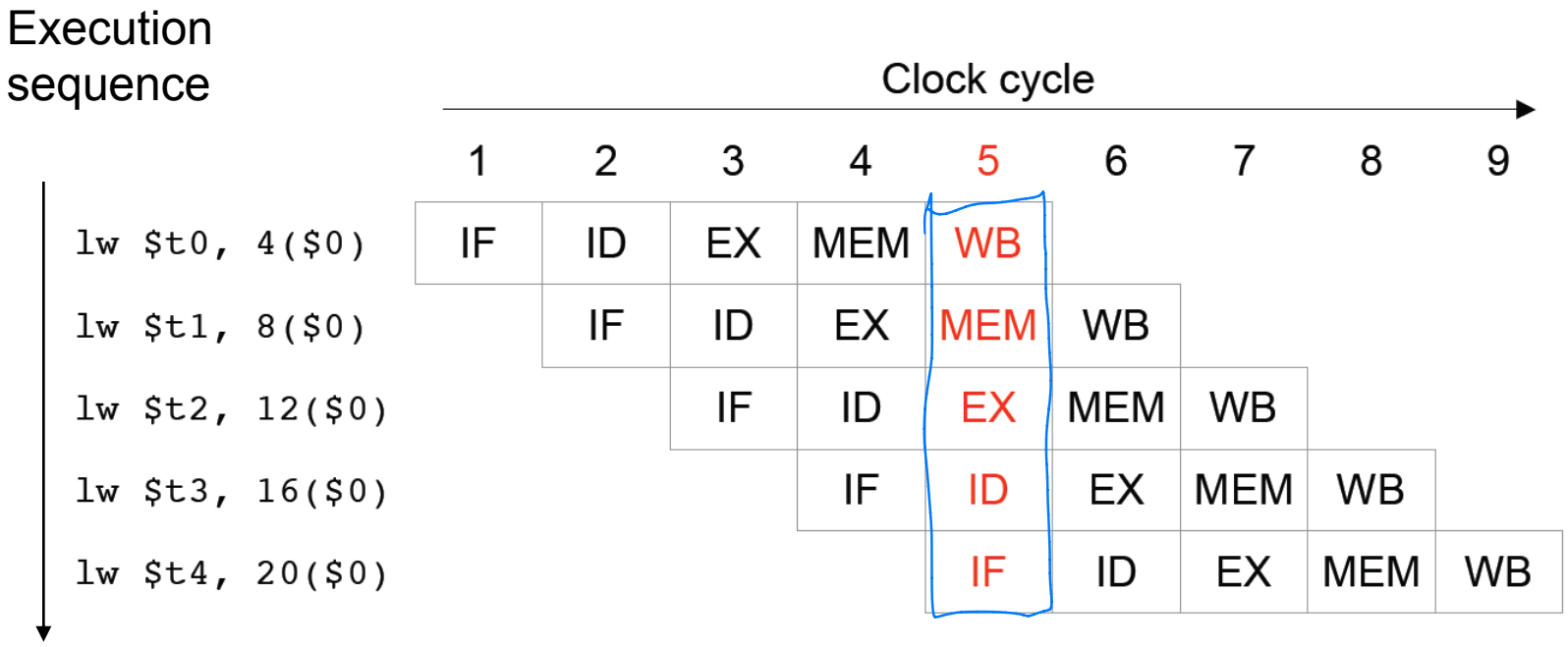


IF and ID together



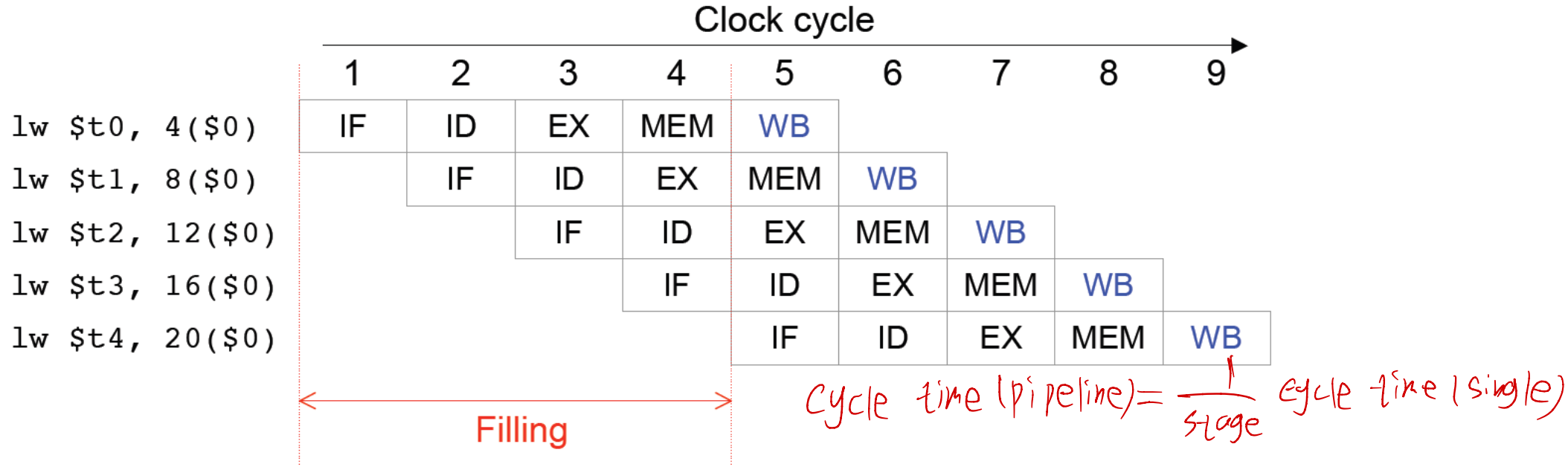
Pipelining LW instructions

- Pipeline depth is the number of stages, in here, 5.
- Cycle 1 – 4 : the pipeline is filing, as there are unused functional units.
- Cycle 5: the pipeline is full, all functional units are used.
- Cycle 6 – 9: the pipeline is flushing (emptying).



Pipeline execution diagram.

Pipelined processor



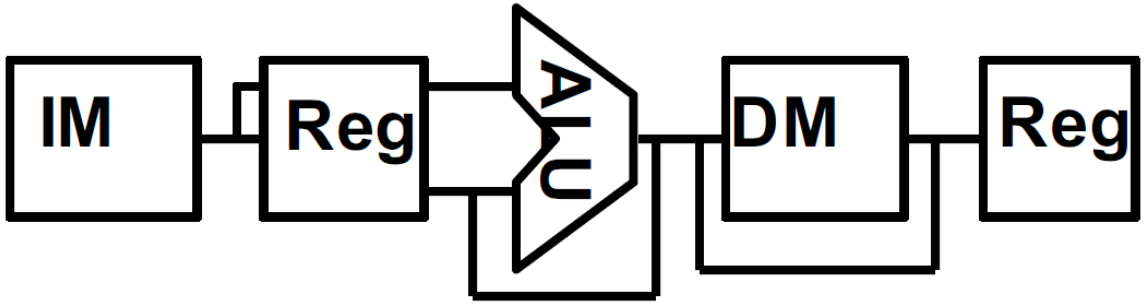
- Time on filling the pipeline (assume cycle time = 200ps)
  - 4 cycles = 4 x 200ps 800ps
- Time on pipeline is full and emptying
  - 10000 cycles = 10000 x 200ps
- Total time
  - 10004 x 200ps ≈ 2M ps
- Speedup? Why cycle time is 200ps?

Once the pipeline is full, one instruction is completed every cycle, so CPI = 1.

A Pipelined MIPS Processor

- Start the next instruction before the current one has completed.
  - improves throughput - total amount of work done in a given time.
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced.
- clock cycle (pipeline stage time) is limited by the **slowest** stage.
  - for some stages don't need the whole clock cycle (e.g., WB).
  - for some instructions, some stages are wasted cycles (i.e., nothing is done during MEM and WB cycle for beq instruction).

Graphically Representing MIPS Pipeline

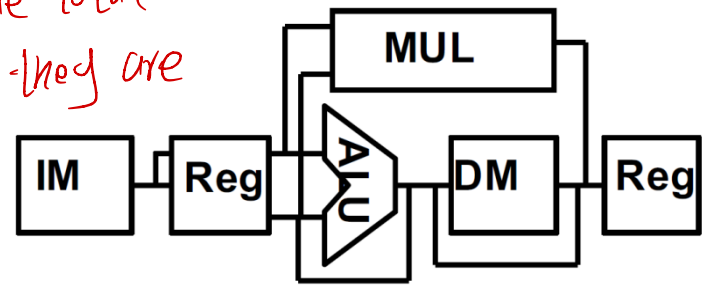


- Can help with answering questions like:
  - How many cycles does it take to execute this code?
  - What is the ALU doing during cycle 4?
  - Is there a hazard, why does it occur, and how can it be fixed?

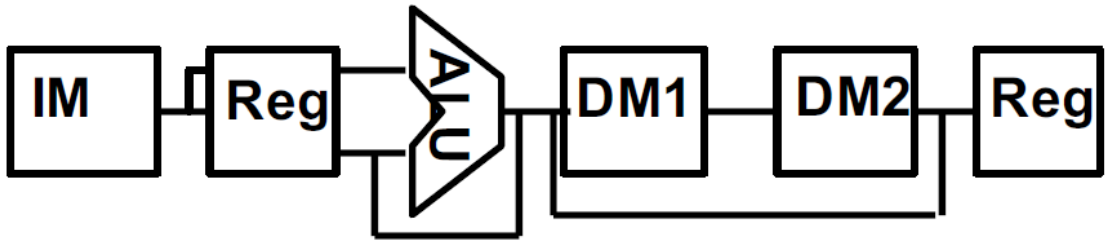
Other Pipeline Structures Are Possible

- What about the (slow) multiply operation?
  - Make the clock twice as slow or …
  - let it take two cycles (since it doesn't use the DM stage)

to separate the total time such that they are close.



- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)

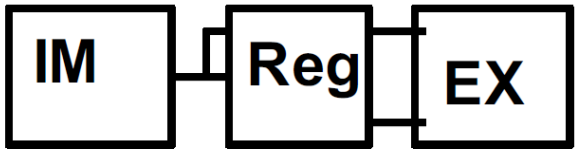


Other Sample Pipeline Alternatives

- ARM7

for mobile phones

simple hardware → complex compiler

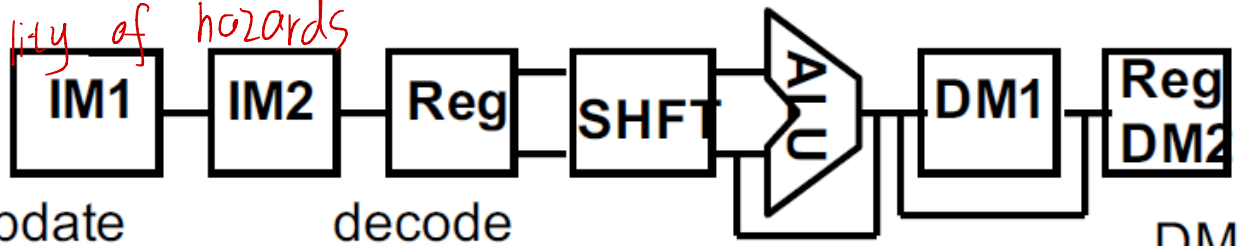


PC update	decode	ALU op
IM access	reg	DM access
	access	shift/rotate
		commit result
		(write back)

single task → high correlation → high possibility of hazards

- XScale

more stages → high possibility of hazards



PC update				DM write
BTB access	decode			reg write
start IM access	reg 1 access	ALU op		
		shift/rotate	start DM access	
IM access		reg 2 access	exception	

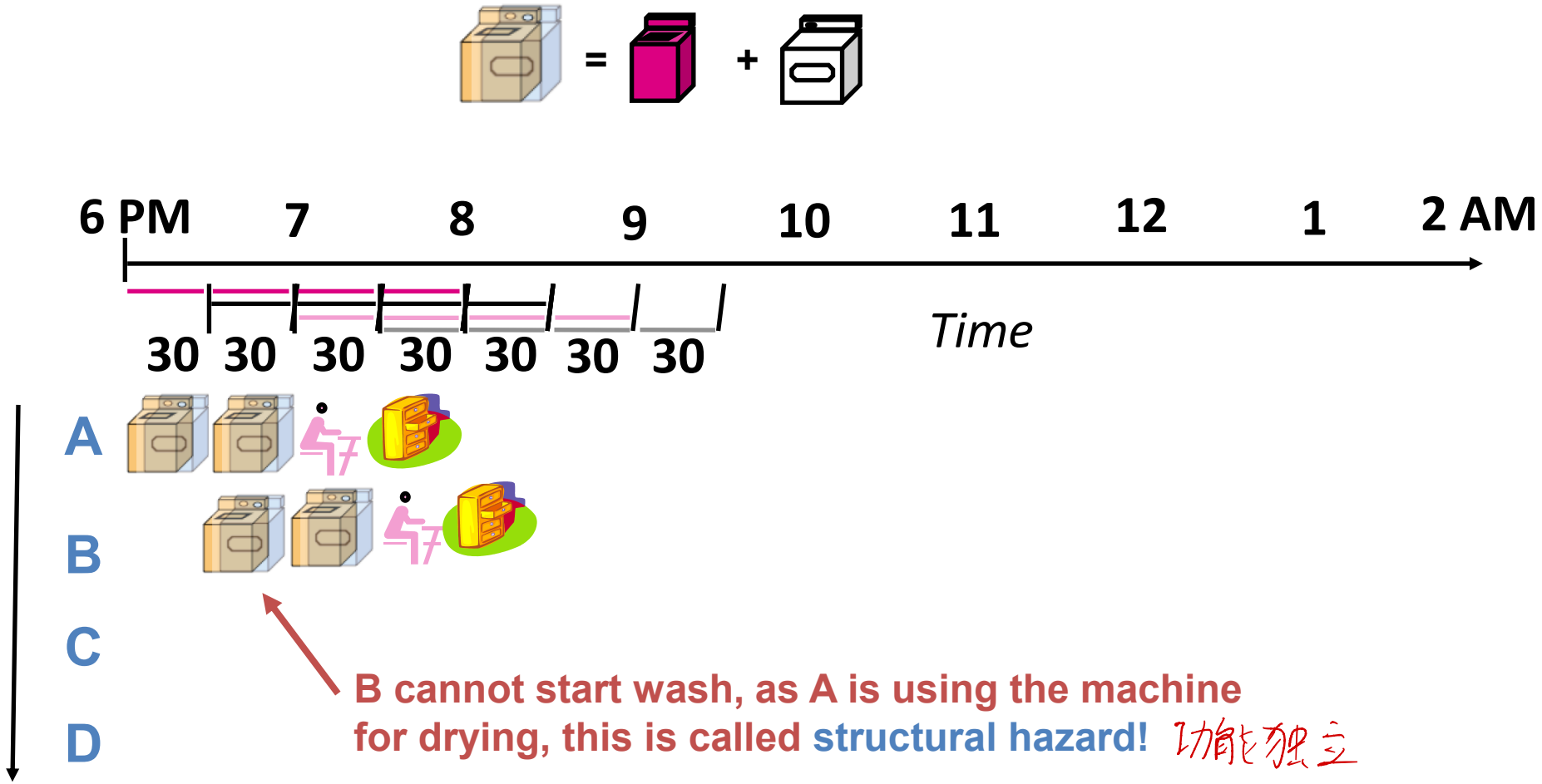
Can Pipelining Get Us Into Trouble?

- Yes: Pipeline Hazards
  - structural hazards:
    - a required resource is busy
  - data hazards:
    - attempt to use data before it is ready
  - control hazards:
    - deciding on control action depends on previous instruction

- Can usually resolve hazards by waiting
  - pipeline control must detect the hazard
  - and take action to resolve hazards

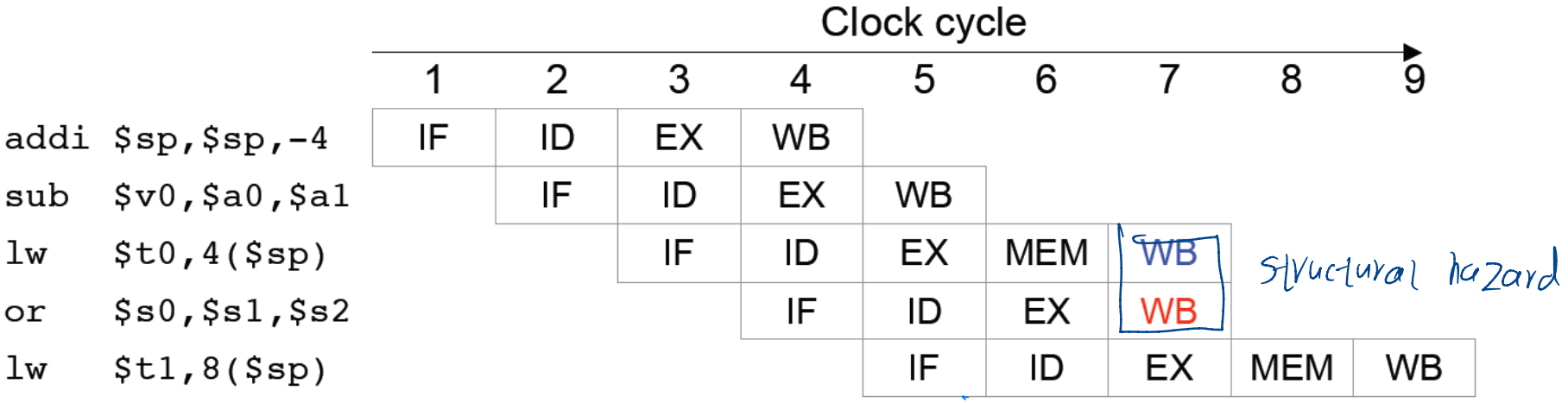
Laundry example

- What happen if we only have one machine which combines washing and drying functions?



Structural Hazards

- Structural hazard: The hardware cannot support the combination of instructions that are set to execute in the given clock cycle.
- Structural hazard examples.
  - R-type instructions only require 4 stages: IF, ID, EX, and WB.
    - Load uses Register File's Write Port during its 5th stage.
    - R-type uses Register File's Write Port during its 4th stage

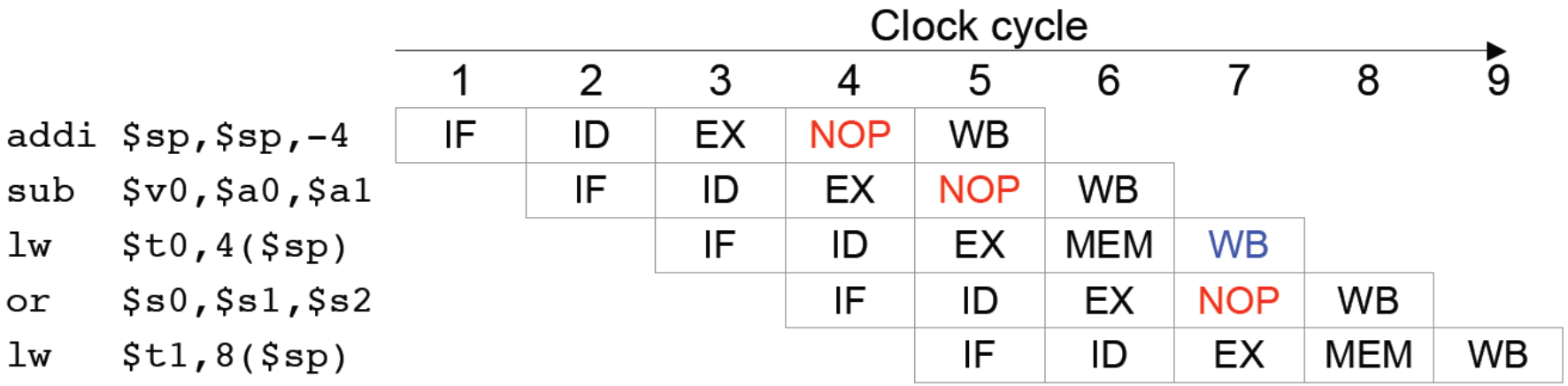


Attempt to use the same resource in two different ways at the same time.
However, the hardware cannot support two writes simultaneously.

Resolve Structural Hazards 1

- Enforce uniformity 強制等长
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions

R-Type	IF	ID	EX	NOP	WB



  - sw and beq have NOP stages too …

sw	IF	ID	EX	MEM	NOP

beq	IF	ID	EX	NOP	NOP

Structural Hazards

- There are other conflicts for use of a resource.
- In MIPS pipeline with a single memory
  - Load/store requires data access
    - Hence, pipeline datapaths require separate instruction/data memories, or separate instruction/data cachesinstruction/data memories.
    - Harvard Model.
  - Instruction fetch requires instruction access
    - Since Register File.

Resolve Structural Hazard 2

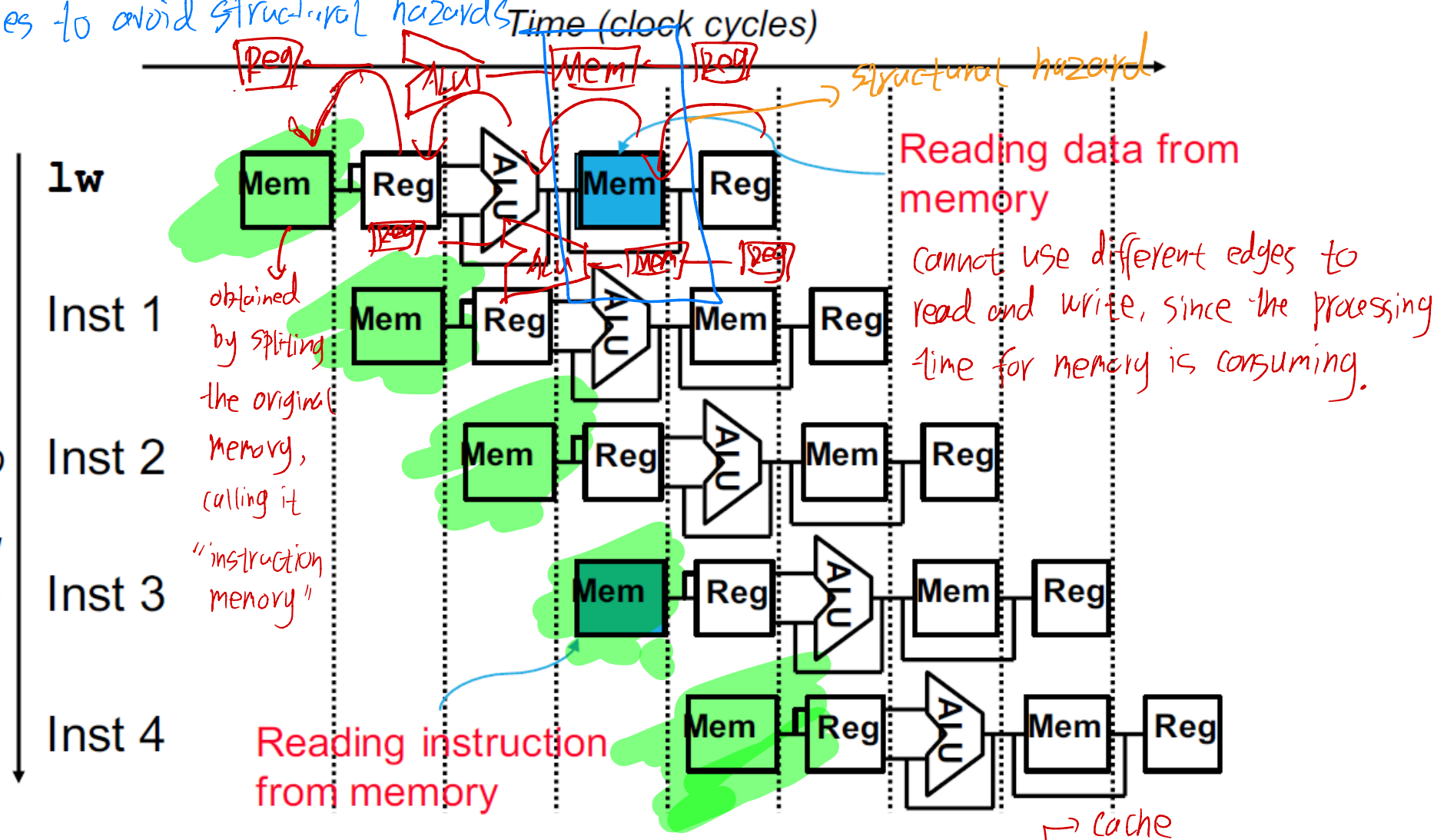
use two memories to avoid structural hazards

How the hazard comes when there is only a memory in the pipeline?

Let's remove the first memory appeared in the figure before the analysis, which is marked by the green color.

After the removal, shifting left all the remaining parts, we will obtain the results shown in the figure. Noting that there will be two memories used(for loading data and fetching the instruction) in one cycle, which causes a structural hazard.

Solution?
Split the original memory to form another new one to separate the instruction and data caches so that we can fetch the instruction and load data simultaneously.



- Fix with separate instr and data memories (I$ and D$)

Resolve Structural Hazard 3

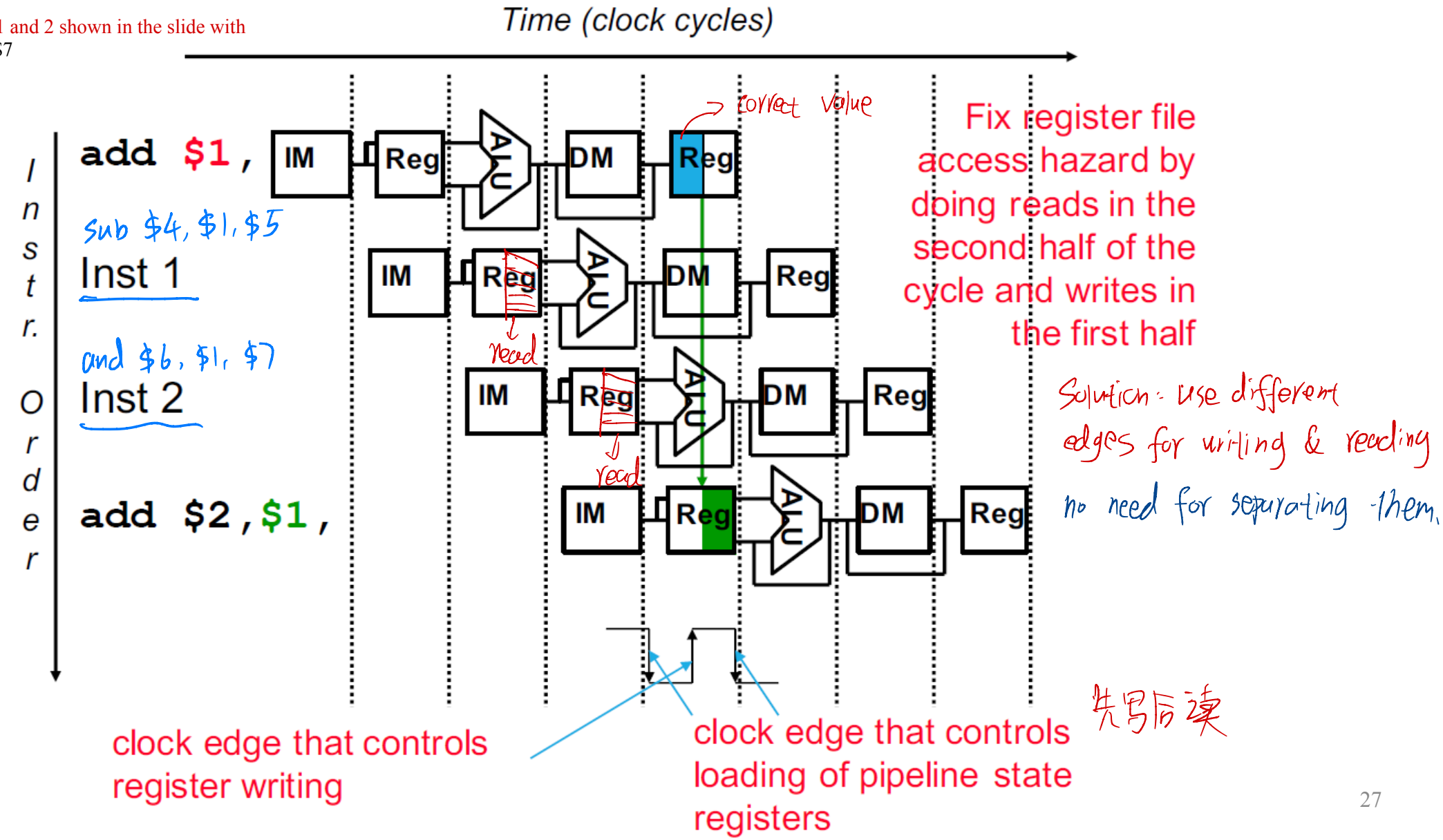
Prerequisite: All parts in the pipeline follow the rule of read-after-write data dependency.

Assume that if we replace Inst 1 and 2 shown in the slide with
sub $4, $1, $5 and and $6, $1, $7

Actually, the structural hazard will occur if the instructions are executed as the following order.

Why?
Note that the value in the register of the first instruction can be available only the "writeback" stage is finished.

However, in Inst 1 and 2, we will obtain a wrong value of register $1 when the correct one is still unavailable. It means that a structural hazard happened before executing the fourth instruction where the register can get the correct value of $1.

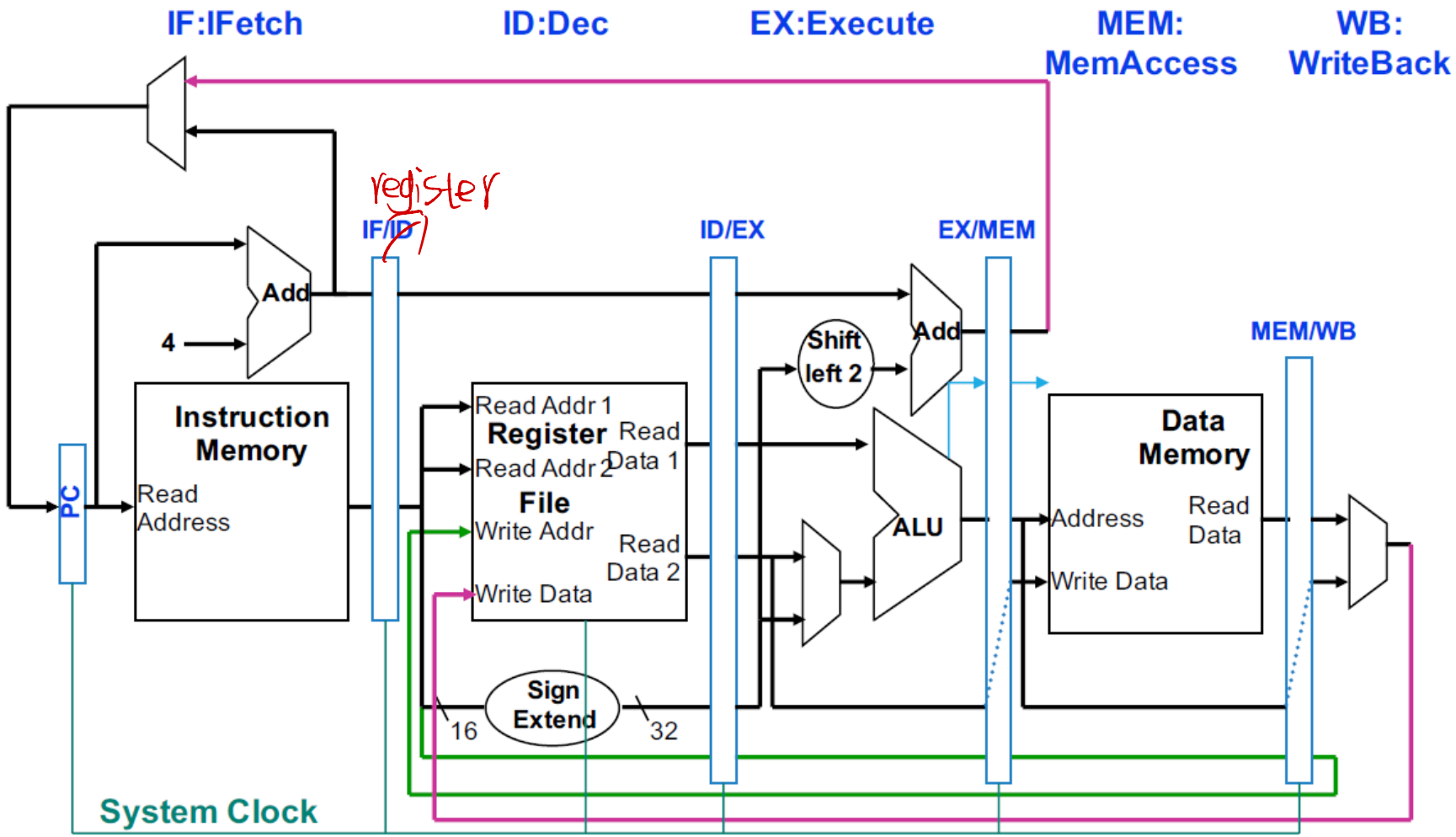


Pipelining the MIPS ISA

- What makes it easy
  - all instructions are the same length (32 bits)
    - can fetch in the 1st stage and decode in the 2nd stage
  - few instruction formats (three) with symmetry across formats
    - can begin reading register file in 2nd stage
  - memory operations occur only in loads and stores
    - can use the execute stage to calculate memory addresses
  - each instruction writes at most one result (i.e., changes the machine state) and does it in the last few pipeline stages (MEM or WB)
  - operands must be aligned in memory so a single data transfer takes only one data memory access

MIPS Pipeline Datapath Additions

- State registers between each pipeline stage to isolate them.

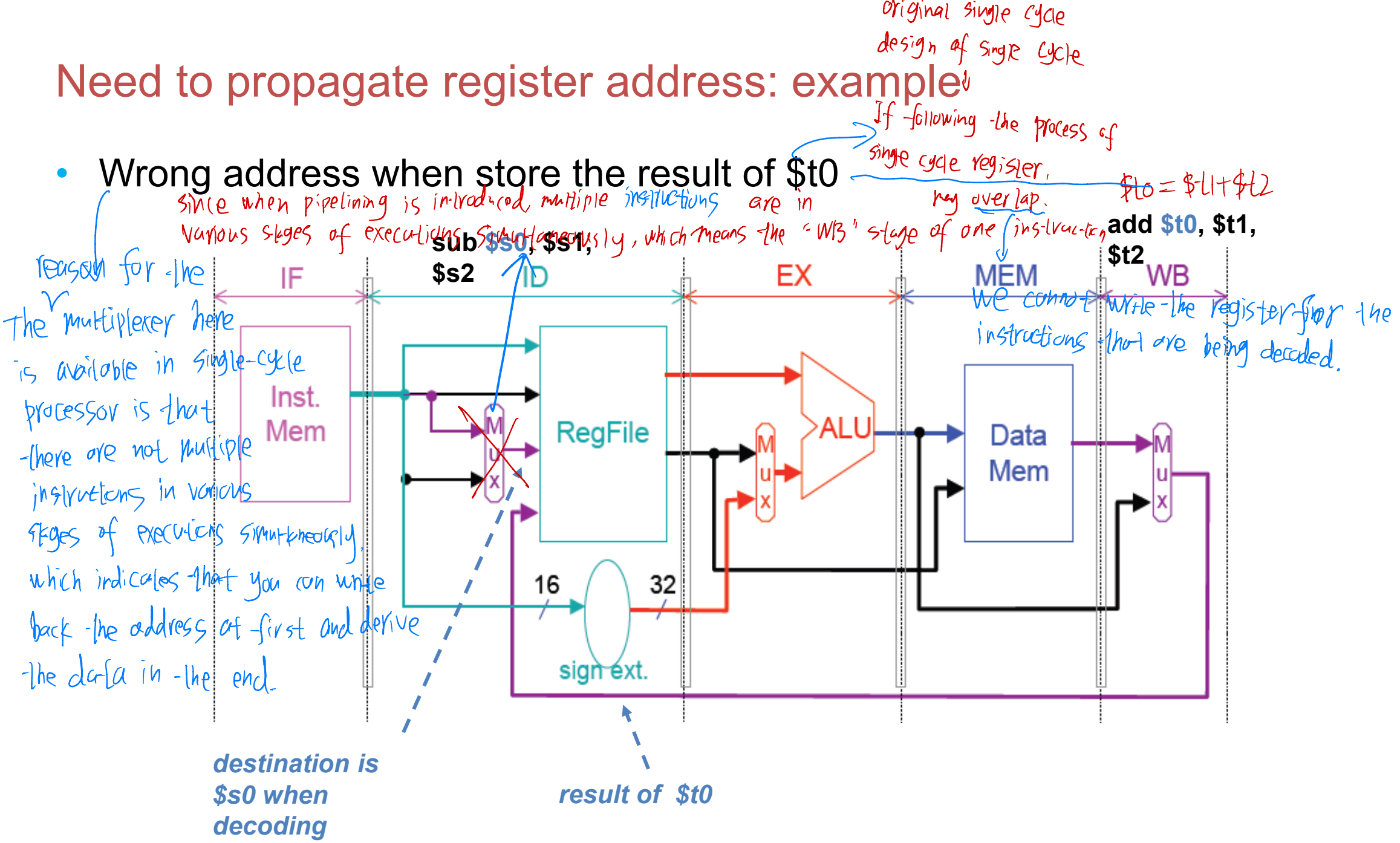


Value Propagation

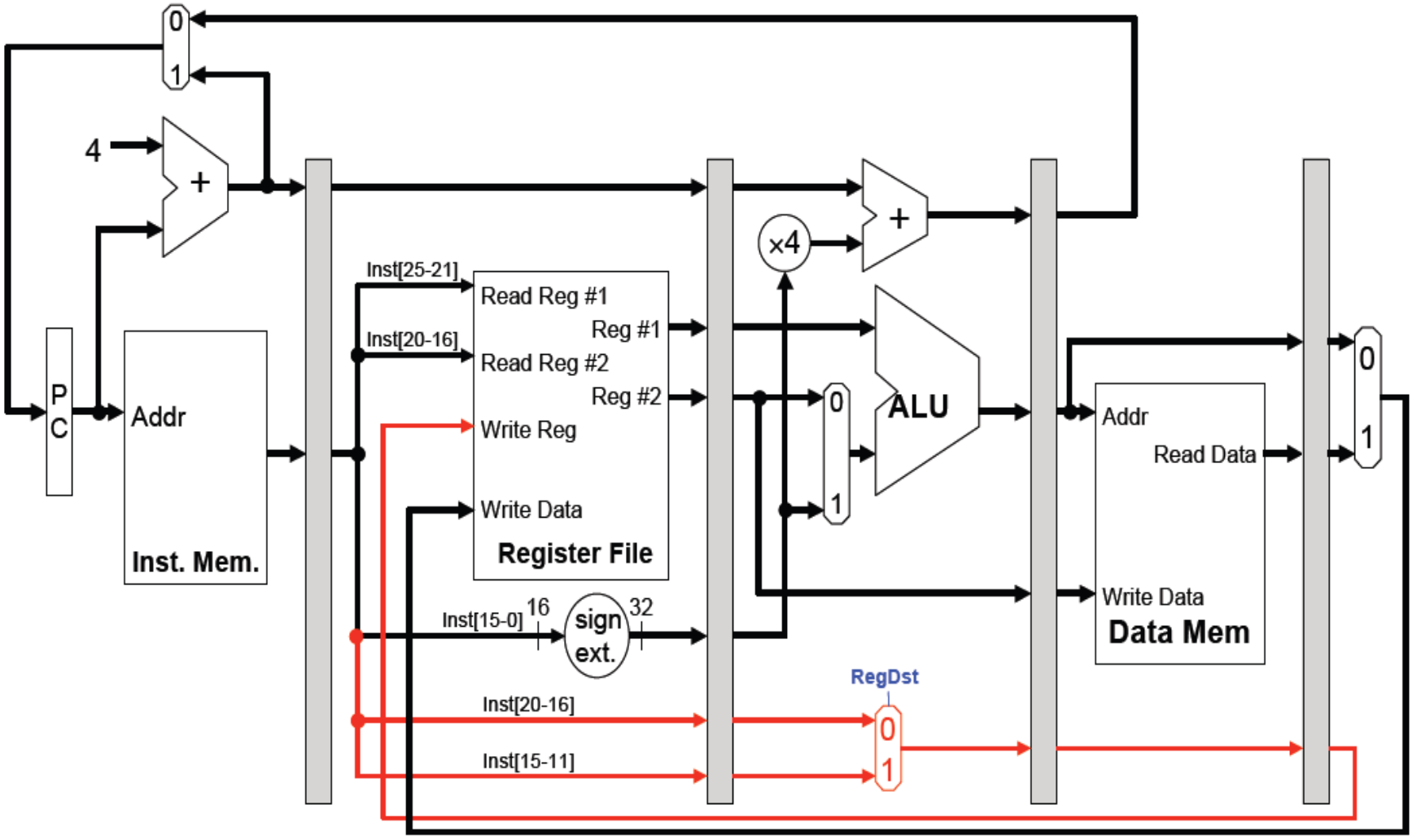
- Data propagation
  - Any data required in later stages must be propagated through the pipeline register.
  - E.g. data from R[rs], R[rt] must be propagated to the ALU through the ID/EX register.
  - E.g. for SW instruction, data from R[rt] must be propagated to the memory through the ID/EX and EX/MEM registers.
- Register address propagation
  - Propagate register address
  - ID stage decodes the address of R[rd], this address must be propagated to WB stage for writing data.

Need to propagate register address: example

- Wrong address when store the result of $t0



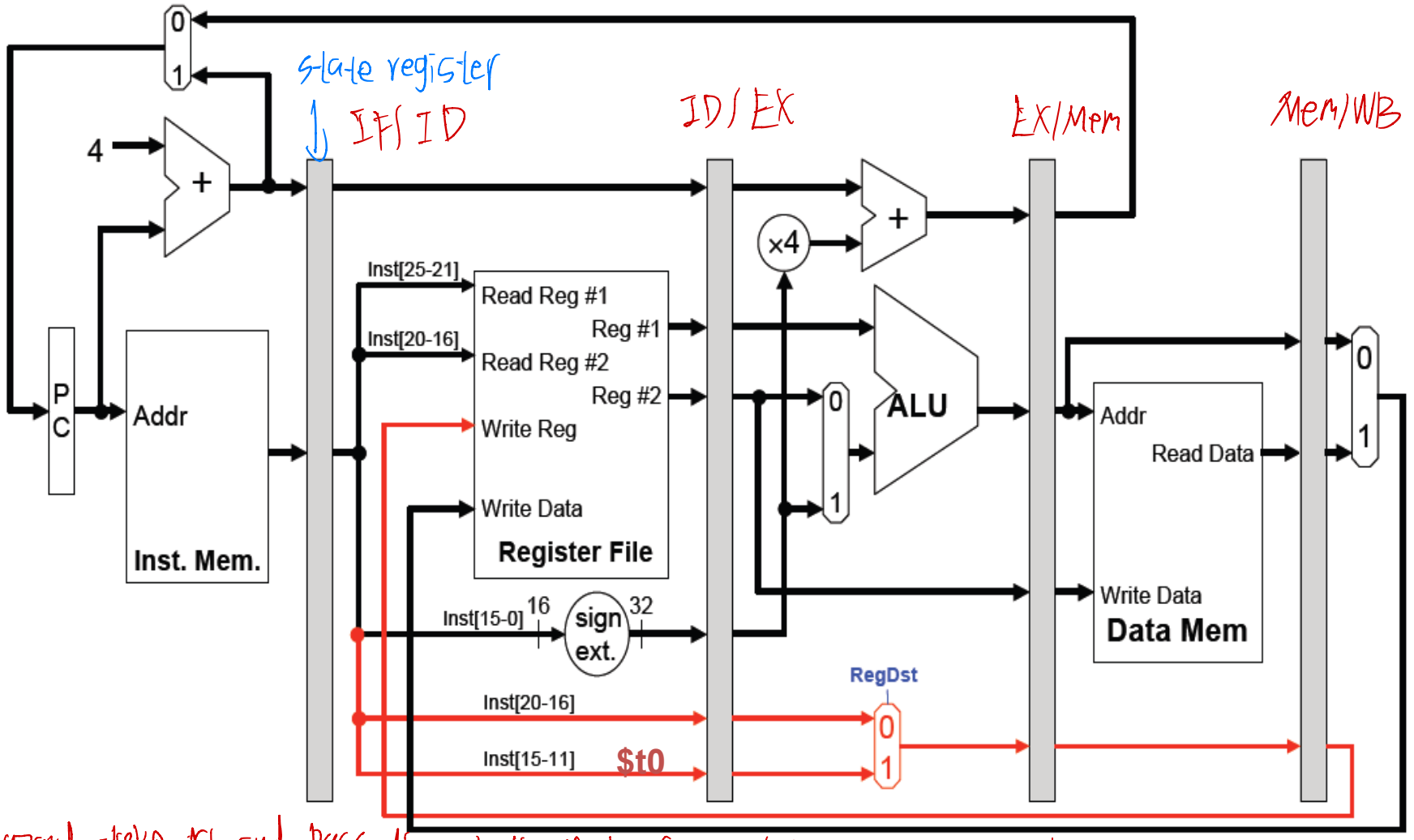
Propagate register address



result of $t0

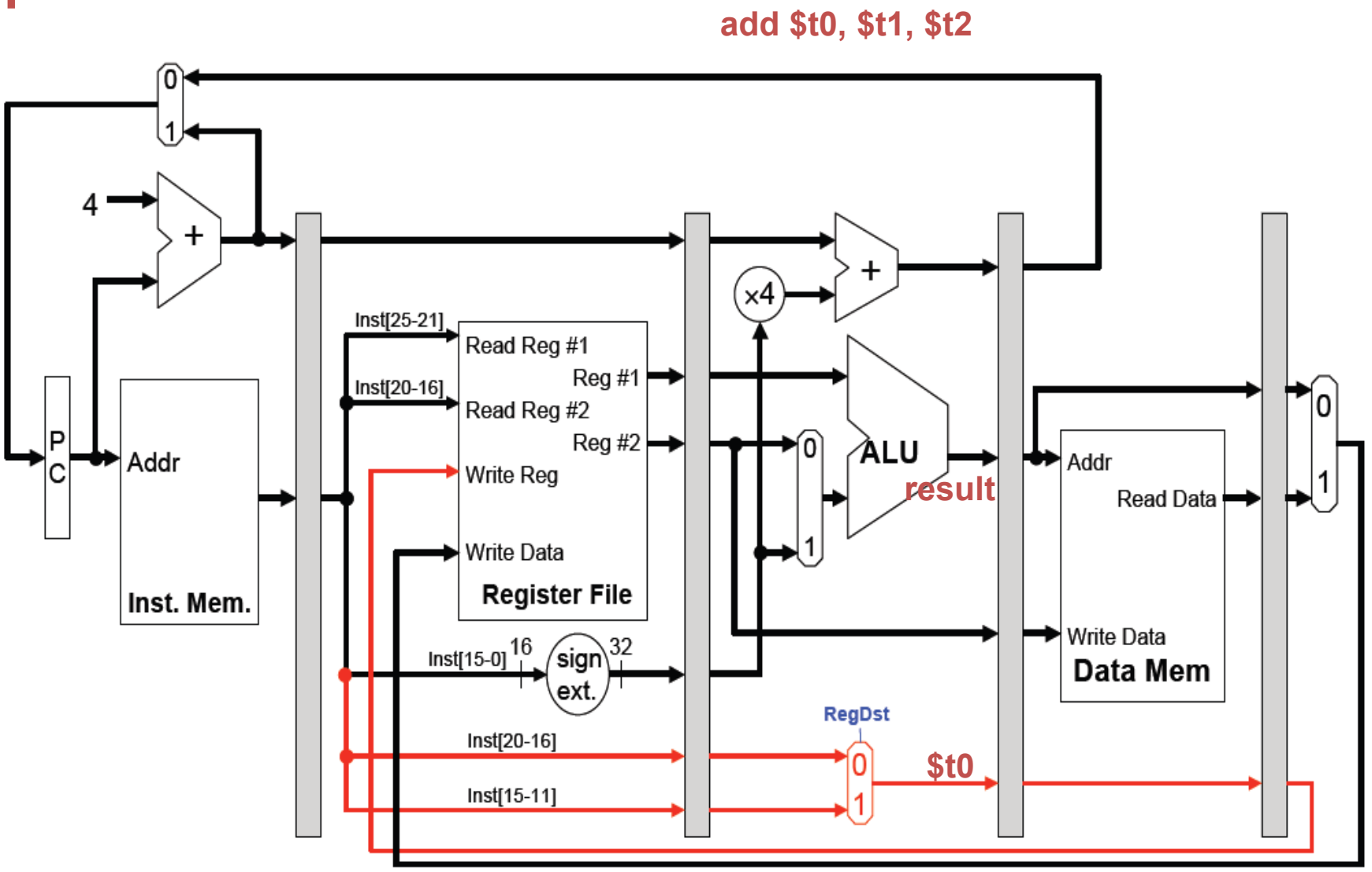
Example: ID

add $t0, $t1, $t2

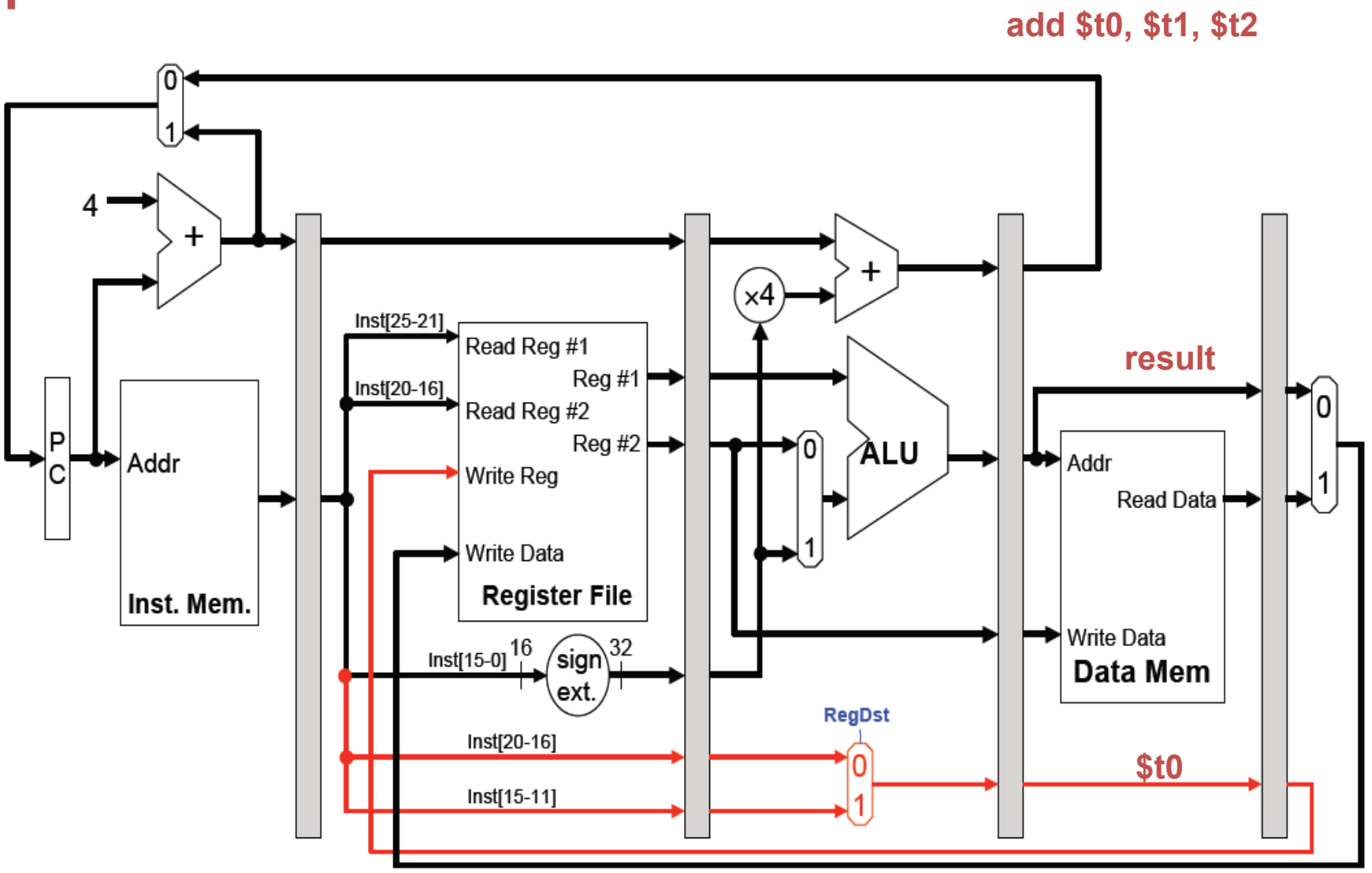


To resolve this hazard, take it and pass through the rest of 3 state registers to change its state to "writeback" stage so that we can make sure obtain correct address after "writeback" stage.

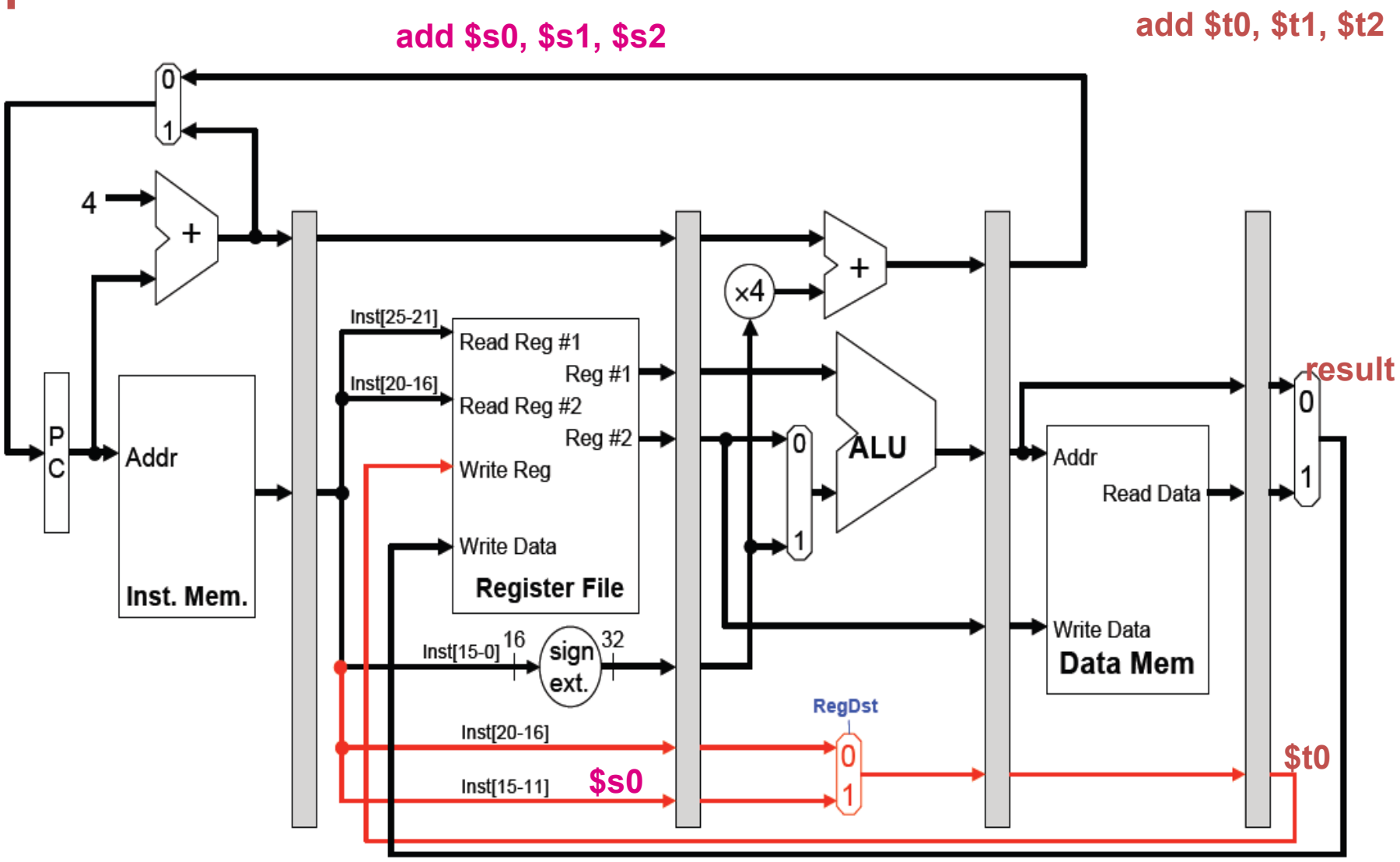
Example: EX


add $t0, $t1, $t2
0
1
4
+
PC
Addr
Inst. Mem.
Inst[25-21]
Inst[20-16]
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
Inst[15-0]
16
sign ext.
32
×4
+
0
1
ALU
result
Addr
Read Data
Write Data
Data Mem
0
1
RegDst
Inst[20-16]
Inst[15-11]
0
1
$t0

Example: MEM


add $t0, $t1, $t2
0
1
4
+
PC
Addr
Inst. Mem.
Inst[25-21]
Inst[20-16]
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
Inst[15-0]
16
sign ext.
32
×4
+
0
1
ALU
result
Addr
Read Data
Write Data
Data Mem
RegDst
Inst[20-16]
Inst[15-11]
$t0

Example: WB

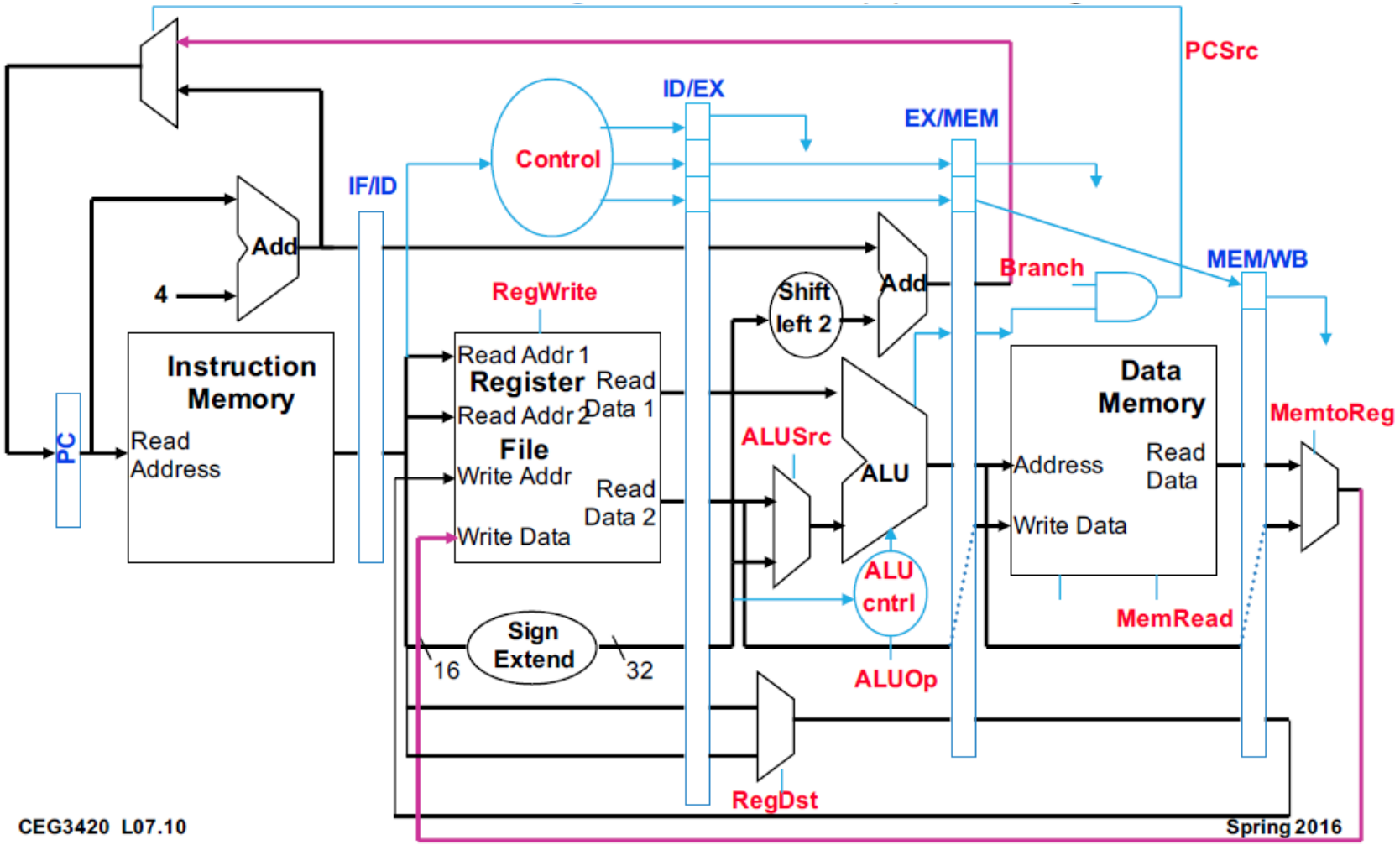

add $s0, $s1, $s2
add $t0, $t1, $t2
0
1
4
+
PC
Addr
Inst. Mem.
Inst[25-21]
Inst[20-16]
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
Inst[15-0]
16
sign ext.
32
×4
+
0
1
ALU
Addr
Read Data
Write Data
Data Mem
result
RegDst
Inst[20-16]
Inst[15-11]
$s0
$t0

Propagation of control signals

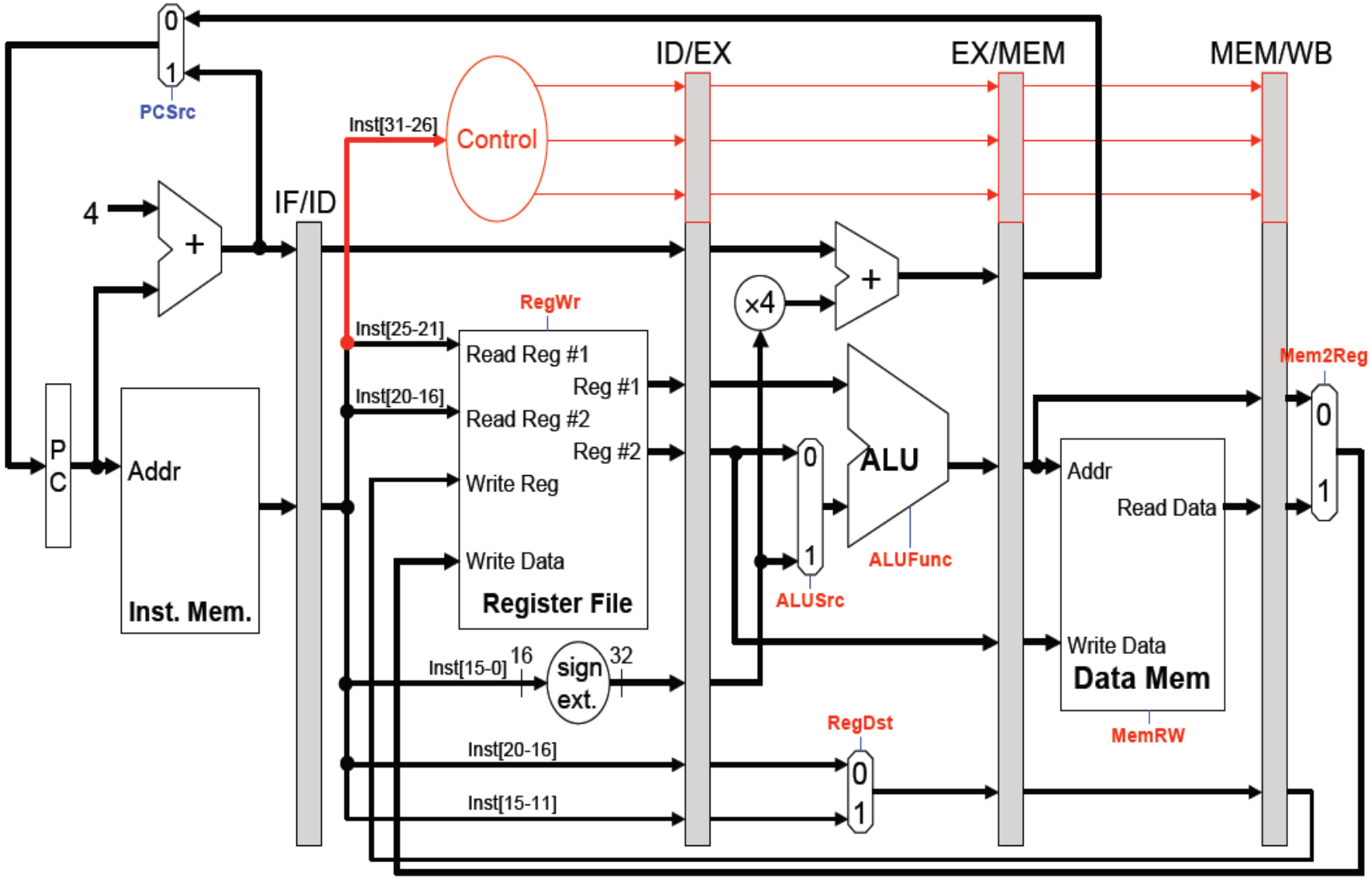
- The control signals are generated in the same way as in the single-cycle processor.
  - After an instruction is fetched, the processor decodes it and produces the appropriate control values in ID stage.
- Some control signals will not be used until later stages.
  - These signals must be propagated through the pipeline until they reach the appropriate stage.
- We can just propagate them using the pipeline registers, along with the data.

MIPS Pipeline Control Path

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages

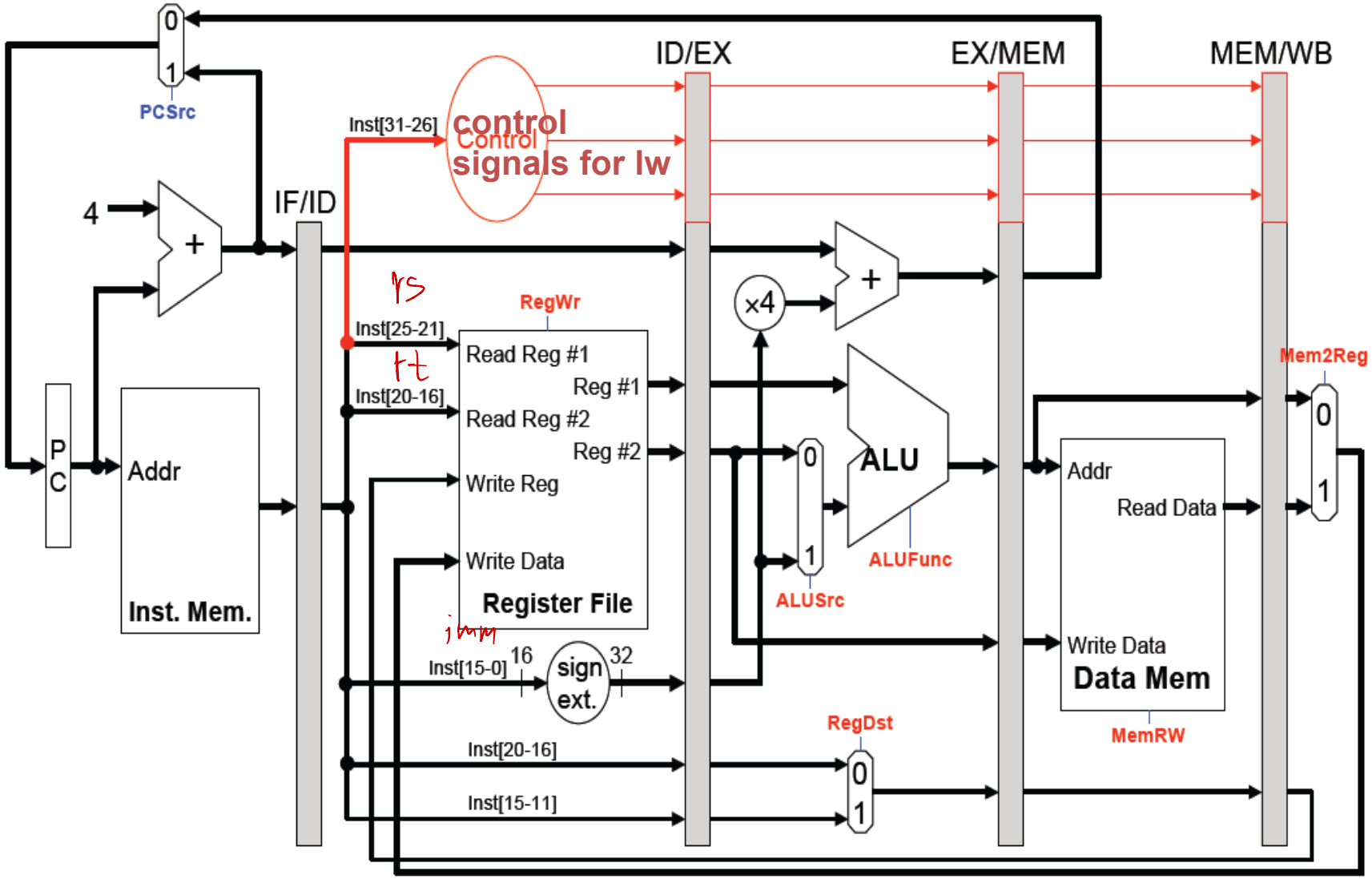


Propagate control signals

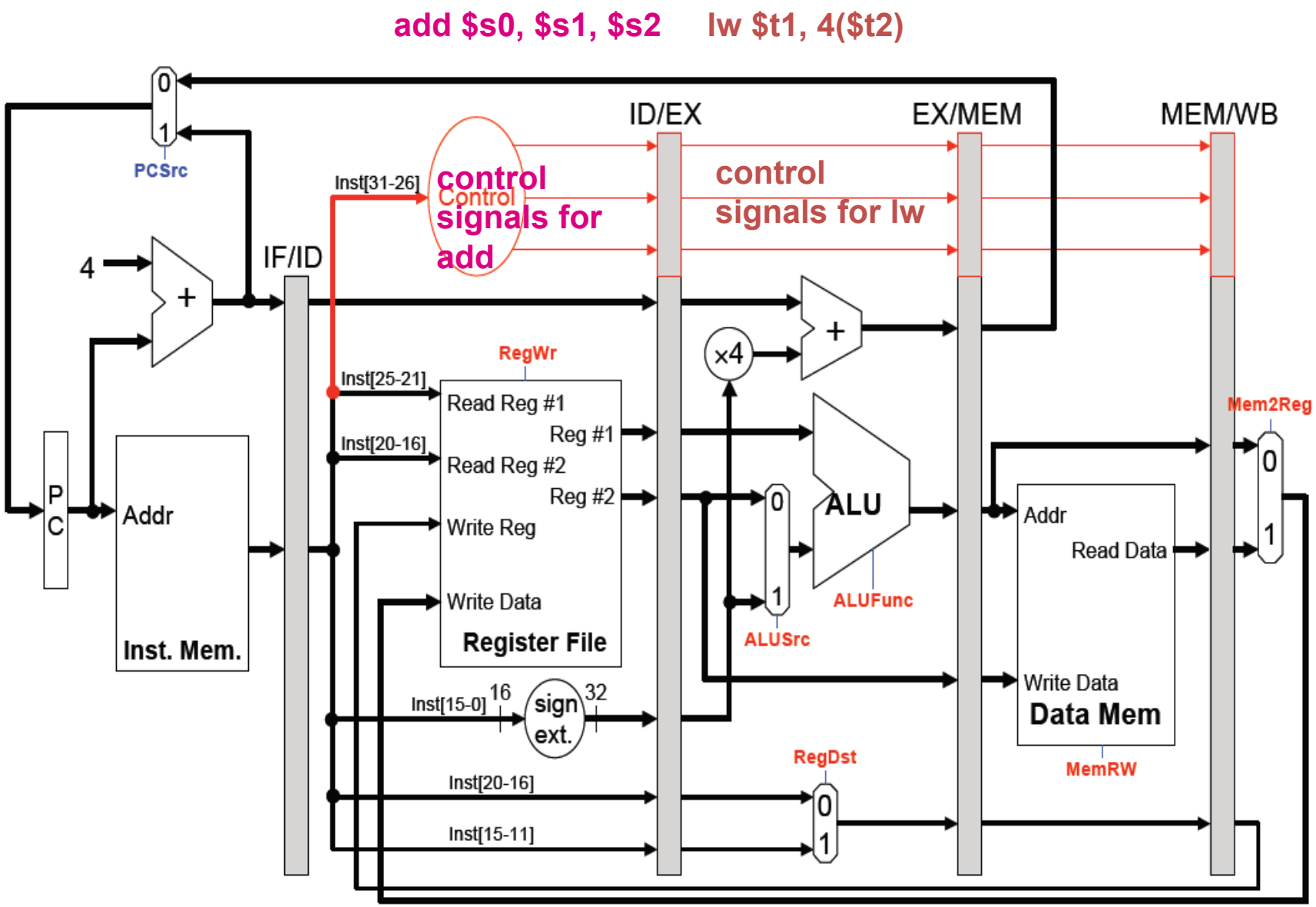


Propagate control signals

lw $t1, 4($t2)

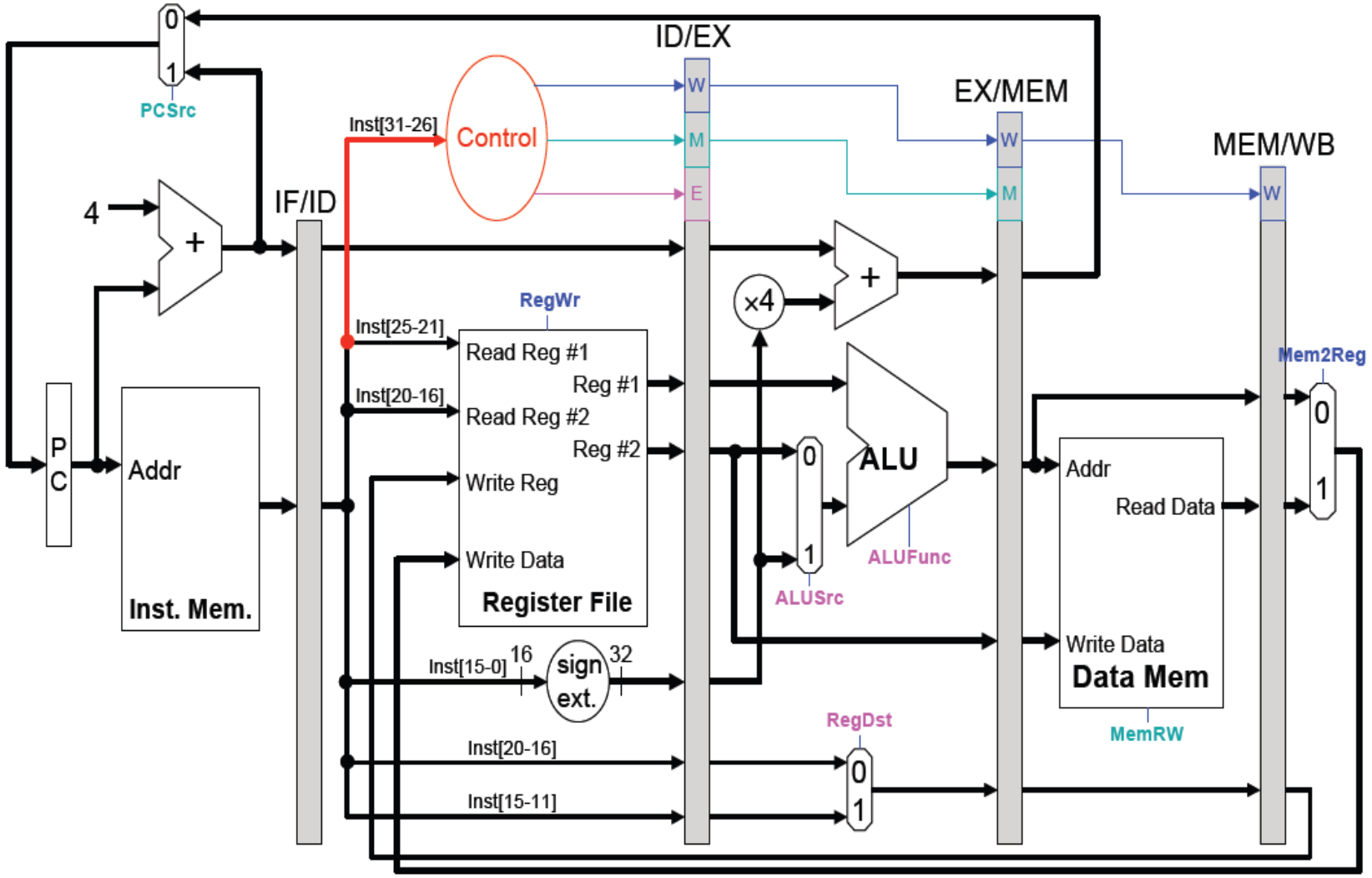


Propagate control signals



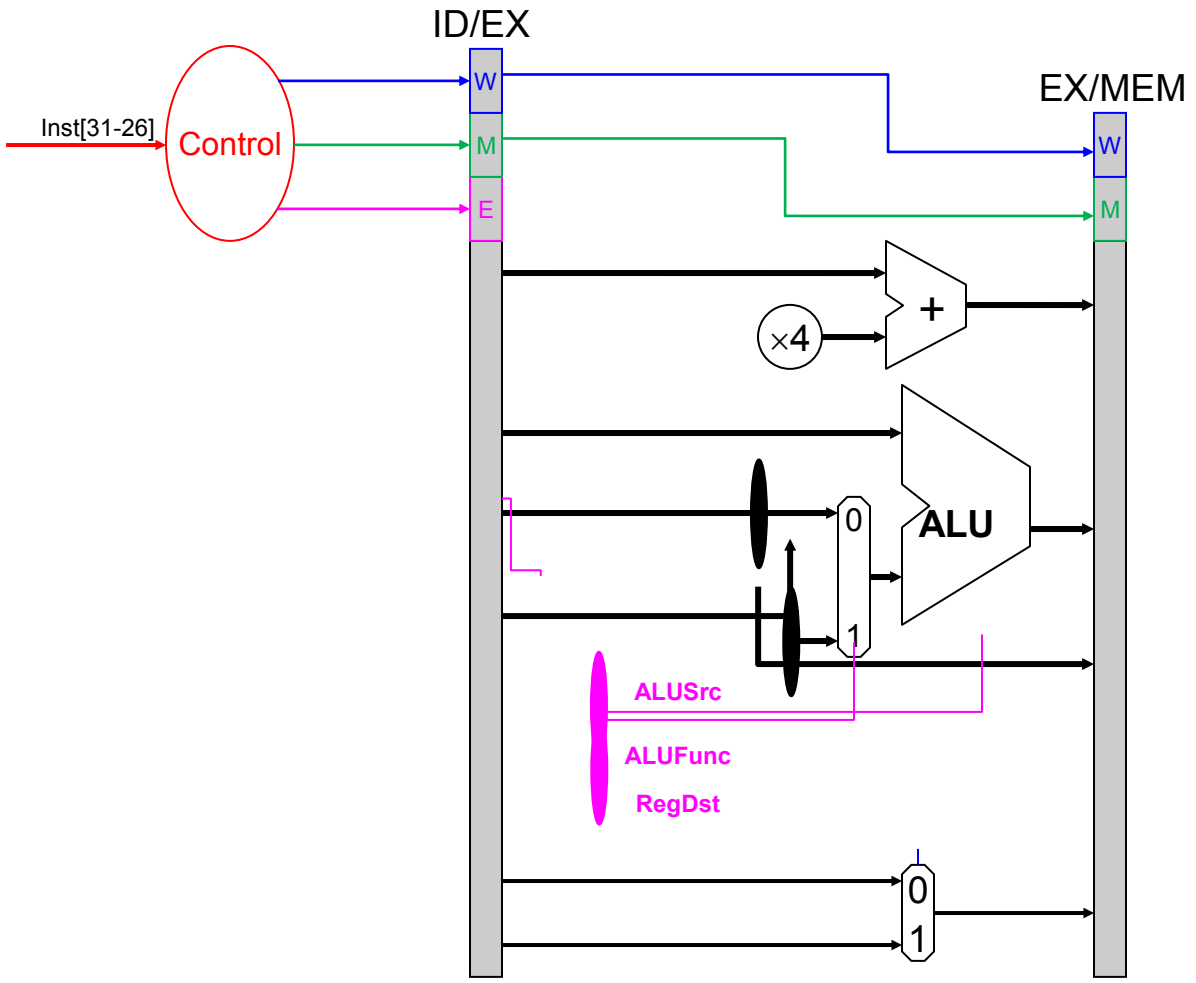
Propagate control signals: W, M, E

- **Not necessary to propagate all control signals from ID stage to WB stage. As a control signal is only used in one stage.**



Control signal usage: EX

- ALUsrc, ALUFunc, RegDst are only required (used) in EX stage.



Pipeline Control

- IF Stage: read Instr Memory (always asserted) and write PC (on System Clock)
- ID Stage: no optional control signals to set

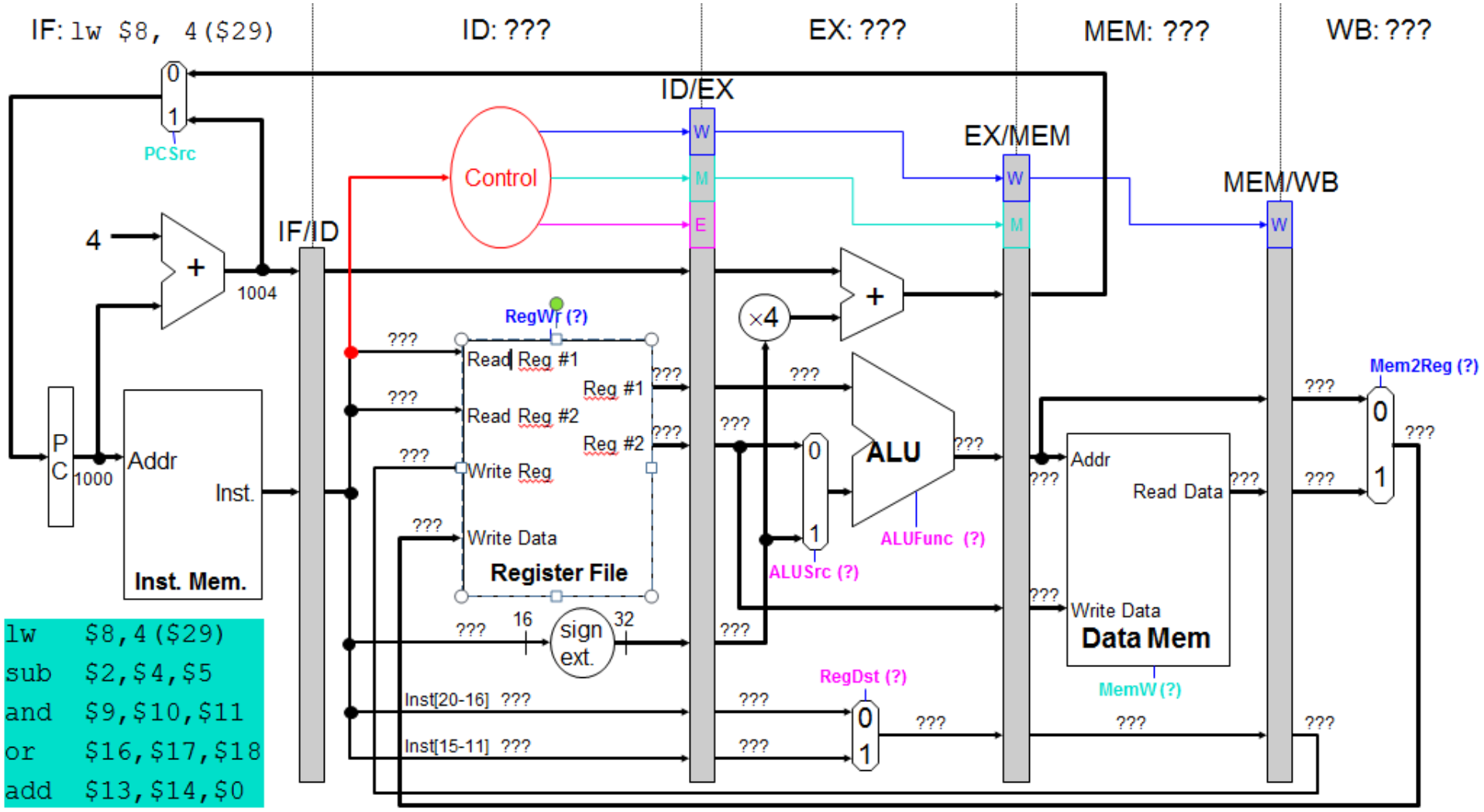
	EX Stage				MEM Stage			WB Stage	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Brch	Mem Read	Mem Write	Reg Write	Mem toReg
R	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	1	1	0	0	0	X

An example of pipeline execution

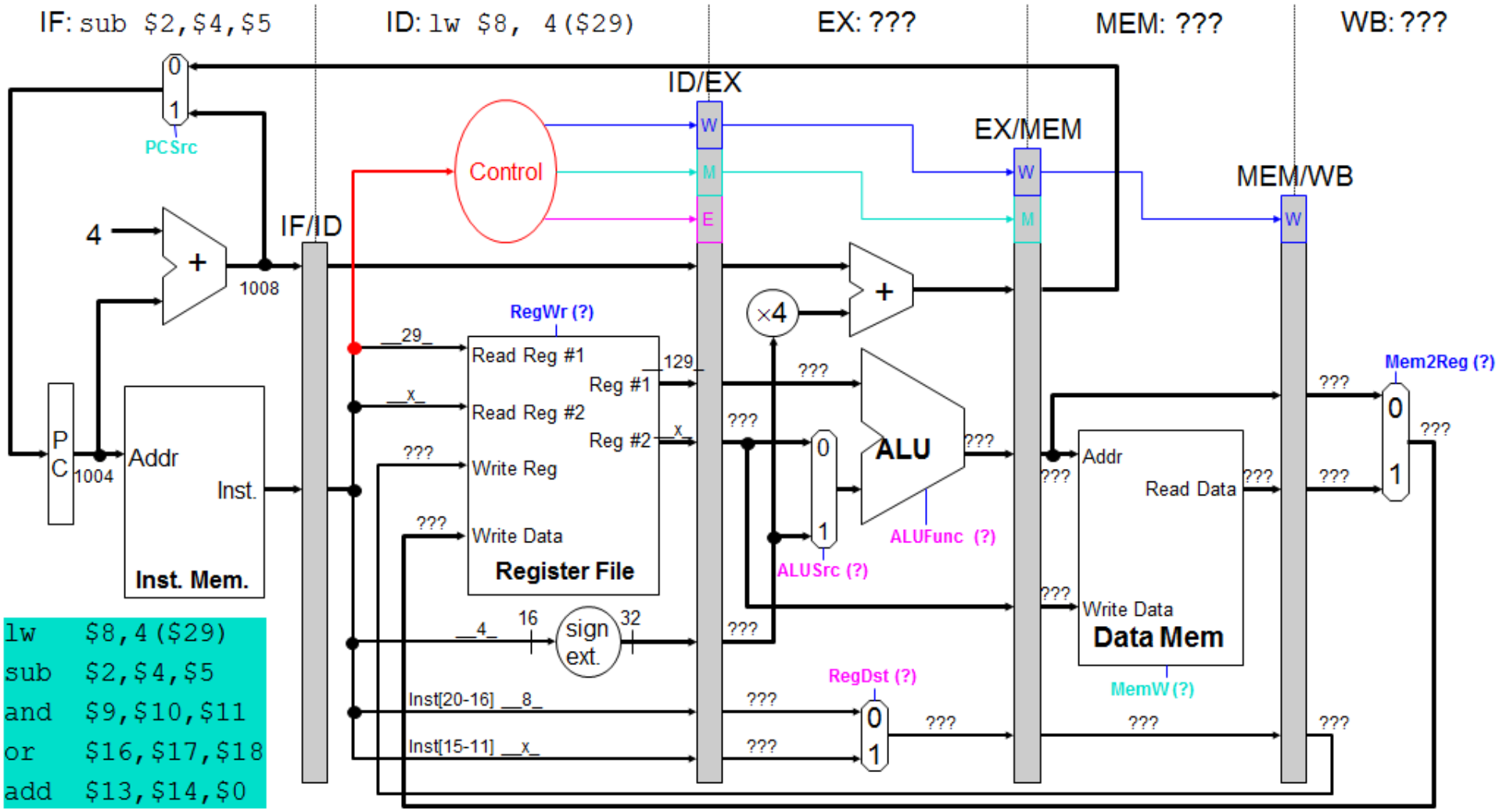
			Clock cycle								
			1	2	3	4	5	6	7	8	9
1000:	lw	$8,4($29)	IF	ID	EX	MEM	WB				
1004:	sub	$2,$4,$5		IF	ID	EX	MEM	WB			
1008:	and	$9,$10,$11			IF	ID	EX	MEM	WB		
1012:	or	$16,$17,$18				IF	ID	EX	MEM	WB	
1016:	add	$13,$14,$10					IF	ID	EX	MEM	WB

- Assumptions:
  - Each register contains a value which is equal to its register address plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - All data memory locations contain a constant value 99.

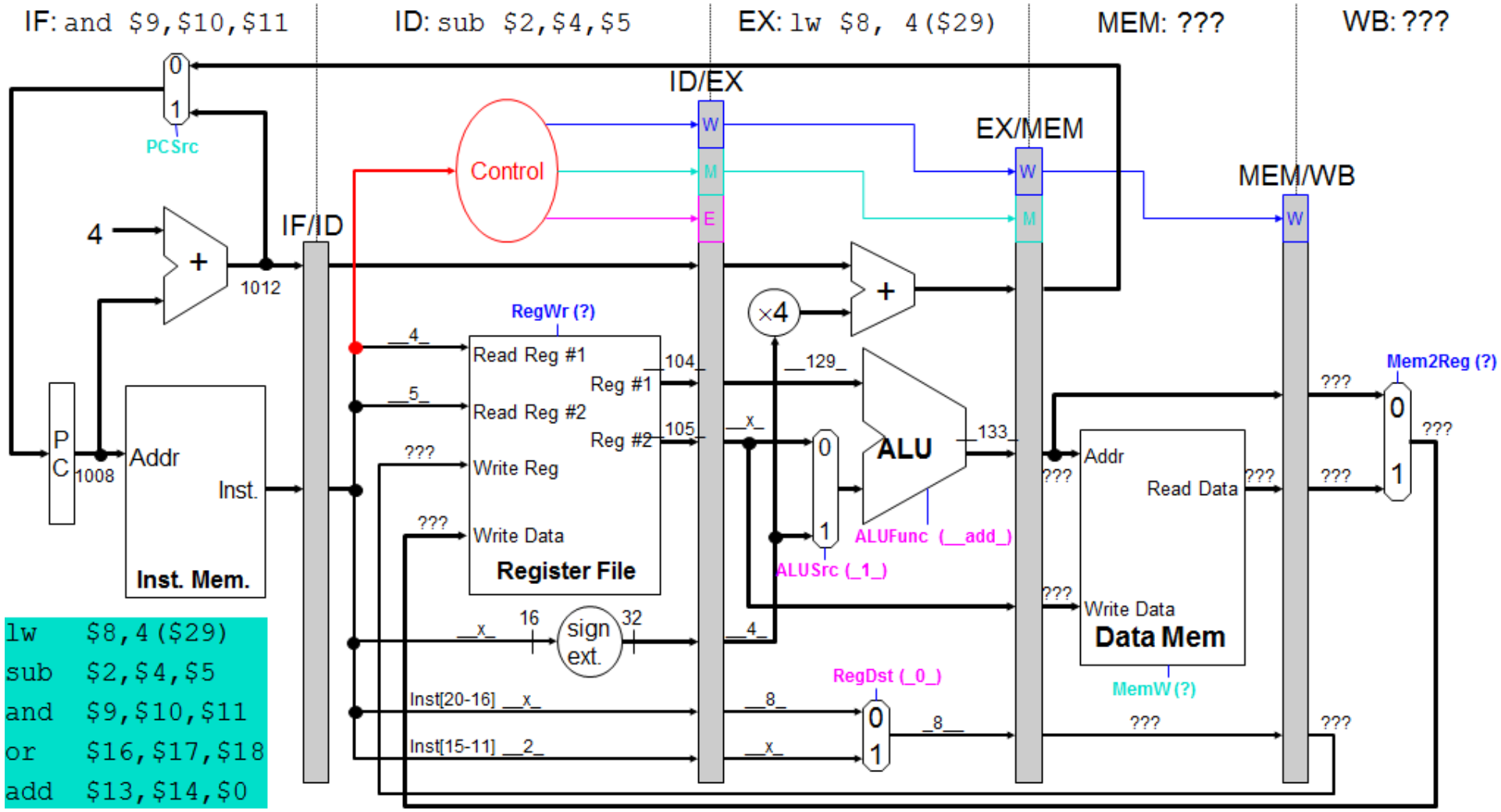
Cycle 1 (filling)



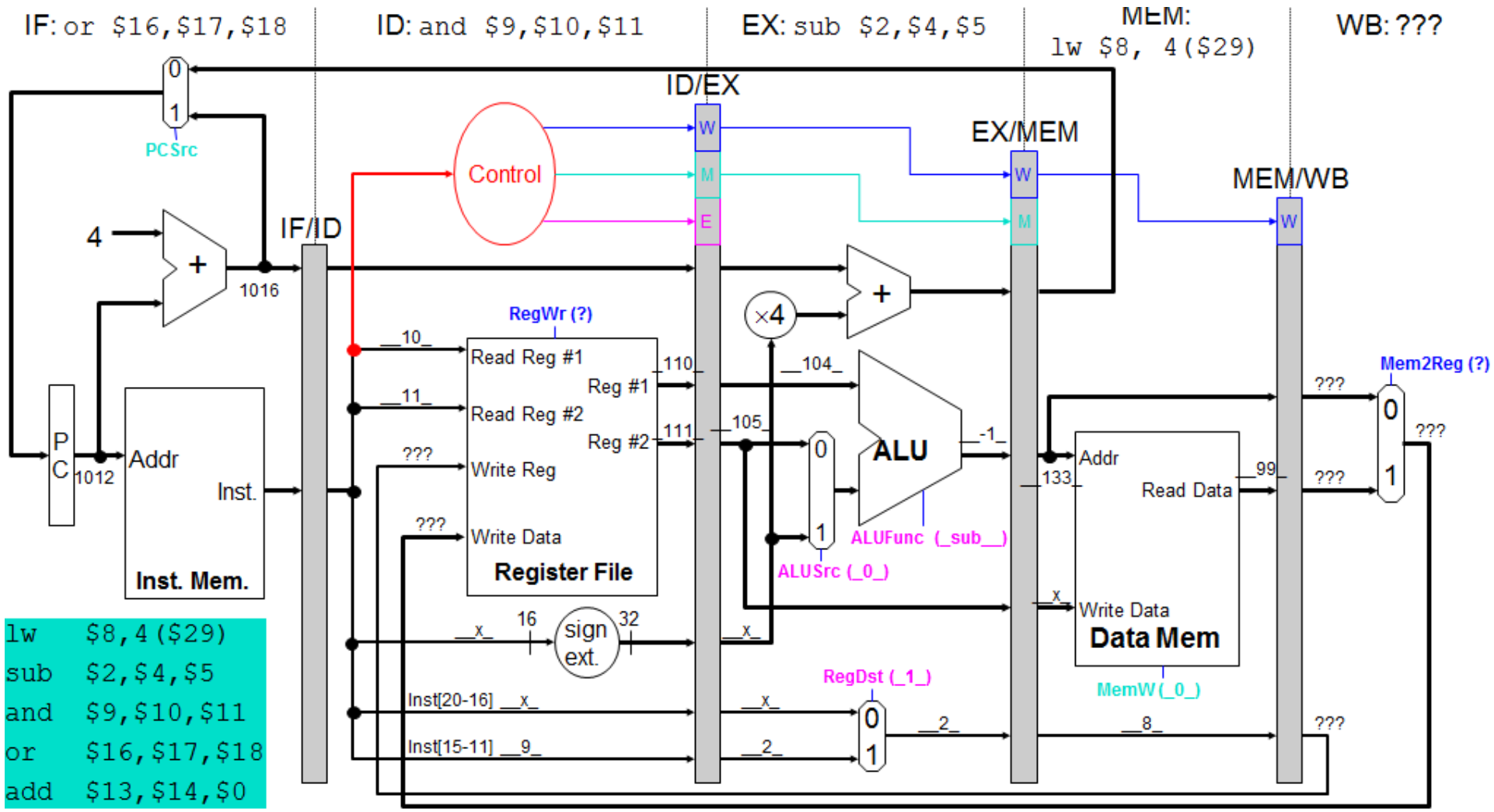
Cycle 2


IF: sub $2,$4,$5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
4
1008
1004
PC
Addr
Inst.
Inst. Mem.
RegWr (?)
Read Reg #1
Read Reg #2
Write Reg
Write Data
Reg #1
Reg #2
Register File
sign ext.
16
32
×4
ALU
ALUFunc (?)
ALUSrc (?)
RegDst (?)
Addr
Read Data
Write Data
Data Mem
MemW (?)
Mem2Reg (?)
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

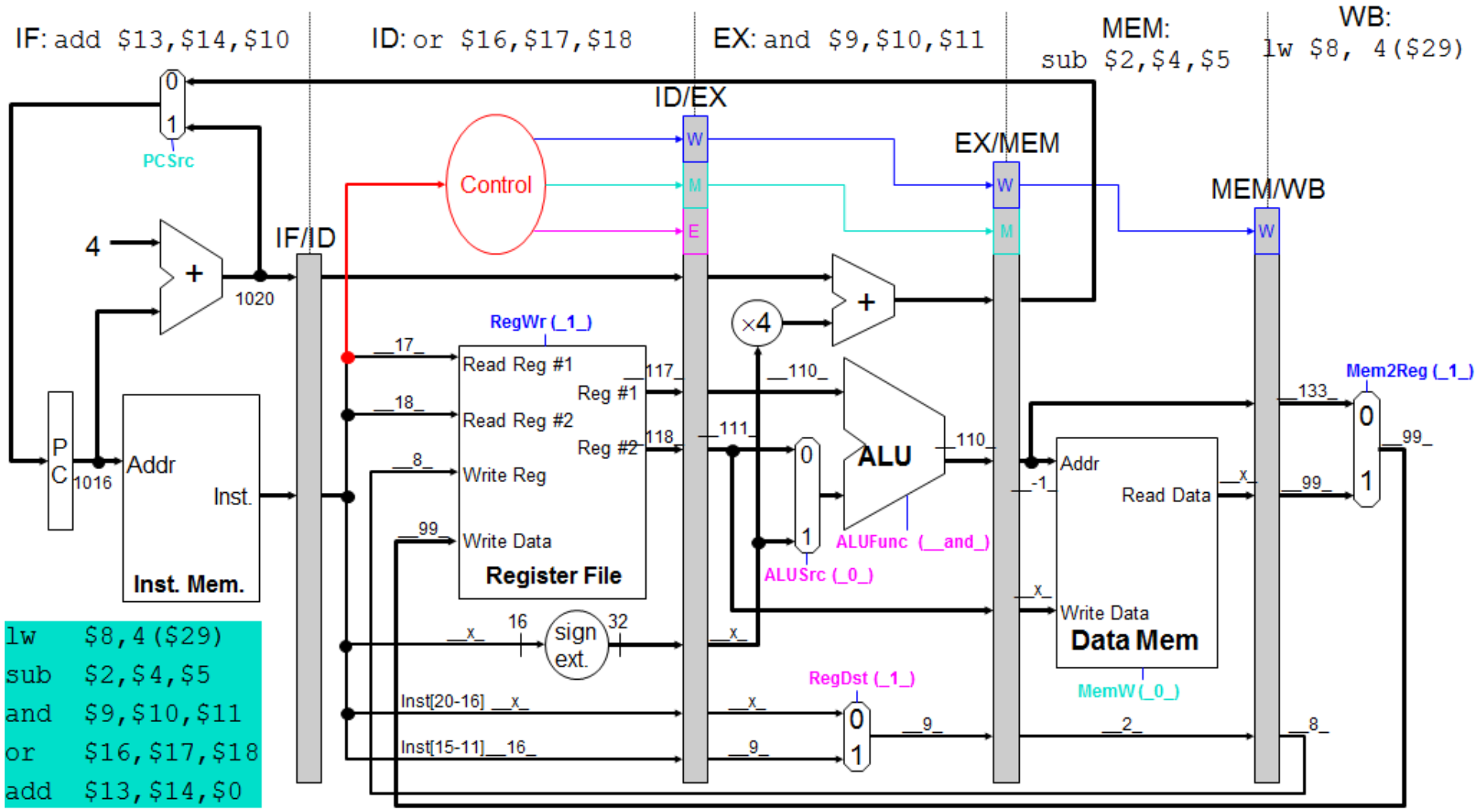
Cycle 3


IF: and $9,$10,$11
ID: sub $2,$4,$5
EX: lw $8, 4($29)
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
W
M
E
4
+
1012
PC
1008
Addr
Inst.
Inst. Mem.
RegWr (?)
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
104
105
129
133
×4
ALU
ALUFunc (__add_)
ALUSrc (_1_)
16
sign ext.
32
Inst[20-16]
Inst[15-11]
RegDst (_0_)
Addr
Read Data
Write Data
Data Mem
MemW (?)
Mem2Reg (?)
???
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

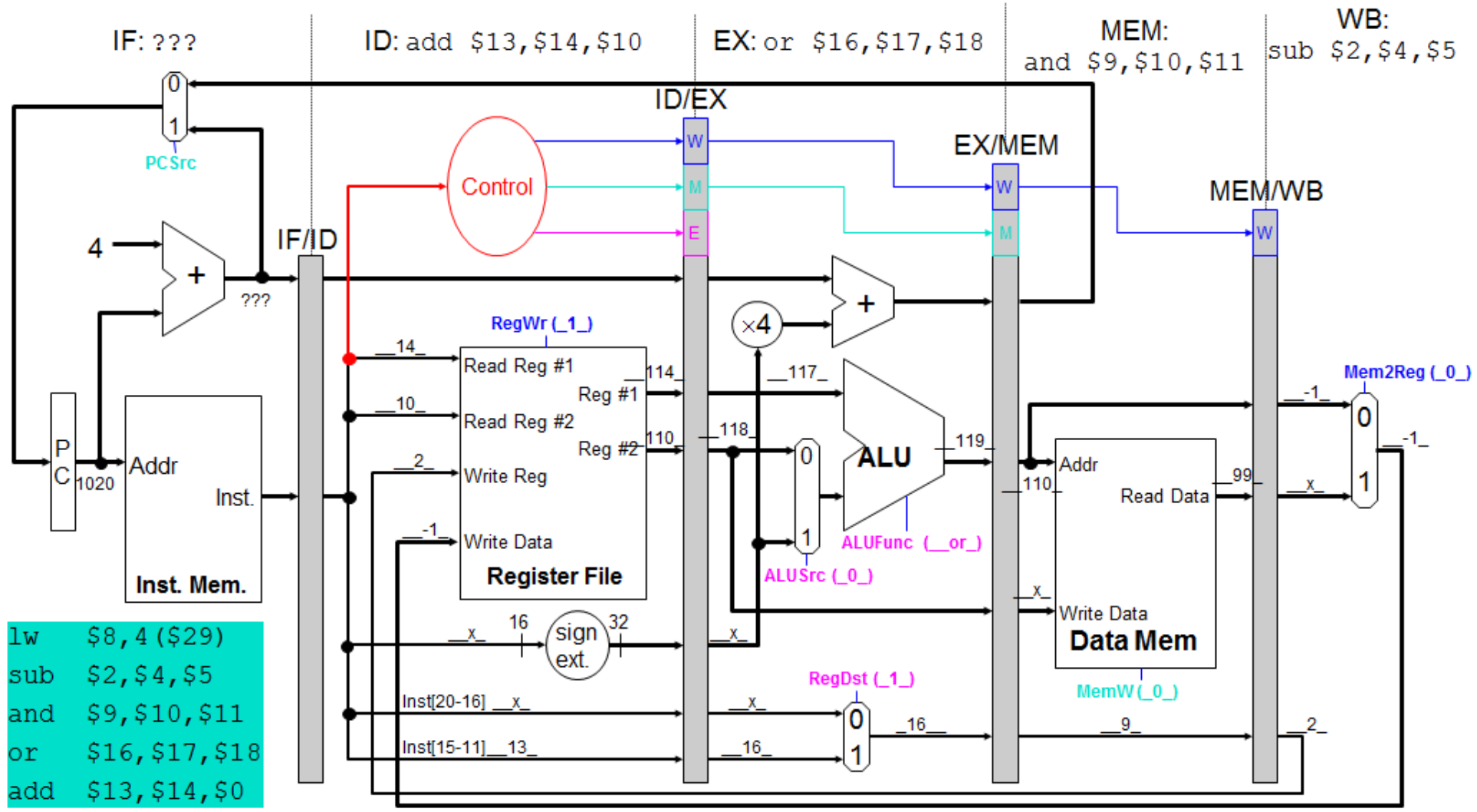
Cycle 4


IF: or $16,$17,$18
ID: and $9,$10,$11
EX: sub $2,$4,$5
MEM: lw $8, 4($29)
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
RegWr (?)
Read Reg #1
Read Reg #2
Write Reg
Write Data
Register File
Reg #1
Reg #2
Addr
Inst.
Inst. Mem.
PC
1012
1016
4
×4
ALU
ALUFunc (_sub_)
ALUSrc (_0_)
sign ext.
16
32
Addr
Read Data
Write Data
Data Mem
MemW (_0_)
Mem2Reg (?)
RegDst (_1_)
Inst[20-16]
Inst[15-11]
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

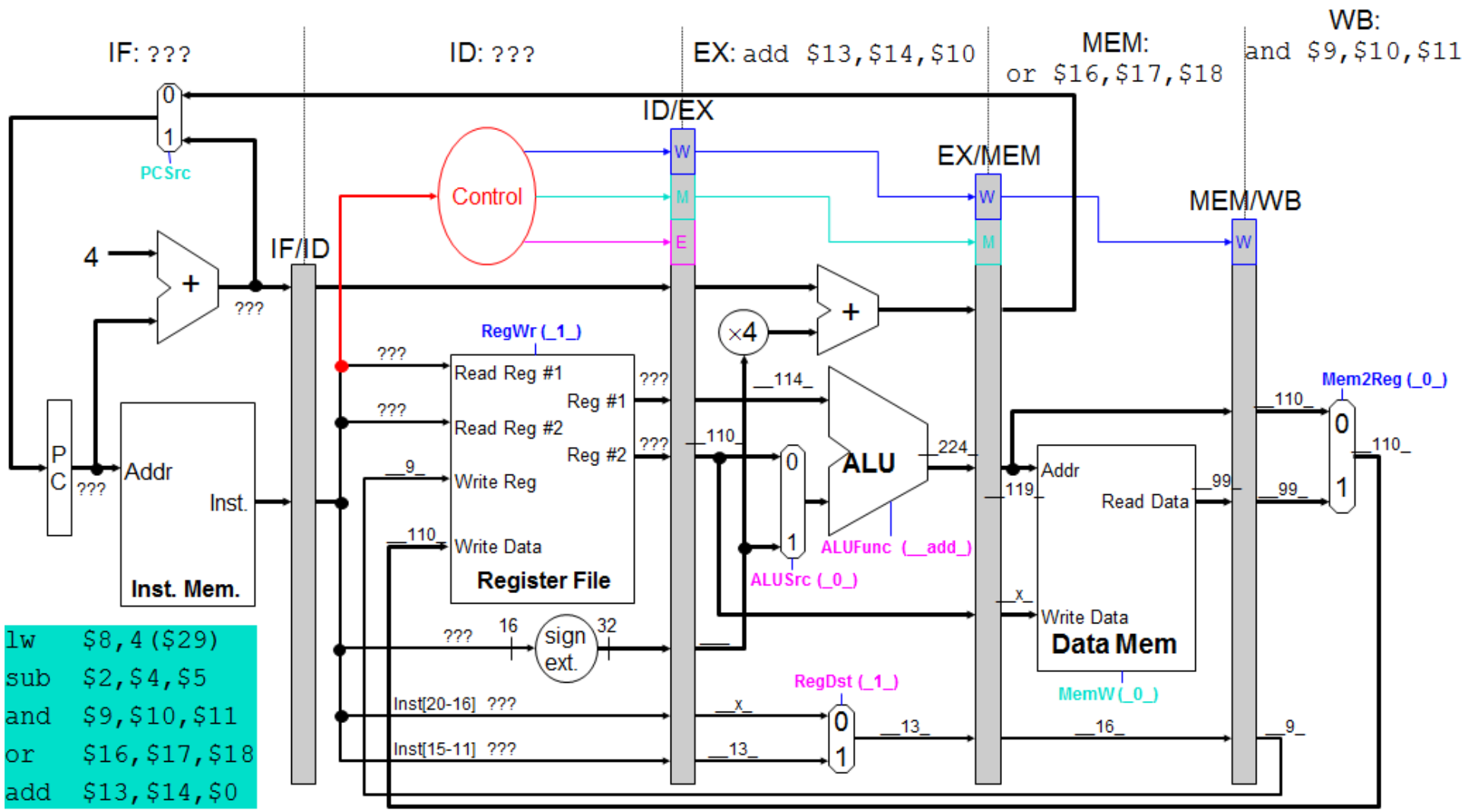
Cycle 5 (full)


IF: add $13,$14,$10
ID: or $16,$17,$18
EX: and $9,$10,$11
MEM: sub $2,$4,$5
WB: lw $8, 4($29)
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
W
M
E
PCSrc
4
+
1020
PC
1016
Addr
Inst.
Inst. Mem.
RegWr (_1_)
Read Reg #1
Read Reg #2
Write Reg
Write Data
Reg #1
Reg #2
Register File
17
18
8
99
117
118
16
sign ext.
32
×4
110
111
ALU
ALUFunc (__and_)
ALUSrc (_0_)
Addr
Read Data
Write Data
Data Mem
MemW (_0_)
RegDst (_1_)
Inst[20-16]
Inst[15-11]
Mem2Reg (_1_)
133
99
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

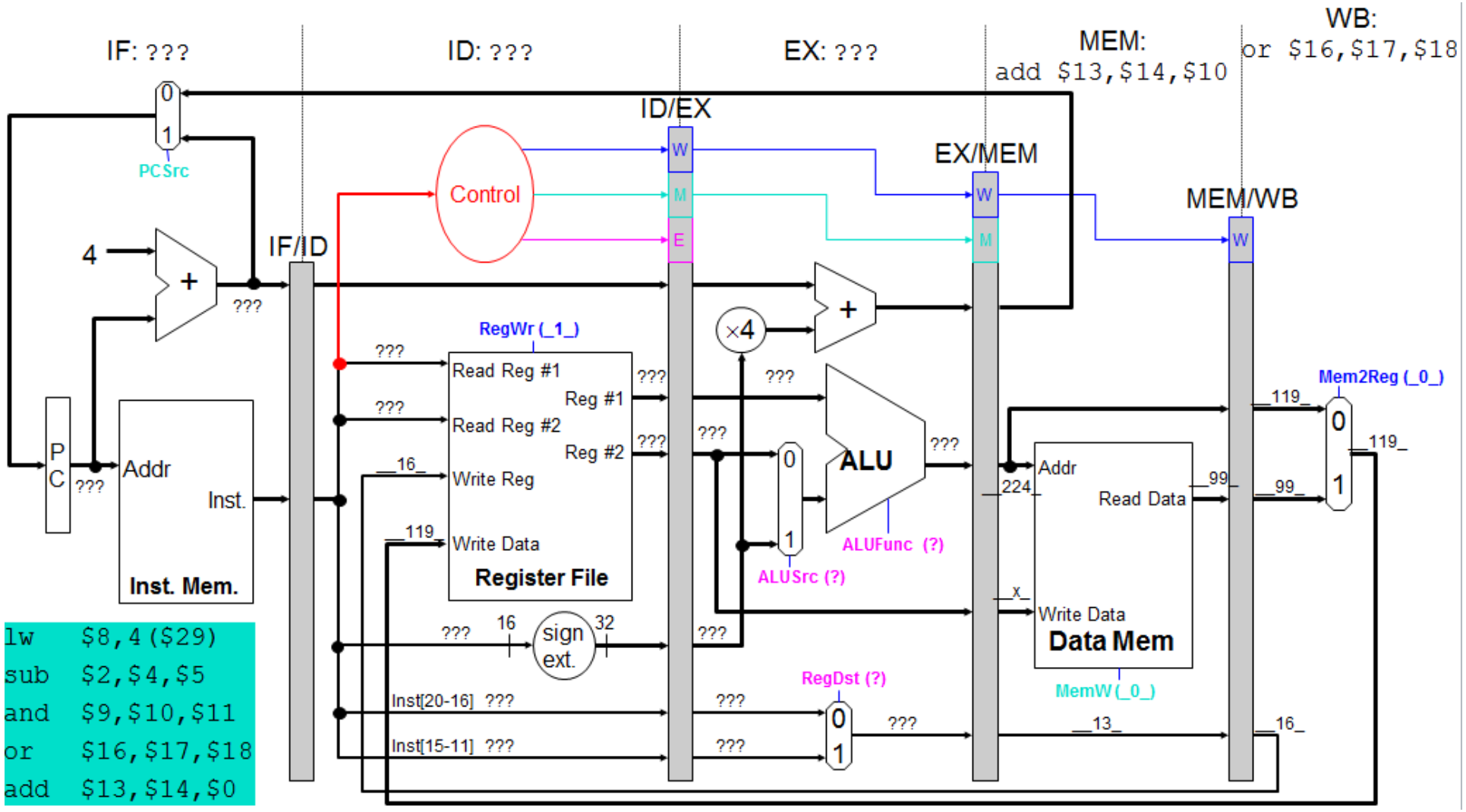
Cycle 6 (flushing)



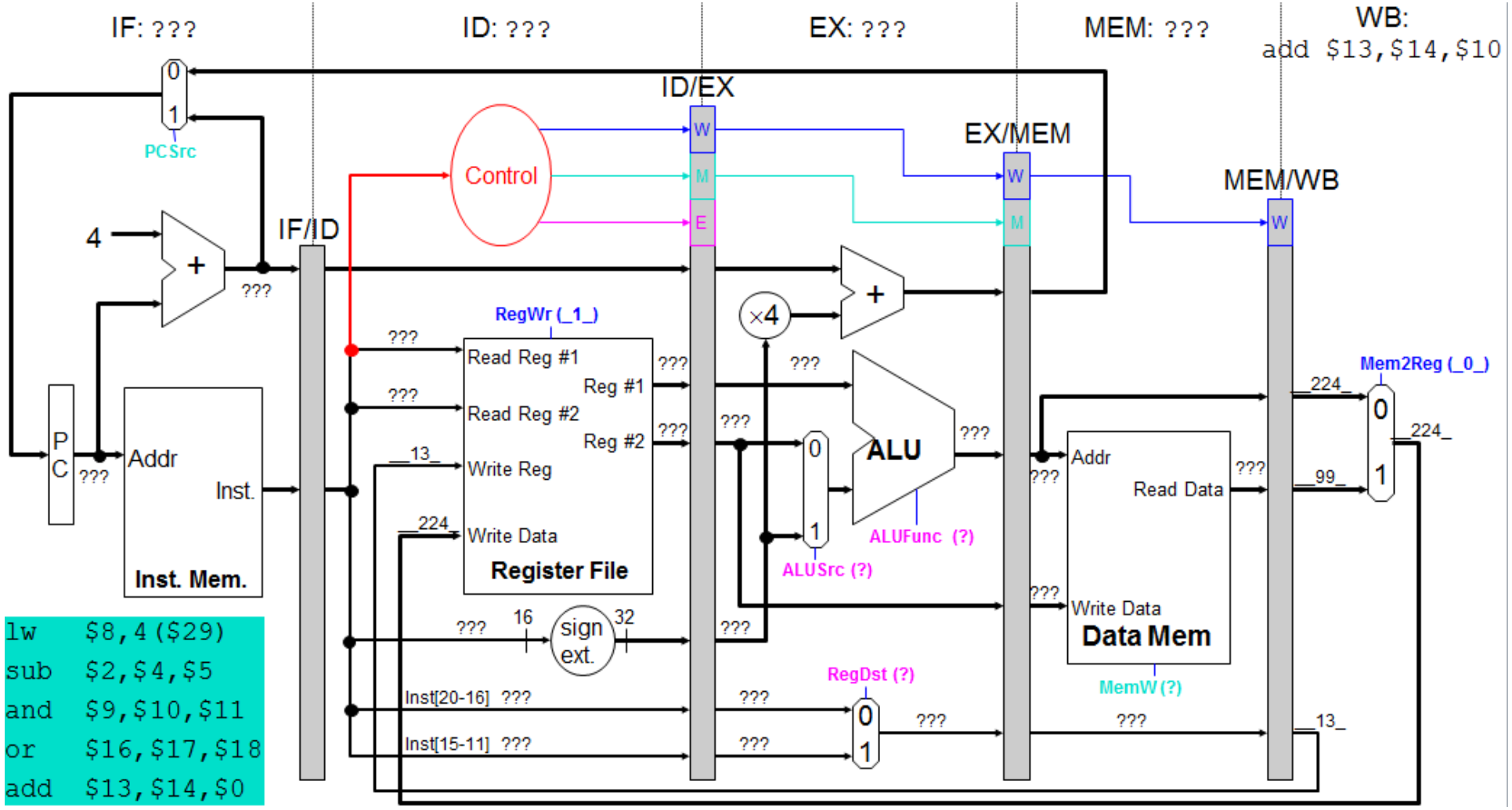
Cycle 7


IF: ???
ID: ???
EX: add $13,$14,$10
MEM: or $16,$17,$18
WB: and $9,$10,$11
ID/EX
EX/MEM
MEM/WB
PCSrc
Control
IF/ID
RegWr (_1_)
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
Addr
Inst.
Inst. Mem.
PC
ALU
ALUFunc (__add_)
ALUSrc (_0_)
sign ext.
RegDst (_1_)
Addr
Read Data
Write Data
Data Mem
MemW (_0_)
Mem2Reg (_0_)
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

Cycle 8


IF: ???
ID: ???
EX: ???
MEM: add $13,$14,$10
WB: or $16,$17,$18
PCSrc
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
W
M
E
4
+
???
RegWr (_1_)
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
×4
ALU
ALUFunc (?)
ALUSrc (?)
Mem2Reg (_0_)
PC
Addr
Inst.
Inst. Mem.
Read Data
Data Mem
MemW (_0_)
sign ext.
16
32
RegDst (?)
Inst[20-16] ???
Inst[15-11] ???
119
99
224
13
x
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

Cycle 9


IF: ???
ID: ???
EX: ???
MEM: ???
WB:
add $13,$14,$10
ID/EX
EX/MEM
MEM/WB
PCSrc
Control
IF/ID
4
+
???
RegWr (_1_)
Read Reg #1
Reg #1
Read Reg #2
Reg #2
Write Reg
Write Data
Register File
__13_
_224_
PC
Addr
Inst.
Inst. Mem.
×4
ALU
ALUFunc (?)
ALUSrc (?)
16
sign ext.
32
Inst[20-16] ???
Inst[15-11] ???
RegDst (?)
Addr
Read Data
Write Data
Data Mem
MemW (?)
Mem2Reg (_0_)
_224_
_99_
__13_
lw $8,4($29)
sub $2,$4,$5
and $9,$10,$11
or $16,$17,$18
add $13,$14,$0

Summary

- Utilize capabilities of the datapath by pipelined instruction processing.
  - Different stages use distinct functional units.
  - Multiple instructions are processed simultaneously.
  - Potential speedup is equal to the number of pipeline stages.
  - Performance limited by length of longest stage (plus fill/flush).
- Single cycle processor
  - CPI = 1, long cycle time.
- Multi-cycle processor
  - Multiple cycles to execute one instruction, shorter cycle time
  - Throughput is 1/CPI, which is less than 1.
- Pipelined processor
  - Each instruction takes multiple cycles, but multiple instructions can be processed simultaneously.
  - Shorter cycle time.
  - Increased throughput, potential throughput is 1 (one inst. per cycle).

Pipelining: ideal example

		Clock cycle								
		1	2	3	4	5	6	7	8	9
`lw`	`$8,4($29)`	IF	ID	EX	MEM	WB				
`sub`	`$2,$4,$5`		IF	ID	EX	MEM	WB			
`and`	`$9,$10,$11`			IF	ID	EX	MEM	WB		
`or`	`$16,$17,$18`				IF	ID	EX	MEM	WB	
`add`	`$13,$14,$0`					IF	ID	EX	MEM	WB

- Ideal code segment.
  - Start a new instruction every clock cycle.
  - Complete one instruction every clock cycle.
- No dependency between instructions.
  - Each instruction read/write distinct registers.
  - Data required by current instruction is not produced by previous instruction.
- Pipelined design can be generated easily.

However

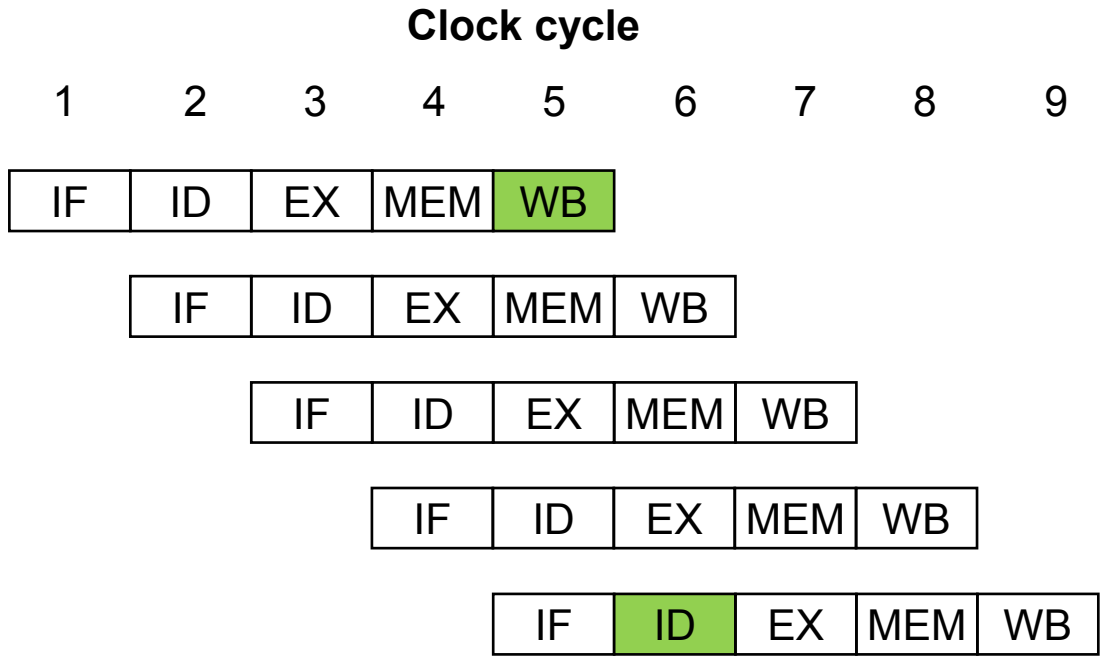
- There are many dependencies between instructions.
  - Read after write dependency.
  - E.g. instruction “and, or, add, sw” require (read) data $2, which will be produced (written) by instruction “sub”.
- It is fine for single cycle processor.
  - As an instruction will not start until the previous finish completely.
- What happen for a pipelined processor?

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

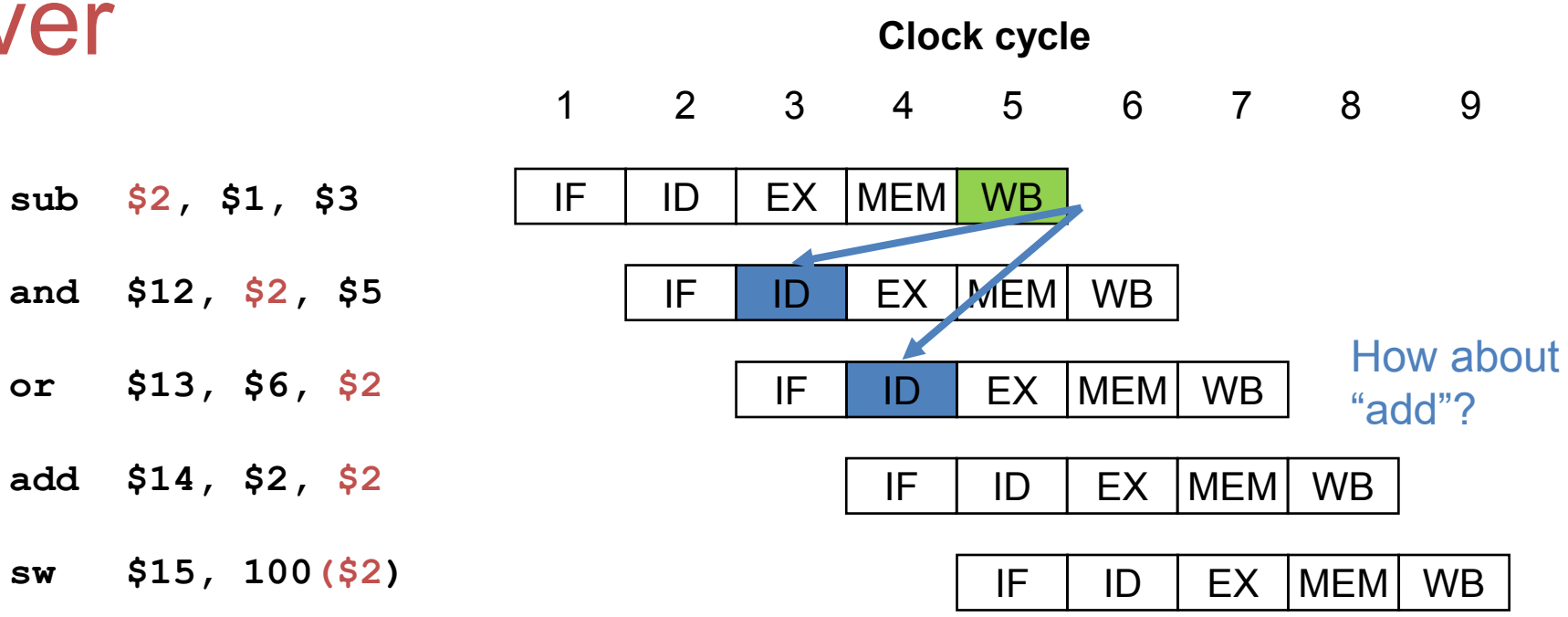
Instruction execution

- SW is fine
  - When does SW read the data of $2?
  - $2 has been written back to register when the ID stage of SW starts.

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



However



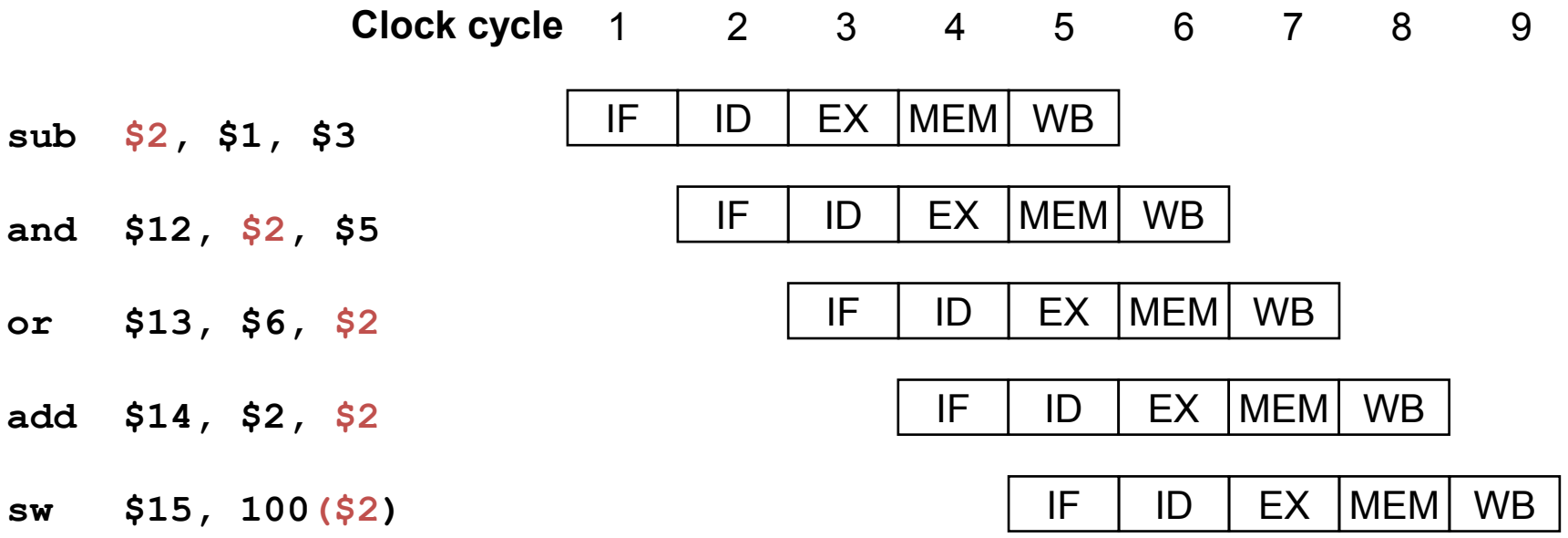
- "and, or" instructions.
  - $2 is not ready when they start reading $2.
  - An old value of $2 will be read in this circumstance.
- Data hazard
  - An instruction attempts to use data before it is ready.
- Back (red) arrows in time designate data hazard.

Solutions for solving data hazard

- Pipeline stall
  - Delay the execution of instructions such that they can read the correct data.
  - May reduce performance.
  - We will address this later.
- Forwarding
  - Forward the data from one stage to another stage that request the data. In other word, the data bypass the pipeline register.
  - E.g. forward the $2 from ALU output of “sub” to “and, or”.
- Question: How?
  - Explained in the following slides

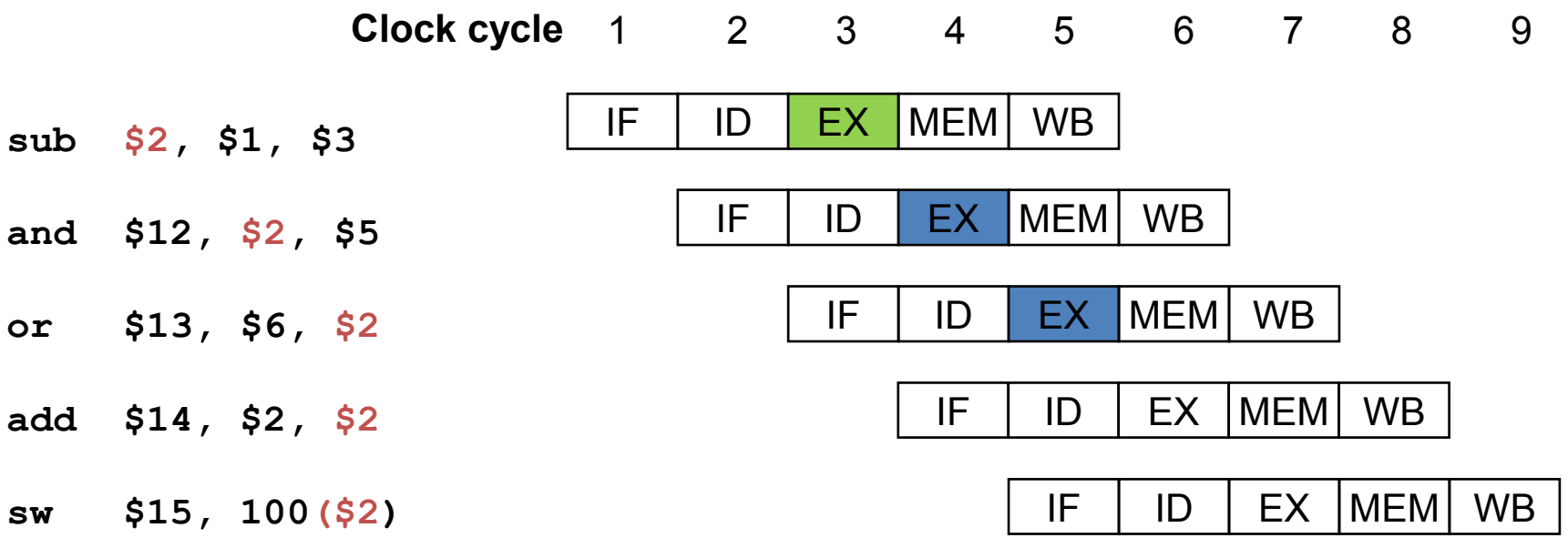
Observing the example

- When is the result ($2) produced?
- When is the result ($2) consumed?



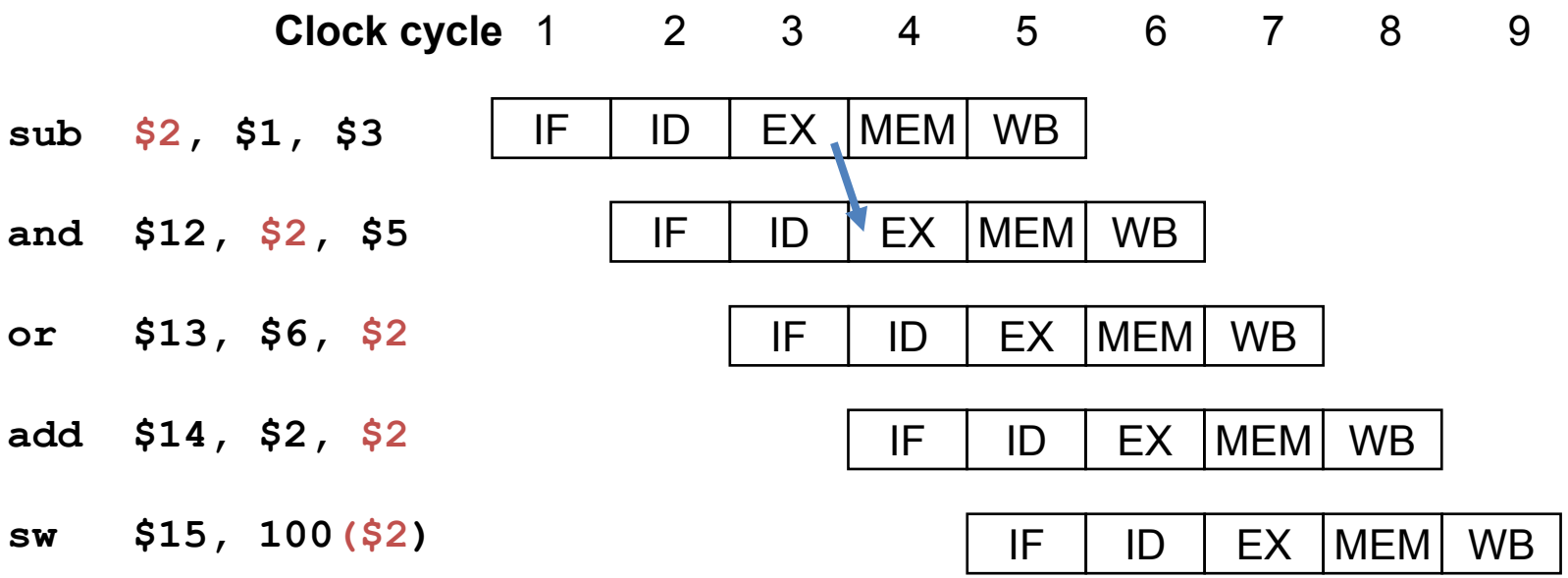
Observing the example

- When is the result ($2) produced? EX of “sub”
- When is the result ($2) consumed? EX of “and, or”



Forwarding

- Forward the result from the ALU output of “sub” to “and” instruction.
- Bypass the pipeline register and write back stage.



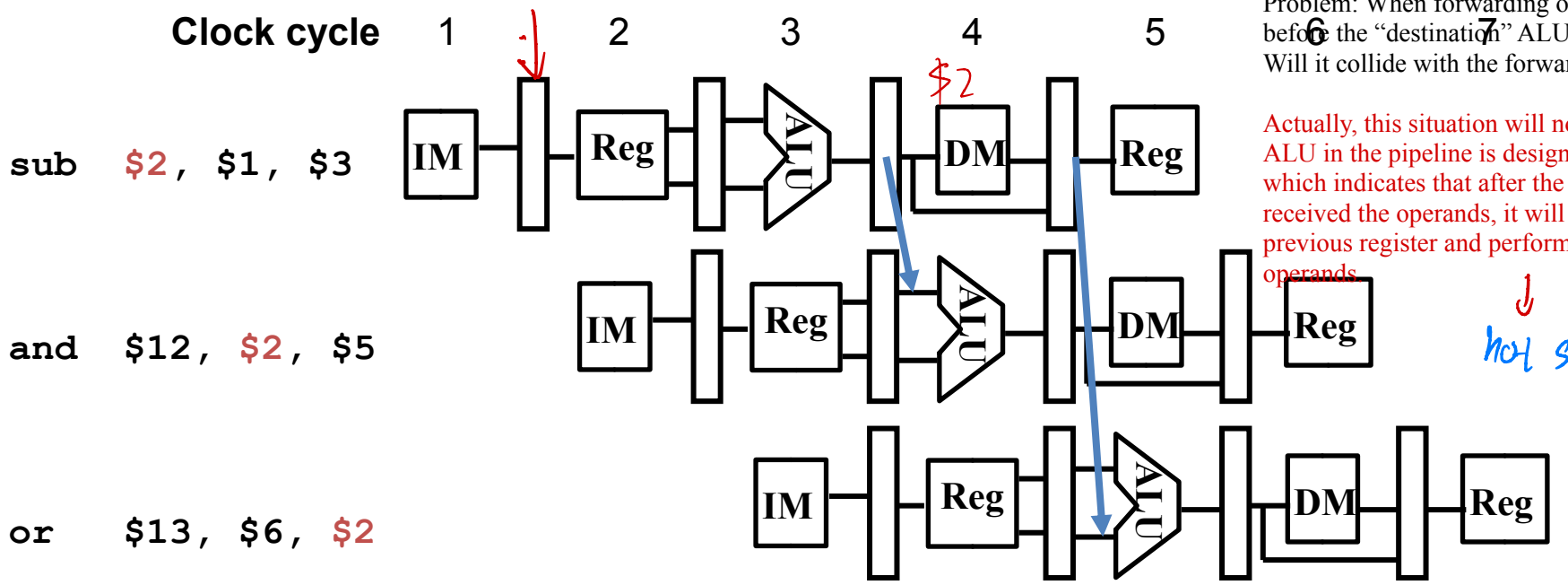
Forwarding

- Since at the beginning of clock cycle 4, pipeline register has already contained the result of ALU ($2).
- At clock 4: forward ALU result from EX/MEM register to ALU input.
- At clock 5: forward ALU result from MEM/WB register to ALU input.
  - As ALU result ($2) is now propagated to MEM/WB register.

Why forwarding?
In fact, eliminating the data hazard by stalling the pipeline is inefficient.

Remember that there does exist a state(buffer) register between stages in the pipeline.

For that instruction that needs the previous data to execute, we can actually read from the buffer to get the correct value without having to wait for reading the register file in the end of the pipeline. Note that forwarding paths is valid only if the destination stage that receives the "forwarded" value is after the source stage.

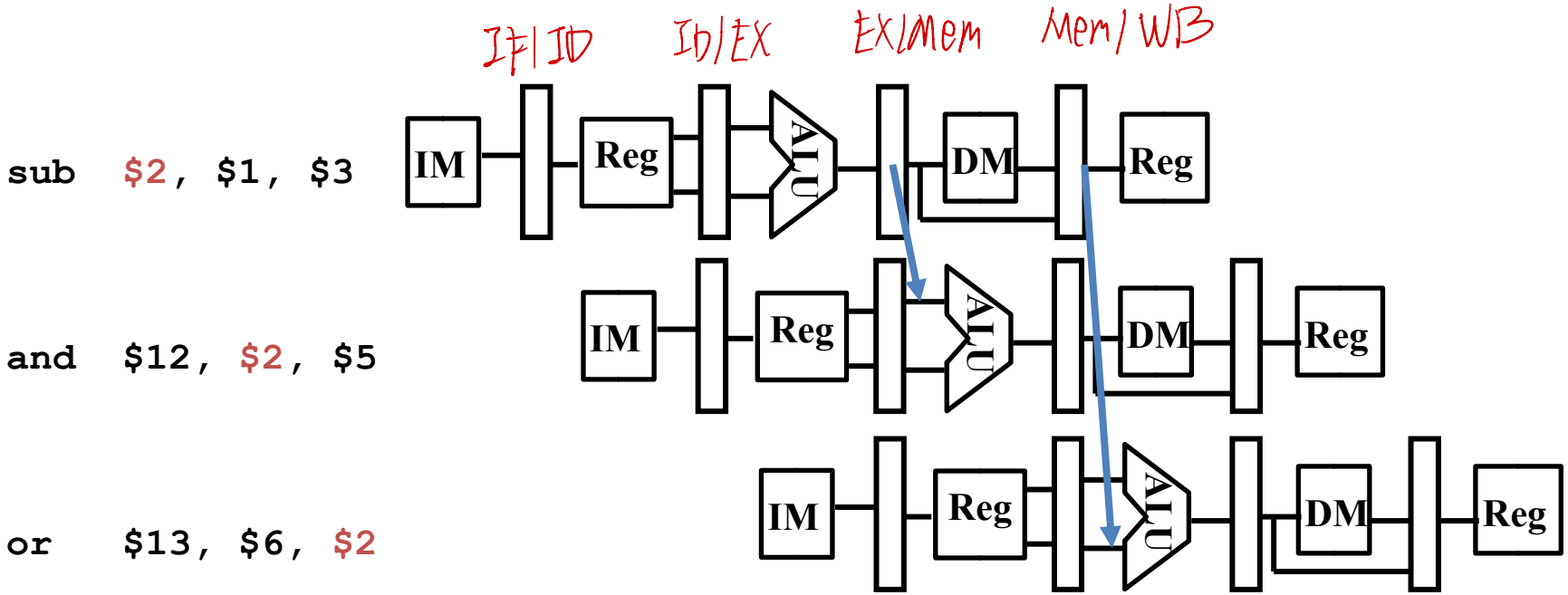


Problem: When forwarding occurred in the ALU, the register before the "destination" ALU is reading data concurrently. Will it collide with the forwarding?

Actually, this situation will not happen, because actually the ALU in the pipeline is designed in a synchronous way, which indicates that after the "destination" ALU has received the operands, it will latch the new data from the previous register and perform calculations using provided operands.

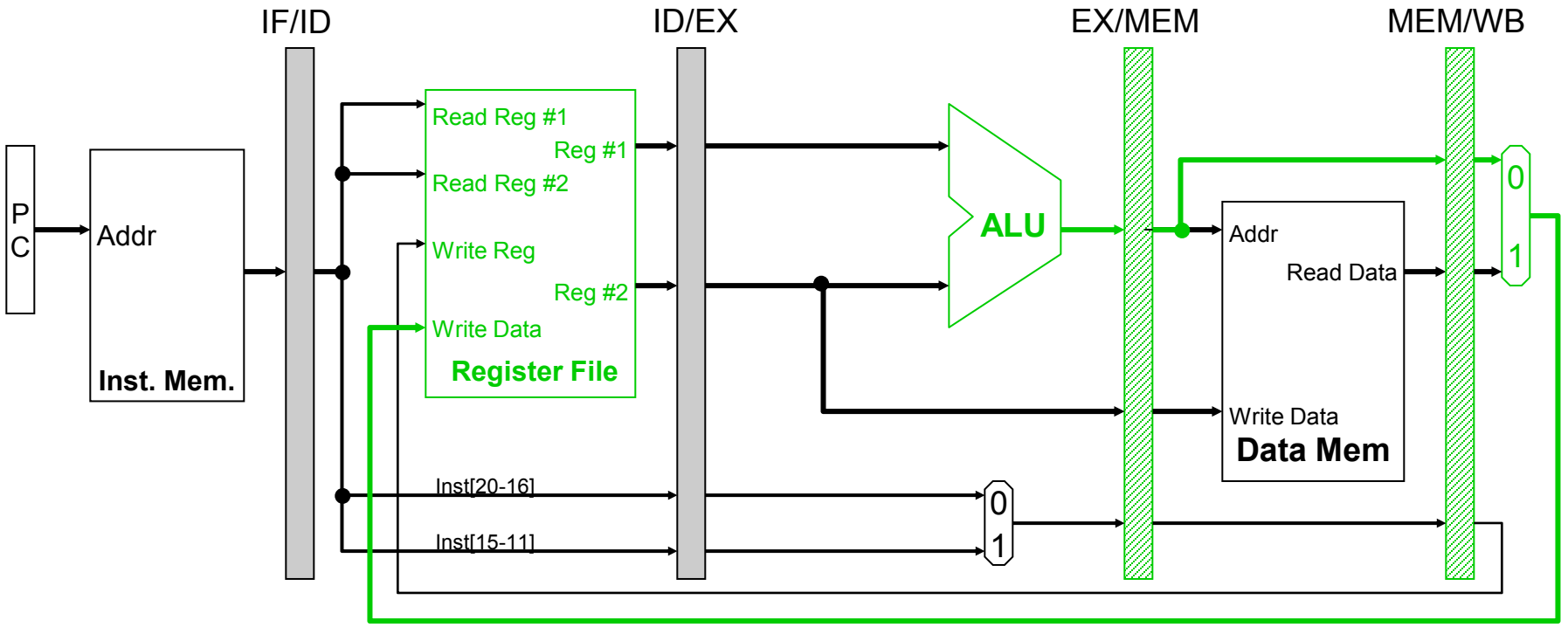
Forwarding hardware

- Add a control unit (forwarding unit) to select the correct ALU value for the EX stage.
  - If there is no hazard, the unit selects ALU's operands from register file, just like before.
  - If there is a hazard, the unit selects operands from either the EX/MEM or MEM/WB pipeline registers.
- As ALU has two inputs, add two new multiplexers to select the ALU operands.

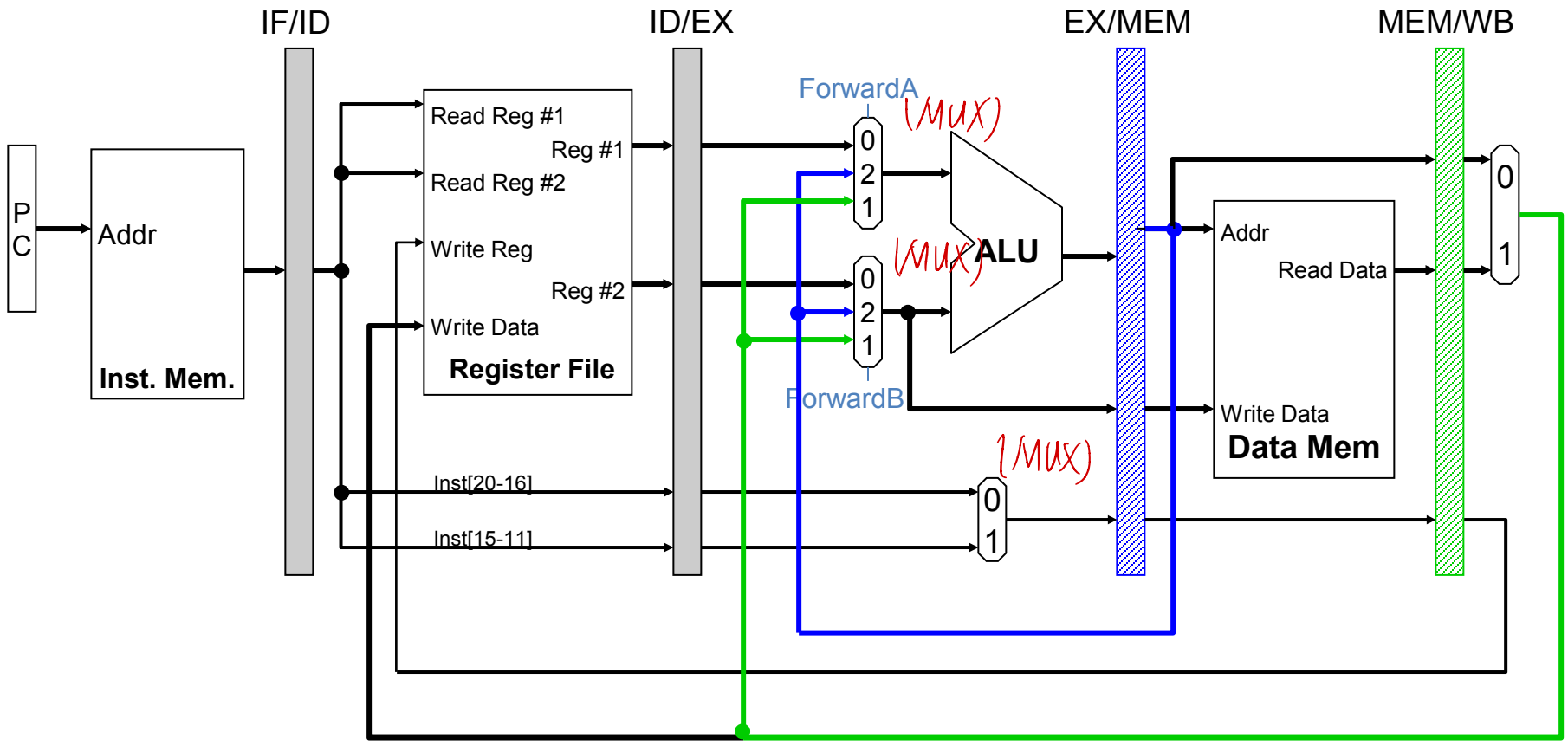


Normal flow of ALU result

- The ALU result generated at the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.



Simplified datapath with two new multiplexers



Next step: add a forwarding unit (control unit) to control the selection of the two multiplexers.

Question: how can the hardware know that there is data hazard? → how to detect data hazard?

Forward Unit Output Signals

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

How to detect data hazard?

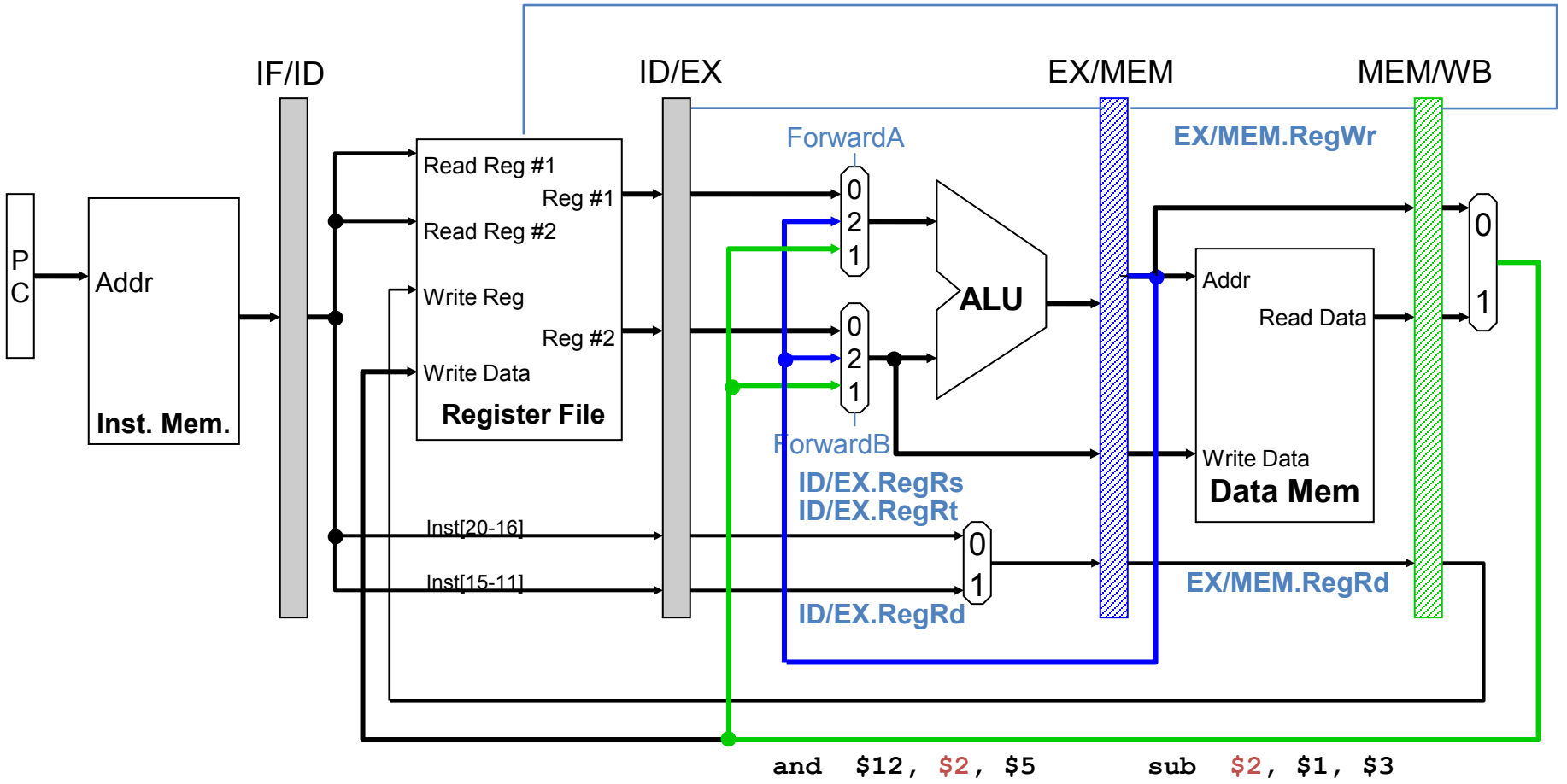
- 1. EX Forward Unit:

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
        ForwardA = 10
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 0)
and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
        ForwardB = 10
```

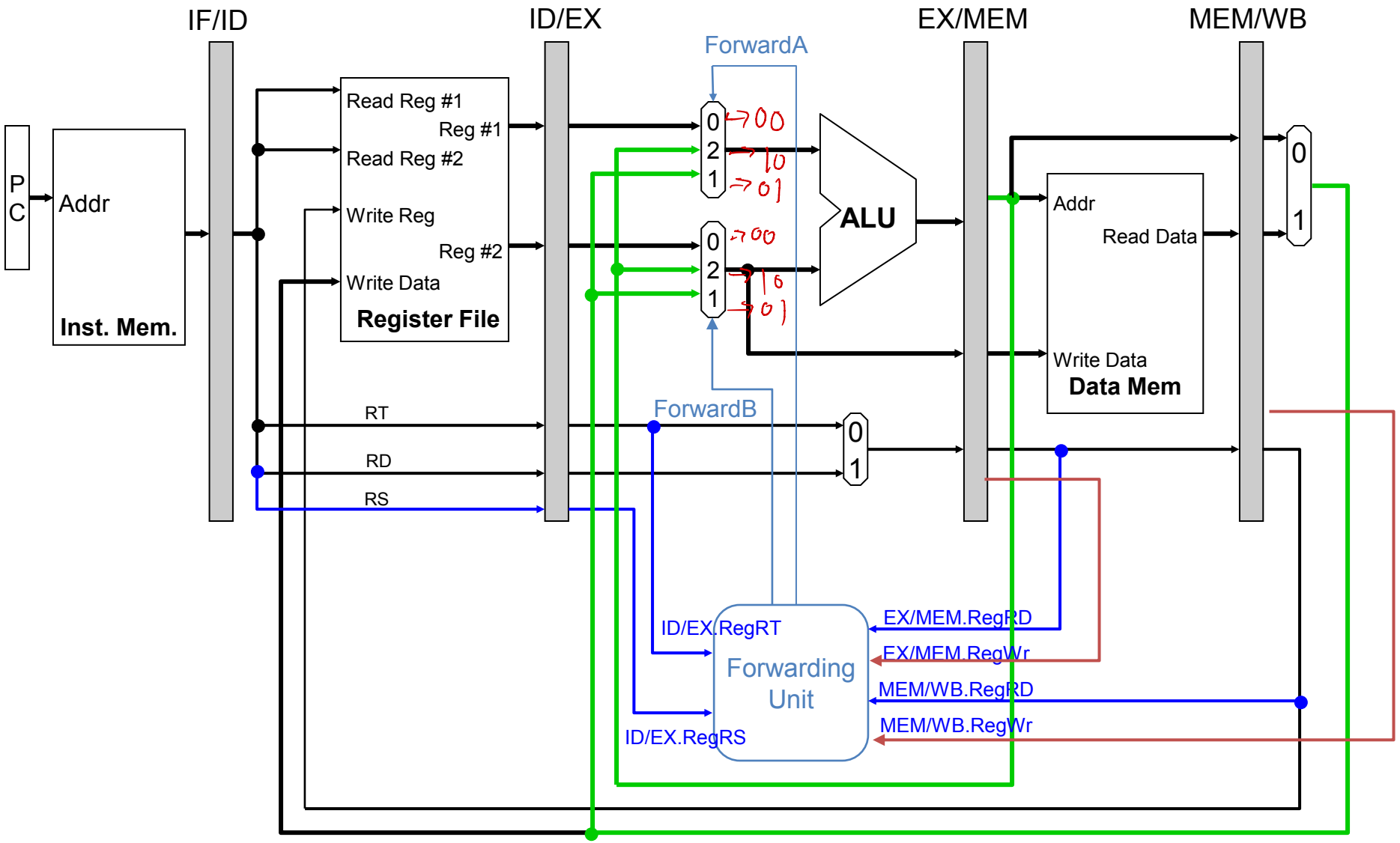
- 2. MEM Forward Unit:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRs))
        ForwardA = 01
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 0)
and (MEM/WB.RegisterRd == ID/EX.RegisterRt))
        ForwardB = 01
```

How to detect data hazard?



Datapath with forwarding unit (control unit)

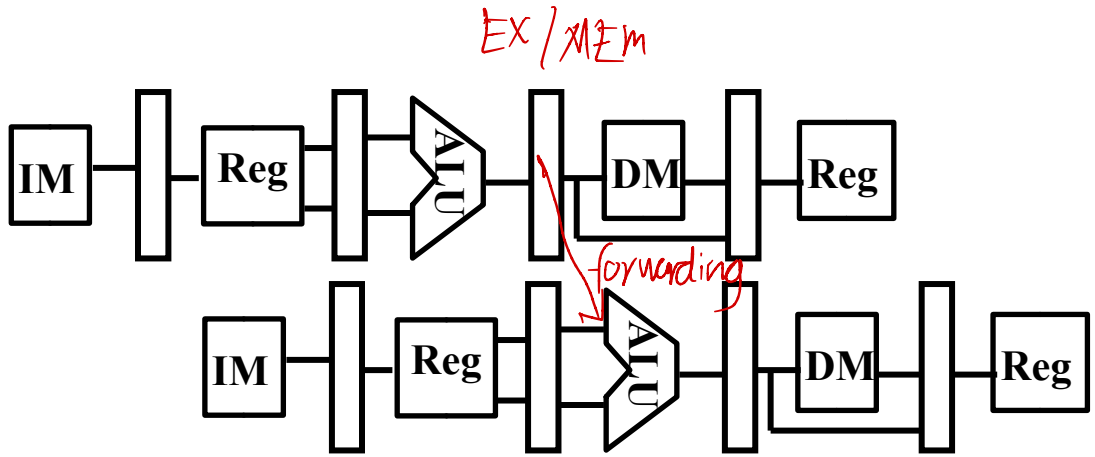


EX/MEM hazard

- An EX/MEM hazard occurs between two adjacent instructions and:
  - The first instruction will write data to register file.
  - The first instruction's destination is the same as one of the second instruction's read operands, e.g. register $2 here.

The origin of the hazard is same as the one mentioned in page 64.

```
sub  $2, $1, $3
and  $12, $2, $5
```

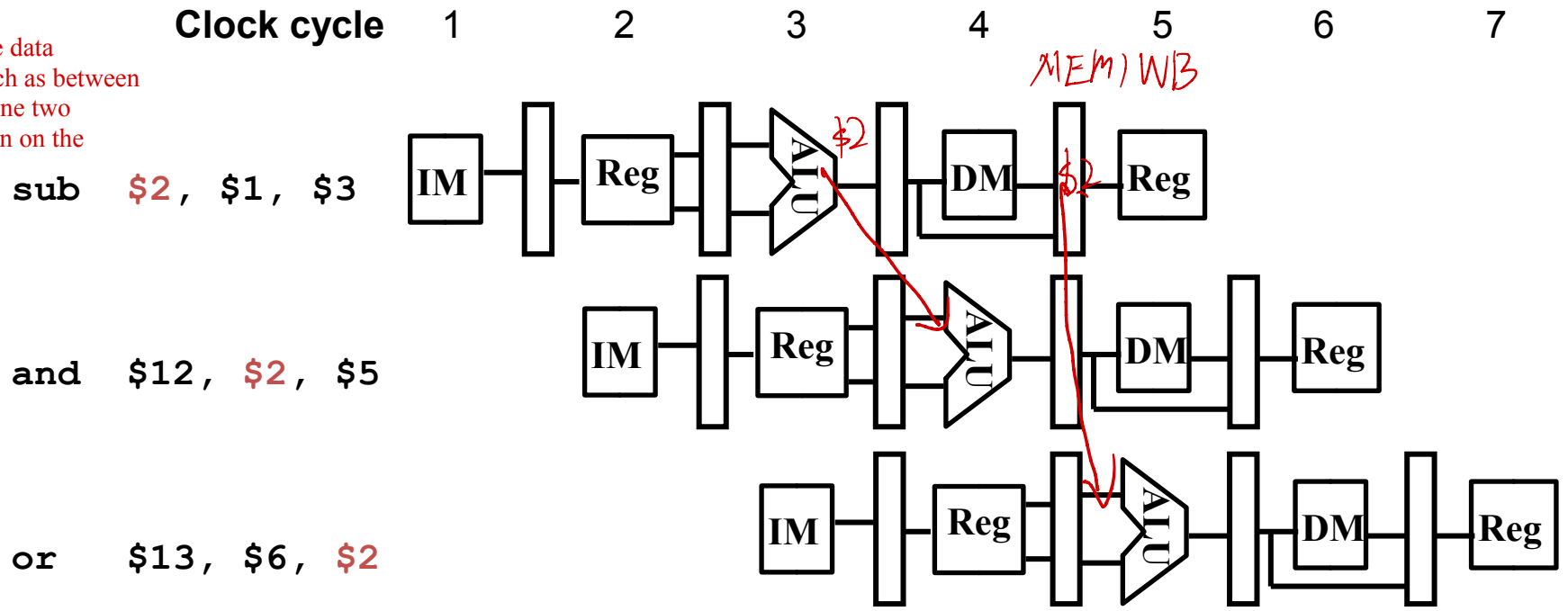


MEM/WB hazard

- Occur between current instruction and the instruction two stages ahead.

Is the data only allowed to be forwarded between adjacent instructions?

No. We can also forward the data between multiple stages, such as between current instruction and the one two stages ahead, which is shown on the figure.



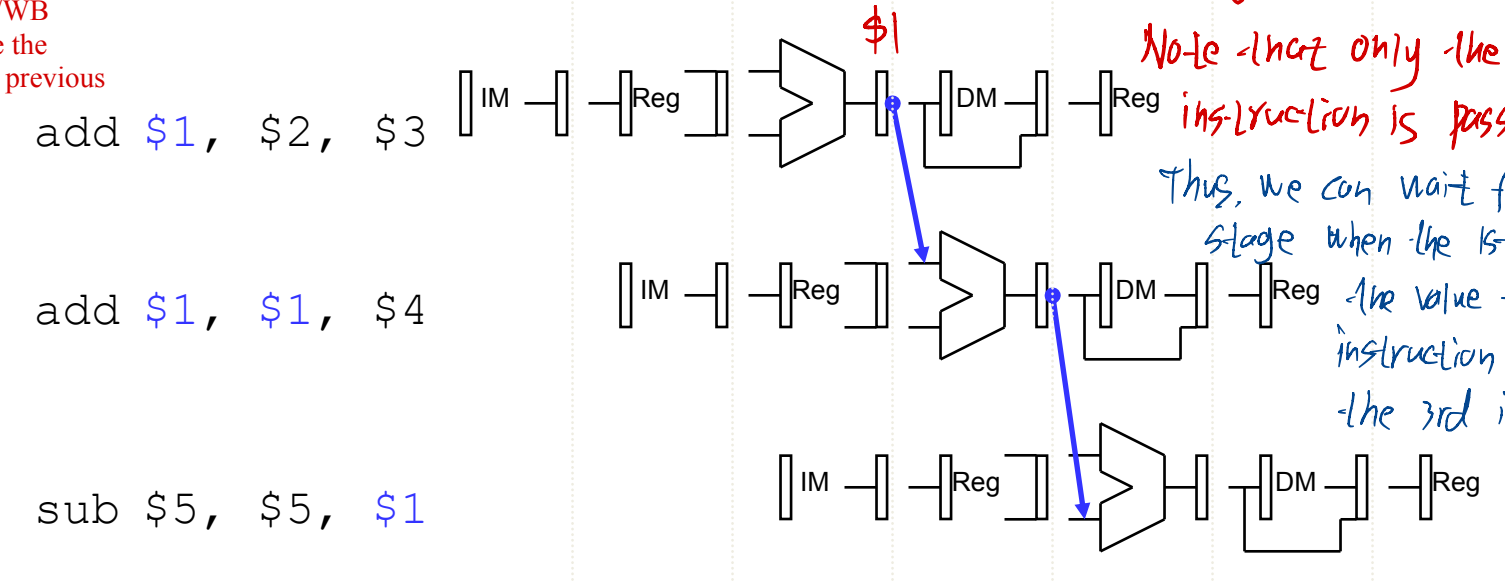
Exception: not MEM/WB hazard, it is EX/MEM hazard

- One new problem is if the same register is updated twice in previous two instructions.

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

- Register $1 is written by *both* of the previous two instructions: which instruction will actually produce the result for SUB?
  - Only the most recent result (from the second ADD) should be forwarded.

Thus, it is impossible that MEM/WB hazard occurs for this case where the same register is updated twice in previous two instructions.



It is unnecessary to forward the data in register $1 when the 1st instruction is executed. Note that only the result produced by the 2nd instruction is passed to the final instruction. Thus, we can wait for the end of "writeback" stage when the 1st execution proceeds, allowing the value to be written for the 2nd instruction and forward the data for the 3rd instruction.

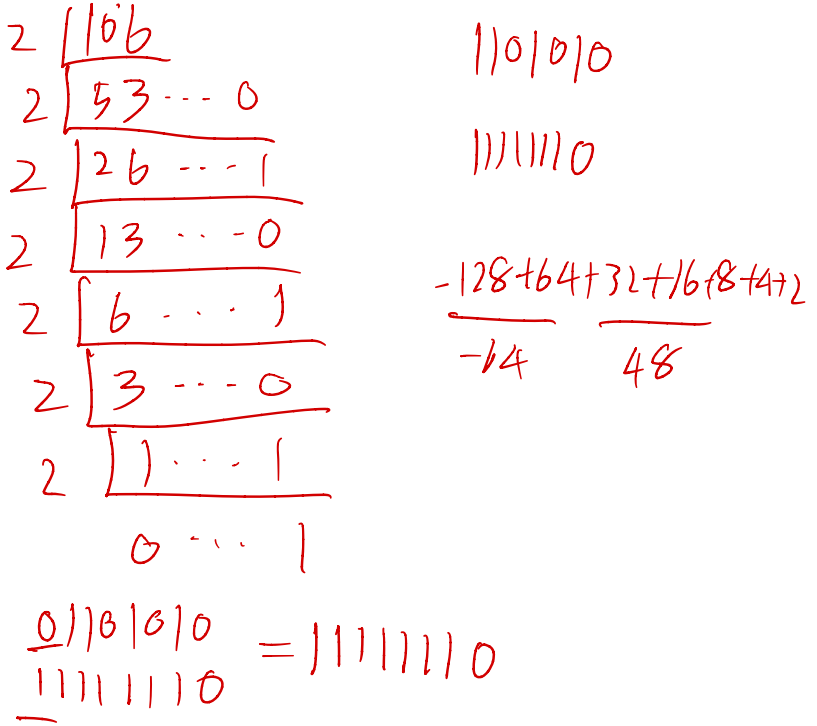
↓

EX/MEM hazard

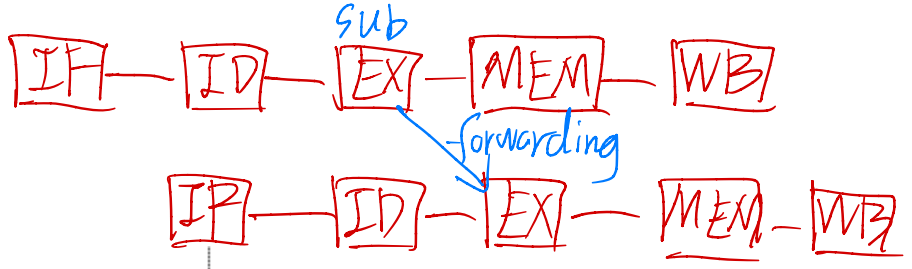
Forwarding example:

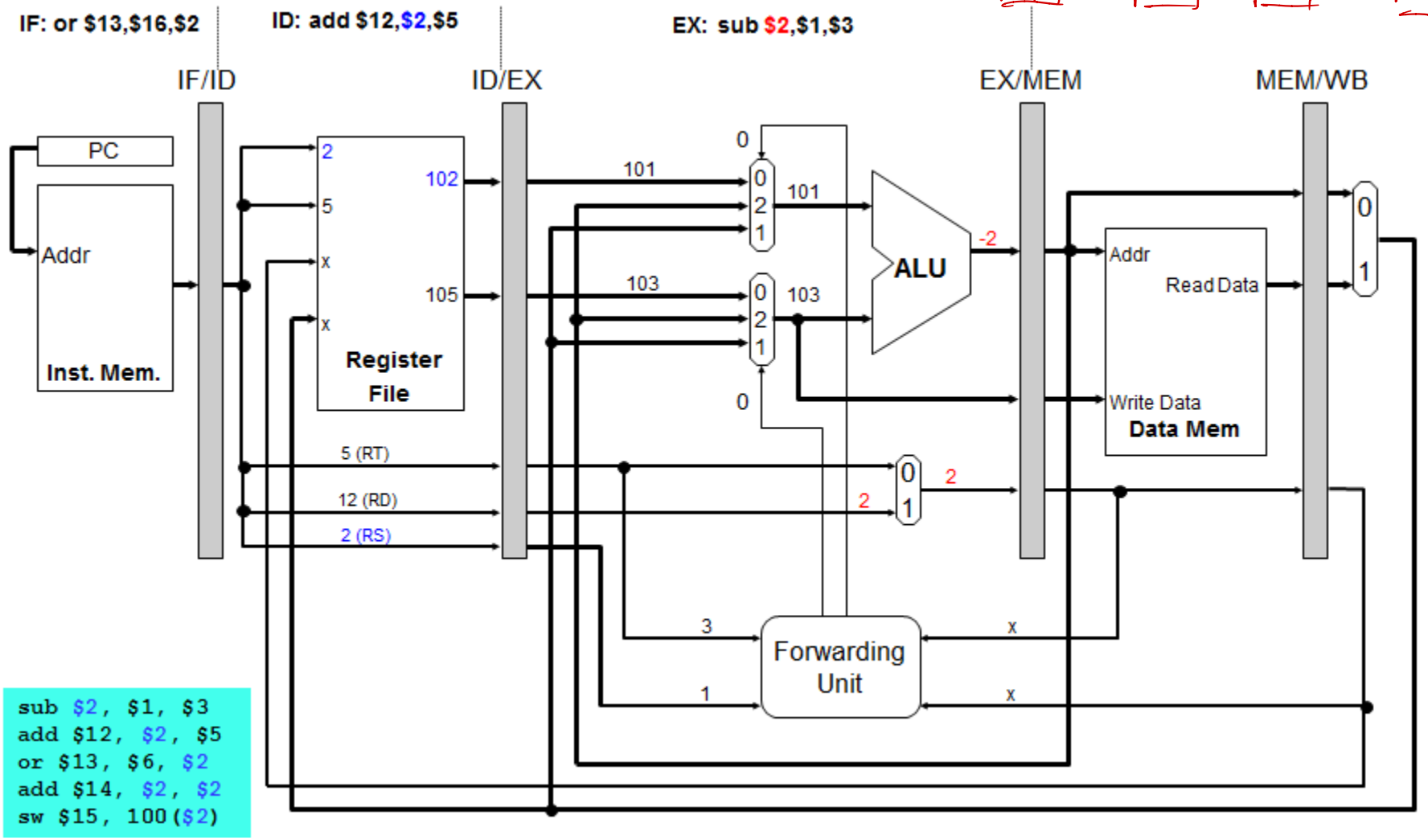
- Assume that each register contains a value which is its number plus 100. For instance, register $2 contains 102, register $6 contains 106, and so forth.

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



Cycle 3

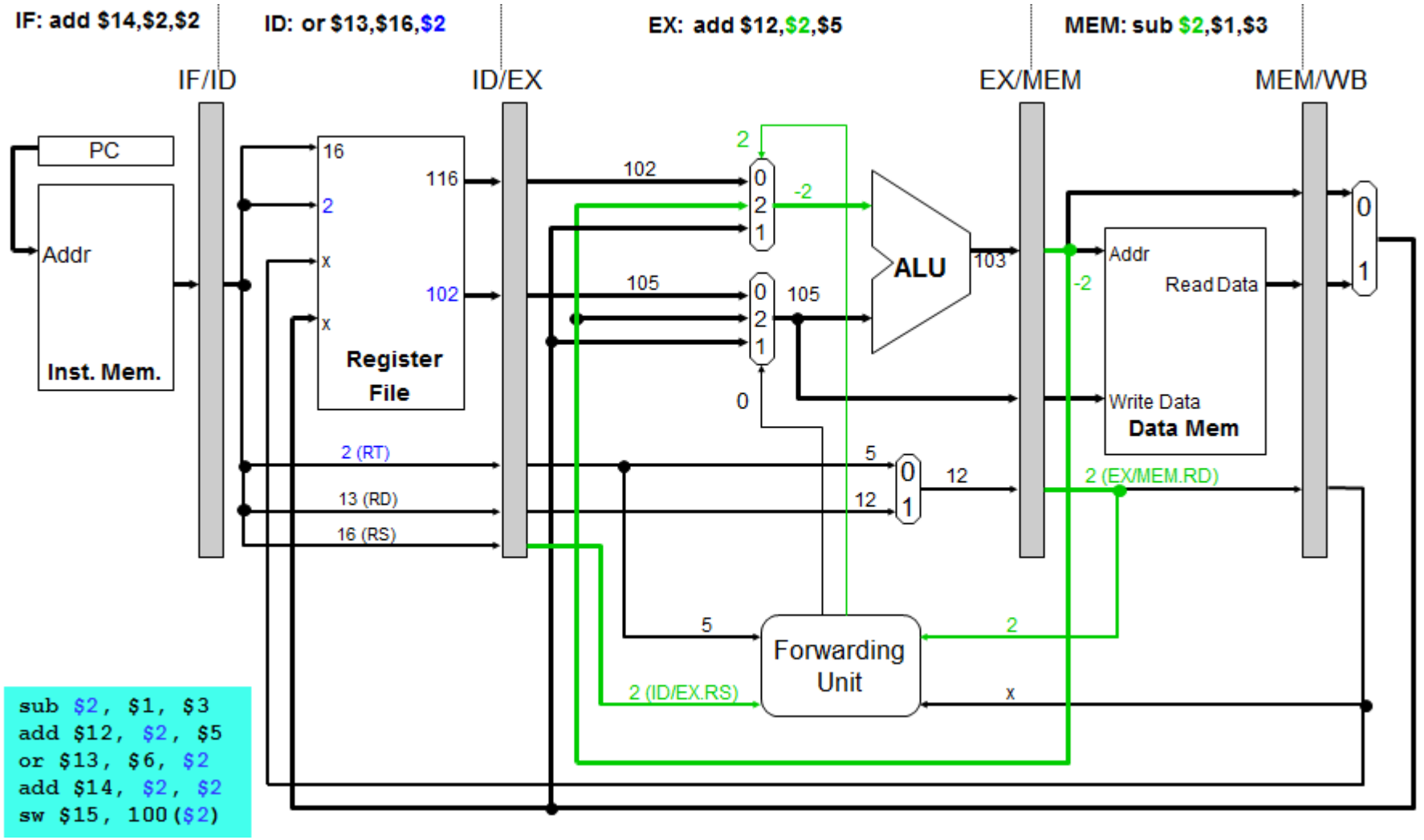




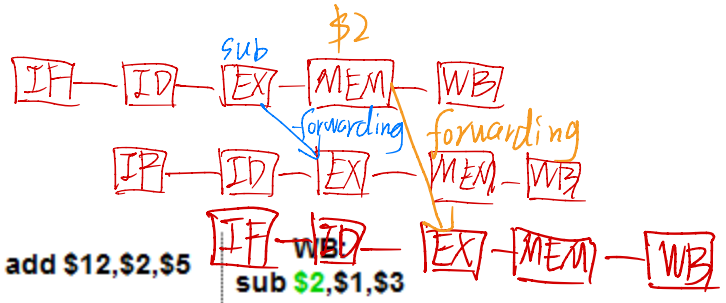

```
sub $2, $1, $3
add $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

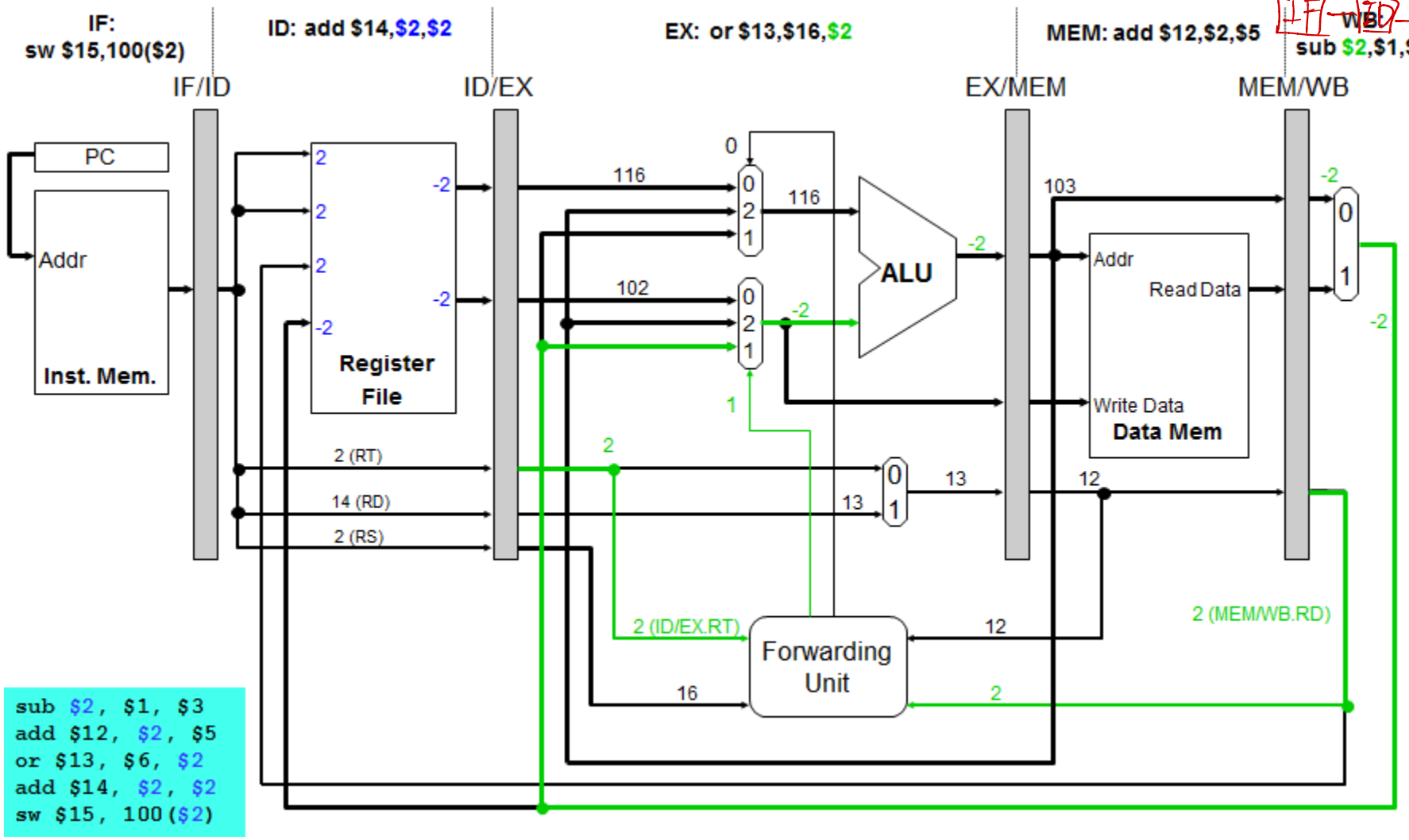
Cycle 4: forwarding $2 from EX/MEM

EX hazard



Cycle 5: forwarding $2 from MEM/WB

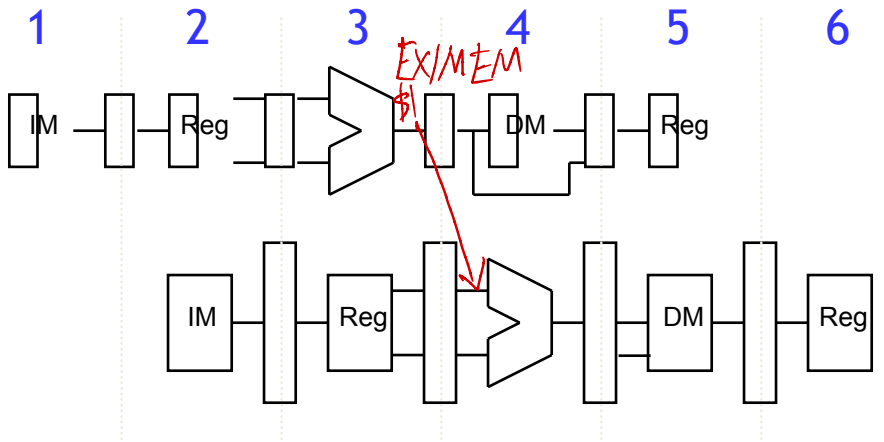




Forwarding for SW

CASE 1:

```
add $1, $2, $3
sw  $4, 0($1)
```

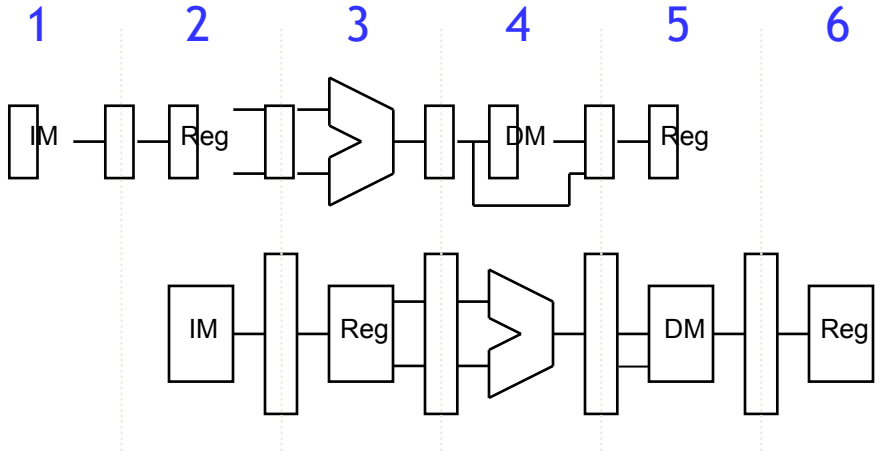


mem[$1的] = $4

1st operand

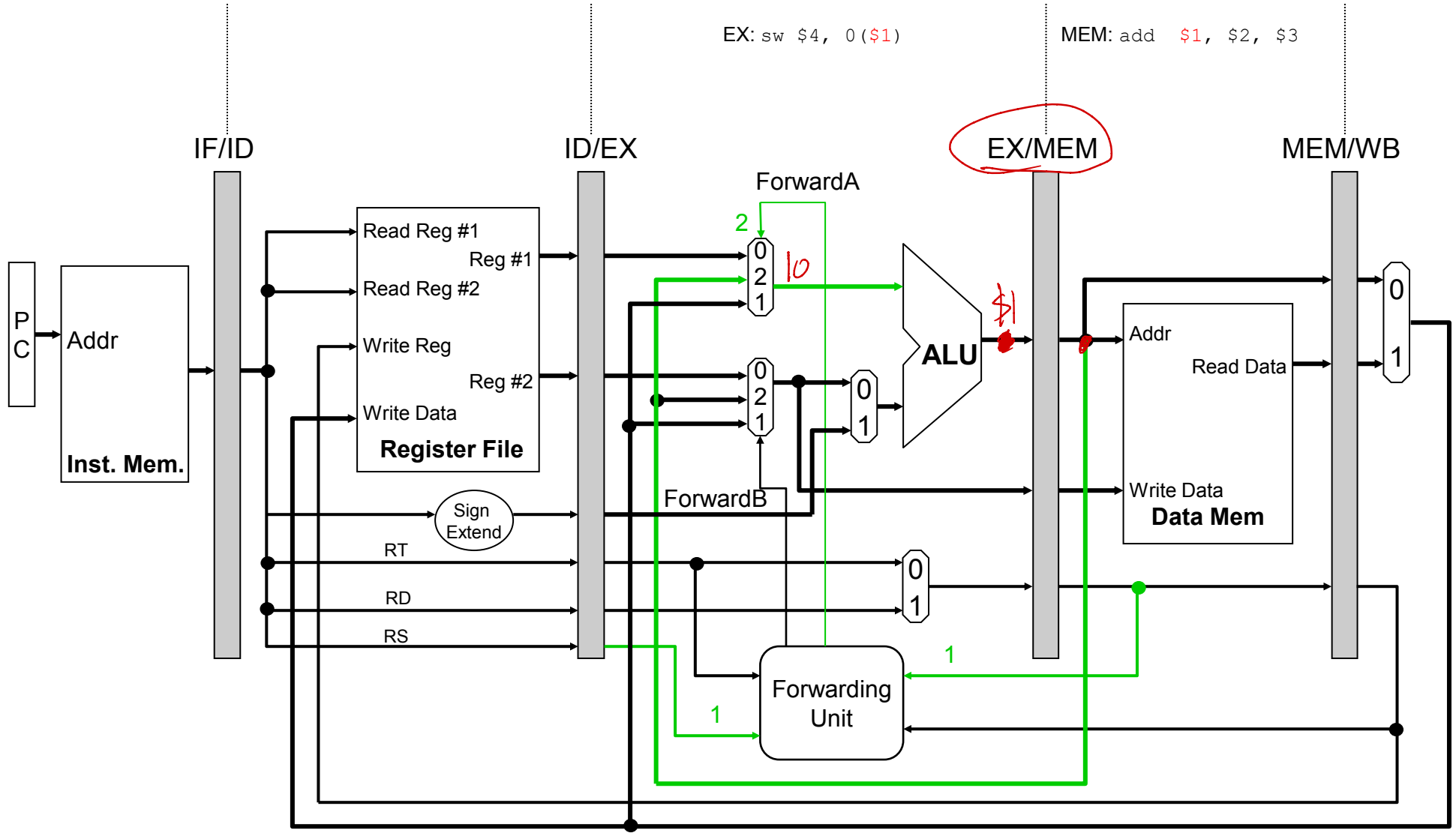
CASE 2:

```
add $1, $2, $3
sw  $1, 0($4)
```

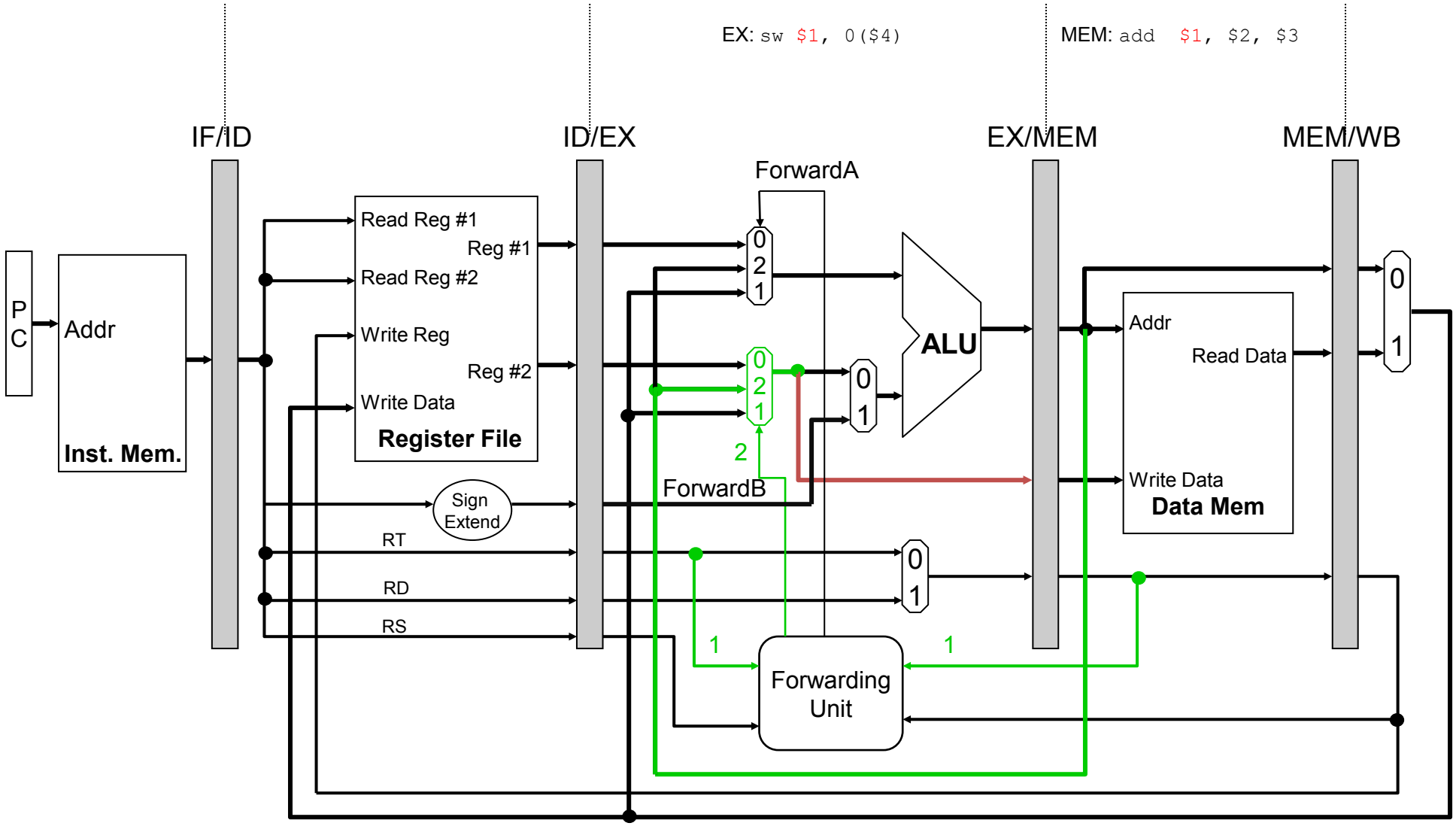


mem[$4的] = $1

CASE 1: forward to R[rs]

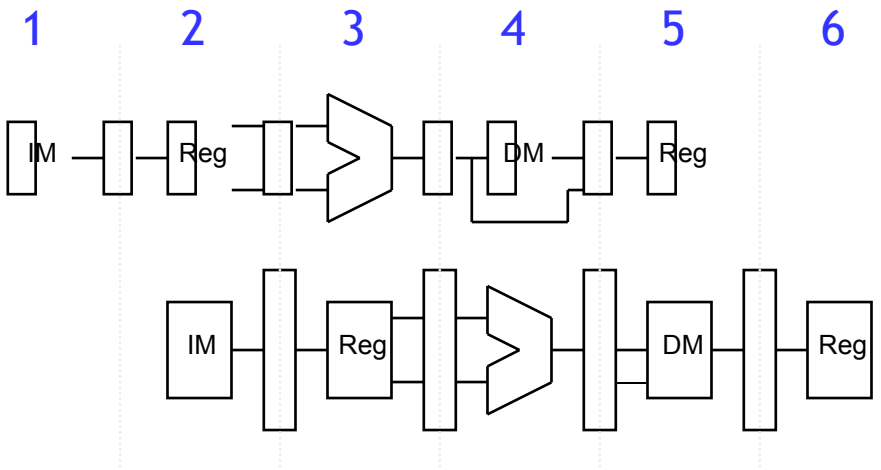


CASE 2: forward to R[rt]



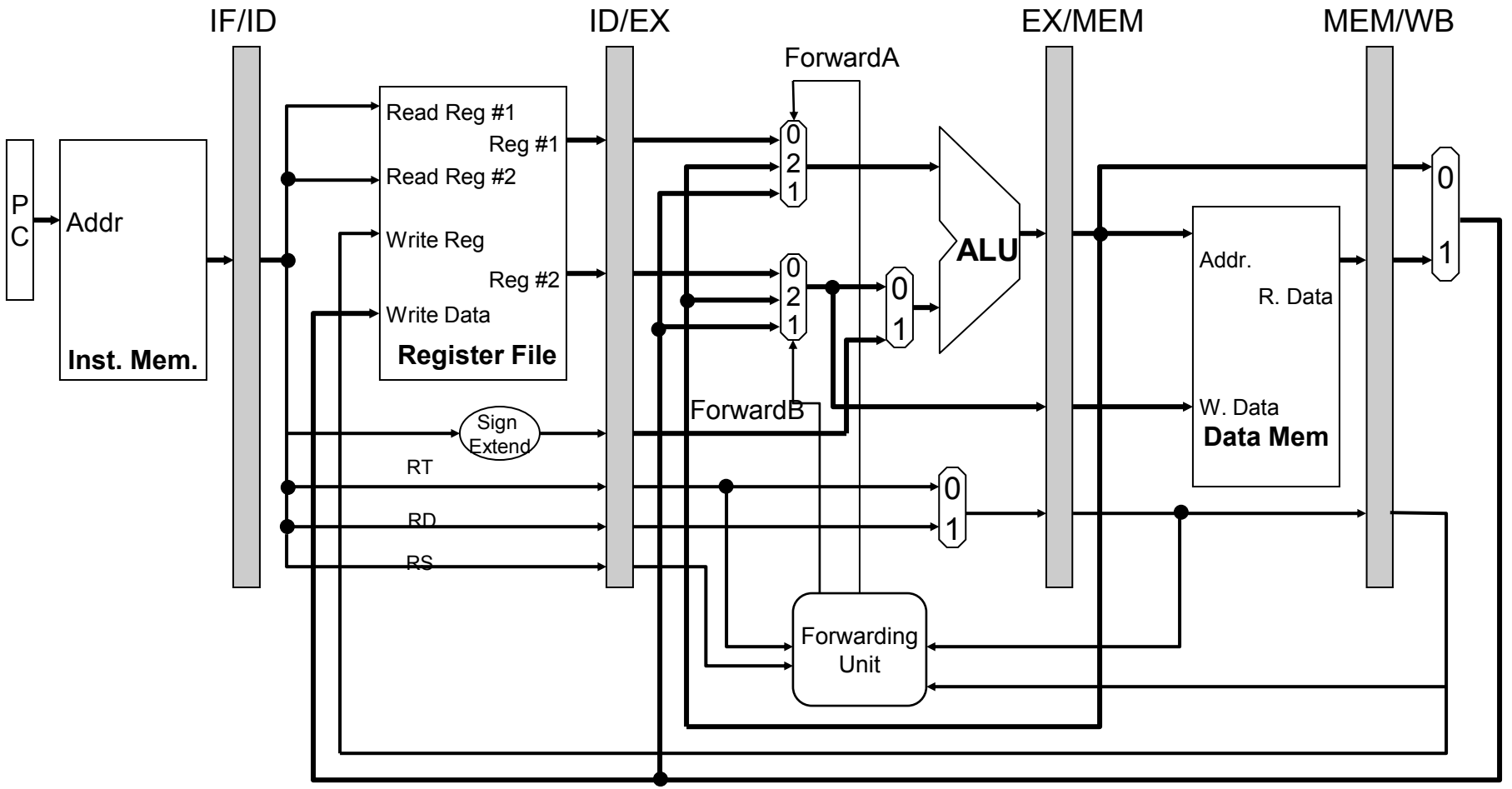
CASE 3

```
lw   $1, 0($2)
sw   $1, 0($4)
```



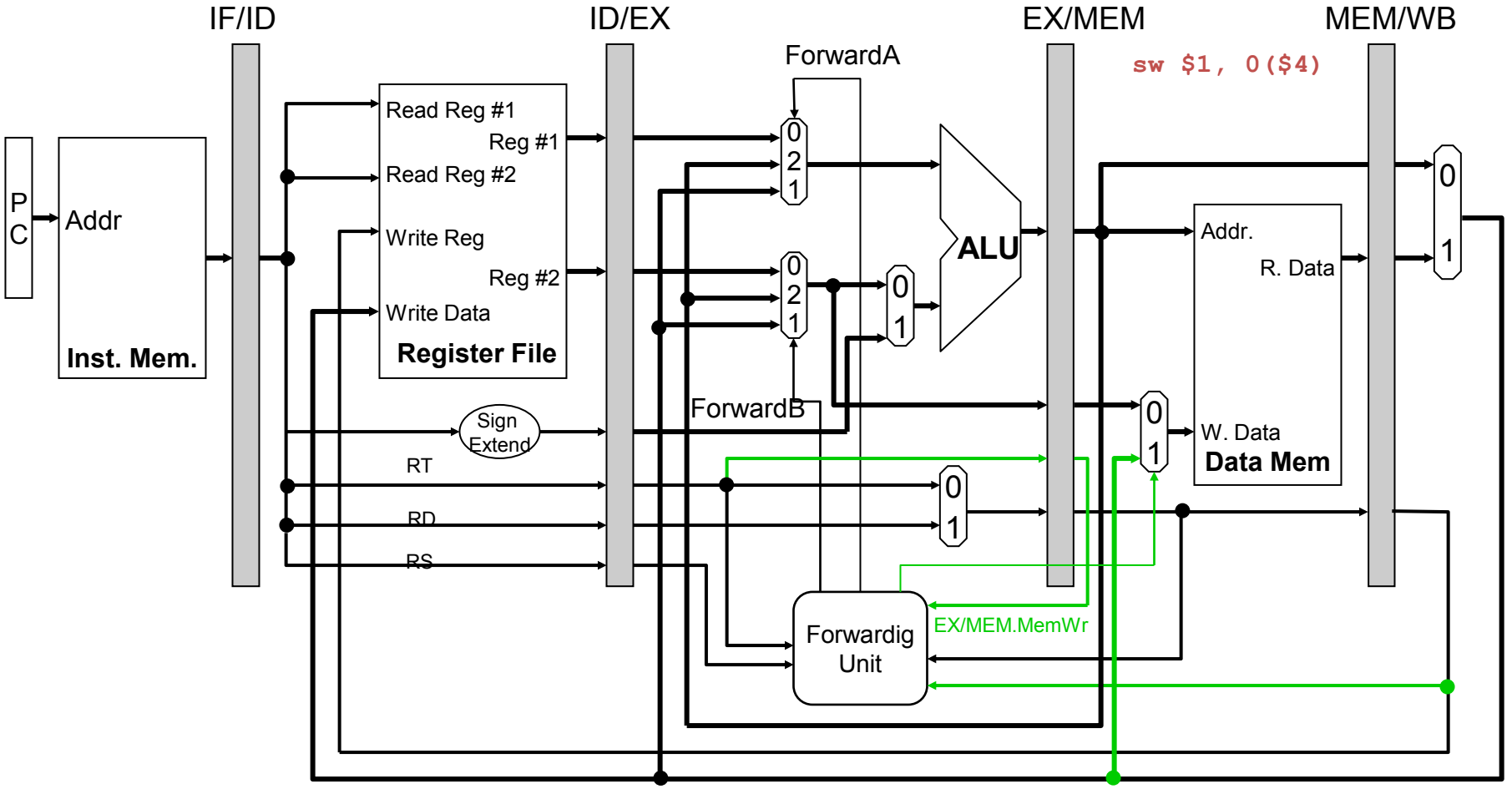
Can the datapath handle case 3?

```
lw   $1, 0($2)
sw   $1, 0($4)
```

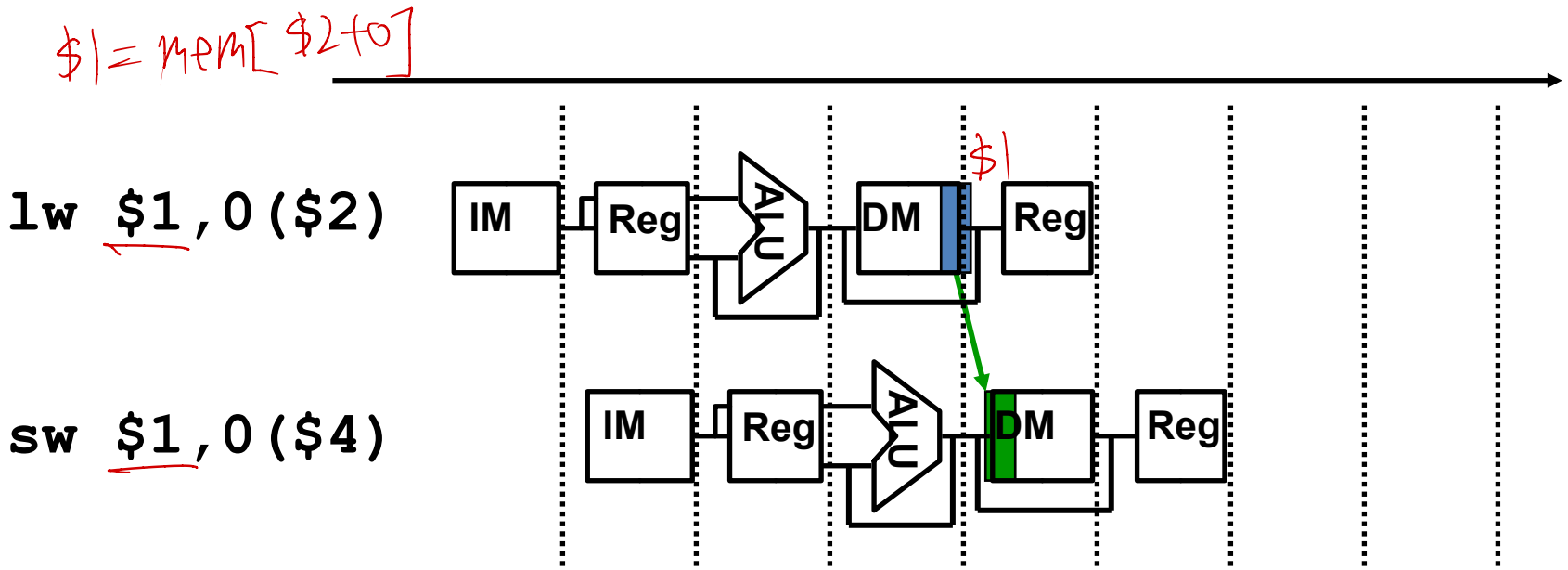


Modify the datapath

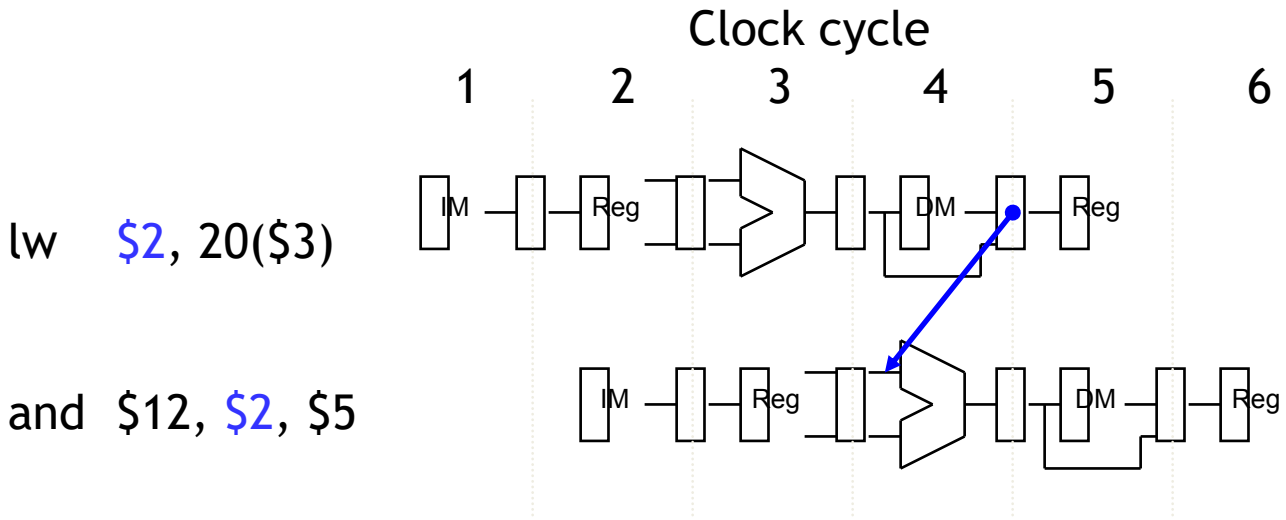
```
lw   $1, 0($2)
sw   $1, 0($4)
```



Handle case 3



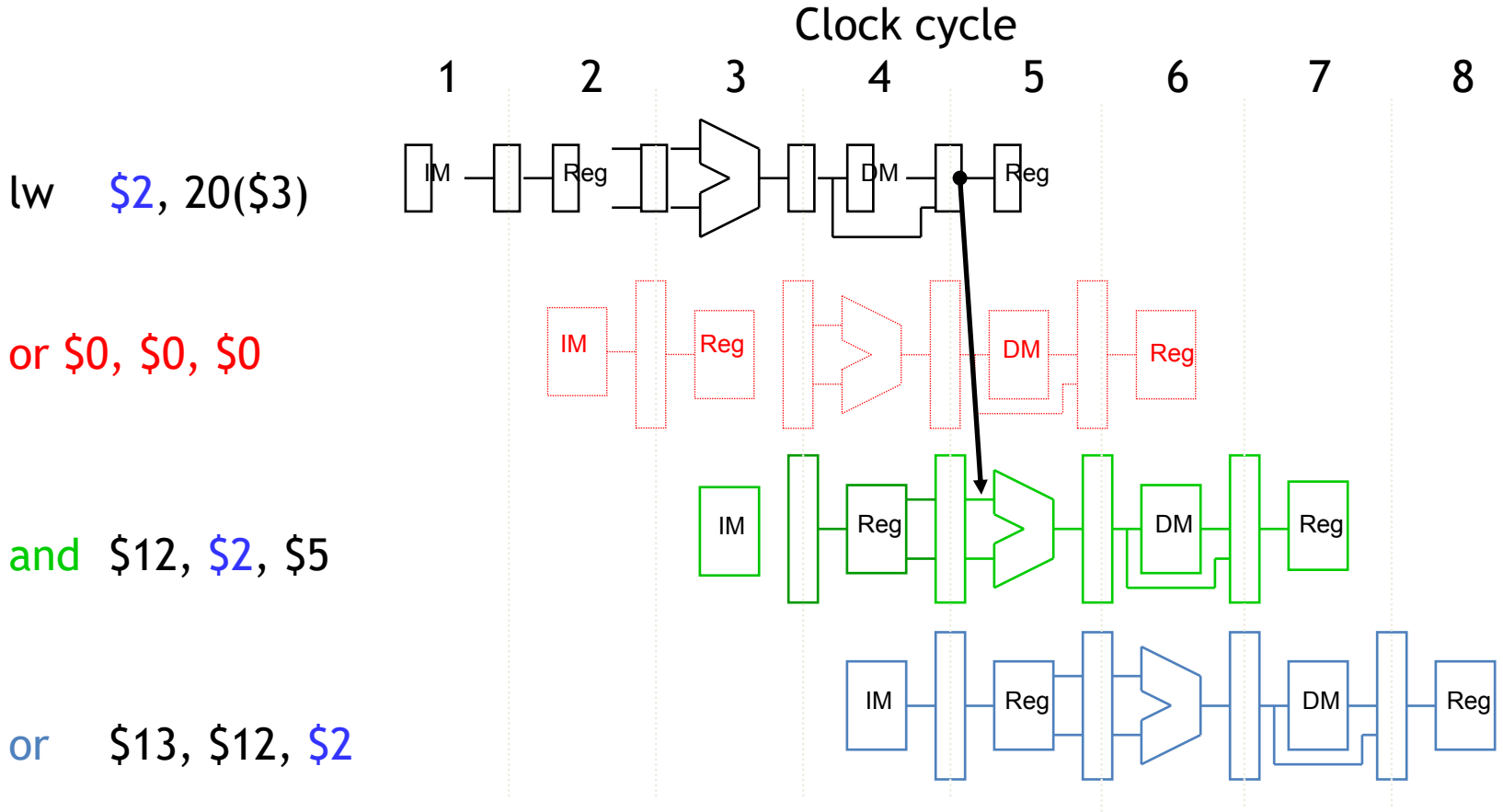
Can forwarding resolve this example?



- $2 is only available at the beginning of cycle 5.
- $2 is needed by “and” at the beginning of cycle 4.
- Forwarding doesn’t work.
  - Data is not available when another instruction needs it.
  - Forwarding: data is already available in one of the pipeline stages when another instruction needs it.
- “True” data hazard.

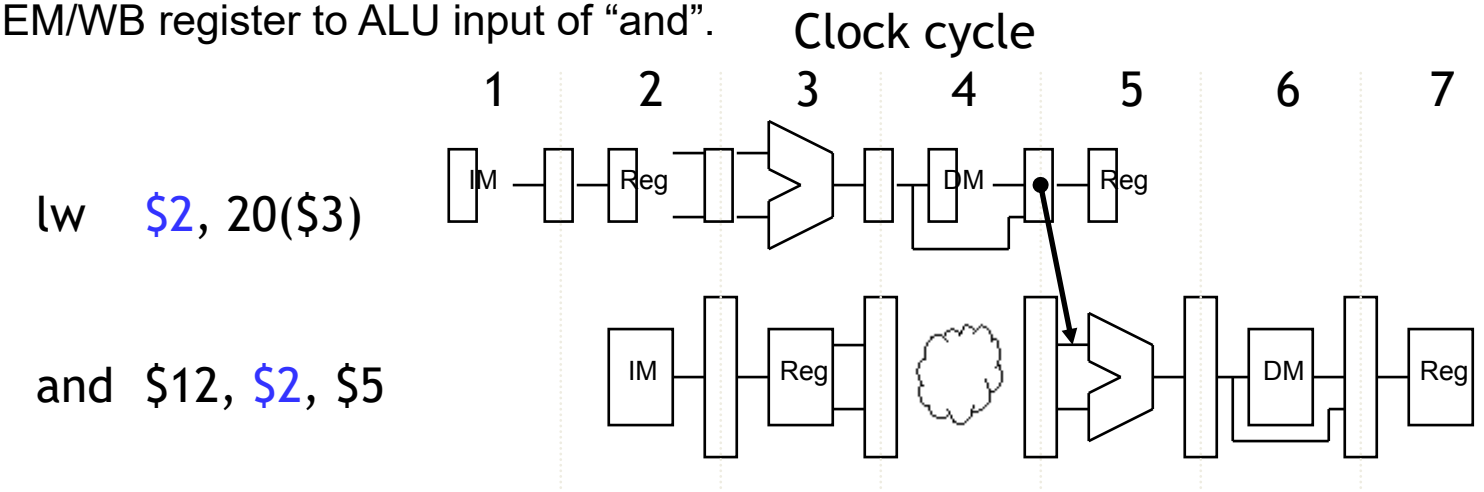
Resolve true data hazard

- By inserting NOP instructions at compile time (software approach).
- NOP is a dummy instruction doing nothing.
- A NOP can be: or $0, $0, $0
  - Doesn't change the value of any register and memory.

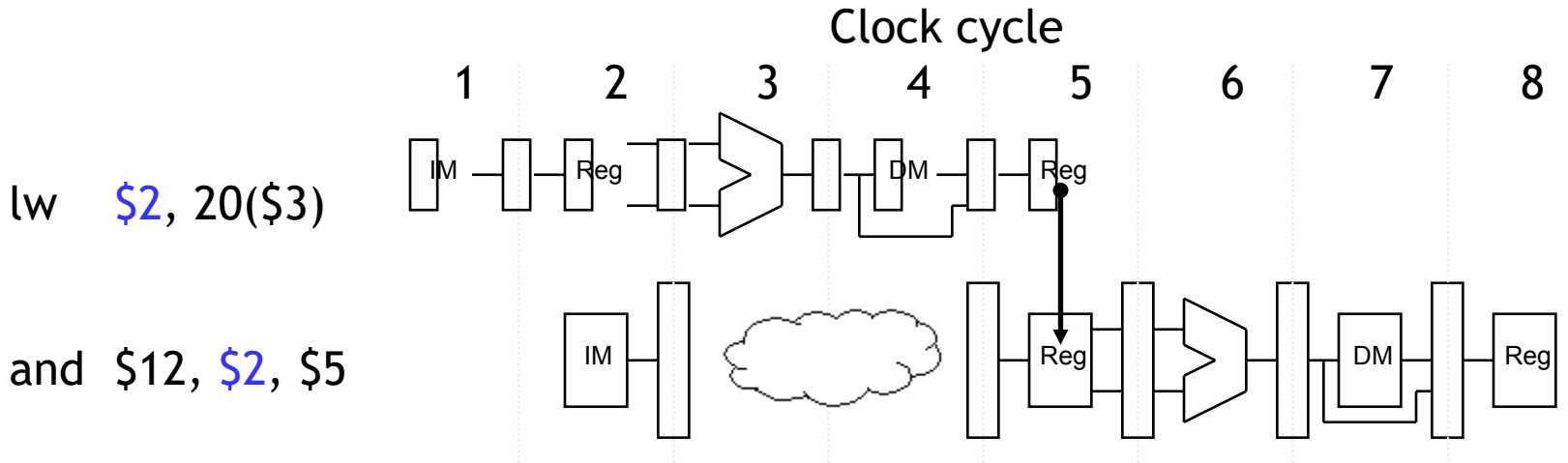


Resolve true data hazard

- By stall the pipeline + forwarding at run time (hardware approach).
- Delay the execution of instruction (adding delay slot).
- E.g. delay the EX stage of “and” by one cycle, and use forwarding to pass data from MEM/WB register to ALU input of “and”.

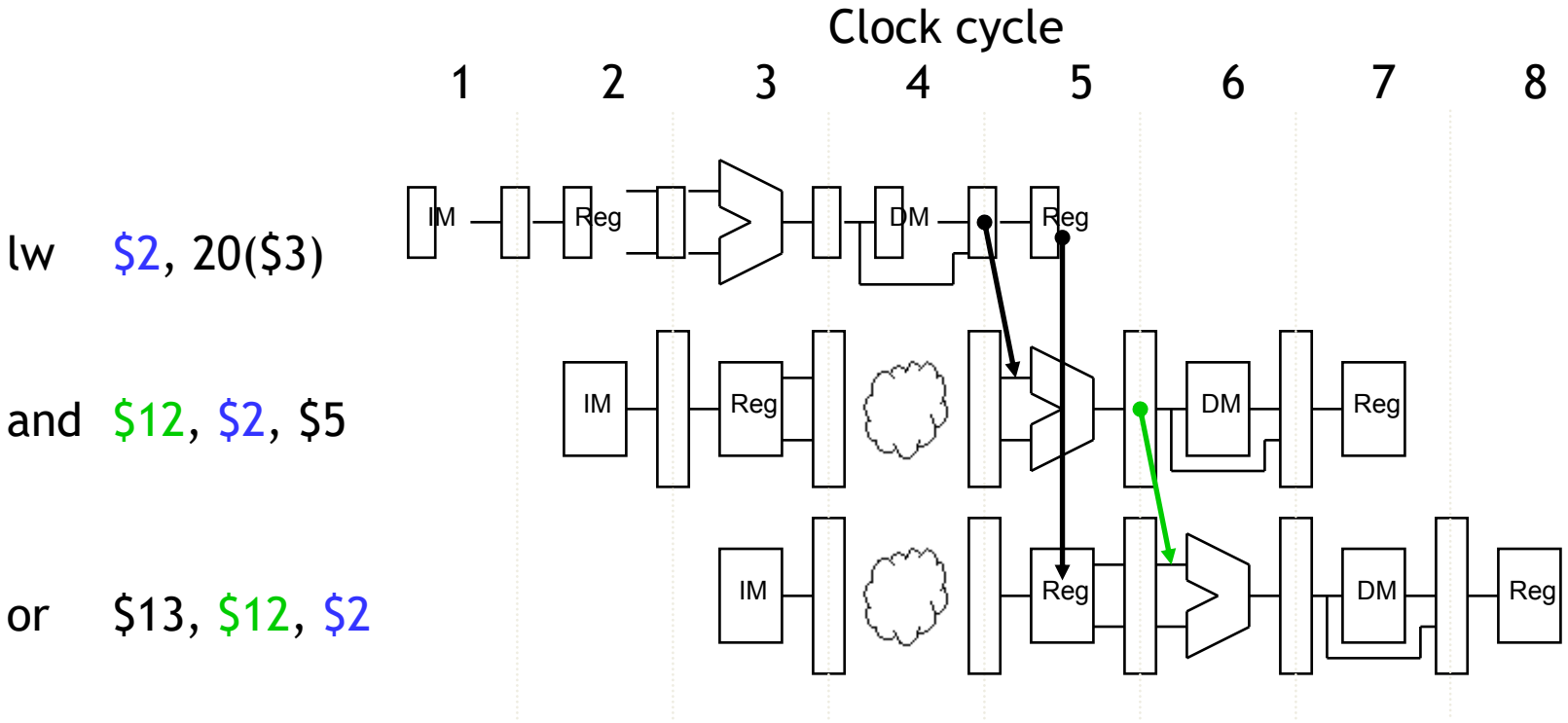


- If there is no forwarding, we need to stall two cycles (adding two delay slots).



Stall

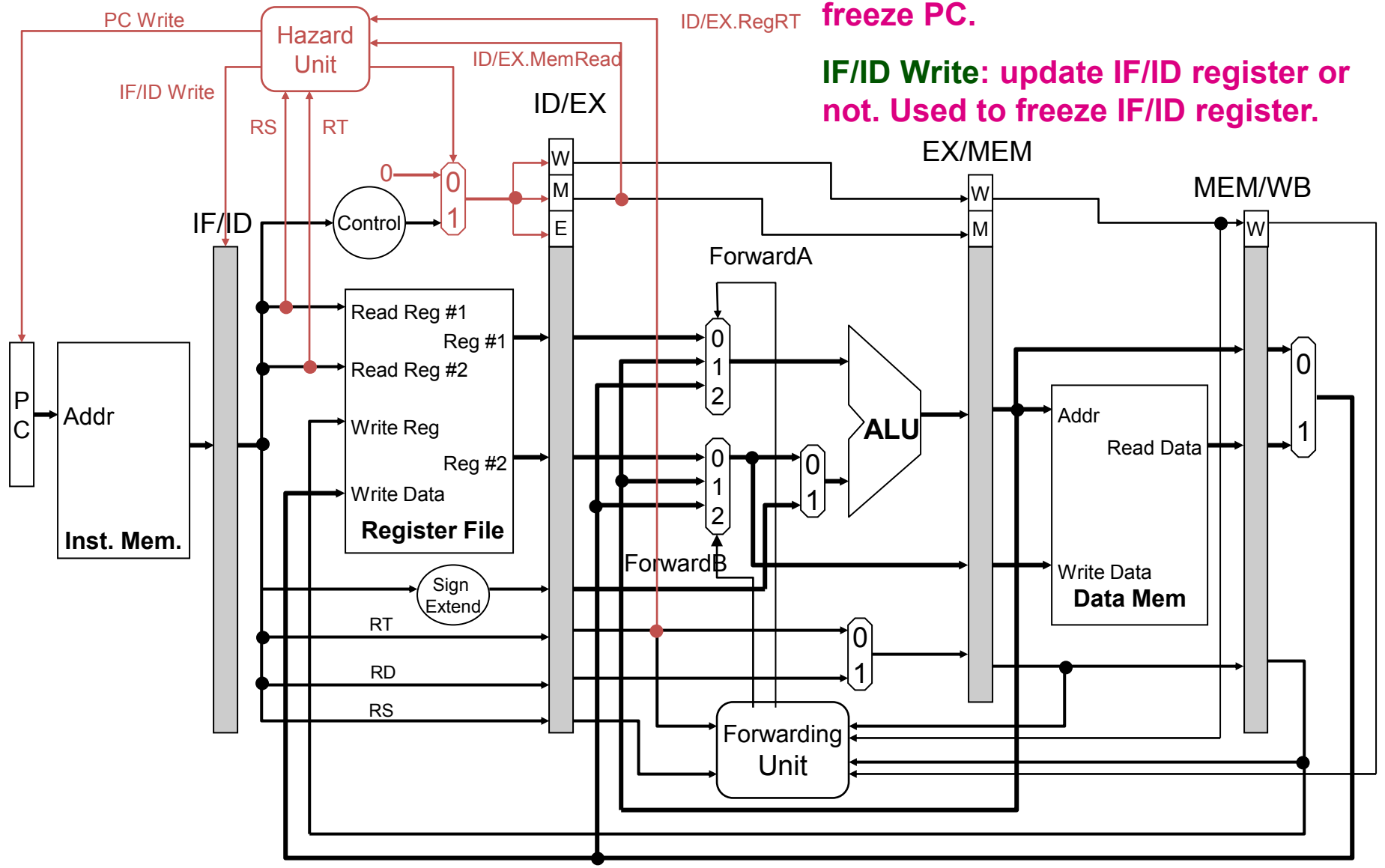
- If we stall one instruction, we need to stall the subsequent instructions too.
  - Prevent structural hazard.
- Can use stall to resolve any hazard
  - But reduce performance.



Adding hazard detection (stall detection)

PC Write: update PC or not. Used to freeze PC.

IF/ID Write: update IF/ID register or not. Used to freeze IF/ID register.



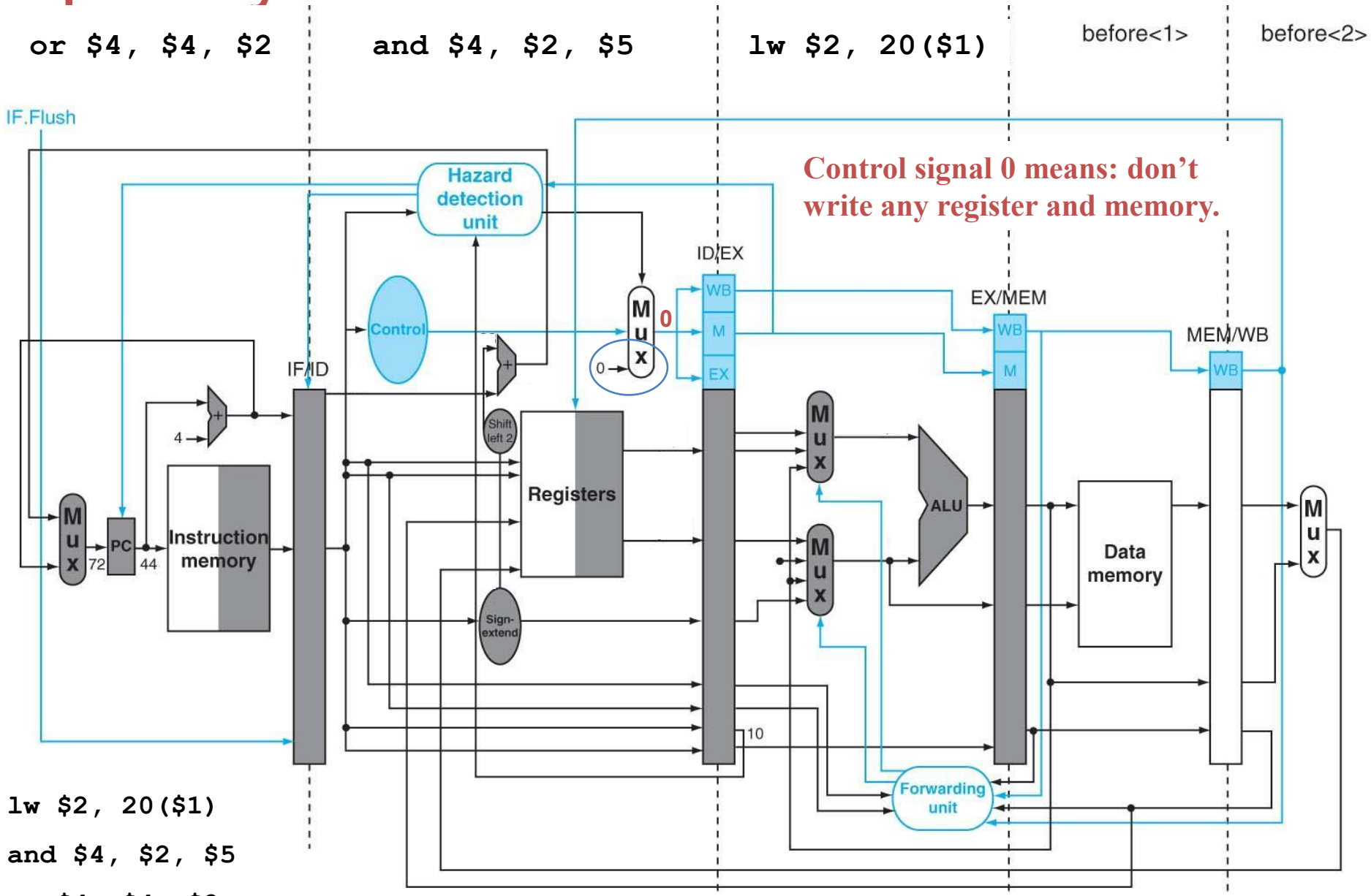
Hazard detection unit (stall detection)

- Inputs of the hazard detection unit.
  - IF/ID.RegRs and IF/ID.RegRt, the source registers for the second instruction.
  - ID/EX.MemRead and ID/EX.RegRt, to determine if the first instruction is LW and, if so, to which register it will write.

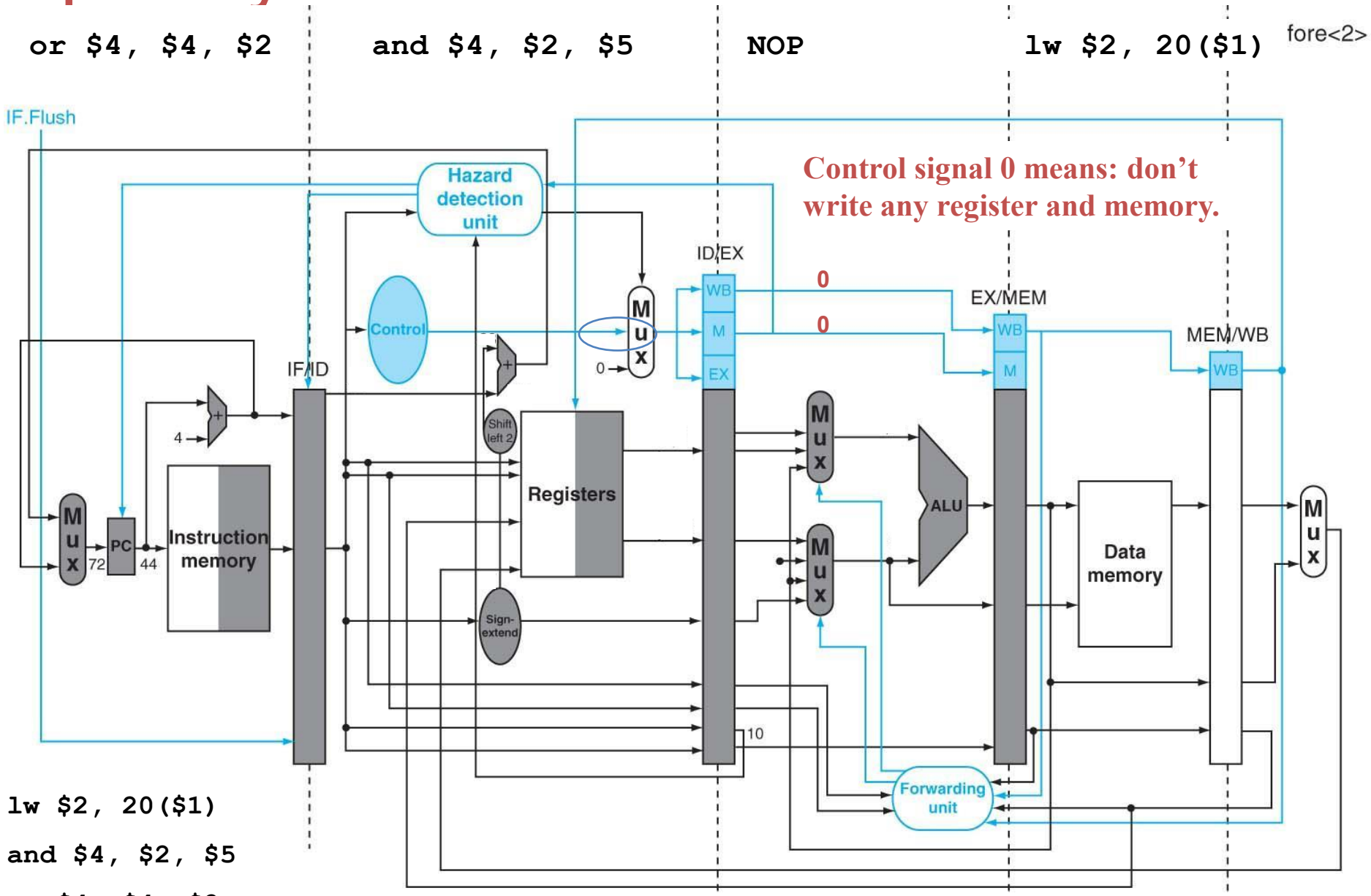
```
if (ID/EX.MemRead and
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))
      stall the pipeline
```

- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls (Freeze PC and IF/ID) or continues.
  - A mux select, which forces control signals of the EX and future stages to 0 in case of a stall (NOP insertion).

Example: cycle 3



Example: cycle 4



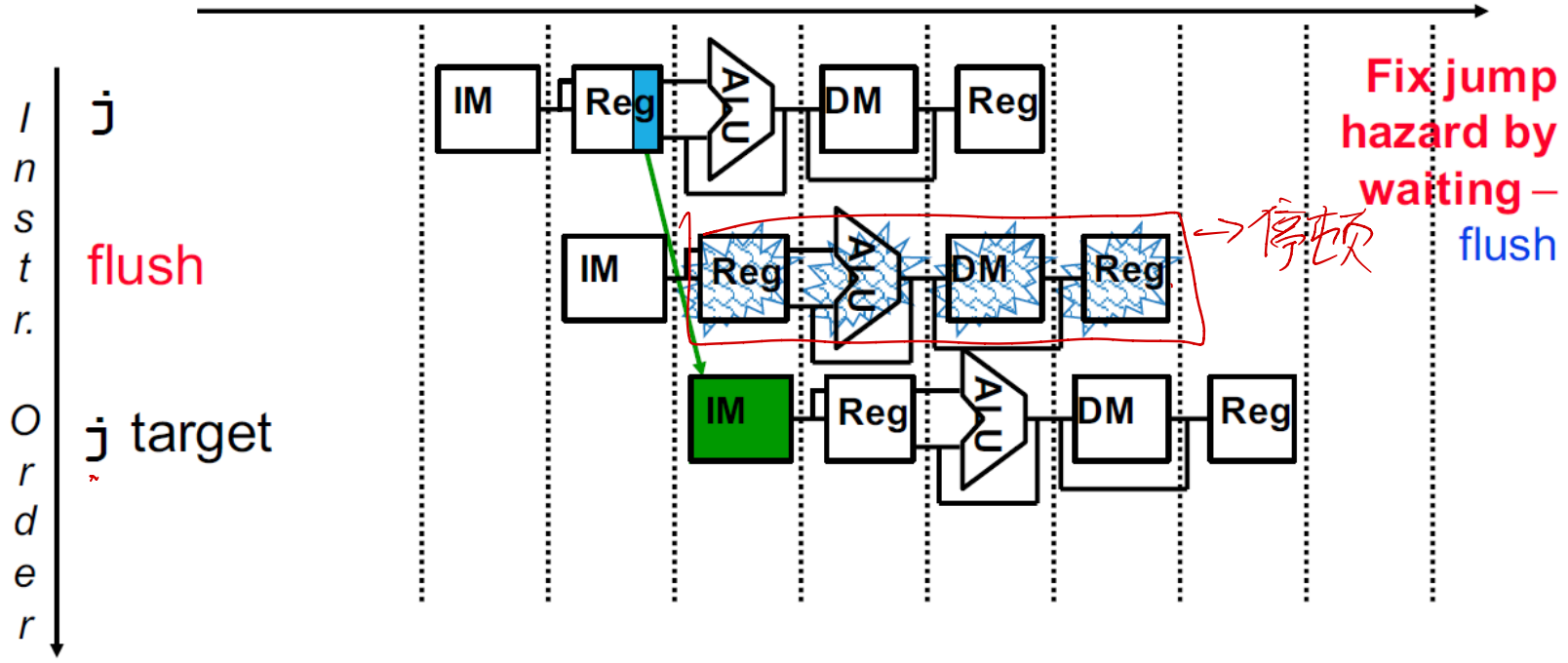
Control Hazards

- When the flow of instruction addresses is not sequential (i.e., PC = PC + 4); incurred by change of flow instructions
  - Unconditional branches (j, jal, jr)
  - Conditional branches (beq, bne)
  - Exceptions
- Possible approaches
  - Stall (impacts CPI)
  - Move decision point as early in the pipeline as possible, thereby reducing the number of stall cycles
  - Delay decision (requires compiler support)
  - Predict and hope for the best !
- Control hazards occur less frequently than data hazards, but there is *nothing* as effective against control hazards as forwarding is for data hazards.

Control Hazards 1: Jumps Incur One Stall

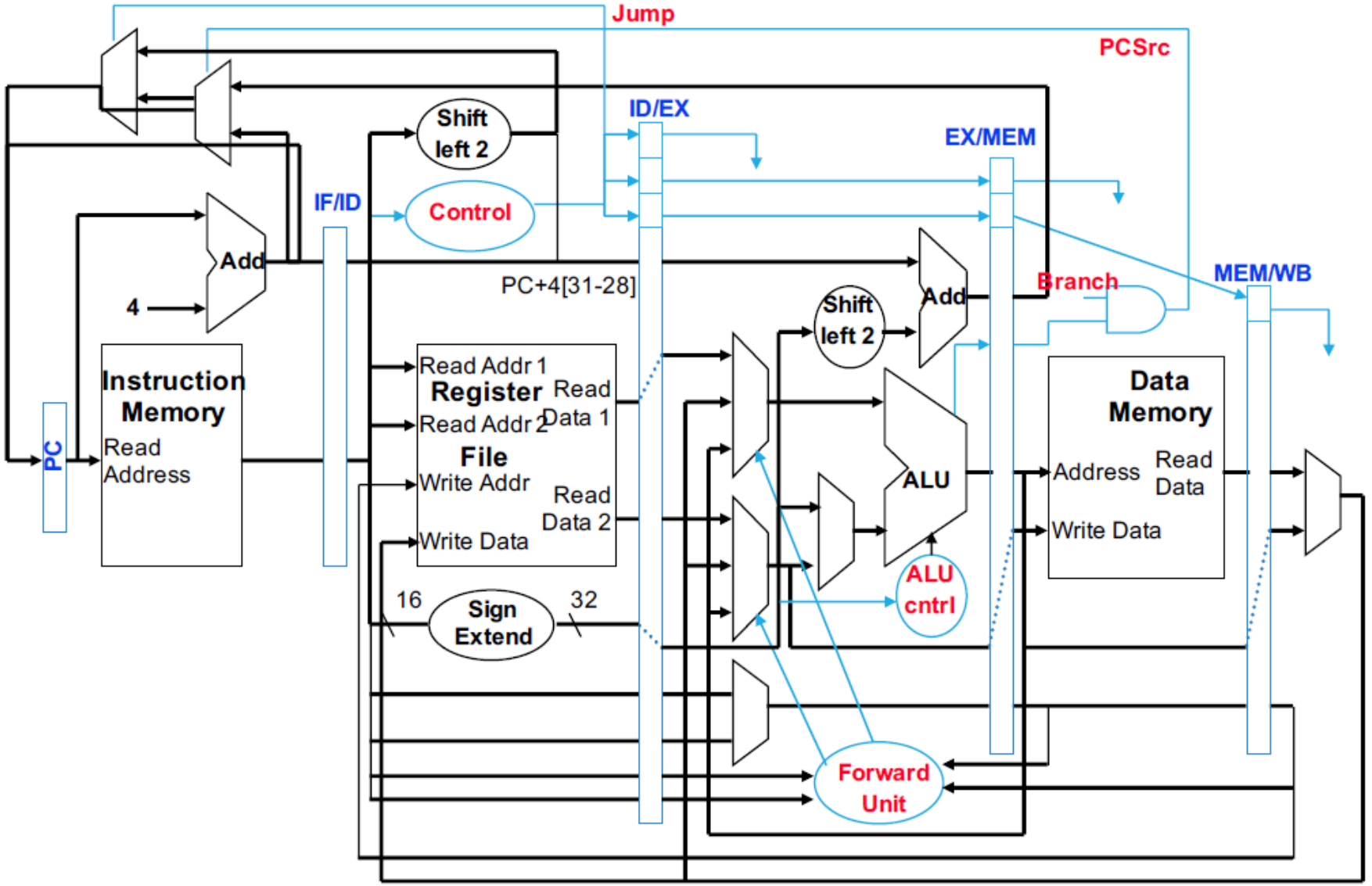
connot predict, since it is unconditional

- Jumps not decoded until ID, so one flush is needed
  - To flush, set IF.Flush to zero the instruction field of the IF/ID pipeline register (turning it into a nop)

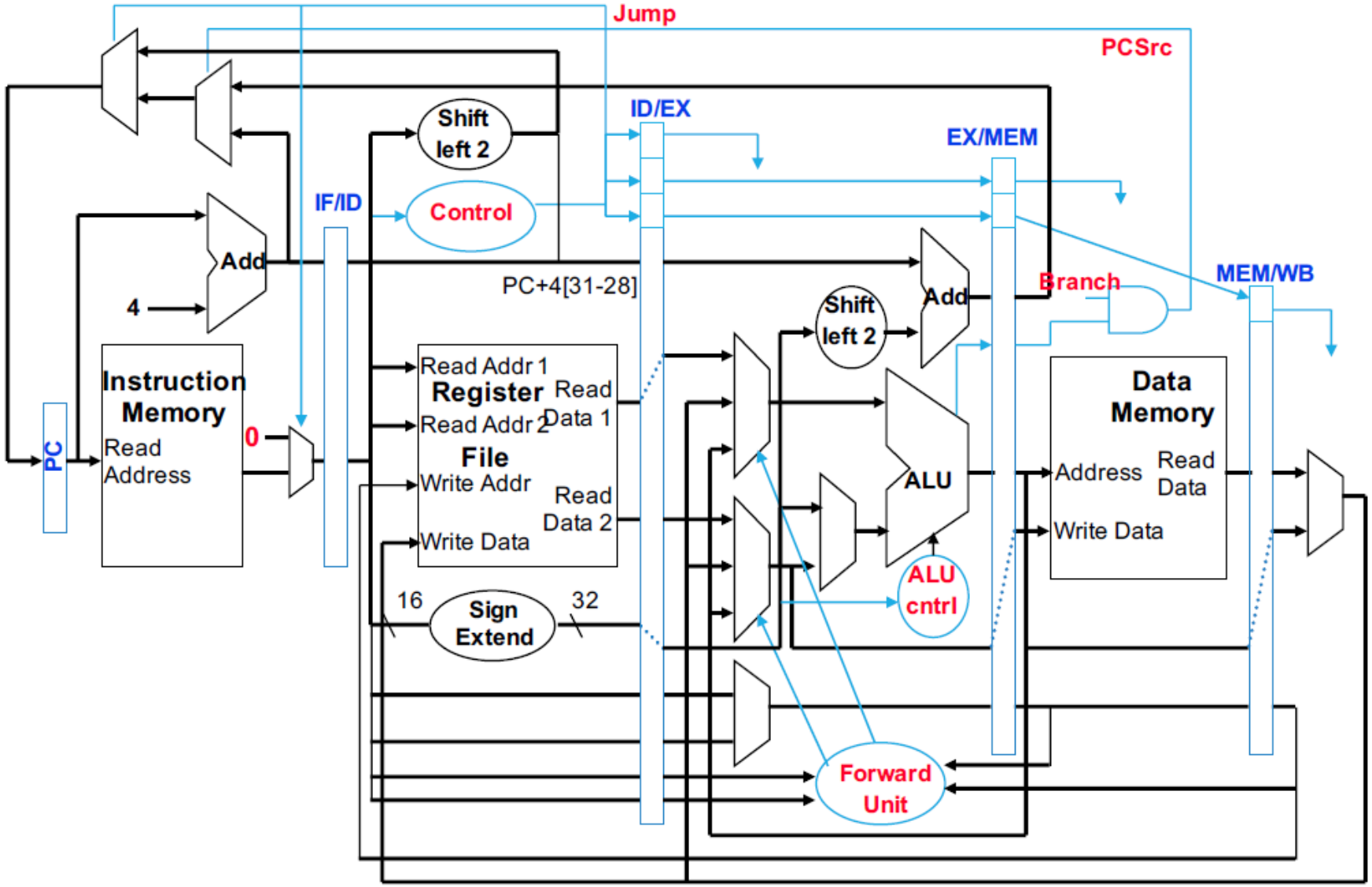


- Fortunately, jumps are very infrequent – only 3% of the SPECint instruction mix

Datapath Branch and Jump Hardware

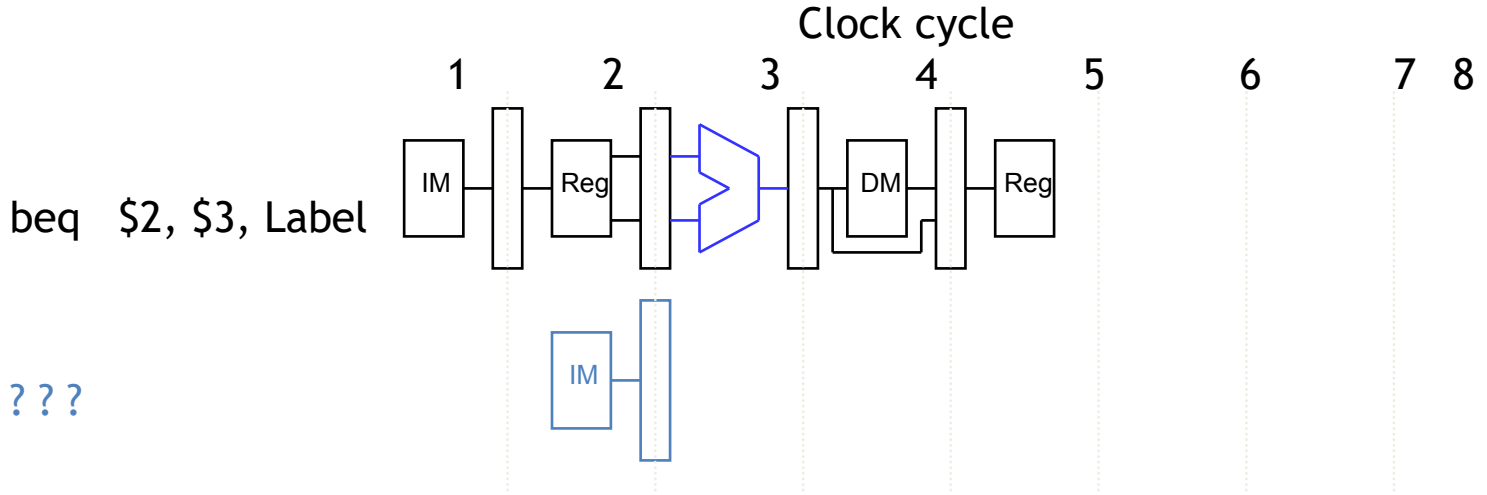


Supporting ID Stage Jumps

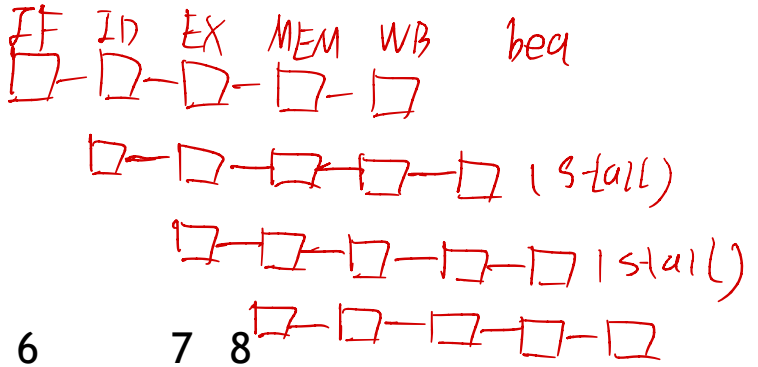


Control Hazard 2: Branch Instr

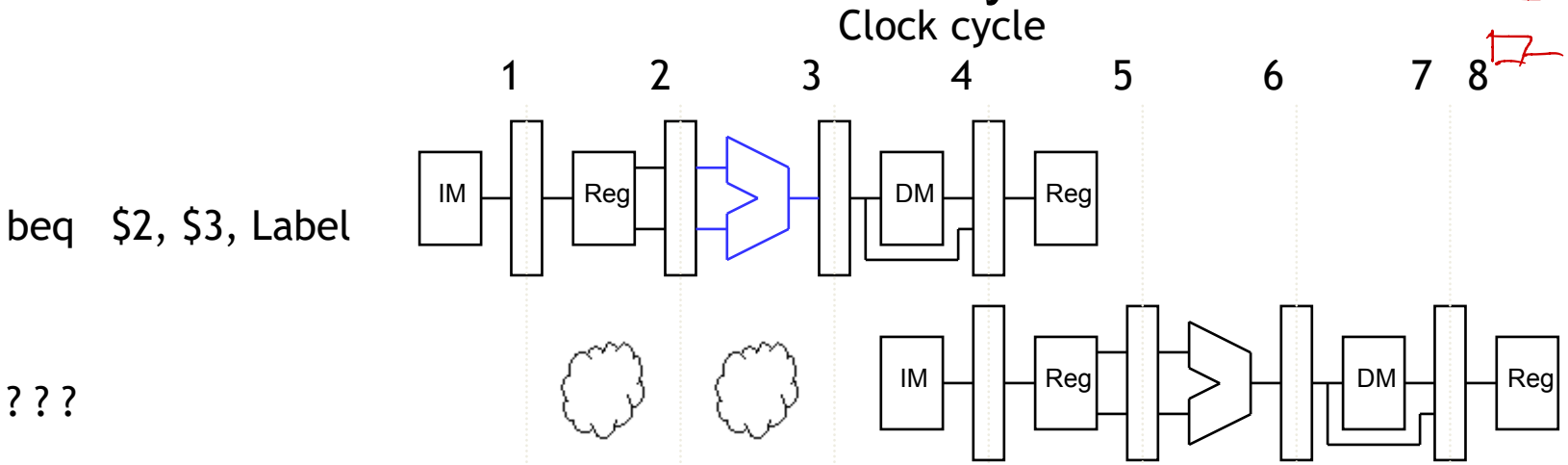
- Branch instruction.
  - Branch is calculated at EX stage.
  - Branch decision is not known until EX is finished, i.e. next instruction cannot be loaded until EX is finished → control hazard.



At compile time, insert NOP to stall the pipeline



- Insert two NOPs below BEQ to stall for two cycles.



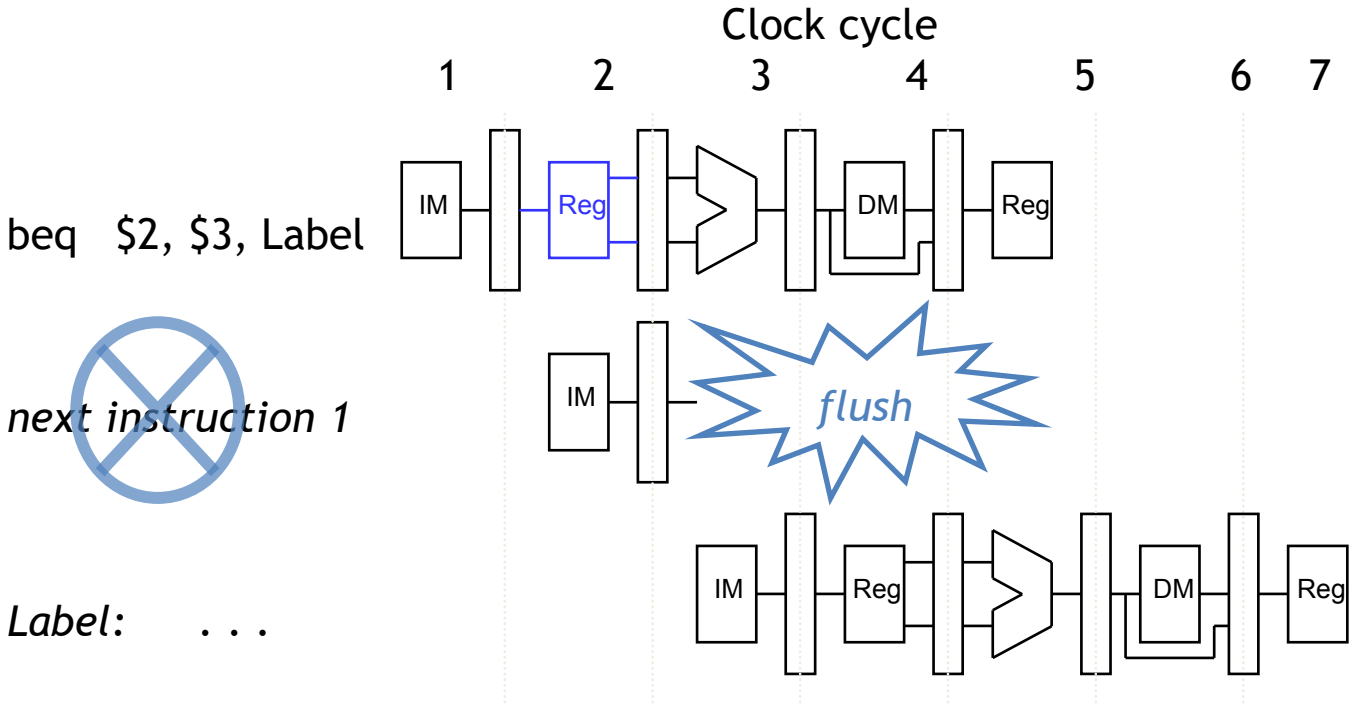
- But stall often reduces performance of the pipeline.

```
Loop:   addi $t1, $t1, 1
        bne  $t1, $t2, Loop
```

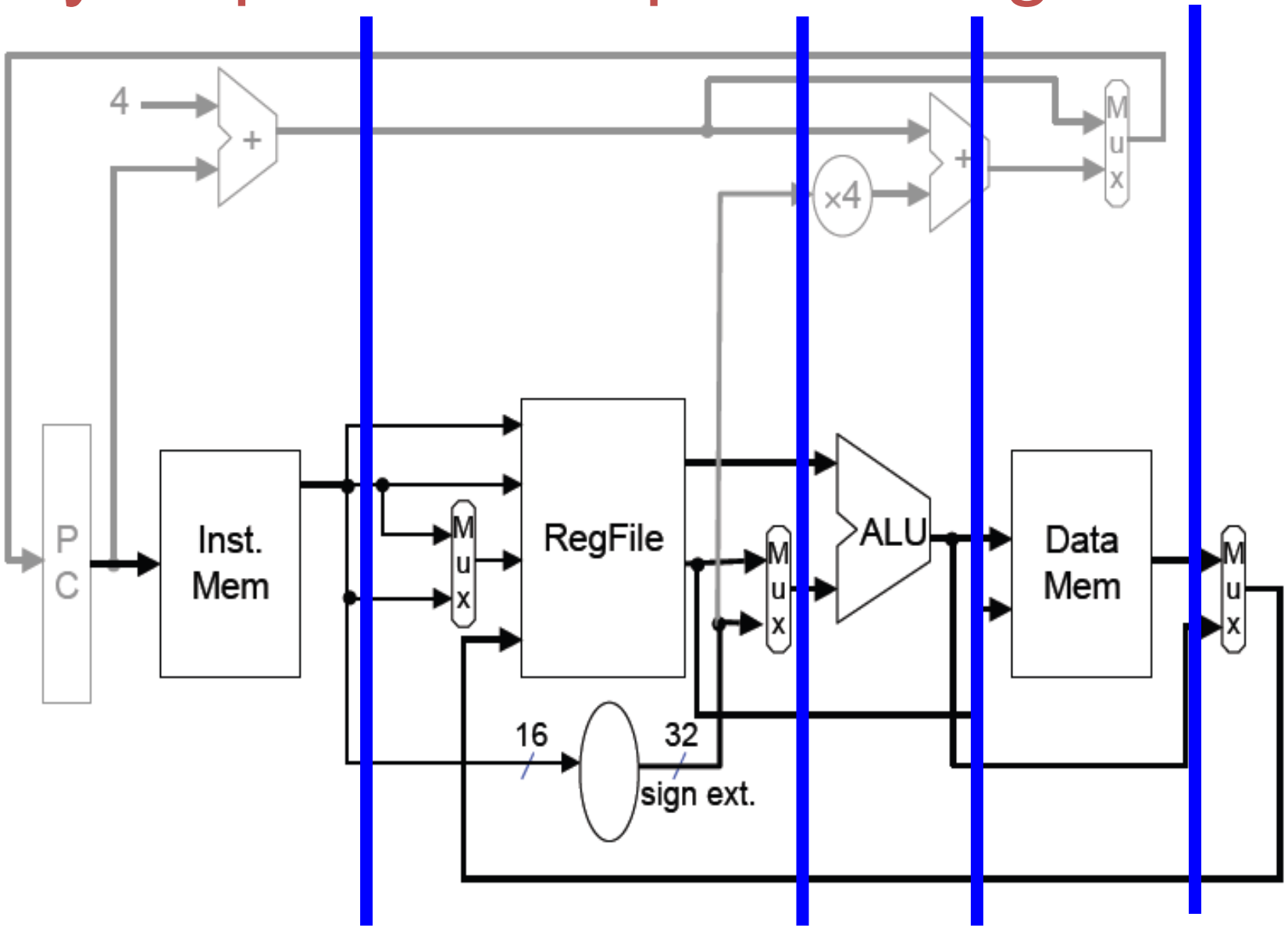
  - Assume $t1 = 0 and $t2 = 1000 initially. We are expecting the loop can finish within 2000 cycles, but with branch stalling, it takes 4000 cycles.

Calculate the branch condition earlier

- Can we calculate the branch condition earlier?
  - Yes. Add extra functional unit to calculate the branch condition at ID stage → The branch condition will be available at cycle 3.
  - If misprediction, only need to flush one instruction.

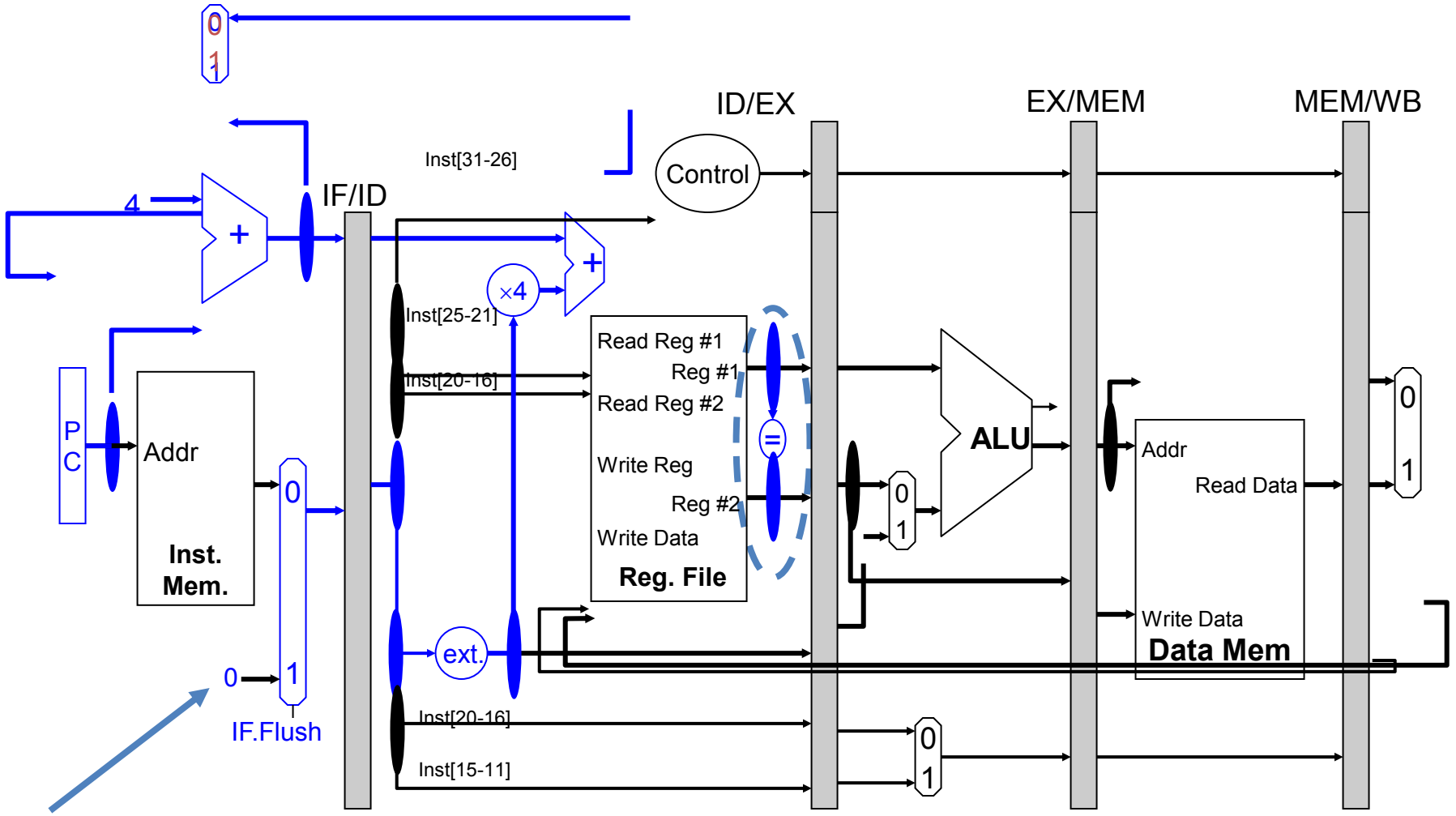


Single cycle processor: partitioning



Branch detection at ID stage

check if two registers are equal.



Used to insert NOP instruction into pipeline when there is a branch instruction.

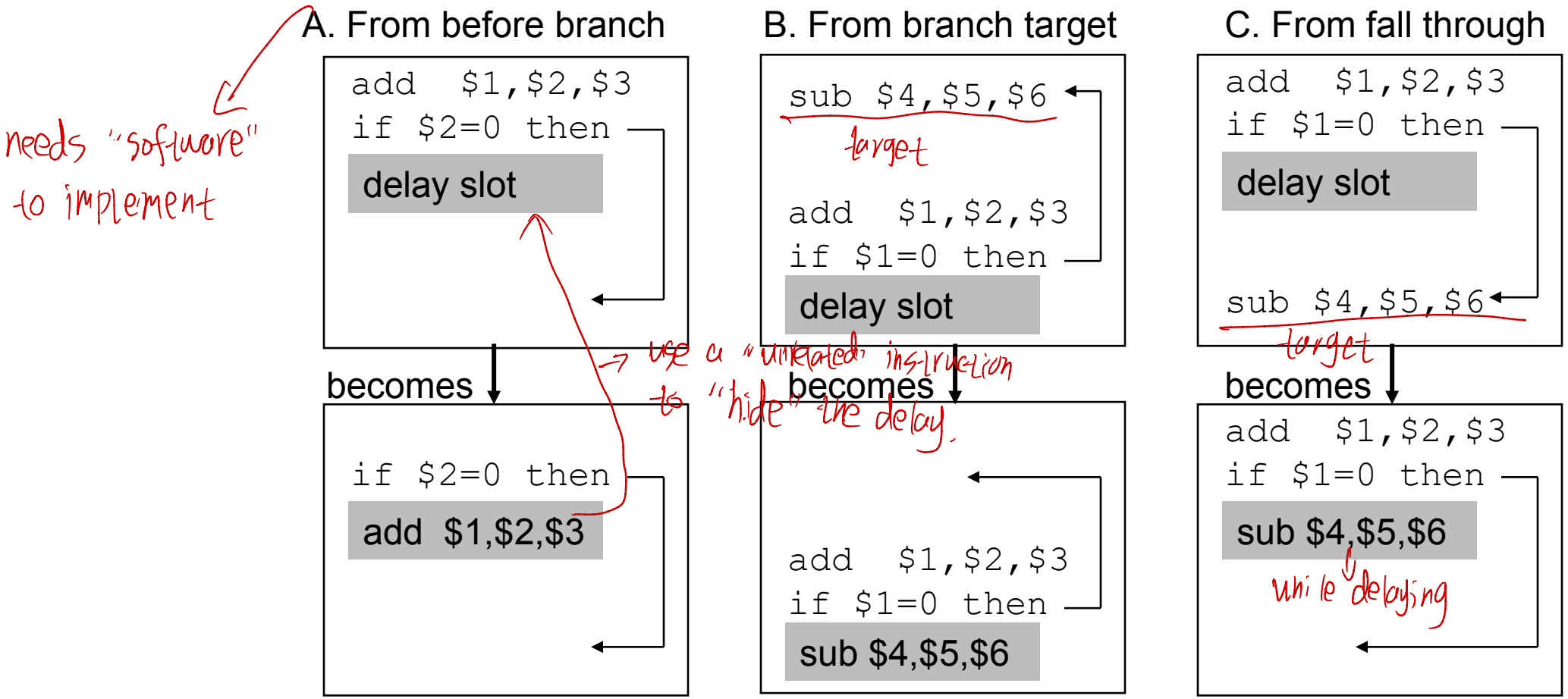
Two “Types” of Stalls

- Nop instruction (or bubble) inserted between two instructions in the pipeline (as done for load-use situations)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle (“bounce” them in place with write control signals)
  - Insert `nop` by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction squashing) were an instruction in the pipeline is replaced with a nop instruction (as done for instructions located sequentially after j instructions)
  - Zero the control bits for the instruction to be flushed

Delayed Branches

- If the branch hardware has been moved to the ID stage, then we can eliminate all branch stalls with delayed branches which are defined as always executing the next sequential instruction after the branch instruction – the branch takes effect *after* that next instruction.
- MIPS compiler moves an instruction to immediately after the branch that is not affected by the branch (a safe instruction) thereby hiding the branch delay.
- With deeper pipelines, the branch delay grows requiring more than one delay slot
  - Delayed branches have lost popularity compared to more expensive but more flexible (dynamic) hardware branch prediction
  - Growth in available transistors has made hardware branch prediction relatively cheaper

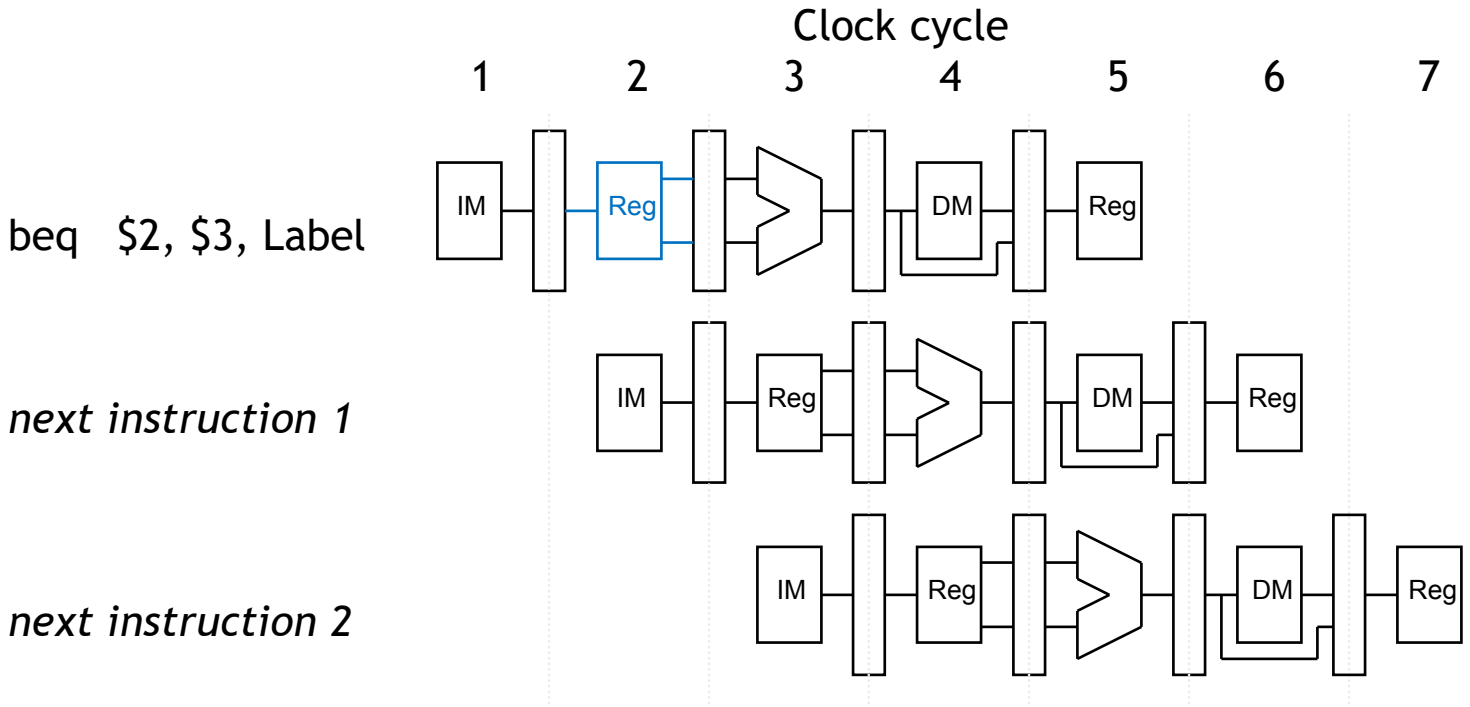
Scheduling Branch Delay Slots



- A is the best choice, fills delay slot and reduces IC
- In B and C, the `sub` instruction may need to be moved
- In B and C, must be okay to execute `sub` when branch fails

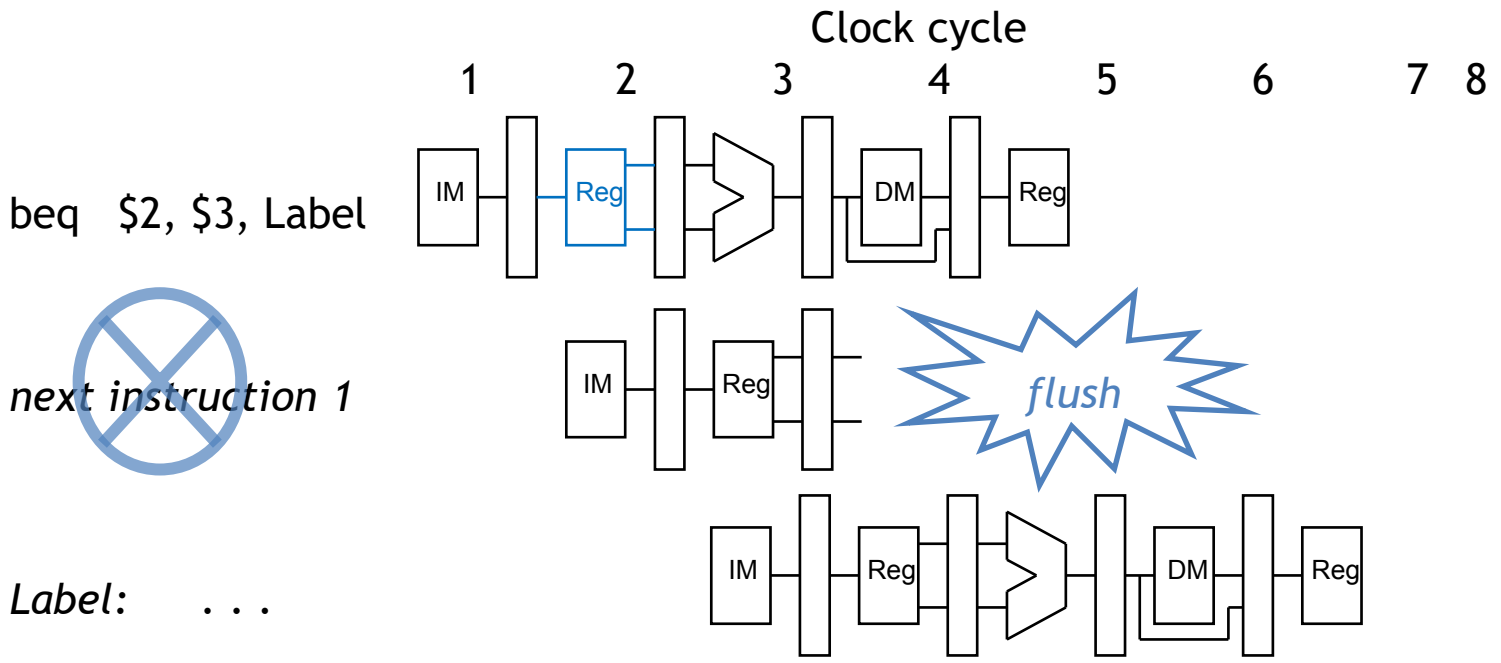
At run time, using branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue to execution the next instruction, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch Prediction

- If our guess is wrong, that means we have already started executing two incorrect instructions. As a result, we need to discard, or flush, those instructions from the pipeline and begin executing the right one from the branch target address, Label.
- If misprediction, need to flush one instructions.



Static Branch Prediction

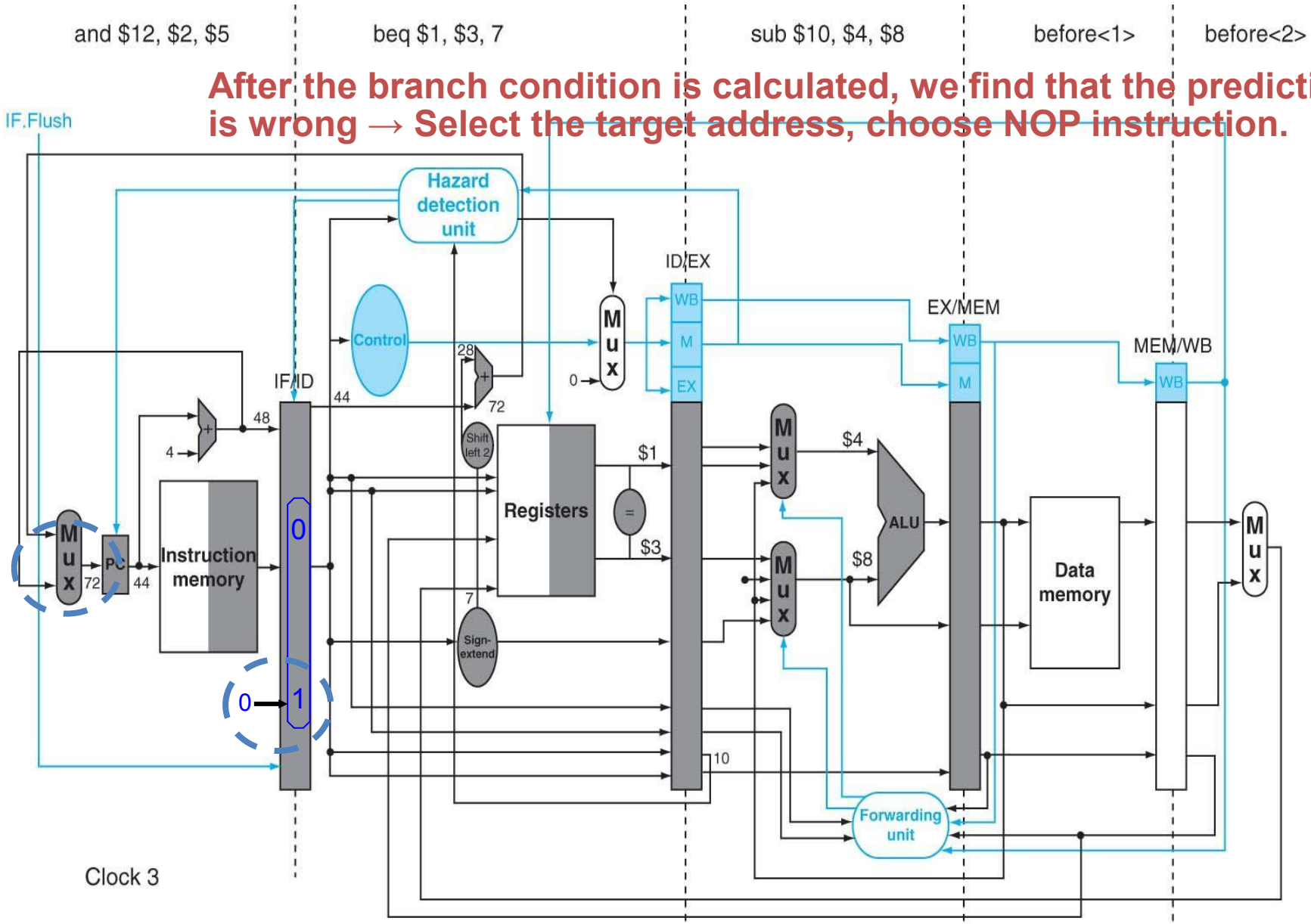
- Resolve branch hazards by assuming a given outcome and proceeding without waiting to see the actual branch outcome
  - Predict not taken – always predict branches will not be taken, continue to fetch from the sequential instruction stream, only when branch *is* taken does the pipeline stall
    - If taken, flush instructions after the branch in IF stage if branch logic in ID – one stall
    - ensure that those flushed instructions haven't changed the machine state – automatic in the MIPS pipeline since machine state changing operations are at the tail end of the pipeline (MemWrite (in MEM) or RegWrite (in WB))
    - restart the pipeline at the branch destination
  - Predict taken – predict branches will always be taken
    - Predict taken *always* incurs one stall cycle (if branch destination hardware has been moved to the ID stage)
    - Is there a way to "cache" the address of the branch target instruction ??

Example

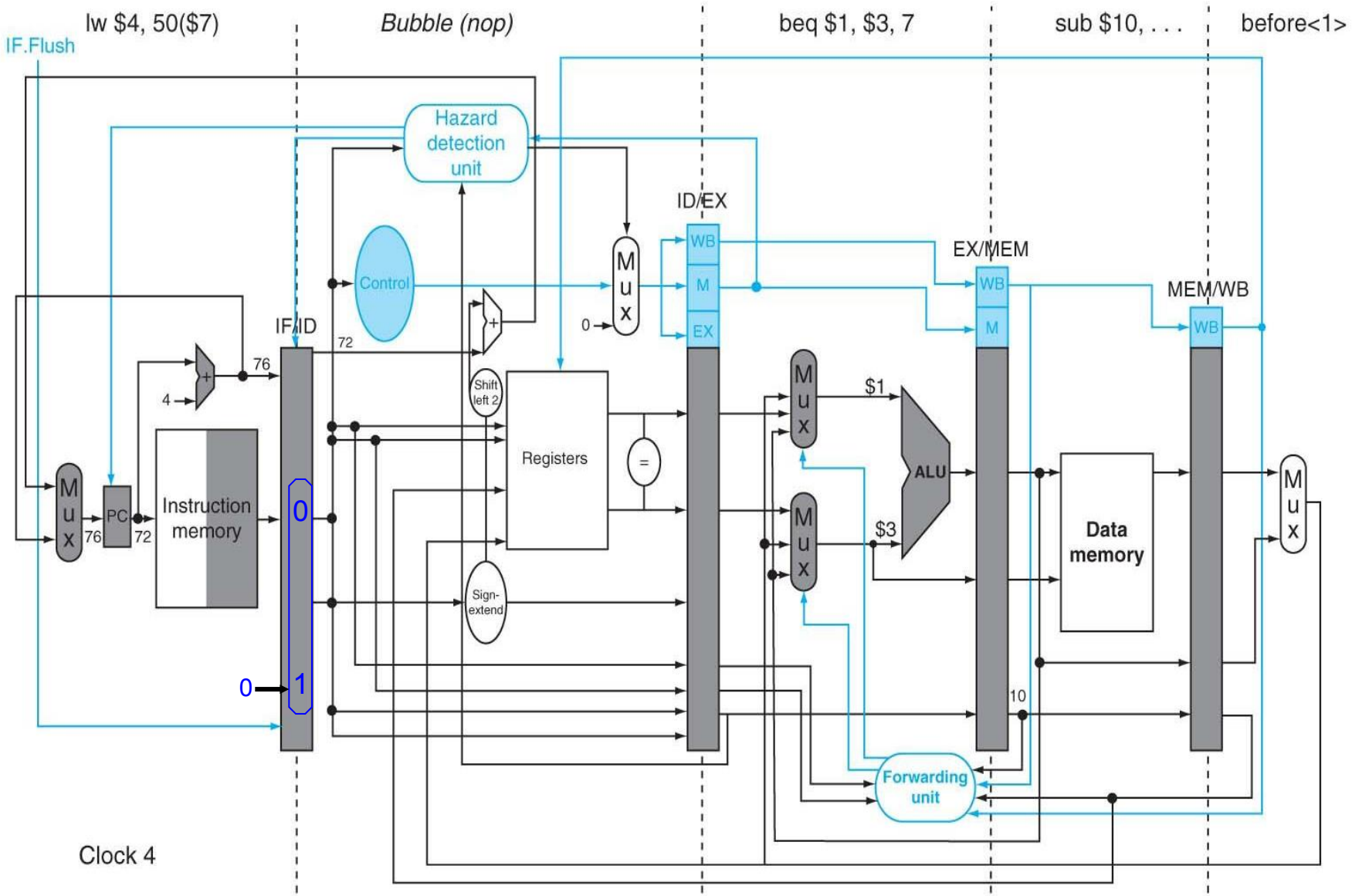
- Assumption: The prediction strategy is that branch not taken.

```
36#          sub $10, $4, $8
40#          beq $1, $3, TARGET
44#          and $12, $2, $5
48#          or $13, $2, $6
52#          add $14, $4, $2
56#          slt $15, $6, $7
...
72# TARGET:     lw $4, 50($7)
```

Guess branch not taken: load instruction “and”



Fetch target instruction “lw”, insert “nop”



Performance gain of branch prediction

- In general, branch prediction is worth
  - Example: for (i = 0; i < 100; i++)

    if we always predict (i < 100) is true → 99% correct.
  - Mispredicting a branch means that one clock cycles are wasted.
- A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - If our predictions are just occasionally correct, e.g. 10% accuracy, then it is better to stall and waste more than one cycles for *every* branch.
- We must be careful that instructions do not modify registers or memory before they get flushed.

Branching Structures

- Predict not taken works well for "top of the loop" branching structures
  - But such loops have jumps at the bottom of the loop to return to the top of the loop – and incur the jump stall overhead
- Predict not taken doesn't work well for "bottom of the loop" branching structures

```
Loop: beq $1,$2,Out
      1nd loop instr
            .
            .
            .
      last loop instr
      j   Loop
Out:  fall out instr
```

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

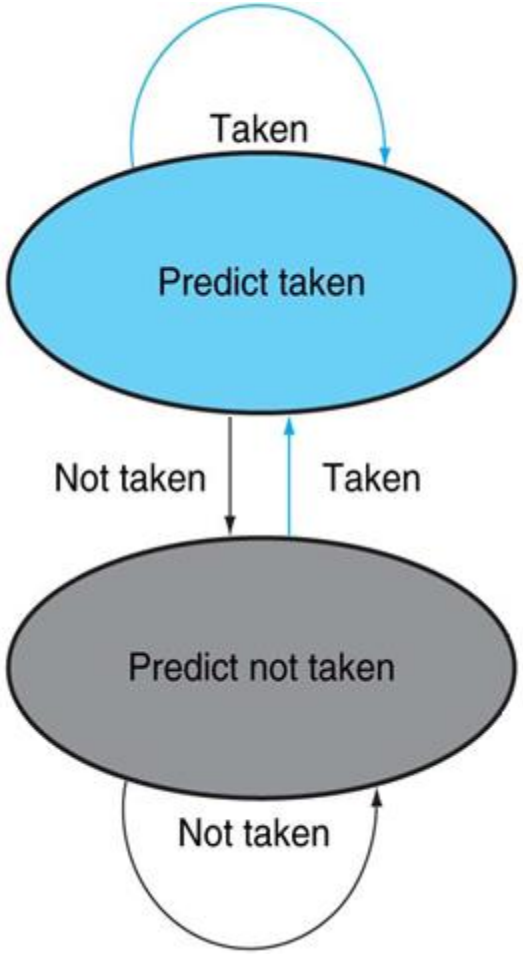
Since in most cases, the prediction will fail

Static Branch Prediction, con't

- As the branch penalty increases (for deeper pipelines), a simple static prediction scheme will hurt performance. With more hardware, it is possible to try to predict branch behavior dynamically during program execution.
  - Dynamic branch prediction – predict branches at runtime using *run-time* information.

Dynamic prediction (1-bit prediction scheme)

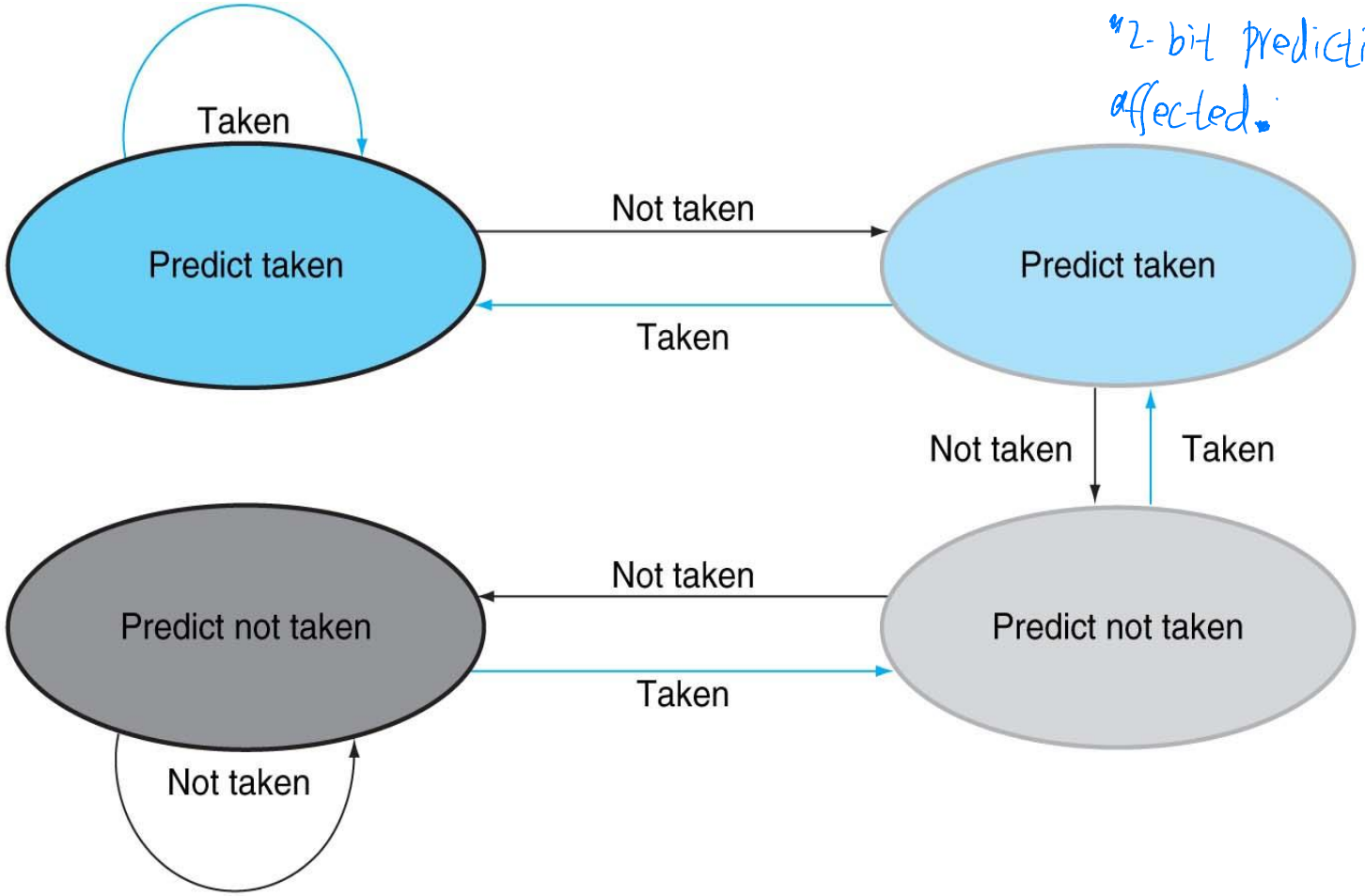
- Predict branch behavior during program execution.



Dynamic prediction (2-bit prediction scheme)

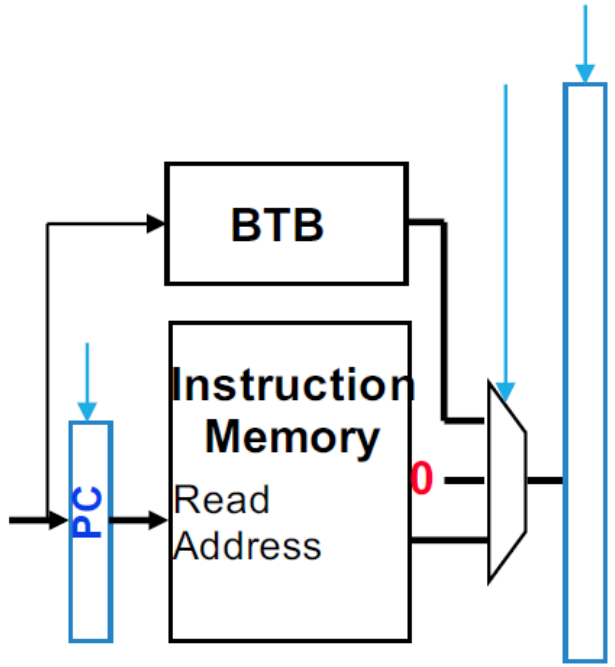
fits well for "for" loop, since in some cases there are some "if-else" clauses in the loop, "2-bit prediction cannot be readily affected.

- Predict branch behavior during program execution.



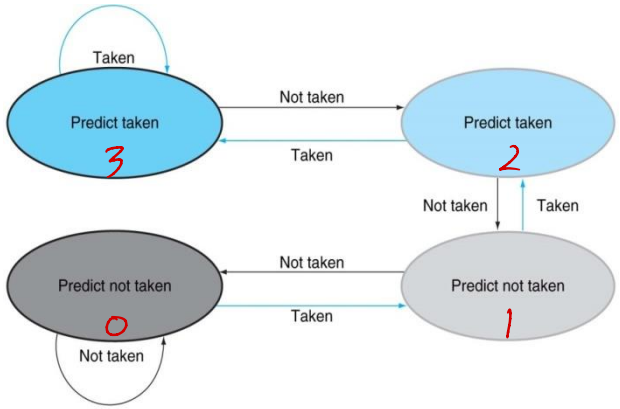
Dynamic prediction

- A branch prediction buffer (aka branch history table (BHT)) in the IF stage addressed by the lower bits of the PC, contains bit(s) passed to the ID stage through the IF/ID pipeline register that tells whether the branch was taken the last time it was execute.
- The BHT predicts *when* a branch is taken, but does not tell *where* its taken to!
- If the prediction is correct, stalls can be avoided no matter which direction they go.



Example of dynamic branch prediction

- Assume that we start off in state 3. A program contains 7 beq instructions.
- Prediction accuracy: 4/7=57%



Instruction	Actual execution	Current FSM state	Prediction	Correct?	Next FSM state
beq …	Taken	3	Taken	Yes	3
beq …	Not taken	3	Taken	No	2
beq …	Taken	2	Taken	Yes	3
beq …	Taken	3	Taken	Yes	3
beq …	Not Taken	3	Taken	No	2
beq …	Not Taken	2	Taken	No	1
beq …	Not Taken	1	Not taken	Yes	0

1-bit Prediction Accuracy

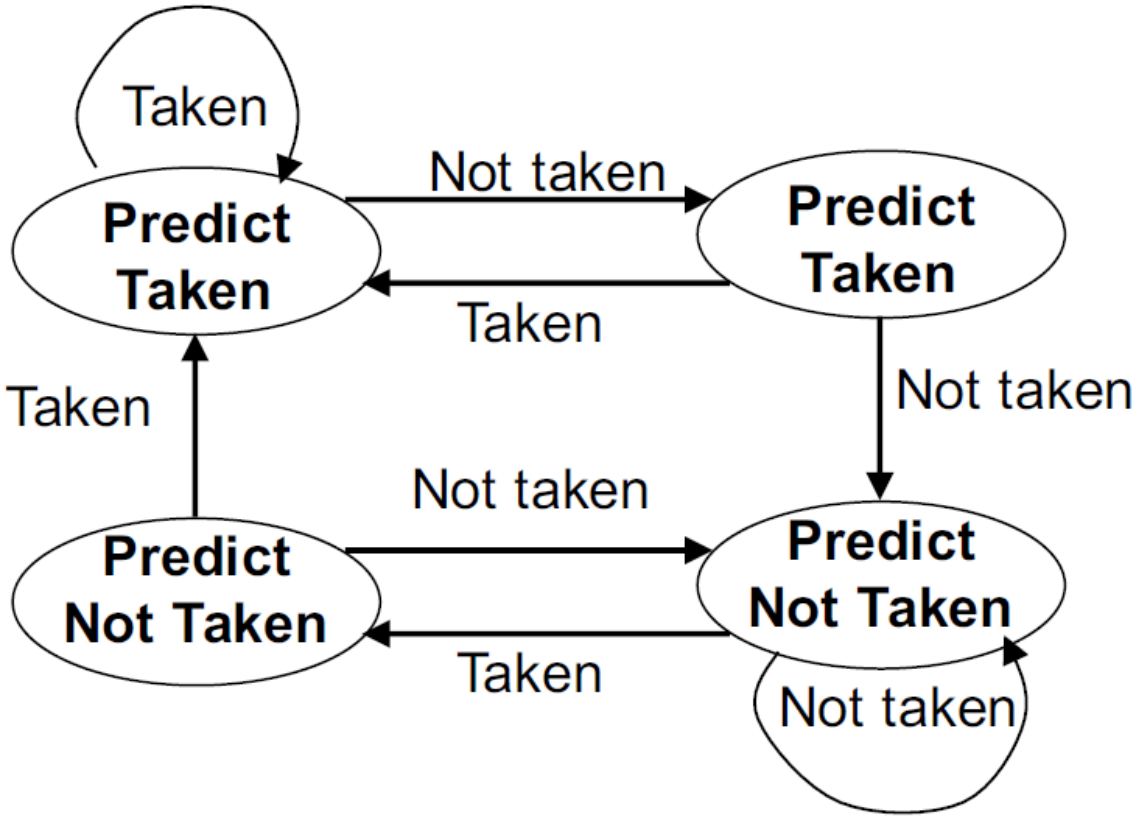
- A 1-bit predictor will be incorrect twice when not taken
  - Assume predict_bit = 0 to start (indicating branch not taken) and loop control is at the bottom of the loop code
  1. First time through the loop, the predictor mispredicts the branch since the branch is taken back to the top of the loop; invert prediction bit (predict_bit = 1)
  2. As long as branch is taken (looping), prediction is correct
  3. Exiting the loop, the predictor again mispredicts the branch since this time the branch is not taken falling out of the loop; invert prediction bit (predict_bit = 0)

```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- For 10 times through the loop we have a 80% prediction accuracy for a branch that is taken 90% of the time

2-bit Prediction Accuracy

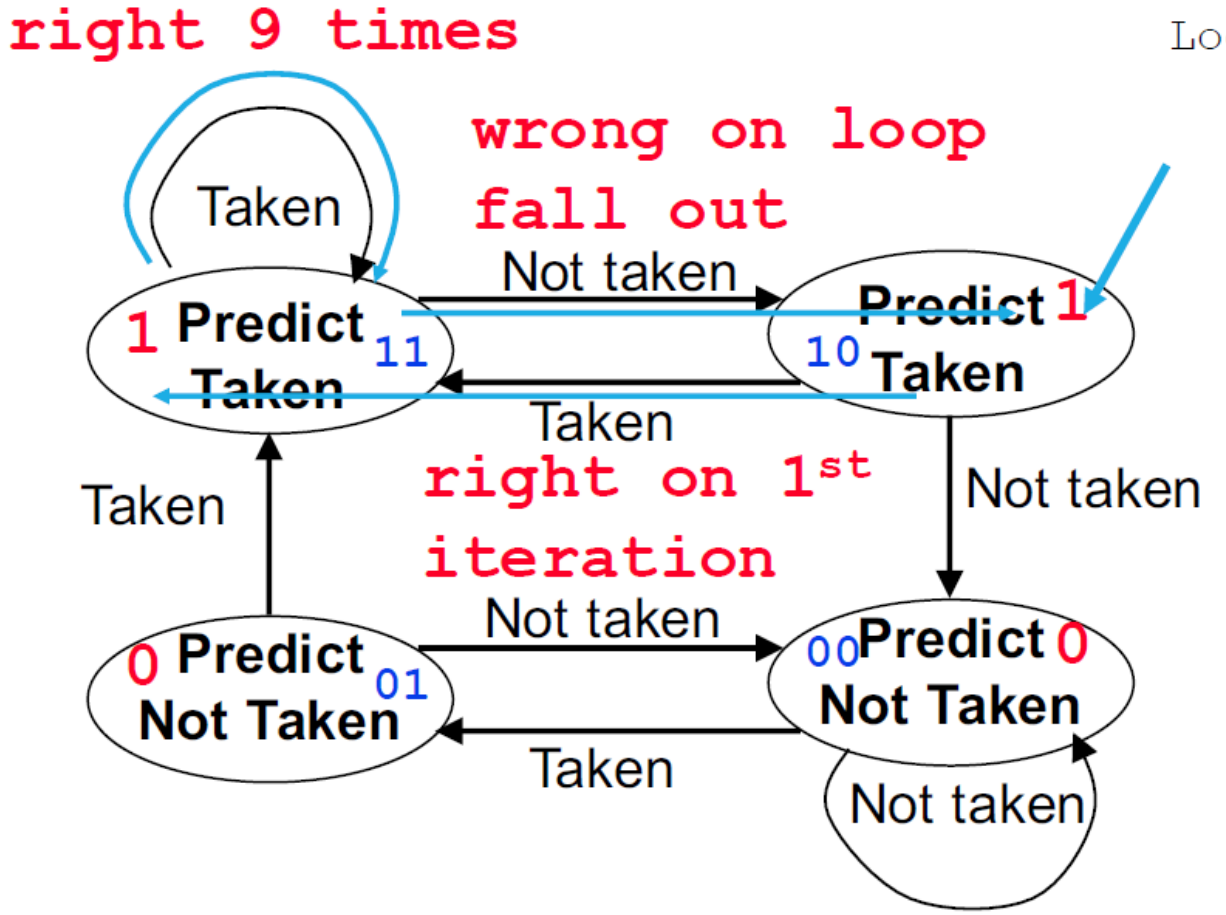
- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

2-bit Prediction Accuracy

- A 2-bit scheme can give 90% accuracy since a prediction must be wrong twice before the prediction bit is changed.




```
Loop: 1st loop instr
      2nd loop instr
            .
            .
            .
      last loop instr
      bne $1,$2,Loop
      fall out instr
```

- BHT also stores the initial FSM state

Summary

- Control hazard: attempts to make a decision before branch condition is evaluated.
  - E.g. branch instructions.
  - Solution: insert NOP at compile time; branch prediction with flush.
  - Modify the datapath to detect the branch one cycle earlier.