Appendix C

Basic pipeline concepts and implementation

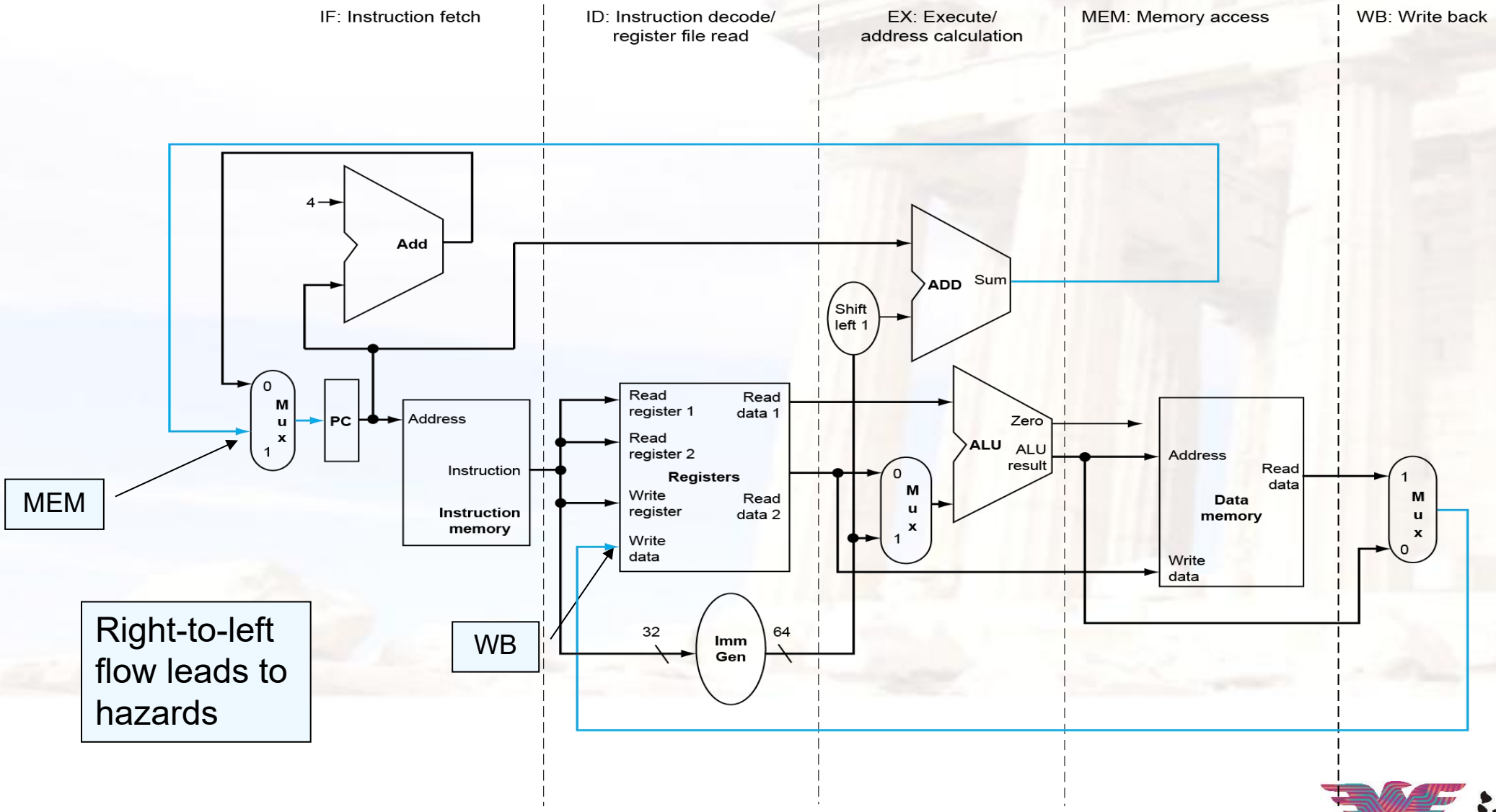
Pipelining RISC-V instruction set

- Since there are five separate stages, we can have a pipeline in which one instruction is in each stage.
- CPI is decreased to 1, since one instruction will be issued (or finished) each cycle.
- During any cycle, one instruction is present in each stage.

	Clock Number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

- Ideally, performance is increased five fold !

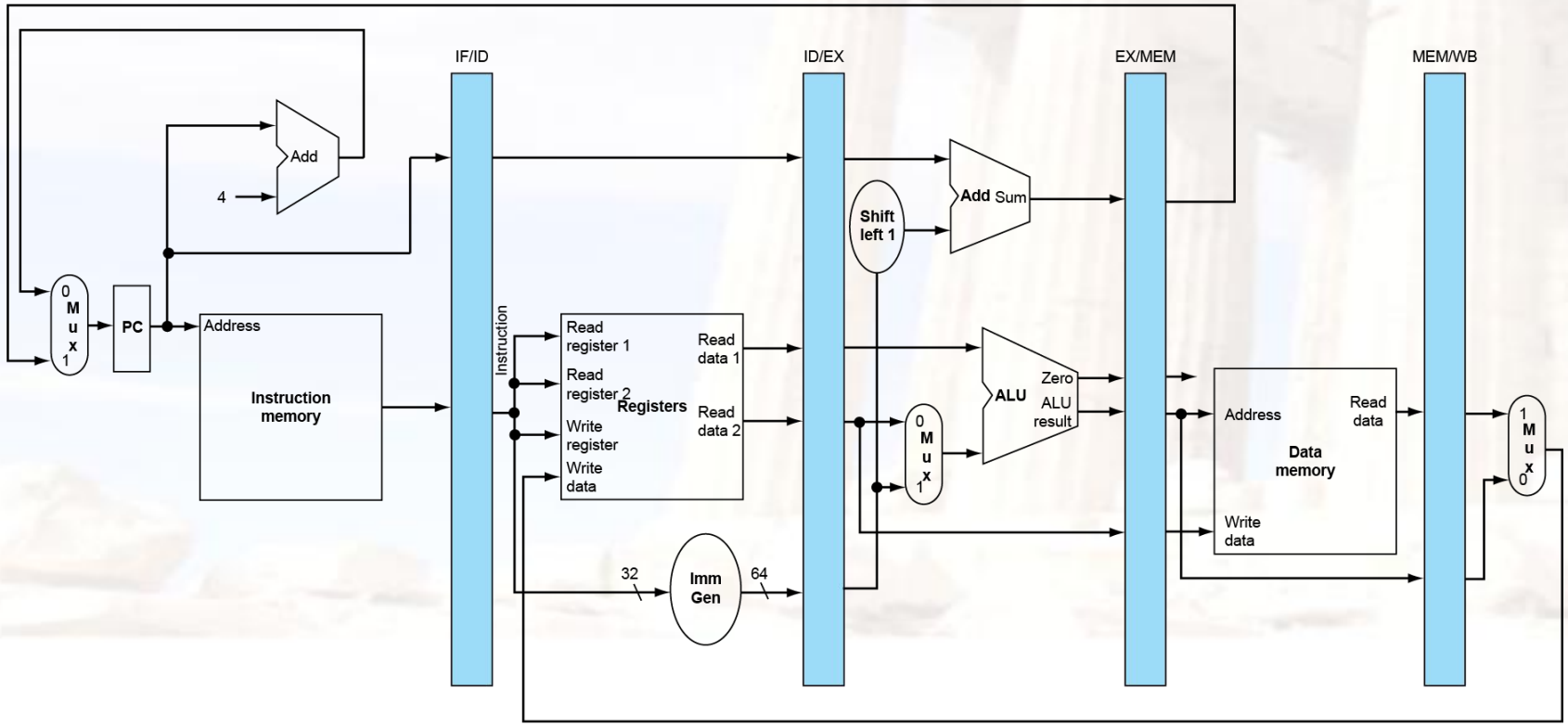
Pipelined Datapath and Control



Right-to-left flow leads to hazards

Five stage pipeline

- Need registers (or latch) between stages
  - To hold information produced in previous cycle



浙江大学 ZHEJIANG UNIVERSITY

Pipeline Hazards

- **Taxonomy of Hazards**
  - **Structural hazards**
    - These are conflicts over hardware resources.
  - **Data hazards**
    - Instruction depends on result of prior computation which is not ready (computed or stored) yet
  - **Control hazards**
    - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

How to solve the structural hazard

- Multiple accesses to memory
  - Split instruction and data memory / multiple memory port / instruction buffer
  - Memory bandwidth need to be imporved by 5 folds.
- Multiple accesses to the register file
  - Double bump
- Not fully pipelined functional units
  - Fully pipeline the functional unit
  - Using multiple functional unites
- Real machine often with structural hazards.

Summary of Structural hazard

Taxonomy of Hazards

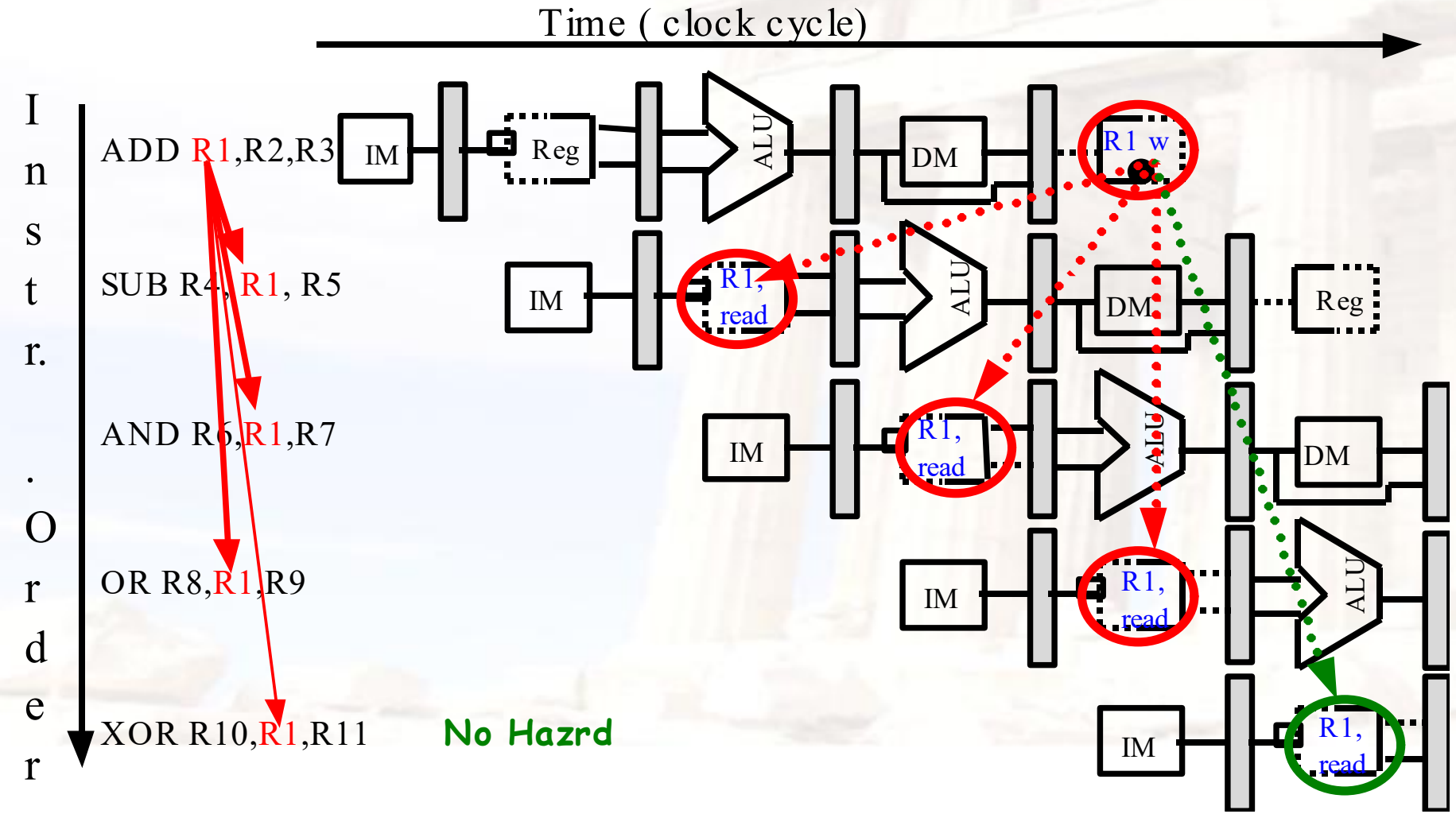
- **Structural hazards**
  - These are conflicts over hardware resources.
  - OK, maybe add extra hardware resources;
    or full pipelined the functional units;
    otherwise still have to stall
  - Allow machine with Structural hazard, since it happens not so often
- **Data hazards**
  - Instruction depends on result of prior computation which is not ready (computed or stored) yet
- **Control hazards**
  - branch condition and the branch PC are not available in time to fetch an instruction on the next clock

Data hazard

- Data hazards occur when the pipeline changes the order of read/write accesses to operands (Violation of data dependence) ycomparing with that in sequential executing, .
- Let's see an Example

```
DADD R1,  R1, R3
DSUB R4,  R1, R5
AND  R6,  R1, R7
OR   R8,  R1, R9
XOR  R10, R1, R11
```

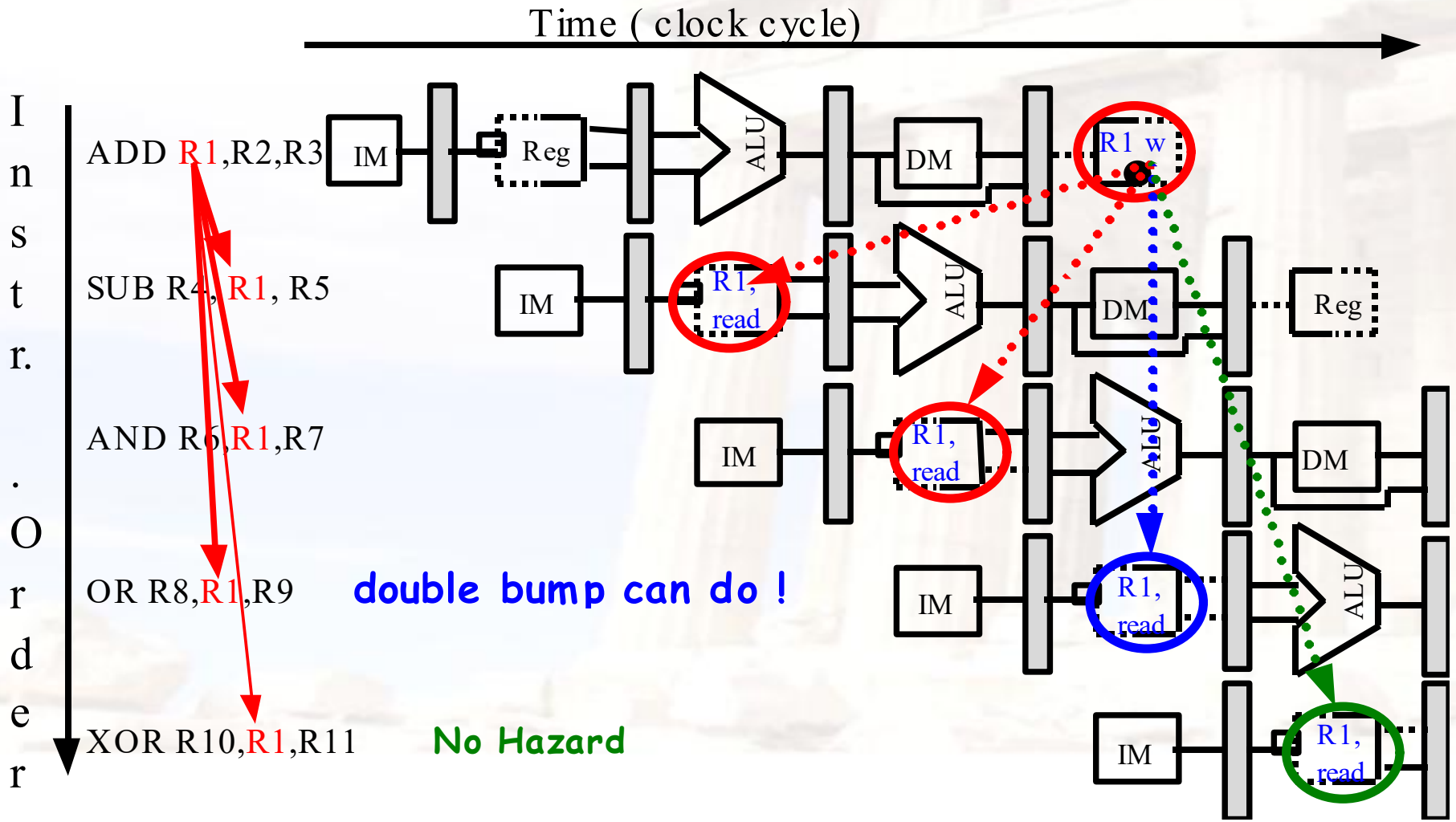
Coping with data hazards:example



Data hazard

- Basic structure
  - An instruction in flight wants to use a data value that's not "done" yet
  - "Done" means "it's been computed" and "it's located where I would normally expect to go look in the pipe hardware to find it"
- Basic cause
  - You are used to assuming a purely sequential model of instruction execution
  - Instruction N finishes before instruction N+k, for k >= 1
  - There are dependencies now between "nearby" instructions ("near" in sequential order of fetch from memory)
- Consequence+
  - Data hazards -- instructions want data values that are not done yet, or not in the right place yet

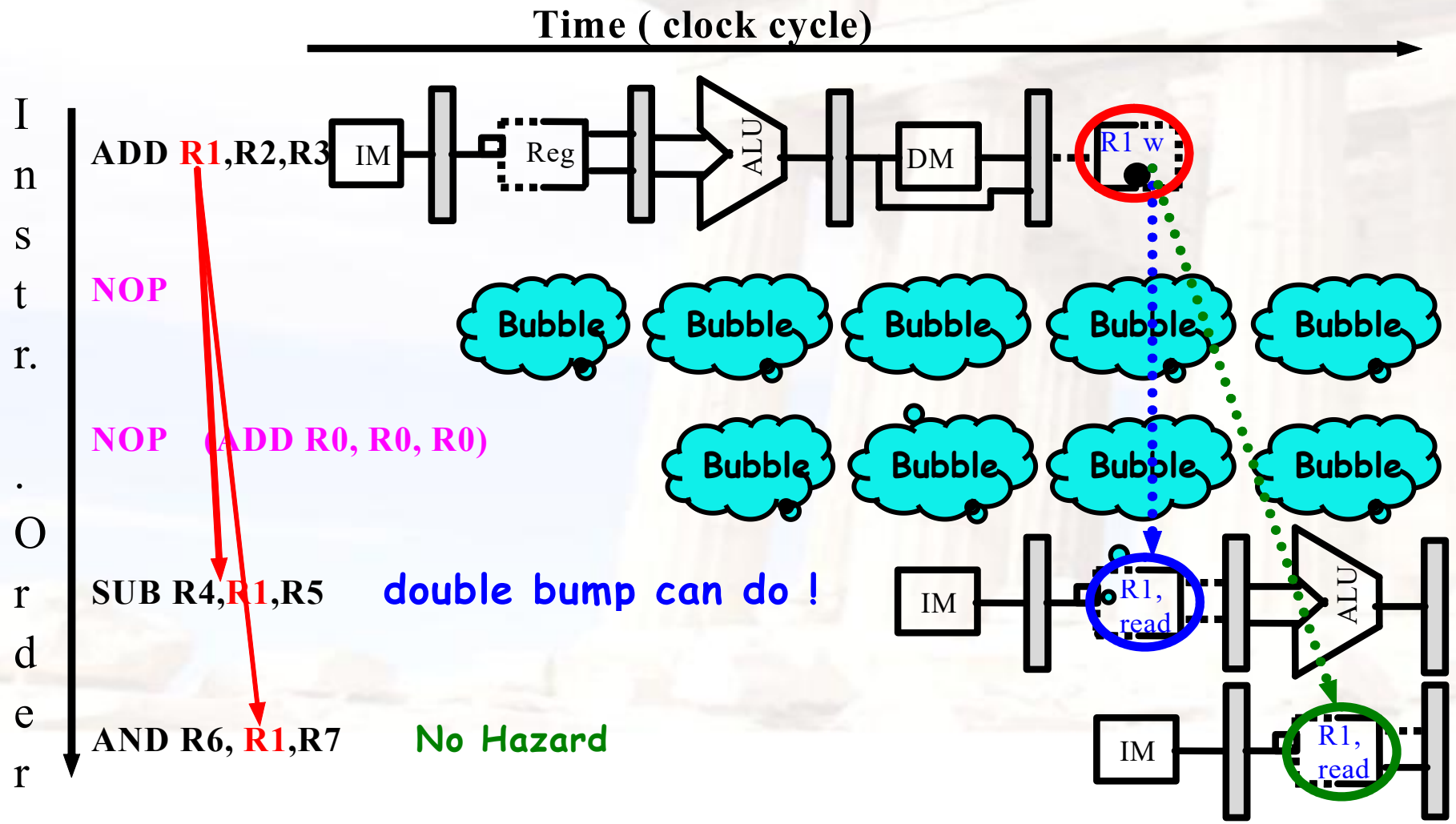
Somecases “Double Bump” can do !



The simplest way to solve data hazard: stall

- **Proposed solution**
  - **Don't** let them **overlap** like this…?
- **Mechanics**
  - Don't let the instruction flow through the pipe
  - In particular, don't let it **WRITE** any bits anywhere in the pipe hardware that represents **REAL** CPU state (e.g., register file, memory)
  - Let the instruction **wait** until the hazard resolved.
  - Name for this operation: **PIPELINE STALL**

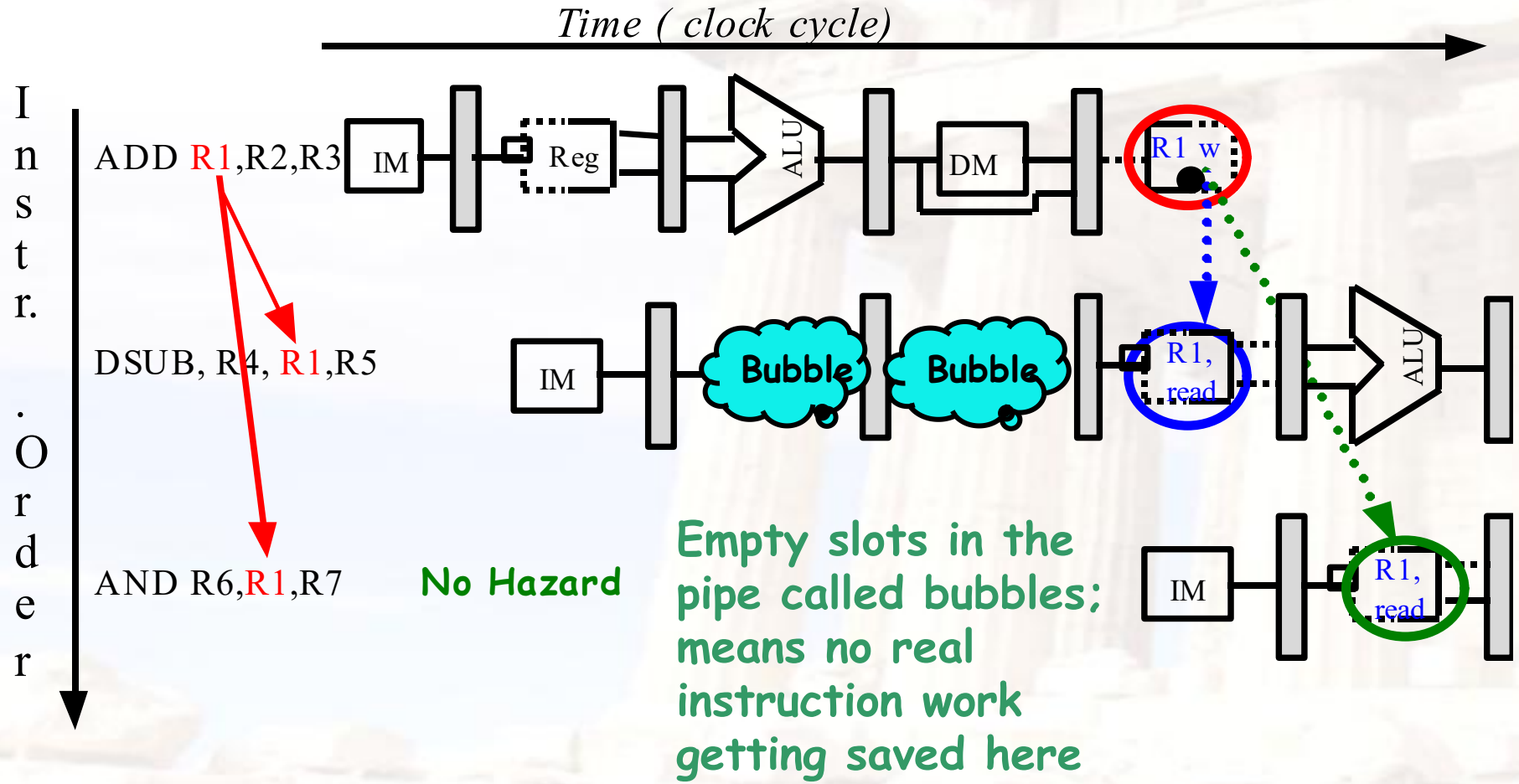
How do we stall ? Insert nop by compiler



How do we stall? Add hardware Interlock !

- Add extra hardware to detect stall situations
  - Watches the instruction field bits
  - Looks for "read versus write" conflicts in particular pipe stages
  - Basically, a bunch of careful "case logic"
- Add extra hardware to push bubbles thru pipe
  - Actually, relatively easy
  - Can just let the instruction you want to stall GO FORWARD through the pipe…
  - …but, TURN OFF the bits that allow any results to get written into the machine state
  - So, the instruction "executes" (it does the work), but doesn't "save"

Interlock: insert stalls



How the interlock is implementated ?

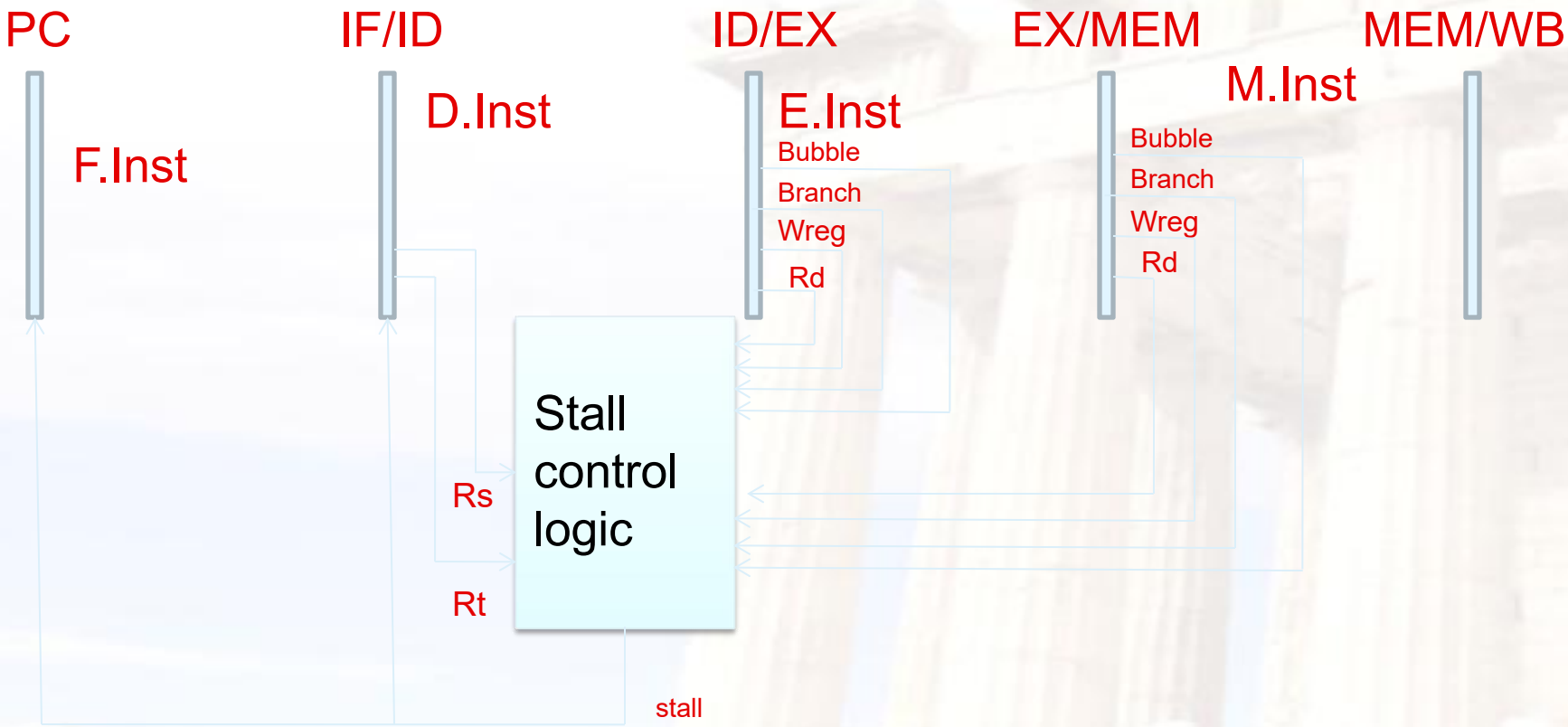
How the Interlock functions?

- The Interlock can simulate the NOP:

  Once it is detected need to add a stall, then

  - Clear the ID/EX.IR to be the instruction of NOP.
    - disable the write signal: "wreg, wmem".
  - Keep IF/ID. IR unchanged for one more clock cycle.
    - Disable the write signal: "WritePC, WriteIR".

How to Stall ? Add a stall control logic !



1) To produce a stall signal. If stall ==1 then
2) Use stall to disable writing PC write and IF/ID latch.
3) Push a bubble into next stage.
   - ✓ Bubble = 1 and disable control signal Wreg and Wmem.
   - ✓ Puch a nop forward into ID/EX

How to stall (when data hazard)?

- If stall == 0 //stop the latter ones
  - then PC ← new PC; IF/ID.IR ← IF.IR
  - IF/ID.NPC ← PC + 4
- If stall == 1 // push bubble forward
  - then ID/EX.nop ←1 else ID/EX.nop ←0
- When initial

  ID/EX.nop ←0, EX/MEM.nop ← 0, Mem.WB.nop ←0

Are the stalls the same for Data hazard / control hazard ?

Solve the data hazard by Forwarding

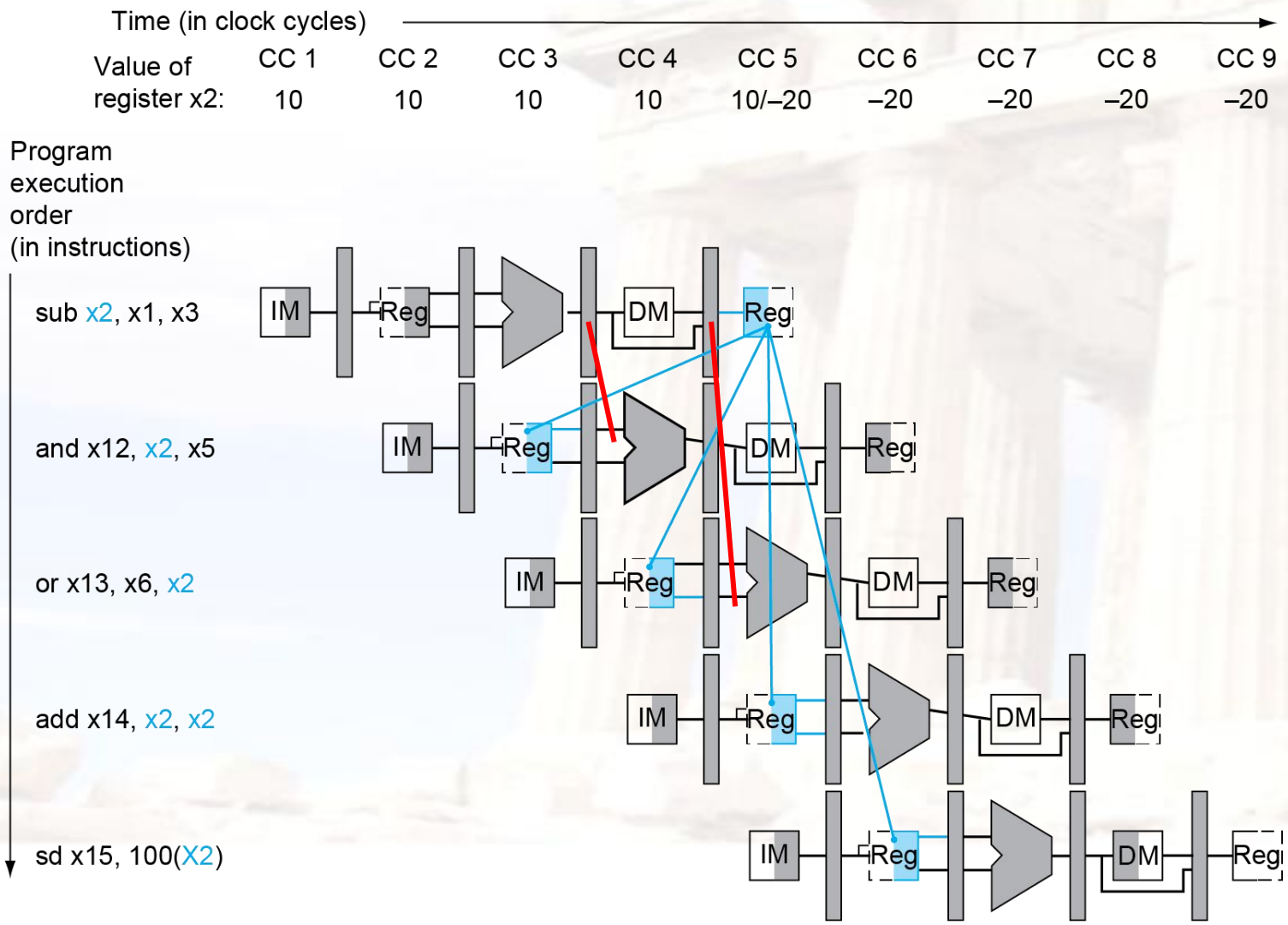
Data Dependence

I2 RAW on I1

I3 RAW on I1

I4 RAW on I1

I5 RAW on I1



Data Hazard

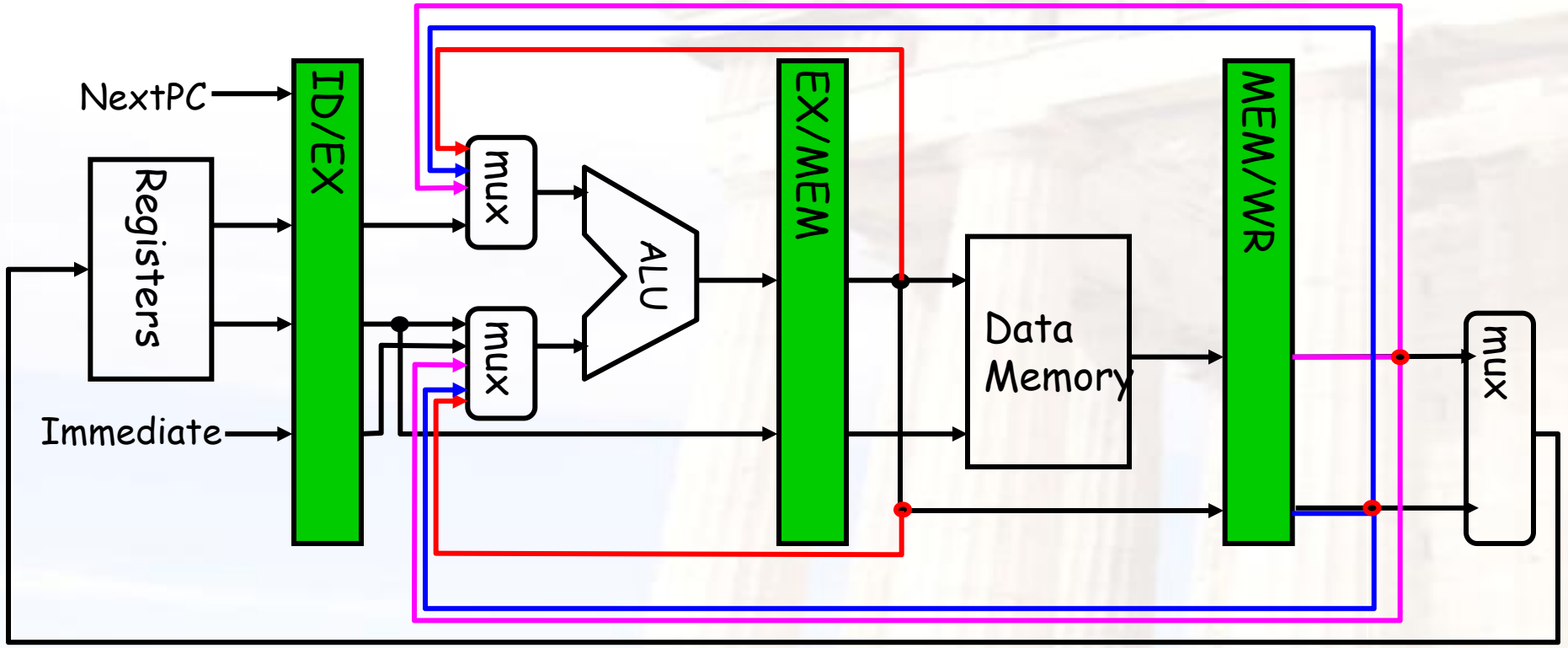
I2 RAW hazard with I1

I3 RAW hazard with I1

RAW hazard，but no stall using double bump

No RAW hazard

Hardware Change for Forwarding



Source → sink

EX/Mem.ALUoutput → ALU input

MEM/WB.ALUoutput → ALU input

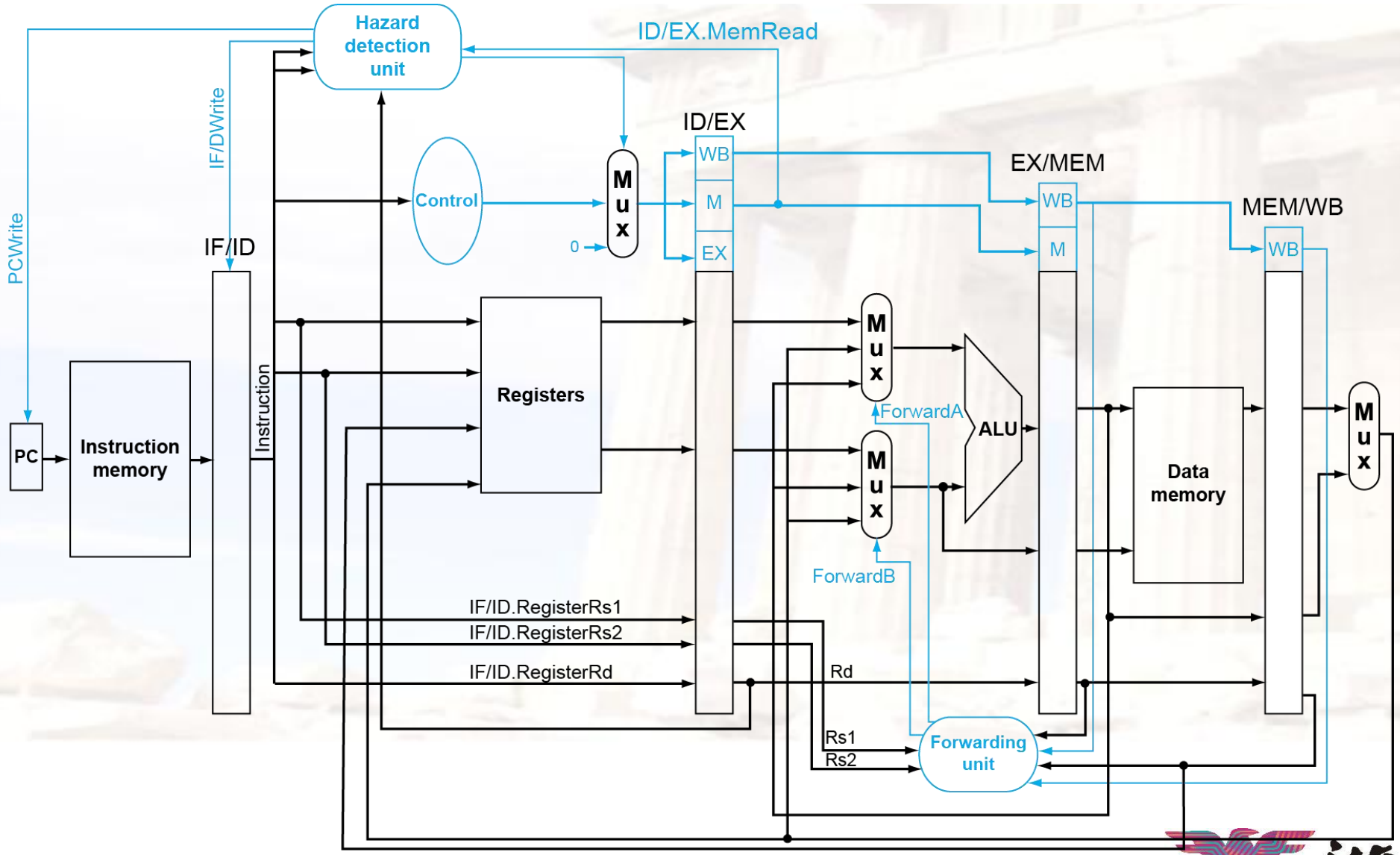
MEM/WB.LMD → ALU input

When to use the red. Blue purple forwording path ?

When to use the forwarding path ?

- **EX/Mem.ALUoutput → ALU** input
  - The previous instruction in EX/MEM is ALU
  - The instruction in ID/EX has Rs1 or Rs2 source register
  - EX/MEM.Rd == ID/EX.Rs1 or EX/MEM.Rd ==ID/EX.Rs2
- **MEM/WB.ALUoutput → ALU input**
  - The previous instruction in MEM/WB is ALU
  - The instruction in ID/EX has Rs1 or Rs2 source register
  - MEM/WB.Rd == ID/EX.Rs1 or MEM/WB.Rd == IF/ID.Rs2
- **MEM/WB.LMD → ALU input**
  - The previous instruction in MEM/WB is Load
  - The instruction in ID/EX has Rs1 or Rs2 source register
  - MEM/WB.Rd == ID/EX.Rs1 or MEM/WB.Rd == ID/EX.Rs2

Forwarding at EX stage



Fowarding can only solve some problems

```
ADD  x6,  x28, x29
SUB  x30,  x6,  x14
```

```
ADD  x6,  x28, x29
AND  x14,  x5, x7
SUB  x30,  x6,  x14
```

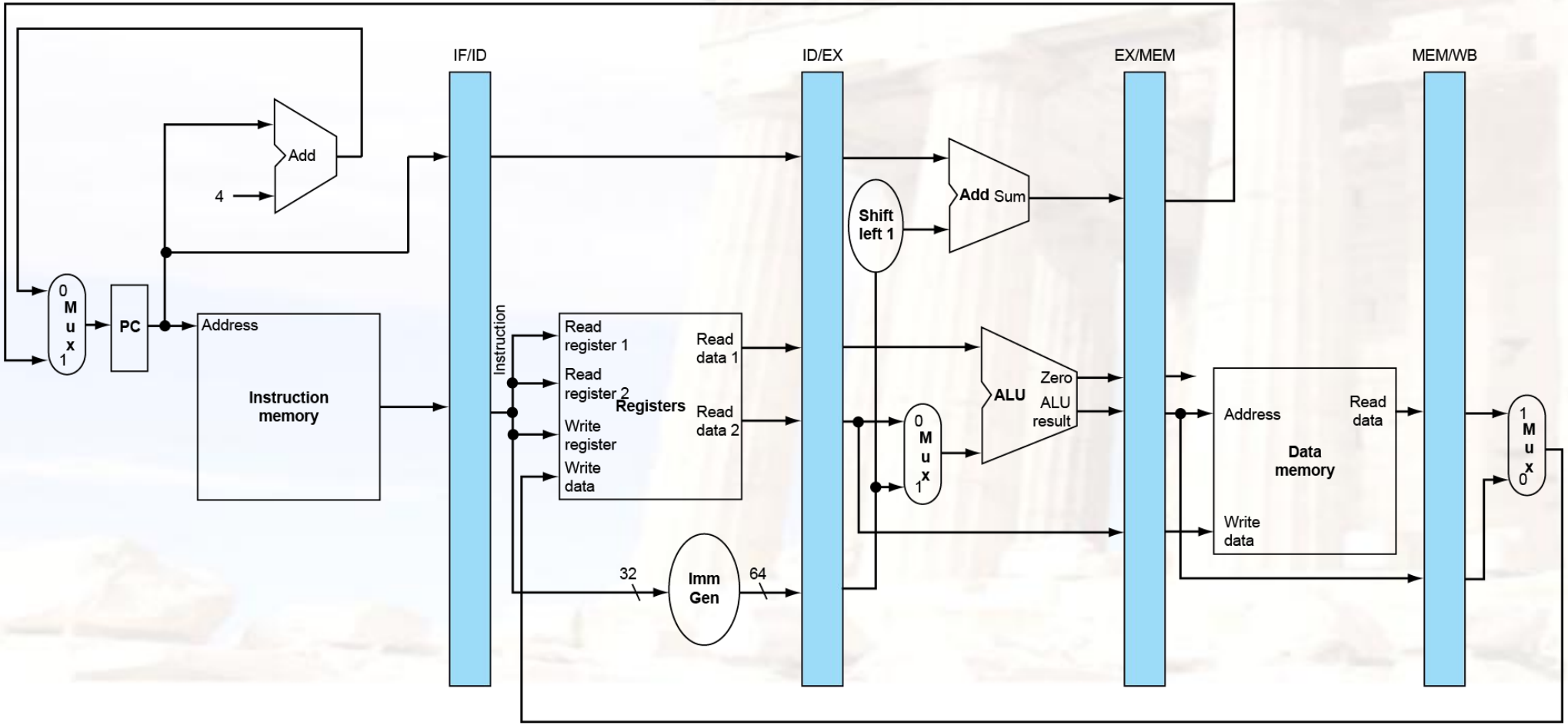
/ LW x6, 4(x10)

Control hazard Example: Branches

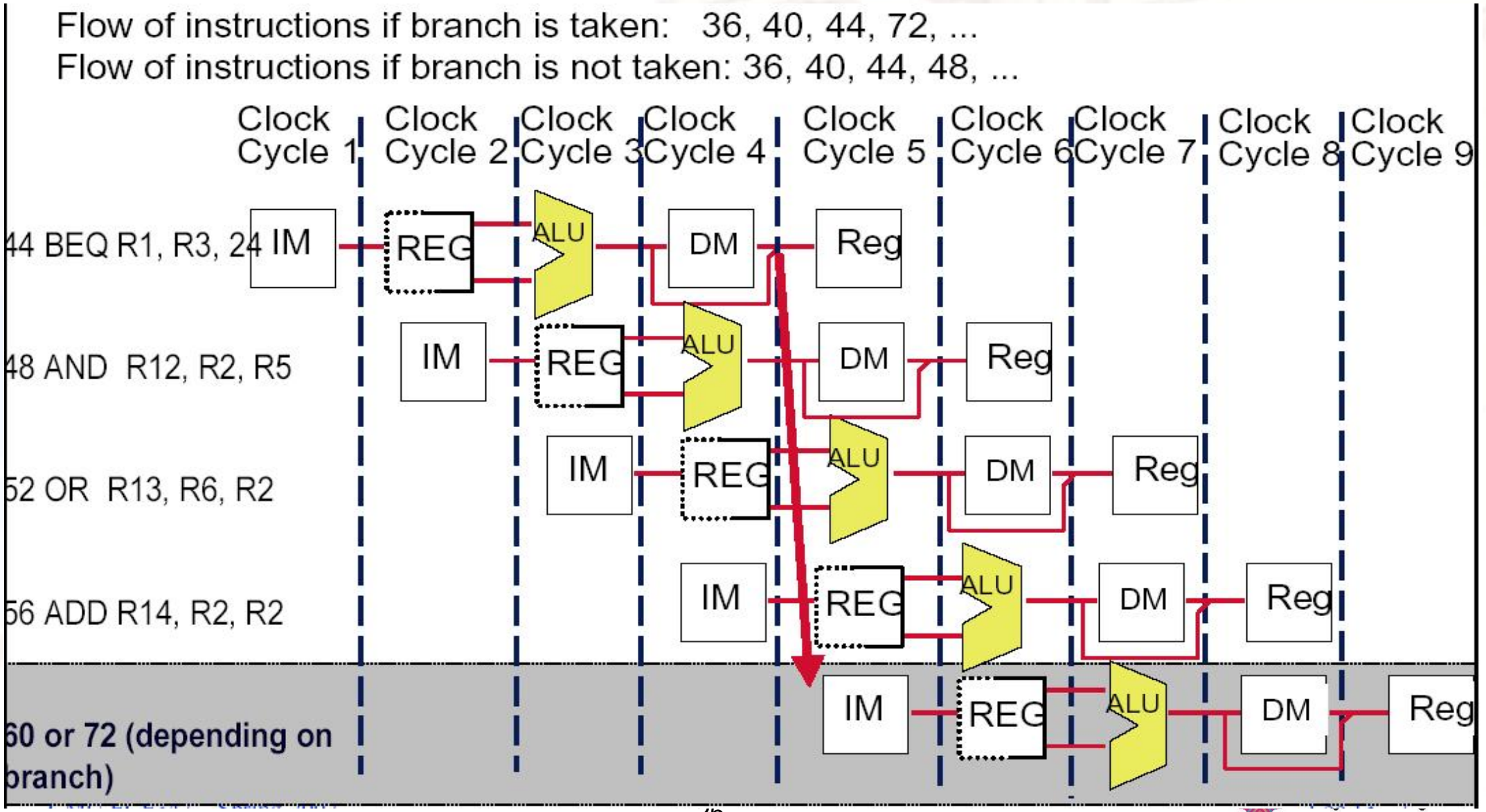
Address	Instruction	
36	NOP	
40	ADD R30,R30,R30	
44	BEQ R1, R3, 24	<- this branchs to address 72
48	AND R12, R2, R5	We execute all these if R1 != R3
52	OR R13, R6, R2	
56	ADD R14, R2, R2	
60	...	
64	...	
68		
72	LW R4, 50(R7)	We execute just these if R1 == R3
76	...	

Flow of instructions if branch is taken: 36, 40, 44, 72, ...
Flow of instructions if branch is not taken: 36, 40, 44, 48, ...

Recall: Basic Pipelined Datapath



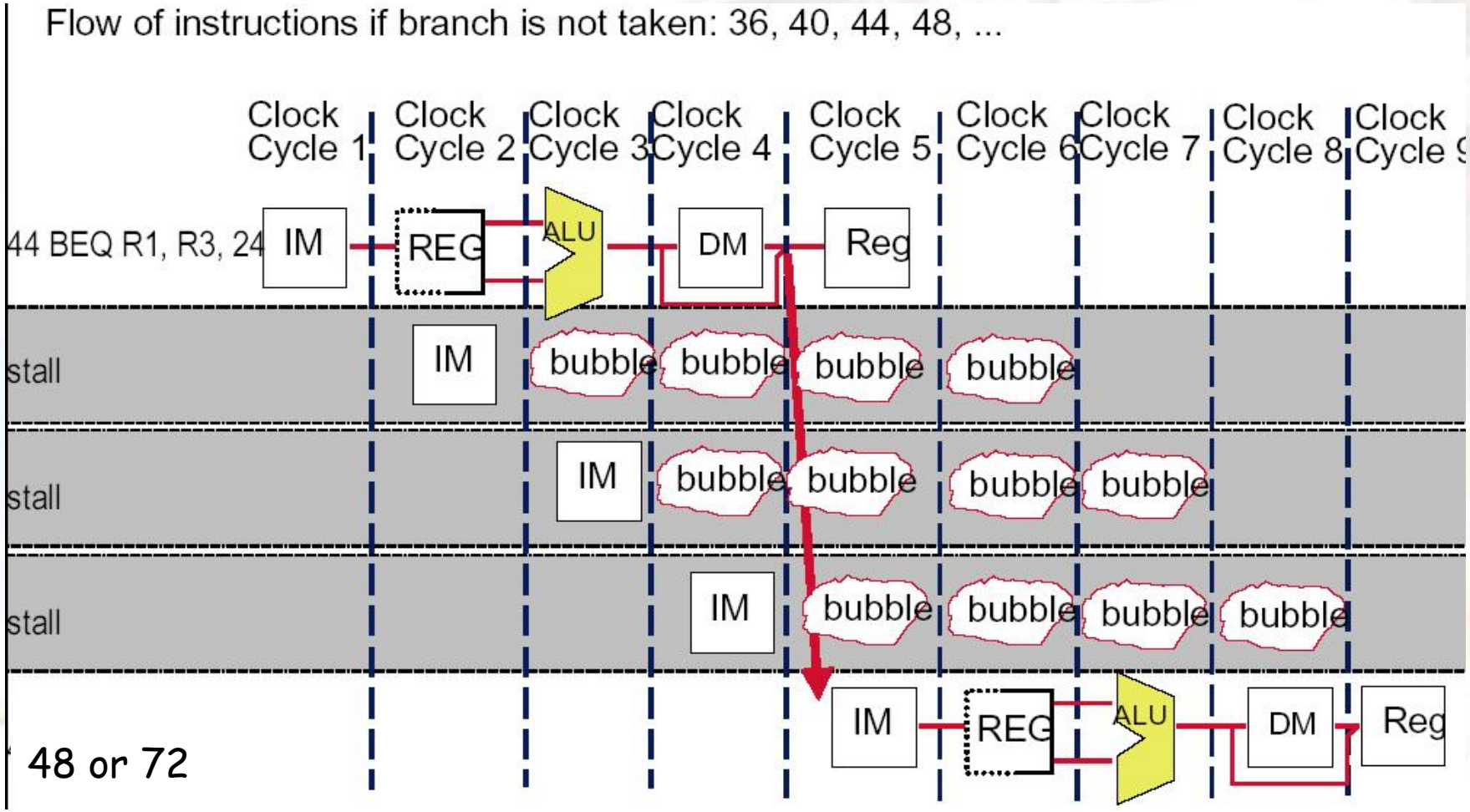
Control hazard



Dealing with the control hazard

- Four simple solutions
  - Stall
  - Prediction
    - Predict-untaken: treat every branch as not taken
    - Predict-taken: treat every branch as taken
    - Delayed branch (omit)
- Note:
  - Fixed hardware
  - Compile time scheme using knowledge of hardware scheme and of branch behavior

Recall: solve the hazard by inserting stalls



2

4

Flushing the pipeline

- Simplest hardware:
  - Holding or deleting any instruction after branch until the branch destination is know.
  - Penalty is fixed.
  - Can not be reduced by software.

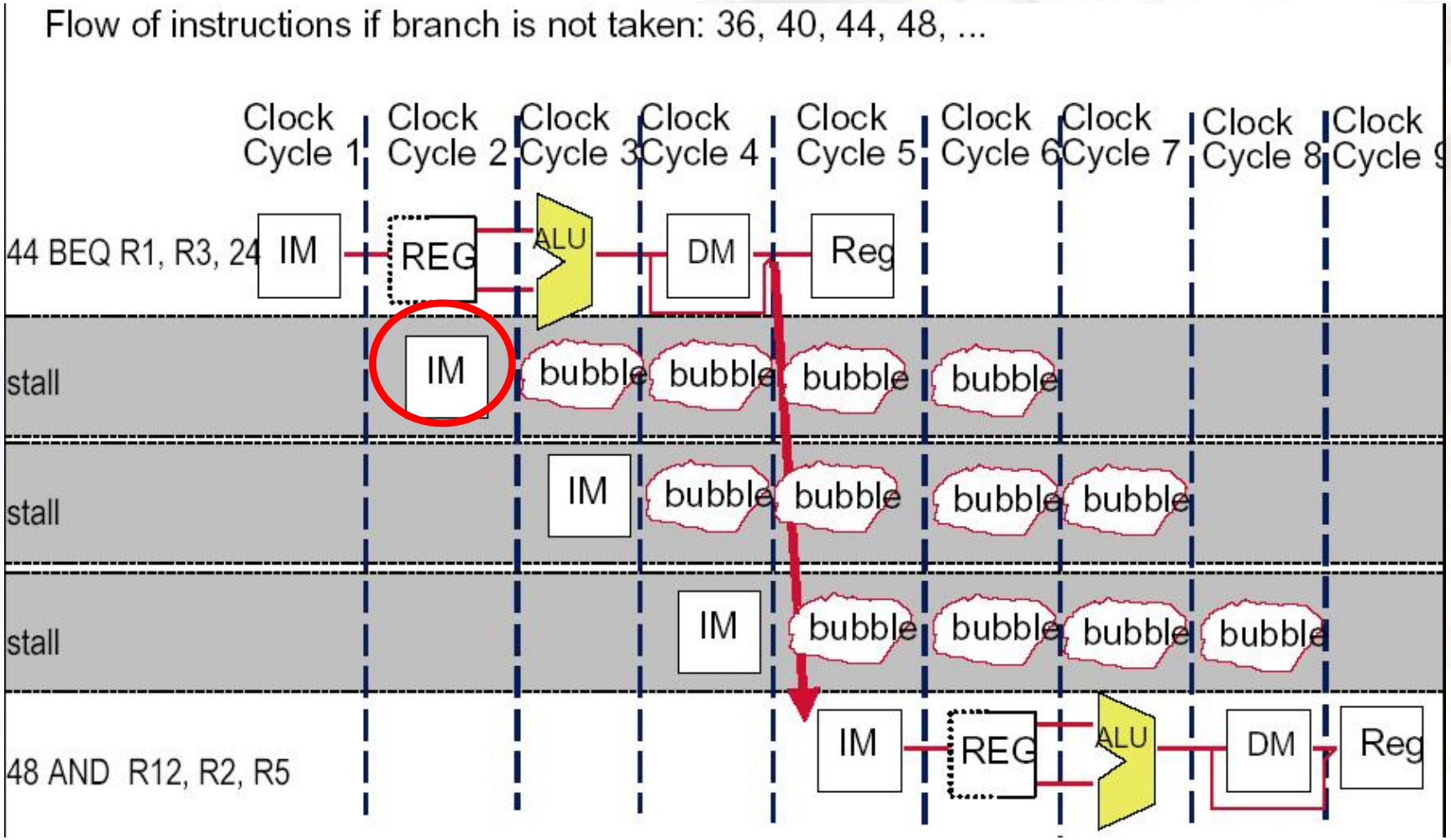
- Implementation. Note difference with data hazard stall
  - Stop the later instruction: keep PC unchanged
  - Nop → IF/ID.IR clear the wrong instruction fetched
  - Nop → ID/EX.IR push a bubble forward

Stalls greatly hurt the performance

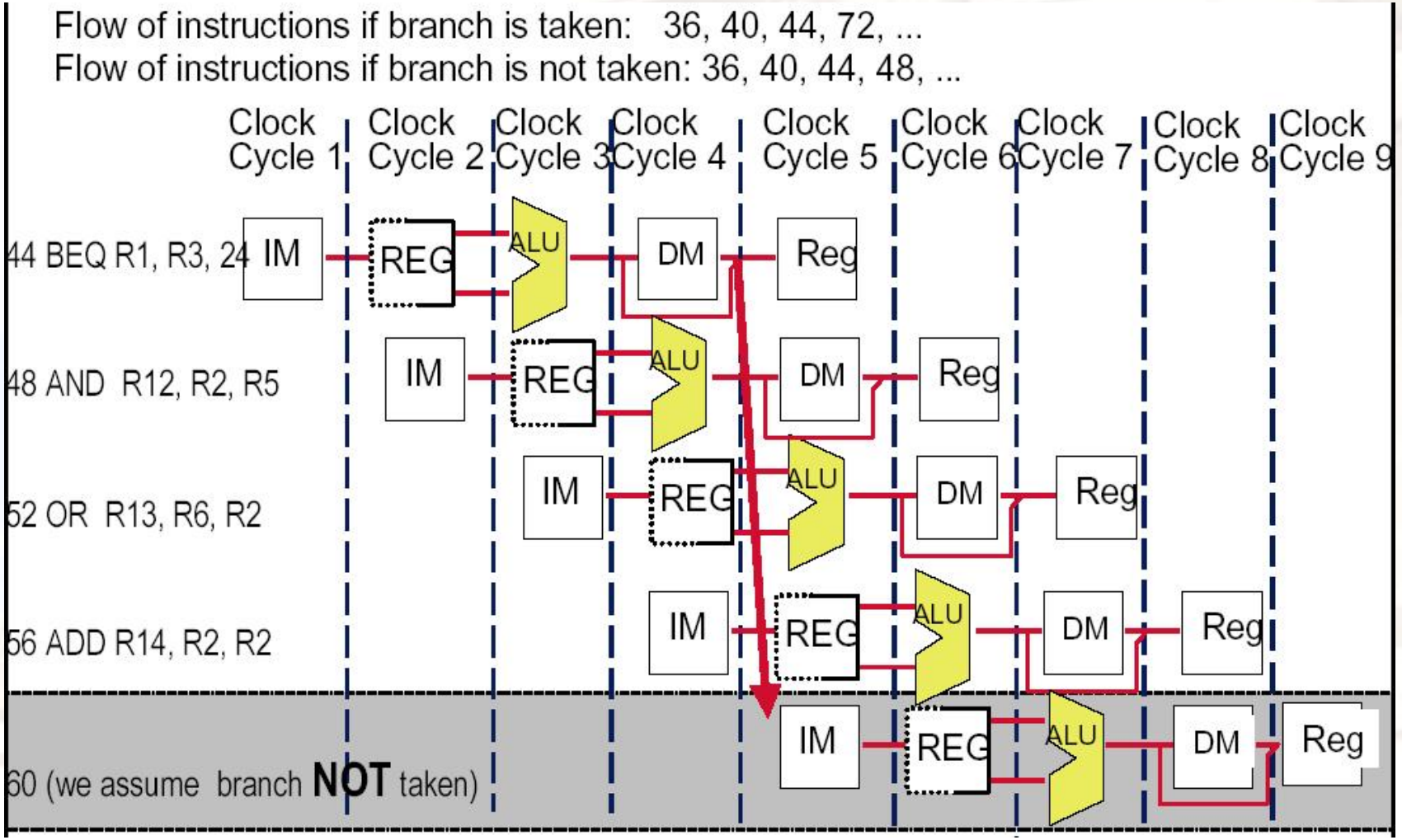
- Problem:
  - With a 30% branch frequency and an ideal CPI of 1, how much the performace is by inserting stalls ?

- Answer:
  - CPI = 1+30%$\times 3 = 1.9$
  - this simple solution achieves only about **half** of the ideal performance.

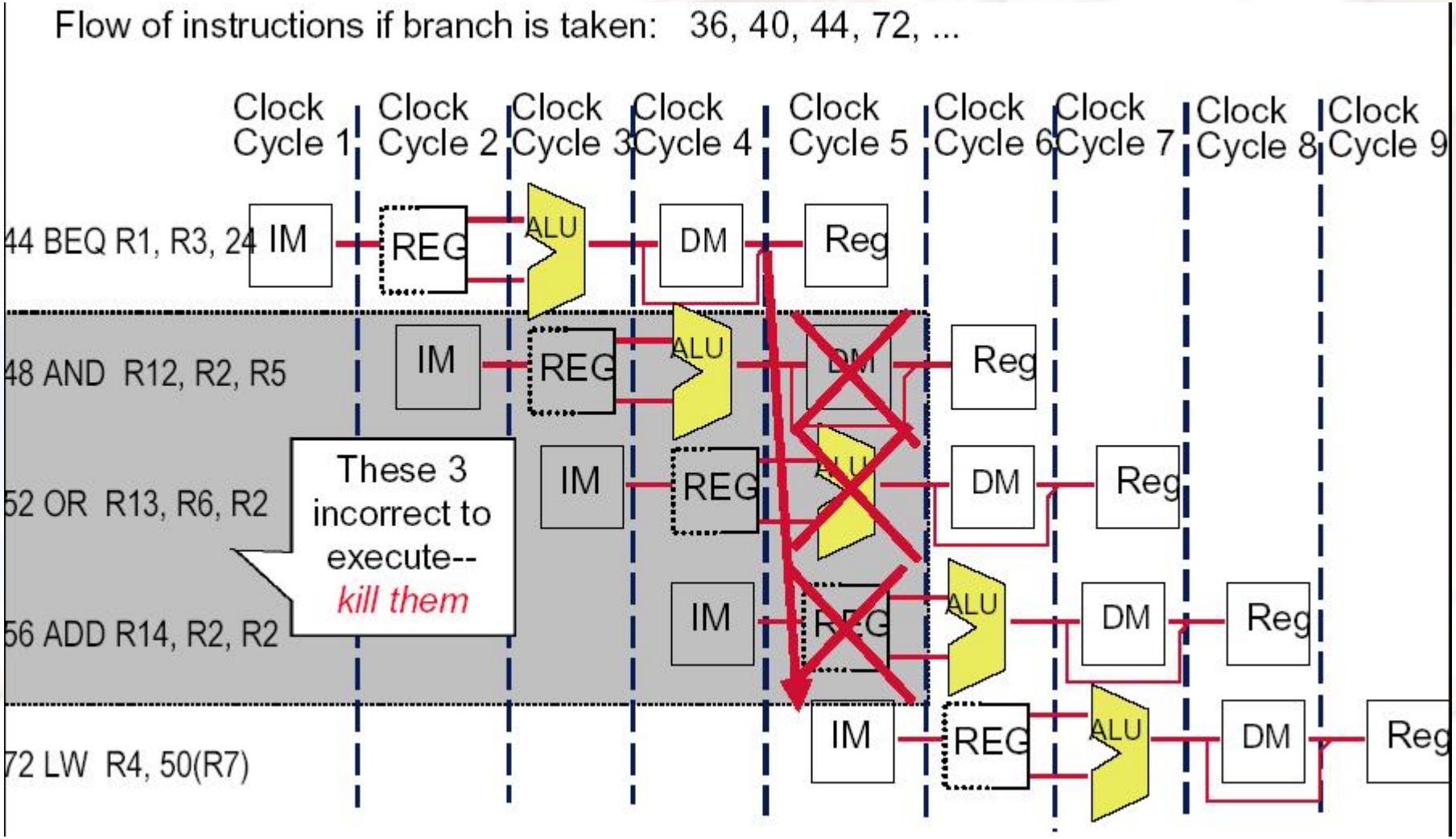
Always Stall Hurts the Not- taken case



How about assume Branch Not Taken



How to do with the branch taken ?



Predict –not-taken

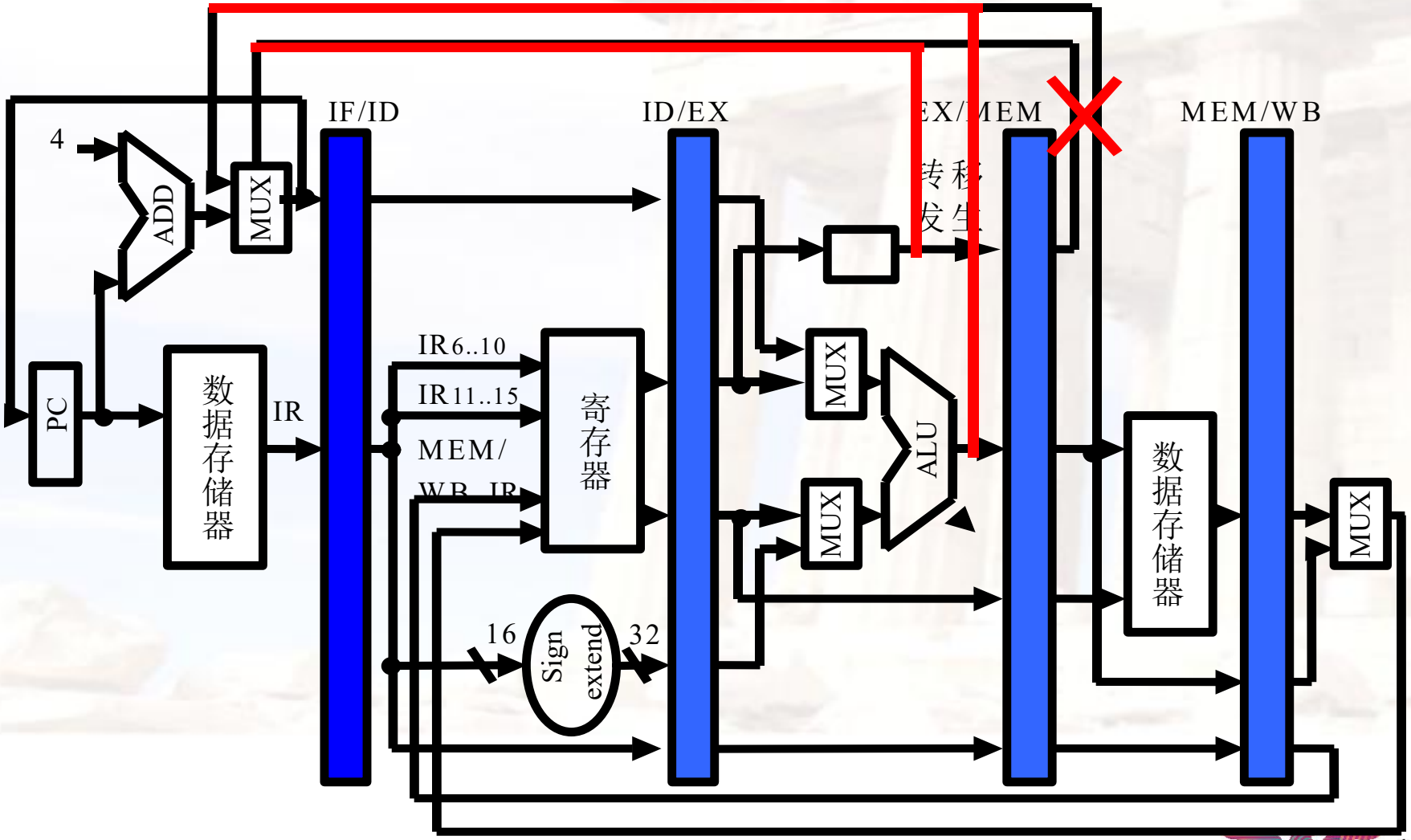
- Hardware:
  - Treat every branch as not taken (or as the formal instruction)
    - When branch is not taken, the fetched instruction just continues to flow on. No stall at all.
    - If the branch is taken, then restart the fetch at the branch target, which cause 3 stall.(should turn the fetched instruction into a no-op)
    - Perf = 1+ br% (not taken% * 0 + take%* 3)
- Compiler:
  - Can improve the performance by coding the most frequent case in the untaken path.

Implementation of predict-not-taken

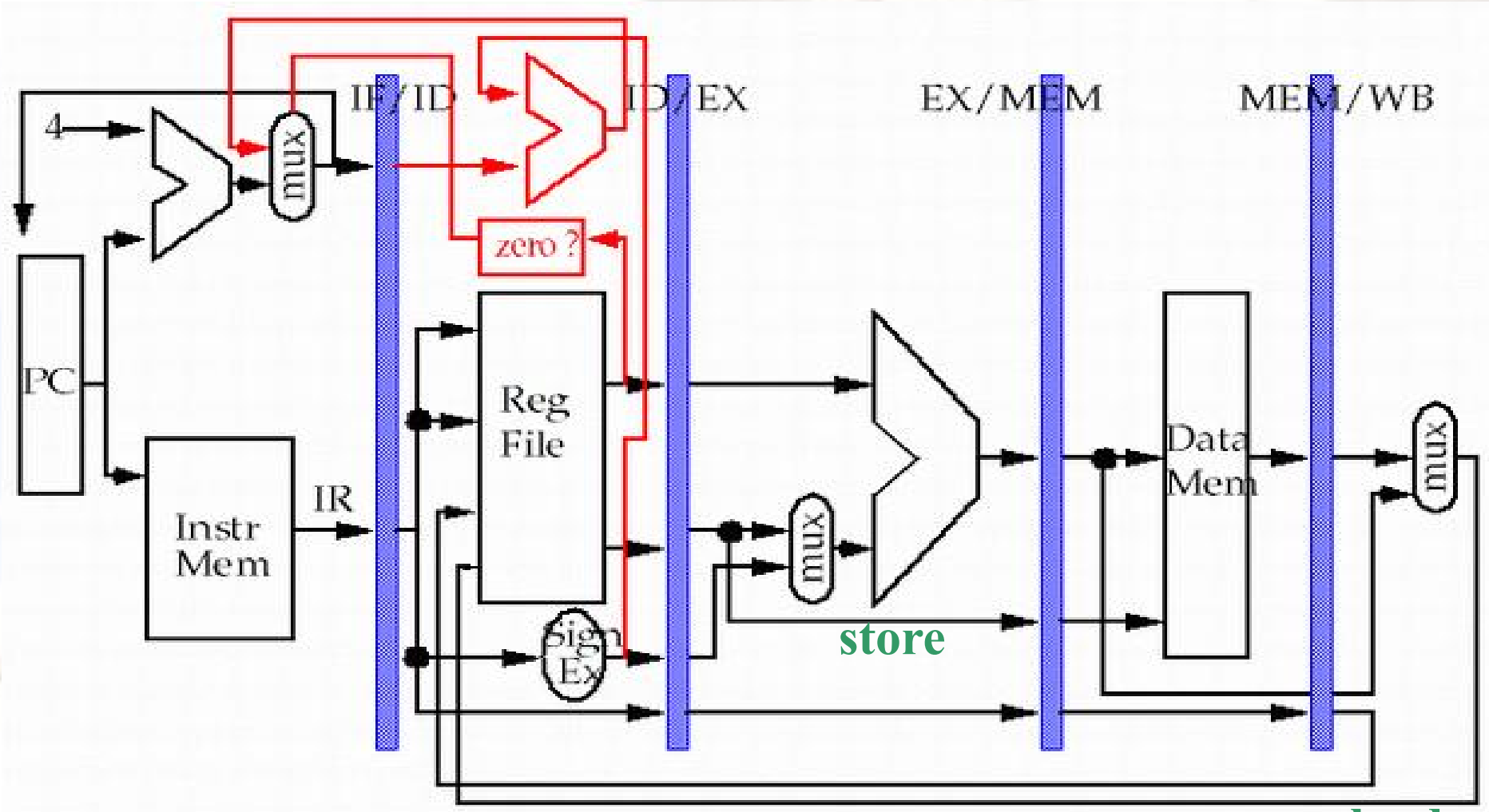
- When the branch is not taken, run as the branch is an ALU.

- When the branch is taken， then
  - Change the PC to the branch target (turn to the right direction)
  - Kill the wrong instructions have been predicted to execute.
    - Nop→IF/ID
    - Nop→ID/EX
    - Nop→EX/MEM

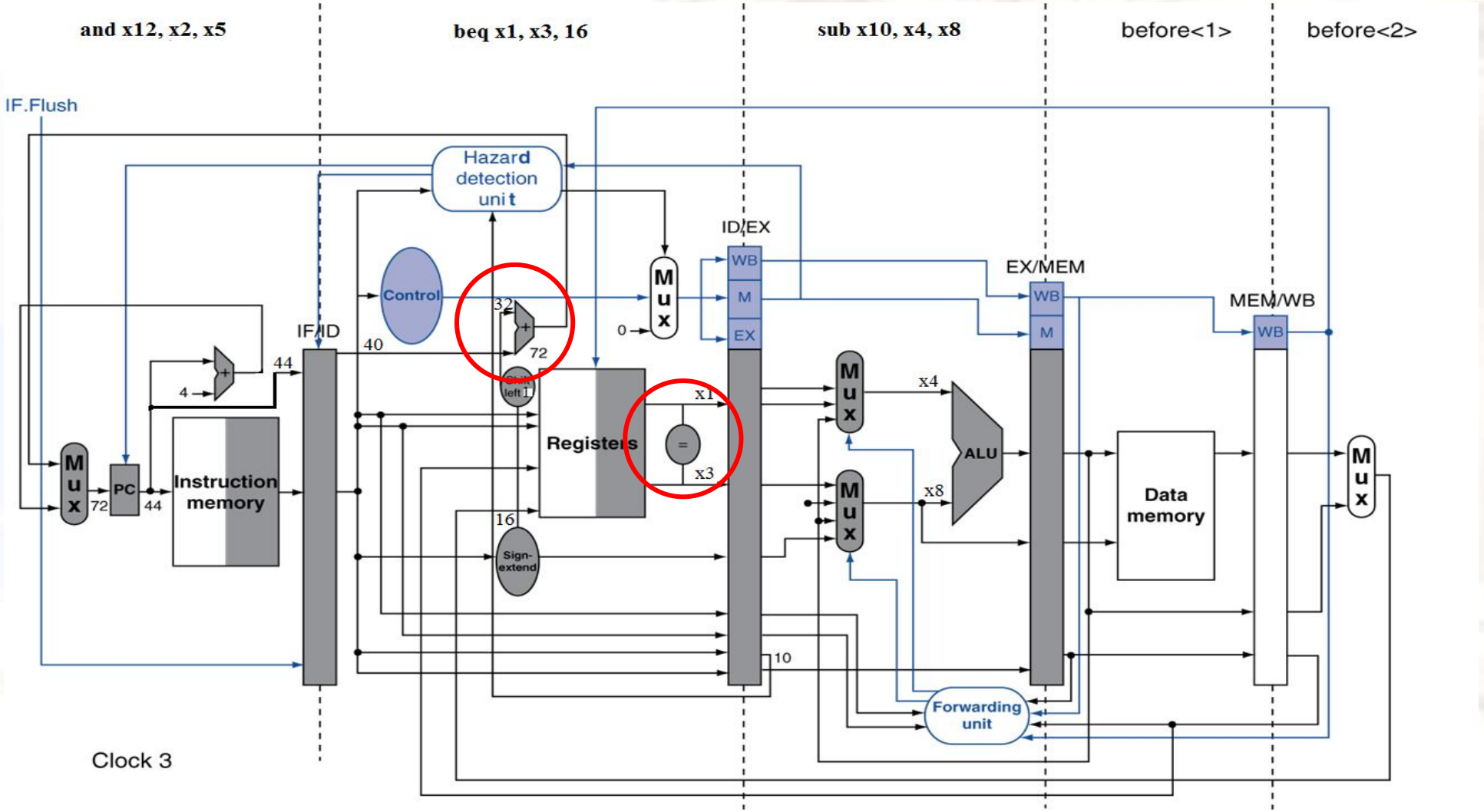
Move the Branch Computation Forward



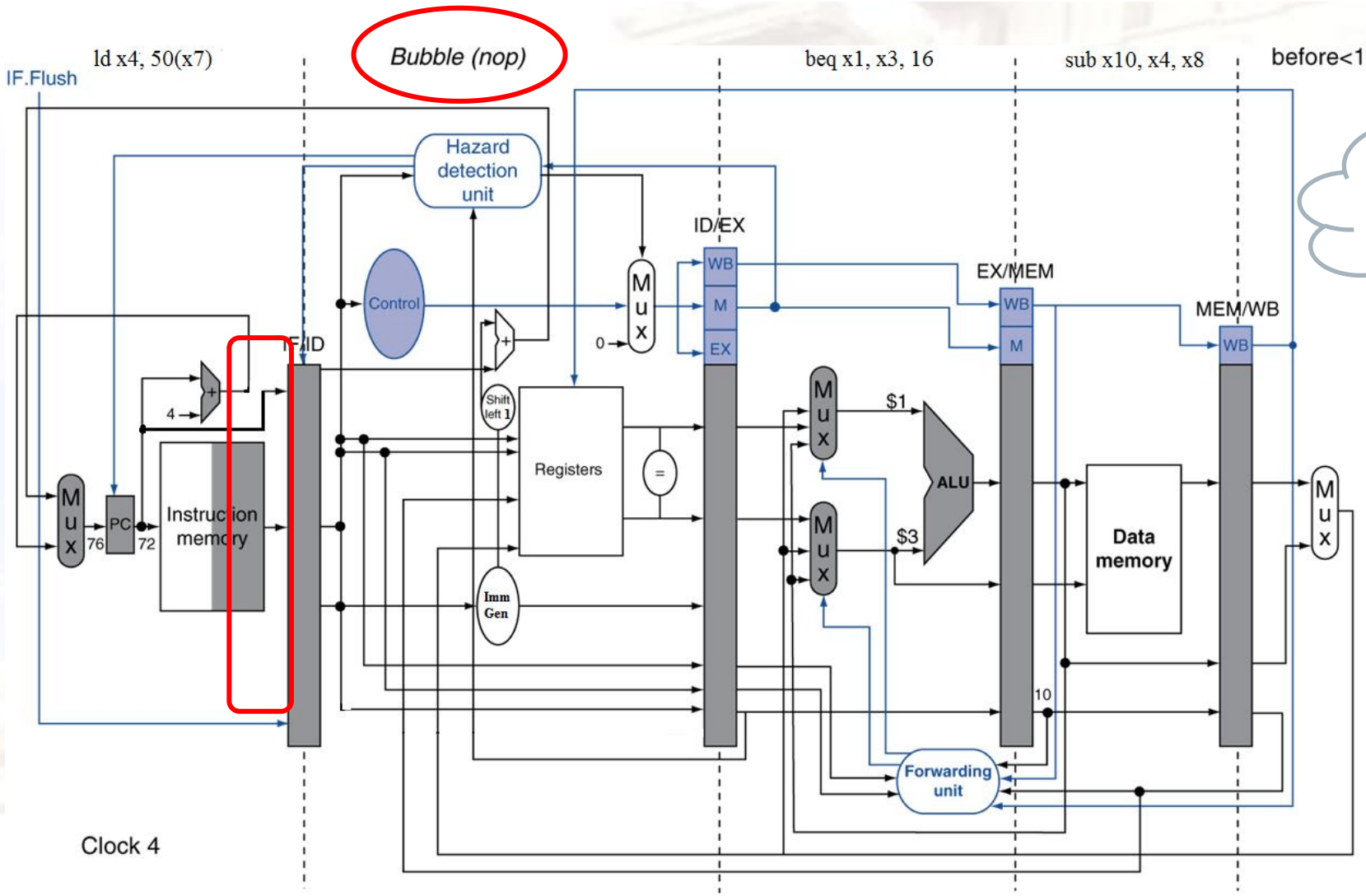
Move the Branch Computation more Forward



Example: Branch Taken



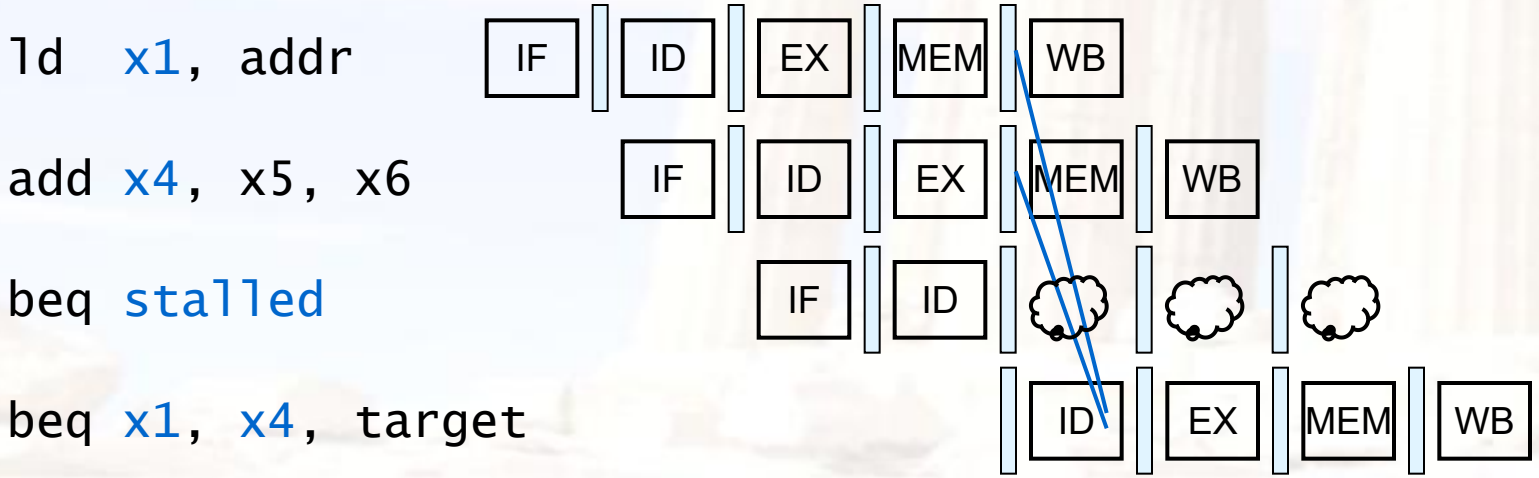
Example: Branch Taken



What's the possible problem ?

Data Hazards for Branches

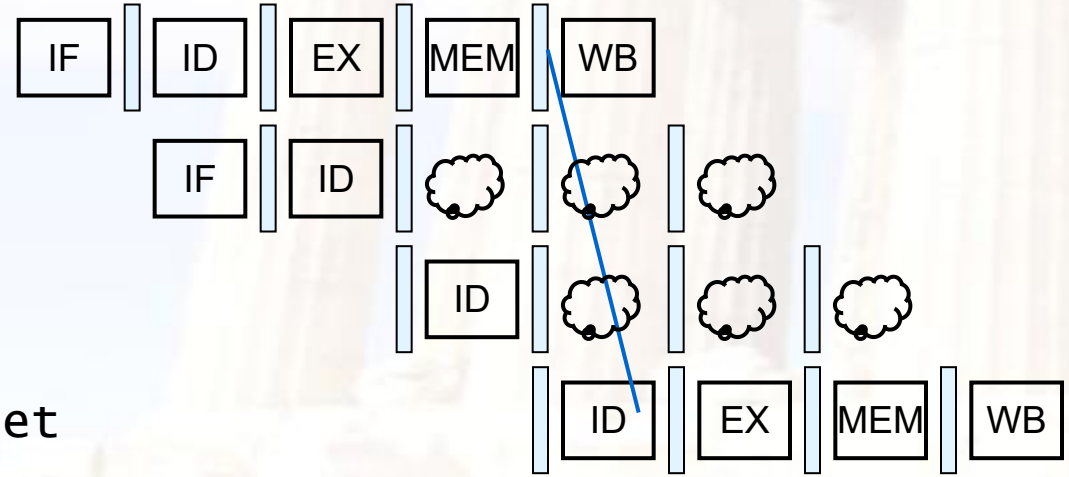
- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
ld  x1, addr
beq stalled
beq stalled
beq x1, x0, target
```



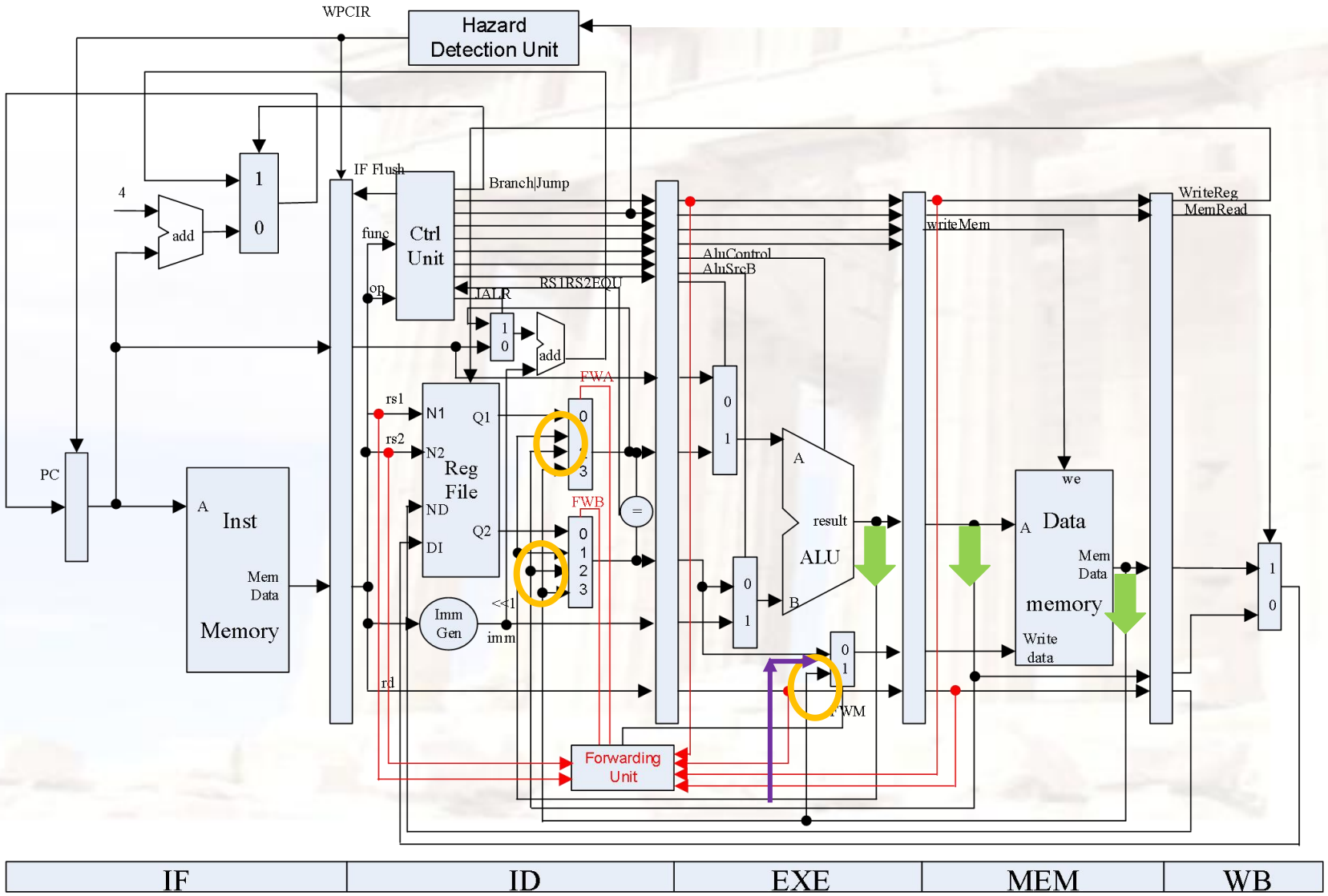
Can the stalls on the previous page be reduced ?

- Yes. Move the forwarding path from EX stage to ID stage.

though at the cost of Clock cycle time be increased.

Please note:

the positions of the sources ➡/sinks ○ of the forwarding paths.



Question: When the purple forwarding path will be needed ?

Fowarding paths in Lab1