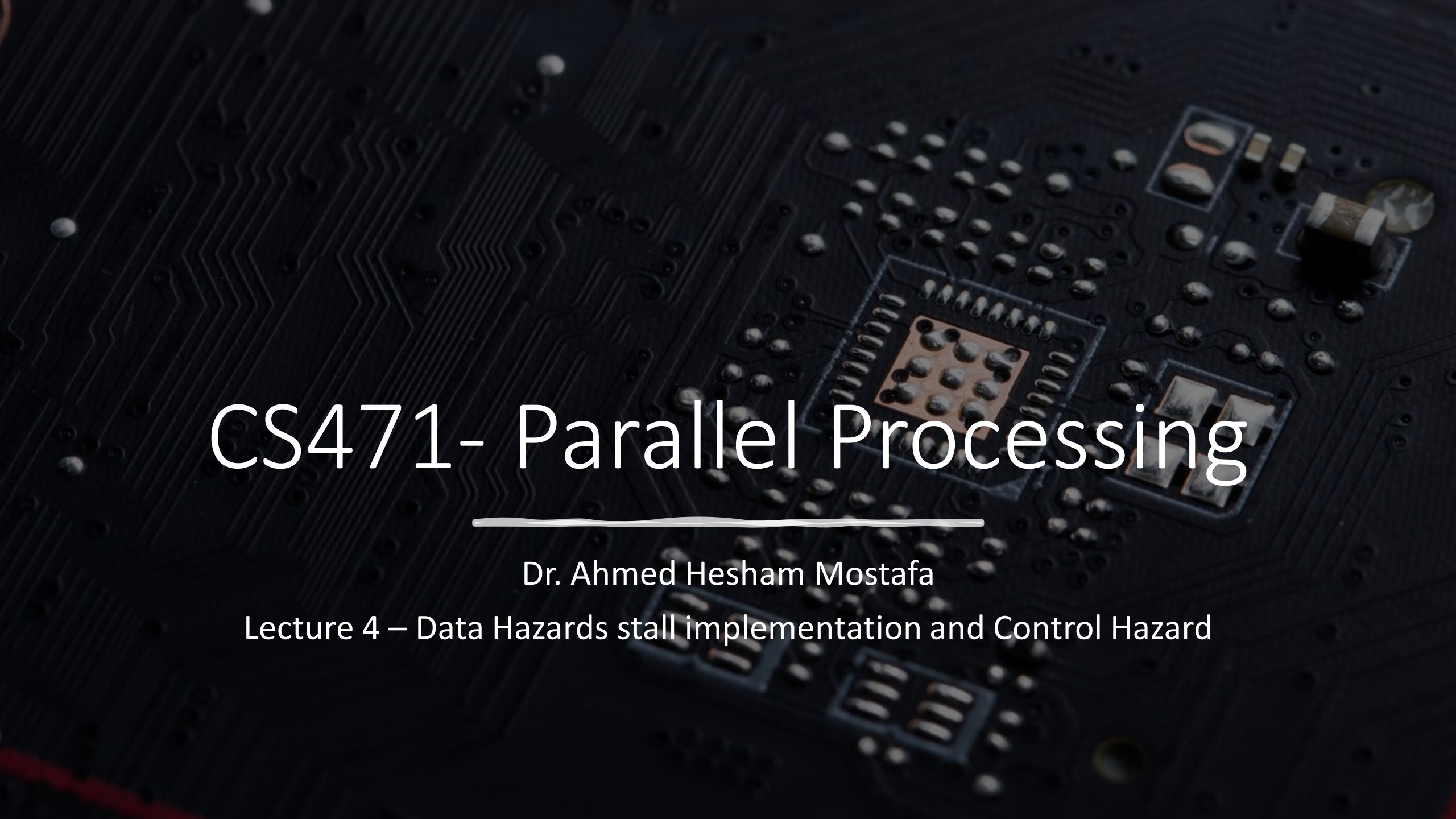


CS471- Parallel Processing

Dr. Ahmed Hesham Mostafa

Lecture 4 – Data Hazards stall implementation and Control Hazard

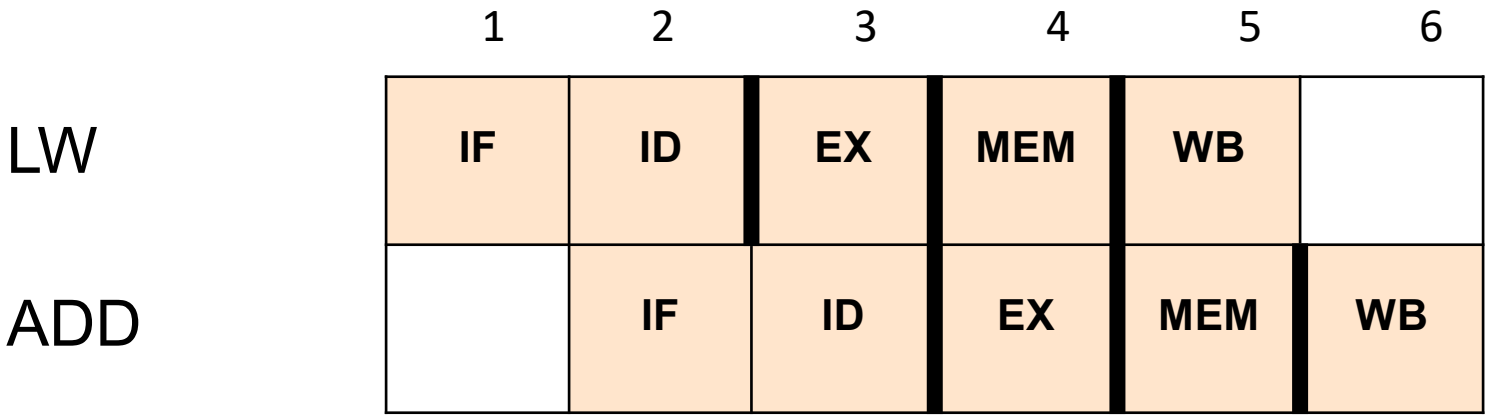
Data Hazards and Stalls

- So far, we've only addressed "potential" data hazards, where the forwarding unit was able to detect and resolve them without affecting the performance of the pipeline.
- There are also "unavoidable" data hazards, which the forwarding unit cannot resolve, and whose resolution does affect pipeline performance.
- We thus add an (unavoidable) hazard detection unit, which detects them and introduces stalls to resolve them.

Data Hazards & Stalls

- Identify the true data hazard in this sequence:

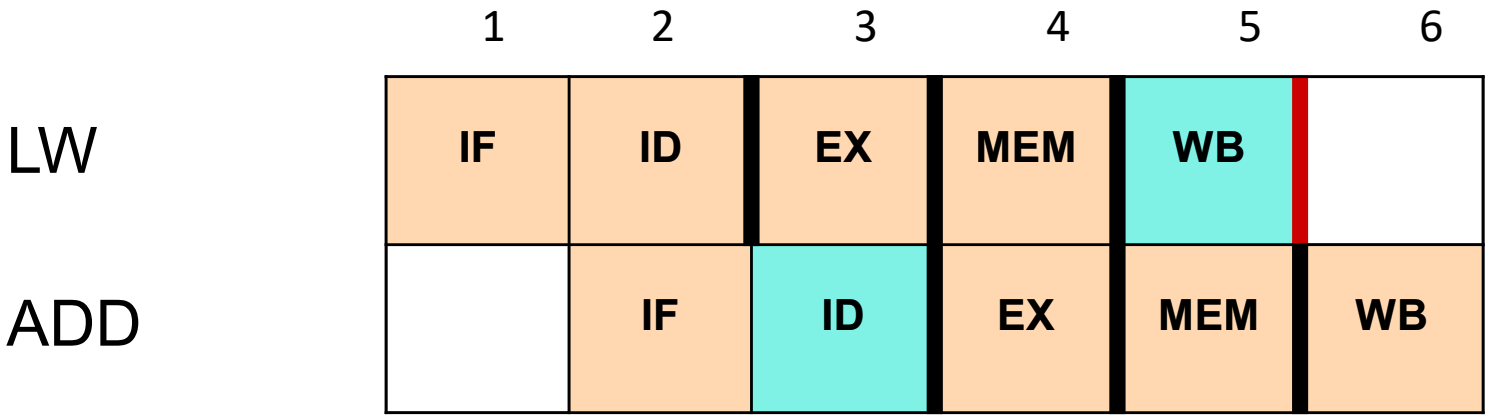
```
LW    $s0,  100($t0)      ;$s0 = memory value
ADD   $t2,  $s0,  $t3     ;$t2 = $s0 + $t3
```



Data Hazards & Stalls

- Identify the true data hazard in this sequence:

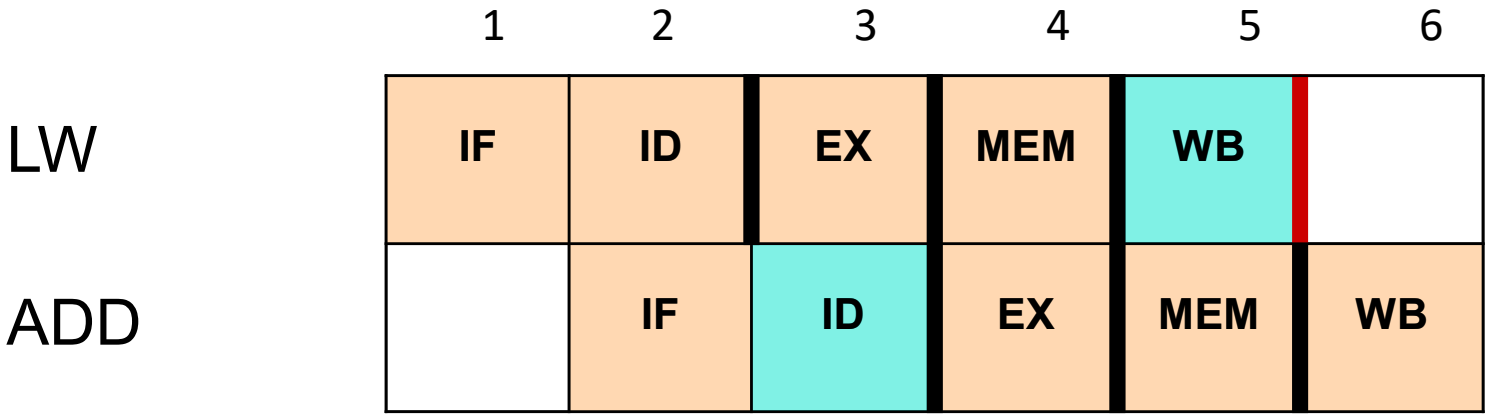
```
LW    $s0,   100($t0)      ;$s0 = memory value
ADD   $t2,   $s0,   $t3    ;$t2 = $s0 + $t3
```



LW doesn't write $s0 to Reg File until the end of CC5, but ADD reads $s0 from Reg File in CC3

Data Hazards & Stalls

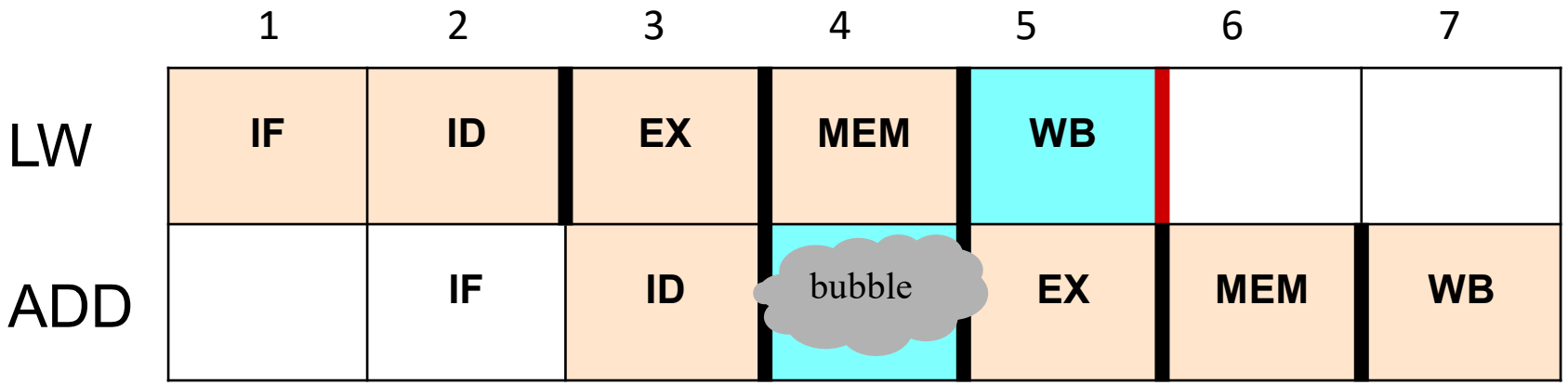
```
LW    $s0,   100($t0)      ;$s0 = memory value
ADD   $t2,   $s0,   $t3     ;$t2 = $s0 + $t3
```



EX/MEM forwarding won't work, because the data isn't loaded from memory until CC4 (so it's not in EX/MEM register)
MEM/WB forwarding won't work either, because ADD executes in CC4

Data Hazards & Stalls

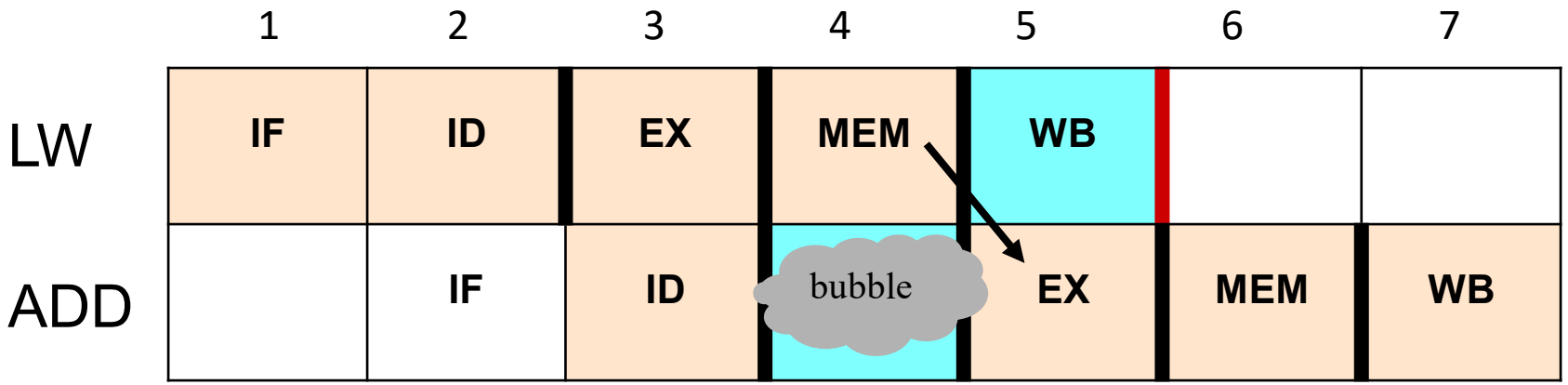
```
LW    $s0,  100($t0)      ;$s0 = memory value
ADD   $t2,  $s0,  $t3     ;$t2 = $s0 + $t3
```



We must handle this hazard by "stalling" the pipeline for 1 Clock Cycle (bubble)

Data Hazards & Stalls

```
LW    $s0,   100($t0)      ;$s0 = memory value
ADD   $t2,   $s0,   $t3     ;$t2 = $s0 + $t3
```

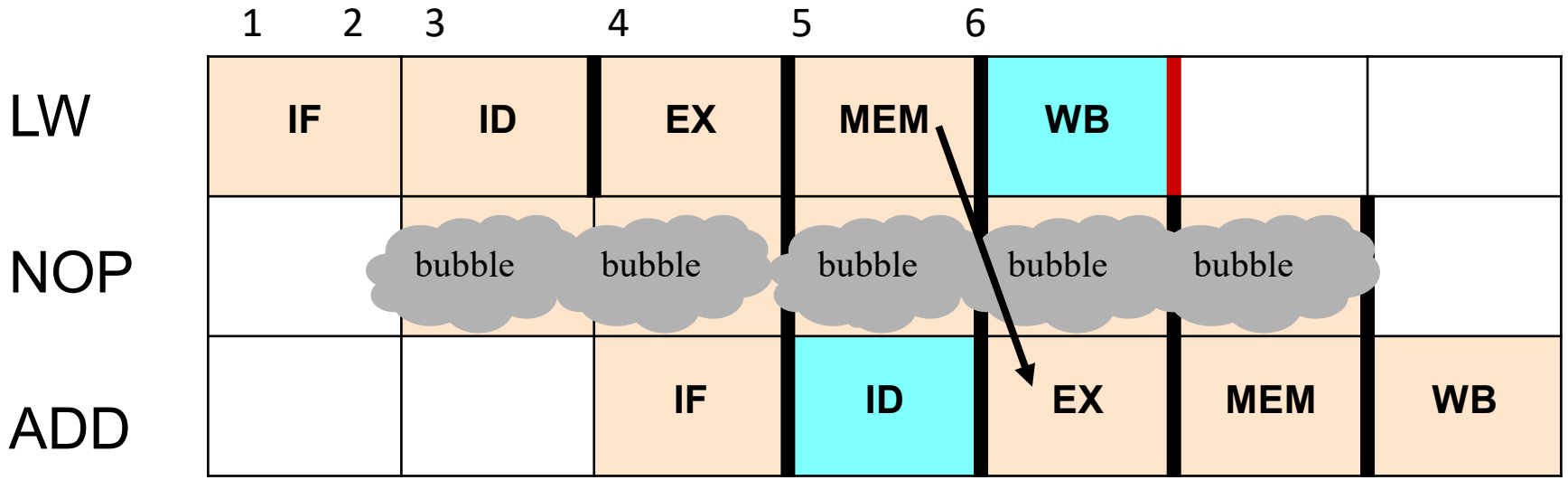


We can then use MEM/WB forwarding, but of course there is still a performance loss

Data Hazards & Stalls: implementation

- Stall Implementation #1: Compiler detects hazard and inserts a NOP (no reg changes (SLL $0, $0, 0))

```
LW    $s0,   100($t0)      ;$s0 = memory value
NOP        ;dummy instruction
ADD   $t2,   $s0,   $t3    ;$t2 = $s0 + $t3
```



- Problem: we have to rely on the compiler

Data Hazards & Stalls: implementation

- Stall Implementation #2: Add a “hazard detection unit” to stall current instruction for 1 CC if:
- ID-Stage Hazard Detection and Stall Condition:

Load word	lw	rt,imm(rs)
Store word	sw	rt,imm(rs)

If ((ID/EX.MemRead = 1) & ;only a LW reads mem

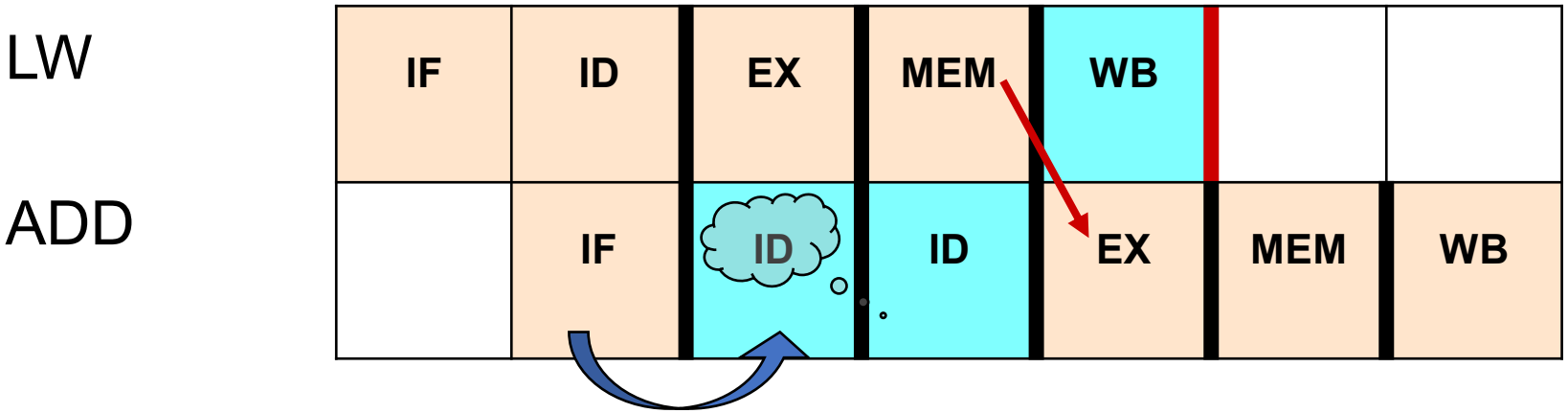
((ID/EX.RegRT = IF/ID.RegRS) || ;RS will read load dest (RT)

(ID/EX.RegRT = IF/ID.RegRT))) ;RT will read load dest

```
          RT          RS
LW    $s0,   100($t0)      ;$s0 = memory value
ADD   $t2,   $s0,   $t3    ;$t2 = $s0 + $t3
      RD     RS     RT
```

Data Hazards & Stalls: implementation

- The effect of this stall will be to repeat the ID Stage of the current instruction. Then we do the MEM/WB forwarding on the next Clock Cycle



We do this by preserving the current values in IF/ID for use on the next Clock Cycle

Control Hazards

- What about branches?

A control hazard occurs if there is a control instruction (e.g. BEQ) and the program counter (PC) following the control instruction is not known until the control instruction computes if the branch should be taken

- e.g.

```
        beq r1, r2, L
        add r3, r0, r3
        sub r5, r4, r6
L:      or  r3, r2, r4
```

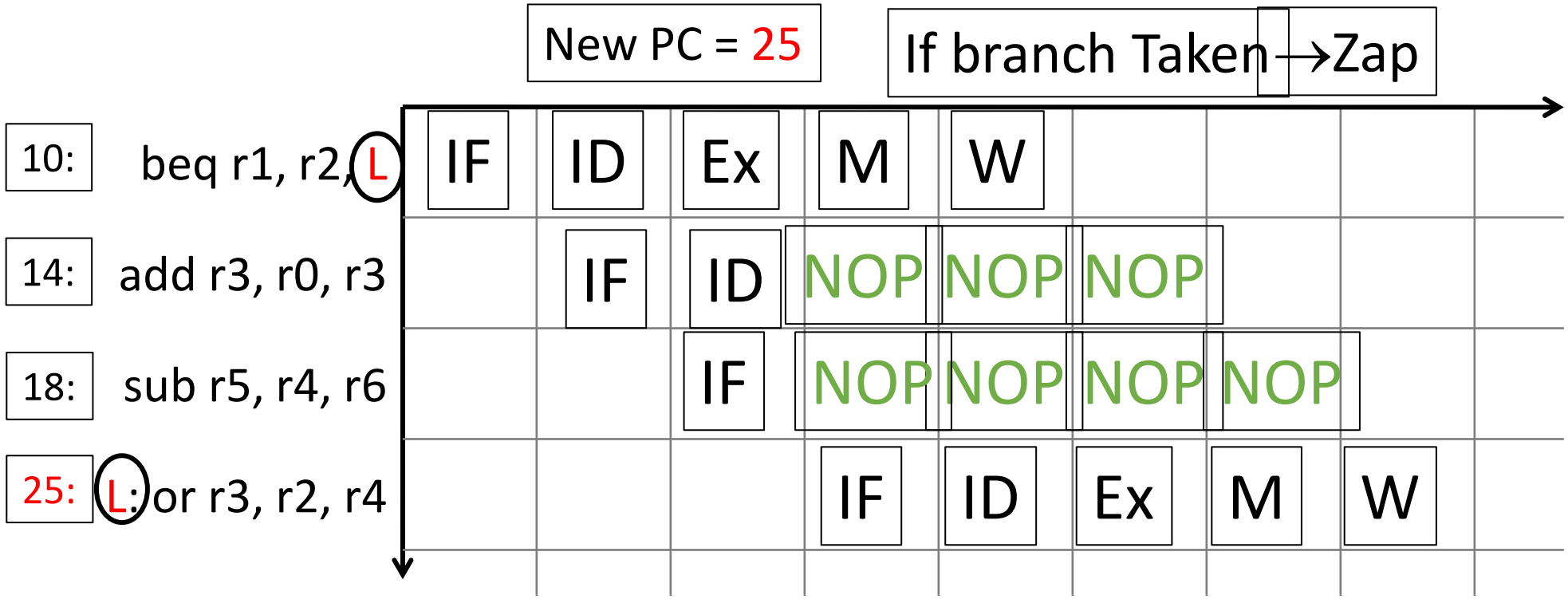
Control Hazards

- **Control Hazards**
  - instructions are fetched in stage 1 (IF)
  - branch and jump decisions occur in stage 3 (EX)
  - i.e. next PC is not known until ***2 cycles*** *after* branch/jump

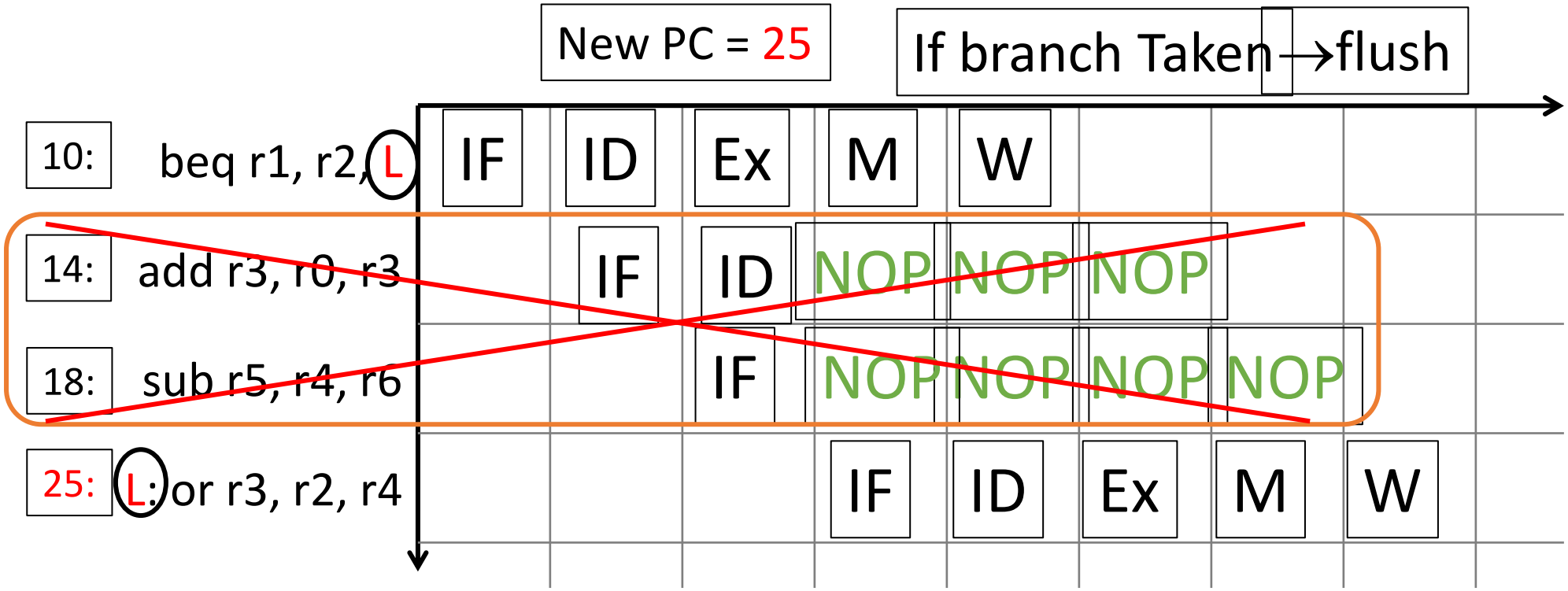
Control Hazards

- What happens to instr following a branch, if the branch ***is taken***?
- Stall (+ Zap/Flush)
  - prevent PC update
  - clear IF/ID pipeline register
    - instruction just fetched might be the wrong one, so convert to nop
  - allow the branch to continue into the EX stage

Control Hazards - if the branch *is taken*



Control Hazards - if the branch *is taken*



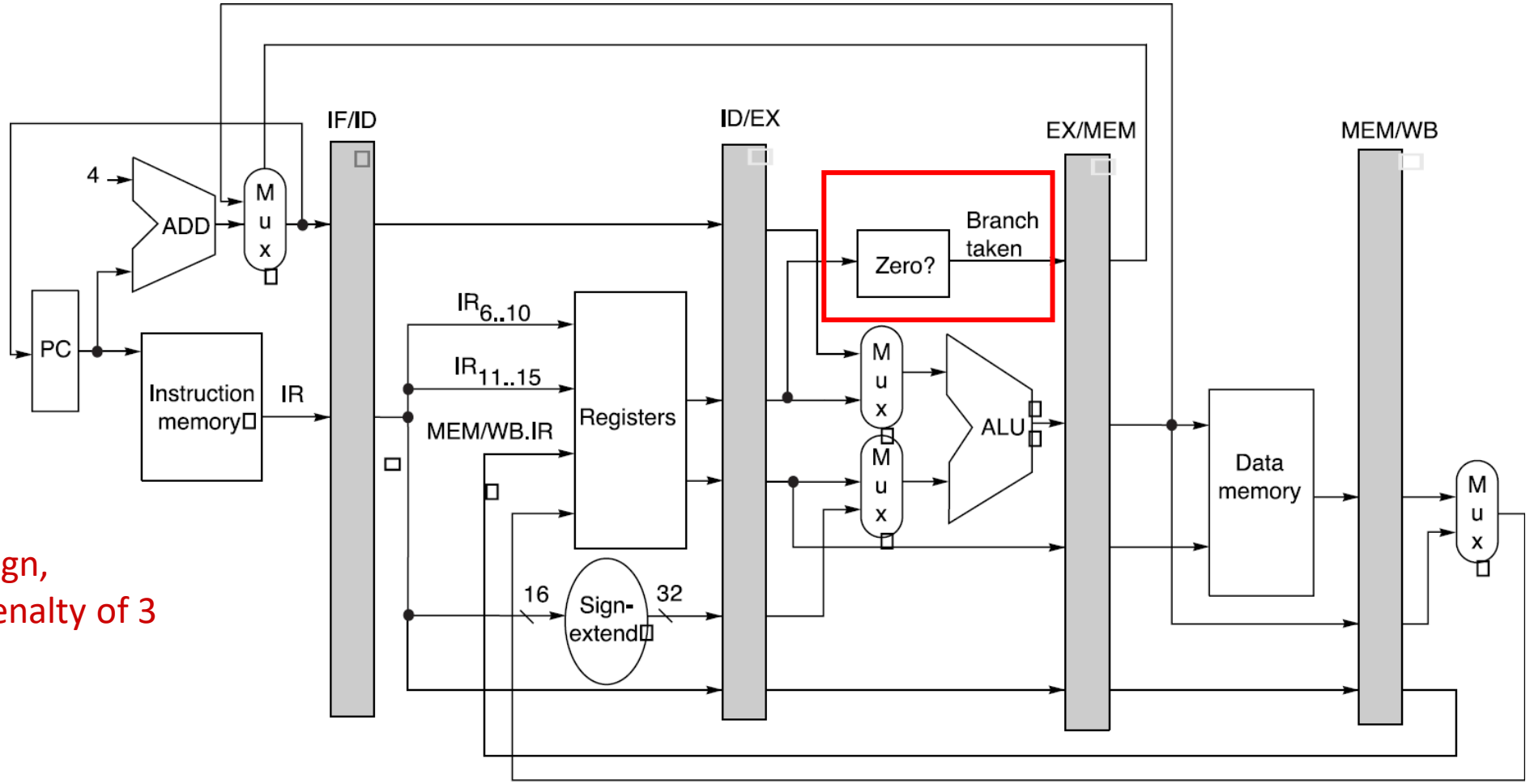
Control Hazards

- Control hazards occur because the PC following a control instruction is not known until the control instruction computes if the branch should be taken or not.
- If the branch is taken, then need to zap/flush instructions.
- There is a performance penalty for branches:
- Need to stall, then may need to zap (flush) subsequent instructions that have already been fetched.

Reduce the cost of control hazards?

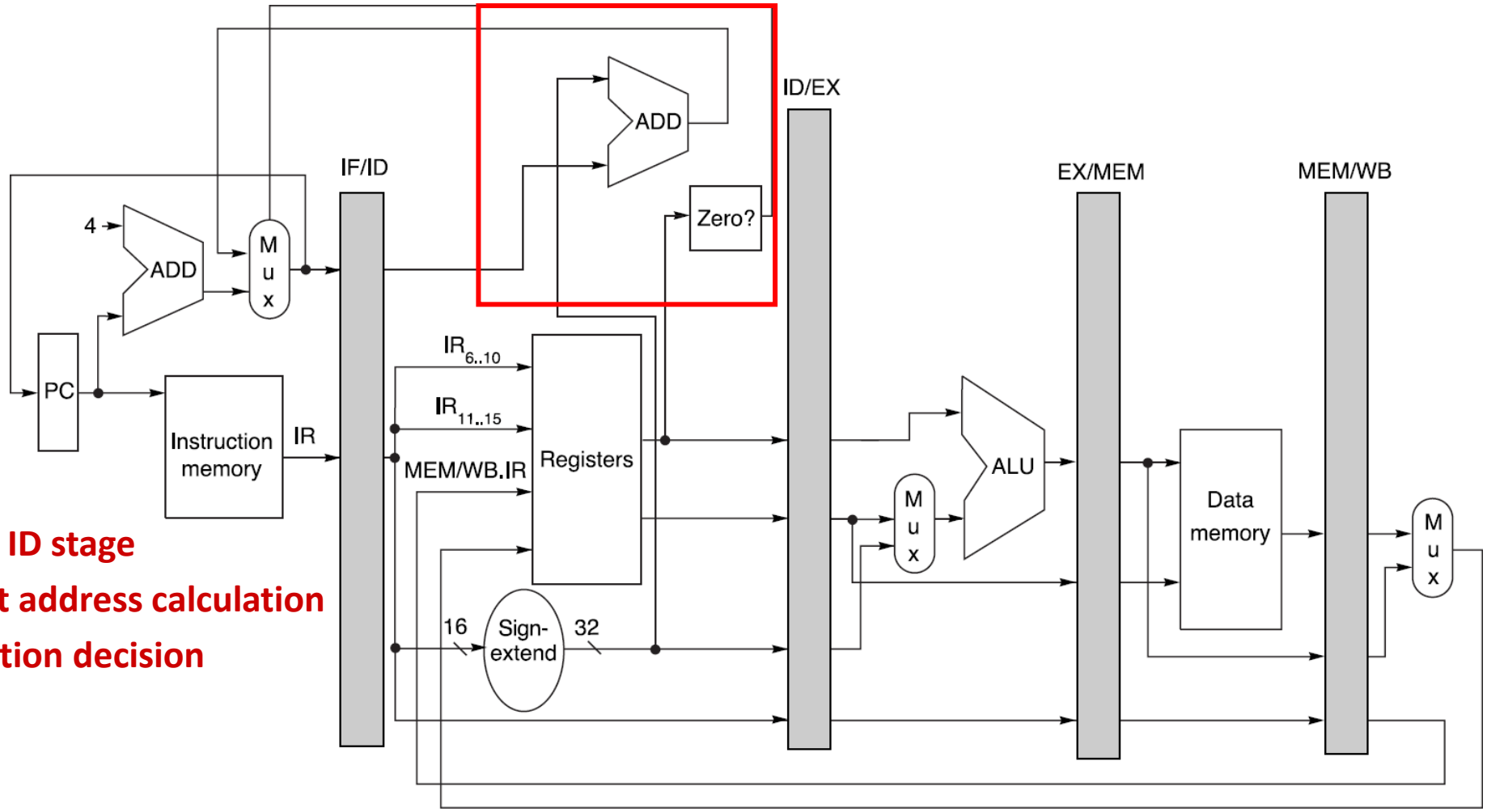
- Can we forward/bypass values for branches?
  - We can move branch calc from EX to ID
  - will require new bypasses into ID stage; or can just zap the second instruction
- What happens to instructions following a branch, if branch taken?
  - Still need to zap/flush instructions
- Is there still a performance penalty for branches
  - Yes, need to stall, then may need to zap (flush) subsequent instructions that have already been fetched

Reduce the cost of control hazards move branch calc from EX to ID



In our original Design, branches have a penalty of 3 cycles

Reduce the cost of control hazards move branch calc from EX to ID



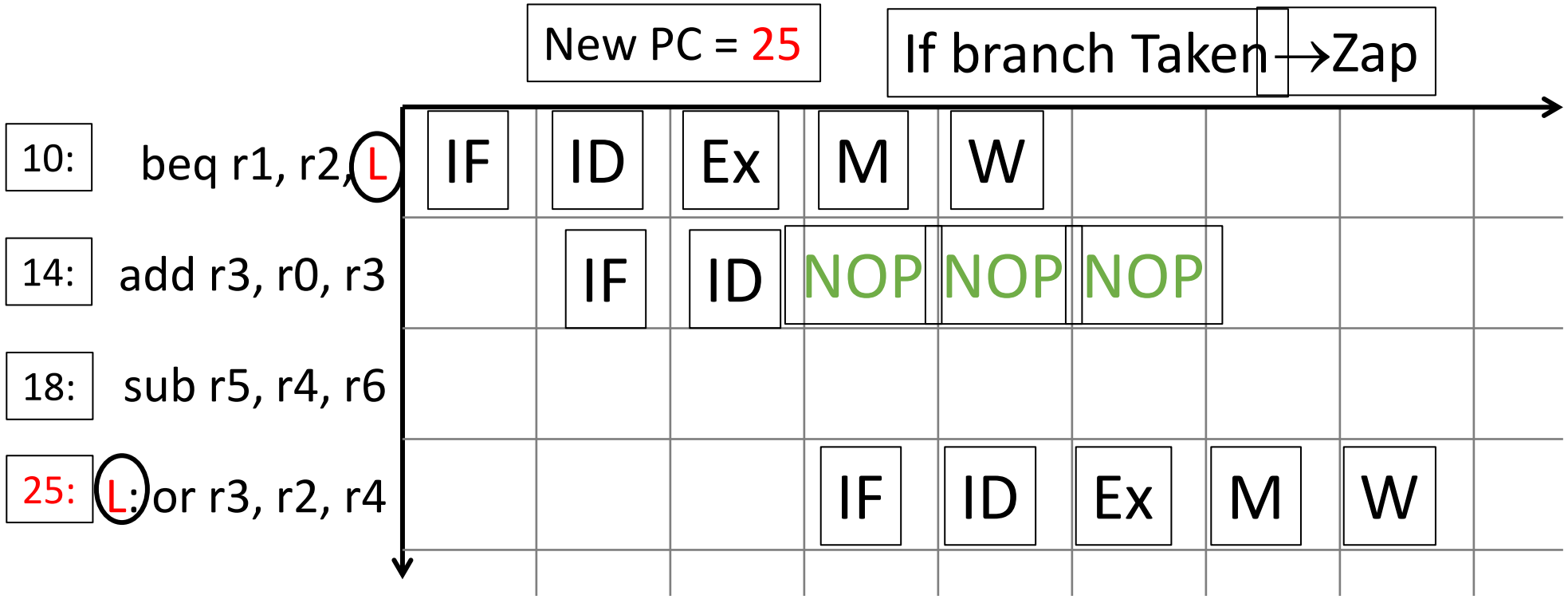
Move following to ID stage

a) Branch-target address calculation

b) Branch condition decision

Reduced penalty (1 cycle) when branch take!

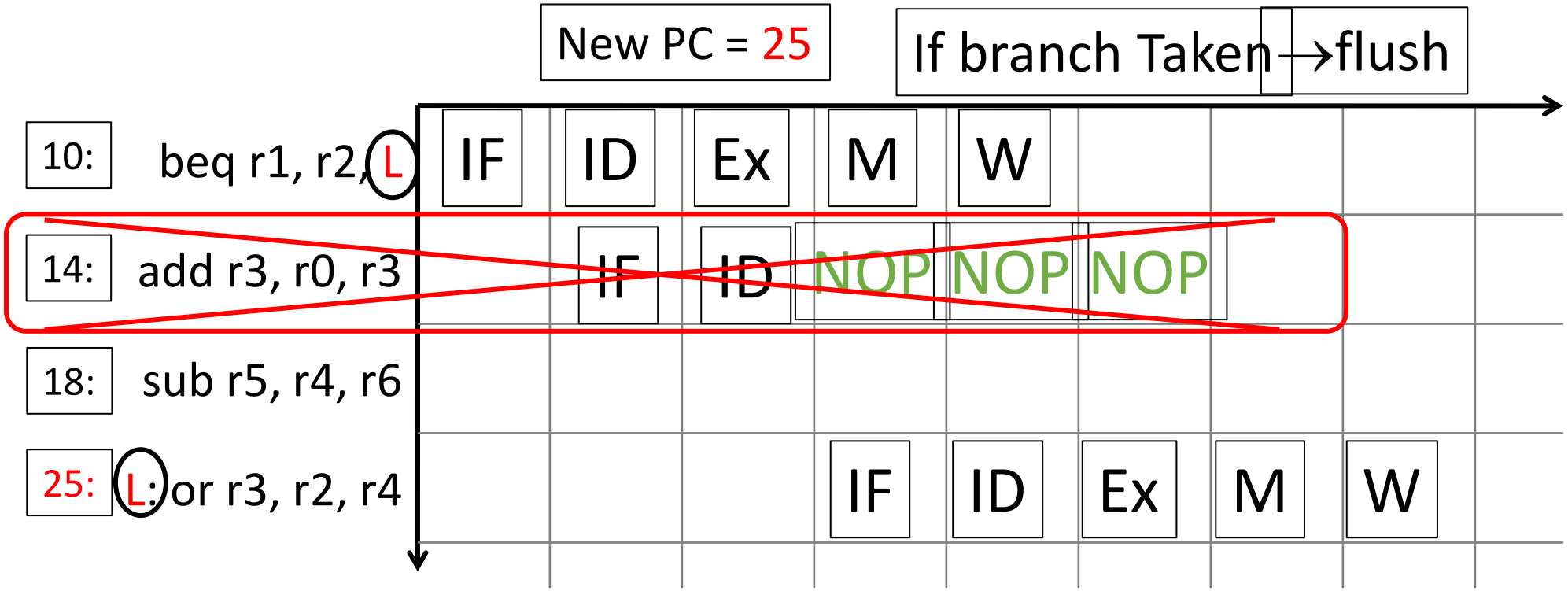
Reduce the cost of control hazards move branch calc from EX to ID



Can optimize and move branch and jump decision to stage 2 (ID)

i.e. next PC is not known until ***1 cycles*** *after* branch/jump

Reduce the cost of control hazards move branch calc from EX to ID



Can optimize and move branch and jump decision to stage 2 (ID)

i.e. next PC is not known until ***1 cycles*** *after* branch/jump

Reduce the cost of control hazards
Delay slot

- ISA says N instructions after branch/jump are ***always*** executed
- MIPS has 1 branch delay slot
- Whether the branch taken or not, the instruction following branch is ***always*** executed
- Delay Slots can potentially increase performance due to control hazards by putting a useful instruction in the delay slot since the instruction in the delay slot will ***always*** be executed.
- Requires software (compiler) to make use of delay slot.
- Put nop in delay slot if not able to put useful instruction in the delay slot.

Control Hazard Solutions

- Stall - stop fetching instr. until result is available
  - Significant performance penalty
  - Hardware required to stall
- Delayed branch - specify in architecture that following instruction is always executed
  - Compiler re-orders instructions into delay slot
  - Insert "NOP" (no-op) operations when can't use (~50%)
  - This is how original MIPS worked
- Predict - assume an outcome and continue fetching (undo if prediction is wrong)
  - Performance penalty only when guess wrong
  - Hardware required to "squash" instructions

References

- Lecture Notes, Lecture 4: Pipelining Basics & Hazards, Kai Bu, kaibu@zju.edu.cn
- EmbeddedNotes - 7 - Pipeline - 2 - Hazards (Arabic) – YouTube
- EmbeddedNotes - 8 - Pipeline - 3 - Branch Prediction (Arabic) - YouTube
- Lecture Notes, Introduction to Computer Architecture, at the University of California, Santa Barbara. © Behrooz Parhami
- Lecture Notes, Prof. Hakim Weatherspoon, CS 3410, Spring 2015,Cornell University
- Lecture Notes, Prof. Hong Jiang,Courtesy of Prof. Yifeng Zhu, U. of Maine
- Portions of these slides are derived from:Dave Patterson © UCB

Thanks