EI 338: Computer Systems Engineering
(Operating Systems & Computer Architecture)

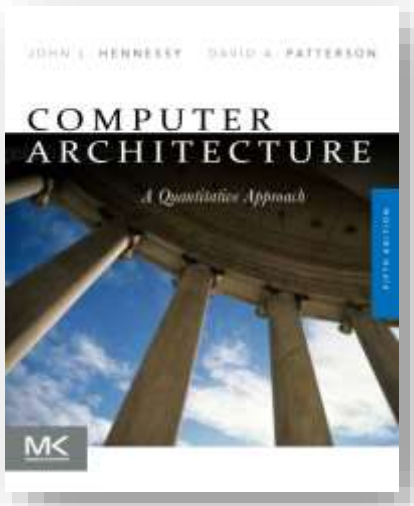
Dept. of Computer Science & Engineering
Chentao Wu
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Download lectures

- ftp://public.sjtu.edu.cn
- User: wuct
- Password: wuct123456

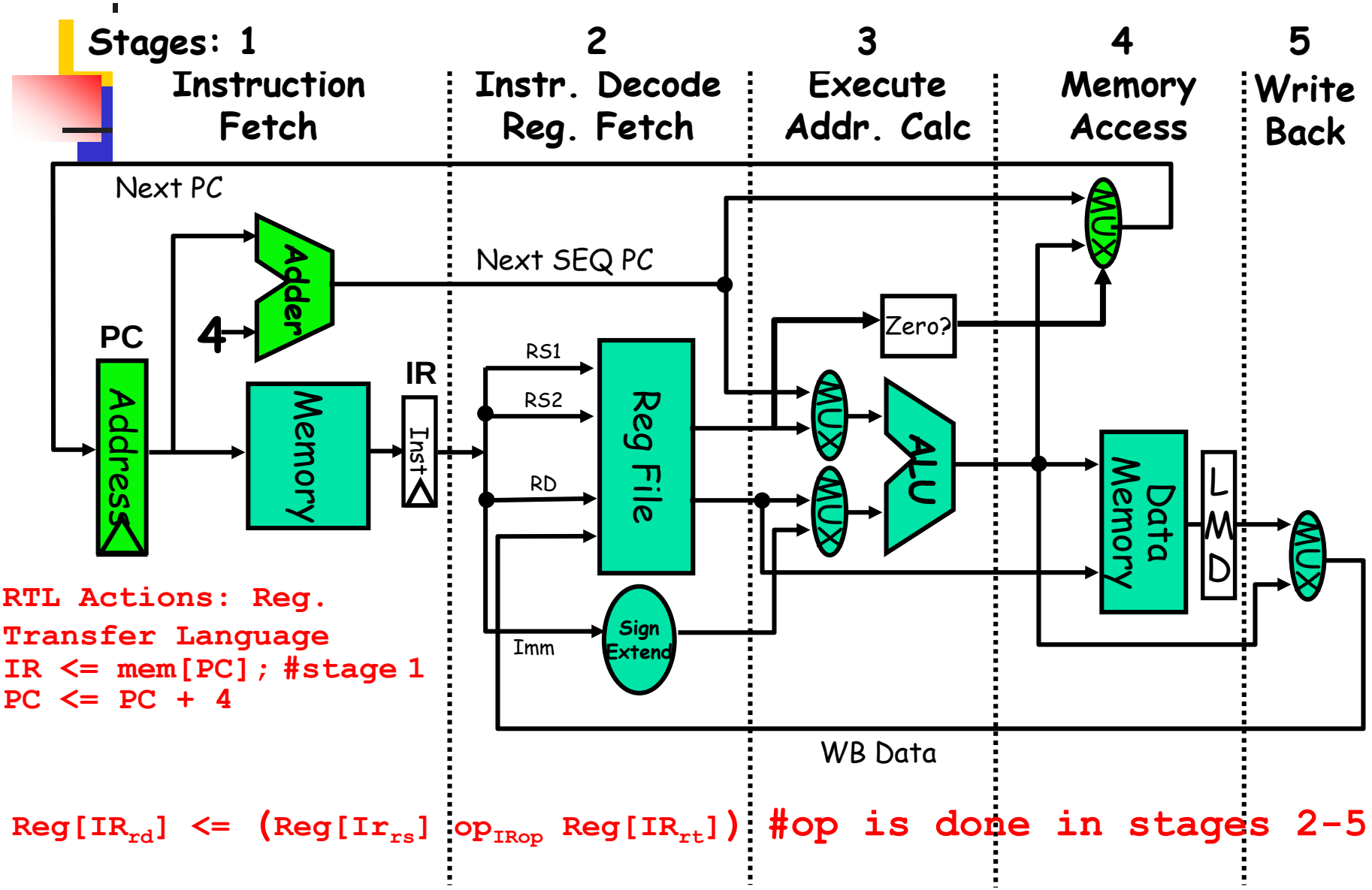
- http://www.cs.sjtu.edu.cn/~wuct/cse/



Appendix C

Pipelining

5 Steps of a (pre-pipelined) MIPS Datapath

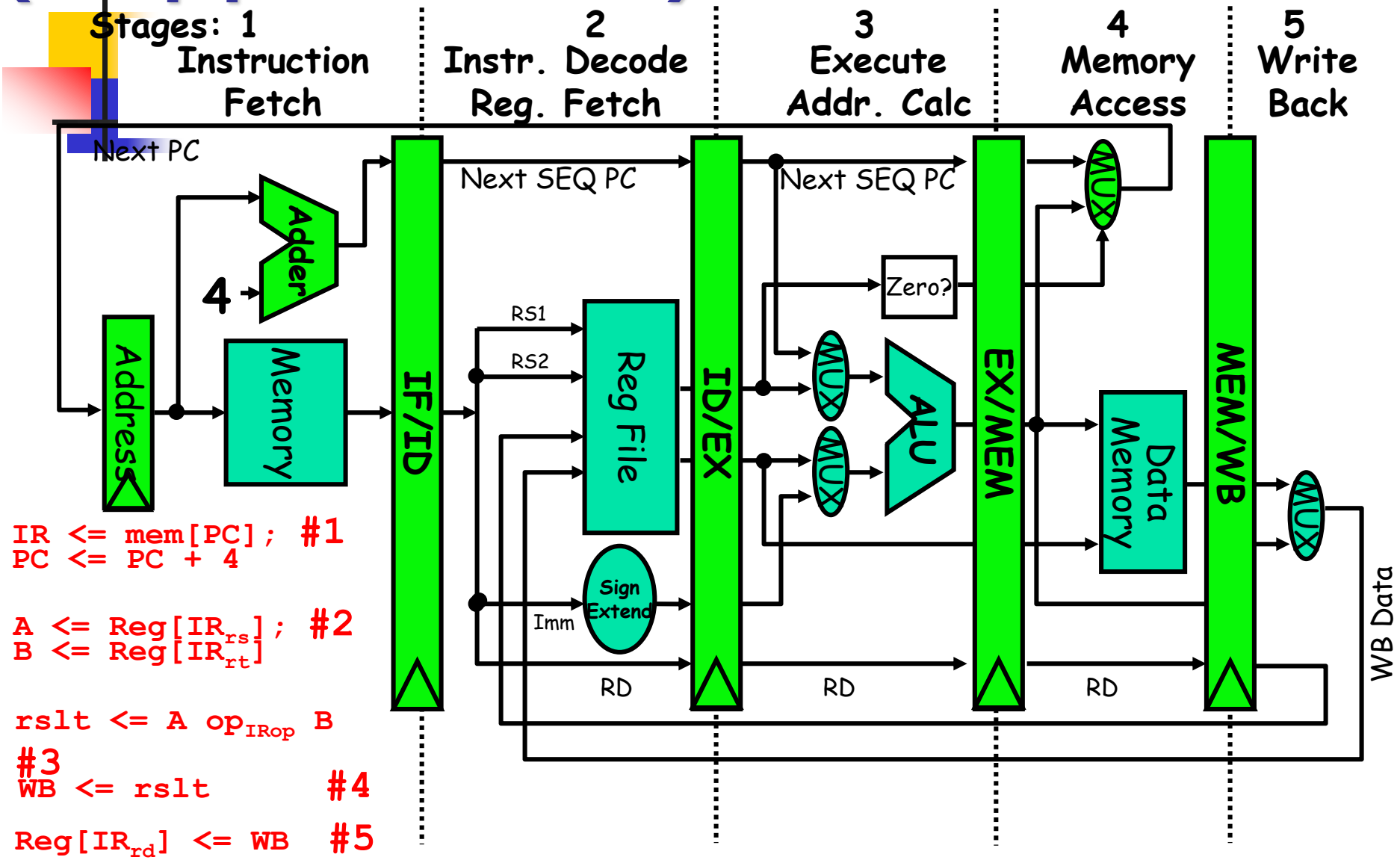


RTL Actions: Reg. Transfer Language

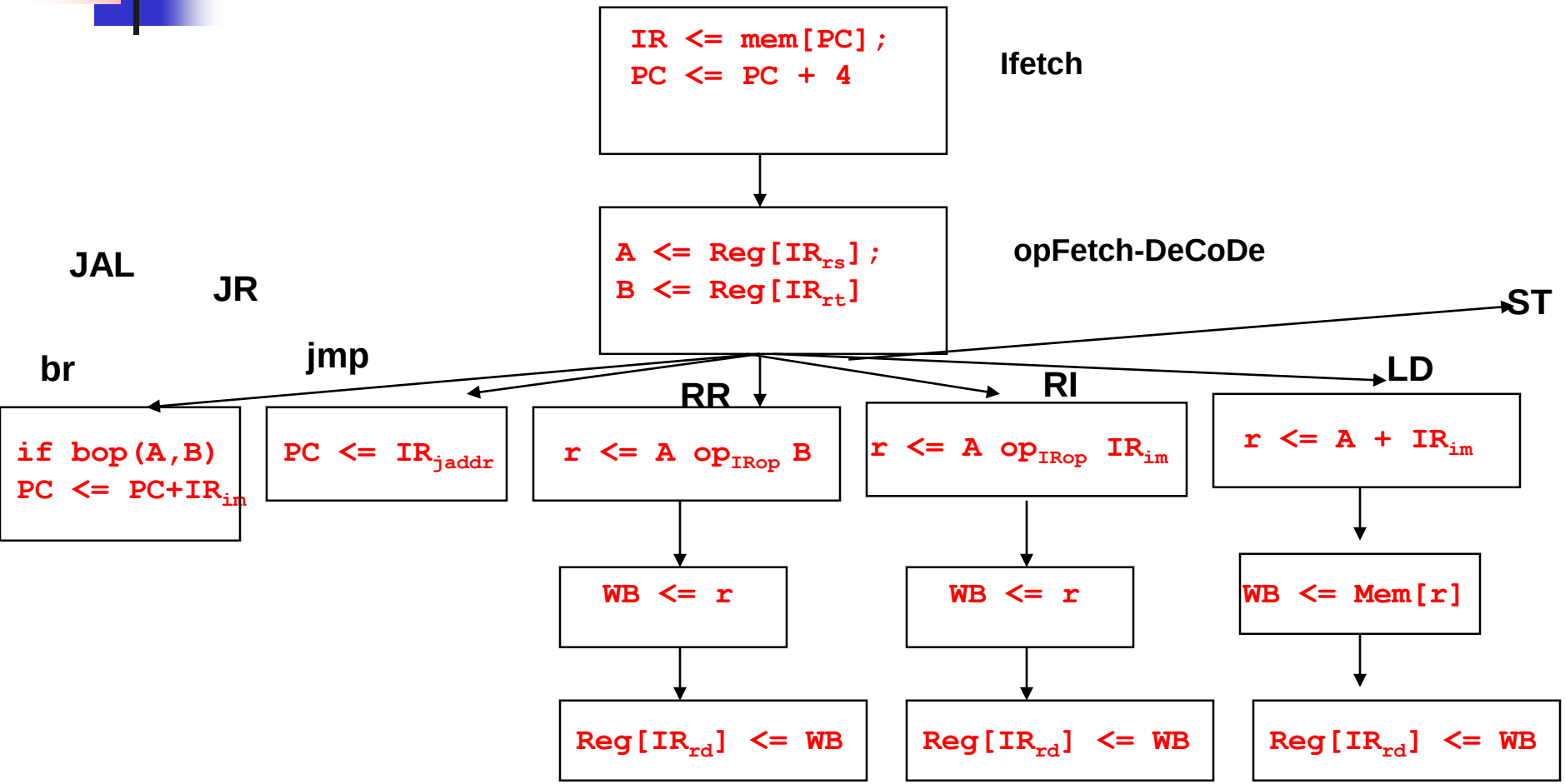
```
IR <= mem[PC]; #stage 1
PC <= PC + 4
```

$Reg[IR_{rd}] <= (Reg[Ir_{rs}]\ op_{IRop}\ Reg[IR_{rt}])$ #op is done in stages 2-5

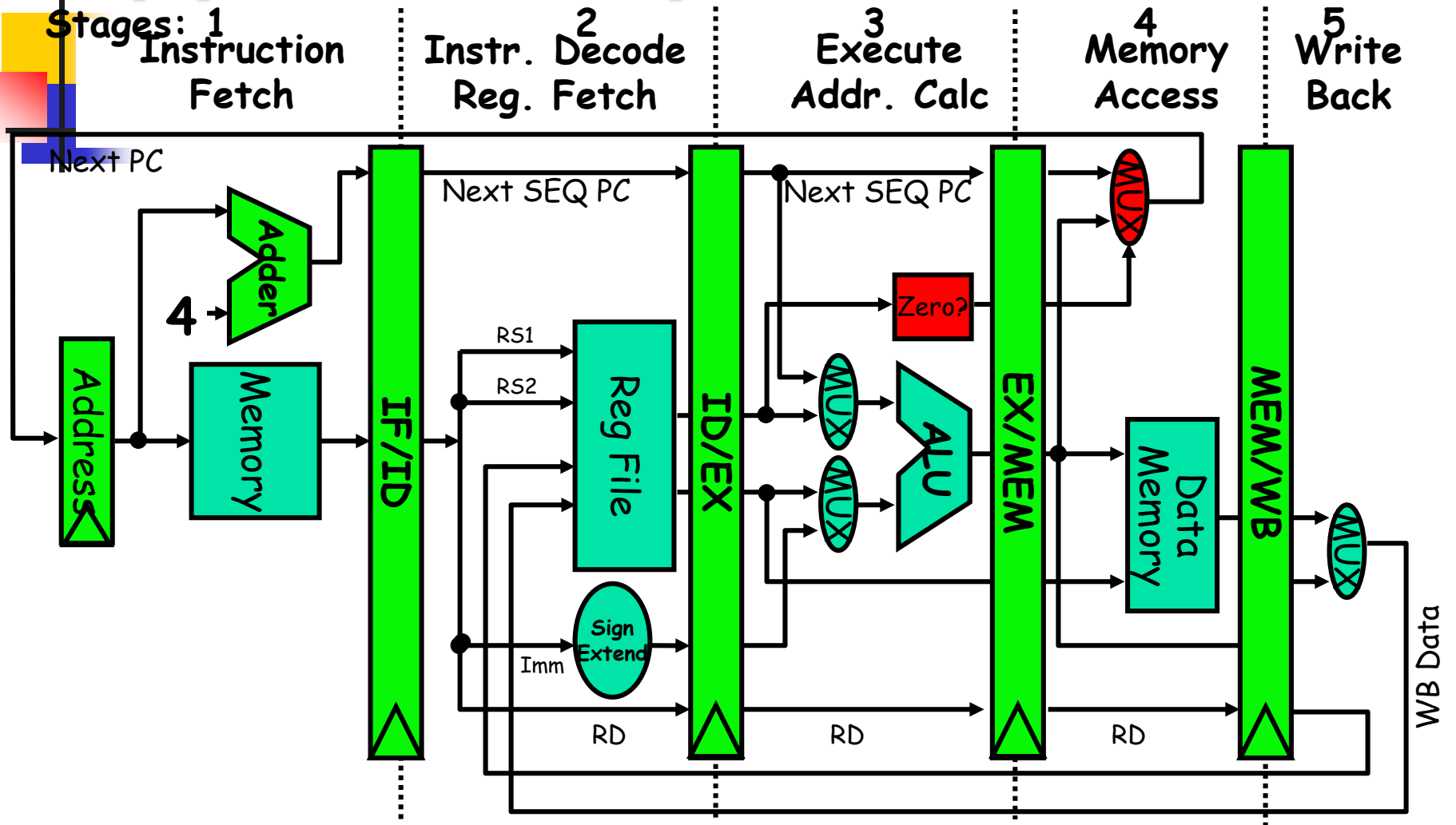
5-Stage MIPS Datapath (has pipeline latches)



Instruction Set Processor Controller

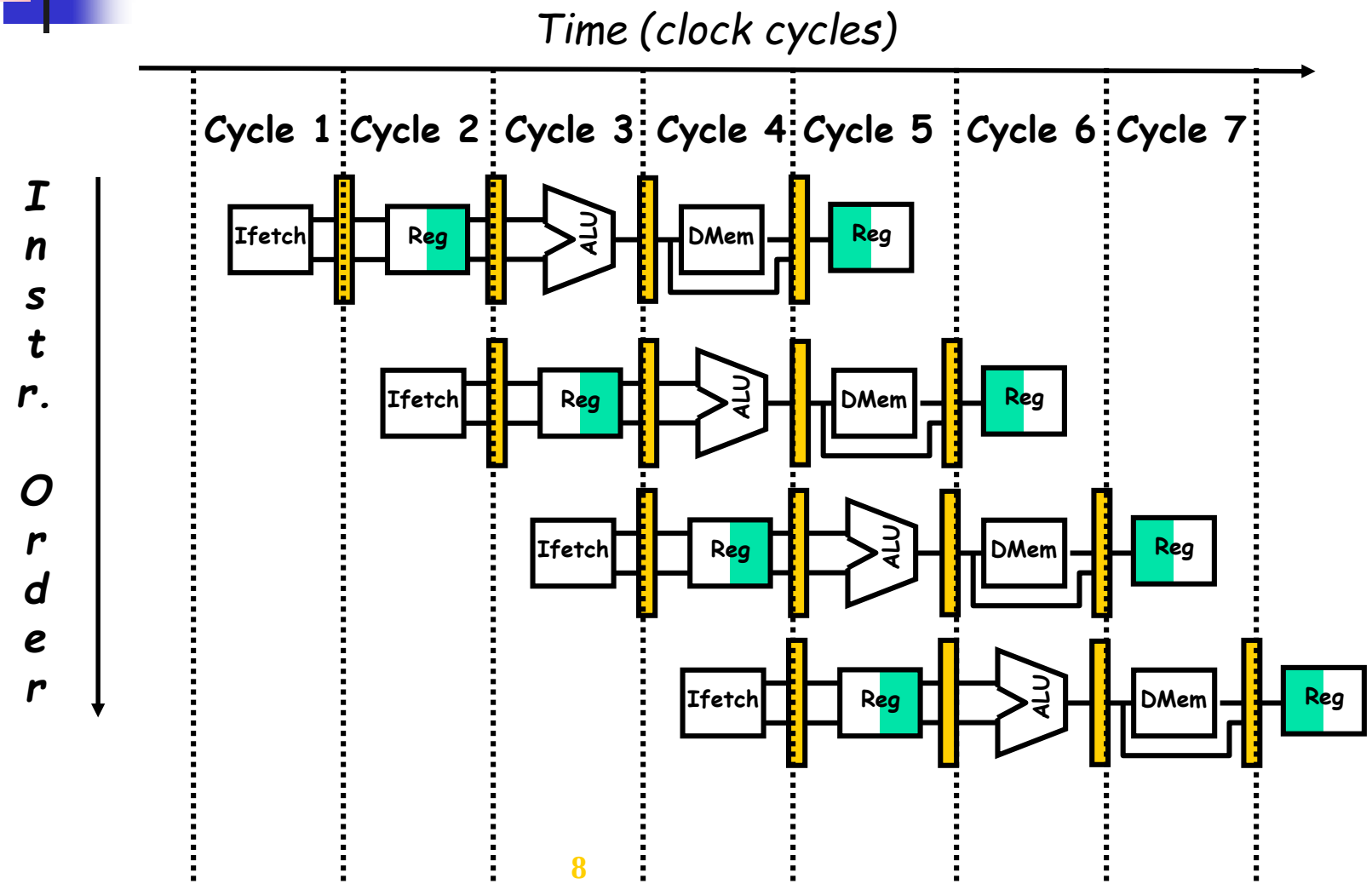


5-Stage MIPS Datapath (has pipeline latches)



- Data stationary control
  - local decode for each instruction phase / pipeline stage

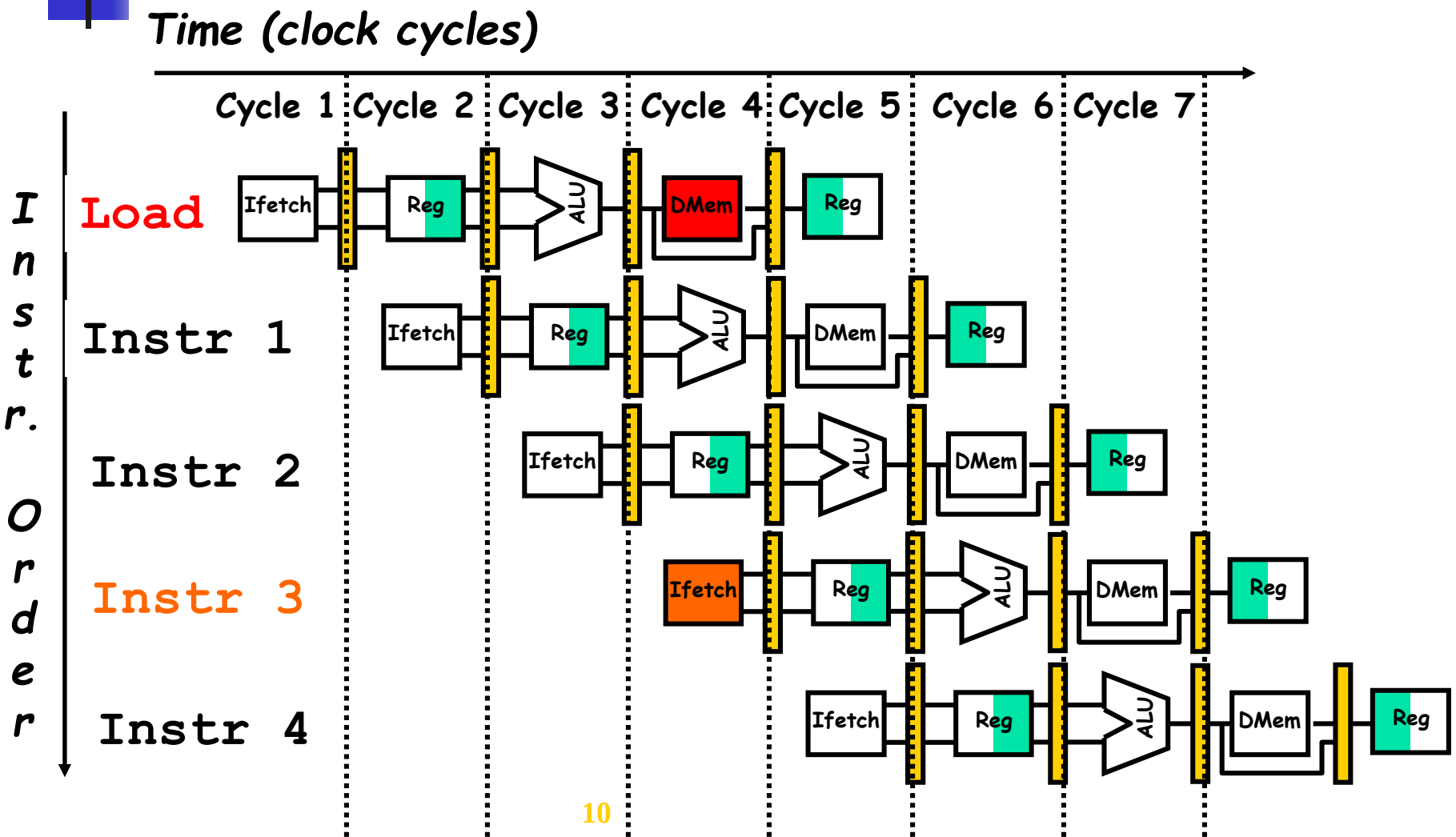
Visualizing Pipelining



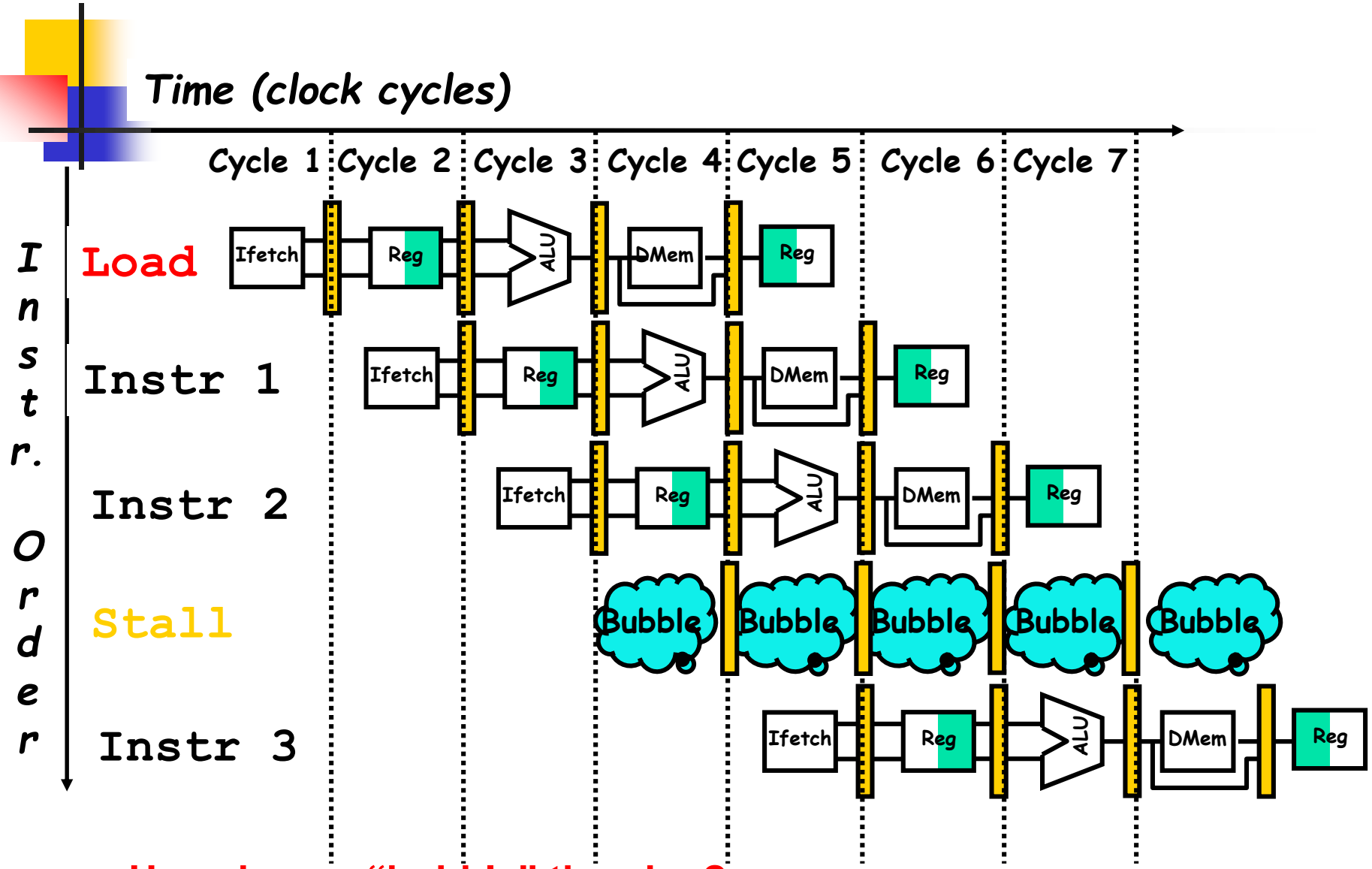
Pipelining is not quite that easy!

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (having a single person to fold and put clothes away at same time)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (having a missing sock in a later wash; cannot put away)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

One Memory_Port / Structural_Hazards



One Memory Port/Structural Hazards



How do you "bubble" the pipe?

Code SpeedUp Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

For simple RISC pipeline, Ideal CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{unpipelined}}{\text{Cycle Time}_{pipelined}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory ("Harvard Architecture")
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Assume loads are 20% of instructions executed

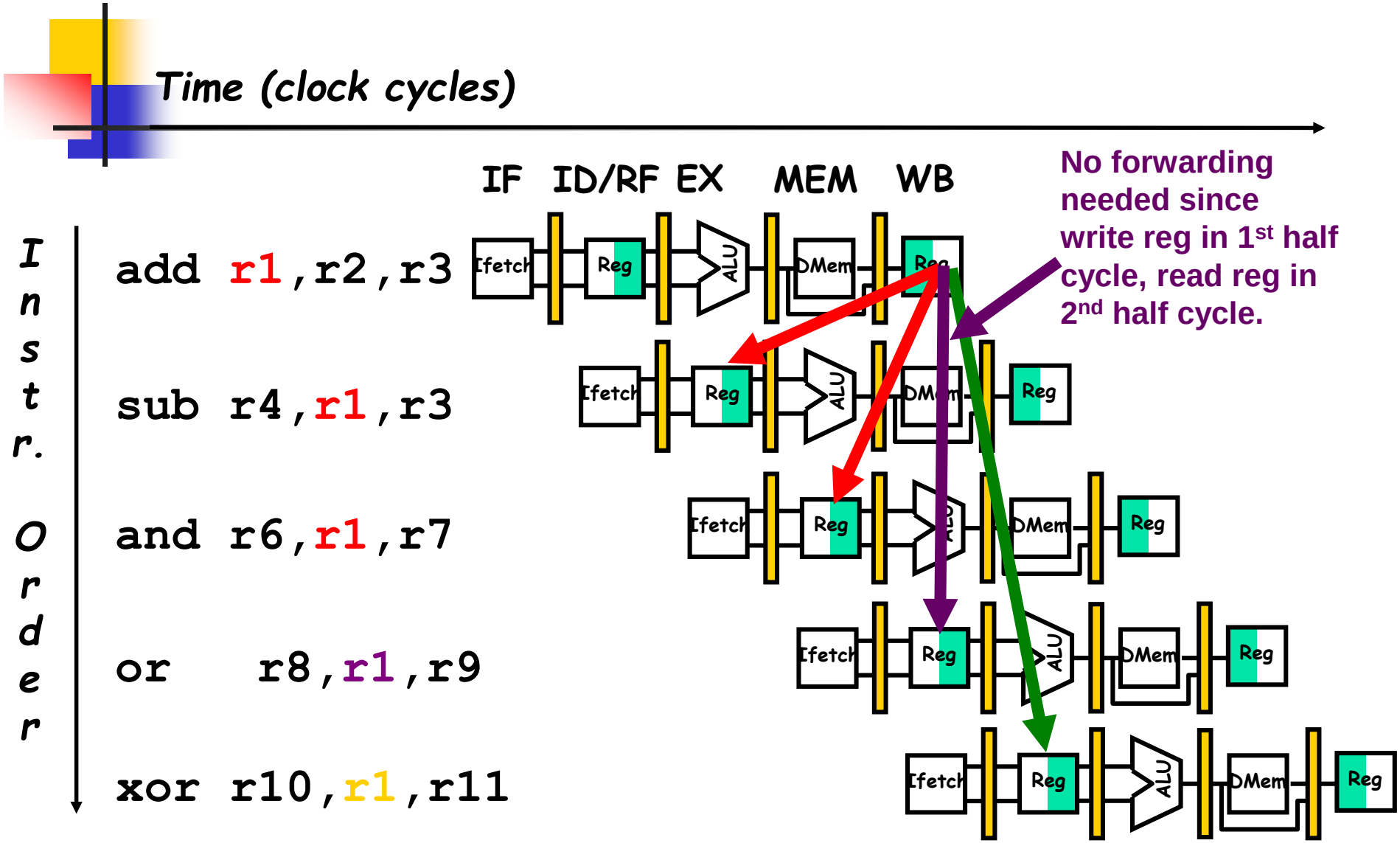
$$SpeedUp_A = Pipeline\ Depth/(1 + 0) \times (clock_{unpipe}/clock_{pipe})$$
$$= Pipeline\ Depth$$

$$SpeedUp_B = Pipeline\ Depth/(1 + 0.2 \times 1) \times (clock_{unpipe}/(clock_{unpipe} / 1.05)$$
$$= (Pipeline\ Depth/1.20) \times 1.05 \quad \{105/120 = 7/8\}$$
$$= 0.875 \times Pipeline\ Depth$$

$$SpeedUp_A / SpeedUp_B = Pipeline\ Depth/(0.875 \times Pipeline\ Depth) = 1.14$$

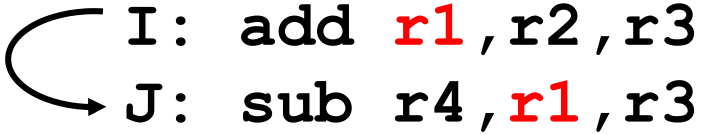
- Machine A is 1.14 times faster

Data Hazard on Register R1 (If No Forwarding)



Three Generic Data Hazards

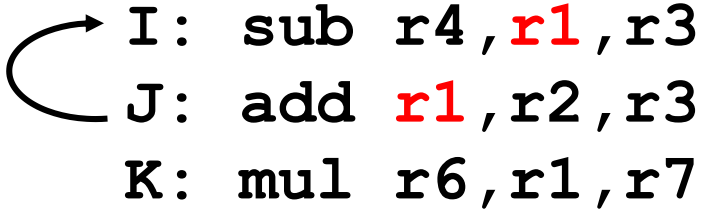
- **Read After Write (RAW)**
  $Instr_J$ tries to read operand before $Instr_I$ writes it



- **Caused by a "(True) Dependence" (in compiler nomenclature). This hazard results from an actual need for communicating a new data value.**

Three Generic Data Hazards

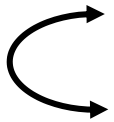
- **Write After Read (WAR)**
  $Instr_J$ writes operand *before* $Instr_I$ reads it



- **Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".**
- **Cannot happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Register reads are always in stage 2, and**
  - **Register writes are always in stage 5**

Three Generic Data Hazards

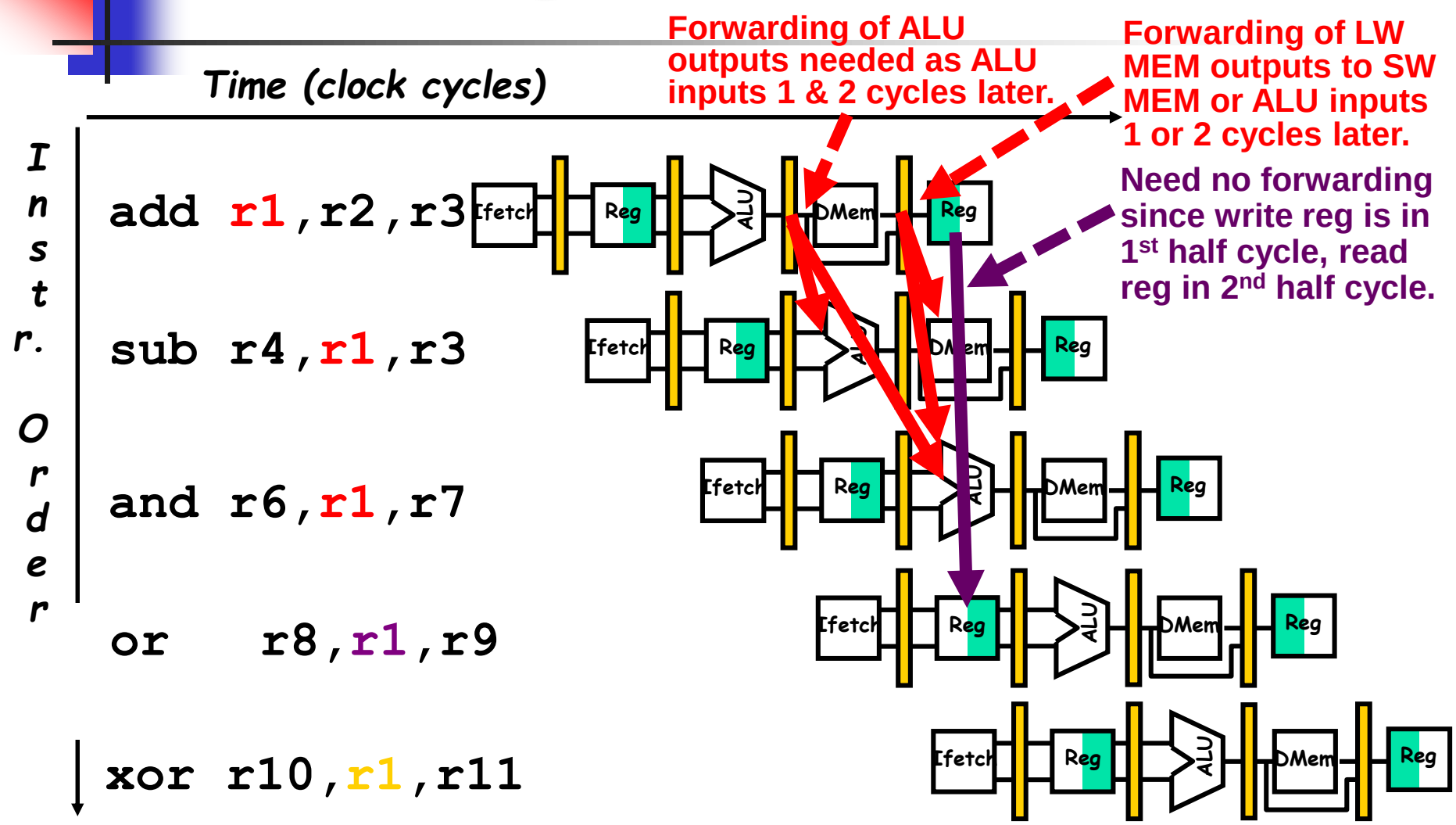
- **Write After Write (WAW)**
  $Instr_J$ writes operand *before* $Instr_I$ writes it.



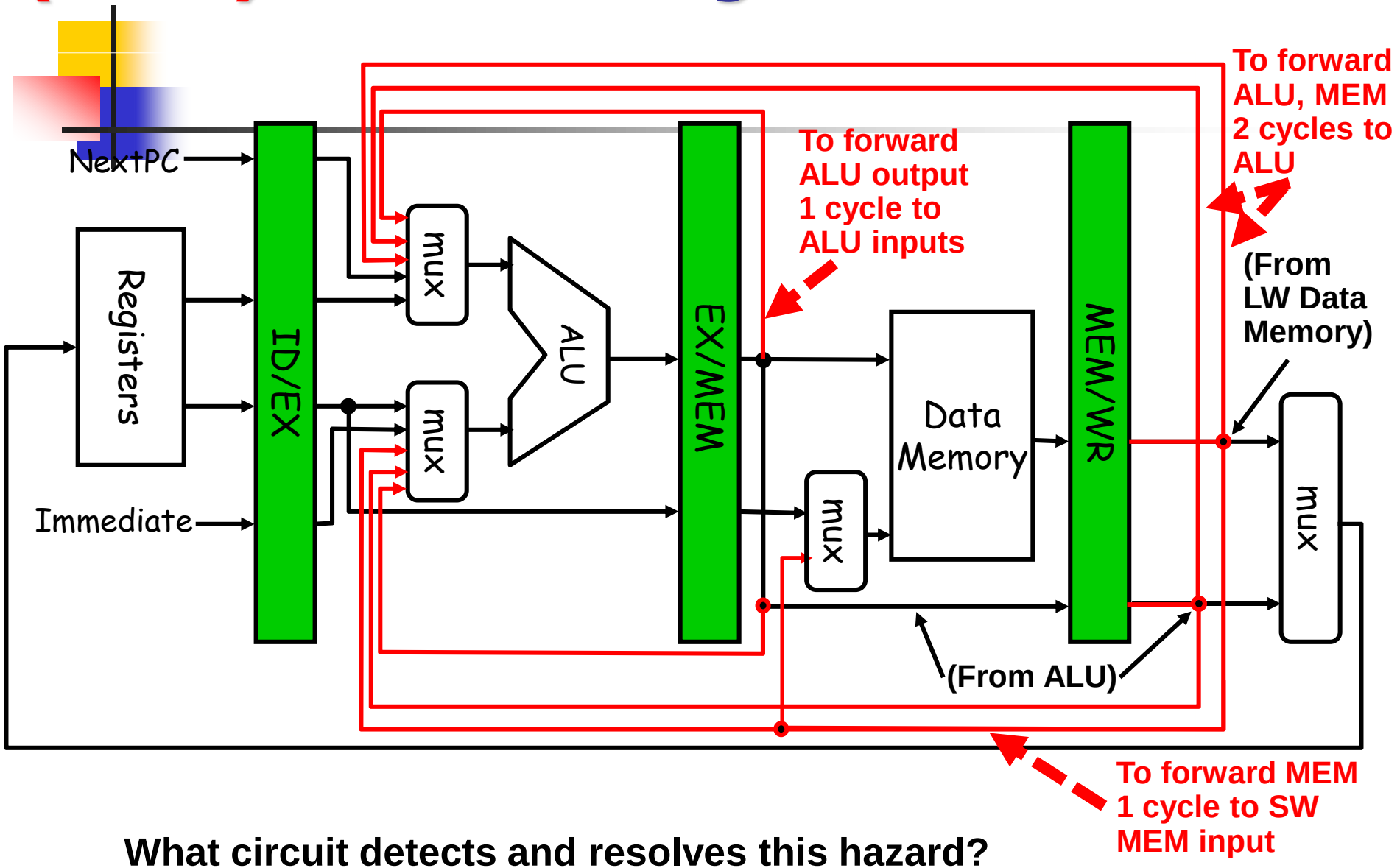
```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- **Called an "output dependence" by compiler writers**
  This also results from the reuse of name "r1".
- **Cannot happen in MIPS 5 stage pipeline because:**
  - **All instructions take 5 stages, and**
  - **Register writes are always in stage 5**
- **Will see WAR and WAW in more complicated pipes**

Forwarding to Avoid Data Hazard

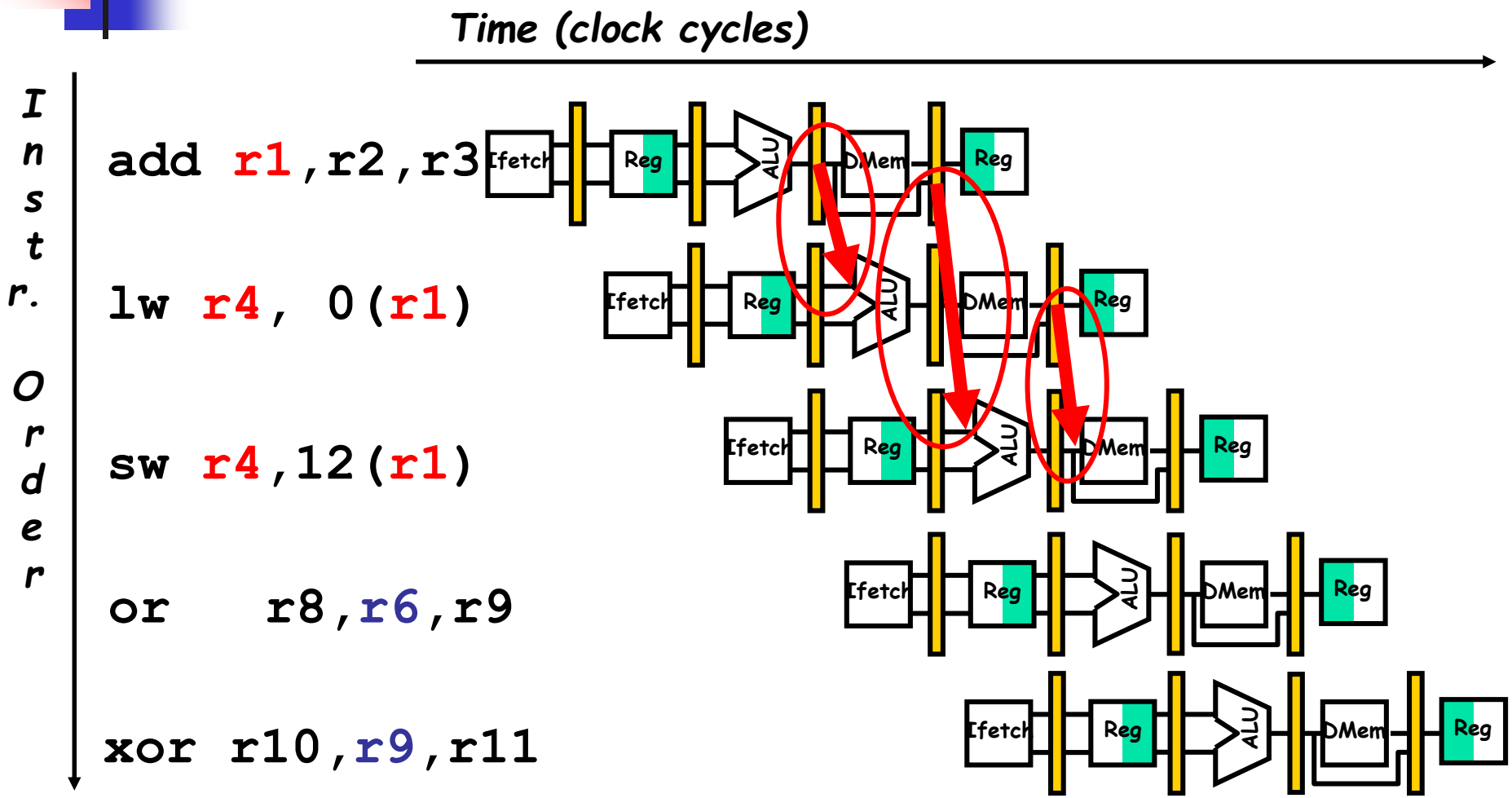


HW Datapath Changes (in red) for Forwarding

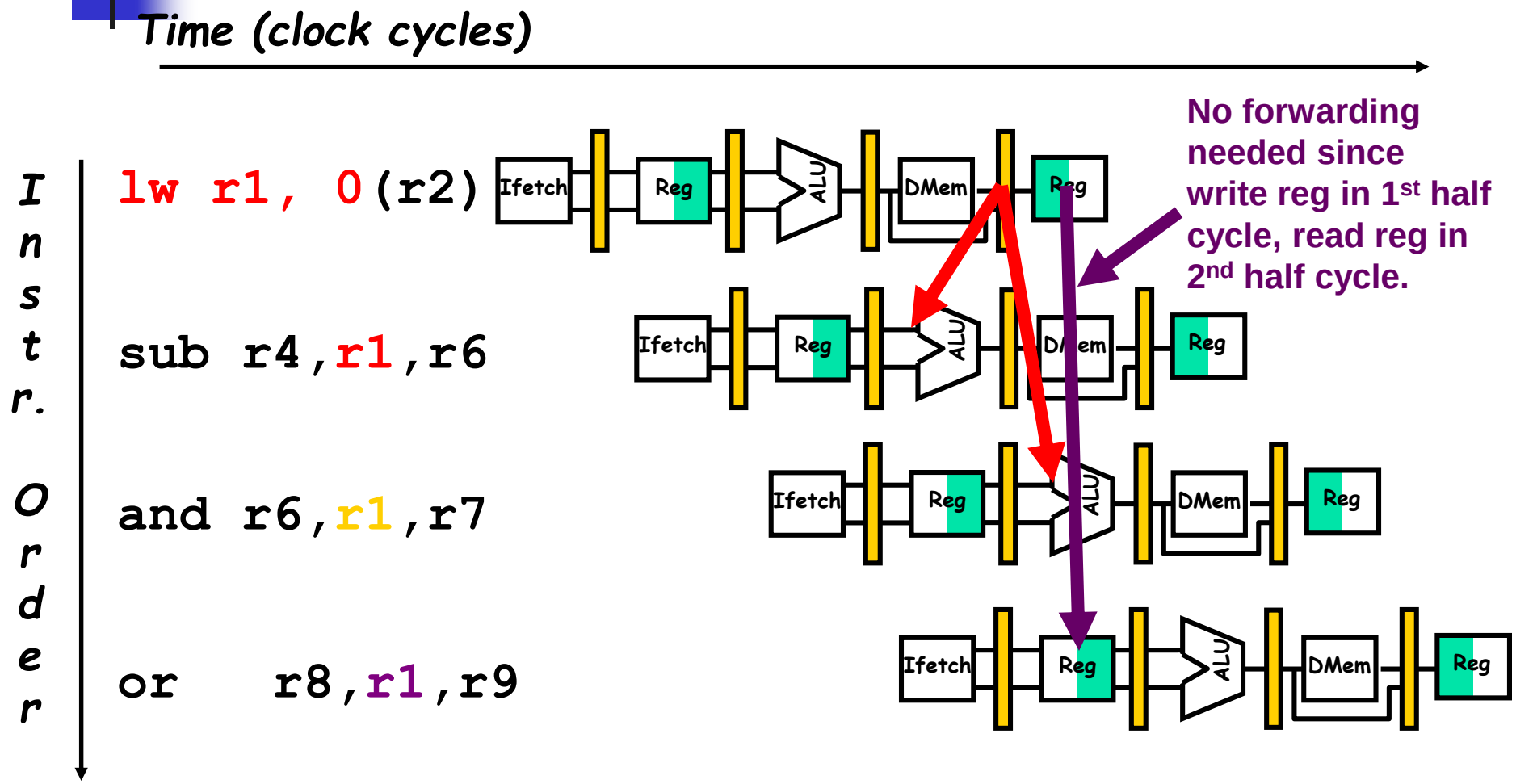


What circuit detects and resolves this hazard?

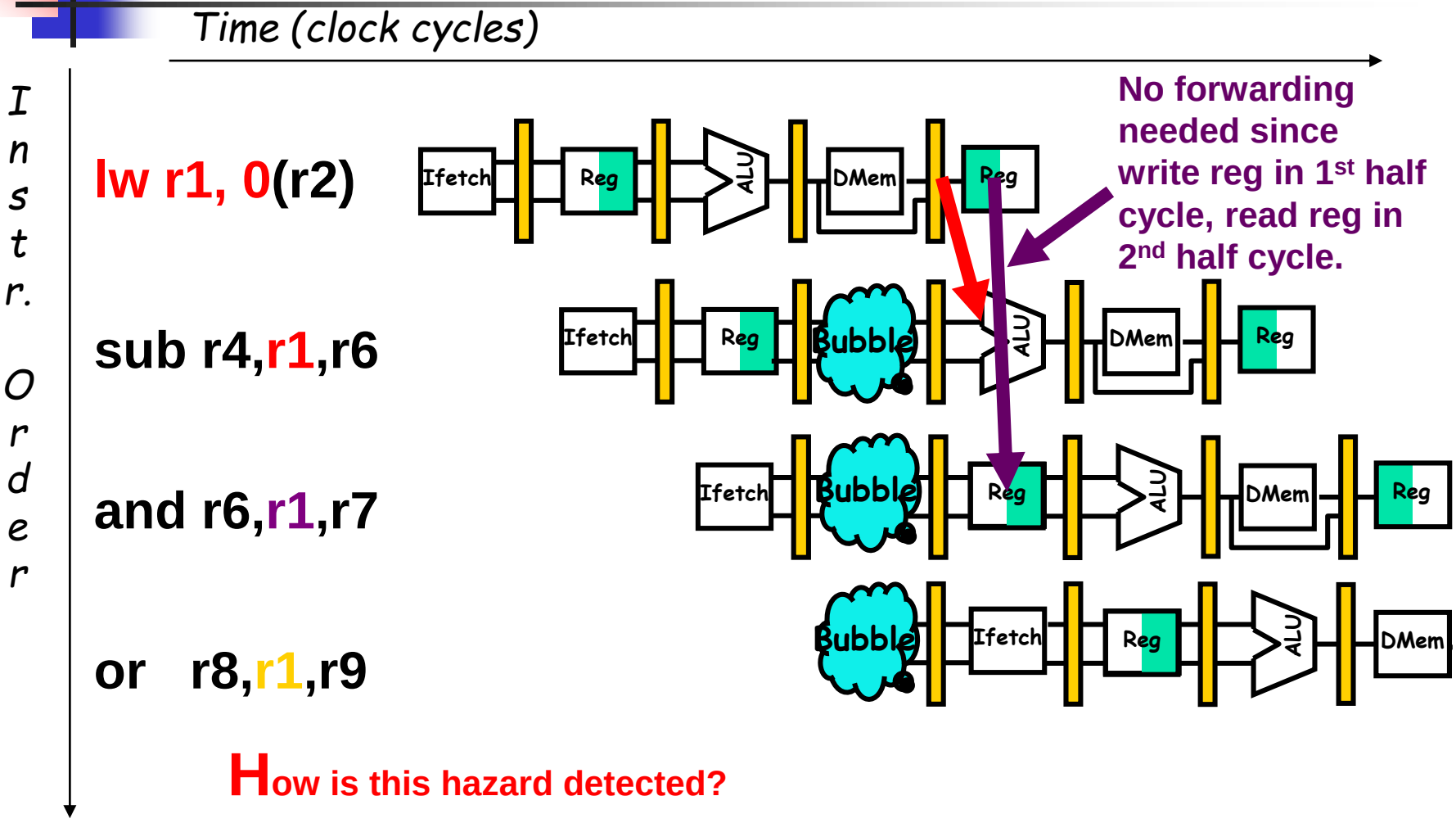
Forwarding Avoids ALU-ALU & LW-SW Data Hazards



LW-ALU Data Hazard Even with Forwarding



Data Hazard Even with Forwarding



Software Scheduling to Avoid Load Hazards

Try producing fast code with no stalls for

```
a = b + c;
d = e – f;
```

assuming a, b, c, d ,e, and f are in memory.

Slow code:

```
          LW    Rb,b
          LW    Rc,c
Stall ==> ADD   Ra,Rb,Rc
          SW    a,Ra
          LW    Re,e
          LW    Rf,f
Stall ==> SUB   Rd,Re,Rf
          SW    d,Rd
```

Fast code (no stalls):

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

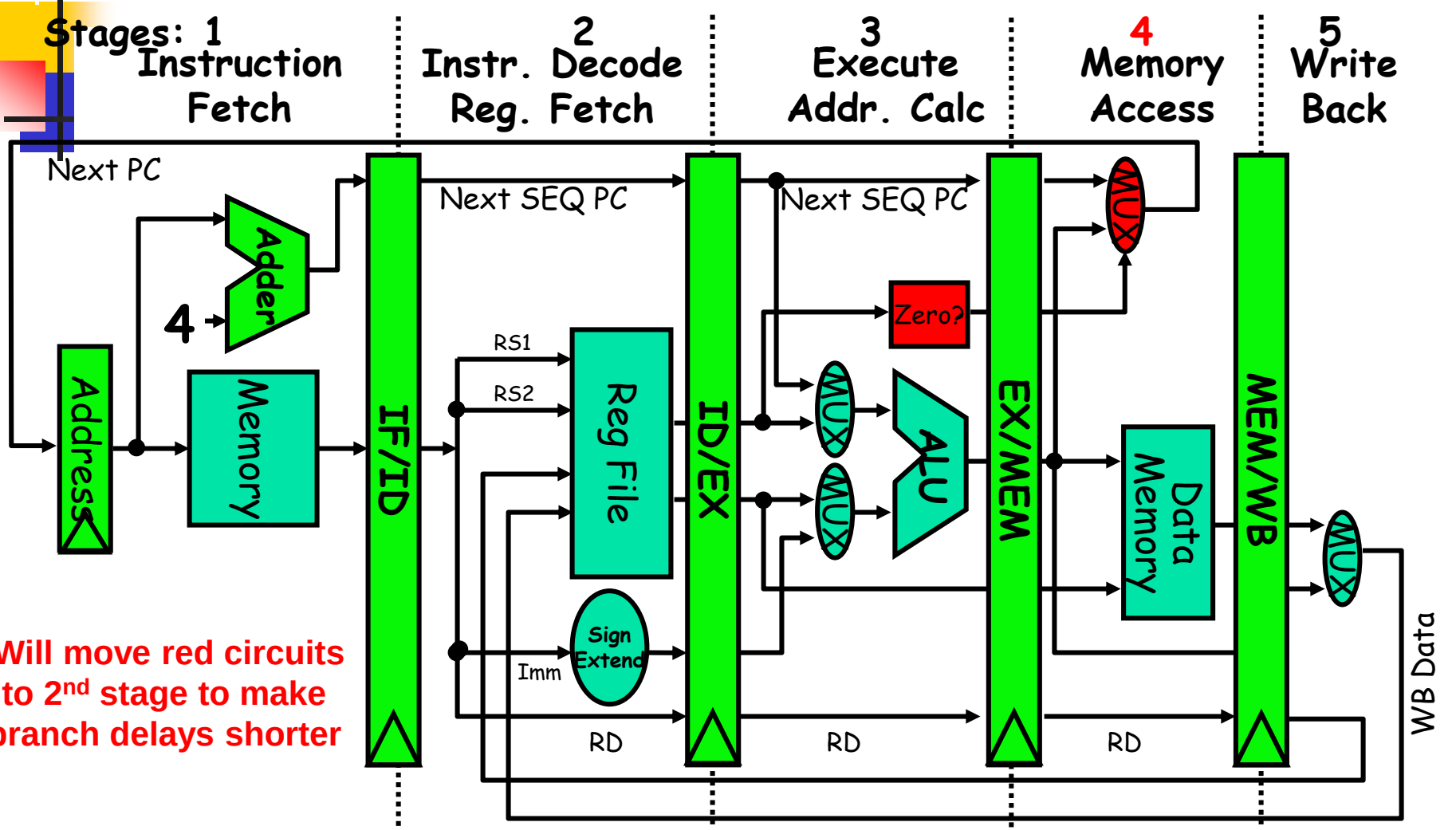
Compiler optimizes for performance. Hardware checks for safety.

Important technique !

Outline

- MIPS – An ISA for Pipelining
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion

5-Stage MIPS Datapath (has pipeline latches)



- Old simple design put branch completion in stage 4 (Mem)

Control Hazard on Branch - Three Cycle Stall (Caused if Decide Branches in 4th Stage)

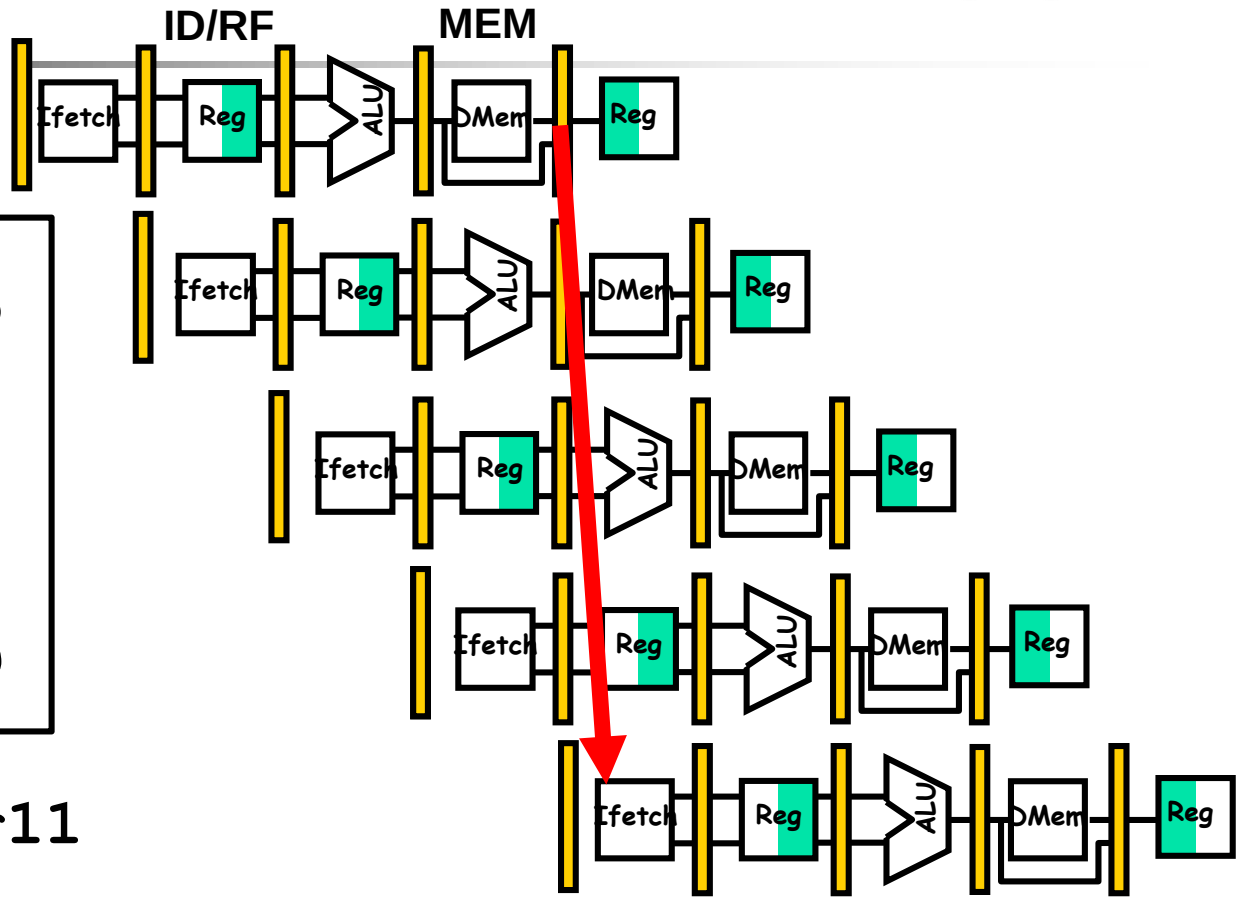
```
10: beq r1,r3,34

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

34: xor r10,r1,r11
```



What can be done with the 3 instructions between beq & xor?

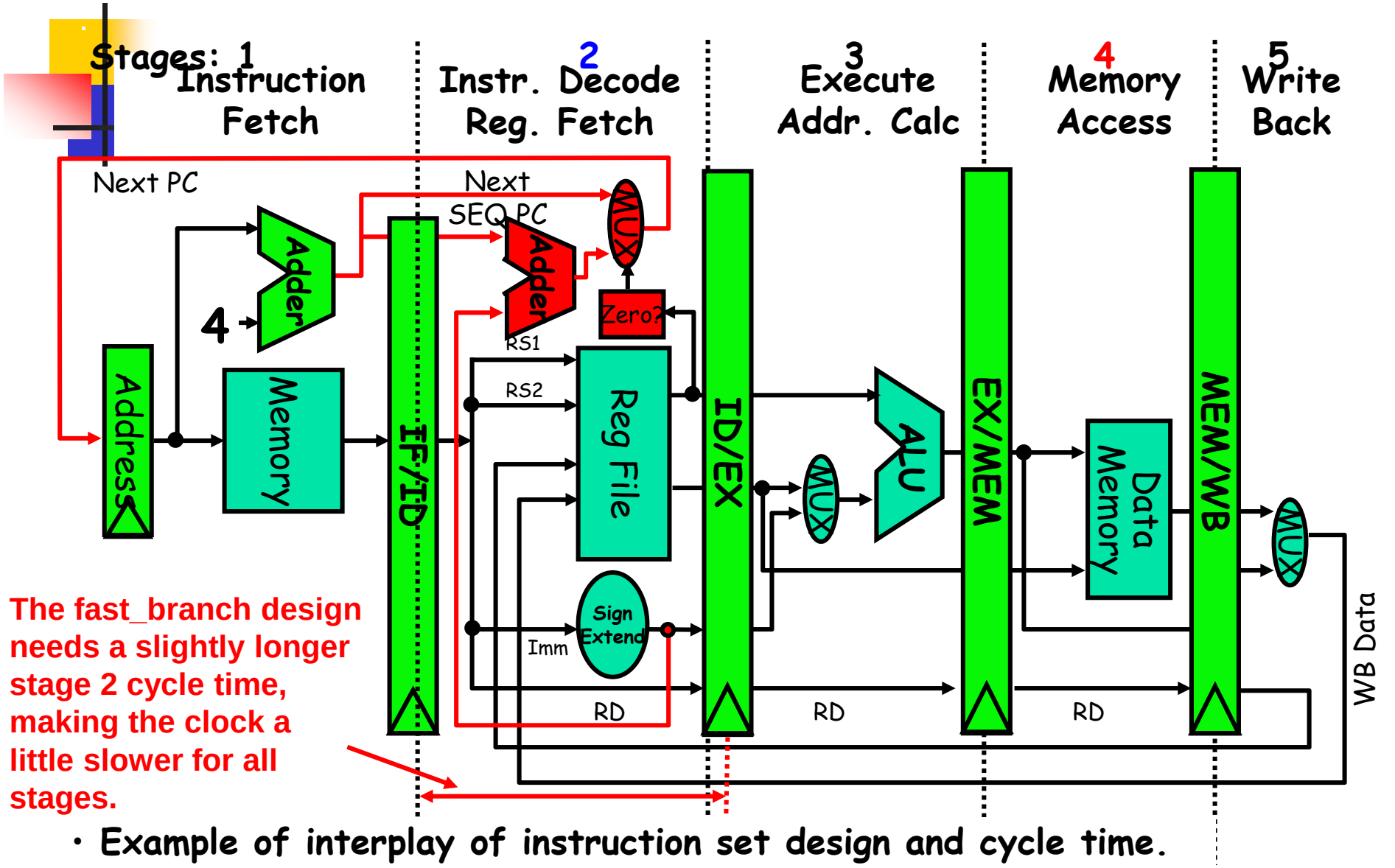
Code between beq&xor must not start until know beq not branch => 3 stalls

Adding 3 cycle stall after every branch (1/7 of instructions) => CPI += 3/7. BAD!

Branch Stall Impact if Commit in Stage 4

- If CPI = 1 and 15% of instructions are branches,
  Stall 3 cycles => new CPI = 1.45 (1+3*.15) Too much!
- Two-part solution:
  - Determine sooner whether branch taken or not, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- Original 1986 MIPS Solution:
  - Move zero_test to ID/RF (Inst Decode & Register Fetch) stage(2) (4=MEM)
  - Add extra adder to calculate new PC (Program Counter) in ID/RF stage
  - Result is 1 clock cycle penalty for branch versus 3 when decided in MEM

New Pipelined MIPS Datapath: Faster Branch



The fast_branch design needs a slightly longer stage 2 cycle time, making the clock a little slower for all stages.

- Example of interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clearly known

#2: Predict Branch Not Taken – Easy Solution

- **Execute the next instructions in sequence**
- **PC+4 already calculated, so use it to get next instruction**
- **Nullify bad instructions in pipeline if branch is actually taken**
- **Nullify easier since pipeline state updates are late (MEM, WB)**
- **47% MIPS branches not taken on average**

Four Branch Hazard Alternatives

#3: Predict Branch Taken

- **53% MIPS branches taken on average**
- **But have not calculated branch target address in MIPS**
  - **MIPS still incurs 1 cycle branch penalty**
  - **Some other CPUs: branch target known before outcome**

Last of Four Branch Hazard Alternatives

#4: Delayed Branch (Used Only in 1st MIPS "Killer Micro")

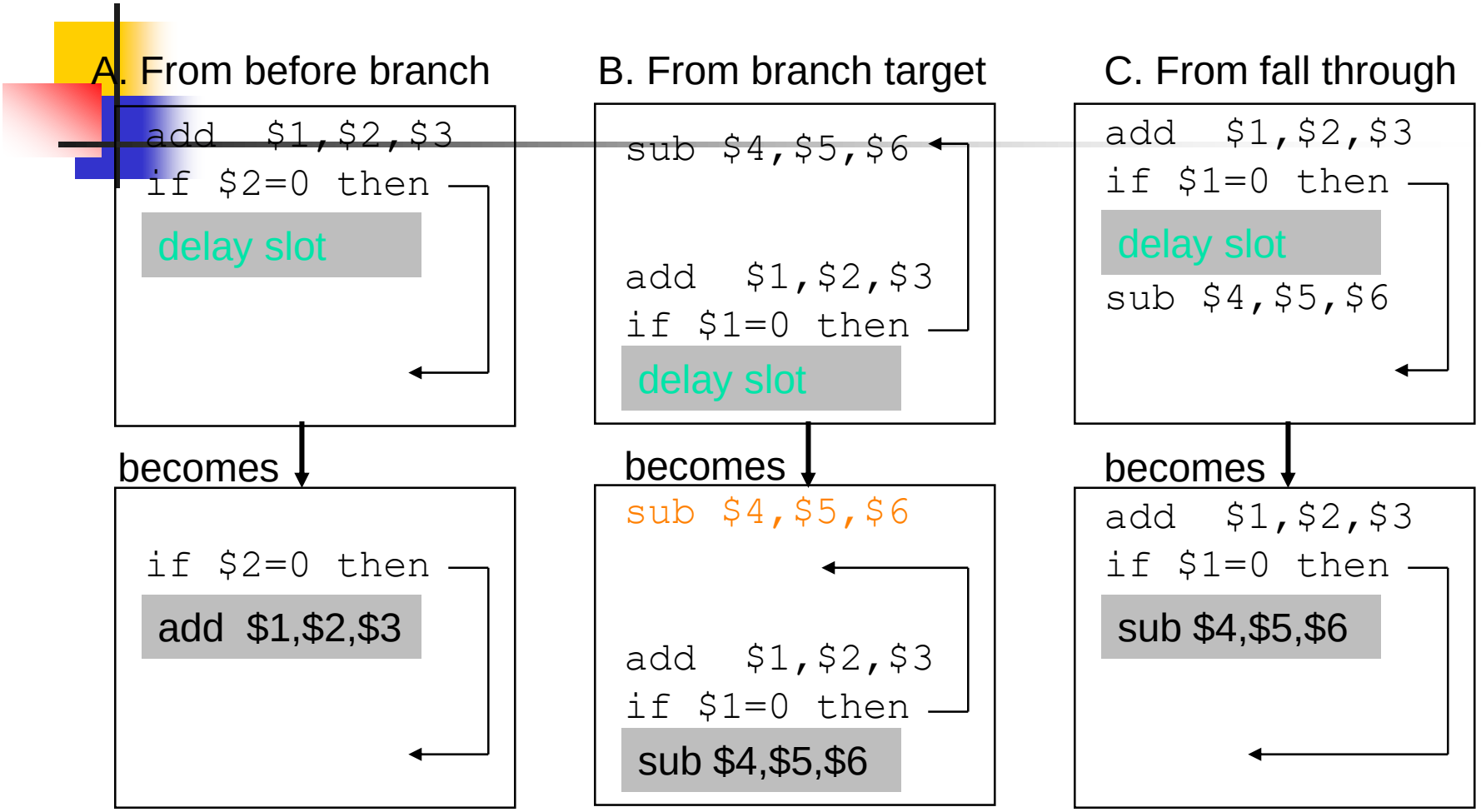
- **Define branch to take place AFTER a following instruction**

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
branch target if taken
```

Branch delay of length *n*

- **1 slot delay allows proper decision and branch target address in 5 stage pipeline**
- **MIPS 1st used this (Later versions of MIPS did not; pipeline deeper)**

Scheduling Branch Delay Slots



- **A is the best choice, fills delay slot & reduces instruction count (IC)**
- **In B, the `sub` instruction may need to be copied, increasing IC**
- **In B and C, must be okay to execute an extra `sub` when branch fails**

Delayed Branch Not Used in Modern CPUs

- **Compiler effectiveness 1/2 for single branch delay slot:**
  - **Fills about 60% of branch delay slots**
  - **About 80% of instructions executed in branch delay slots useful in computation**
  - **Only half of (60% x 80%) slots usefully filled; cannot fill 2 or more**

Delayed Branch Not Used in Modern CPUs

- **Delayed Branch downside: As processor designs use deeper pipelines and multiple issue, the branch delay grows and needs many more delay slots**
  - **Delayed branching soon lost effectiveness and popularity compared to more expensive but more flexible dynamic approaches**
  - **Growth in available transistors soon permitted dynamic approaches that keep records of branch locations, taken/not-taken decisions, and target addresses**
  - **Multi-issue 2 => 3 delay slots needed, 4 => 7 slots, 8 => 15 slots**

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume 4% unconditional jump, 10% conditional branch-taken, 6% conditional branch-not-taken, base CPI = 1.

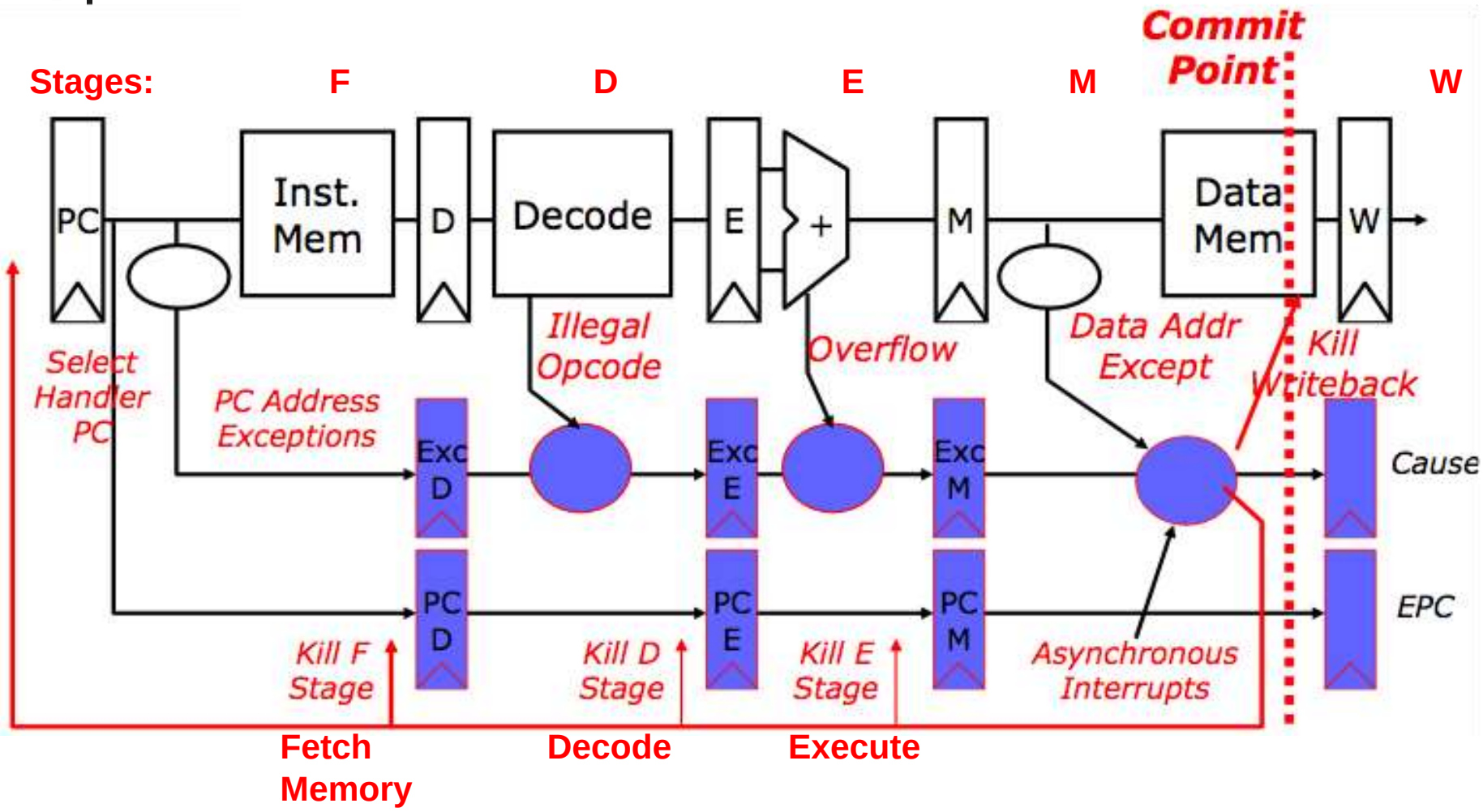
Scheduling Scheme	*Branch penalty*	*CPI*	*speedup vs. no-pipe*	*5 cycles*	*speedup vs. stall_pipeline*
Stall pipeline (Stage 4)	3	1.60	3.1		1.00
Predict taken (Stage 2)	1	1.20	4.2		1.33
Predict not taken (St.2)	1	1.14	4.4		1.40
Delayed branch (Stg 2)	0.5	1.10	4.5		1.45

(Sample calculations) 1.60=1+3(4+10+6)% (4.5=5/1.10) (1.45=1.6/1.1)
1.20=1+1(4+10+6)% (to calculate taken target)
1.14=1+1(4+10)% (refetch for jump, taken-branch)

Another Problem with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution {caused by instructions executing}
  - Examples: divide by zero, undefined opcode
- **Interrupt**: Hardware signal to switch the processor to a new instruction stream {not directly caused by code}
  - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- **Precise Interrupt Problem:** Must seem as if the exception or interrupt appeared between 2 instructions (I_i and I_{i+1}) although several instructions were executing at the time
  - All instructions up to and including I_i are totally completed
  - No effect of any instruction after I_i is allowed to be saved
- After a precise interrupt, the interrupt (exception) handler either aborts the program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



Key observation: "Architected" states change only in memory (M) and register write (W) stages.

And In Conclusion: Control and Pipelining

- Quantify and summarize performance
  - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, single-point failure
- Control via State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers by stalling:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch or branch (taken/not-taken) prediction
- Exceptions and interrupts add complexity

Homework

- C.1