EI338: Computer Systems and Engineering
(Computer Architecture & Operating Systems)

Chentao Wu 吴晨涛

Associate Professor

Dept. of Computer Science and Engineering

Shanghai Jiao Tong University

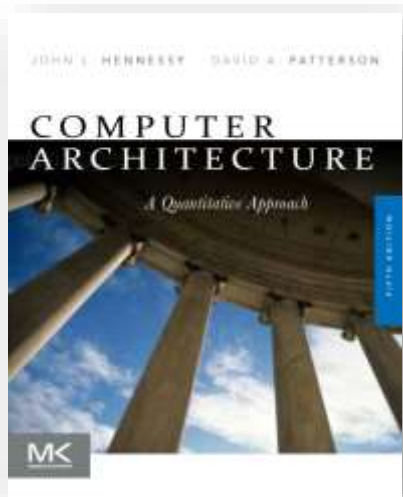SEIEE Building 3-513

wuct@cs.sjtu.edu.cn

Download Lectures

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

Appendix C

Pipelining

5 Steps of a (pre-pipelined) MIPS Datapath

Stages: 1
Instruction Fetch

2
Instr. Decode Reg. Fetch

3
Execute Addr. Calc

4
Memory Access

5
Write Back
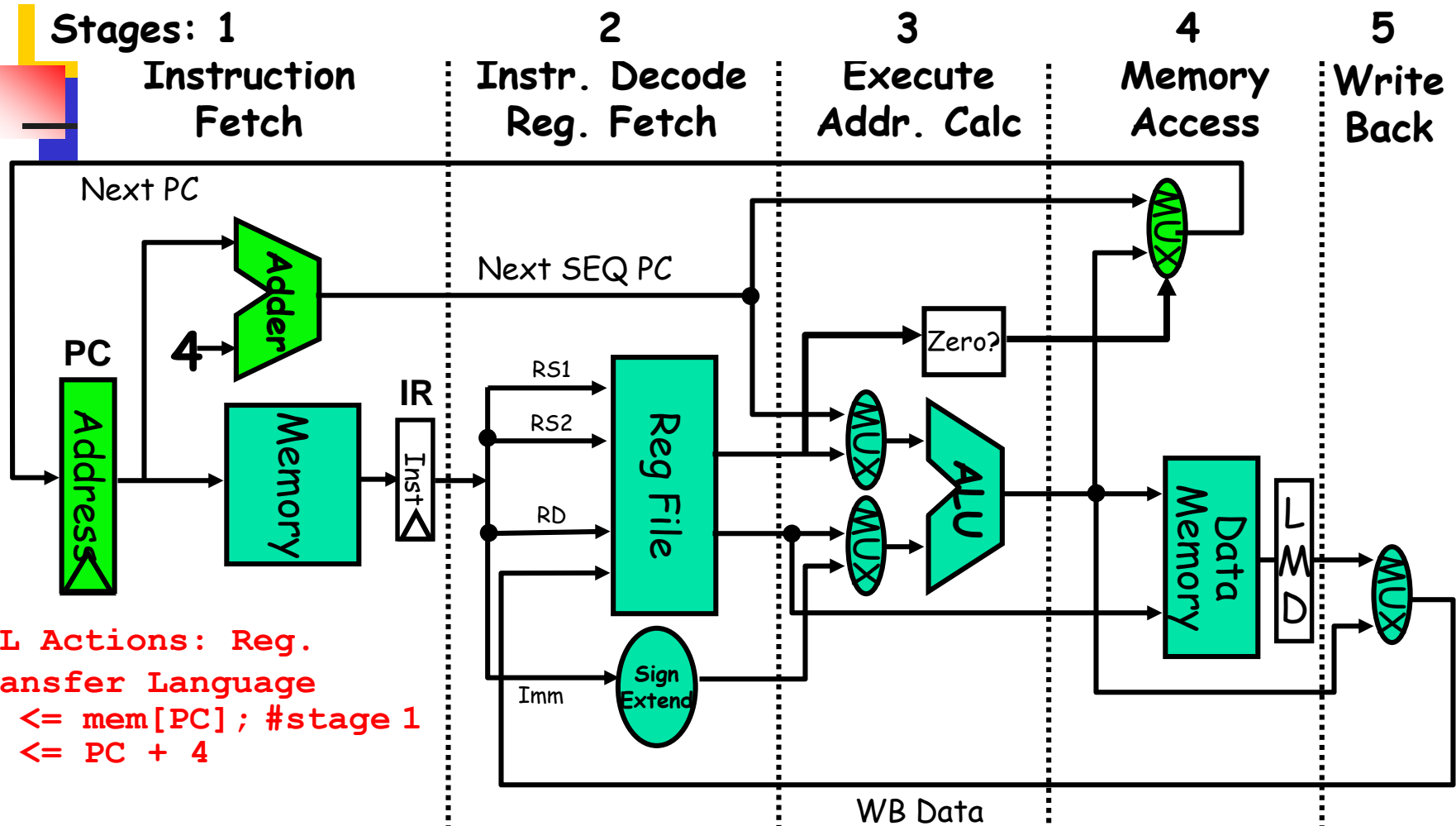
RTL Actions: Reg. Transfer Language
IR <= mem[PC]; #stage 1
PC <= PC + 4

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

5-Stage MIPS Datapath (has pipeline latches)

Stages: 1 Instruction Fetch | 2 Instr. Decode Reg. Fetch | 3 Execute Addr. Calc | 4 Memory Access | 5 Write Back

Next PC

Next SEQ PC

Next SEQ PC

4

Adder

Address

Memory

IF/ID

RS1

RS2

Reg File

ID/EX

Zero?

MUX

ALU

EX/MEM

Data Memory

MEM/WB

MUX

Sign Extend

Imm

RD

RD

RD

WB Data

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

A <= Reg[IR$_{rs}$]; #2
B <= Reg[IR$_{rt}$]

rslt <= A op$_{IRop}$ B
#3
WB <= rslt   #4

Reg[IR$_{rd}$] <= WB   #5
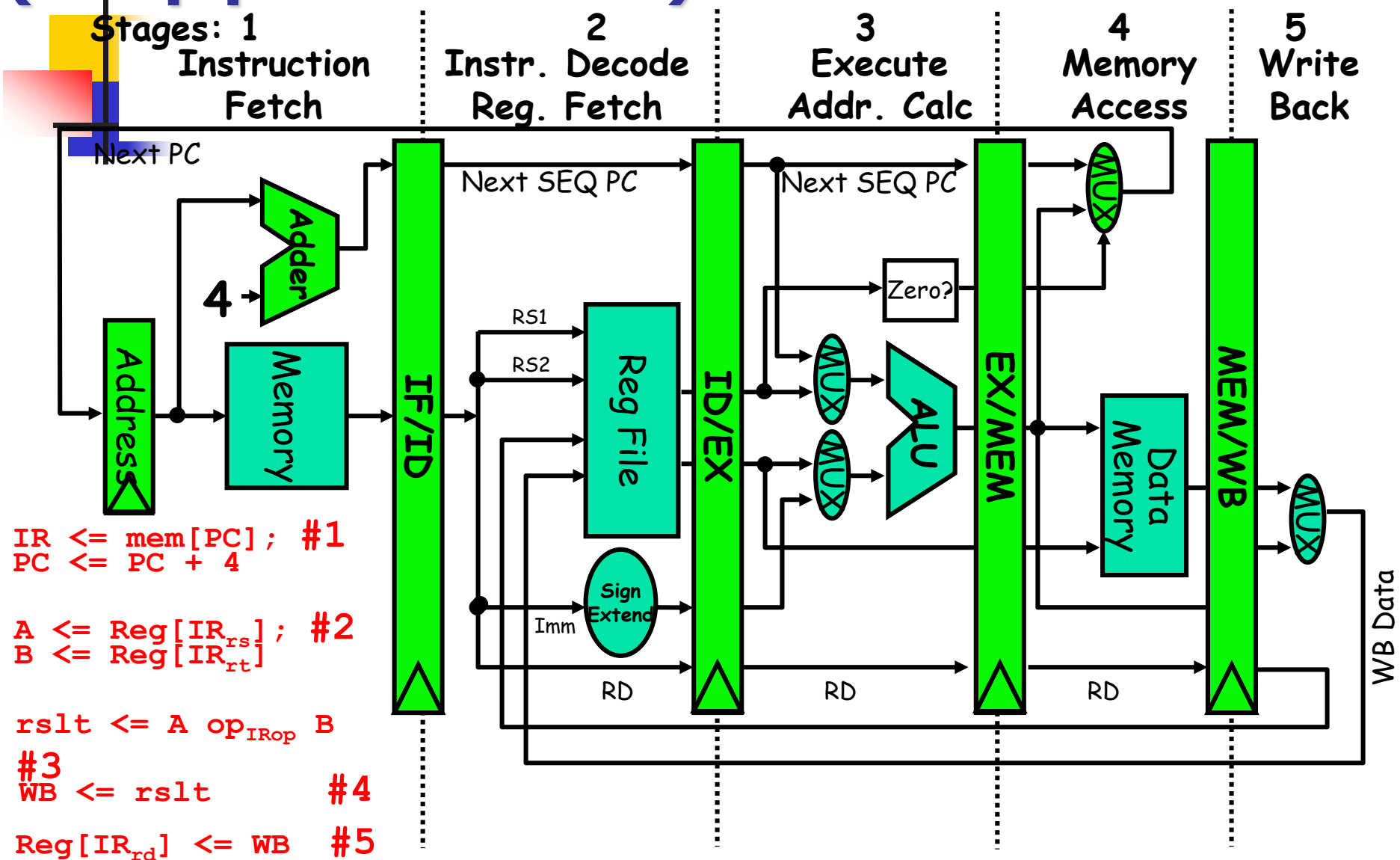
Instruction Set Processor Controller

IR <= mem[PC];
PC <= PC + 4

Ifetch

A <= Reg[IR$_{rs}$];
B <= Reg[IR$_{rt}$]

opFetch-DeCoDe

JAL

JR

ST

br

jmp

RR

RI

LD

if bop(A,B)
PC <= PC+IR$_{im}$

PC <= IR$_{jaddr}$

r <= A op$_{IRop}$ B

r <= A op$_{IRop}$ IR$_{im}$

r <= A + IR$_{im}$

WB <= r

WB <= r

WB <= Mem[r]

Reg[IR$_{rd}$] <= WB

Reg[IR$_{rd}$] <= WB

Reg[IR$_{rd}$] <= WB

5-Stage MIPS Datapath (has pipeline latches)

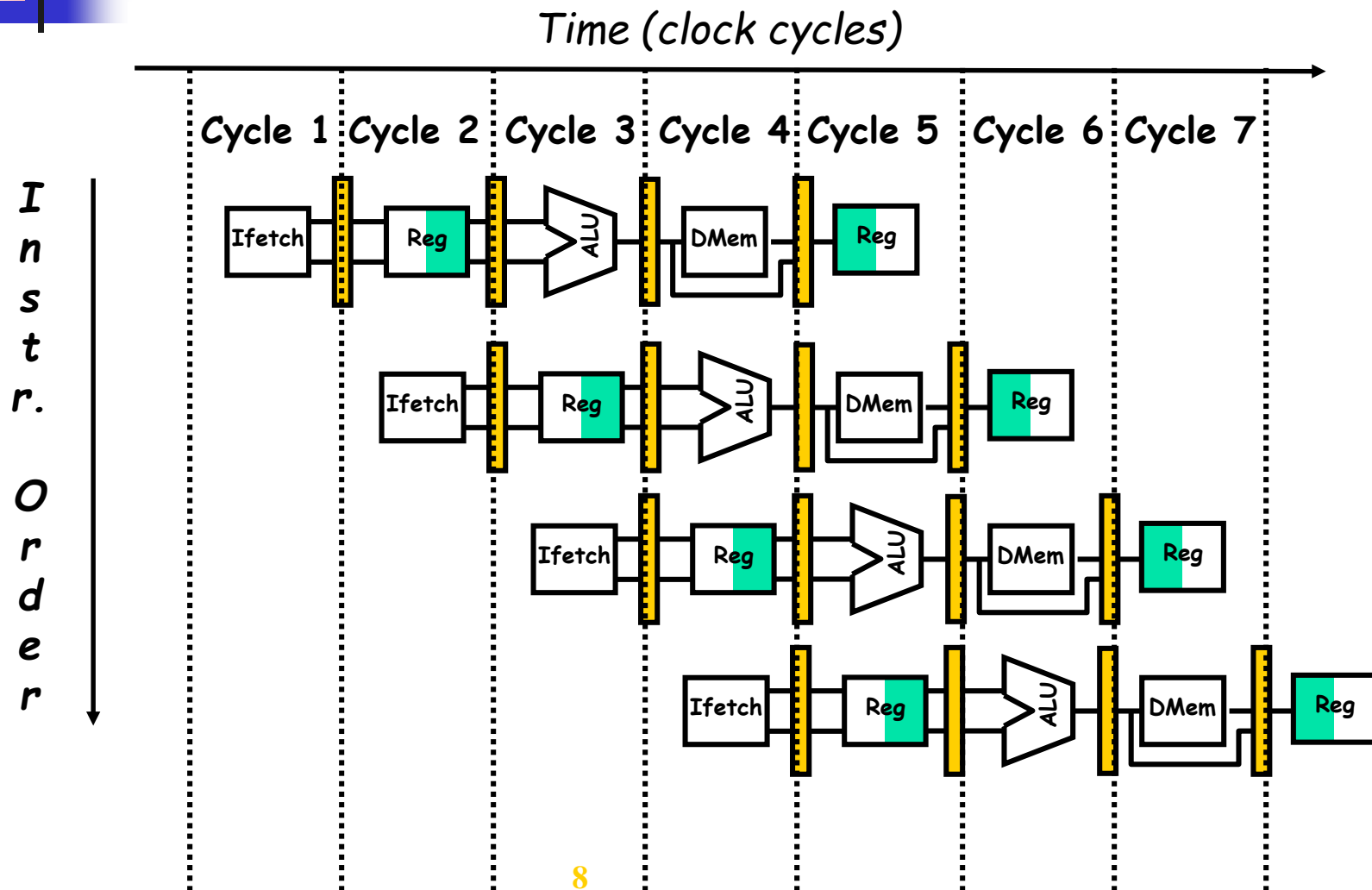| Stages: 1 Instruction Fetch | 2 Instr. Decode Reg. Fetch | 3 Execute Addr. Calc | 4 Memory Access | 5 Write Back |
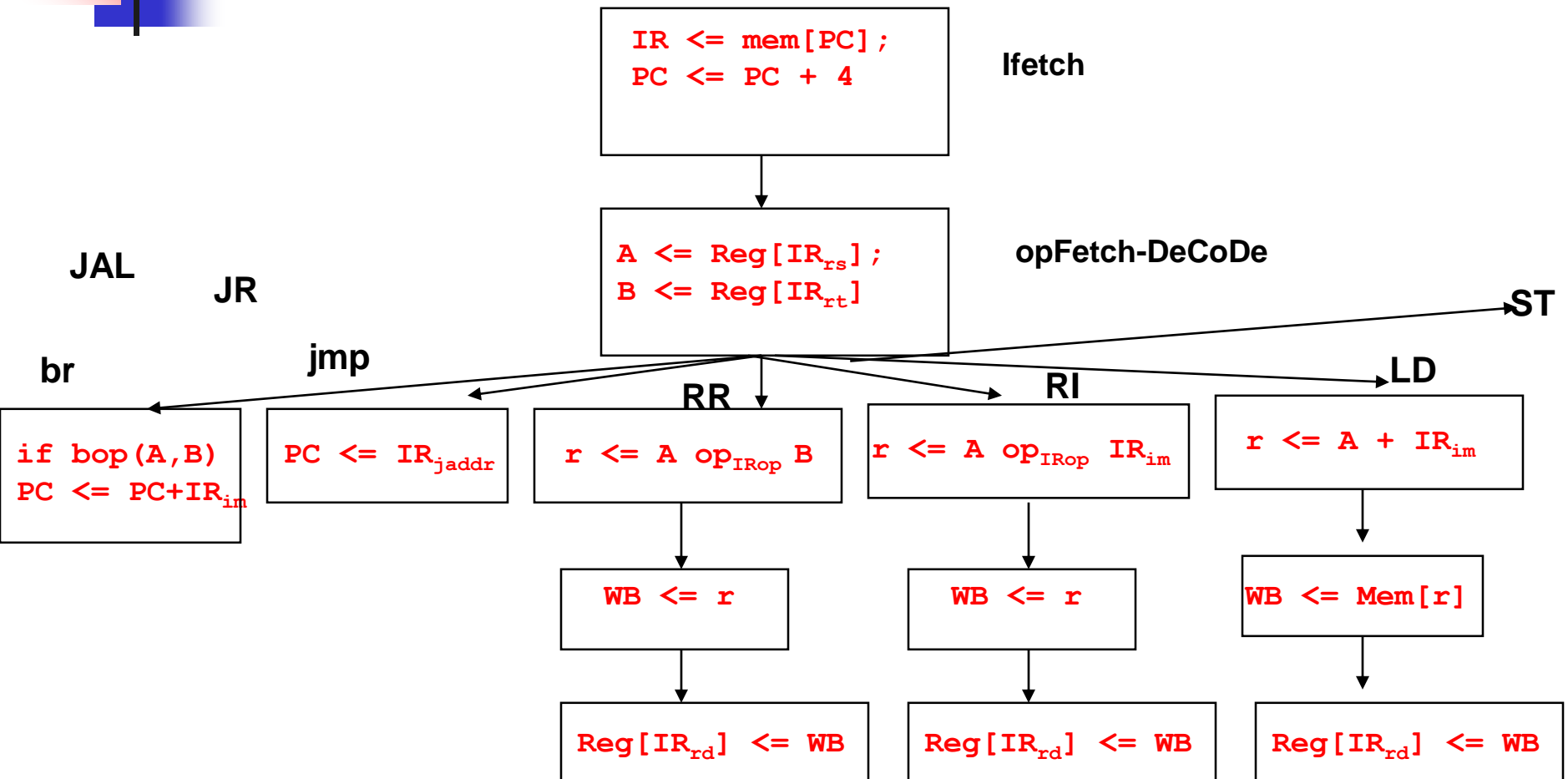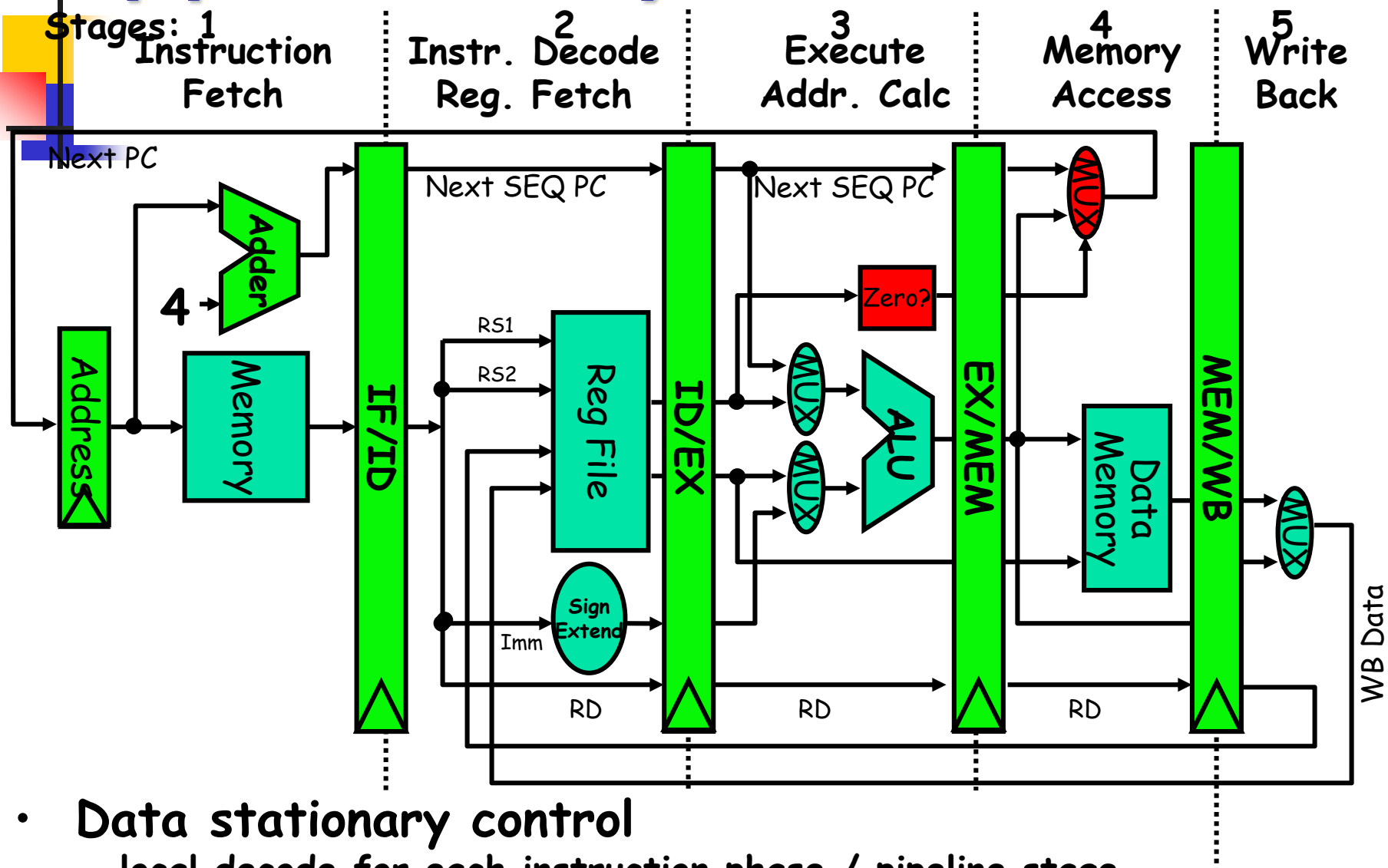
- **Data stationary control**
  - local decode for each instruction phase / pipeline stage

# Visualizing Pipelining

Time (clock cycles)

Pipelining is not quite that easy!

- **Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle**

    - <u>**Structural hazards**</u>**: HW cannot support this combination of instructions (having a single person to fold and put clothes away at same time)**

    - <u>**Data hazards**</u>**: Instruction depends on result of prior instruction still in the pipeline (having a missing sock in a later wash; cannot put away)**

    - <u>**Control hazards**</u>**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).**
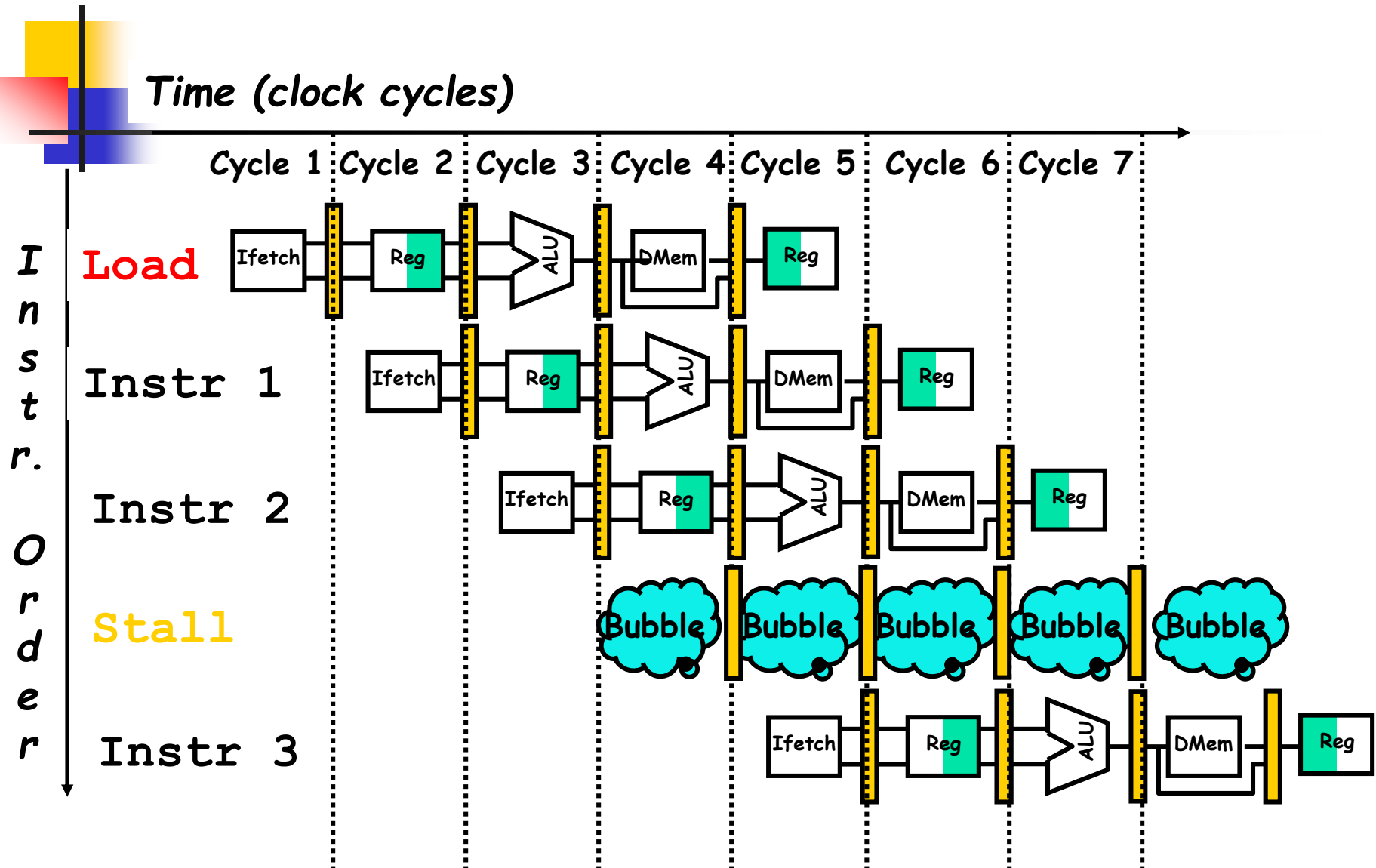
One Memory_Port / Structural_Hazards

Time (clock cycles)
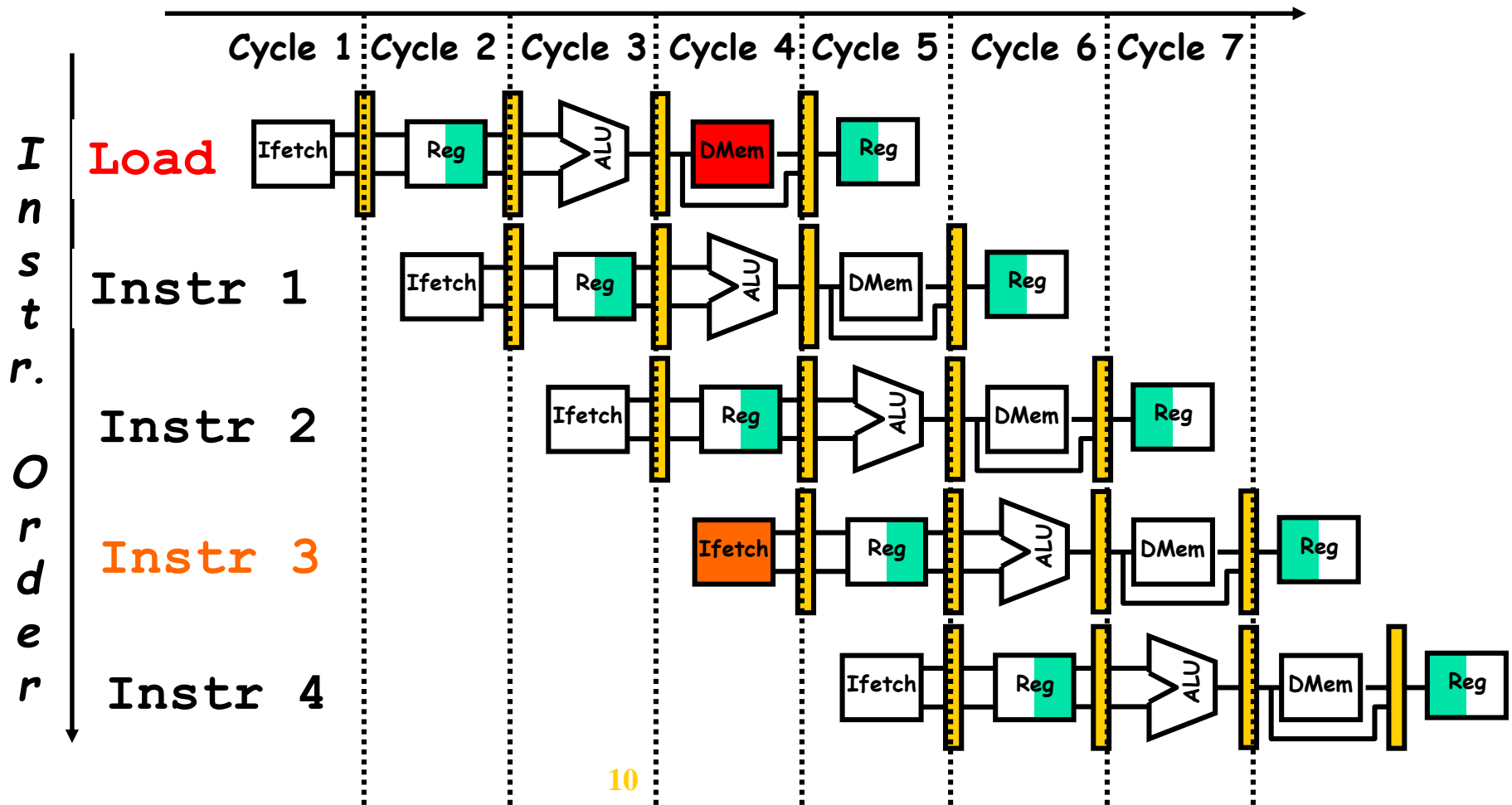
One Memory Port/Structural Hazards

Time (clock cycles)

How do you "bubble" the pipe?

Code SpeedUp Equation for Pipelining

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

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

For simple RISC pipeline, Ideal CPI = 1:

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

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**

$$\text{SpeedUp}_A = \text{Pipeline Depth}/(1 + 0) \times (\text{clock}_{unpipe}/\text{clock}_{pipe})$$
$$= \text{Pipeline Depth}$$

$$\text{SpeedUp}_B = \text{Pipeline Depth}/(1 + 0.2 \times 1) \times (\text{clock}_{unpipe}/(\text{clock}_{unpipe} / 1.05)$$

$$= (\text{Pipeline Depth}/1.20) \times 1.05 \qquad \{105/120 = 7/8\}$$
$$= 0.875 \times \text{Pipeline Depth}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipeline Depth}/(0.875 \times \text{Pipeline Depth}) = 1.14$$

- **Machine A is 1.14 times faster**

Data Hazard on Register R1 (If No Forwarding)

Time (clock cycles)

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.
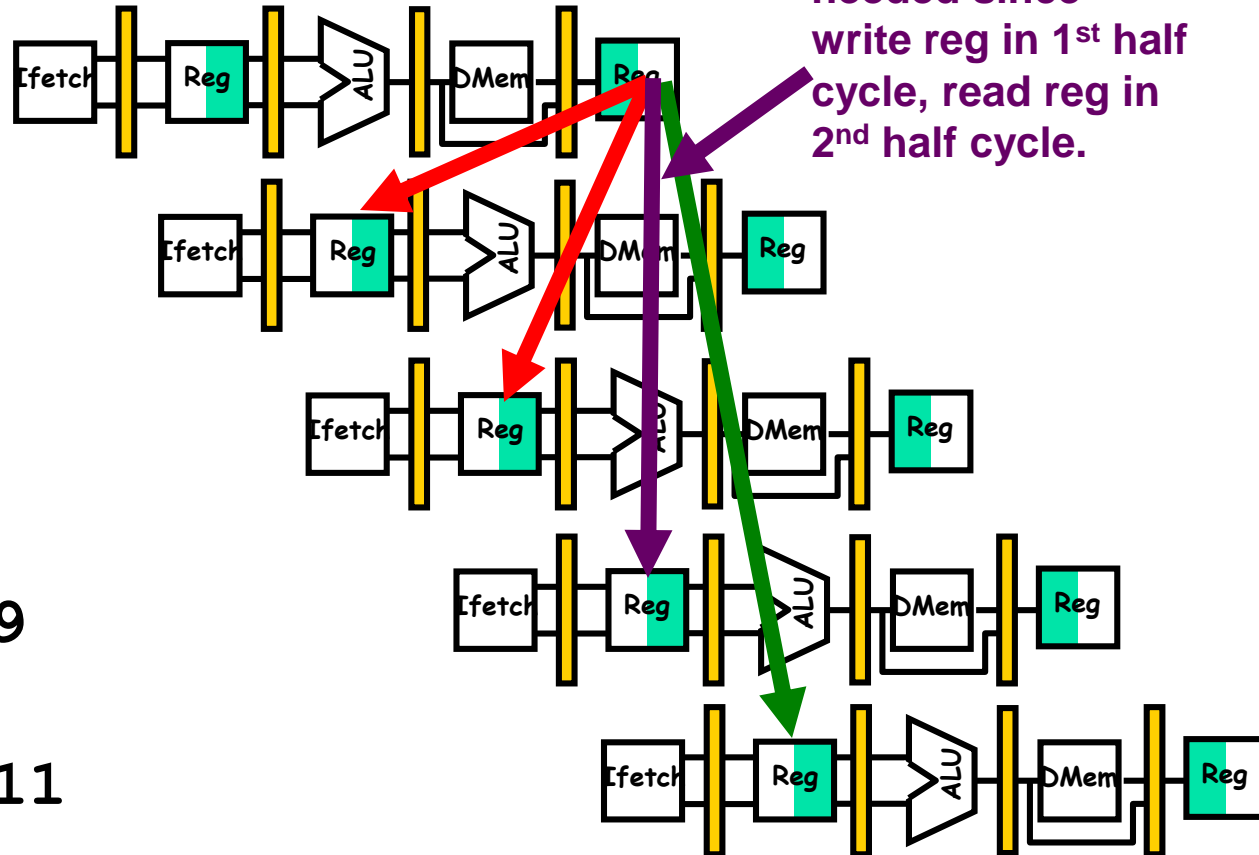
Instr. Order

```
add  r1,r2,r3

sub  r4,r1,r3

and  r6,r1,r7

or   r8,r1,r9

xor r10,r1,r11
```

Three Generic Data Hazards

- **Read After Write (RAW)**
  Instr$_J$ tries to read operand before Instr$_I$ writes it

  ```
  I: add r1,r2,r3
  J: sub r4,r1,r3
  ```

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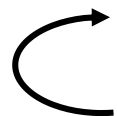
Three Generic Data Hazards

- **Write After Read (WAR)**
  Instr$_J$ writes operand *before* Instr$_I$ reads it

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

- **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

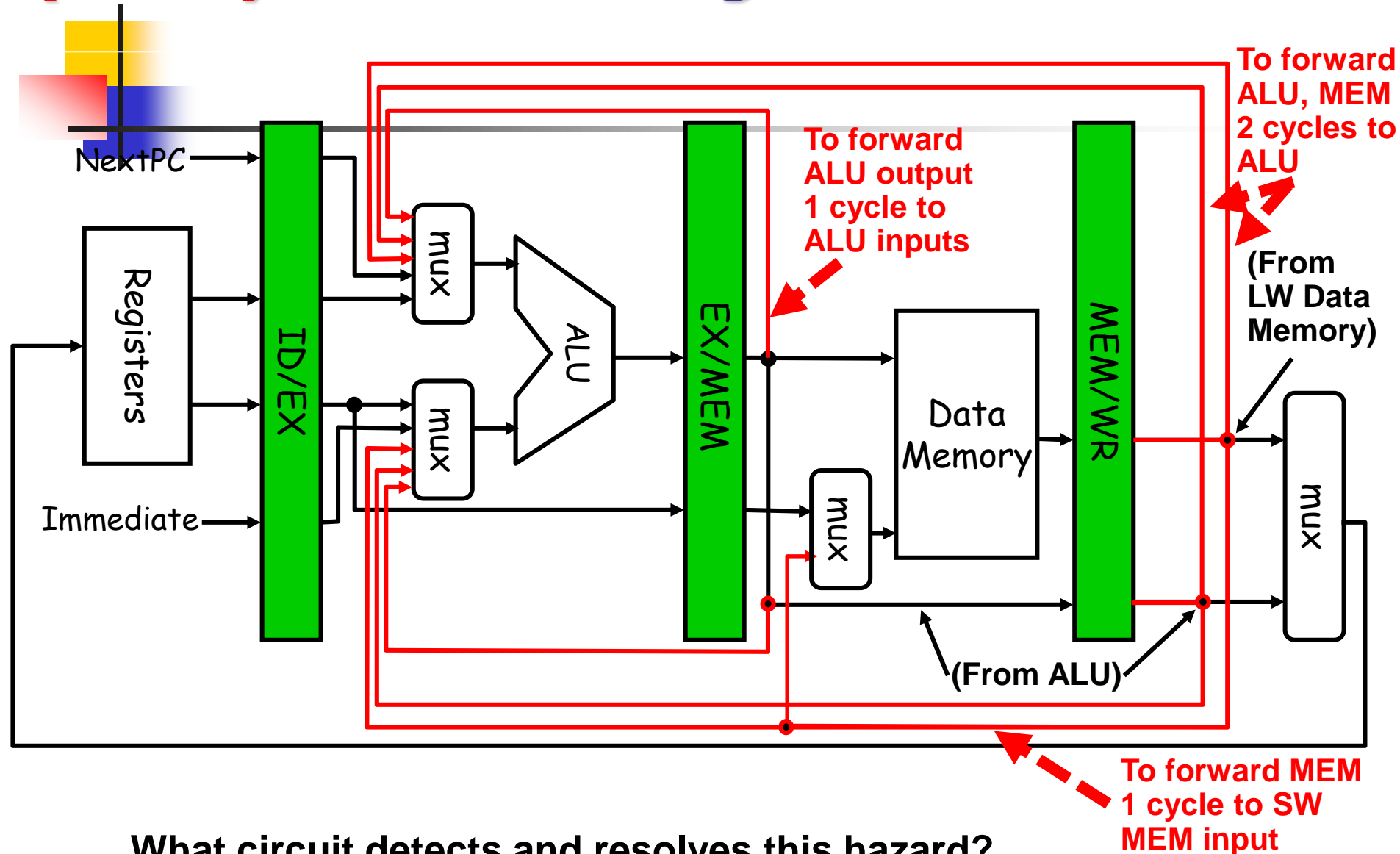**Forwarding of ALU outputs needed as ALU inputs 1 & 2 cycles later.**

Forwarding of LW MEM outputs to SW MEM or ALU inputs 1 or 2 cycles later.

Need no forwarding since write reg is in 1st half cycle, read reg in 2nd half cycle.

Time (clock cycles)

Instr. Order

add **r1**,r2,r3

sub r4,**r1**,r3

and r6,**r1**,r7

or   r8,**r1**,r9

xor r10,**r1**,r11

To forward ALU output 1 cycle to ALU inputs

To forward ALU, MEM 2 cycles to ALU

(From LW Data Memory)

(From ALU)

To forward MEM 1 cycle to SW MEM input

NextPC

Registers

Immediate

ID/EX

mux

mux

ALU

EX/MEM

mux

Data Memory

MEM/WR

mux

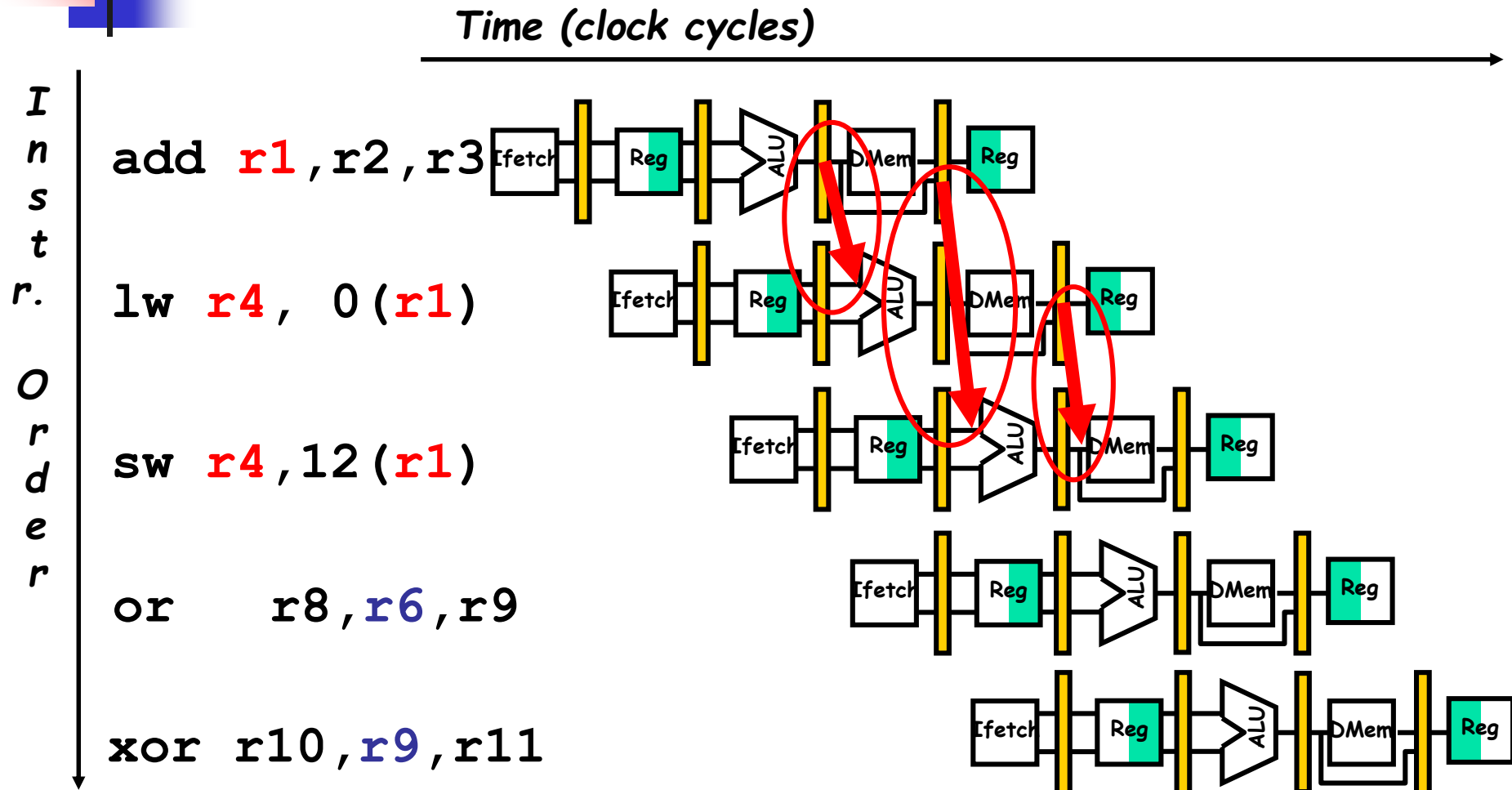**What circuit detects and resolves this hazard?**

Forwarding Avoids Alu-Alu & Lw-Sw Data Hazards

Time (clock cycles)

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.

Instr. Order

`lw r1, 0(r2)`   Ifetch | Reg | ALU | DMem | Reg

`sub r4,r1,r6`   Ifetch | Reg | ALU | DMem | Reg

`and r6,r1,r7`   Ifetch | Reg | ALU | DMem | Reg

`or   r8,r1,r9`   Ifetch | Reg | ALU | DMem | Reg
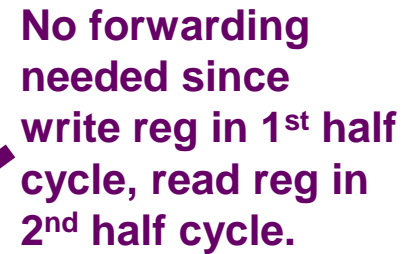
Data Hazard Even with Forwarding

Time (clock cycles)

I n s t r. O r d e r

No forwarding needed since write reg in 1st half cycle, read reg in 2nd half cycle.

lw r1, 0(r2)

| Ifetch | Reg | ALU | DMem | Reg |

sub r4,r1,r6

| Ifetch | Reg | Bubble | ALU | DMem | Reg |

and r6,r1,r7

| Ifetch | Bubble | Reg | ALU | DMem | Reg |

or r8,r1,r9

| Bubble | Ifetch | Reg | ALU | DMem |

How is this hazard detected?
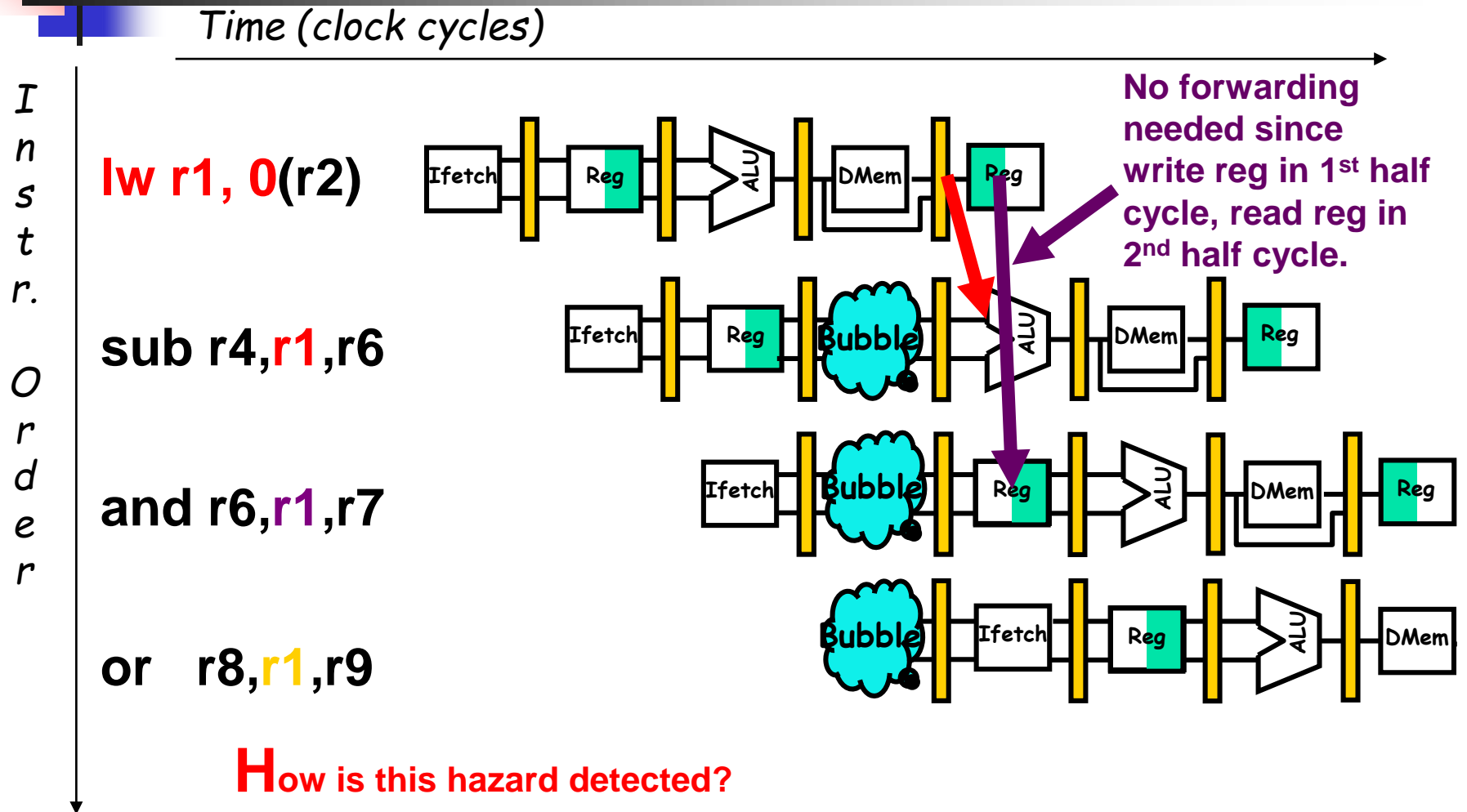
Software Scheduling to Avoid Load Hazards

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:
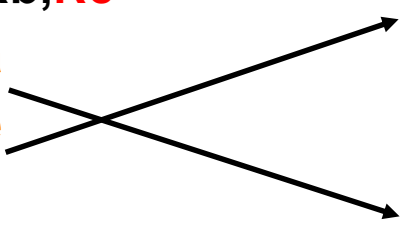
Fast code (no stalls):

	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

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.
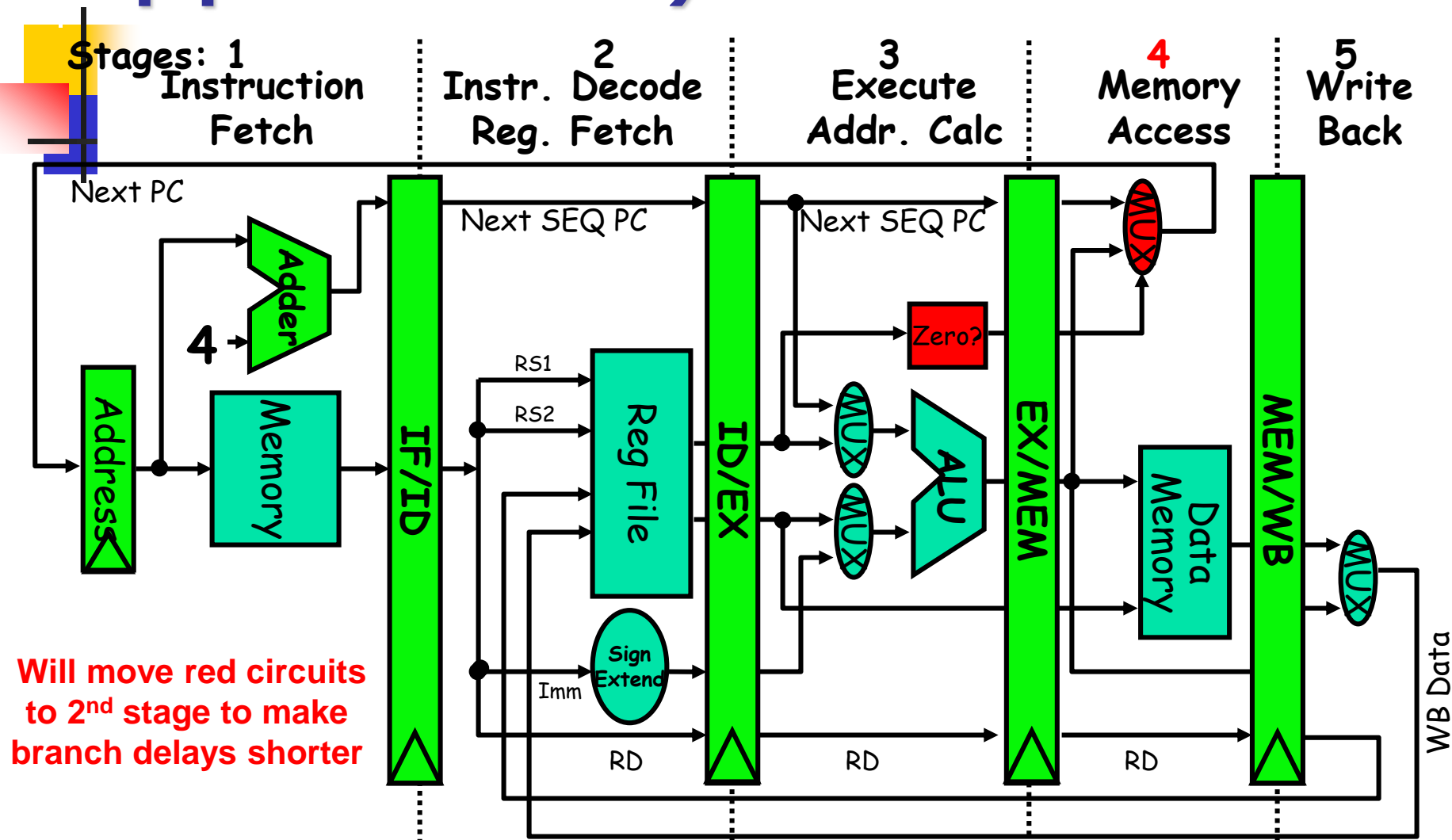
Important technique !

Outline

- **MIPS – An ISA for Pipelining**
- **5 stage pipelining**
- **Structural and Data Hazards**
- **Forwarding**
- **Branch Schemes**
- **Exceptions and Interrupts**
- **Conclusion**
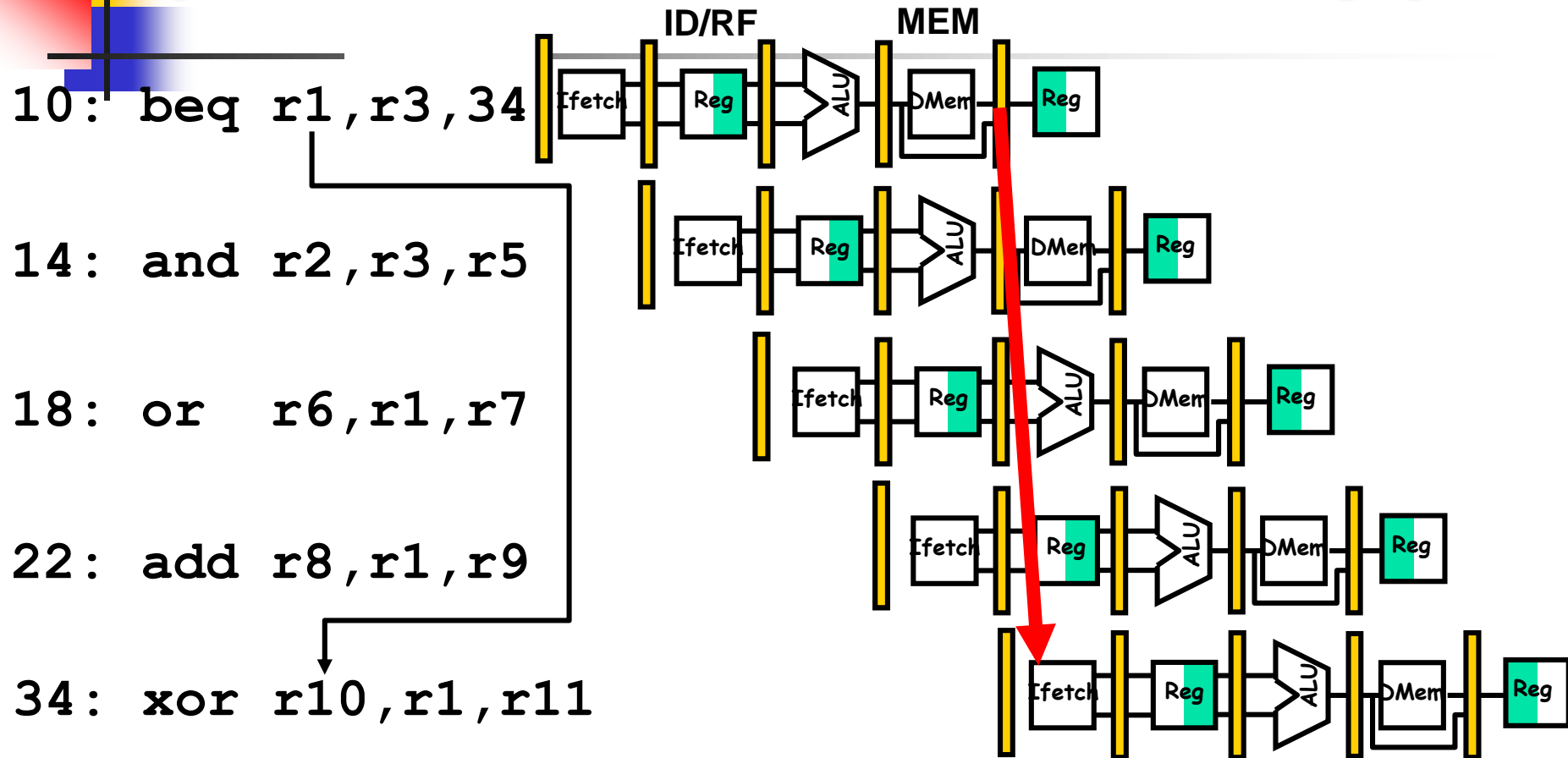
5-Stage MIPS Datapath (has pipeline latches)

Stages:

| 1 Instruction Fetch | 2 Instr. Decode Reg. Fetch | 3 Execute Addr. Calc | 4 Memory Access | 5 Write Back |

Next PC

Next SEQ PC

Next SEQ PC

Adder

4

Address

Memory

IF/ID

RS1

RS2

Reg File

ID/EX

MUX

MUX

ALU

Zero?

EX/MEM

MUX

Data Memory

MEM/WB

MUX

WB Data

Sign Extend

Imm

RD

RD

RD

Will move red circuits to 2nd stage to make branch delays shorter

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

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

ID/RF   **MEM**

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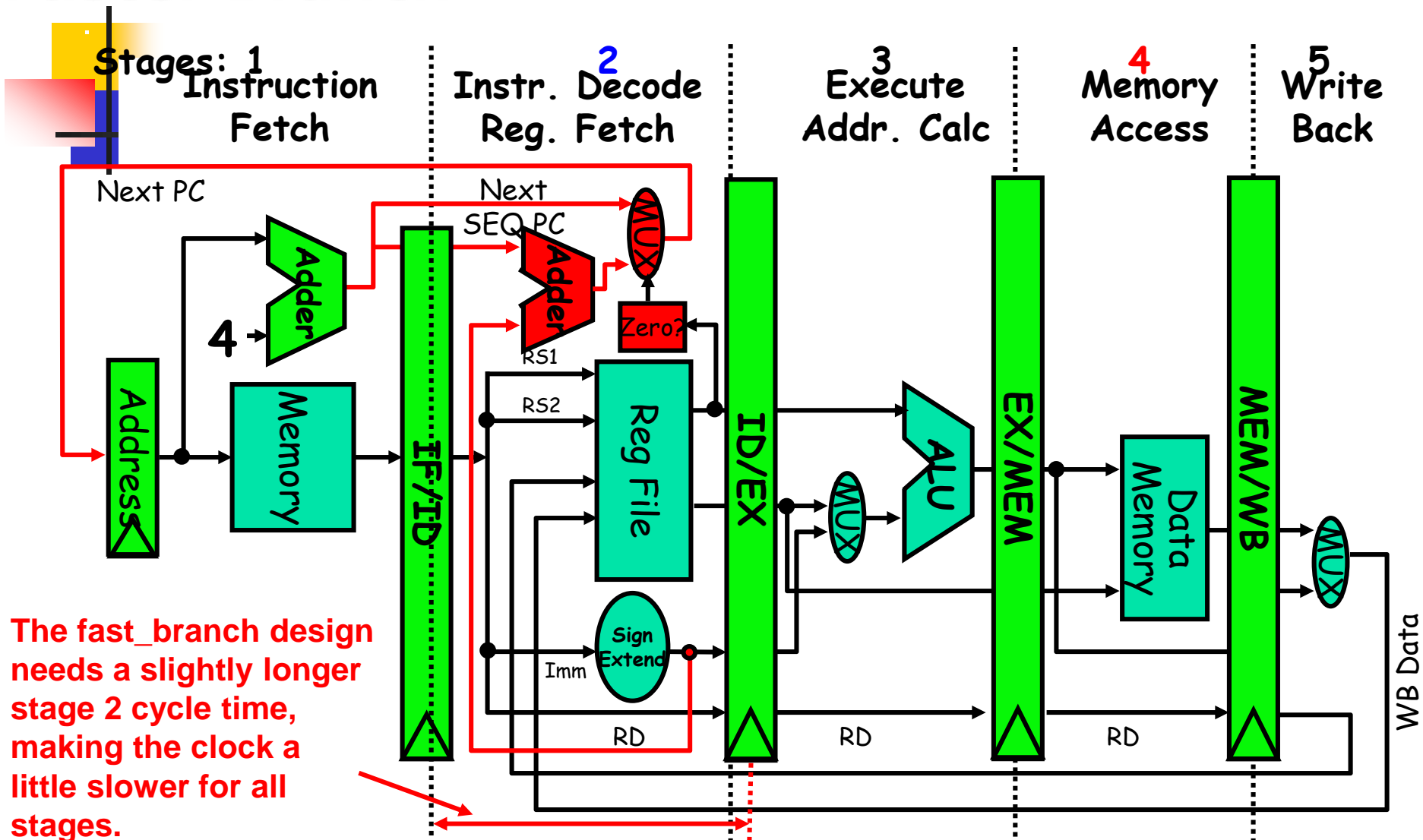Adding 3 cycle stall after every branch (1/7 of instructions) => CPI += 3/7. BAD!

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 $\neq$ 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

Stages:
1. Instruction Fetch
2. Instr. Decode Reg. Fetch
3. Execute Addr. Calc
4. Memory Access
5. Write Back

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**
- **<u>But have not calculated branch target address in MIPS</u>**
  - **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
```
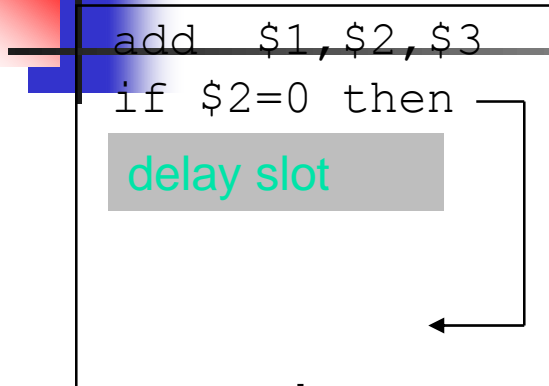
Branch delay of length n

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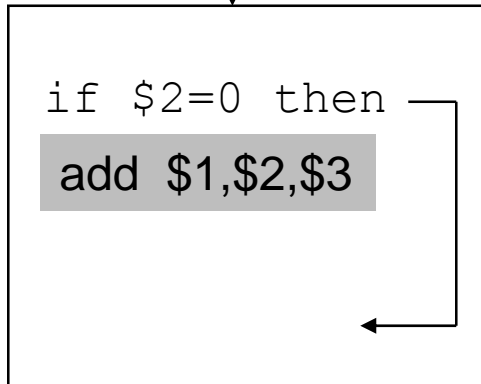- **MIPS 1st used this (Later versions of MIPS did not; pipeline deeper)**
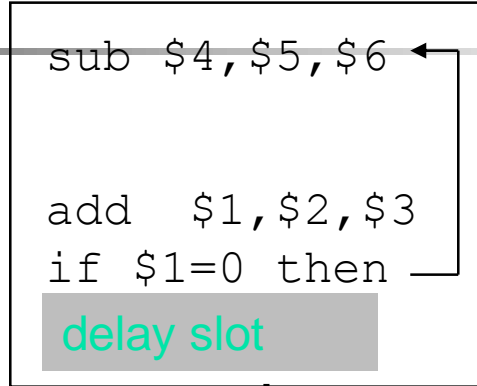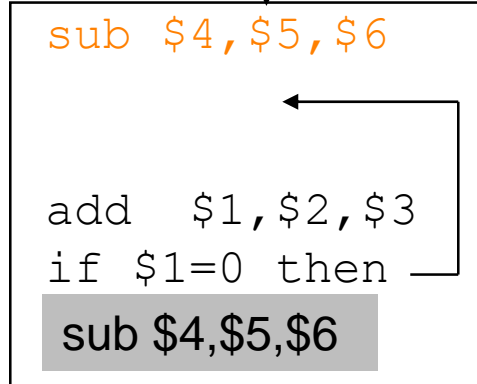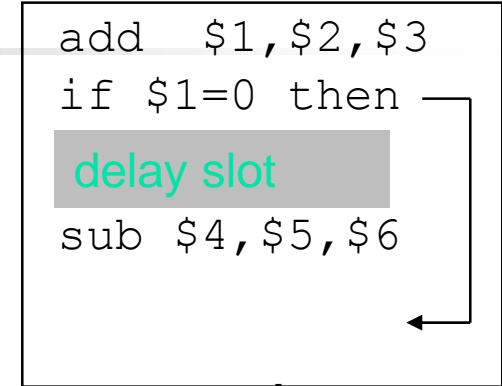
Scheduling Branch Delay Slots

A. From before branch

```
add  $1,$2,$3
if $2=0 then

    delay slot
```

becomes

```
if $2=0 then

  add $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6



add  $1,$2,$3
if $1=0 then

  delay slot
```

becomes

```
sub $4,$5,$6



add  $1,$2,$3
if $1=0 then

  sub $4,$5,$6
```

C. From fall through

```
add  $1,$2,$3
if $1=0 then

  delay slot
sub $4,$5,$6
```

becomes

```
add  $1,$2,$3
if $1=0 then

  sub $4,$5,$6
```
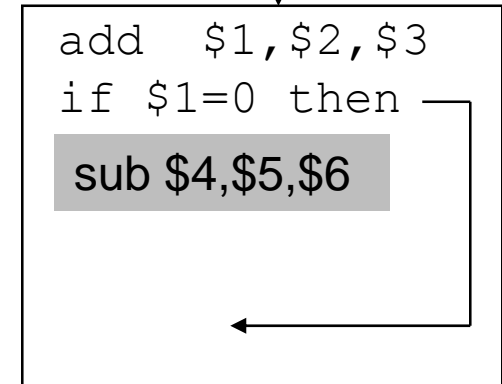
- **A is the best choice, fills delay slot & reduces instruction count (IC)**
- **In B, the `sub` instruction may need to be copied, increasing IC**
- **In B and C, must be okay to execute an extra `sub` when branch fails**

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 1.60=1+3(4+10+6)%   (4.5=5/1.10)   (1.45=1.6/1.1)
 calcu-   1.20=1+1(4+10+6)%   (to calculate taken target)
 **lations) 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 <u>between</u> 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
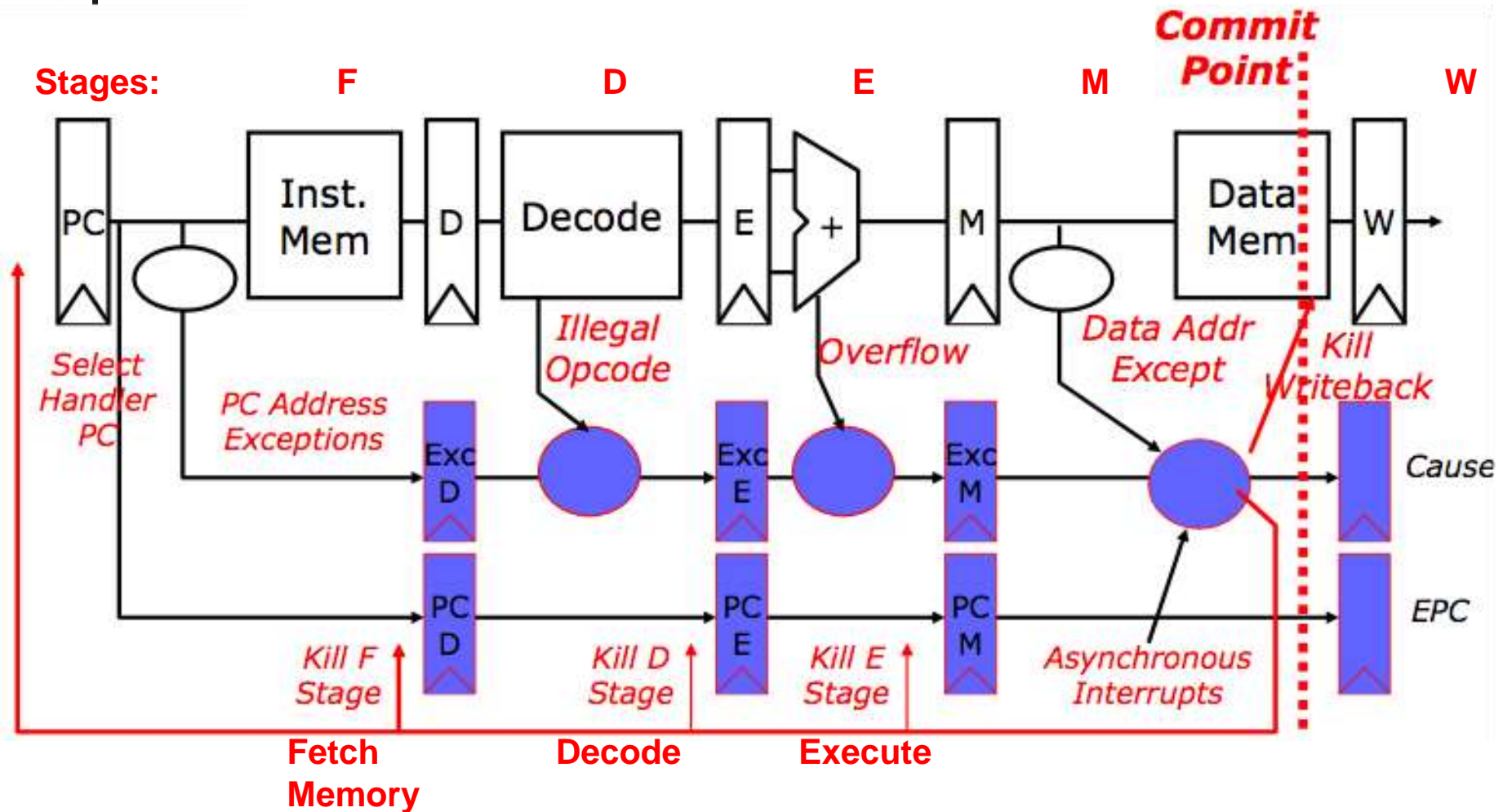
Key observation: "Architected" states change only in memory (M) and register write (W) stages.

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**