Computer Architecture

Lecture 3: Pipelining

Iakovos Mavroidis

Computer Science Department
University of Crete

Previous Lecture

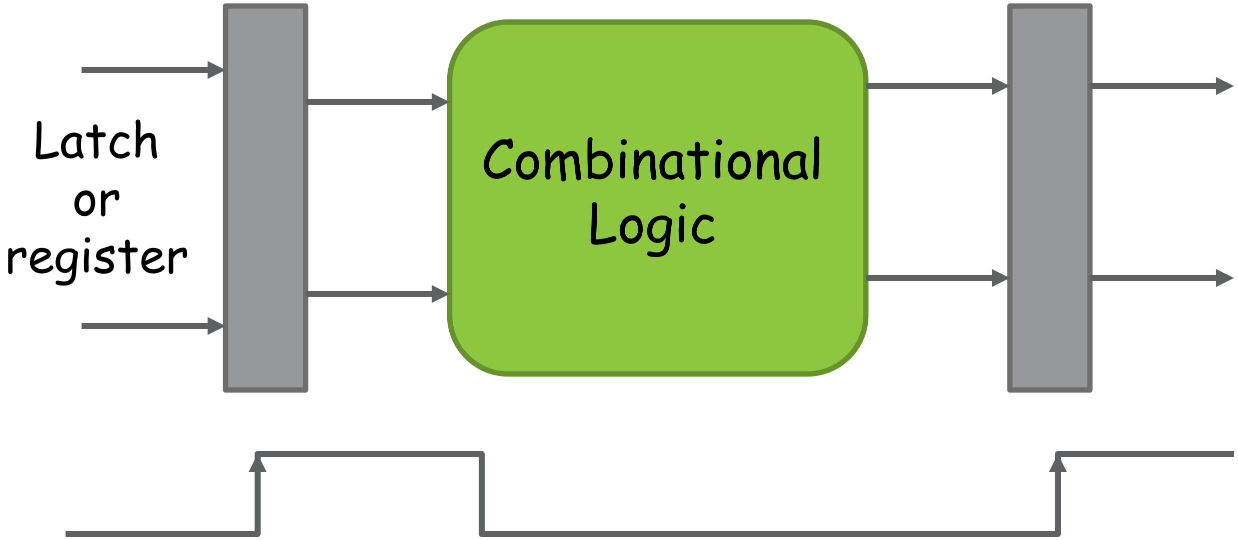
- Measurements and metrics : Performance, Cost, Dependability, Power
- Guidelines and principles in the design of computers

▪Monday 8/10	→	Friday 12/10
▪Monday 15/10	→	Friday 19/10
▪Wednesday 17/10	→	TBD

Outline

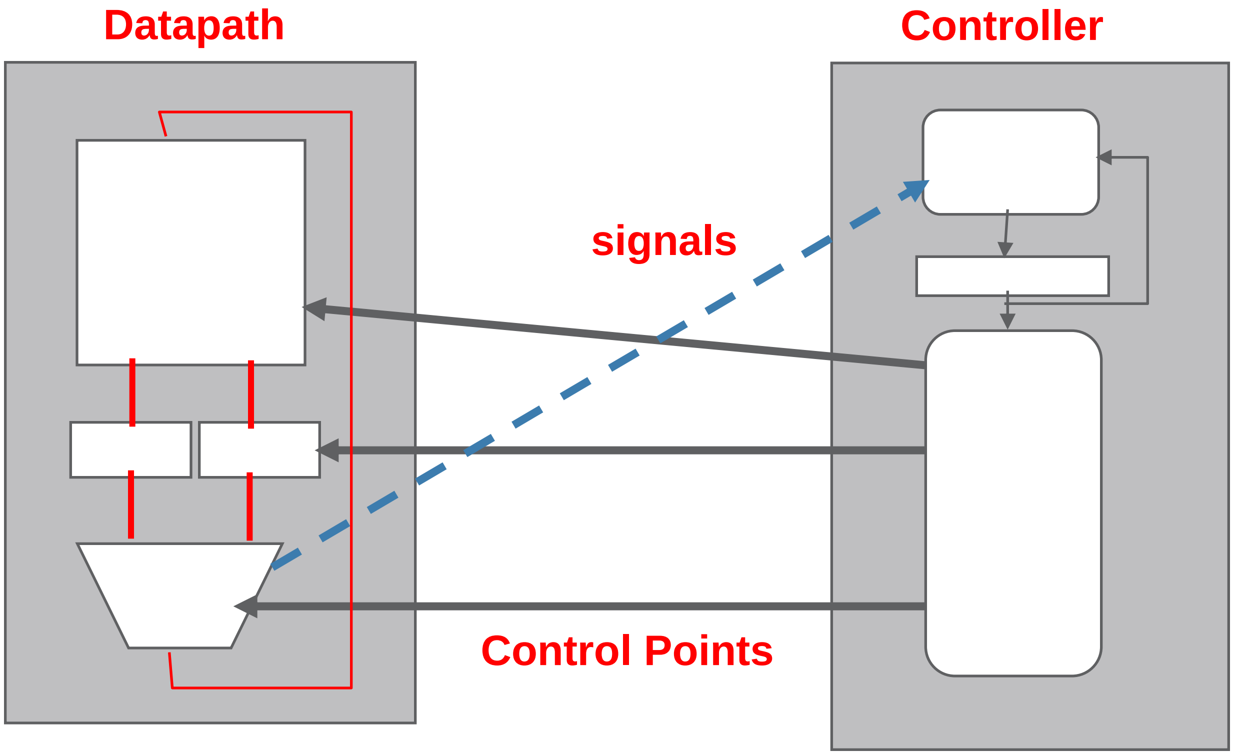
- Processor review
- Hazards
  - Structural
  - Data
  - Control
- Performance
- Exceptions

Clock Cycle



- ❑Old days: 10 levels of gates
- ❑Today: determined by numerous time-of-flight issues + gate delays
  - clock propagation, wire lengths, drivers

Datapath vs Control



- ❑ Datapath: Storage, FU, interconnect sufficient to perform the desired functions
  - Inputs are Control Points
  - Outputs are signals
- ❑ Controller: State machine to orchestrate operation on the data path
  - Based on desired function and signals

“Typical” RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: 32bit MIPS

Register-Register

31 26	25 21	20 16	15 11	10 6	5 0
Op	Rs1	Rs2	Rd		Opx

Register-Immediate

31 26	25 21	20 16	15 0
Op	Rs1	Rd	immediate

Branch

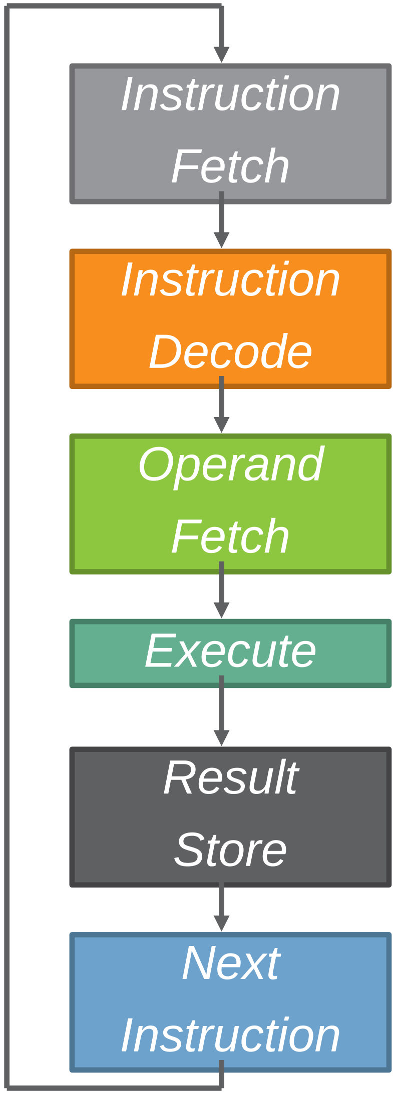
31 26	25 21	20 16	15 0
Op	Rs1	Rs2/Opx	immediate

Jump / Call

31	25 0
Op	target

Example: lw $2, 100($5)
add $4, $5, $6
beq $3, $4, label

Example Execution Steps



Instruction Fetch

Obtain instruction from program storage

Instruction Decode

Determine required actions and instruction size

Operand Fetch

Locate and obtain operand data

Execute

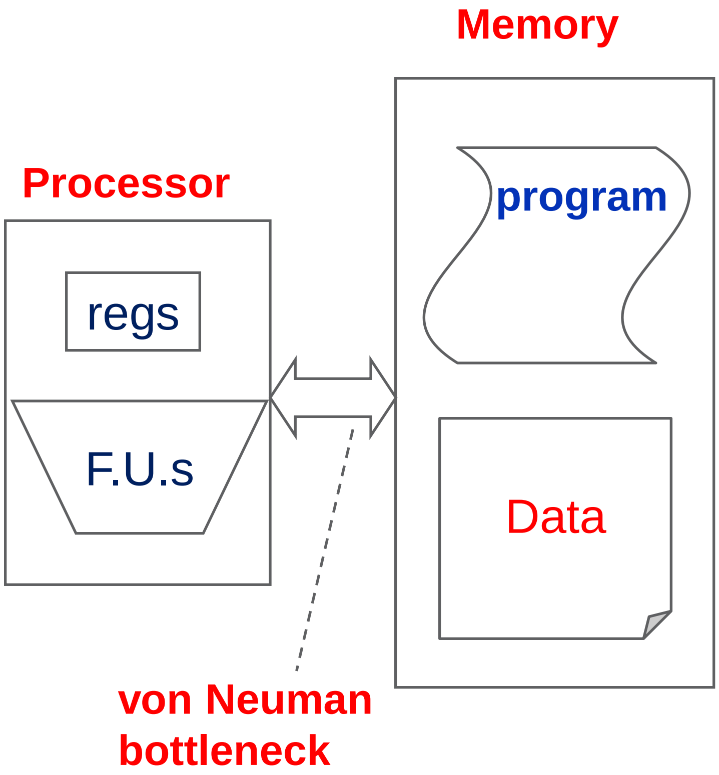
Compute result value or status

Result Store

Deposit results in storage for later use

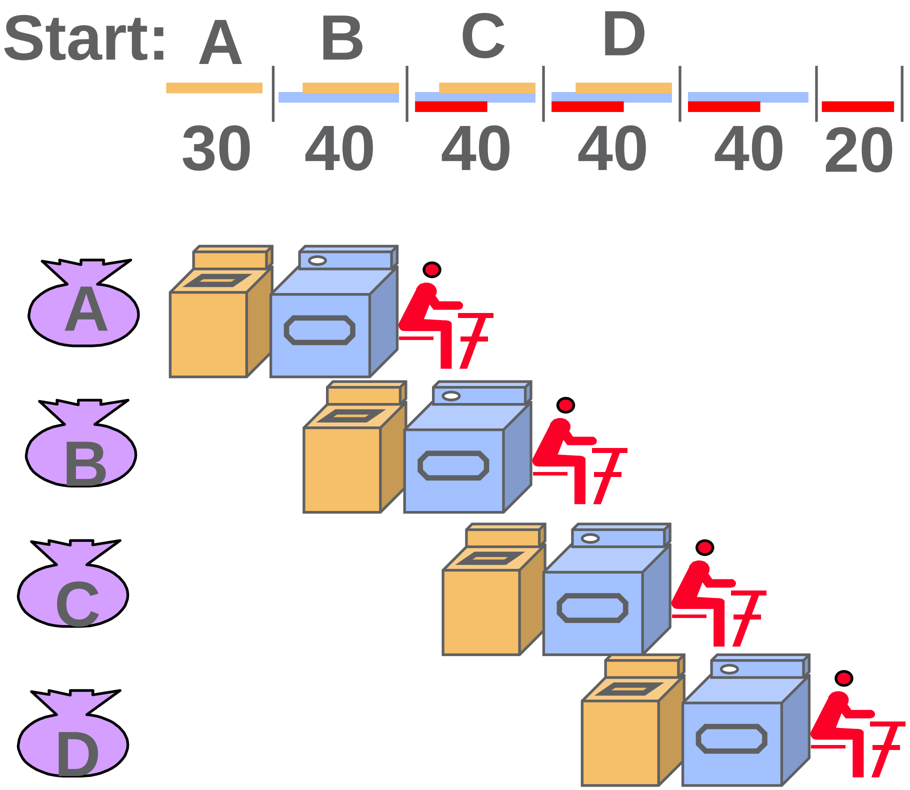
Next Instruction

Determine successor instruction



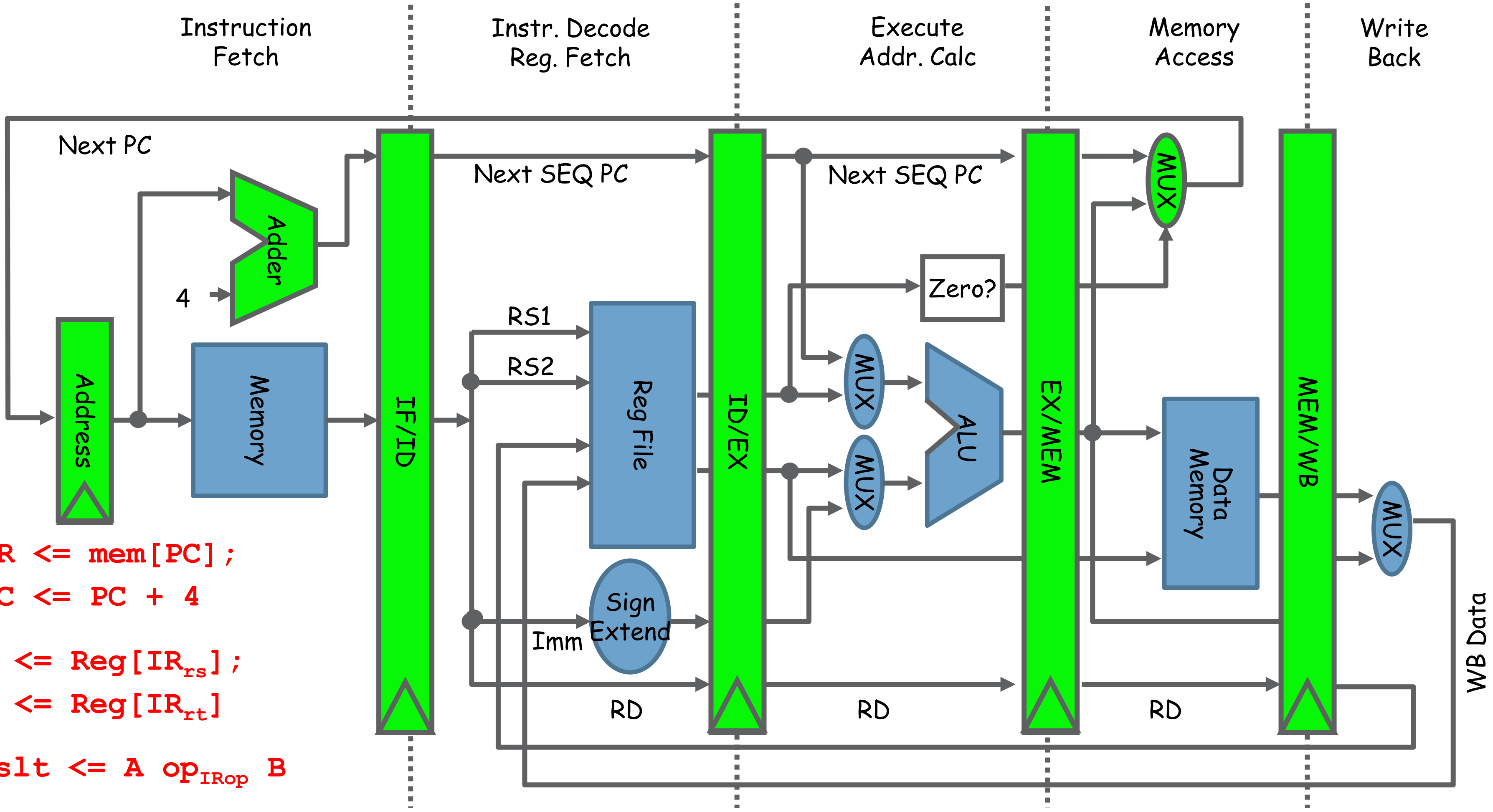
5-stage execution is a bit different (see next slides)…

Pipelining: Latency vs Throughput



Pipelining doesn’t help **latency** of single task, it helps **throughput** of entire workload

5-stage Instruction Execution - Datapath



IR <= mem[PC];

PC <= PC + 4

A <= Reg[IR_{rs}];

B <= Reg[IR_{rt}]

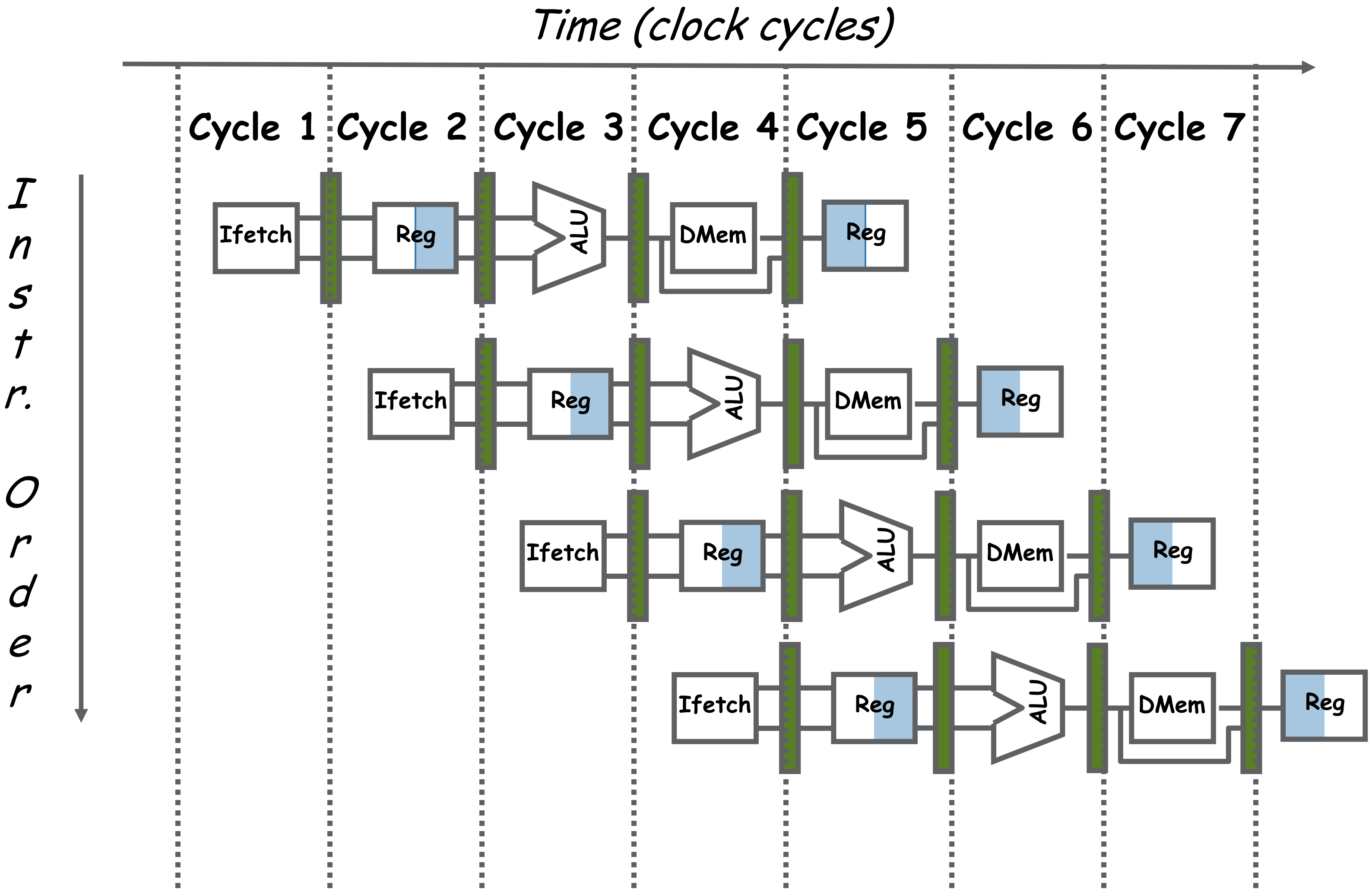
rslt <= A op_{IRop} B

WB <= rslt

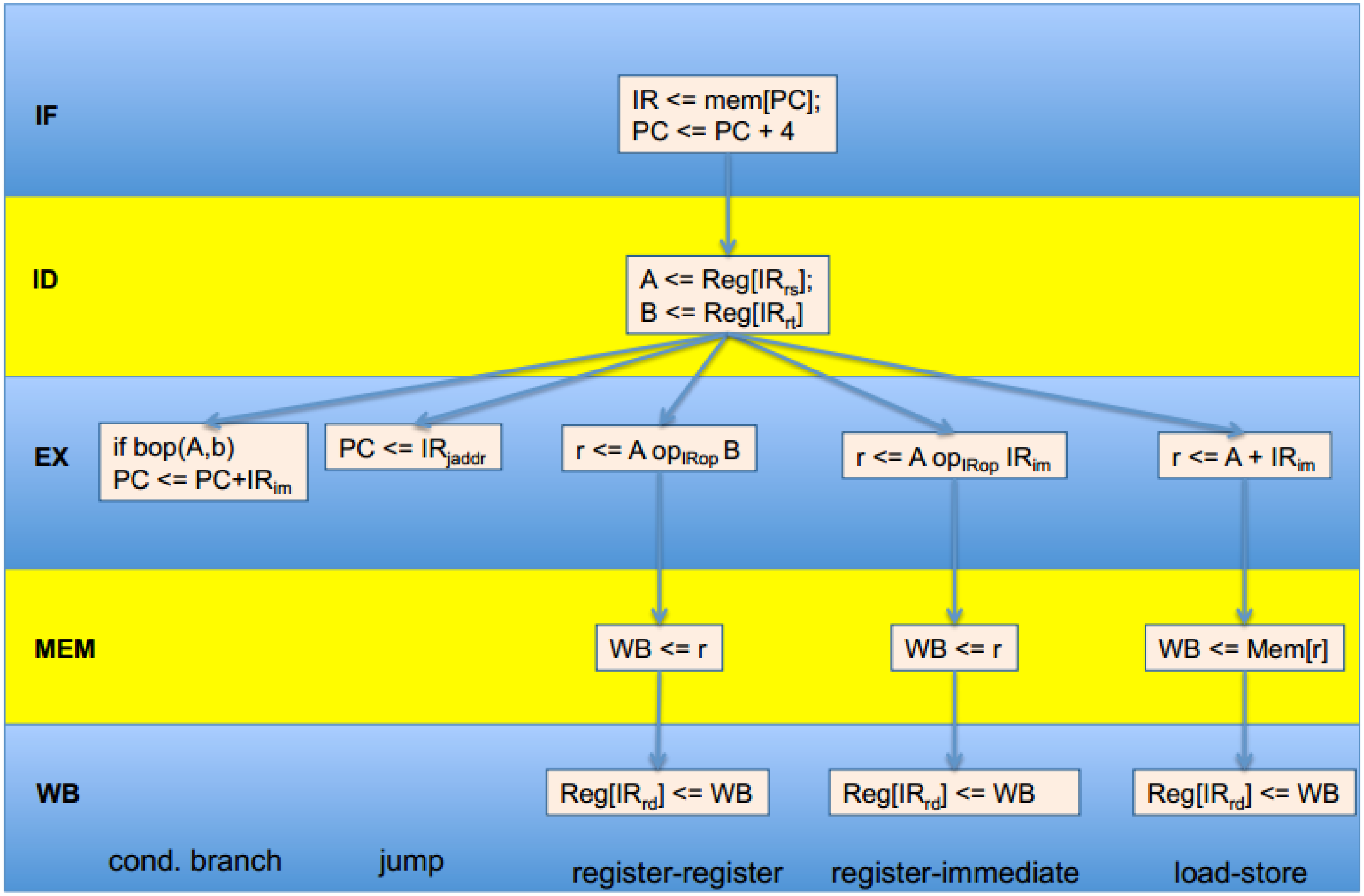
Reg[IR_{rd}] <= WB

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

Visualizing Pipelining



5-stage Instruction Execution - Control



IF

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

ID

A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

EX

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}

MEM

WB <= r

WB <= r

WB <= Mem[r]

WB

Reg[IR_{rd}] <= WB

Reg[IR_{rd}] <= WB

Reg[IR_{rd}] <= WB

cond. branch

jump

register-register

register-immediate

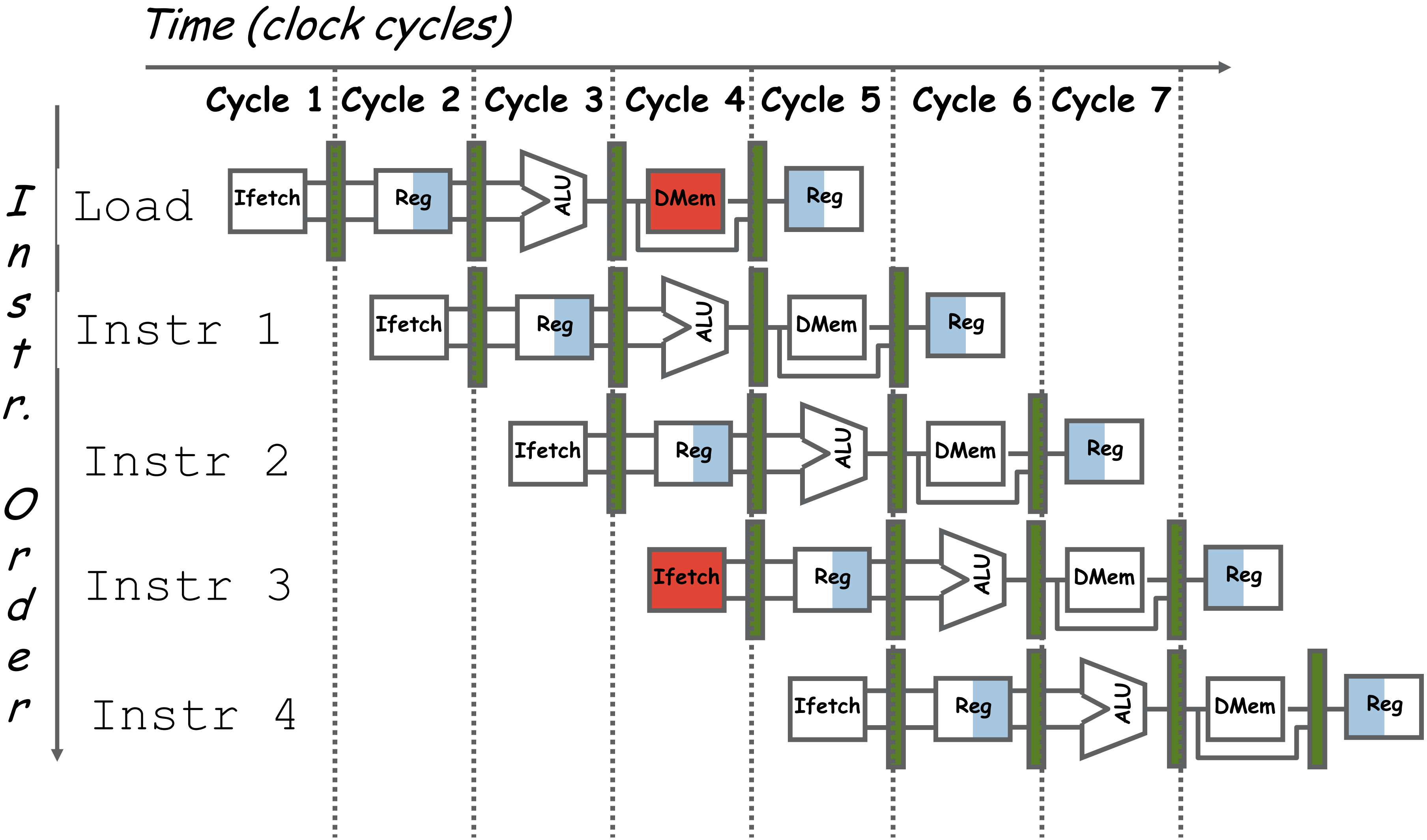
load-store

Pipeline Registers: IR, A, B, r, WB

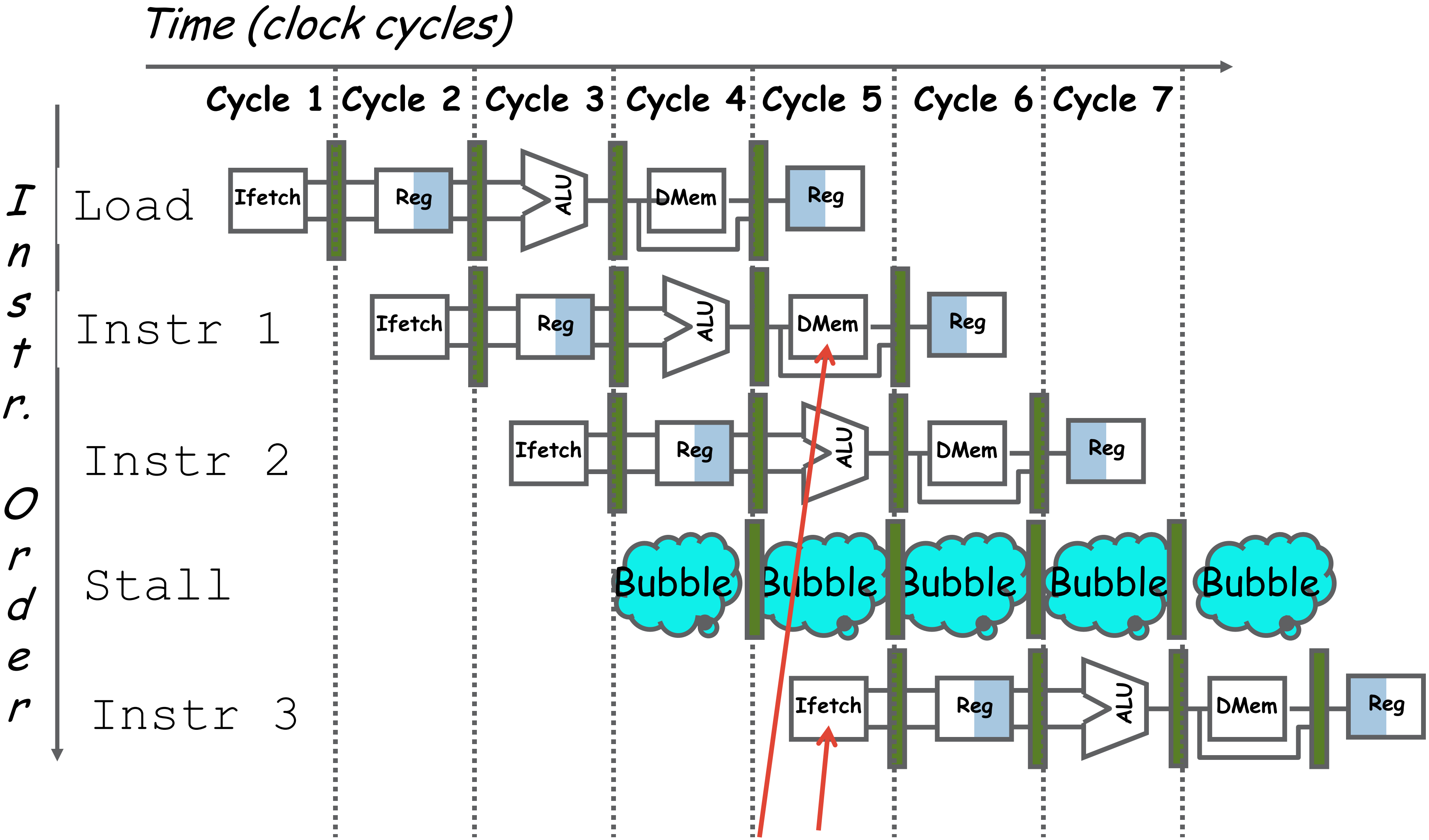
Limits in Pipelining

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

Example of Structural Hazard



Example of Structural Hazard



How do you "bubble" this pipe (if instr1 = load)?

Speed Up Equation of Pipelining

$$\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

For simple RISC pipeline, Ideal CPI = 1:

$$\text{Speedup} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

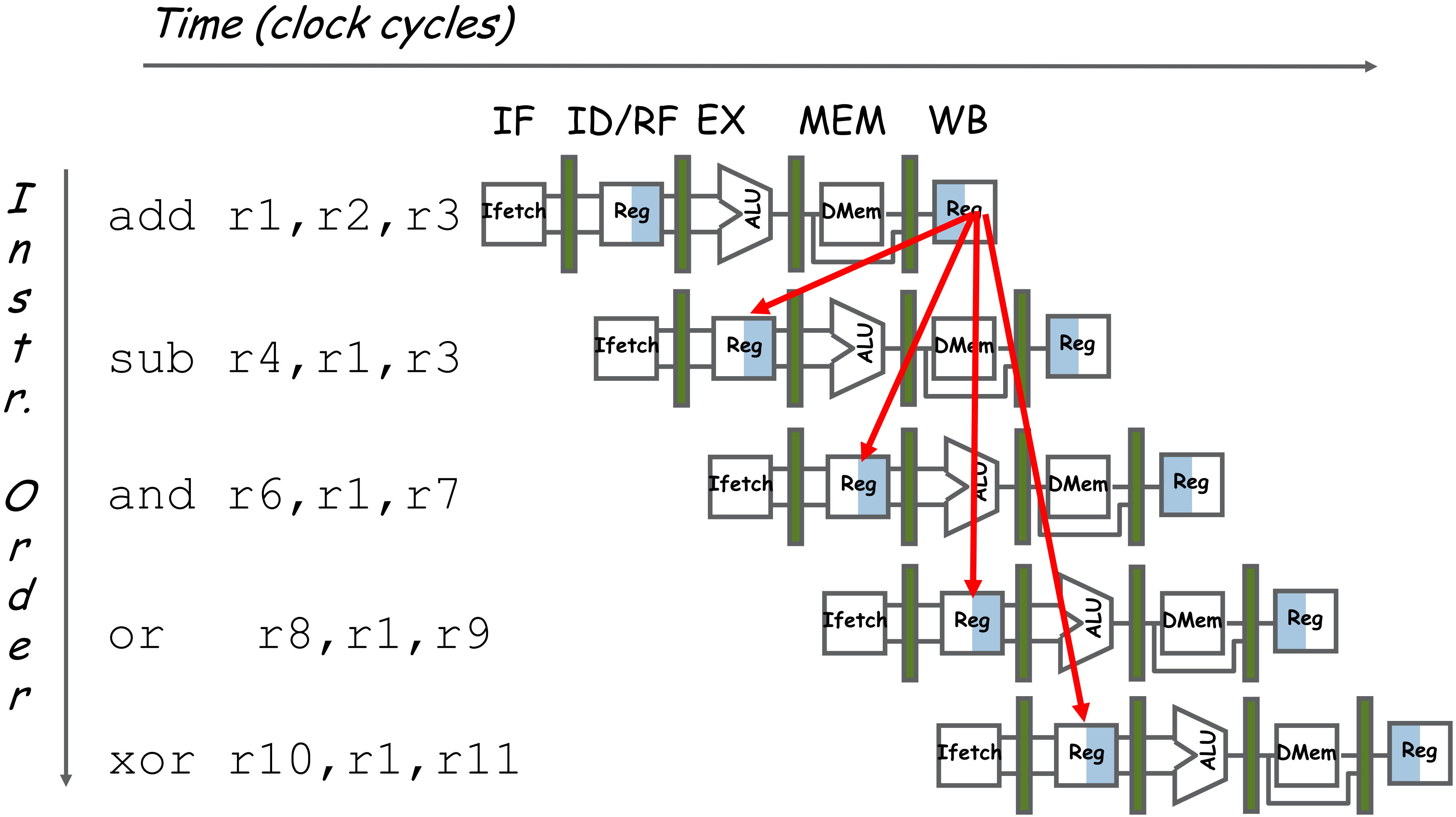
Example: Dual-port vs Single-port

- ❑Machine A: Dual read ported memory (“Harvard Architecture”)
- ❑Machine B: Single read ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- ❑Ideal CPI = 1 for both
- ❑Suppose that Loads are 40% of instructions executed

$$
\begin{aligned}
\text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\
&= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\
&= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
\end{aligned}
$$

- ❑Machine A is 1.33 times faster

Data Hazard



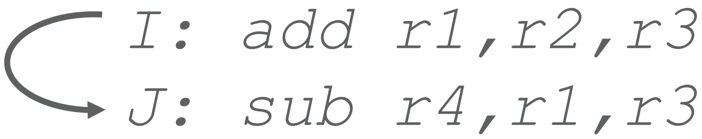
Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```

Read After Write

- ❑ 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 "Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Write After Read

- 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”.

- Can’t happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

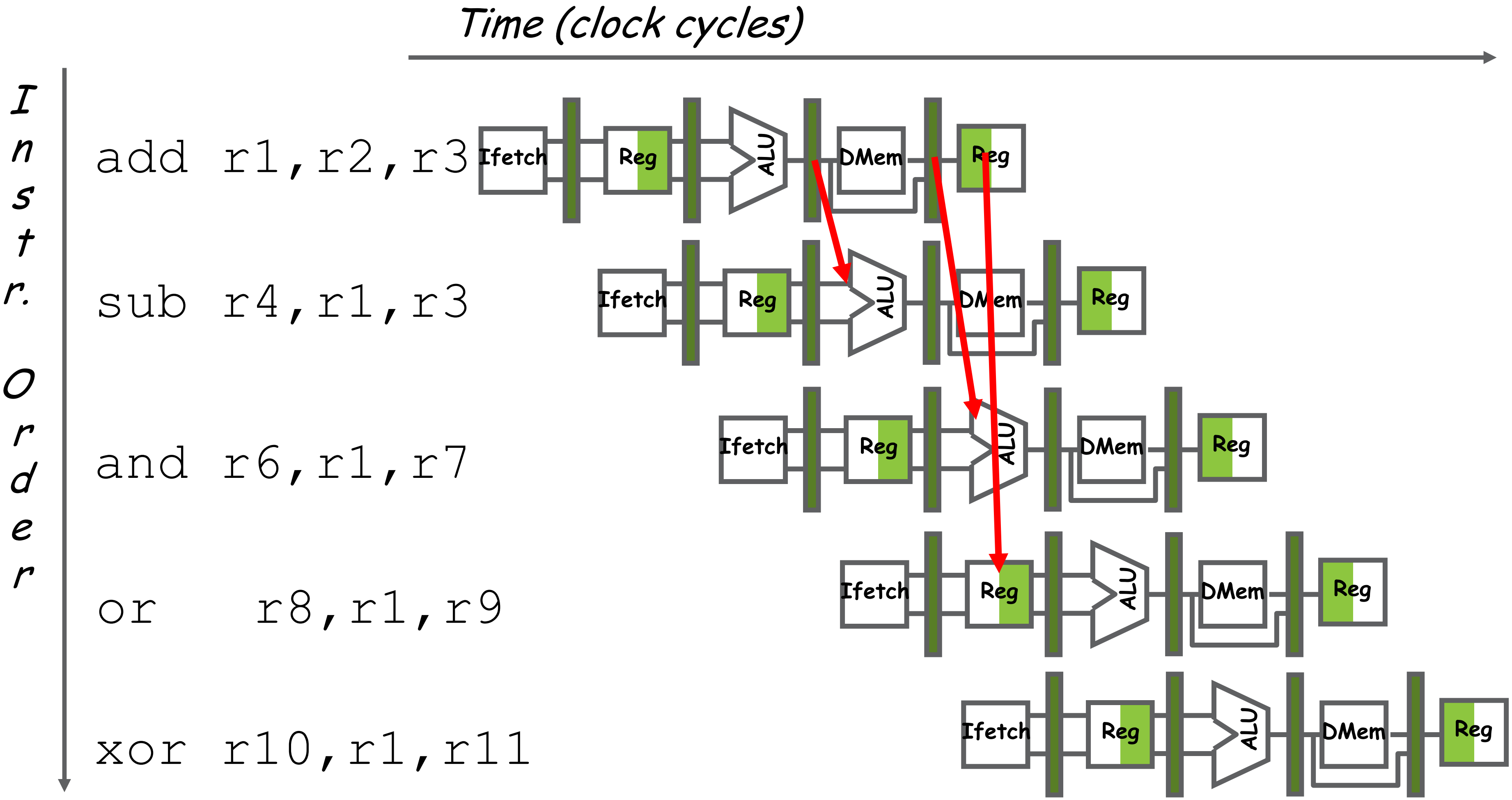
Write After Write

- 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".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in more complicated pipes

Forwarding to avoid data hazards

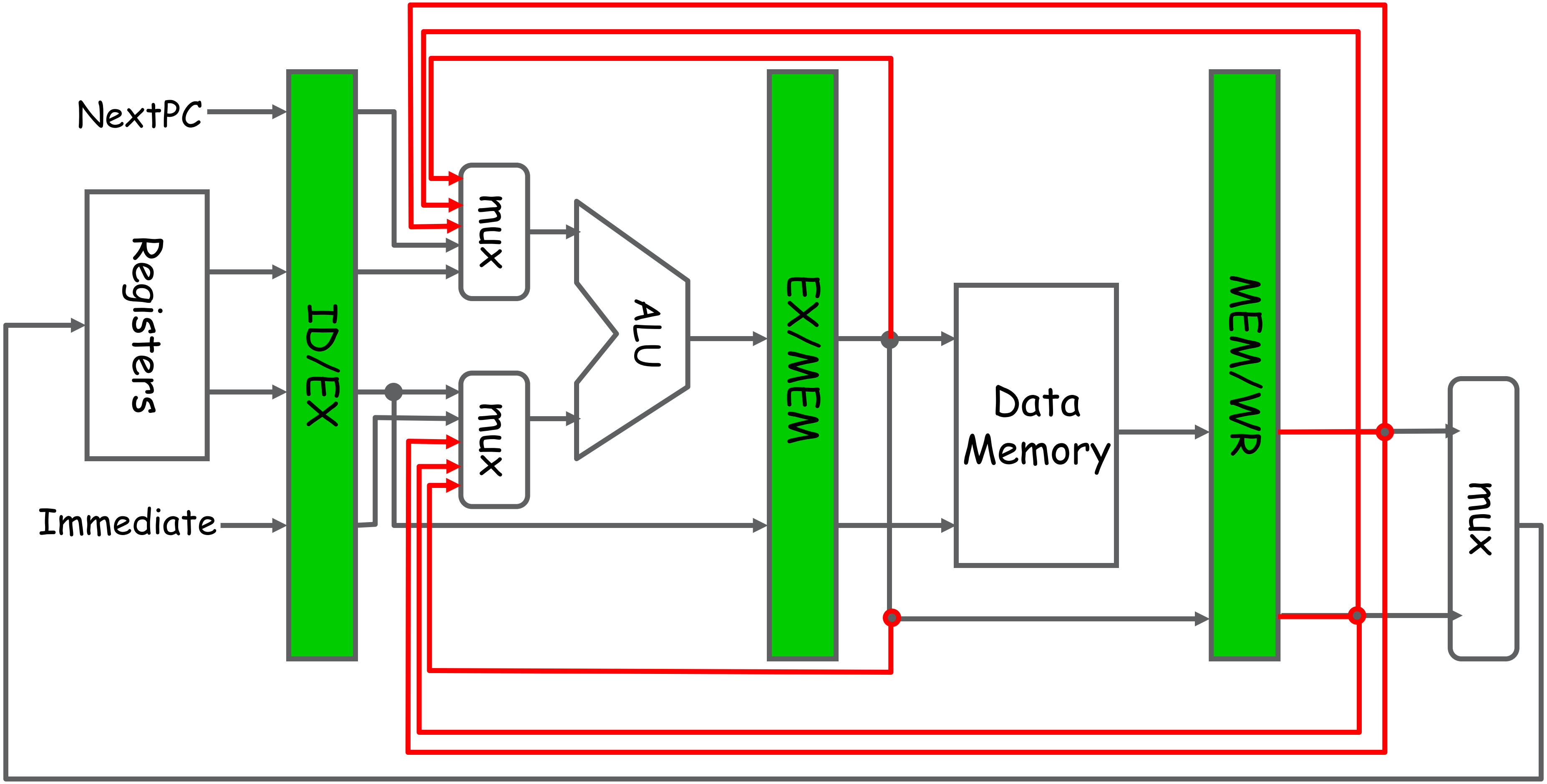


Time (clock cycles)

Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```

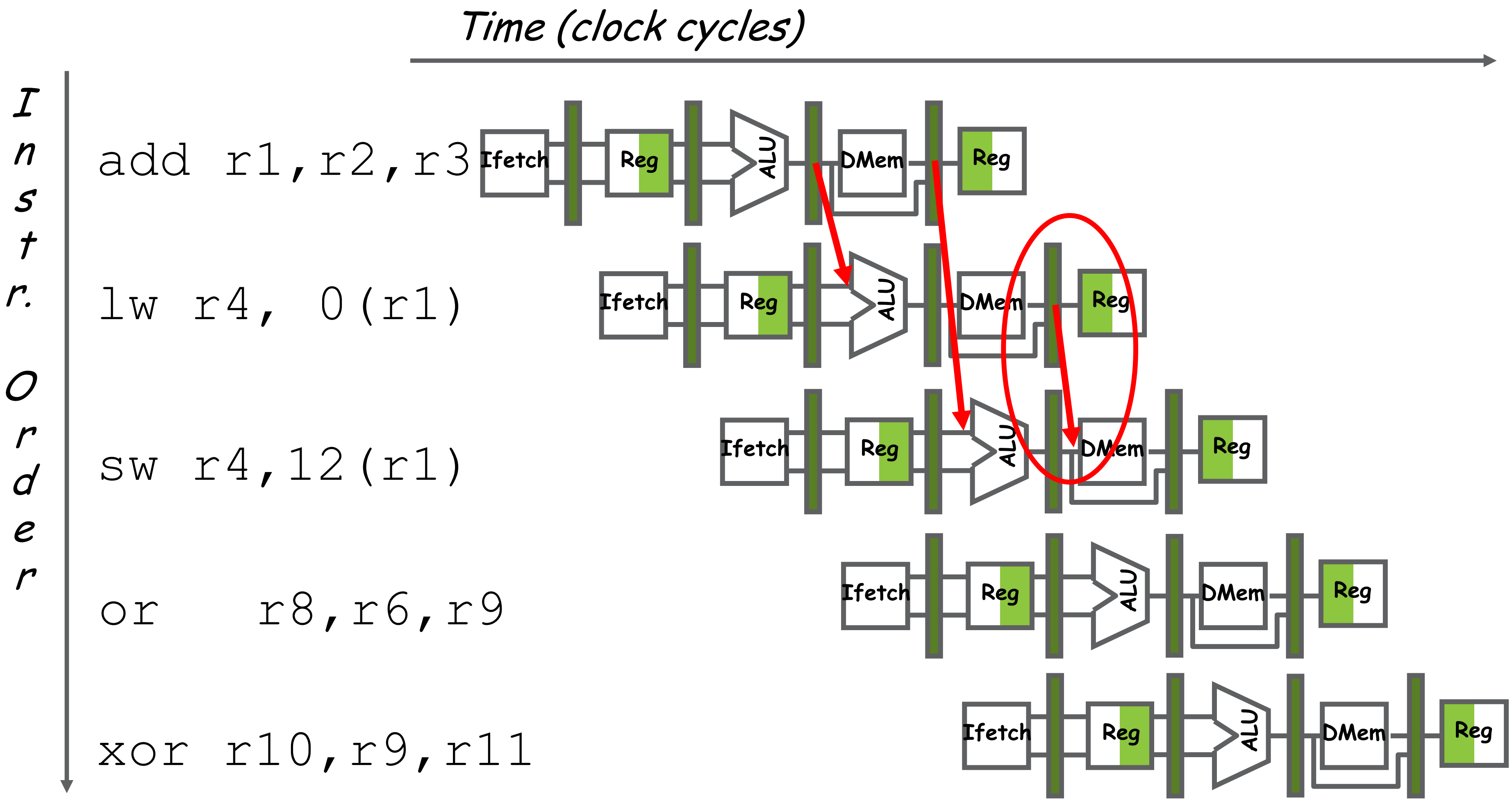
HW Change for Forwarding



What circuit detects and resolves this hazard?

Why we need forwarding lines for both inputs of the ALU?

Forwarding to Avoid LW-SW Data Hazard



Time (clock cycles)

Instr. Order

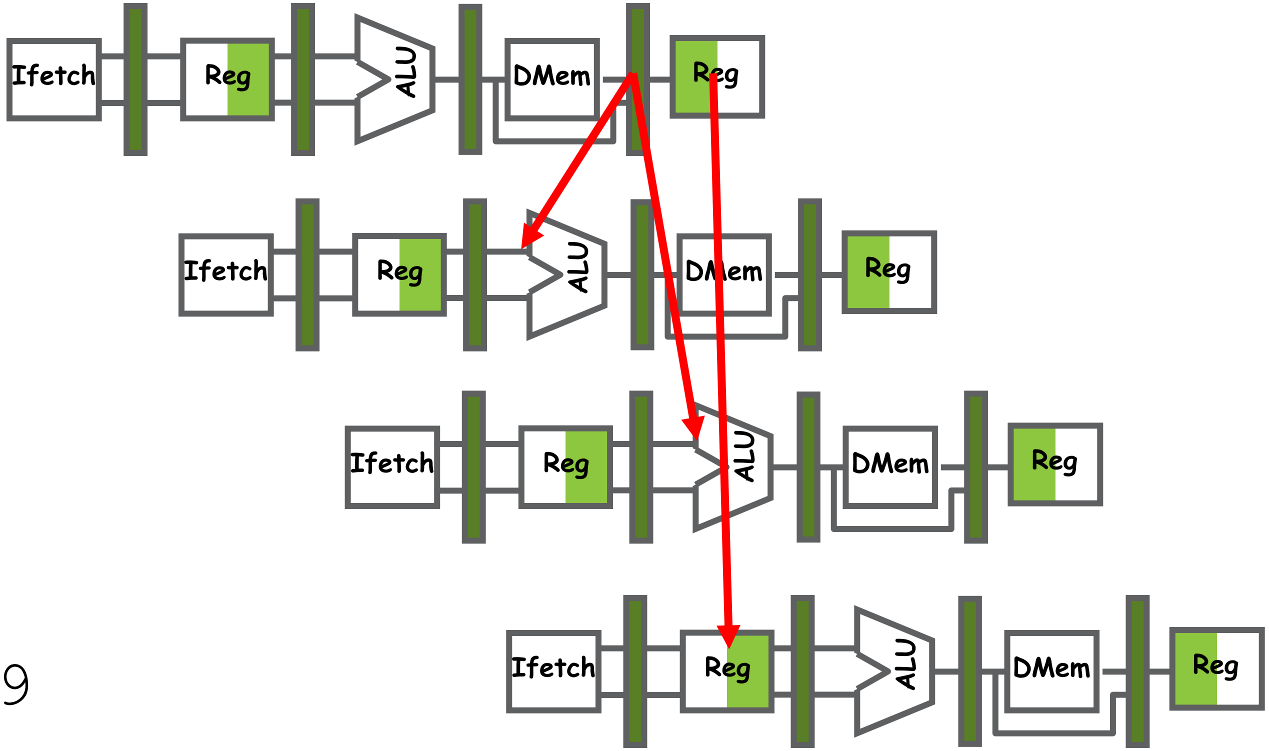
```
add r1,r2,r3
lw  r4, 0(r1)
sw  r4,12(r1)
or   r8,r6,r9
xor r10,r9,r11
```

Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

```
lw  r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or   r8,r1,r9
```



Data Hazard Even with Forwarding

Time (clock cycles)

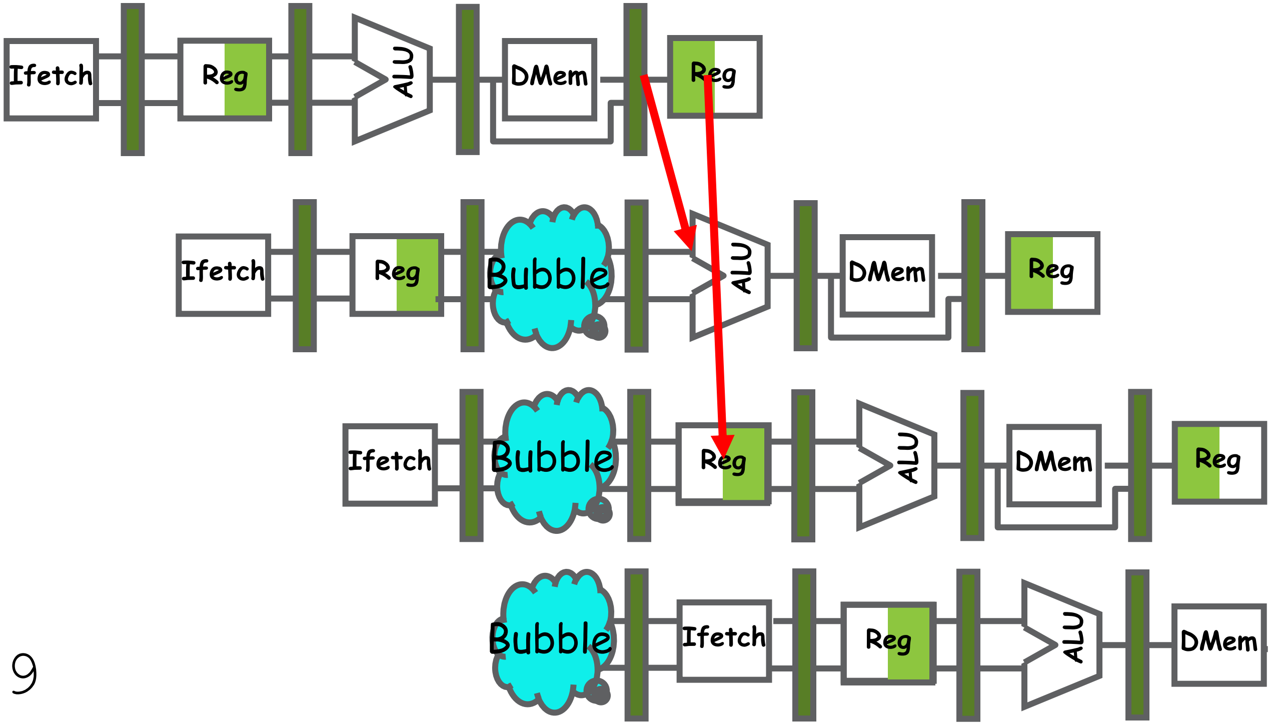
Instr. Order

```
lw  r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
```



Software Scheduling to Avoid Load Hazards

Try producing fast code for

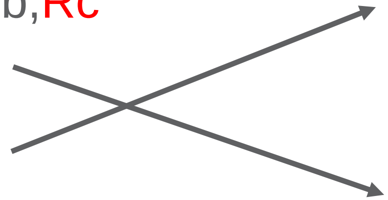
$$a = b + c;$$

$$d = e – f;$$

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

Slow code:

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



Fast code:

```
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
```

Control Hazard on Branches
Three Stage Stall

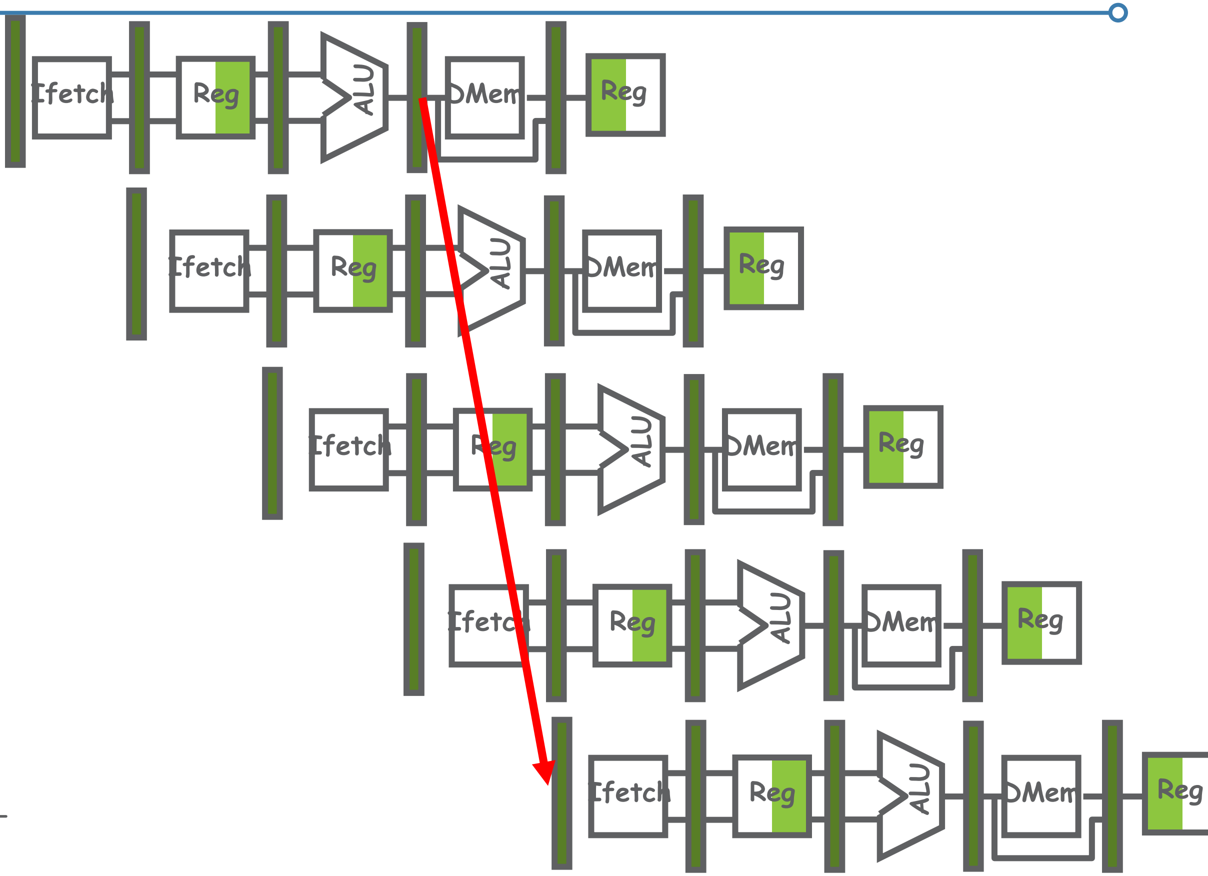
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```

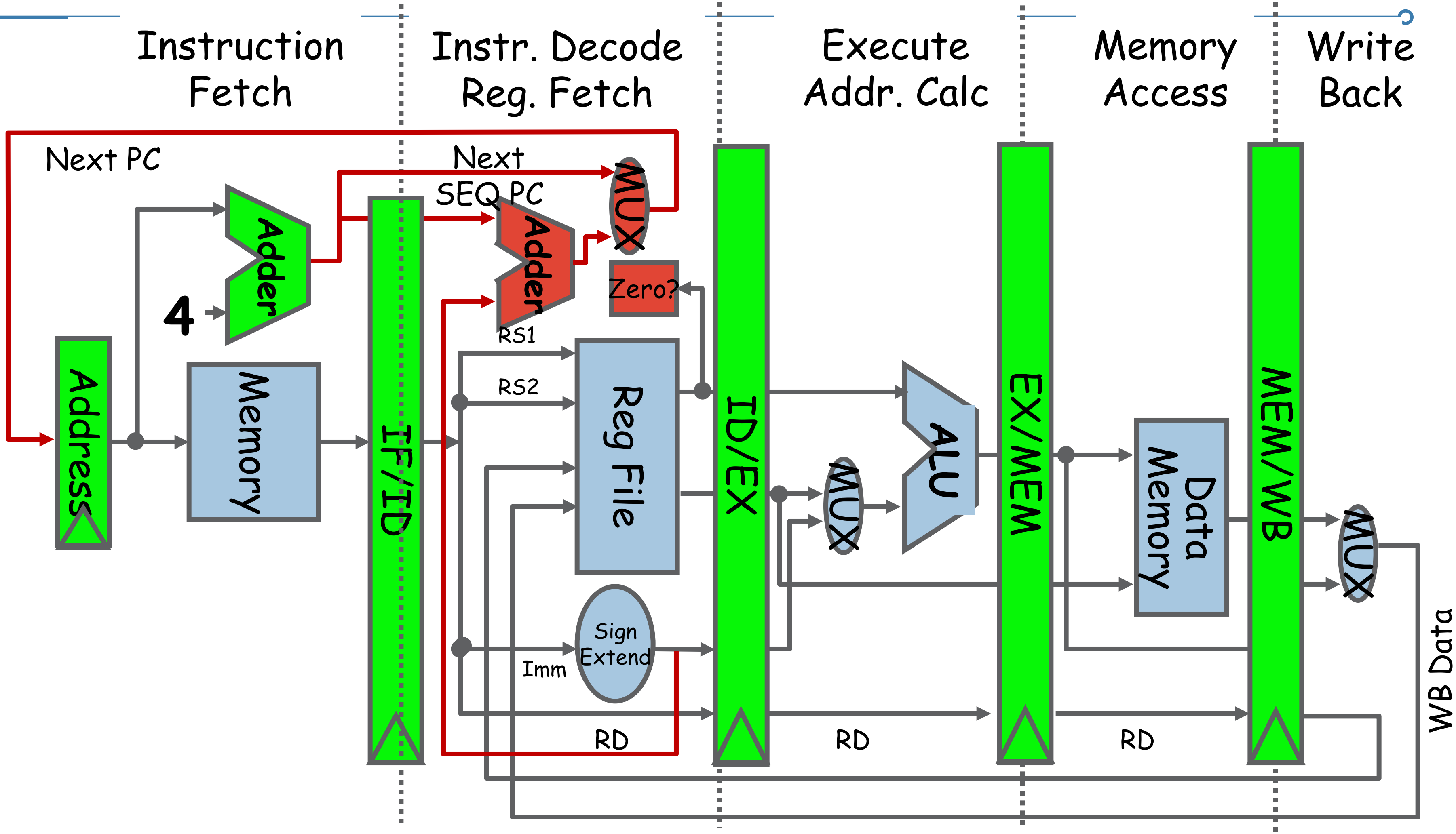


What do you do with the 3 instructions in between?
How do you do it?
Where is the "commit"?

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath



- Interplay of instruction set design and cycle time.

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- “Squash” instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven’t calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
- What happens when hit not taken branch?

Four Branch Hazard Alternatives

#4: Delayed Branch

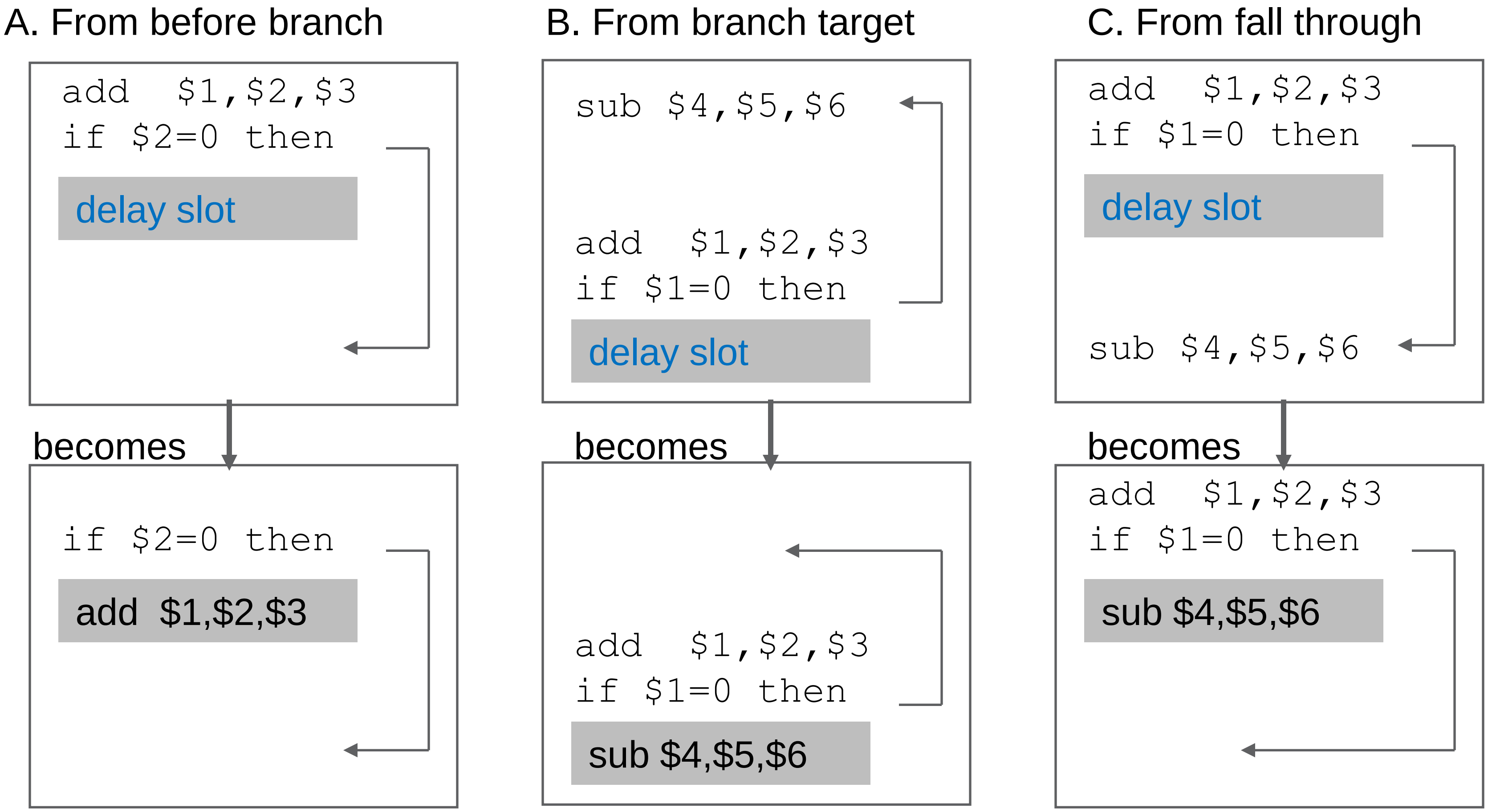
- Define branch to take place AFTER a following instruction

```
branch instruction
   sequential successor$_1$
   sequential successor$_2$
   ........
   sequential successor$_n$
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

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

```
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
```

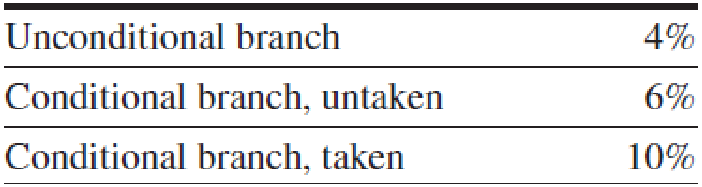
- ❑ A is the best choice, fills delay slot & reduces instruction count (IC)
- ❑ In B, the `sub` instruction may need to be copied, increasing IC
- ❑ In B and C, must be okay to execute `sub` when branch fails

Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

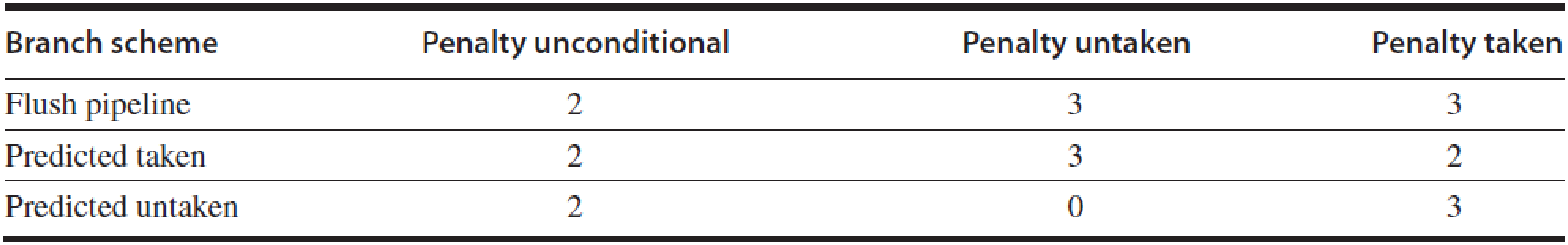
Evaluating Branch Alternatives

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

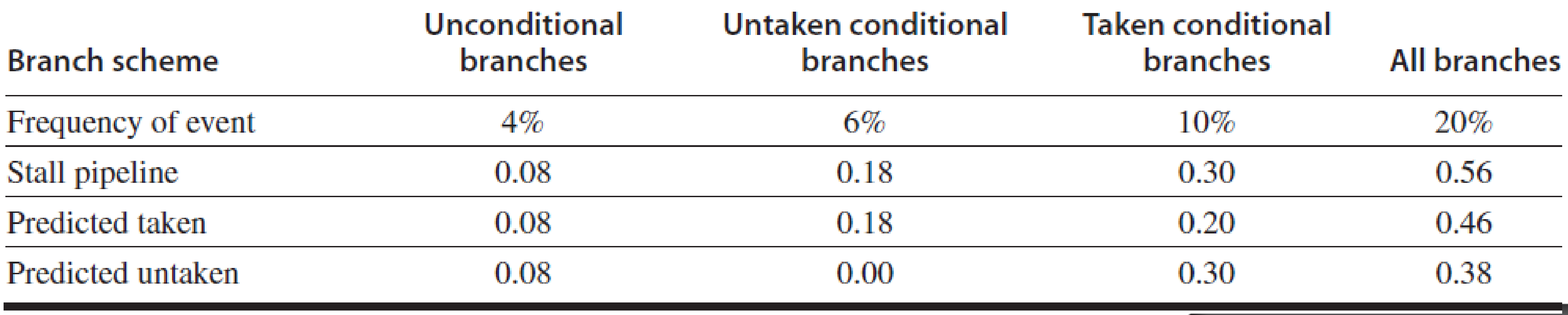


Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Deep pipeline in this example



Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

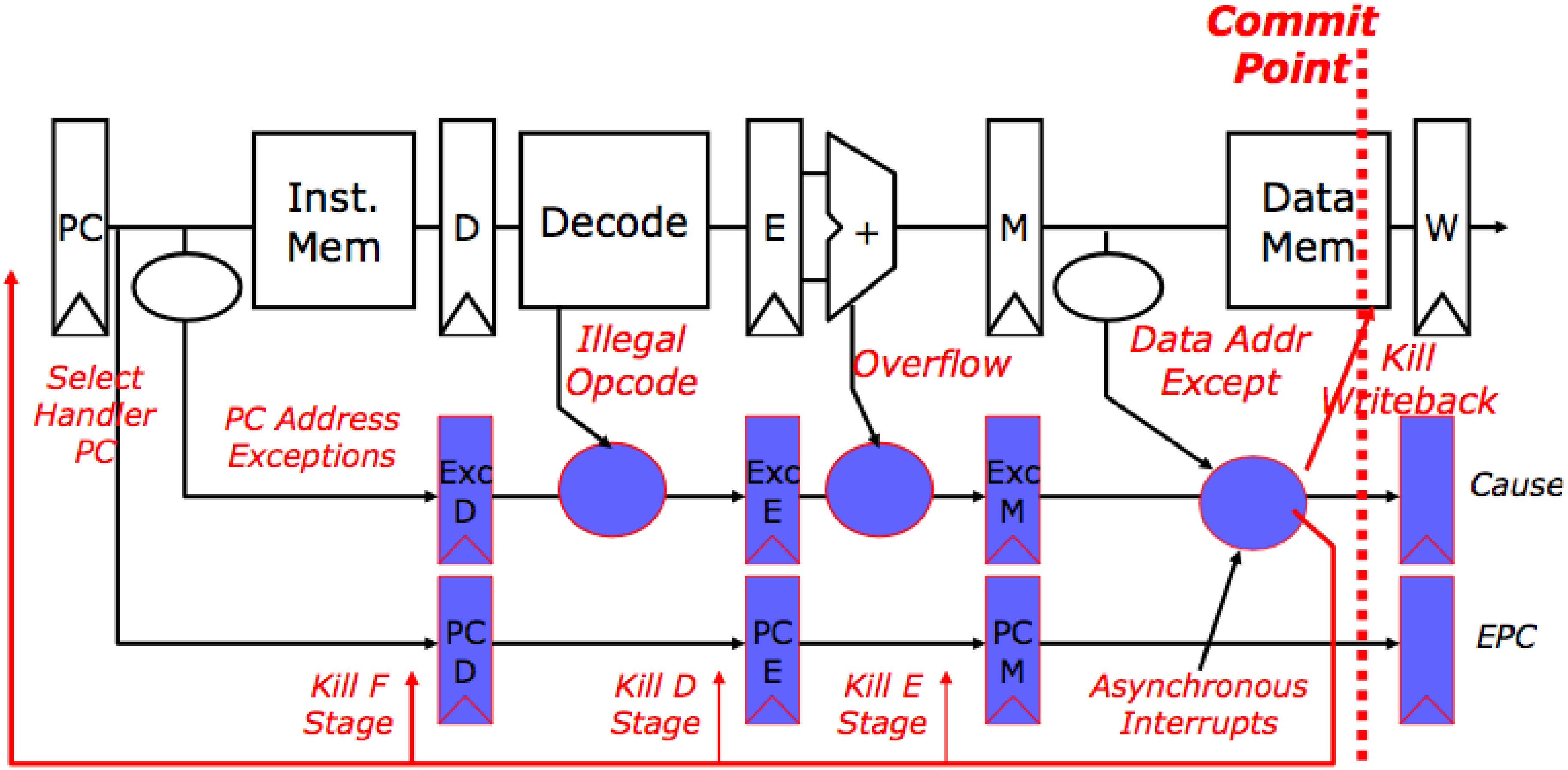


Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Problems with Pipelining

- Exception: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- Interrupt: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- Problem: It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
  - The effect of all instructions up to and including I_i is totalling complete
  - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

Summary: Pipelining

- ❑Next time: Read Appendix A
- ❑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:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- ❑Exceptions, Interrupts add complexity