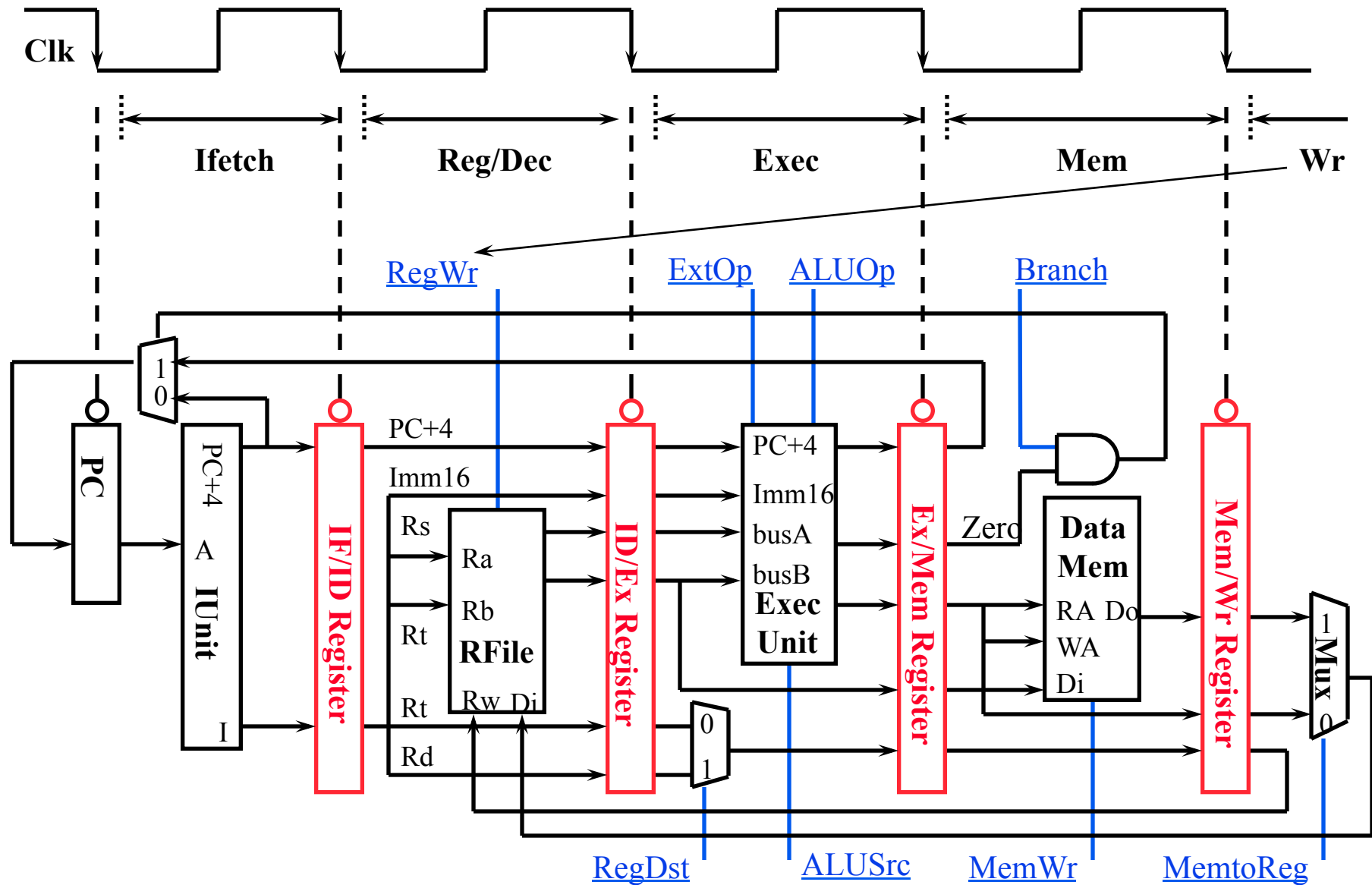# ECE 361
Computer Architecture
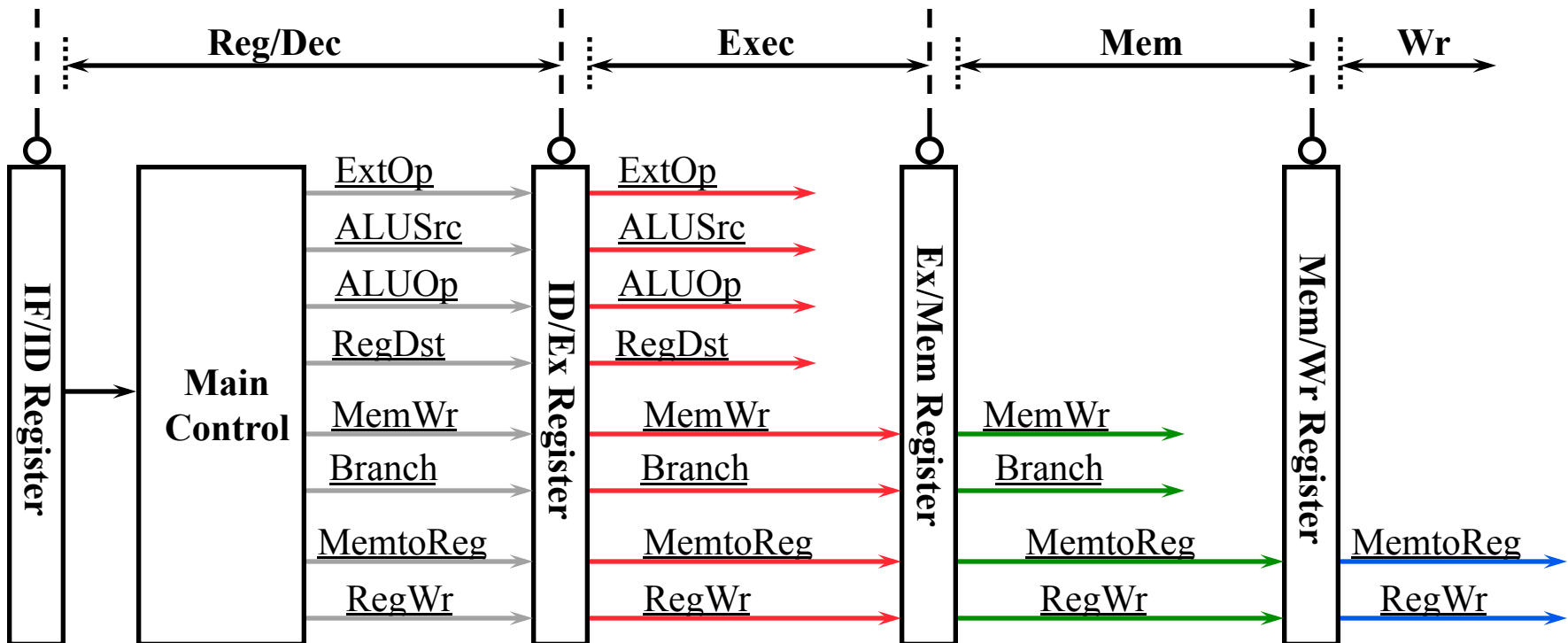
Lecture 12
Pipeline Control

Review: A Pipelined Datapath

Review: Pipeline Control "Data Stationary Control"

° **The Main Control generates the control signals during Reg/Dec**
- **Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later**
- **Control signals for Mem (MemWr Branch) are used 2 cycles later**
- **Control signals for Wr (MemtoReg MemWr) are used 3 cycles later**

Review: Pipeline Summary

° **Pipeline Processor:**

- **Natural enhancement of the multiple clock cycle processor**

- **Each functional unit can only be used once per instruction**

- **If an instruction is going to use a functional unit:**

    - **it must use it at the same stage as all other instructions**

- **Pipeline Control:**

    - **Each stage's control signal depends ONLY on the instruction that is currently in that stage**
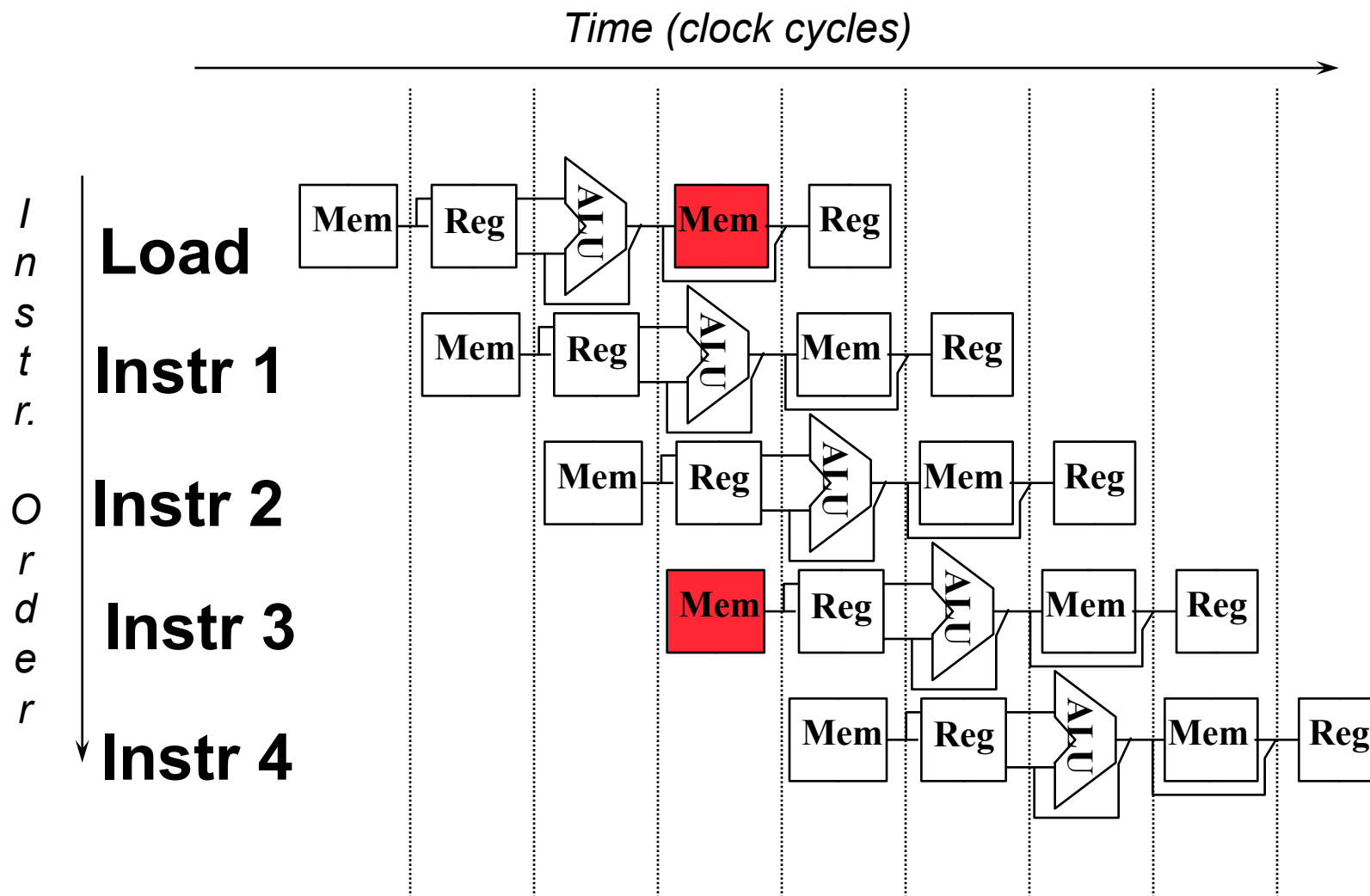
Outline of Today's Lecture

° **Recap and Introduction**

° **Introduction to Hazards**

° **Forwarding (aka bypassing)**

° **1 cycle Load Delay**

° **1 cycle Branch Delay**

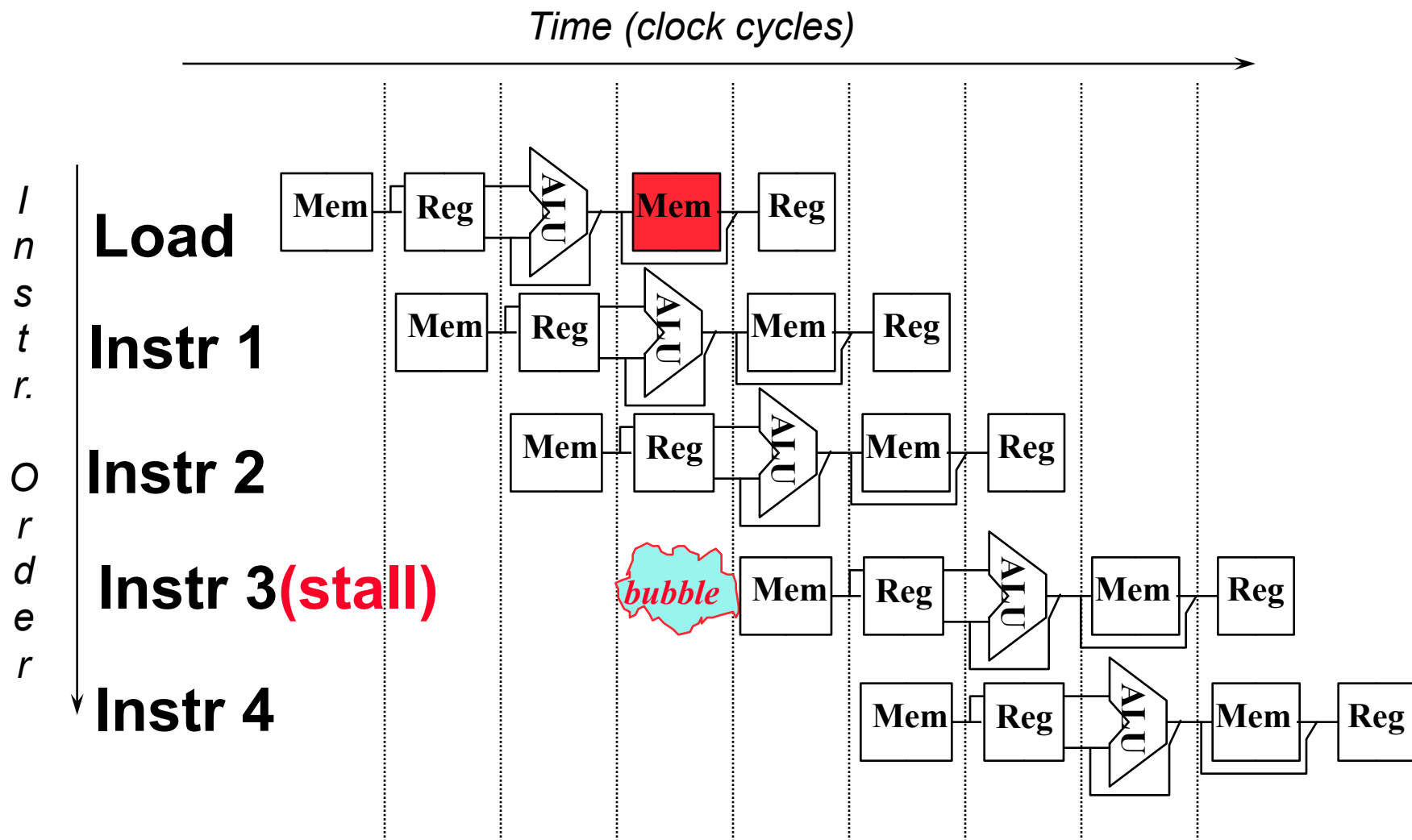° **What makes pipelining hard**

° **Summary**

Its not that easy for computers

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

- **structural hazards: HW cannot support this combination of instructions**

- **data hazards: instruction depends on result of prior instruction still in the pipeline**

- **control hazards: pipelining of branches & other instructions that change the PC**

° **Common solution is to stall the pipeline until the hazard is resolved, inserting one or more "bubbles" in the pipeline**
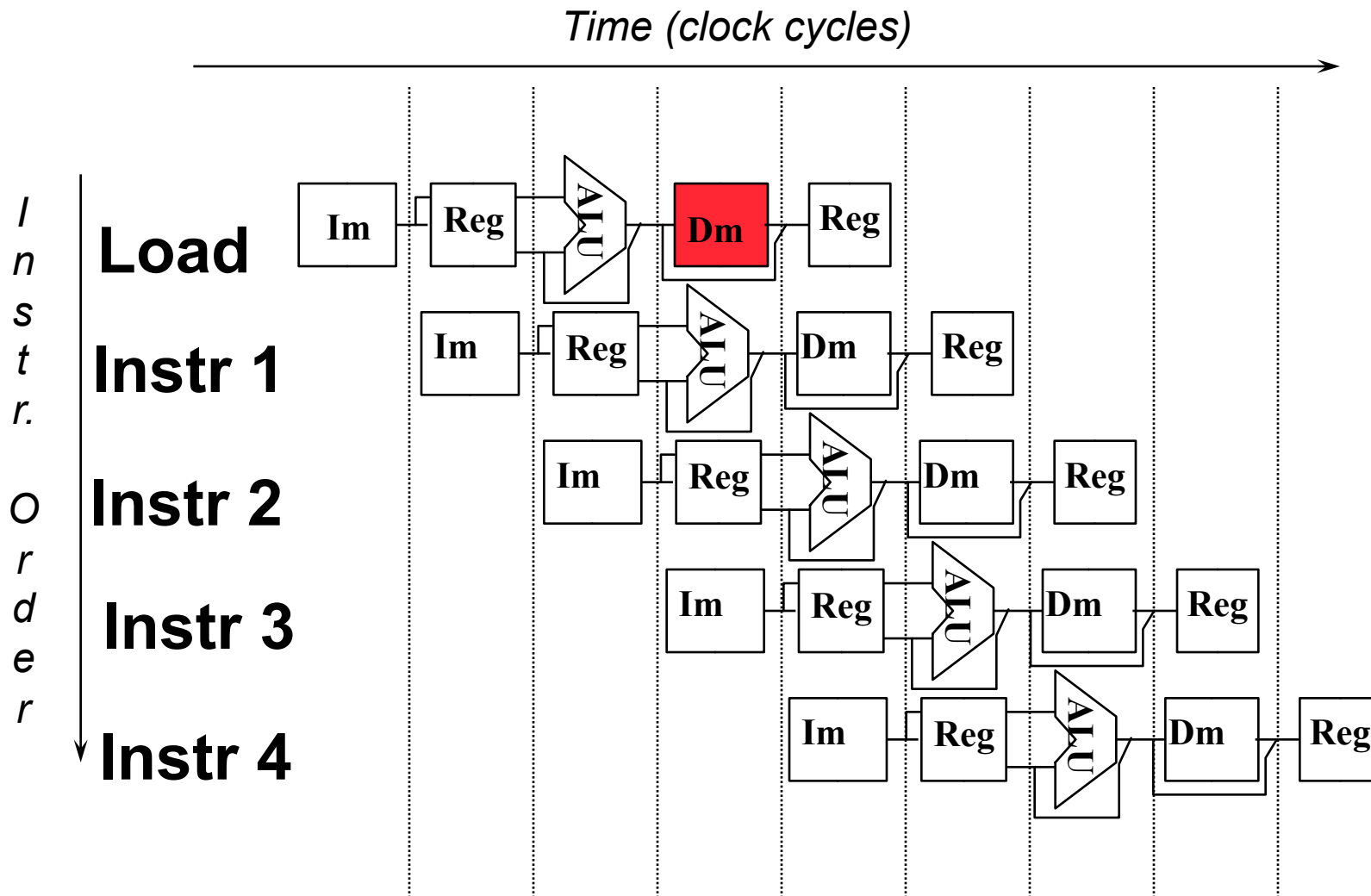
Single Memory is a Structural Hazard

Option 1: Stall to resolve Memory Structural Hazard

Time (clock cycles)

I
n
s
t
r.

O
r
d
e
r

Load

Instr 1

Instr 2

Instr 3 (stall)

Instr 4

Mem — Reg — ALU — Mem — Reg

Mem — Reg — ALU — Mem — Reg

Mem — Reg — ALU — Mem — Reg

bubble — Mem — Reg — ALU — Mem — Reg

Mem — Reg — ALU — Mem — Reg

Option 2: Duplicate to Resolve Structural Hazard

- **Separate Instruction Cache (Im) & Data Cache (Dm)**

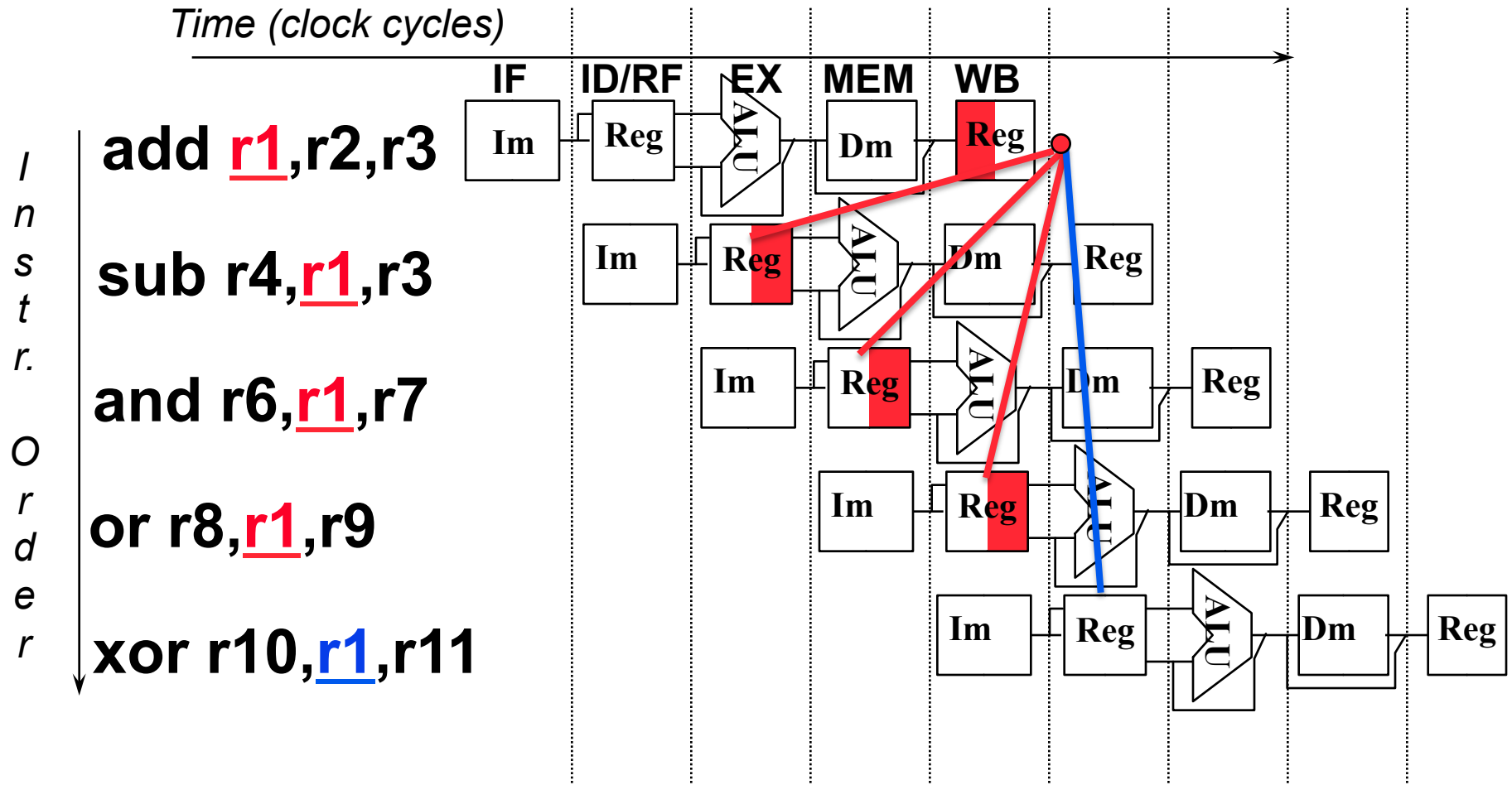**Data Hazard on r1**

add <u>r1</u> ,r2,r3

sub r4, <u>r1</u> ,r3

and r6, <u>r1</u> ,r7

or r8, <u>r1</u> ,r9

xor r10, <u>r1</u> ,r11

Data Hazard on r1:

- **Dependencies backwards in time are hazards**

Time (clock cycles)

	IF	ID/RF	EX	MEM	WB

add **r1**,r2,r3

sub r4,**r1**,r3

and r6,**r1**,r7

or r8,**r1**,r9

xor r10,**r1**,r11

*I n s t r.   O r d e r*

Option1: HW Stalls to Resolve Data Hazard

- **Dependencies backwards in time are hazards**
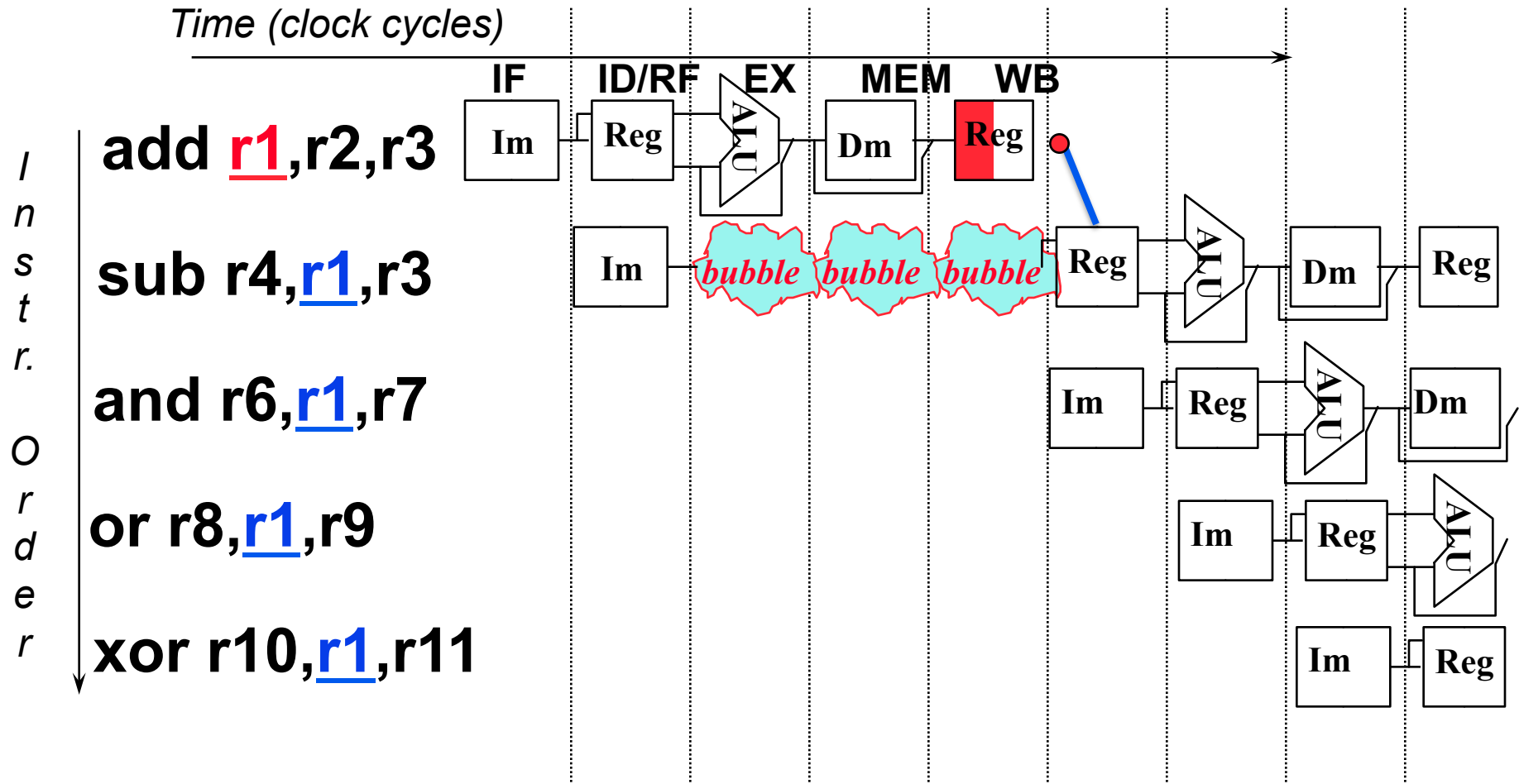
But recall use of "Data Stationary Control"

° **The Main Control generates the control signals during Reg/Dec**
- **Control signals for Exec (ExtOp, ALUSrc, ...) are used 1 cycle later**
- **Control signals for Mem (MemWr Branch) are used 2 cycles later**
- **Control signals for Wr (MemtoReg MemWr) are used 3 cycles later**
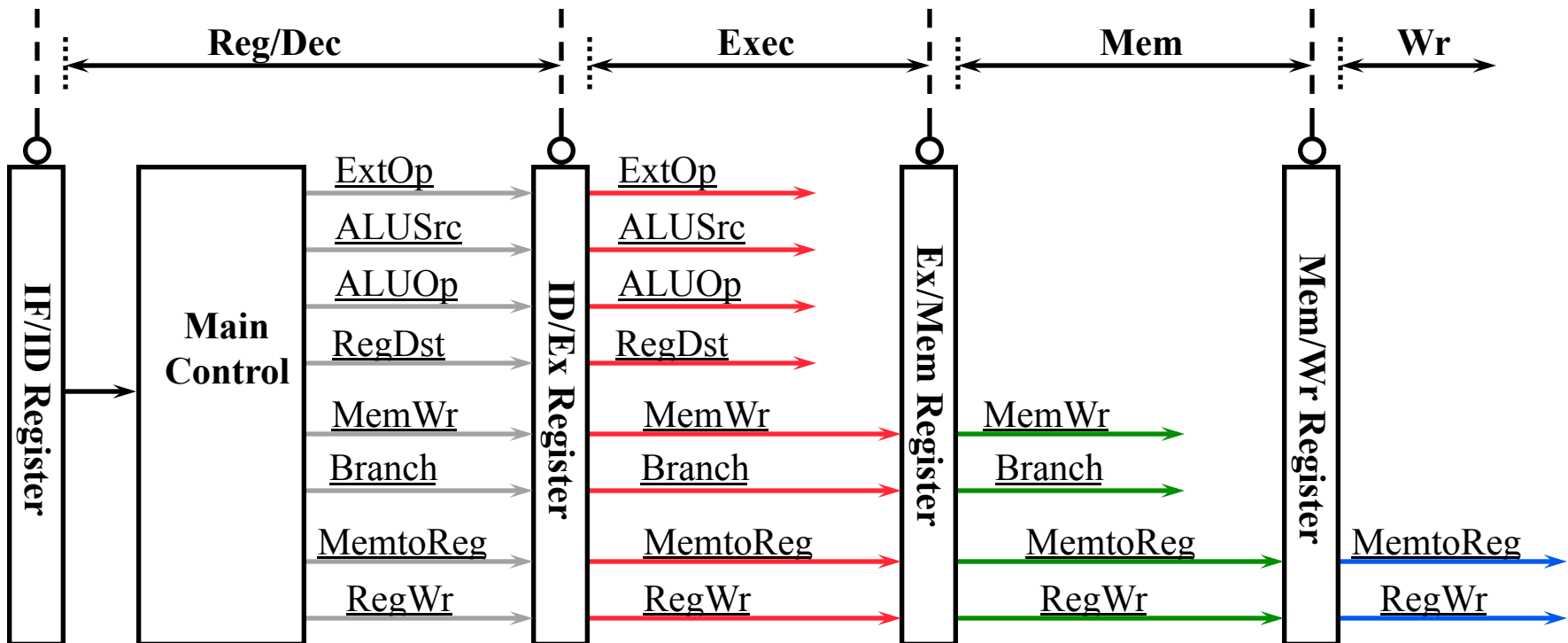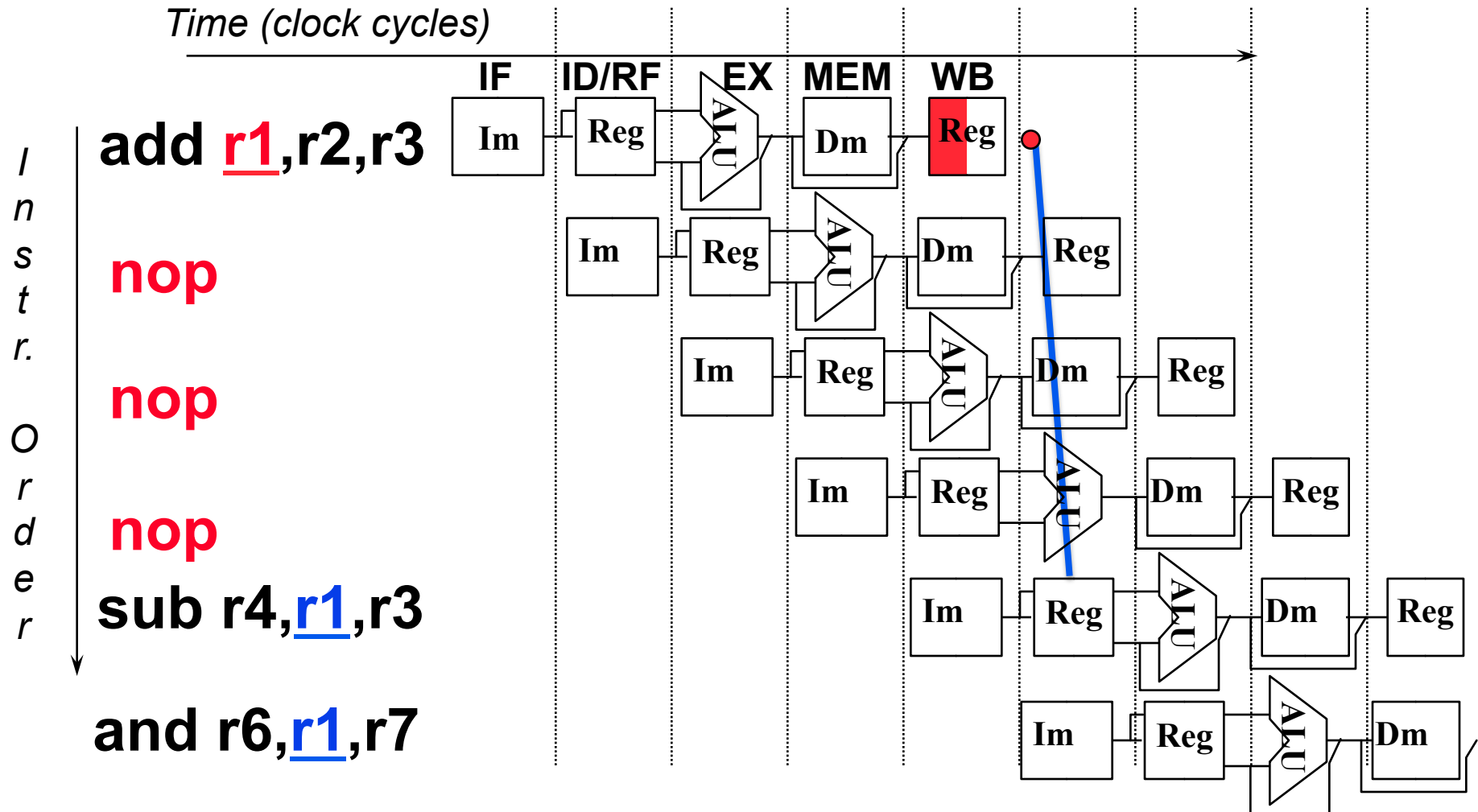
Option 1: How HW really stalls pipeline

- HW doesn't change PC => keeps fetching same instruction & sets control signals to benign values (0)
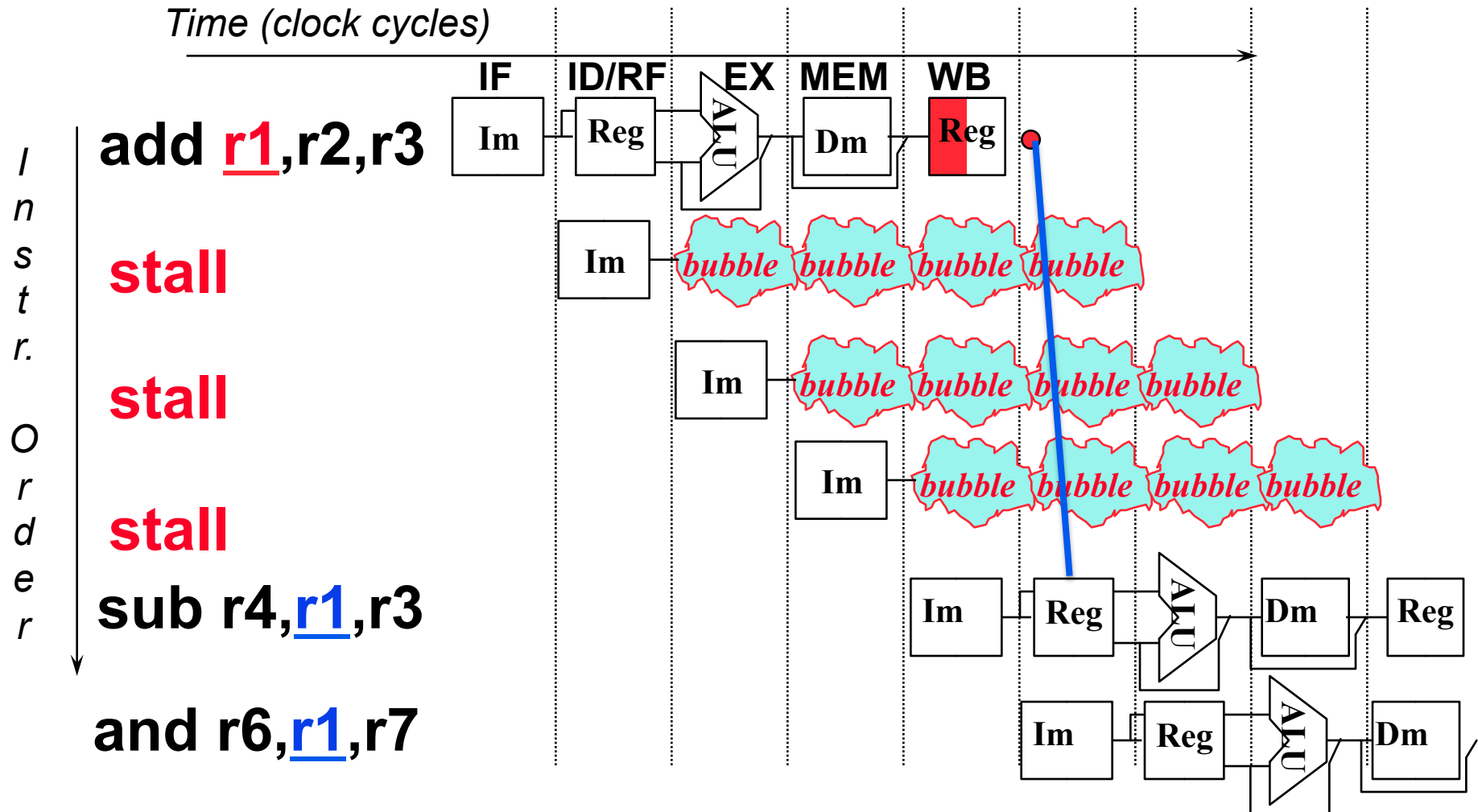
Option 2: SW inserts independent instructions

- **Worst case inserts NOP instructions**

Time (clock cycles)

| | IF | ID/RF | EX | MEM | WB |

add **r1**,r2,r3

Instr. Order

nop

nop

nop

sub r4,**r1**,r3

and r6,**r1**,r7

Option 3 Insight: Data is available!

- **Pipeline registers already contain needed data**

Time (clock cycles)

	IF	ID/RF	EX	MEM	WB

Instr. Order

add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

xor r10,r1,r11
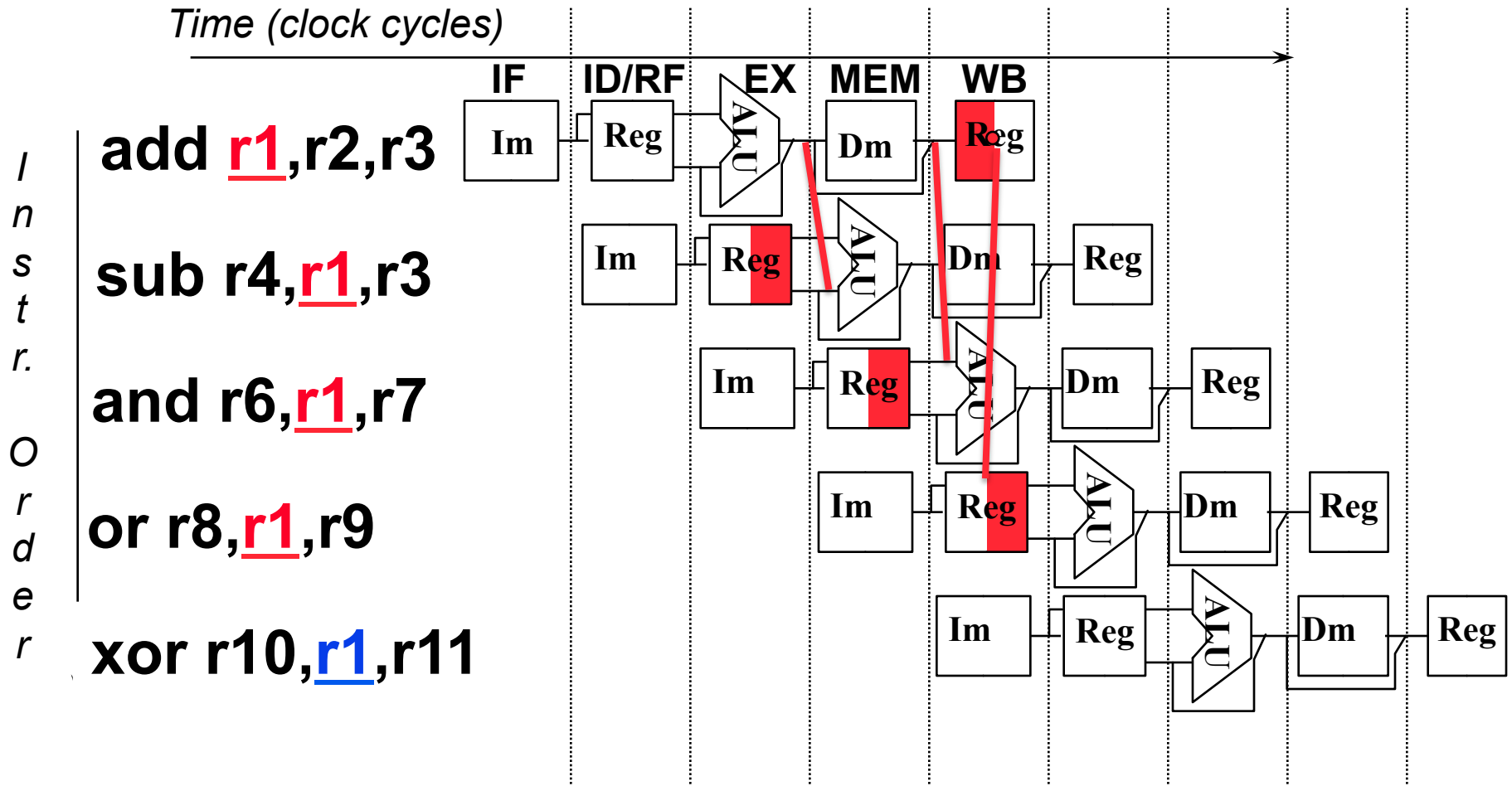
HW Change for "Forwarding" (aka Bypassing):

- **Increase multiplexors to add paths from pipeline registers**
- **Assumes register read during write gets new value (otherwise more results to be forwarded)**

ID/EX   EX/MEM   MEM/WB

Zero ?

Mux

ALU

Mux

Data Memory
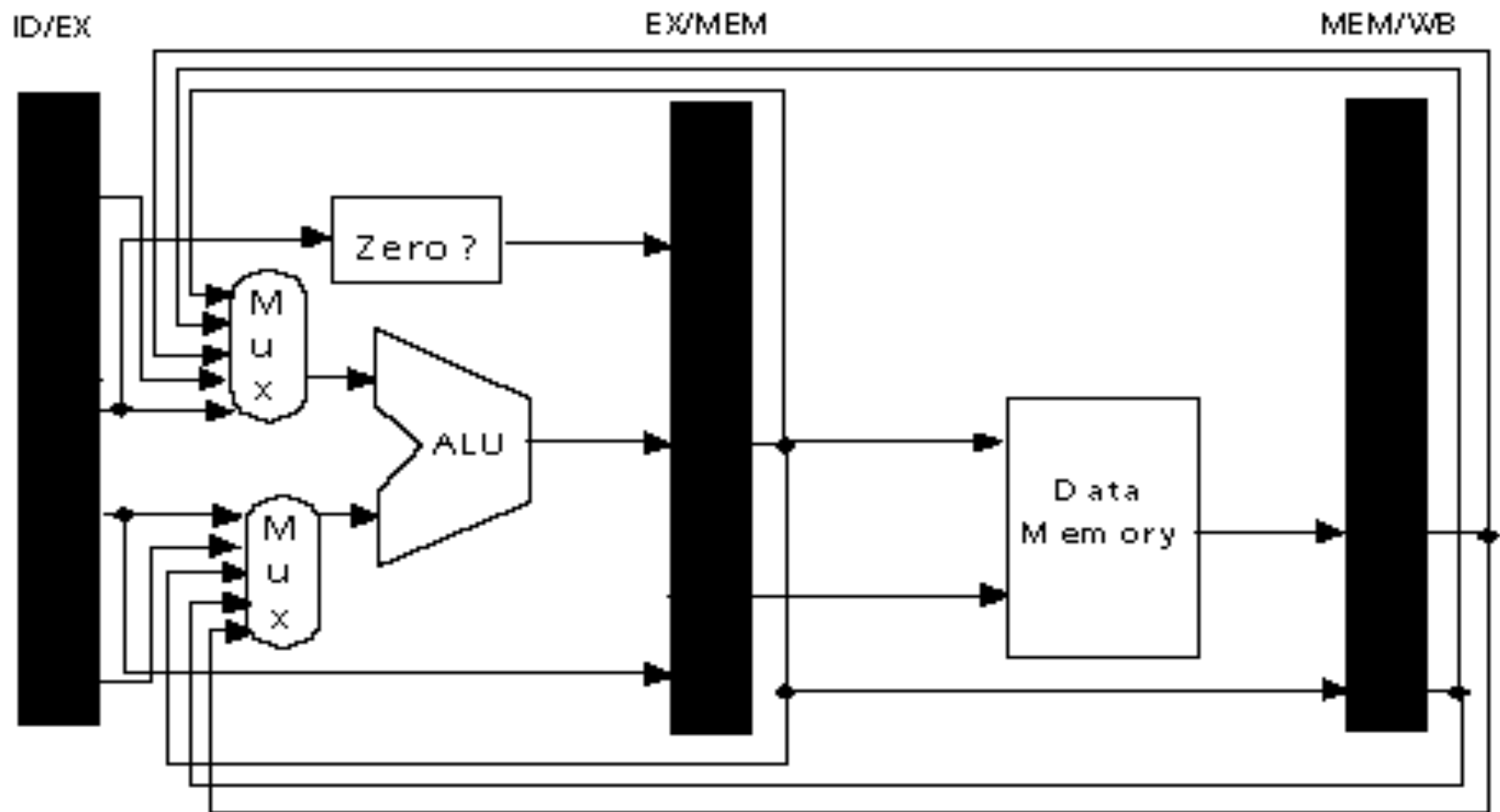
From Last Lecture: The Delay Load Phenomenon

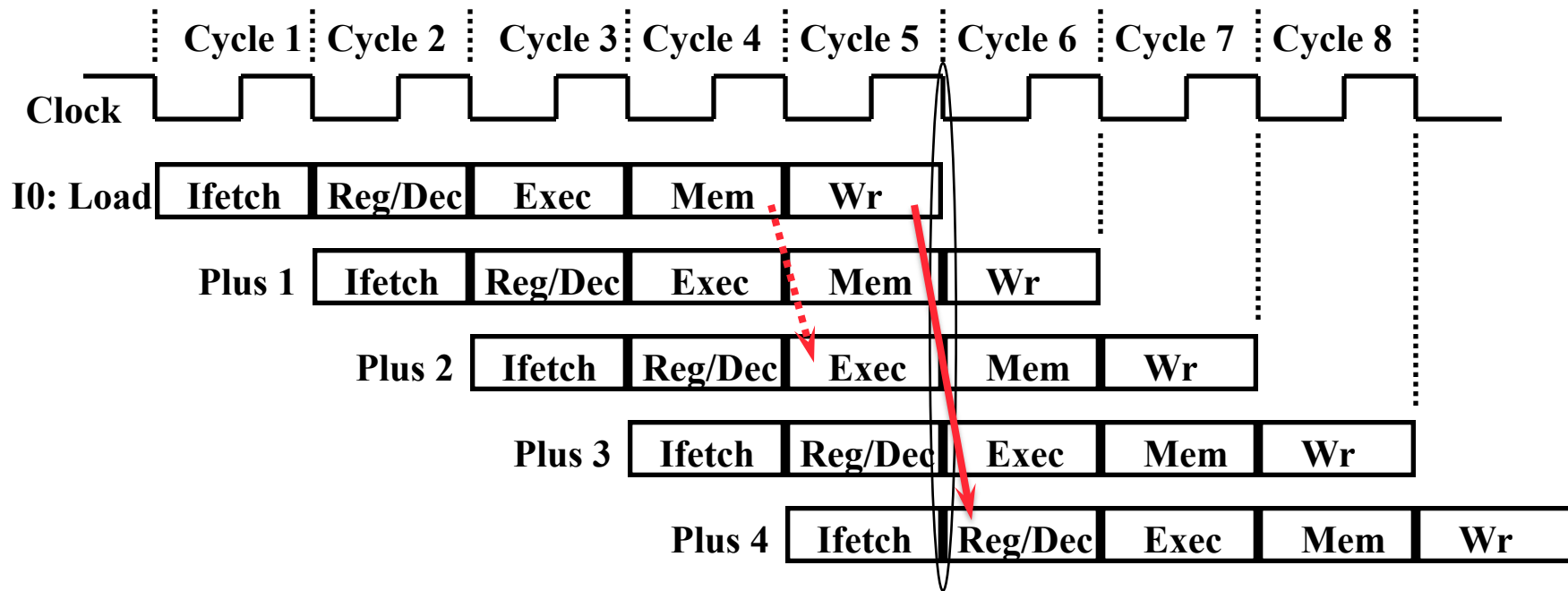|	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8

Clock

I0: Load | Ifetch | Reg/Dec | Exec | Mem | Wr |

Plus 1 | Ifetch | Reg/Dec | Exec | Mem | Wr |

Plus 2 | Ifetch | Reg/Dec | Exec | Mem | Wr |

Plus 3 | Ifetch | Reg/Dec | Exec | Mem | Wr |

Plus 4 | Ifetch | Reg/Dec | Exec | Mem | Wr |

° **Although Load is fetched during Cycle 1:**
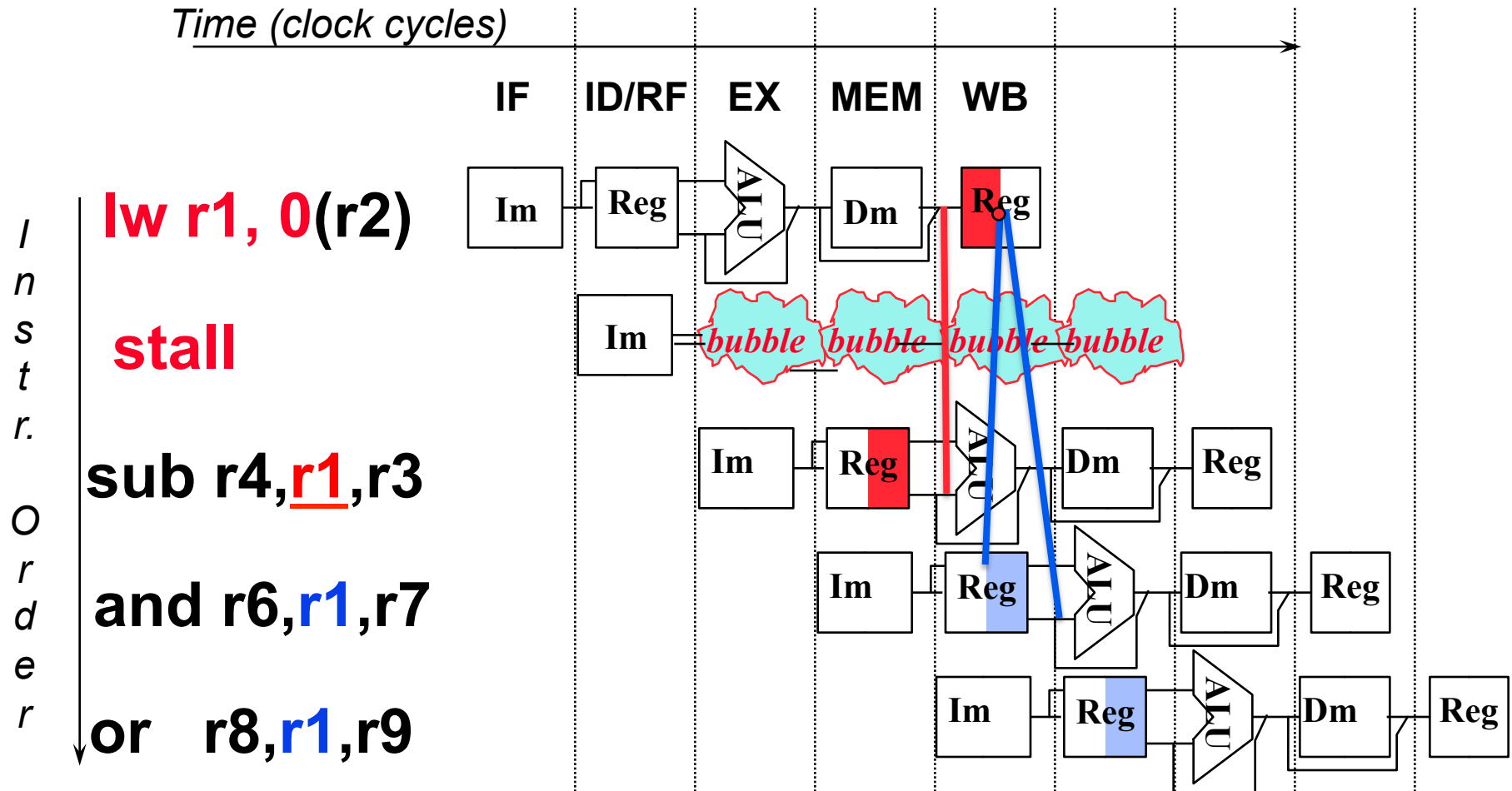
- **The data is NOT written into the Reg File until the end of Cycle 5**
- **We cannot read this value from the Reg File until Cycle 6**
- **3-instruction delay before the load take effect**

Forwarding reduces Data Hazard to 1 cycle:

Time (clock cycles)

IF   ID/RF   EX   MEM   WB

Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or   r8,r1,r9
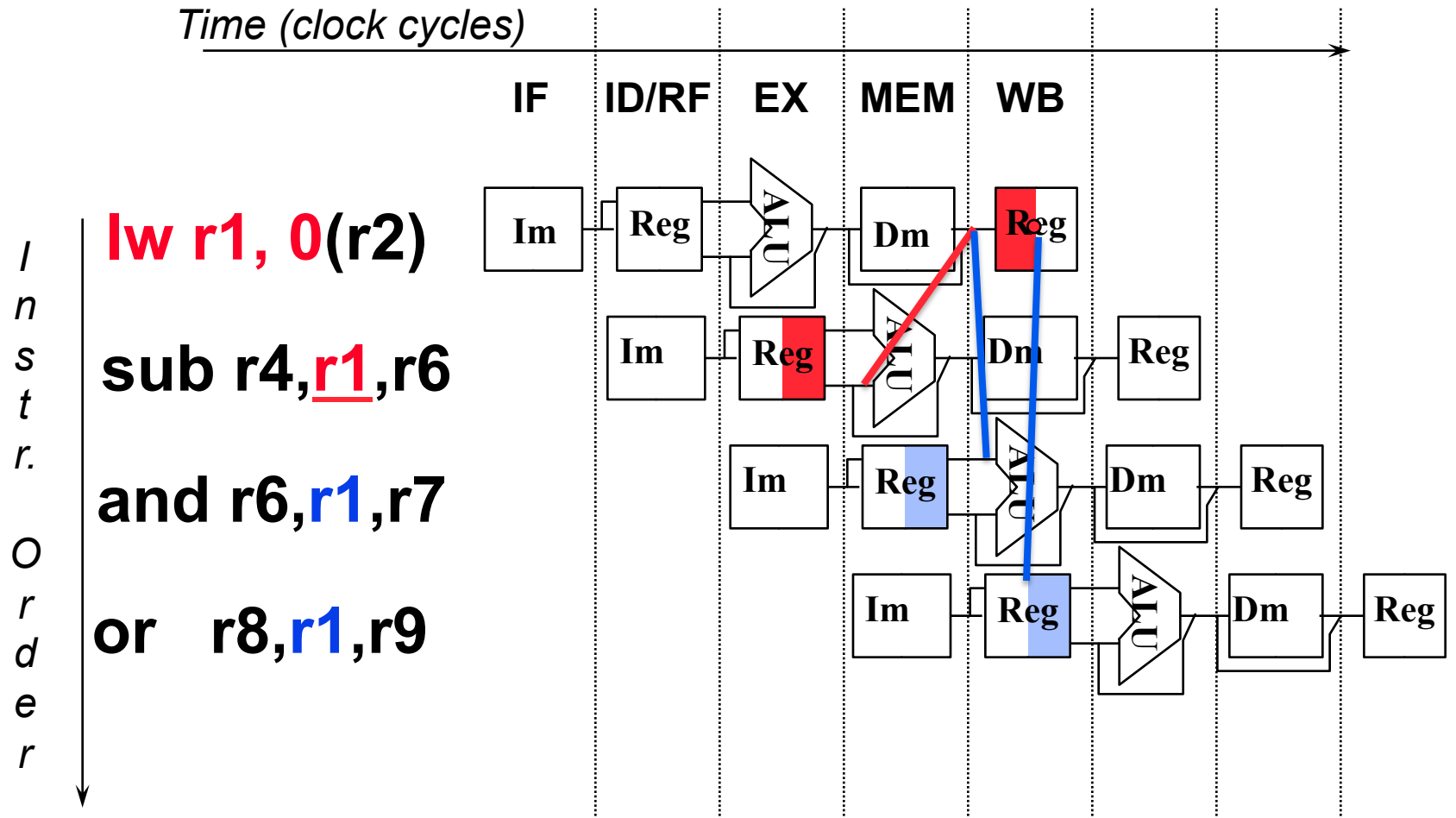
Option1: HW Stalls to Resolve Data Hazard

- "Interlock": checks for hazard & stalls

Time (clock cycles)

	IF	ID/RF	EX	MEM	WB

Instr. Order

lw r1, 0(r2)

 stall

sub r4,r1,r3

and r6,r1,r7

or   r8,r1,r9

Option 2: SW inserts independent instructions

- **Worst case inserts NOP instructions**
- **MIPS I solution: No HW checking**

Time (clock cycles)

	IF	ID/RF	EX	MEM	WB

lw r1, 0(r2)

nop

sub r4,r1,r3

and r6,r1,r7

or   r8,r1,r9

Instr. Order
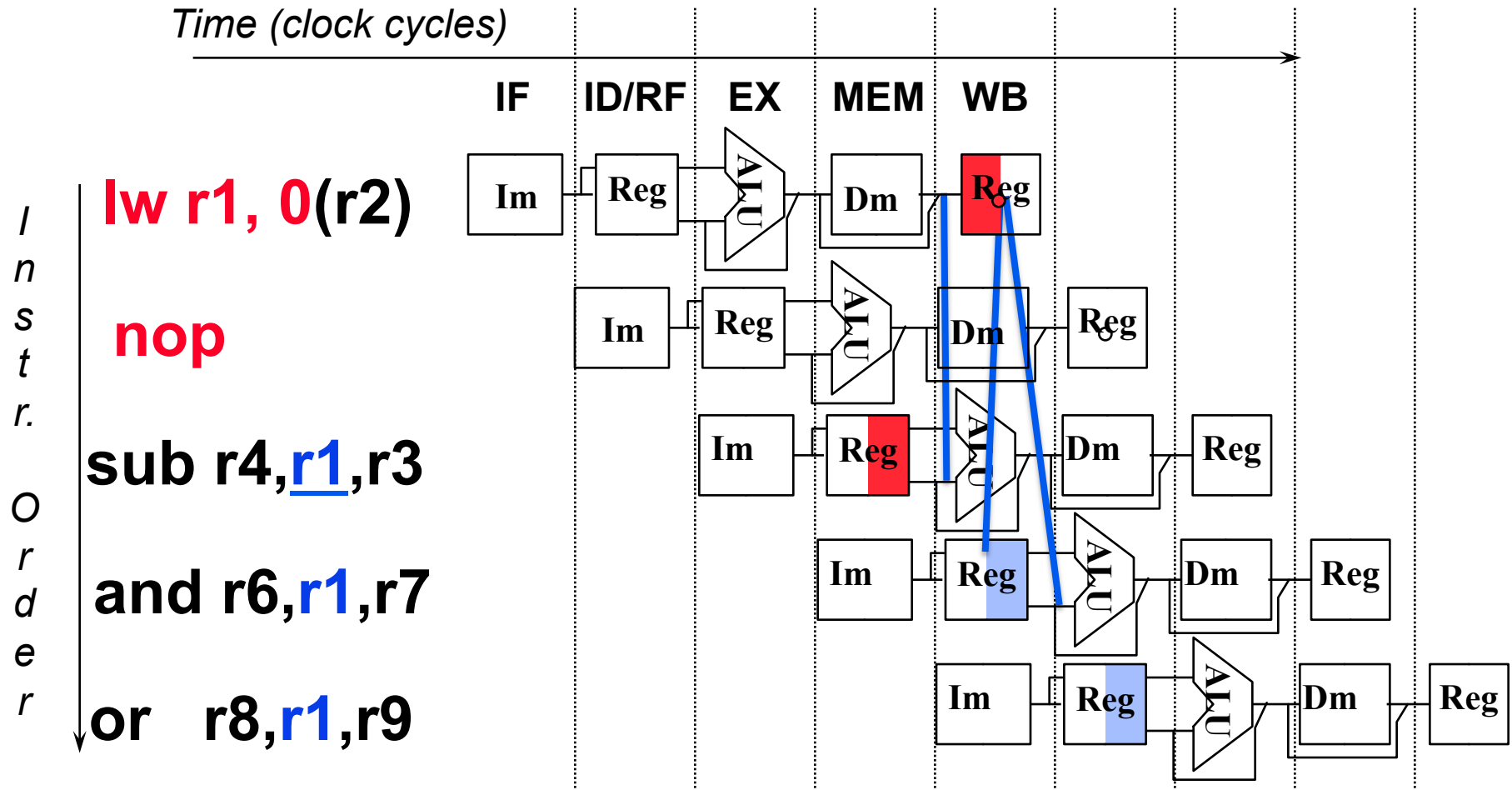
Software Scheduling to Avoid Load Hazards

Try producing fast code for

      a = b + c;

      d = e – f;

assuming a, b, c, d ,e, and f

in memory.
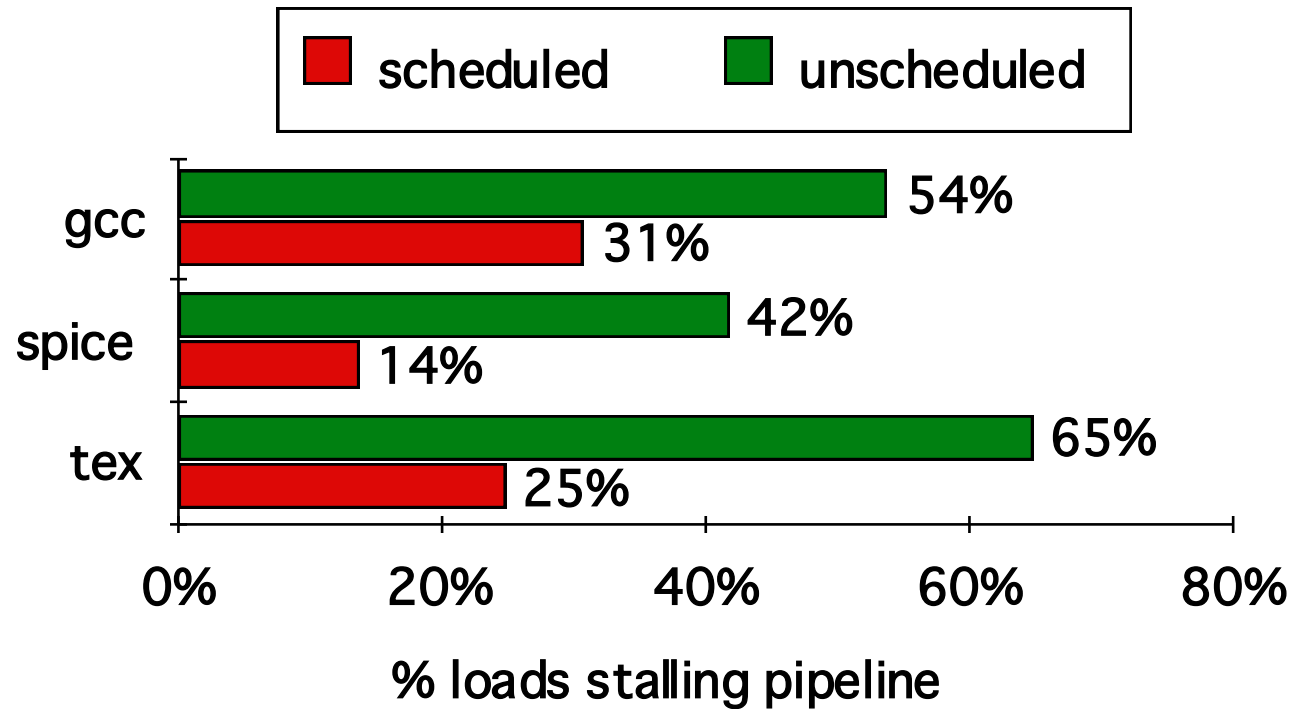
Slow code:

			Fast code:	
	LW	Rb,b	LW	Rb,b
Bubble {	LW	Rc,c	LW	Rc,c
	ADD	Ra,Rb,Rc	LW	Re,e
	SW	a,Ra	ADD	Ra,Rb,Rc
	LW	Re,e	LW	Rf,f
Bubble {	LW	Rf,f	SW	a,Ra
	SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
	SW	d,Rd	SW	d,Rd

Compiler Avoiding Load Stalls:

% loads stalling pipeline

From Last Lecture: The Delay Branch Phenomenon

	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9	Cycle 10	Cycle 11

Clk

12: Beq | Ifetch | Reg/Dec | Exec | Mem | Wr
(target is 1000)

16: R-type | Ifetch | Reg/Dec | Exec | Mem | Wr

20: R-type | Ifetch | Reg/Dec | Exec | Mem | Wr

24: R-type | Ifetch | Reg/Dec | Exec | Mem | Wr

**1000: Target of Br** | Ifetch | Reg/Dec | Exec | Mem | Wr
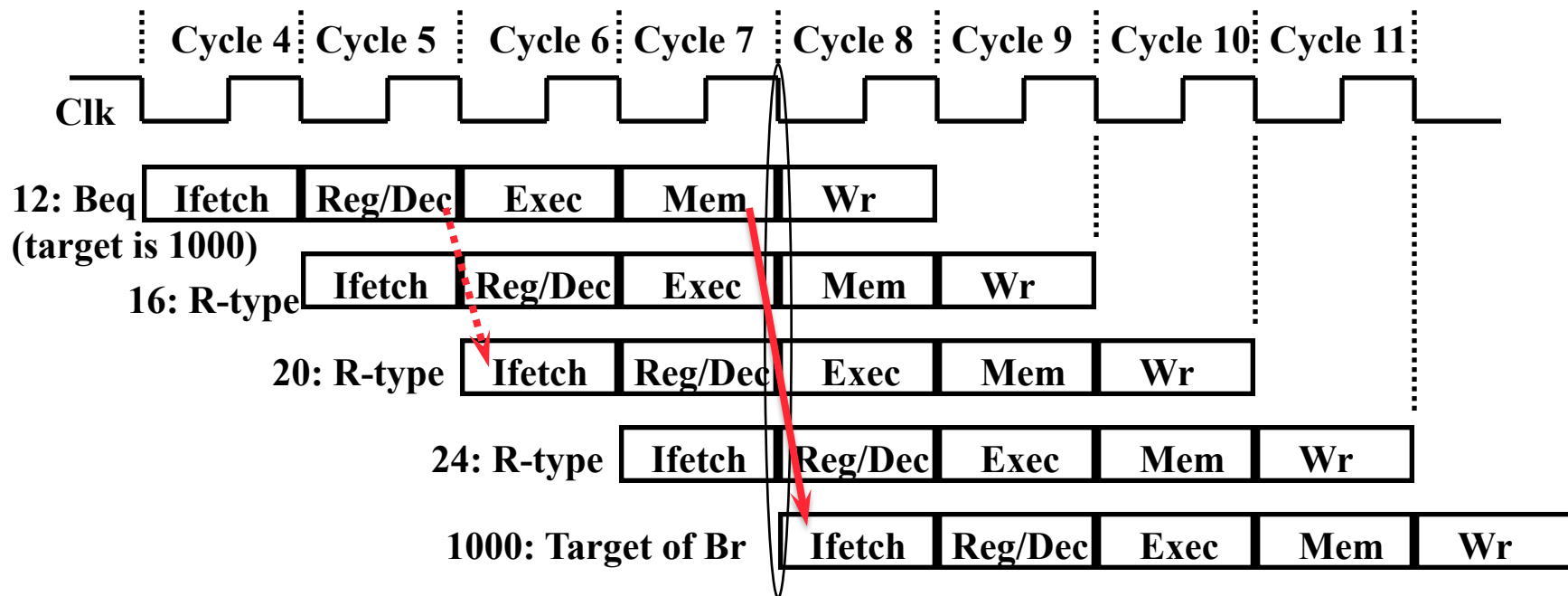
° **Although Beq is fetched during Cycle 4:**

- **Target address is NOT written into the PC until the end of Cycle 7**
- **Branch's target is NOT fetched until Cycle 8**
- **3-instruction delay before the branch take effect**

Branch Stall Impact

° **If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!**

° **2 part solution:**

  - **Determine branch taken or not sooner, AND**
  - **Compute taken branch address earlier**

° **MIPS branch tests = 0 or != 0**

° **Solution Step 1:**

  - **Move Zero test to ID/RF stage**
  - **Adder to calculate new PC in ID/RF stage**
  - **1 clock cycle penalty for branch vs. 3**

Step 1: move HW forward to reduce branch delay
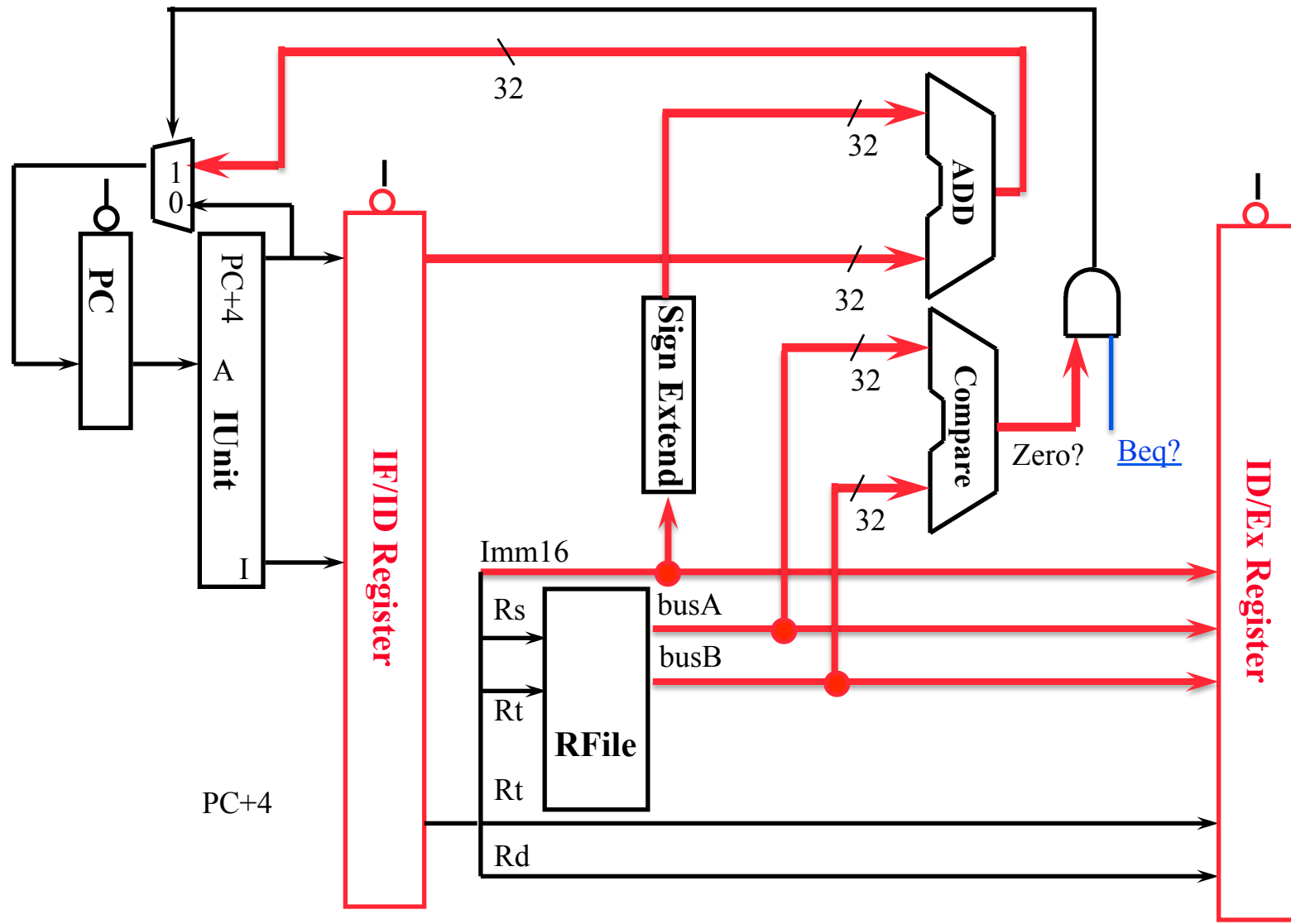
Step 2: Define Branch as Delayed

° **Worst case, SW inserts NOP into branch delay**

° **Where to get instructions to fill branch delay slot?**

- **Before branch instruction**
- **From the target address: only valuable when branch taken**
- **From fall through: only valuable when branch not taken**

° **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**

When is pipelining hard?

° **Interrupts: 5 instructions executing in 5 stage pipeline**

- **How to stop the pipeline?**
- **Restart?**
- **Who caused the interrupt?**

Stage	*Problem for which interrupts occur*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt (e.g., divide-by-zero)
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

When is pipelining hard?

° **Complex Addressing Modes and Instructions**

° **Address modes: Autoincrement causes register change during instruction execution**

- **Interrupts?**

- **Now worry about write hazards since write no longer last stage**

    - **Write After Read (WAR): Write occurs before independent read**

    - **Write After Write (WAW): Writes occur in wrong order, leaving wrong result in registers**

    - **(Previous data hazard called RAW, for Read After Write)**

° **Memory-memory Move instructions**

- **Multiple page faults**

- **make progress?**

When is pipelining hard?

° **Floating Point:** long execution time

° Also, may pipeline FP execution unit so that can initiate new instructions without waiting full latency

FP Instruction	Latency	Initiation Rate	(MIPS R4000)
Add, Subtract	4	3	
Multiply	8	4	
Divide	36	35	
Square root	112	111	
Negate	2	1	
Absolute value	2	1	
FP compare	3	2	

° Divide, Square Root take -10X to -30X longer than Add

- Exceptions?

- Adds WAR and WAW hazards since pipelines are no longer same length

# Hazard Detection

Suppose instruction *i* is about to be issued and a predecessor instruction *j* is in the instruction pipeline.

$$Rregs\ (\ i\ )\ =\ \text{Registers read by instruction } i$$

$$Wregs\ (\ i\ )\ =\ \text{Registers written by instruction } i$$

° **A RAW hazard exists on register ρ if $\exists\ \rho,\ \ \rho \in Rregs(\ i\ ) \cap Wregs(\ j\ )$**

- **Keep a record of pending writes (for inst's in the pipe) and compare with operand regs of current instruction.**

- **When instruction issues, reserve its result register.**

- **When on operation completes, remove its write reservation.**
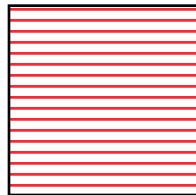
° **A WAW hazard exists on register ρ if $\exists\ \rho,\ \ \rho \in Wregs(\ i\ ) \cap Wregs(\ j\ )$**

° **A WAR hazard exists on register ρ if $\exists\ \rho,\ \ \rho \in Wregs(\ i\ ) \cap Rregs(\ j\ )$**

Avoiding Data Hazards by Design

Suppose instructions are executed in a pipelined fashion such that Instructions are initiated in order.

° *WAW avoidance:* if writes to a particular resource (e.g., reg) are performed in the same stage for all instructions, then no WAW hazards occur.

proof: writes are in the same time sequence as instructions.

I	R/D	E	W		
	I	R/D	E	W	
		I	R/D	E	W
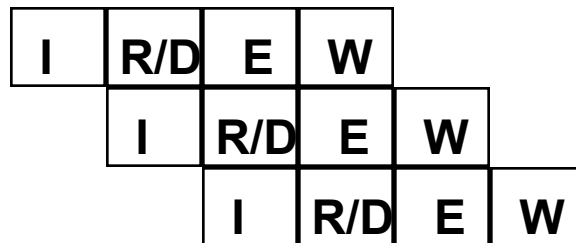
° *WAR avoidance:* if in all instructions, reads of a resource occur at an earlier stage than writes to that resource occur in any instruction, then no WAR hazards occur.

proof: A successor instruction must issue later, hence it will perform writes only after all reads for the current instruction.

First Generation RISC Pipelines

° **All instructions follow same pipeline order ("static schedule").**

° **Register write in last stage**

    – **Avoid WAW hazards**

° **All register reads performed in first stage after issue.**

    – **Avoid WAR hazards**

° **Memory access in stage 4.**

    – **Avoid all memory hazards**

° **Control hazards resolved by delayed branch (with fast path).**

° **RAW hazards resolved by bypass**

    – **Except RAW hazard on load, which is resolved by delayed load.**

Substantial pipelining with very little cost or complexity.

Machine organization is (slightly) exposed!

Relies very heavily on "hit assumption" of memory accesses in cache

Speedup and Pipeline Depth

$$Speedup = \frac{AvgInstructionTime_{unpipelined}}{AvgInstructionTime_{pipelined}}$$

$$= \frac{CPI_{unpipelined} \times ClockCycle_{unpipelined}}{CPI_{pipelined} \times ClockCycle_{pipelined}}$$

$$= \frac{CPI_{unpipelined}}{CPI_{ideal\_pipelined} + PipelineStallCyclesPerInstruction} \times \frac{ClockCycle_{unpipelined}}{ClockCycle_{pipelined}}$$

$$\approx \frac{PipelineDepth}{1 + PipelineStallCyclesPerInstruction} \times \frac{ClockCycle_{unpipelined}}{ClockCycle_{pipelined}}$$

Review: Summary of Pipelining Basics

° **Speedup & Pipeline Depth:**

$$Speedup \approx \frac{PipelineDepth}{1+PipelineStallsPerInstruction} \times \frac{ClockCycle_{unpipelined}}{CloclCycle_{pipelined}}$$

° **Hazards limit performance on computers:**

- **structural: need more HW resources**
- **data: need forwarding, compiler scheduling**
- **control: early evaluation & PC, delayed branch, prediction**

° **Increasing length of pipe increases impact of hazards since pipelining helps instruction bandwidth, not latency**

° **Compilers key to reducing cost of data and control hazards**

- **load delay slots**
- **branch delay slots**

° **Exceptions, Instruction Set, FP makes pipelining harder**

° **Longer pipelines => Branch prediction, more instruction parallelism?**