



CS 211: Computer Architecture

Instructor: Prof. Bhagi Narahari

Dept. of Computer Science

Course URL:

www.seas.gwu.edu/~narahari/cs211/

How to improve performance?

- Recall performance is function of
 - CPI: cycles per instruction
 - Clock cycle
 - Instruction count
- Reducing any of the 3 factors will lead to improved performance

How to improve performance?

- First step is to apply concept of pipelining to the instruction execution process
 - Overlap computations
- What does this do?
 - Decrease clock cycle
 - Decrease effective CPU time compared to original clock cycle
- Appendix A of Textbook
 - Also parts of Chapter 2

Pipeline Approach to Improve System Performance

- Analogous to fluid flow in pipelines and assembly line in factories
- Divide process into “stages” and send tasks into a pipeline
 - Overlap computations of different tasks by operating on them concurrently in different stages

Instruction Pipeline

- **Instruction execution process lends itself naturally to pipelining**
 - **overlap the subtasks of instruction fetch, decode and execute**

Linear Pipeline Processor

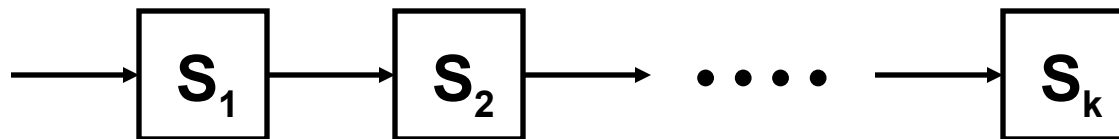
3

Linear pipeline processes a sequence of subtasks with linear precedence

At a higher level - Sequence of processors

Data flowing in streams from stage S_1 to the final stage S_k

Control of data flow : *synchronous* or *asynchronous*



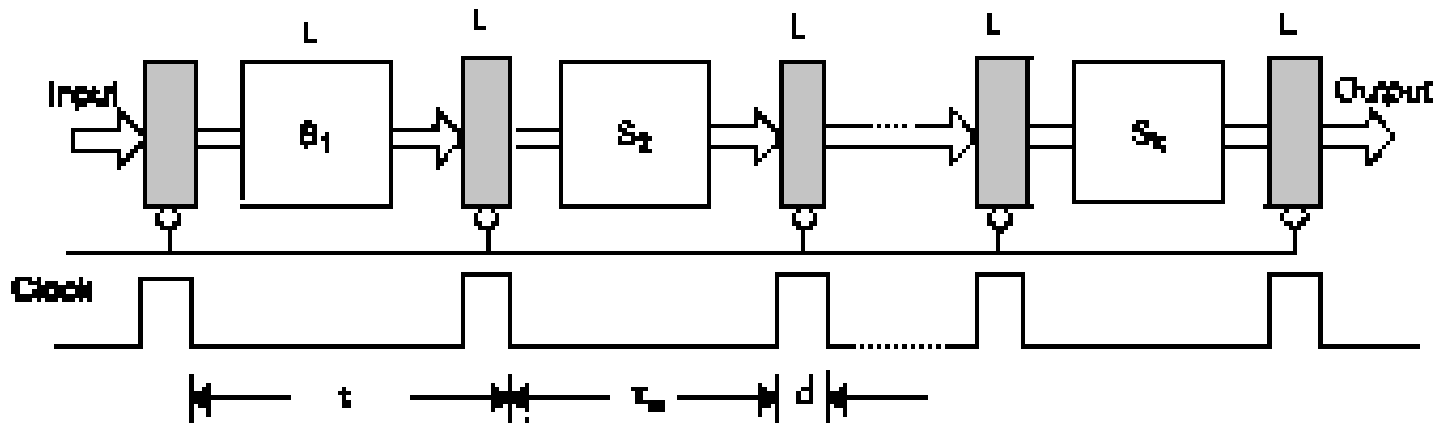
Synchronous Pipeline

4

All transfers simultaneous

One task or operation enters the pipeline per cycle

Processors reservation table : diagonal



Time Space Utilization of Pipeline

Full pipeline after 4 cycles

S3			T1	T2
S2		T1	T2	T3
S1	T1	T2	T3	T4
	1	2	3	4

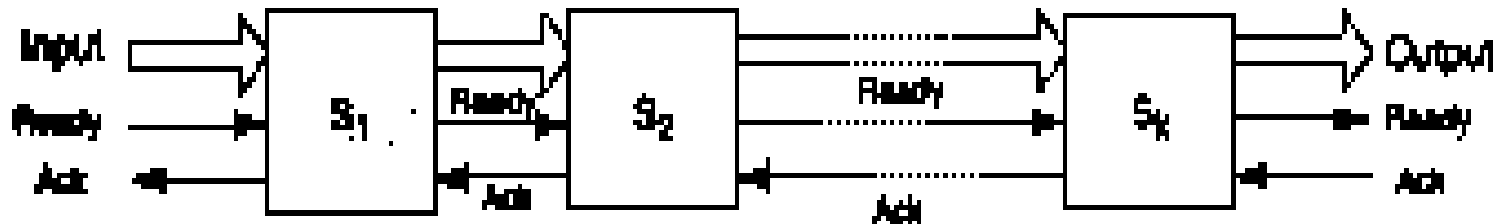
Time (in pipeline cycles)

Asynchronous Pipeline

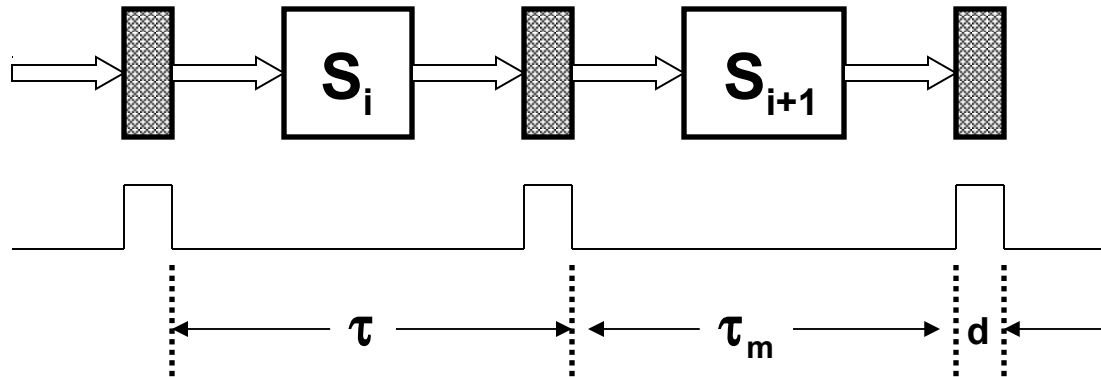
Transfers performed when individual processors are ready

Handshaking protocol between processors

Mainly used in multiprocessor systems with message-passing



Pipeline Clock and Timing



Clock cycle of the pipeline : τ

Latch delay : d

$$\tau = \max \{ \tau_m \} + d$$

Pipeline frequency : f

$$f = 1 / \tau$$

Speedup and Efficiency

k-stage pipeline processes *n* tasks in *k* + (*n*-1) clock cycles:

k cycles for the first task and *n*-1 cycles for the remaining *n*-1 tasks

Total time to process *n* tasks

$$T_k = [k + (n-1)] \tau$$

For the non-pipelined processor

$$T_1 = n k \tau$$

Speedup factor

$$S_k = \frac{T_1}{T_k} = \frac{n k \tau}{[k + (n-1)] \tau} = \frac{n k}{k + (n-1)}$$

Efficiency and Throughput

Efficiency of the k -stages pipeline :

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)}$$

Pipeline throughput (the number of tasks per unit time) :
note equivalence to IPC

$$H_k = \frac{n}{[k + (n-1)] \tau} = \frac{n f}{k + (n-1)}$$

Pipeline Performance: Example

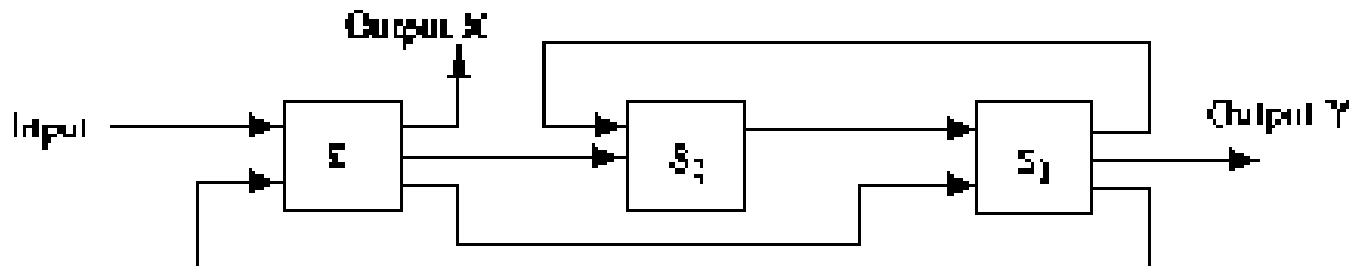
- Task has 4 subtasks with time: $t_1=60$, $t_2=50$, $t_3=90$, and $t_4=80$ ns (nanoseconds)
- latch delay = 10
- Pipeline cycle time = $90+10 = 100$ ns
- For non-pipelined execution
 - time = $60+50+90+80 = 280$ ns
- Speedup for above case is: $280/100 = 2.8$!!
- Pipeline Time for 1000 tasks = $1000 + 4-1 = 1003 \times 100$ ns
- Sequential time = 1000×280 ns
- Throughput = $1000/1003$
- What is the problem here ?
- How to improve performance ?

Non-linear pipelines and pipeline control algorithms

- Can have non-linear path in pipeline...
 - How to schedule instructions so they do no conflict for resources
- How does one control the pipeline at the microarchitecture level
 - How to build a scheduler in hardware ?
 - How much time does scheduler have to make decision ?

Non-linear Dynamic Pipelines

- Multiple processors (k -stages) as linear pipeline
- Variable functions of individual processors
- Functions may be dynamically assigned
- Feedforward and feedback connections



Reservation Tables

- **Reservation table : displays time-space flow of data through the pipeline analogous to opcode of pipeline**
 - Not diagonal, as in linear pipelines
- **Multiple reservation tables for different functions**
 - Functions may be dynamically assigned
- **Feedforward and feedback connections**
- **The number of columns in the reservation table : evaluation time of a given function**

Reservation Tables (Examples)

→ Time

		1	2	3	4	5	6	7	8
S1 S2 S3	S ₁	X					X		X
	S ₂		X		X				
	S ₃			X		X		X	

→ Time

		1	2	3	4	5	6
S1 S2 S3	S ₁	Y				Y	
	S ₂			Y			
	S ₃		Y		Y		Y

Latency Analysis

- ***Latency*** : the number of clock cycles between two initiations of the pipeline
- ***Collision*** : an attempt by two initiations to use the same pipeline stage at the same time
- Some latencies cause collision, some not

Collisions (Example)

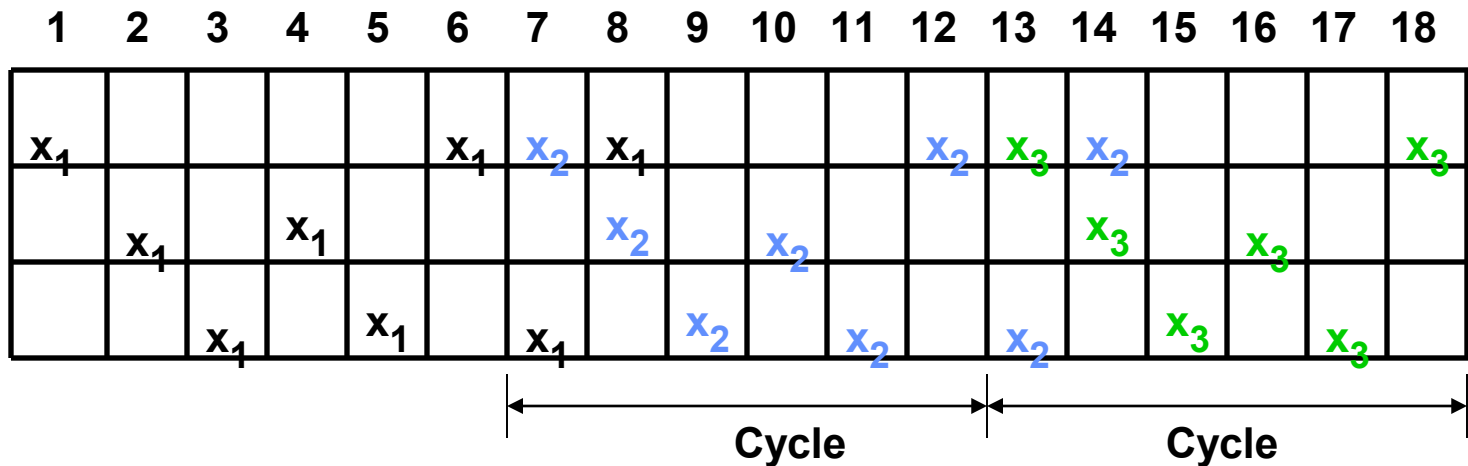
Time →

	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

1	2	3	4	5	6	7	8	9	10
x_1		x_2		x_3	x_1	x_4	x_1x_2		x_2x_3
	x_1		x_1x_2		x_2x_3		x_3x_4		x_4
		x_1		x_1x_2		$x_1x_2x_4$		$x_2x_3x_4$	

Latency = 2

Latency Cycle



Latency cycle : the sequence of initiations which has repetitive subsequence and without collisions

Latency sequence length : the number of time intervals within the cycle

Average latency : the sum of all latencies divided by the number of latencies along the cycle

Collision Free Scheduling

Goal : to find the shortest average latency

Lengths : for reservation table with n columns, maximum forbidden latency is $m \leq n - 1$, and permissible latency p is
 $1 \leq p \leq m - 1$

Ideal case : $p = 1$ (static pipeline)

Collision vector : $C = (C_m C_{m-1} \dots C_2 C_1)$

[$C_i = 1$ if latency i causes collision]

[$C_i = 0$ for permissible latencies]

Collision Vector

Reservation Table

x					x
	x		x		
		x		x	

x₁					x₁
	x₁		x₁		
		x₁		x₁	

Value x_1

x₂					x₂
	x₂		x₂		
		x₂		x₂	

Value x_2

$$C = (? \ ? \ \dots \ ? \ ?)$$

Back to our focus: Computer Pipelines

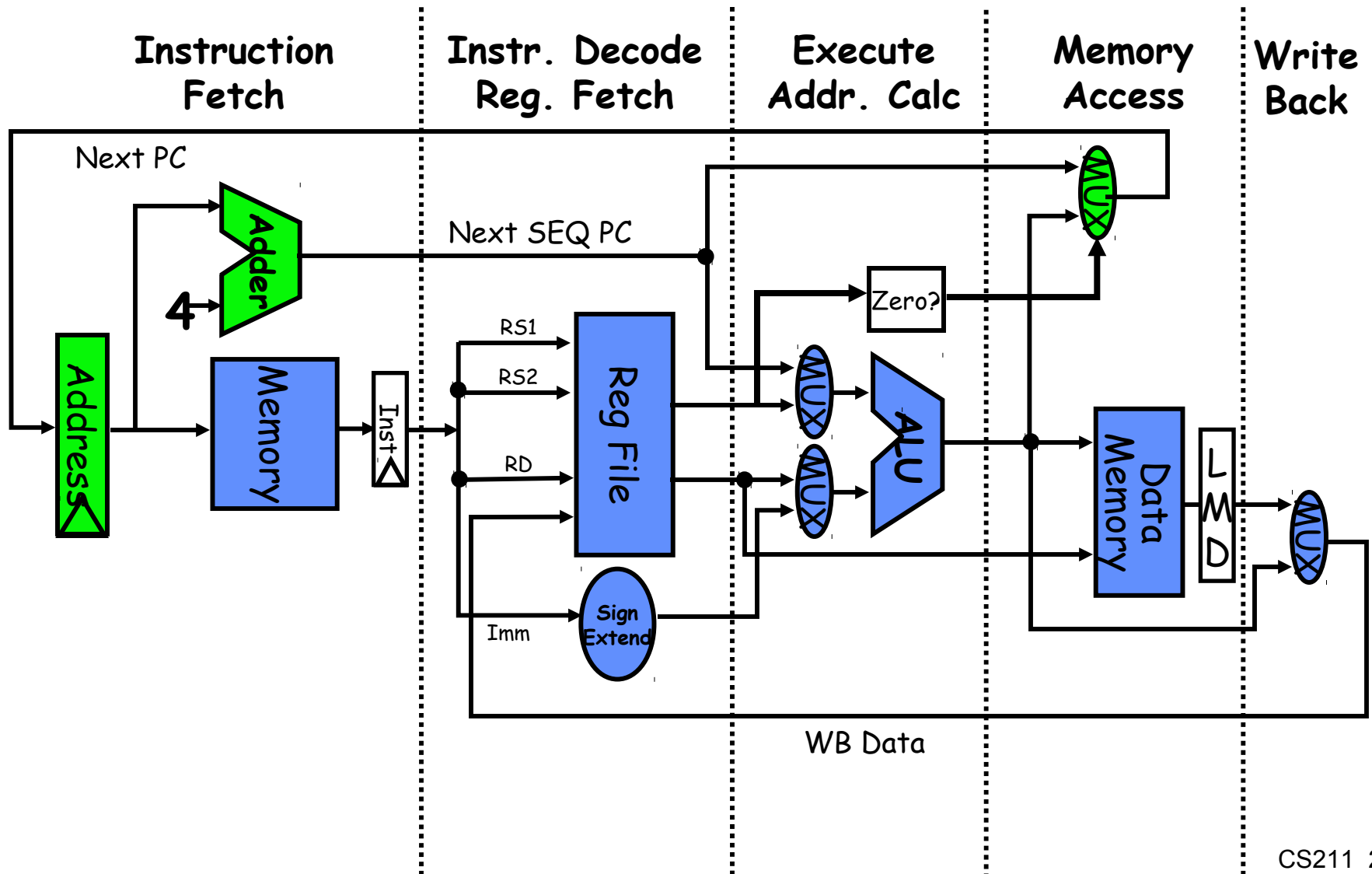
- Execute **billions of instructions**, so throughput is what matters
- MIPS desirable features:
 - all instructions same length,
 - registers located in same place in instruction format,
 - memory operands only in loads or stores

Designing a Pipelined Processor

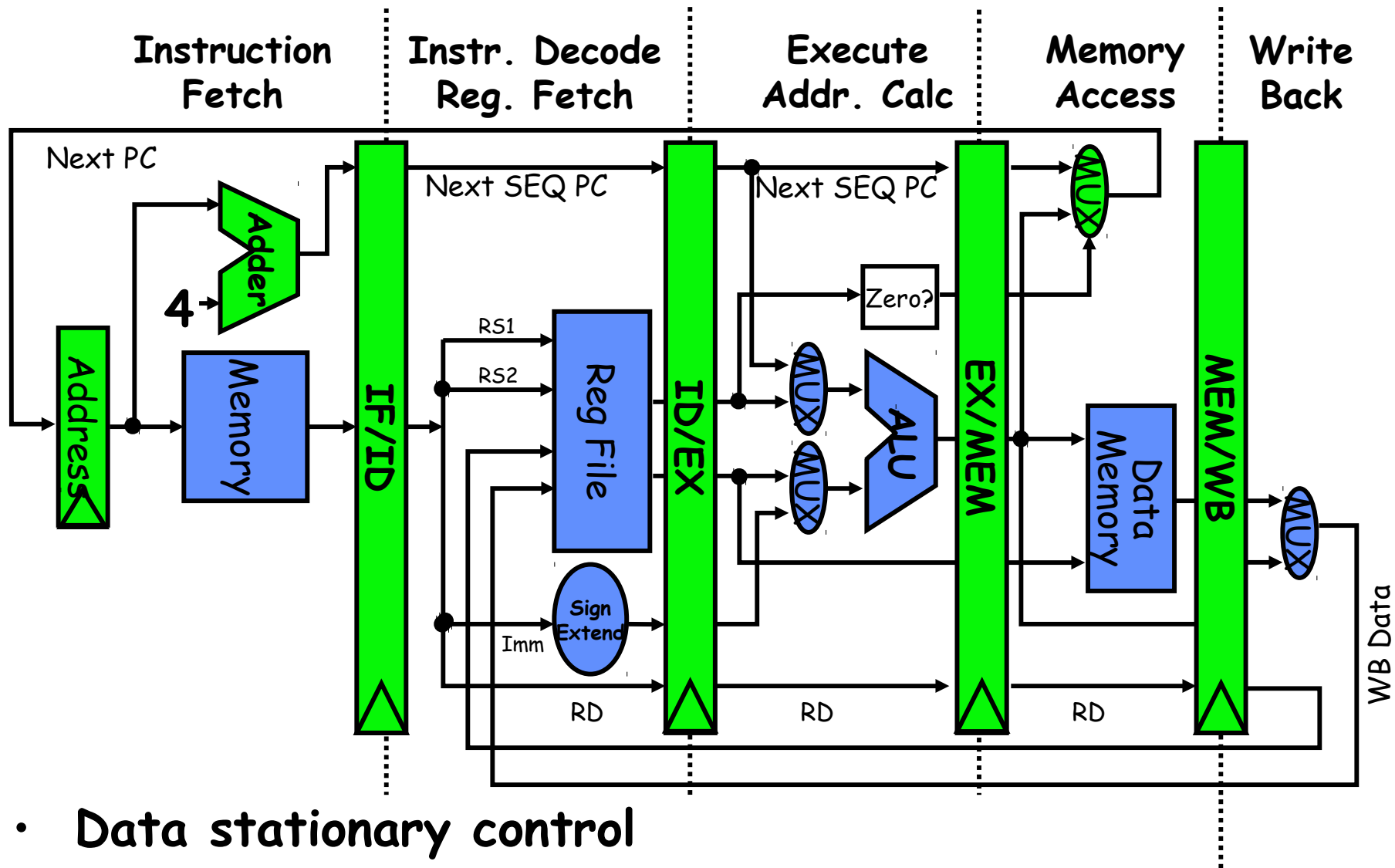
- Go back and examine your datapath and control diagram
- associated resources with states
- ensure that flows do not conflict, or figure out how to resolve
- assert control in appropriate stage

5 Steps of MIPS Datapath

What do we need to do to pipeline the process ?

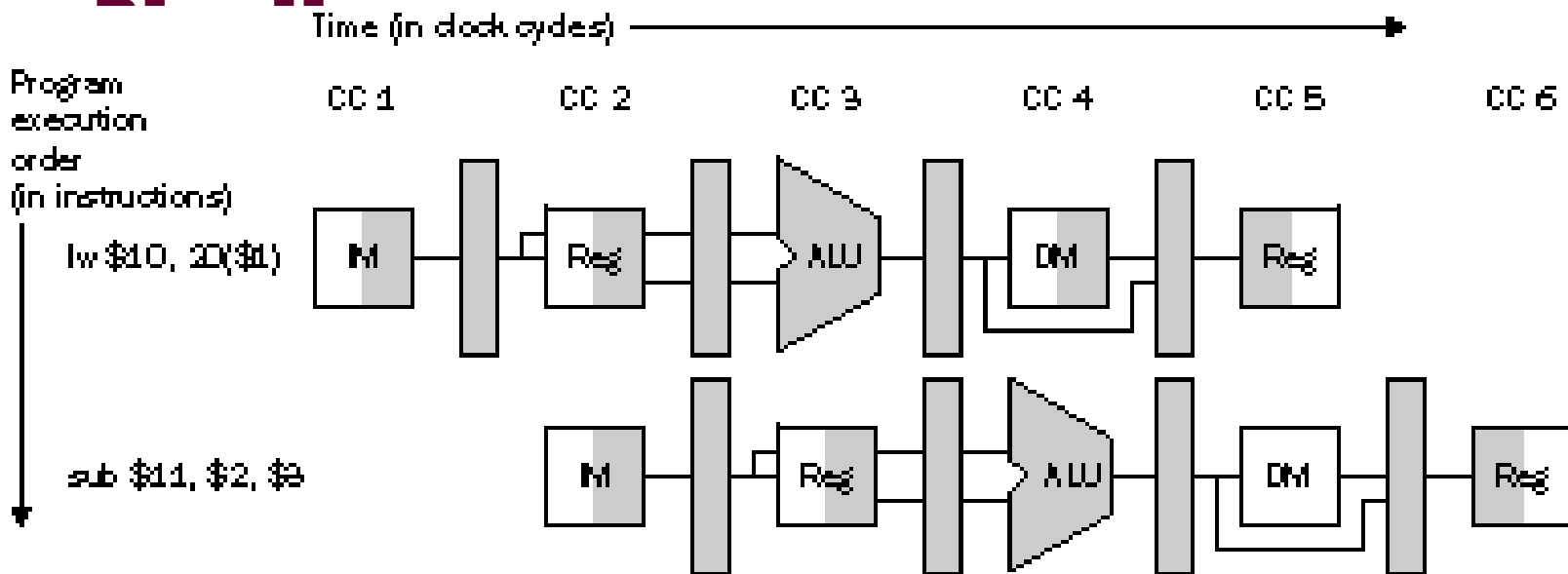


5 Steps of MIPS/DLX Datapath



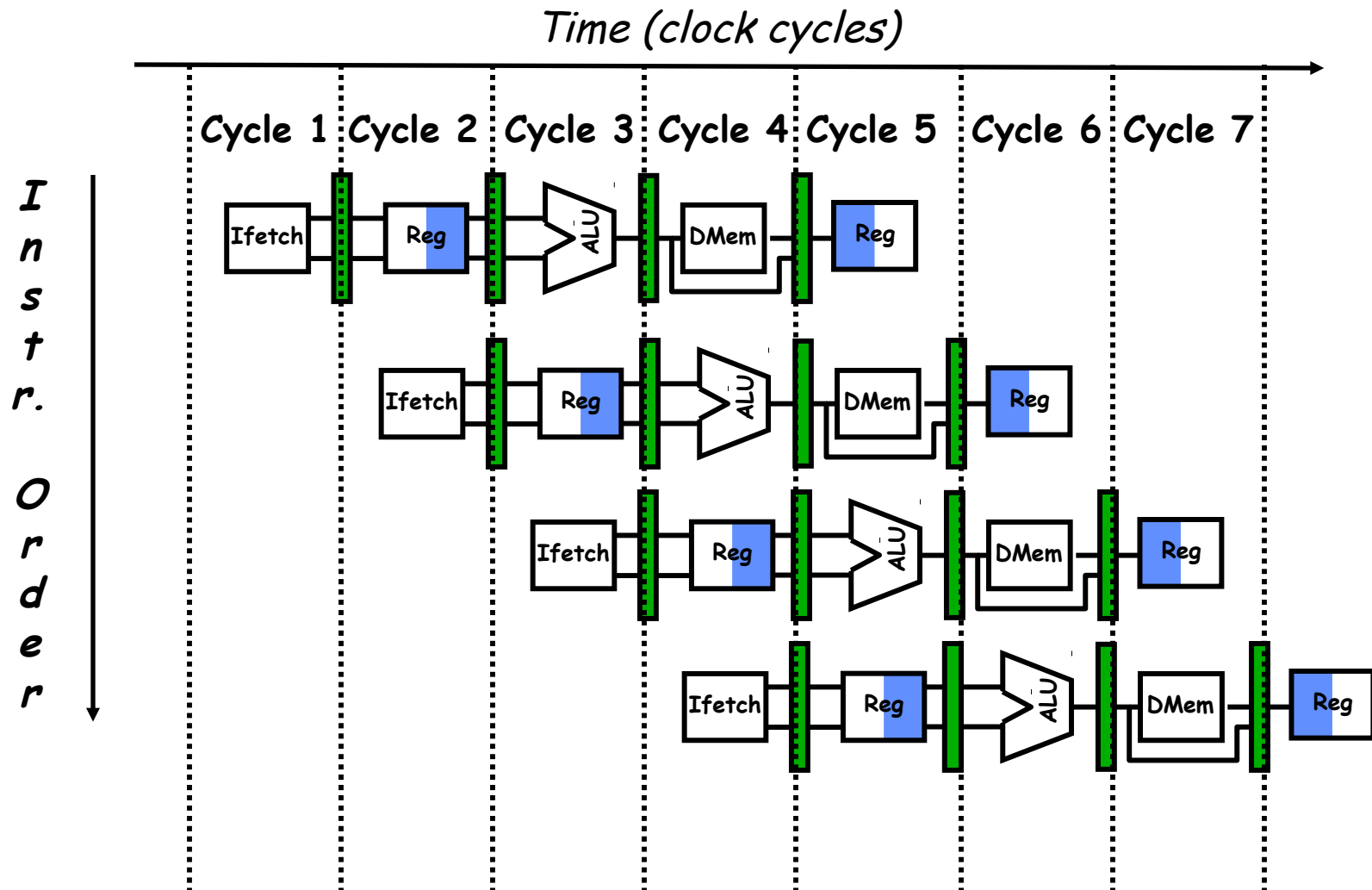
- Data stationary control
 - local decode for each instruction phase / pipeline stage

Graphically Representing

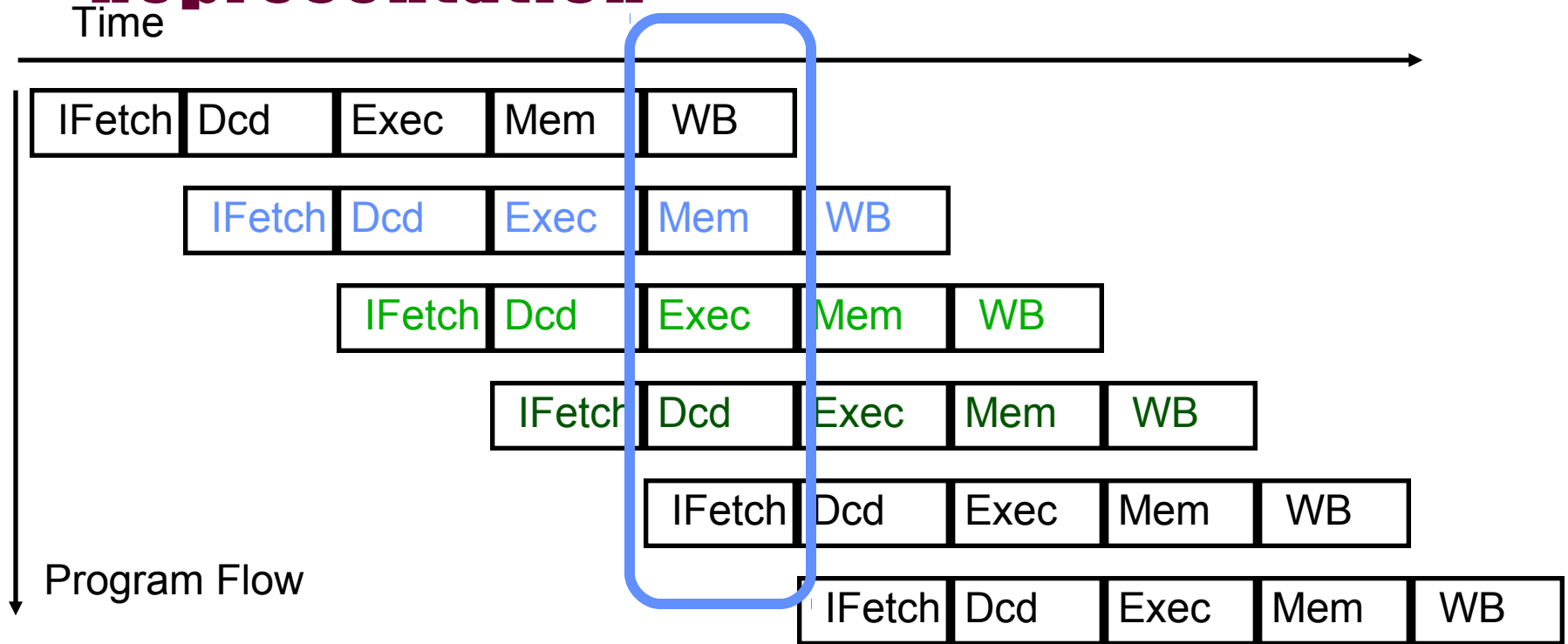


- Can help with answering questions like:
 - how many cycles does it take to execute this code?
 - what is the ALU doing during cycle 4?
 - use this representation to help understand datapaths

Visualizing Pipelining



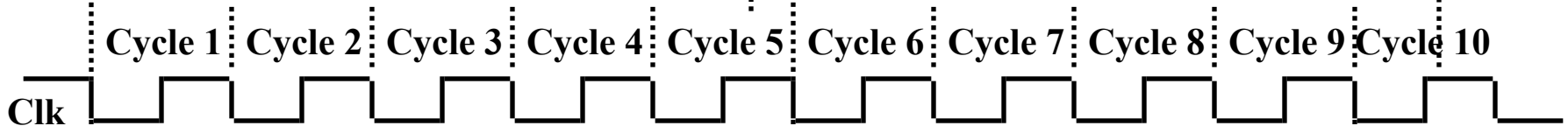
Conventional Pipelined Execution Representation



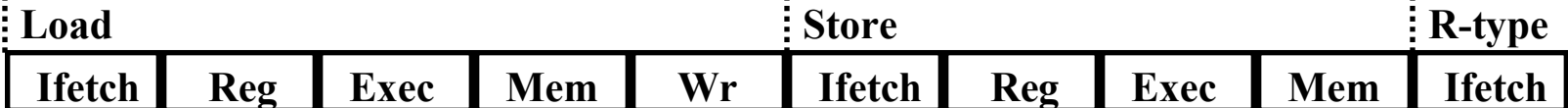
Single Cycle, Multiple Cycle, vs. Pipeline



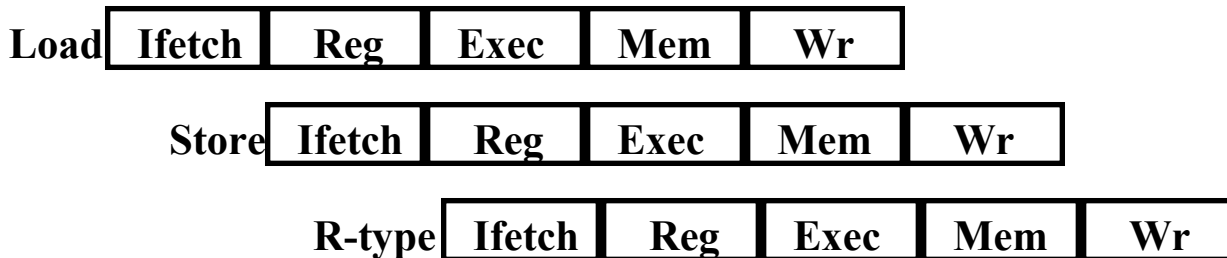
Single Cycle Implementation:



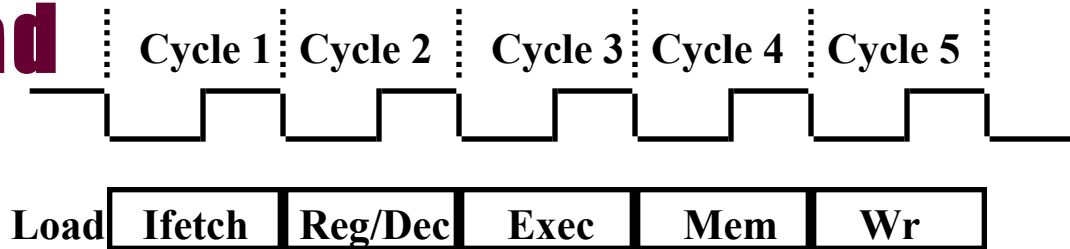
Multiple Cycle Implementation:



Pipeline Implementation:

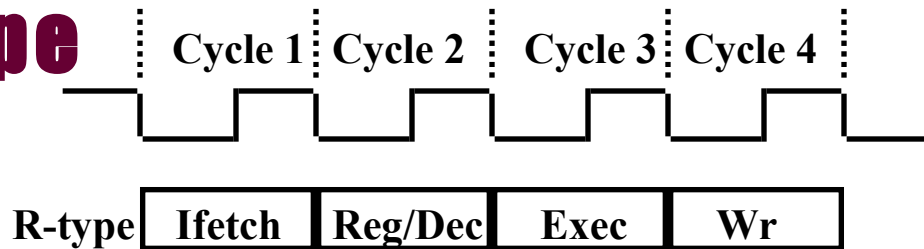


The Five Stages of Load



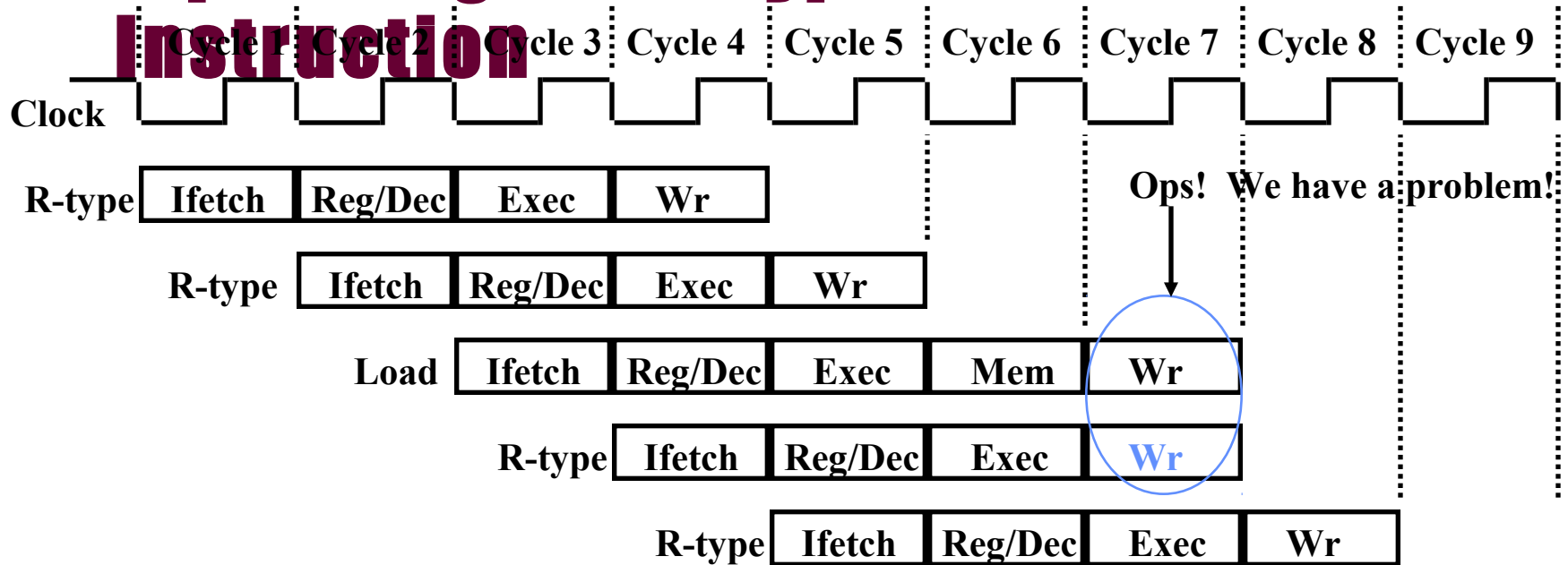
- **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec: Calculate the memory address**
- **Mem: Read the data from the Data Memory**
- **Wr: Write the data back to the register file**

The Four Stages of R-type



- **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- **Reg/Dec: Registers Fetch and Instruction Decode**
- **Exec:**
 - ALU operates on the two register operands
 - Update PC
- **Wr: Write the ALU output back to the register file**

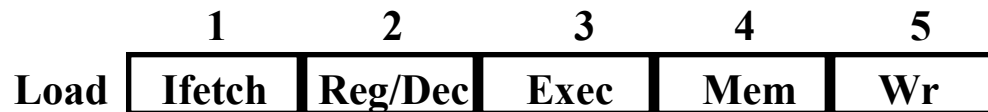
Pipelining the R-type and Load Instruction



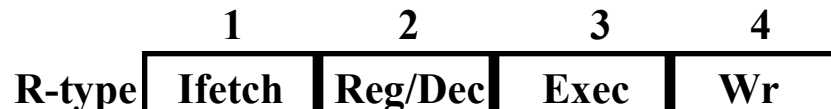
- We have pipeline conflict or structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

Important Observation

- Each functional unit can only be used **once** per instruction
- Each functional unit must be used at the **same** stage for all instructions:
 - Load uses Register File's Write Port during its **5th** stage

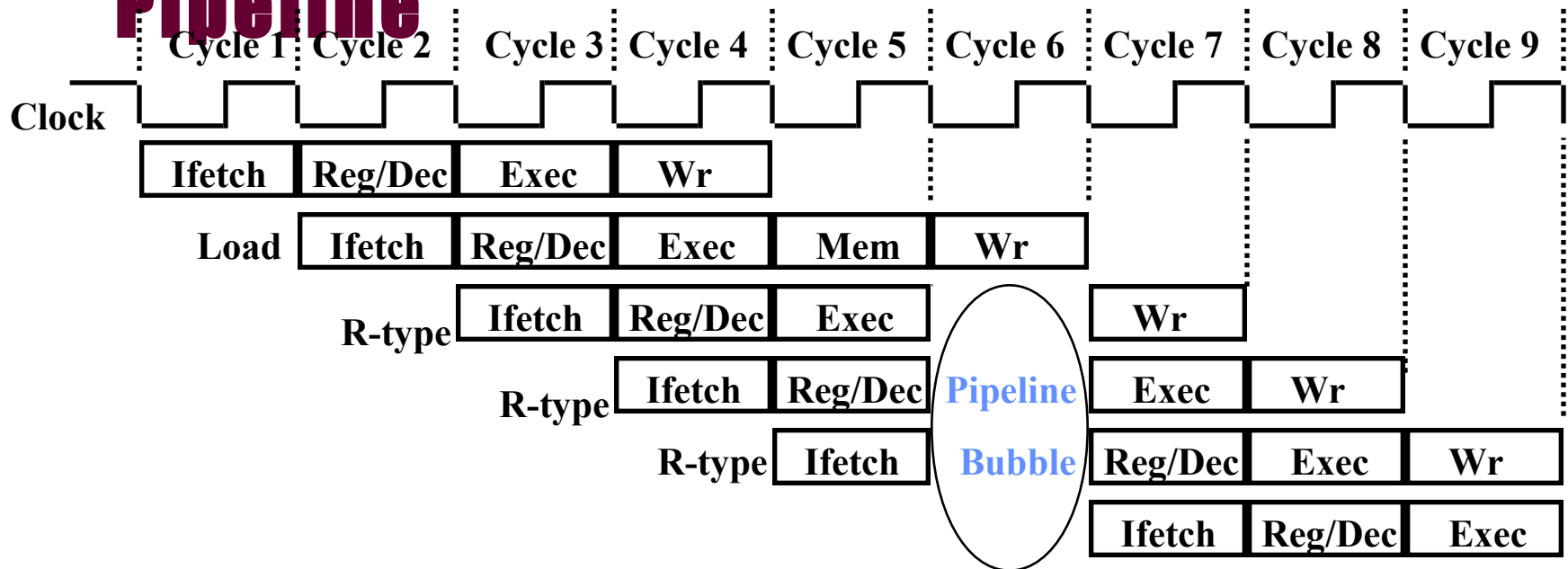


- R-type uses Register File's Write Port during its **4th** stage



- 2 ways to solve this pipeline hazard.

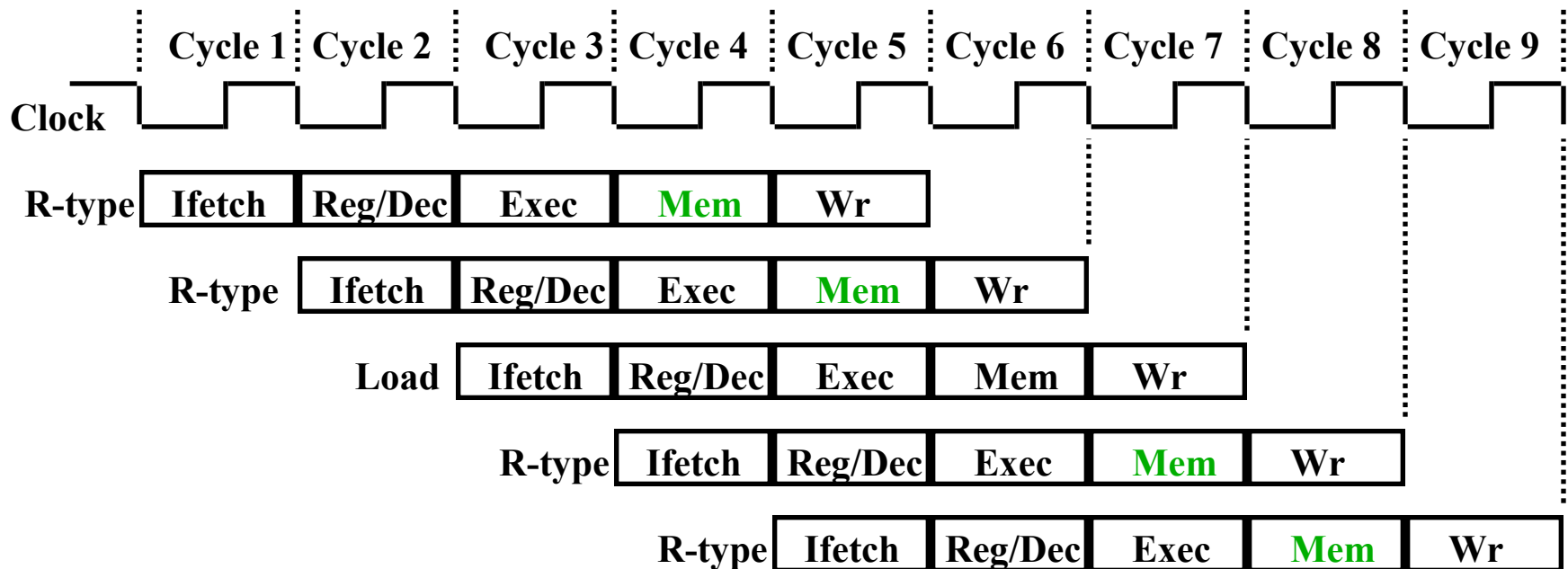
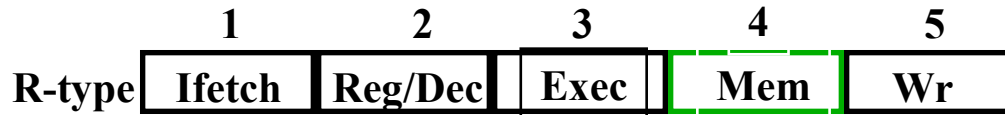
Solution 1: Insert “Bubble” into the Pipeline



- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex.
 - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

Solution 2: Delay R-type's Write by One Cycle

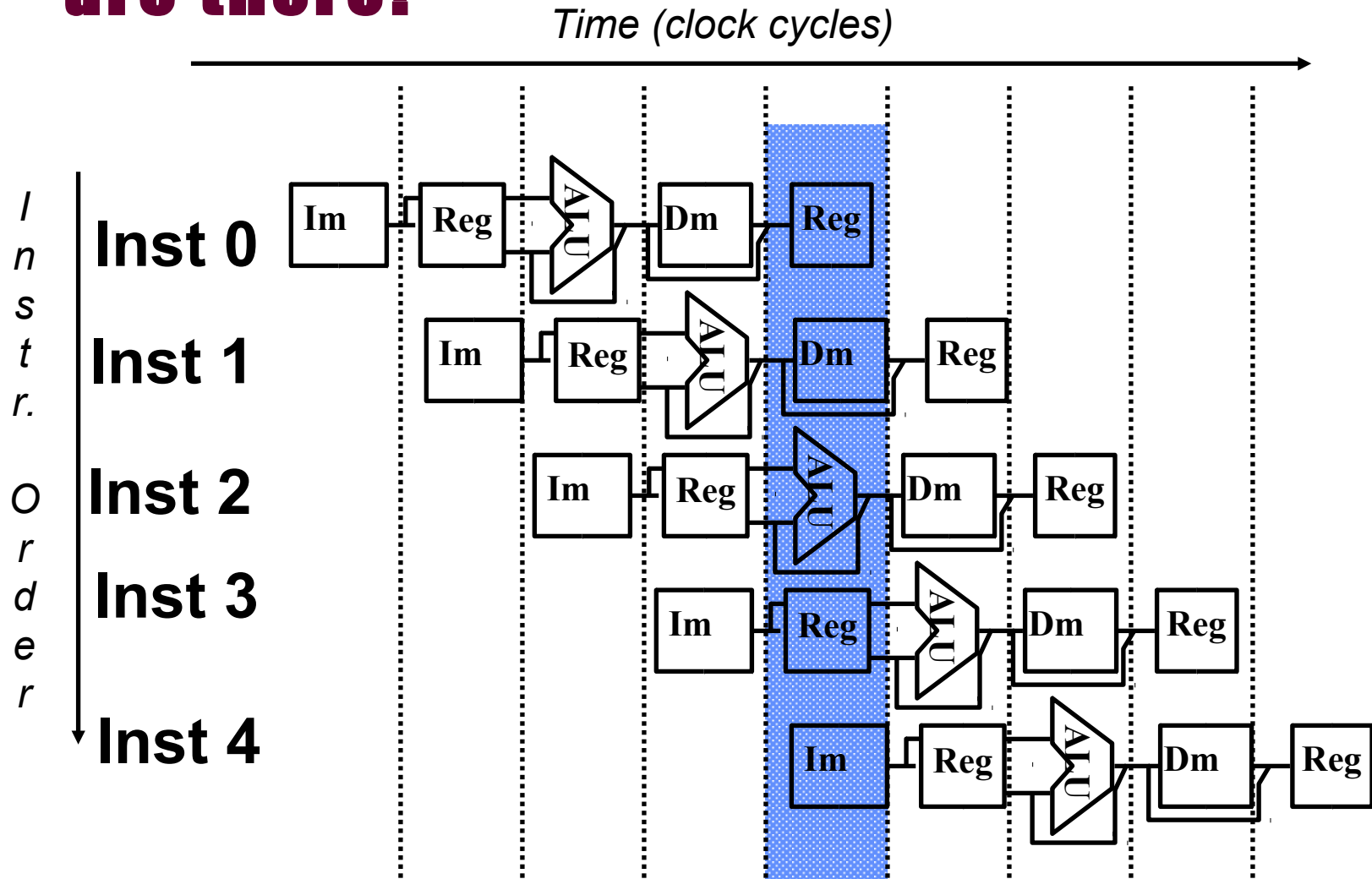
- Delay R-type's register write by one cycle:
 - Now R-type instructions also use Reg File's write port at Stage 5
 - Mem stage is a **NOOP** stage: nothing is being done.



Why Pipeline?

- Suppose we execute 100 instructions
- Single Cycle Machine
 - $45 \text{ ns/cycle} \times 1 \text{ CPI} \times 100 \text{ inst} = 4500 \text{ ns}$
- Multicycle Machine
 - $10 \text{ ns/cycle} \times 4.6 \text{ CPI (due to inst mix)} \times 100 \text{ inst} = 4600 \text{ ns}$
- Ideal pipelined machine
 - $10 \text{ ns/cycle} \times (1 \text{ CPI} \times 100 \text{ inst} + 4 \text{ cycle drain}) = 1040 \text{ ns}$

Why Pipeline? Because the resources are there!



Problems with Pipeline processors?

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle and introduce stall cycles which increase CPI
 - **Structural hazards**: HW cannot support this combination of instructions - two dogs fighting for the same bone
 - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
 - » **Data dependencies**
 - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
 - » **Control dependencies**
- Can always resolve hazards by stalling
- More stall cycles = more CPU time = less performance
 - Increase performance = decrease stall cycles

Back to our old friend: CPU time equation

- Recall equation for CPU time

$$\begin{aligned}\text{CPU time} &= \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Seconds}}{\text{Cycle}} \\ &= IC * CPI * Clk\end{aligned}$$

- So what are we doing by pipelining the instruction execution process ?
 - Clock ?
 - Instruction Count ?
 - CPI ?
 - » How is CPI effected by the various hazards ?

Speed Up Equation for Pipelining

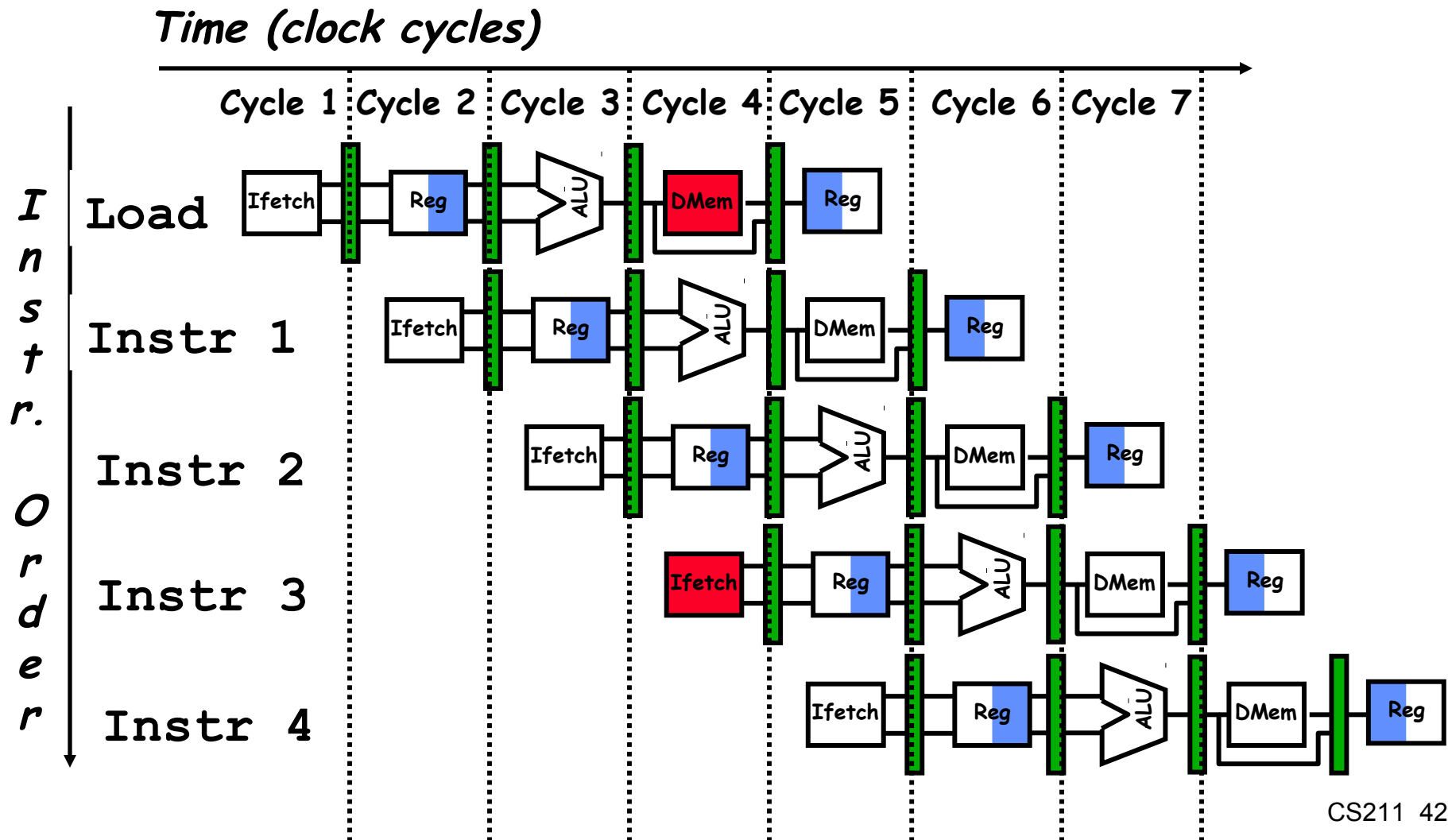
$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

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

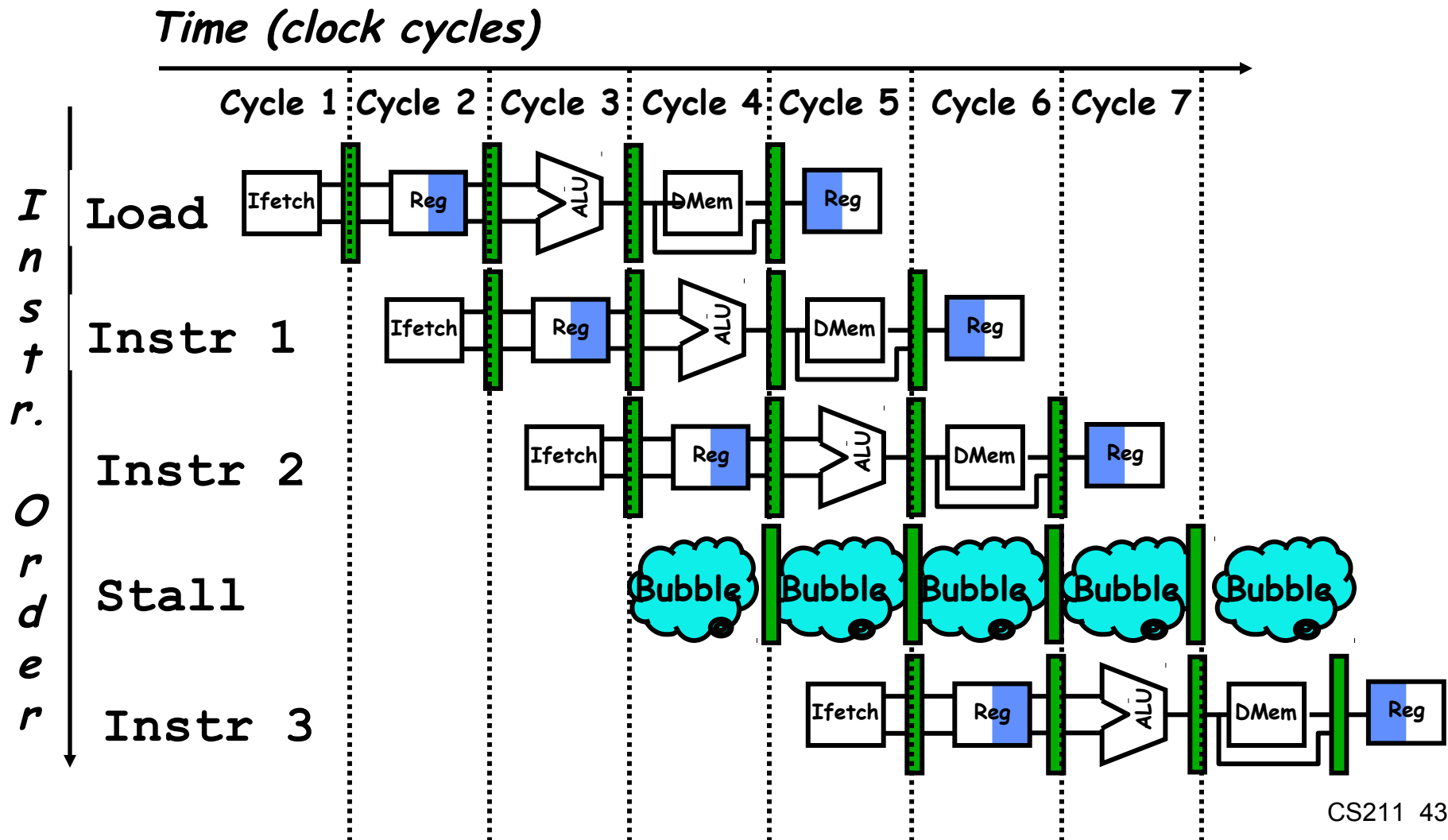
For simple RISC pipeline, $CPI = 1$:

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

One Memory Port/Structural Hazards



One Memory Port/Structural Hazards



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
- Note - Loads will cause stalls of 1 cycle
- Recall our friend:
 - $\text{CPU} = \text{IC} * \text{CPI} * \text{Cik}$
 - » $\text{CPI} = \text{ideal CPI} + \text{stalls}$

Example_

- 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
- Loads are 40% of instructions executed

$$\begin{aligned}\text{SpeedUp}_A &= \text{Pipe. Depth} / (1 + 0) \times (\text{clock}_{\text{unpipe}} / \text{clock}_{\text{pipe}}) \\ &= \text{Pipeline Depth}\end{aligned}$$

$$\begin{aligned}\text{SpeedUp}_B &= \text{Pipe. Depth} / (1 + 0.4 \times 1) \times \\ &\quad (\text{clock}_{\text{unpipe}} / (\text{clock}_{\text{unpipe}} / 1.05)) \\ &= (\text{Pipe. Depth} / 1.4) \times 1.05 \\ &= 0.75 \times \text{Pipe. Depth}\end{aligned}$$

$$\text{SpeedUp}_A / \text{SpeedUp}_B = \text{Pipe. Depth} / (0.75 \times \text{Pipe. Depth}) = 1.33$$

- Machine A is 1.33 times faster


Data Dependencies

- True dependencies and False dependencies
 - false implies we can remove the dependency
 - true implies we are stuck with it!
- Three types of data dependencies defined in terms of how succeeding instruction depends on preceding instruction
 - RAW: Read after Write or Flow dependency
 - WAR: Write after Read or anti-dependency
 - WAW: Write after Write

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 “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.**

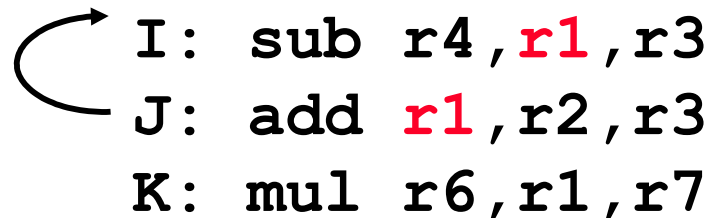
RAW Dependency

- Example program (a) with two instructions
 - i1: load r1, a;
 - i2: add r2, r1,r1;
- Program (b) with two instructions
 - i1: mul r1, r4, r5;
 - i2: add r2, r1, r1;
- Both cases we cannot read in i2 until i1 has completed writing the result
 - In (a) this is due to load-use dependency
 - In (b) this is due to define-use dependency

Three Generic Data Hazards

- **Write After Read (WAR)**

Instr_j writes operand before Instr_i reads it

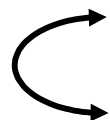


```
I:  sub  r4, r1, r3
J:  add  r1, r2, r3
K:  mul  r6, r1, r7
```

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- 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

Three Generic Data Hazards

- **Write After Write (WAW)**
Instr_j writes operand before Instr_i writes it.
- 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 later more complicated pipes



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

WAR and WAW Dependency

- **Example program (a):**
 - i1: mul r1, r2, r3;
 - i2: add r2, r4, r5;
- **Example program (b):**
 - i1: mul r1, r2, r3;
 - i2: add r1, r4, r5;
- **both cases we have dependence between i1 and i2**
 - in (a) due to r2 must be read before it is written into
 - in (b) due to r1 must be written by i2 after it has been written into by i1

What to do with WAR and WAW ?

- Problem:
 - i1: mul r1, r2, r3;
 - i2: add r2, r4, r5;
- Is this really a dependence/hazard ?

What to do with WAR and WAW

- Solution: Rename Registers
 - i1: mul r1, r2, r3;
 - i2: add r6, r4, r5;
- Register renaming can solve many of these false dependencies
 - note the role that the compiler plays in this
 - specifically, the register allocation process--i.e., the process that assigns registers to variables

Hazard Detection in H/W

- Suppose instruction i is about to be issued and a predecessor instruction j is in the instruction pipeline
- How to detect and store potential hazard information
 - Note that hazards in machine code are based on register usage
 - Keep track of results in registers and their usage
 - » Constructing a register data flow graph
- For each instruction i construct set of Read registers and Write registers
 - $Rregs(i)$ is set of registers that instruction i reads from
 - $Wregs(i)$ is set of registers that instruction i writes to
 - Use these to define the 3 types of data hazards

Hazard Detection in Hardware

- A RAW hazard exists on register ρ if $\rho \in \text{Rregs}(i) \cap \text{Wregs}(j)$
 - 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 an operation completes, remove its write reservation.



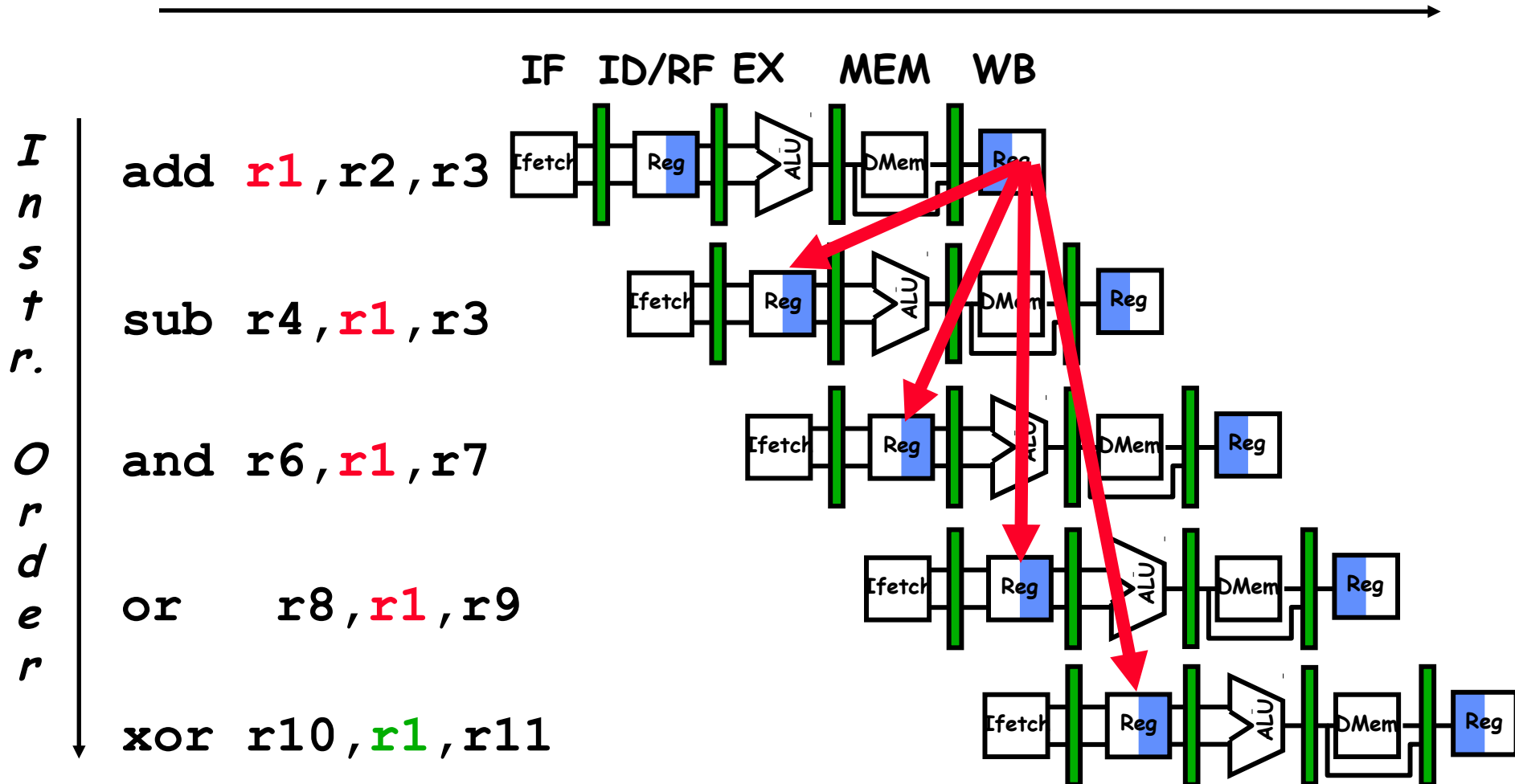
- A WAW hazard exists on register ρ if $\rho \in \text{Wregs}(i) \cap \text{Wregs}(j)$
- A WAR hazard exists on register ρ if $\rho \in \text{Wregs}(i) \cap \text{Rregs}(j)$

Internal Forwarding: Getting rid of some hazards

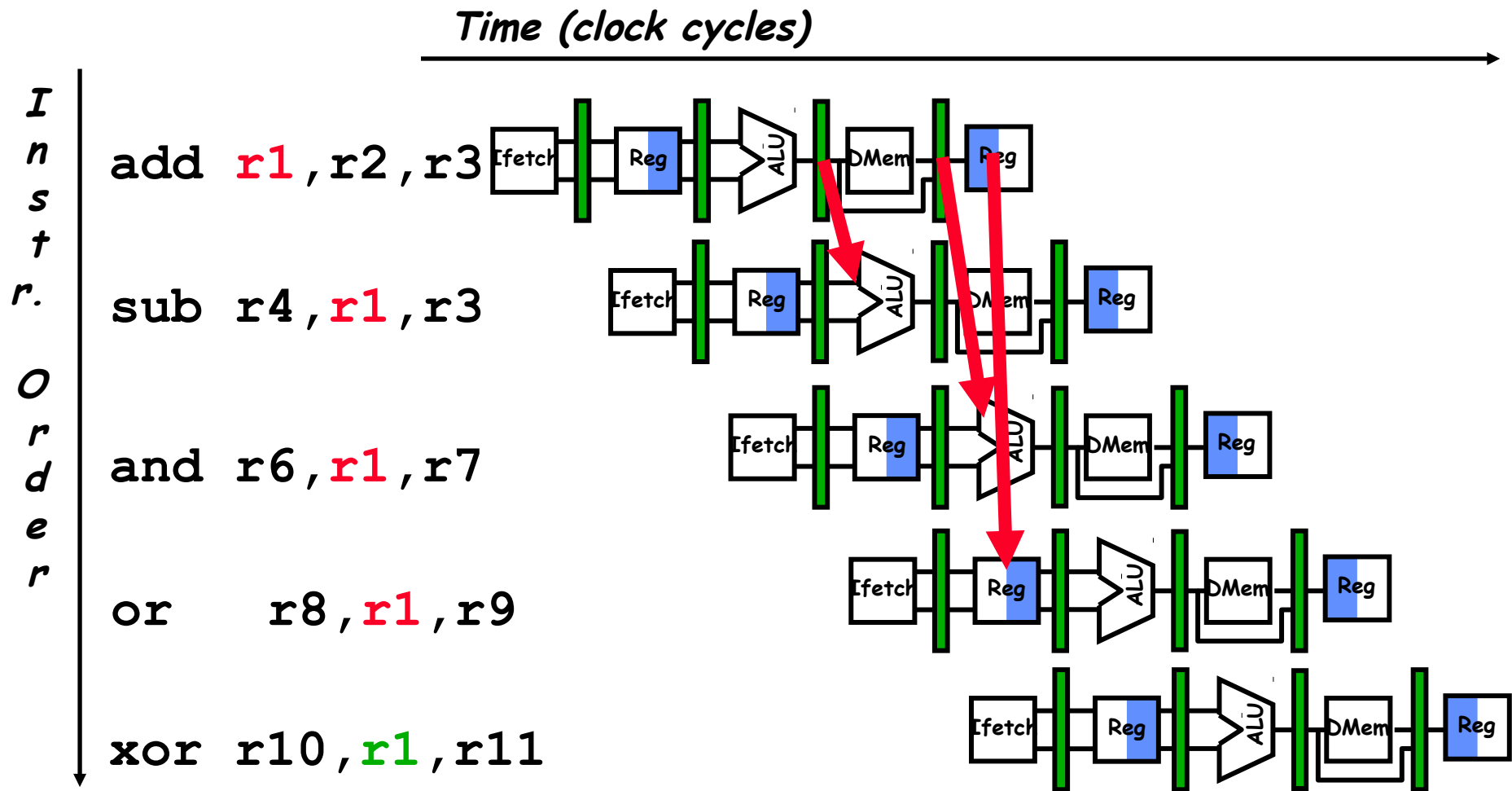
- **In some cases the data needed by the next instruction at the ALU stage has been computed by the ALU (or some stage defining it) but has not been written back to the registers**
- **Can we “forward” this result by bypassing stages ?**

Data Hazard on R1

Time (clock cycles)



Forwarding to Avoid Data Hazard



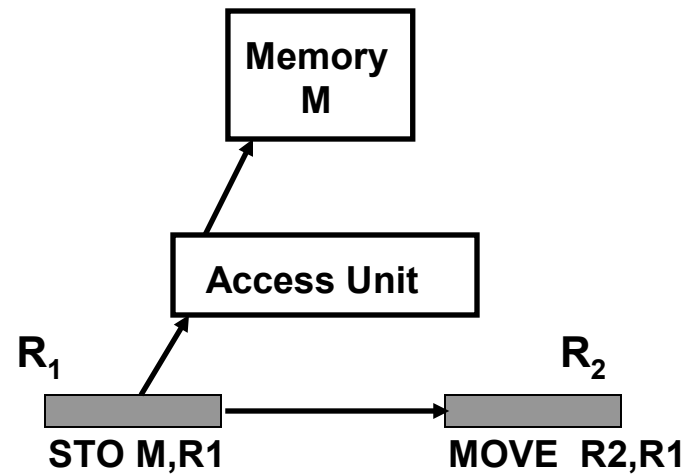
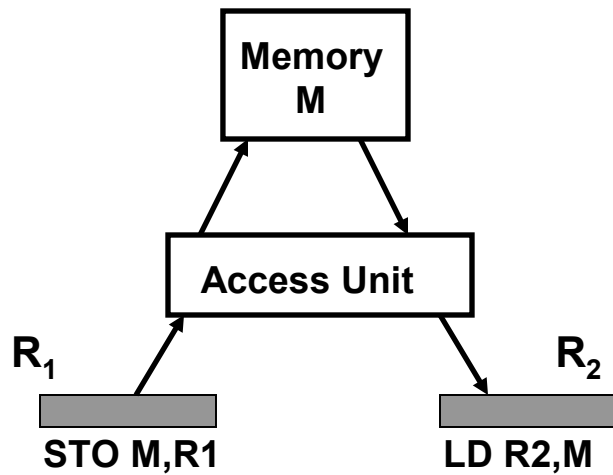
Internal Forwarding of Instructions

- Forward result from ALU/Execute unit to execute unit in next stage
- Also can be used in cases of memory access
- in some cases, operand fetched from memory has been computed previously by the program
 - can we “forward” this result to a later stage thus avoiding an extra read from memory ?
 - Who does this ?
- Internal forwarding cases
 - Stage i to Stage $i+k$ in pipeline
 - store-load forwarding
 - load-store forwarding
 - store-store forwarding

Internal Data Forwarding

38

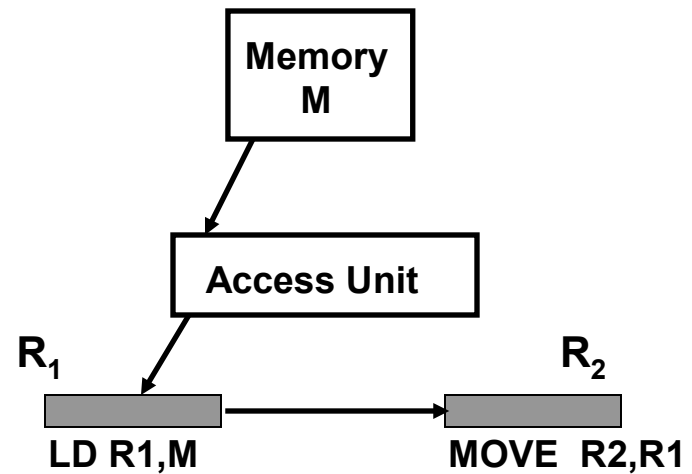
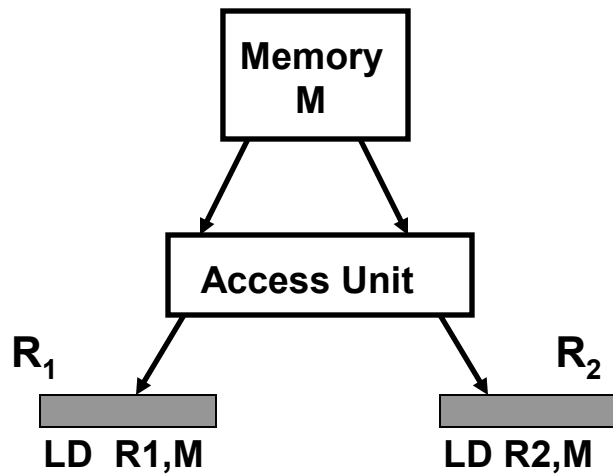
Store-load forwarding



Internal Data Forwarding

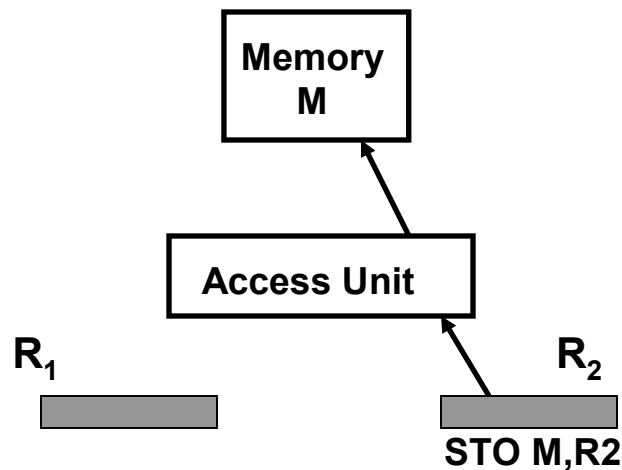
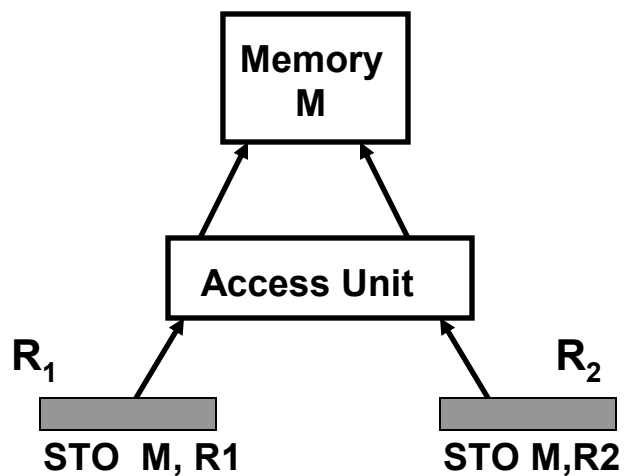
39

Load-load forwarding

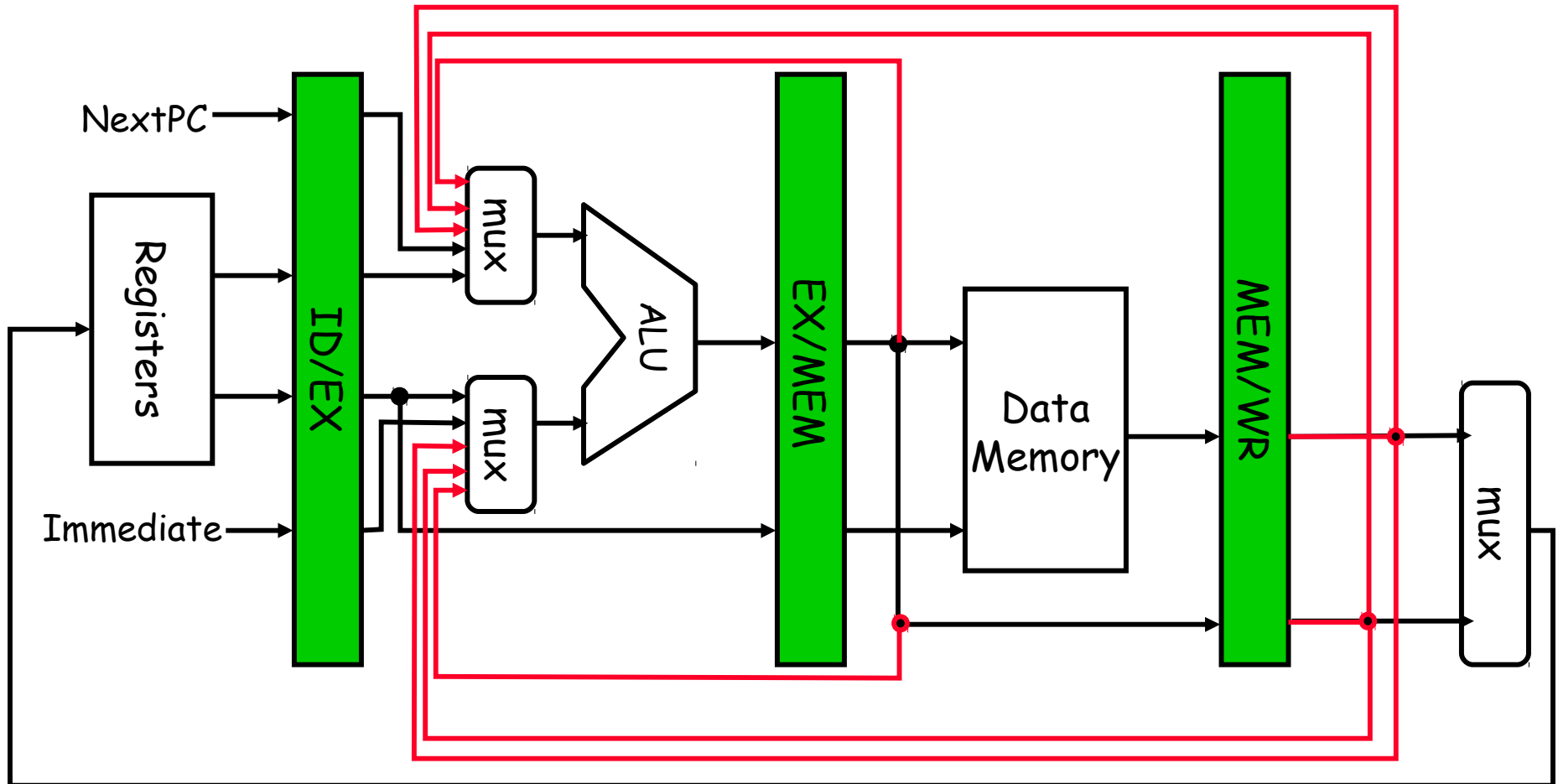


Internal Data Forwarding

Store-store forwarding

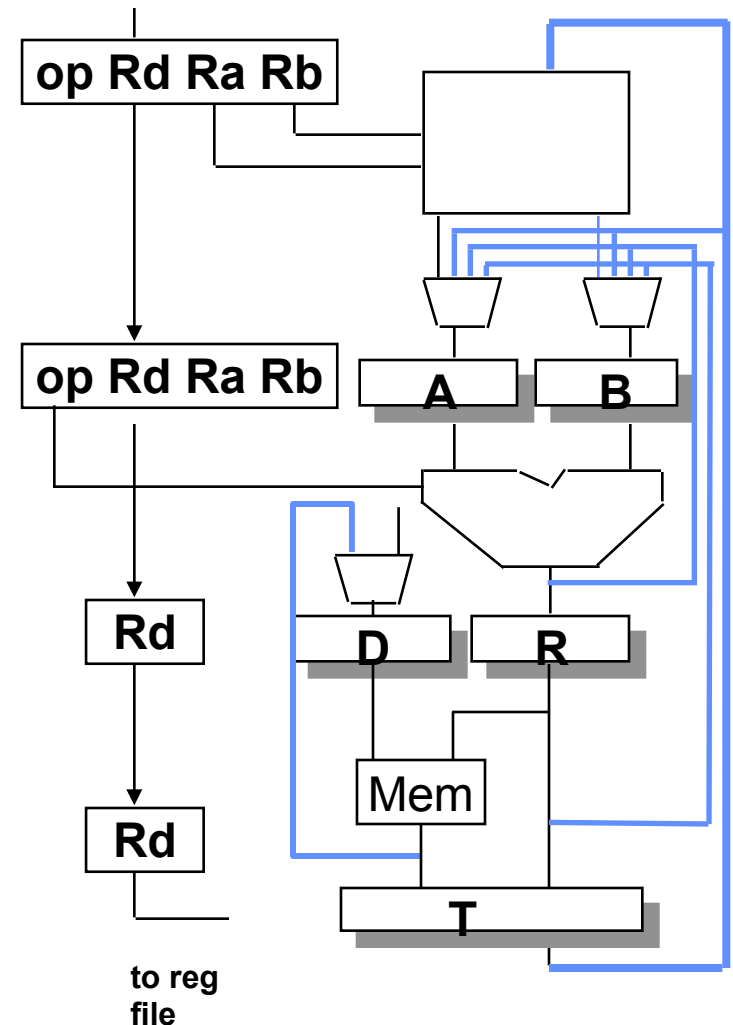


HW Change for Forwarding



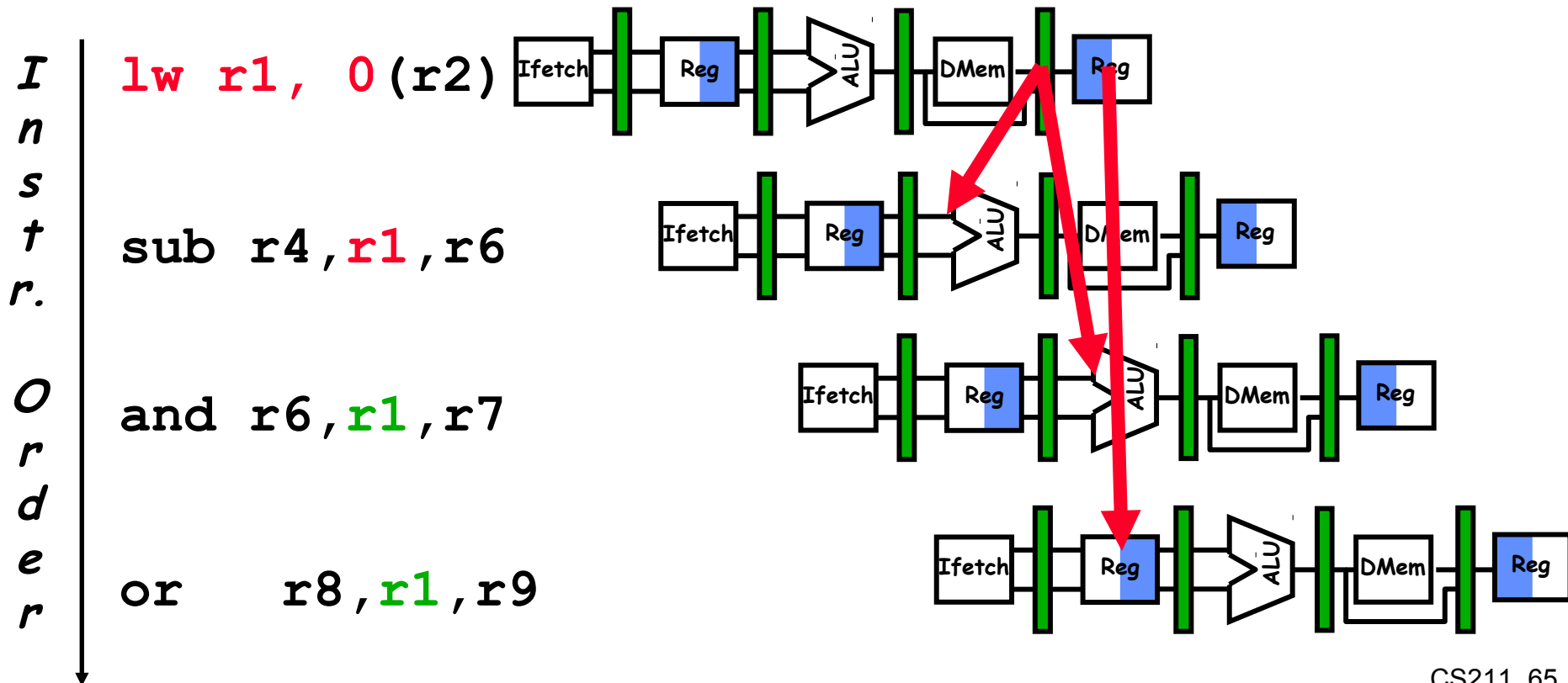
What about memory operations?

- If instructions are initiated in order and operations always occur in the same stage, there can be no hazards between memory operations!
- What does delaying WB on arithmetic operations cost?
 - cycles ?
 - hardware ?
- What about data dependence on loads?
 - $R1 \leftarrow R4 + R5$
 - $R2 \leftarrow \text{Mem}[R2 + I]$
 - $R3 \leftarrow R2 + R1$ \Rightarrow “**Delayed Loads**”
- Can recognize this in decode stage and introduce bubble while stalling fetch stage
- Tricky situation:
 - $R1 \leftarrow \text{Mem}[R2 + I]$
 - $\text{Mem}[R3+34] \leftarrow R1$Handle with bypass in memory stage!

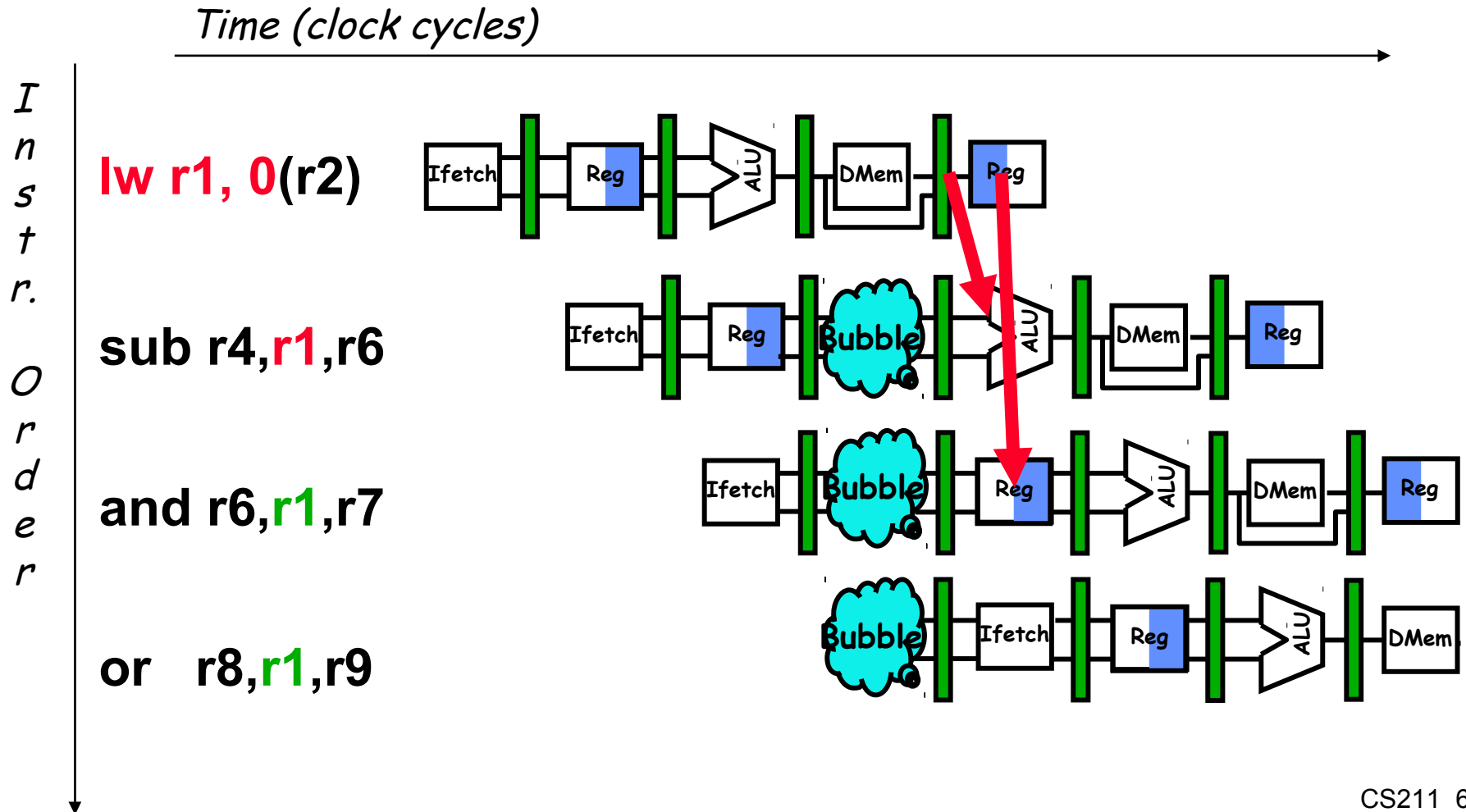


Data Hazard Even with Forwarding

Time (clock cycles)



Data Hazard Even with Forwarding



What can we (S/W) do?

Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

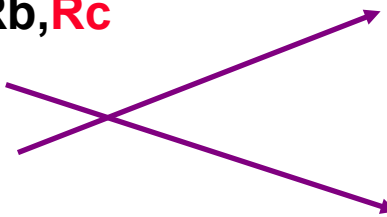
assuming $a, b, c, d, e,$ and f in memory.

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

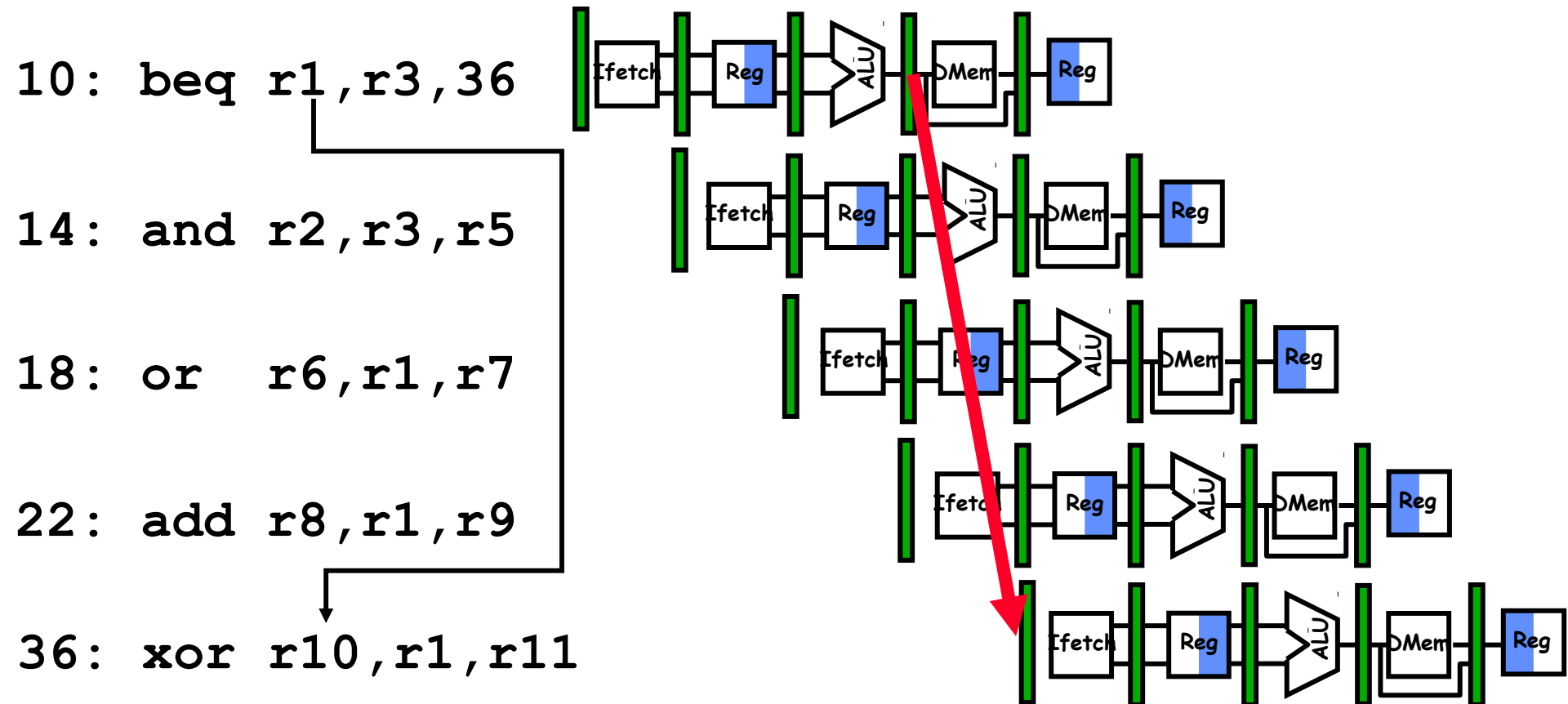


Control Hazards: Branches

- **Instruction flow**
 - Stream of instructions processed by Inst. Fetch unit
 - Speed of “input flow” puts bound on rate of outputs generated
- **Branch instruction affects instruction flow**
 - Do not know next instruction to be executed until branch outcome known
- **When we hit a branch instruction**
 - Need to compute target address (where to branch)
 - Resolution of branch condition (true or false)
 - Might need to ‘flush’ pipeline if other instructions have been fetched for execution

Control Hazard on Branches

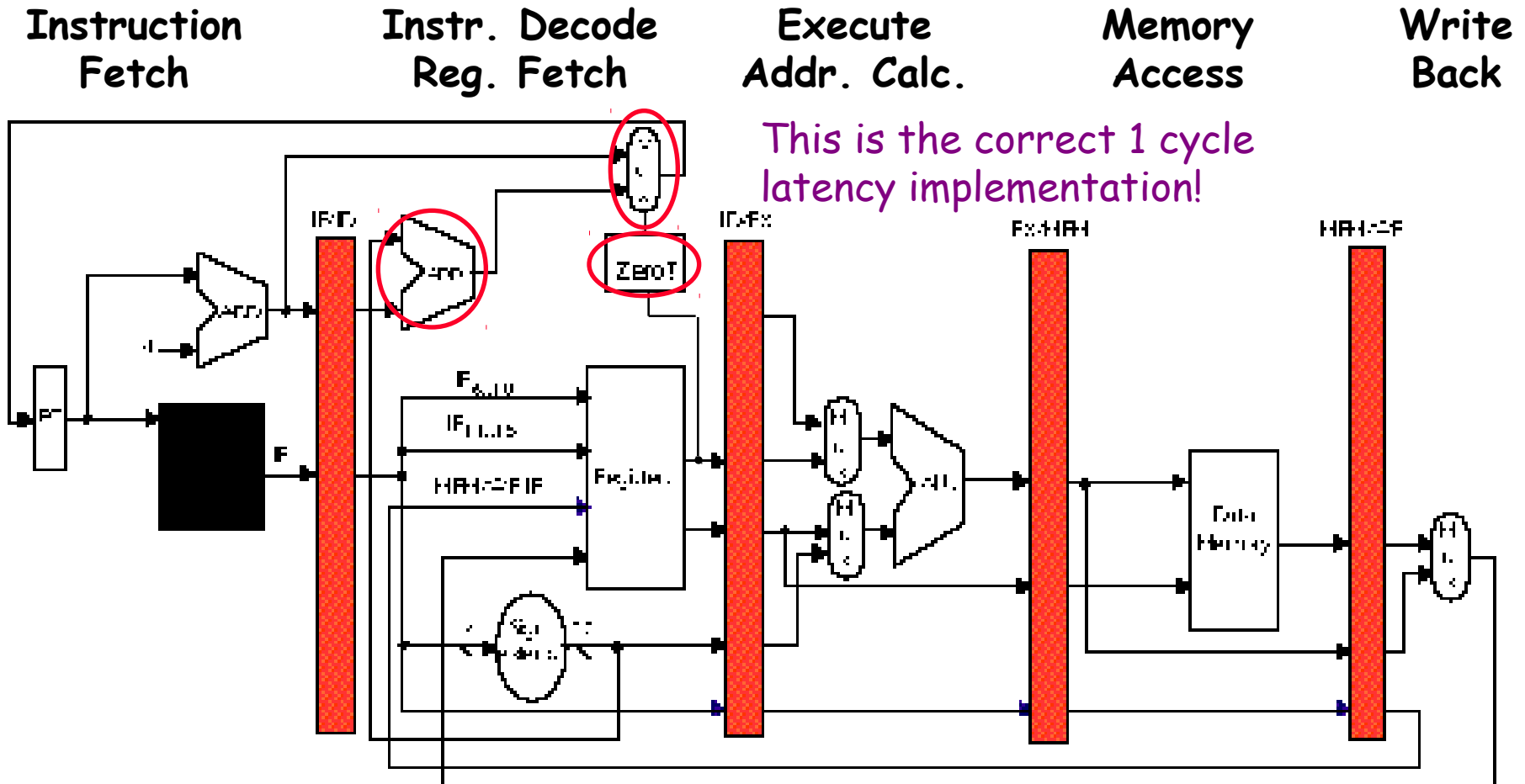
Three Stage Stall



Branch Stall Impact

- If $CPI = 1$, 30% branch,
Stall 3 cycles \Rightarrow 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 $\neq 0$
- 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 (DLX) Datapath



Four Branch Hazard Alternatives

#1: Stall until branch direction is clear – flushing pipe

#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% DLX branches not taken on average
- PC+4 already calculated, so use it to get next instruction

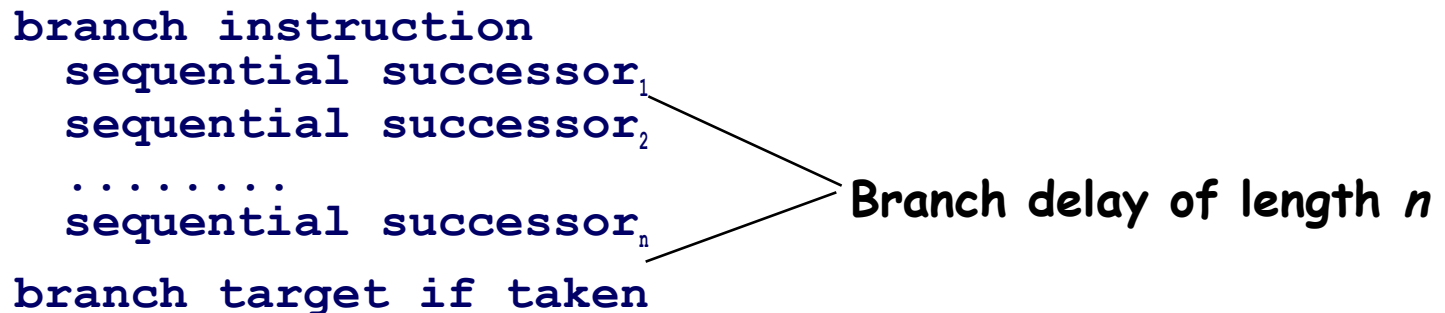
#3: Predict Branch Taken

- 53% DLX branches taken on average
- But haven't calculated branch target address in DLX
 - » DLX still incurs 1 cycle branch penalty
 - » Other machines: branch target known before outcome

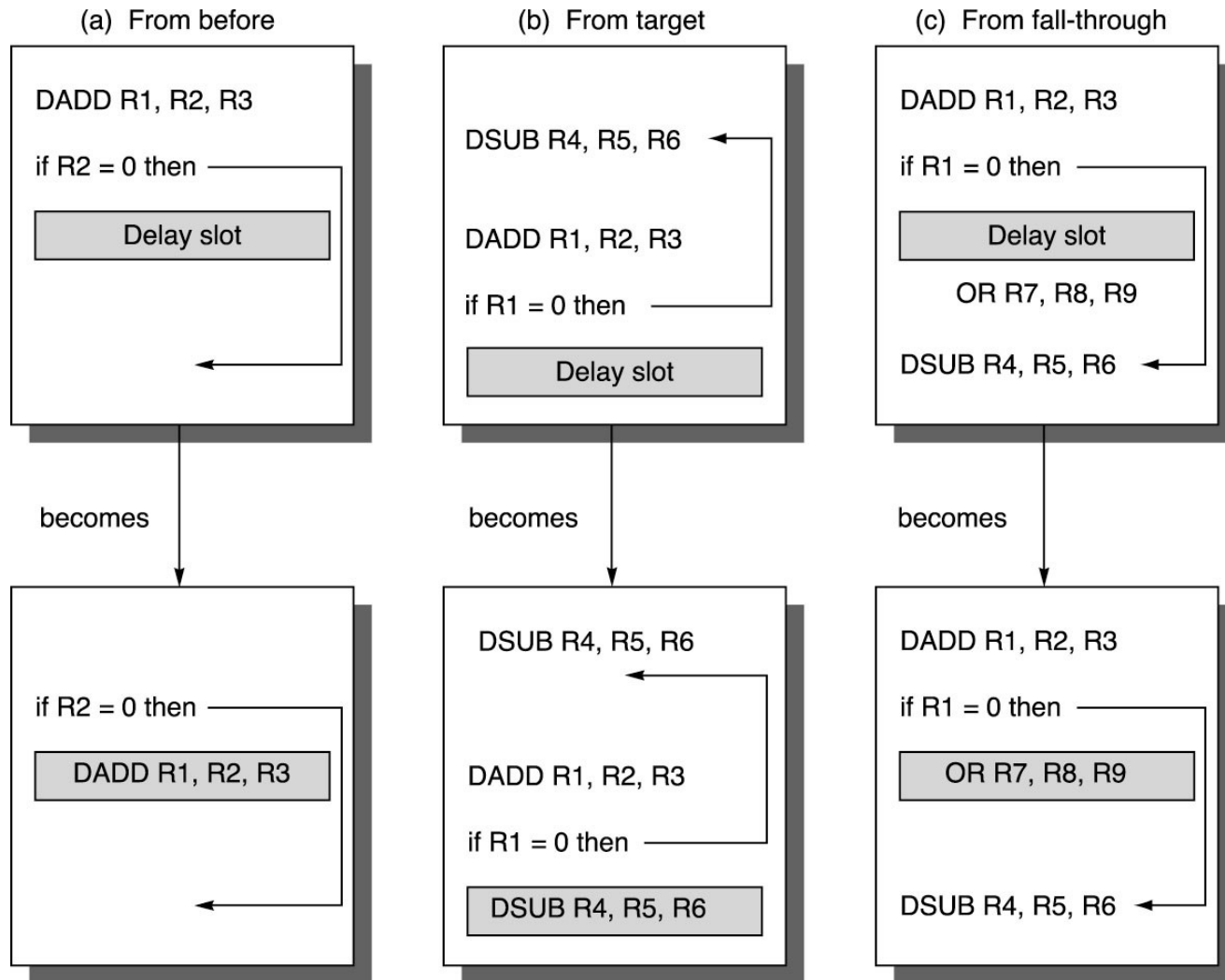
Four Branch Hazard Alternatives

#4: Delayed Branch

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



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



Delayed Branch

- **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
 - Cancelling branches allow more slots to be filled
- **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: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)**

Evaluating Branch Alternatives

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

<i>Scheduling scheme</i>	<i>Branch penalty</i>	<i>CPI</i>	<i>speedup v. unpipelined</i>	<i>speedup v. stall</i>
Stall pipeline	3	1.42	3.5	1.0
Predict taken	1	1.14	4.4	1.26
Predict not taken	1	1.09	4.5	1.29
Delayed branch	0.5	1.07	4.6	1.31

Conditional & Unconditional = 14%, 65% change PC

Branch Prediction based on history

- Can we use history of branch behaviour to predict branch outcome ?
- Simplest scheme: use 1 bit of “history”
 - Set bit to Predict Taken (T) or Predict Not-taken (NT)
 - Pipeline checks bit value and predicts
 - » If incorrect then need to invalidate instruction
 - Actual outcome used to set the bit value
- Example: let initial value = T, actual outcome of branches is- NT, T,T,NT,T,T
 - Predictions are: T, NT,T,T,NT,T
 - » 3 wrong (in red), 3 correct = 50% accuracy
- In general, can have k-bit predictors: more when we cover superscalar processors.

Summary :

Control and Pipelining

- Just overlap tasks; easy if tasks are independent
- Speed Up \leq 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

Summary #1/2:

Pipelining

- What makes it easy
 - all instructions are the same length
 - just a few instruction formats
 - memory operands appear only in loads and stores
- What makes it hard? HAZARDS!
 - structural hazards: suppose we had only one memory
 - control hazards: need to worry about branch instructions
 - data hazards: an instruction depends on a previous instruction
- Pipelines pass control information down the pipe just as data moves down pipe
- Forwarding/Stalls handled by local control
- Exceptions stop the pipeline

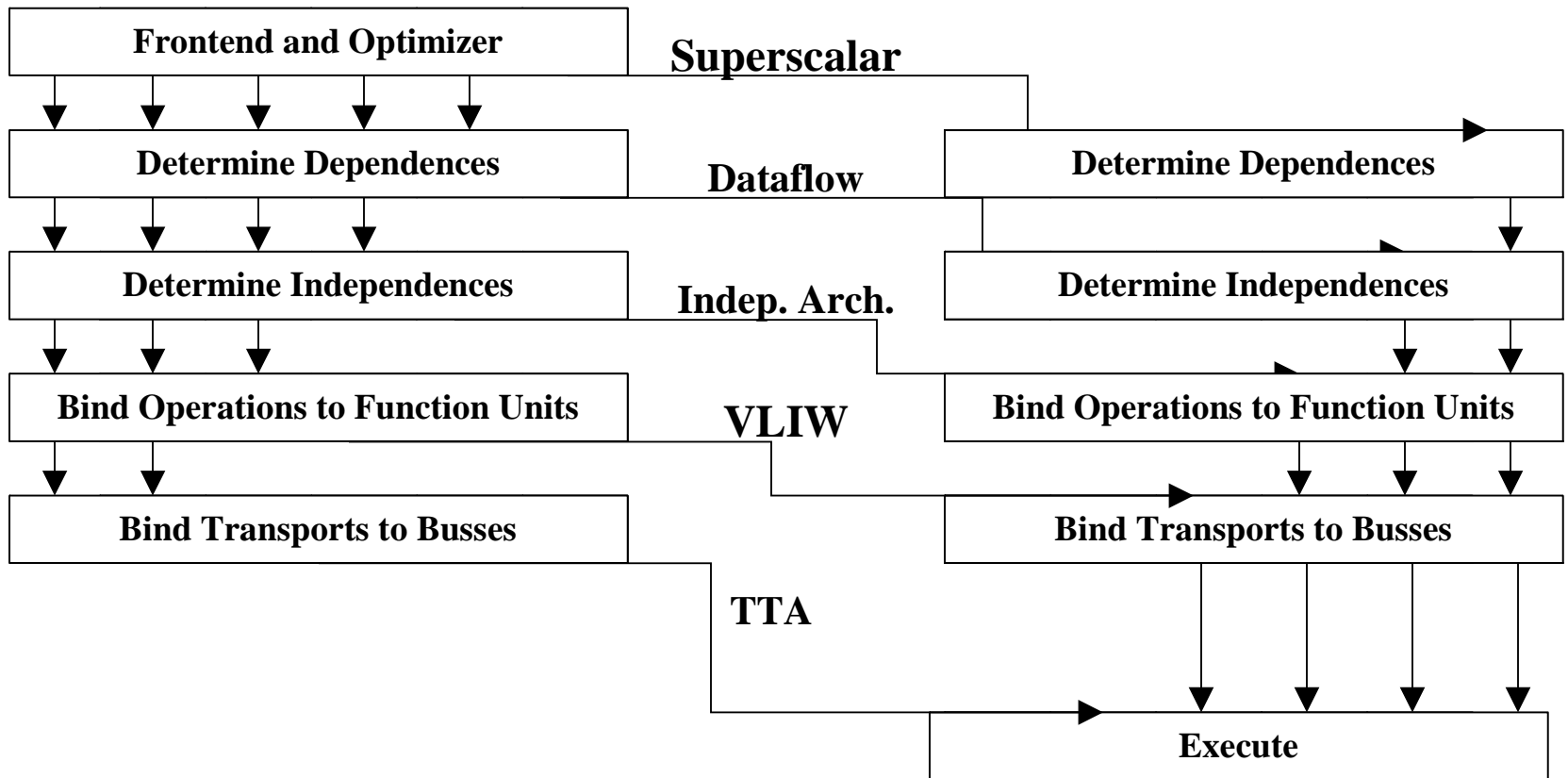
Introduction to ILP

- **What is ILP?**
 - Processor and Compiler design techniques that speed up execution by causing individual machine operations to execute in parallel
- **ILP is transparent to the user**
 - Multiple operations executed in parallel even though the system is handed a single program written with a sequential processor in mind
- **Same execution hardware as a normal RISC machine**
 - May be more than one of any given type of hardware

Compiler vs. Processor

Compiler

Hardware



B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel: History overview, and perspective. The Journal of Supercomputing, 7(1-2):9-50, May 1993.