Computer Architecture, Fall 2019

Pipeline and Hazards

Great Idea #4: Parallelism

Software

- Parallel Requests
  - Assigned to computer
  - e.g. search “Garcia”
- Parallel Threads
  - Assigned to core
  - e.g. lookup, ads
- Parallel Instructions
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions
- Parallel Data
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words
- Hardware descriptions
  - All gates functioning in parallel at same time

Leverage Parallelism & Achieve High Performance

Hardware

Warehouse Scale Computer

Smart Phone

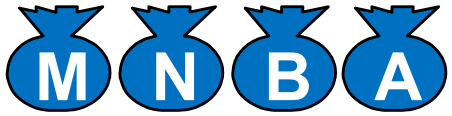
Review of Last Lecture

- Implementing controller for your datapath
  - Take decoded signals from instruction and generate control signals

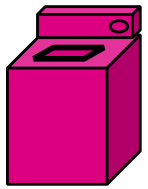
Agenda

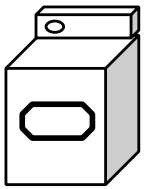
- Pipelined Execution
- Pipelined Datapath

Pipeline Analogy: Doing Laundry

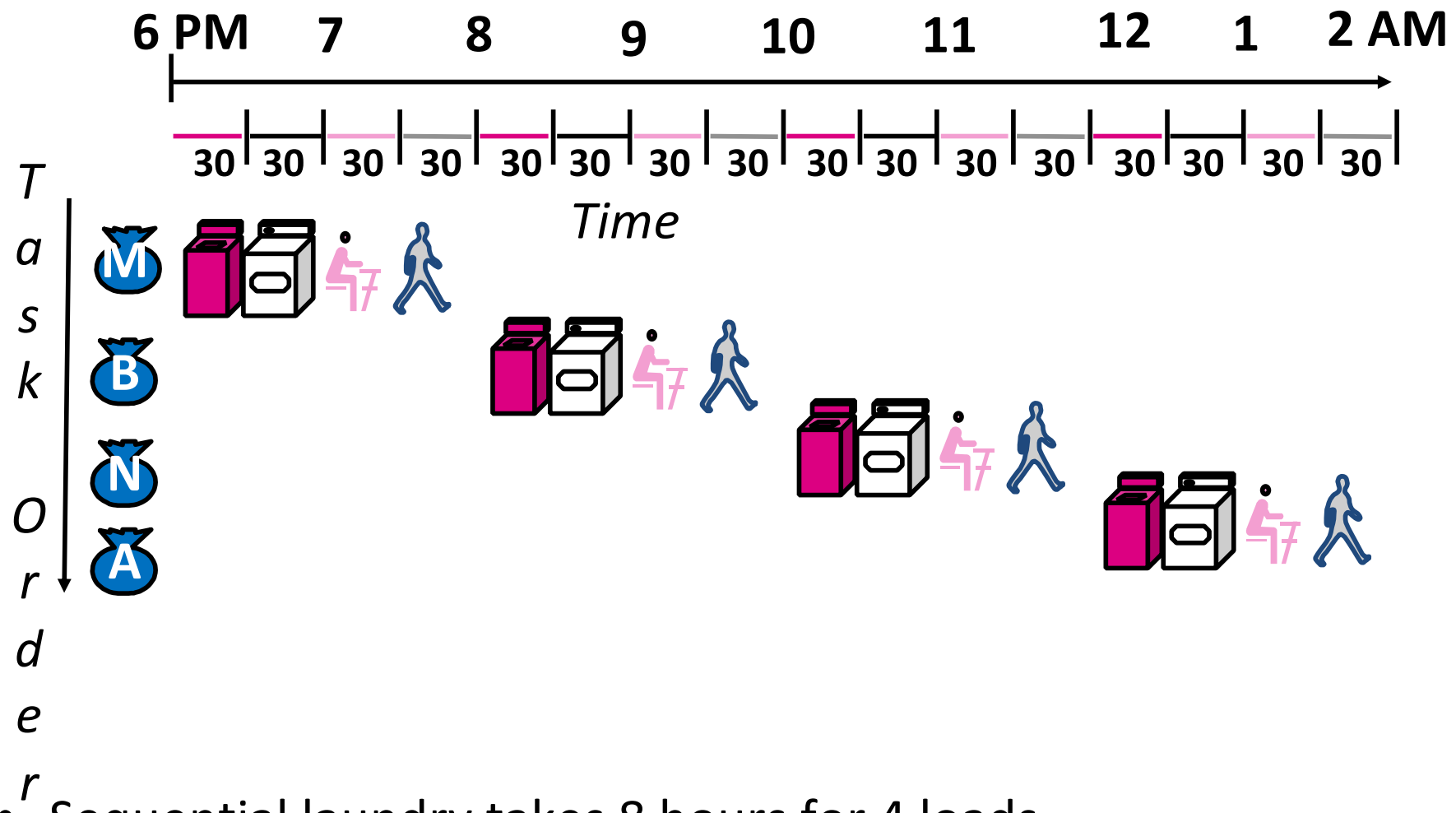


- Morgan, Nick, Branden, and Ayush each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers



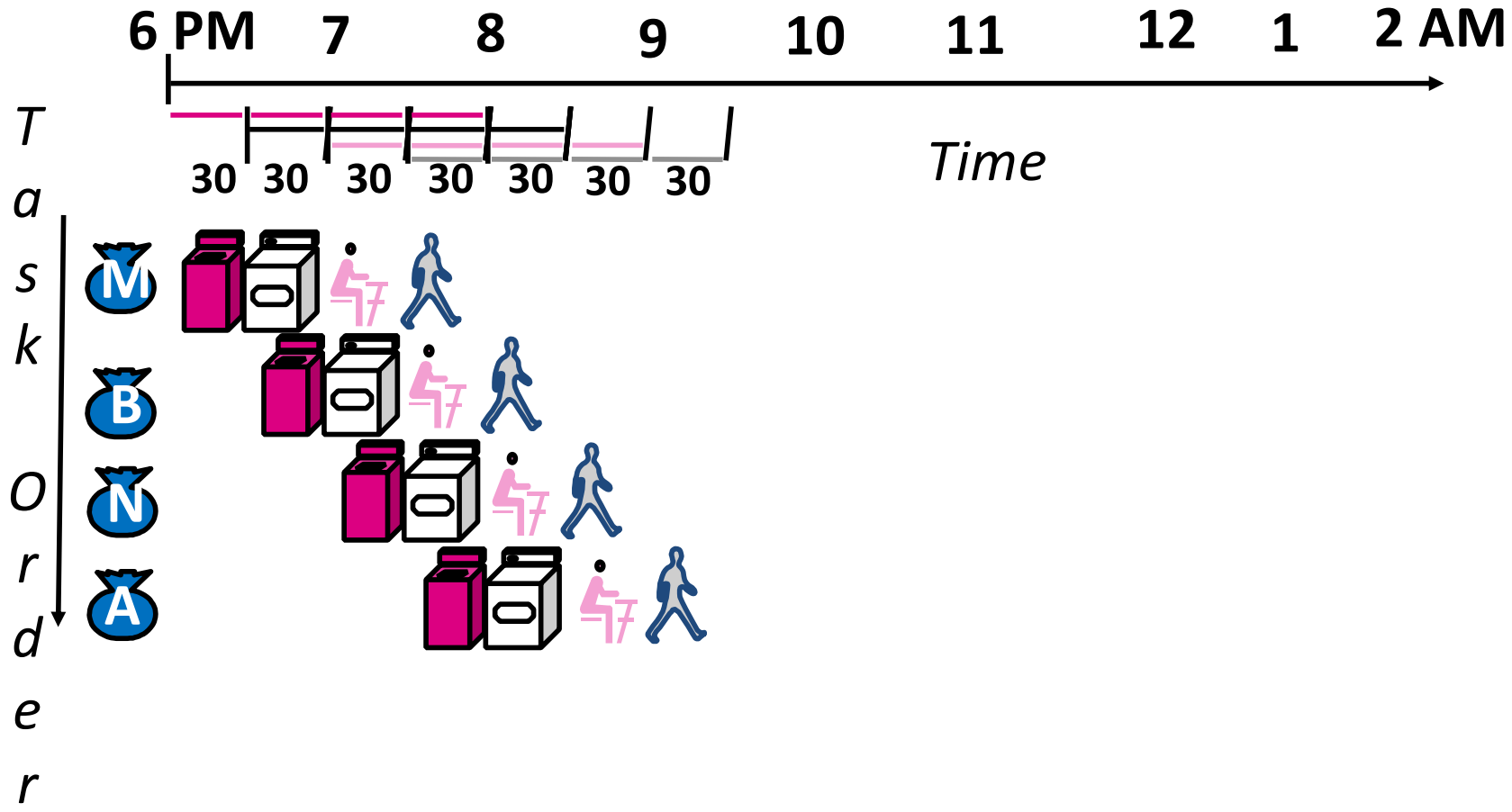


Sequential Laundry



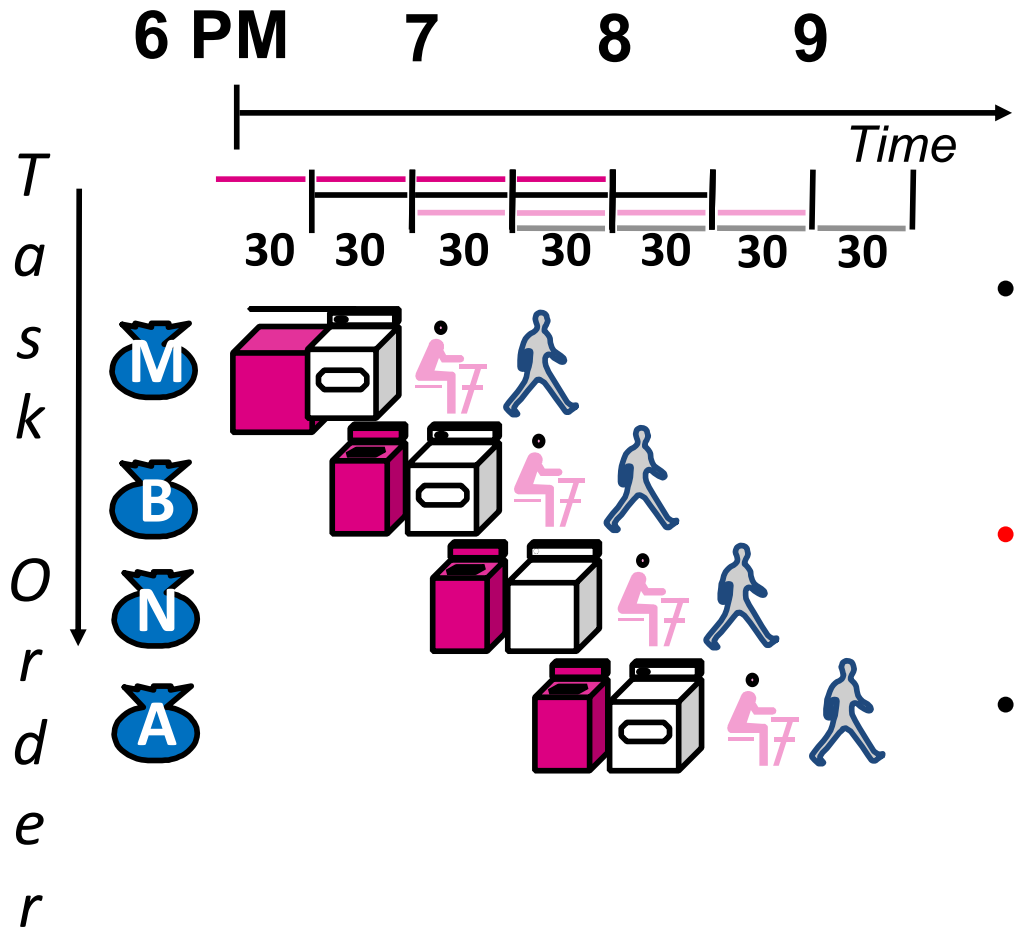
- Sequential laundry takes 8 hours for 4 loads
- 1 load finishes every 2 hours, and Ayush is up til 2AM...

Pipelined Laundry



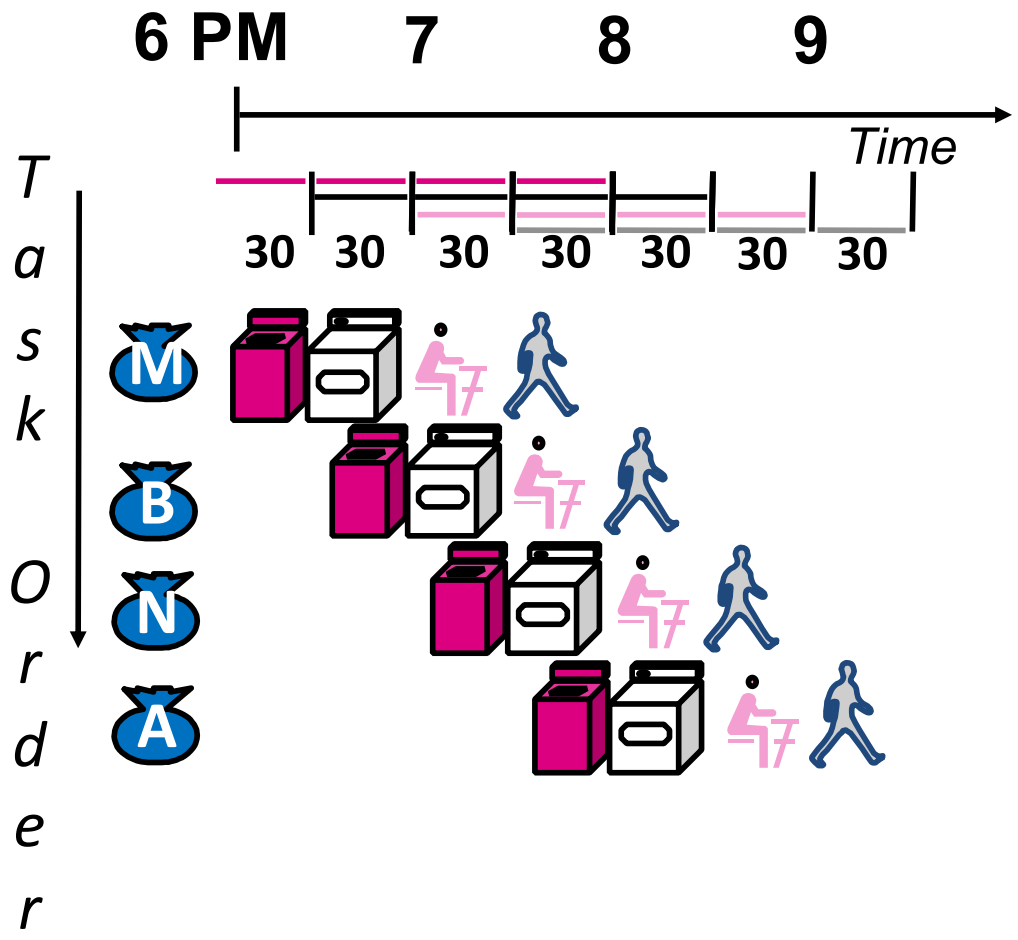
- Pipelined laundry takes 3.5 hours for 4 loads!
- 1 load finishes every half hour (after the first load, which takes 2 hours)

Pipelining Lessons (1/2)



- Pipelining doesn't decrease *latency* of single task; it increases *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup ~ number of pipeline stages
- Speedup reduced by time to *fill* and *drain* the pipeline:
  8 hours/3.5 hours which gives 2.3X speedup v. potential 4X in this example

Pipelining Lessons (2/2)



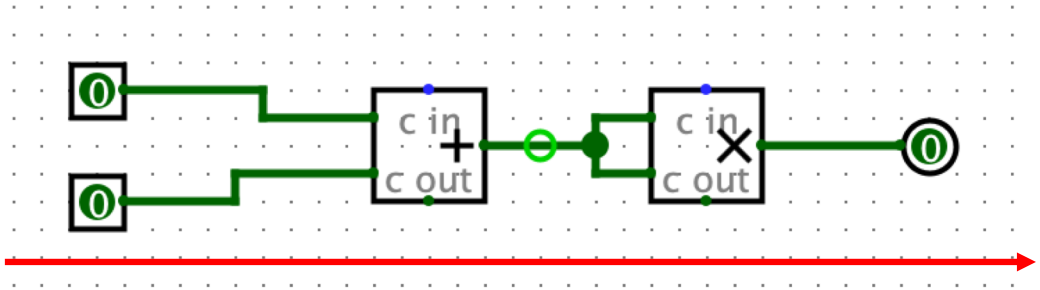
- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
  - Pipeline rate limited by *slowest* pipeline stage
  - Unbalanced lengths of pipeline stages reduces speedup

Agenda

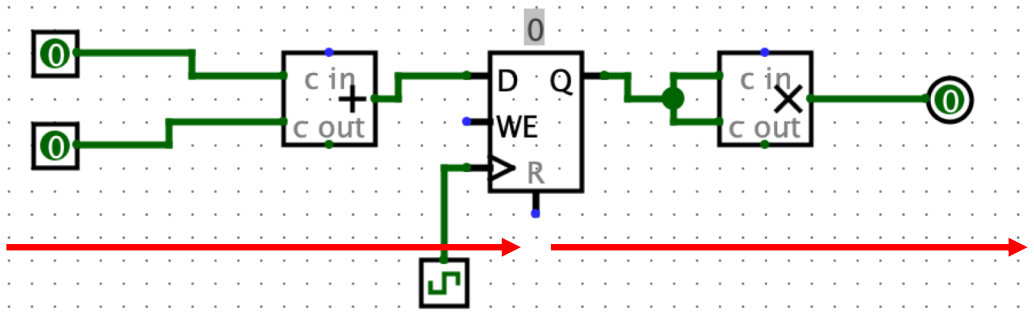
- Pipelined Execution
- Pipelined Datapath

Quick review: Circuit pipelining!

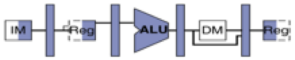
- When we calculate cycle time, or critical path, we do so between state elements, inputs, and

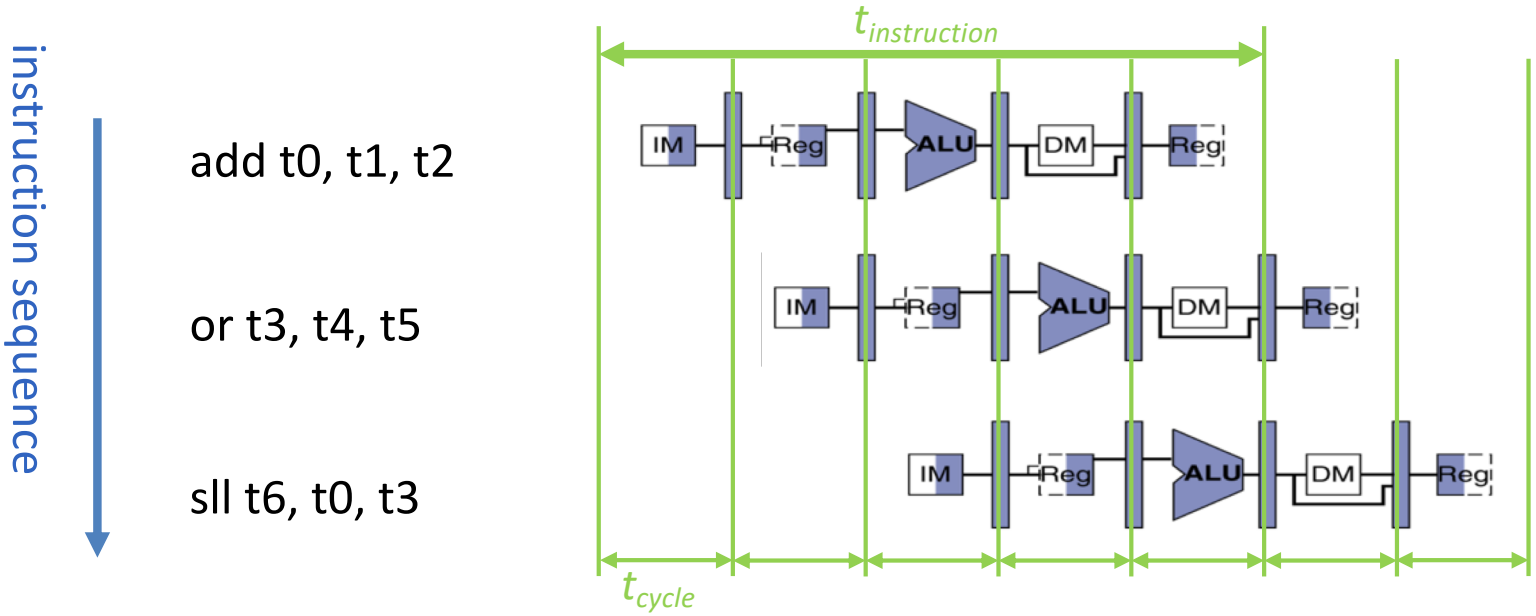


- Adding registers between circuit components *decreases* our critical path and *increases* our frequency



Pipelining with RISC-V

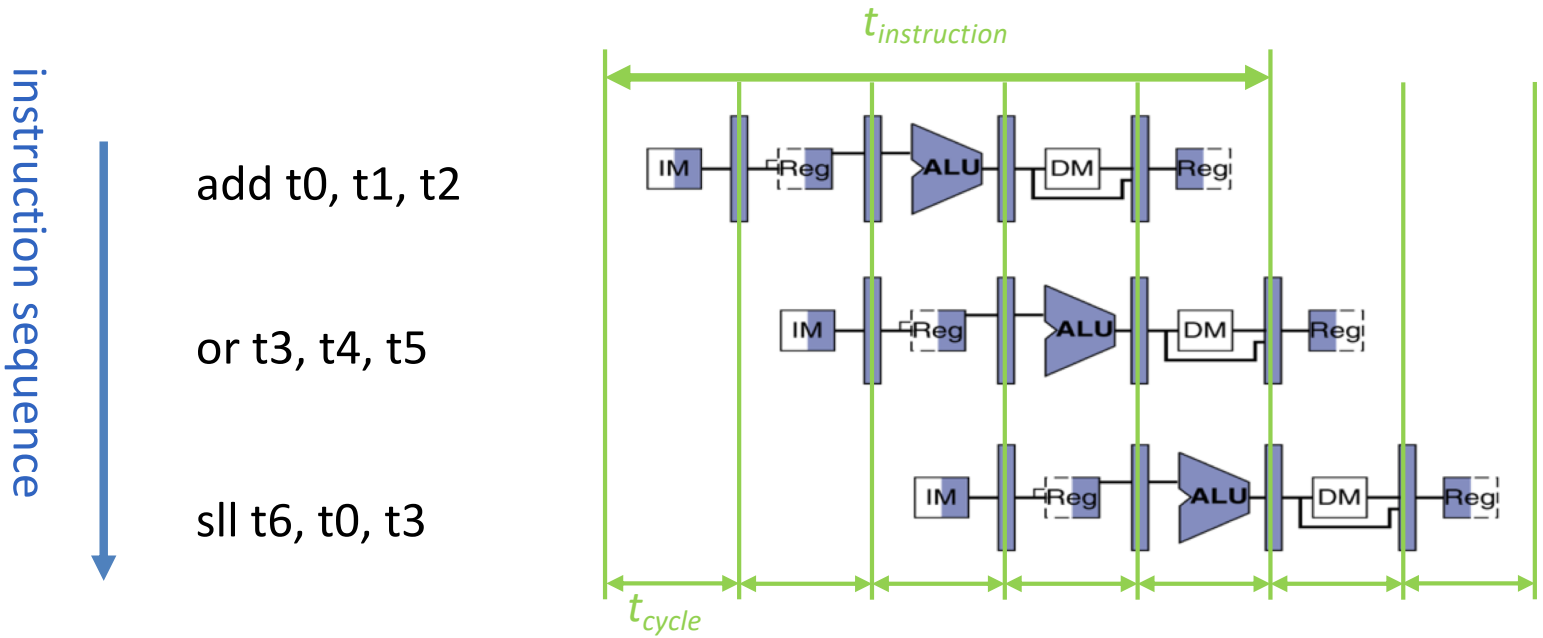
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch		200 ps	200 ps
Reg Read		100 ps	200 ps
ALU		200 ps	200 ps
Memory		200 ps	200 ps
Register Write		100 ps	200 ps
$t_{instruction}$		**800 ps**	**1000 ps**



Pipeline Performance

- Use T_c ("time between completion of instructions") to measure speedup
- Speedup due to increased *throughput*
  - *Latency* for each instruction does not decrease, in fact it may increase if our stages are uneven!
- It takes longer for the *first* instruction to finish, but every instruction after finishes *faster*!

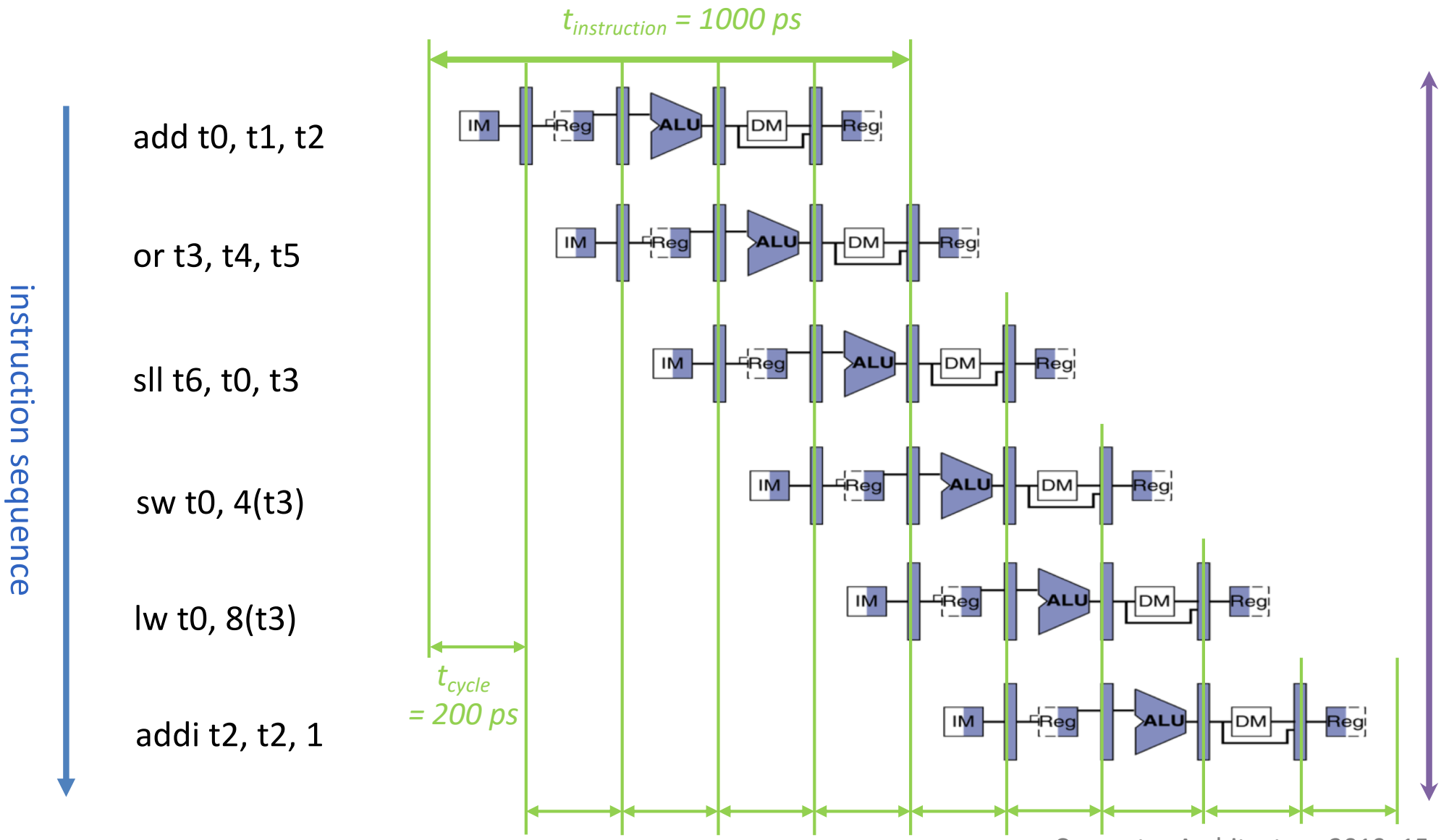
Pipelining with RISC-V



	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?

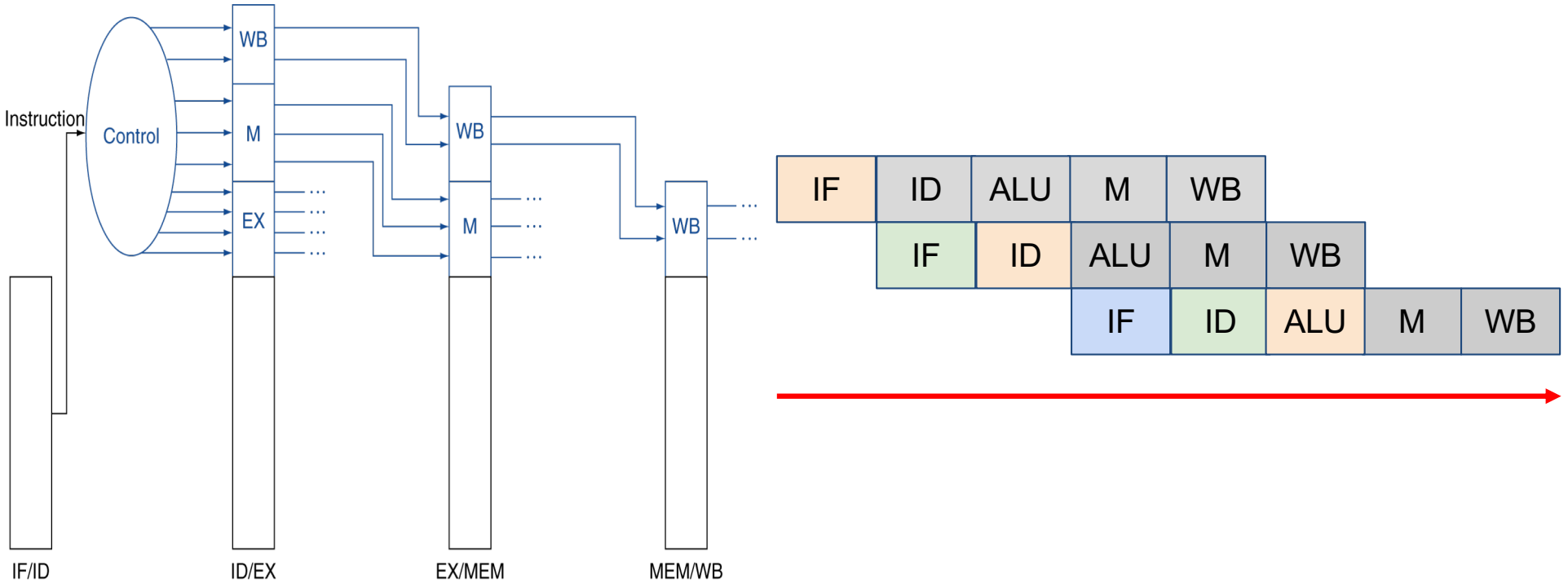


Instruction Level Parallelism (ILP)

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction level parallelism*
- Later: Other types of parallelism
  - DLP: same operation on lots of data (SIMD)
  - TLP: executing multiple threads “simultaneously” (OpenMP)

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages
- At any given point, there are up to 5 different instructions in the datapath! We must keep track of 5 different sets of control bits!



Question: Assume the stage times shown below.

Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

Question: Assume the stage times shown below.

Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

No mem access

throughput:

(IF+ID+EX+WB) = 600 →

(4*max_stage)/4 = 200

old/new = 600/200 = 3x faster

Question: Assume the stage times shown below.

Suppose we *remove loads and stores* from our ISA. Consider going from a single-cycle implementation to a **4-stage** pipelined version.

Instr Fetch	Reg Read	ALU Op	Mem Access	Reg Write
200ps	100 ps	200ps	200ps	100 ps

1) The *latency* will be 1.25x slower.
2) The *throughput* will be 3x faster.

	1	2
(A)	F	F
(B)	F	T
(C)	T	F
(D)	T	T

No mem access! Latency:
IF+ID+EX+WB = 600 →
4*max_stage = 800
old/new = 600/800 = negative
speedup! 800/600 = 1.33x slower!

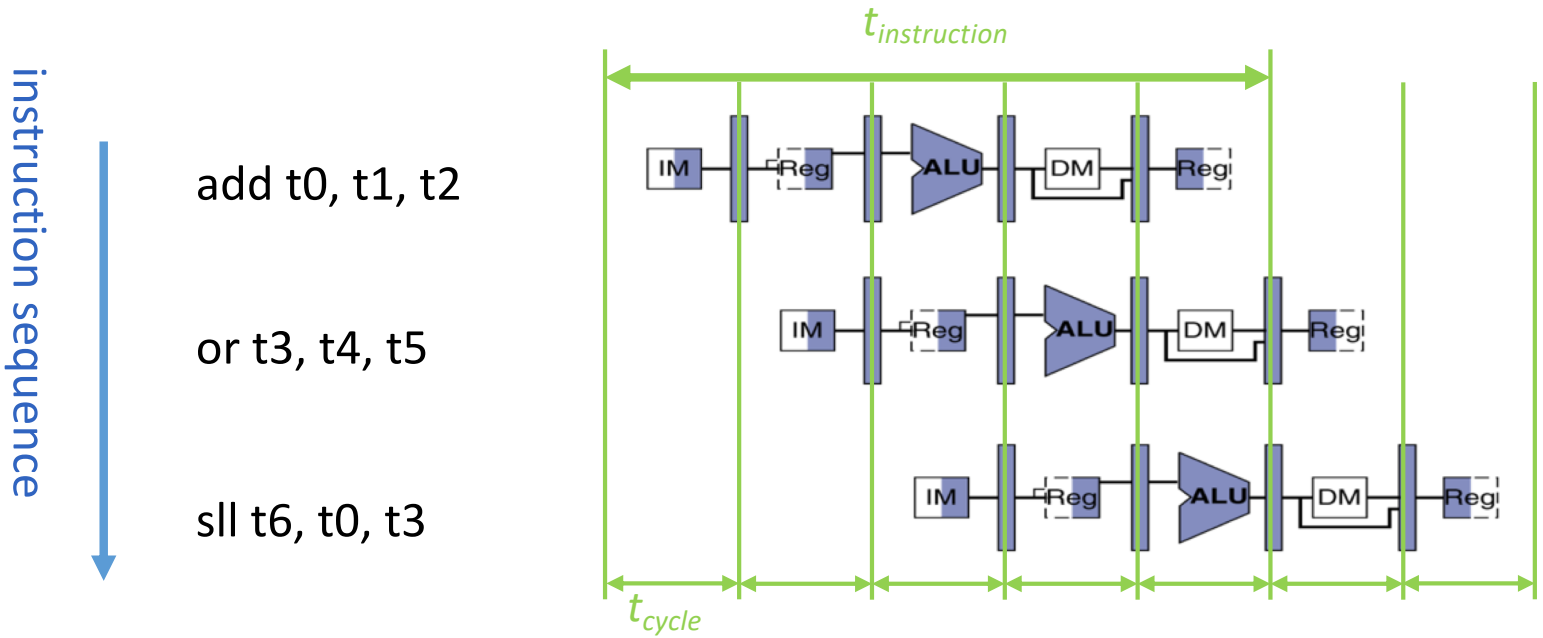
Summary

- Pipelining improves performance by exploiting Instruction Level Parallelism
  - 5-stage pipeline for RV32I: IF, ID, EX, MEM, WB
  - Executes multiple instructions in parallel
  - Each instruction has the same latency, but there's better throughput
  - Think: what problems does pipelining introduce?

Agenda

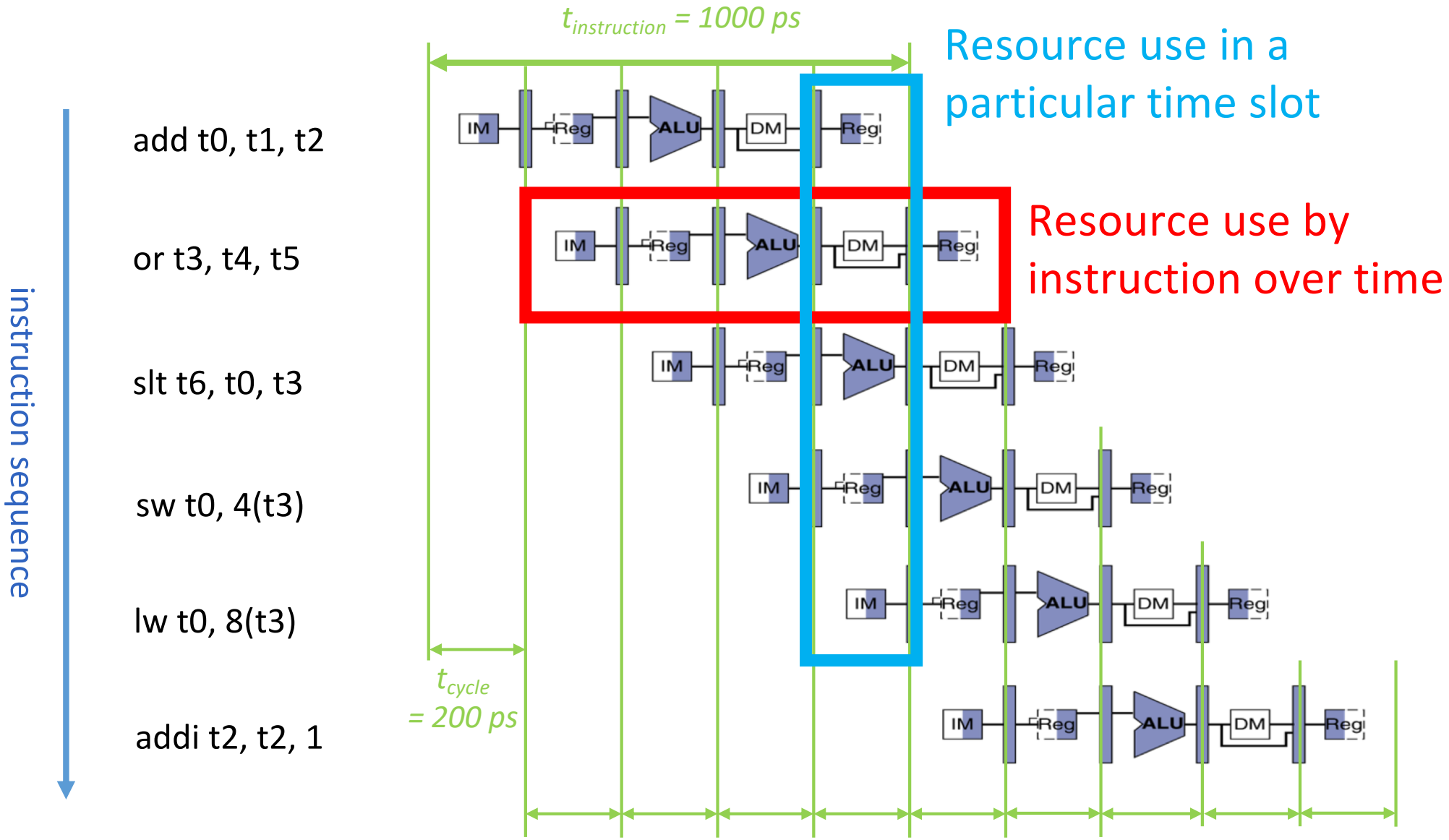
- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Recap: Pipelining with RISC-V

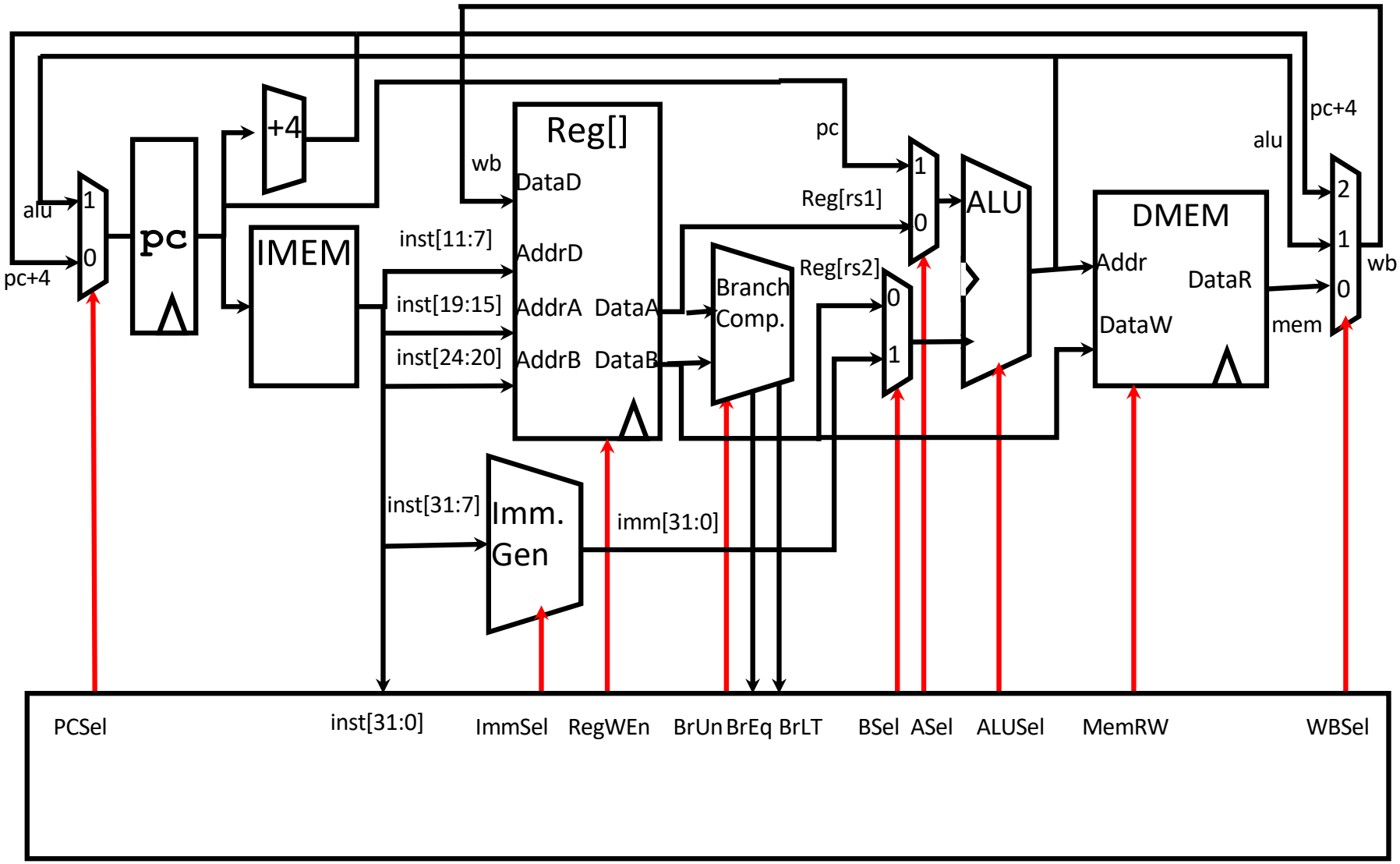


	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

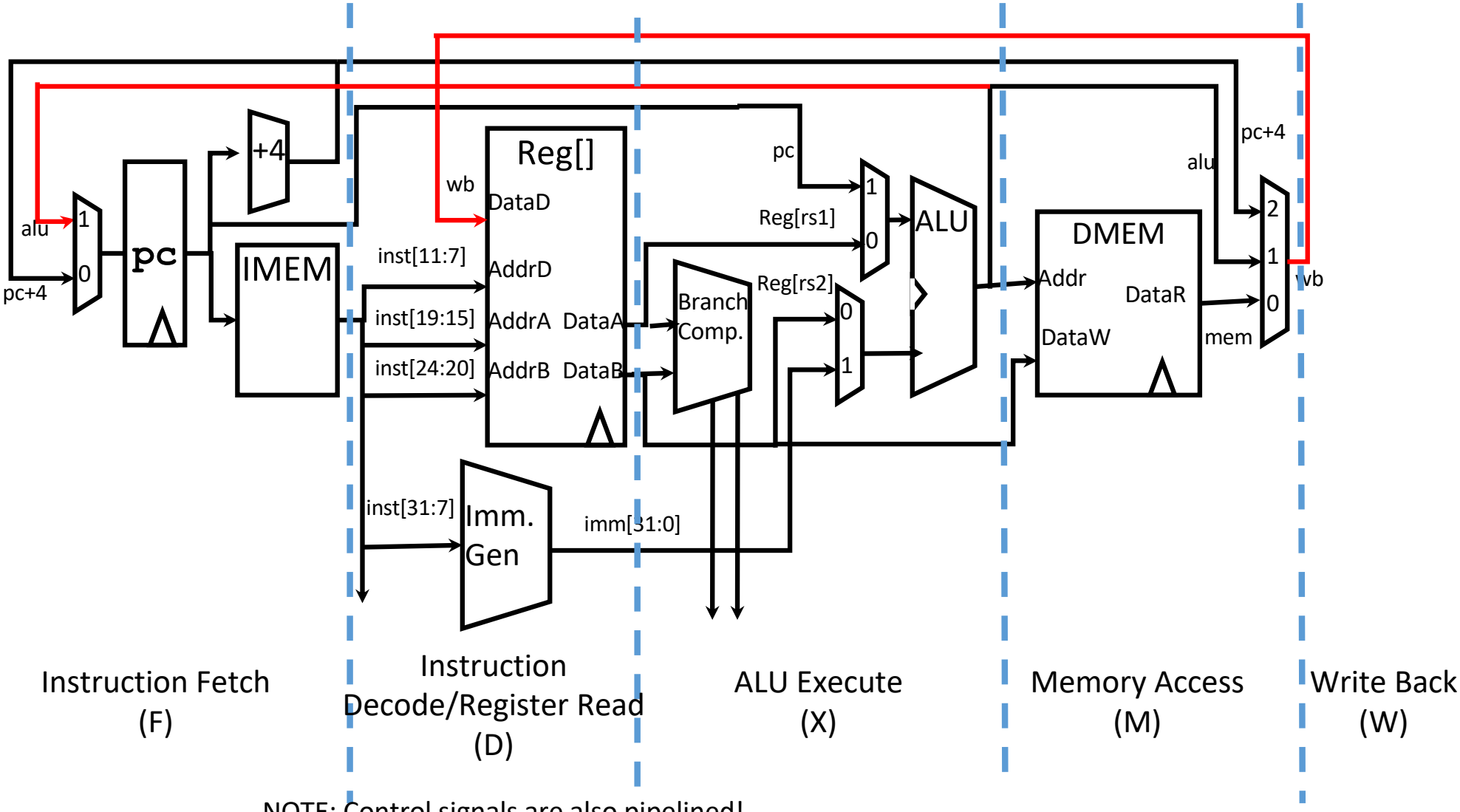
RISC-V Pipeline



Single-Cycle RISC-V RV32I Datapath

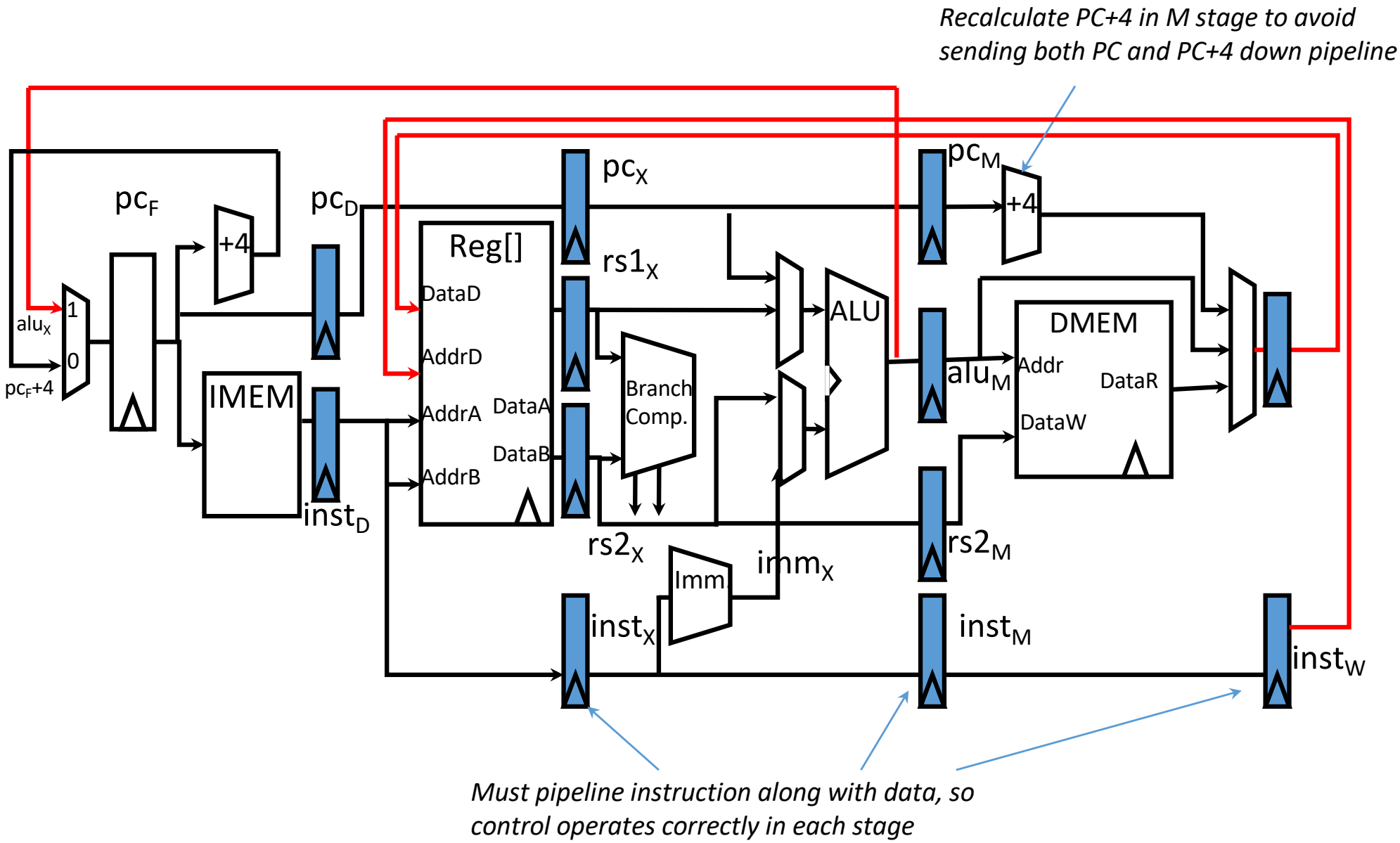


Pipelining RISC-V RV32I Datapath

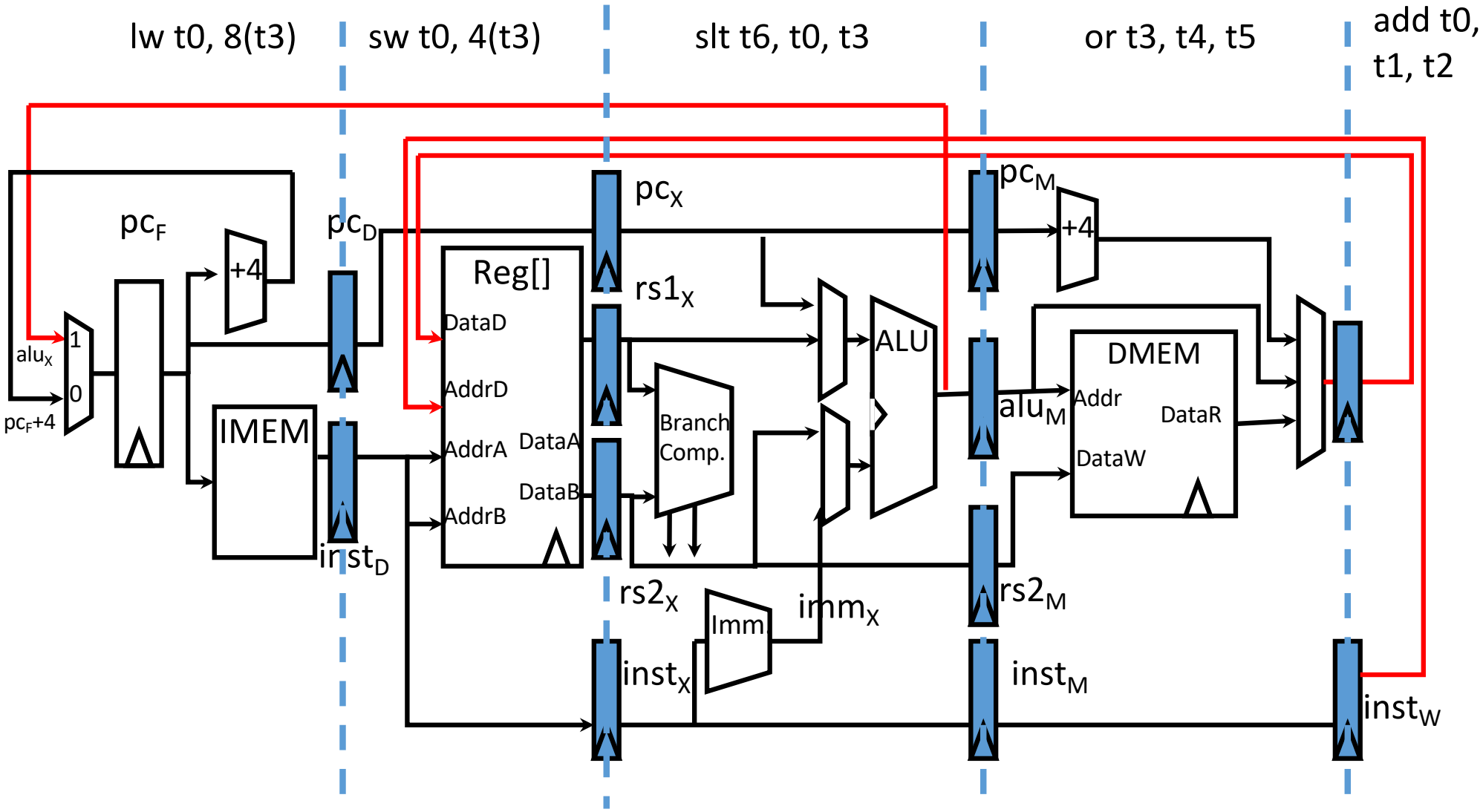


NOTE: Control signals are also pipelined!

Pipelined RISC-V RV32I Datapath



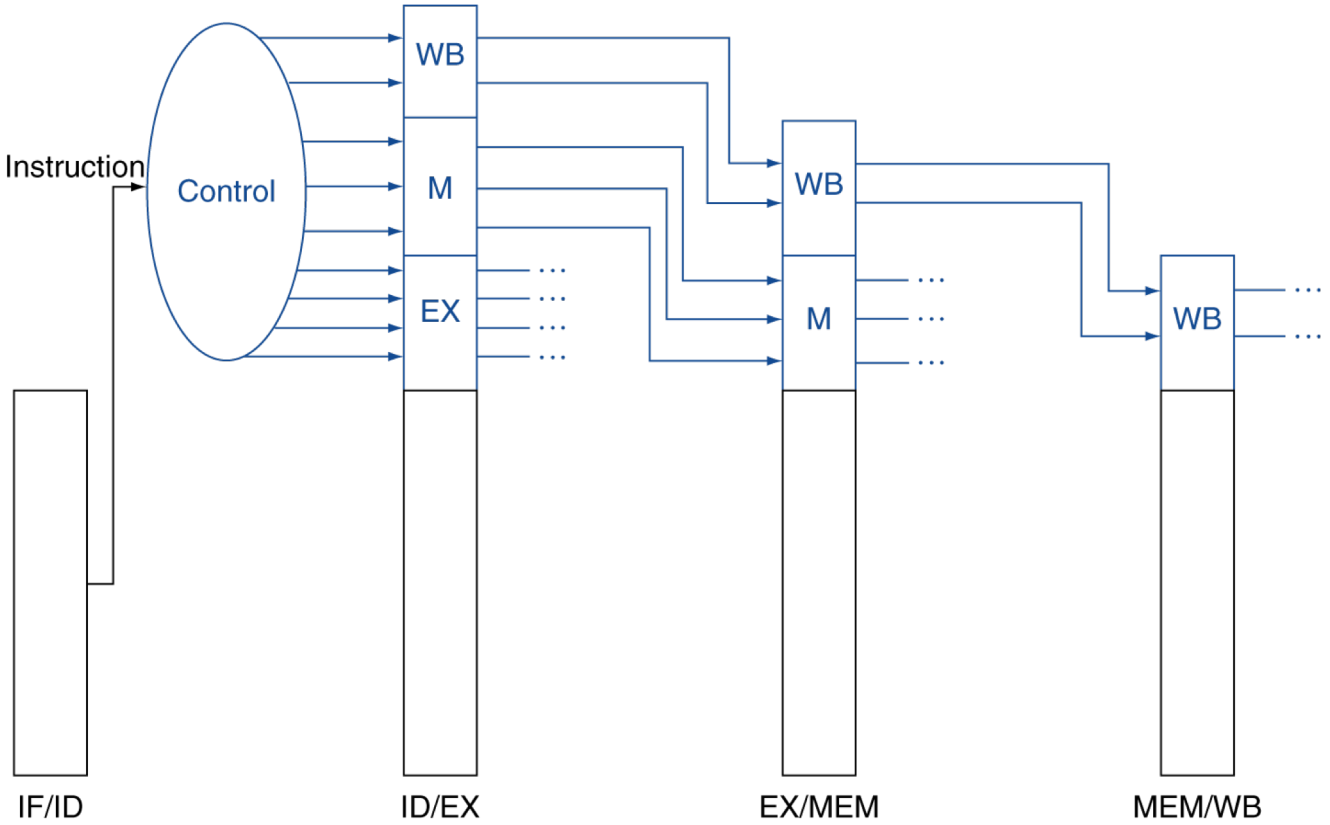
Each stage operates on different instruction



Pipeline registers separate stages, hold data for each instruction in flight

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation
  - Information is stored in pipeline registers for use by later stages



Question: Which of the following signals for RISC-V does NOT need to be passed into the EX pipeline stage for a beq instruction?

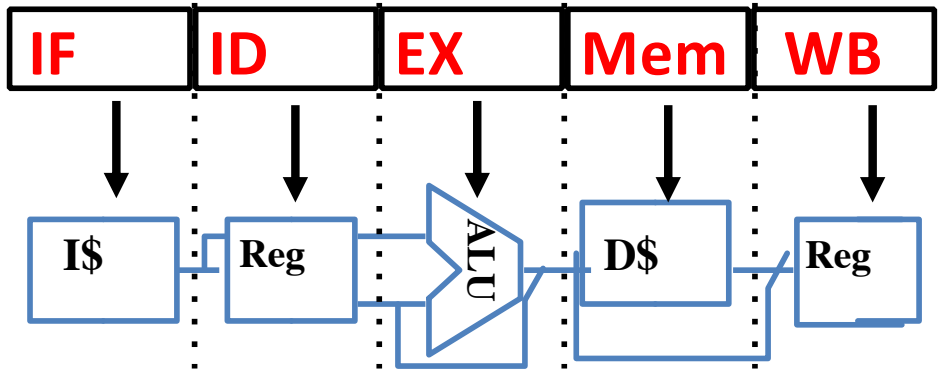
```
beq t0 t1 Label
```

(A) BrUn

(B) MemWr

(C) RegWr

(D) WBSel



Agenda

- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Hazards Ahead!

Pipelining Hazards

A *hazard* is a situation that prevents starting the next instruction in the next clock cycle

1) *Structural hazard*
   - A required resource is busy (e.g. needed in multiple stages)
2) *Data hazard*
   - Data dependency between instructions
   - Need to wait for previous instruction to complete its data write
3) *Control hazard*
   - Flow of execution depends on previous instruction

Agenda

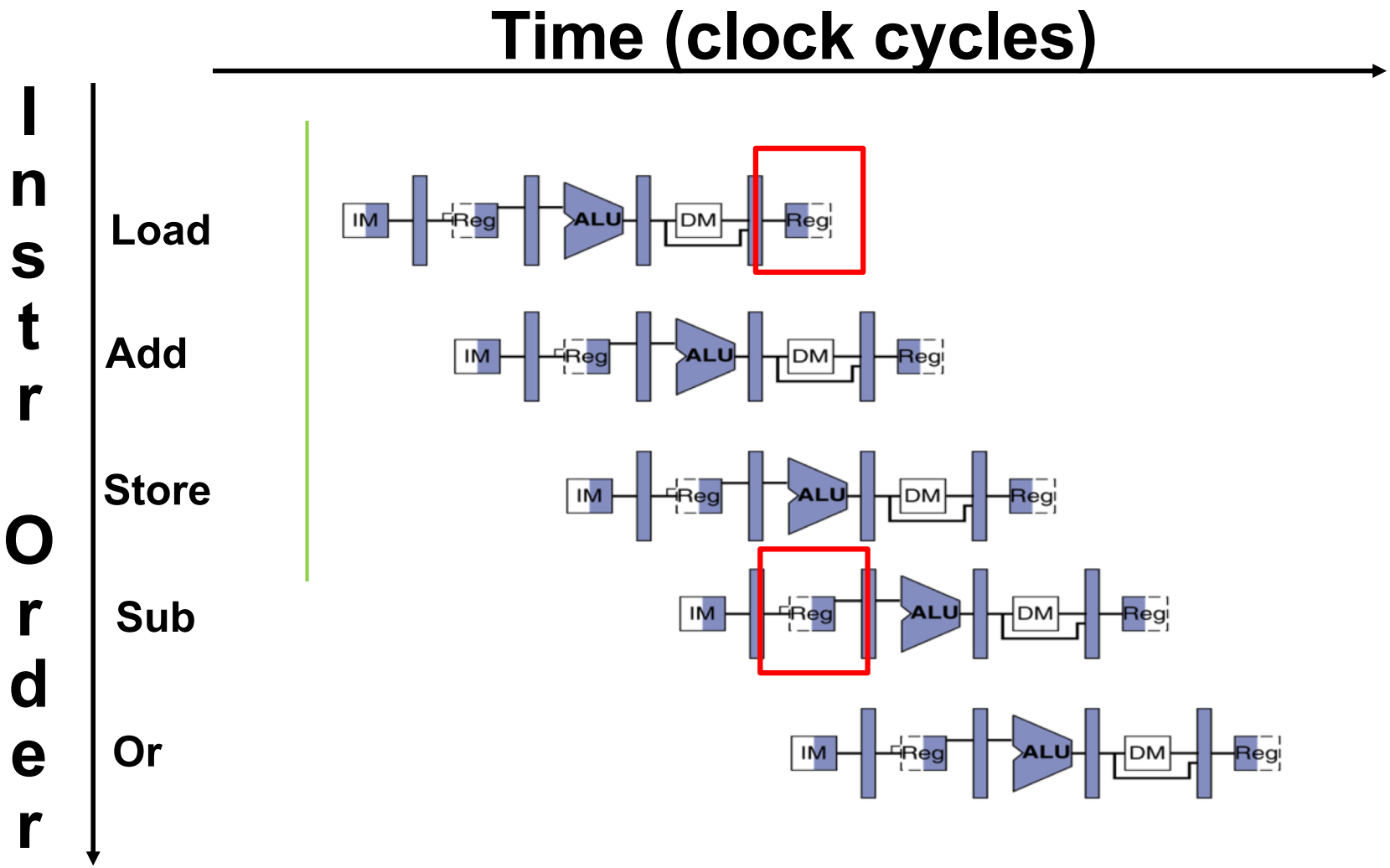
- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Structural Hazard

- **Problem:** Two or more instructions in the pipeline compete for access to a single physical resource
- **Solution 1:** Instructions take turns using resource, some instructions have to stall (wait)
- **Solution 2:** Add more hardware to machine
- Can always solve a structural hazard by adding more hardware

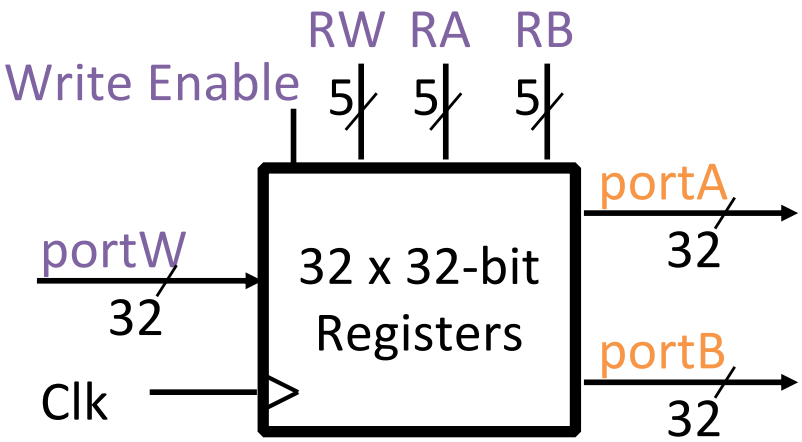
Structural Hazard: Regfile!

- RegFile: Used in ID and WB!



Regfile Structural Hazards

- Each instruction:
  - can read up to two operands in decode stage
  - can write one value in writeback stage
- Avoid structural hazard by having separate “ports”
  - two independent read ports and one independent write port
- Three accesses per cycle can happen simultaneously

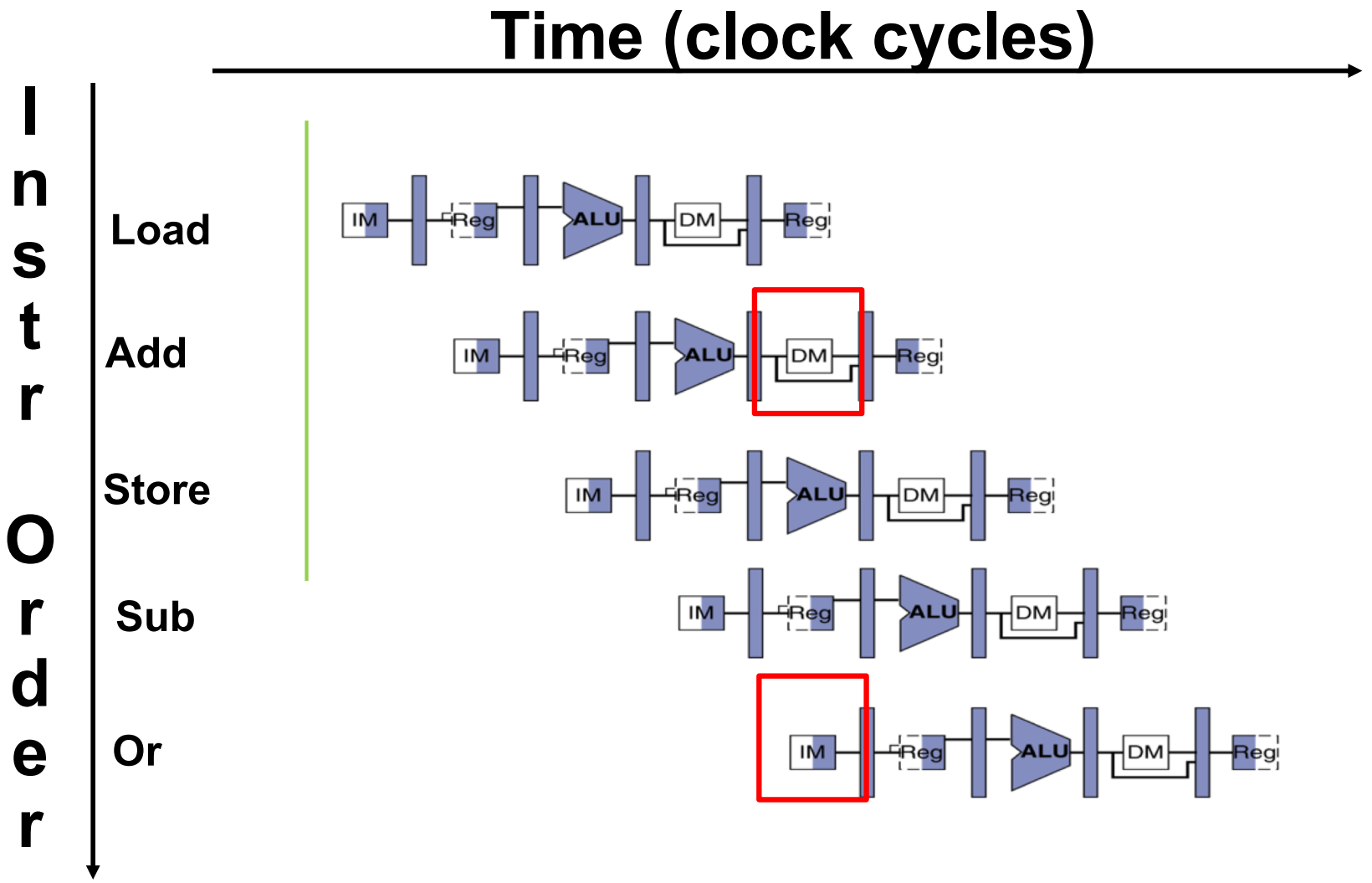


Regfile Structural Hazards

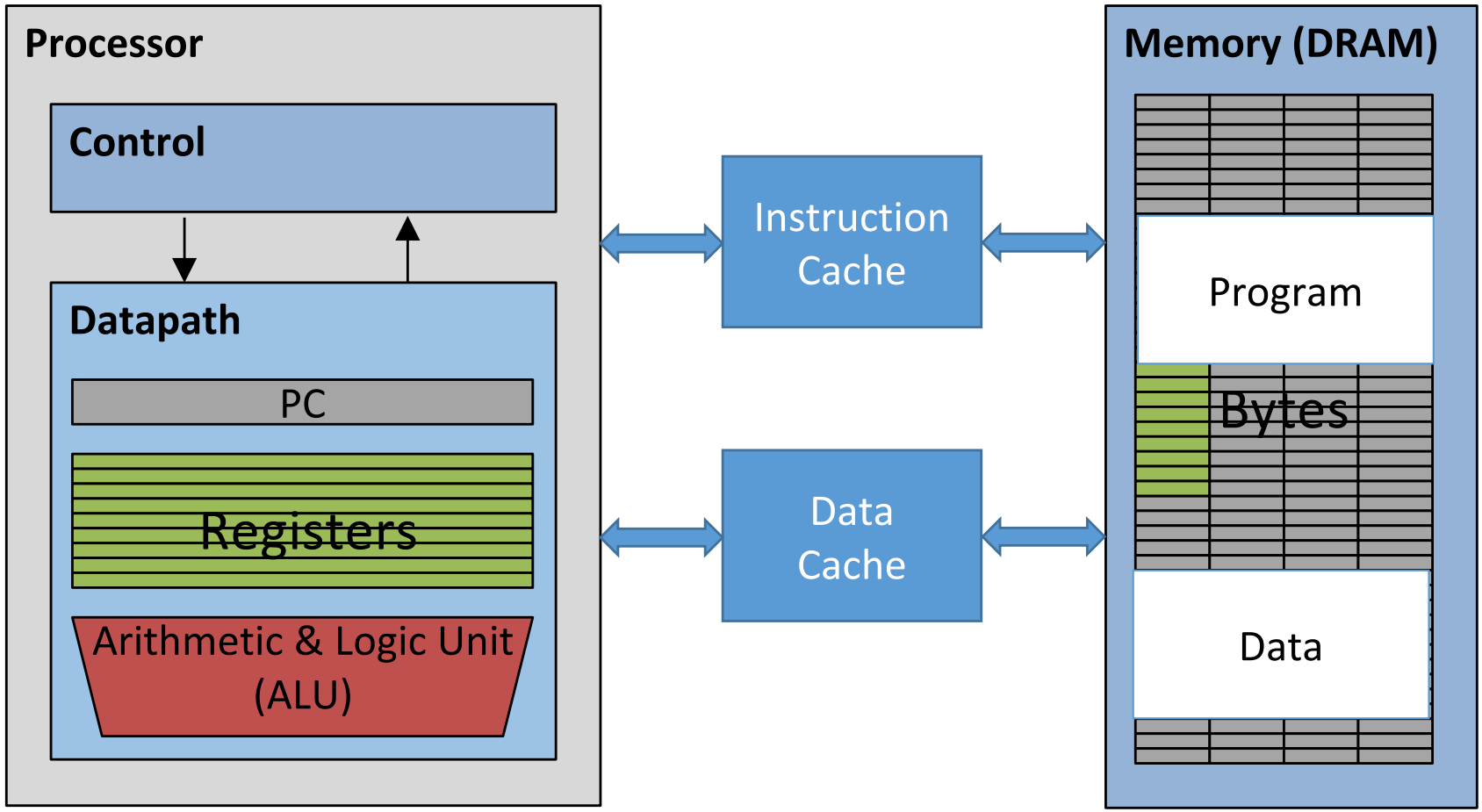
- We use *two solutions* simultaneously :
  1) Build RegFile with independent read and write ports (good for single-stage)
  2) Double Pumping: split RegFile access in two! Prepare to write during 1st half, write on *falling* edge, read during 2nd half of each clock cycle
     - Will save us a cycle later...
     - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Structural Hazard: Memory!

- Memory units: Used in IF and MEM!



Instruction and Data Caches



Caches: small and fast “buffer” memories

Structural Hazards – Summary

- Conflict for use of a resource
- In RISC-V pipeline with a single memory unit
  - Load/store requires data access
  - Without separate memory units, instruction fetch would have to stall for that cycle
    - All other operations in pipeline would have to wait
- Pipelined datapaths require separate instruction/data memory units
  - Or separate instruction/data caches
- RISC ISAs (including RISC-V) designed to avoid structural hazards
  - e.g. at most one memory access/instruction

Agenda

- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

2. Data Hazards (1/2)

- Consider the following sequence of instructions:

```
add  t0, t1, t2
sub  t4, t0, t3
and  t5, t0, t6
or   t7, t0, t8
xor  t9, t0, t10
```

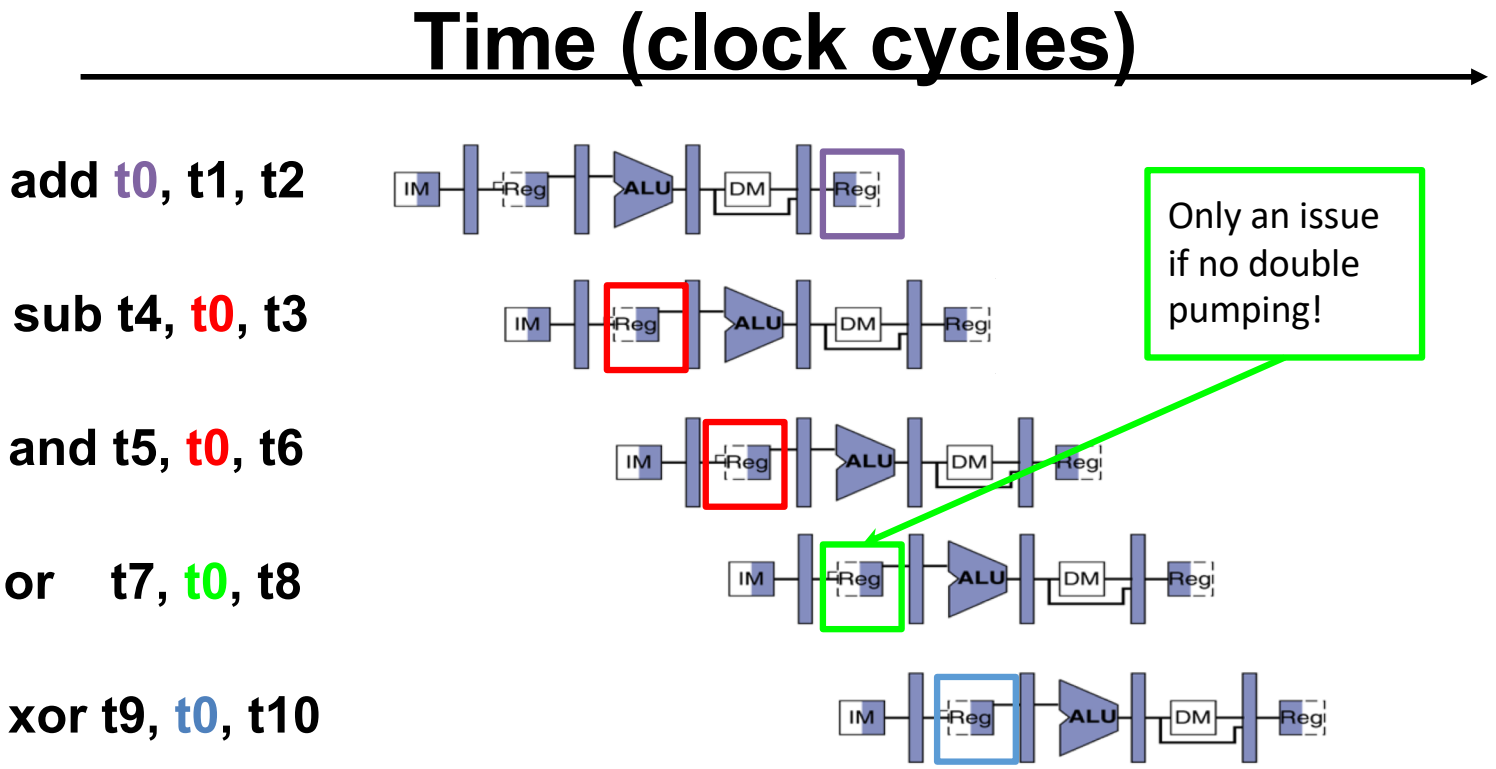
Stored during WB

Read during ID

2. Data Hazards (2/2)

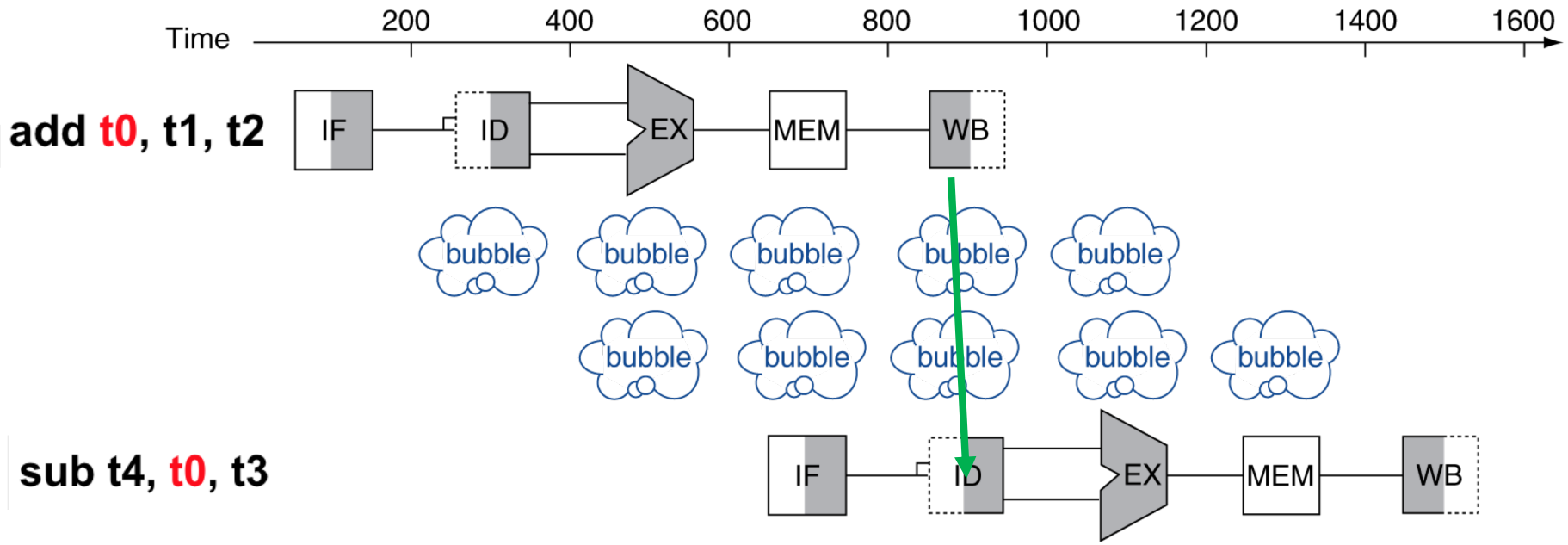
Identifying data hazards:

- Where is data **WRITTEN**?
- Where is data **READ**?
- Does the WRITE happen AFTER the READ?



Solution 1: Stalling

- Problem: Instruction depends on result from previous instruction
  - add t0, t1, t2
    sub t4, t0, t3



- Bubble:
  - effectively NOP: affected pipeline stages do “nothing” (add x0 x0 x0)

Stalls and Performance

- Stalls reduce performance
  - Decrease throughput of “valid” or useful instructions
  - Can also be seen as increasing the latency of our stalled instruction
- But stalls are required to get correct results
- Compiler can arrange code to avoid hazards and stalls!
  - Requires knowledge of the pipeline structure, and knowledge of instruction interactions

Data Hazard Solution: Forwarding

- Forward result as soon as it is available, even though it's not stored in RegFile yet

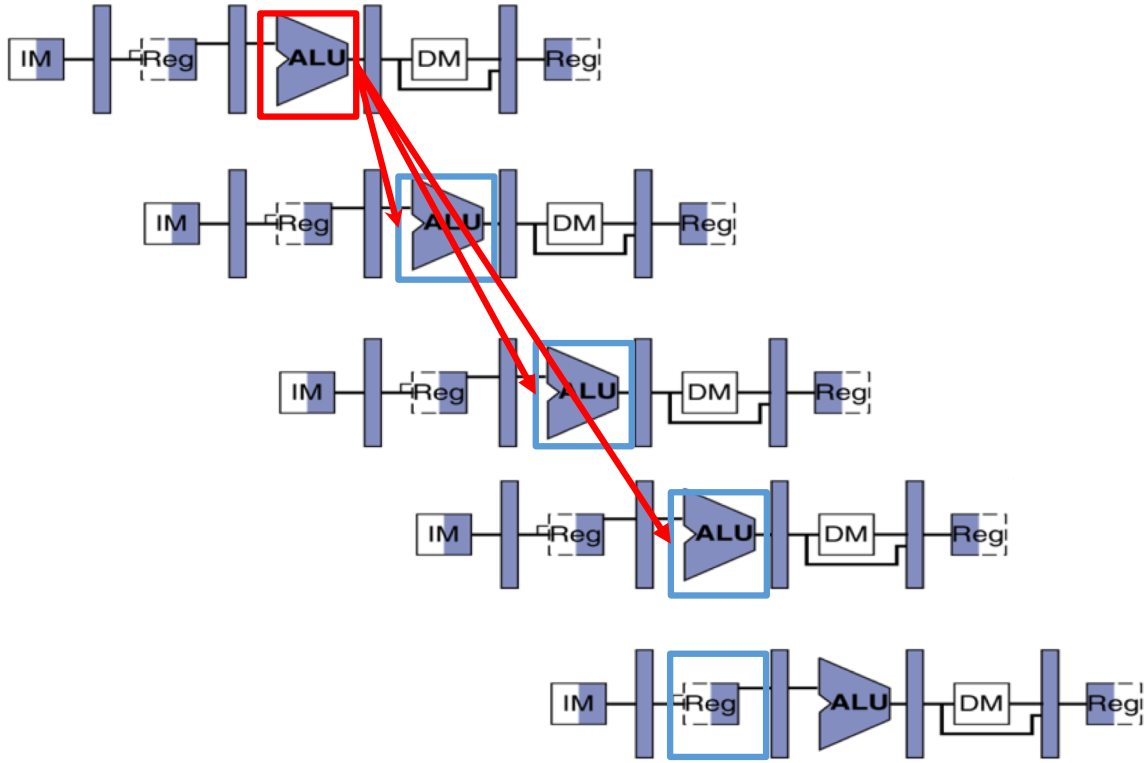
add t0, t1, t2

sub t4, t0, t3

and t5, t0, t6

or t7, t0, t8

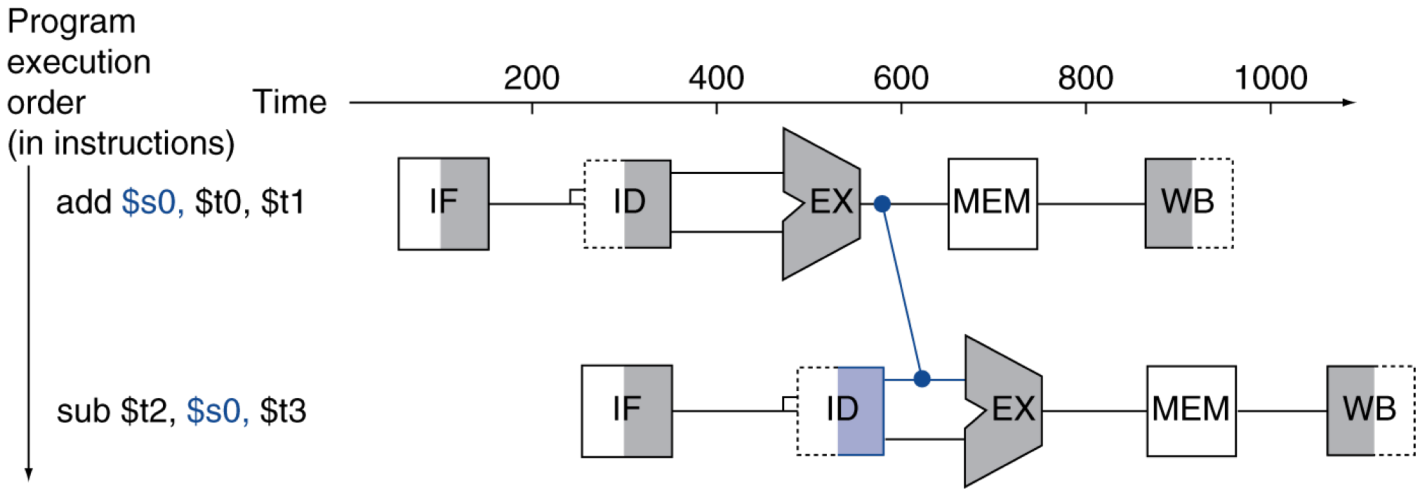
xor t9, t0, t10



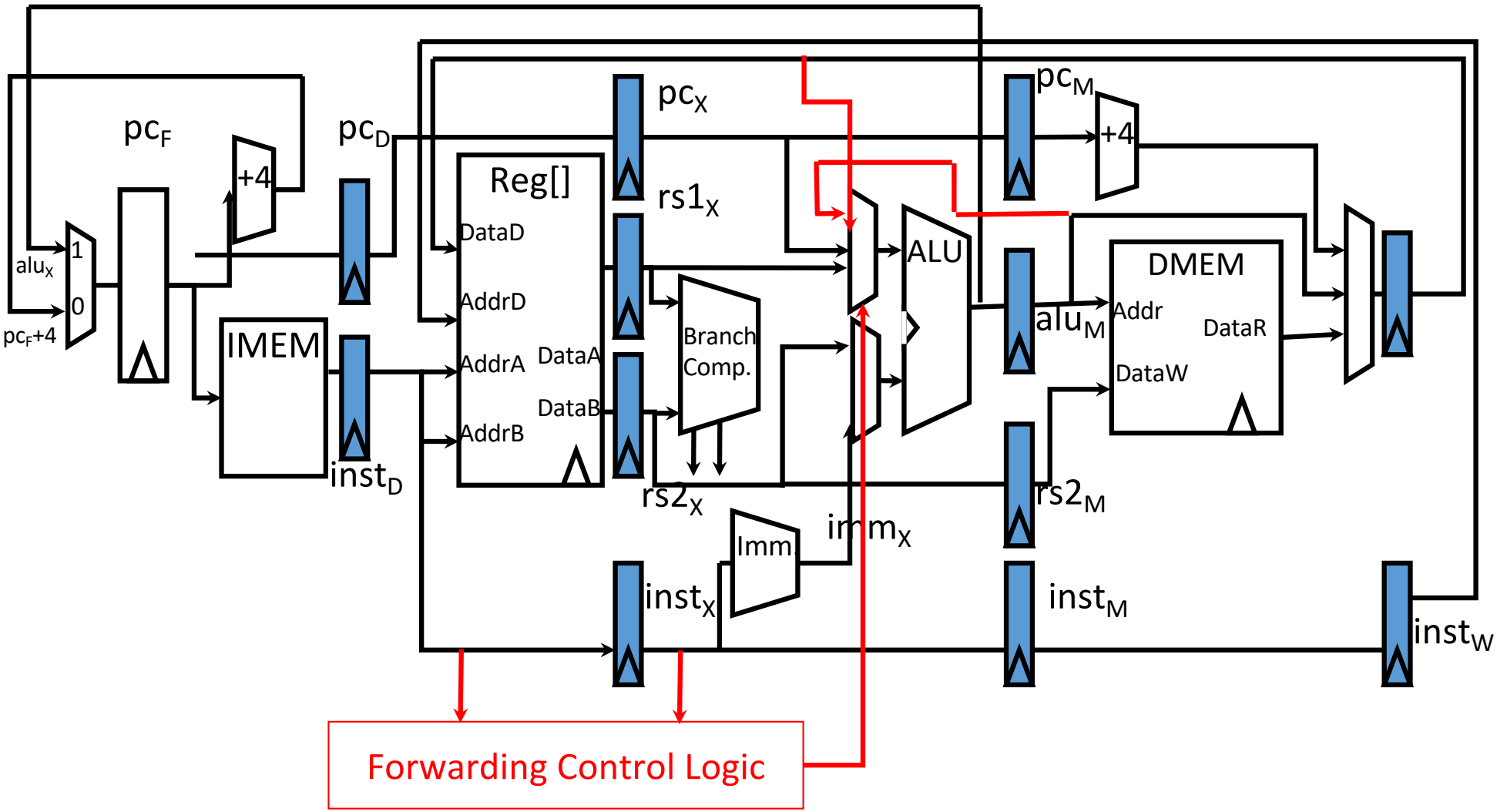
Forwarding: grab operand from pipeline stage, rather than register file

Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra hardware in the datapath (and extra control!)

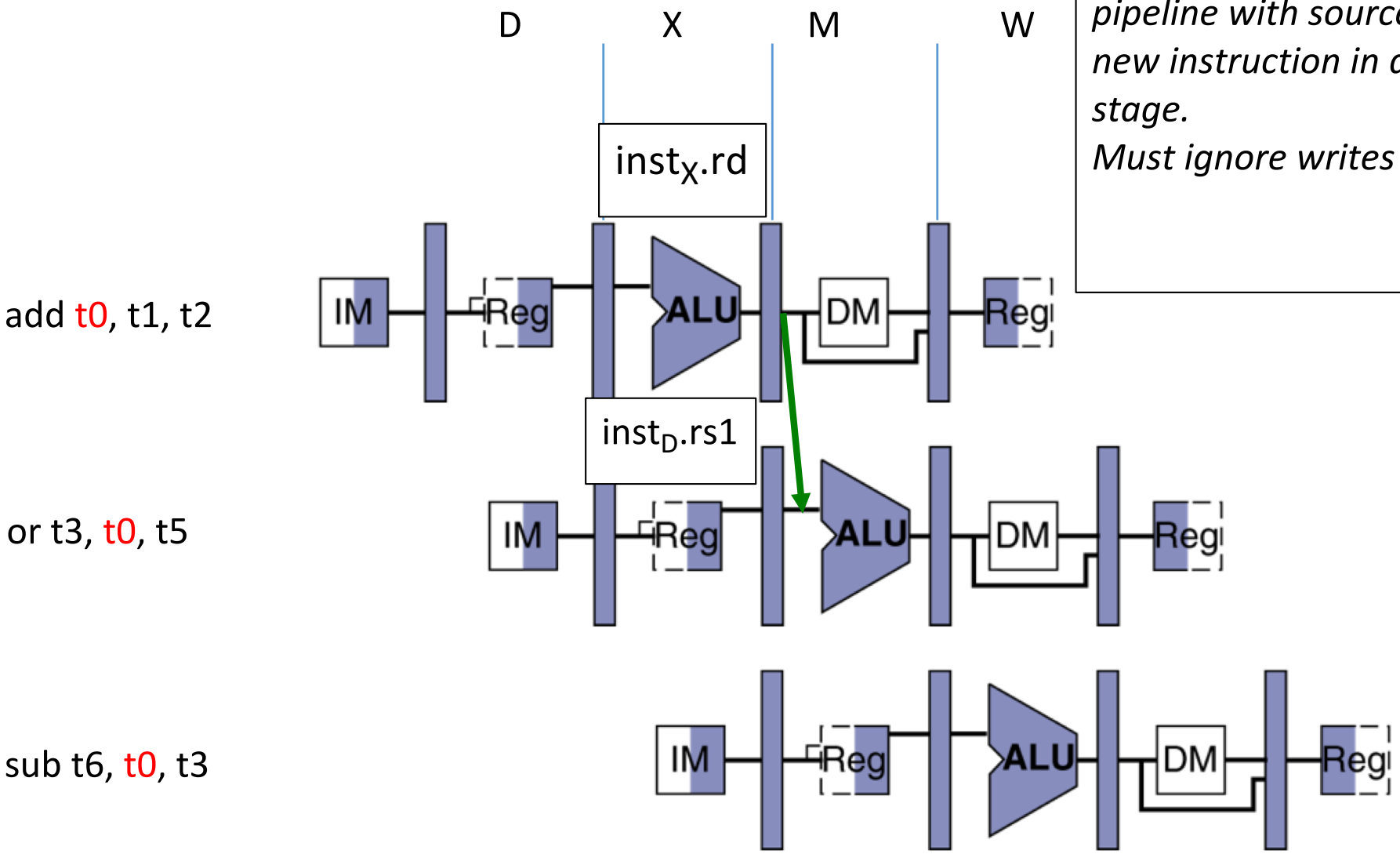


Forwarding Path



Detect Need for Forwarding (example)

Compare destination of older instructions in pipeline with sources of new instruction in decode stage.
Must ignore writes to x0!

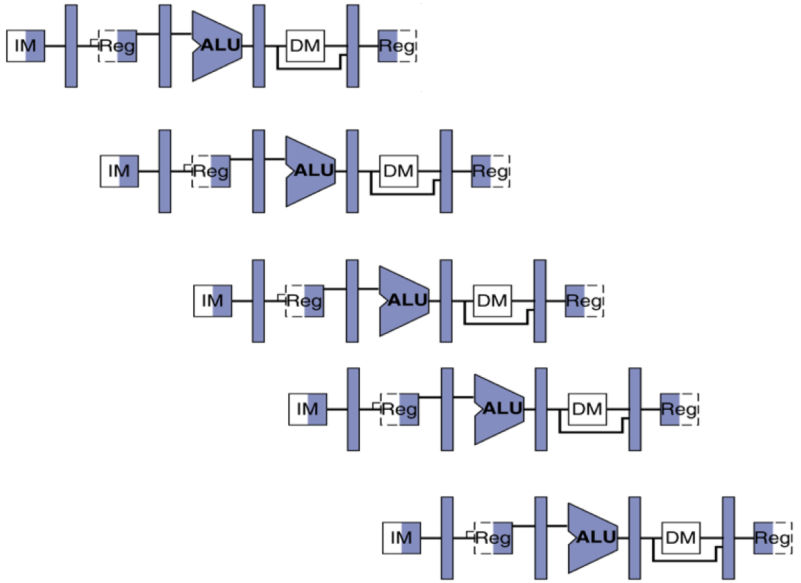


Question

In our 5-stage pipeline, how many subsequent instructions do we need to look at to detect data hazards for this **add**? Assume we have double-pumping.

A) 1 instruction
B) 2 instructions
C) 3 instructions
D) 4 instructions
E) 5 instructions

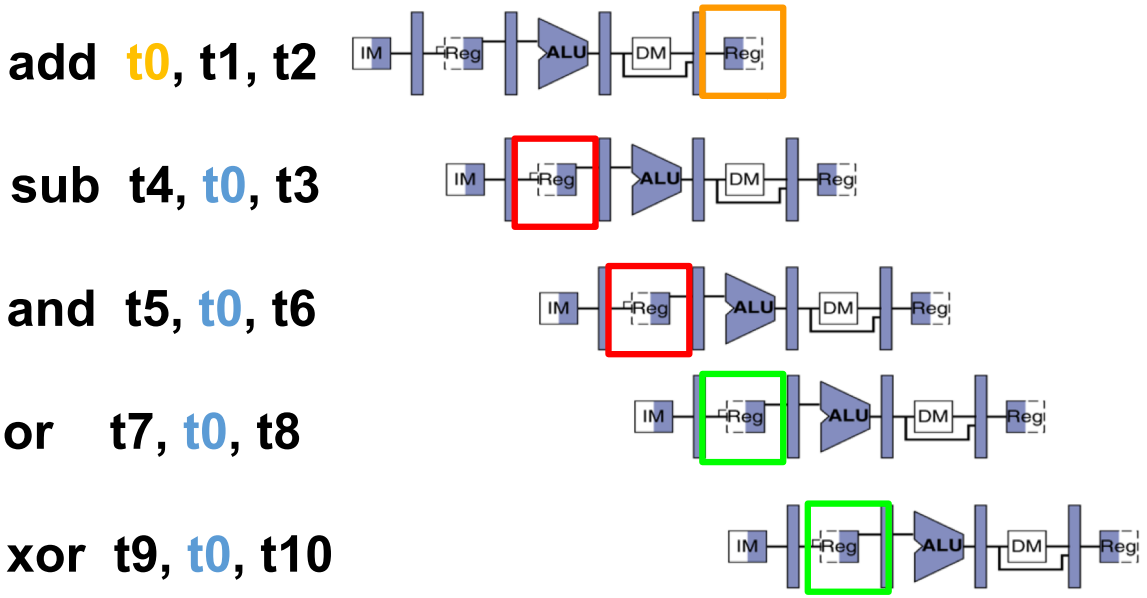
```
add  t0, t1, t2
sub  t4, t0, t3
and  t5, t0, t6
or   t7, t0, t8
xor  t9, t0, t10
```



Question

In our 5-stage pipeline, how many subsequent instructions do we need to look at to detect data hazards for this add? Assume we have double-pumping.

A) 1 instruction
B) 2 instructions
C) 3 instructions
D) 4 instructions
E) 5 instructions



Agenda

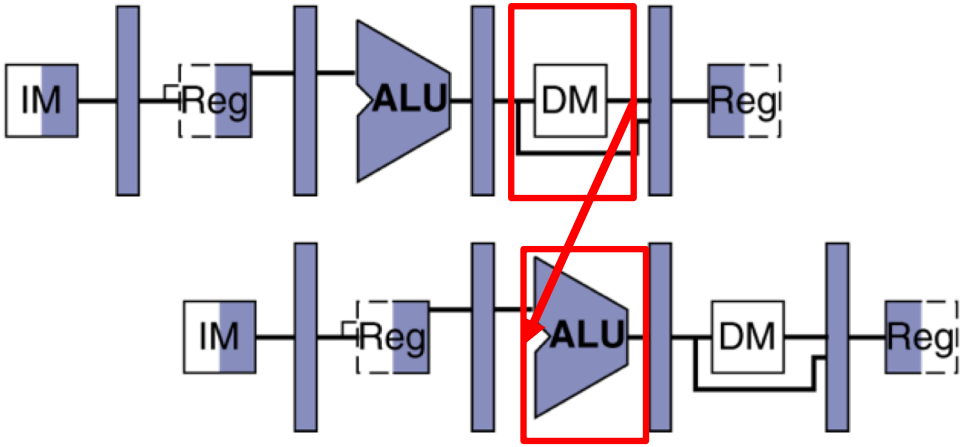
- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Data Hazard: Loads (1/4)

- **Recall:** Dataflow backwards in time are hazards

lw t0, 0(t1)

sub t3, t0, t2



- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load (sub), then forward after the load is done (more hardware)

Data Hazard: Loads (2/4)

- *Hardware* stalls pipeline
  - Called “hardware interlock”

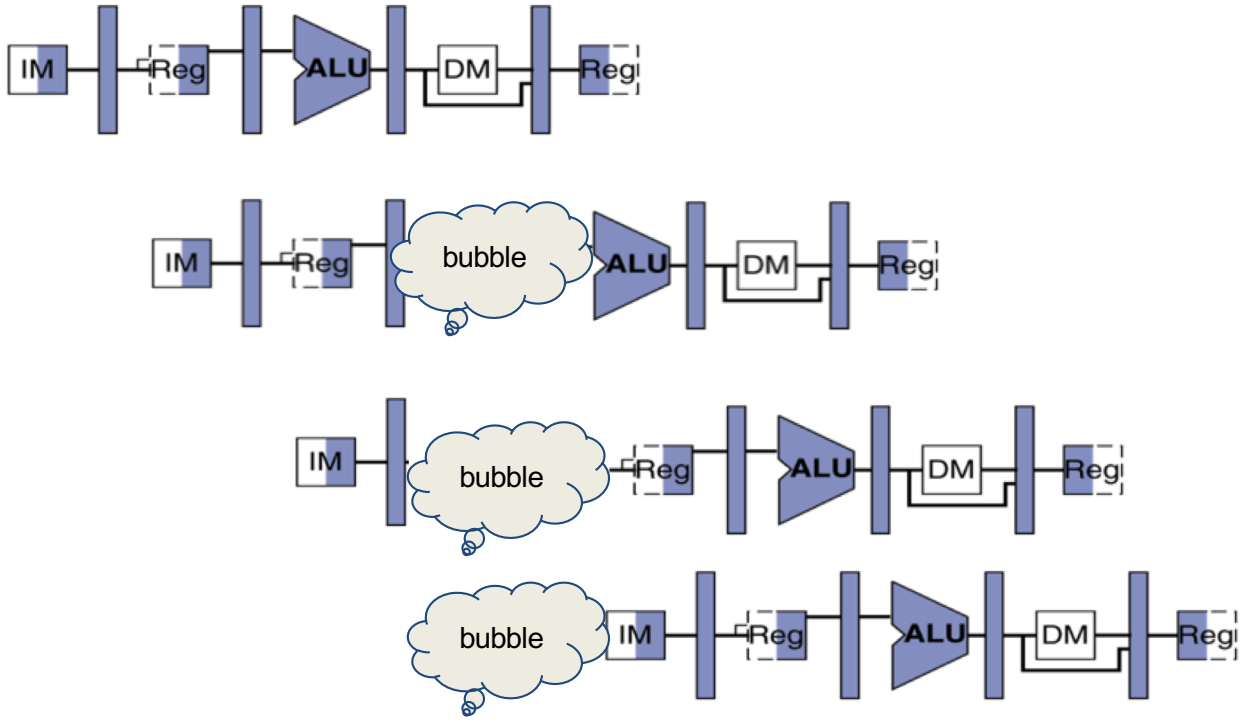
This is what happens in hardware in a “hardware interlock”

lw t0, 0(t1)

sub t3, t0, t2

and t5, t0, t4

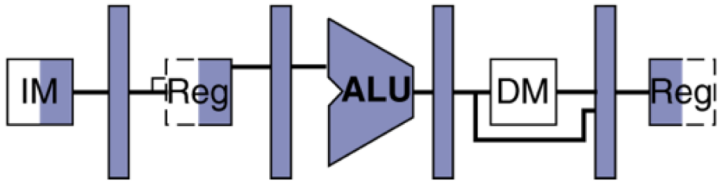
or t7, t0, t6



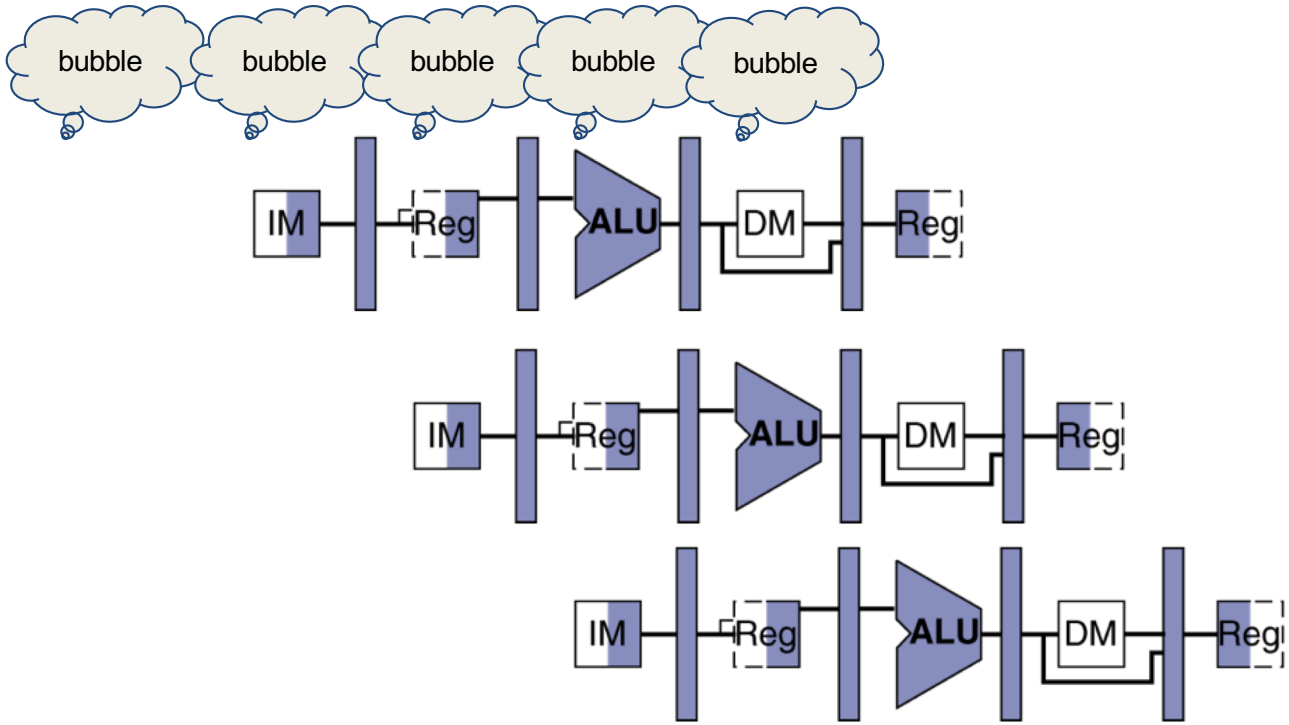
Data Hazard: Loads (3/4)

- Stall is equivalent to `nop`

lw t0, 0(t1)



nop



sub t3, t0, t2

and t5, t0, t4

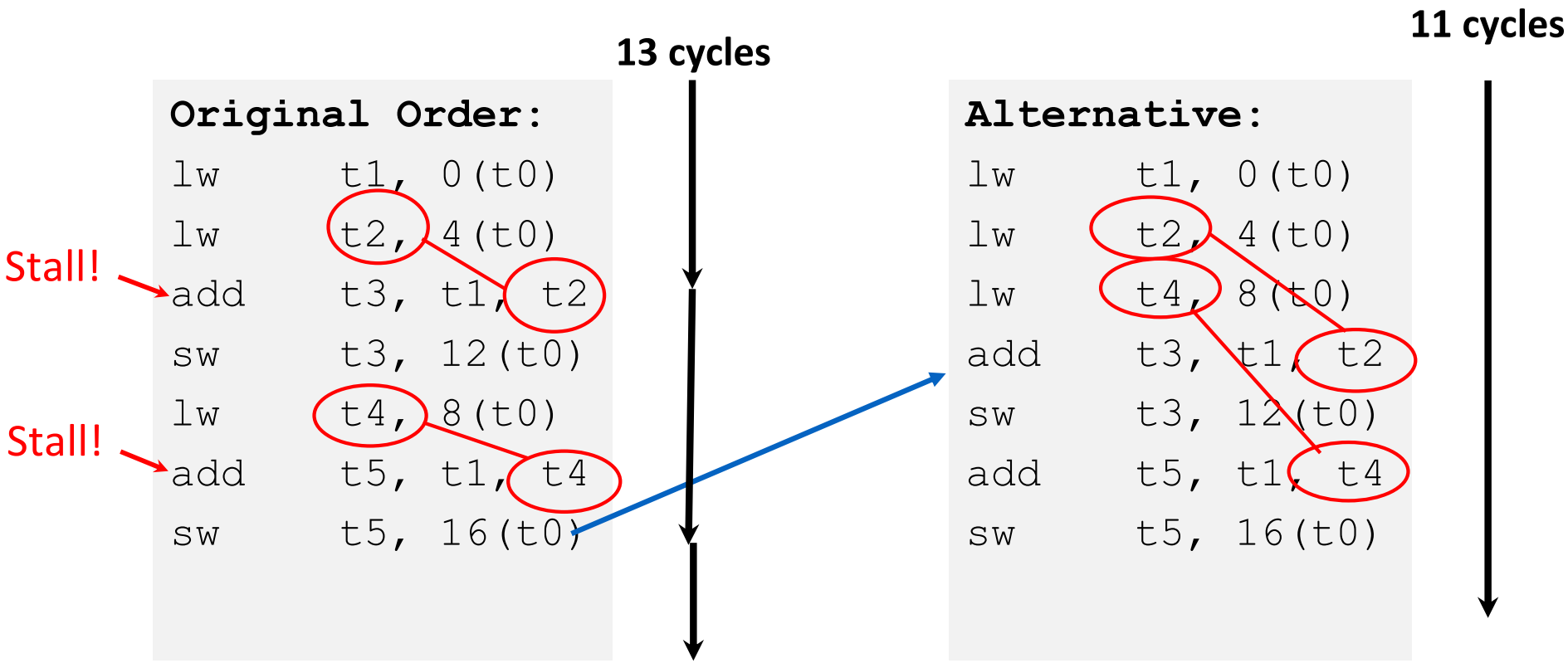
or t7, t0, t6

Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware will stall for one cycle
  - Equivalent to inserting an explicit **`nop`** in the slot
    - except the latter uses more code space
  - Performance loss
- **Idea:** Let the compiler/assembler put an unrelated instruction in that slot → no stall!

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction!
- RISC-V code for `D=A+B;  E=A+C;`



Agenda

- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

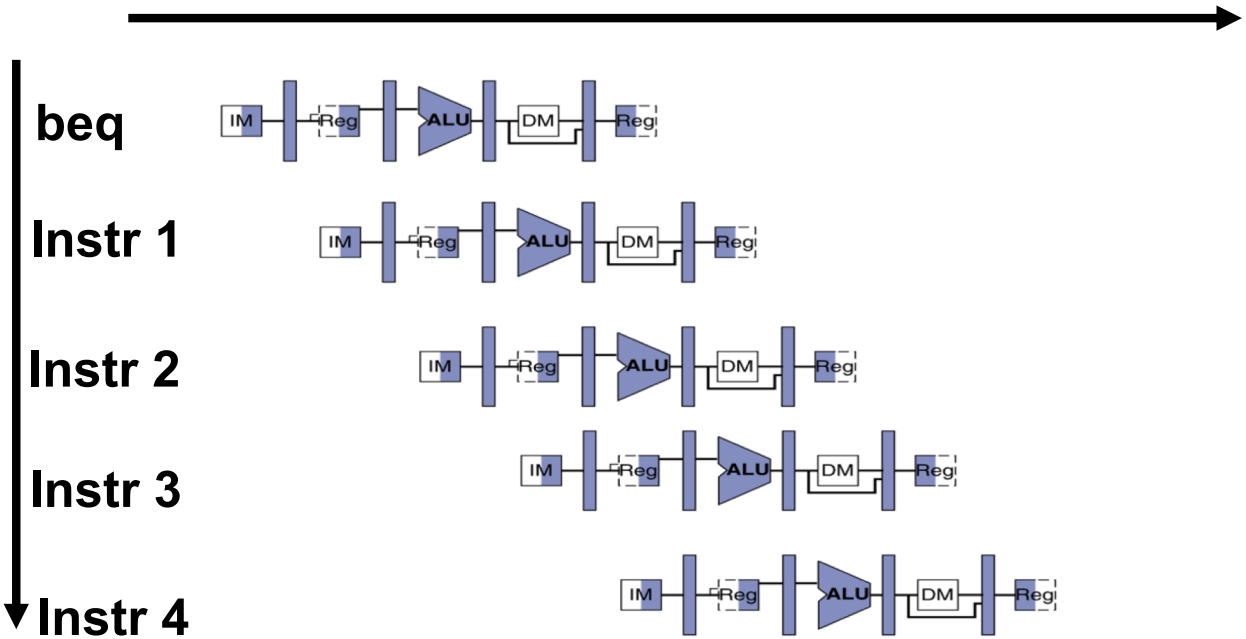
3. Control Hazards

- Branch (`beq`, `bne`, . . .) determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can't always fetch correct instruction
    - Result isn't known until end of execute
- **Simple Solution:** Stall on *every* branch until we have the new PC value
  - How long must we stall?

Question

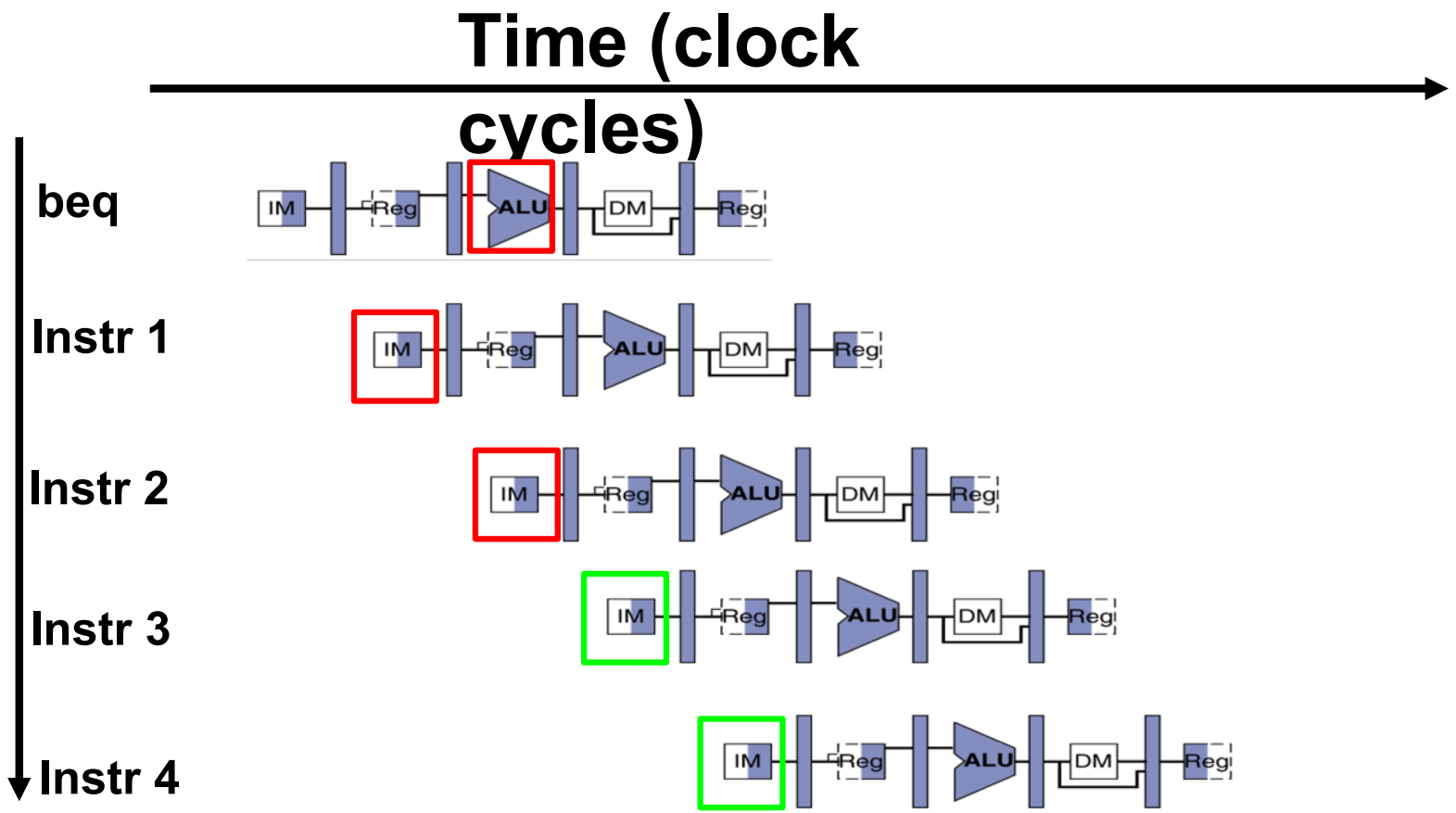
- How many instructions after **beq** are affected by the control hazard?

A) 1
B) 2
C) 3
D) 4
E) 5



Branch Stall

- How many bubbles required for branch?

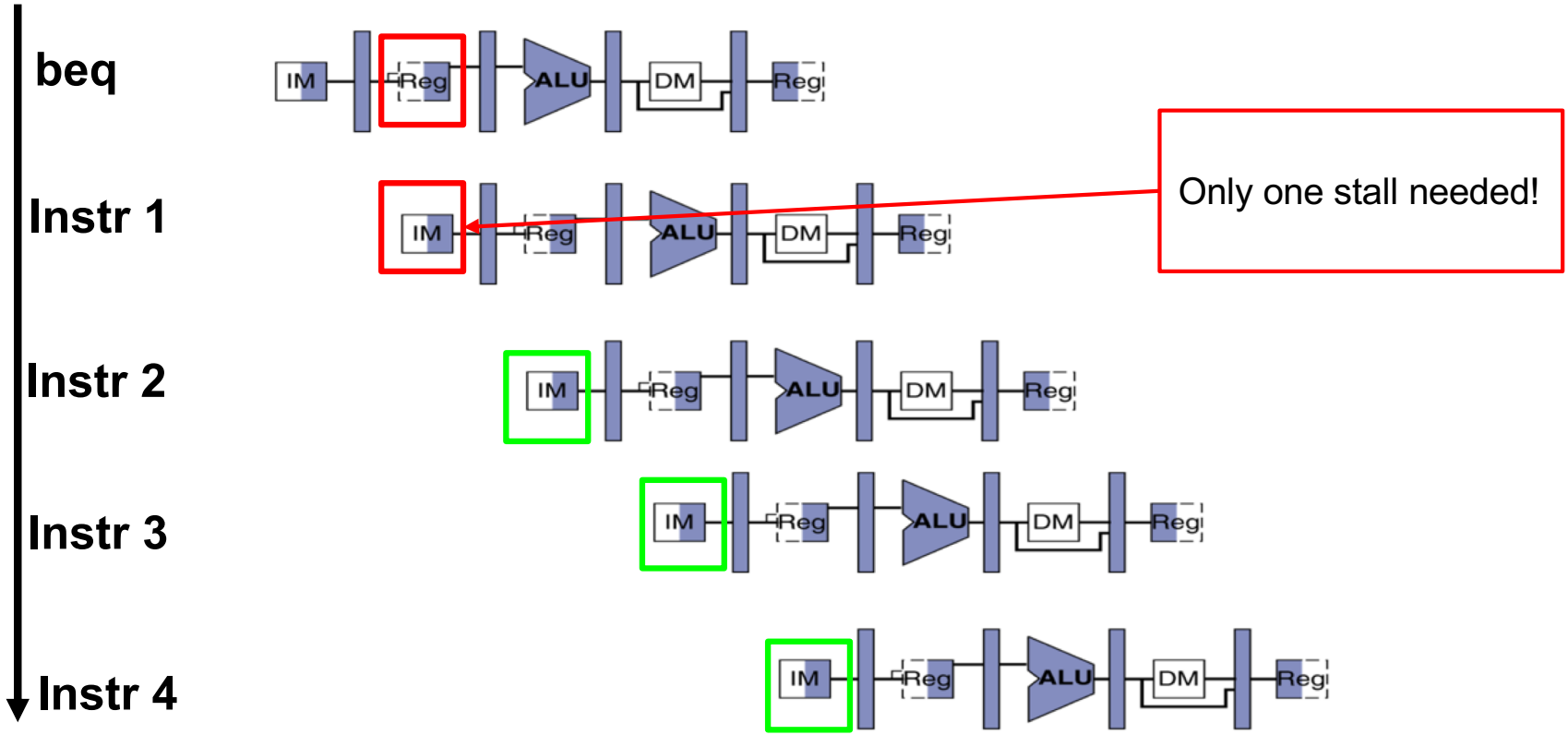


3. Control Hazard: Branching

- **Option #1:** Move **branch comparator** to ID stage
  - As soon as instruction is decoded, immediately make a decision and set the new value of PC
  - **Benefit:** Branch decision made in 2nd stage, so only one `nop` is needed instead of two
  - **Side Note:** Have to compute new PC value (PC + imm) in ID instead of EX
    - Adds extra copy of new-PC logic in ID stage
    - Branches are idle in EX, MEM, and WB

Improved Branch Stall

- When is comparison result available?

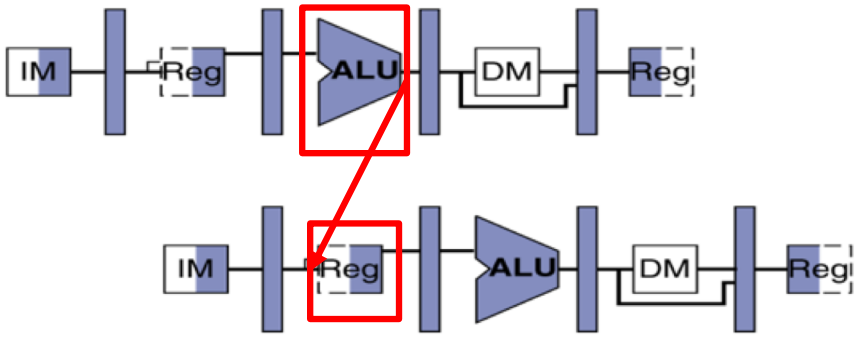


Data Hazard: Branches!

- **Recall:** Dataflow backwards in time are hazards

add t0, t0, t1

beq x0, t0, foo



- Now that t0 is needed earlier (ID instead of EX), we can't forward it to the beq's ID stage
  - Must *stall* after add, then forward (more hardware)

Observations

- **Takeaway**: Moving branch comparator to ID stage would add redundant hardware and introduce new problems
- Can we work with the nature of branches?
  - If branch not taken, then instructions fetched sequentially after branch are correct
  - If branch or jump taken, then need to flush incorrect instructions from pipeline by converting to NOPs

Agenda

- Structural Hazards
- Data Hazards
  - Forwarding
- Data Hazards (Continued)
  - Load Delay Slot
- Control Hazards
  - Branch and Jump Delay Slots
  - Branch Prediction

3. Control Hazard: Branching

- **RISC-V Solution:** *Branch Prediction* – guess outcome of a branch, fix afterwards if necessary
  - Must cancel (*flush*) all instructions in pipeline that depended on guess that was wrong
  - How many instructions do we end up flushing?

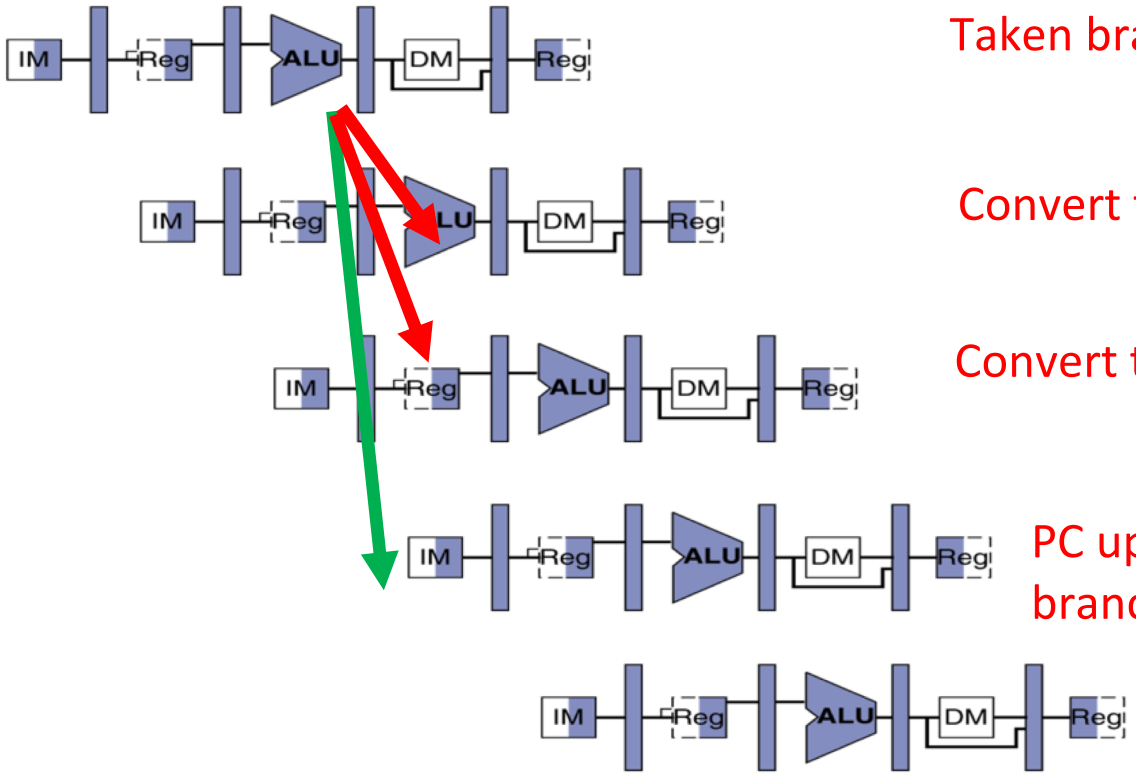
Kill Instructions after Branch if Taken

beq t0, t1, label

sub t2, s0, t5

or t6, s0, t3

label: xxxxxx



Taken branch

Convert to NOP

Convert to NOP

PC updated reflecting branch outcome

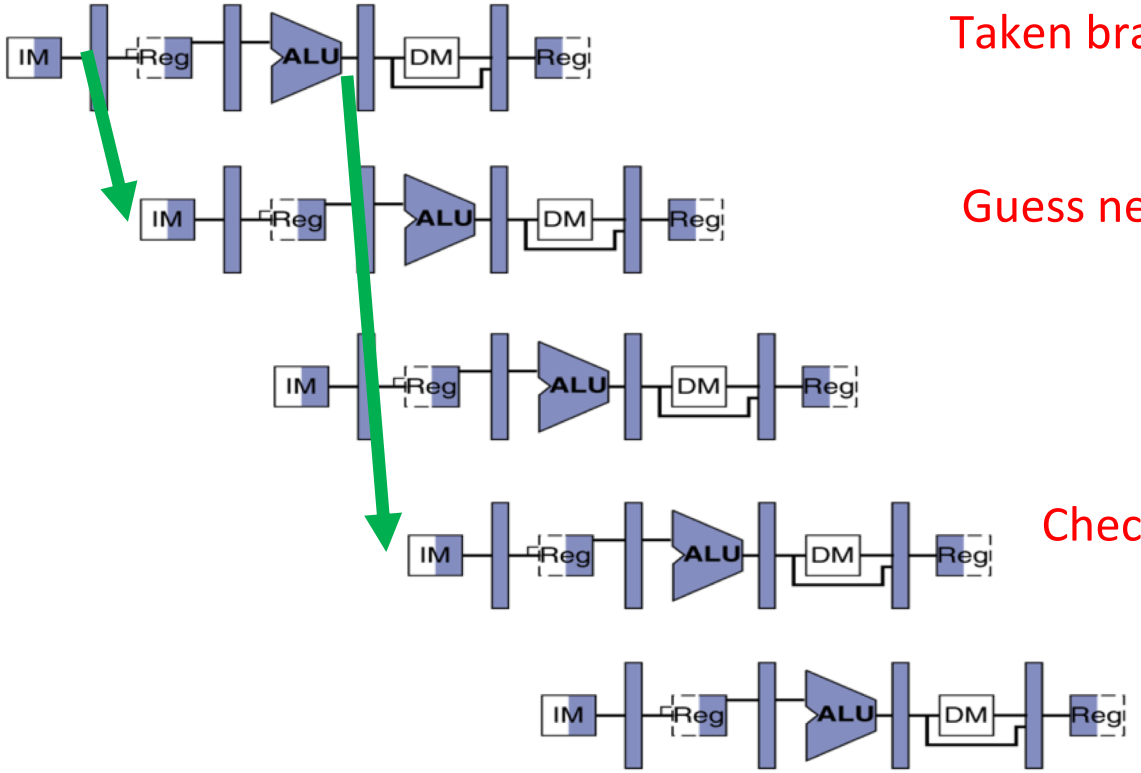
Two instructions are affected by an incorrect branch, just like we'd have to insert two NOP's/stalls in the pipeline to wait on the correct value!

Branch Prediction

beq t0, t1, label

label: …..

…..



Taken branch

Guess next PC!

Check guess correct

In the correct case, we don't have any stalls/NOP's at all!

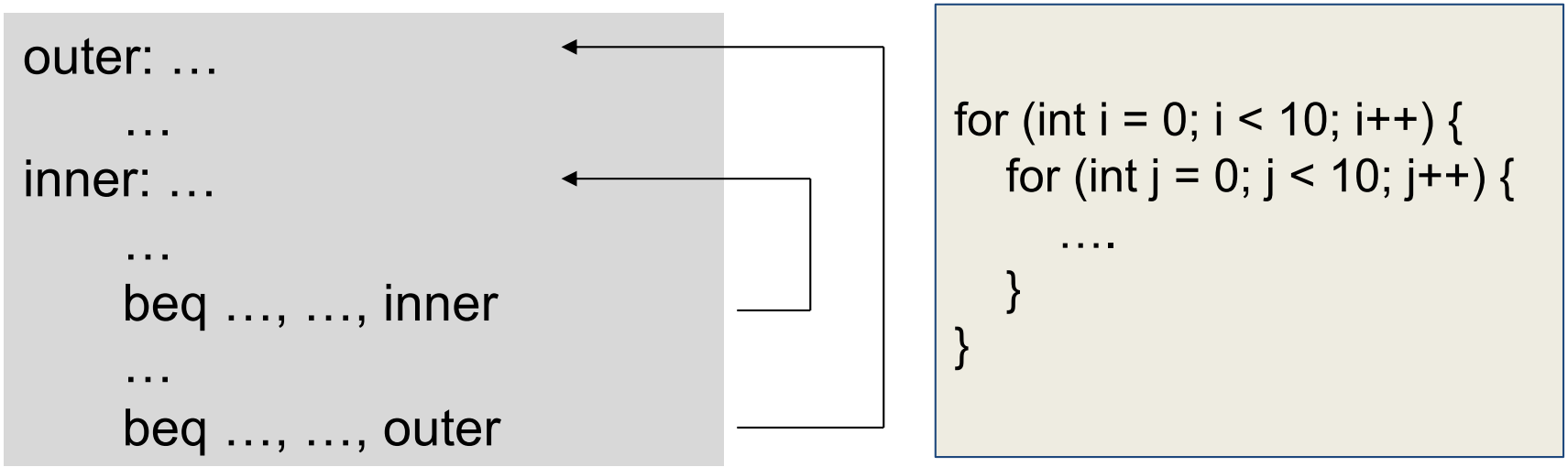
Prediction, if done correctly, is better on average than stalling

Dynamic Branch Prediction

- Branch penalty is more significant in deeper pipelines
- Use *dynamic branch prediction*
  - Have branch prediction mechanism (a.k.a. branch history table) that stores outcomes (taken/not taken) of previous branches
  - To execute a branch
    - Check table and predict the same outcome for next fetch
    - If wrong, flush pipeline and flip prediction

1-Bit Predictor: Shortcoming

- Examine the code below, assuming both loops will be executed multiple times:



- Inner loop branches are predicted wrong twice!
  - Predict as taken on last iteration of inner loop
  - Then predict as not taken on first iteration of inner loop next time around

Branch Predictors

- Branch prediction today is very (very, very…) effective !
  - Multiple models: branch target buffer, branch history table, geometric predictors, etc.
- Contain many bits of state, not easily “saturated”, some consider local vs. global branching
- Interested? Morgan thinks these are really cool, and you can build them in CS152!

Question: For the code sequence below, choose the statement that best describes requirements for correctness

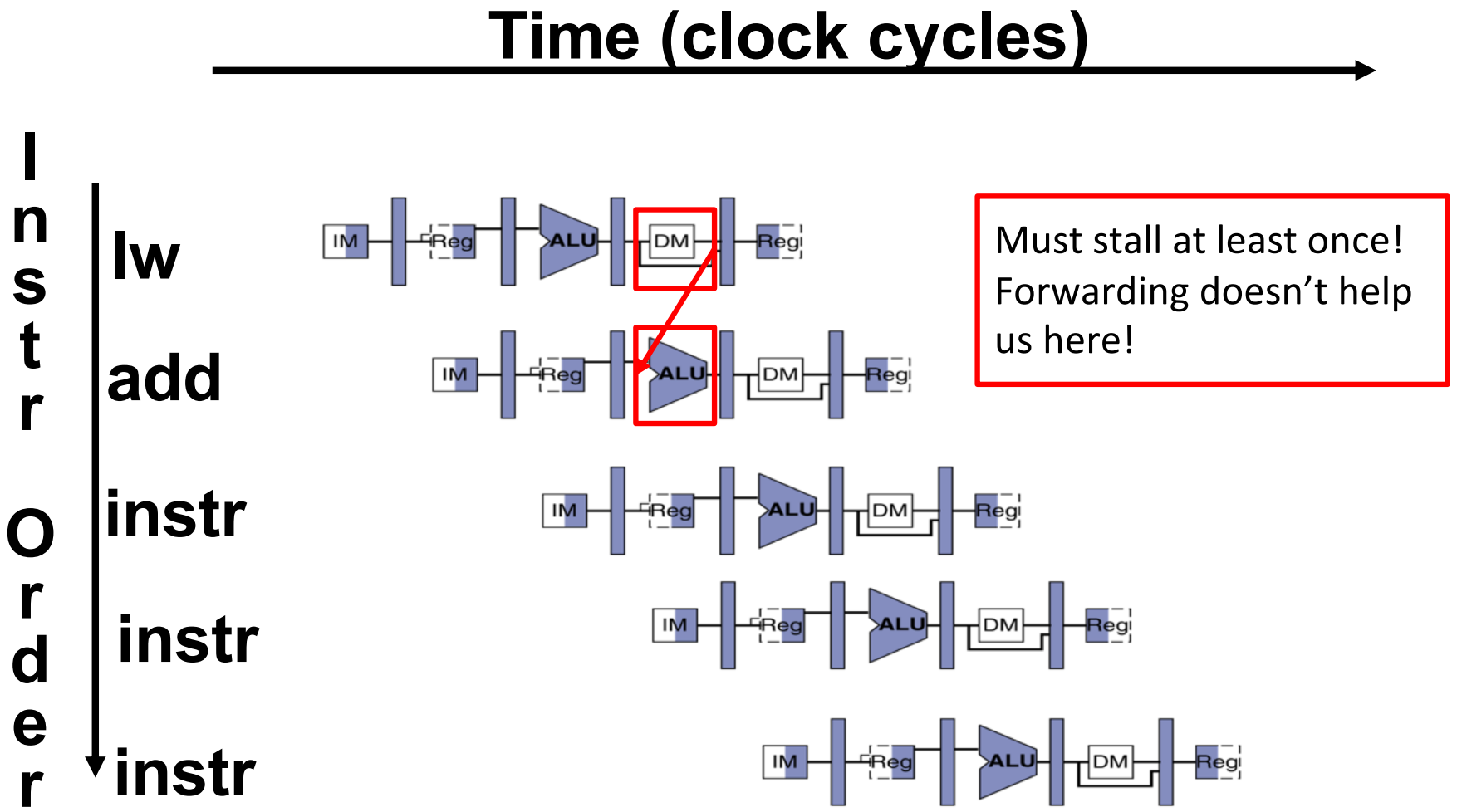
```
lw  t0,0(t0)
add t1,t0,t0
```

A **No stalls as is**

B **No stalls with forwarding**

C **Must stall**

Code Sequence 1



Question: For the code sequence below, choose the statement that best describes requirements for correctness

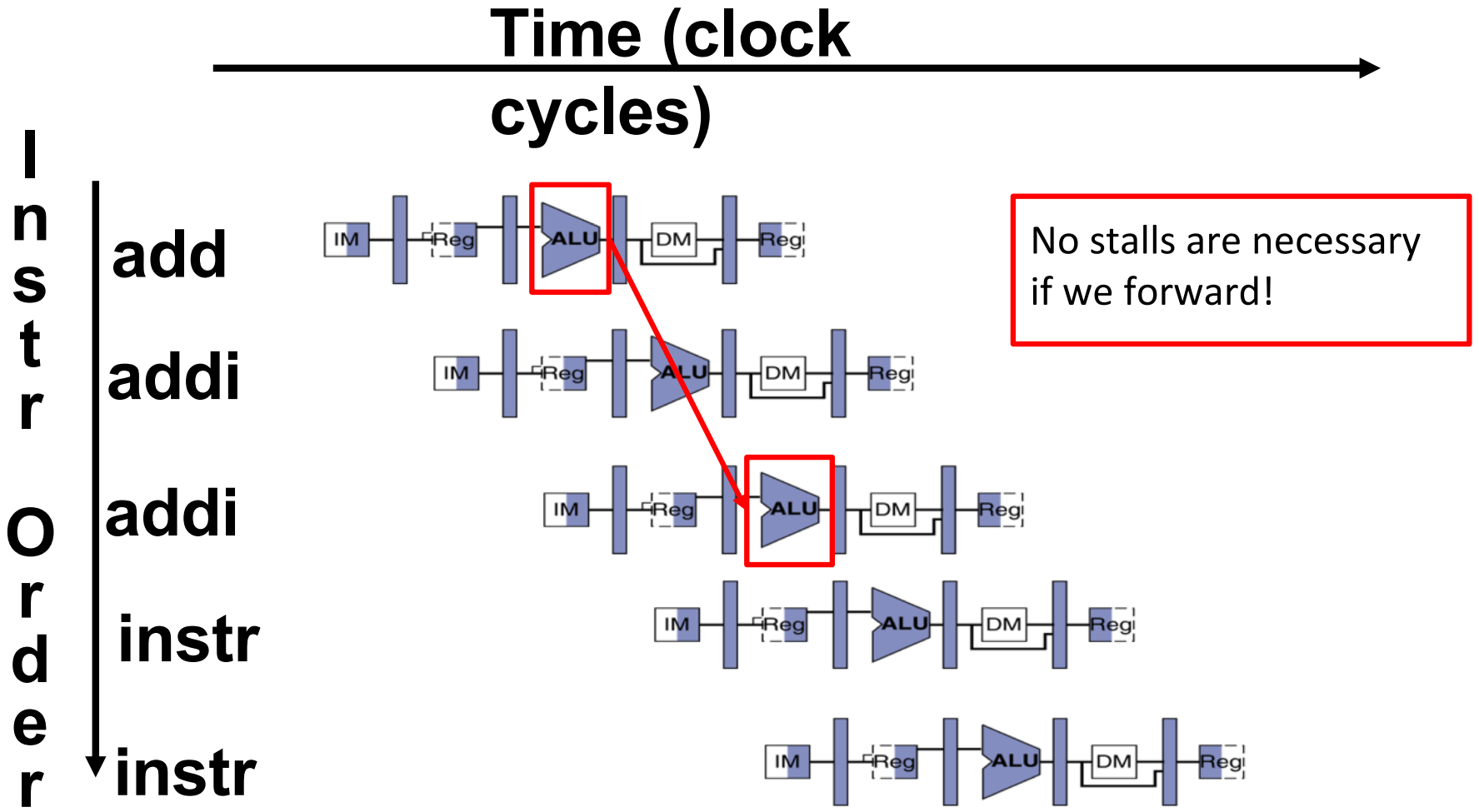
```
add  t1, t0, t0
addi t2, t0, 5
addi t4, t1, 5
```

A **No stalls as is**

B **No stalls with forwarding**

C **Must stall**

Code Sequence 2



Question: For the code sequence below, choose the statement that best describes requirements for correctness

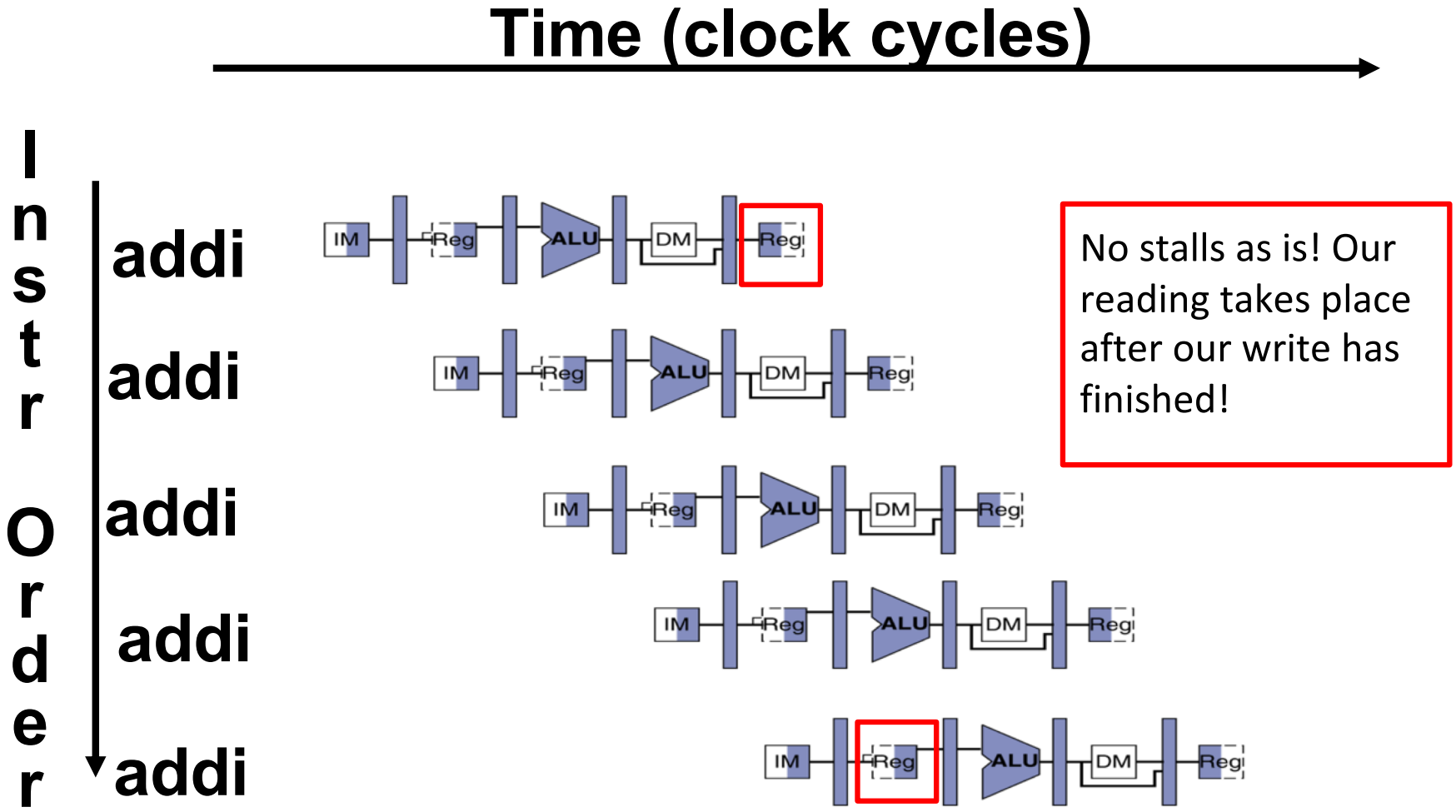
```
3:
 addi t1,t0,1
 addi t2,t0,2
 addi t3,t0,2
 addi t3,t0,4
 addi t5,t1,5
```

A **No stalls as is**

B **No stalls with forwarding**

C **Must stall**

Code Sequence 3



No stalls as is! Our reading takes place after our write has finished!

Agenda

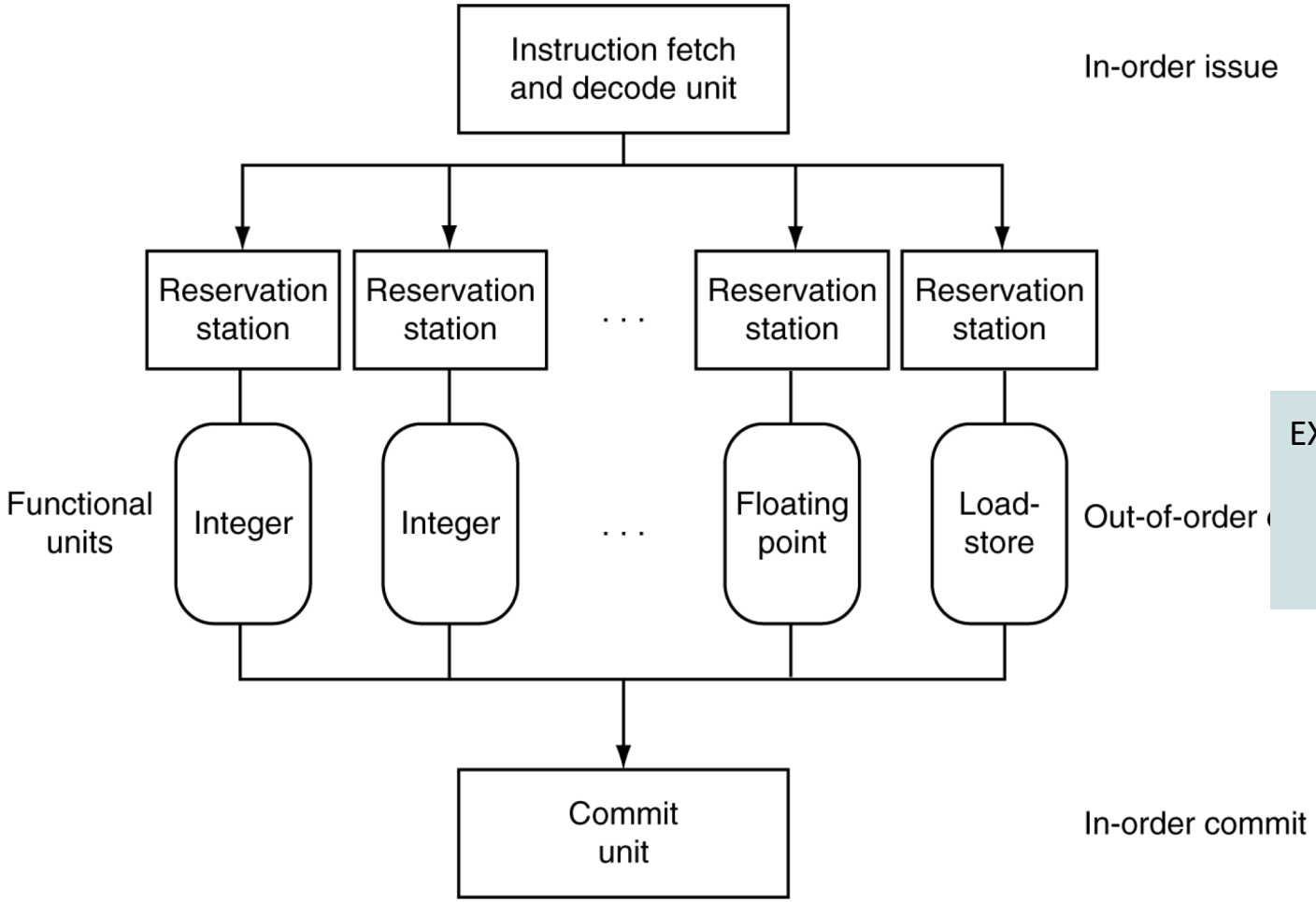
- RISC-V Pipeline
- Hazards
  - Structural
  - Data
    - R-type instructions
    - Load
  - Control
- Superscalar processors

Increasing Processor Performance

- Clock rate
  - Limited by technology and power dissipation
- Pipelining
  - “Overlap” instruction execution
  - Deeper pipeline: 5 => 10 => 15 stages
    - Less work per stage → shorter clock cycle
    - But more potential for hazards (CPI > 1)
- Multi-issue ”super-scalar” processor
  - Multiple execution units (ALUs)
    - Several instructions executed simultaneously
    - CPI < 1 (ideally)

Superscalar Processor

P&H p. 340



DECODE ORDER

1. addi t0 t0 t1
2. lw t2 0(a0)
3. sub t4 a1 a0
4. addi t0 t0 t1

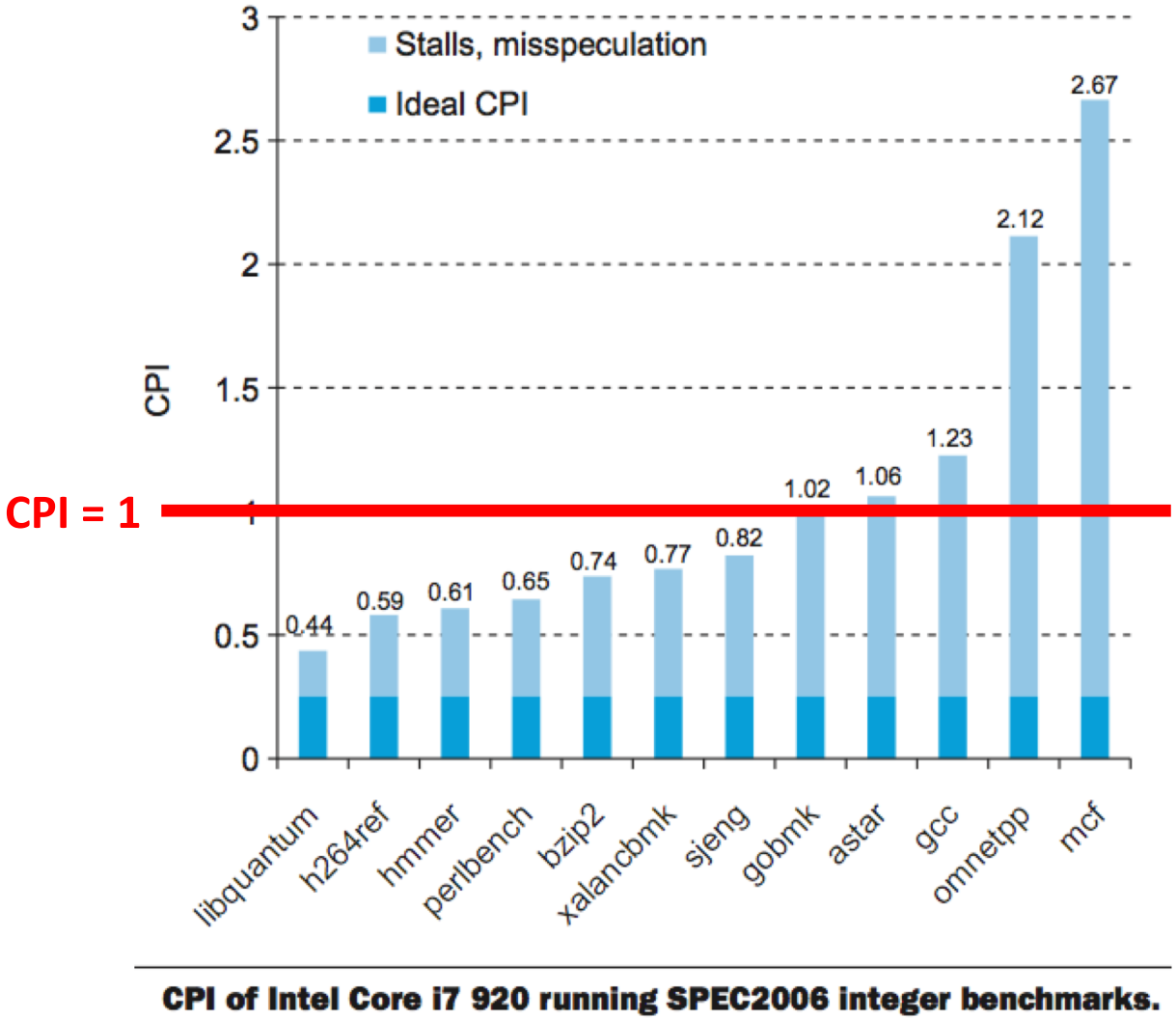
EXECUTION ORDER

1. sub AND addi (1) AND load
2. addi (2)

COMMIT ORDER

1. addi t0 t0 t1
2. lw t2 0(a0)
3. sub t4 a1 a0
4. addi t0 t0 t1

Benchmark: CPI of Intel Core i7



CPI of Intel Core i7 920 running SPEC2006 integer benchmarks.

P&H p. 350

Summary

- Hazards reduce effectiveness of pipelining
  - Cause stalls/bubbles
- Structural Hazards
  - Conflict in use of a datapath component
- Data Hazards
  - Need to wait for result of a previous instruction
- Control Hazards
  - Address of next instruction uncertain/unknown
- Superscalar processors use multiple execution units for additional instruction level parallelism
  - Performance benefit highly code dependent