

CS 61C: Great Ideas in Computer Architecture

Lecture 13: *5-Stage MIPS CPU (Pipelining)*

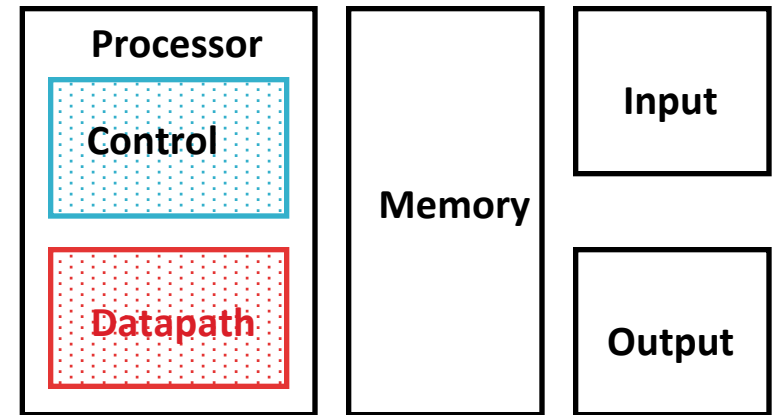
Instructor: Sagar Karandikar
sagark@eecs.berkeley.edu

<http://inst.eecs.berkeley.edu/~cs61c>

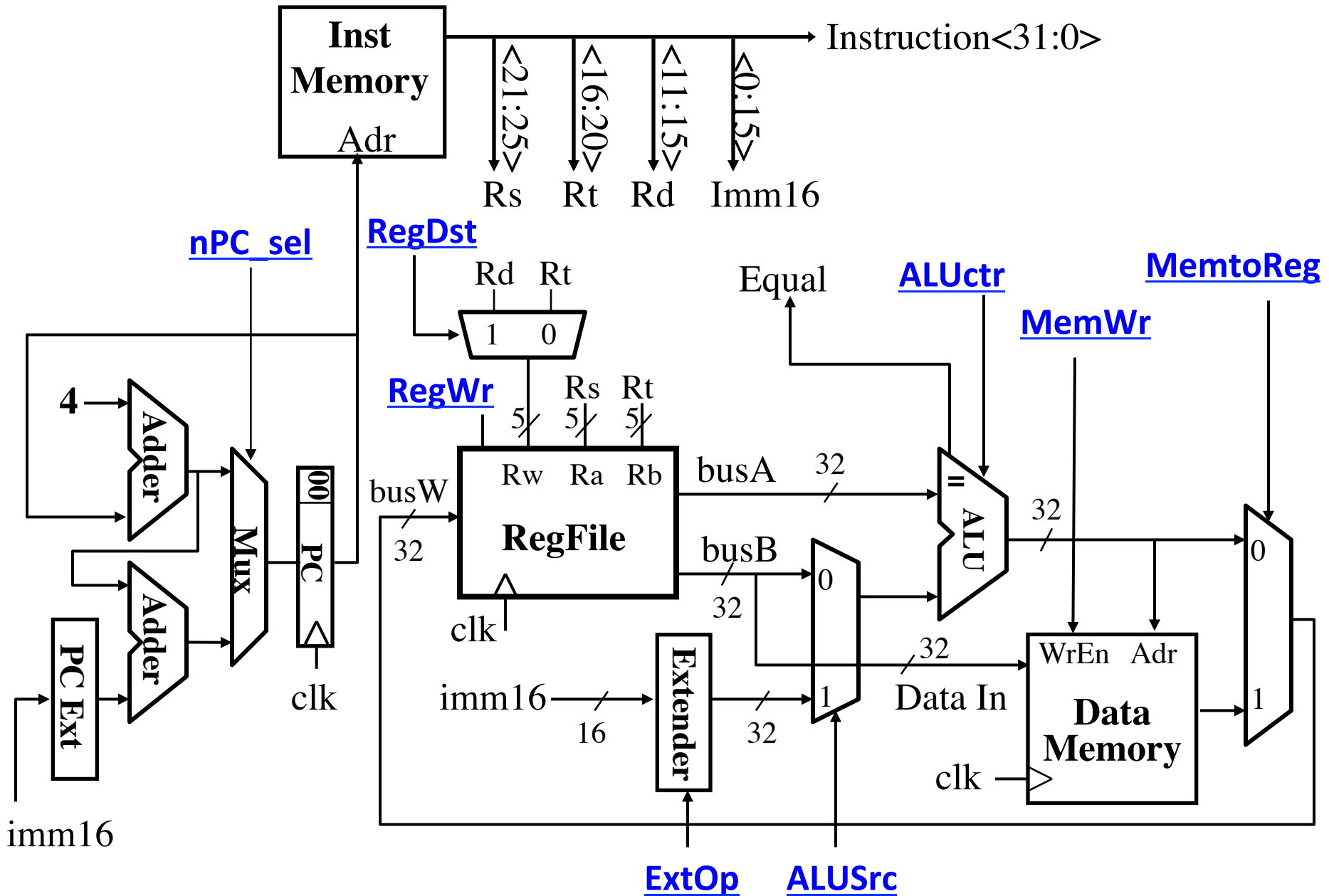
Review: Single-Cycle Processor

- Five steps to design a processor:

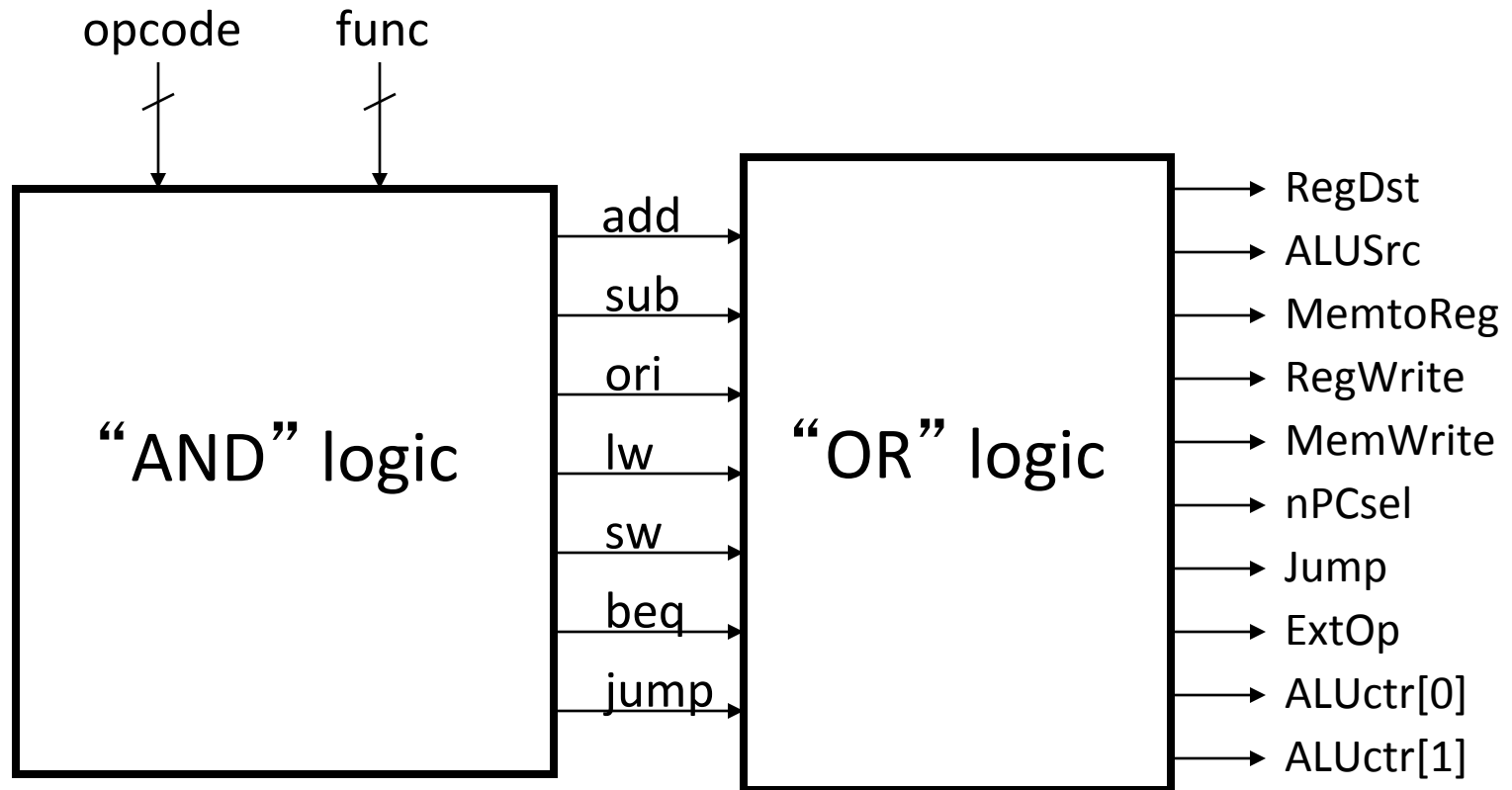
1. Analyze instruction set → datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
5. Assemble the control logic
 - Formulate Logic Equations
 - Design Circuits



Review: A Single-Cycle Datapath



Review: Controller Implementation



Single Cycle Performance

- Assume time for actions are
 - 100ps for register read or write; 200ps for other events
- Clock period is?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What can we do to improve clock rate?
- Will this improve performance as well?
 - Want increased clock rate to mean faster programs

Single Cycle Performance

- Assume time for actions are
 - 100ps for register read or write; 200ps for other events
- Clock period is?

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What can we do to improve clock rate?
- Will this improve performance as well?
 - Want increased clock rate to mean faster programs

New-School Machine Structures (It's a bit more complicated!)

Software

Hardware

Warehouse-Scale
Computer

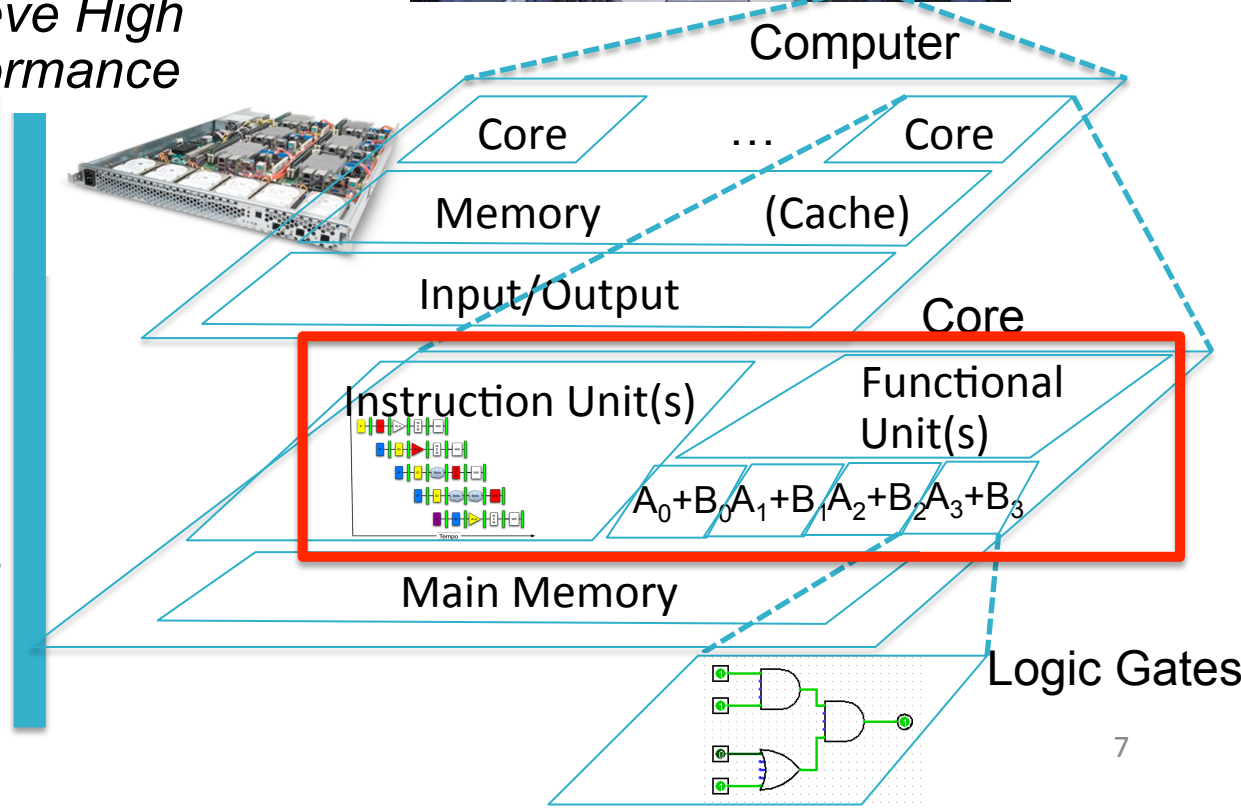


Smart
Phone



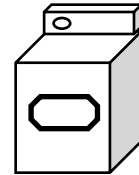
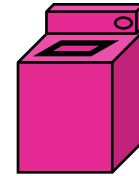
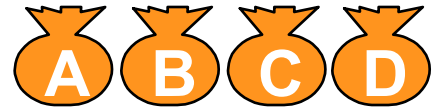
*Harness
Parallelism &
Achieve High
Performance*

- Parallel Requests
Assigned to computer
e.g., Search “Katz”
- 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

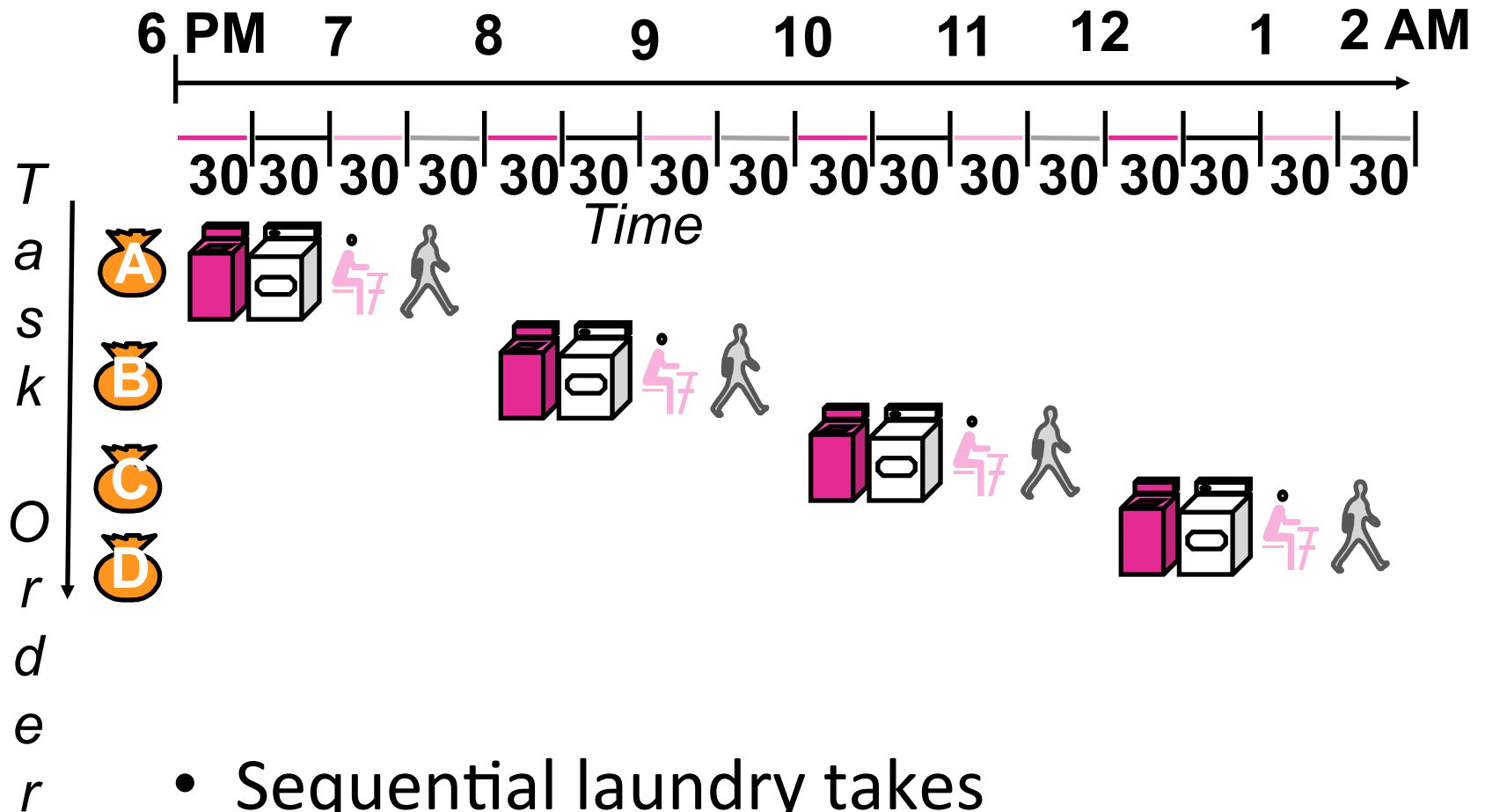


Gotta Do Laundry

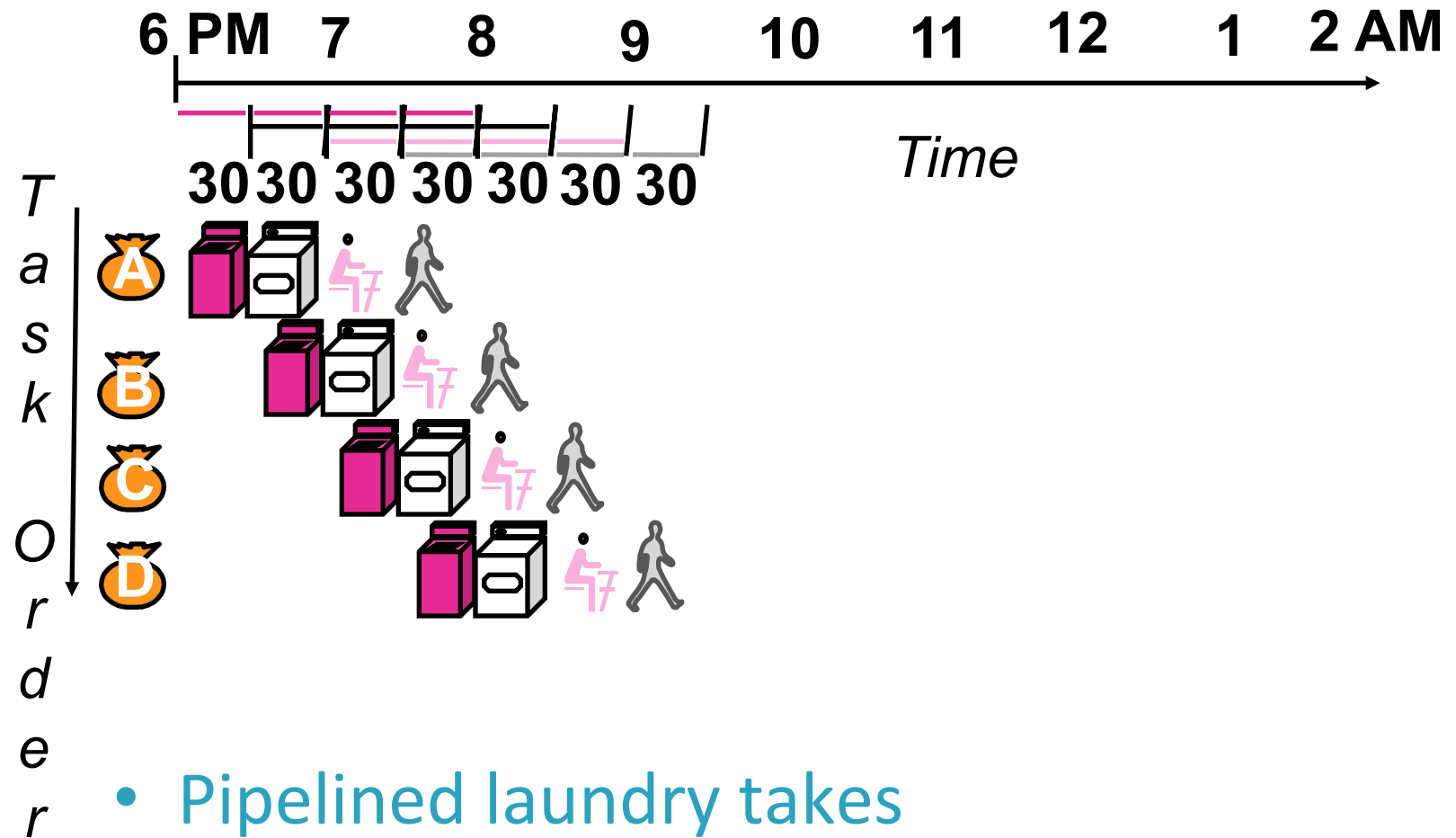
- Ann, Brian, Cathy, Dave
each have one load of clothes to
wash, dry, fold, and put away
 - Washer takes 30 minutes
 - Dryer takes 30 minutes
 - “Folder” takes 30 minutes
 - “Stasher” takes 30 minutes to put
clothes into drawers



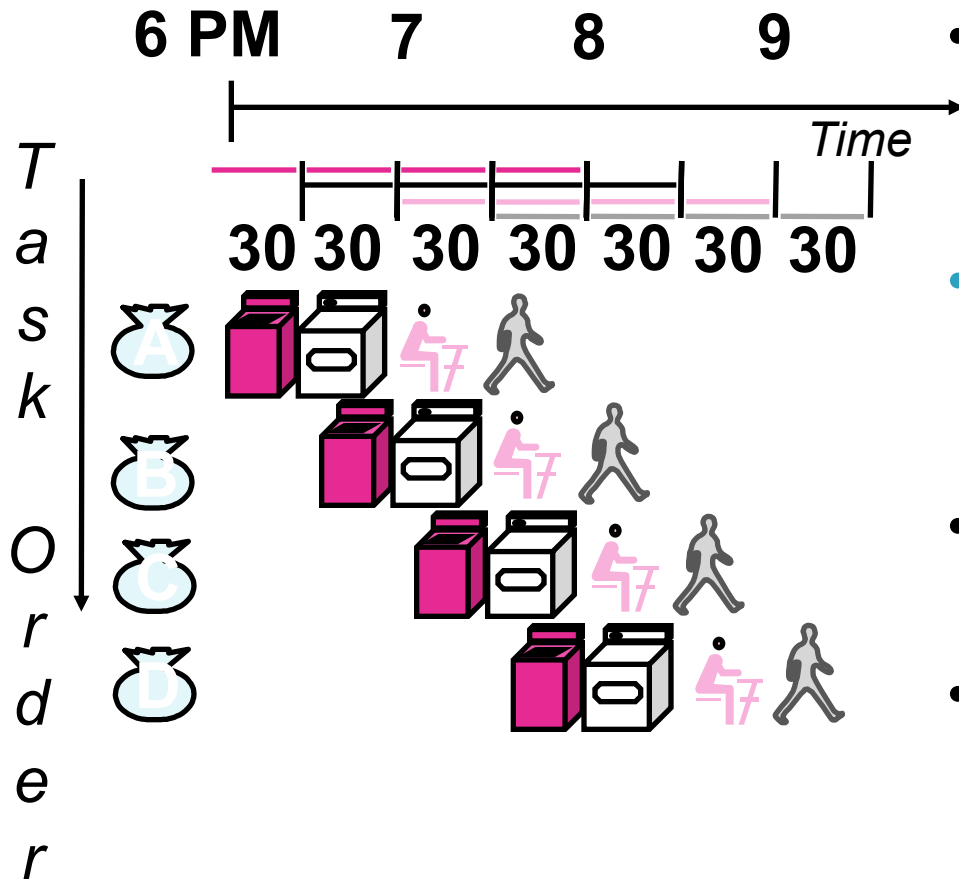
Sequential Laundry



Pipelined Laundry

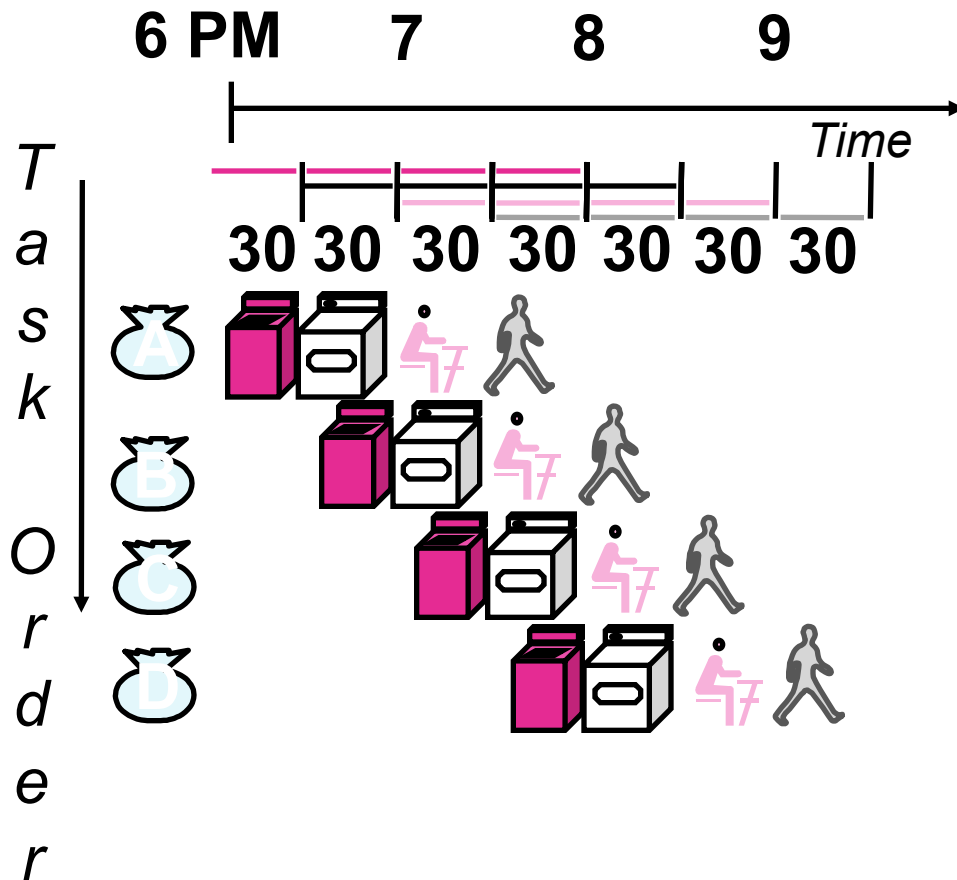


Pipelining Lessons (1/2)



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example

Pipelining Lessons (2/2)

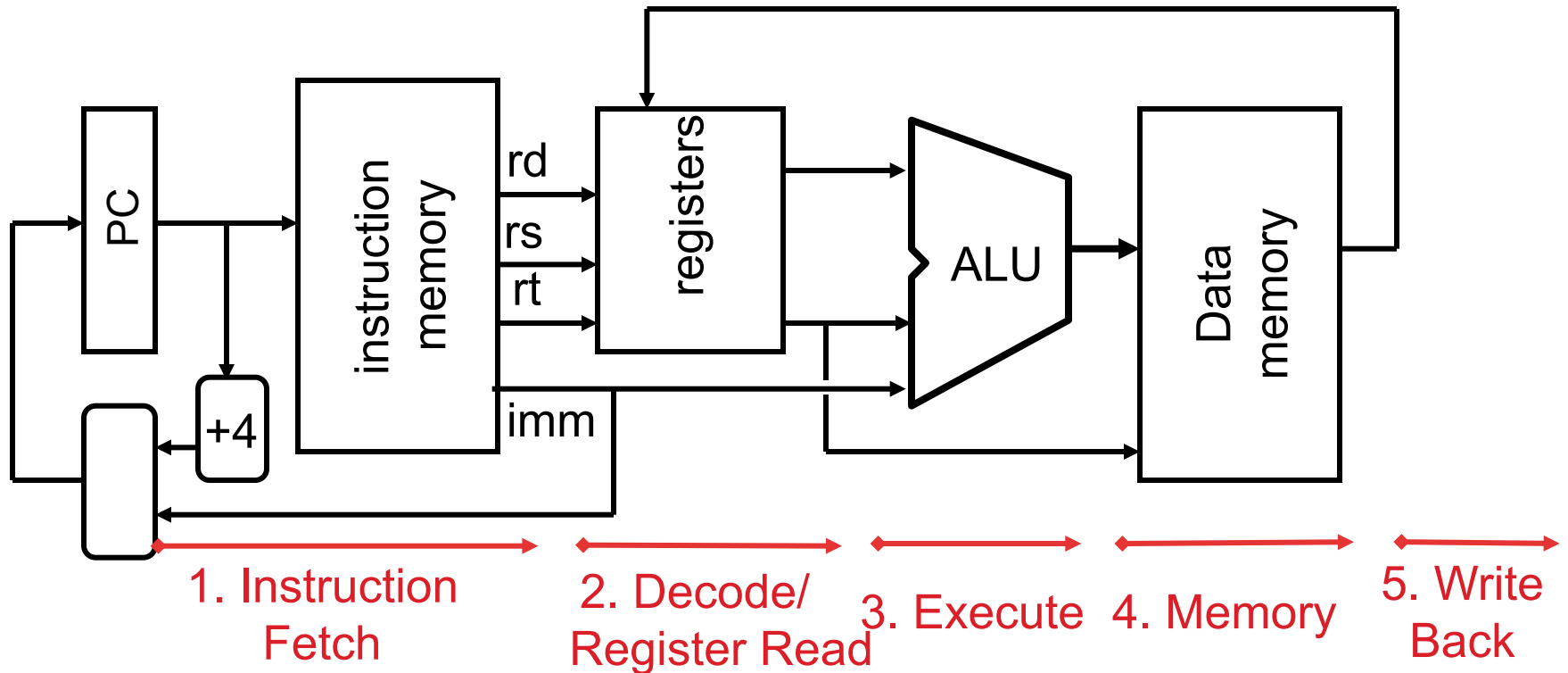


- Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?
- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages reduces speedup

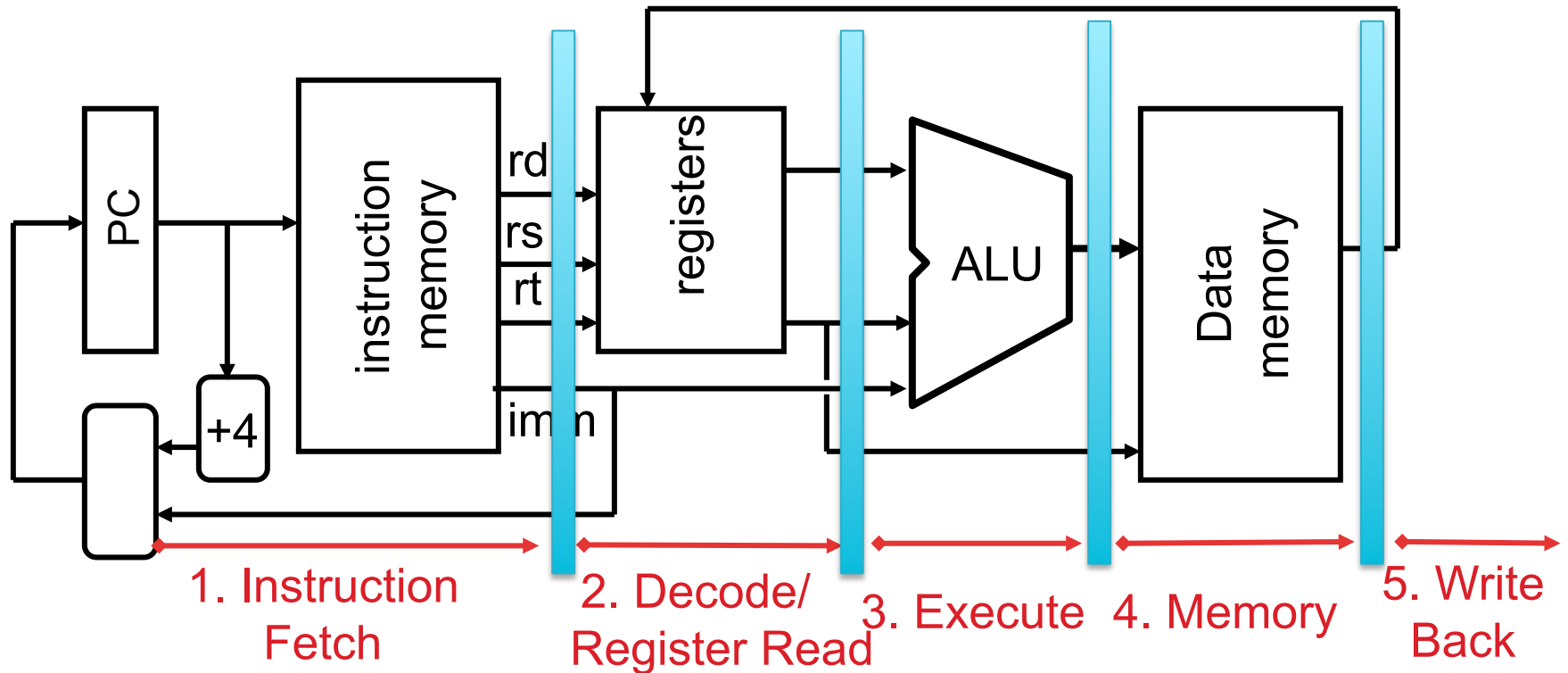
Steps in Executing MIPS

- 1) IFetch: Instruction Fetch, Increment PC
- 2) Decode: Instruction Decode, Read Registers
- 3) Exec:
 - Mem-ref: Calculate Address
 - Arith-log: Perform Operation
- 4) Mem:
 - Load: Read Data from Memory
 - Store: Write Data to Memory
- 5) WB: Write Data Back to Register

Single Cycle Datapath

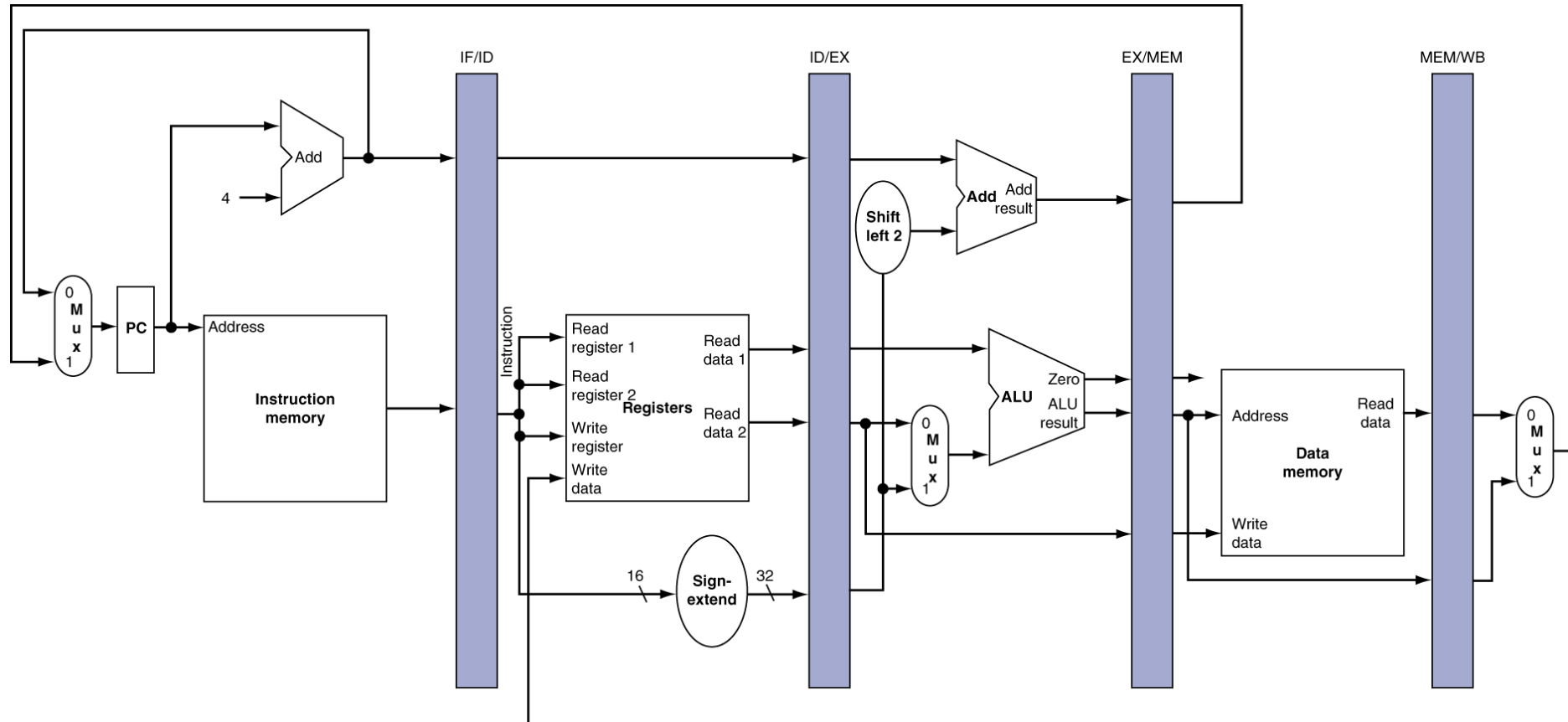


Pipeline registers

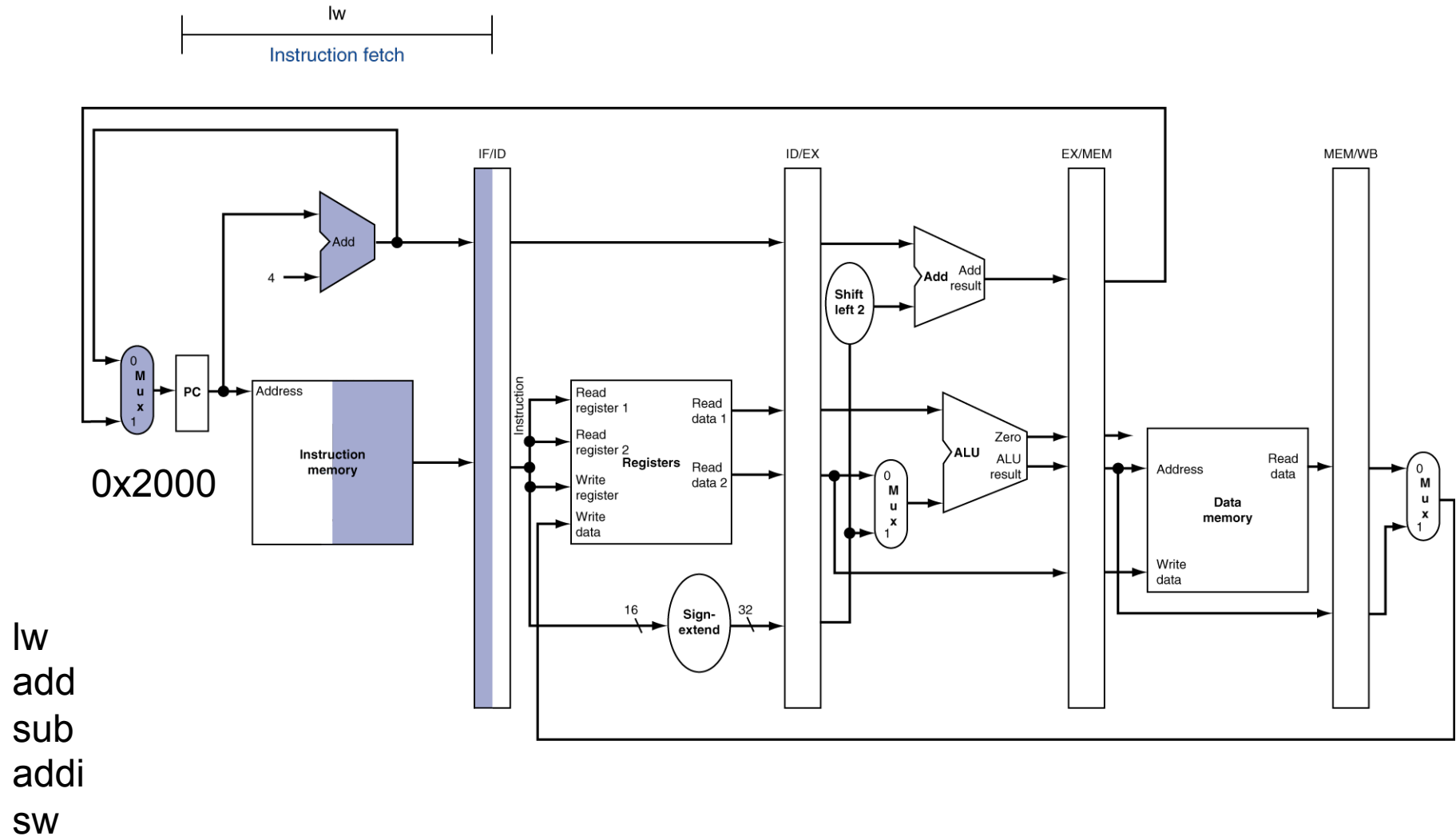


- Need registers between stages
 - To hold information produced in previous cycle

More Detailed Pipeline

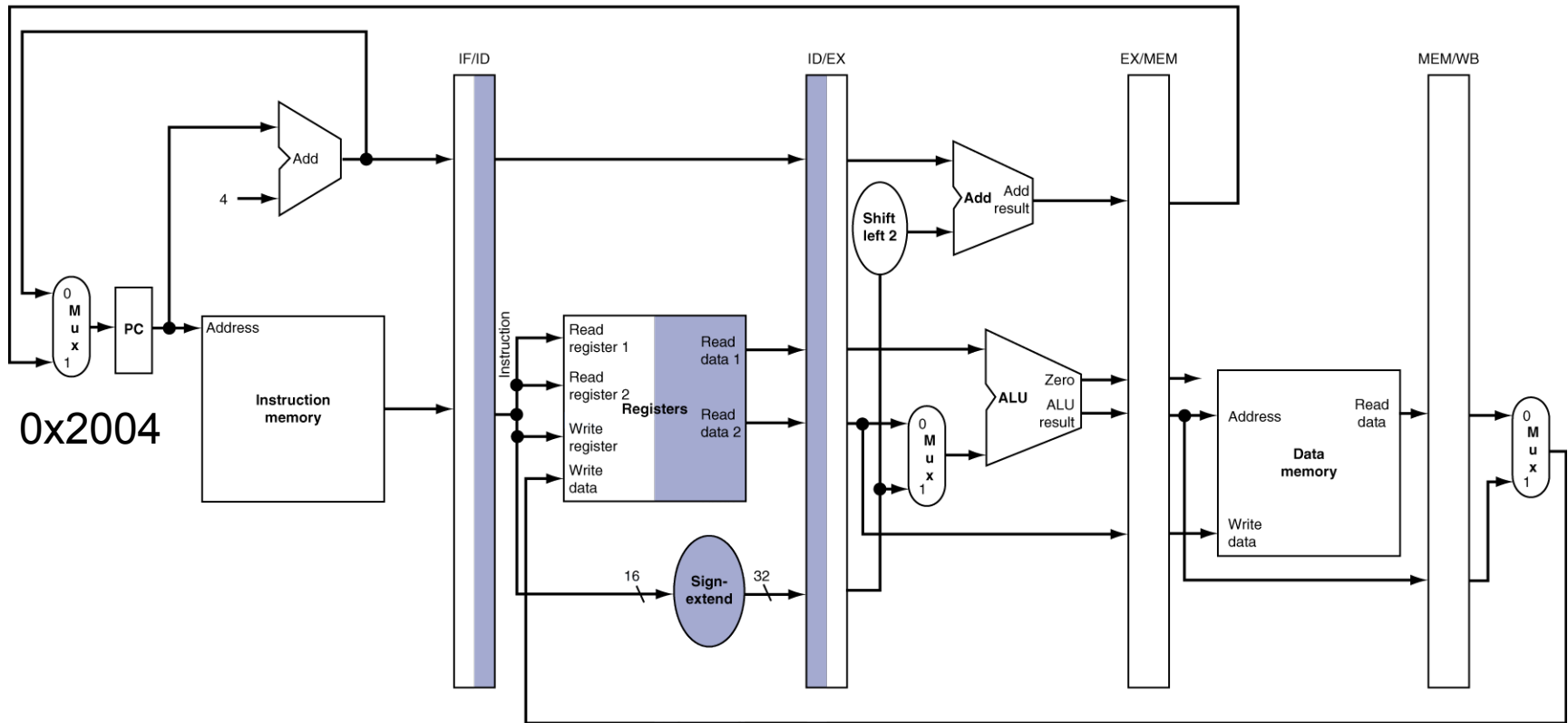


IF for Load, Store, ...



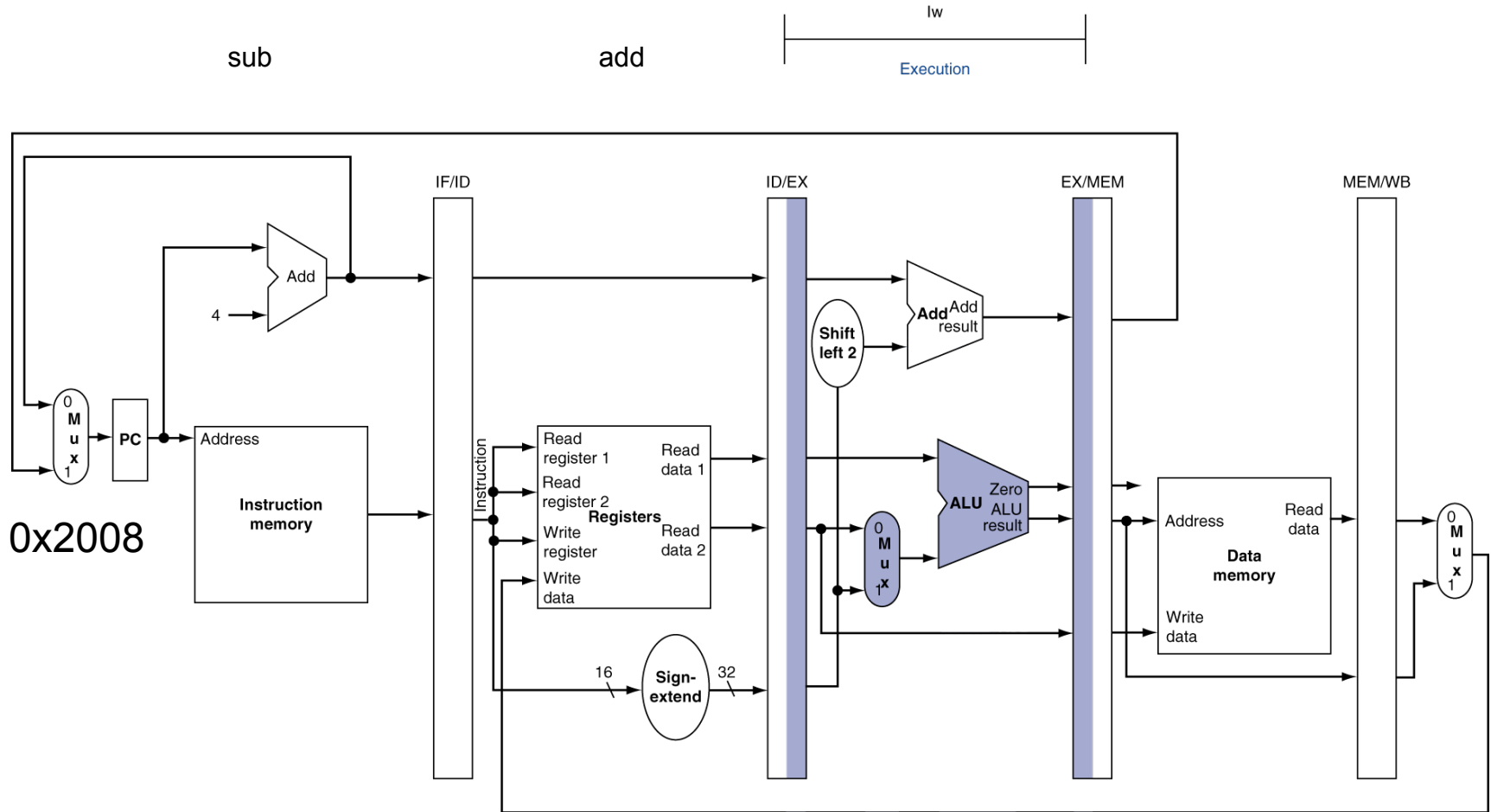
ID for Load, Store, ...

add lw
Instruction decode



lw
add
sub
addi
sw

EX for Load



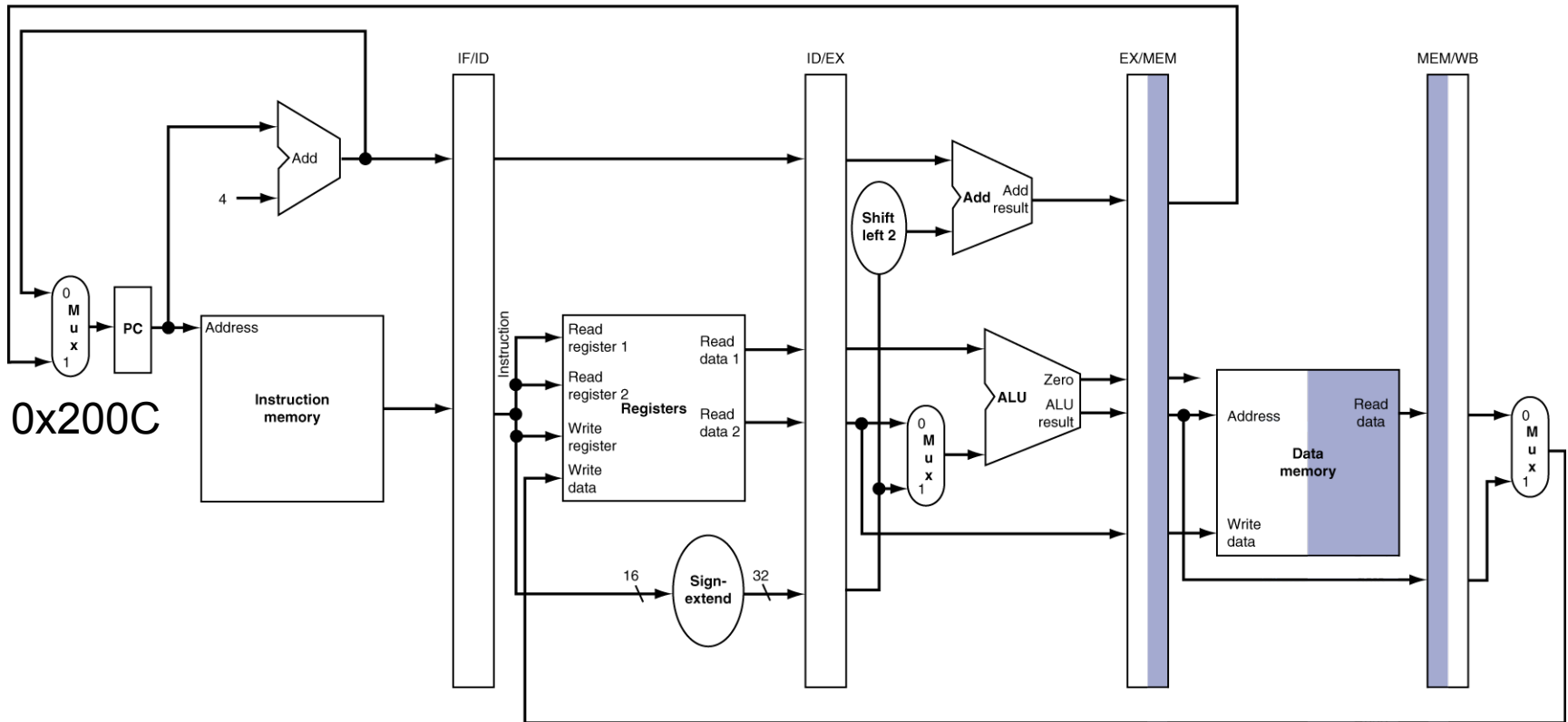
lw
add
sub
addi
sw

MEM for Load

addi

sub

add



lw
add
sub
addi
sw

WB for Load – Oops!

sw

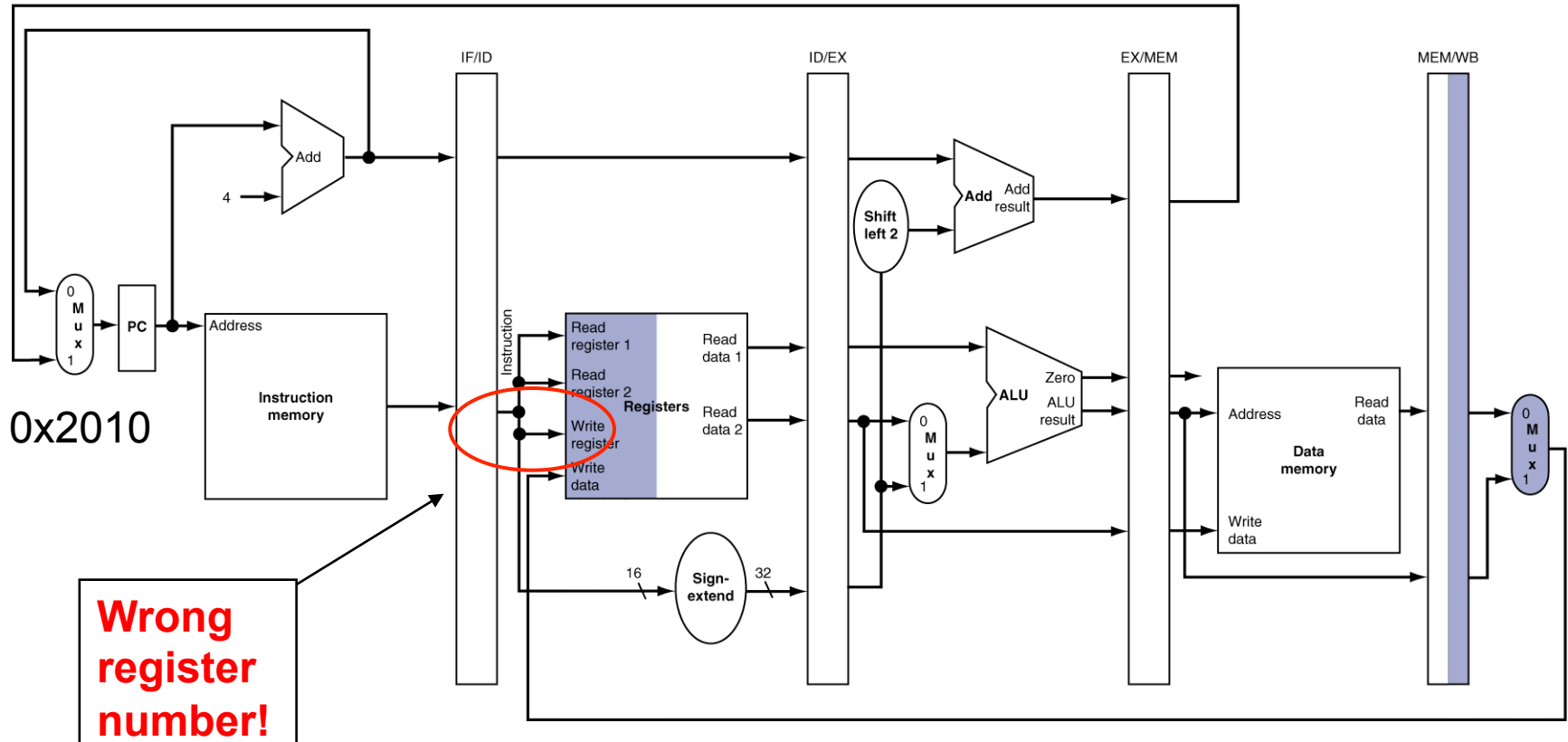
addi

sub

add

lw

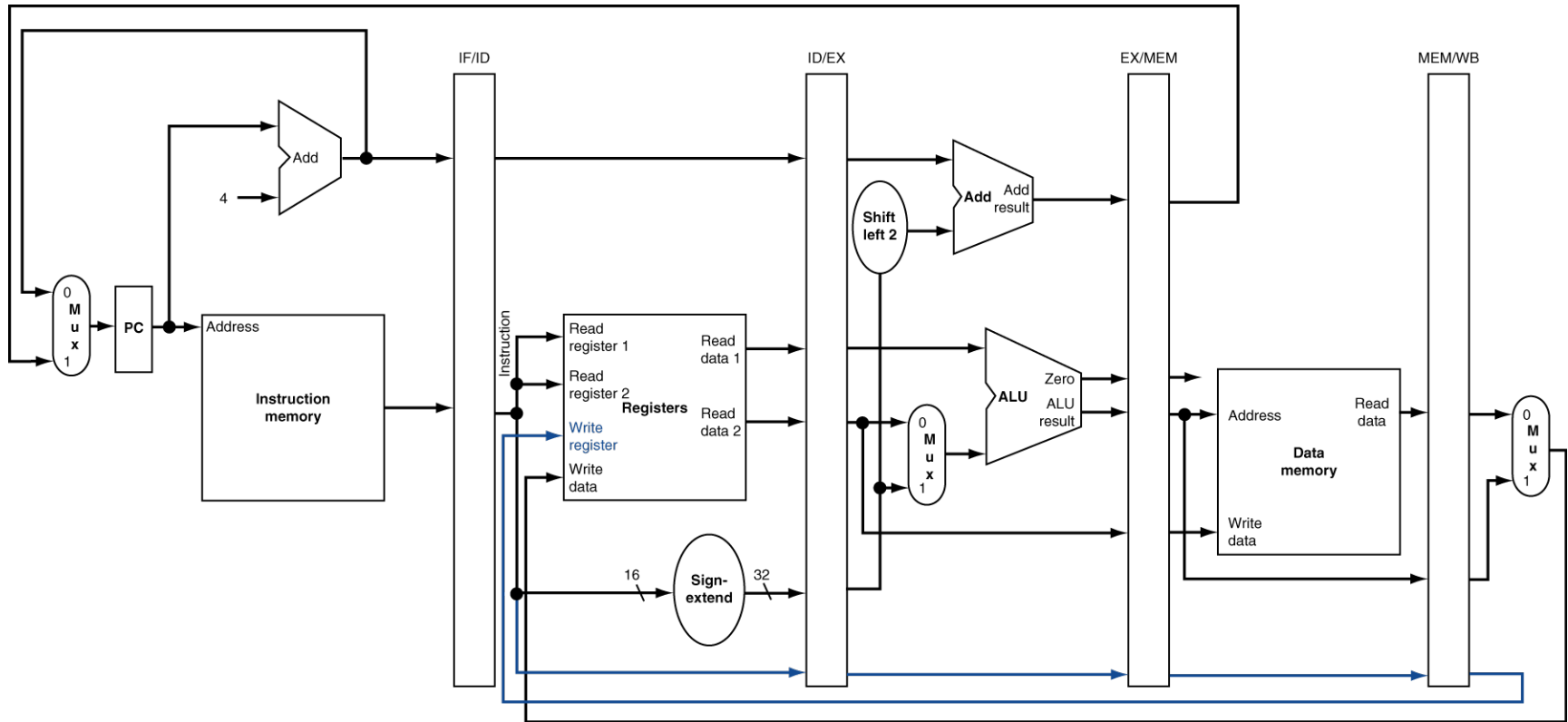
Write back



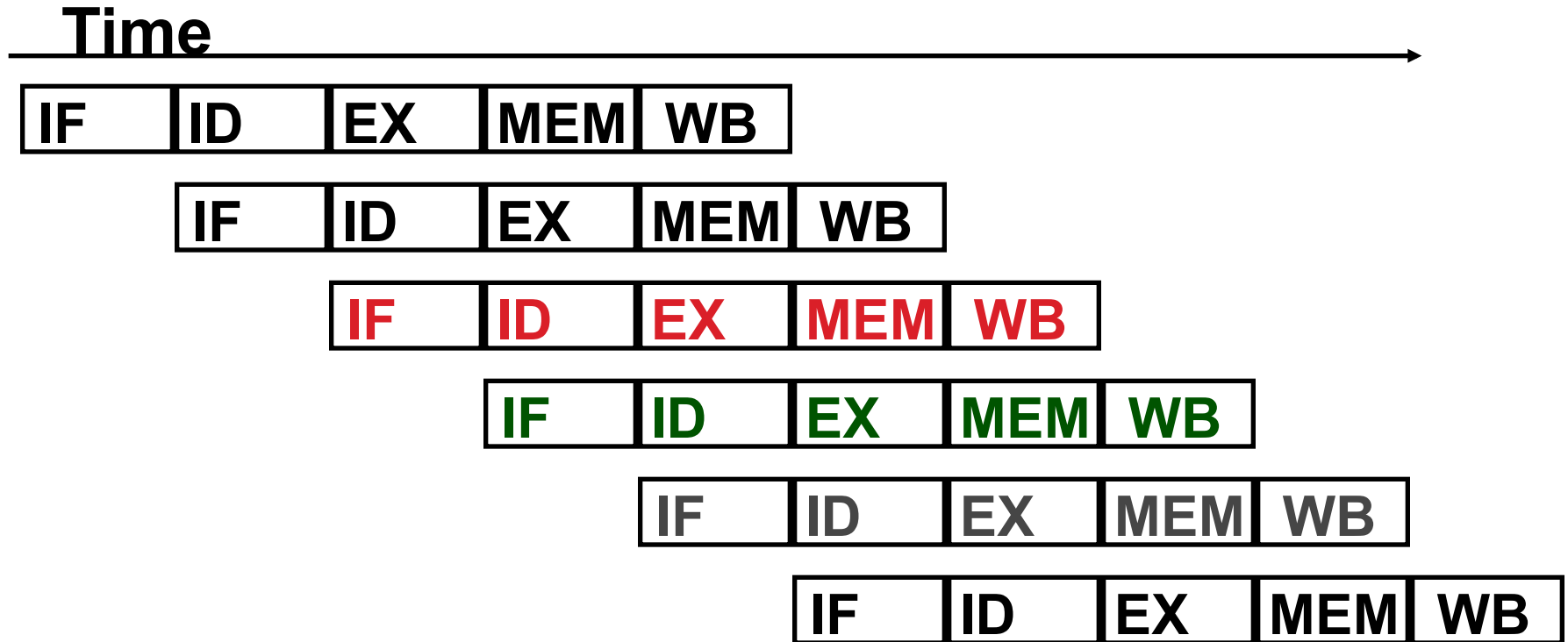
lw
add
sub
addi
sw

**Wrong
register
number!**

Corrected Datapath for Load

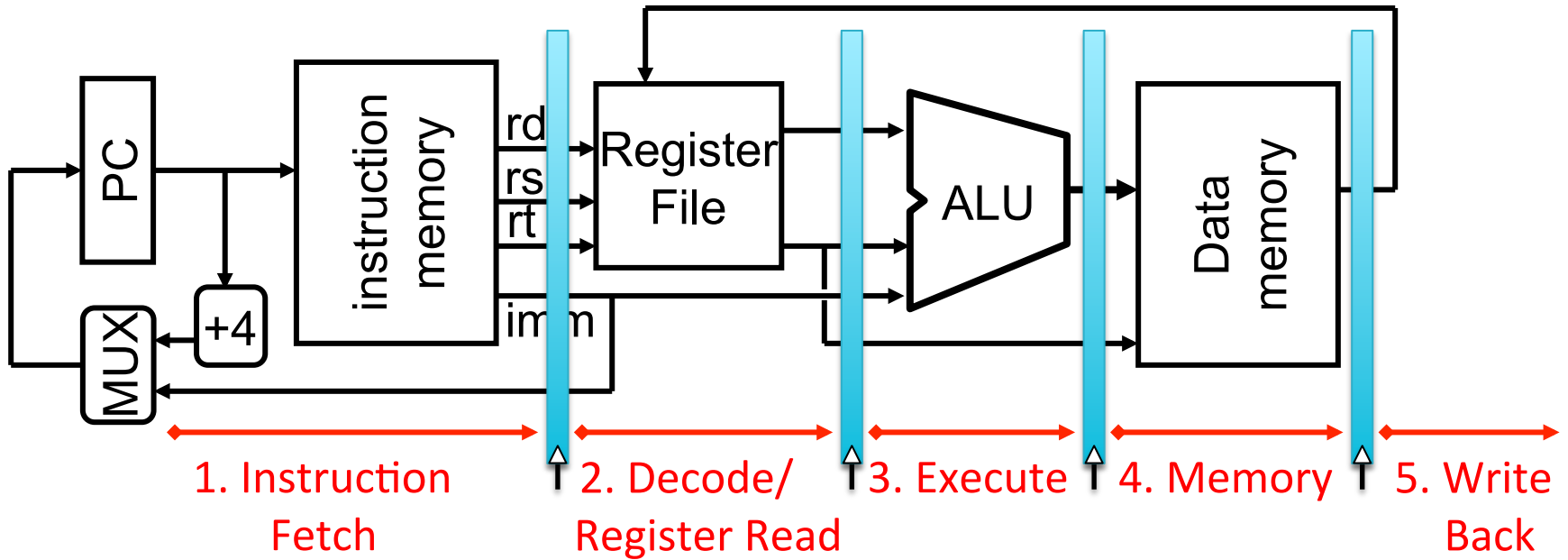


Pipelined Execution Representation

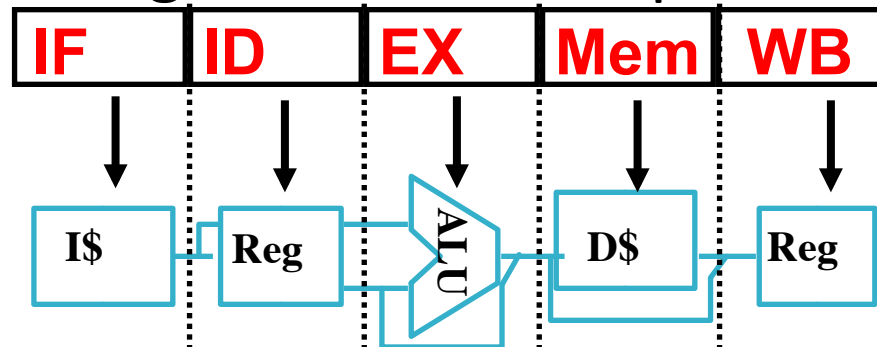


- Every instruction must take same number of steps, so some stages will idle
 - e.g. MEM stage for any arithmetic instruction

Graphical Pipeline Diagrams

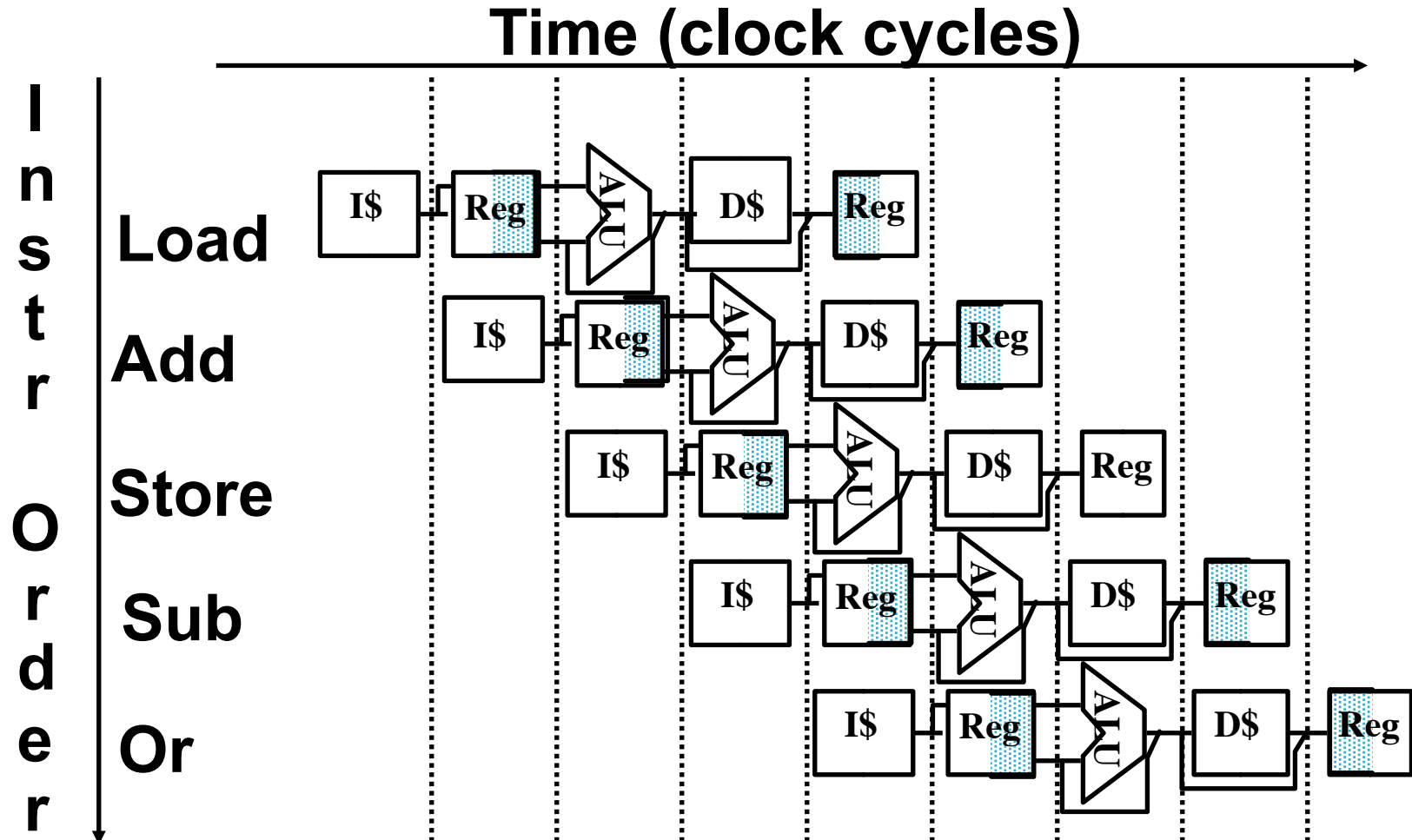


- Use datapath figure below to represent pipeline:



Graphical Pipeline Representation

- RegFile: left half is write, right half is read



Pipelining Performance (1/3)

- Use T_c (“time between completion of instructions”) to measure speedup
 - $T_{c,\text{pipelined}} \geq \frac{T_{c,\text{single-cycle}}}{\text{Number of stages}}$
 - Equality only achieved if stages are *balanced* (i.e. take the same amount of time)
- If not balanced, speedup is reduced
- Speedup due to increased *throughput*
 - *Latency* for each instruction does not decrease

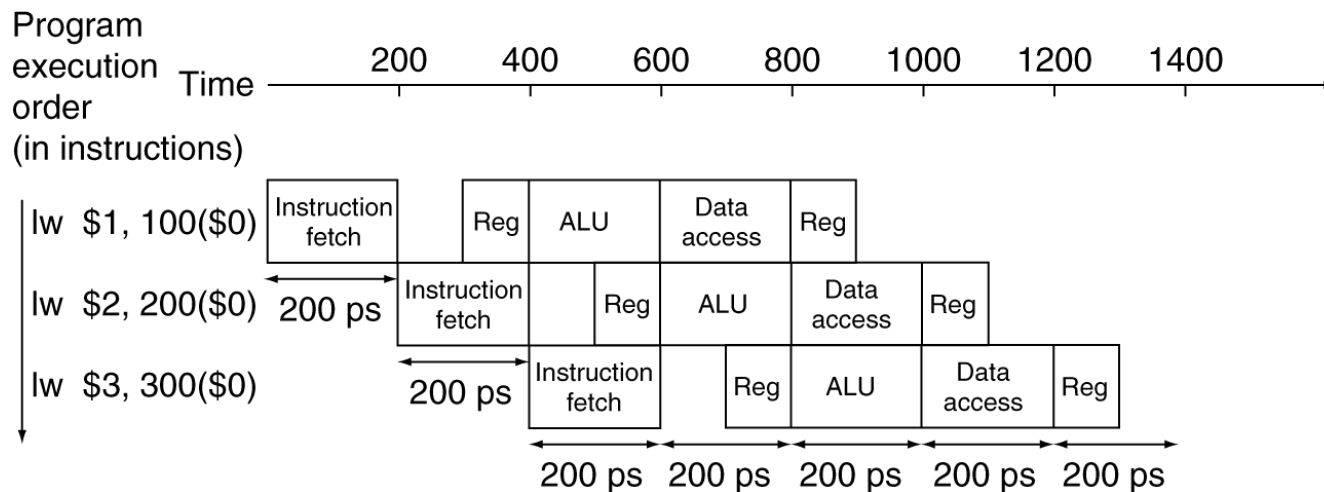
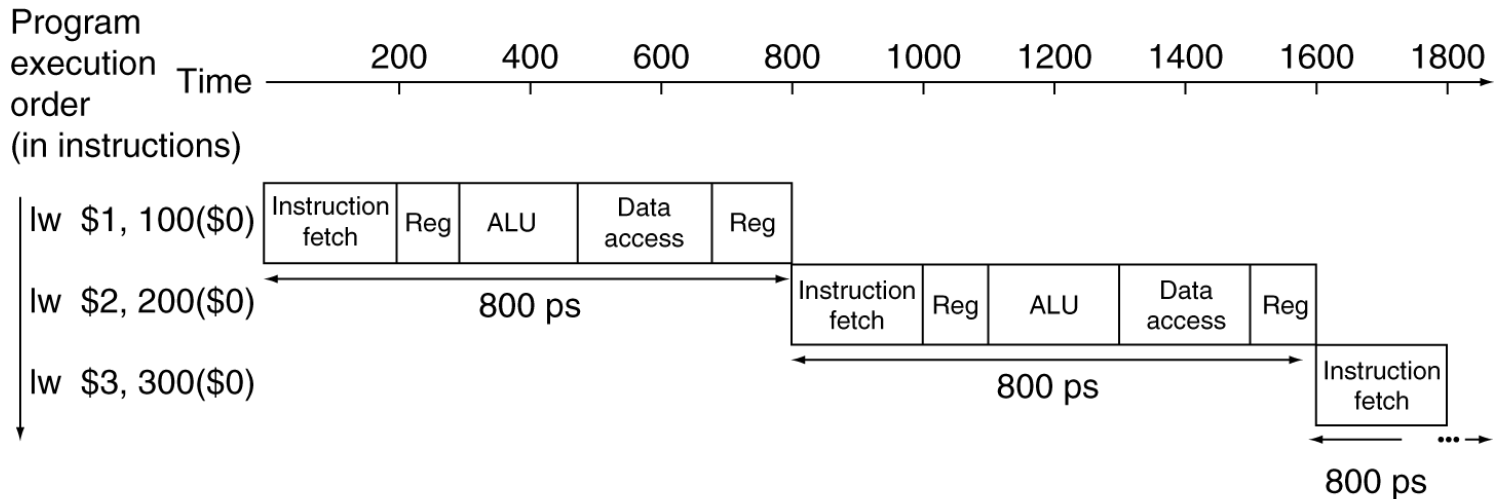
Pipelining Performance (2/3)

- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

- What is pipelined clock rate?
 - Compare pipelined datapath with single-cycle datapath

Pipelining Performance (3/3)



Clicker/Peer Instruction

Which statement is false?

- A: Pipelining increases instruction throughput
- B: Pipelining increases instruction latency
- C: Pipelining increases clock frequency
- D: Pipelining decreases number of components
- E: True

Administrivia

- HW3 Out
- Proj2-2 Out
- Midterm regrades due tomorrow @ 23:59:59

Break

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 read/write

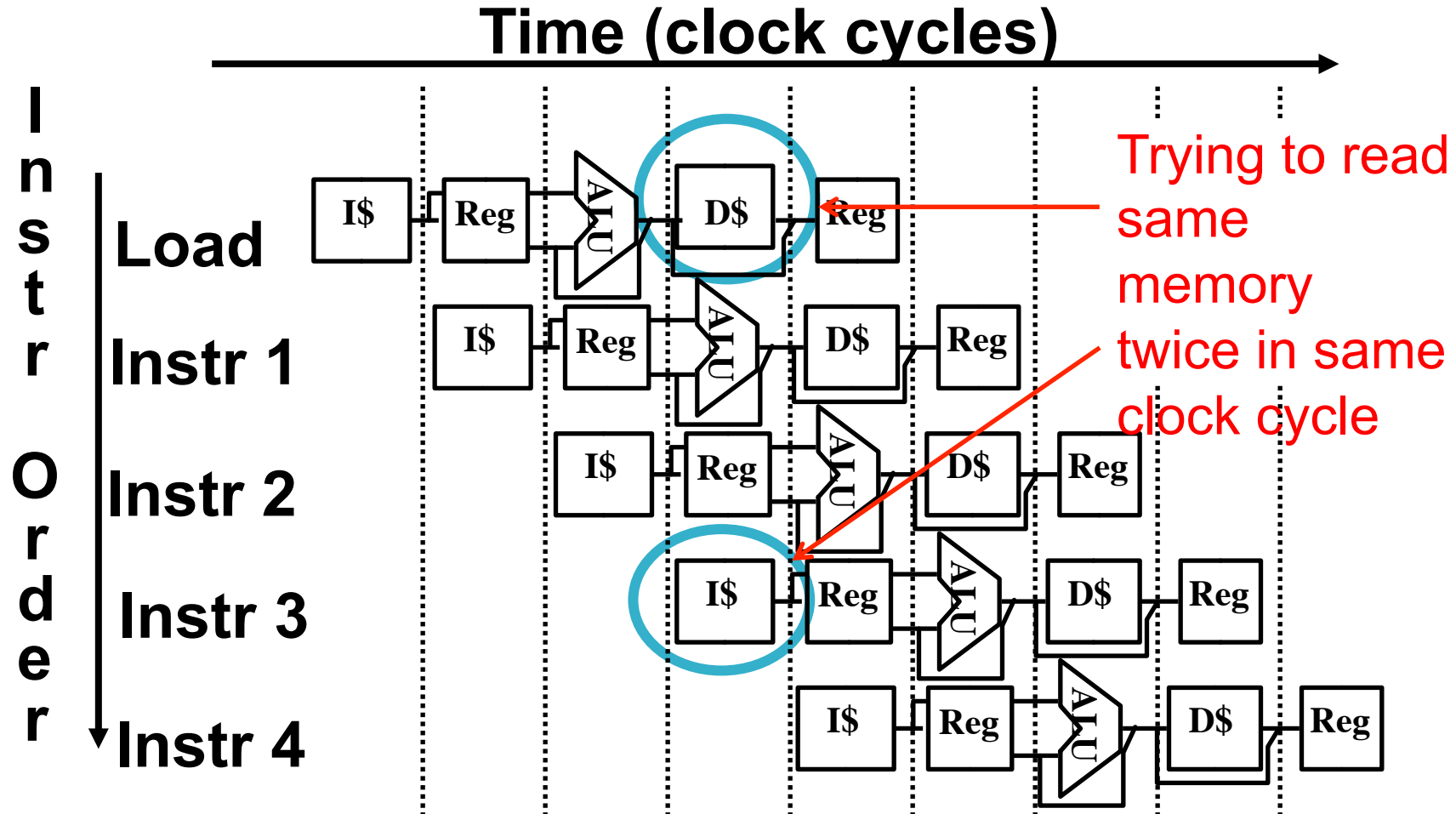
3) *Control hazard*

- Flow of execution depends on previous instruction

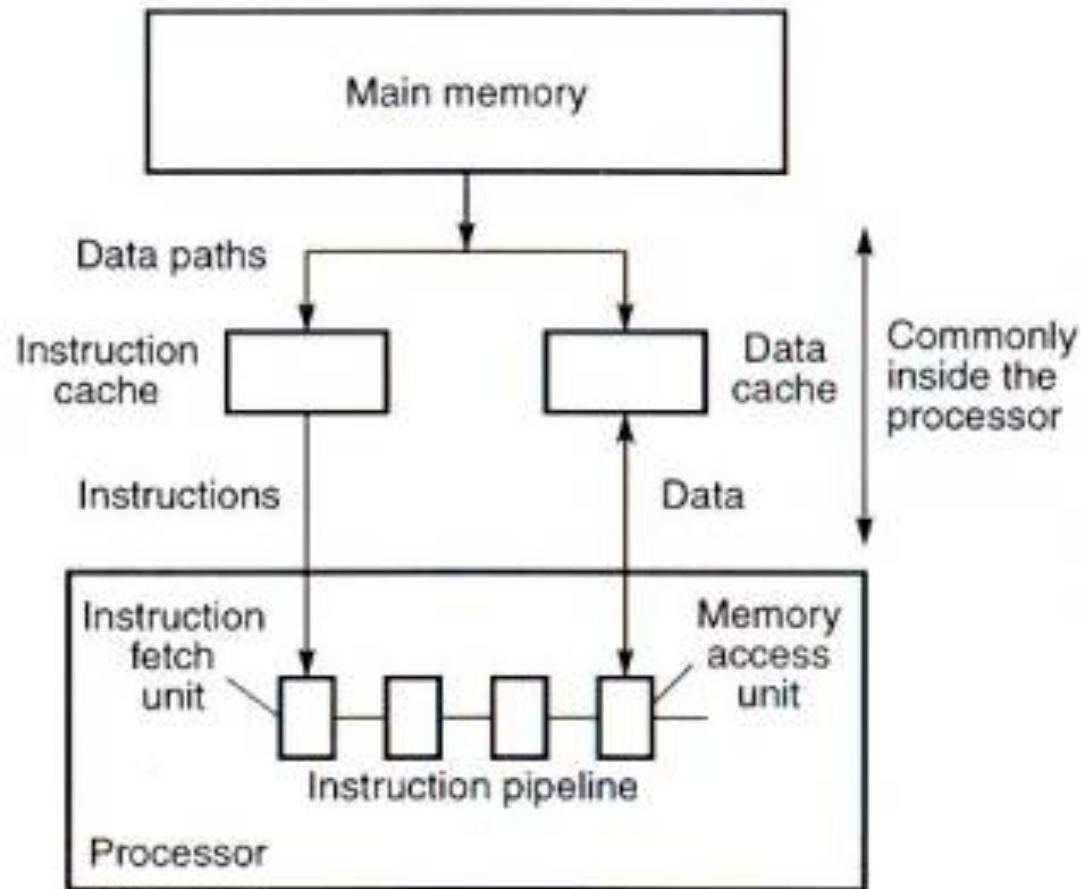
1. Structural Hazards

- Conflict for use of a resource
- MIPS pipeline with a single memory?
 - Load/Store requires memory access for data
 - Instruction fetch would have to *stall* for that cycle
 - Causes a pipeline “*bubble*”
- Hence, pipelined datapaths require separate instruction/data memories
 - Separate L1 I\$ and L1 D\$ take care of this

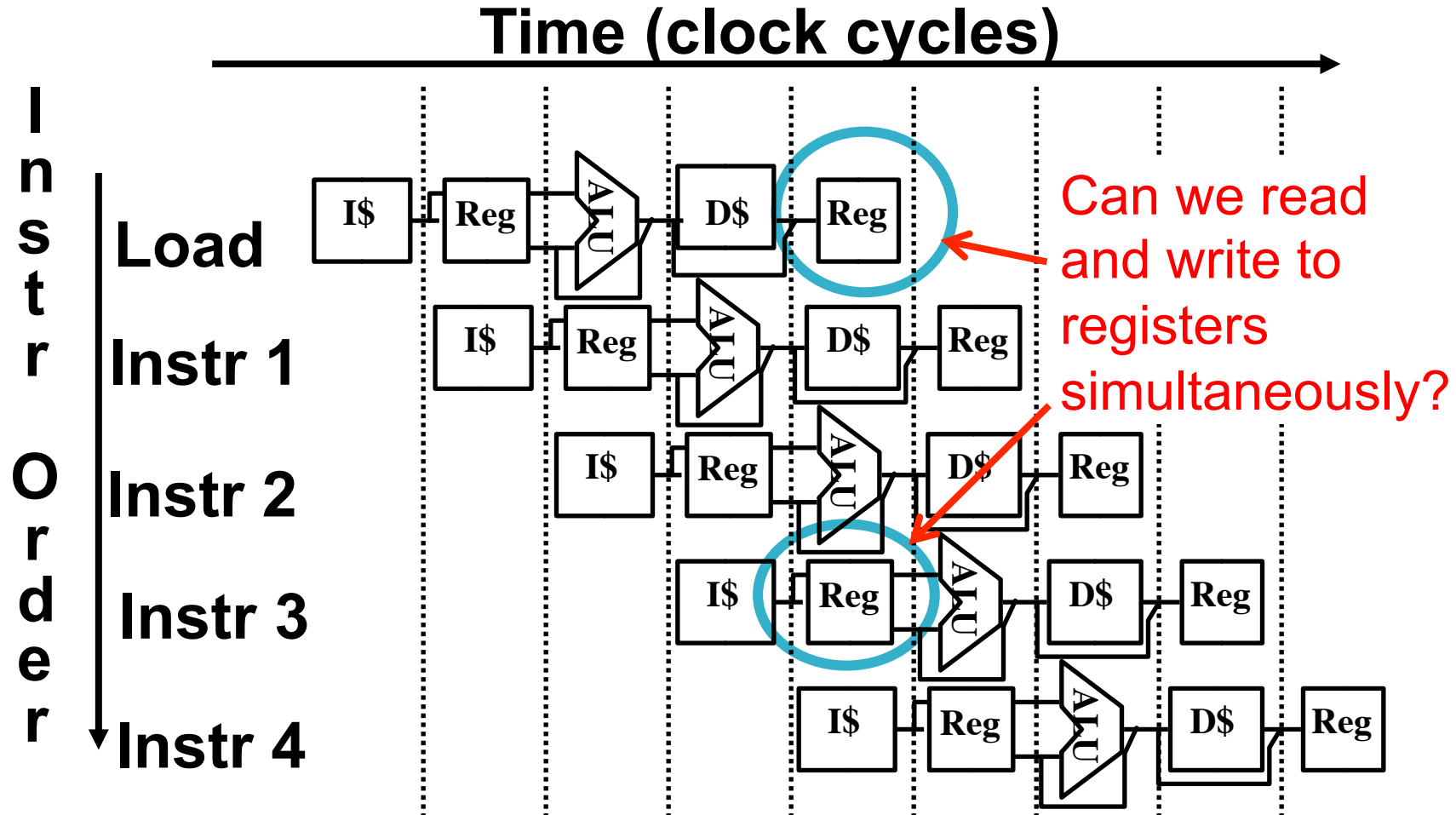
Structural Hazard #1: Single Memory



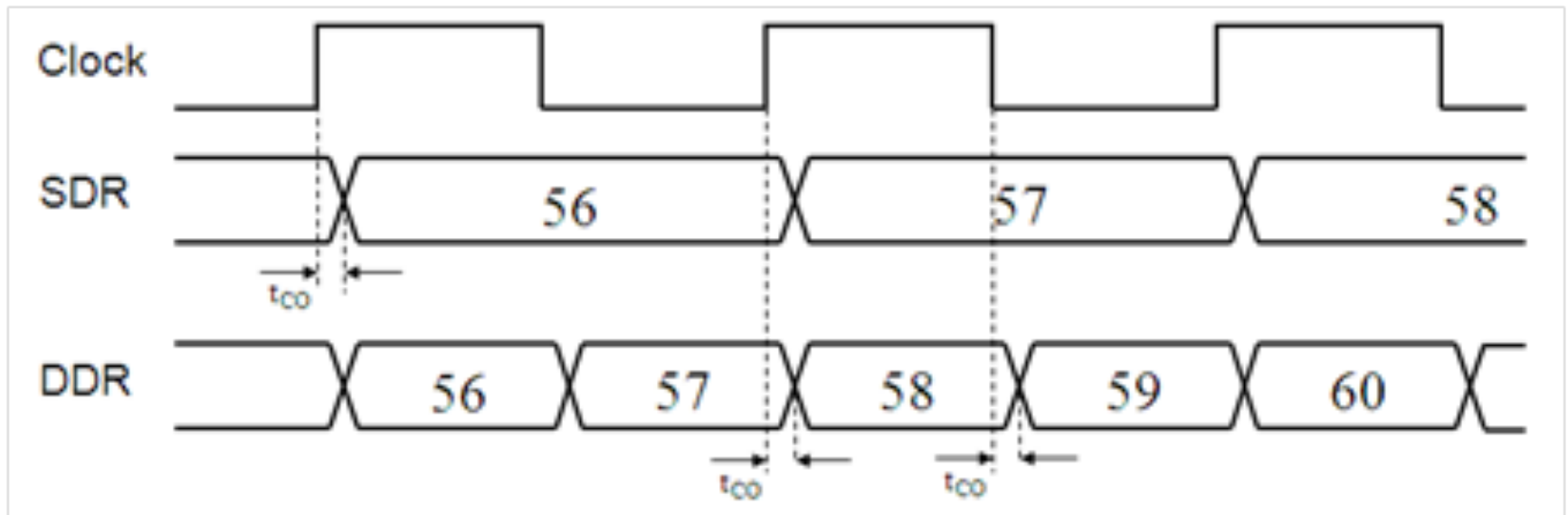
Solving Structural Hazard #1 with Caches



Structural Hazard #2: Registers (1/3)



Solving Structural Hazard #2 for the Regfile: Double Pumping (2/3)



* **This is not how registers work in general!**

Structural Hazard #2: Registers (3/3)

- Two different solutions have been used:
 - 1) Split RegFile access in two: Write during 1st half and Read during 2nd half of each clock cycle
 - Possible because RegFile access is *VERY* fast (takes less than half the time of ALU stage)
 - 2) Build RegFile with independent read and write ports
- **Conclusion:** Read and Write to registers during same clock cycle is okay

Structural hazards can always be removed by adding hardware resources

2. Data Hazards (1/2)

- Consider the following sequence of instructions:

add `$t0`, `$t1`, `$t2`

sub `$t4`, `$t0`, `$t3`

and `$t5`, `$t0`, `$t6`

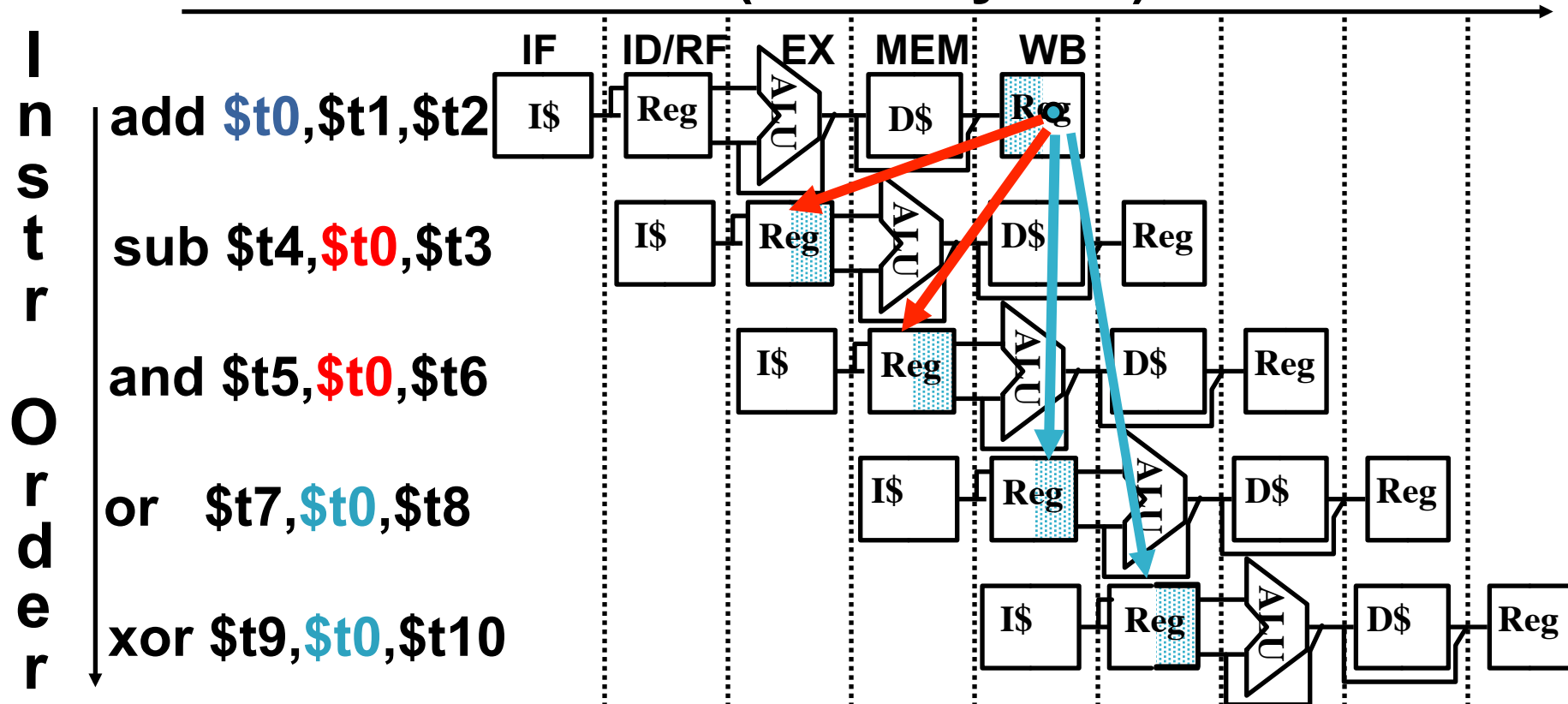
or `$t7`, `$t0`, `$t8`

xor `$t9`, `$t0`, `$t10`

2. Data Hazards (2/2)

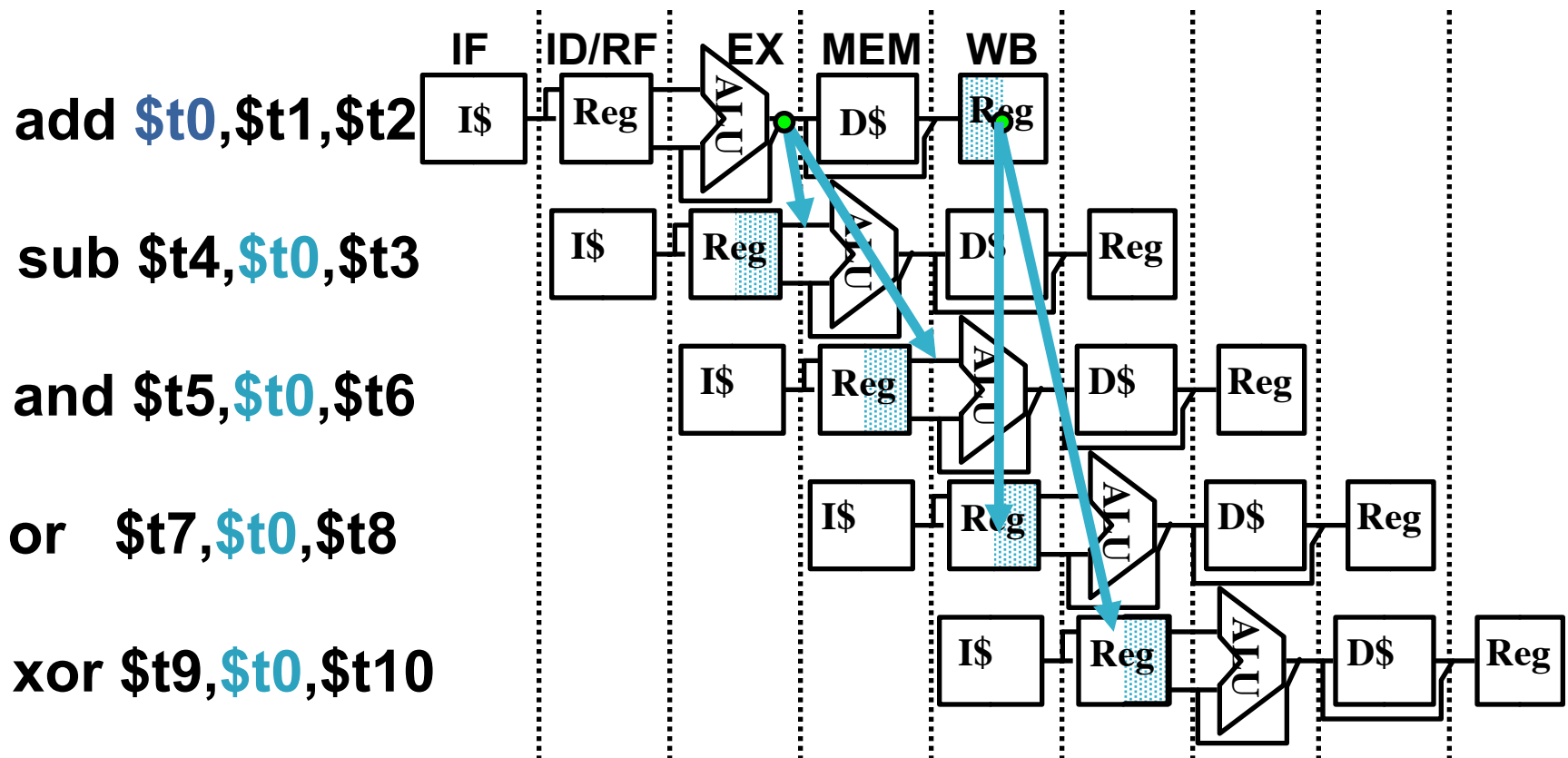
- Data-flow *backwards* in time are hazards

Time (clock cycles)



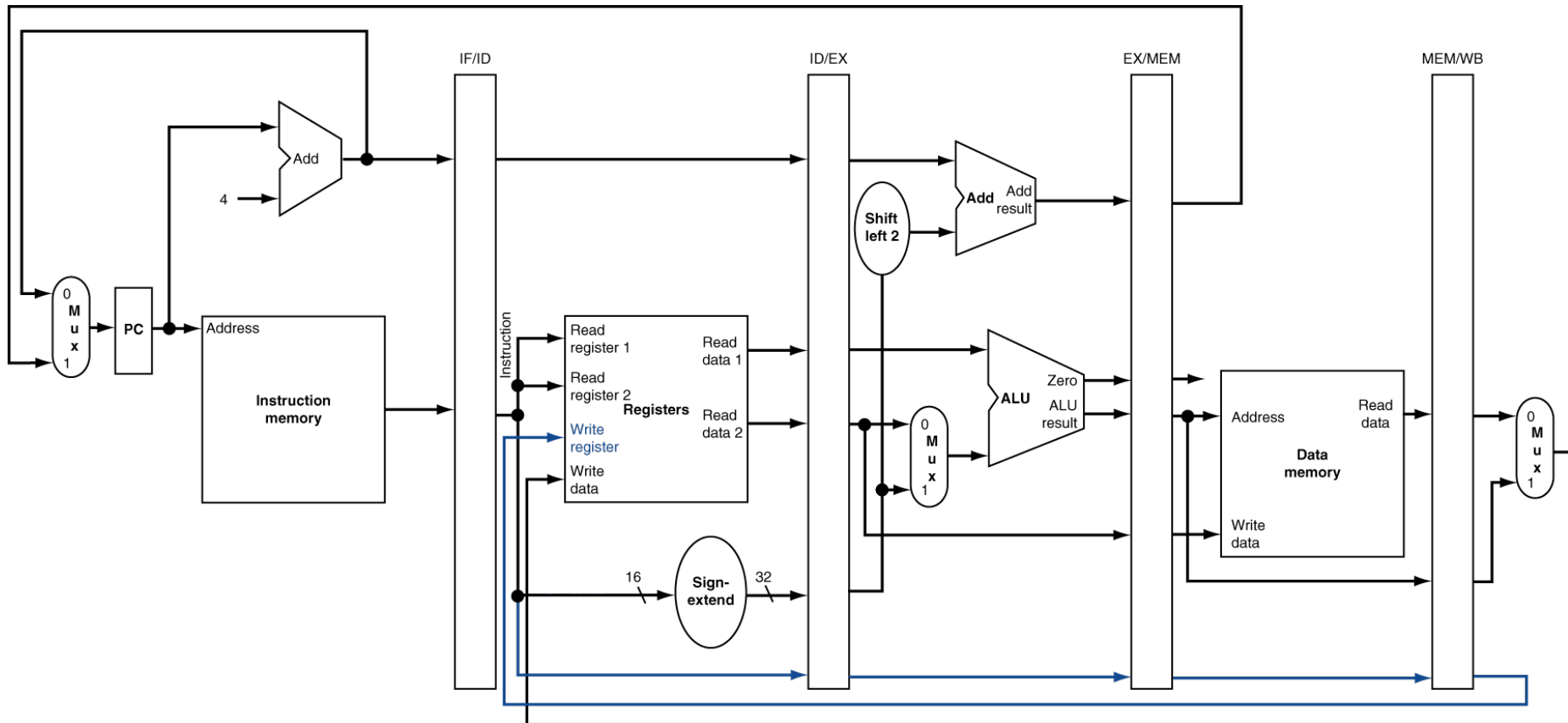
Data Hazard Solution: Forwarding

- Forward result as soon as it is available
 - OK that it's not stored in RegFile yet



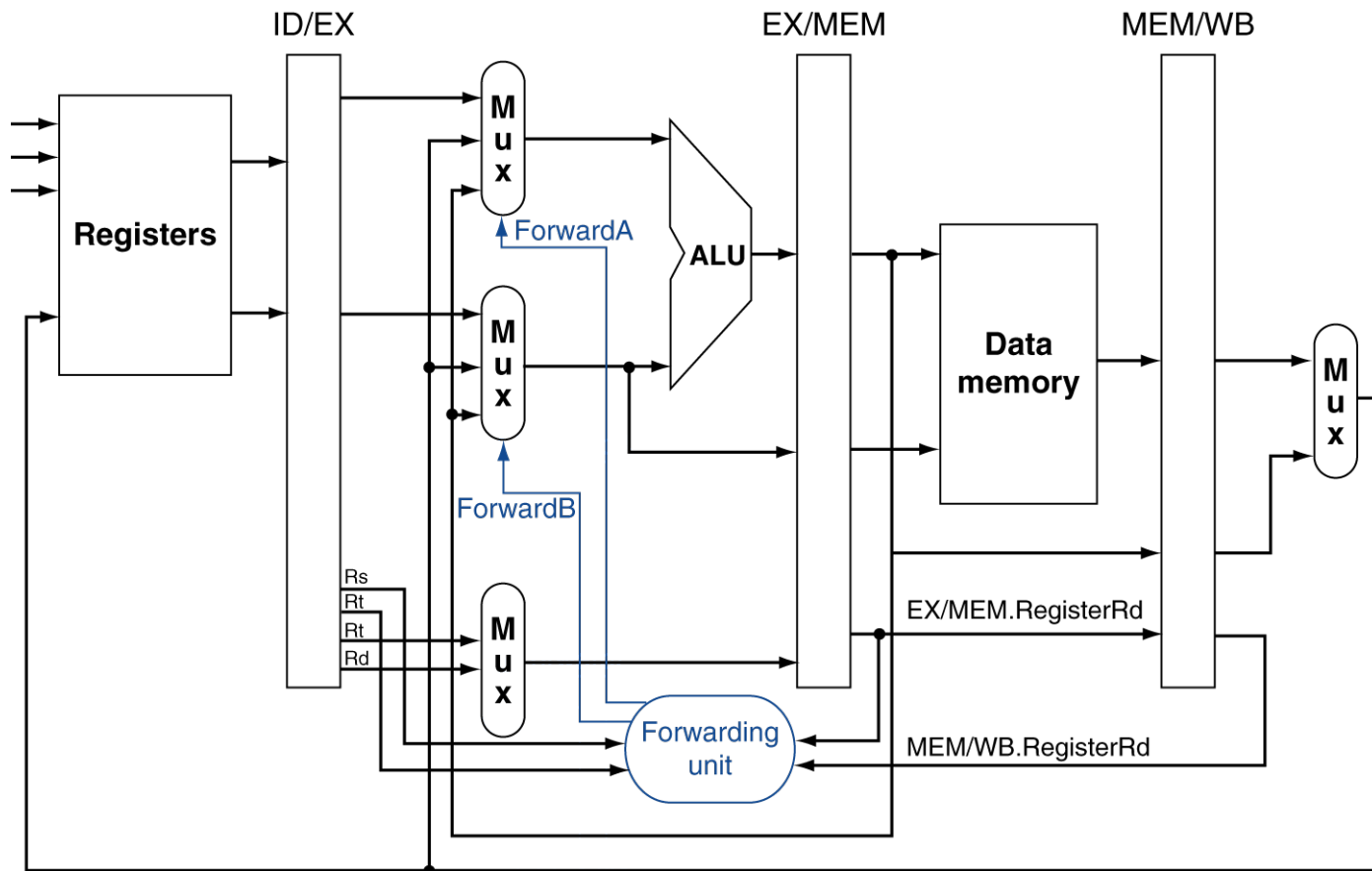
Datapath for Forwarding (1/2)

- What changes need to be made here?



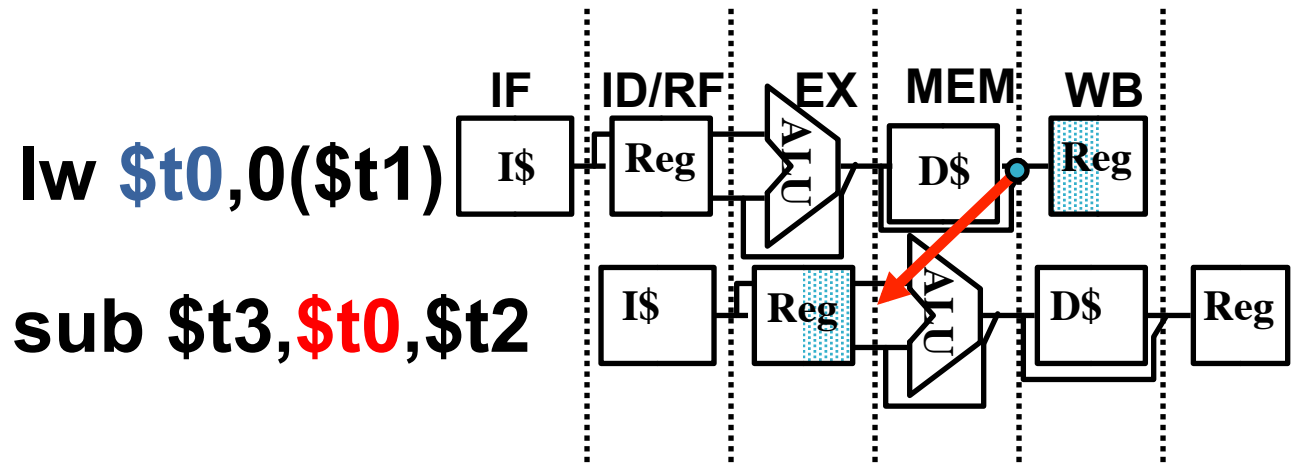
Datapath for Forwarding (2/2)

- Handled by *forwarding unit*



Data Hazard: Loads (1/4)

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



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

Data Hazard: Loads (2/4)

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

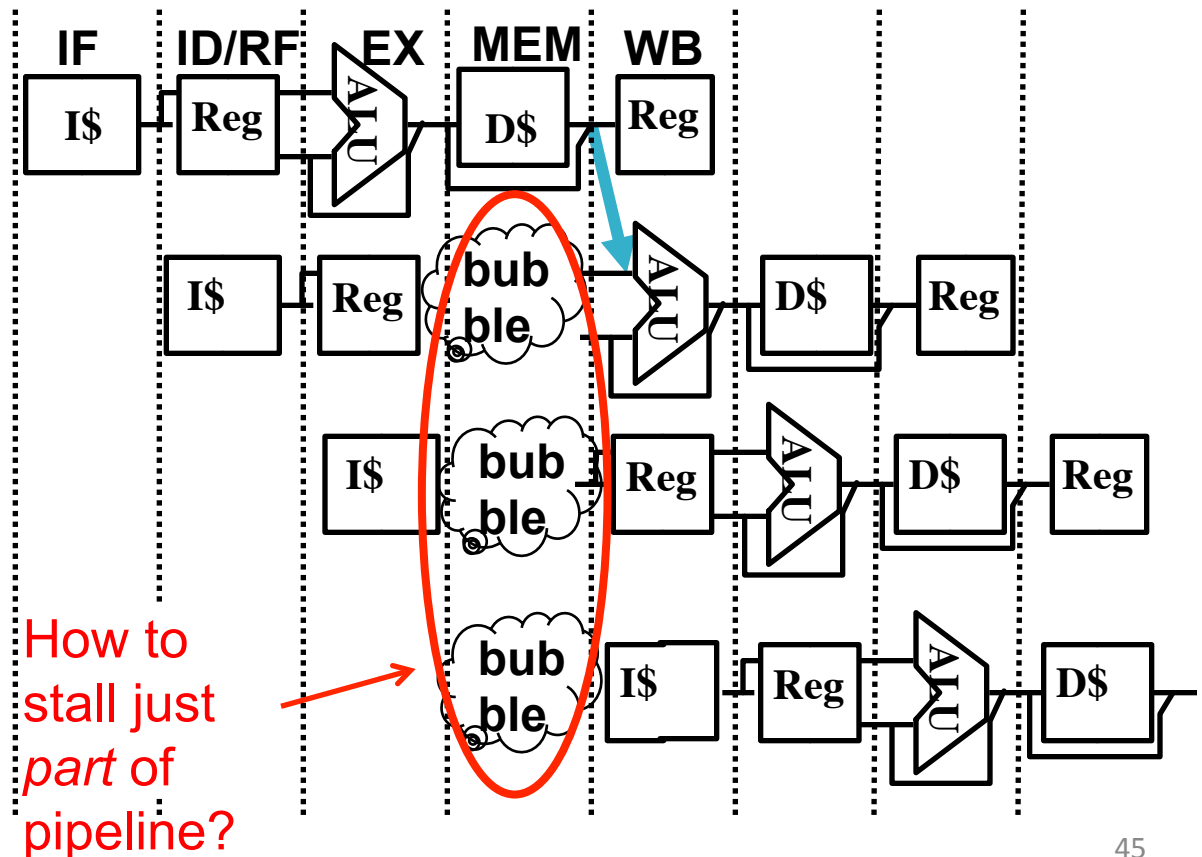
Schematically, this is what we want, but in reality stalls done “horizontally”

lw \$t0, 0(\$t1)

sub \$t3,\$t0,\$t2

and \$t5,\$t0,\$t4

or \$t7,\$t0,\$t6



Data Hazard: Loads (3/4)

- Stalled instruction converted to “bubble”, acts like nop

lw **\$t0**, 0(\$t1)

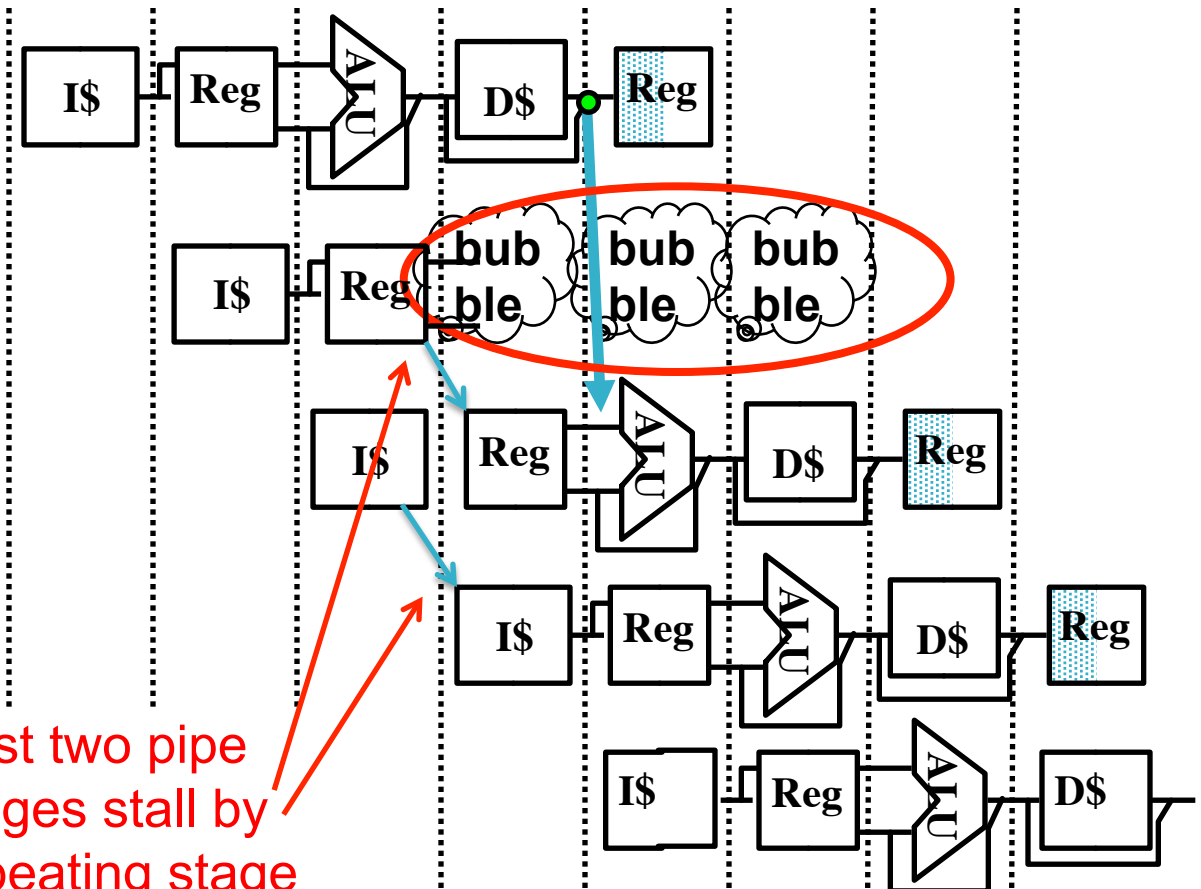
~~sub \$t3, **\$t0**, \$t2~~

sub \$t3, **\$t0**, \$t2

and \$t5, **\$t0**, \$t4

or \$t7, **\$t0**, \$t6

First two pipe
stages stall by
repeating stage
one cycle later

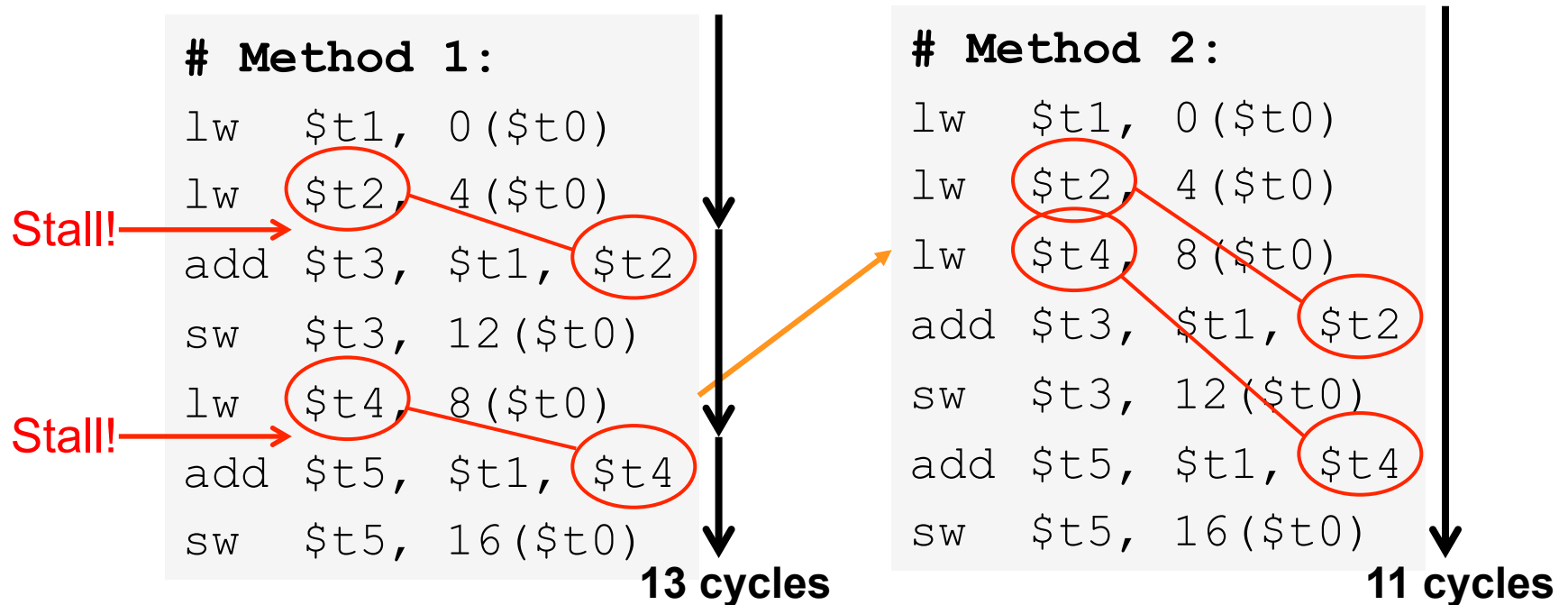


Data Hazard: Loads (4/4)

- Slot after a load is called a *load delay slot*
 - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
 - Letting the hardware stall the instruction in the delay slot is equivalent to putting an explicit `nop` in the slot (except the latter uses more code space)
- **Idea:** Ask the compiler to 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!
- MIPS code for $D=A+B$; $E=A+C$;



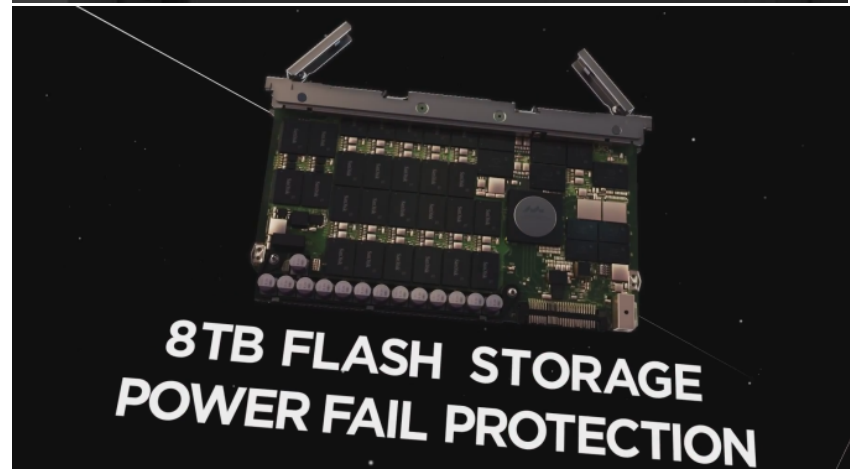
Clicker Question

In general, registers capture values only on the rising edge of the clock – double pumping the RegFile is an optimization, not the norm.

- A. True
- B. False

CS61C In The News: SanDisk announces ½ PetaByte flash drive

- 512TB of flash memory in 3U of rack space
 - That's 2^{49} bytes
- 780,000 I/O/second
- 7 GB/s sustained bandwidth
- < \$1 per GB
- 1 GB $\rightarrow 2^{30} \rightarrow 2^{19}$ dollars = \$524,288



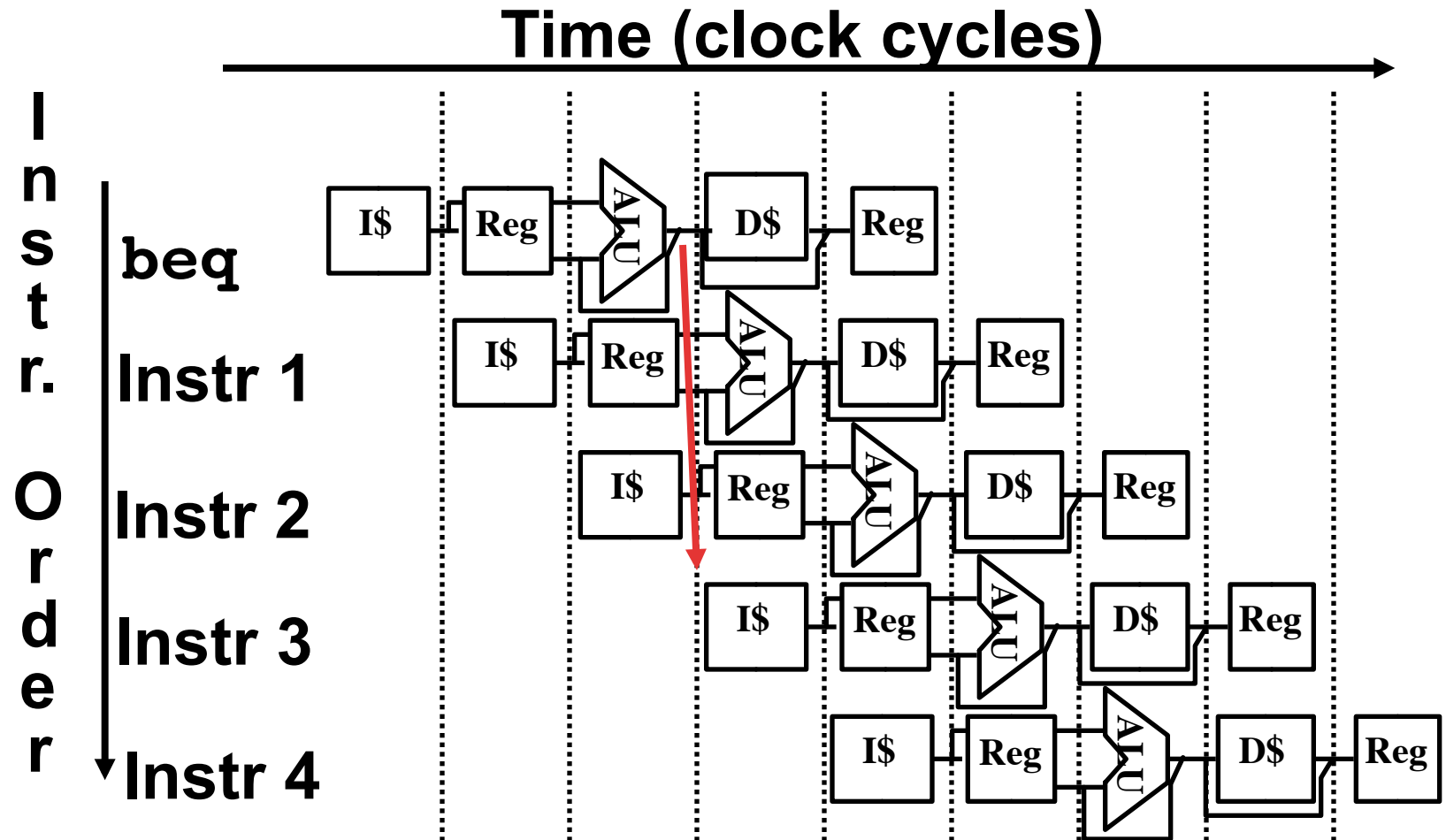
Break

- (Maybe)

3. Control Hazards

- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: *Stall* on every branch until branch condition resolved
 - Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)

Stall => 2 Bubbles/Clocks



Where do we do the compare for the branch?

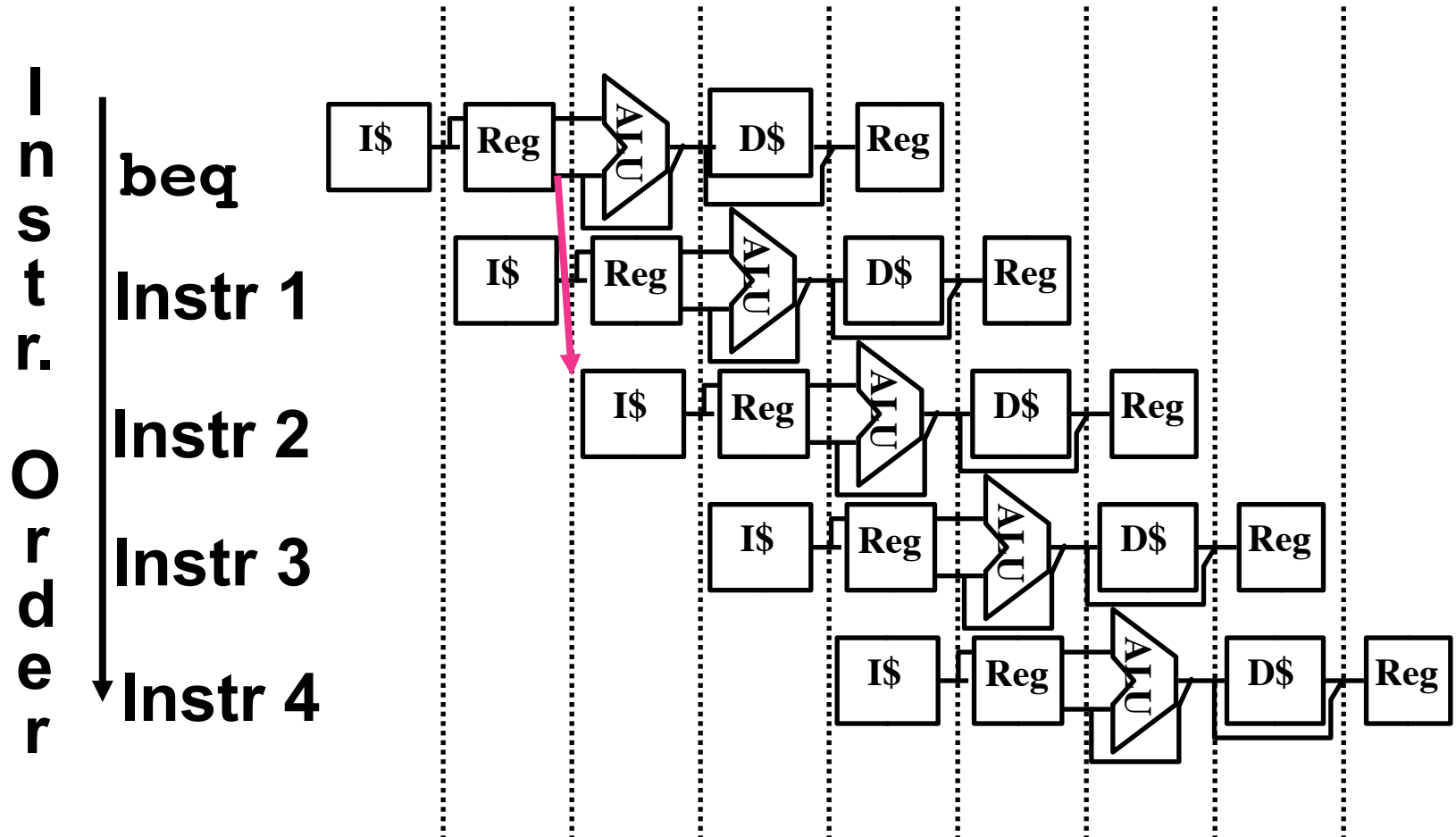
Control Hazard: Branching

- Optimization #1:
 - Insert **special branch comparator** in Stage 2
 - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
 - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
 - Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What's an efficient way to implement the equality comparison?

One Clock Cycle Stall

Time (clock cycles)



Branch comparator moved to Decode stage.

Control Hazards: Branching

- Option 2: *Predict* outcome of a branch, fix up if guess wrong
 - Must cancel all instructions in pipeline that depended on guess that was wrong
 - This is called “*flushing*” the pipeline
- Simplest hardware if we predict that all branches are NOT taken
 - Why?

Control Hazards: Branching

- Option #3: Redefine branches
 - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
 - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the *branch-delay slot*)
- *Delayed Branch* means *we always execute inst after branch*
- This optimization is used with MIPS

Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

or \$8, \$9, \$10

add \$1, \$2, \$3

sub \$4, \$5, \$6

beq \$1, \$4, Exit

xor \$10, \$1, \$11

Exit:

Delayed Branch

add \$1, \$2, \$3

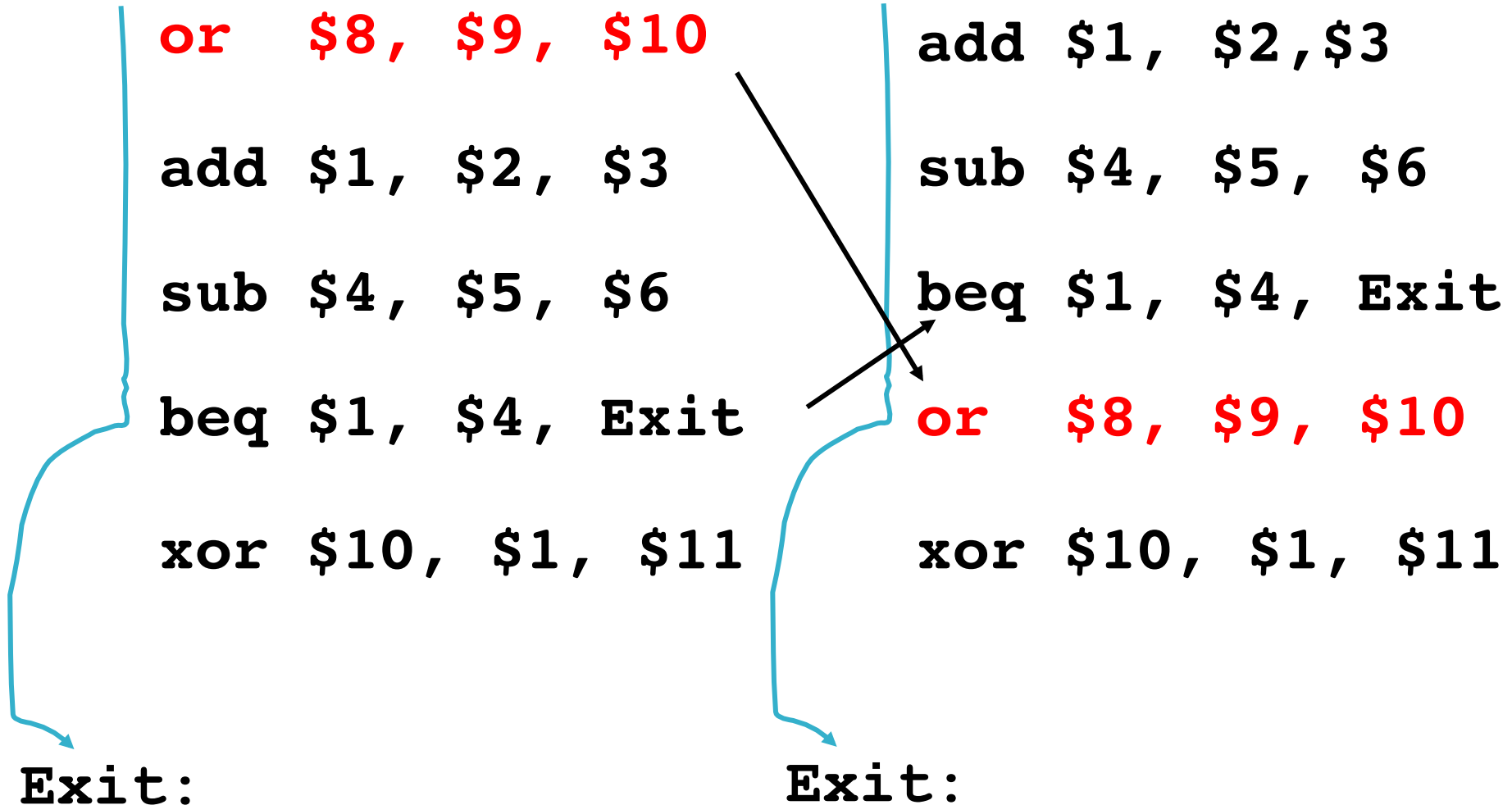
sub \$4, \$5, \$6

beq \$1, \$4, Exit

or \$8, \$9, \$10

xor \$10, \$1, \$11

Exit:



Control Hazards: Branching

- Notes on Branch-Delay Slot
 - Worst-Case Scenario: put a nop in the branch-delay slot
 - Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn't affect the logic of program
 - Re-ordering instructions is common way to speed up programs
 - Compiler usually finds such an instruction 50% of time
 - Jumps also have a delay slot
 - Thus the $R[ra] = PC + 8$

Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 \Rightarrow 10 \Rightarrow 15 stages)
 - Less work per stage \Rightarrow shorter clock cycle
- Multiple issue “superscalar”
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - But dependencies reduce this in practice
- “Out-of-Order” execution
 - Reorder instructions dynamically in hardware to reduce impact of hazards
- *Take CS152 next to learn about these techniques!*

Clicker Question

Which of the following are true?

1. The jump delay slot produces the RTL:

$PC \leftarrow PC + 8$

2. The latency of an instruction usually decreases as a result of pipelining

3. The throughput of the CPU usually increases as a result of pipelining

- A. T, T, T
- B. T, T, F
- C. T, F, T
- D. T, F, F
- E. F, T, T

In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions in different pipeline stages
- Pipeline stages should be balanced for highest clock rate
- Three types of pipeline hazard limit performance
 - Structural (always fixable with more hardware)
 - Data (use interlocks or bypassing to resolve)
 - Control (reduce impact with branch prediction or branch delay slots)