CS 110
Computer Architecture

Pipelining

Instructor:
Sören Schwertfeger

http://shtech.org/courses/ca/

School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

Important Note...

根据
《上海科技大学本科生学籍管理实施细则（试行）》和
《上海科技大学研究生学籍管理实施细则（试行）》
的第十四条和十五条，上科大的所有课程均不安排补考，如学生申请缓考或者未通过考核，必须重新修读该课程。

Summary: Single-cycle Processor

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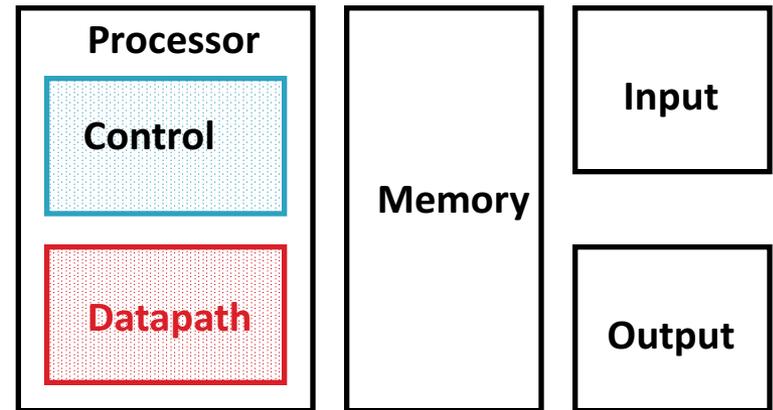   1. Analyze instruction set → datapath requirements
    2. Select set of datapath components & establish clock methodology
    3. Assemble datapath meeting the requirements
    4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
   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

Processor

Control

Datapath

Memory

Input

Output

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

- Clock rate (cycles/second = Hz) = 1/Period (seconds/cycle)
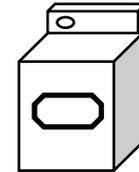
Single Cycle Performance

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

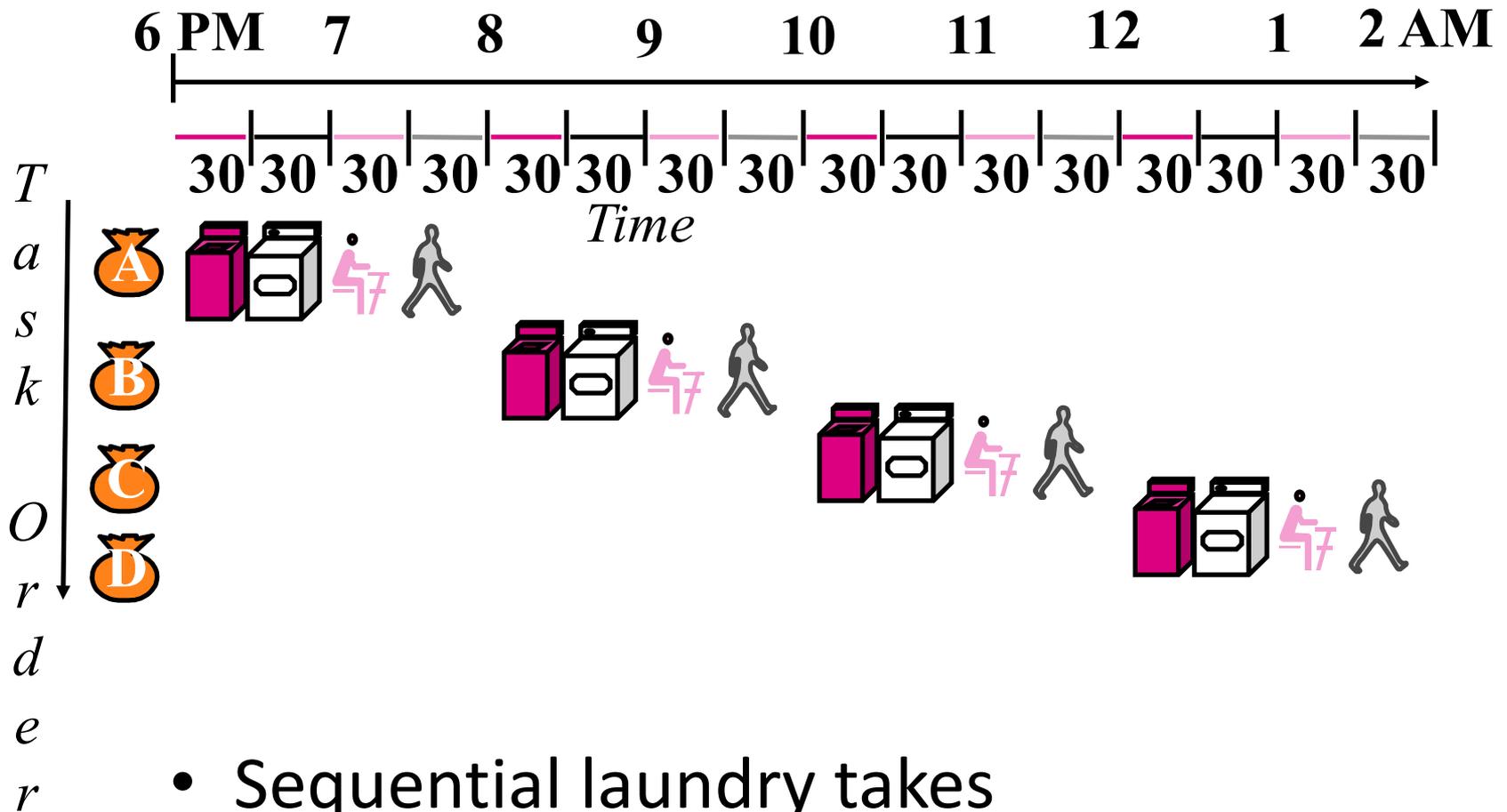| Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

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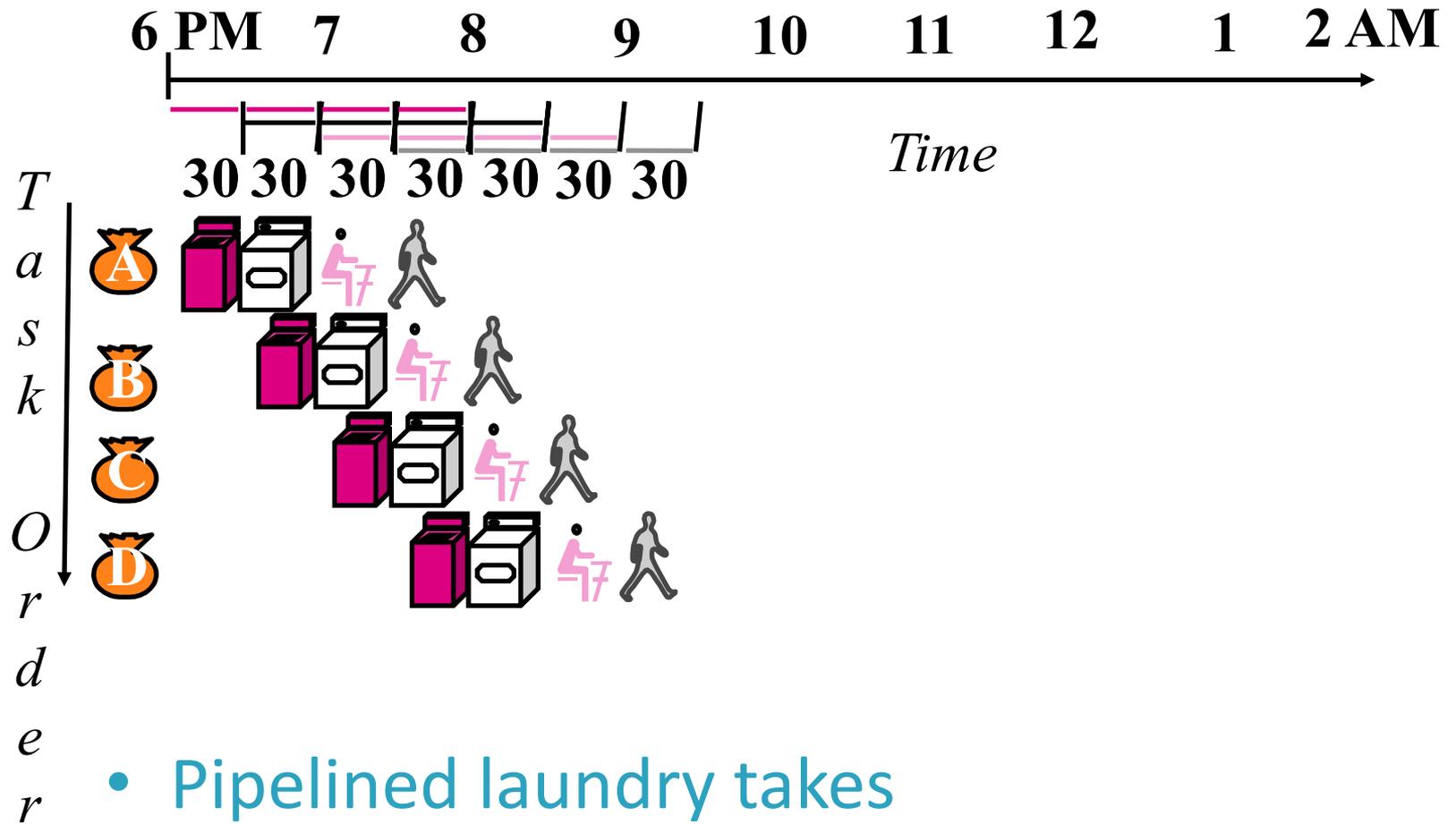 - Want increased clock rate to mean faster programs

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

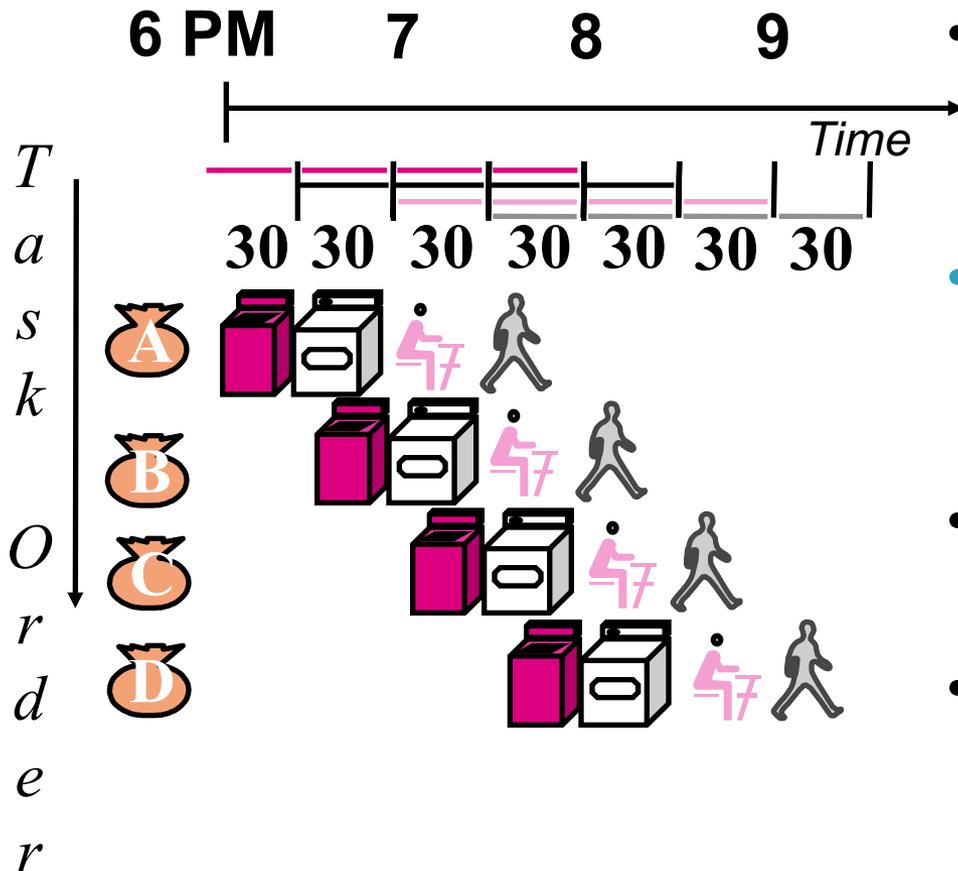 - "Stasher" takes 30 minutes to put clothes into drawers

Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Lessons (1/2)

6 PM   **7**   **8**   **9**

Time

T
a
s
k

O
r
d
e
r

30 30   30   30   30   30   30

A

B

C

D

- 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 (8/3.5) v. 4x (8/2) in this example
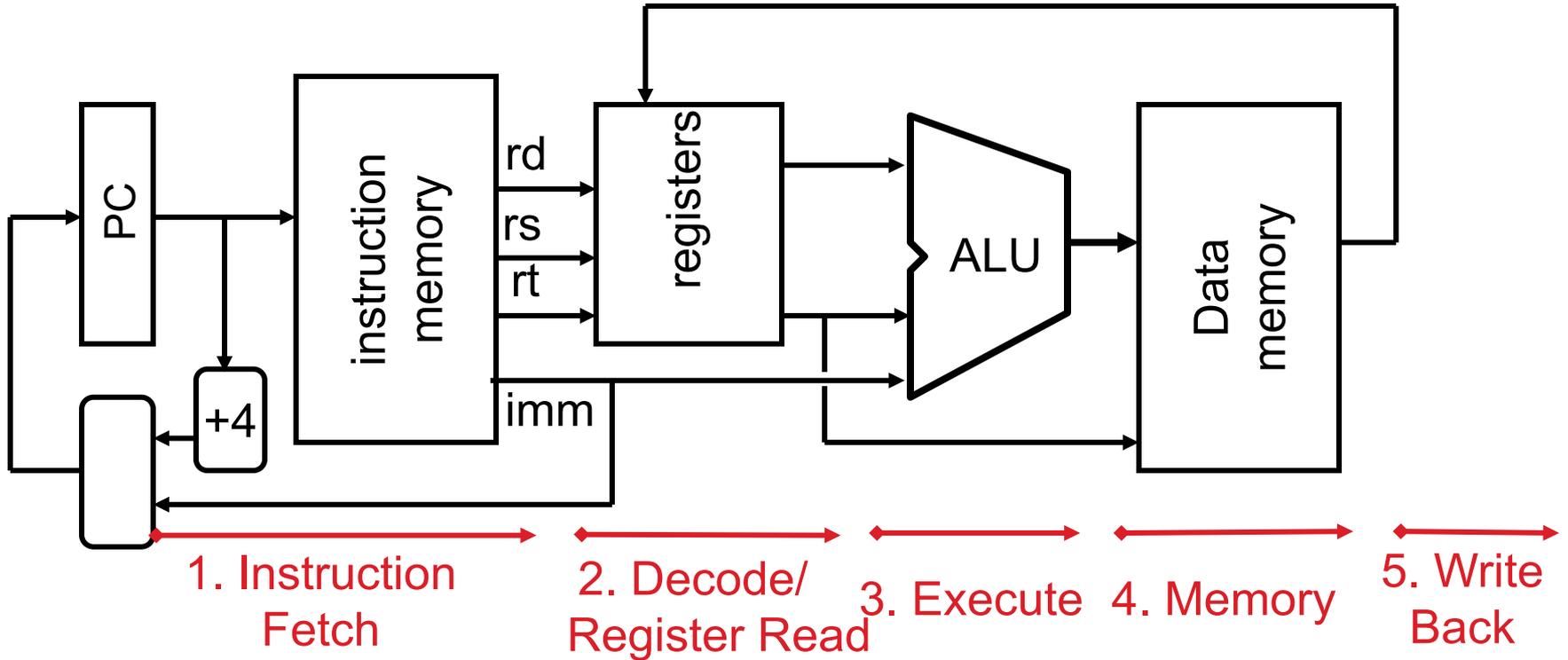
Pipelining Lessons (2/2)
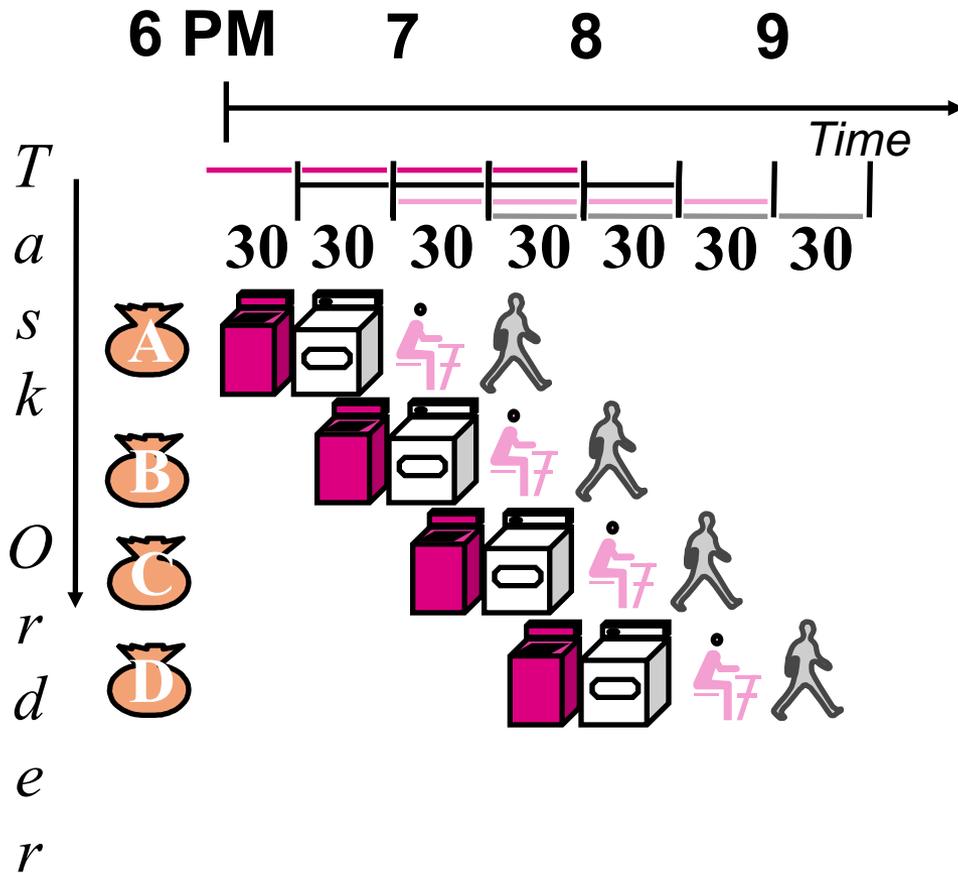
- Suppose new Dryer takes 20 minutes, new Folder takes 20 minutes. How much faster is pipeline?

- Pipeline rate limited by slowest pipeline stage

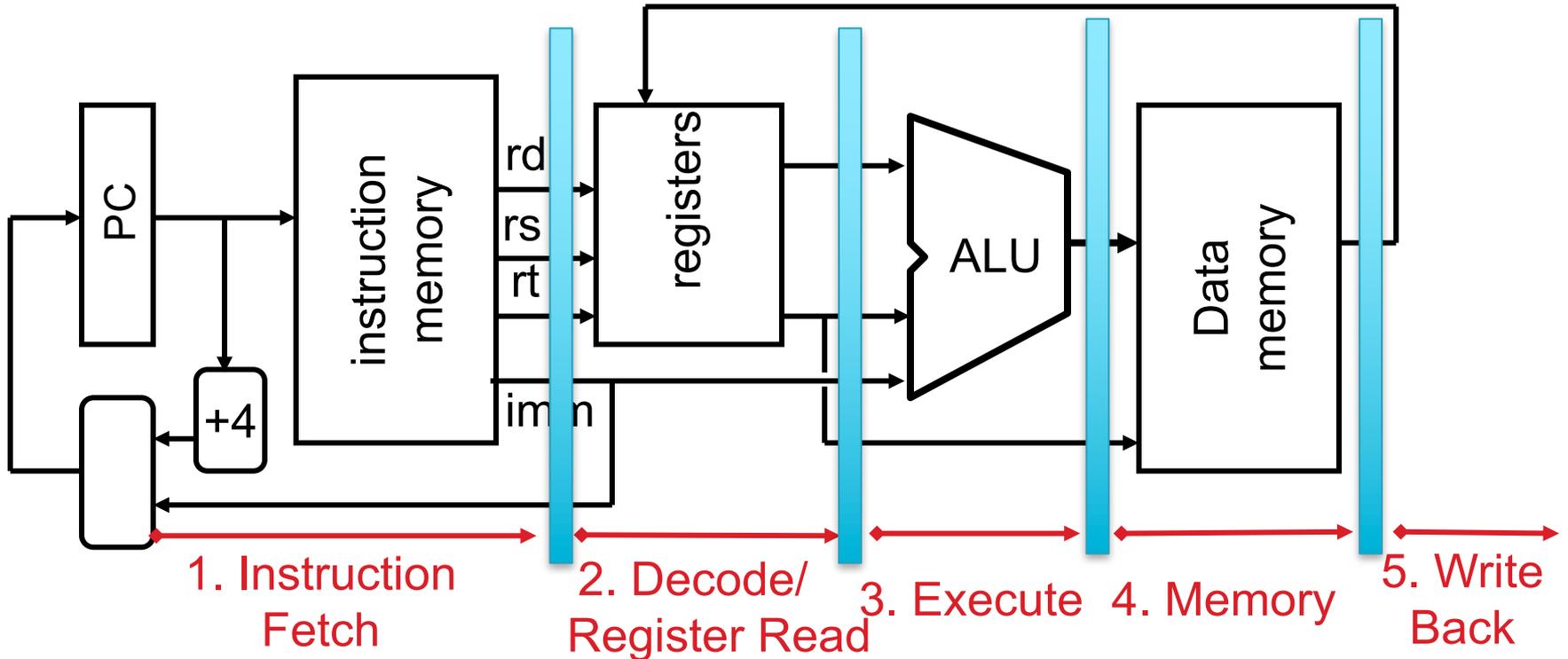- Unbalanced lengths of pipe stages reduces speedup

Execution Steps in MIPS Datapath

1) IFtch: Instruction Fetch, Increment PC

2) Dcd: 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
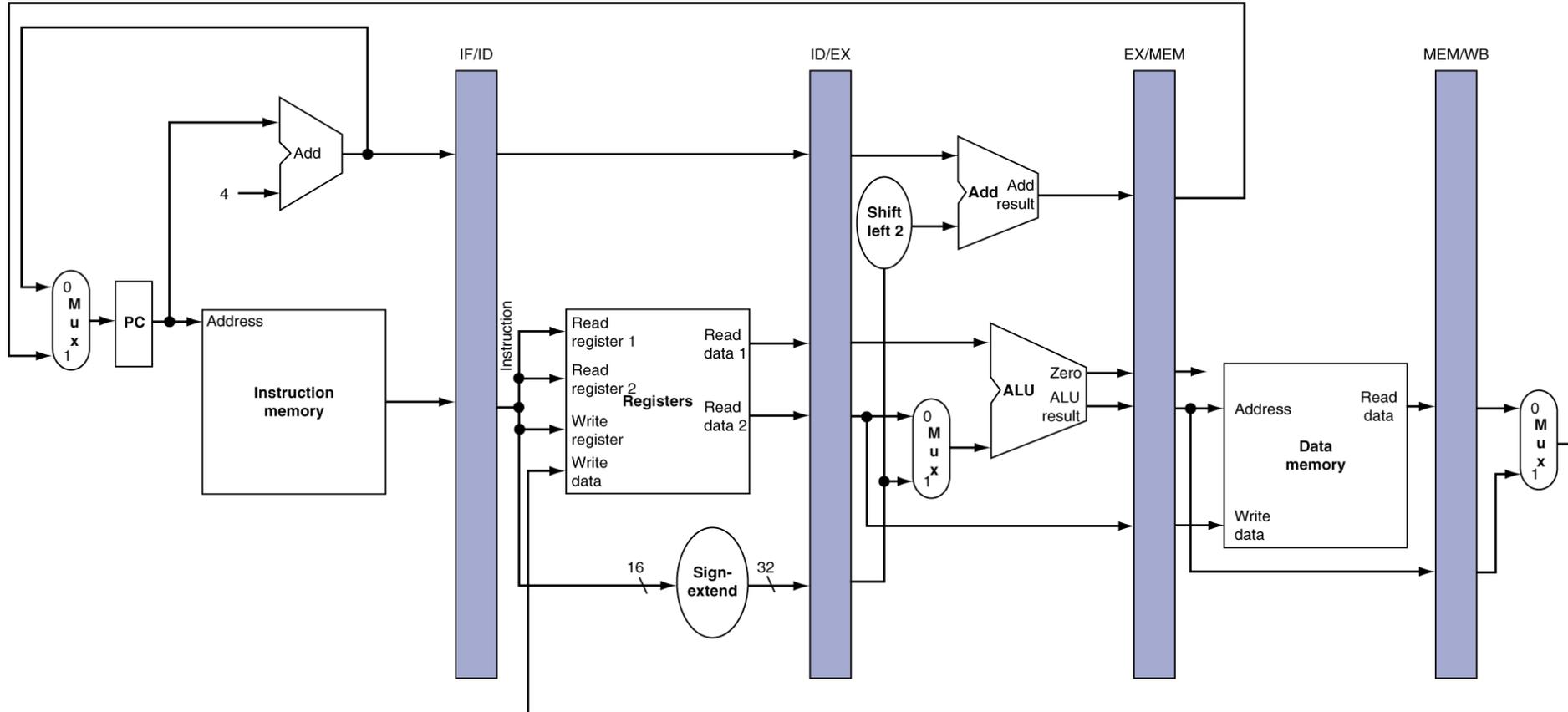
Pipeline registers

PC

instruction memory

+4

rd
rs
rt
imm

registers

ALU

Data memory

1. Instruction Fetch

2. Decode/ Register Read
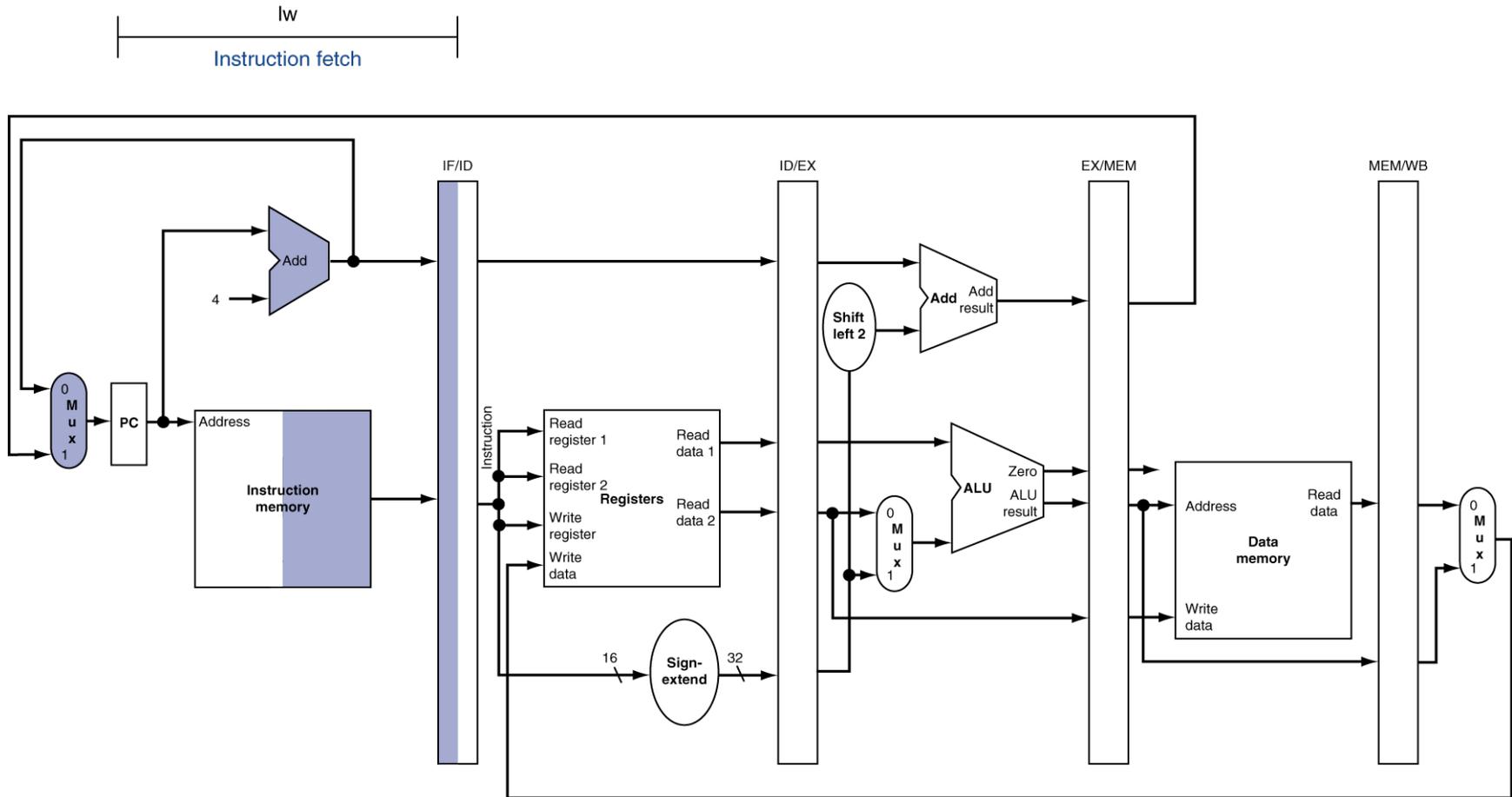
3. Execute

4. Memory

5. Write Back

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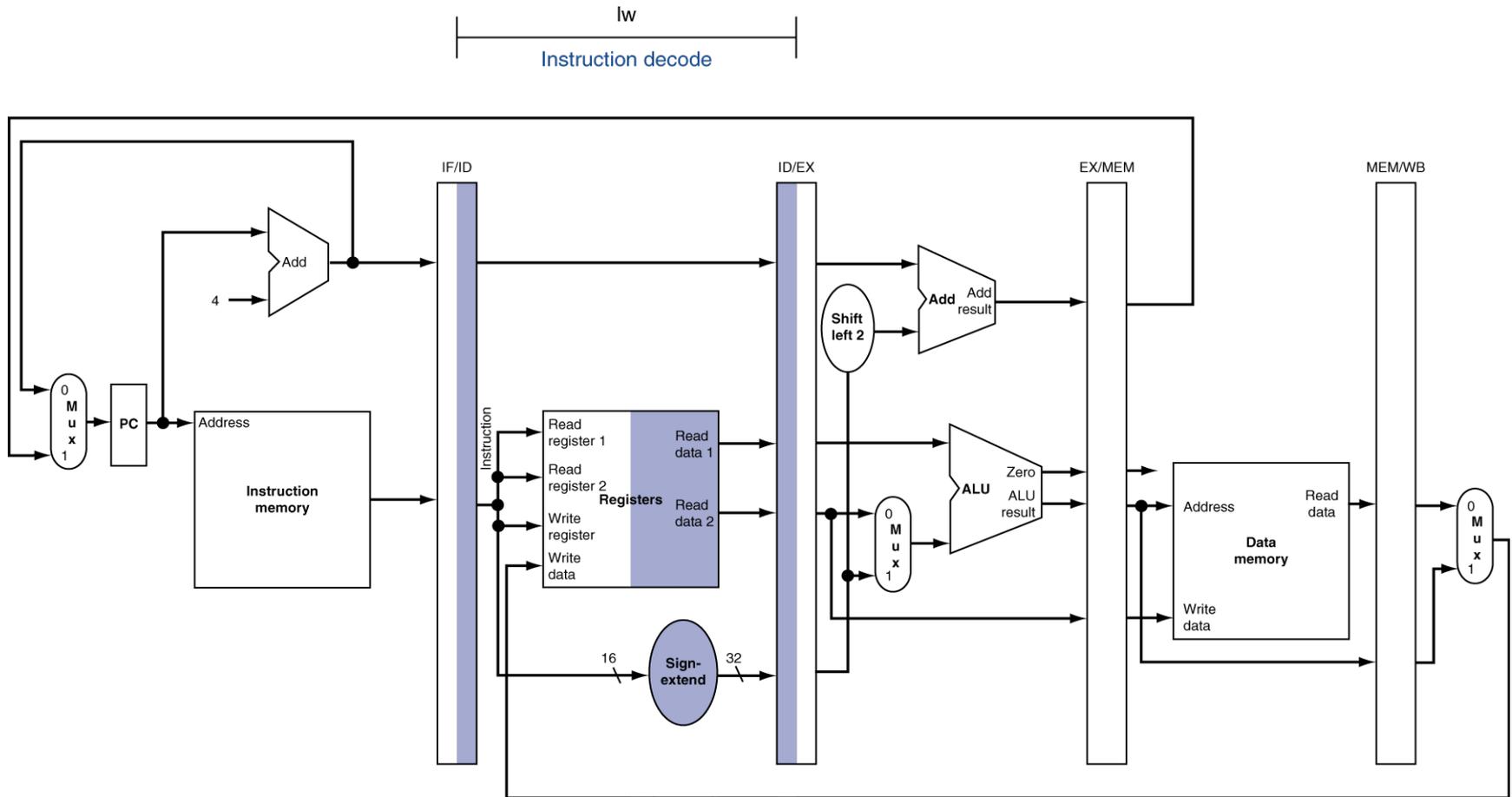 - To hold information produced in previous cycle

More Detailed Pipeline

IF for Load, Store, …

ID for Load, Store, …
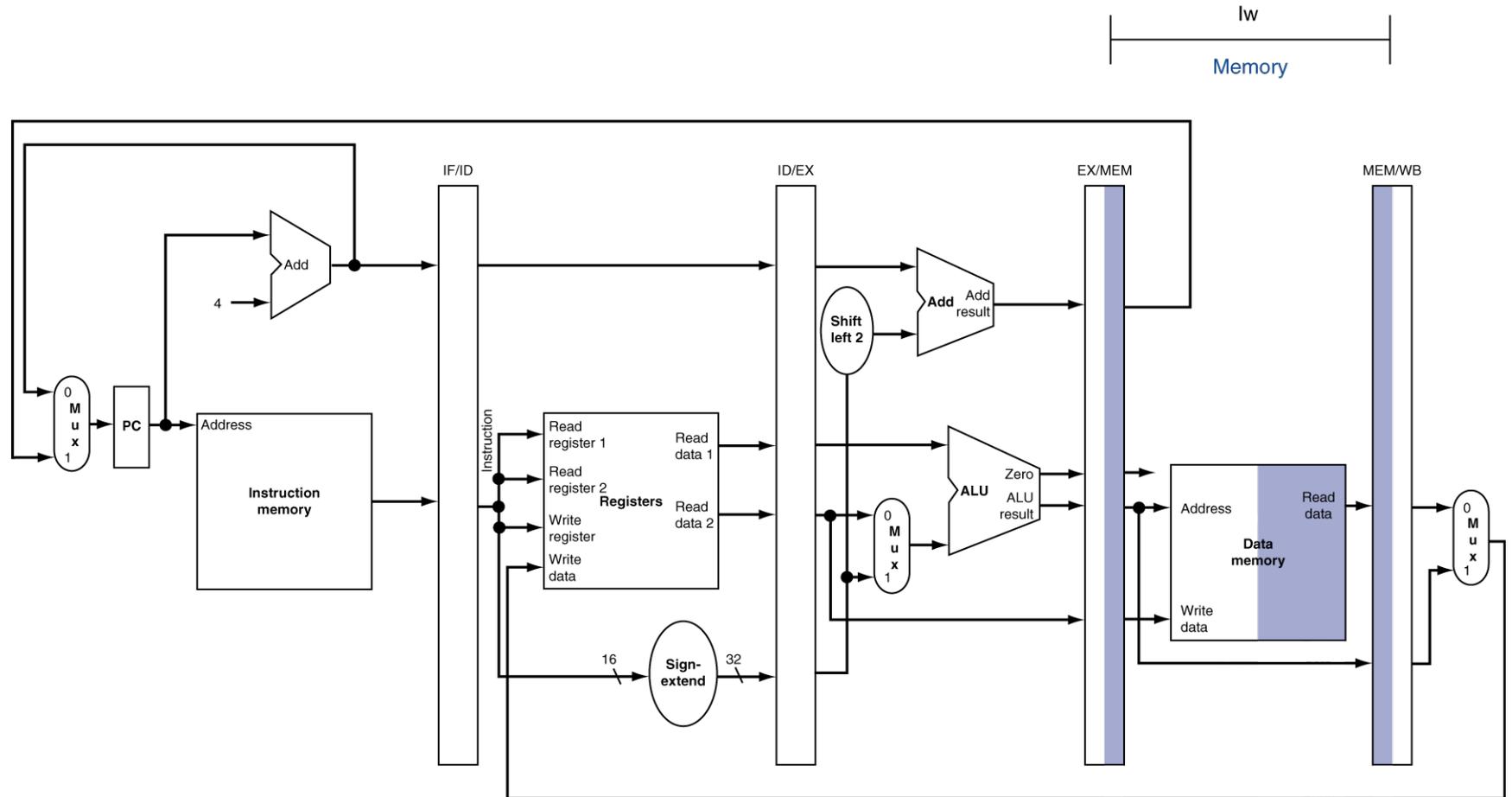
EX for Load

17
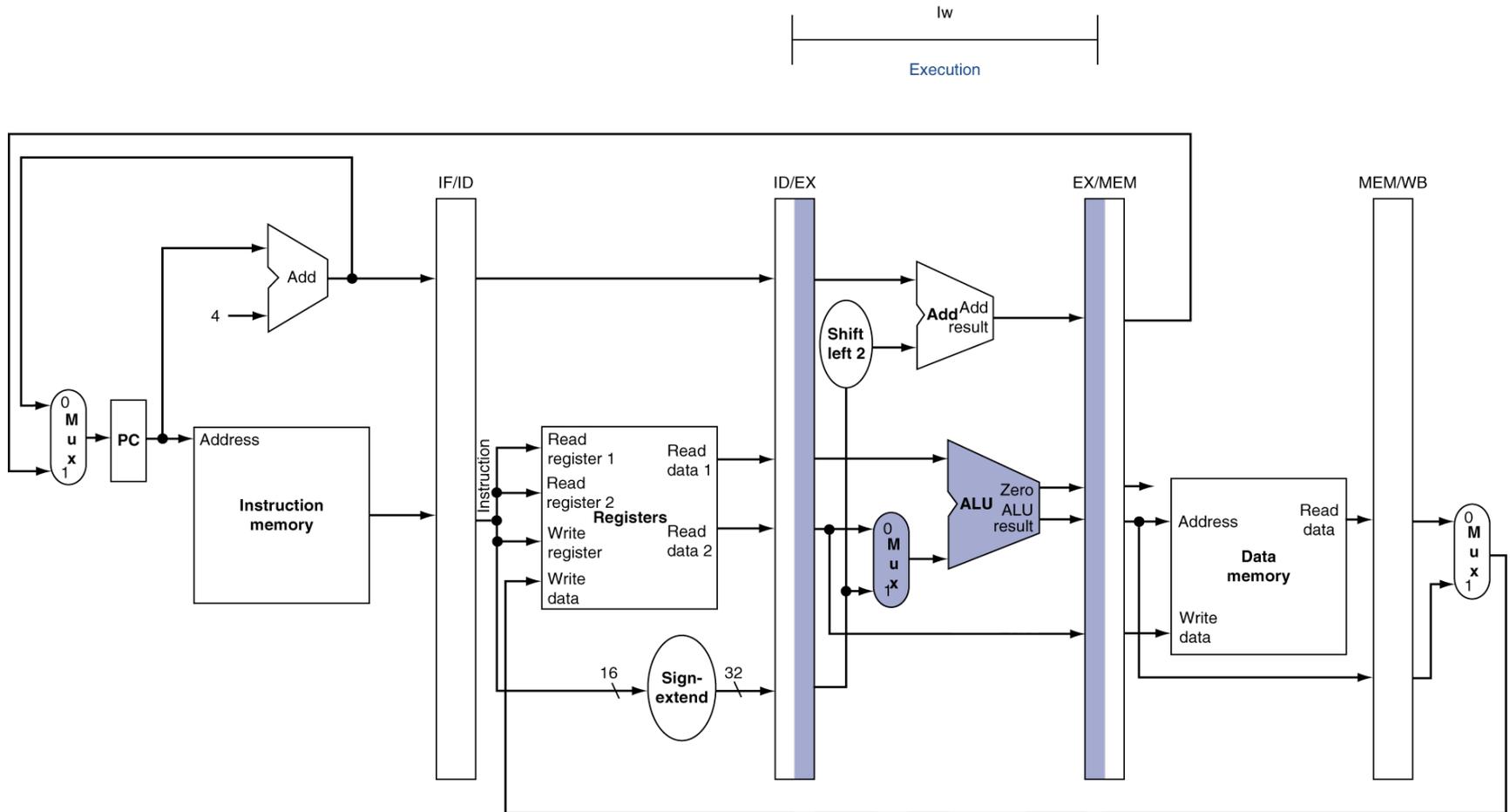
MEM for Load

WB for Load – Oops!
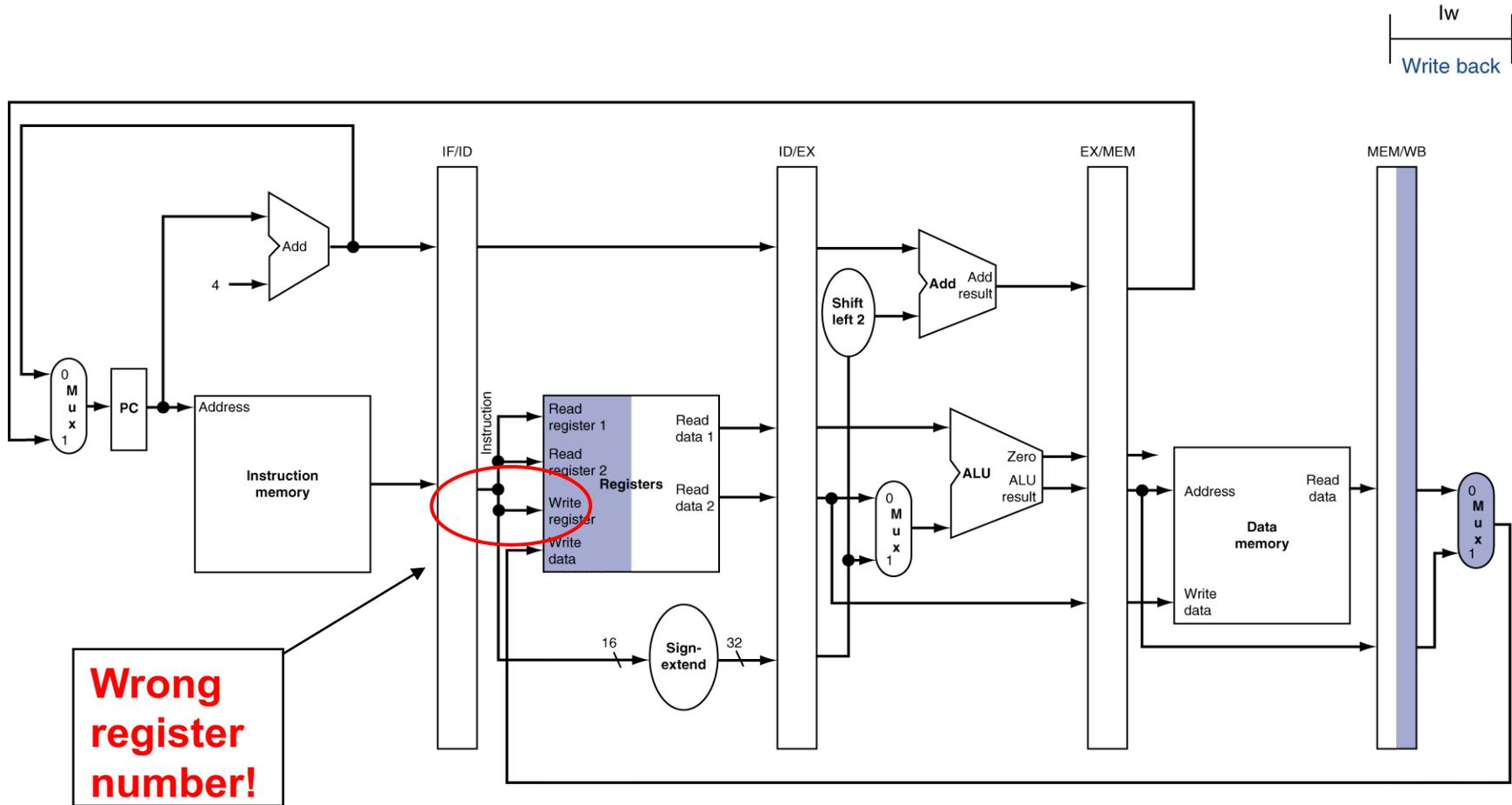
Corrected Datapath for Load
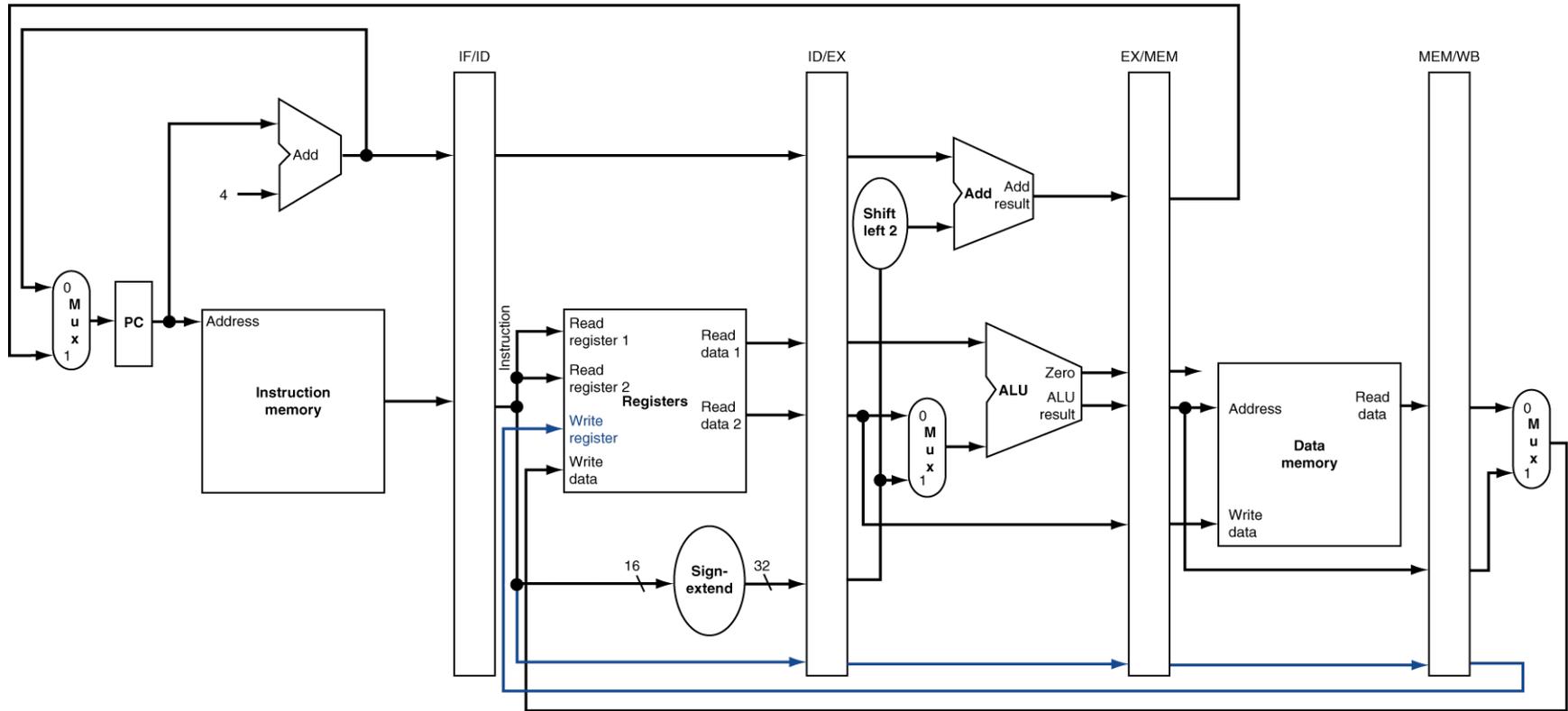
Pipelined Execution Representation

Time →

IF	ID	EX	MEM	WB

IF	ID	EX	MEM	WB

IF	ID	EX	MEM	WB

IF	ID	EX	MEM	WB

IF	ID	EX	MEM	WB

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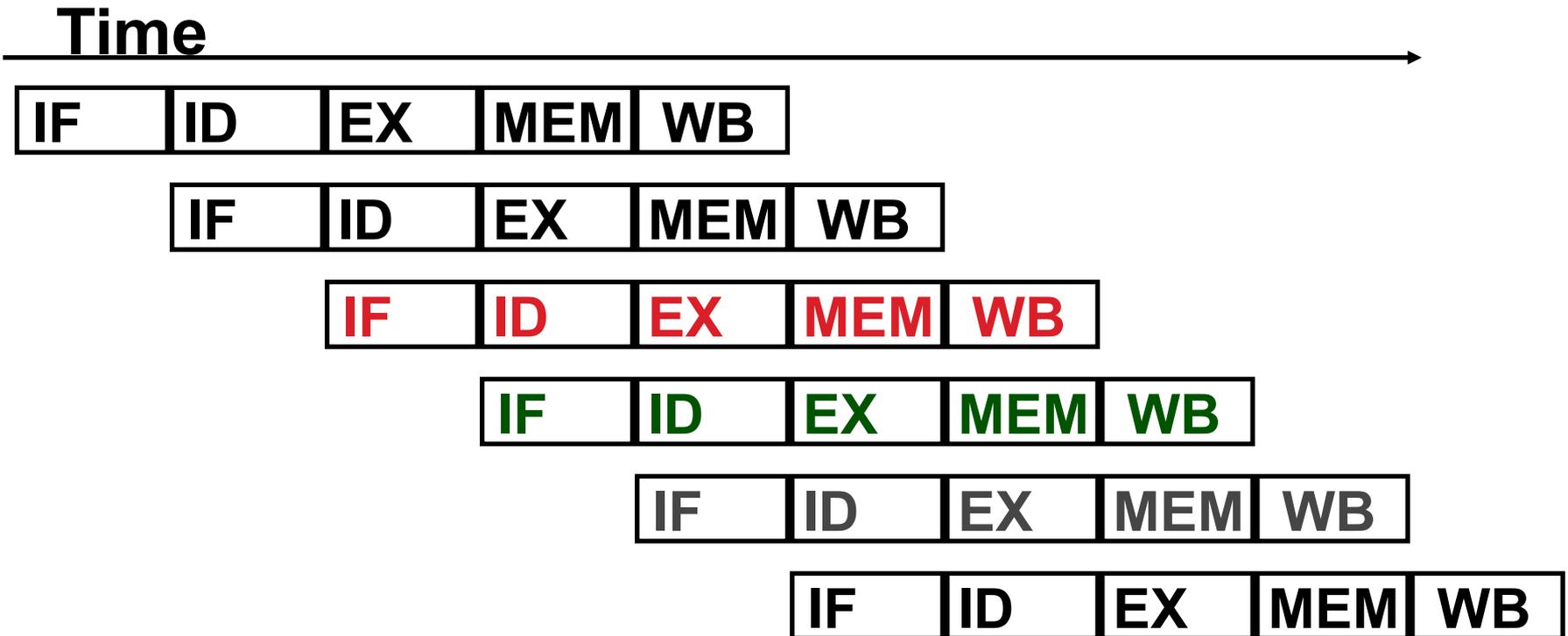
- Every instruction must take same number of steps, so some stages will idle
  - e.g. MEM stage for any arithmetic instruction

Graphical Pipeline Diagrams

1. Instruction Fetch
2. Decode/ Register Read
3. Execute
4. Memory
5. Write Back
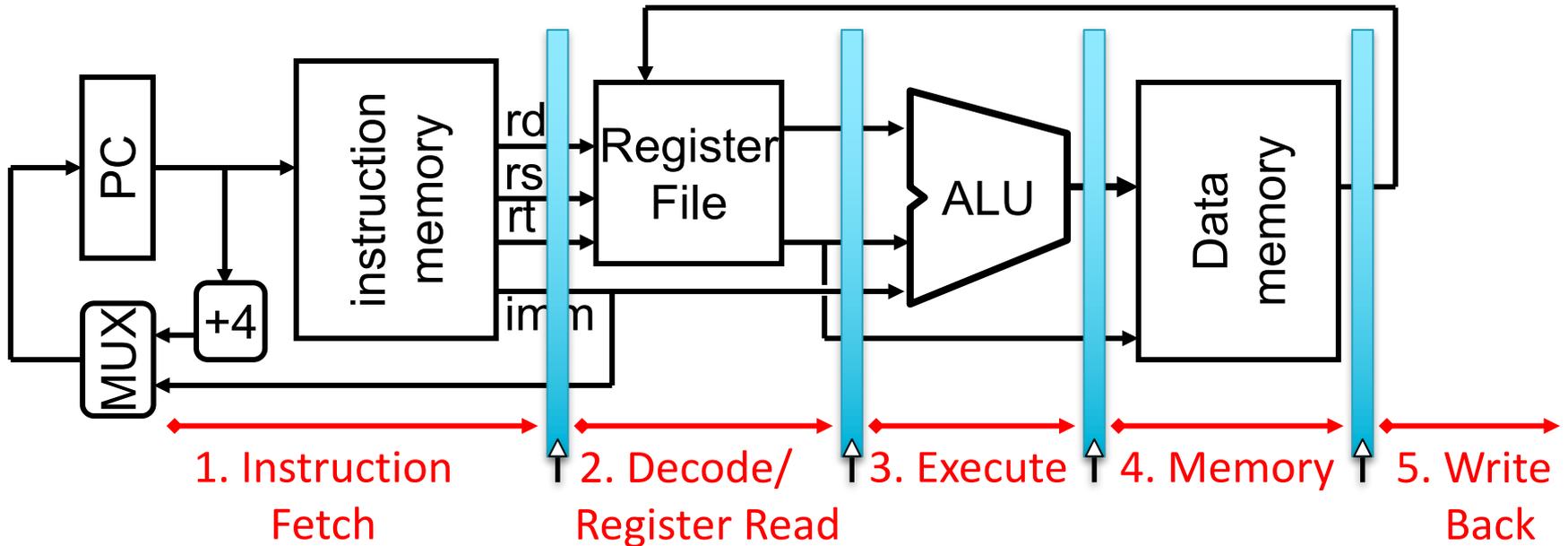
- Use datapath figure below to represent pipeline:

IF	ID	EX	Mem	WB
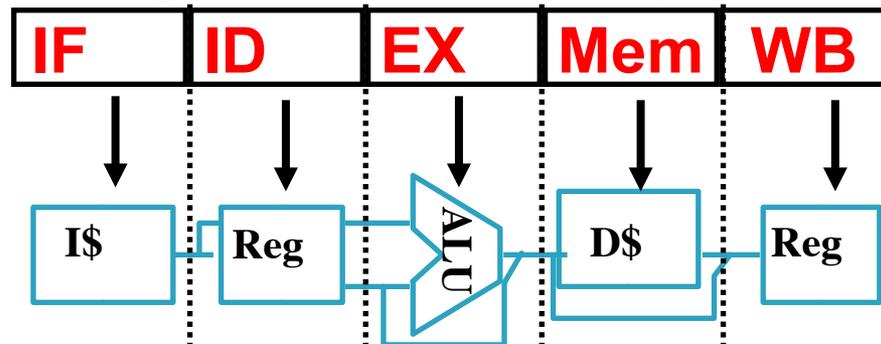
Graphical Pipeline Representation

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

Time (clock cycles)
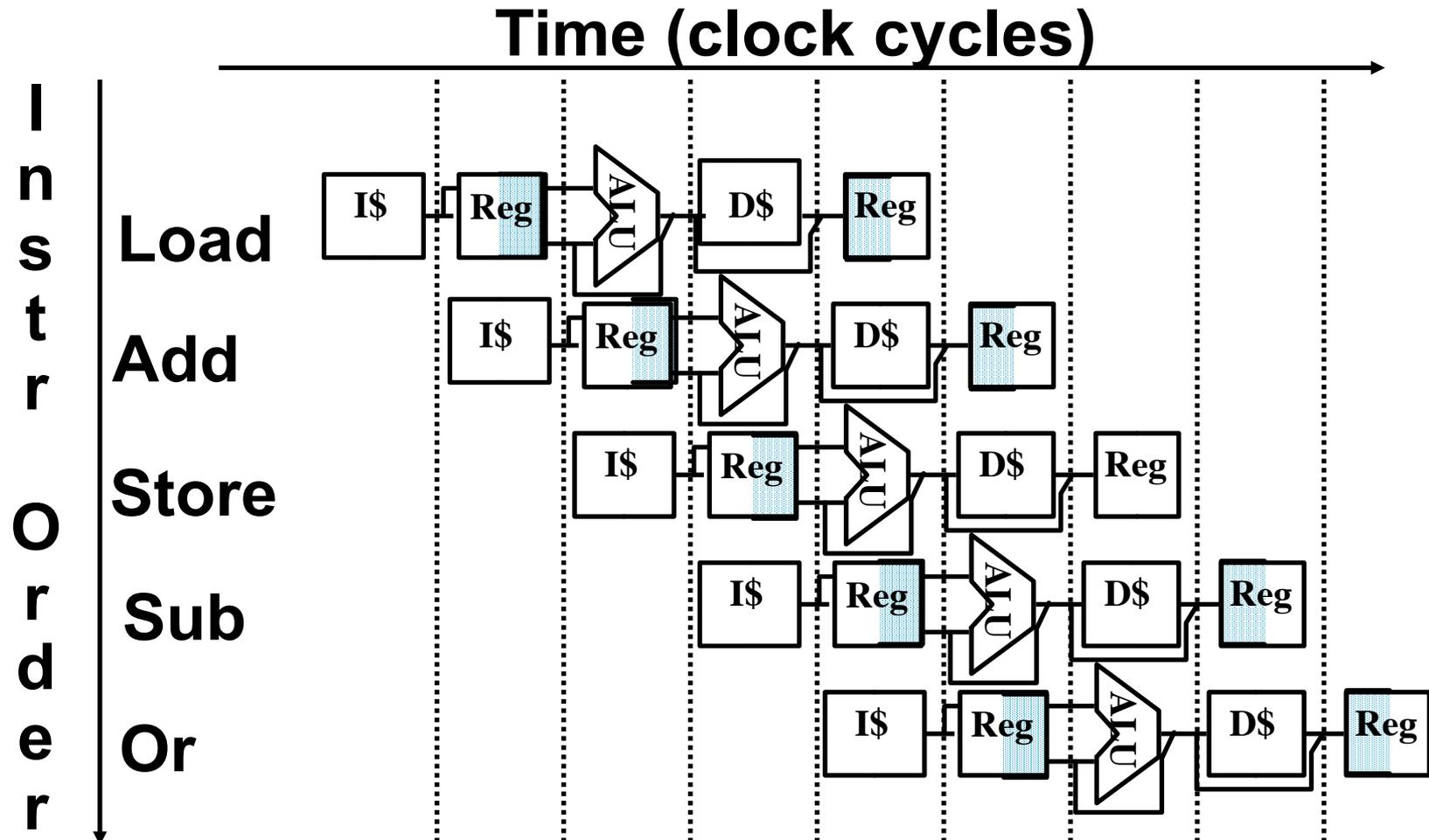
Pipelining Performance (1/3)

- 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)

Single-cycle
T_c = 800 ps
f = 1.25GHz

Program execution order (in instructions)

Time →
200 400 600 800 1000 1200 1400 1600 1800

lw $1, 100($0) — Instruction fetch | Reg | ALU | Data access | Reg

lw $2, 200($0) — 800 ps — Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — 800 ps — Instruction fetch

800 ps

Pipelined
T_c = 200 ps
f = 5GHz

Program execution order (in instructions)

Time →
200 400 600 800 1000 1200 1400

lw $1, 100($0) — Instruction fetch | Reg | ALU | Data access | Reg

lw $2, 200($0) — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0) — 200 ps — Instruction fetch | Reg | ALU | Data access | Reg

200 ps 200 ps 200 ps 200 ps 200 ps

Question

Logic in some stages takes 200ps and in some 100ps. Clk-Q delay is 30ps and setup-time is 20ps. What is the maximum clock frequency at which a pipelined design can operate?

- A: 10GHz

- B: 5GHz

- C: 6.7GHz

- D: 4.35GHz

- E: 4GHz

Question

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

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
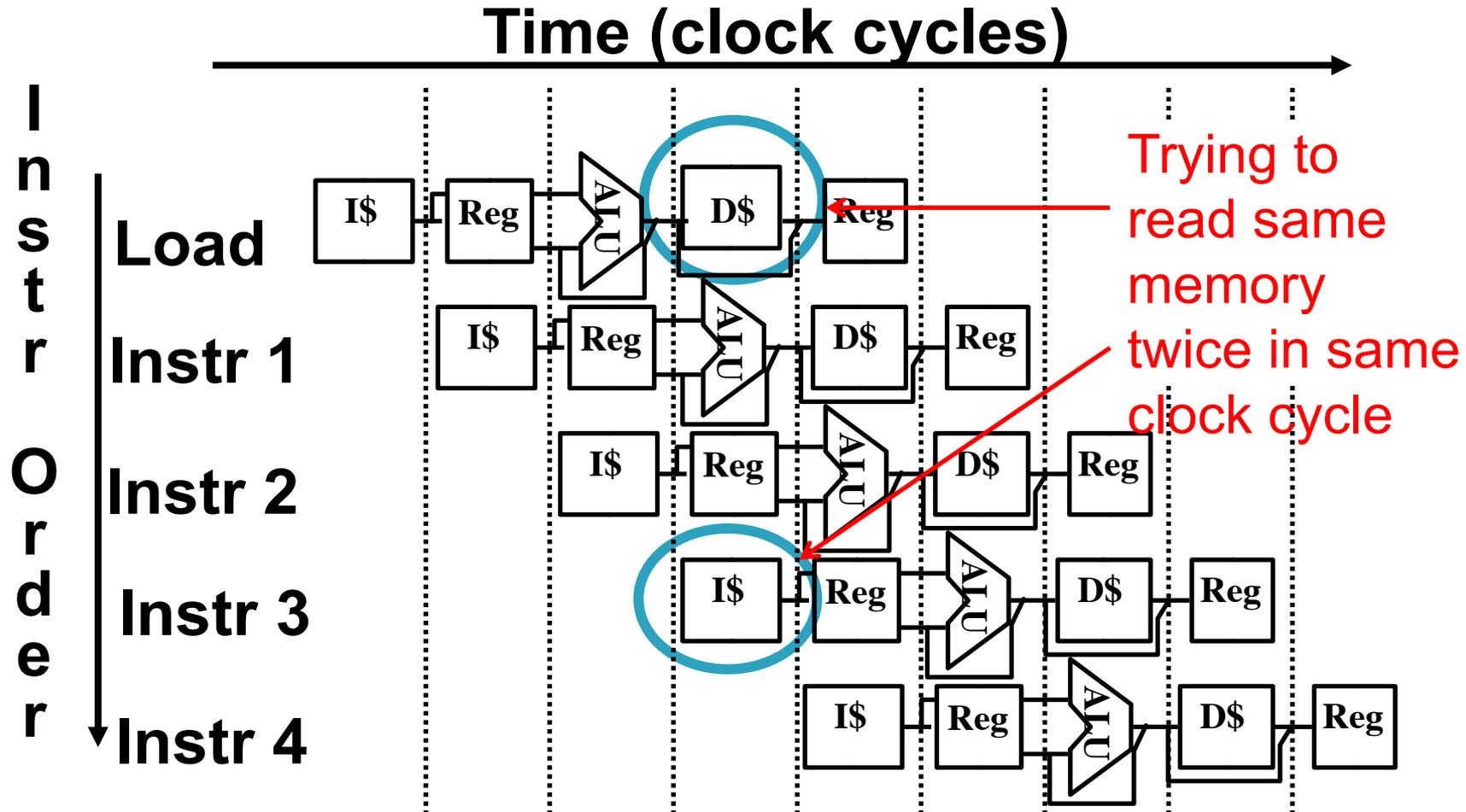   - Need to wait for previous instruction to complete its data read/write

3) *Control hazard*
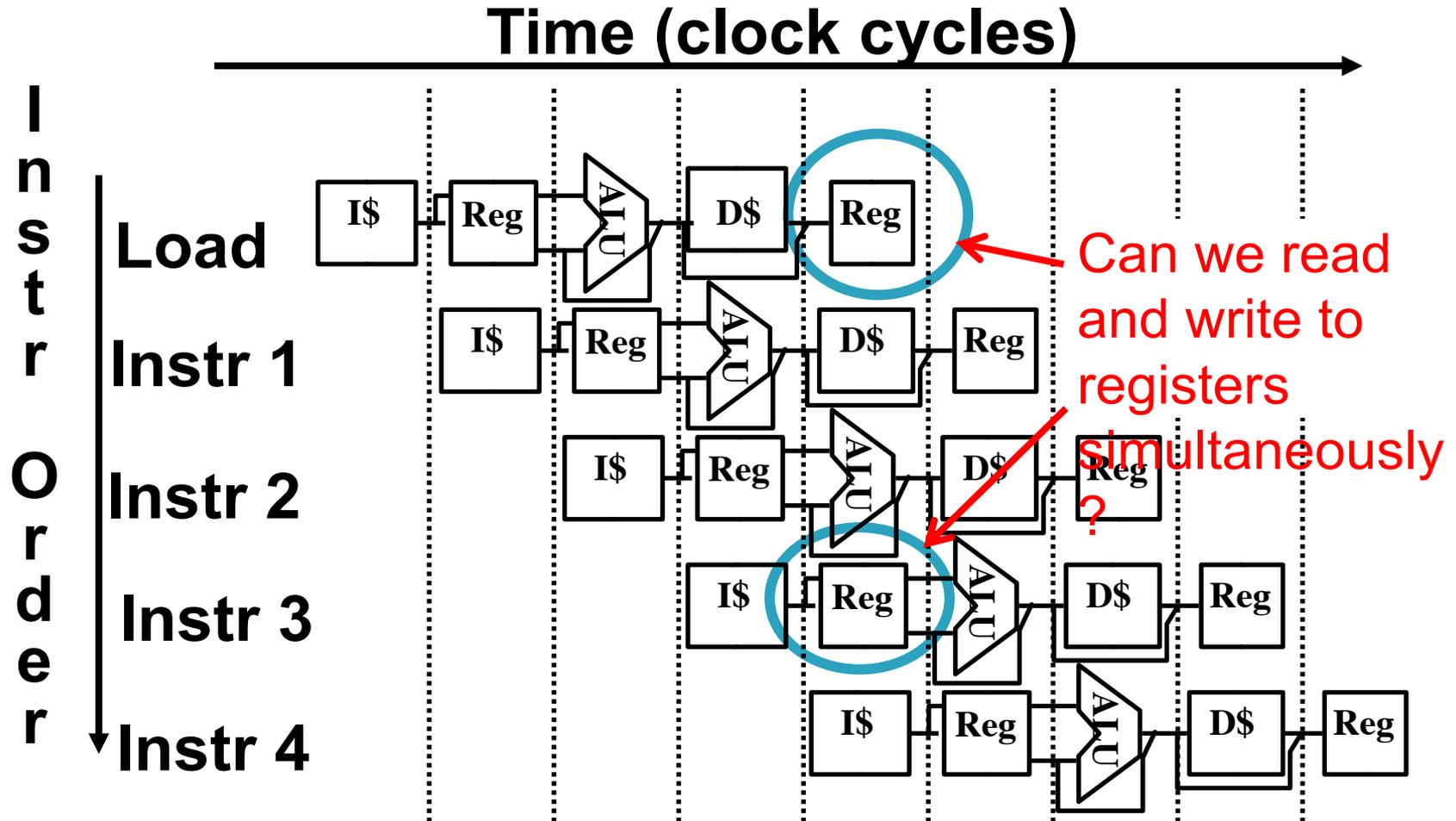   - Flow of execution depends on previous instruction

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

Time (clock cycles)

Instr Order

Load

Instr 1

Instr 2

Instr 3

Instr 4

Trying to read same memory twice in same clock cycle

Structural Hazard #2: Registers (1/2)

Time (clock cycles)

Instr Order

Load
Instr 1
Instr 2
Instr 3
Instr 4

Can we read and write to registers simultaneously?

Structural Hazard #2: Registers (2/2)

- 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:
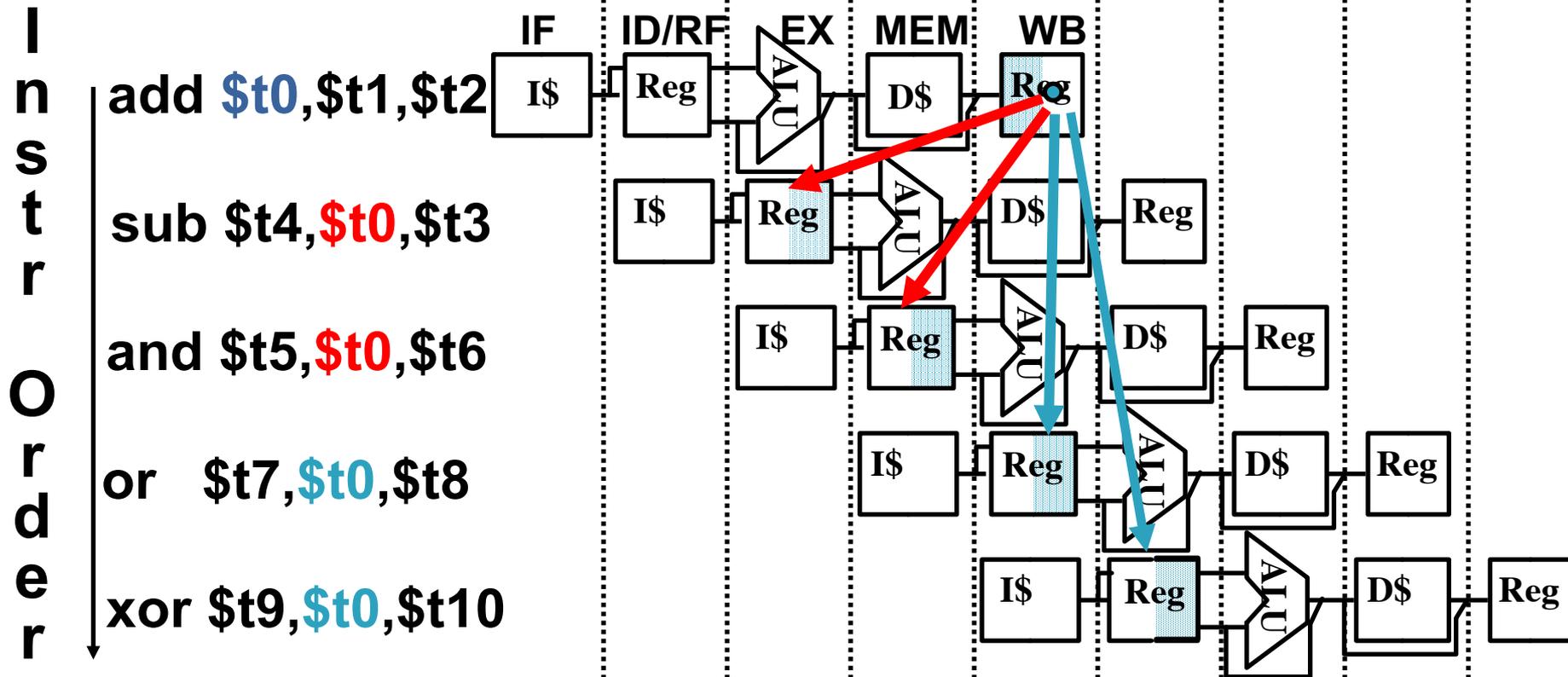
```
add $t0, $t1, $t2
sub $t4, $t0, $t3
and $t5, $t0, $t6
or  $t7, $t0, $t8
xor $t9, $t0, $t10
```
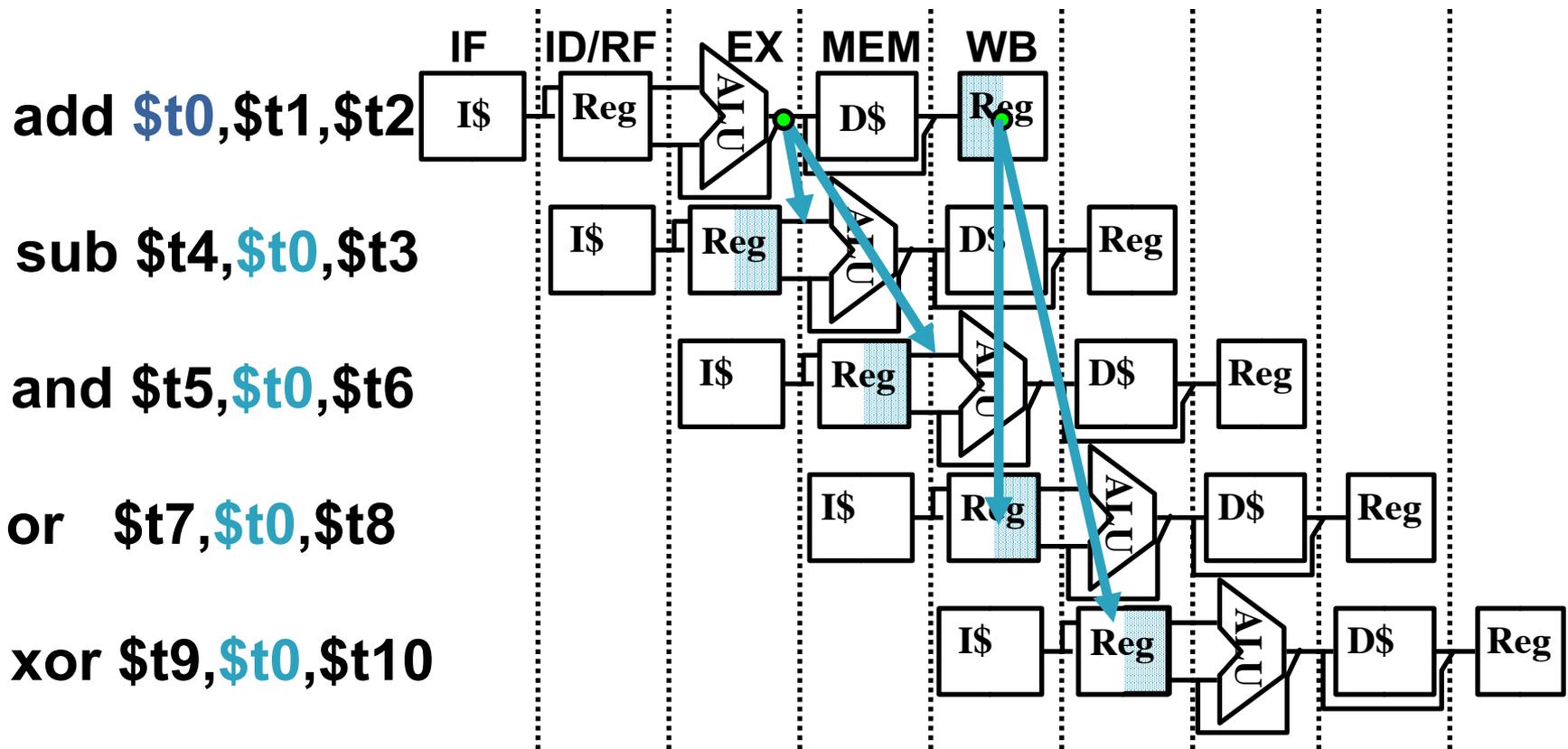
2. Data Hazards (2/2)

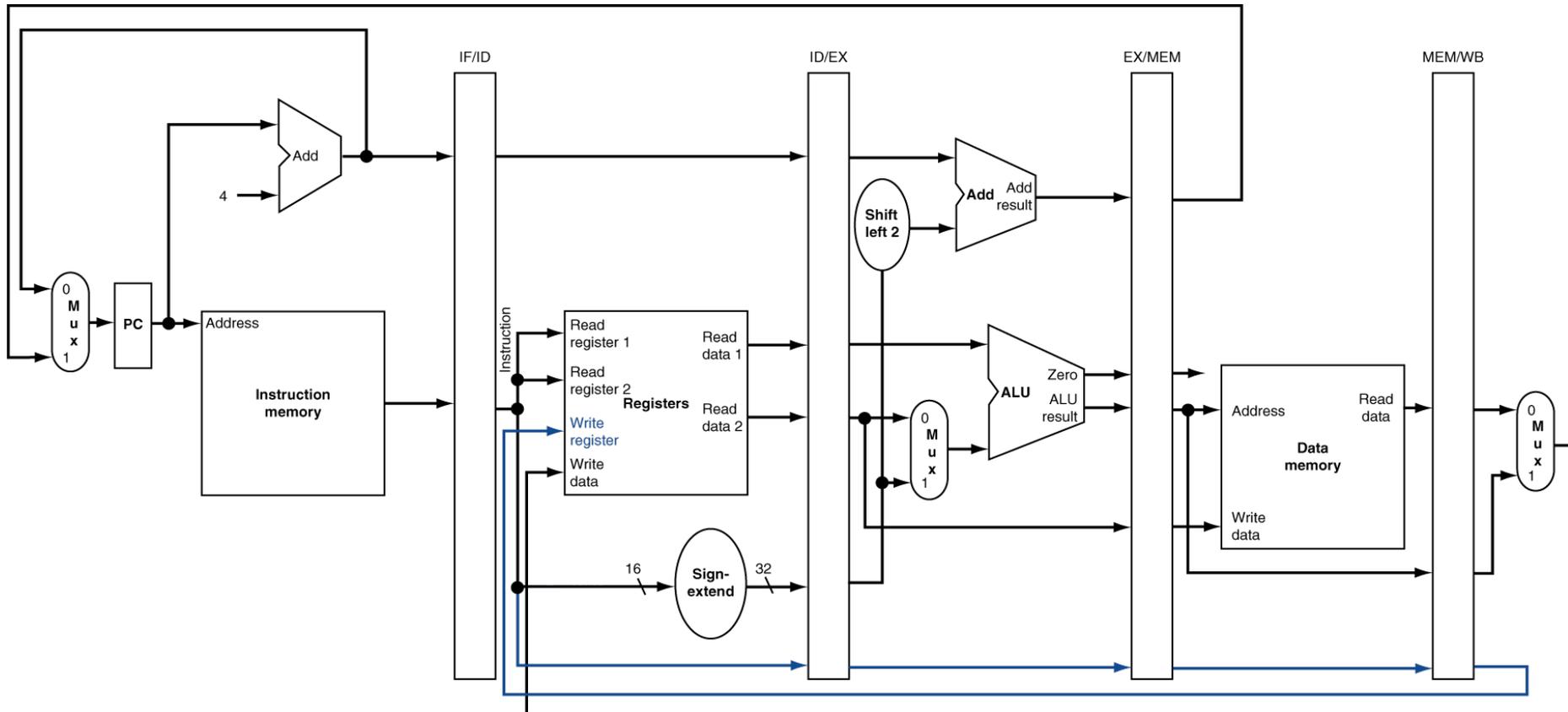- Data-flow *backwards* in time are hazards

Time (clock cycles)

Data Hazard Solution: Forwarding

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

add **$t0**,$t1,$t2

sub $t4,**$t0**,$t3

and $t5,**$t0**,$t6
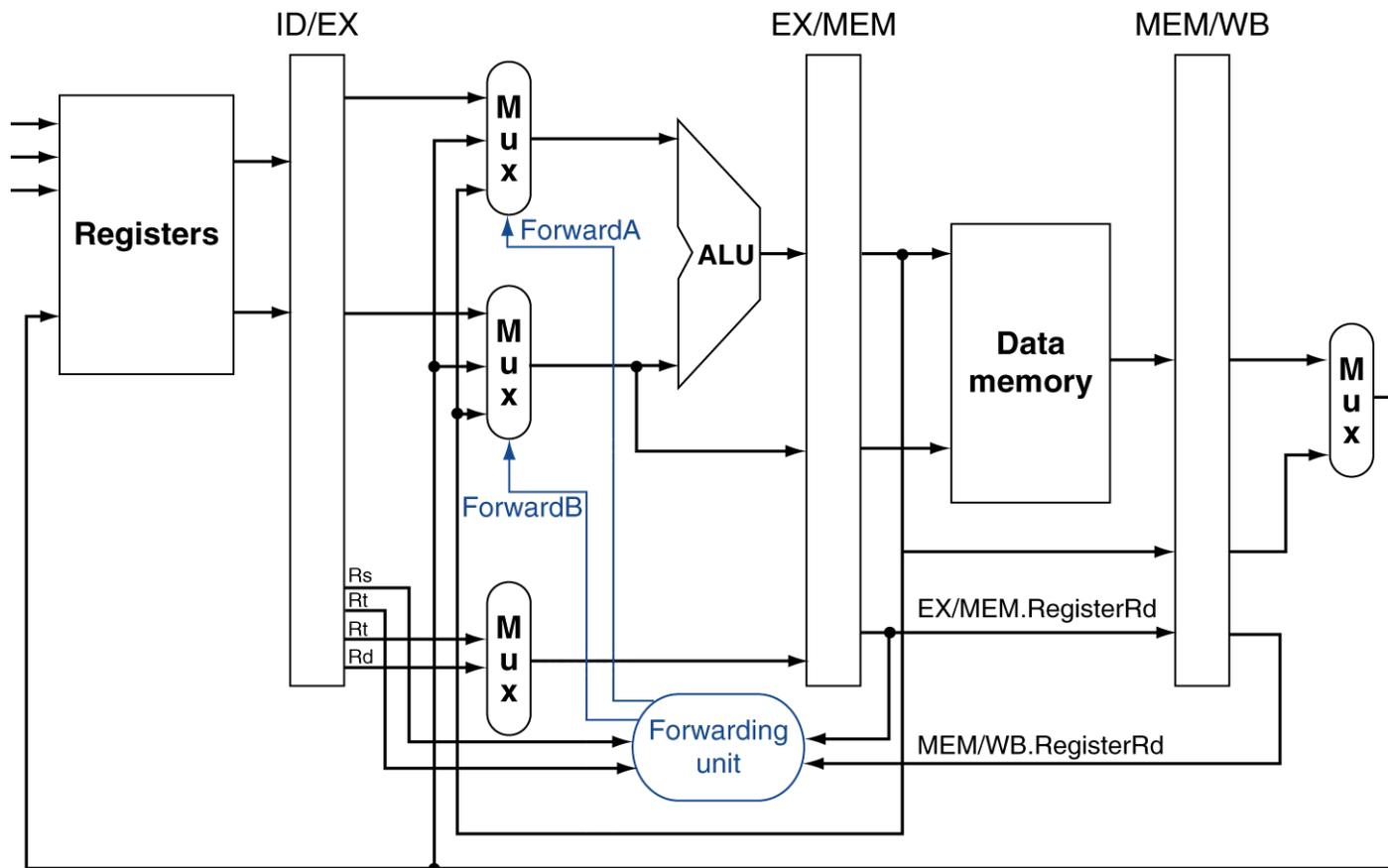
or $t7,**$t0**,$t8

xor $t9,**$t0**,$t10

Datapath for Forwarding (1/2)
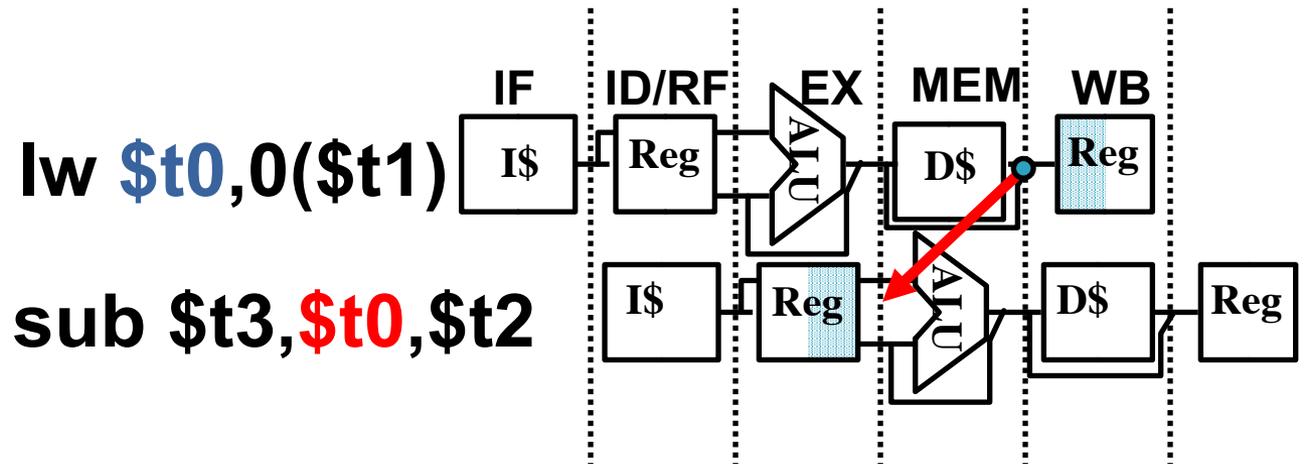
• What changes need to be made here?

Datapath for Forwarding (2/2)

- Handled by *forwarding unit*

Data Hazard: Loads (1/4)

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

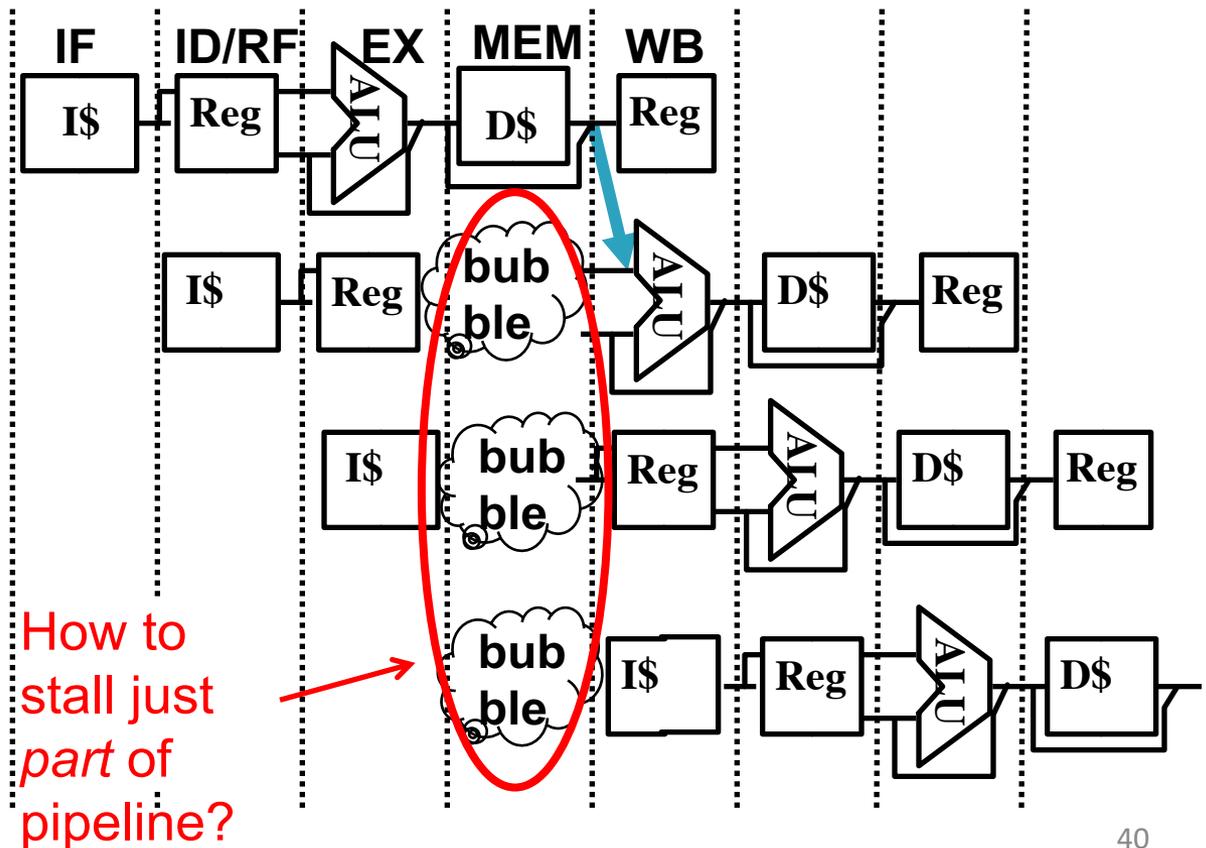
- Can't solve all cases with forwarding
  - Must *stall* instruction dependent on load, then forward (more hardware)

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

How to stall just *part* of pipeline?

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,$t

First two pipe stages stall by repeating stage one cycle later
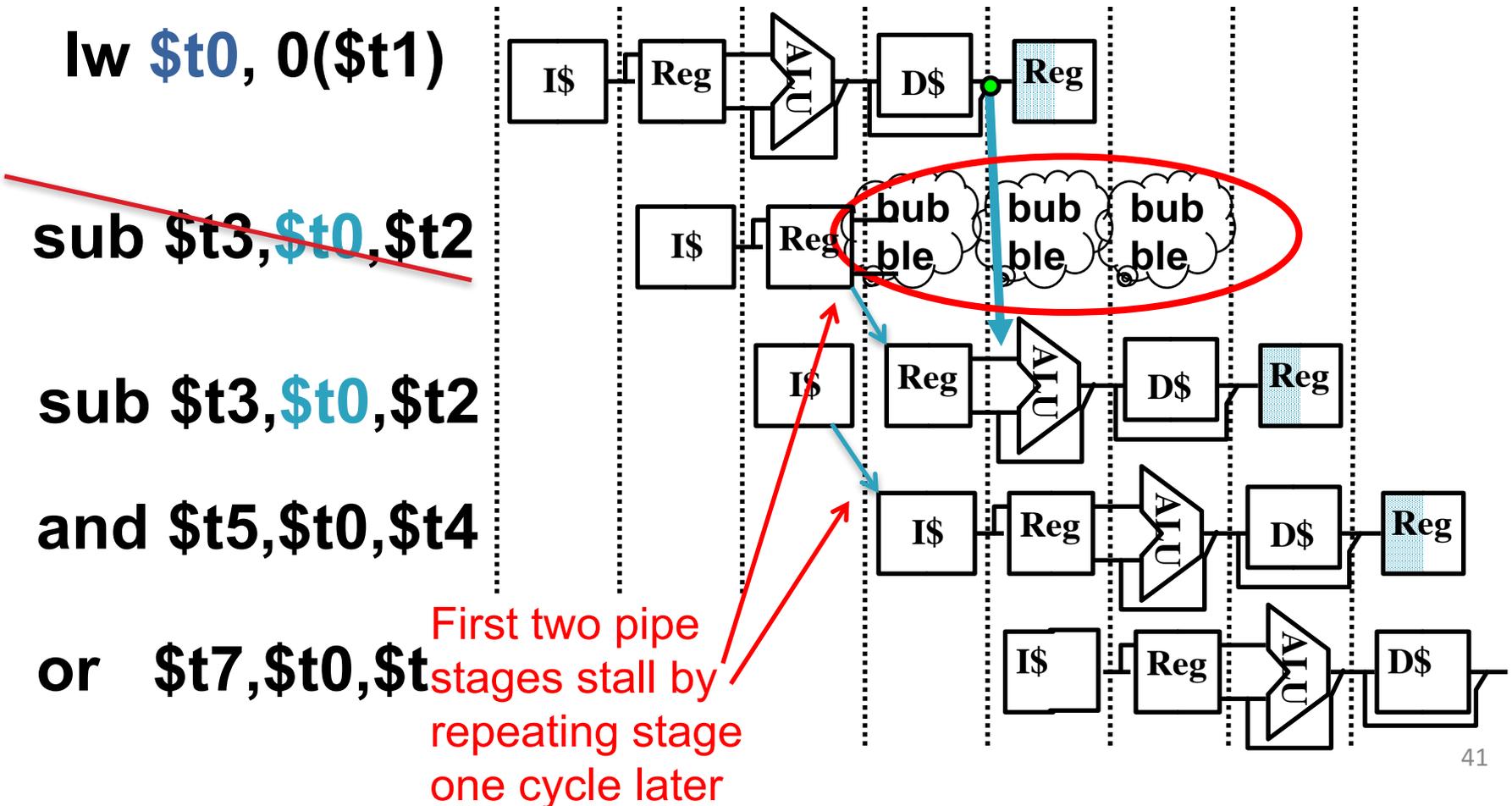
Data Hazard: Loads (4/4)

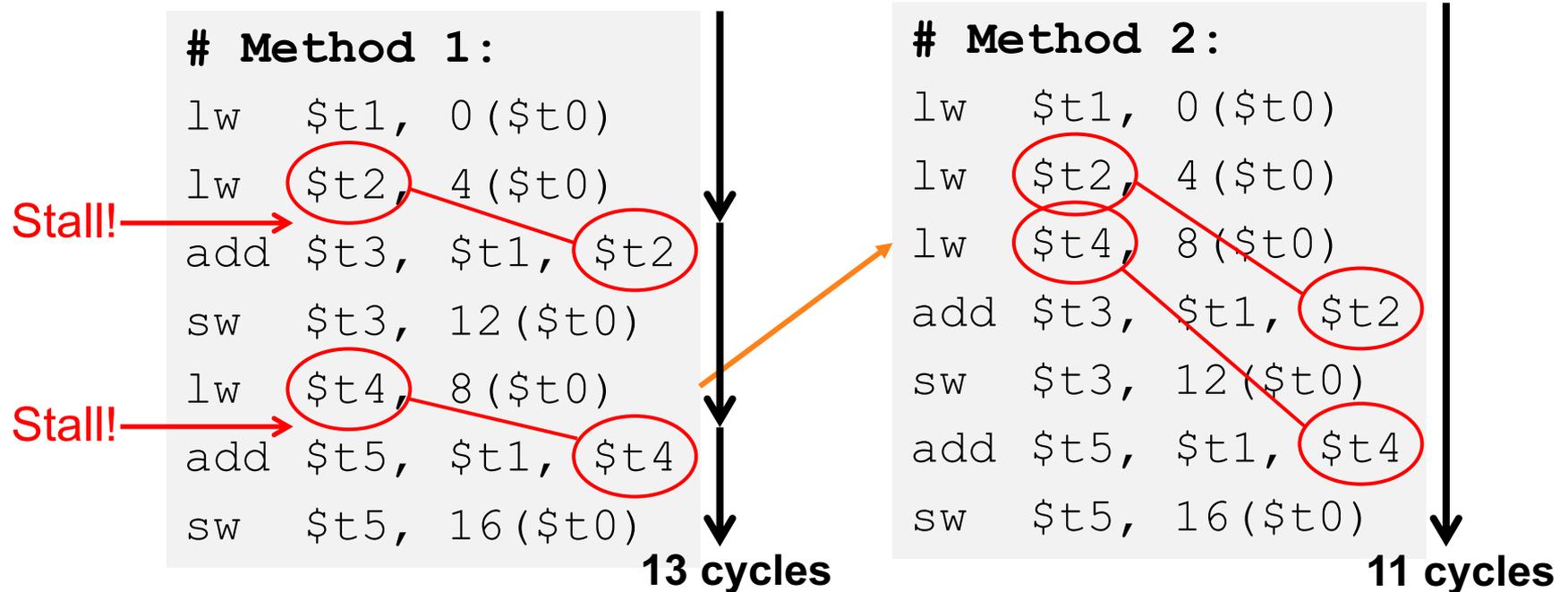- Slot after a load is called a *load delay slot*
  - If that instruction uses the result of the load, then the hardware interlock will stall it for one cycle
  - Letting the hardware stall the instruction in the delay slot is equivalent to putting an explicit `nop` in the slot (except the latter uses more code space)

- **Idea:** Let the compiler 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;`

```
# Method 1:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Stall!

Stall!

13 cycles

```
# Method 2:
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

11 cycles

43

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

Time (clock cycles)

Instr. Order

- beq
- Instr 1
- Instr 2
- Instr 3
- Instr 4
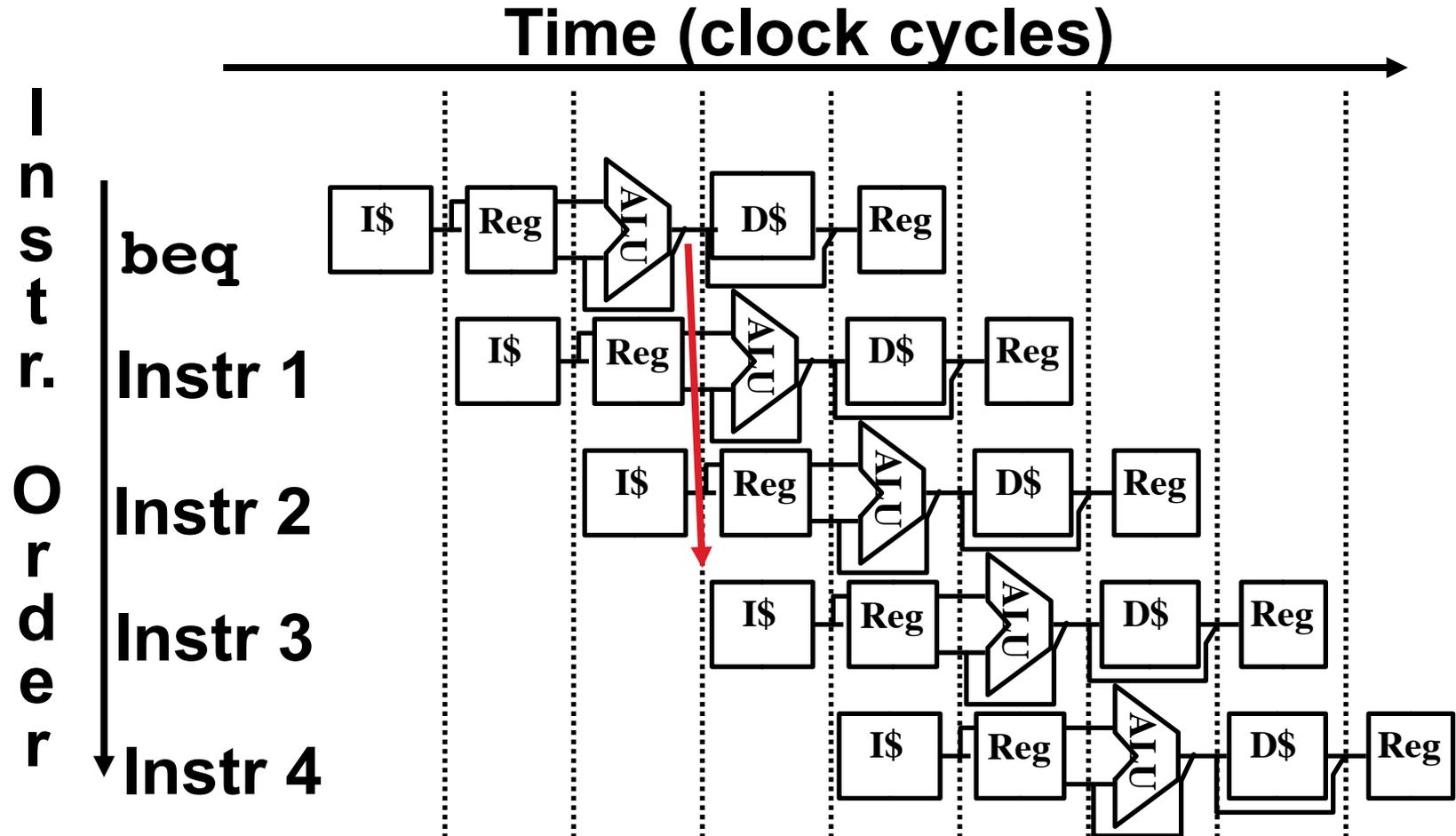
Where do we do the compare for the branch?

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

One Clock Cycle Stall

Time (clock cycles)

Instr. Order

beq

Instr 1

Instr 2
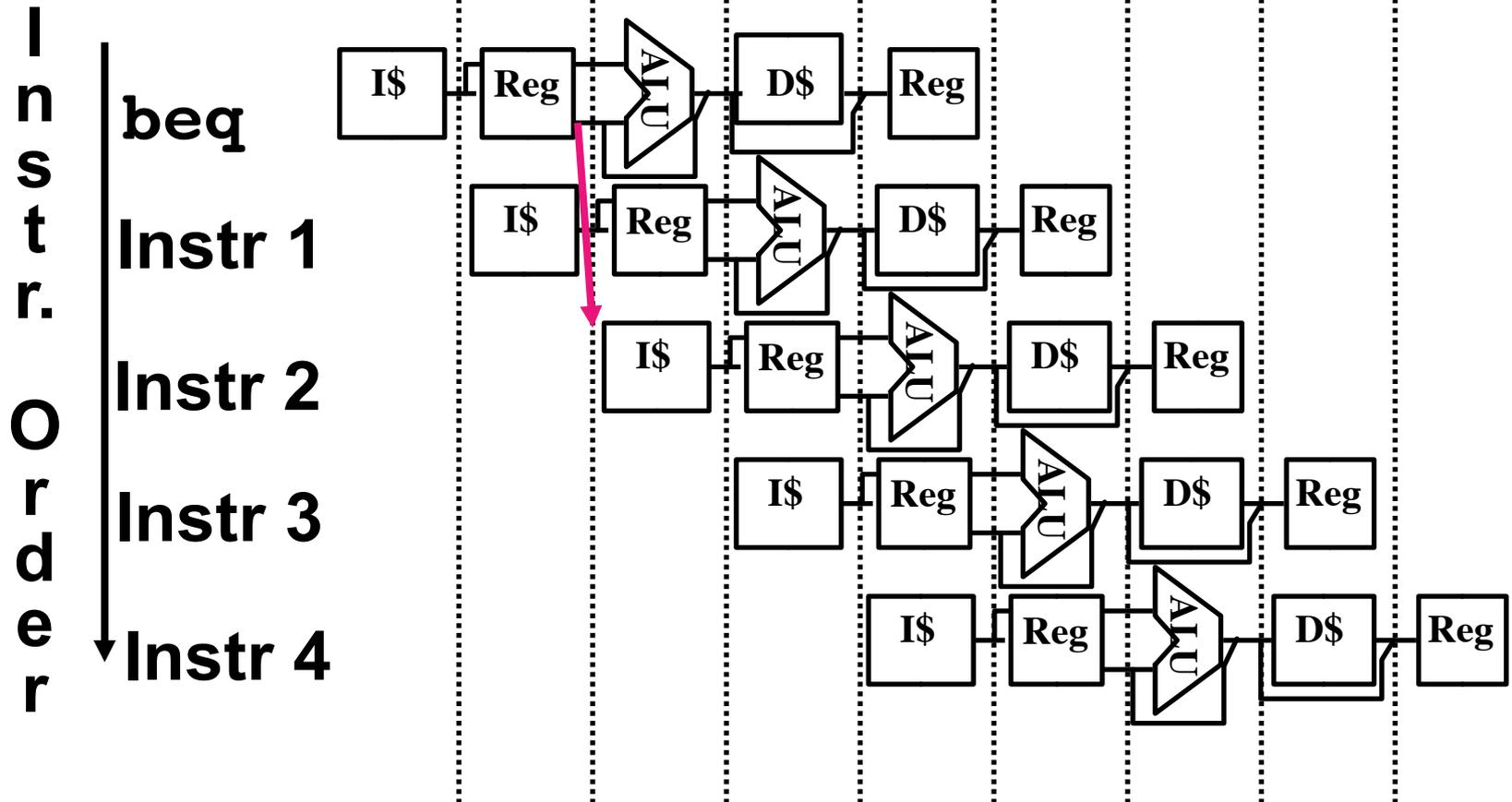
Instr 3

Instr 4

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
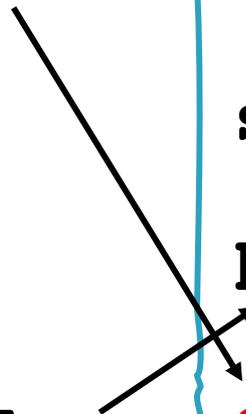
Nondelayed Branch

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 more than 70% of time
    - Jumps also have a delay slot …

Greater Instruction-Level Parallelism (ILP)

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage => shorter clock cycle
- Multiple issue "superscalar"
  - Replicate pipeline stages => 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

Hyperthreading
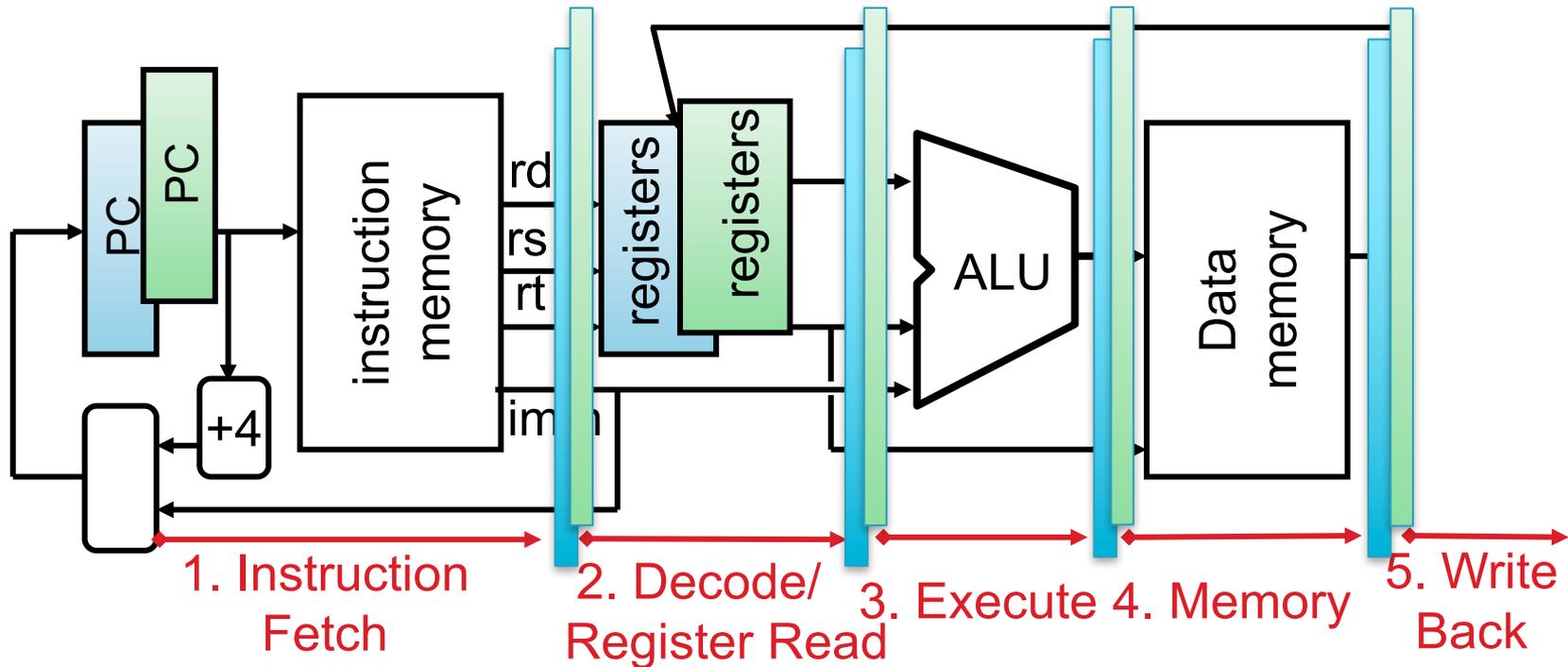
- Duplicate all elements that hold the state (registers)
- Use the same CL blocks
- Use muxes to select which state to use every clock cycle
- => run 2 totally independent threads (same memory -> shared memory!)
- Speedup?
  - No obvious speedup – make use of CL blocks in case of unavailable resources (e.g. wait for memory)

53

Intel Nehalem i7

- Hyperthreading:
  - About 5% die area
  - Up to 30% speed gain (BUT also < 0% possible)
- Pipeline: 20-24 stages!
- Out-of-order execution
  1. Instruction fetch.
  2. Instruction dispatch to an instruction queue
  3. Instruction: Wait in queue until input operands are available => instruction can **leave queue before earlier**, older instructions.
  4. The instruction is issued to the appropriate functional unit and executed by that unit.
  5. The results are queued.
  6. Write to register only after all older instructions have their results written.

GT/s: gigatransfers per second
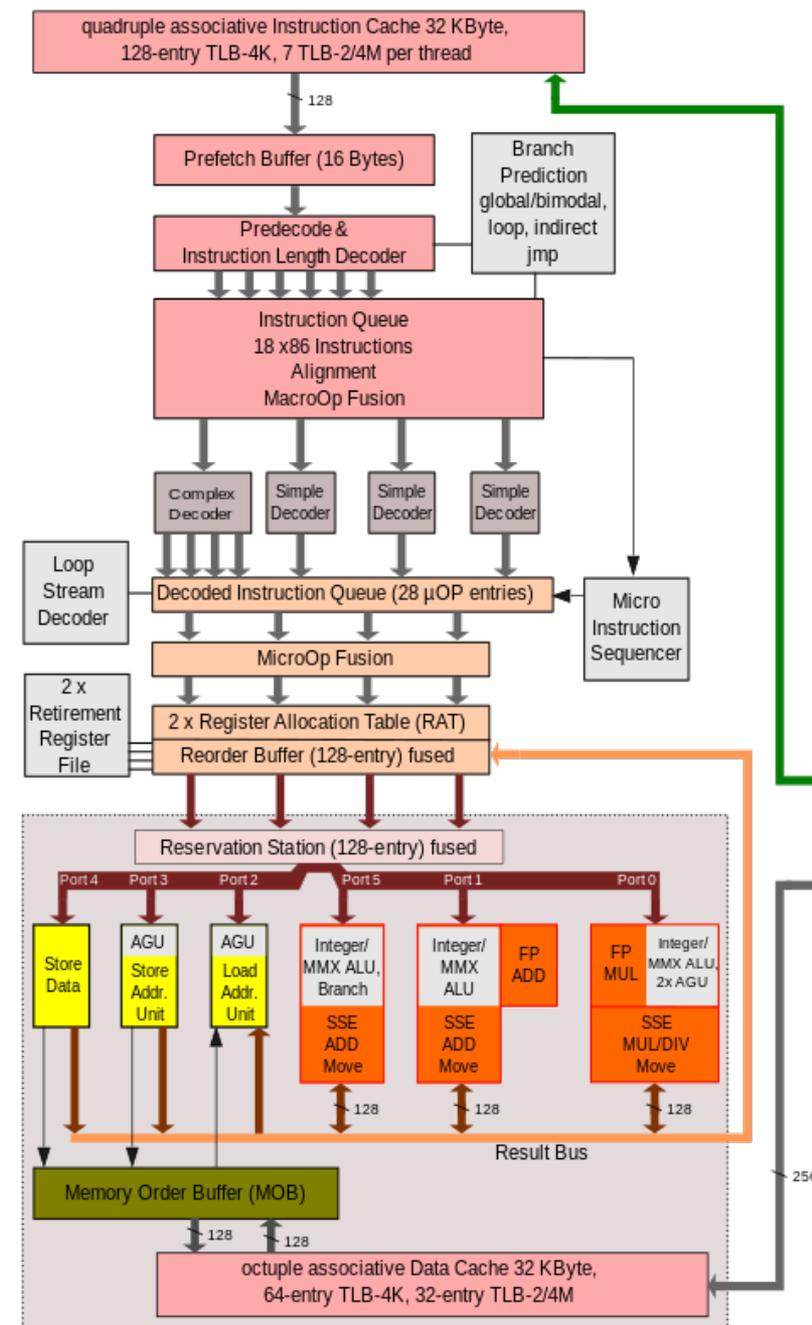
In Conclusion

- Pipelining increases throughput by overlapping execution of multiple instructions in different pipestages

- Pipestages 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)