Computer Architecture

Lecture 6: Pipelining

Dr. Ahmed Sallam

Suez Canal University

Based on original slides by Prof. Onur Mutlu

Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...

Recap of Last Lecture

- Multi-cycle and Microprogrammed Microarchitectures
  - Benefits vs. Design Principles
  - When to Generate Control Signals
  - Microprogrammed Control: uInstruction, uSequencer, Control Store
  - LC-3b State Machine, Datapath, Control Structure
  - An Exercise in Microprogramming
  - Variable Latency Memory, Alignment, Memory Mapped I/O, ...

- Microprogramming
  - Power of abstraction (for the HW designer)
  - Advantages of uProgrammed Control
  - Update of Machine Behavior

Review: A Simple LC-3b Control and Datapath

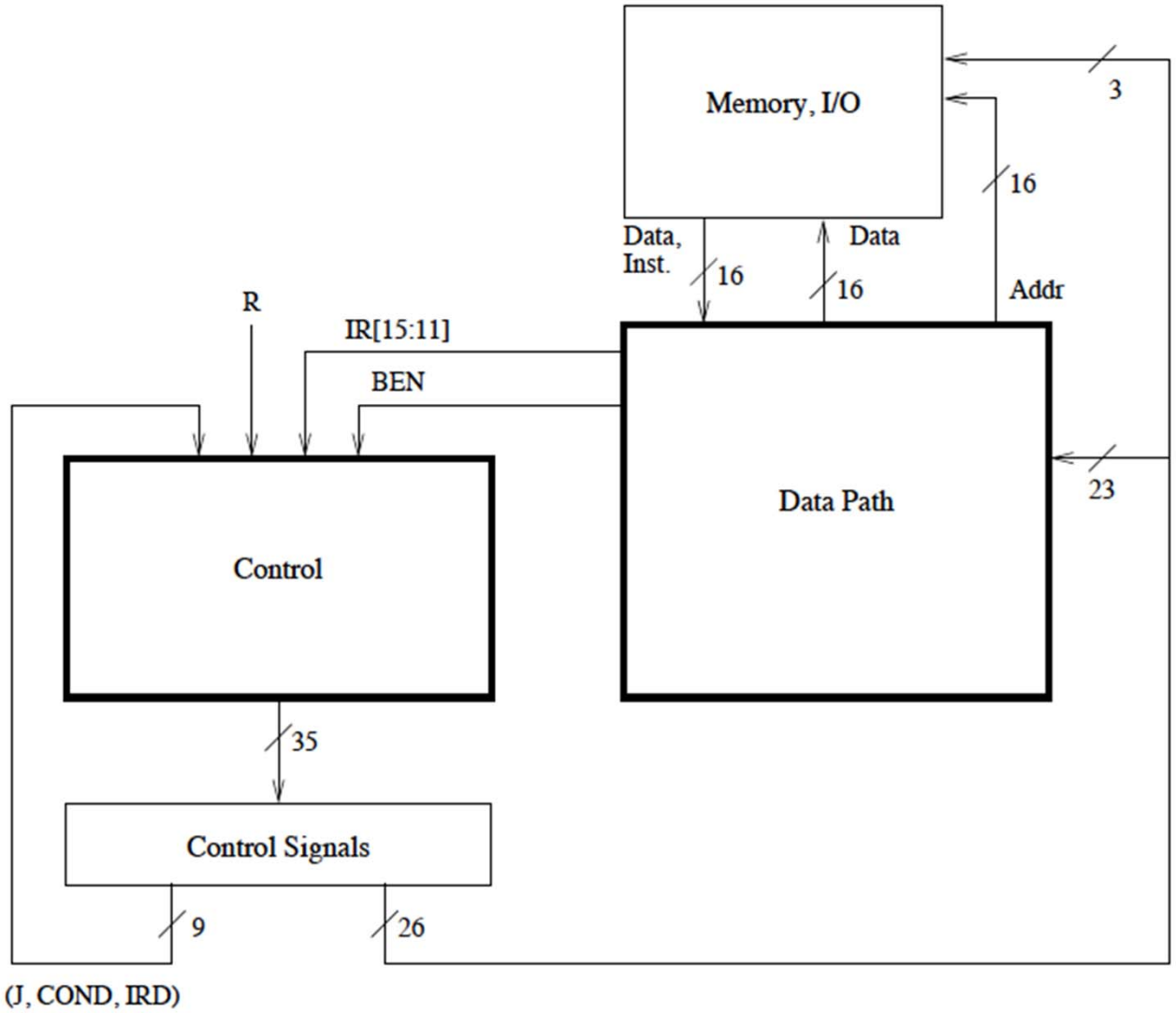
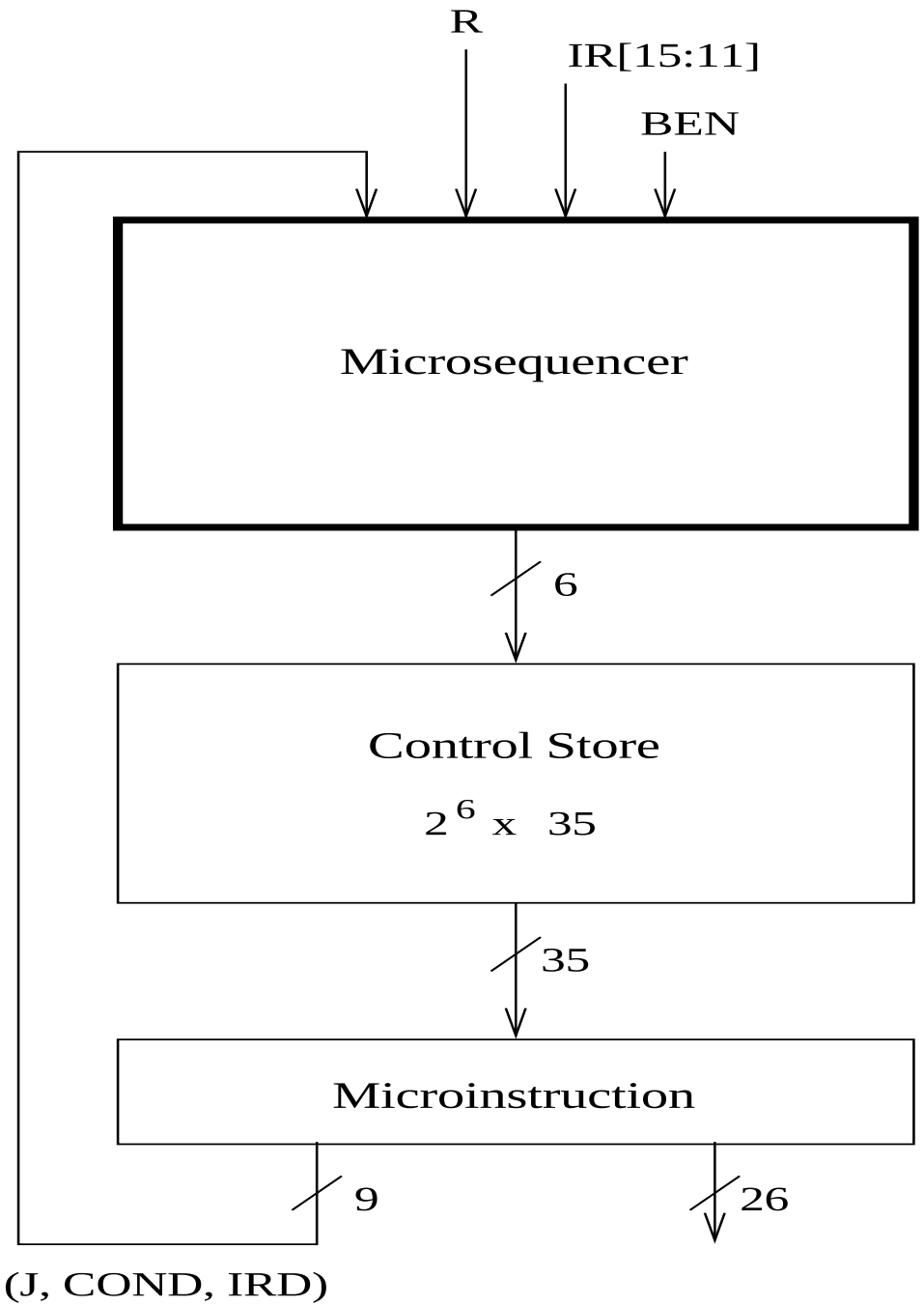
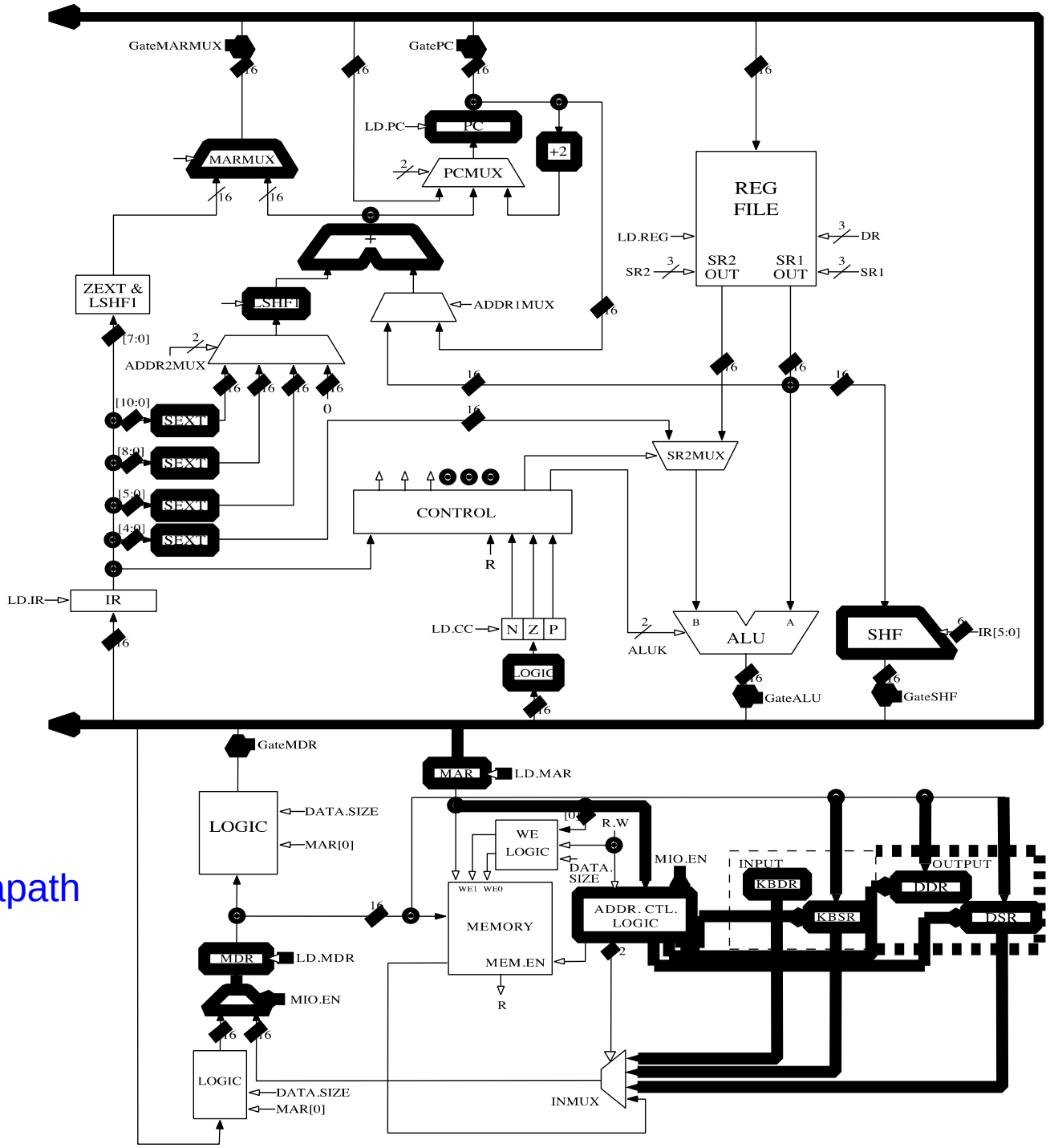


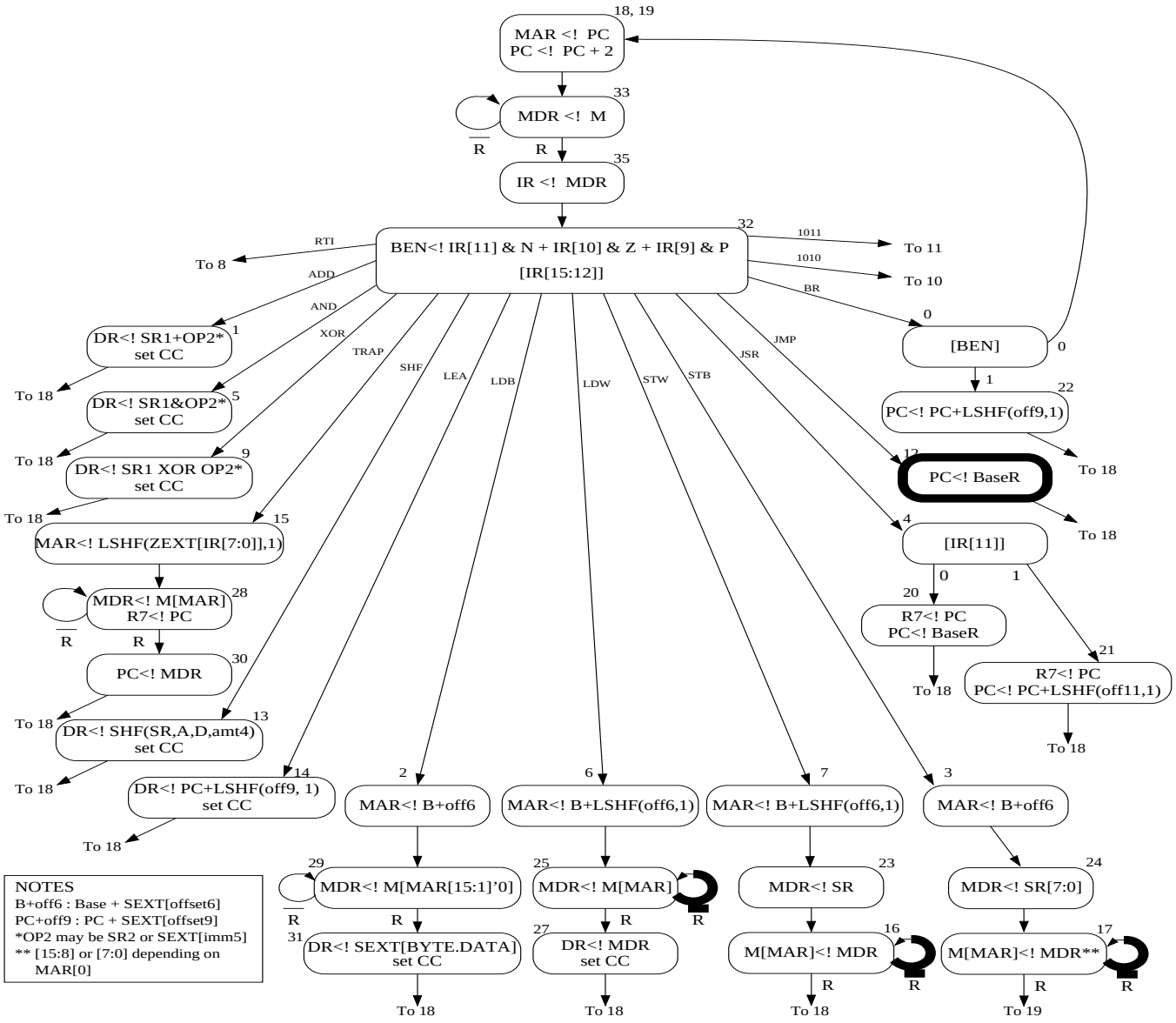
Figure C.1: Microarchitecture of the LC-3b, major components



Simple Design
of the Control Structure

A Simple Datapath
Can Become
Very Powerful




18, 19
MAR <! PC
PC <! PC + 2
33
MDR <! M
R
R
35
IR <! MDR
32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP
1011
To 11
1010
To 10
BR
1
DR<! SR1+OP2*
set CC
To 18
5
DR<! SR1&OP2*
set CC
To 18
9
DR<! SR1 XOR OP2*
set CC
To 18
15
MAR<! LSHF(ZEXT[IR[7:0]],1)
28
MDR<! M[MAR]
R7<! PC
R
R
30
PC<! MDR
To 18
13
DR<! SHF(SR,A,D,amt4)
set CC
To 18
14
DR<! PC+LSHF(off9, 1)
set CC
To 18
0
[BEN]
0
1
22
PC<! PC+LSHF(off9,1)
To 18
12
PC<! BaseR
To 18
4
[IR[11]]
0
1
20
R7<! PC
PC<! BaseR
To 18
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18
2
MAR<! B+off6
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
6
MAR<! B+LSHF(off6,1)
25
MDR<! M[MAR]
R
R
27
DR<! MDR
set CC
To 18
7
MAR<! B+LSHF(off6,1)
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
3
MAR<! B+off6
24
MDR<! SR[7:0]
17
M[MAR]<! MDR**
R
R
To 19
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

Review: The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: microprogramming

- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
  - The sequence of microinstructions needed to implement the desired operation
  - The ability for the control logic to correctly sequence through the microinstructions
  - Any additional datapath elements and control signals needed (no need if the operation can be "translated" into existing control signals)

Review: Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of u-instructions)
  - Microcode (u-code) enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of as a user-invisible ISA (u-ISA)
- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the microcode
  - Can support complex instructions as a sequence of simple microinstructions
- Enables update of machine behavior
  - A buggy implementation of an instruction can be fixed by changing the microcode in the field

Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- You should be very familiar with this right now

Microprogrammed vs. Hardwired Control

- Advantages
- Disadvantages
- You should be very familiar with this right now

Can We Do Better?

- What limitations do you see with the multi-cycle design?

- Limited concurrency
  - Some hardware resources are idle during different phases of instruction processing cycle
  - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
  - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: More concurrency → Higher instruction throughput (i.e., more "work" completed in one cycle)

- Idea: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

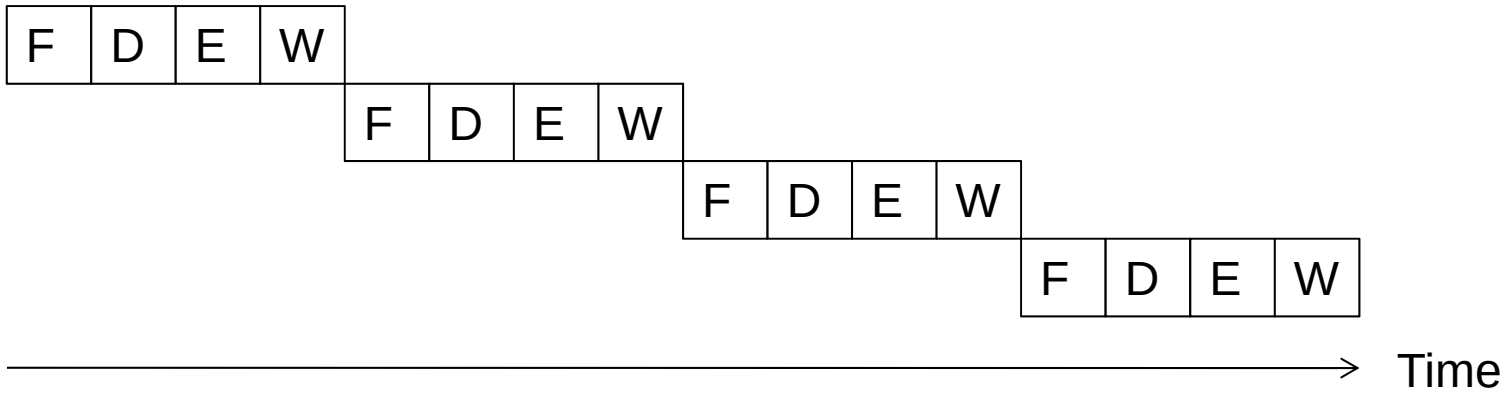
Pipelining

Pipelining: Basic Idea

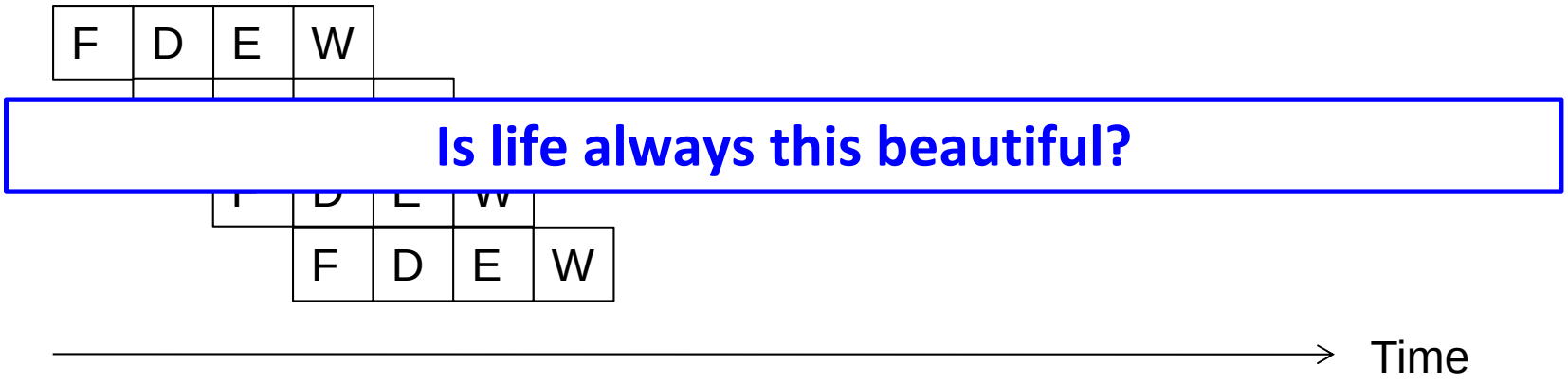
- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: “Assembly line processing” of instructions
- Idea:
  - Divide the instruction processing cycle into distinct “stages” of processing
  - Ensure there are enough hardware resources to process one instruction in each stage
  - Process a different instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this...

Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

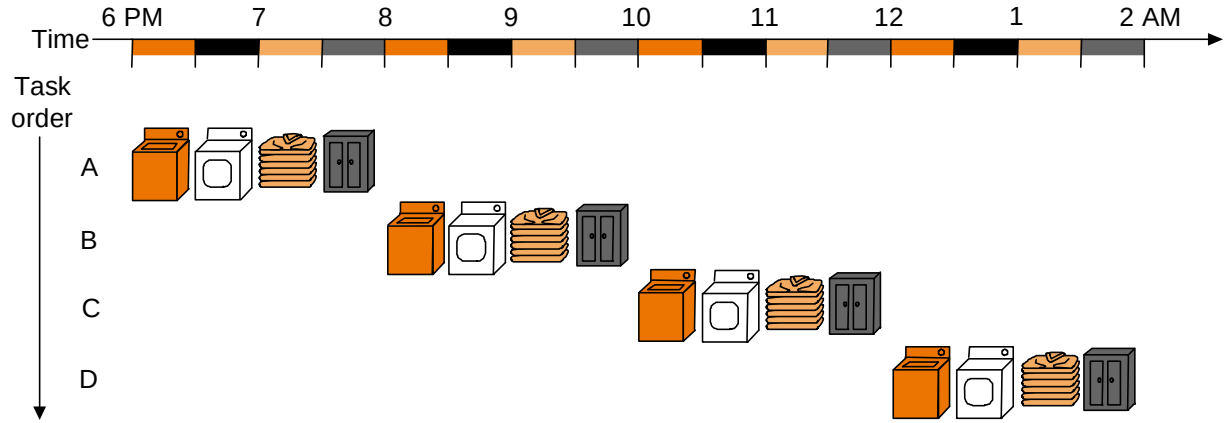


- Pipelined: 4 cycles per 4 instructions (steady state)



UNDERSTANDING PIPELINE

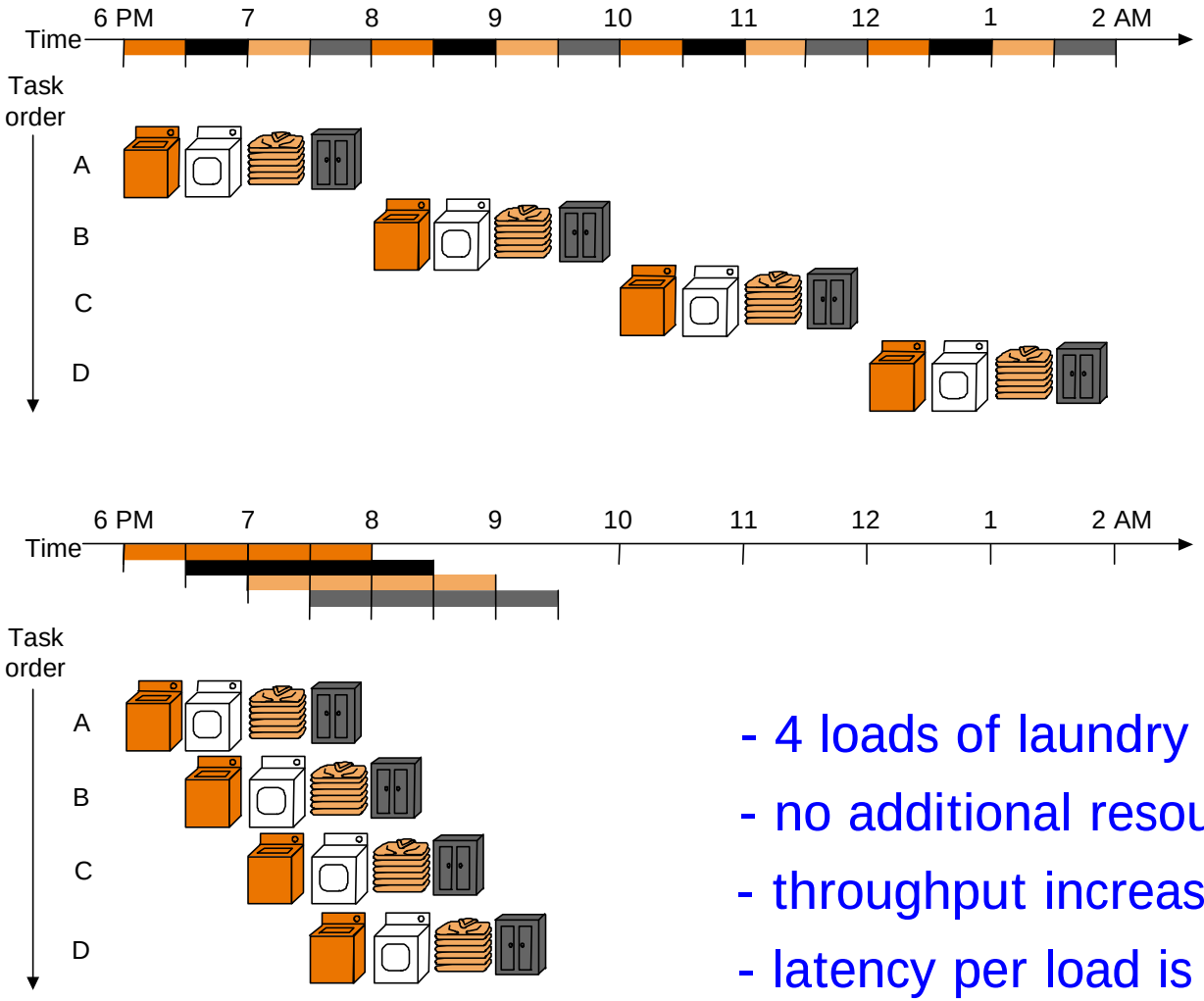
The Laundry Analogy



- "place one dirty load of clothes in the washer"
- "when the washer is finished, place the wet load in the dryer"
- "when the dryer is finished, take out the dry load and fold"
- "when folding is finished, ask your roommate (??) to put the clothes away"

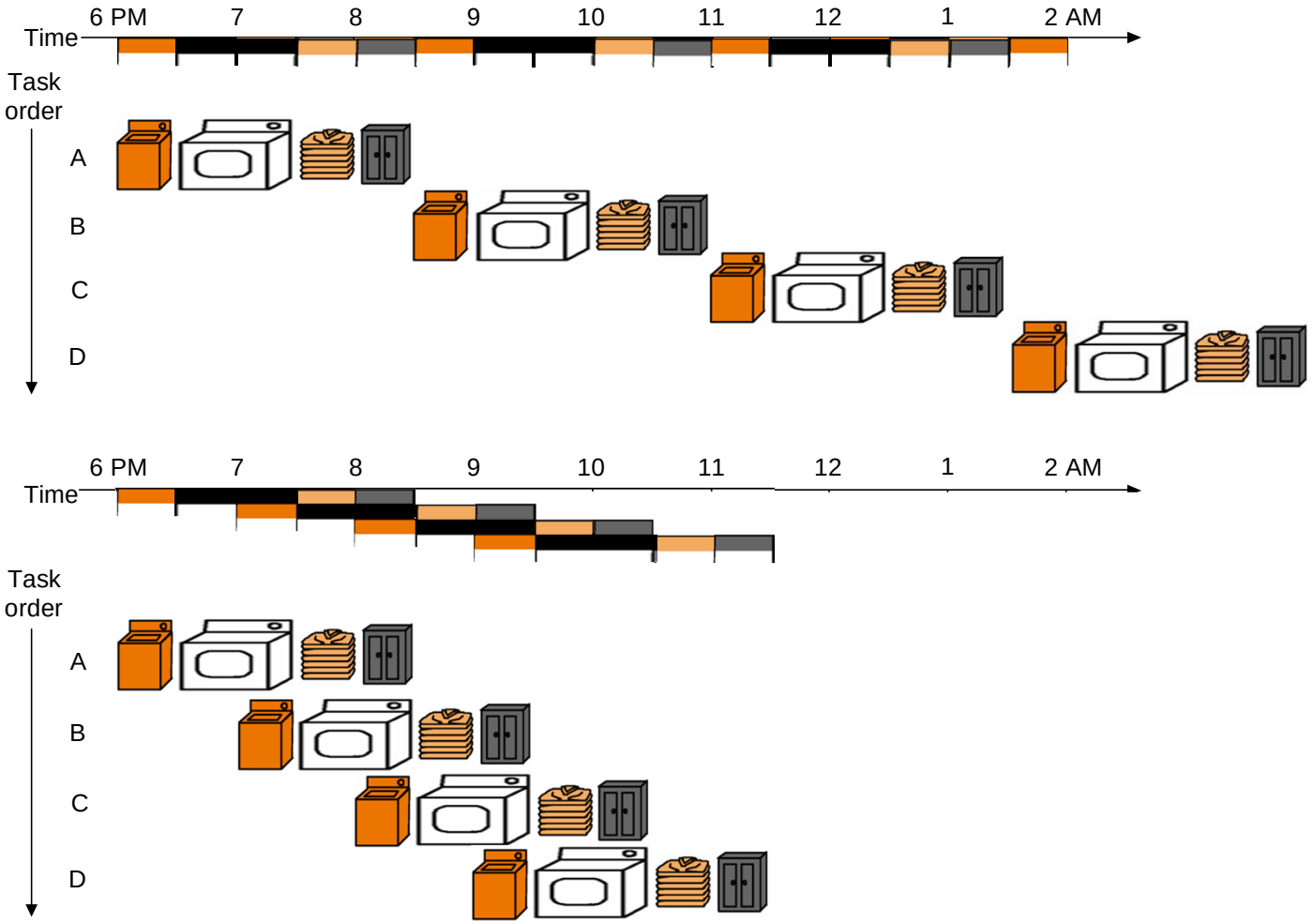
  - steps to do a load are sequentially dependent
  - no dependence between different loads
  - different steps do not share resources

Pipelining Multiple Loads of Laundry



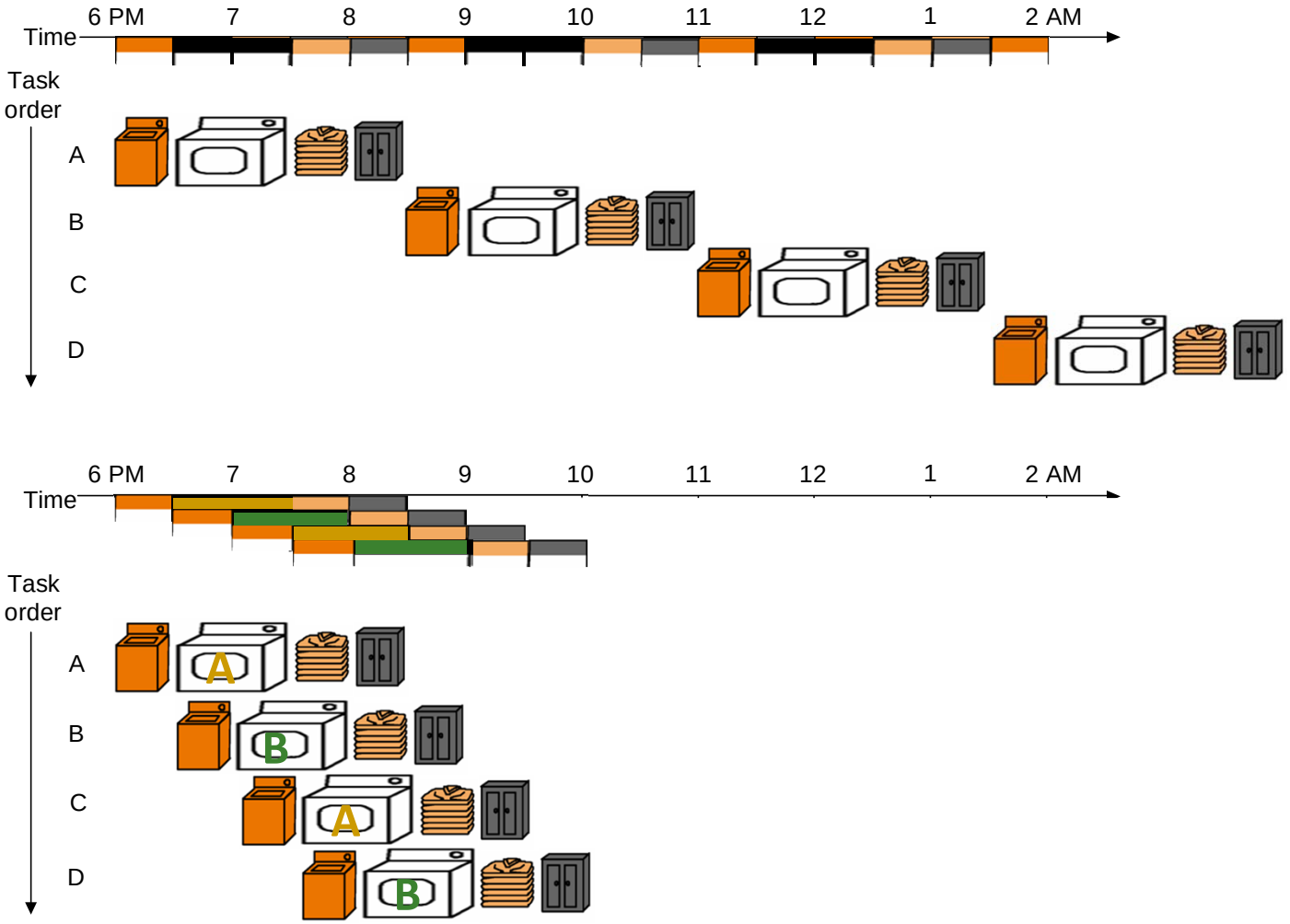
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice



throughput restored (2 loads per hour) using 2 dryers

PERFORMING PIPELINE

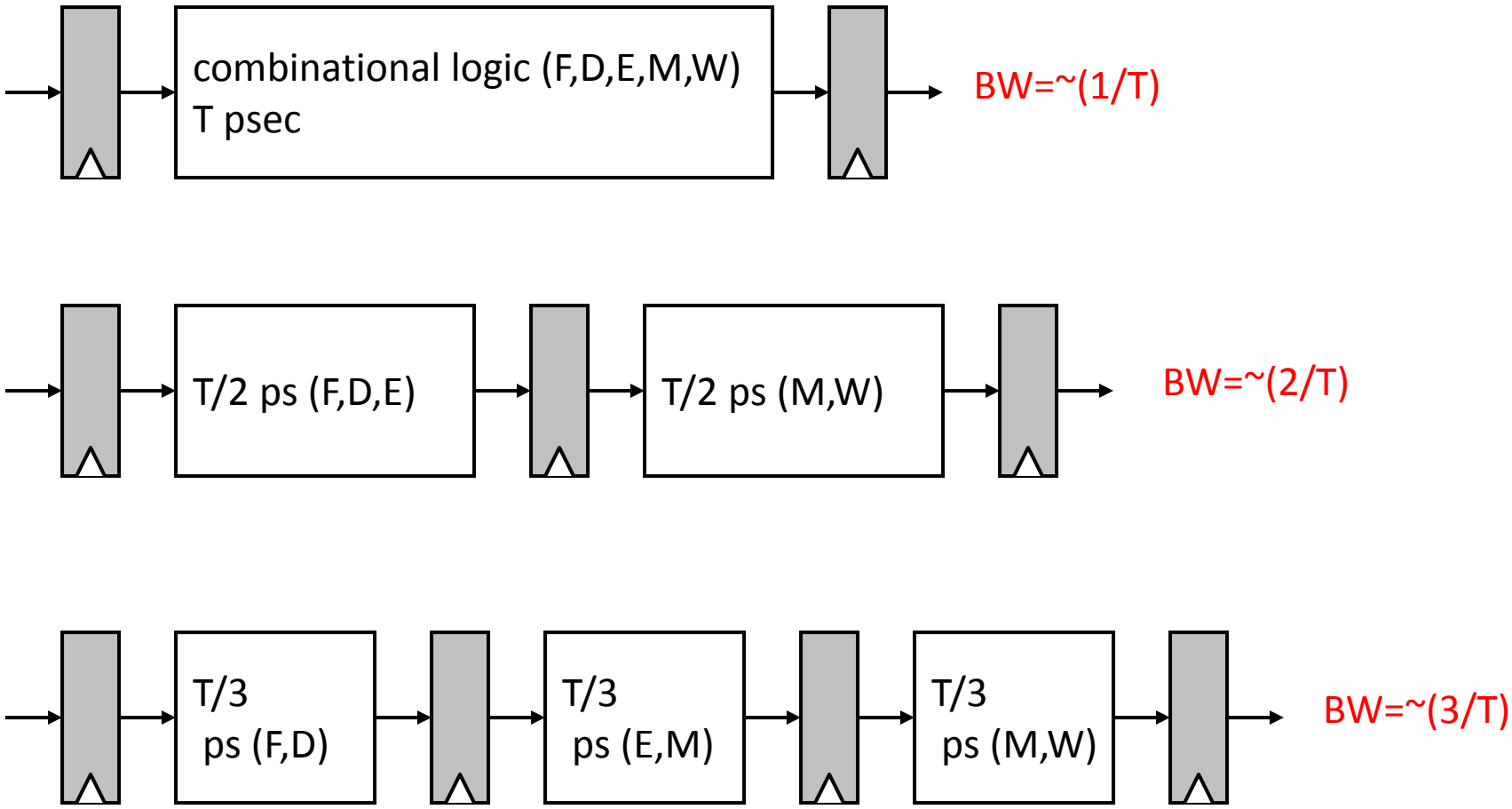
An Ideal Pipeline

- Goal: Increase throughput with little increase in cost
  (hardware cost, in case of instruction processing)

- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)

- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

Ideal Pipelining



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

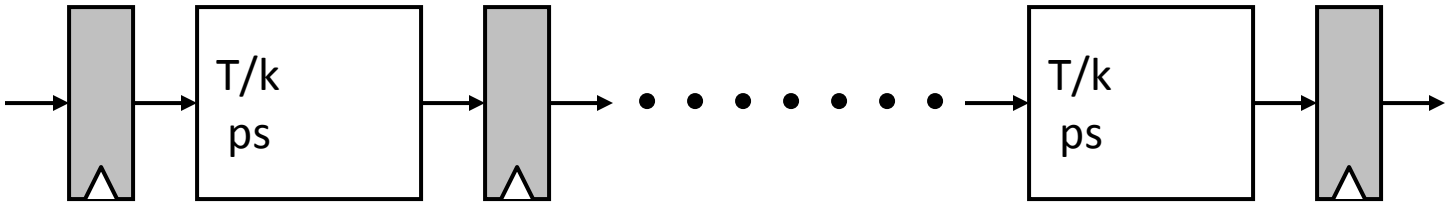
  BW = 1/(T+S) where S = latch delay

- k-stage pipelined version

  $BW_{k\text{-stage}} = 1 / (T/k + S)$

  $BW_{max} = 1 / (1 \text{ gate delay} + S)$

Latch delay reduces throughput (switching overhead b/w stages)



More Realistic Pipeline: Cost

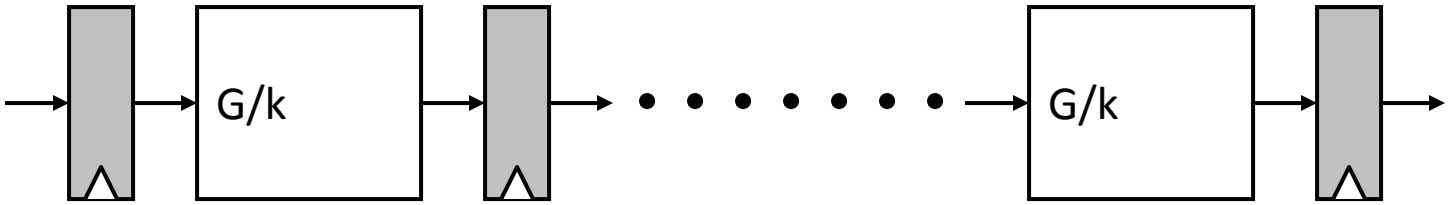
- Nonpipelined version with combinational cost G

  Cost = G+L where L = latch cost

- k-stage pipelined version

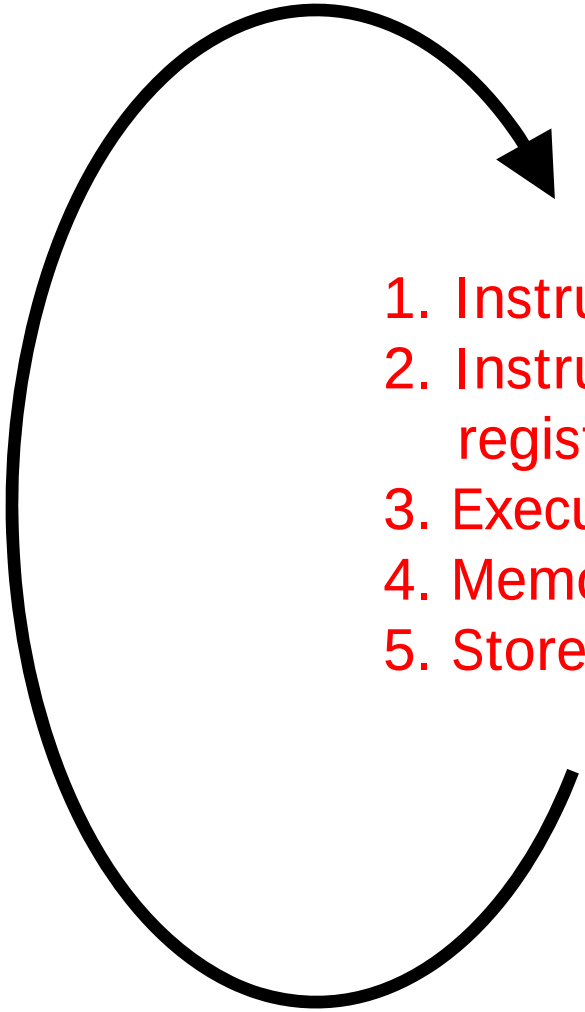
  $Cost_{k\text{-stage}} = G + Lk$

Latches increase hardware cost



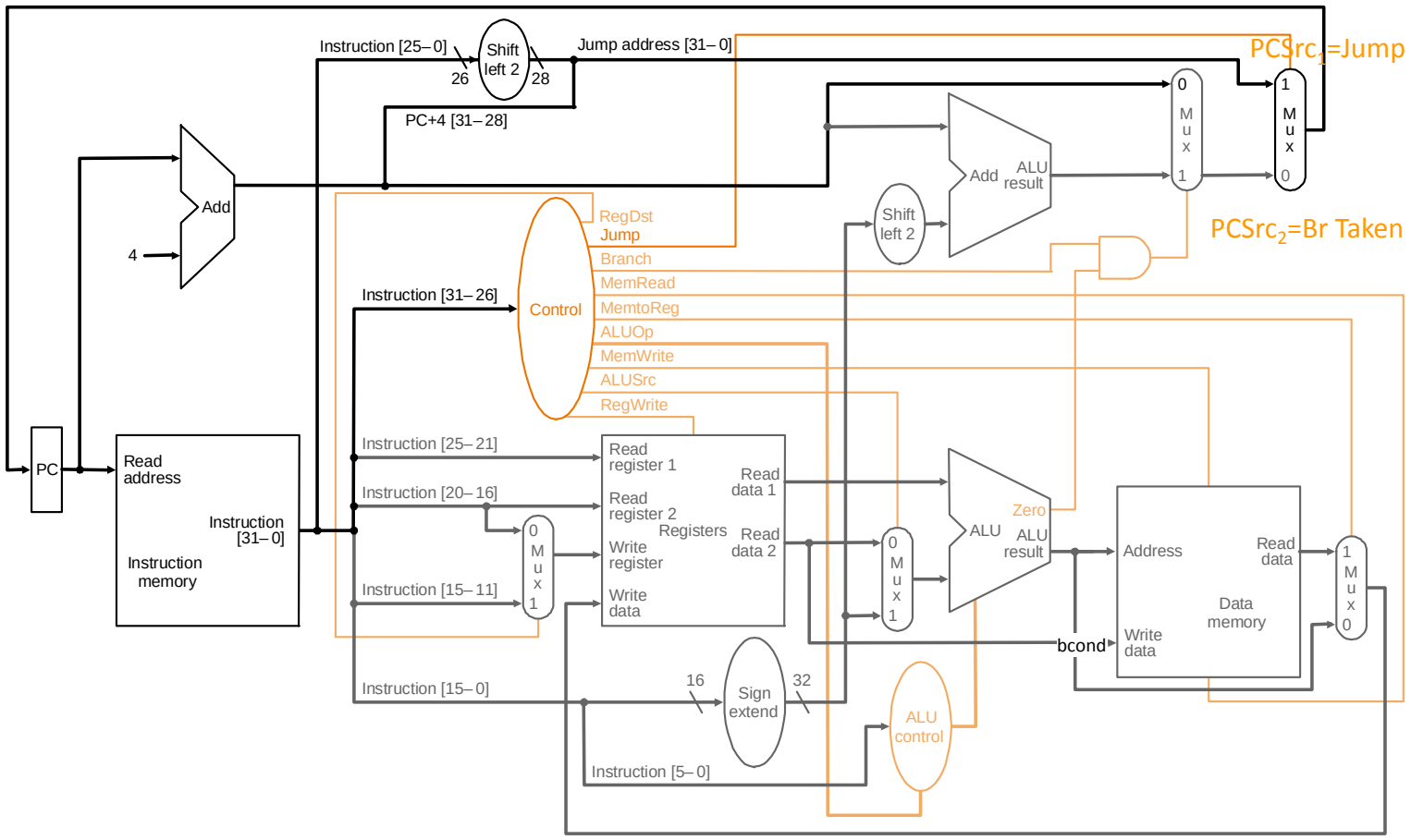
Pipelining Instruction Processing

Remember: The Instruction Processing Cycle



1. Instruction fetch (IF)
2. Instruction decode and
   register operand fetch (ID/RF)
3. Execute/Evaluate memory address (EX/AG)
4. Memory operand fetch (MEM)
5. Store/writeback result (WB)

Remember the Single-Cycle Uarch

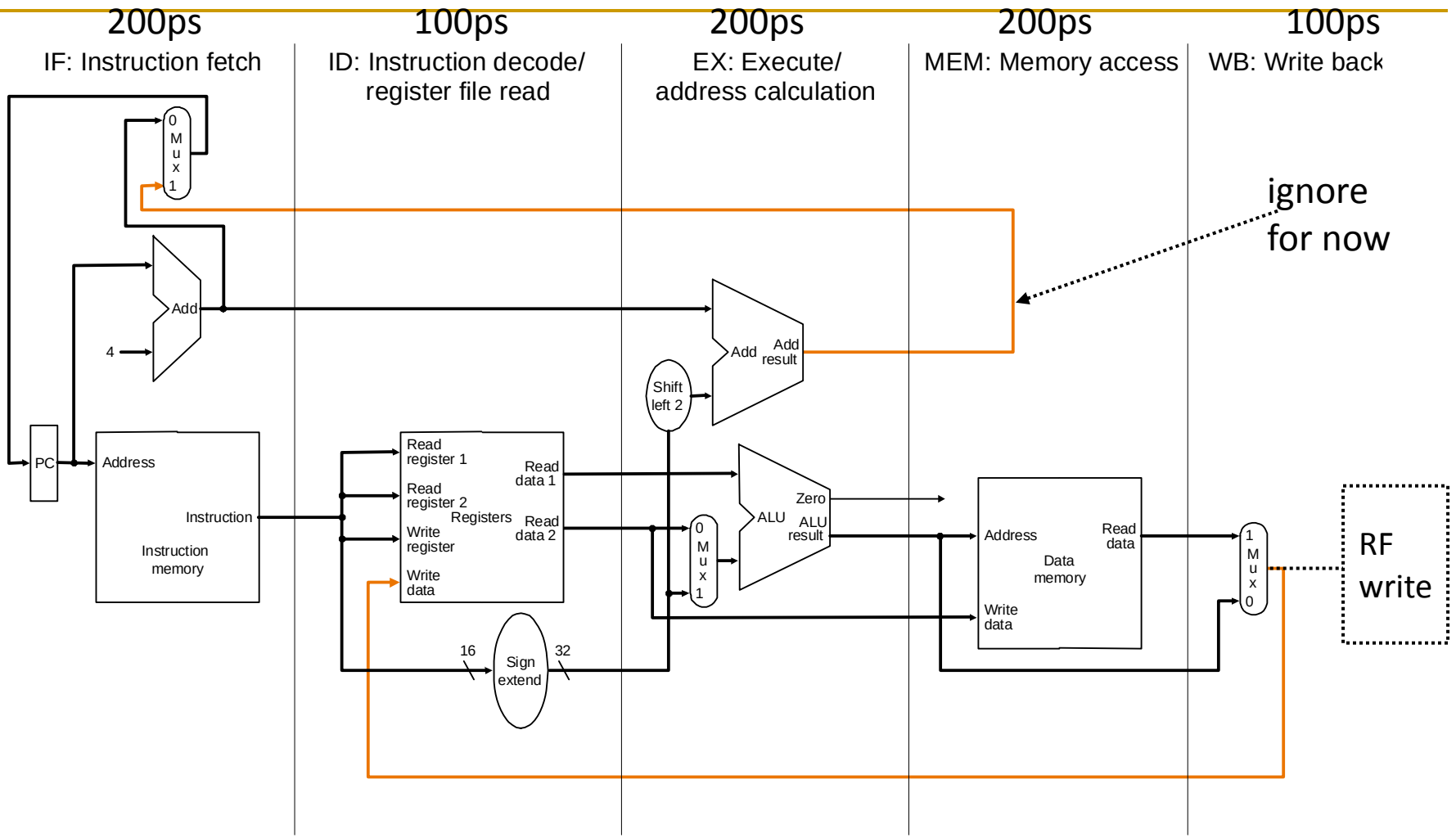


ALU operation

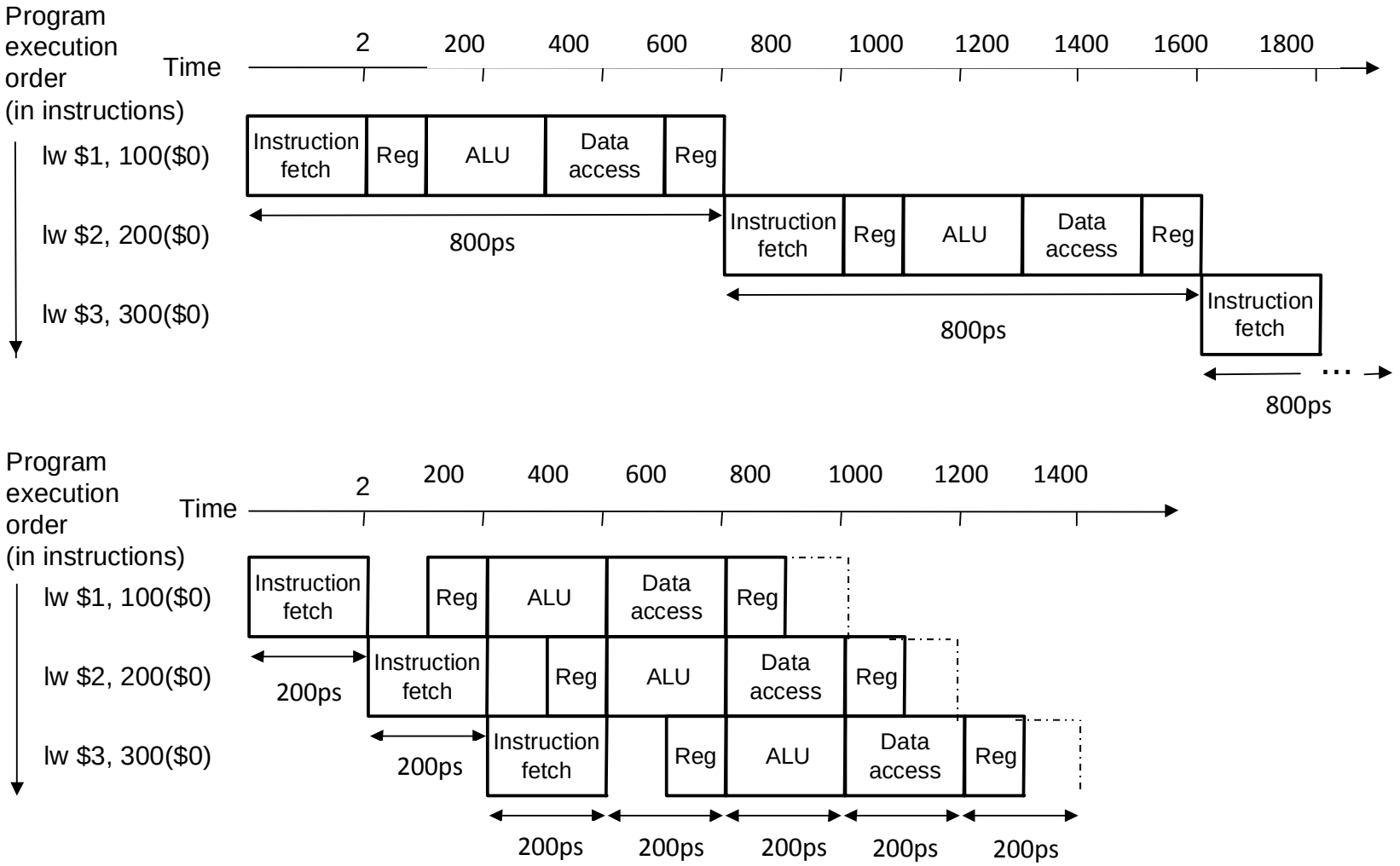
BW=~(1/T)

PIPELINE DATA PATH

Dividing Into Stages



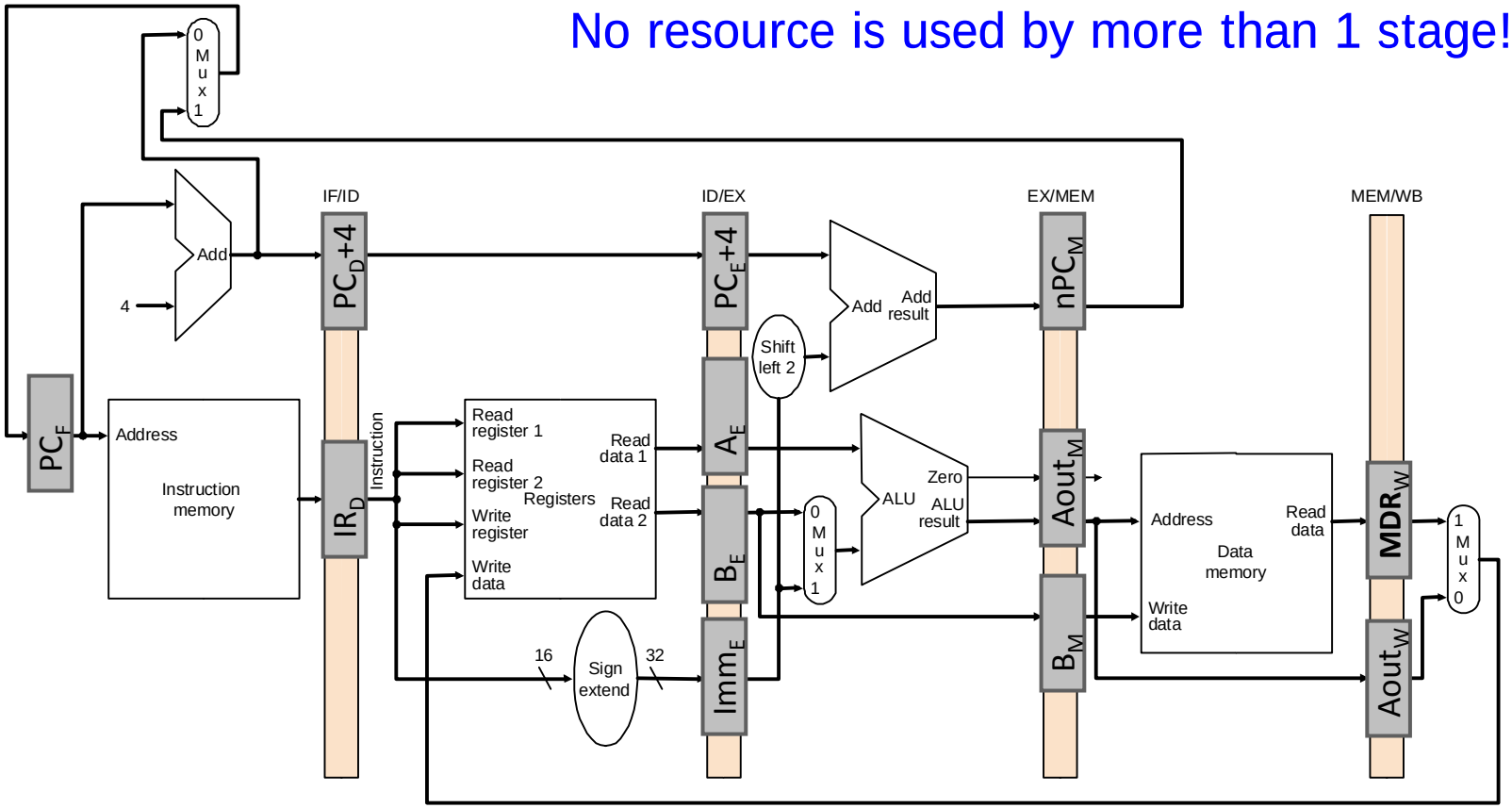
Instruction Pipeline Throughput

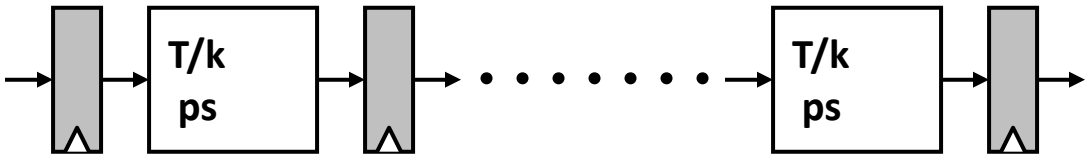


5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

Enabling Pipelined Processing: Pipeline Registers

No resource is used by more than 1 stage!

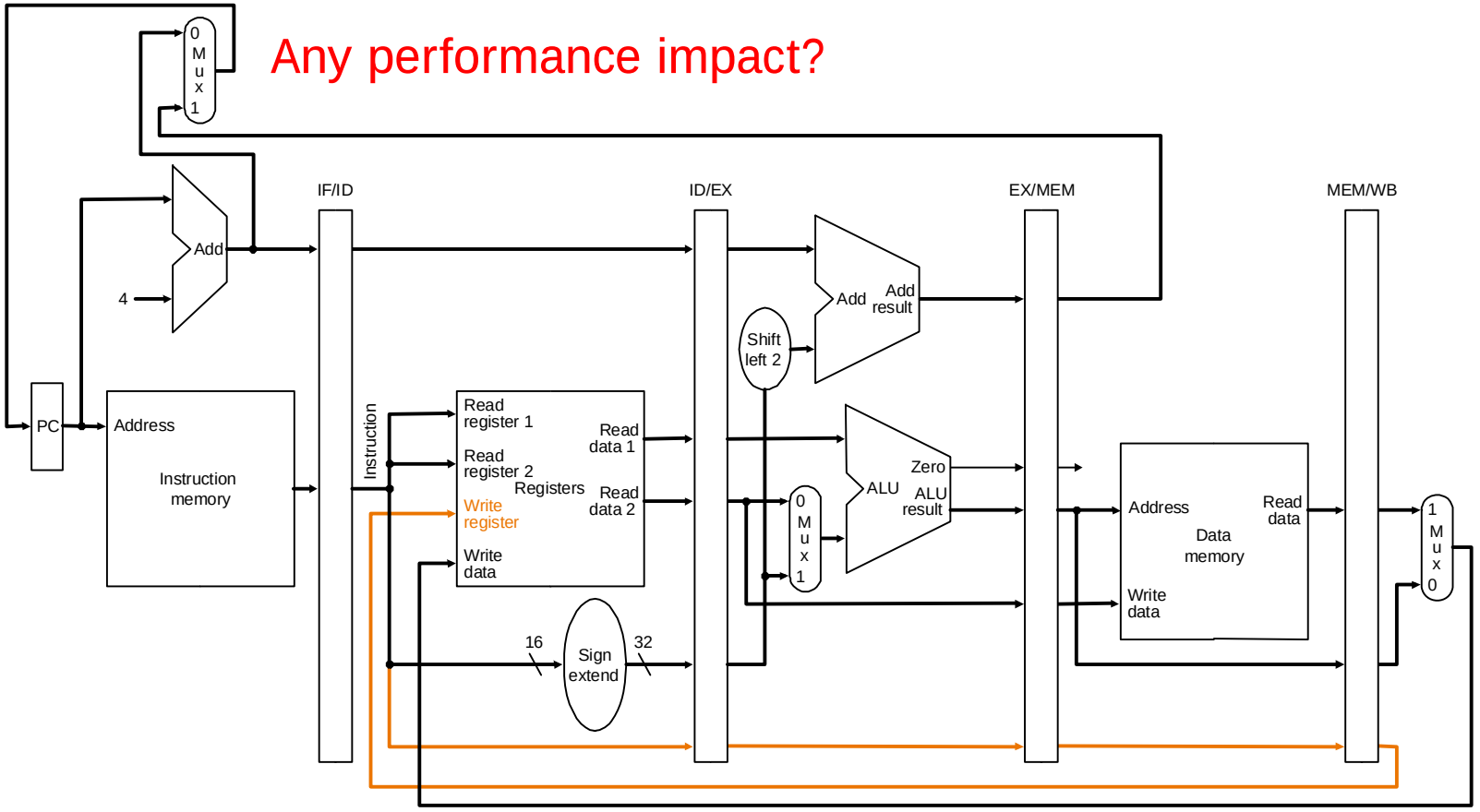




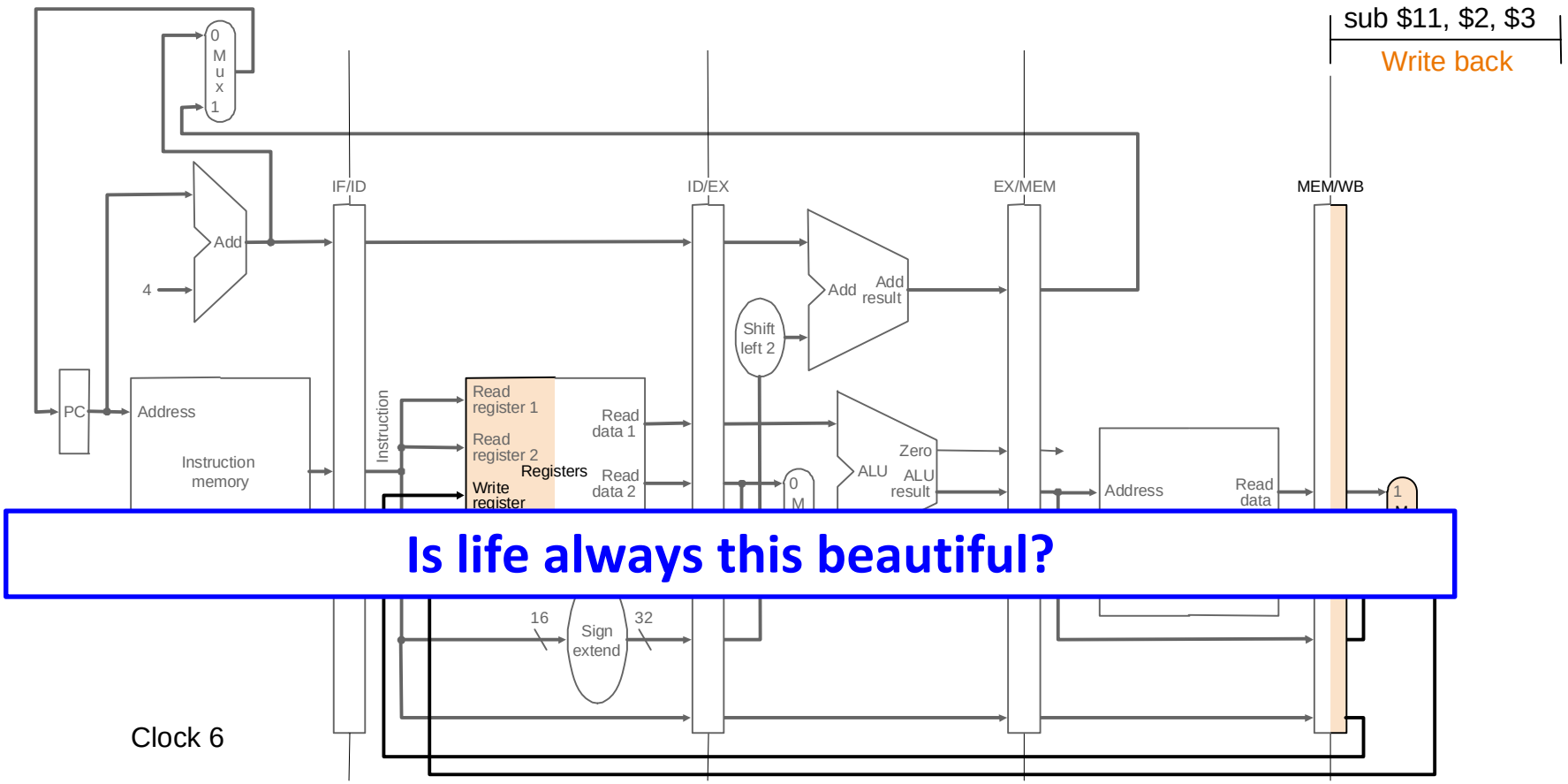
Pipeline performance impact

All instruction classes must follow the same path and timing through the pipeline stages. (compare lw, add)

Any performance impact?

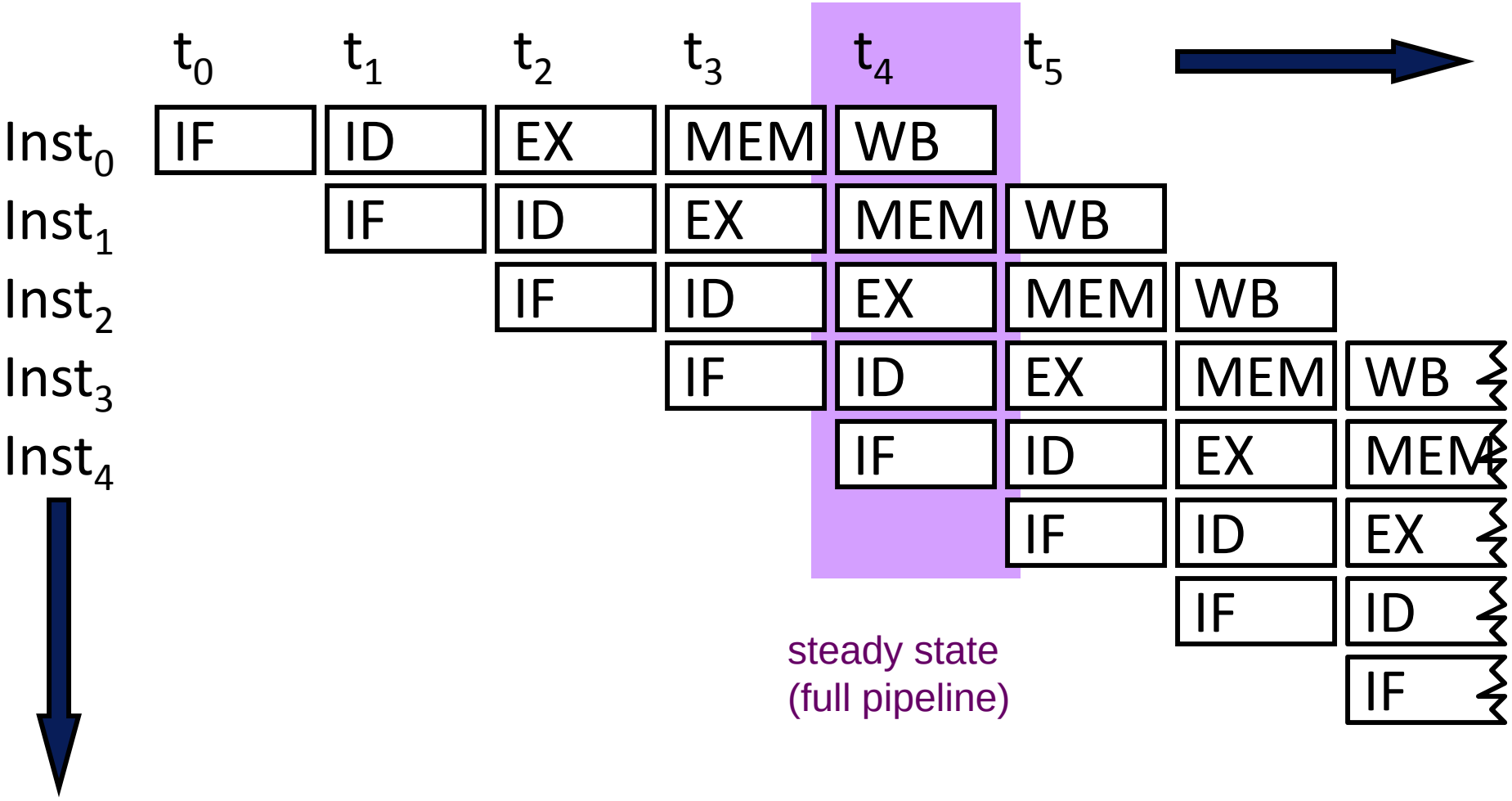


Pipelined Operation Example

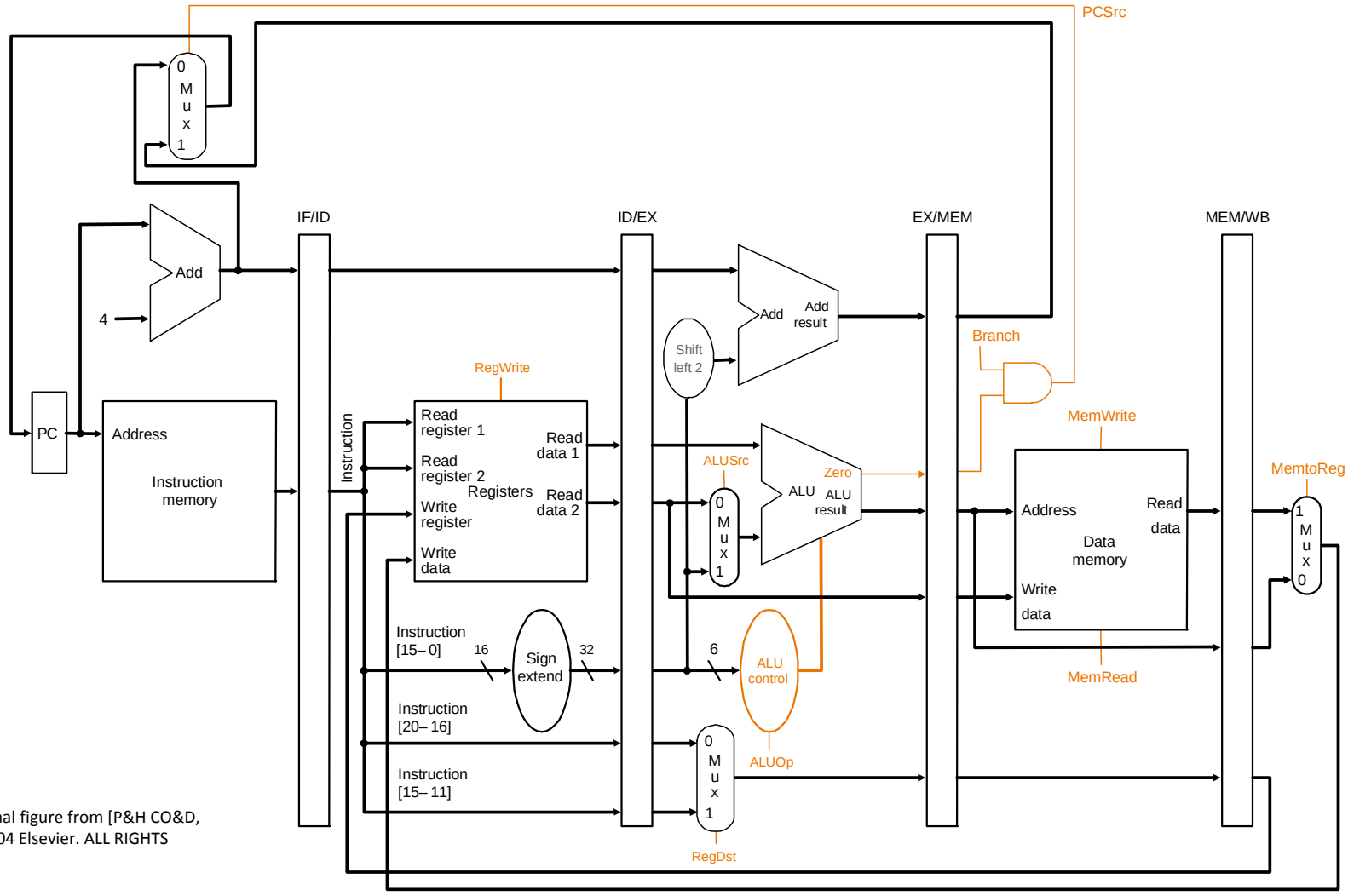


Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Illustrating Pipeline Operation: Operation View



Control Points in a Pipeline



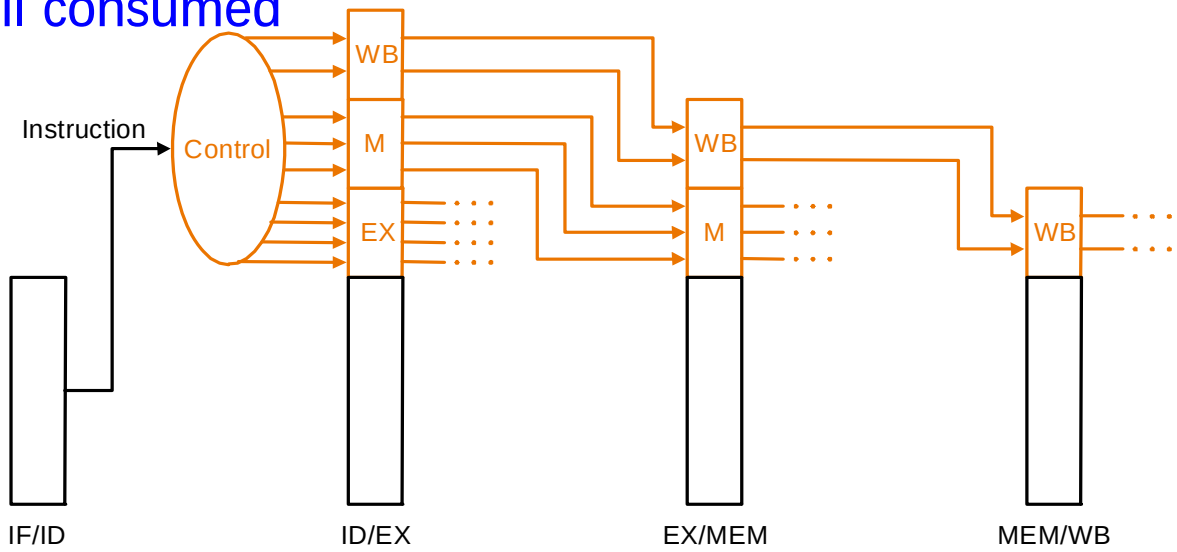
Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

Identical set of control points as the single-cycle datapath!!

PIPELINE CONTROL SIGNALS

Control Signals in a Pipeline

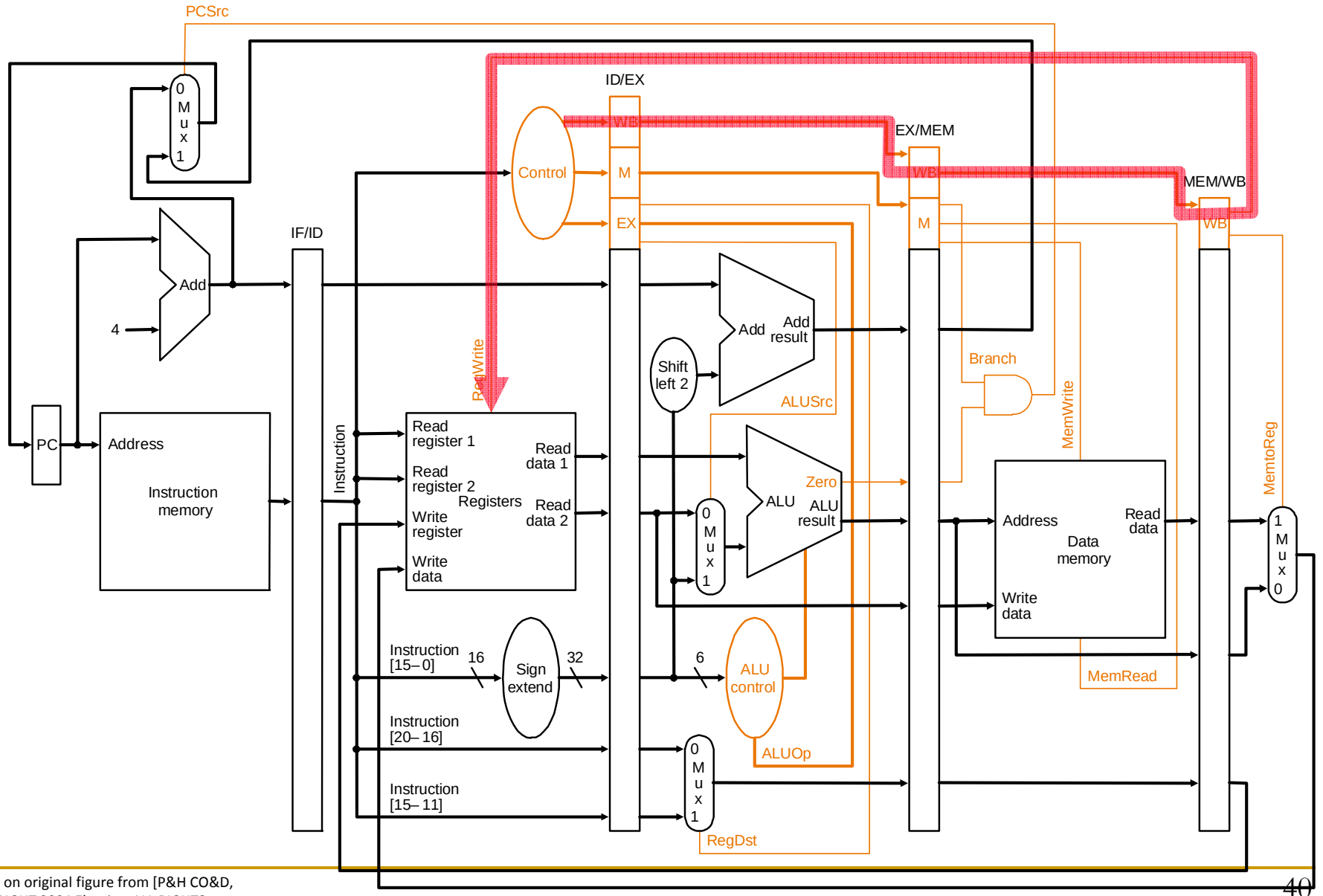
- For a given instruction
  - same control signals as single-cycle, but
  - control signals required at different cycles, depending on stage
  - ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



  - ⇒ Option 2: carry relevant "instruction word/field" down the pipeline and decode locally within each or in a previous stage

Which one is better?

Pipelined Control Signals



Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

PIPELINE ISSUES

Remember: An Ideal Pipeline

- Goal: Increase throughput with little increase in cost (hardware cost, in case of instruction processing)
- Repetition of identical operations
  - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of independent operations
  - No dependencies between repeated operations
- Uniformly partitionable suboperations
  - Processing an be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
  - What about the instruction processing “cycle”?

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations … NOT!
  - ⇒ different instructions → not all need the same stages
    - Forcing different instructions to go through the same pipe stages
    - → external fragmentation (some pipe stages idle for some instructions)
- Uniform suboperations … NOT!
  - ⇒ different pipeline stages → not the same latency
    - Need to force each stage to be controlled by the same clock
    - → internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)
- Independent operations … NOT!
  - ⇒ instructions are not independent of each other
    - Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results
    - → pipeline stalls (pipeline is not always moving)

Issues in Pipeline Design

- Balancing work in pipeline stages
  - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
  - Handling dependences
    - Data
    - Control
  - Handling resource contention
  - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
  - Minimizing *stalls*

Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

DEPENDENCES

A computer program

- Following the Von Neumann model, the program is a sequence of instructions

```
L1: Mov AX, 3
    Add ax, bx
    ....
    jmp l1
```

Dependences and Their Types

- Also called “dependency” or *less desirably* “hazard”
- Dependences dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource

- Solution 1: Eliminate the cause of contention
  - Duplicate the resource or increase its throughput
    - E.g., use separate instruction and data memories (caches)
    - E.g., use multiple ports for memory structures

- Solution 2: Detect the resource contention and stall one of the contending stages
  - Which stage do you stall?
  - Example: What if you had a single read and write port for the register file?

UNDERSTANDING DEPENDENCES

Data Dependences

- Types of data dependences
  - Flow dependence (true data dependence – read after write)
    - Comes from the running program semantic
  - Anti dependence (write after read)
  - Output dependence (write after write)

- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program is correct
  - Flow dependences always need to be obeyed because they constitute true dependence on a value
  - Anti and output dependences exist due to limited number of architectural registers
    - They are dependence on a name, not a value

Data Dependence Types

Flow dependence

$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

Read-after-Write (RAW)

Anti dependence

$r_3 \leftarrow r_1$ op r_2

$r_1 \leftarrow r_4$ op r_5

Write-after-Read (WAR)

Output-dependence

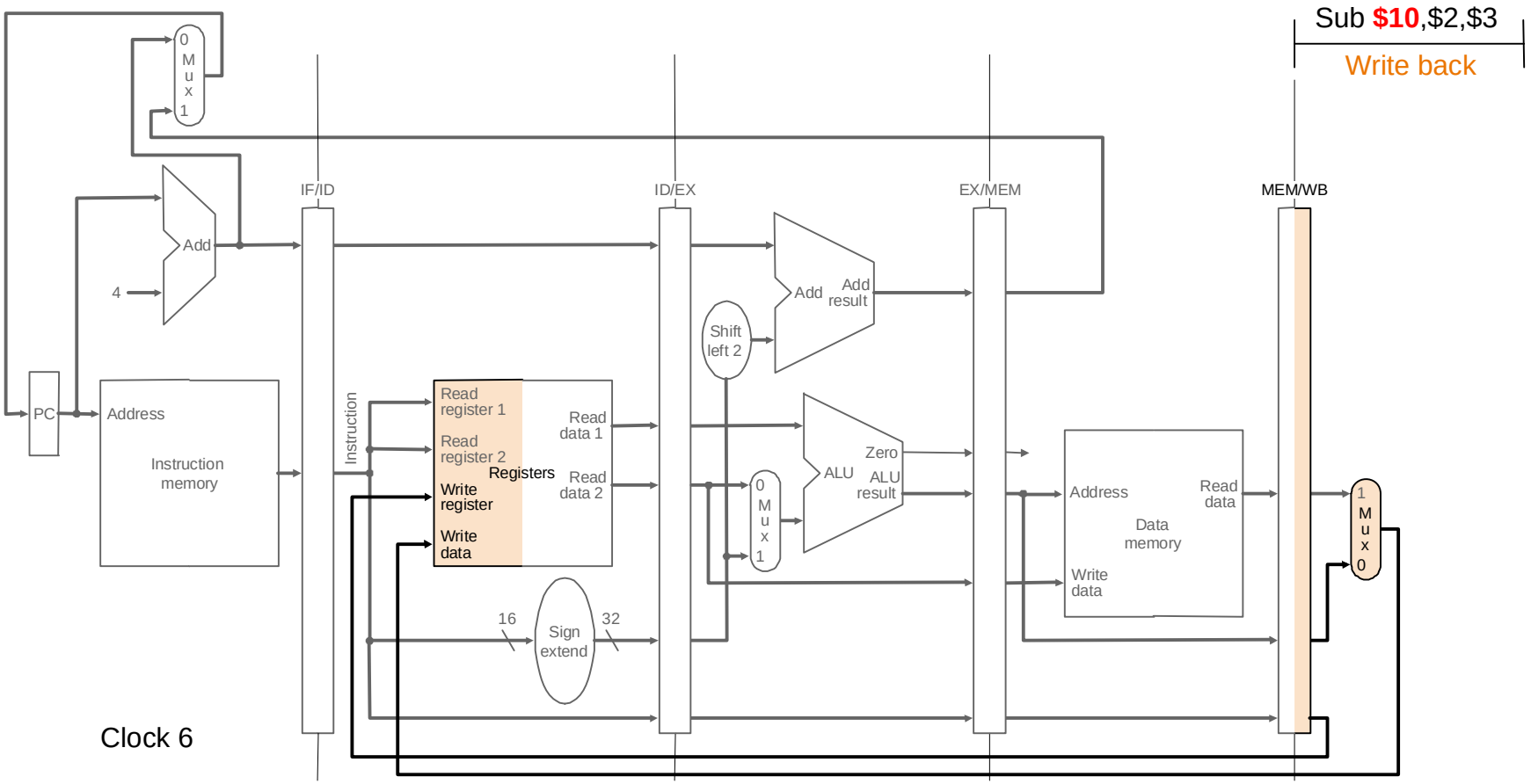
$r_3 \leftarrow r_1$ op r_2

$r_5 \leftarrow r_3$ op r_4

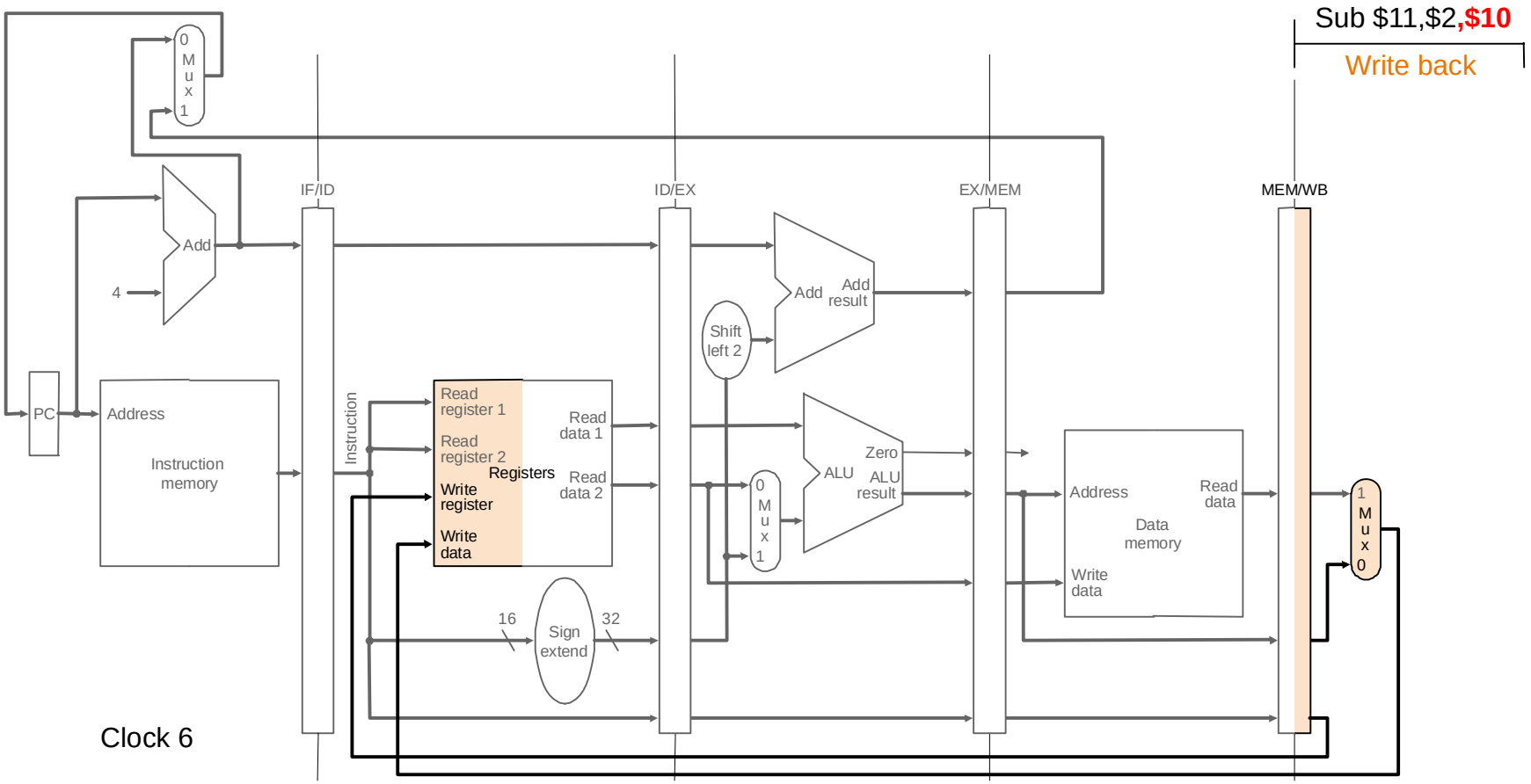
$r_3 \leftarrow r_6$ op r_7

Write-after-Write (WAW)

Example output dependence



Example of flow dependence



Control Dependence

- Question: What should the fetch PC be in the next cycle?
- Answer: The address of the next instruction
  - All instructions are control dependent on previous ones. Why?

- If the fetched instruction is a non-control-flow instruction:
  - Next Fetch PC is the address of the next-sequential instruction
  - Easy to determine if we know the size of the fetched instruction

- If the instruction that is fetched is a control-flow instruction:
  - How do we determine the next Fetch PC?

- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

DEPENDENCES DETECTION

Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution

- Software based interlocking

  vs.

- Hardware based interlocking

- MIPS acronym?
  - Microprocessor without Interlocked Pipeline Stages

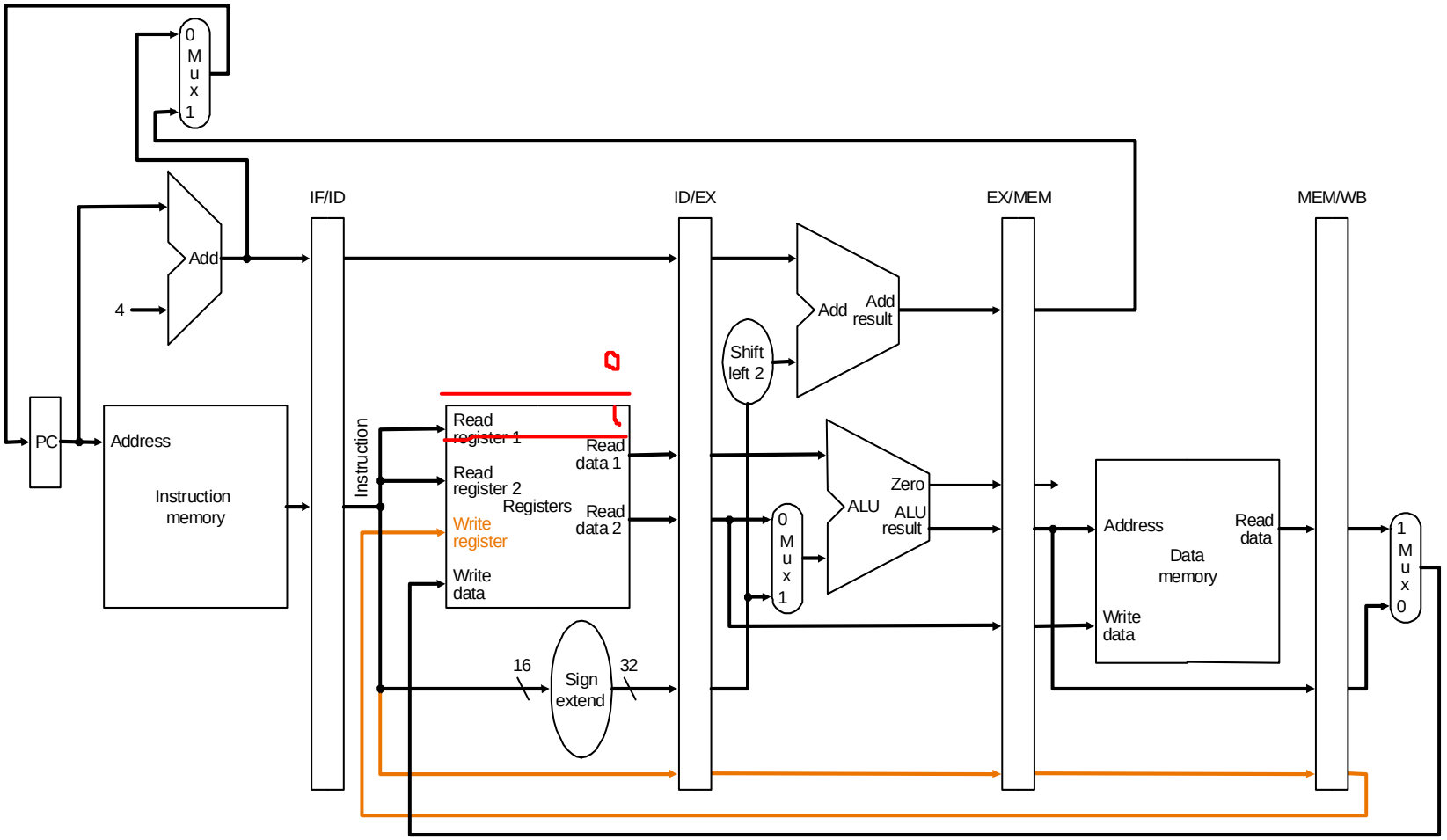
Approaches to Dependence Detection (I)

- Scoreboarding
  - Each register in register file has a Valid bit associated with it
  - An instruction that is writing to the register resets the Valid bit
  - An instruction in Decode stage checks if all its source and destination registers are Valid
    - Yes: No need to stall... No dependence
    - No: Stall the instruction

- Advantage:
  - Simple. 1 bit per register

- Disadvantage:
  - Need to stall for all types of dependences, not only flow dep.

Scoreboarding



Not Stalling on Anti and Output Dependences

- What changes would you make to the scoreboard to enable this?
  - counter for writing operation, not just 1 and 0

Approaches to Dependence Detection (II)

- Combinational dependence check logic
  - Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
  - Yes: stall the instruction/pipeline
  - No: no need to stall... no flow dependence

- Advantage:
  - No need to stall on anti and output dependences

- Disadvantage:
  - Logic is more complex than a scoreboard
  - Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

DATA DEPENDENCE HANDLING

Once You Detect the Dependence in Hardware

- What do you do afterwards?

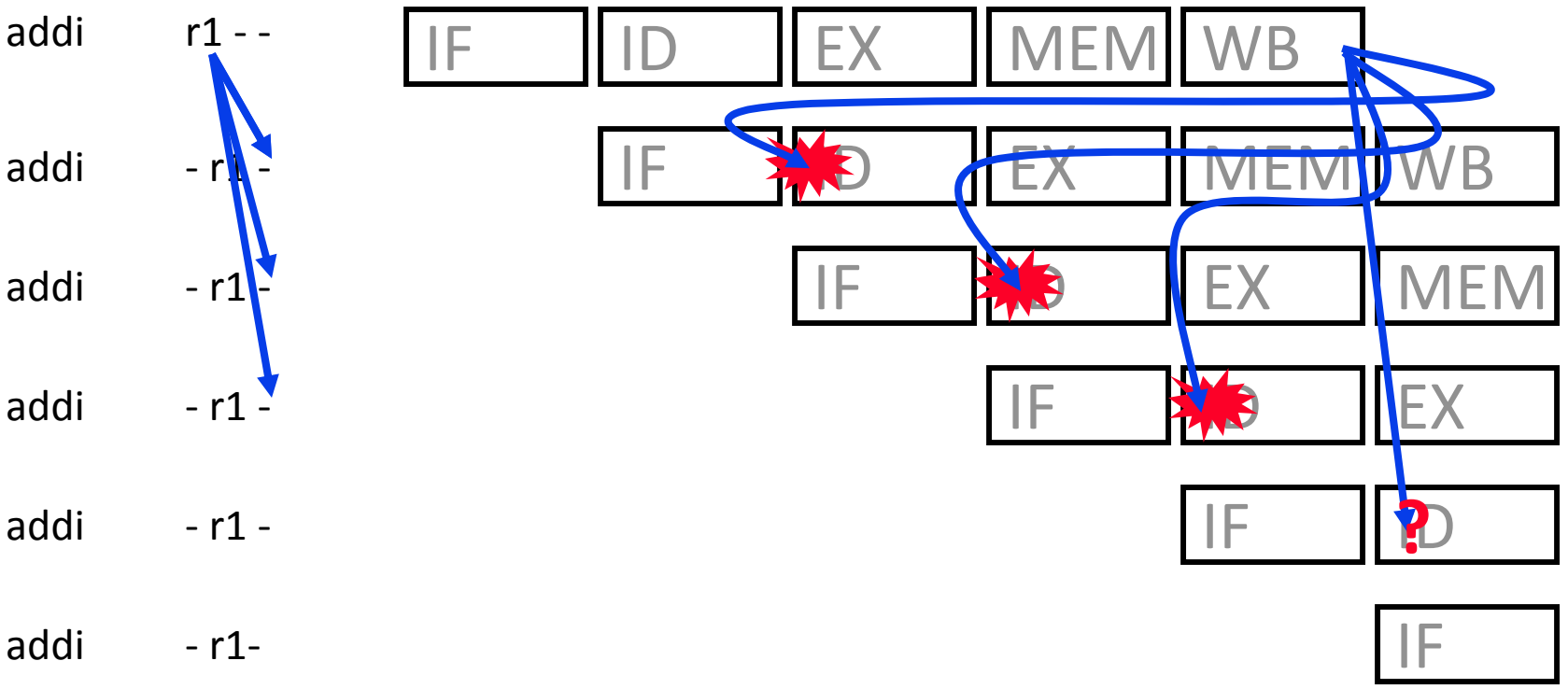
- Observation: Dependence between two instructions is detected before the communicated data value becomes available

How to Handle Data Dependences

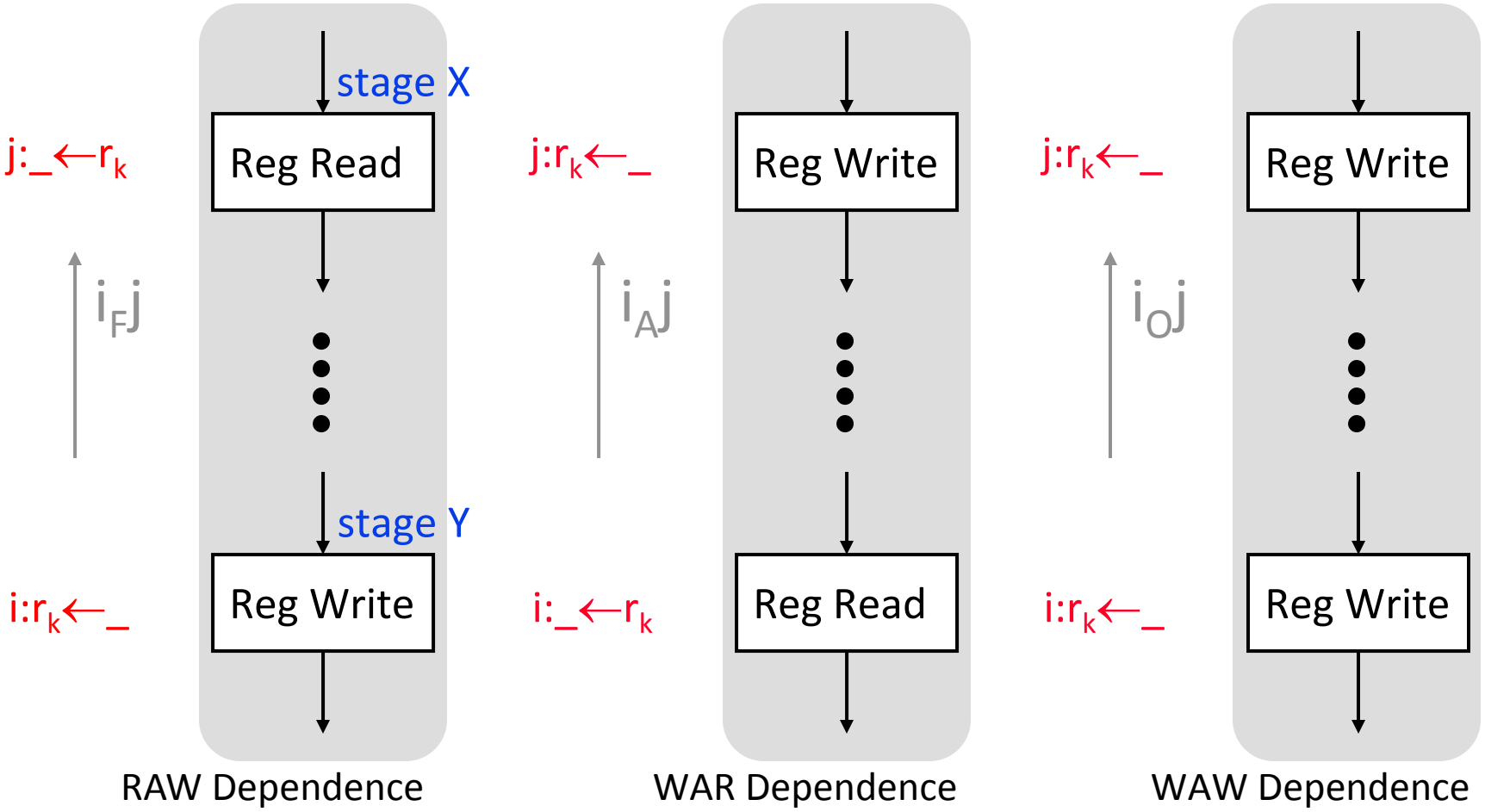
- Anti and output dependences are easier to handle
  - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
  - 1. Detect and wait until value is available in register file
  - 2. Detect and forward/bypass data to dependent instruction
  - Detect and eliminate the dependence at the software level
  - No need for the hardware to detect dependence (MIPS NOP)
  - Do something else (same program "reorder"), (different program "fine-grained multithreading") and No need to detect.
  - Predict the needed value(s), execute "speculatively", and verify

Right place to eliminate dependency

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?



Safe and Unsafe Movement of Pipeline



dist(i,j) $\leq$ dist(X,Y) $\Rightarrow$ Unsafe to keep j moving

dist(i,j) > dist(X,Y) $\Rightarrow$ Safe

RAW Dependence Analysis Example

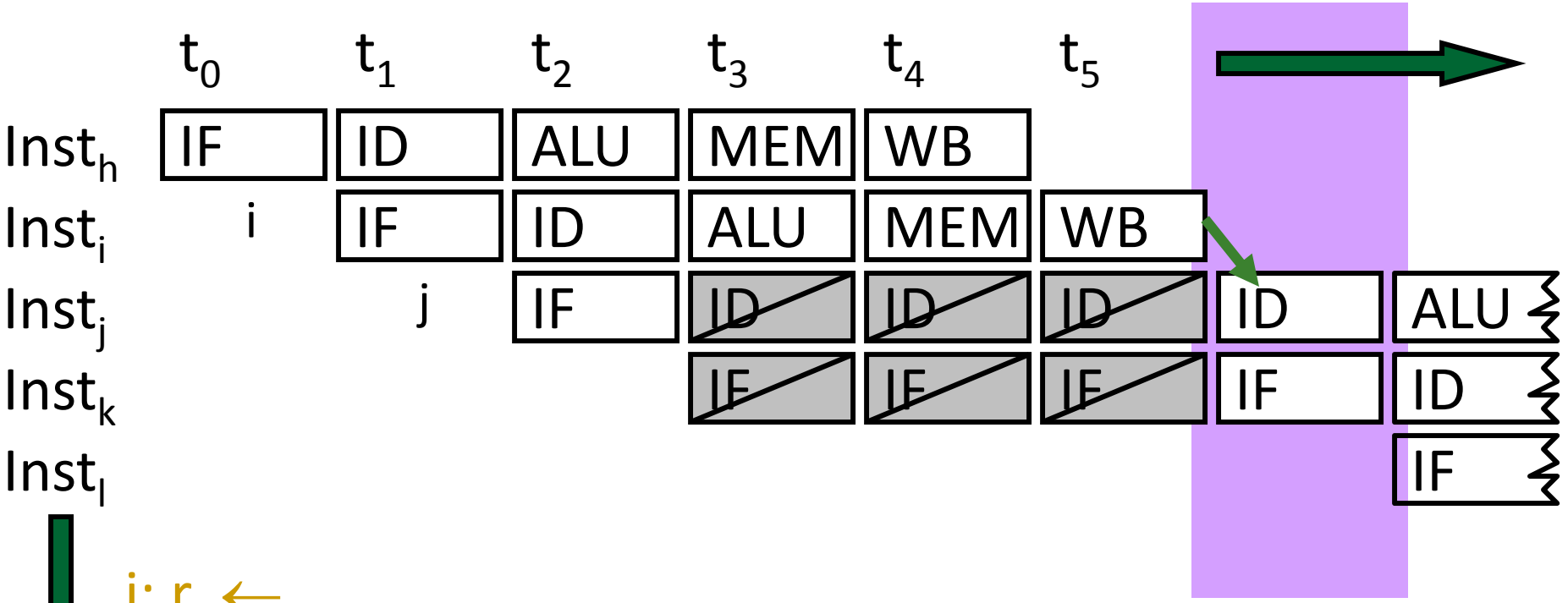
	R/I-Type	LW	SW	Br	J	Jr
IF						
ID	read RF	read RF	read RF	read RF		read RF
EX						
MEM						
WB	write RF	write RF				

- Instructions I_A and I_B (where I_A comes before I_B) have RAW dependence iff
  - I_B (R/I, LW, SW, Br or JR) reads a register written by I_A (R/I or LW)
  - $dist(I_A, I_B) \leq dist(ID, WB) = 3$

What about WAW and WAR dependence?

What about memory data dependence?

Pipeline Stall: Resolving Data Dependence



i: $r_x \leftarrow$ _
bubble
bubble
bubble
j: _ $\leftarrow r_x$

dist(i,j)=4

Stall = make the dependent instruction wait until its source data value is available

1. stop all up-stream stages
2. drain all down-stream stages

Sample Assembly (P&H)

- for (j=i-1; j>=0 && v[j] > v[j+1]; j-=1) { ..... }

```
                addi    $s1, $s0, -1
for2tst:        slti    $t0, $s1, 0
                bne     $t0, $zero, exit2
                sll     $t1, $s1, 2
                add     $t2, $a0, $t1
                lw      $t3, 0($t2)
                lw      $t4, 4($t2)
                slt     $t0, $t4, $t3
                beq     $t0, $zero, exit2
                .........
                addi    $s1, $s1, -1
                j       for2tst
exit2:
```

3 stalls
3 stalls
3 stalls
3 stalls
3 stalls
3 stalls

Readings

- P&H Chapter 4.9-4.11

- Smith and Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, 1995
  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts