

18-447

# Computer Architecture

## Lecture 8: Pipelining

Prof. Onur Mutlu

Carnegie Mellon University

Spring 2013, 2/4/2013

# Reminder: Homework 2

---

- Homework 2 out
  - Due February 11 (next Monday)
  - LC-3b microcode
  - ISA concepts, ISA vs. microarchitecture, microcoded machines

# Reminder: Lab Assignment 2

---

- Lab Assignment 1.5
  - Verilog practice
  - Not to be turned in
- Lab Assignment 2
  - Due Feb 15
  - Single-cycle MIPS implementation in Verilog
  - All labs are individual assignments
  - No collaboration; please respect the honor code

# Lookahead: Extra Credit for Lab Assignment 2

---

- Complete your normal (single-cycle) implementation first, and get it checked off in lab.
- Then, implement the MIPS core using a microcoded approach similar to what we will discuss in class.
- We are not specifying any particular details of the microcode format or the microarchitecture; you can be creative.
- For the extra credit, the microcoded implementation should execute the same programs that your ordinary implementation does, and you should demo it by the normal lab deadline.
- You will get maximum 4% of course grade
- Document what you have done and demonstrate well

# Readings for Today

---

- Pipelining

- P&H Chapter 4.5-4.8
- Optional: Hamacher et al. book, Chapter 6, “Pipelining”

- Pipelined LC-3b Microarchitecture

- <http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=18447-lc3b-pipelining.pdf>

# Today's Agenda

---

- Finish microprogrammed microarchitectures
- Start pipelining

# Review: Last Lecture

---

- An exercise in microprogramming

# Review: An Exercise in Microprogramming



# Handouts

---

- 7 pages of Microprogrammed LC-3b design
- <http://www.ece.cmu.edu/~ece447/s13/doku.php?id=manuals>
- <http://www.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=lc3b-figures.pdf>

# A Simple LC-3b Control and Datapath

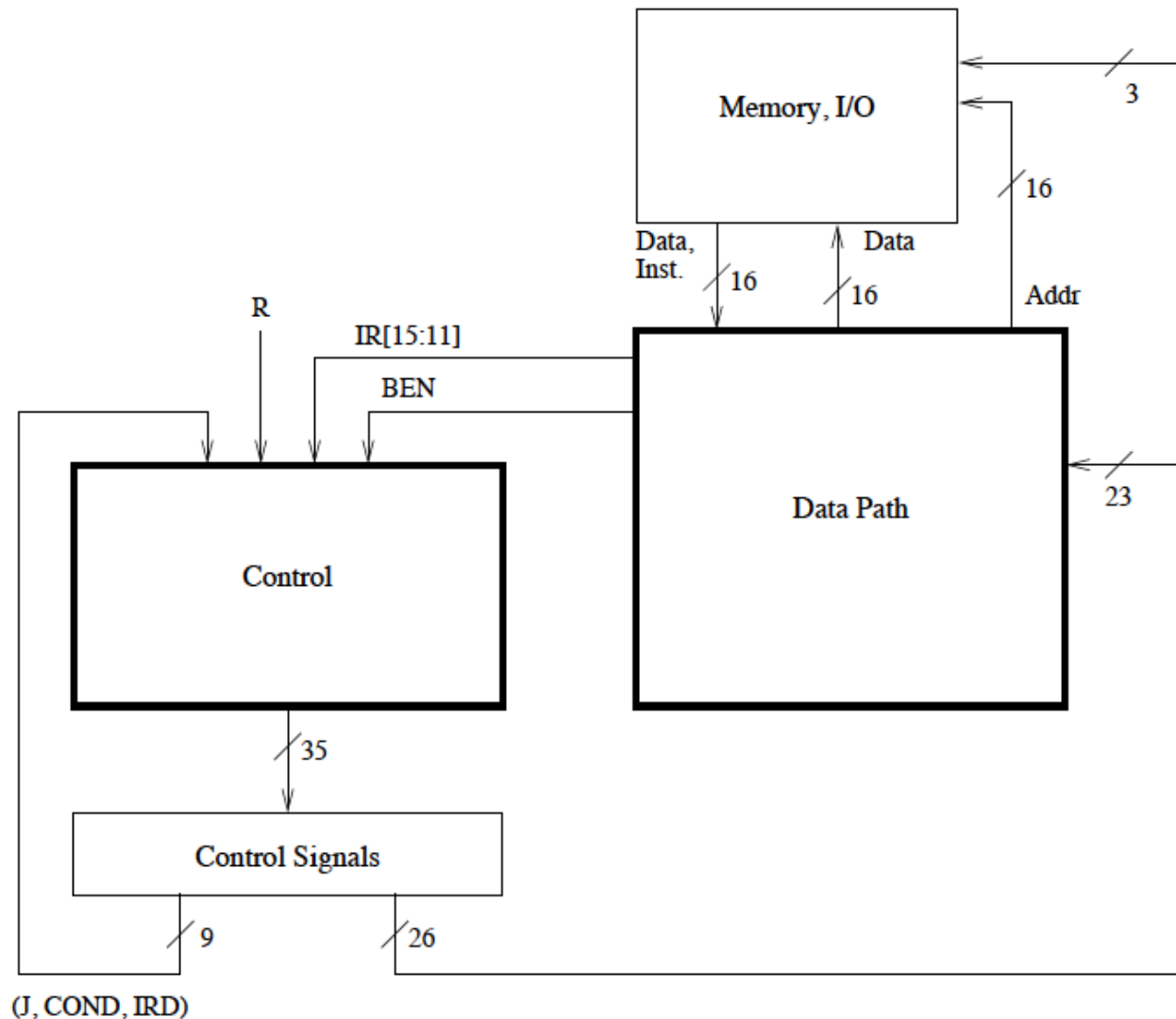
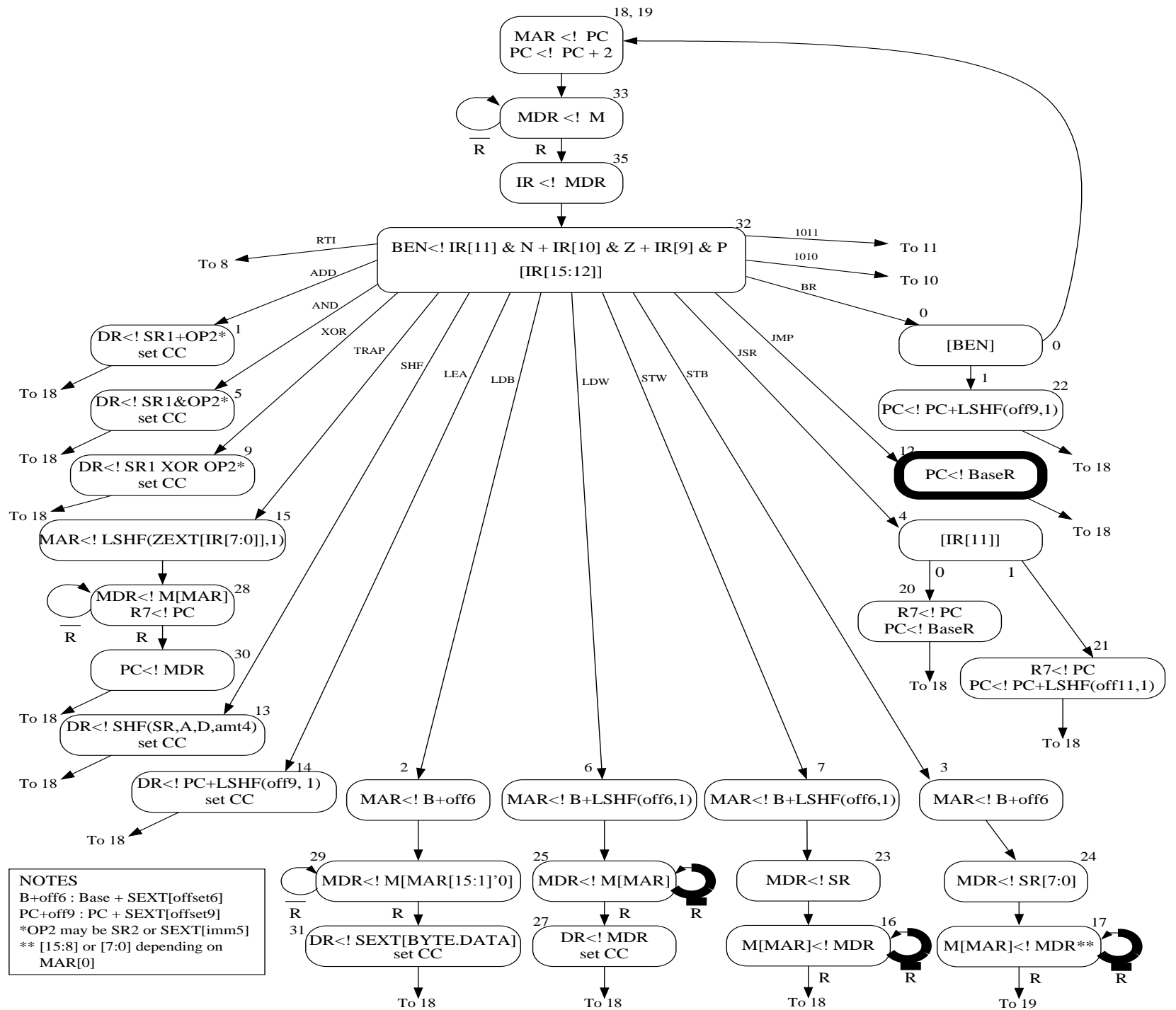
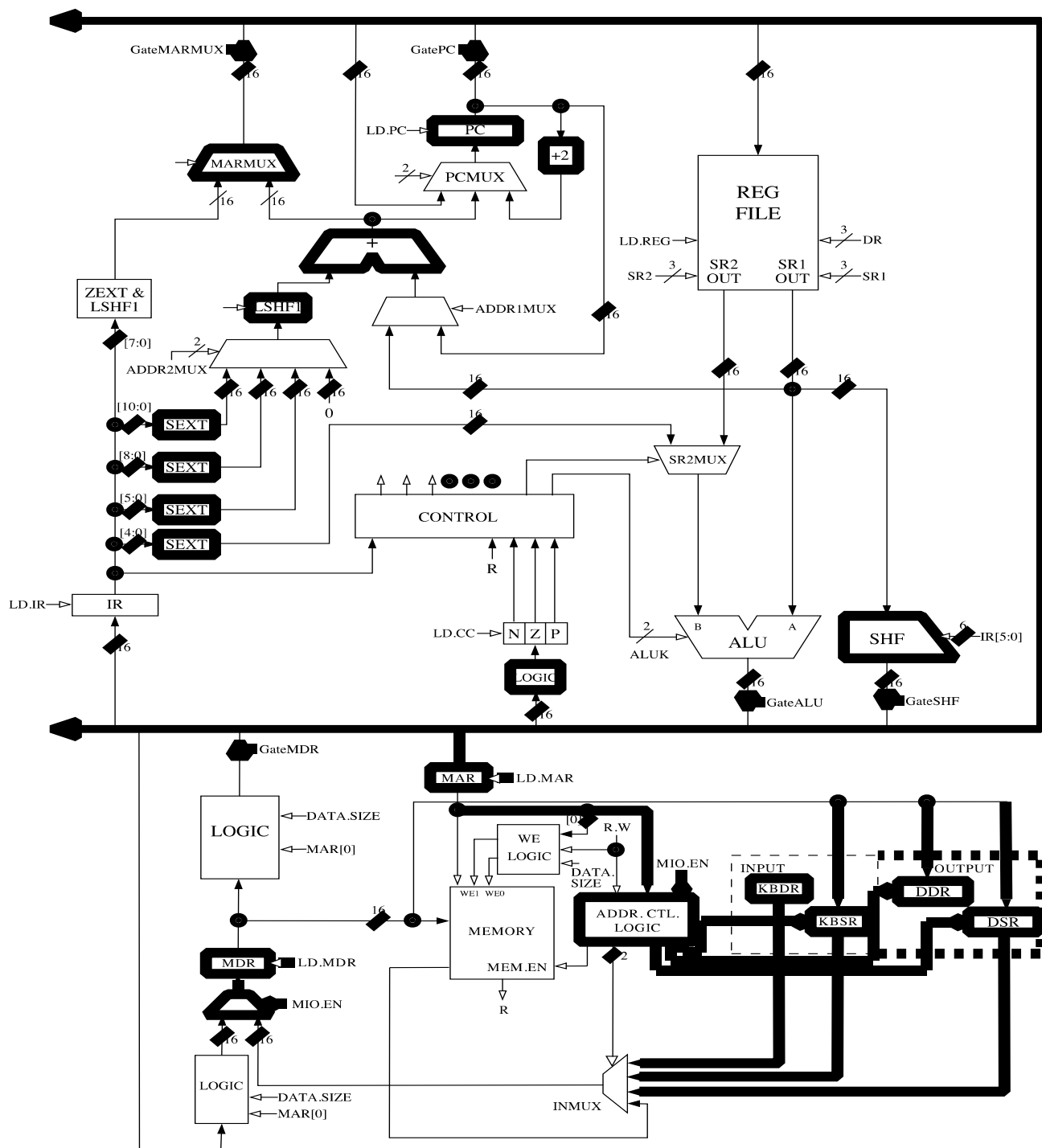
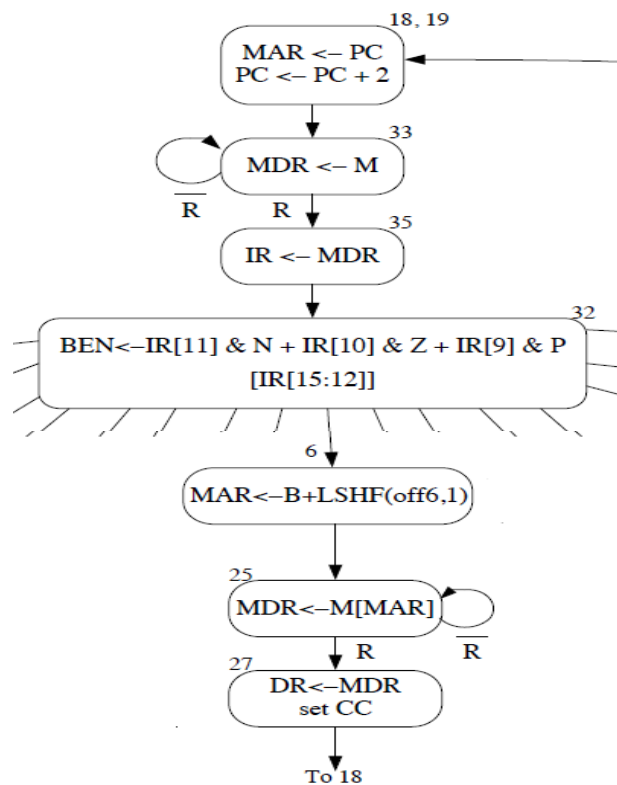


Figure C.1: Microarchitecture of the LC-3b, major components

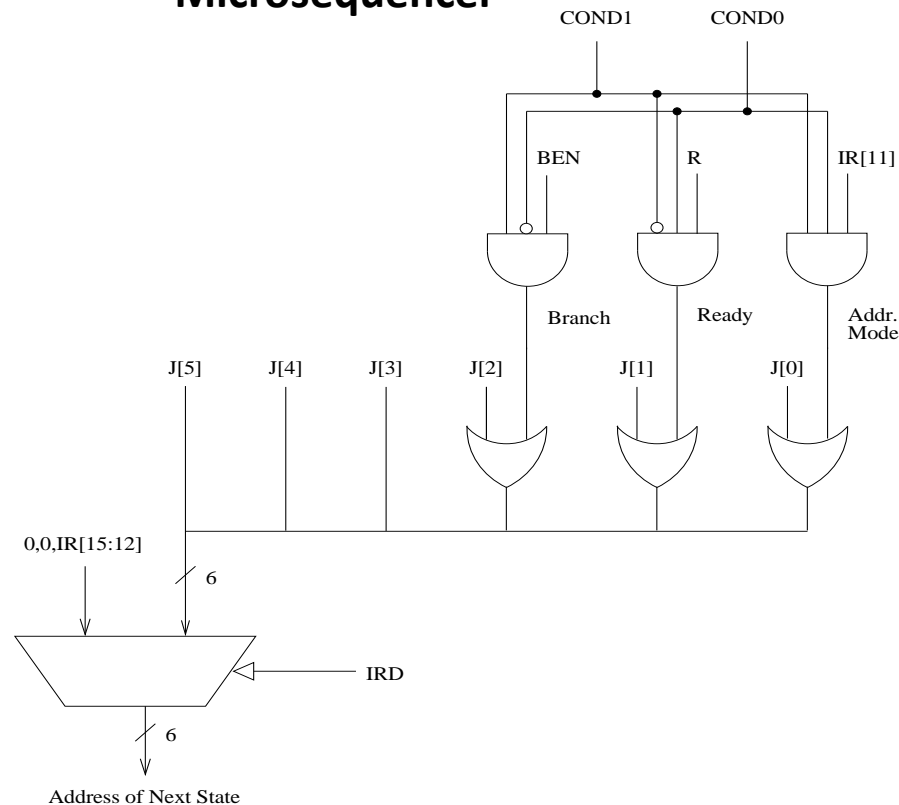


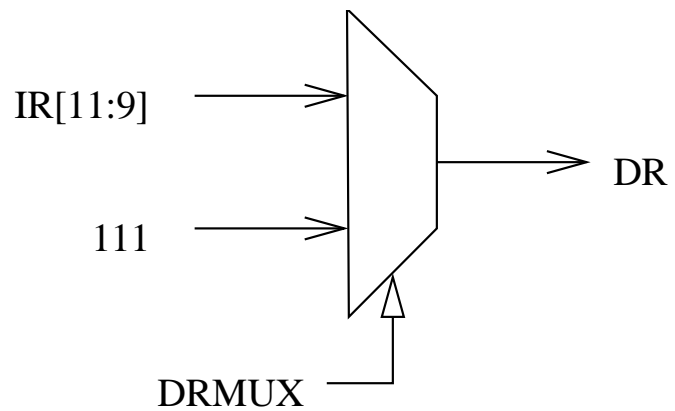


## State Machine for LDW

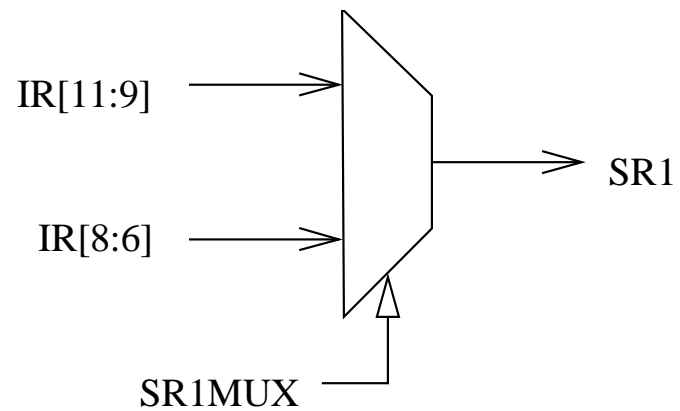


## Microsequencer

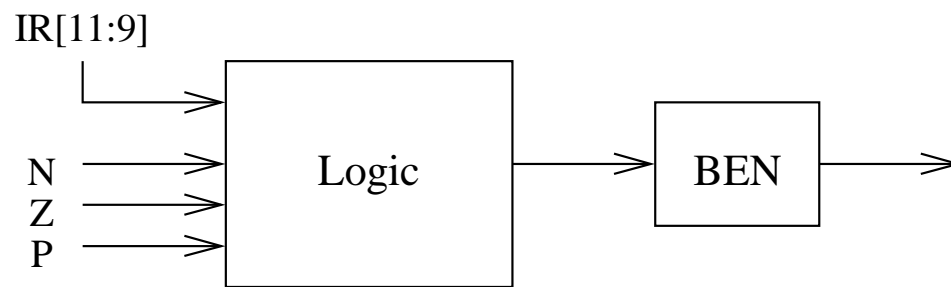
[illegible]



(a)



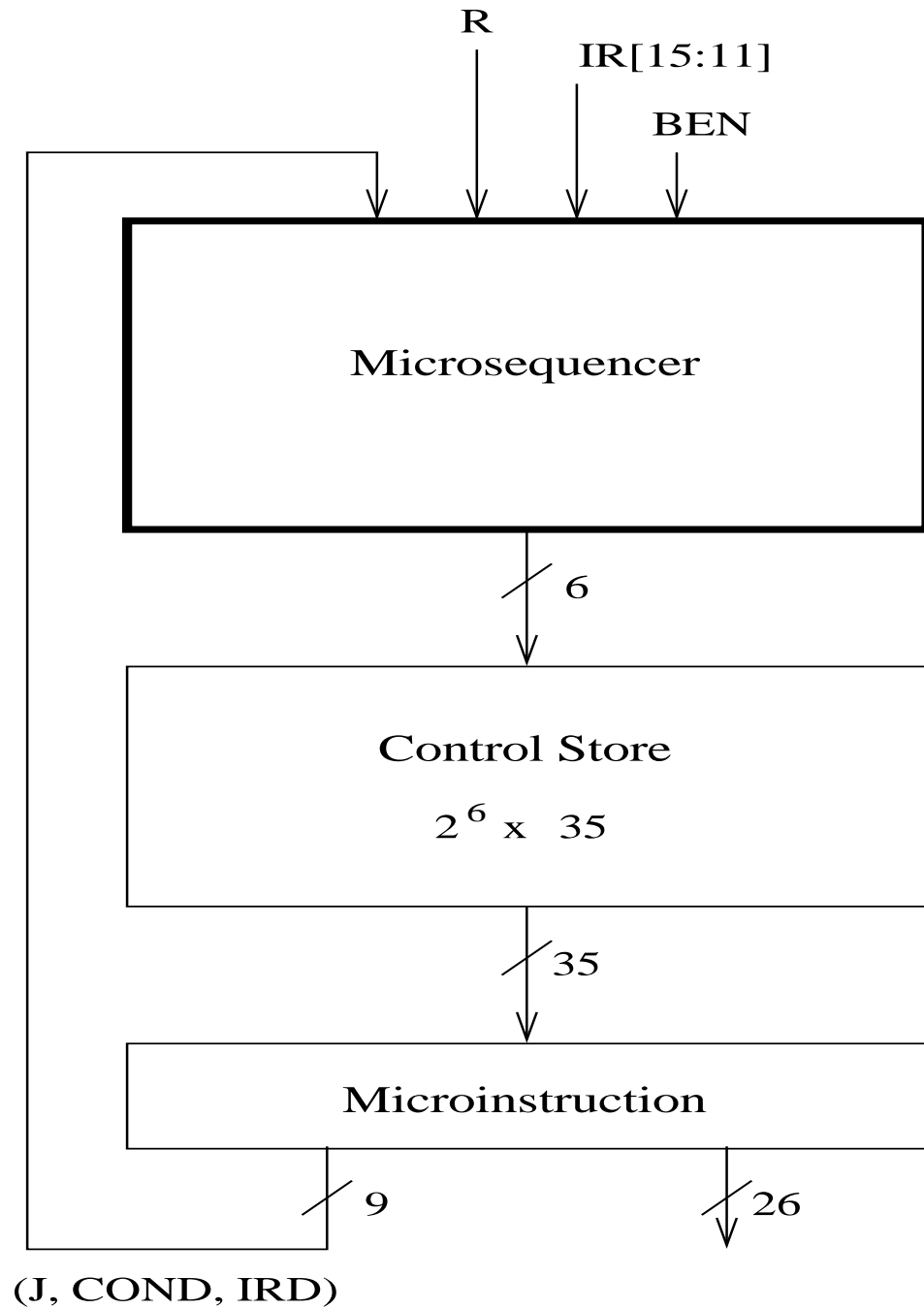
(b)



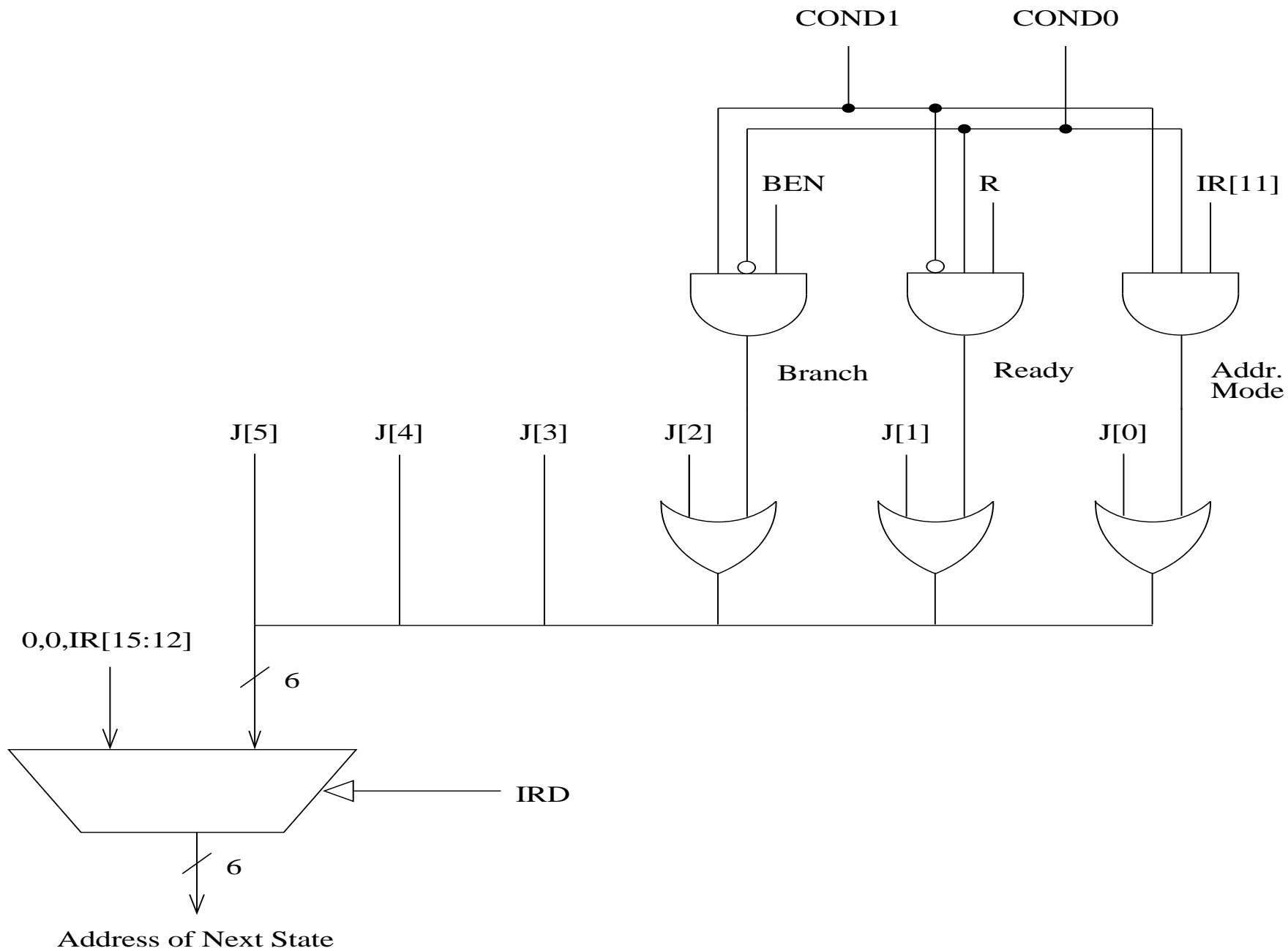
(c)

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateSHF/1:	NO, YES	
PCMUX/2:	PC+2 BUS ADDER	;select pc+2 ;select value from bus ;select output of address adder
DRMUX/1:	11.9 R7	;destination IR[11:9] ;destination R7
SR1MUX/1:	11.9 8.6	;source IR[11:9] ;source IR[8:6]
ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO offset6 PCoffset9 PCoffset11	;select the value zero ;select SEXT[IR[5:0]] ;select SEXT[IR[8:0]] ;select SEXT[IR[10:0]]
MARMUX/1:	7.0 ADDER	;select LSHF(ZEXT[IR[7:0]],1) ;select output of address adder
ALUK/2:	ADD, AND, XOR, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
DATA.SIZE/1:	BYTE, WORD	
LSHF1/1:	NO, YES	

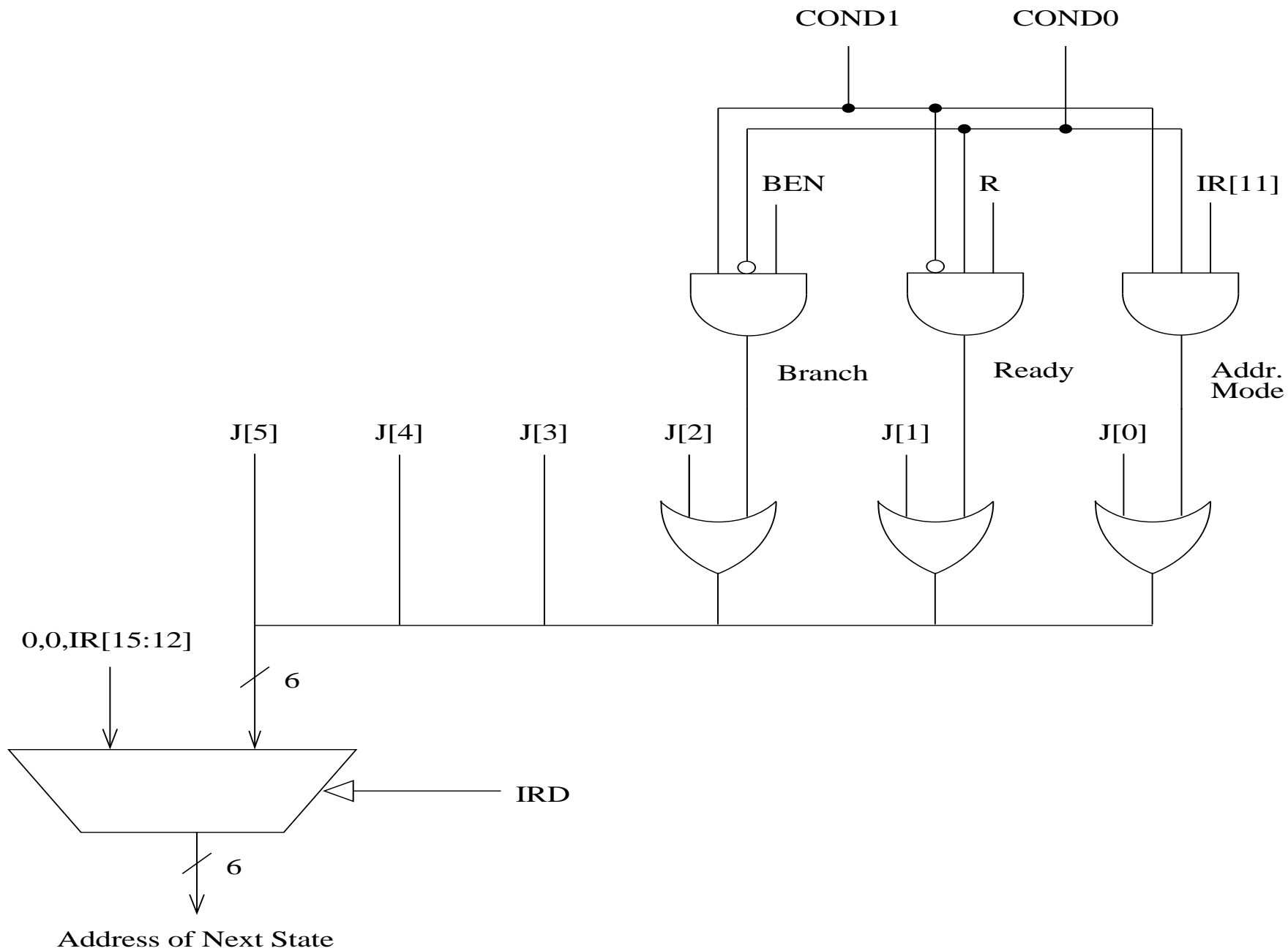
Table C.1: Data path control signals







IRD	Cond	J	LD.MAR	LD.MDR	LD.IR	LD.BEN	LD.REG	LD.CC	LD.PC	GnGnC	GnGnMDR	GnGnLU	GnGnSHF	PCMUX	DRMUX	SRIMUX	ADDRIMUX	ADDRZMUX	MARKMUX	ALUX	MIO.EN	R.W	DATASIZE	Lshift	
																									000000 (State 0)
																									000001 (State 1)
																									000010 (State 2)
																									000011 (State 3)
																									000100 (State 4)
																									000101 (State 5)
																									000110 (State 6)
																									000111 (State 7)
																									001000 (State 8)
																									001001 (State 9)
																									001010 (State 10)
																									001011 (State 11)
																									001100 (State 12)
																									001101 (State 13)
																									001110 (State 14)
																									001111 (State 15)
																									010000 (State 16)
																									010001 (State 17)
																									010010 (State 18)
																									010011 (State 19)
																									010100 (State 20)
																									010101 (State 21)
																									010110 (State 22)
																									010111 (State 23)
																									011000 (State 24)
																									011001 (State 25)
																									011010 (State 26)
																									011011 (State 27)
																									011100 (State 28)
																									011101 (State 29)
																									011110 (State 30)
																									011111 (State 31)
																									100000 (State 32)
																									100001 (State 33)
																									100010 (State 34)
																									100011 (State 35)
																									100100 (State 36)
																									100101 (State 37)
																									100110 (State 38)
																									100111 (State 39)
																									101000 (State 40)
																									101001 (State 41)
																									101010 (State 42)
																									101011 (State 43)
																									101100 (State 44)
																									101101 (State 45)
																									101110 (State 46)
																									101111 (State 47)
																									110000 (State 48)
																									110001 (State 49)
																									110010 (State 50)
																									110011 (State 51)
																									110100 (State 52)
																									110101 (State 53)
																									110110 (State 54)
																									110111 (State 55)
																									111000 (State 56)
																									111001 (State 57)
																									111010 (State 58)
																									111011 (State 59)
																									111100 (State 60)



# Review: End of the Exercise in Microprogramming

# The Microsequencer: Some Questions

---

- When is the IRD signal asserted?
- What happens if an illegal instruction is decoded?
- What are condition (COND) bits for?
- How is variable latency memory handled?
- How do you do the state encoding?
  - Minimize number of state variables
  - Start with the 16-way branch
  - Then determine constraint tables and states dependent on COND

# The Control Store: Some Questions

---

- What control signals can be stored in the control store?

vs.

- What control signals have to be generated in hardwired logic?
  - i.e., what signal cannot be available without processing in the datapath?

# Variable-Latency Memory

---

- The ready signal (R) enables memory read/write to execute correctly
  - Example: transition from state 33 to state 35 is controlled by the R bit asserted by memory when memory data is available
- Could we have done this in a single-cycle microarchitecture?

# The Microsequencer: Advanced Questions

---

- What happens if the machine is interrupted?
- What if an instruction generates an exception?
- How can you implement a complex instruction using this control structure?
  - Think REP MOVS



# 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 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 control signals needed (no need if the operation can be “translated” into existing control signals)

# Let's Do Some More Microprogramming

---

- Implement REP MOVS in the LC-3b microarchitecture
- What changes, if any, do you make to the
  - state machine?
  - datapath?
  - control store?
  - microsequencer?
- Show all changes and microinstructions
- Coming up in Homework 3

# Aside: Alignment Correction in Memory

---

- Remember unaligned accesses
- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
  - Convenience to the programmer/compiler
- How does the hardware ensure this works correctly?
  - Take a look at state 29 for LDB
  - States 24 and 17 for STB
  - Additional logic to handle unaligned accesses

# Aside: Memory Mapped I/O

---

- Address control logic determines whether the specified address of LDx and STx are to memory or I/O devices
- Correspondingly enables memory or I/O devices and sets up muxes
- Another instance where the final control signals (e.g., MEM.EN or INMUX/2) cannot be stored in the control store
  - Dependent on address

# Advantages of Microprogrammed Control

---

- Allows a very simple datapath to do powerful computation by controlling the datapath (using a sequencer)
  - High-level ISA translated into microcode (sequence of microinstructions)
  - Microcode enables a minimal datapath to emulate an ISA
  - Microinstructions can be thought of a user-invisible ISA
- Enables easy extensibility of the ISA
  - Can support a new instruction by changing the ucode
  - Can support complex instructions as a sequence of simple microinstructions
- If I can sequence an arbitrary instruction then I can sequence an arbitrary “program” as a microprogram sequence
  - will need some new state (e.g. loop counters) in the microcode for sequencing more elaborate programs

# Update of Machine Behavior

---

- The ability to update/patch microcode in the field (after a processor is shipped) enables
  - Ability to add new instructions without changing the processor!
  - Ability to “fix” buggy hardware implementations
  
- Examples
  - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
  - IBM System z: Similar to 370/145.
    - Heller and Farrell, “[Millicode in an IBM zSeries processor](#),” IBM JR&D, May/Jul 2004.
  - B1700 microcode can be updated while the processor is running
    - User-microprogrammable machine!

# An Example Microcoded Multi-Cycle MIPS Design

---

- P&H, Appendix D
- Any ISA can be implemented this way
- We will not cover this in class
- However, you can do the extra credit assignment for Lab 2
  - Partial credit even if your full design does not work
  - Maximum 4% of your grade in the course

# A Microprogrammed MIPS

## Design: Lab 2 Extra Credit



# Microcoded Multi-Cycle MIPS Design

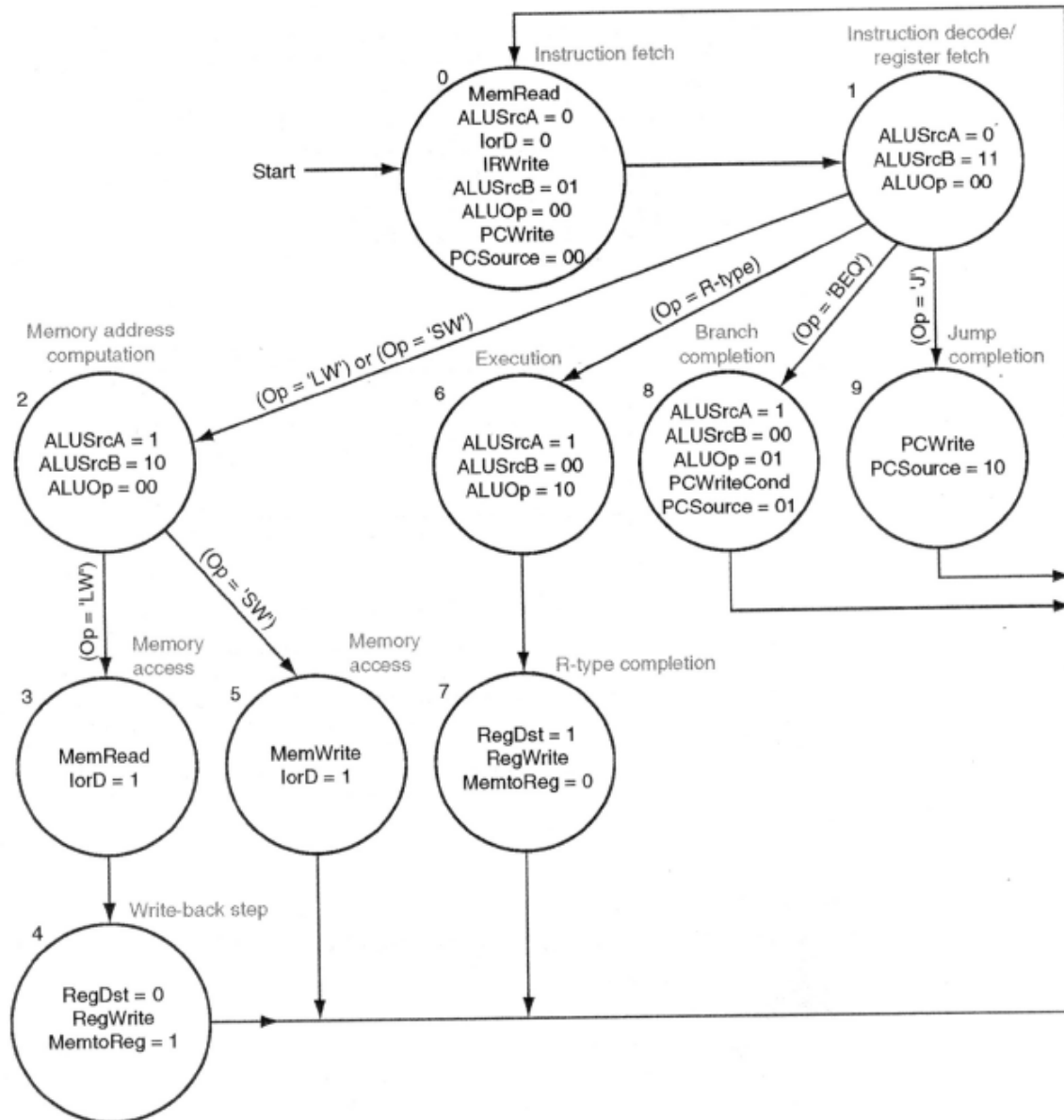
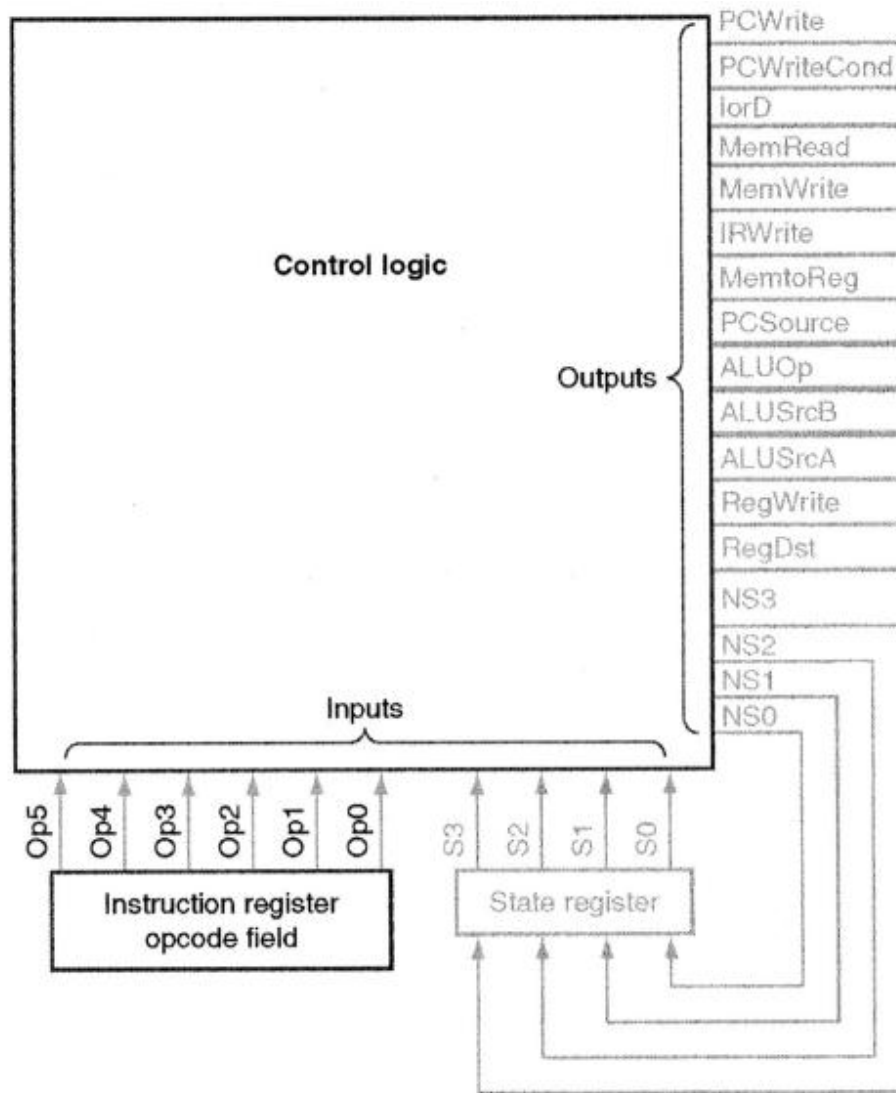


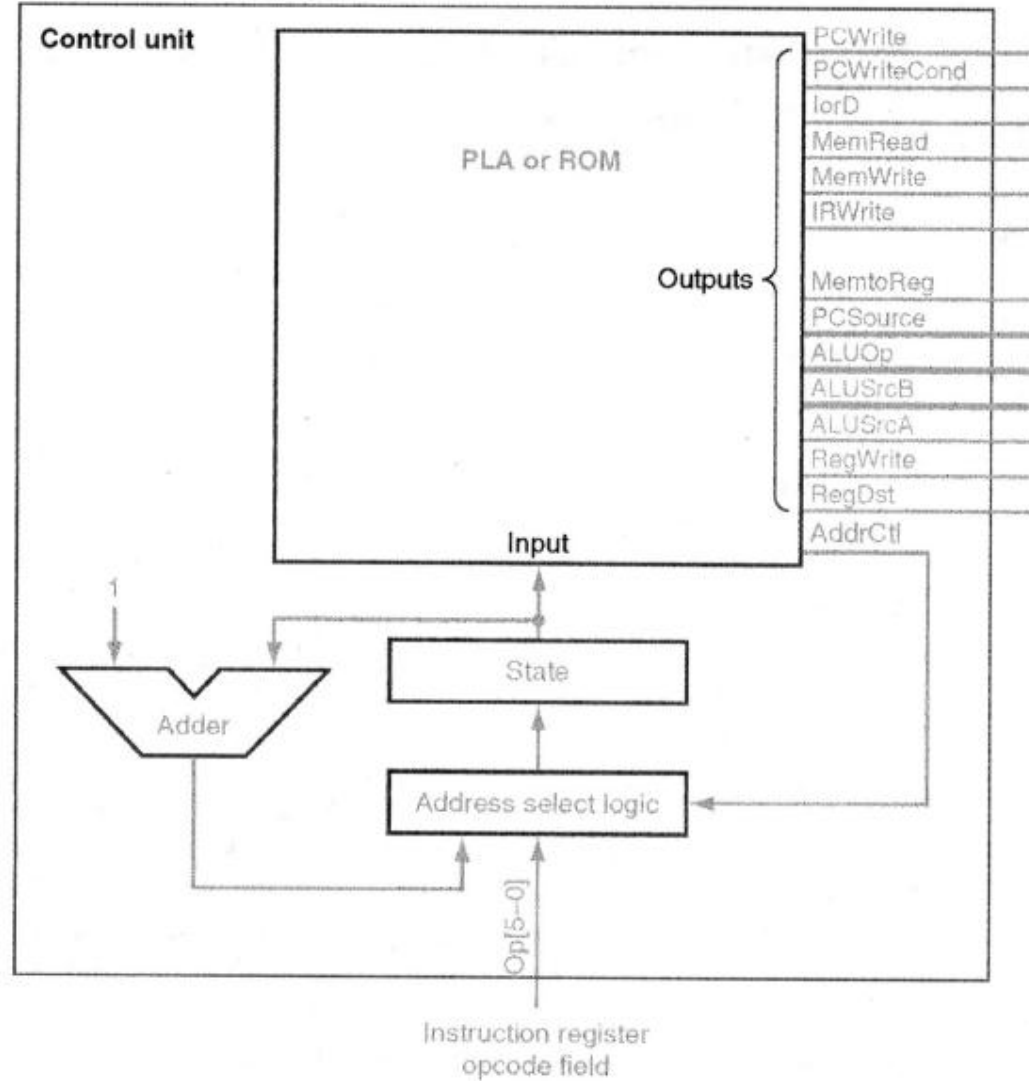
FIGURE D.3.1 The finite-state diagram for multicycle control.

# Control Logic for MIPS FSM



**FIGURE D.3.2** The control unit for MIPS will consist of some control logic and a register to hold the state. The state register is written at the active clock edge and is stable during the clock cycle.

# Microprogrammed Control for MIPS FSM

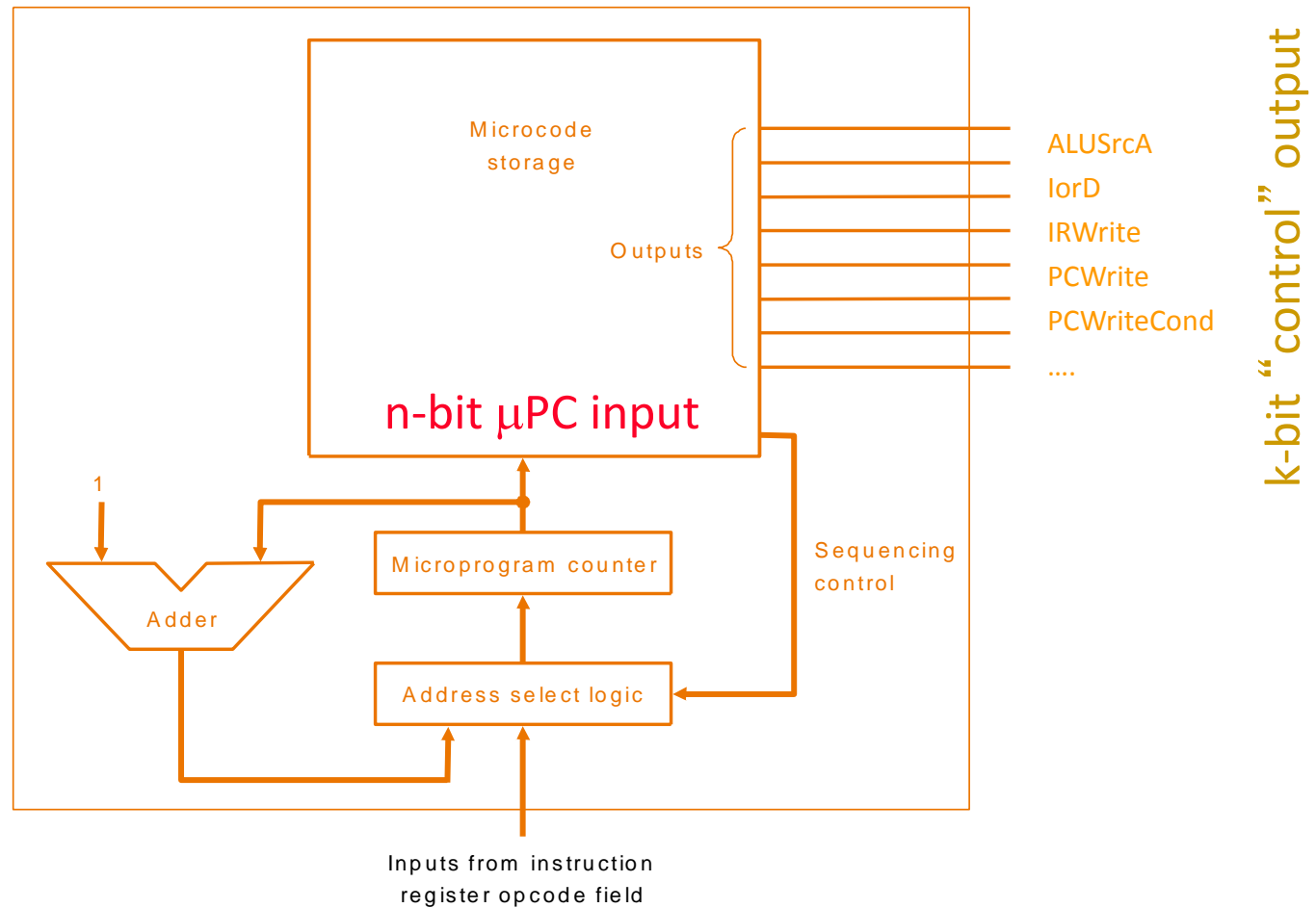


**FIGURE D.4.1 The control unit using an explicit counter to compute the next state.** In this control unit, the next state is computed using a counter (at least in some states). By comparison, Figure D.3.2 encodes the next state in the control logic for every state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.

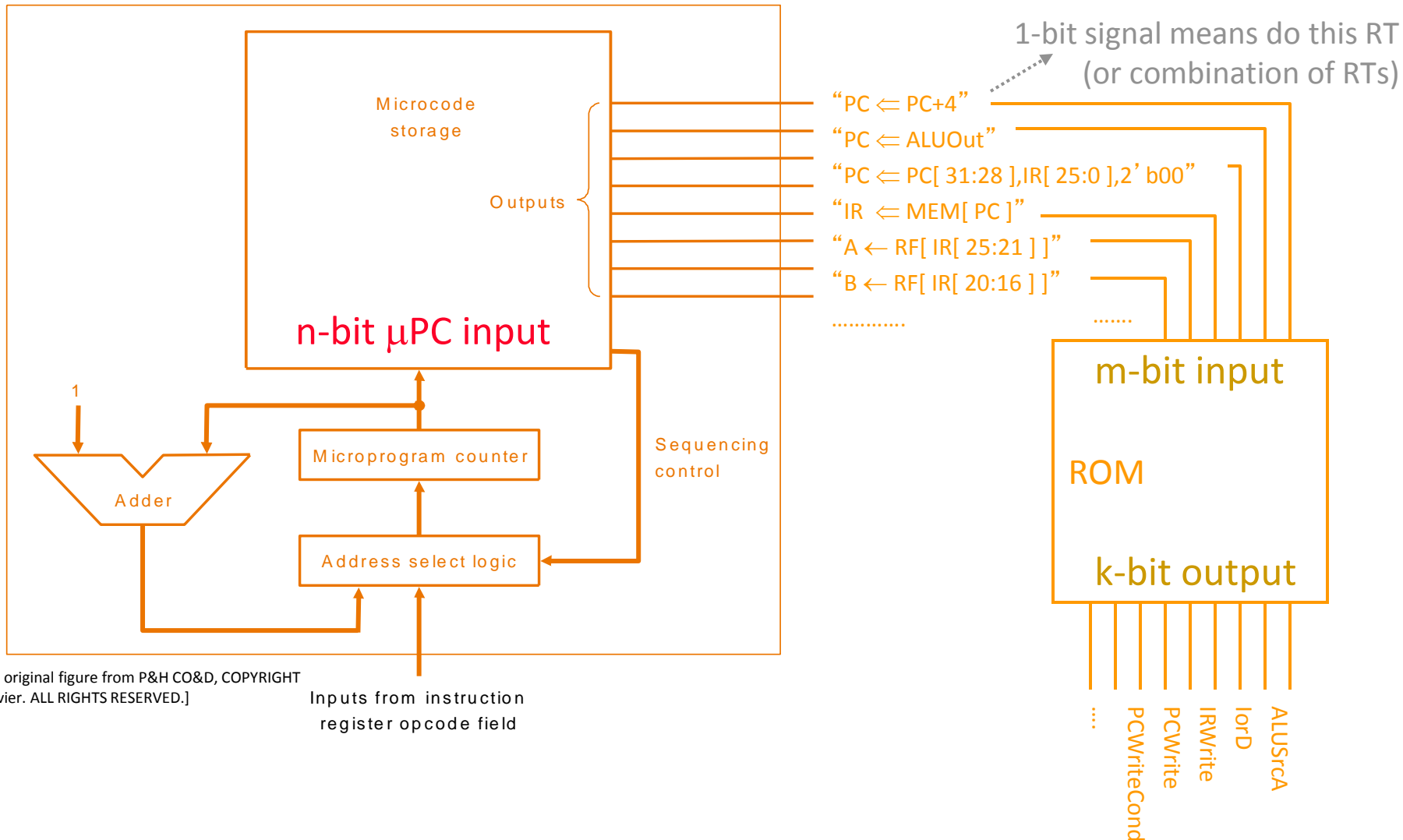
# End: A Microprogrammed MIPS

## Design: Lab 2 Extra Credit

# Horizontal Microcode



# Vertical Microcode



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

Inputs from instruction register opcode field

If done right (i.e.,  $m \ll n$ , and  $m \ll k$ ), two ROMs together ( $2^n \times m + 2^m \times k$  bit) should be smaller than horizontal microcode ROM ( $2^n \times k$  bit)

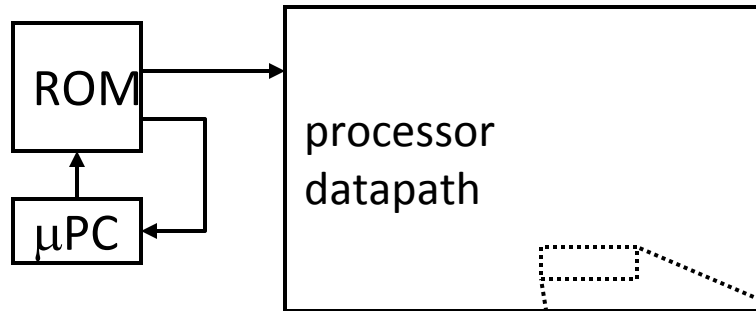
# Nanocode and Millicode

---

- *Nanocode*: a level below traditional  $\mu$ code
  - $\mu$ programmed control for sub-systems (e.g., a complicated floating-point module) that acts as a slave in a  $\mu$ controlled datapath
- *Millicode*: a level above traditional  $\mu$ code
  - ISA-level subroutines that can be called by the  $\mu$ controller to handle complicated operations and system functions
  - E.g., Heller and Farrell, “[Millicode in an IBM zSeries processor](#),” IBM JR&D, May/Jul 2004.
- In both cases, we avoid complicating the main  $\mu$ controller
- You can think of these as “microcode” at different levels of abstraction

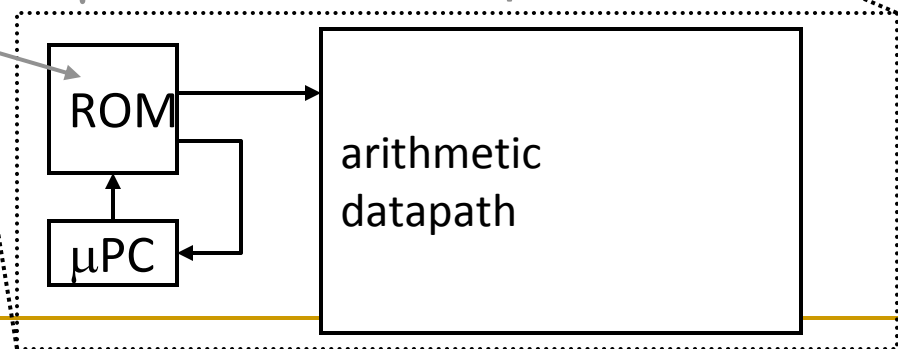
# Nanocode Concept Illustrated

a “ $\mu$ coded” processor implementation



We refer to this as “nanocode” when a  $\mu$ coded subsystem is embedded in a  $\mu$ coded system

a “ $\mu$ coded” FPU implementation





# 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: Concurrency → throughput (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: Basic Idea

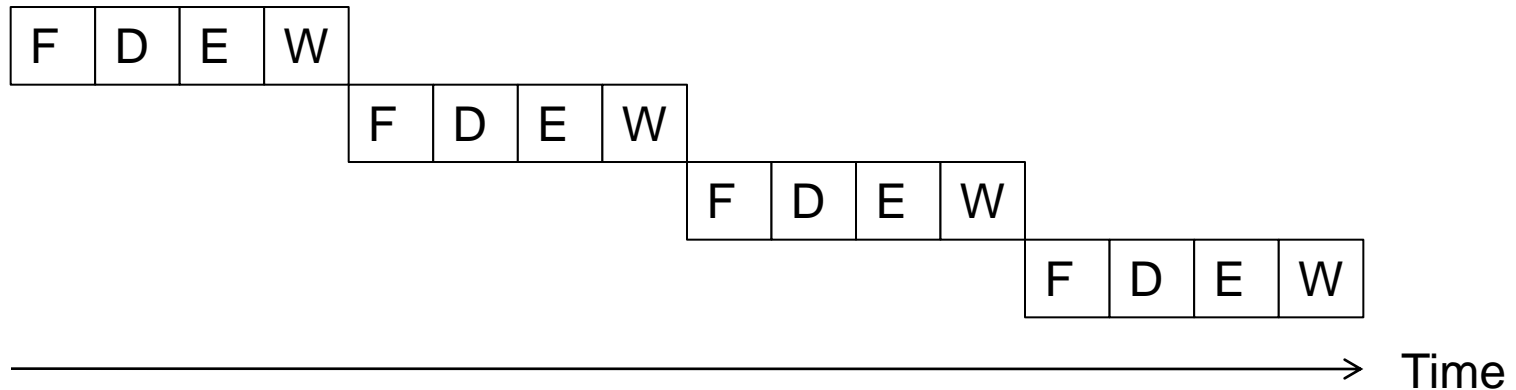
---

- 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...

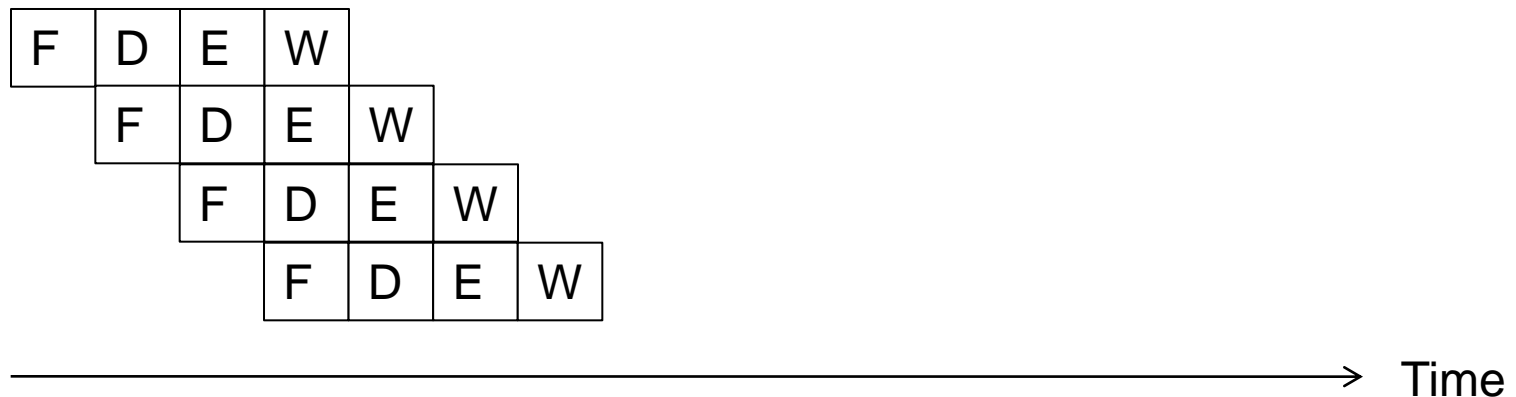
# Example: Execution of Four Independent ADDs

---

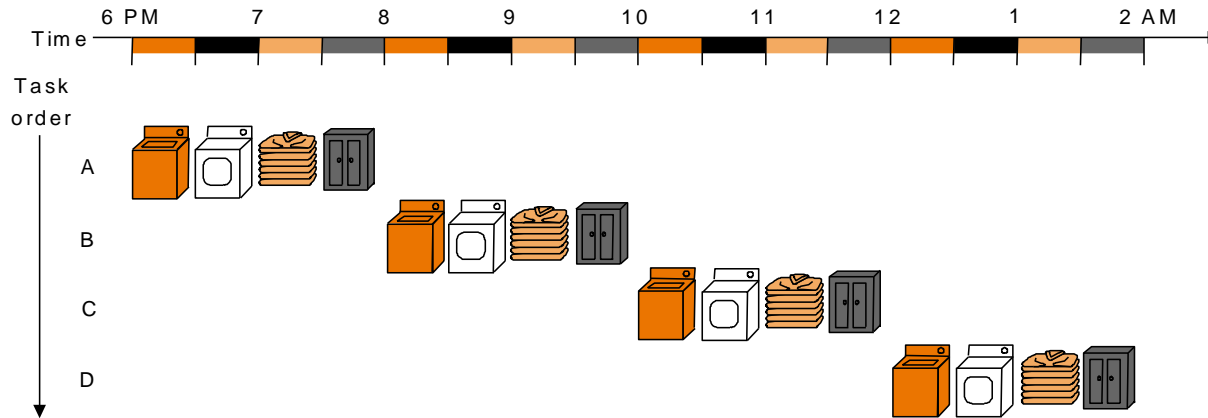
- Multi-cycle: 4 cycles per instruction



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

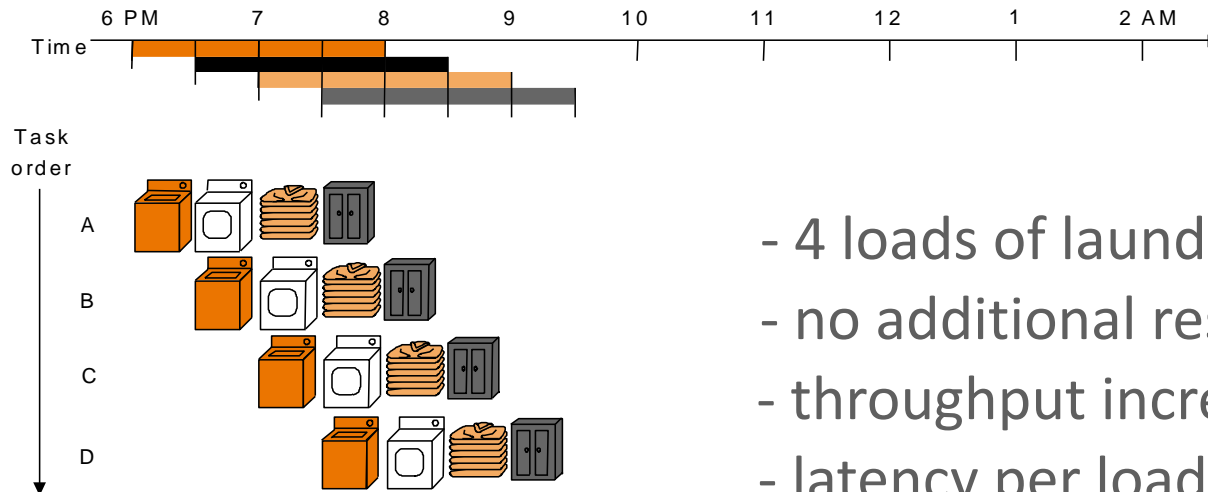
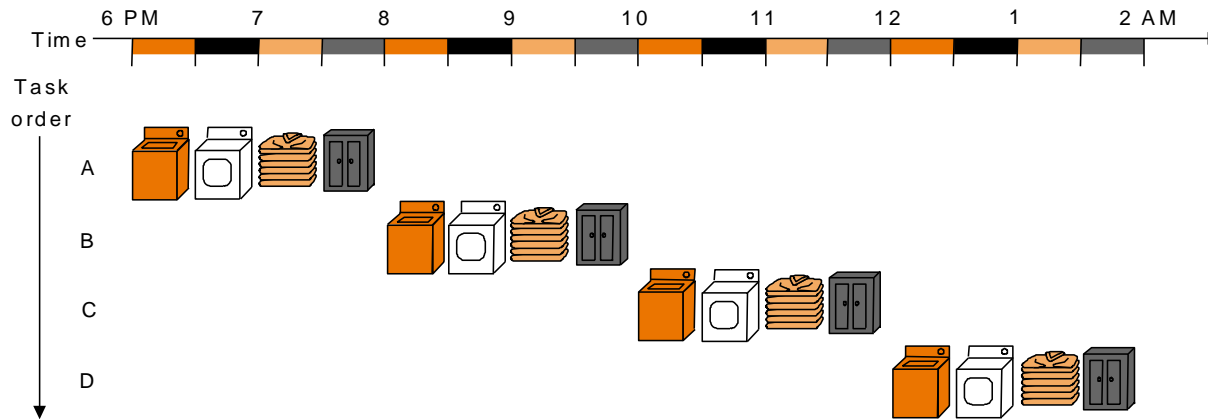


# The Laundry Analogy



- “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

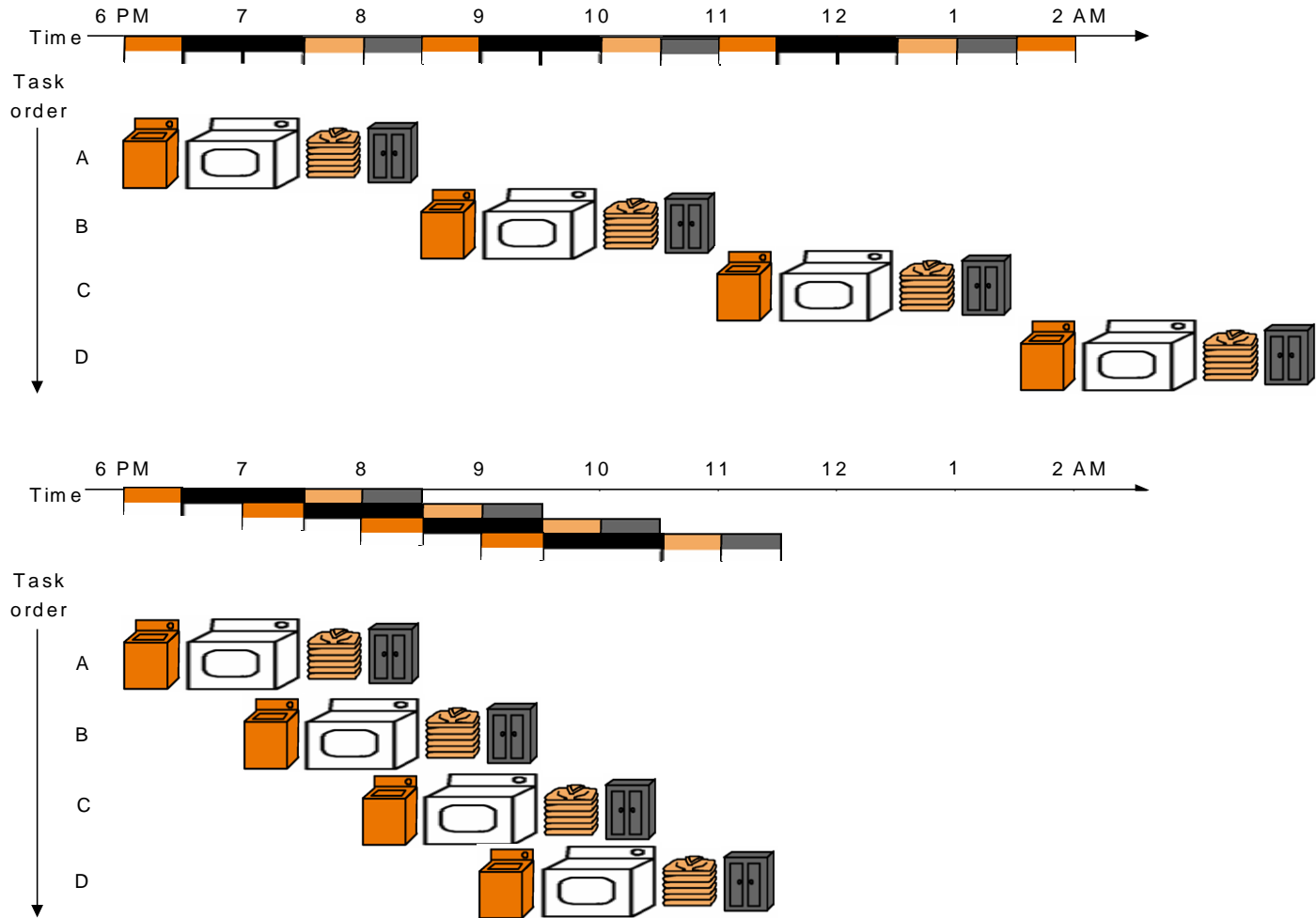
# Pipelining Multiple Loads of Laundry



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

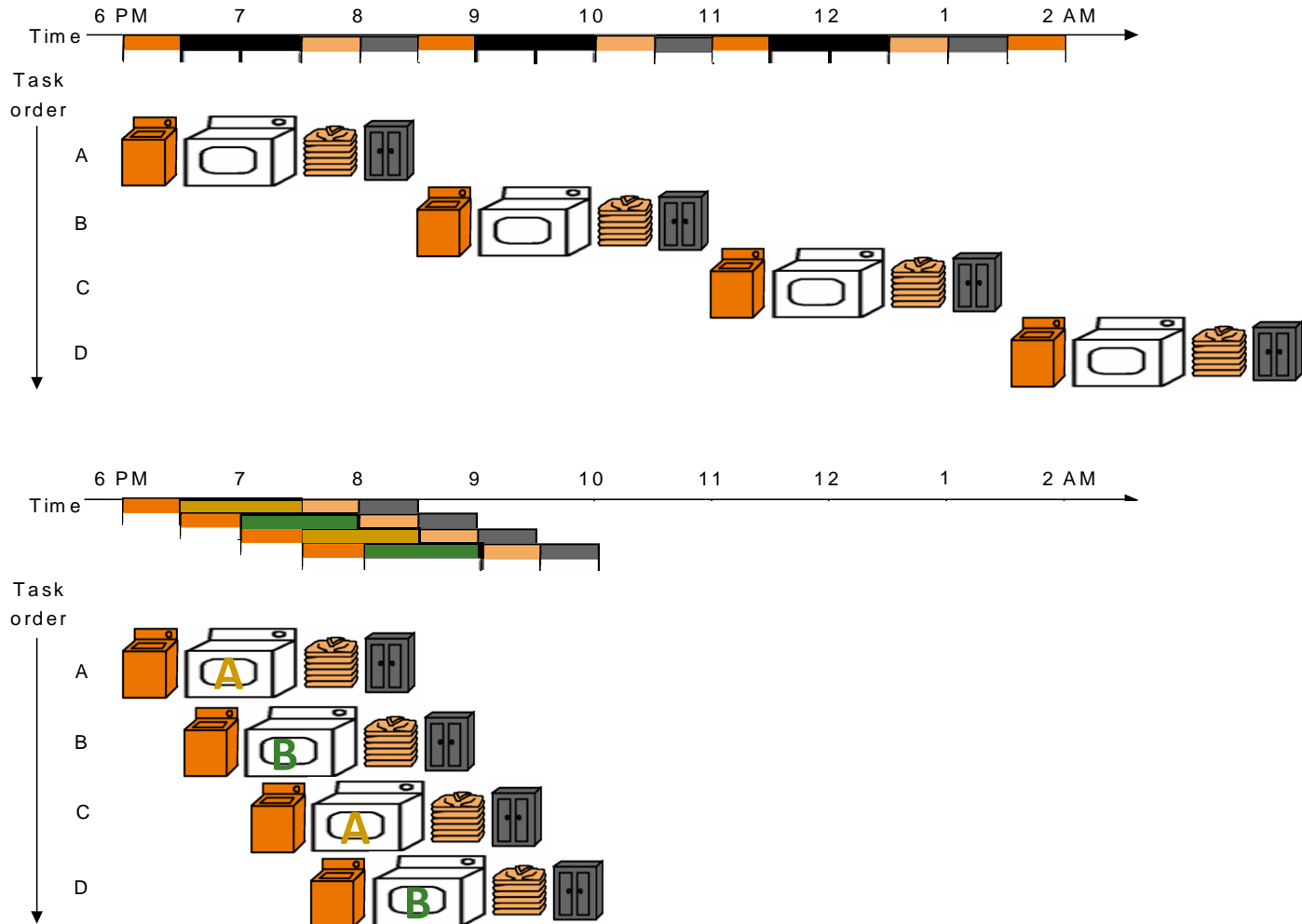


# Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

# Pipelining Multiple Loads of Laundry: In Practice



Throughput restored (2 loads per hour) using 2 dryers

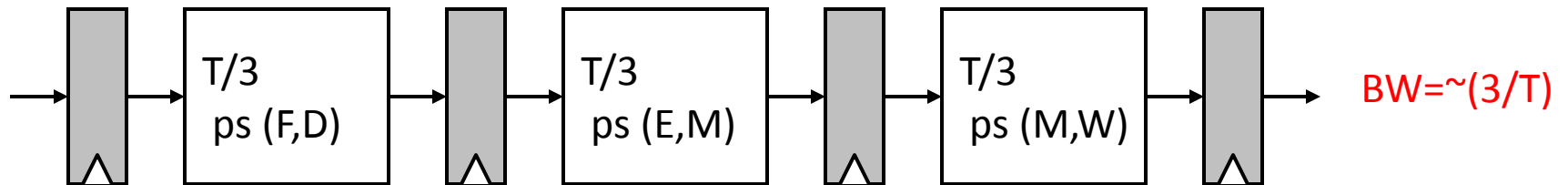
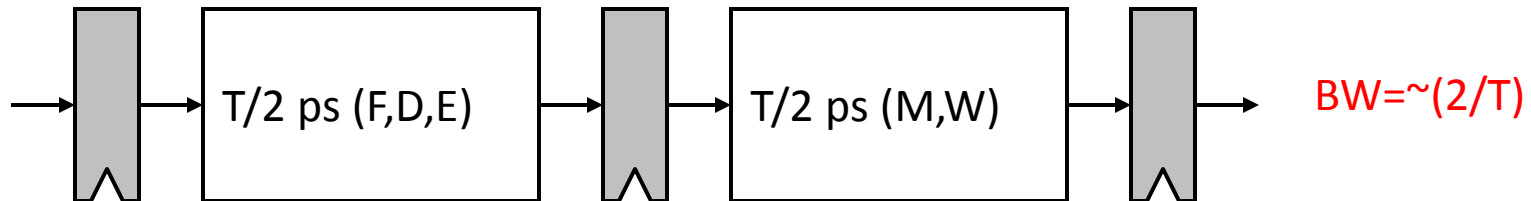
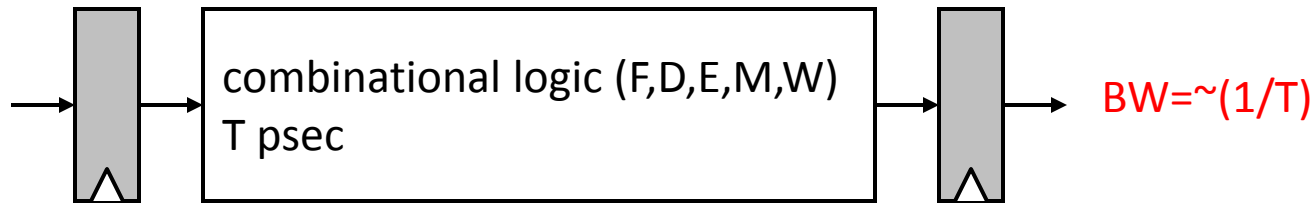
# An Ideal Pipeline

---

- 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
- 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  $T$

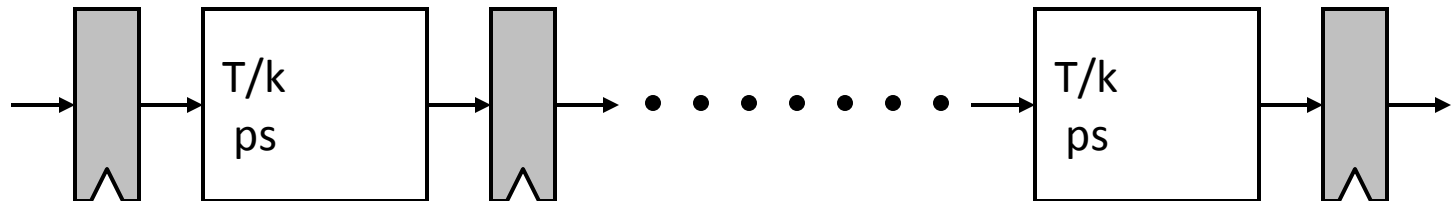
$$BW = 1/(T+S) \text{ where } S = \text{latch delay}$$



- k-stage pipelined version

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

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



# More Realistic Pipeline: Cost

---

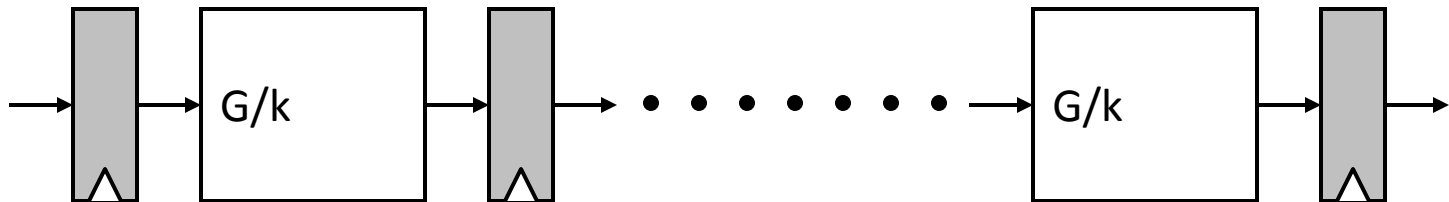
- Nonpipelined version with combinational cost  $G$

$\text{Cost} = G + L$  where  $L$  = latch cost



- $k$ -stage pipelined version

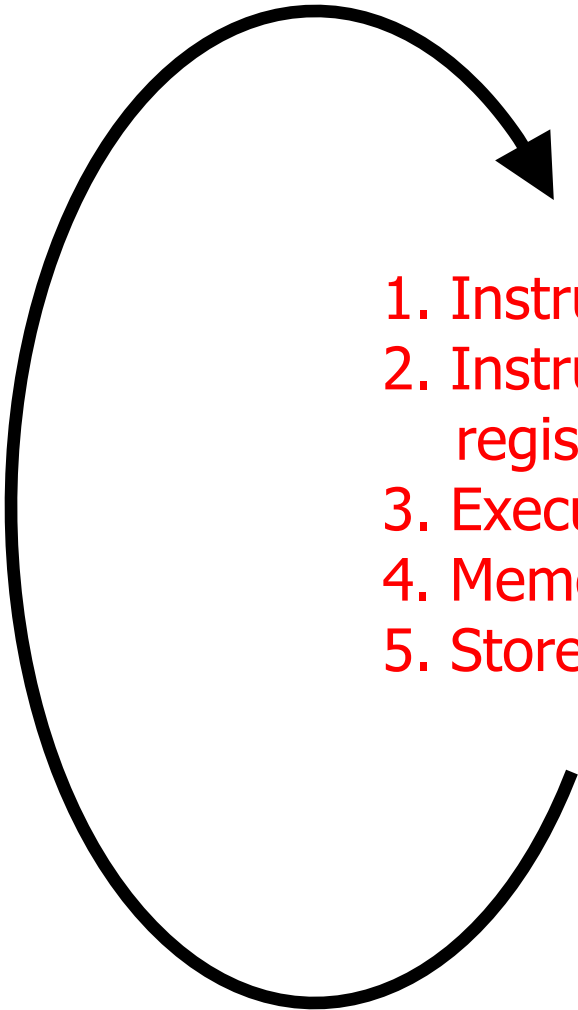
$\text{Cost}_{k\text{-stage}} = G + Lk$



# Pipelining Instruction Processing

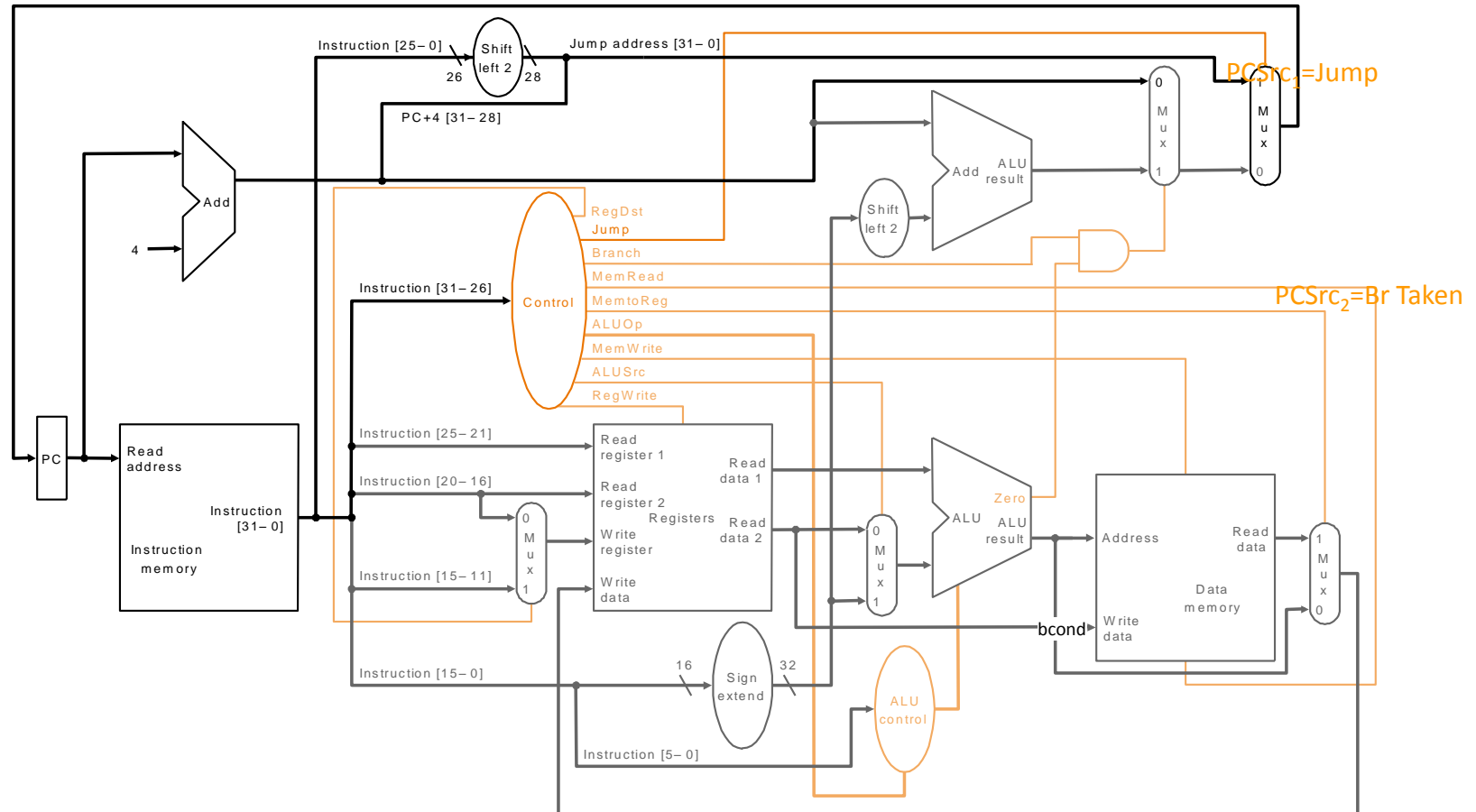
# Remember: The Instruction Processing Cycle

---

- 
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)



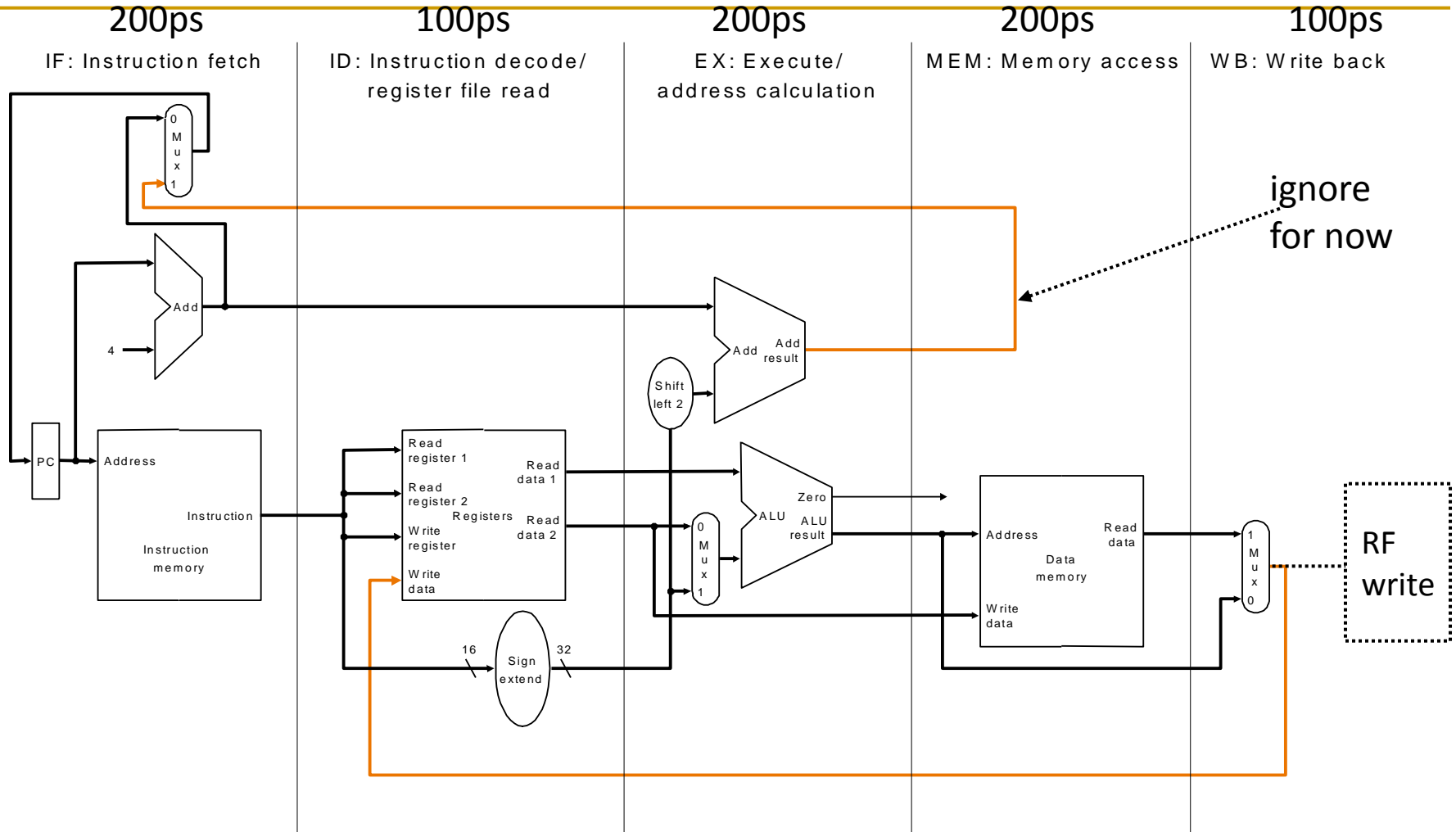
# Remember the Single-Cycle Uarch



ALU operation



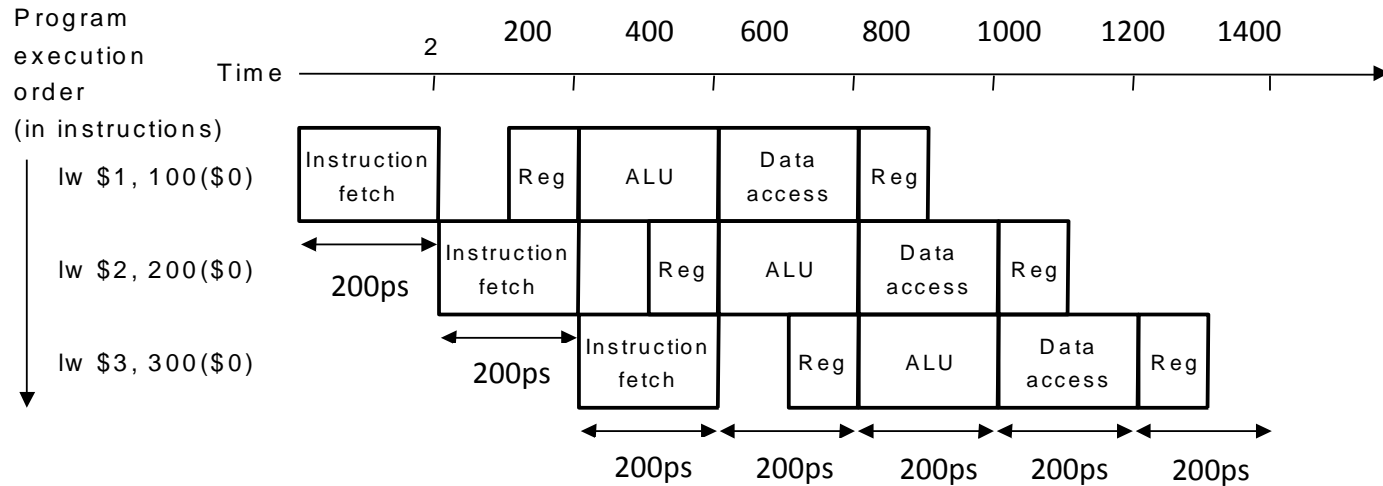
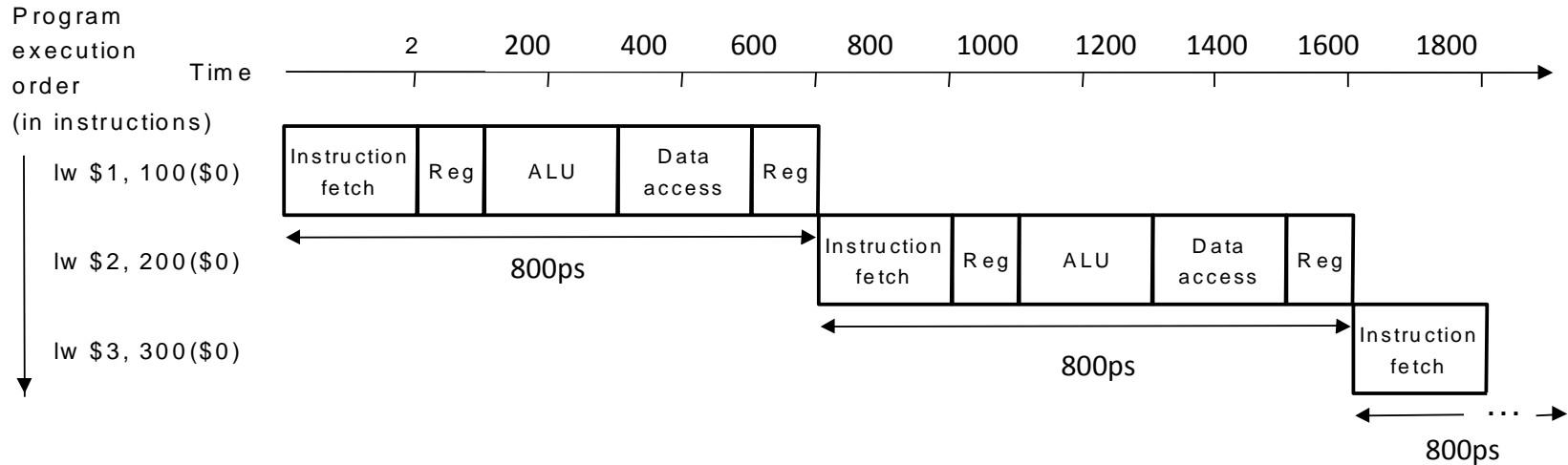
# Dividing Into Stages



Is this the correct partitioning?

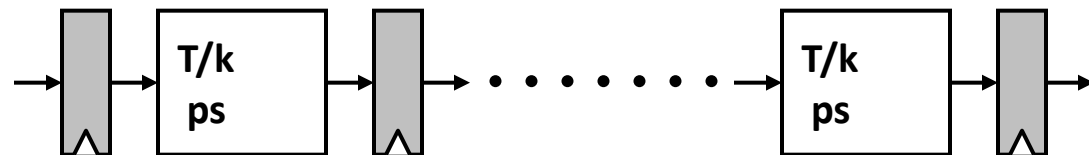
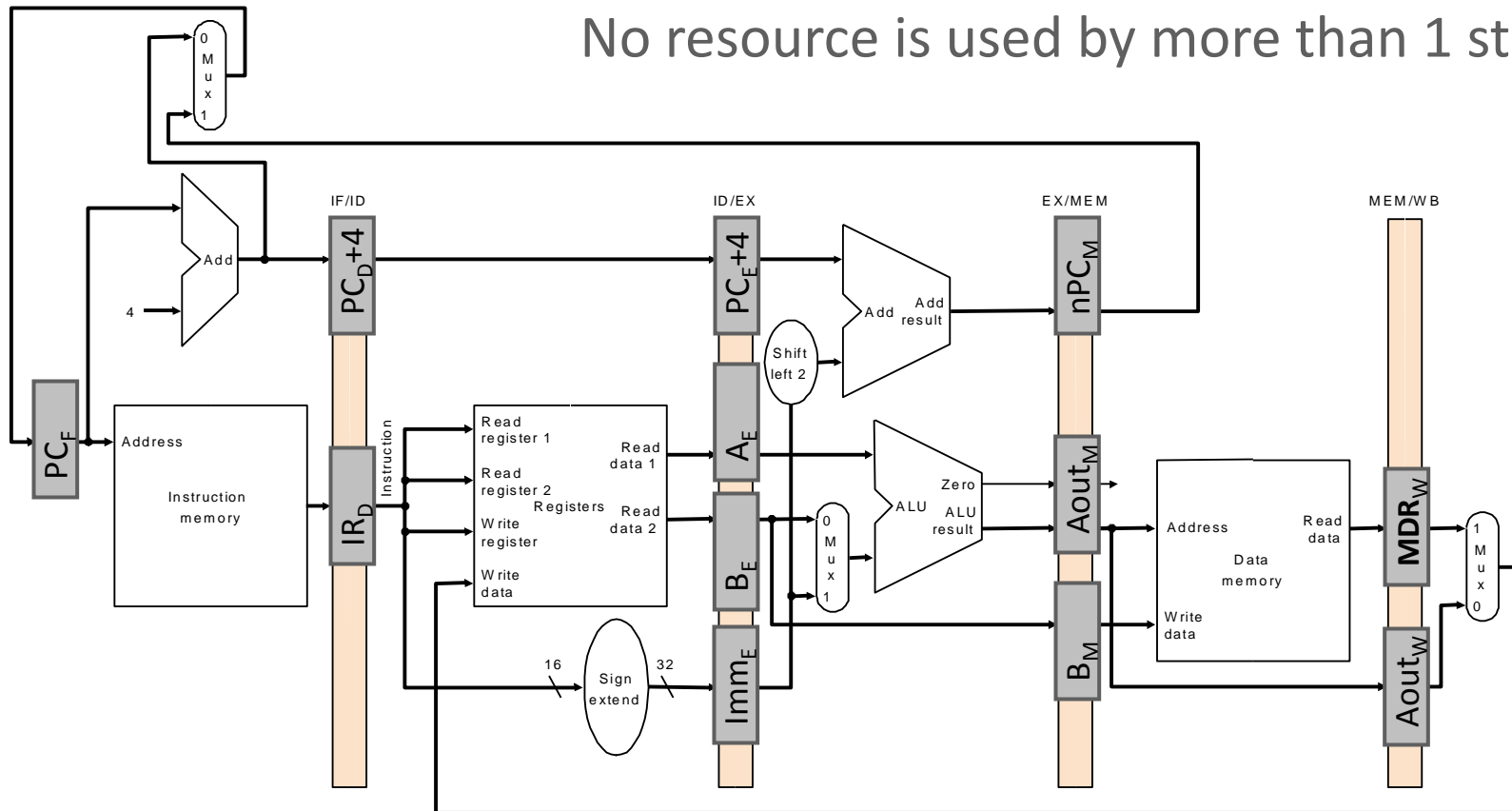
Why not 4 or 6 stages? Why not different boundaries?

# Instruction Pipeline Throughput



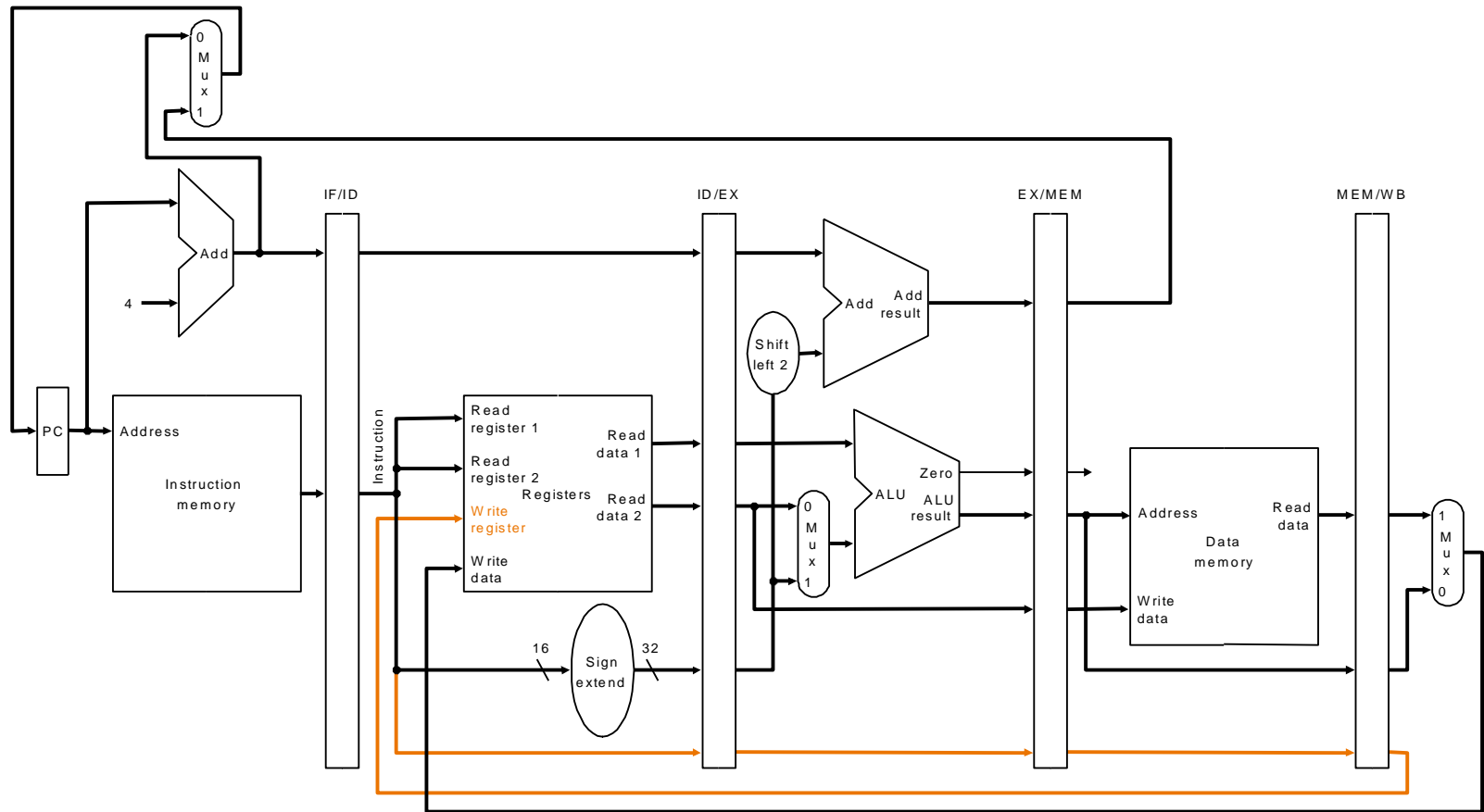
5-stage speedup is 4, not 5 as predicated by the ideal model. Why?

# Enabling Pipelined Processing: Pipeline Registers

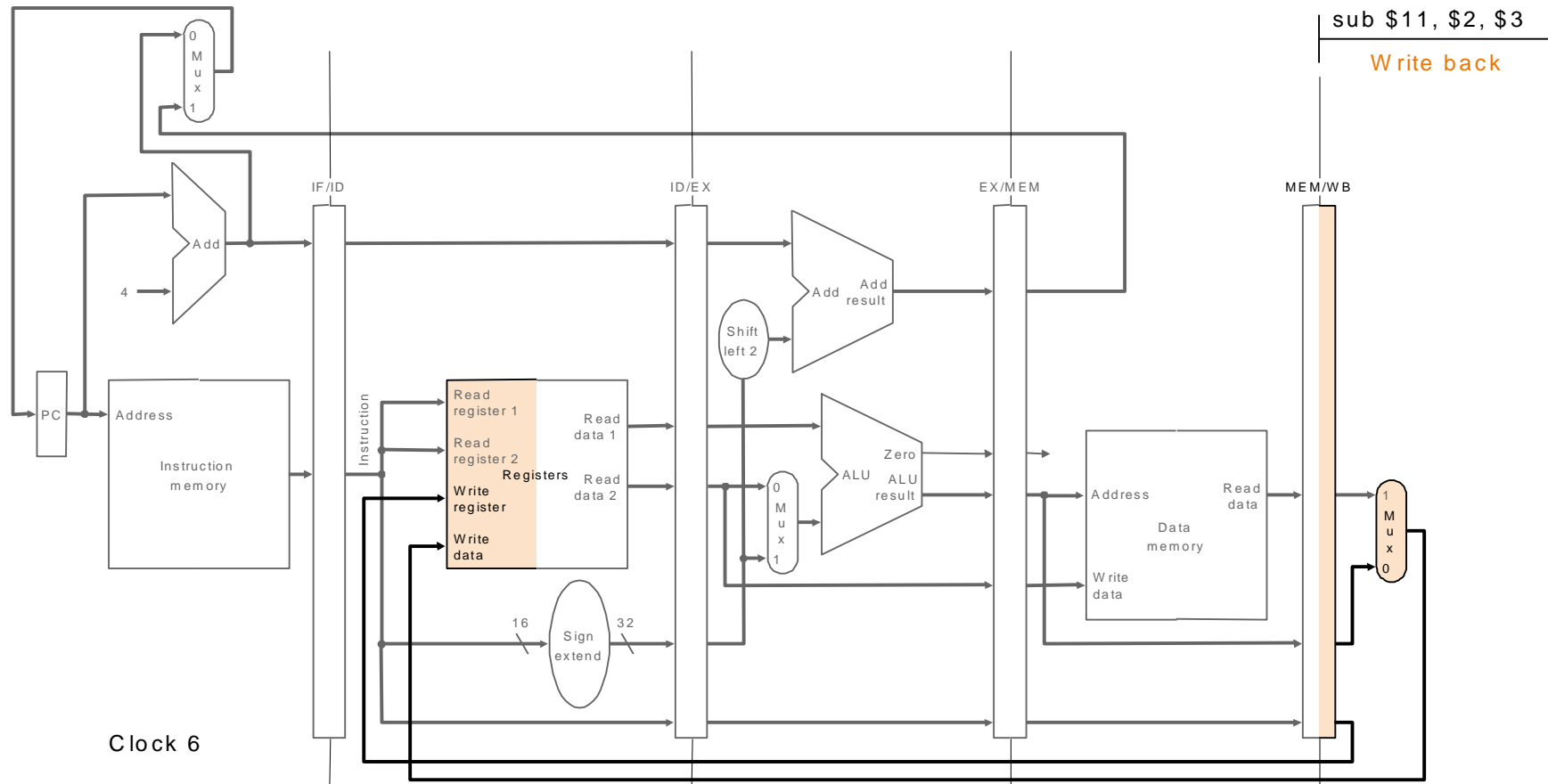


# Pipelined Operation Example

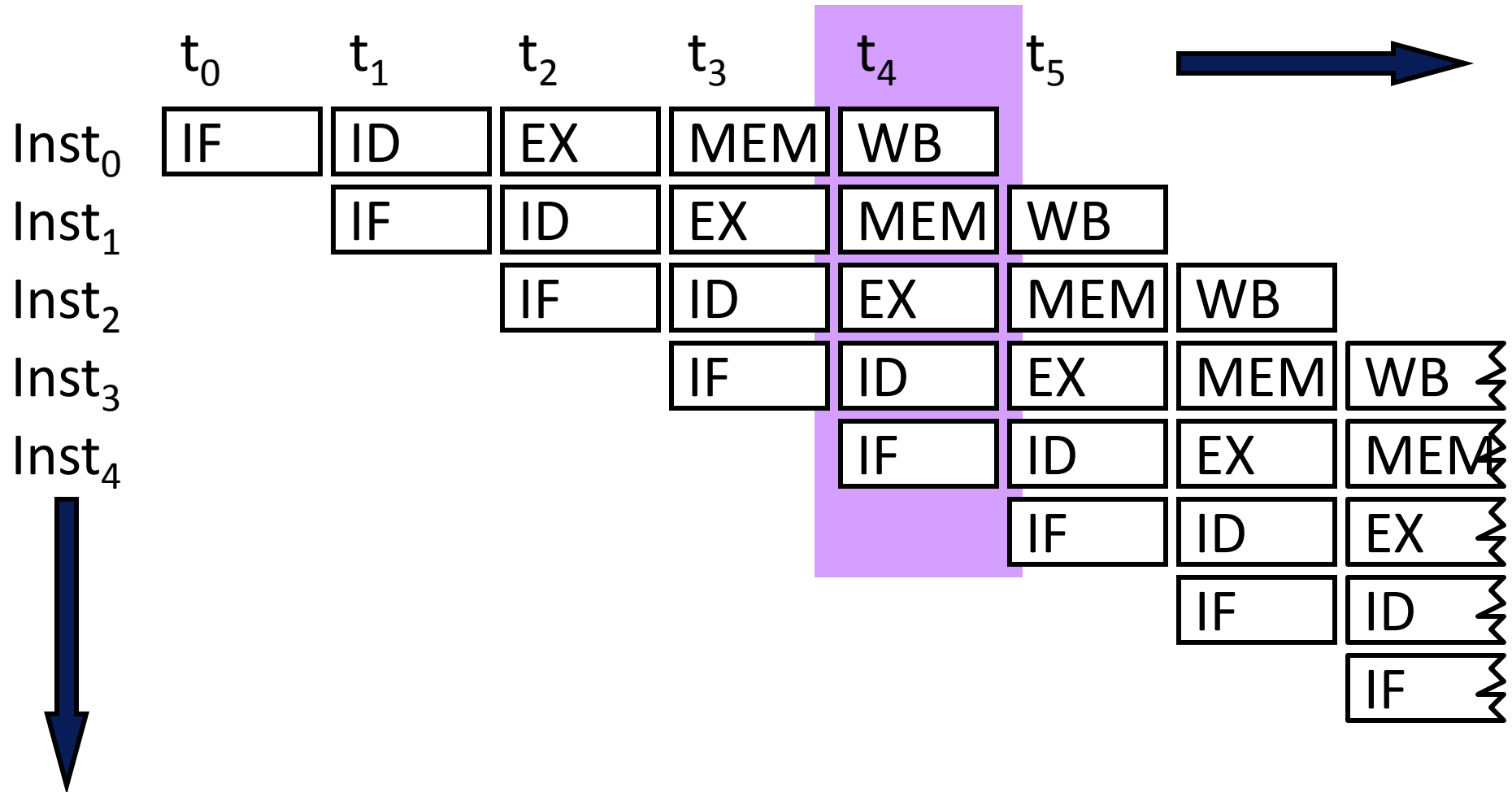
All instruction classes must follow the same path and timing through the pipeline stages. Any performance impact?



# Pipelined Operation Example



# Illustrating Pipeline Operation: Operation View



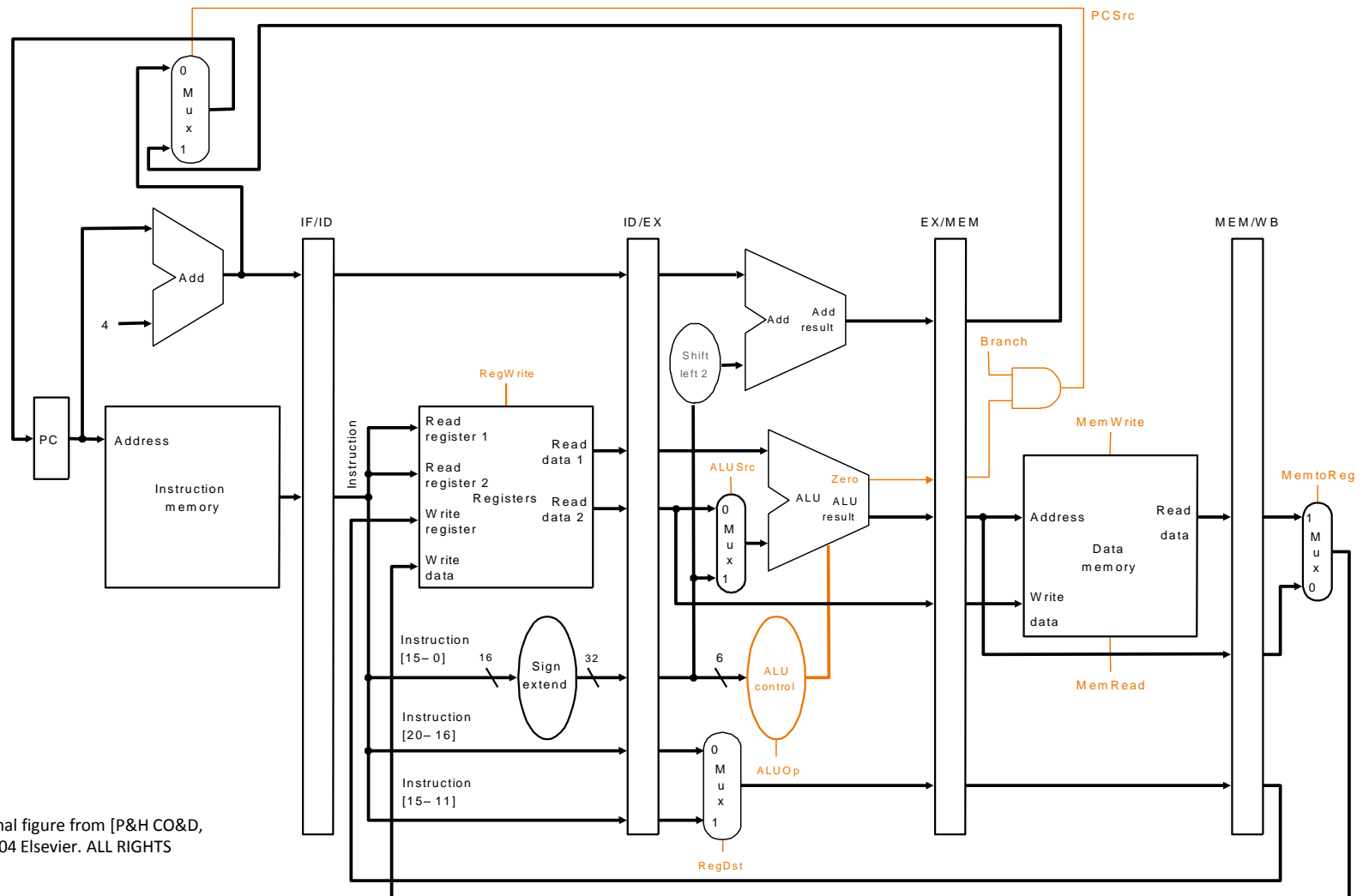
# Illustrating Pipeline Operation: Resource View

---

	t <sub>0</sub>	t <sub>1</sub>	t <sub>2</sub>	t <sub>3</sub>	t <sub>4</sub>	t <sub>5</sub>	t <sub>6</sub>	t <sub>7</sub>	t <sub>8</sub>	t <sub>9</sub>	t <sub>10</sub>
IF	I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>	I <sub>10</sub>
ID		I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>	I <sub>9</sub>
EX			I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>	I <sub>8</sub>
MEM				I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>	I <sub>7</sub>
WB					I <sub>0</sub>	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	I <sub>6</sub>



# Control Points in a Pipeline

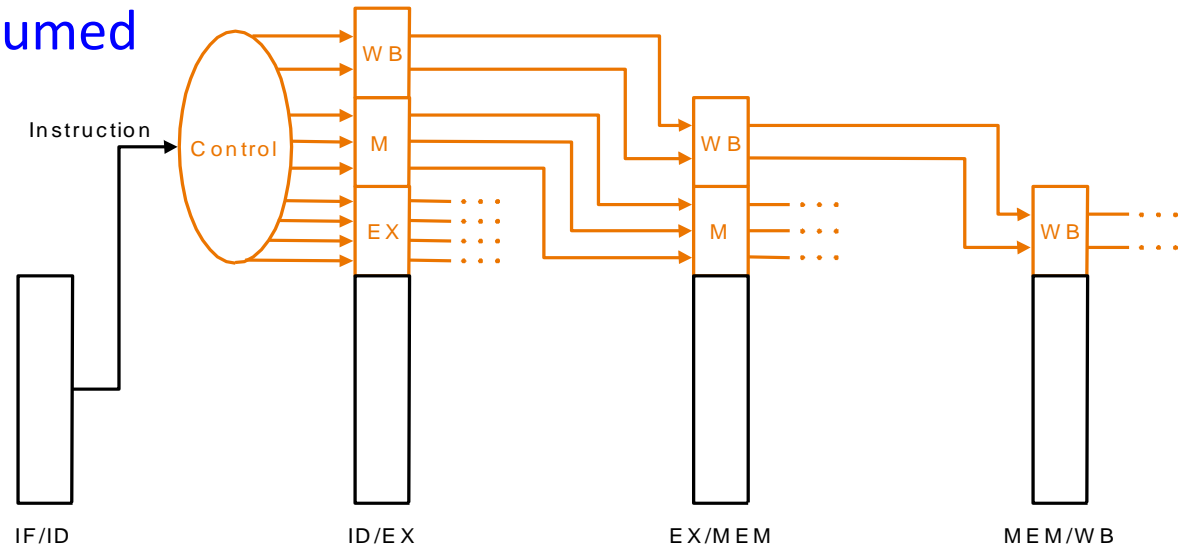


Based on original figure from [P&H CO&D,  
COPYRIGHT 2004 Elsevier. ALL RIGHTS  
RESERVED.]

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

# Control Signals in a Pipeline

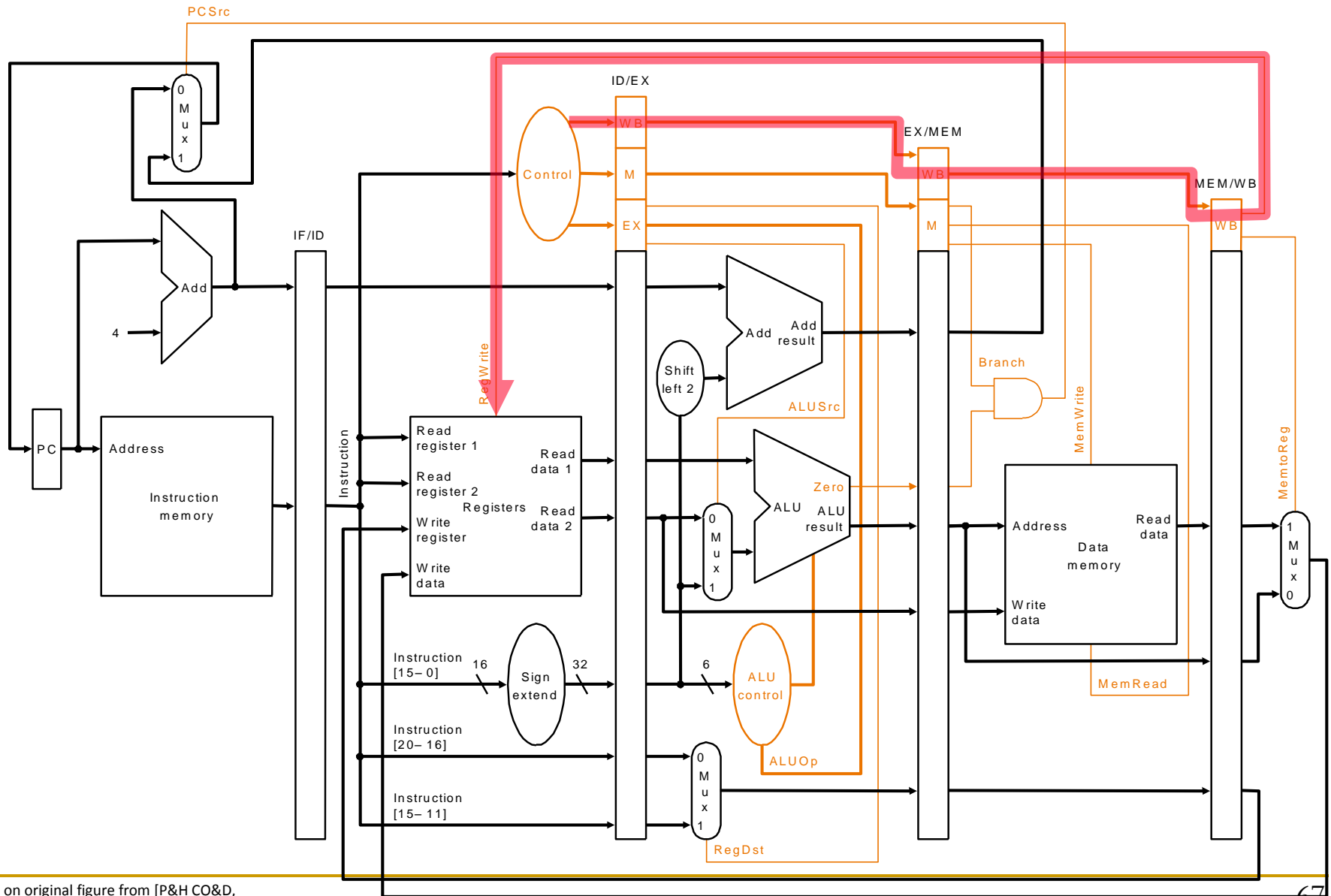
- For a given instruction
  - ❑ same control signals as single-cycle, but
  - ❑ control signals required at different cycles, depending on stage
- ⇒ decode once using the same logic as single-cycle and buffer control signals until consumed



- ⇒ or carry relevant “instruction word/field” down the pipeline and decode locally within each stage (still same logic)

Which one is better?

# Pipelined Control Signals



# 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
- 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”?

# Instruction Pipeline: Not An Ideal Pipeline

---

## ■ Identical operations ... NOT!

⇒ different instructions do not need all stages

- Forcing different instructions to go through the same multi-function pipe
- external fragmentation (some pipe stages idle for some instructions)

## ■ Uniform suboperations ... NOT!

⇒ difficult to balance the different pipeline stages

- Not all pipeline stages do the same amount of work
- internal fragmentation (some pipe stages are too-fast but 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 operates correctly
- Pipeline is not always moving (it stalls)

# 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

# Causes of Pipeline *Stalls*

---

- Resource contention
- Dependences (between instructions)
  - Data
  - Control
- Long-latency (multi-cycle) operations

# Dependences and Their Types

---

- Also called “dependency” or *less desirably* “hazard”
- Dependencies dictate ordering requirements between instructions
- Two types
  - Data dependence
  - Control dependence
- Resource contention is sometimes called resource dependence
  - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately



# 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?

# Data Dependences

---


- Types of data dependences
  - Flow dependence (true data dependence – read after write)
  - Output dependence (write after write)
  - Anti dependence (write after read)
  
- Which ones cause stalls in a pipelined machine?
  - For all of them, we need to ensure semantics of the program are 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
    - We will later see what we can do about them

# Data Dependence Types

---

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write  
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read  
(WAR)

Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write  
(WAW)

# 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