18-447
Computer Architecture
Lecture 5: Intro to Microarchitecture: Single-Cycle

Prof. Onur Mutlu
Carnegie Mellon University

Spring 2015, 1/26/2015

Agenda for Today & Next Few Lectures

- Start Microarchitecture
- Single-cycle Microarchitectures
- Multi-cycle Microarchitectures
- Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, …

Recap of Two Weeks and Last Lecture

- Computer Architecture Today and Basics (Lectures 1 & 2)
- Fundamental Concepts (Lecture 3)
- ISA basics and tradeoffs (Lectures 3 & 4)

- Last Lecture: ISA tradeoffs continued + MIPS ISA
  - Instruction length
  - Uniform vs. non-uniform decode
  - Number of registers
  - Addressing modes
  - Aligned vs. unaligned access
  - RISC vs. CISC properties
  - MIPS ISA Overview

Assignment for You

- Not to be turned in

- As you learn the MIPS ISA, think about what tradeoffs the designers have made
  - in terms of the ISA properties we talked about

- And, think about the pros and cons of design choices
  - In comparison to ARM, Alpha
  - In comparison to x86, VAX

- And, think about the potential mistakes
  - Branch delay slot?
  - Load delay slot?
  - No FP, no multiply, MIPS (initial)

Look Backward

Food for Thought for You

- How would you design a new ISA?
- Where would you place it?
- What design choices would you make in terms of ISA properties?
- What would be the first question you ask in this process?
  - "What is my design point?"

Look Forward & Up

Review: Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- SIMD (single instruction multiple data) vs. SISD
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- …

Think Programmer vs. (Micro)architect

Review: A Note on RISC vs. CISC

- Usually, …

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

Now That We Have an ISA

- How do we implement it?
- i.e., how do we design a system that obeys the hardware/software interface?
- Aside: "System" can be solely hardware or a combination of hardware and software
  - Remember "Translation of ISAs"
  - A virtual ISA can be converted by "software" into an implementation ISA
- We will assume "hardware" for most lectures

Implementing the ISA: Microarchitecture Basics

How Does a Machine Process Instructions?

- What does processing an instruction mean?
- Remember the von Neumann model

AS = Architectural (programmer visible) state before an instruction is processed

↓

Process instruction

↓

AS' = Architectural (programmer visible) state after an instruction is processed

- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The “Process instruction” Step

- ISA specifies abstractly what AS’ should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between AS and AS’ during instruction execution
    - One state transition per instruction

- Microarchitecture implements how AS is transformed to AS’
  - There are many choices in implementation
  - We can have programmer-invisible state to optimize the speed of instruction execution: multiple state transitions per instruction
    - Choice 1: AS → AS’ (transform AS to AS’ in a single clock cycle)
    - Choice 2: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS’ (take multiple clock cycles to transform AS to AS’)

A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute
- Only combinational logic is used to implement instruction execution
  - *No intermediate, programmer-invisible state updates*

AS = Architectural (programmer visible) state
at the beginning of a clock cycle

↓

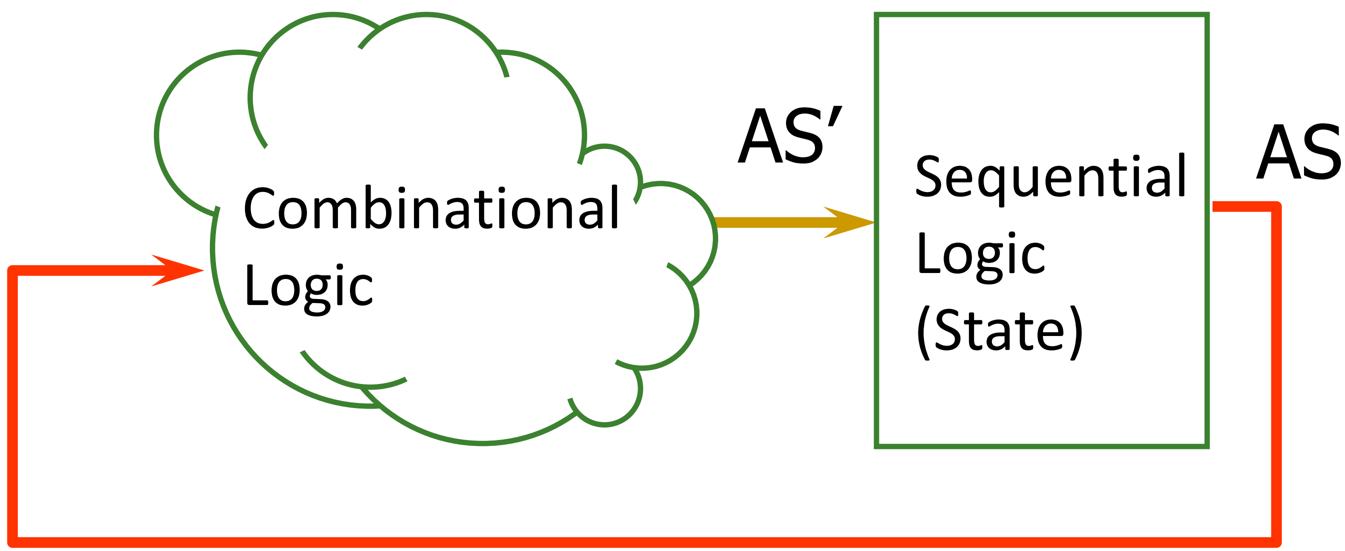
Process instruction in one clock cycle

↓

AS' = Architectural (programmer visible) state
at the end of a clock cycle

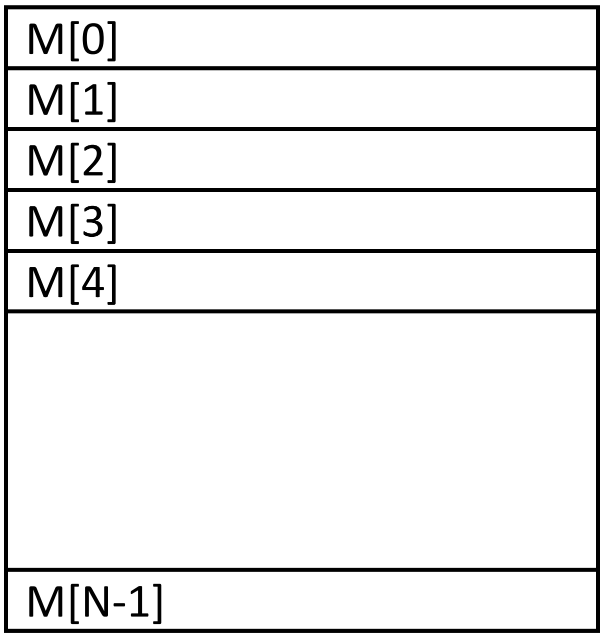
A Very Basic Instruction Processing Engine

- Single-cycle machine

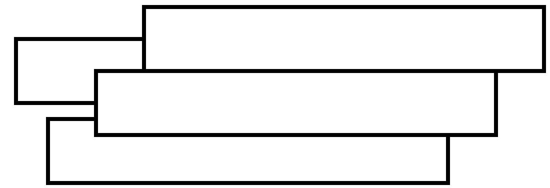


- What is the *clock cycle time* determined by?
- What is the *critical path* of the combinational logic determined by?

Remember: Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address



Registers
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

Single-cycle vs. Multi-cycle Machines

- Single-cycle machines
  - Each instruction takes a single clock cycle
  - All state updates made at the end of an instruction's execution
  - Big disadvantage: The slowest instruction determines cycle time → long clock cycle time

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made only at the end of an instruction's execution
  - Advantage over single-cycle: The slowest "stage" determines cycle time

- Both single-cycle and multi-cycle machines literally follow the von Neumann model at the microarchitecture level

Instruction Processing "Cycle"

- Instructions are processed under the direction of a "control unit" step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six phases:

- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

- Not all instructions require all six stages (see P&P Ch. 4)

Instruction Processing "Cycle" vs. Machine Clock Cycle

- Single-cycle machine:
  - All six phases of the instruction processing cycle take a *single machine clock cycle* to complete

- Multi-cycle machine:
  - All six phases of the instruction processing cycle can take *multiple machine clock cycles* to complete
  - In fact, each phase can take multiple clock cycles to complete

Instruction Processing Viewed Another Way

- Instructions transform Data (AS) to Data' (AS')
- This transformation is done by functional units
  - Units that "operate" on data
- These units need to be told what to do to the data

- An instruction processing engine consists of two components
  - Datapath: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - Control logic: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

Single-cycle vs. Multi-cycle: Control & Data

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)

- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)

- We will see the difference clearly in *microprogrammed multi-cycle microarchitectures*

Many Ways of Datapath and Control Design

- There are many ways of designing the data path and control logic

- Single-cycle, multi-cycle, pipelined datapath and control
- Single-bus vs. multi-bus datapaths
  - See your homework 2 question
- Hardwired/combinational vs. microcoded/microprogrammed control
  - Control signals generated by combinational logic versus
  - Control signals stored in a memory structure

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}
- Execution time of a program
  - Sum over all instructions [{CPI} x {clock cycle time}]
  - {# of instructions} x {Average CPI} x {clock cycle time}

- Single cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

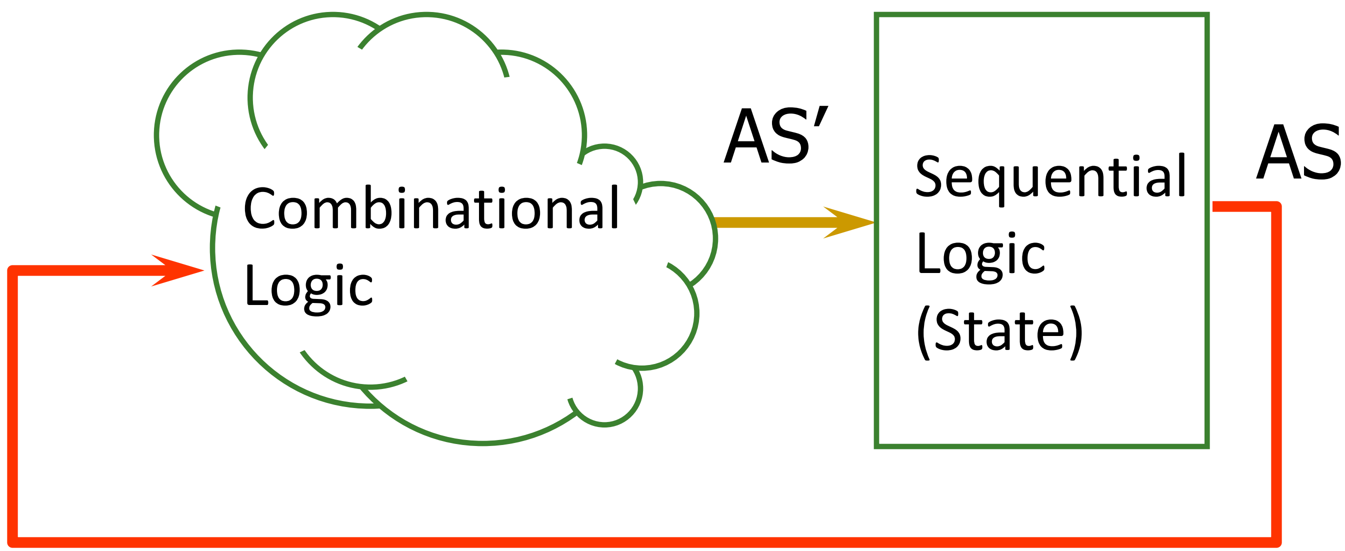
Now, we have two degrees of freedom to optimize independently

A Single-Cycle Microarchitecture

A Closer Look

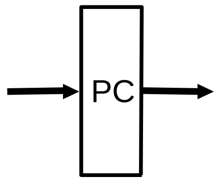
Remember…

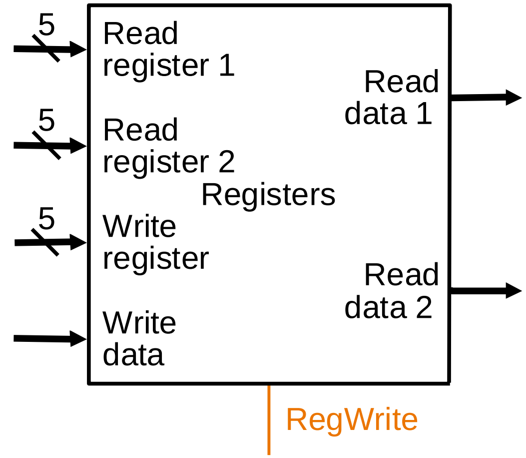
- Single-cycle machine

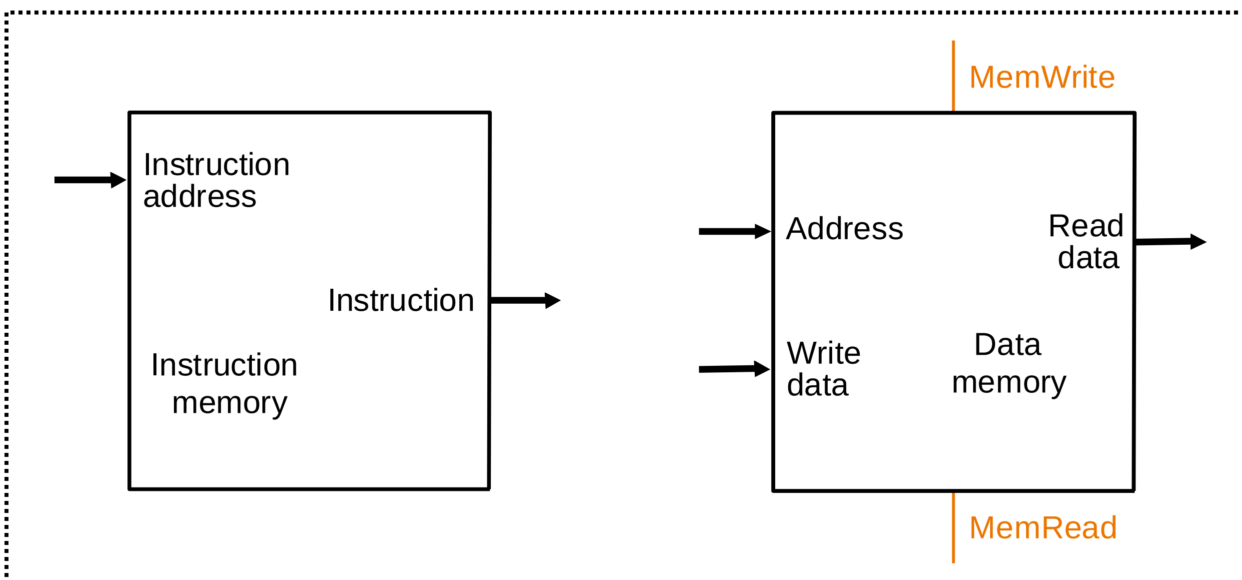


Let’s Start with the State Elements

- Data and control inputs





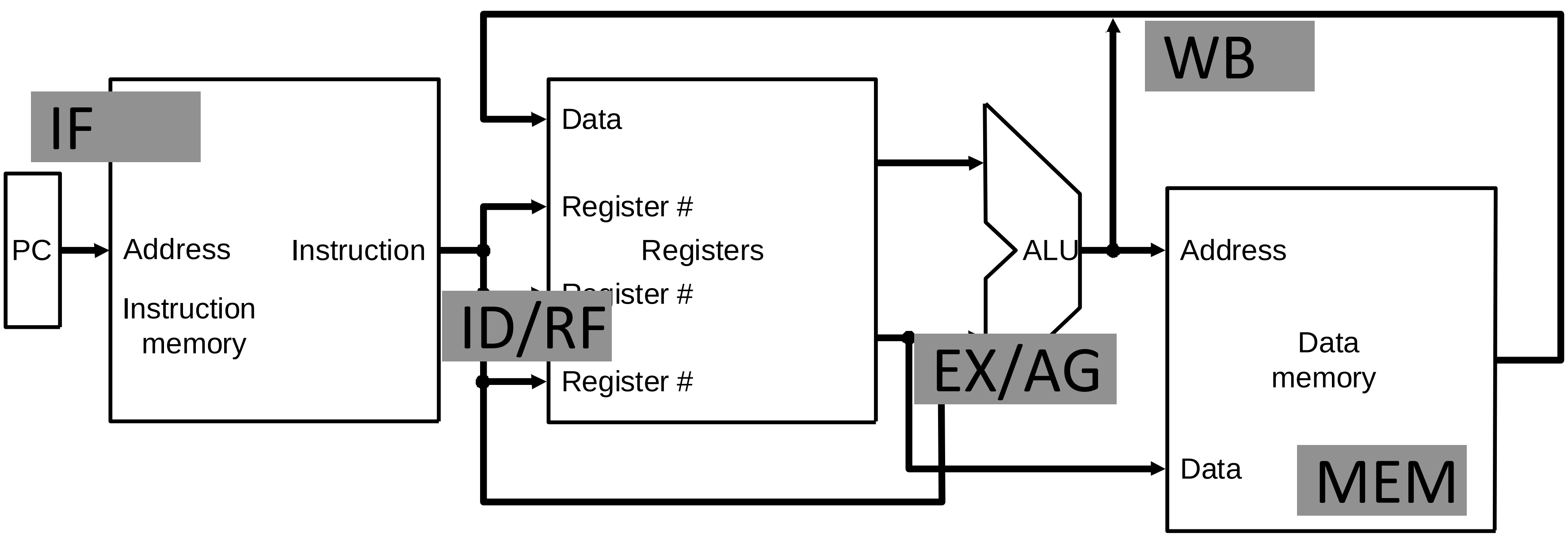


For Now, We Will Assume

- "Magic" memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is asserted
    - Cannot affect read output in between clock edges
- Single-cycle, synchronous memory
  - Contrast this with memory that tells when the data is ready
  - i.e., Ready bit: indicating the read or write is done

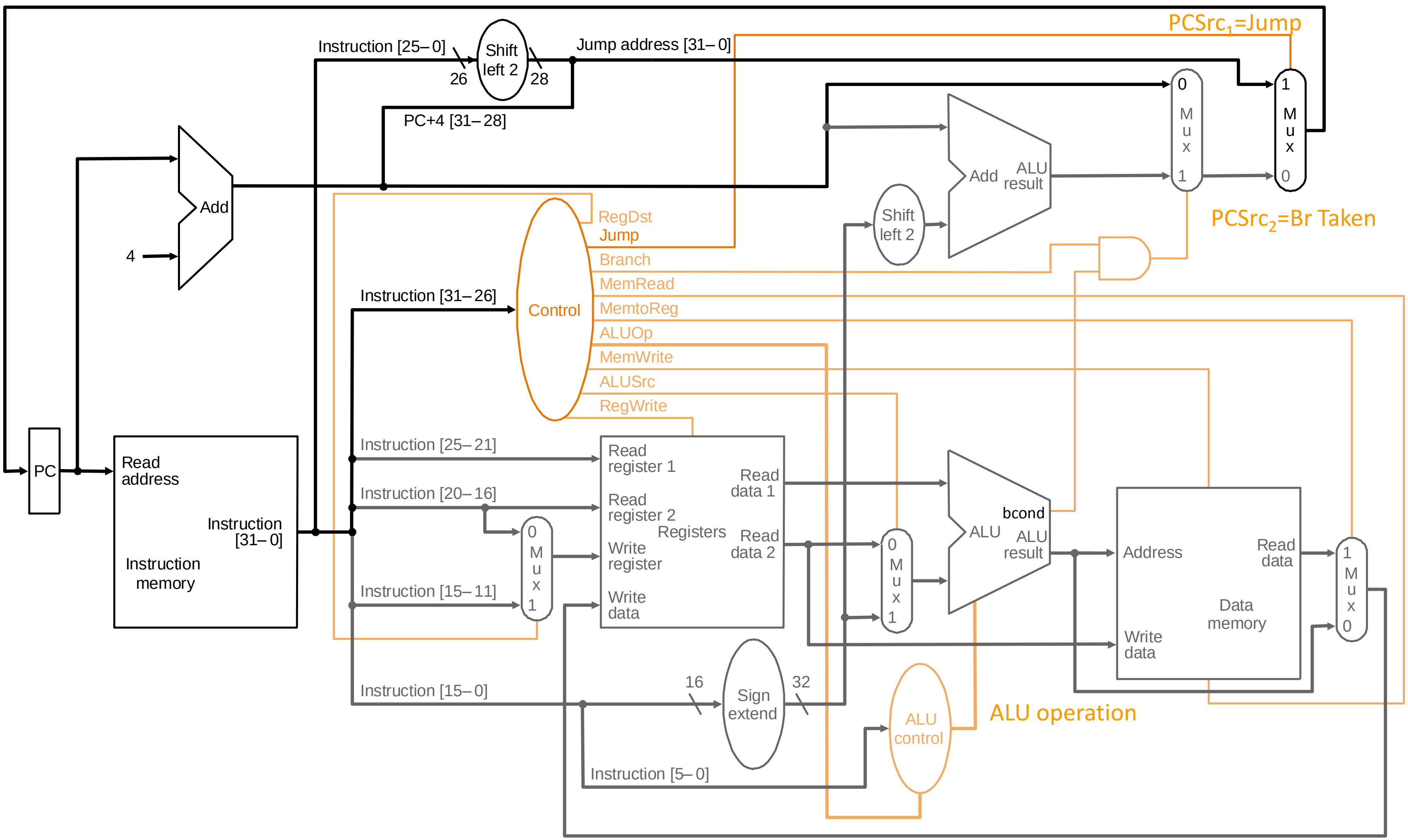
Instruction Processing

- 5 generic steps (P&H book)
  - Instruction fetch (IF)
  - Instruction decode and register operand fetch (ID/RF)
  - Execute/Evaluate memory address (EX/AG)
  - Memory operand fetch (MEM)
  - Store/writeback result (WB)



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

What Is To Come: The Full MIPS Datapath



**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier.

JAL, JR, JALR omitted

Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- Assembly (e.g., register-register signed addition)

  ADD rd_{reg} rs_{reg} rt_{reg}

- Machine encoding

0	rs	rt	rd	0	ADD
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

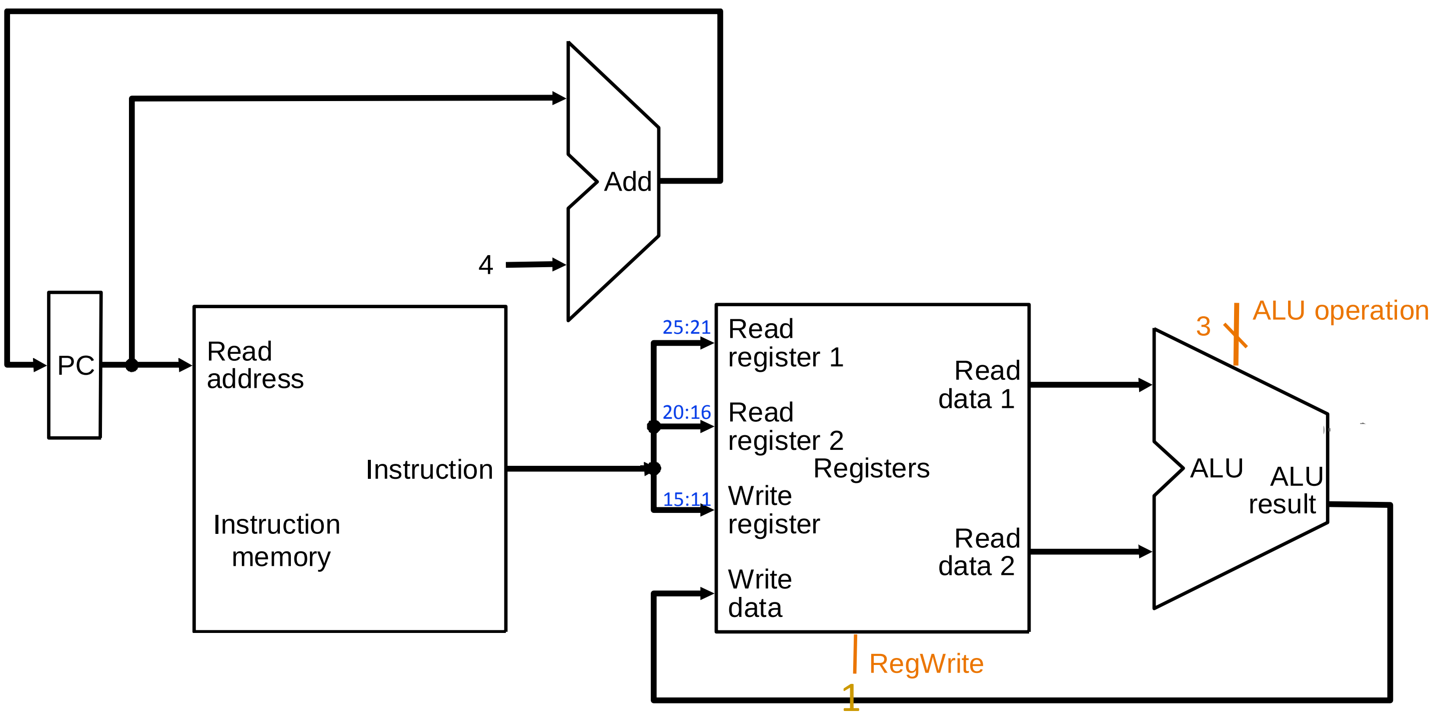
- Semantics

  if MEM[PC] == ADD rd rs rt

  GPR[rd] $\leftarrow$ GPR[rs] + GPR[rt]

  PC $\leftarrow$ PC + 4

ALU Datapath



if MEM[PC] == ADD rd rs rt

GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

Combinational state update logic

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

I-Type ALU Instructions

- Assembly (e.g., register-immediate signed additions)

  ADDI rt_{reg} rs_{reg} $immediate_{16}$

- Machine encoding

ADDI	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

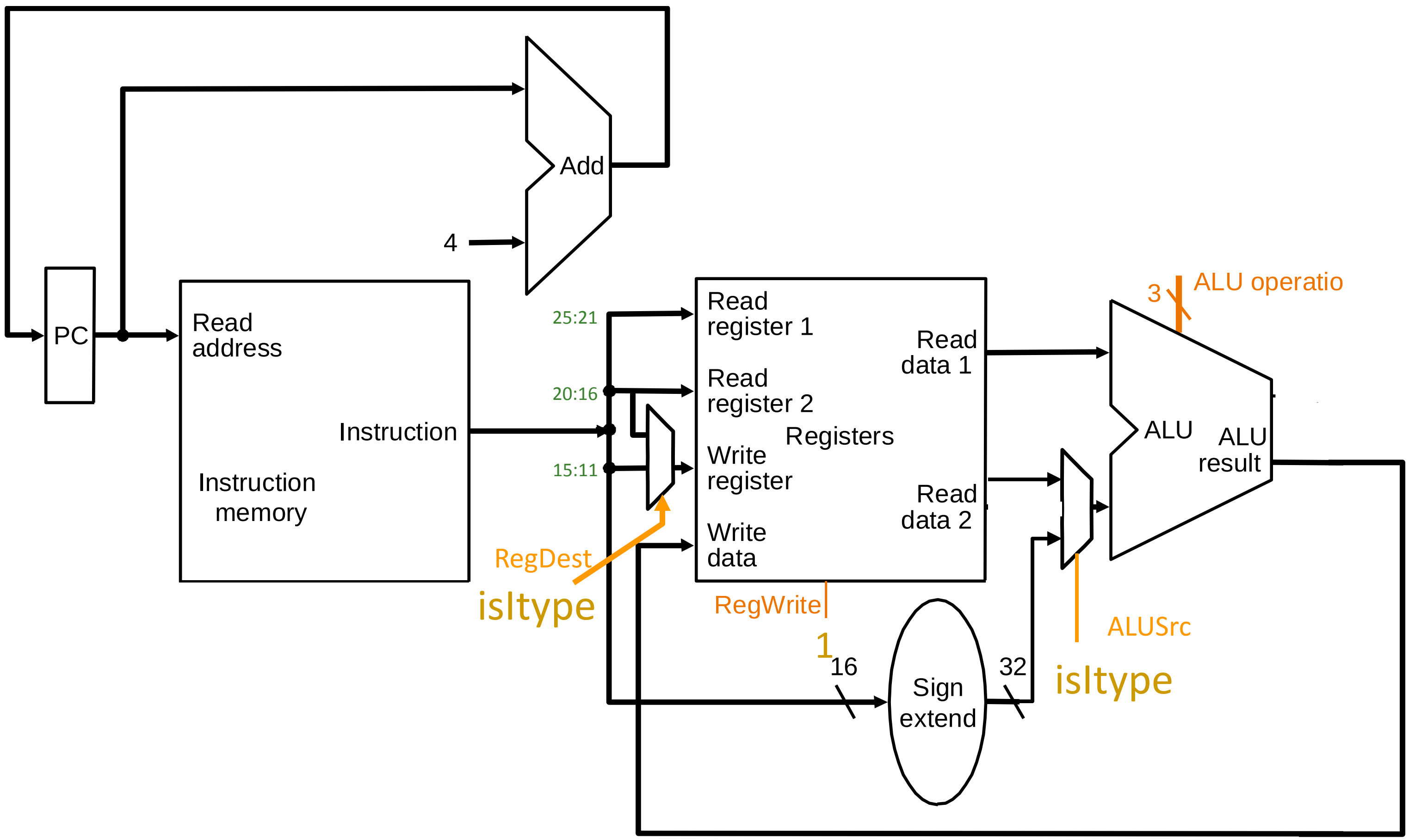
- Semantics

  if MEM[PC] == ADDI rt rs immediate

  GPR[rt] ← GPR[rs] + sign-extend (immediate)

  PC ← PC + 4

Datapath for R and I-Type ALU Insts.



IF	ID	EX	MEM	WB

if MEM[PC] == ADDI rt rs immediate

GPR[rt] ← GPR[rs] + sign-extend (immediate)

PC ← PC + 4

Combinational state update logic

Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

- Assembly (e.g., load 4-byte word)

  LW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding

LW	base	rt	offset
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics

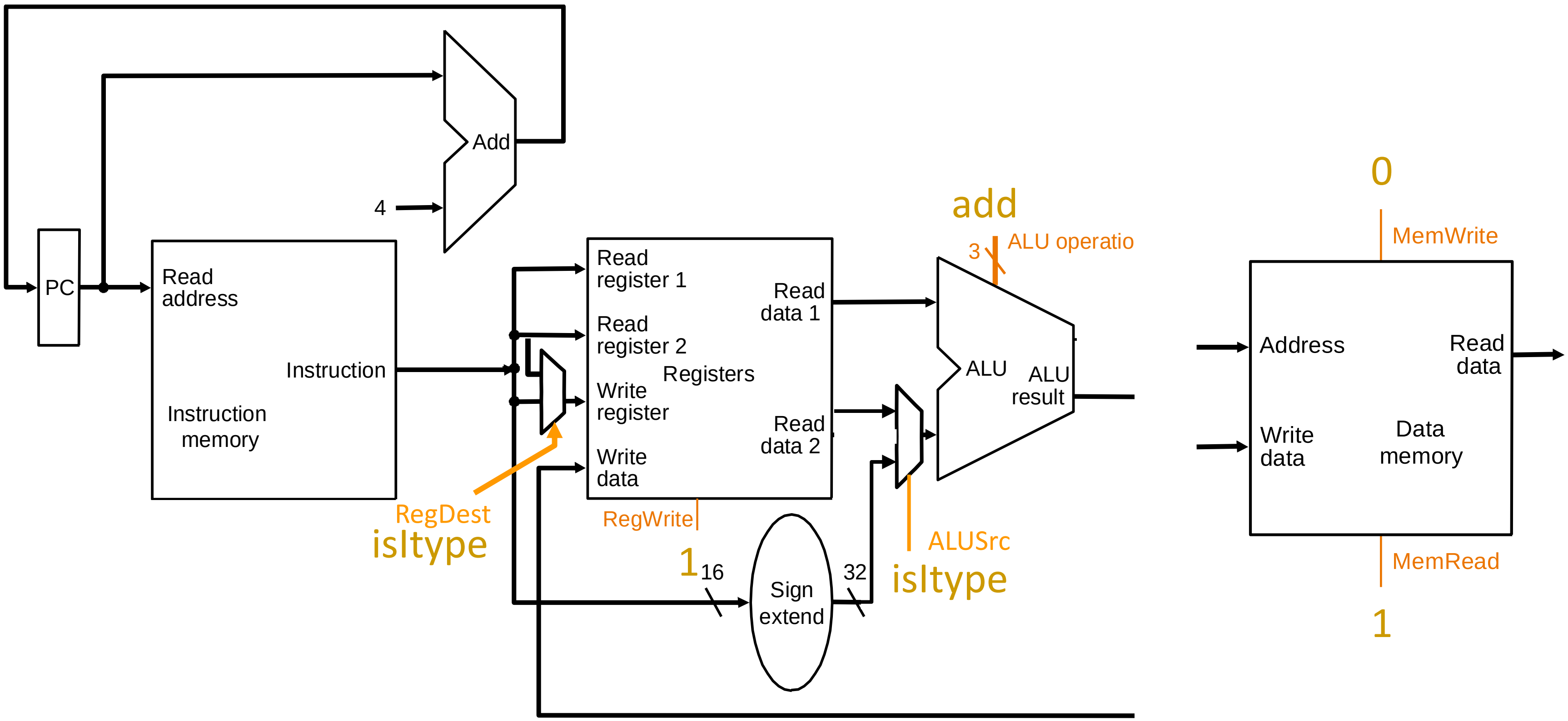
  if MEM[PC]==LW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  GPR[rt] ← MEM[translate(EA)]

  PC ← PC + 4

LW Datapath



if MEM[PC]==LW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

GPR[rt] ← MEM[translate(EA)]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

Store Instructions

- Assembly (e.g., store 4-byte word)

  SW rt_{reg} $offset_{16}$ ($base_{reg}$)

- Machine encoding

SW	base	rt	offset
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics

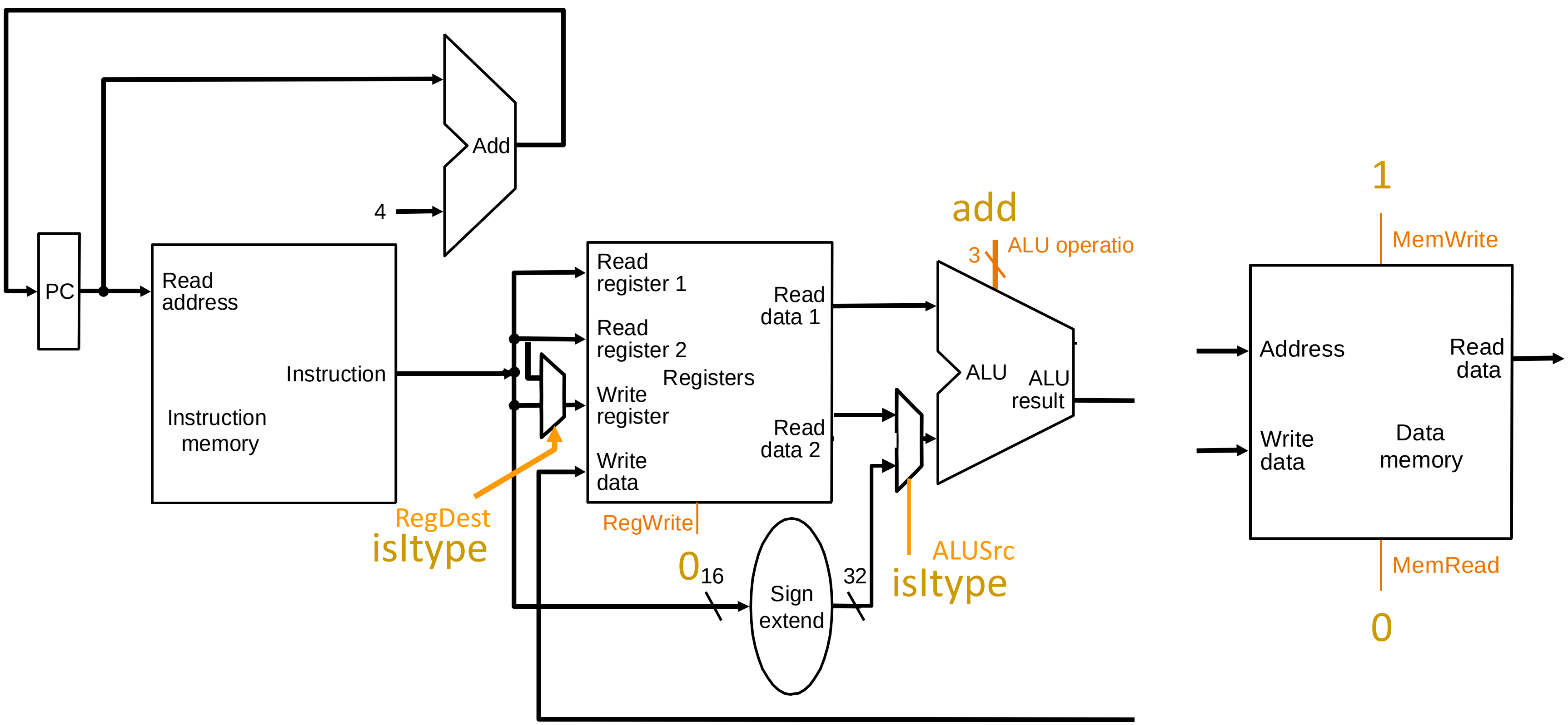
  if MEM[PC]==SW rt $offset_{16}$ (base)

  EA = sign-extend(offset) + GPR[base]

  MEM[translate(EA)] $\leftarrow$ GPR[rt]

  PC $\leftarrow$ PC + 4

SW Datapath



if MEM[PC]==SW rt offset$_{16}$ (base)

EA = sign-extend(offset) + GPR[base]

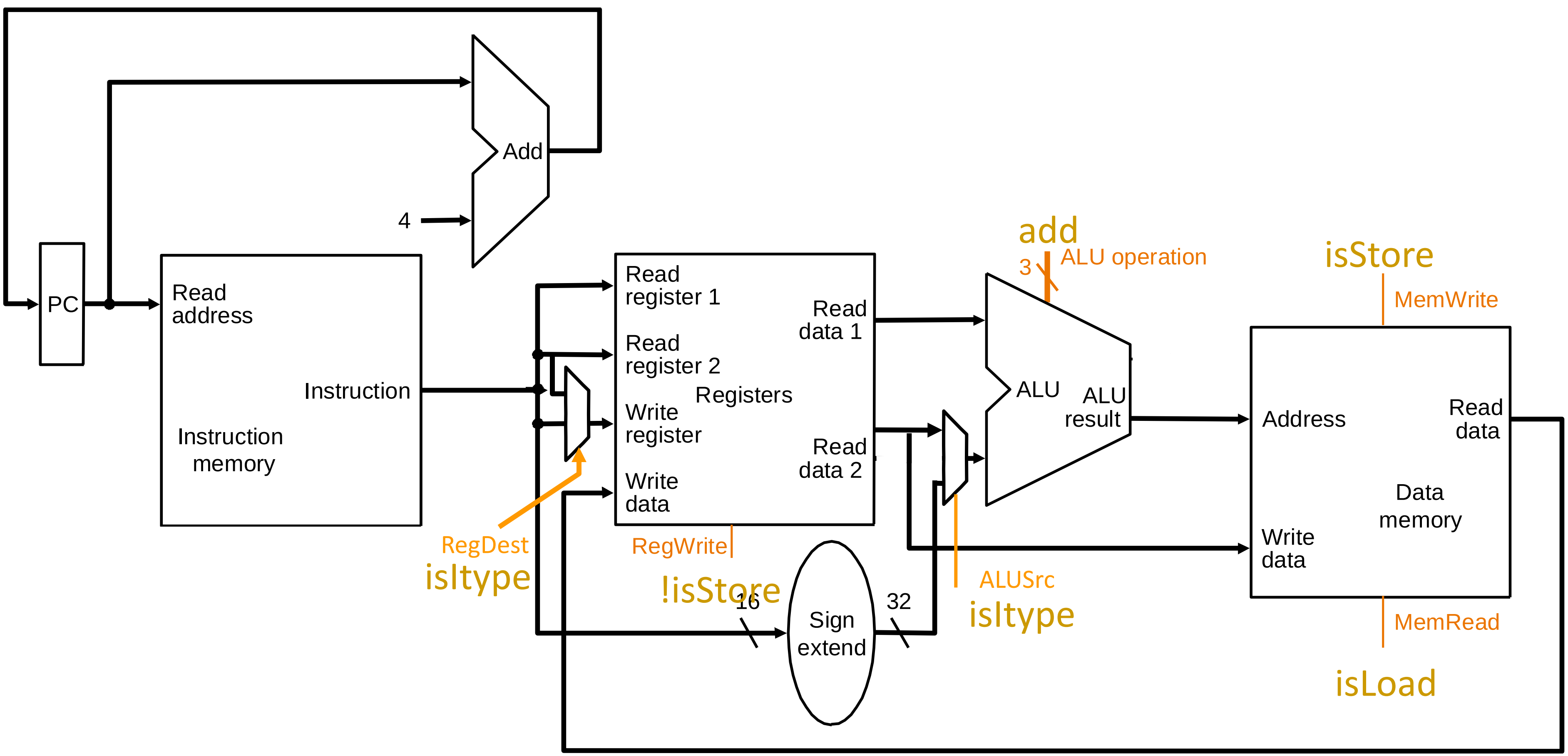
MEM[translate(EA)] ← GPR[rt]

PC ← PC + 4

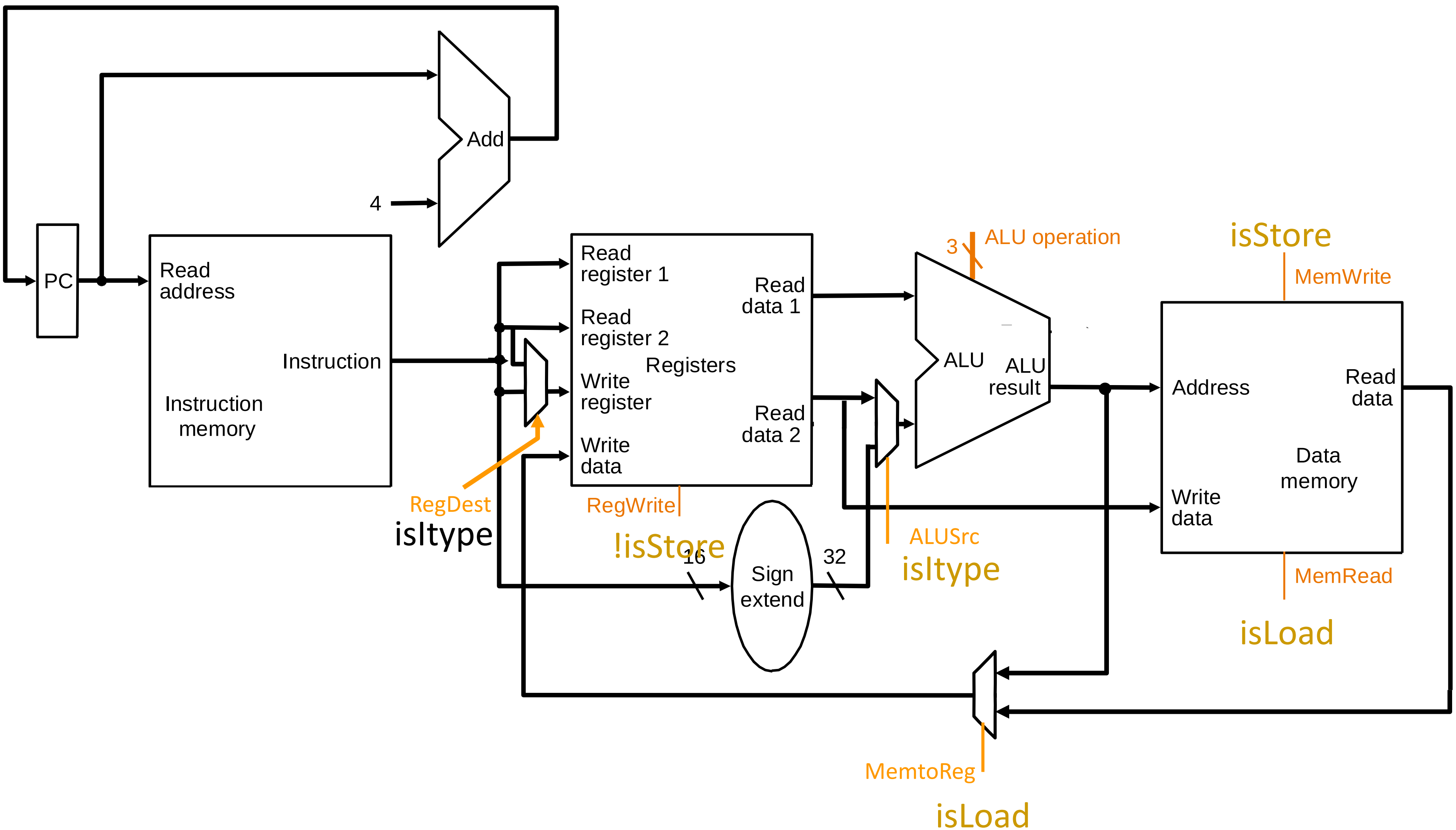
IF	ID	EX	MEM	WB

Combinational state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Insts.



Single-Cycle Datapath for *Control Flow Instructions*

Unconditional Jump Instructions

- Assembly

  J $immediate_{26}$

- Machine encoding

J	immediate
6-bit	26-bit

J-type

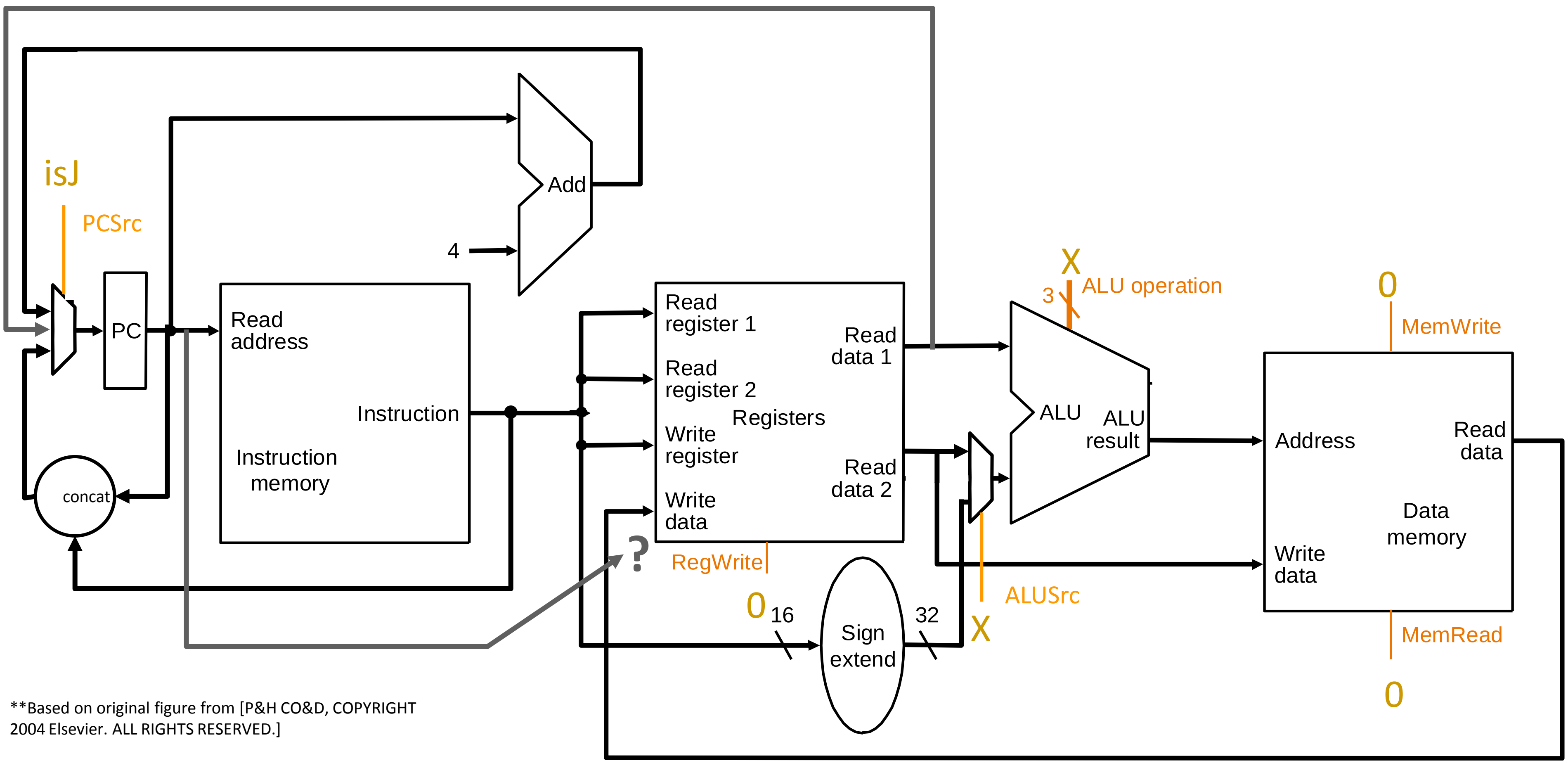
- Semantics

if MEM[PC]==J $immediate_{26}$

  target = { PC[31:28], $immediate_{26}$, 2' b00 }

  PC $\leftarrow$ target

Unconditional Jump Datapath



if MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2’ b00 }

What about JR, JAL, JALR?

Aside: MIPS Cheat Sheet

- http://www.ece.cmu.edu/~ece447/s15/lib/exe/fetch.php?media=mips_reference_data.pdf
- On the 447 website

Conditional Branch Instructions

- Assembly (e.g., branch if equal)

  BEQ rs_{reg} rt_{reg} $immediate_{16}$

- Machine encoding

BEQ	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics (assuming no branch delay slot)

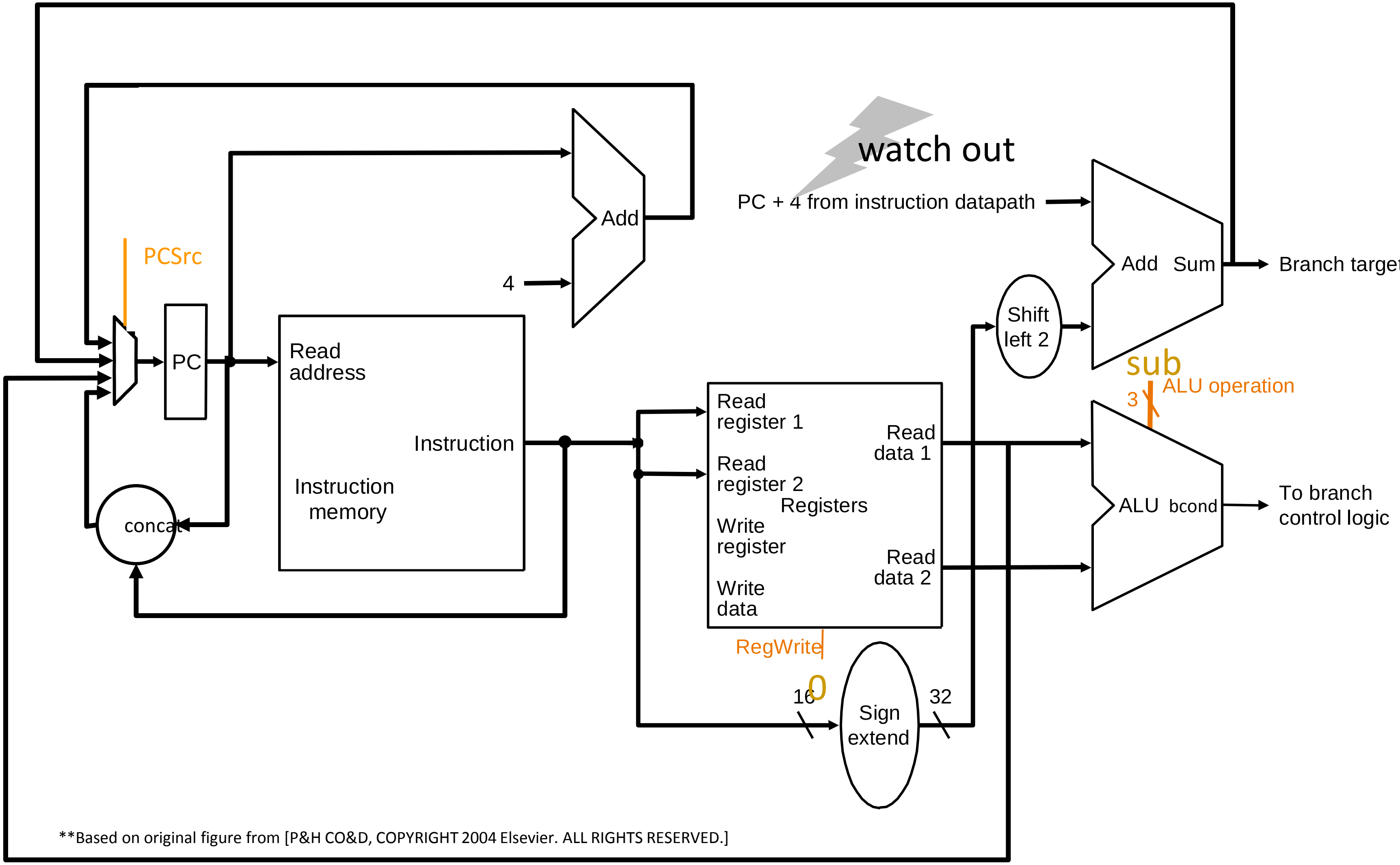
  if MEM[PC]==BEQ rs rt $immediate_{16}$

  target = PC + 4 + sign-extend(immediate) x 4

  if GPR[rs]==GPR[rt] then PC $\leftarrow$ target

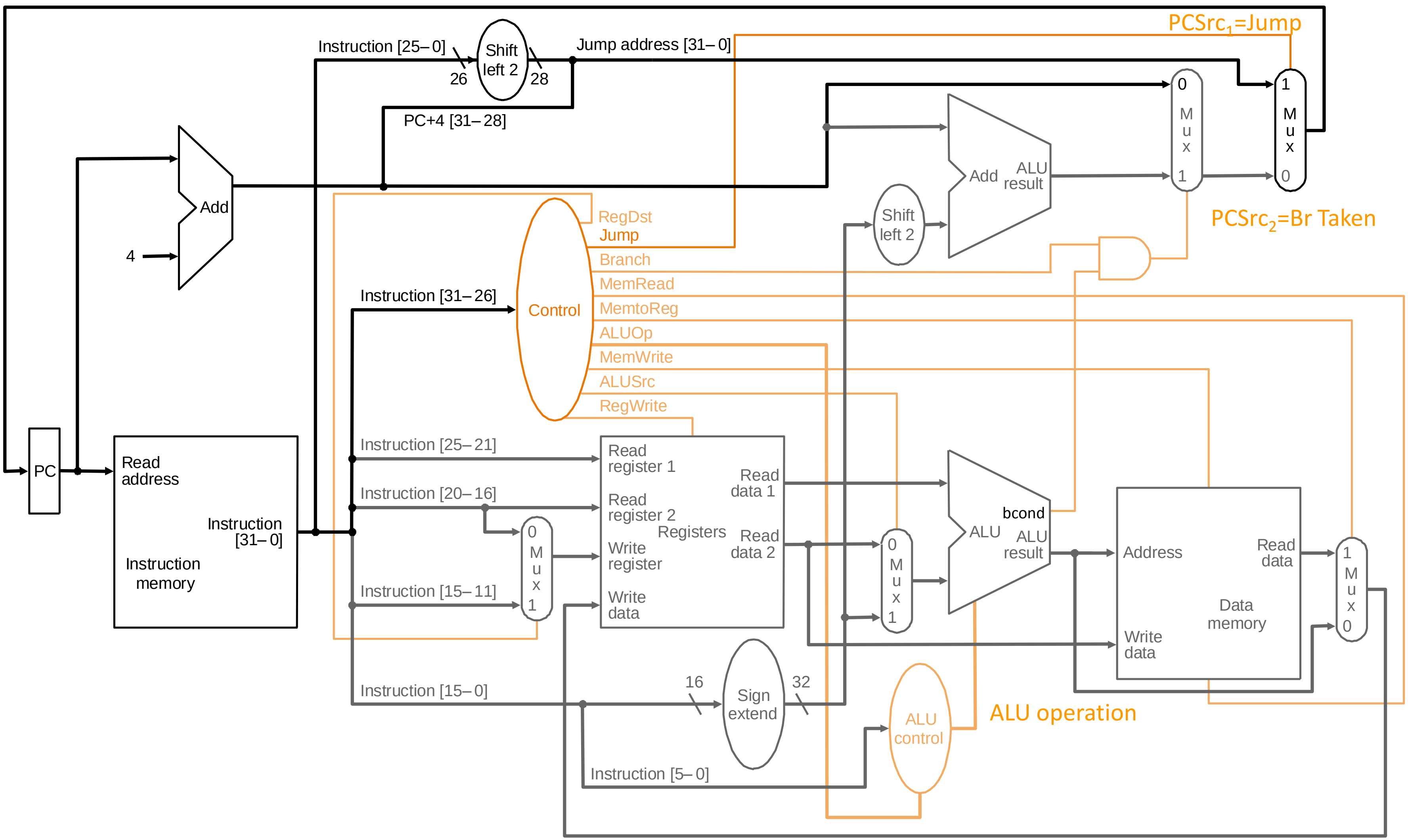
  else PC $\leftarrow$ PC + 4

Conditional Branch Datapath (for you to finish)



How to uphold the delayed branch semantics?

Putting It All Together



JAL, JR, JALR omitted

Single-Cycle Control Logic

Single-Cycle Hardwired Control

- As combinational function of Inst=MEM[PC]

R-type

31–26	25–21	20–16	15–11	10–6	5–0
0	rs	rt	rd	shamt	funct
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

I-type

31–26	25–21	20–16	15–0
opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

J-type

31–26	25–0
opcode	immediate
6-bit	26-bit

- Consider
  - All R-type and I-type ALU instructions
  - LW and SW
  - BEQ, BNE, BLEZ, BGTZ
  - J, JR, JAL, JALR

Single-Bit Control Signals

	When De-asserted	When asserted	Equation
RegDest	GPR write select according to rt, i.e., inst[20:16]	GPR write select according to rd, i.e., inst[15:11]	opcode==0
ALUSrc	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) **&&** (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	steer memory load to GPR wr. port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

JAL and JALR require additional RegDest and MemtoReg options

Single-Bit Control Signals

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	opcode==LW
MemWrite	Memory write disabled	Memory write enabled	opcode==SW
$PCSrc_1$	According to $PCSrc_2$	next PC is based on 26-bit immediate jump target	(opcode==J) || (opcode==JAL)
$PCSrc_2$	next PC = PC + 4	next PC is based on 16-bit immediate branch target	(opcode==Bxx) && "bcond is satisfied"

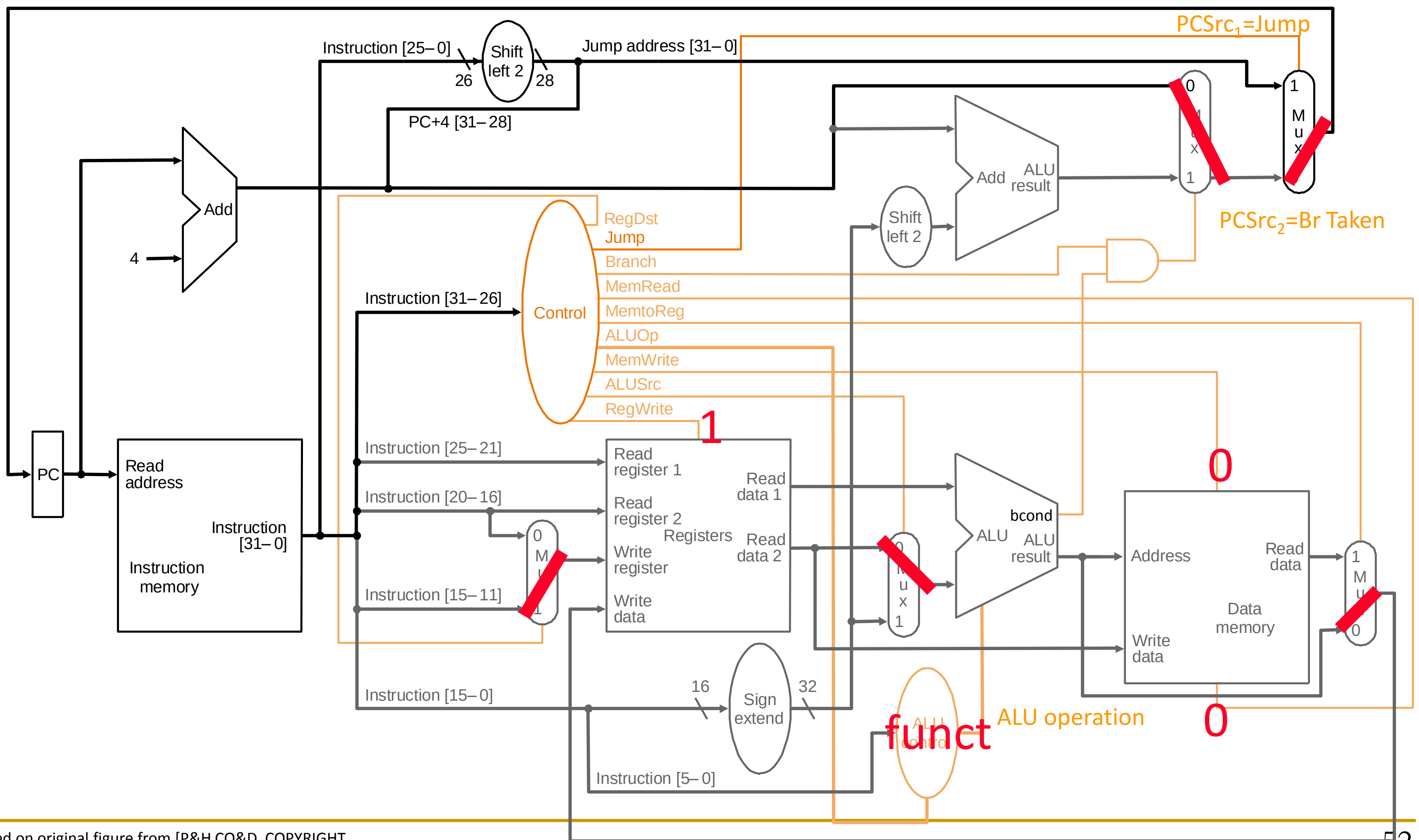
JR and JALR require additional PCSrc options

ALU Control

- case opcode
  '0' ⇒ select operation according to funct
  'ALUi' ⇒ selection operation according to opcode
  'LW' ⇒ select addition
  'SW' ⇒ select addition
  'Bxx' ⇒ select bcond generation function
  __ ⇒ don't care

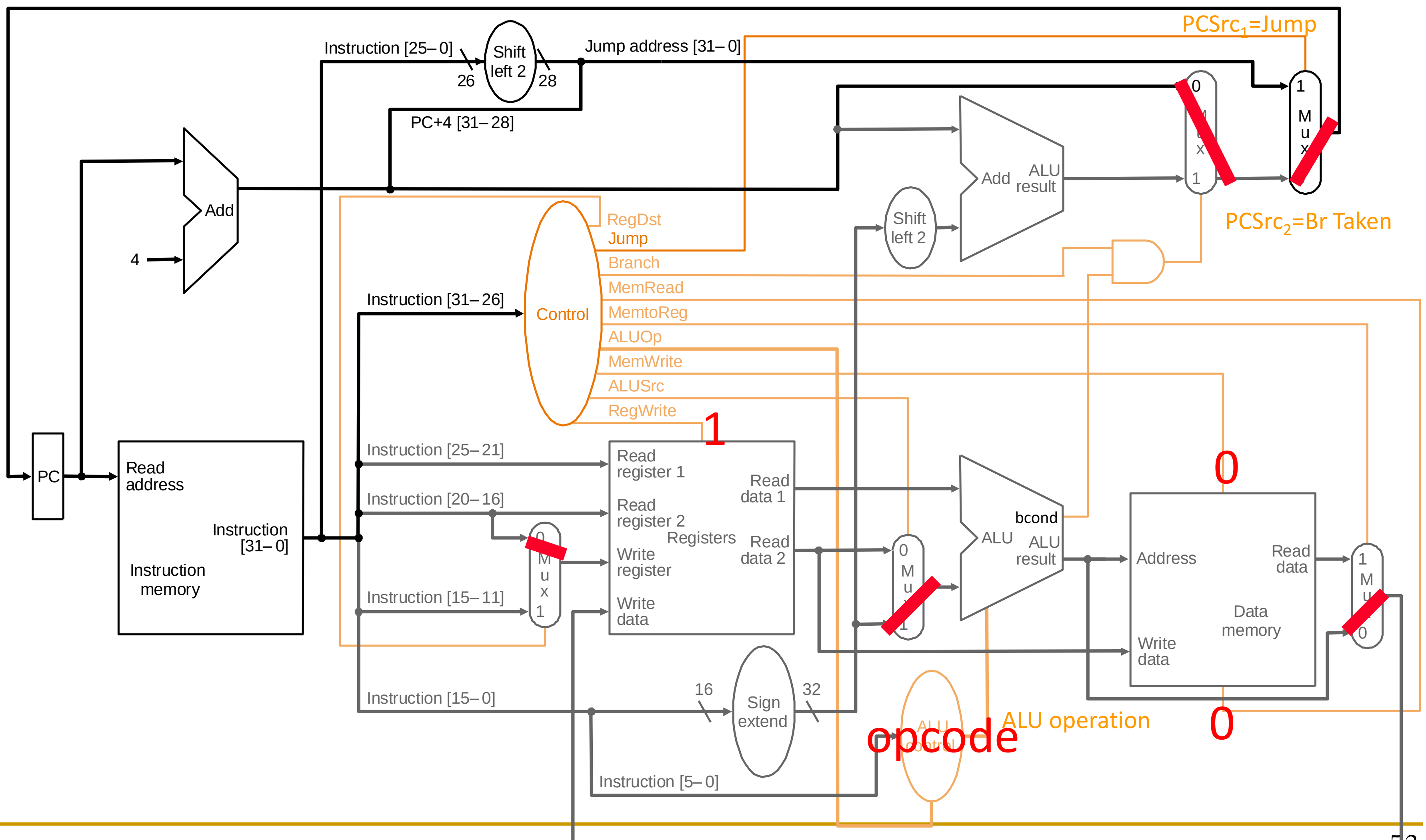
- Example ALU operations
  - ADD, SUB, AND, OR, XOR, NOR, etc.
  - bcond on equal, not equal, LE zero, GT zero, etc.

R-Type ALU



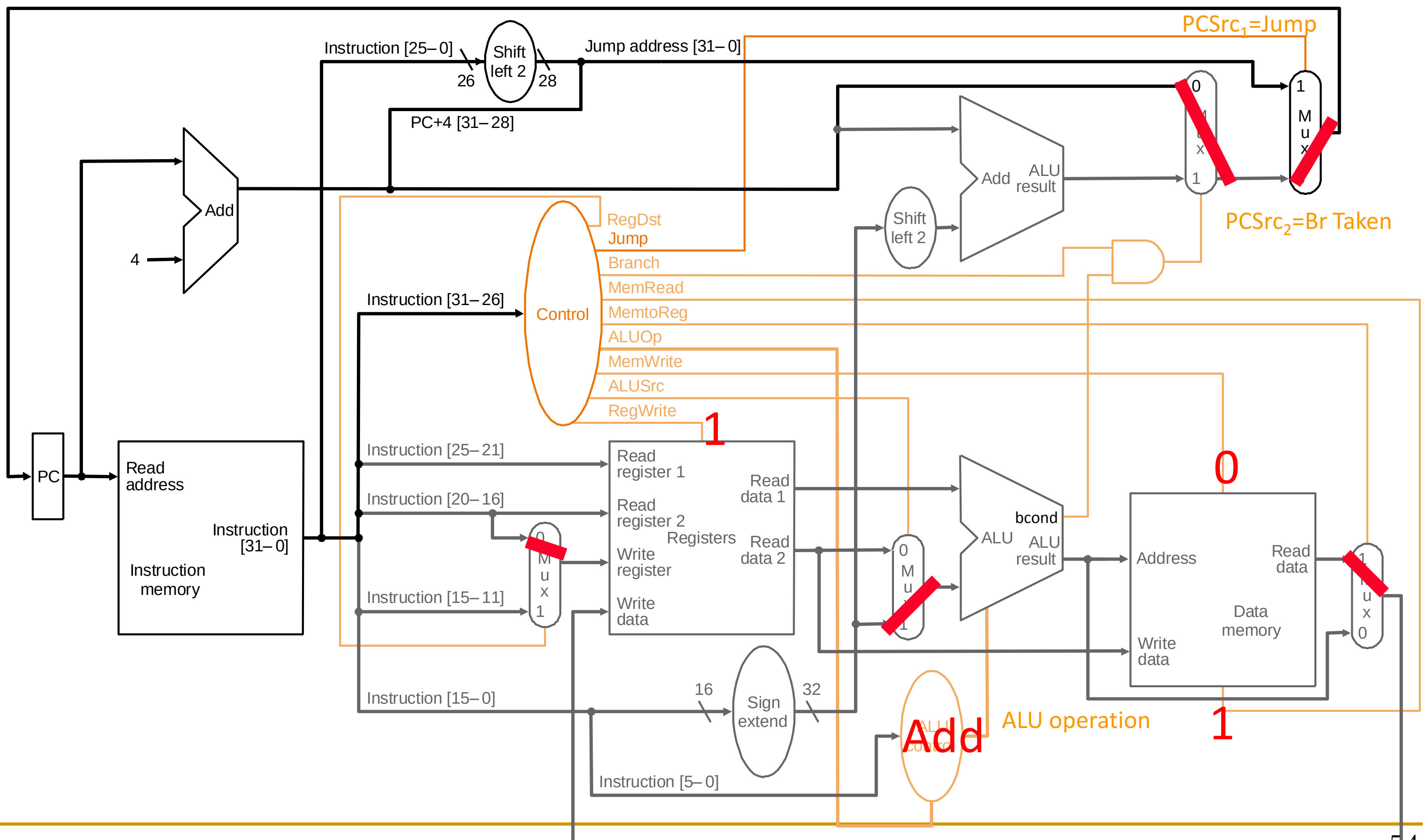
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



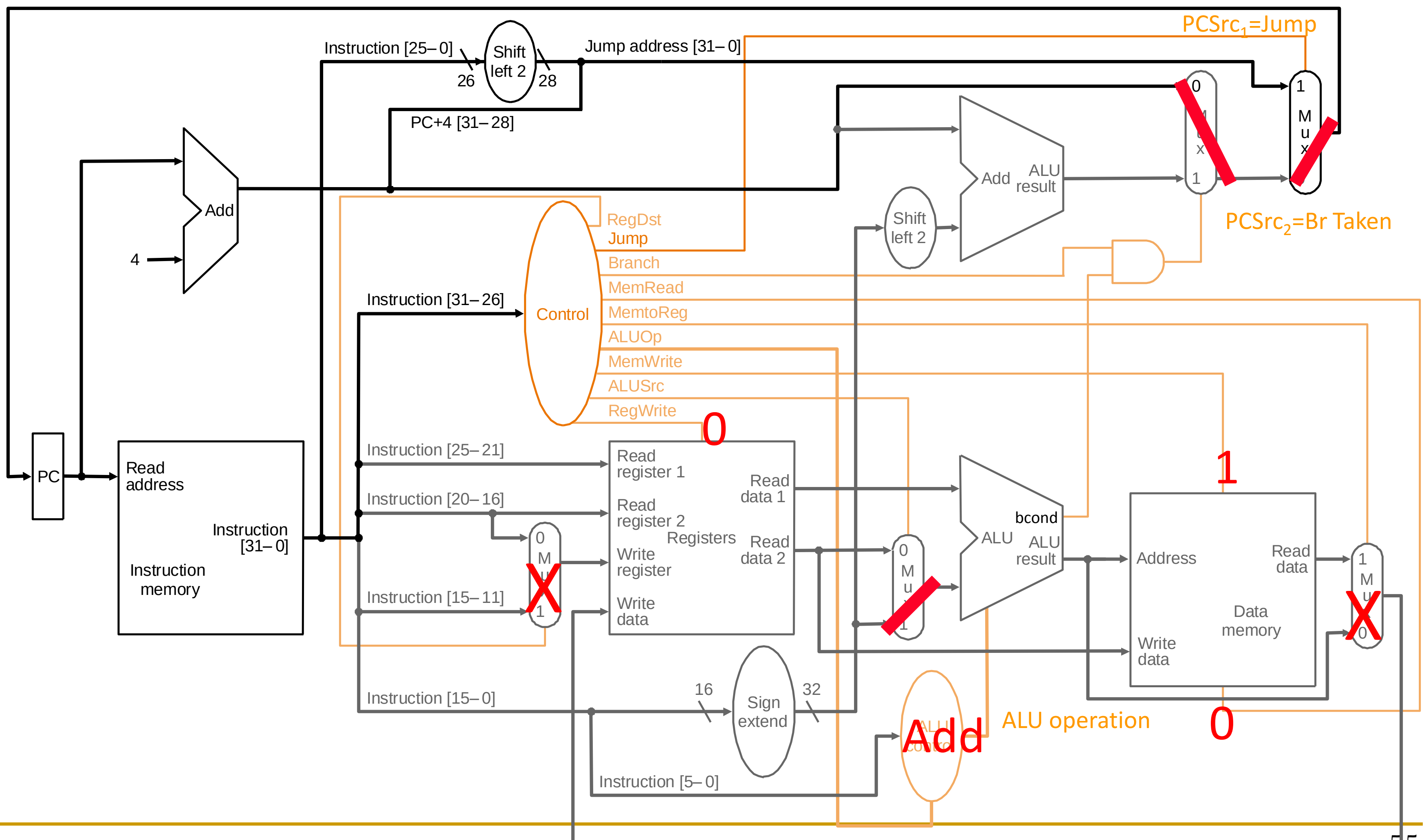
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

LW



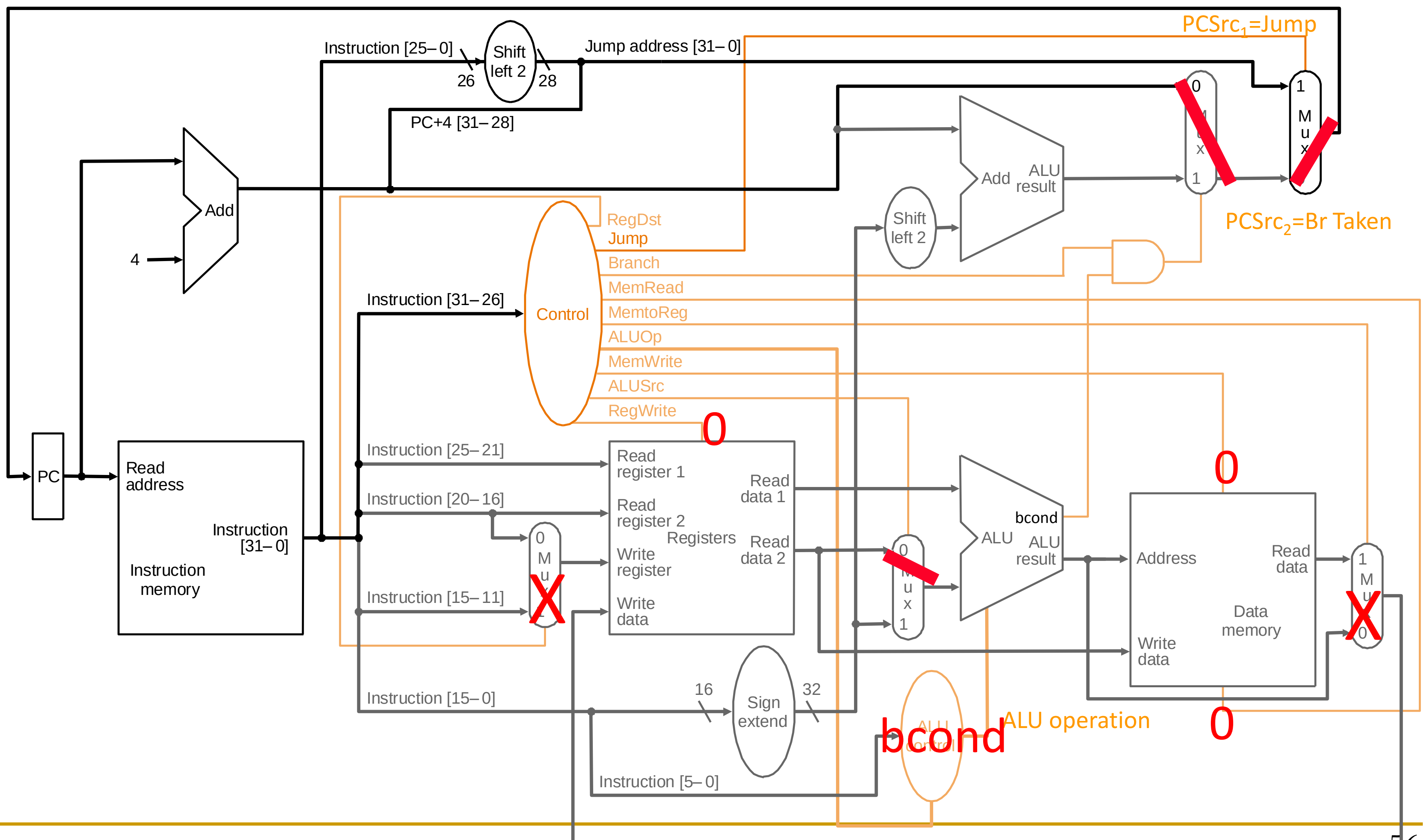
PCSrc1=Jump
Instruction [25– 0]
26
Shift left 2
28
Jump address [31– 0]
PC+4 [31– 28]
Add
4
0
M
u
x
1
1
M
u
x
Add
ALU result
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31– 26]
1
0
PC
Read address
Instruction [31– 0]
Instruction memory
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 11]
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
bcond
ALU
ALU result
Address
Read data
Data memory
Write data
16
Sign extend
32
Instruction [15– 0]
Instruction [5– 0]
Add
ALU operation
1

SW


PCSrc1=Jump
PCSrc2=Br Taken
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
ALU result
0 M u x 1
1 M u x
Add
4
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
0
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
0 M u x 1
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
ALU
bcond
ALU result
1
Address
Read data
Data memory
Write data
1 M u x 0
Instruction [15–0]
16
Sign extend
32
Add
ALU operation
0
Instruction [5–0]
X
X

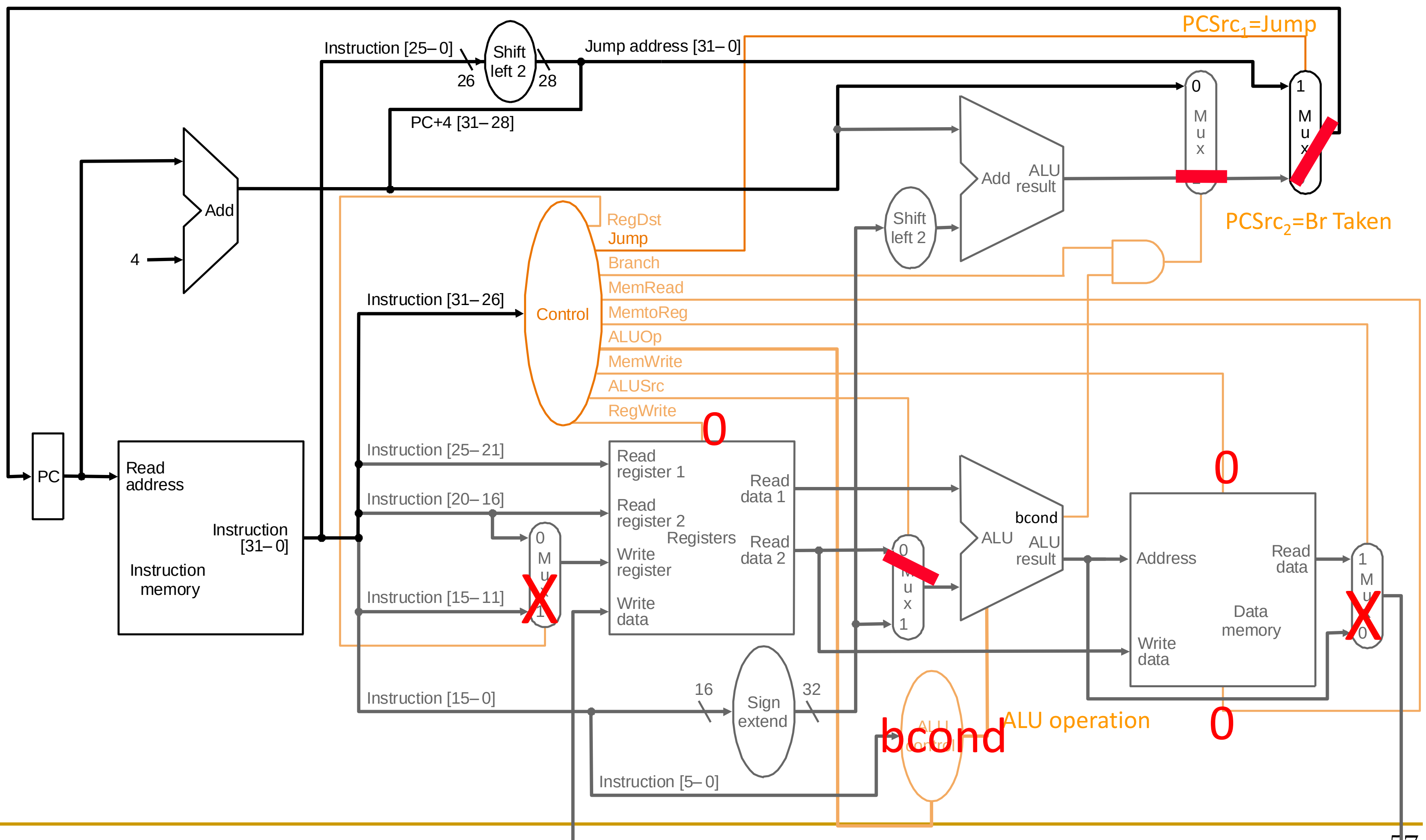
Branch (Not Taken)

Some control signals are dependent on the processing of data

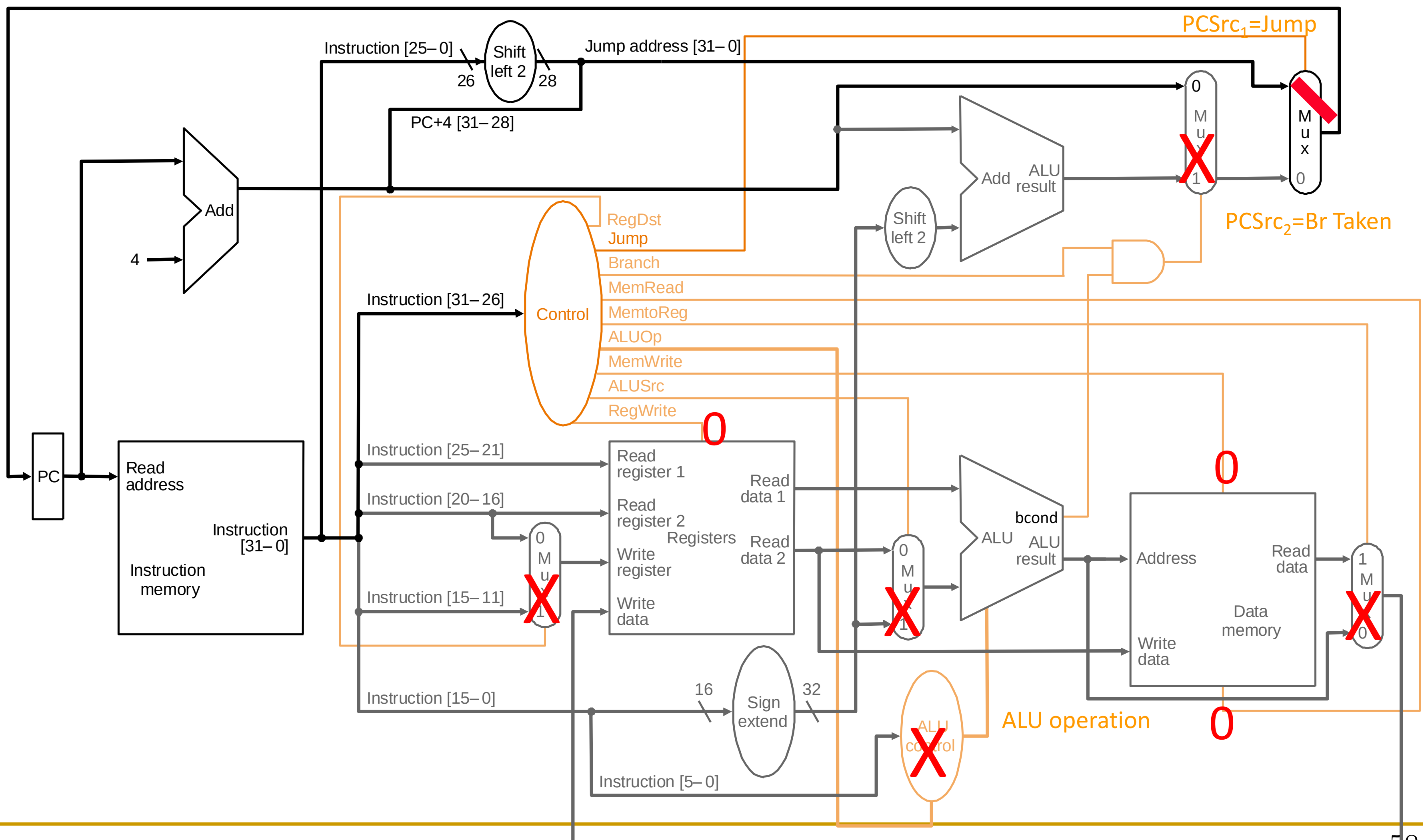


Branch (Taken)

Some control signals are dependent on the processing of data



Jump



PCSrc$_1$=Jump
Instruction [25–0]
26
Shift left 2
28
Jump address [31–0]
PC+4 [31–28]
0
M u x
1
M u x
0
Add
ALU result
Add
4
Shift left 2
PCSrc$_2$=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
0
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
PC
Read address
Instruction [31–0]
Instruction memory
0
M u x
1
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
0
M u x
1
bcond
ALU
ALU result
0
Address
Read data
Data memory
Write data
1
M u x
0
Instruction [15–0]
16
Sign extend
32
ALU control
Instruction [5–0]
ALU operation
0

What is in That Control Box?

- Combinational Logic → Hardwired Control
  - Idea: Control signals generated combinationally based on instruction
  - Necessary in a single-cycle microarchitecture…

- Sequential Logic → Sequential/Microprogrammed Control
  - Idea: A memory structure contains the control signals associated with an instruction
  - Control Store

Evaluating the Single-Cycle Microarchitecture

A Single-Cycle Microarchitecture

- Is *this* a good idea/design?

- When is this a good design?

- When is this a bad design?

- How can we design a better microarchitecture?

A Single-Cycle Microarchitecture: Analysis

- Every instruction takes 1 cycle to execute
  - CPI (Cycles per instruction) is strictly 1

- How long each instruction takes is determined by how long the slowest instruction takes to execute
  - Even though many instructions do not need that long to execute

- Clock cycle time of the microarchitecture is determined by how long it takes to complete the slowest instruction
  - Critical path of the design is determined by the processing time of the slowest instruction

What is the Slowest Instruction to Process?

- Let's go back to the basics

- All six phases of the instruction processing cycle take a *single machine clock cycle* to complete
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result

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)

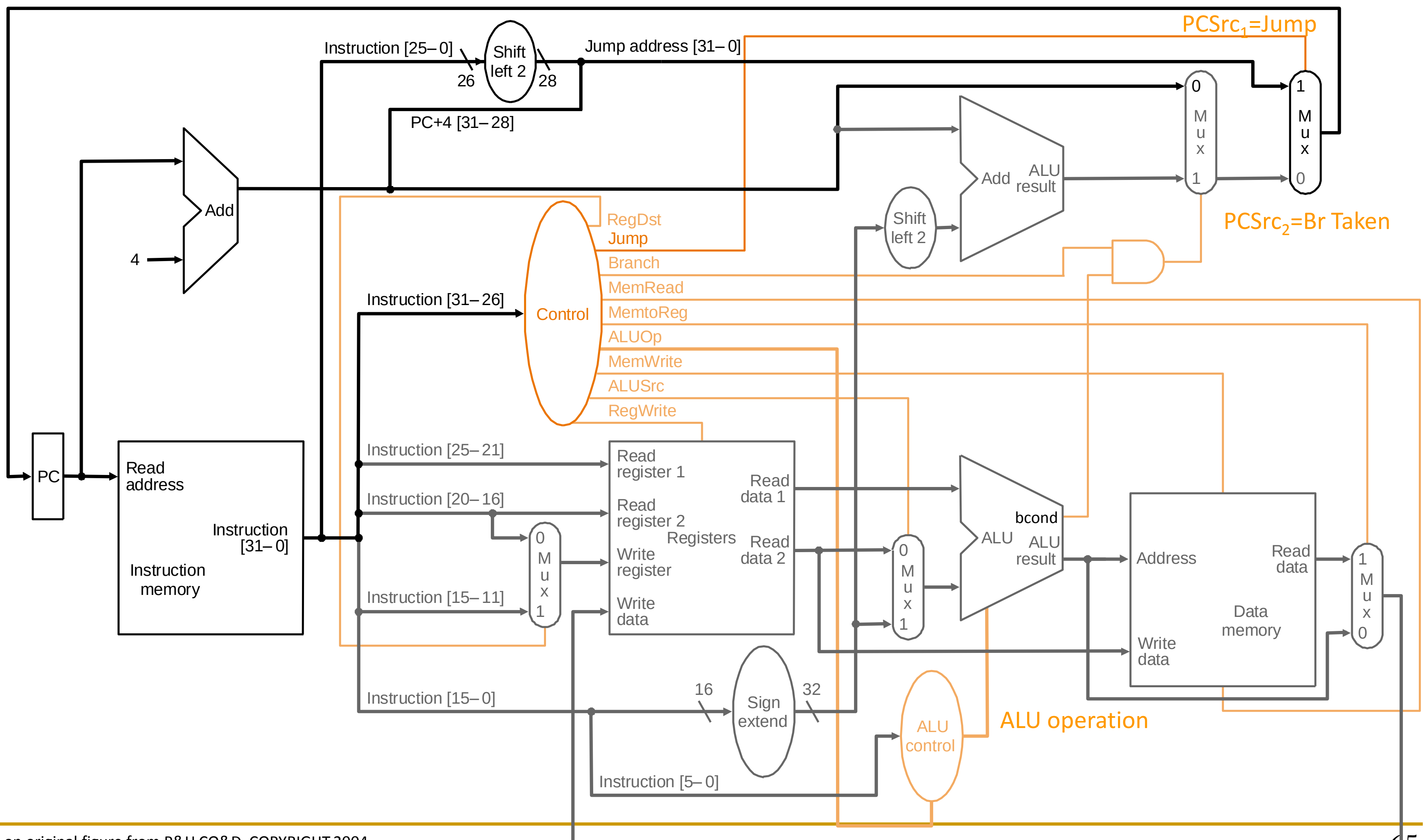
- Do each of the above phases take the same time (latency) for all instructions?

Single-Cycle Datapath Analysis

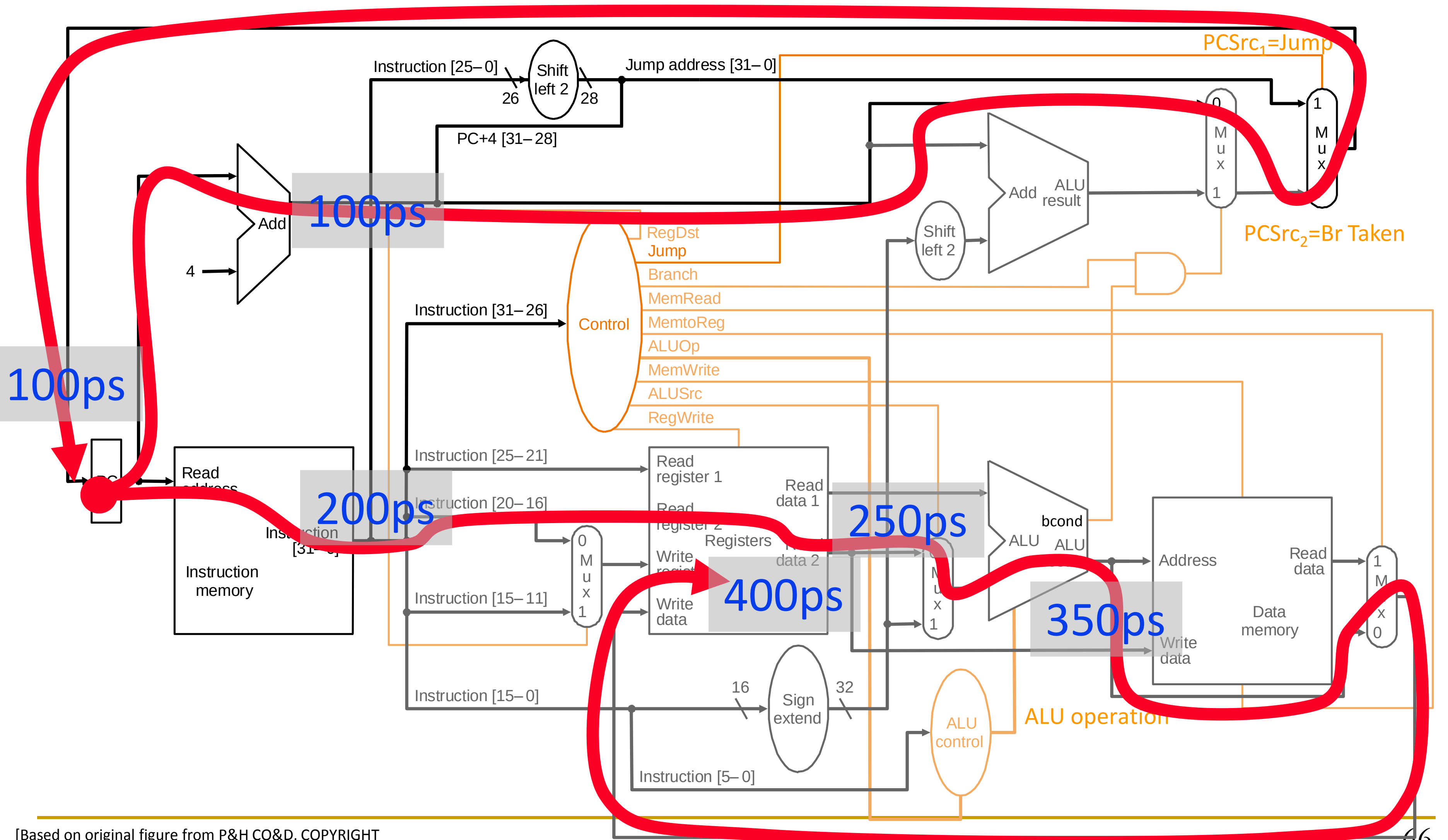
- Assume
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

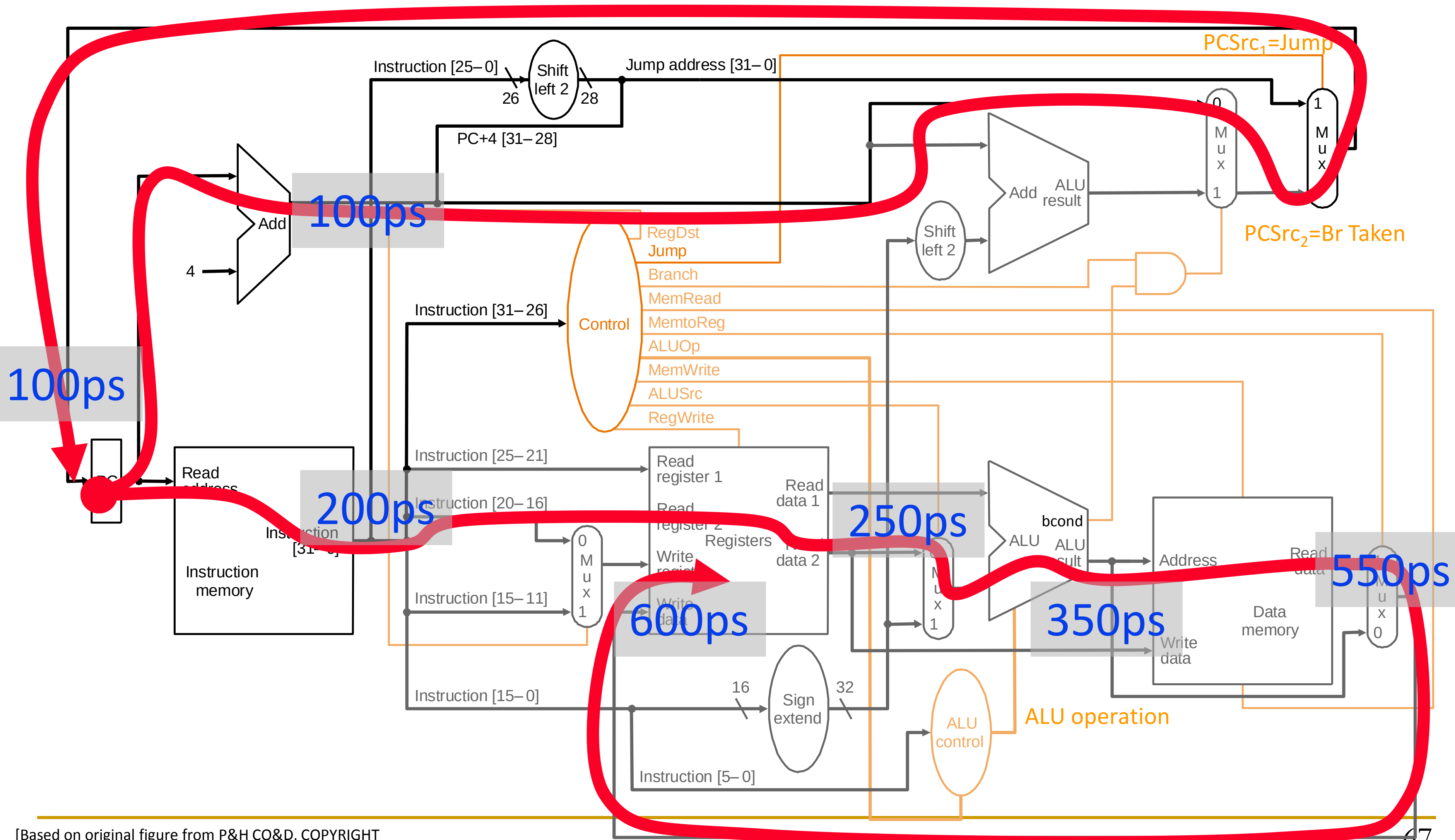
Let's Find the Critical Path



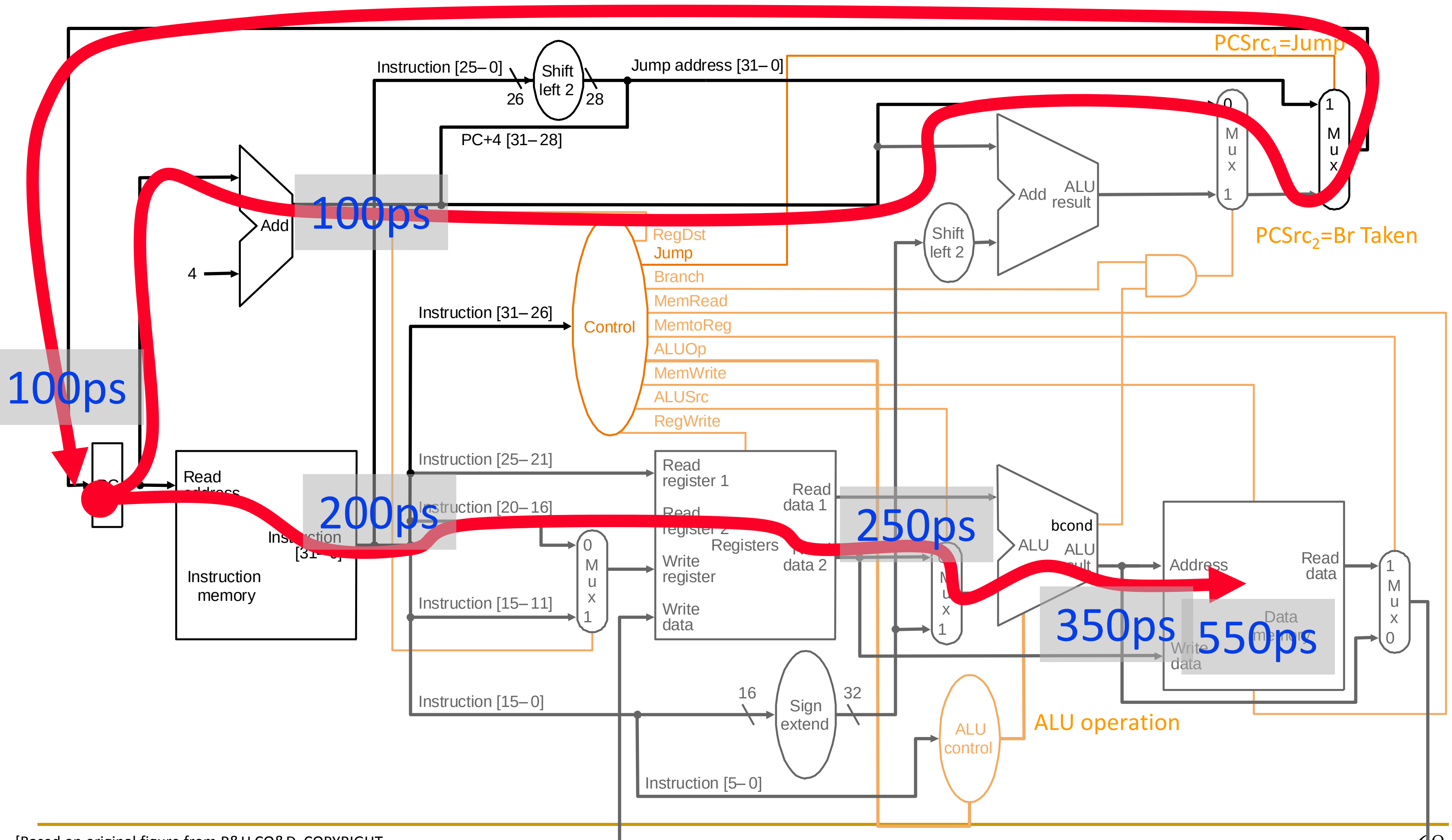
R-Type and I-Type ALU



LW

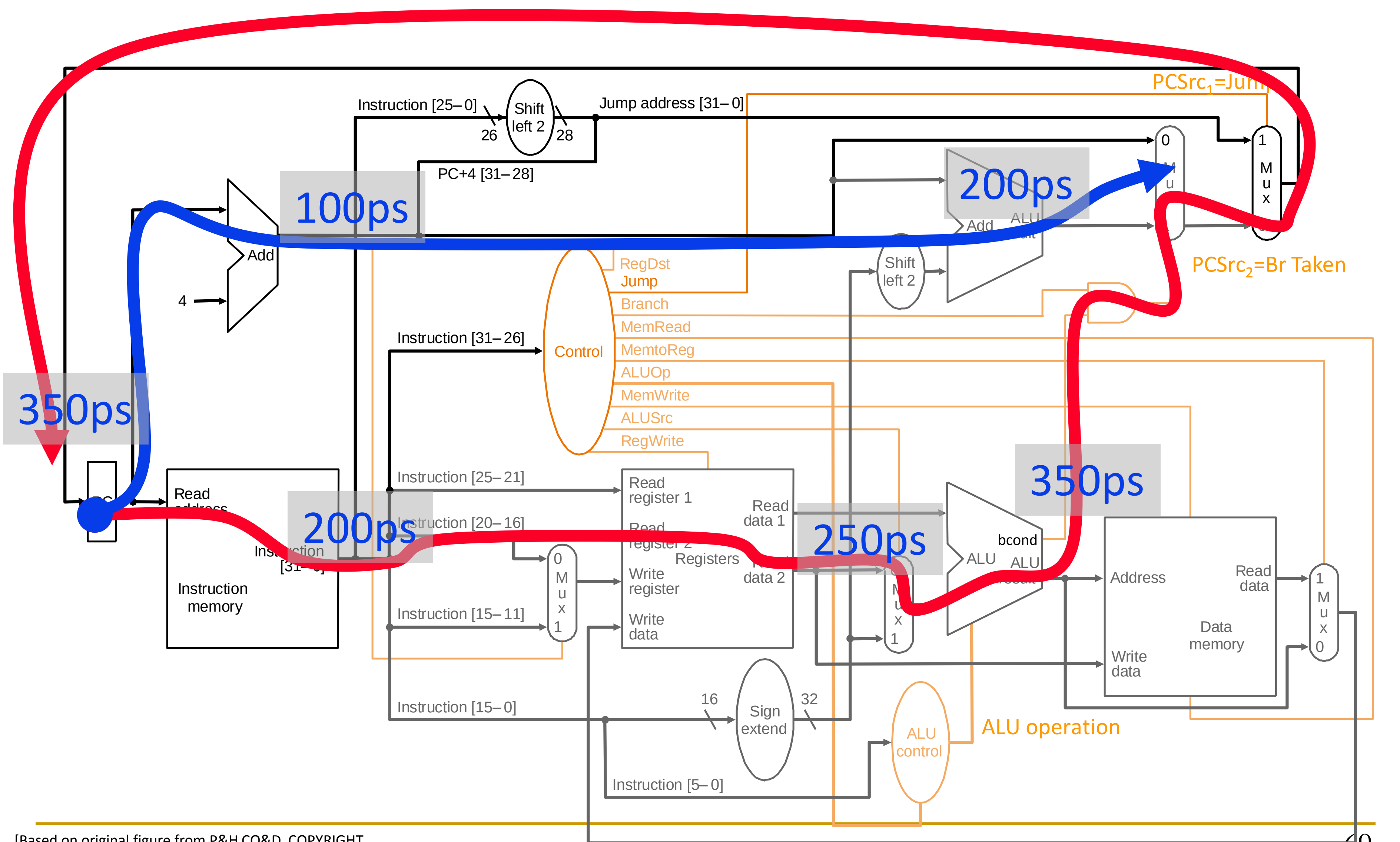

PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
100ps
100ps
M u x
0
1
ALU result
PCSrc2=Br Taken
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read data 1
Registers
Read data 2
Write
Instruction memory
Read
200ps
250ps
350ps
550ps
600ps
bcond
ALU
Address
Data memory
Write data
16
32
Sign extend
ALU control
ALU operation

SW

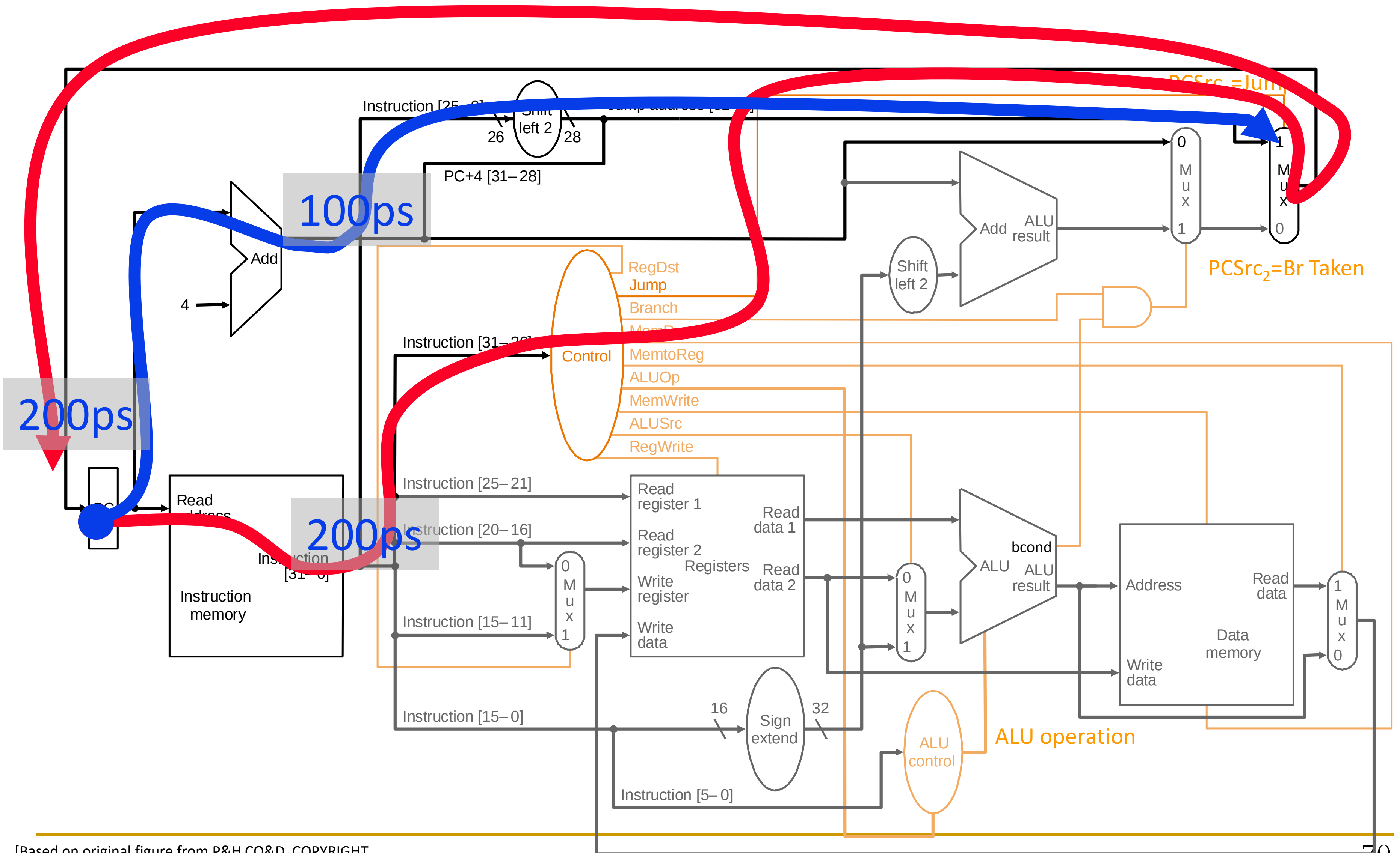


PCSrc$_1$=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
100ps
Add
ALU result
Shift left 2
PCSrc$_2$=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
100ps
Read address
Instruction [31–0]
Instruction memory
200ps
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
250ps
bcond
ALU
ALU result
350ps
Address
Read data
Write data
Data memory
550ps
Instruction [15–0]
16
Sign extend
32
ALU control
ALU operation
Instruction [5–0]

Branch Taken



Jump



PCSrc₁=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
100ps
Add
4
200ps
PC
Read address
Instruction [31–0]
Instruction memory
200ps
Instruction [31–26]
Control
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign extend
32
Shift left 2
Add
ALU result
ALU
bcond
ALU result
ALU control
ALU operation
Address
Read data
Write data
Data memory
PCSrc₂=Br Taken

What About Control Logic?

- How does that affect the critical path?

- Food for thought for you:
  - Can control logic be on the critical path?
  - A note on CDC 5600: control store access too long…

What is the Slowest Instruction to Process?

- Memory is not magic

- What if memory *sometimes* takes 100ms to access?

- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?

- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction

- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle

- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?

- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

- Critical path design
  - Find and decrease the maximum combinational logic delay
  - Break a path into multiple cycles if it takes too long

- Bread and butter (common case) design
  - Spend time and resources on where it matters most
    - i.e., improve what the machine is really designed to do
  - Common case vs. uncommon case

- Balanced design
  - Balance instruction/data flow through hardware components
  - Design to eliminate bottlenecks: balance the hardware for the work

Single-Cycle Design vs. Design Principles

- Critical path design
- Bread and butter (common case) design
- Balanced design

How does a single-cycle microarchitecture fare in light of these principles?

Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles

- Remember: “principled design” from our first lecture
  - Frank Lloyd Wright: “architecture […] based upon principle, and not upon precedent”

Aside: From Lecture 1

- "architecture [...] based upon principle, and not upon precedent"

Aside: System Design Principles

- We will continue to cover key principles in this course
- Here are some references where you can learn more

- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)
- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966. (Flynn's Bottleneck → Balanced design)
- Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS Conference, April 1967. (Amdahl's Law → Common-case design)
- Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
    - http://research.microsoft.com/pubs/68221/acrobat.pdf

Aside: One Important Principle

- Keep it simple

- "Everything should be made as simple as possible, but no simpler."
  - Albert Einstein

- And, do not forget: "An engineer is a person who can do for a dime what any fool can do for a dollar."

- For more, see:
  - Butler W. Lampson, "Hints for Computer System Design," ACM Operating Systems Review, 1983.
  - http://research.microsoft.com/pubs/68221/acrobat.pdf

Multi-Cycle Microarchitectures