

Digital Design & Computer Arch.

Lecture 11: Microarchitecture Fundamentals

Prof. Onur Mutlu

ETH Zürich

Spring 2021

1 April 2021

Readings

■ This week

- Introduction to microarchitecture and single-cycle microarchitecture
 - H&H, Chapter 7.1-7.3
 - P&P, Appendices A and C
- Multi-cycle microarchitecture
 - H&H, Chapter 7.4
 - P&P, Appendices A and C

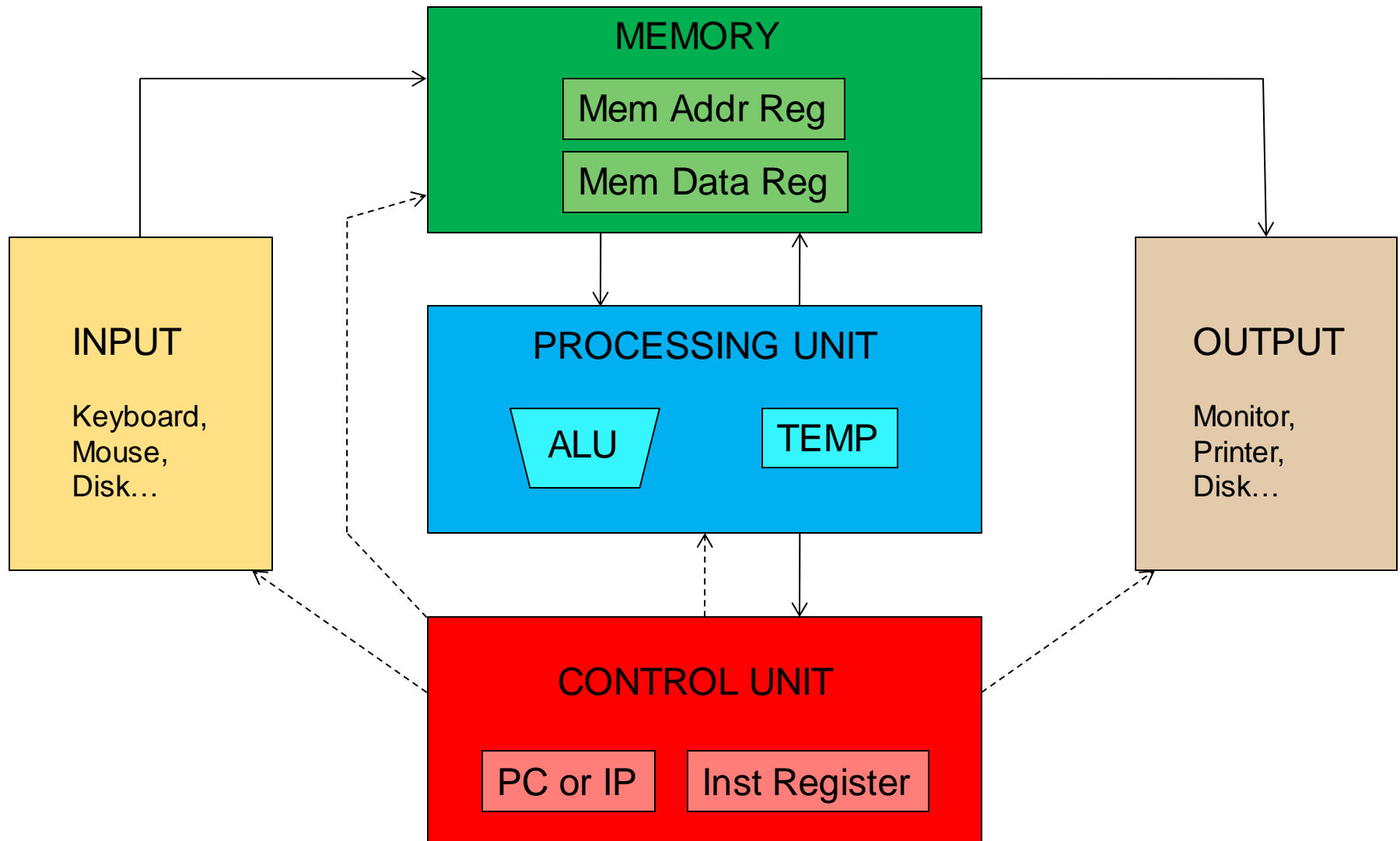
■ Next week

- Pipelining
 - H&H, Chapter 7.5
- Pipelining Issues
 - H&H, Chapter 7.7, 7.8.1-7.8.3

Agenda for Today & Next Few Lectures

- Instruction Set Architectures (ISA): LC-3 and MIPS
- Assembly programming: LC-3 and MIPS
- Microarchitecture (principles & single-cycle uarch)
- Multi-cycle microarchitecture
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution

Recall: The von Neumann Model



Recall: LC-3: A von Neumann Machine

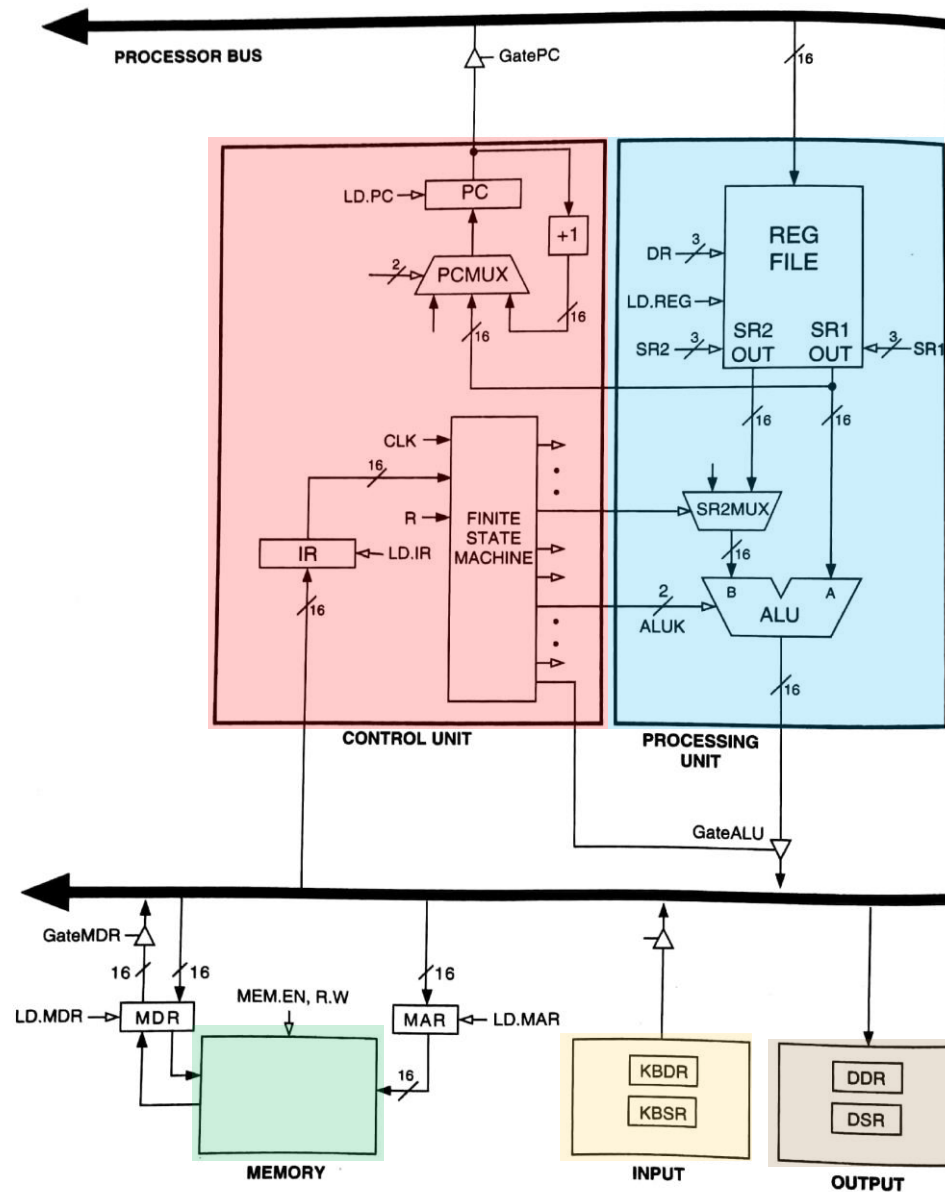
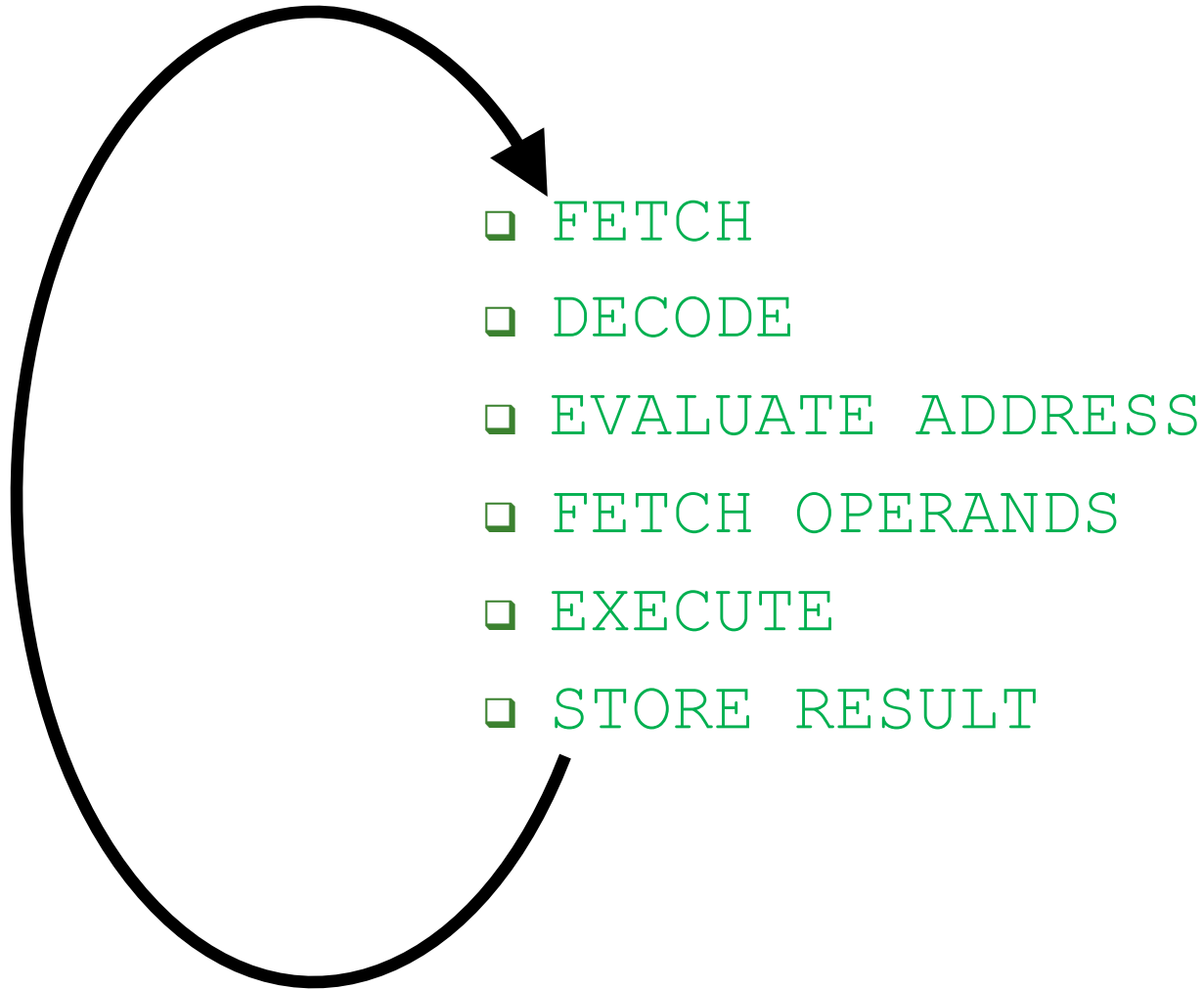


Figure 4.3 The LC-3 as an example of the von Neumann model

Recall: The Instruction Cycle



Recall: The Instruction Set Architecture

- The ISA is the **interface between** what the **software** commands and what the **hardware** carries out
- The ISA specifies
 - The **memory organization**
 - Address space (LC-3: 2^{16} , MIPS: 2^{32})
 - Addressability (LC-3: 16 bits, MIPS: 8 bits)
 - Word- or Byte-addressable
 - The **register set**
 - R0 to R7 in LC-3
 - 32 registers in MIPS
 - The **instruction set**
 - Opcodes
 - Data types
 - Addressing modes
 - Semantics of instructions

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

Microarchitecture

- An **implementation** of the ISA
- How do we implement the ISA?
 - We will discuss this for many lectures
- There can be many implementations of the same ISA
 - MIPS R2000, R10000, ...
 - x86: Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, ... AMD K5, K7, K9, Bulldozer, BobCat, ...
 - IBM POWER 4, 5, 6, 7, 8, 9, 10
 - ARM Cortex-M*, ARM Cortex-A*, NVIDIA Denver, Apple A*, M1, ...
 - Alpha 21064, 21164, 21264, 21364, ...
 - ...

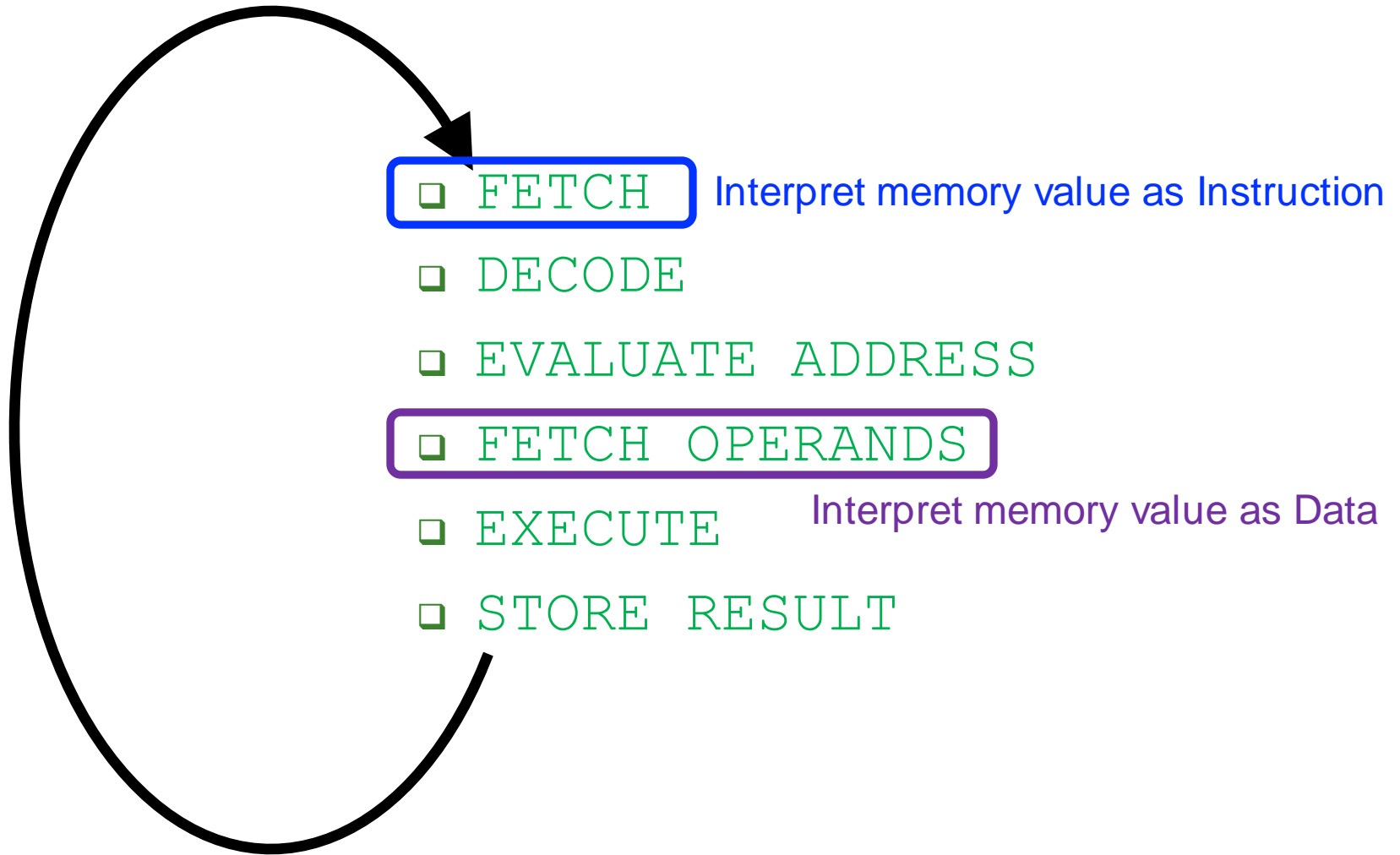
(A Bit More on) ISA Design and Tradeoffs

The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
- **Stored program**
 - Instructions stored in a linear memory array
 - **Memory is unified** between instructions and data
 - The interpretation of a stored value depends on the control signals

When is a value interpreted as an instruction?
- **Sequential instruction processing**

Recall: The Instruction Cycle



Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle.

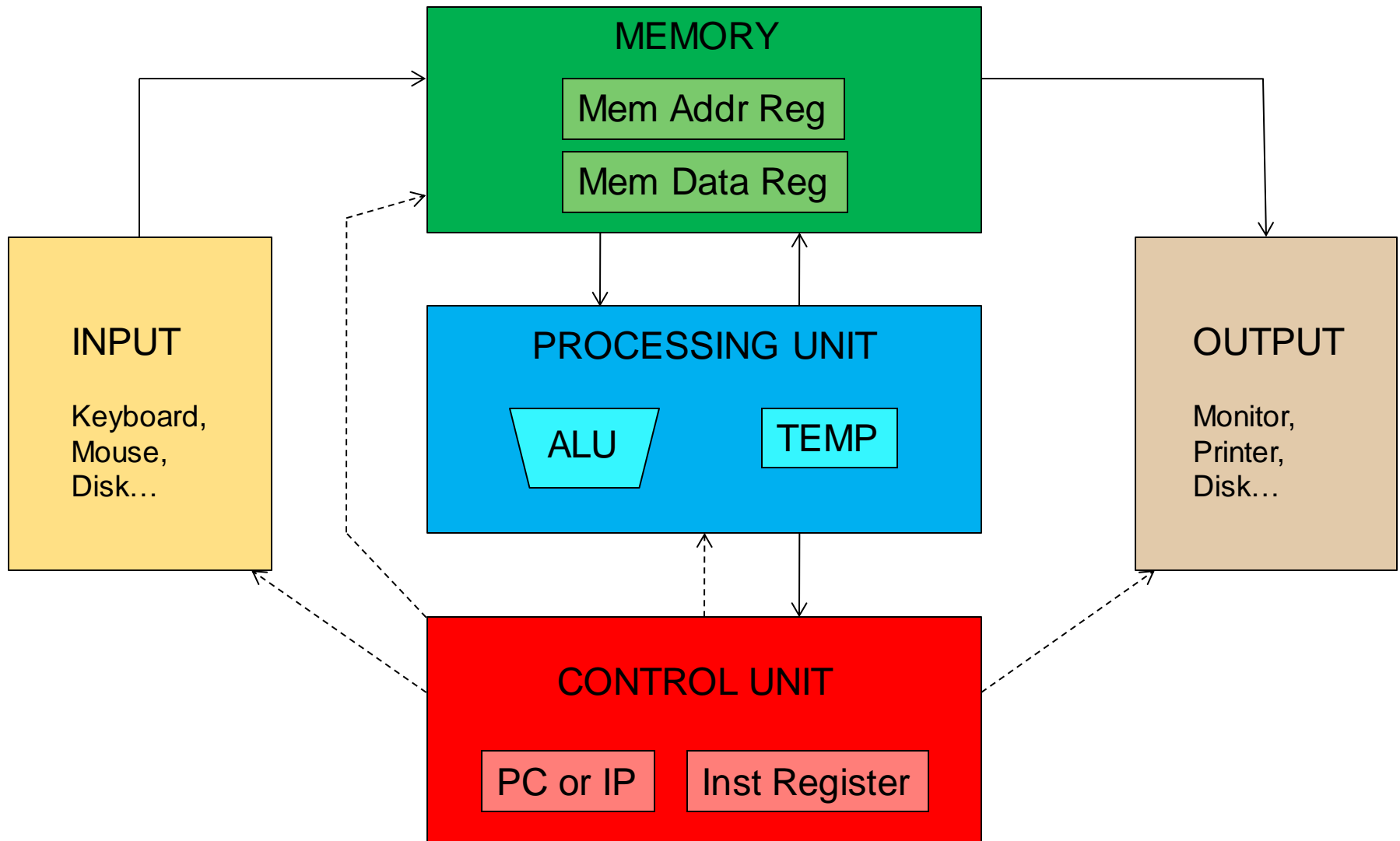
The von Neumann Model/Architecture

- Von Neumann model is also called *stored program computer* (instructions in memory). It has two key properties:
- **Stored program**
 - Instructions stored in a linear memory array
 - **Memory is unified** between instructions and data
 - **The interpretation of a stored value depends on the control signals**
When is a value interpreted as an instruction?
- **Sequential instruction processing**
 - One instruction processed (fetched, executed, completed) at a time
 - **Program counter (instruction pointer)** identifies the current instruction
 - **Program counter is advanced sequentially** except for control transfer instructions

The von Neumann Model/Architecture

- Recommended reading
 - Burks, Goldstein, von Neumann, "Preliminary discussion of the logical design of an electronic computing instrument," 1946.
- Required reading
 - Patt and Patel book, Chapter 4, "The von Neumann Model"
- Stored program
- Sequential instruction processing

The Von Neumann Model (of a Computer)



The Von Neumann Model (of a Computer)

- Q: Is this the only way that a computer can process computer programs?

The von Neumann Model

- In order to build a computer, we need an execution model for processing computer programs

- John von Neumann proposed a fundamental model in 1946

- The von Neumann Model consists of 5 components

- Memory (stores the program and data)
- Processing unit
- Input
- Output
- Control unit (controls the order in which instructions are carried out)



- Throughout this lecture, we will examine two examples of the von Neumann model

- LC-3
- MIPS

Burks, Goldstein, von Neumann,
"Preliminary discussion of the logical design
of an electronic computing instrument," 1946.

- A: No.
- Qualified Answer: No. But, it has been the dominant way
 - i.e., the dominant paradigm for computing
 - for N decades

The Dataflow Execution Model of a Computer

The Dataflow Model (of a Computer)

- Von Neumann model: An instruction is fetched and executed in **control flow order**
 - As specified by the **program counter (instruction pointer)**
 - Sequential unless explicit control flow instruction

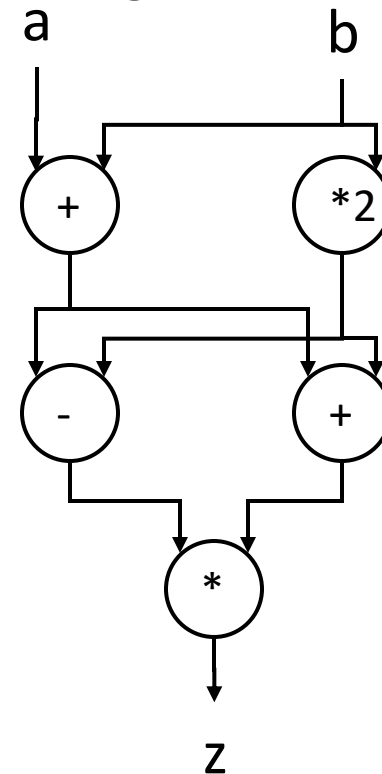
- Dataflow model: An instruction is fetched and executed in **data flow order**
 - i.e., when its operands are ready
 - i.e., there is **no program counter (instruction pointer)**
 - Instruction ordering specified by data flow dependence
 - Each instruction specifies “who” should receive the result
 - An instruction can “fire” whenever all operands are received
 - Potentially many instructions can execute at the same time
 - Inherently more parallel

Von Neumann vs. Dataflow

- Consider a Von Neumann program
 - What is the significance of the program order?
 - What is the significance of the storage locations?

```
v <= a + b;  
w <= b * 2;  
x <= v - w  
y <= v + w  
z <= x * y
```

Sequential

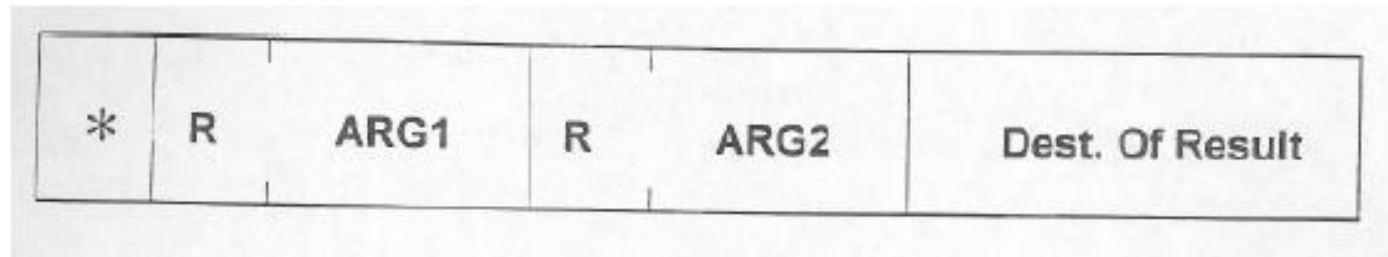
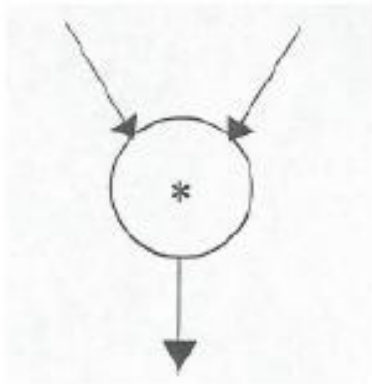


Dataflow

- Which model is more natural to you as a programmer?

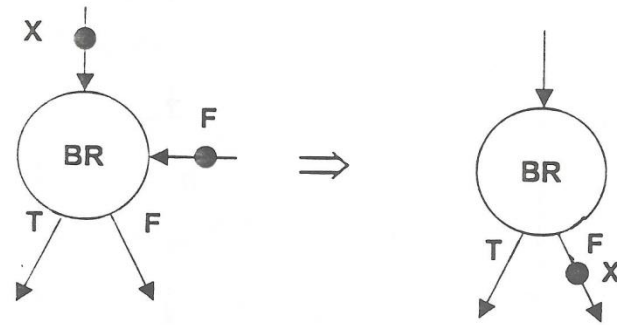
More on Dataflow

- In a dataflow machine, a program consists of dataflow nodes
 - A dataflow node fires (fetched and executed) when all its inputs are ready
 - i.e. when all inputs have tokens
- Dataflow node and its ISA representation

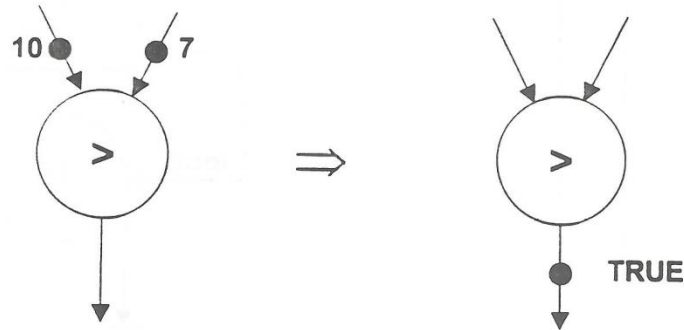


Example Dataflow Nodes

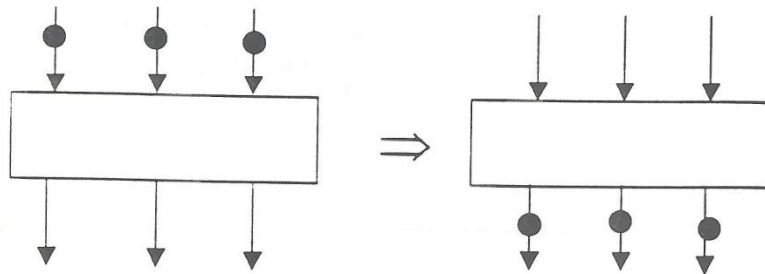
***Conditional**



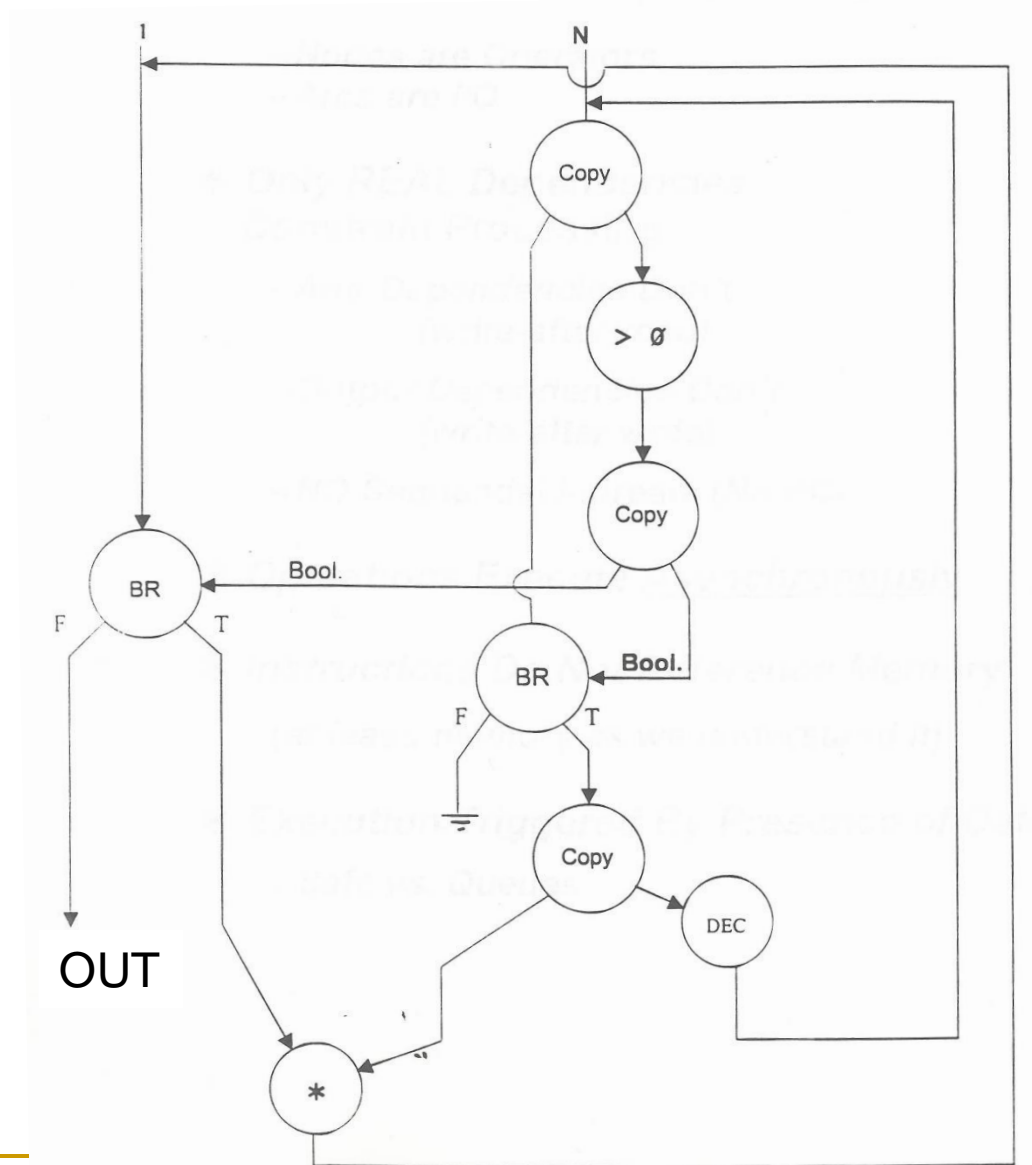
***Relational**



***Barrier Synchron**



A Simple Example Dataflow Program



ISA-level Tradeoff: Program Counter

- Do we need a Program Counter (PC or IP) in the ISA?
 - Yes: Control-driven, sequential execution
 - An instruction is executed when the PC points to it
 - PC automatically changes sequentially (except for control flow instructions)
 - No: Data-driven, parallel execution
 - An instruction is executed when all its operand values are available ([dataflow](#))
- Tradeoffs: MANY high-level ones
 - Ease of programming (for average programmers)?
 - Ease of compilation?
 - Performance: Extraction of parallelism?
 - Hardware complexity?

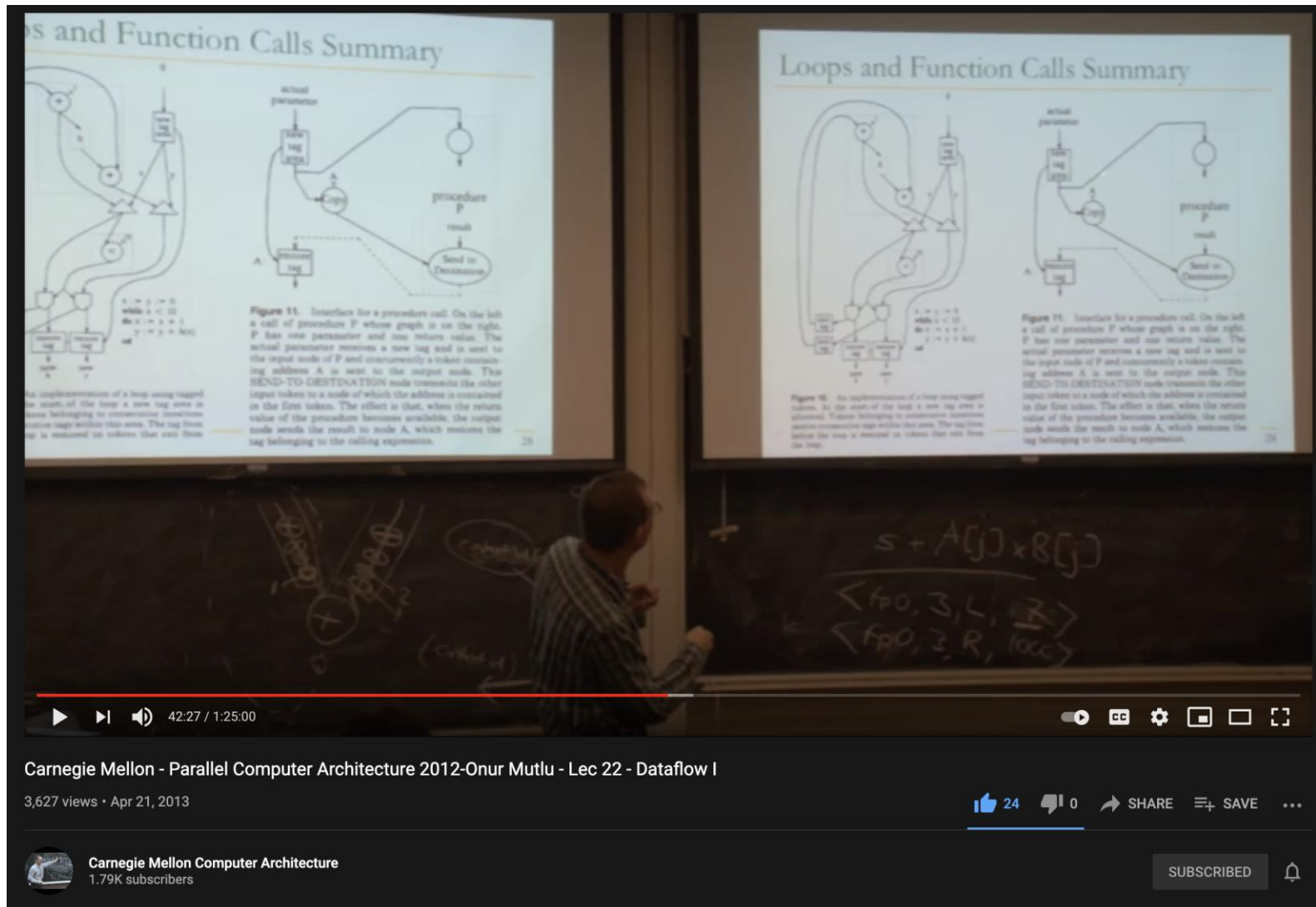
ISA vs. Microarchitecture Level Tradeoff

- A similar tradeoff (control vs. data-driven execution) can be made at the microarchitecture level
- ISA: Specifies how the **programmer sees** the instructions to be executed
 - Programmer sees a sequential, control-flow execution order vs.
 - Programmer sees a dataflow execution order
- Microarchitecture: How the **underlying implementation actually executes** instructions
 - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
 - Programmer should see the order specified by the ISA

Let's Get Back to the von Neumann Model

- But, if you want to learn more about dataflow...
- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- A later lecture
- If you are really impatient:
 - ❑ <http://www.youtube.com/watch?v=D2uue7izU2c>
 - ❑ <http://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.2.1-dataflow-part1.ppt>

Lecture Video on Dataflow Model



The video player shows a lecture on the Dataflow Model. The main content area displays two slides titled "Loops and Function Calls Summary".

Slide 1 (Left): Shows a diagram of a loop using tagged tokens. The diagram includes a loop body with a "SEND-TO-DESTINATION" node. A caption below the diagram reads: "Figure 11. Interface for a procedure call. On the left a call of procedure P whose graph is on the right. P has one parameter and one return value. The actual parameter receives a new tag and is sent to the input node of P and concurrently a token containing address A is sent to the output node. This SEND-TO-DESTINATION node transmits the other input token to a node of which the address is contained in the first token. The effect is that, when the return value of the procedure becomes available, the output node sends the result to node A, which removes the tag belonging to the calling expression."

Slide 2 (Right): Shows a diagram of a loop using tagged tokens. The diagram includes a loop body with a "SEND-TO-DESTINATION" node. A caption below the diagram reads: "Figure 12. An implementation of a loop using tagged tokens. At the start of the loop a new tag area is allocated. Tokens belonging to consecutive iterations arrive consecutively into the area. The tag from before the loop is removed on tokens that exit from the loop."

The video player interface includes a progress bar, play/pause button, volume icon, and a description of the video content.

Video Description: Carnegie Mellon - Parallel Computer Architecture 2012-Onur Mutlu - Lec 22 - Dataflow I

Video Statistics: 3,627 views • Apr 21, 2013

Channel: Carnegie Mellon Computer Architecture (1.79K subscribers)

Engagement: 24 likes, 0 dislikes, SHARE, SAVE, ...

Subscription: SUBSCRIBED

The von Neumann Model

- All major *instruction set architectures* today use this model
 - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, ...
- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
 - Pipelined instruction execution: *Intel 80486 uarch*
 - Multiple instructions at a time: *Intel Pentium uarch*
 - Out-of-order execution: *Intel Pentium Pro uarch*
 - Separate instruction and data caches
- But, what happens underneath that is **not consistent** with the von Neumann model is **not exposed** to software
 - Difference between ISA and microarchitecture

What is Computer Architecture?

- **ISA+implementation definition:** The science and art of designing, selecting, and interconnecting hardware components and designing the hardware/software interface to create a computing system that meets functional, performance, energy consumption, cost, and other specific goals.
- **Traditional (ISA-only) definition:** “The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior *as distinct from* the organization of the dataflow and controls, the logic design, and the physical implementation.”
Gene Amdahl, IBM Journal of R&D, April 1964

ISA vs. Microarchitecture

■ ISA

- Agreed upon interface between software and hardware
 - SW/compiler assumes, HW promises
- What the software writer needs to know to write and debug system/user programs

■ Microarchitecture

- Specific implementation of an ISA
- Not visible to the software

■ Microprocessor

- **ISA, uarch**, circuits
- “Architecture” = ISA + microarchitecture

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
 - Gas pedal: interface for “acceleration”
 - Internals of the engine: implement “acceleration”
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
 - Add instruction vs. Adder implementation
 - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture (**see H&H Chapter 5.2.1**)
 - x86 ISA has many implementations:
 - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, Comet Lake, AMD K5, K7, K9, Bulldozer, BobCat, ...
- Microarchitecture usually changes faster than ISA
 - Few ISAs (x86, ARM, SPARC, MIPS, Alpha, RISC-V) but many uarchs
 - *Why?*

■ Instructions

- ❑ Opcodes, Addressing Modes, Data Types
- ❑ Instruction Types and Formats
- ❑ Registers, Condition Codes

■ Memory

- ❑ Address space, Addressability, Alignment
- ❑ Virtual memory management

■ Call, Interrupt/Exception Handling

■ Access Control, Priority/Privilege

■ I/O: memory-mapped vs. instr.

■ Task/thread Management

■ Power and Thermal Management

■ Multi-threading support, Multiprocessor support

■ ...



Intel® 64 and IA-32 Architectures
Software Developer's Manual

Volume 1:
Basic Architecture

Microarchitecture

- Implementation of the ISA under specific **design constraints and goals**
- Anything done in hardware without exposure to software
 - Pipelining
 - In-order versus out-of-order instruction execution
 - Memory access scheduling policy
 - Speculative execution
 - Superscalar processing (multiple instruction issue?)
 - Clock gating
 - Caching? Levels, size, associativity, replacement policy
 - Prefetching?
 - Voltage/frequency scaling?
 - Error correction?

Property of ISA vs. Uarch?

- ADD instruction's opcode
 - Bit-serial adder vs. Ripple-carry adder
 - Number of general purpose registers
 - Number of cycles to execute the MUL instruction
 - Number of ports to the register file
 - Whether or not the machine employs pipelined instruction execution
-
- Remember
 - Microarchitecture: Implementation of the ISA under specific design constraints and goals

Design Point

- A set of design considerations and their importance
 - ❑ **leads to tradeoffs** in both ISA and uarch
- Example considerations:
 - ❑ Cost
 - ❑ Performance
 - ❑ Maximum power consumption, thermal
 - ❑ Energy consumption (battery life)
 - ❑ Availability
 - ❑ Reliability and Correctness
 - ❑ Time to Market
 - ❑ Security, safety, predictability, ...
- Design point determined by the “Problem” space (application space), the intended users/*market*

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

Application Space

Dream, and they will appear...

Other examples of the application space that continue to drive the need for unique design points are the following:

- 1) **scientific applications** such as those whose computations control nuclear power plants, determine where to drill for oil, and predict the weather;
- 2) **transaction-based applications** such as those that handle ATM transfers and e-commerce business;
- 3) **business data processing** applications, such as those that handle inventory control, payrolls, IRS activity, and various personnel record keeping, whether the personnel are employees, students, or voters;
- 4) **network applications**, such as high-speed routing of Internet packets, that enable the connection of your home system to take advantage of the Internet;
- 5) **guaranteed delivery (a.k.a. real time)** applications that require the result of a computation by a certain critical deadline;
- 6) **embedded applications**, where the processor is a component of a larger system that is used to solve the (usually) dedicated application;
- 7) **media applications** such as those that decode video and audio files;
- 8) random software packages that desktop users would like to run on their PCs.

Each of these application areas has a very different set of characteristics. Each application area demands a different set of tradeoffs to be made in specifying the microprocessor to do the job.

Patt, "Requirements, bottlenecks, and good fortune: agents for microprocessor evolution,"
Proc. of the IEEE 2001.

Many other workloads:

Genome analysis

Machine learning

Robotics

Web search

Graph analytics

...

Increasingly Demanding Applications

Dream

and, they will come

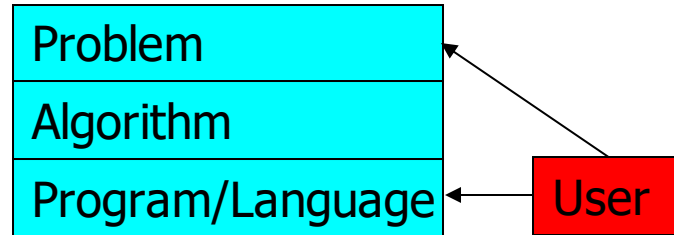
As applications push boundaries, computing platforms will become increasingly strained.

Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Microarchitecture-level tradeoffs
- System and Task-level tradeoffs
 - How to divide the labor between hardware and software
- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
 - *Why **art**?*

Why Is It (Somewhat) Art?

New demands
from the top
(Look Up)



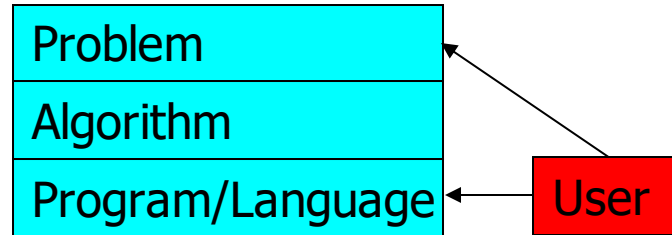
New demands and
personalities of users
(Look Up)

New issues and
capabilities
at the bottom
(Look Down)

- We do not (fully) know the future (applications, users, market)

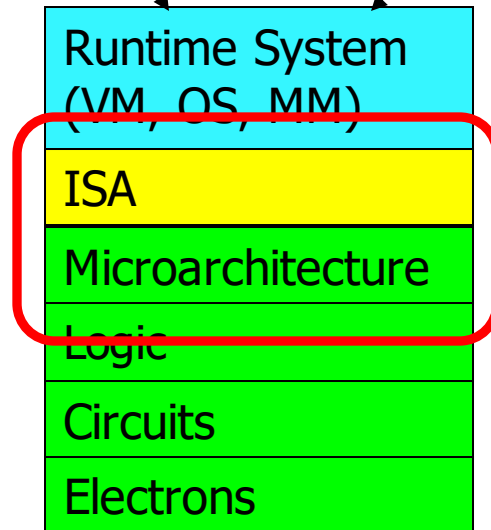
Why Is It (Somewhat) Art?

Changing demands
at the top
(Look Up and Forward)



Changing demands and
personalities of users
(Look Up and Forward)

Changing issues and
capabilities
at the bottom
(Look Down and Forward)



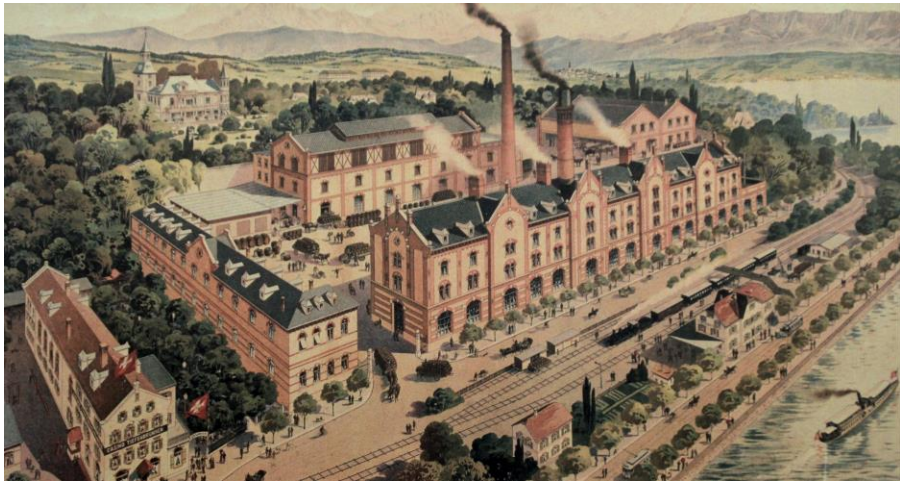
- And, the future is not constant (it changes)!

Analogue from Macro-Architecture

- Future is not constant in macro-architecture, either
- Example: Can a mill be later used as a theater + restaurant + conference room?

Mühle Tiefenbrunnen

- Originally built as a **brewery** in 1889, part of it was converted into a **mill** in 1913, and the other part into a **cold store**
- Today is a **center for a variety of activities**: theater, conferences, restaurants, shops, museum...



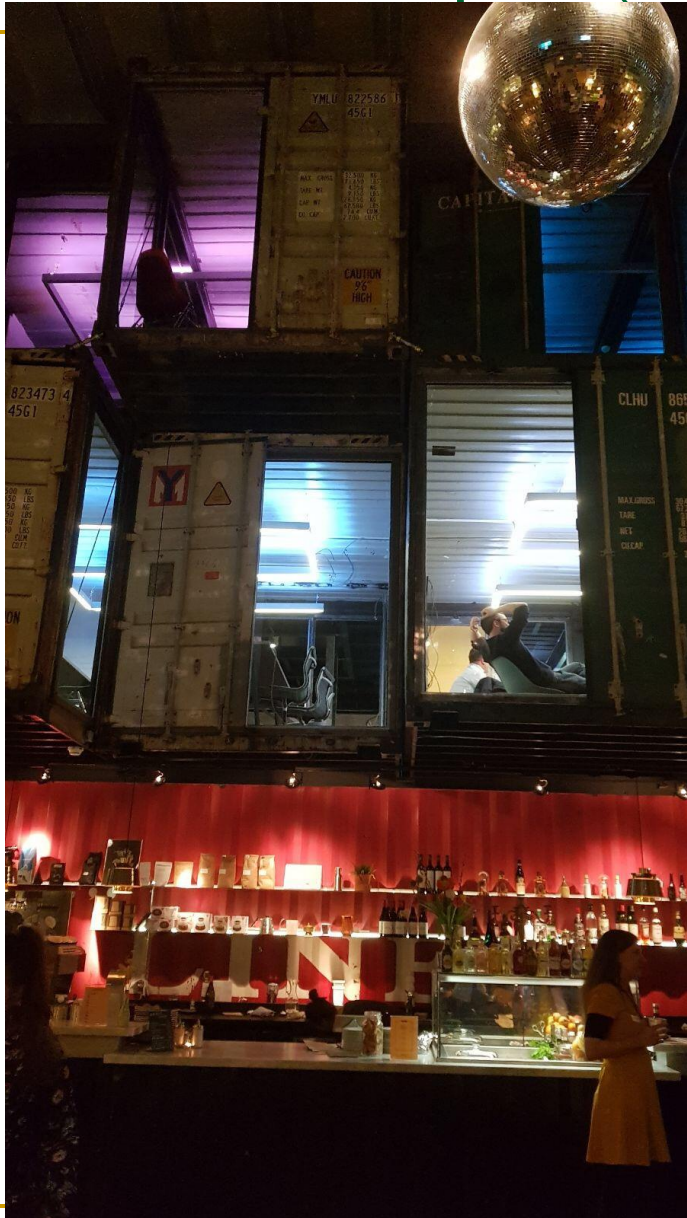
Brewery in 1900

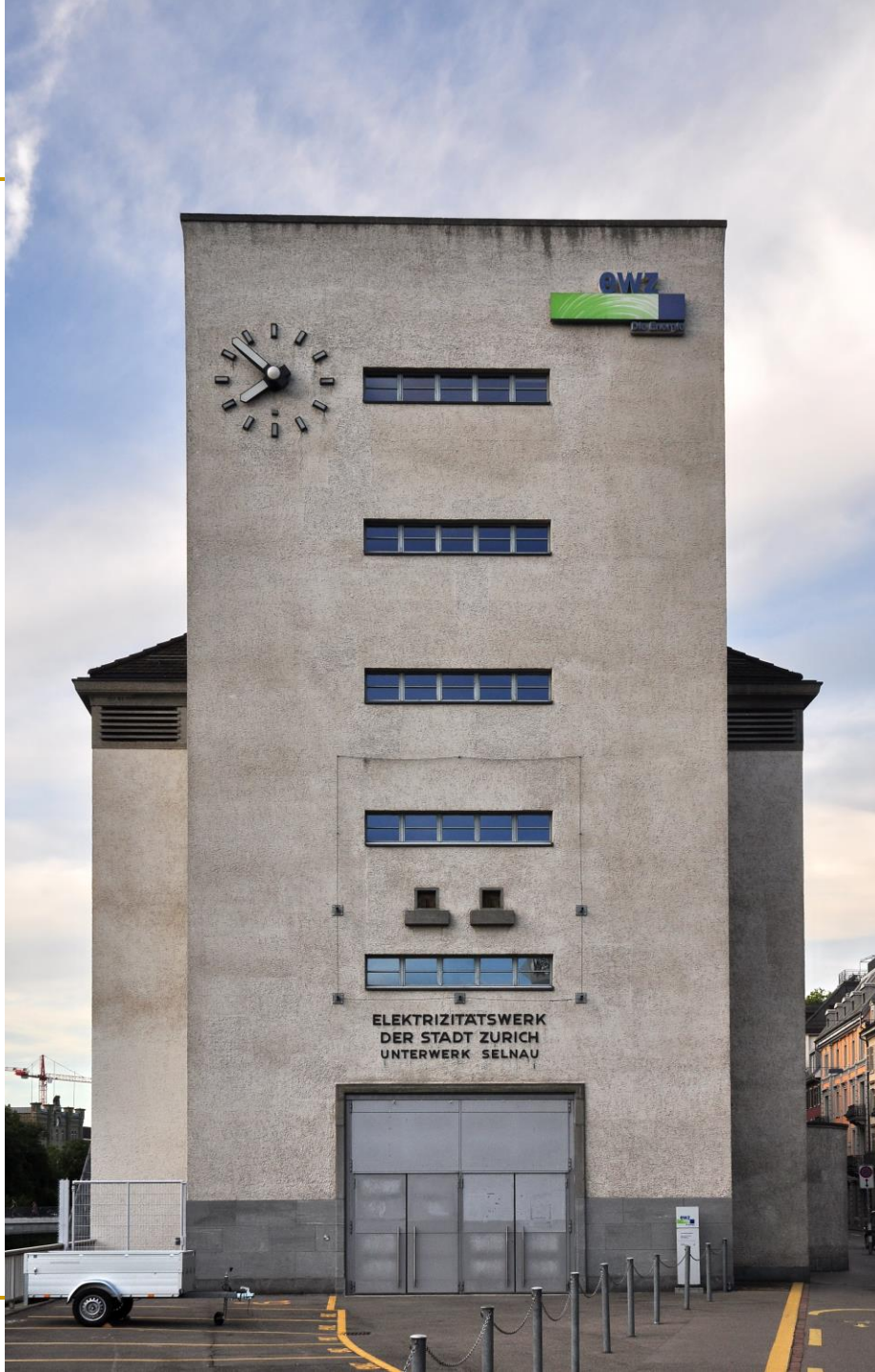


Another Example (I)



Another Example (II)





By Roland zh (Own work) [CC BY-SA 3.0
(<https://creativecommons.org/licenses/by-sa/3.0/>)],
via Wikimedia Commons

Implementing the ISA: Microarchitecture Basics

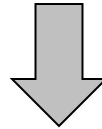
Now That We Have an ISA

- 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
 - “Translation of ISAs”
 - A **virtual ISA** can be converted by “software” into an **implementation ISA**
- We will assume “hardware” implementation for most lectures

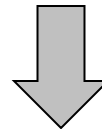
How Does a Machine Process Instructions?

- What does processing an instruction mean?
- We will assume the von Neumann model (for now)

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



Process instruction



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

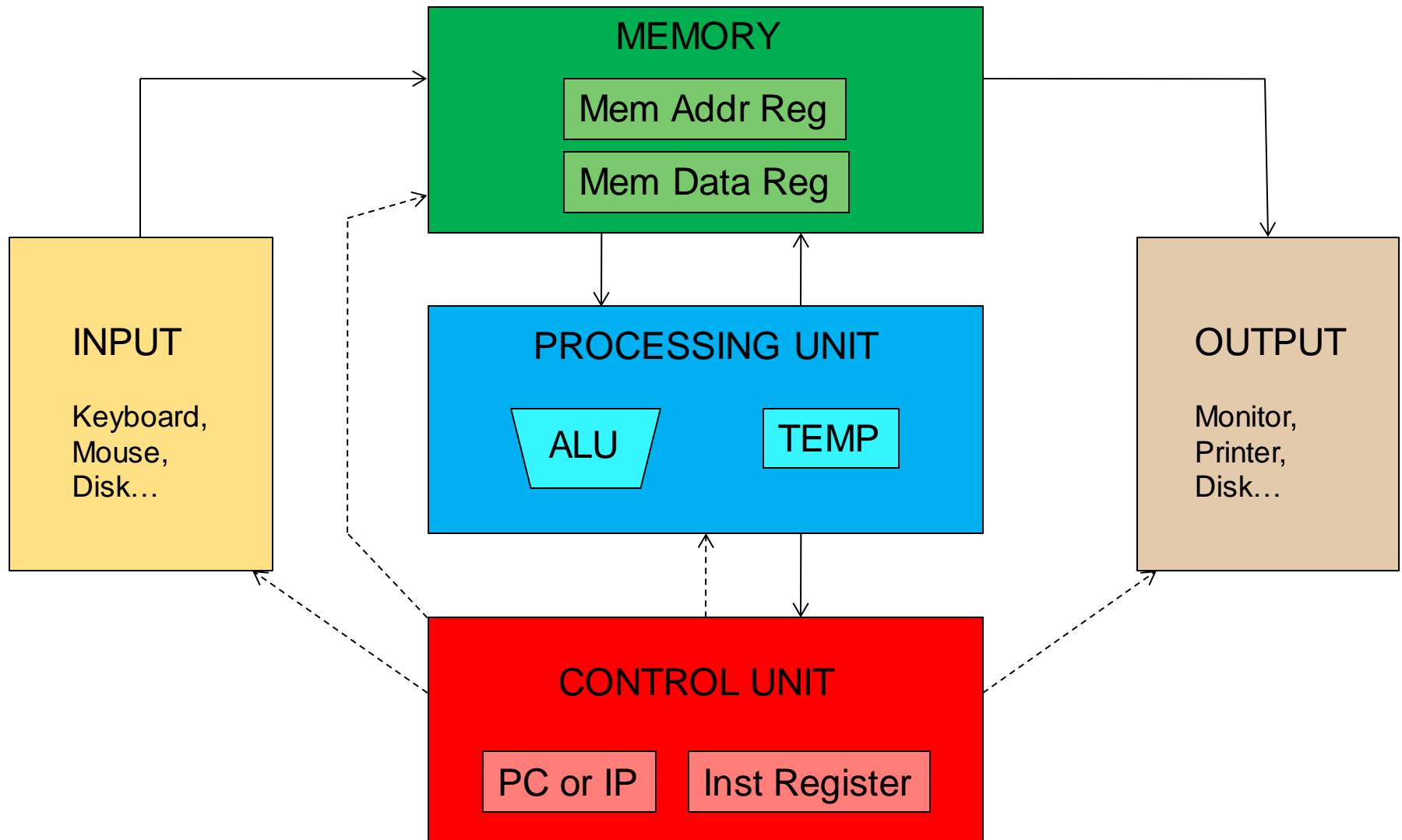
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

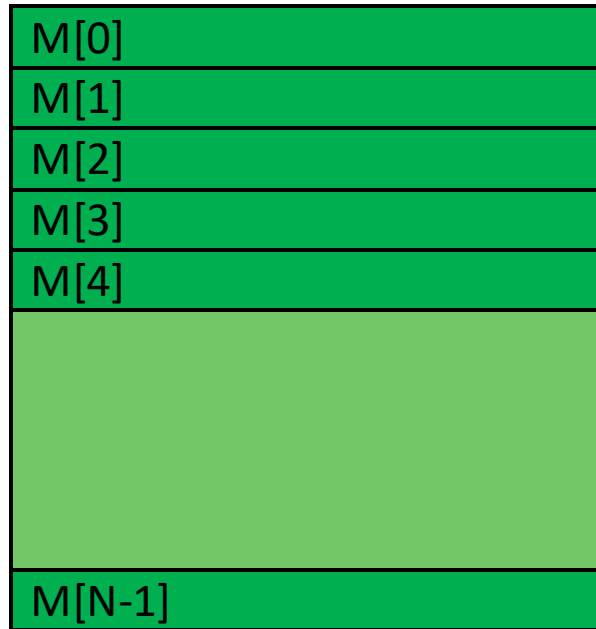
Stored program

Sequential instruction processing

Recall: The Von Neumann Model



Recall: 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 (or next) instruction

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

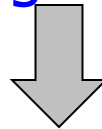
The “Process Instruction” Step

- 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: $AS \rightarrow AS'$ (transform AS to AS' in a single clock cycle)
 - Choice 2: $AS \rightarrow AS+MS1 \rightarrow AS+MS2 \rightarrow AS+MS3 \rightarrow AS'$ (take multiple clock cycles to transform AS to AS')

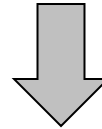
A Very Basic Instruction Processing Engine

- 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



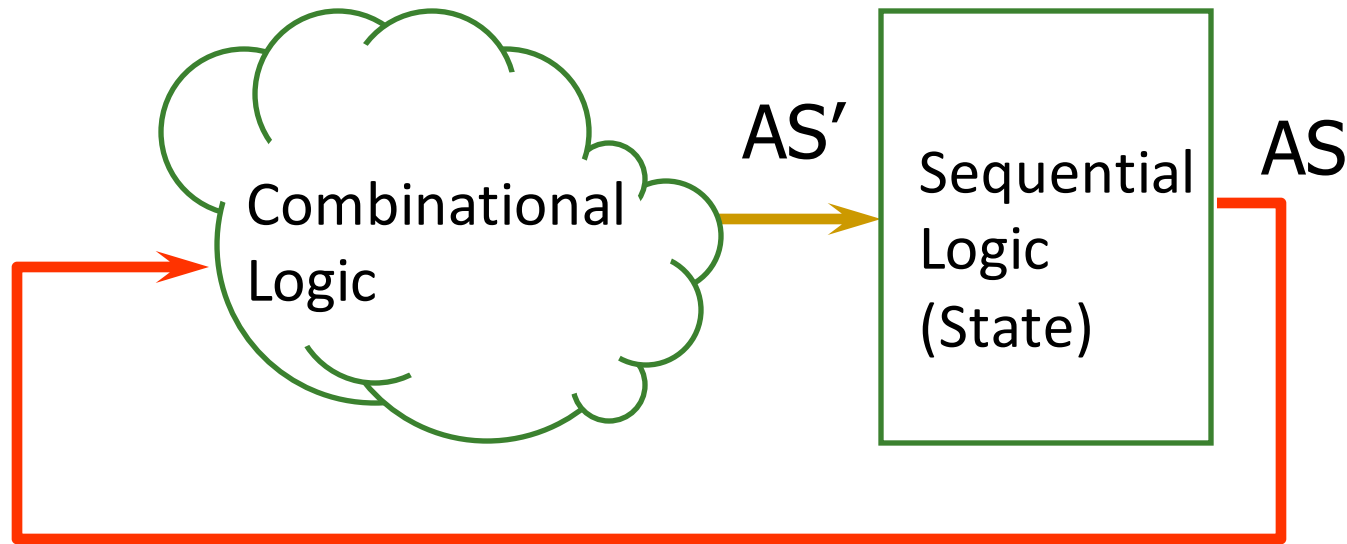
Process instruction in one clock cycle



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* (i.e., longest delay path) of the combinational logic determined by?

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

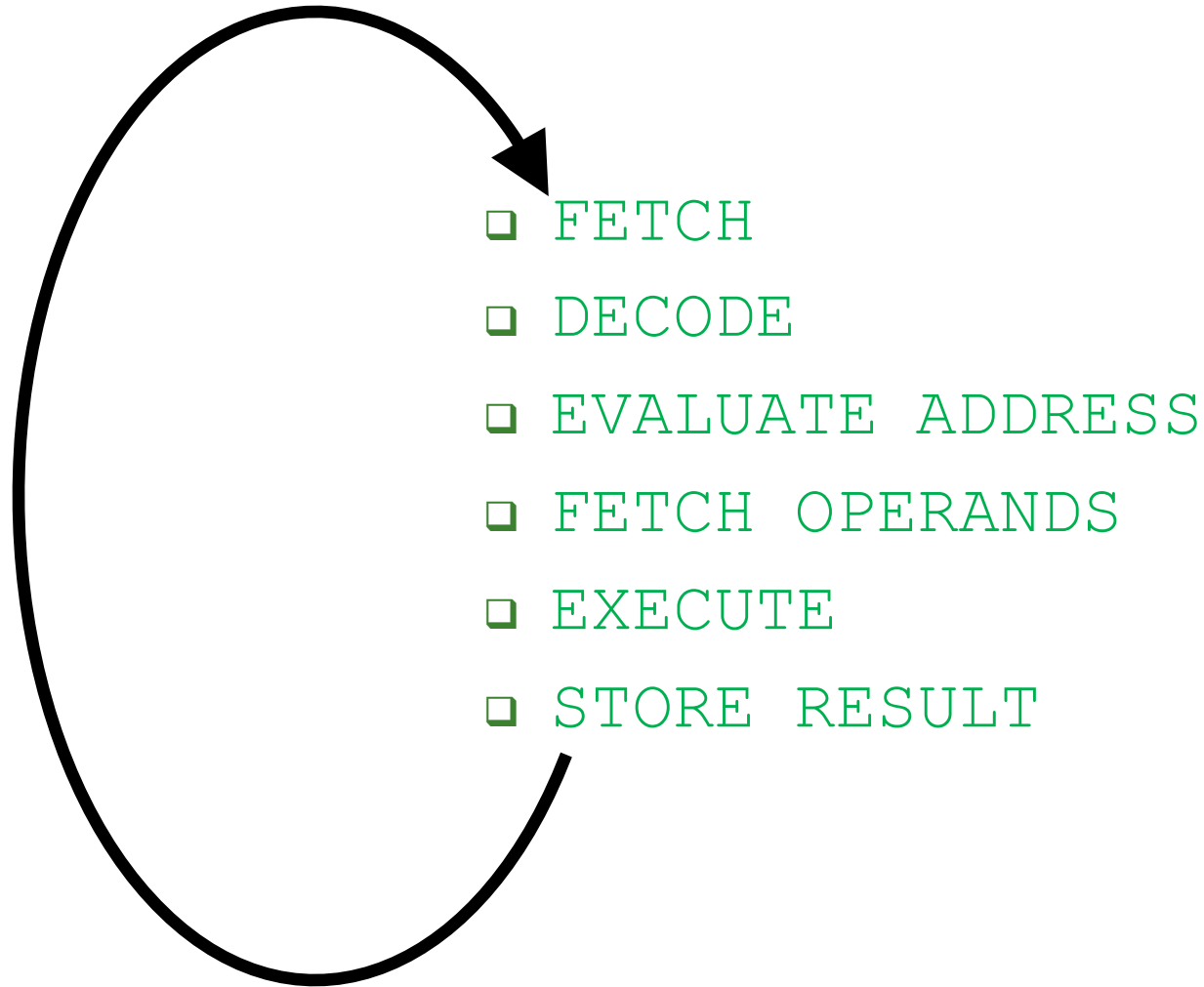
- ❑ Instruction processing broken into multiple cycles/stages
- ❑ State updates can be made during an instruction's execution
- ❑ Architectural state updates made at the end of an instruction's execution
- ❑ Advantage over single-cycle: The slowest "stage" determines cycle time

- 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 steps:
 - Fetch
 - Decode
 - Evaluate Address
 - Fetch Operands
 - Execute
 - Store Result
- Not all instructions require all six steps (see P&P Ch. 4)

Recall: The Instruction Processing “Cycle”



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, muxes, decoders, tri-state bufs) 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**

Recall: LC-3: A von Neumann Machine

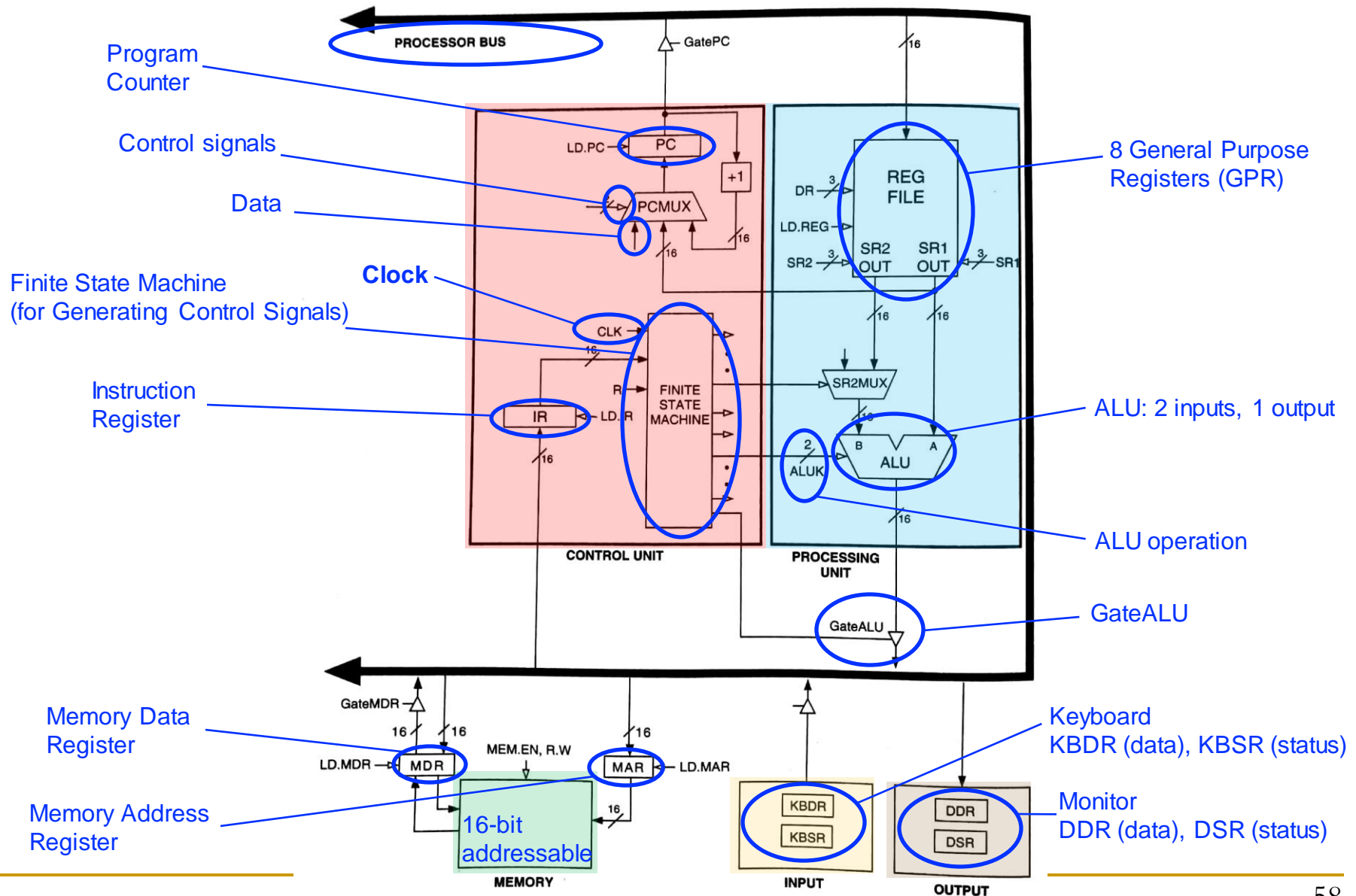


Figure 4.3 The LC-3 as an example of the von Neumann model

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)
- See P&P Appendix C for more (microprogrammed multi-cycle microarchitecture)

Many Ways of Datapath and Control Design

- There are many ways of designing the datapath and control logic
- Example ways
 - Single-cycle, multi-cycle, pipelined datapath and control
 - Single-bus vs. multi-bus datapaths
 - 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 a single instruction
 - **{CPI} x {clock cycle time}** CPI: Cycles Per Instruction
- Execution time of an entire program
 - Sum over all instructions [**{CPI} x {clock cycle time}**]
 - **{# of instructions} x {Average CPI} x {clock cycle time}**
- Single-cycle microarchitecture performance
 - $\text{CPI} = 1$
 - Clock cycle time = long
- Multi-cycle microarchitecture performance
 - CPI = different for each instruction
 - Average CPI \rightarrow hopefully small
 - Clock cycle time = short

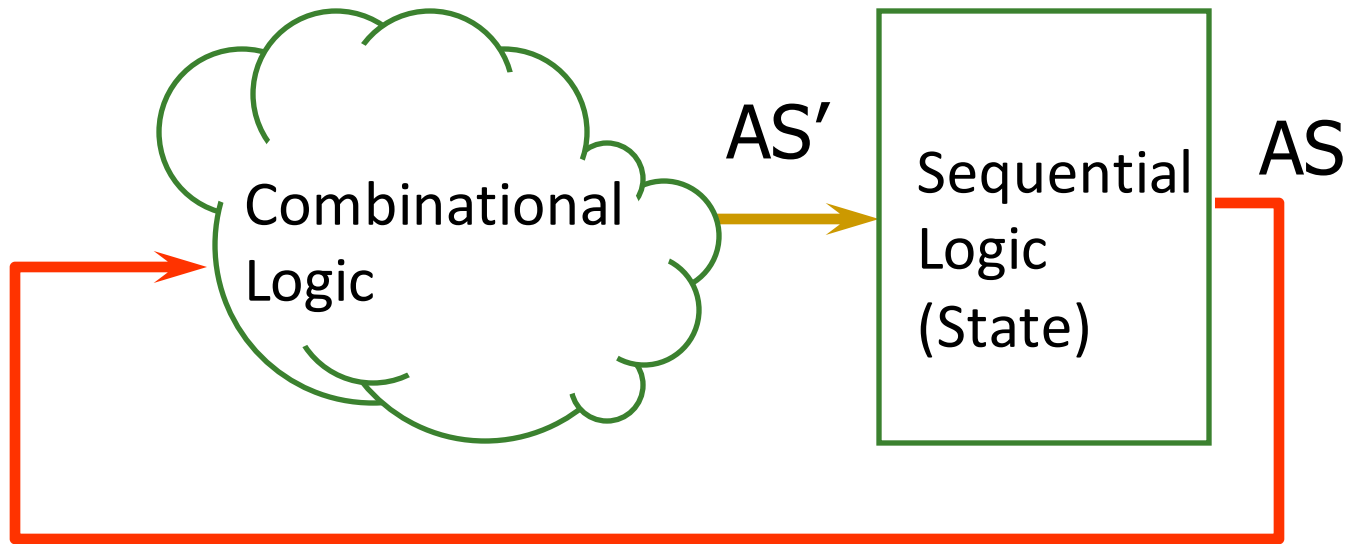
In multi-cycle, we have two degrees of freedom to optimize independently

A Single-Cycle Microarchitecture

A Closer Look

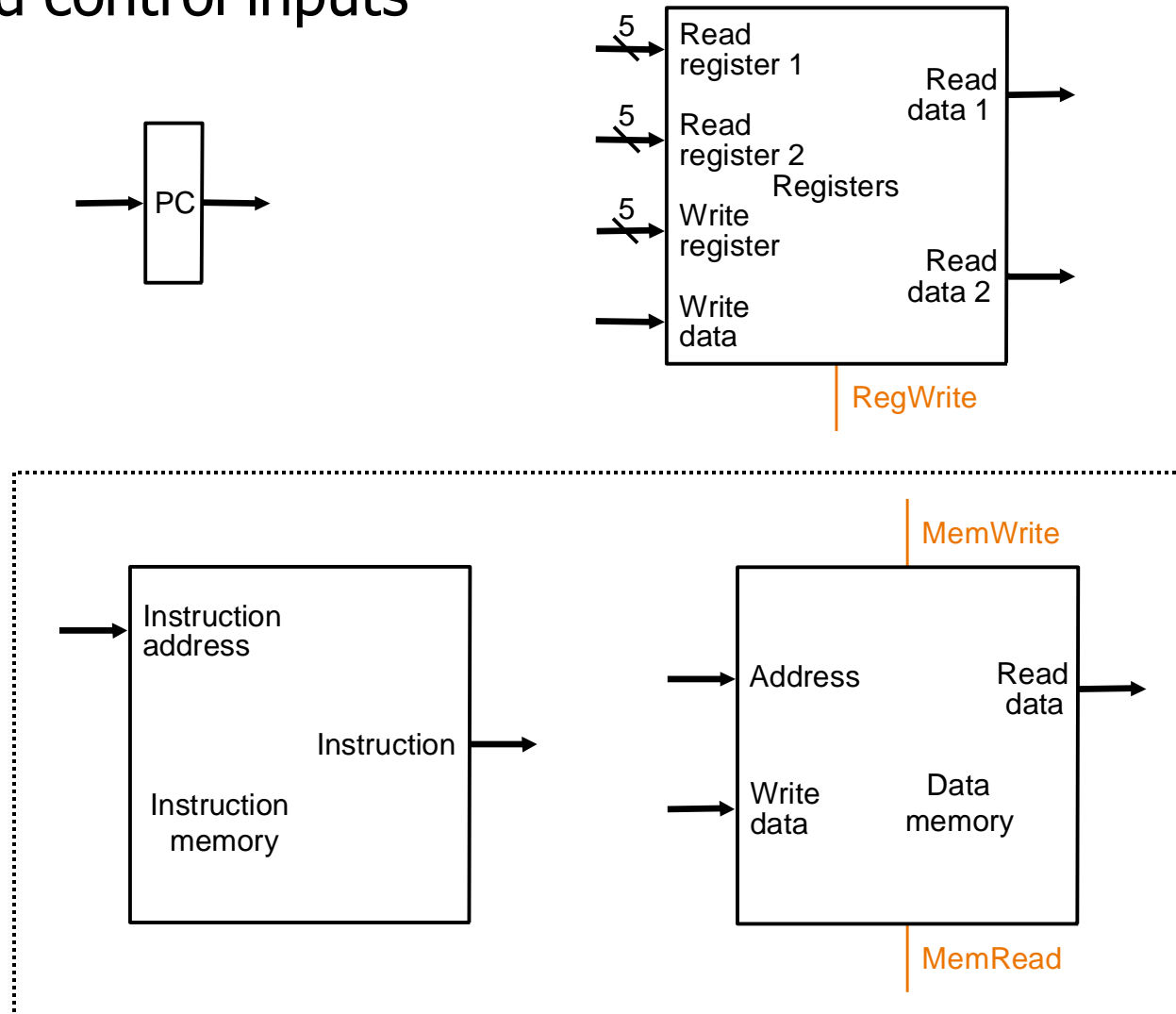
Remember...

- Single-cycle machine

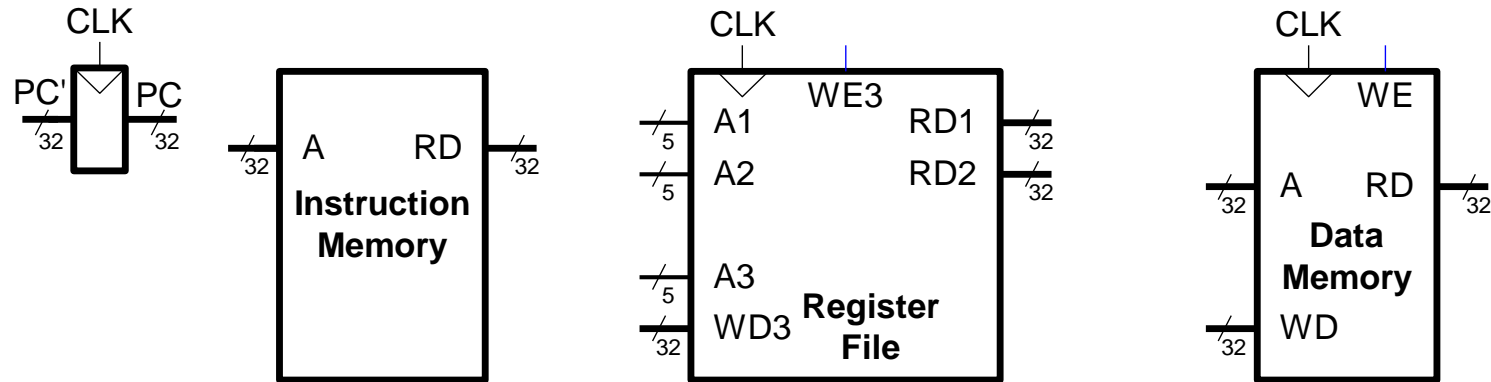


Let's Start with the State Elements

■ Data and control inputs



MIPS State Elements



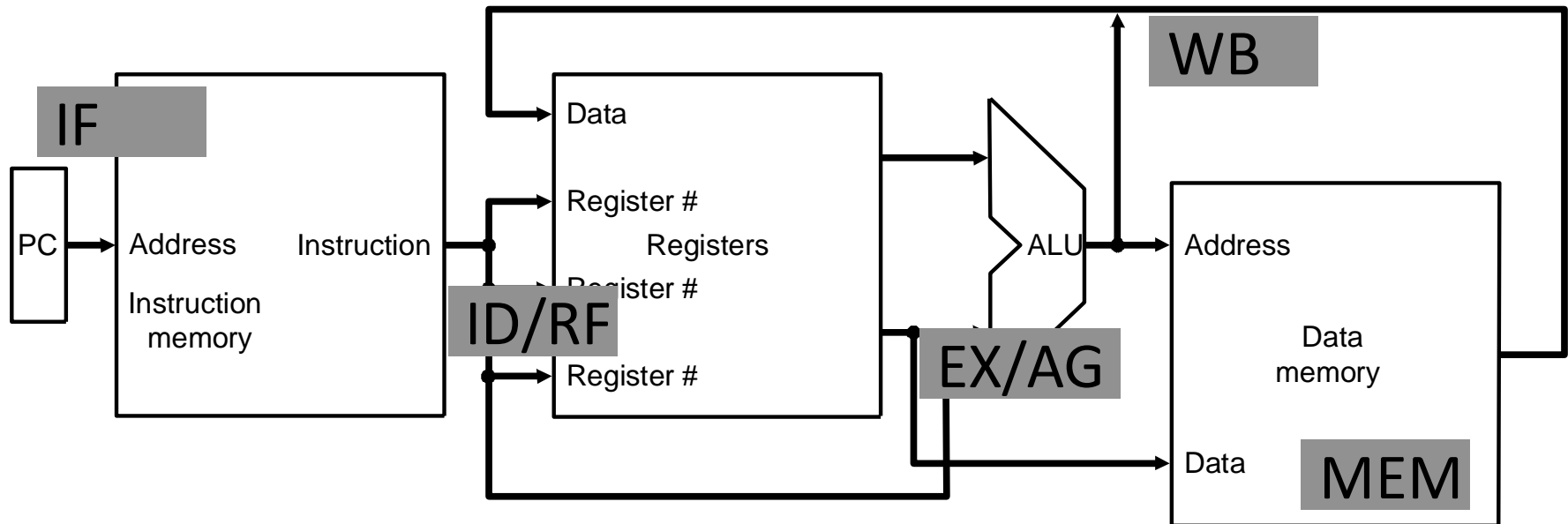
- ❑ **Program counter:**
32-bit register
- ❑ **Instruction memory:**
Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD.
- ❑ **Register file:**
The 32-element, 32-bit register file has 2 read ports and 1 write port
- ❑ **Data memory:**
If the write enable, WE, is 1, it writes 32-bit data WD into memory location at 32-bit address A on the rising edge of the clock.
If the write enable is 0, it reads 32-bit data from address A onto RD.

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 signal: indicating the read or write is done
 - See P&P Appendix C (LC3-b) for multi-cycle memory

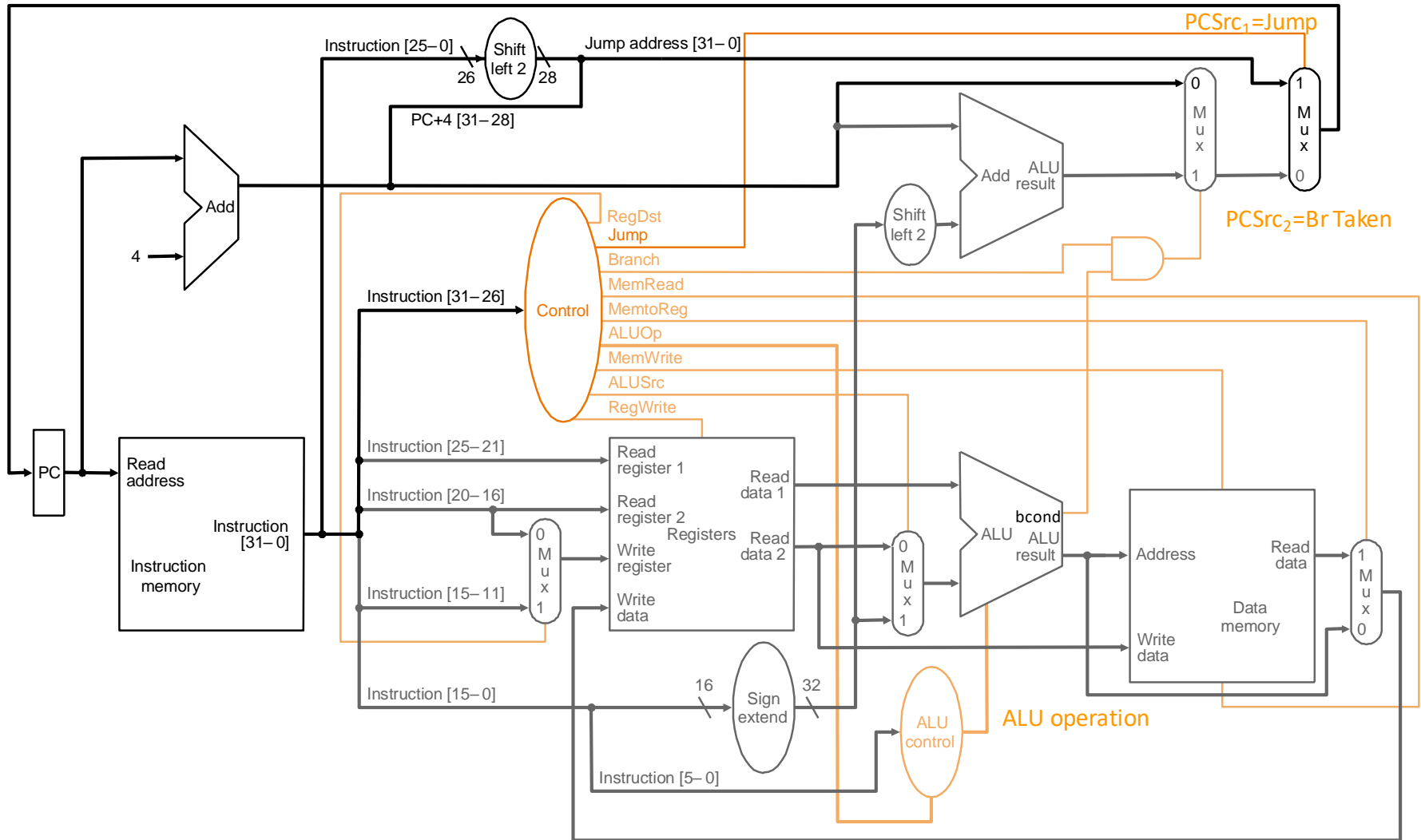
Instruction Processing

- 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)

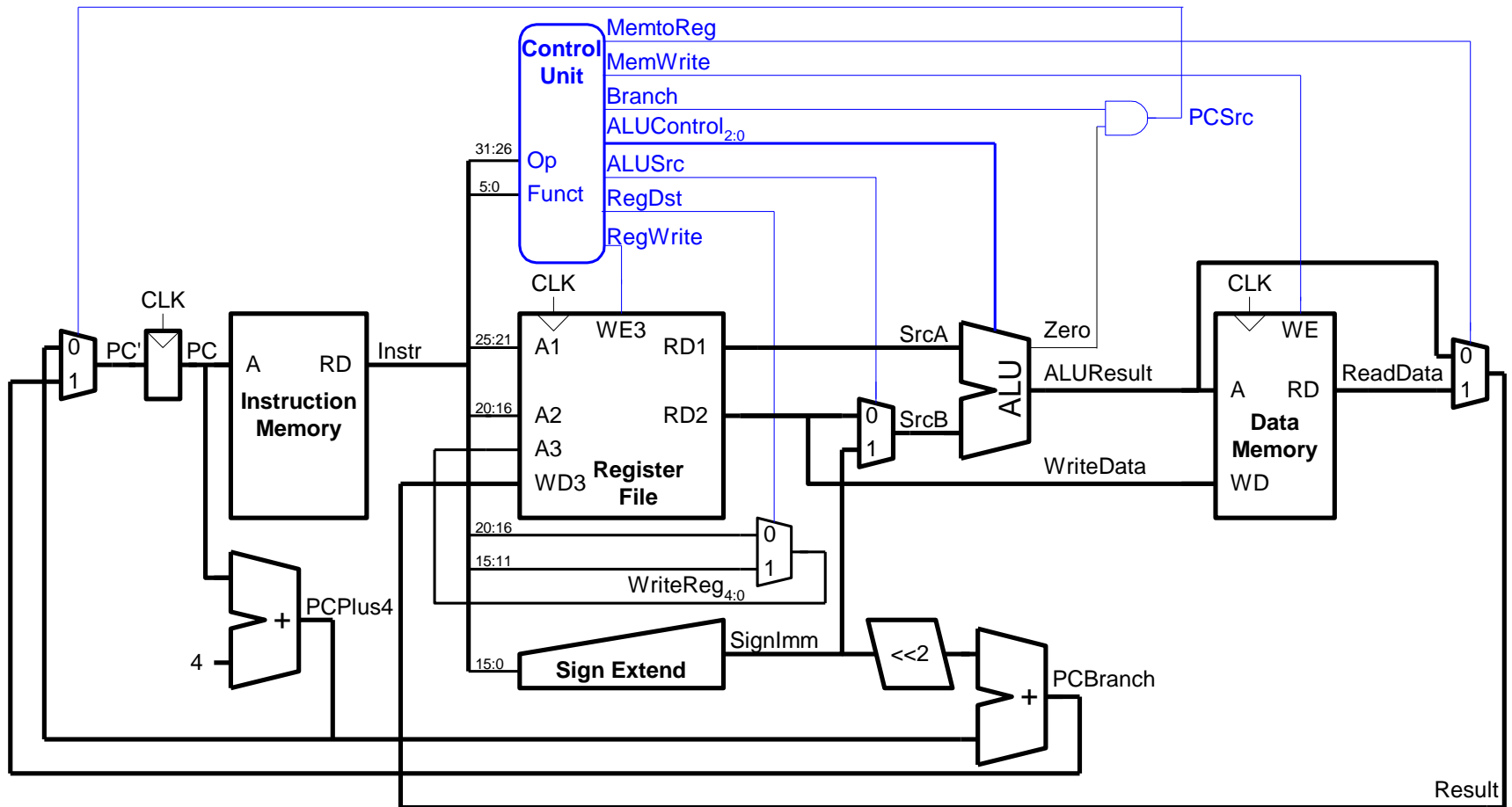


We Need to Provide the
Datapath+Control Logic
to Execute All ISA Instructions

What Is To Come: The Full MIPS Datapath



Another Complete Single-Cycle Processor



Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- R-type: 3 register operands

MIPS assembly (e.g., register-register signed addition)

```
add    $s0, $s1, $s2    # $s0=rd, $s1=rs, $s2=rt
```

Machine Encoding

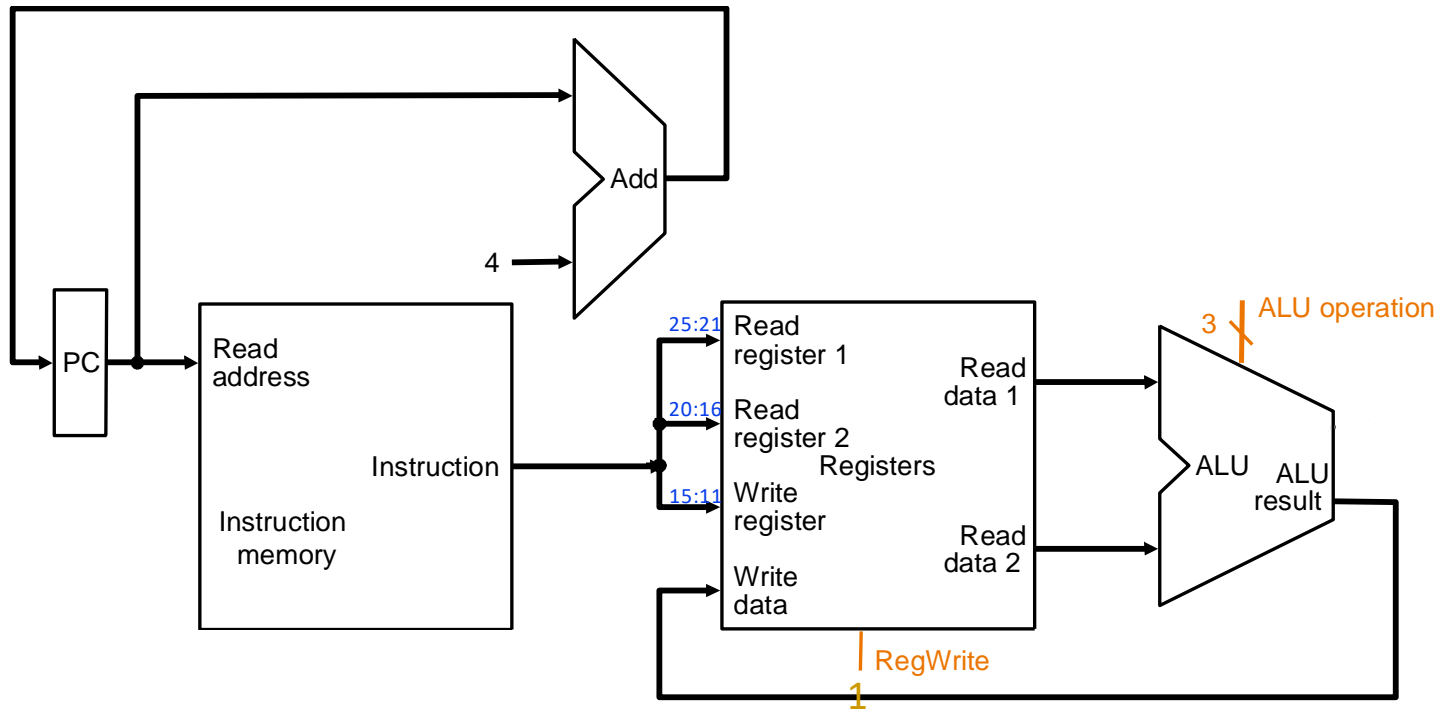
0	rs	rt	rd	0	add (32)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R-Type

- Semantics

```
if MEM[PC] == add rd rs rt
    GPR[rd] ← GPR[rs] + GPR[rt]
    PC ← PC + 4
```


(R-Type) ALU Datapath



IF	ID	EX	MEM	WB
----	----	----	-----	----

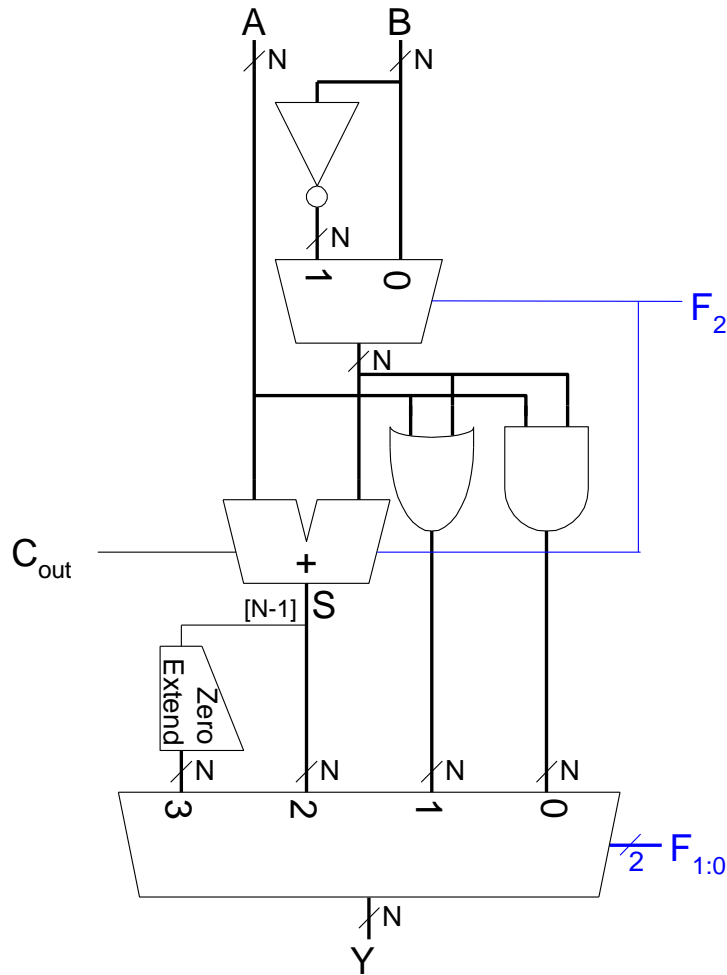


Combinational
state update logic

if $\text{MEM}[\text{PC}] == \text{ADD rd rs rt}$
 $\text{GPR}[\text{rd}] \leftarrow \text{GPR}[\text{rs}] + \text{GPR}[\text{rt}]$
 $\text{PC} \leftarrow \text{PC} + 4$

Example: ALU Design

- ALU operation ($F_{2:0}$) comes from the control logic



$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

I-Type ALU Instructions

- I-type: 2 register operands and 1 immediate

MIPS assembly (e.g., register-immediate signed addition)

```
addi $s0, $s1, 5    # $s0 = rt, $s1 = rs
```

Machine Encoding

addi (0)	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

I-Type

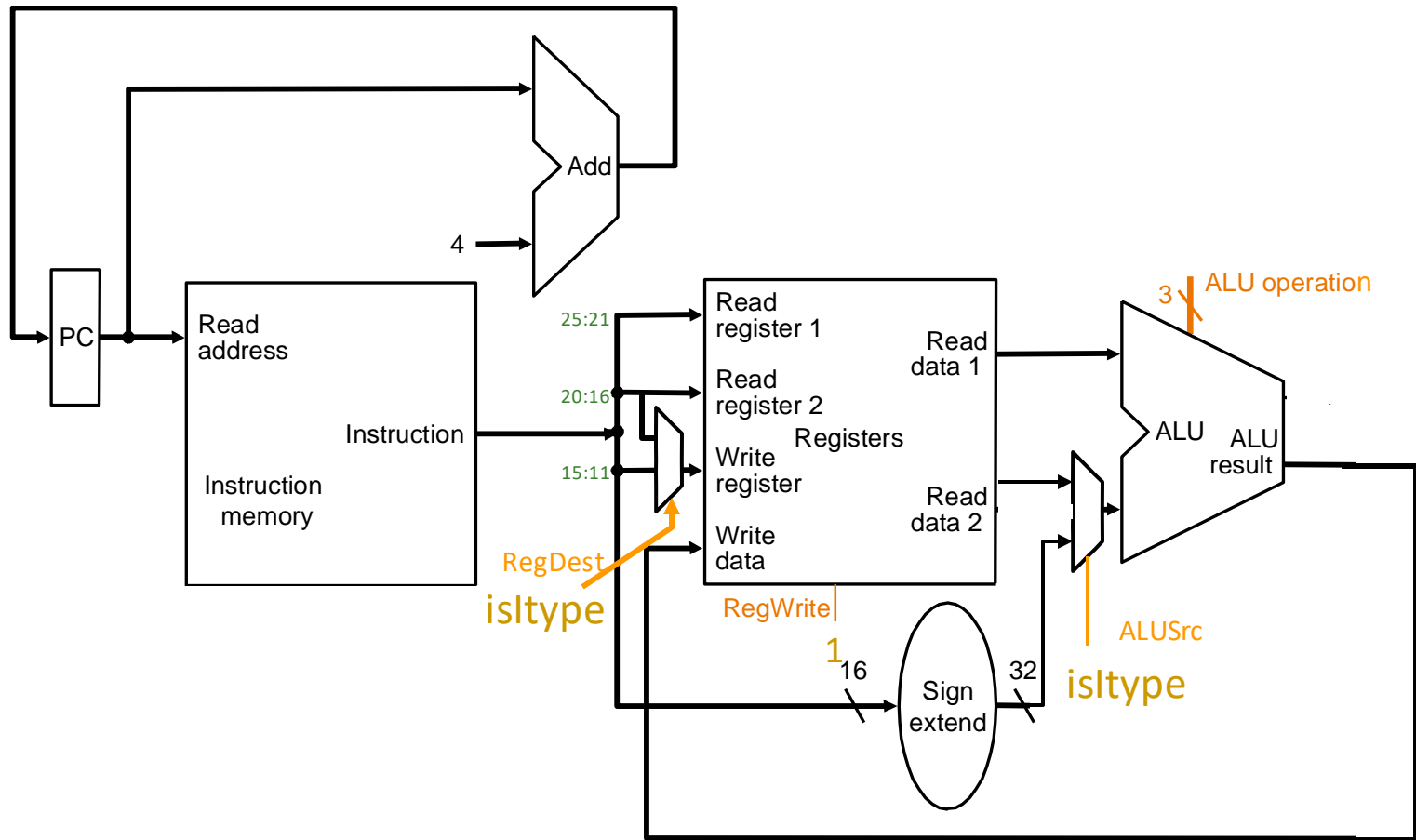
- Semantics

if MEM[PC] == addi rs rt immediate

PC \leftarrow PC + 4

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

Datapath for R- and I-Type ALU Insts.



IF	ID	EX	MEM	WB
----	----	----	-----	----



Combinational
state update logic

if MEM[PC] == ADDI rt rs immediate
 $GPR[rt] \leftarrow GPR[rs] + \text{sign-extend}(\text{immediate})$
 $PC \leftarrow PC + 4$

Recall: ADD with one Literal in LC-3

■ ADD assembly and machine code

LC-3 assembly

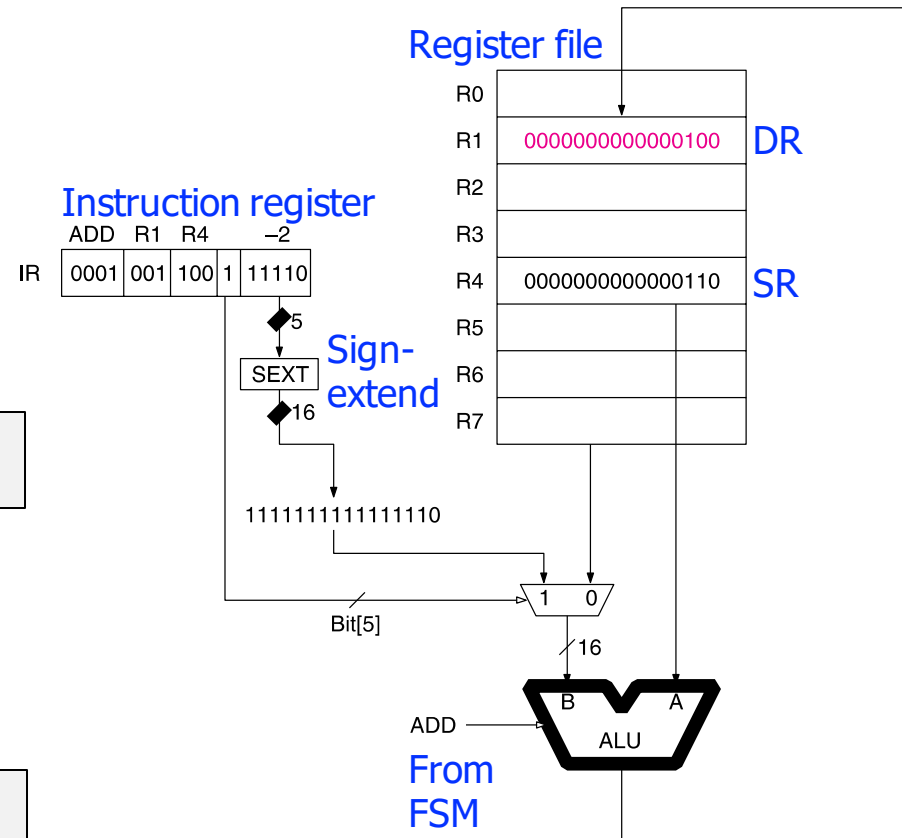
```
ADD R1, R4, #-2
```

Field Values

OP	DR	SR		imm5
1	1	4	1	-2

Machine Code

OP				DR		SR		imm5						
0001				001		100		1	11110					
15		12		11		9		8	6		5	4	0	



Single-Cycle Datapath for *Data Movement Instructions*

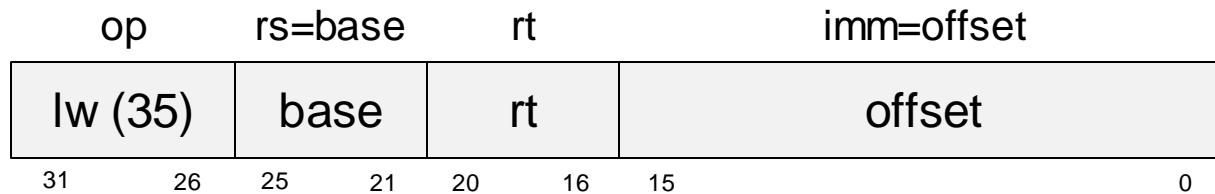
Load Instructions

■ Load 4-byte word

MIPS assembly

```
lw    $s3, 8($s0)    # $s0=rs, $s3=rt
```

Machine Encoding

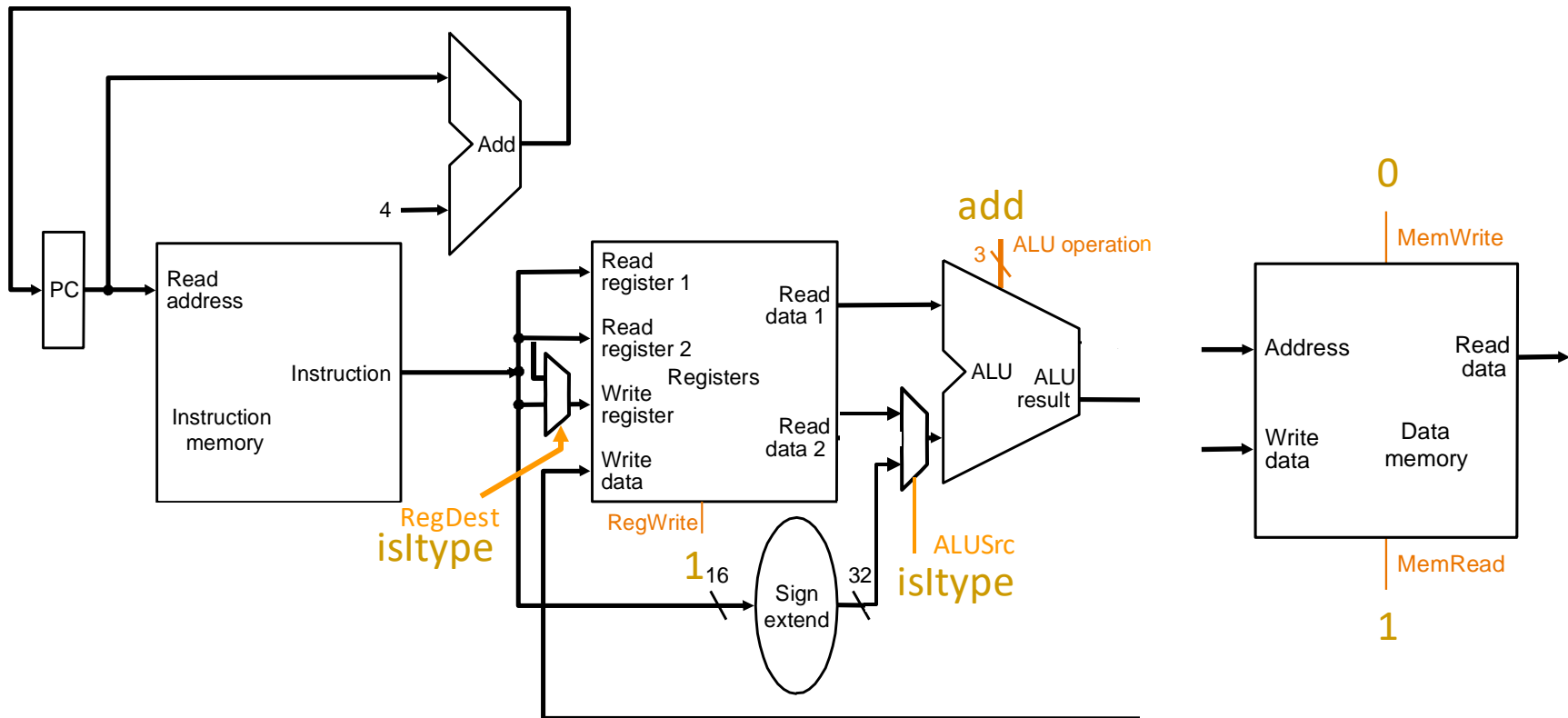


I-Type

■ Semantics

if $\text{MEM}[\text{PC}] == \text{lw rt offset}_{16}(\text{base})$
 $\text{PC} \leftarrow \text{PC} + 4$
 $\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base})$
 $\text{GPR}[\text{rt}] \leftarrow \text{MEM}[\text{translate}(\text{EA})]$

LW Datapath



if $\text{MEM}[\text{PC}] == \text{LW rt offset}_{16}(\text{base})$
 $\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}]$
 $\text{GPR}[\text{rt}] \leftarrow \text{MEM}[\text{translate}(\text{EA})]$
 $\text{PC} \leftarrow \text{PC} + 4$

IF	ID	EX	MEM	WB
----	----	----	-----	----

Combinational
state update logic

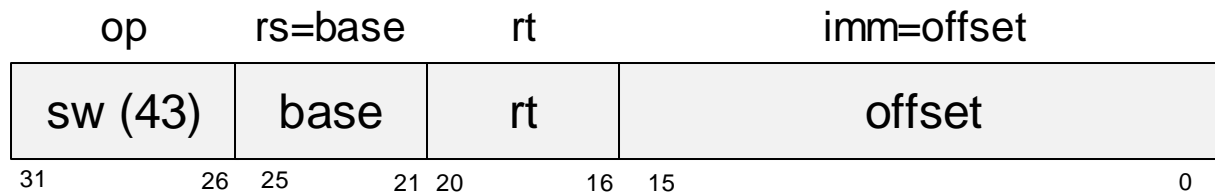
Store Instructions

■ Store 4-byte word

MIPS assembly

```
sw    $s3, 8($s0)  # $s0=rs, $s3=rt
```

Machine Encoding

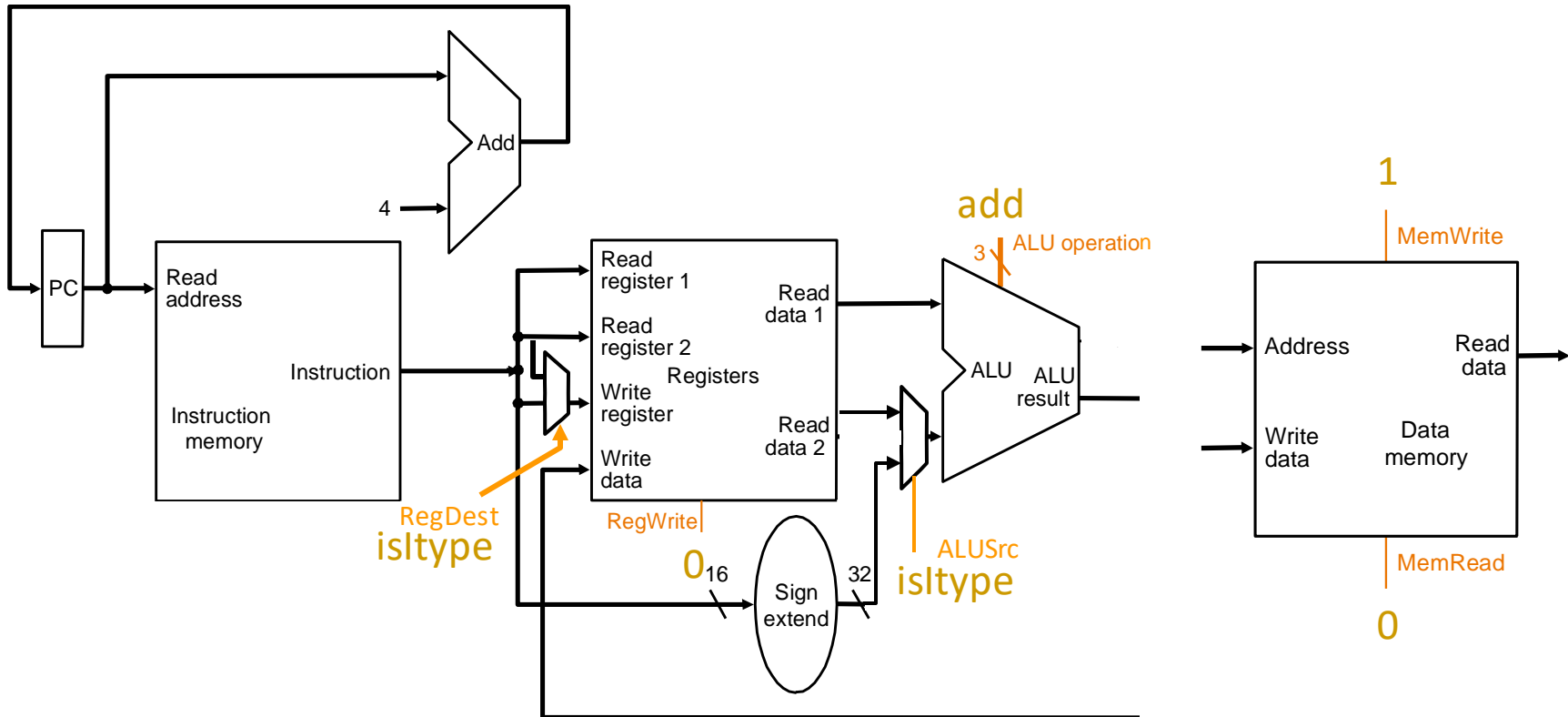


I-Type

■ Semantics

if $\text{Mem}[\text{PC}] == \text{sw rt offset}_{16}(\text{base})$
 $\text{PC} \leftarrow \text{PC} + 4$
 $\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base})$
 $\text{MEM}[\text{translate}(\text{EA})] \leftarrow \text{GPR}[\text{rt}]$

SW Datapath

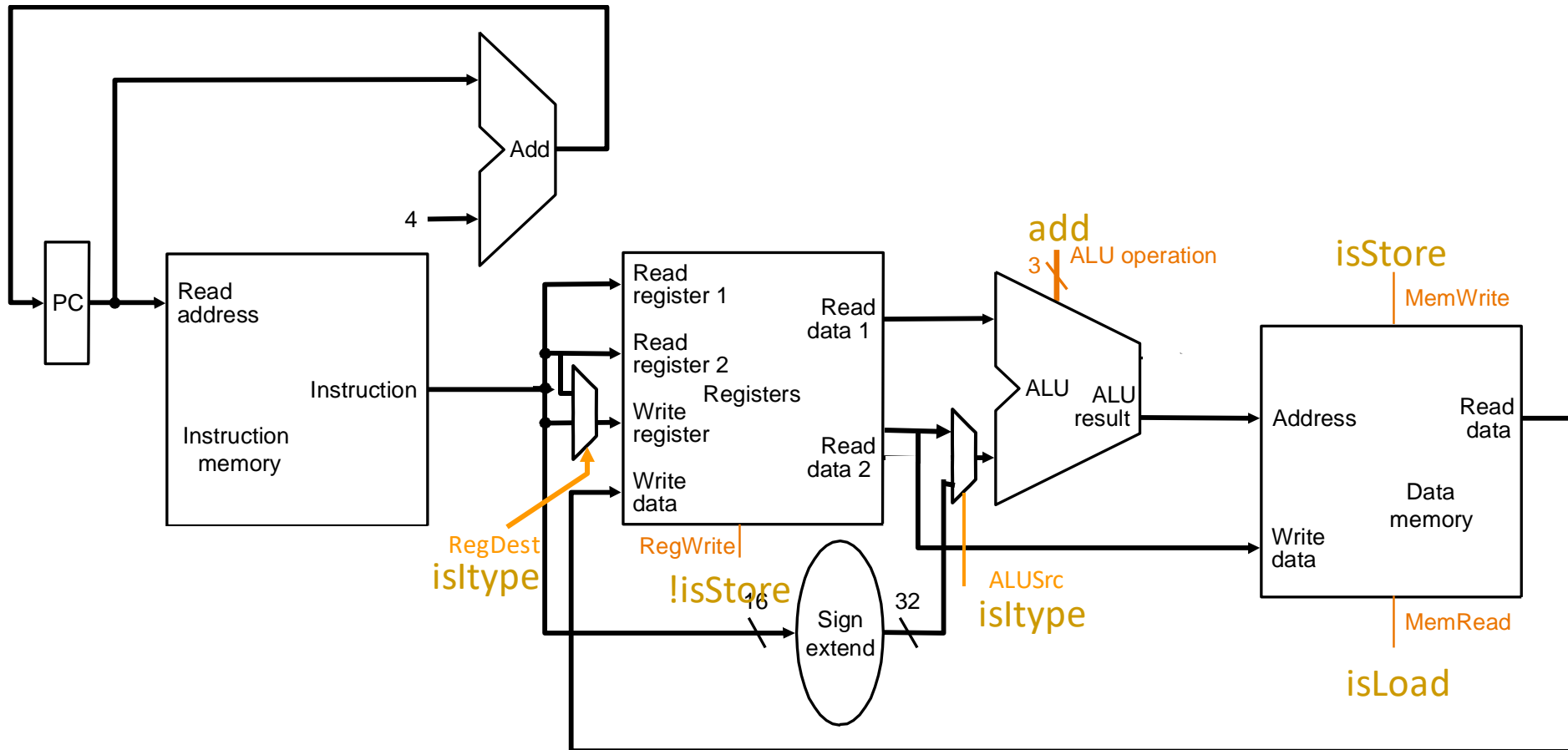


if $\text{MEM}[\text{PC}] == \text{SW rt offset}_{16}(\text{base})$
 $\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}[\text{base}]$
 $\text{MEM}[\text{translate}(\text{EA})] \leftarrow \text{GPR}[\text{rt}]$
 $\text{PC} \leftarrow \text{PC} + 4$

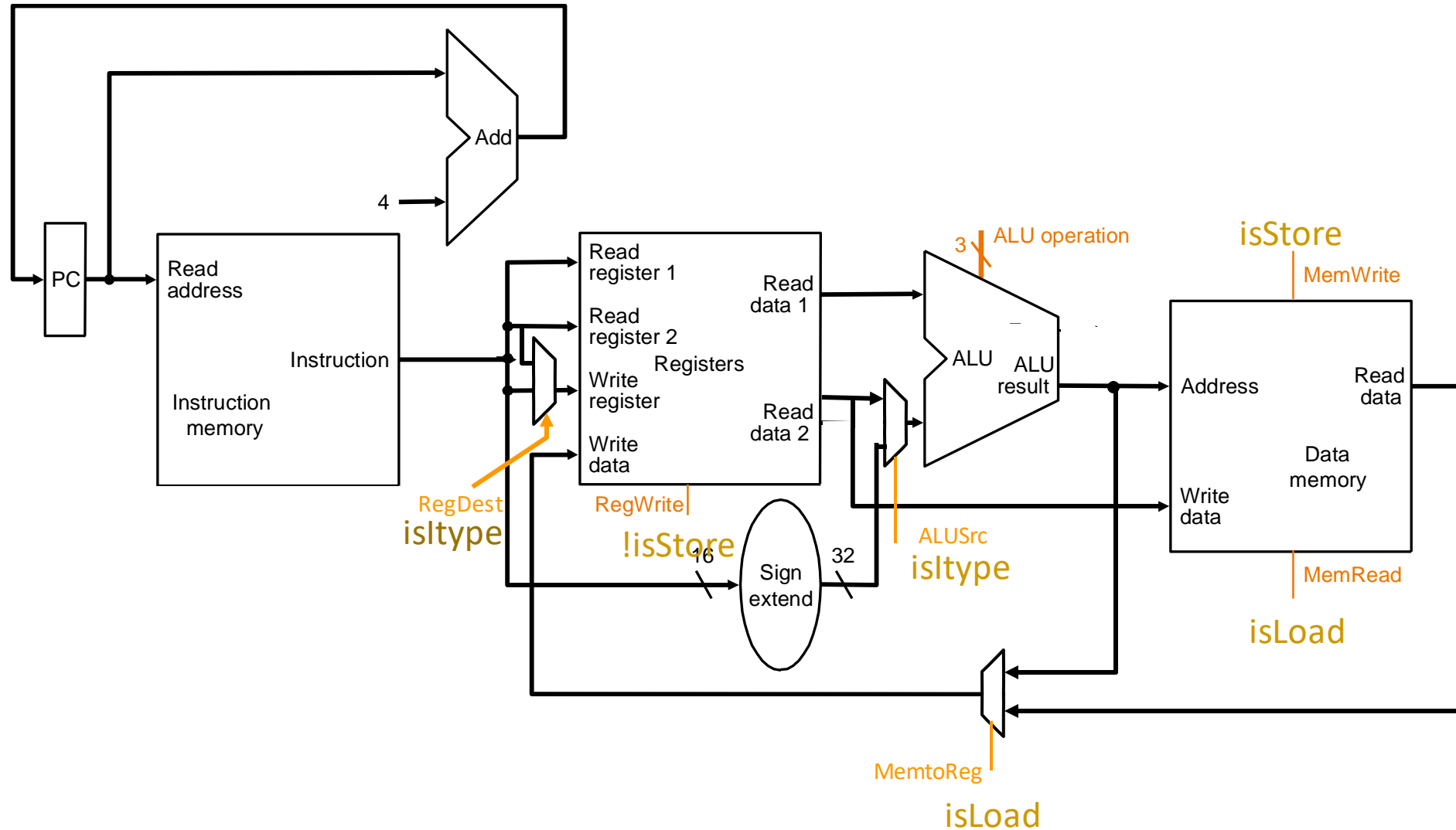
IF	ID	EX	MEM	WB
----	----	----	-----	----

Combinational
state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Insts.



Digital Design & Computer Arch.

Lecture 11: Microarchitecture Fundamentals

Prof. Onur Mutlu

ETH Zürich

Spring 2021

1 April 2021

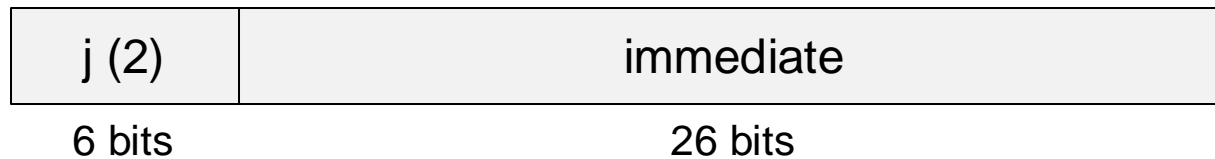
We Did Not Cover Later Slides.
They Will Be Covered in Later Lectures.

Single-Cycle Datapath for *Control Flow Instructions*

Jump Instruction

- Unconditional branch or jump

j target



J-Type

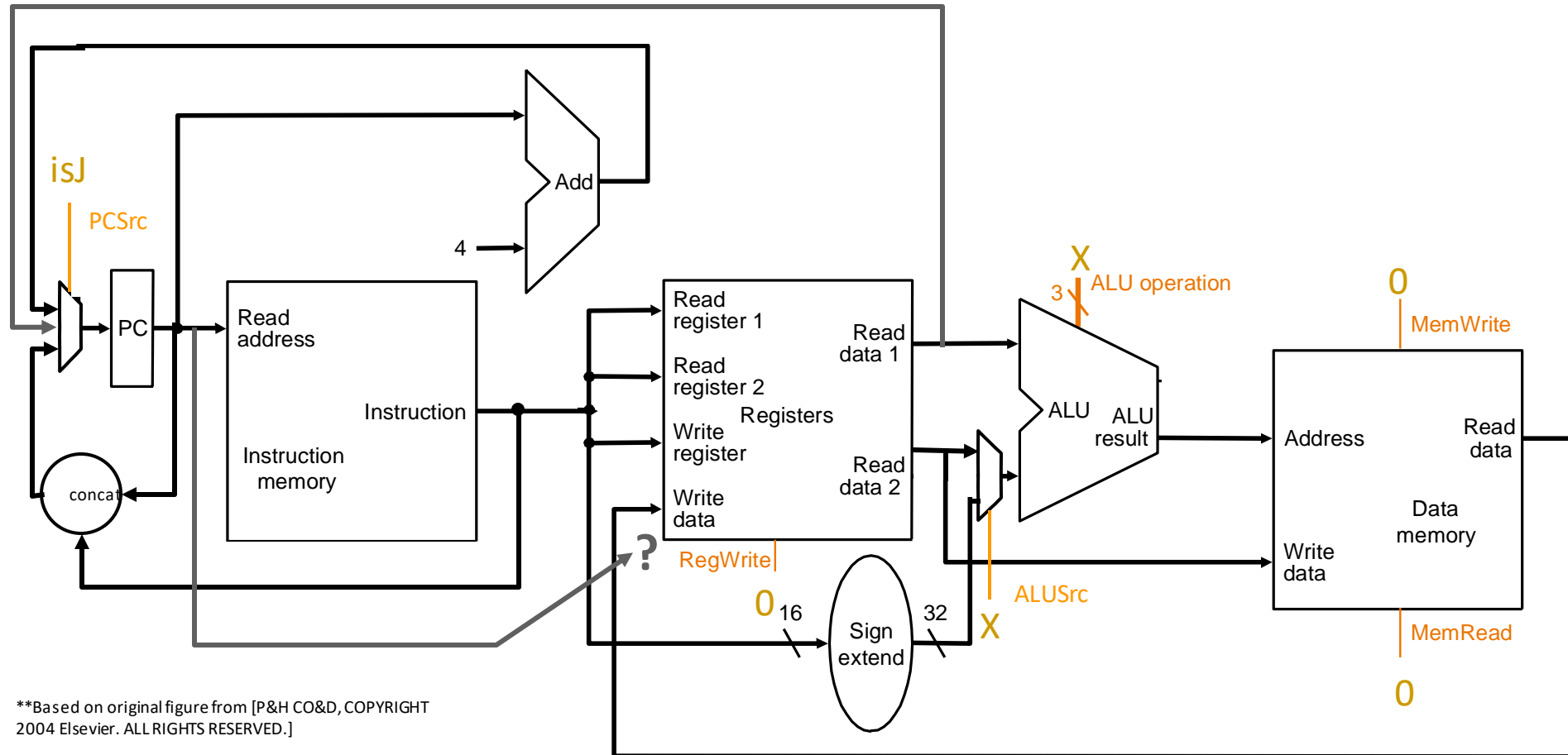
- 2 = opcode
- immediate (target) = target address

- Semantics

if $\text{MEM}[\text{PC}] == j \text{ immediate}_{26}$
 $\text{target} = \{ \text{PC} + [\text{31:28}], \text{immediate}_{26}, 2' \text{ b00} \}$
 $\text{PC} \leftarrow \text{target}$

[†] This is the incremented PC

Unconditional Jump Datapath



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

What about JR, JAL, JALR?

Other Jumps in MIPS

- ❑ jal: jump and link (function calls)

- Semantics

- if $\text{MEM}[\text{PC}] == \text{jal immediate}_{26}$

- $\$ra \leftarrow \text{PC} + 4$

- $\text{target} = \{ \text{PC}^\dagger[31:28], \text{immediate}_{26}, 2'b00 \}$

- $\text{PC} \leftarrow \text{target}$

- ❑ jr: jump register

- Semantics

- if $\text{MEM}[\text{PC}] == \text{jr rs}$

- $\text{PC} \leftarrow \text{GPR}(\text{rs})$

- ❑ jalr: jump and link register

- Semantics

- if $\text{MEM}[\text{PC}] == \text{jalr rs}$

- $\$ra \leftarrow \text{PC} + 4$

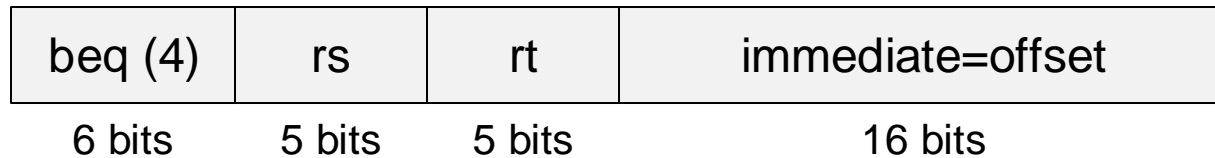
- $\text{PC} \leftarrow \text{GPR}(\text{rs})$

[†] This is the incremented PC

Conditional Branch Instructions

- beq (Branch if Equal)

```
beq  $s0, $s1, offset  # $s0=rs, $s1=rt
```



I-Type

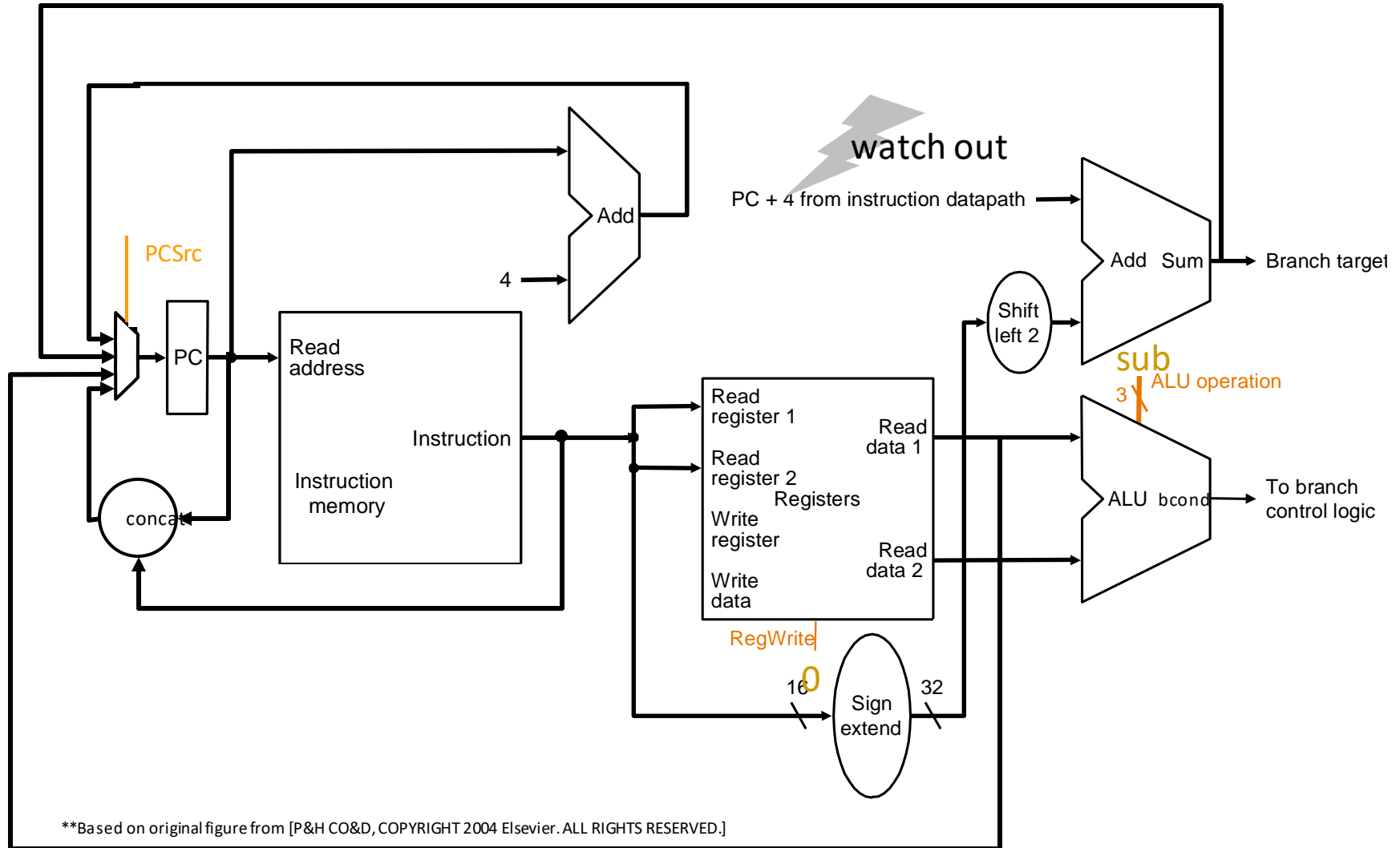
- Semantics (assuming no branch delay slot)

if $\text{MEM}[\text{PC}] == \text{beq } rs \text{ } rt \text{ } \text{immediate}_{16}$
 $\text{target} = \text{PC}^{\dagger} + \text{sign-extend}(\text{immediate}) \times 4$
 if $\text{GPR}[rs] == \text{GPR}[rt]$ then $\text{PC} \leftarrow \text{target}$
 else $\text{PC} \leftarrow \text{PC} + 4$

- Variations: beq, bne, blez, bgtz

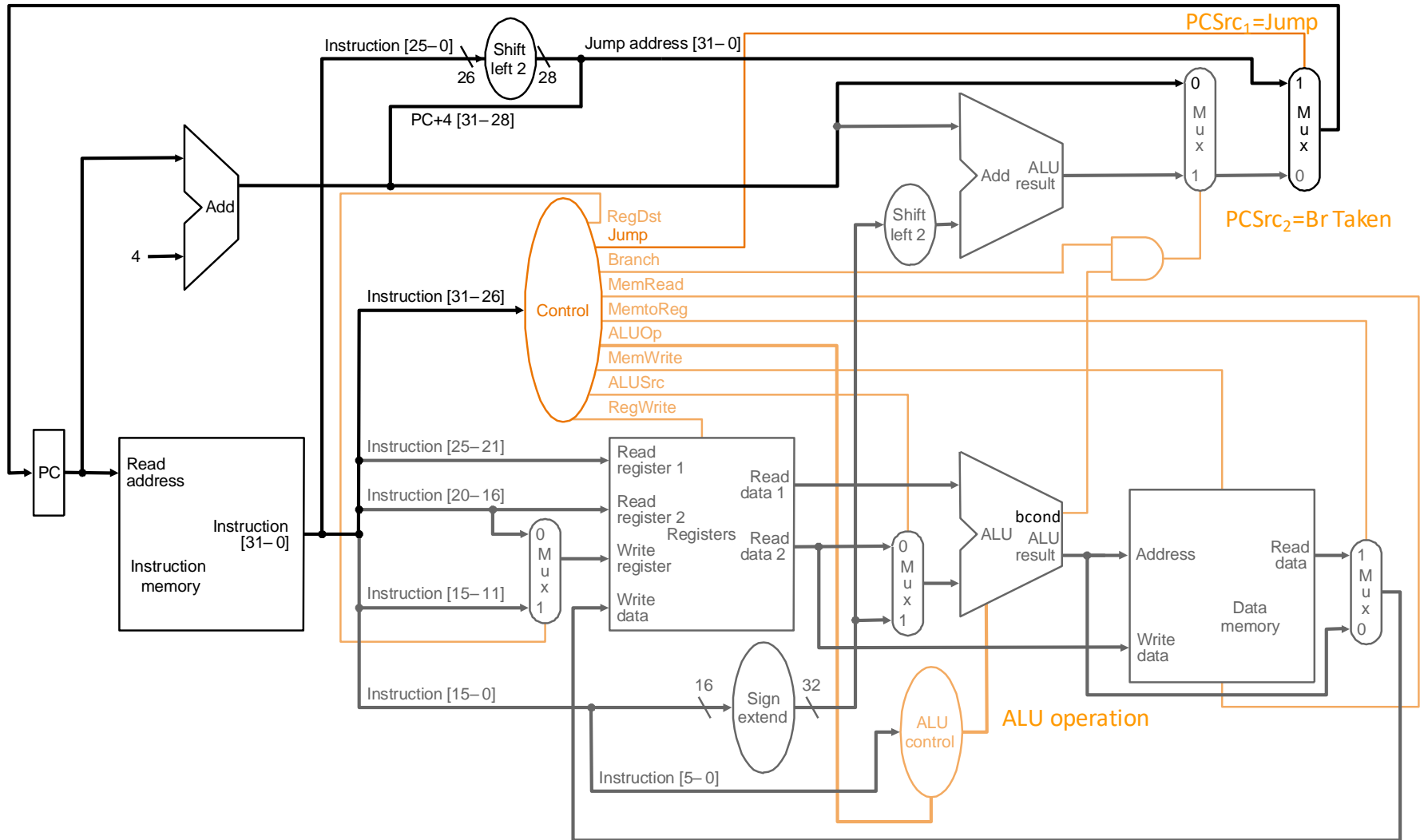
[†] This is the incremented PC

Conditional Branch Datapath (for you to finish)



How to uphold the delayed branch semantics?

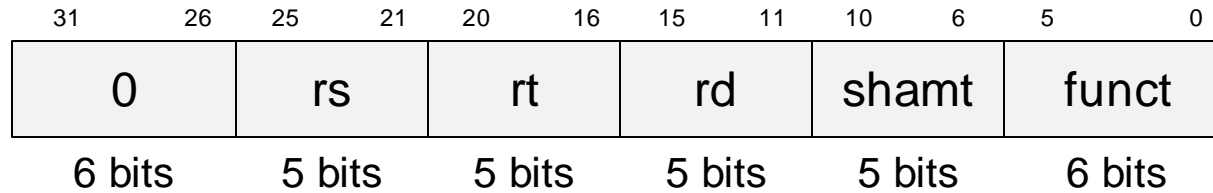
Putting It All Together



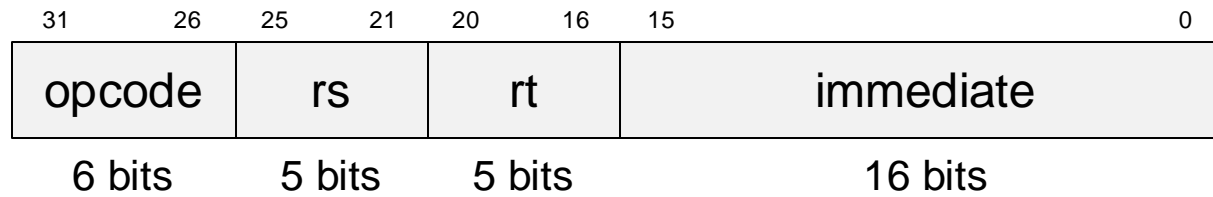
Single-Cycle Control Logic

Single-Cycle Hardwired Control

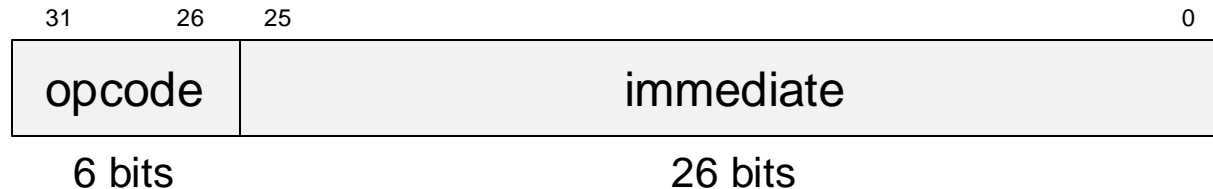
- As combinational function of **Inst=MEM[PC]**



R-Type



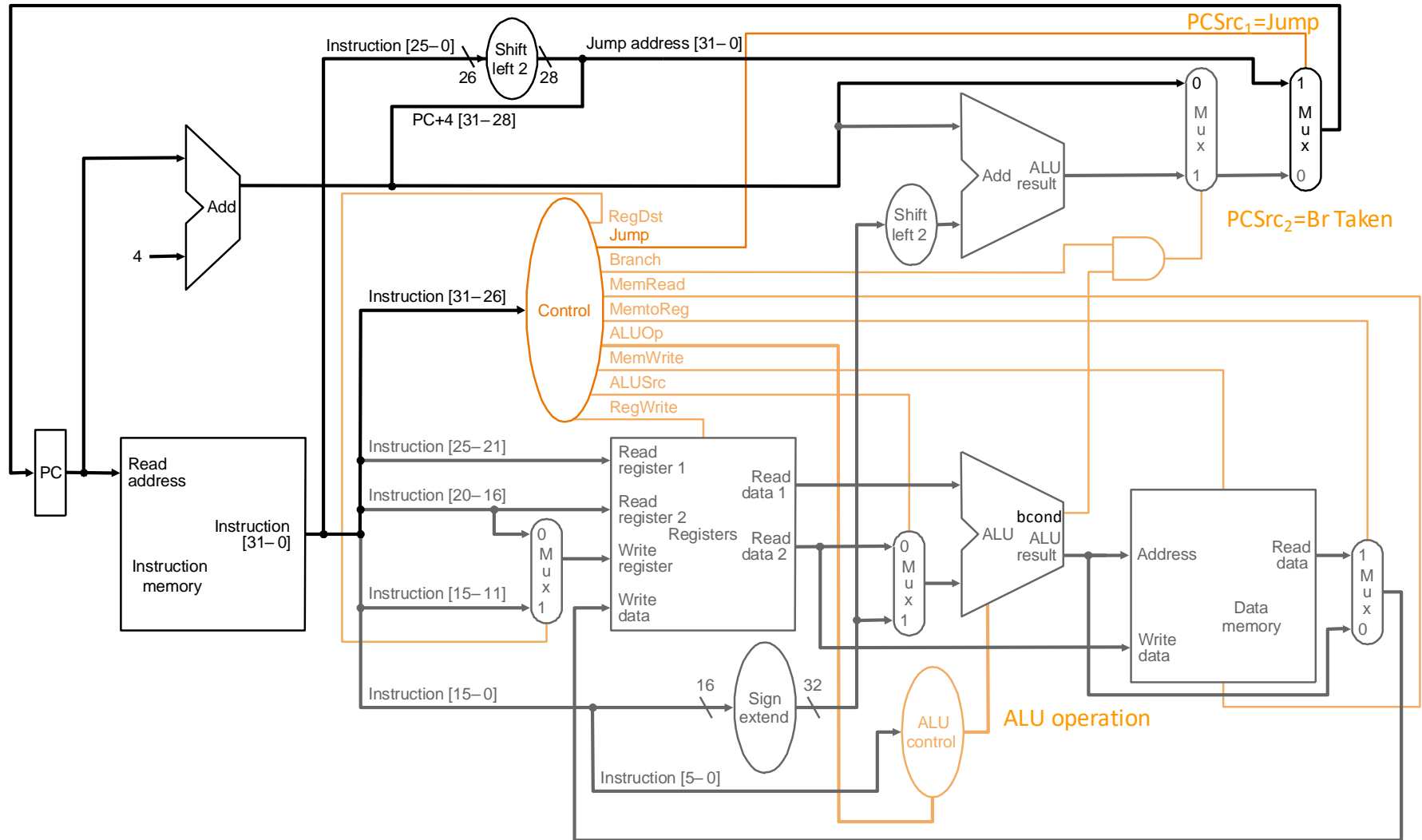
I-Type



J-Type

- Consider
 - ❑ All R-type and I-type **ALU** instructions
 - ❑ **lw** and **sw**
 - ❑ **beq, bne, blez, bgtz**
 - ❑ **j, jr, jal, jalr**

Generate Control Signals (in Orange Color)



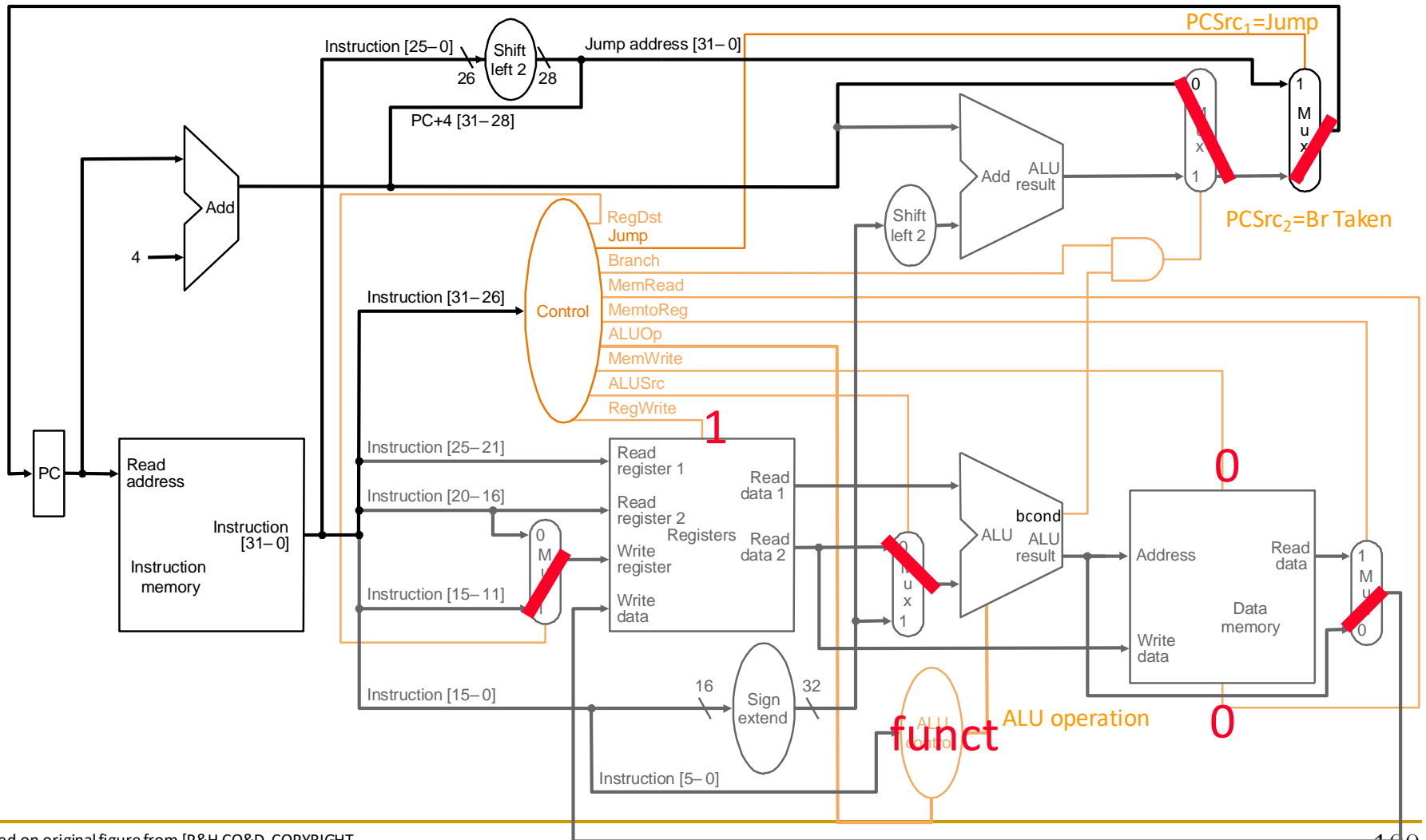
Single-Bit Control Signals (I)

	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	2 nd ALU input from 2 nd GPR read port	2 nd 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 write port	opcode ==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode !=SW) && (opcode !=Bxx) && (opcode !=J) && (opcode !=JR))

Single-Bit Control Signals (II)

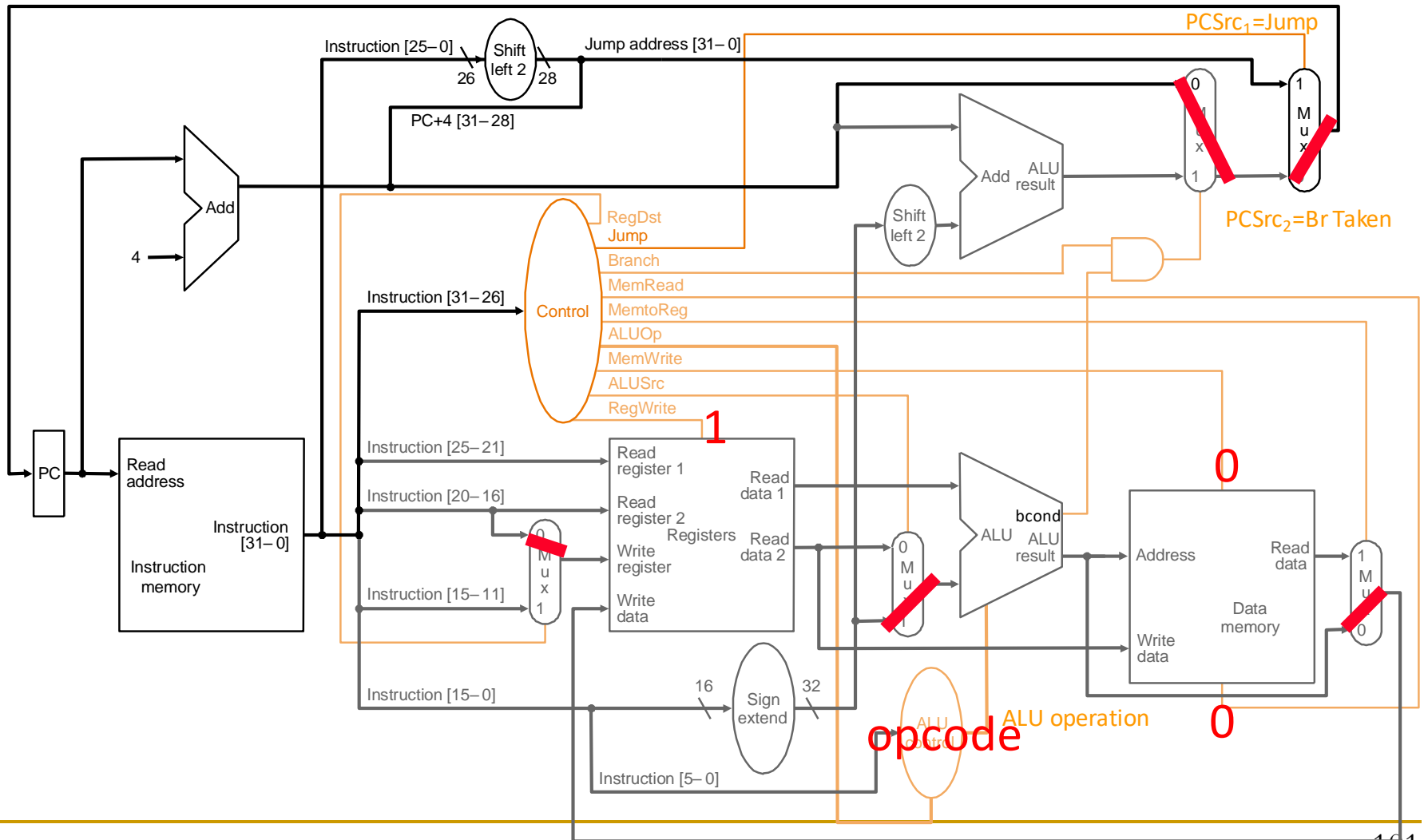
	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	$\text{opcode} == \text{LW}$
MemWrite	Memory write disabled	Memory write enabled	$\text{opcode} == \text{SW}$
PCSrc ₁	According to PCSrc ₂	next PC is based on 26-bit immediate jump target	$(\text{opcode} == \text{J}) \mid \mid (\text{opcode} == \text{JAL})$
PCSrc ₂	next PC = PC + 4	next PC is based on 16-bit immediate branch target	$(\text{opcode} == \text{Bxx}) \ \&\& \text{“bcond is satisfied”}$

R-Type ALU

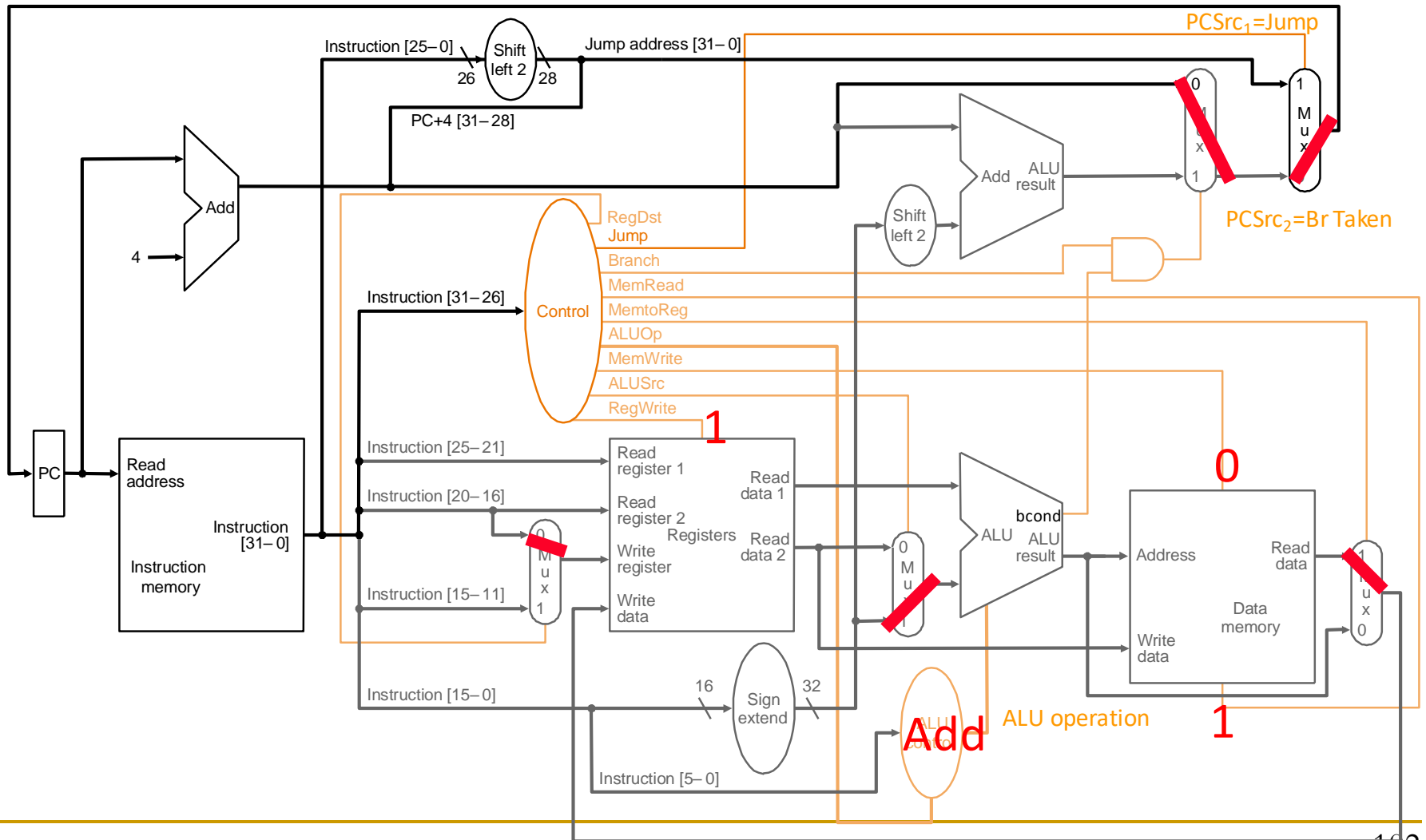


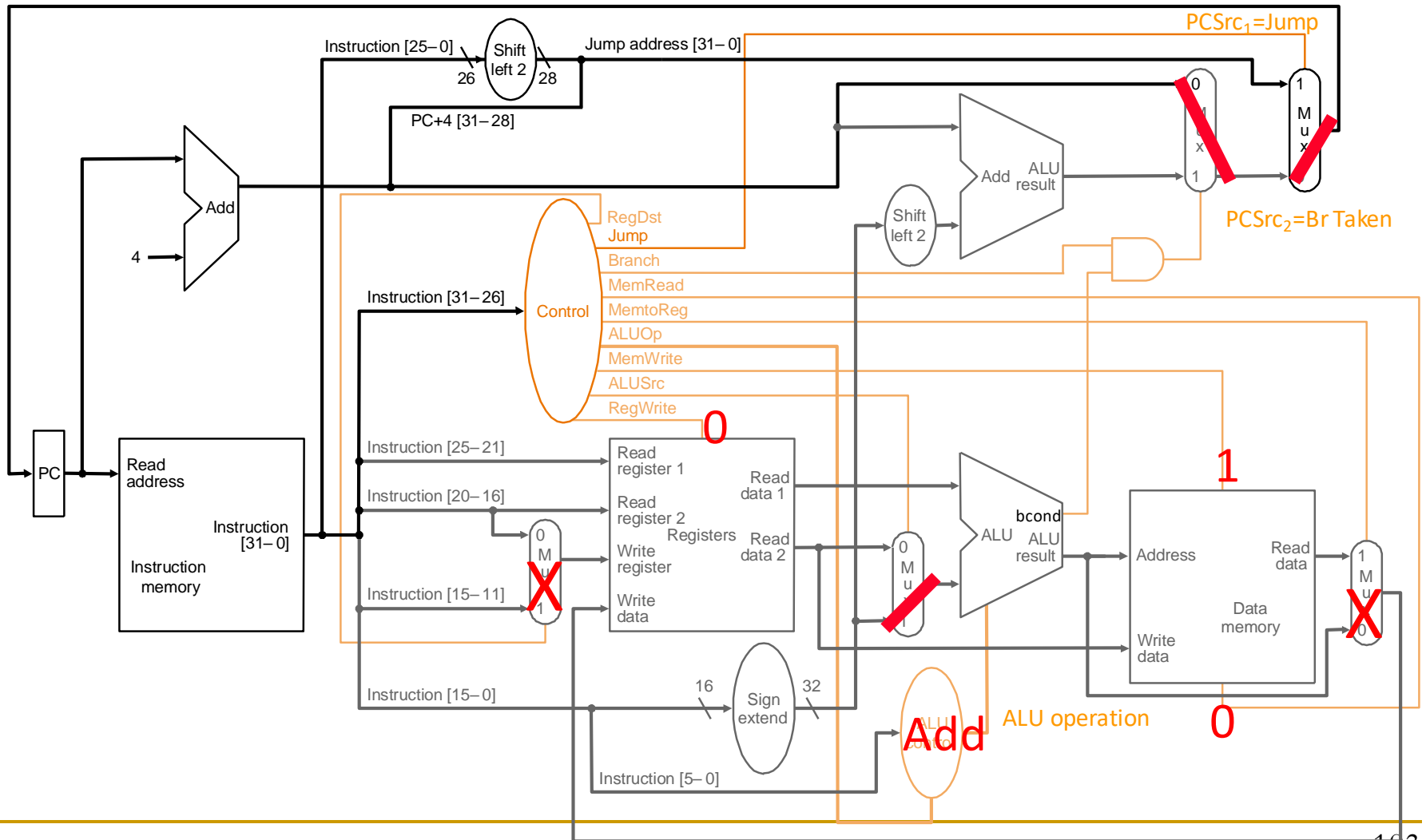
**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

I-Type ALU



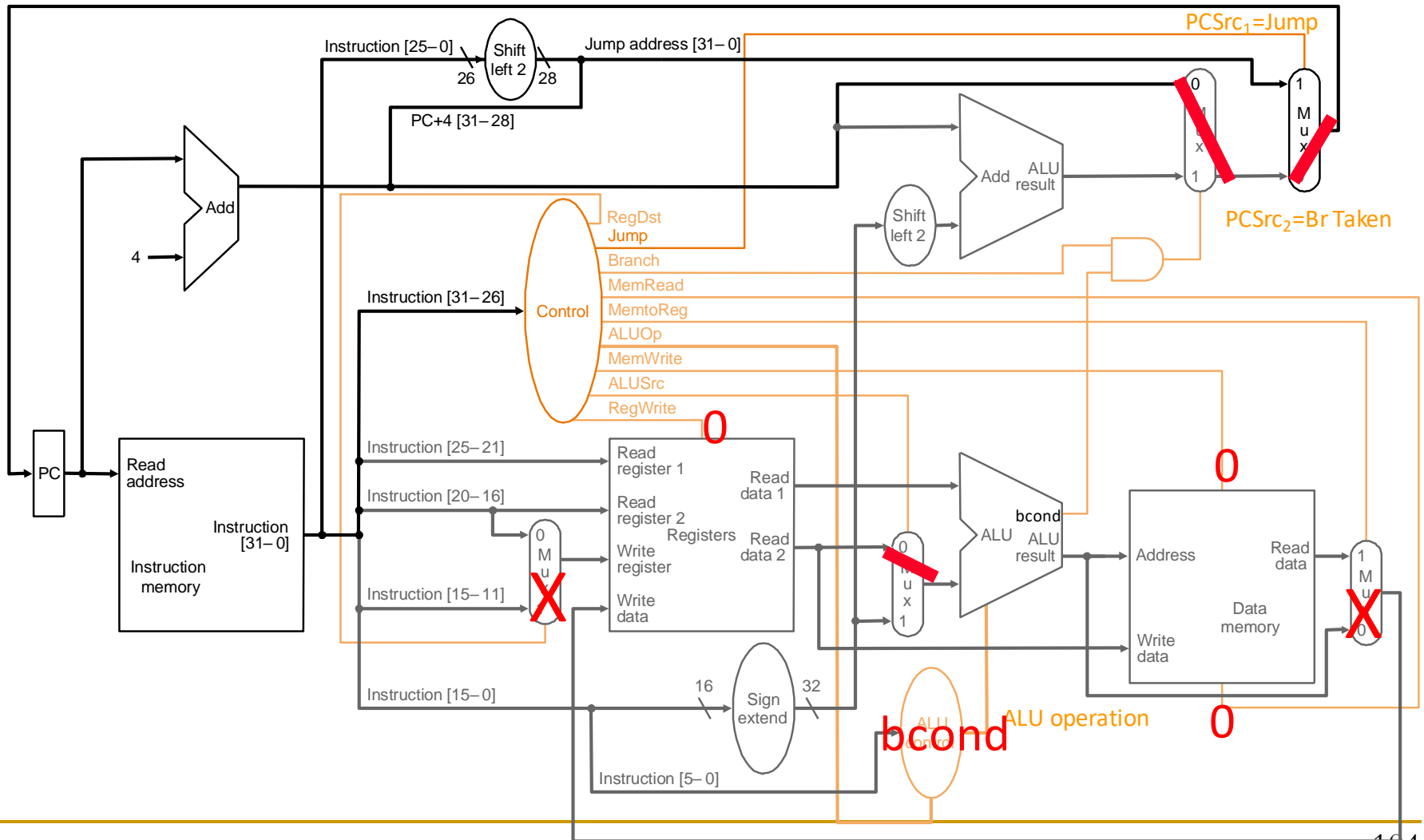
**Based on original figure from [P&H CO&D, COPYRIGHT 2004
Elsevier. ALL RIGHTS RESERVED.]





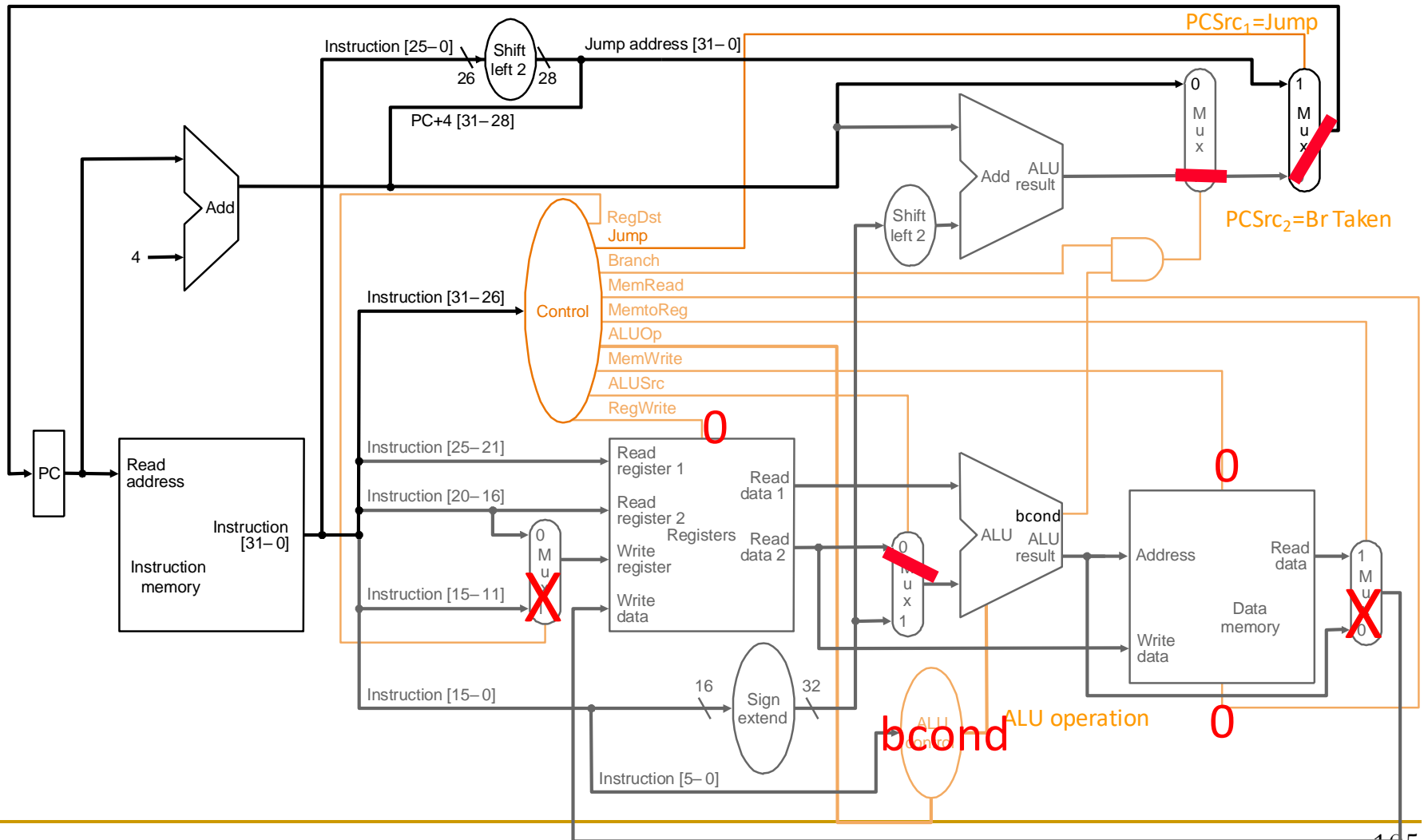
Branch (Not Taken)

Some control signals are dependent on the processing of data



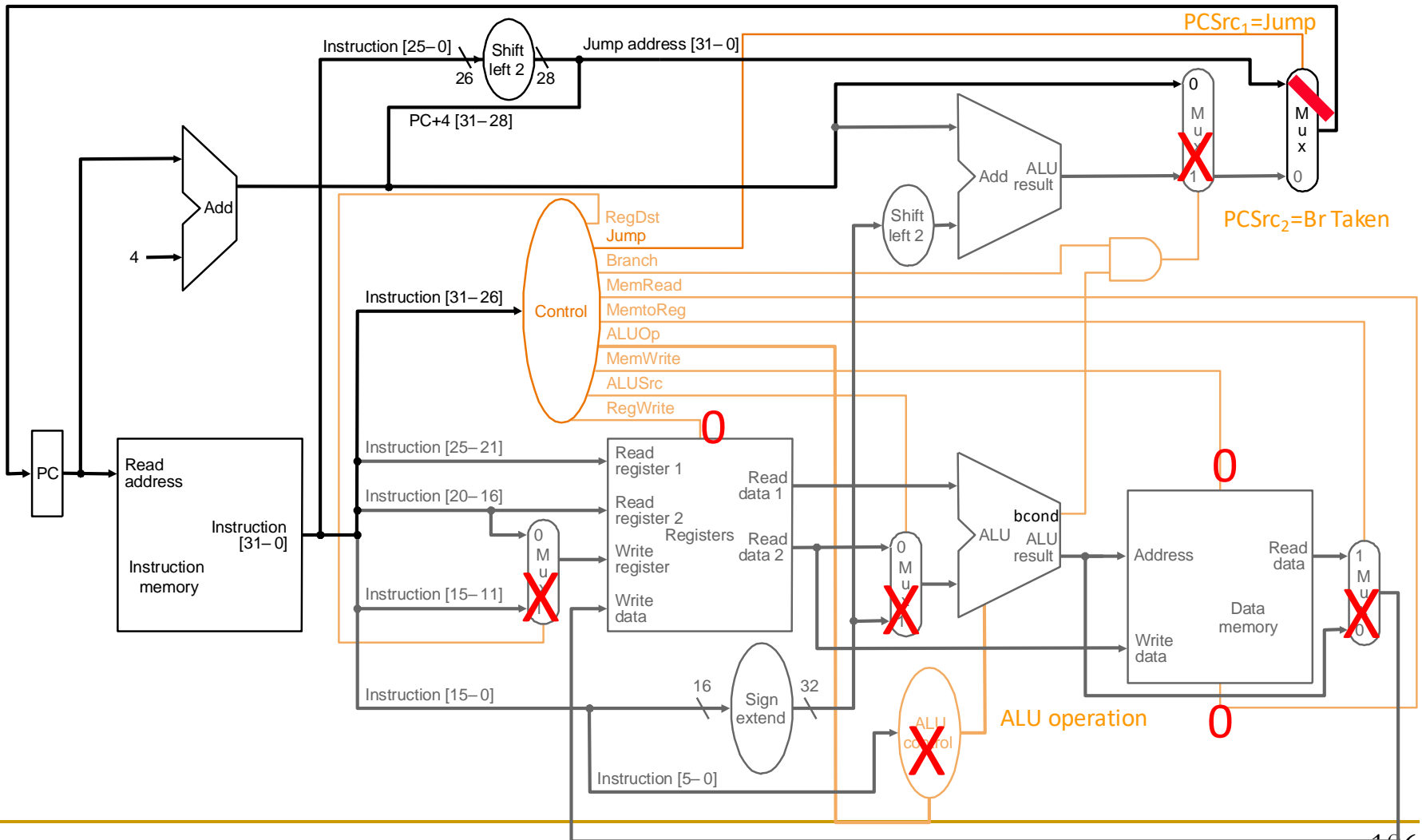
Branch (Taken)

Some control signals are dependent on the processing of data



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

Jump

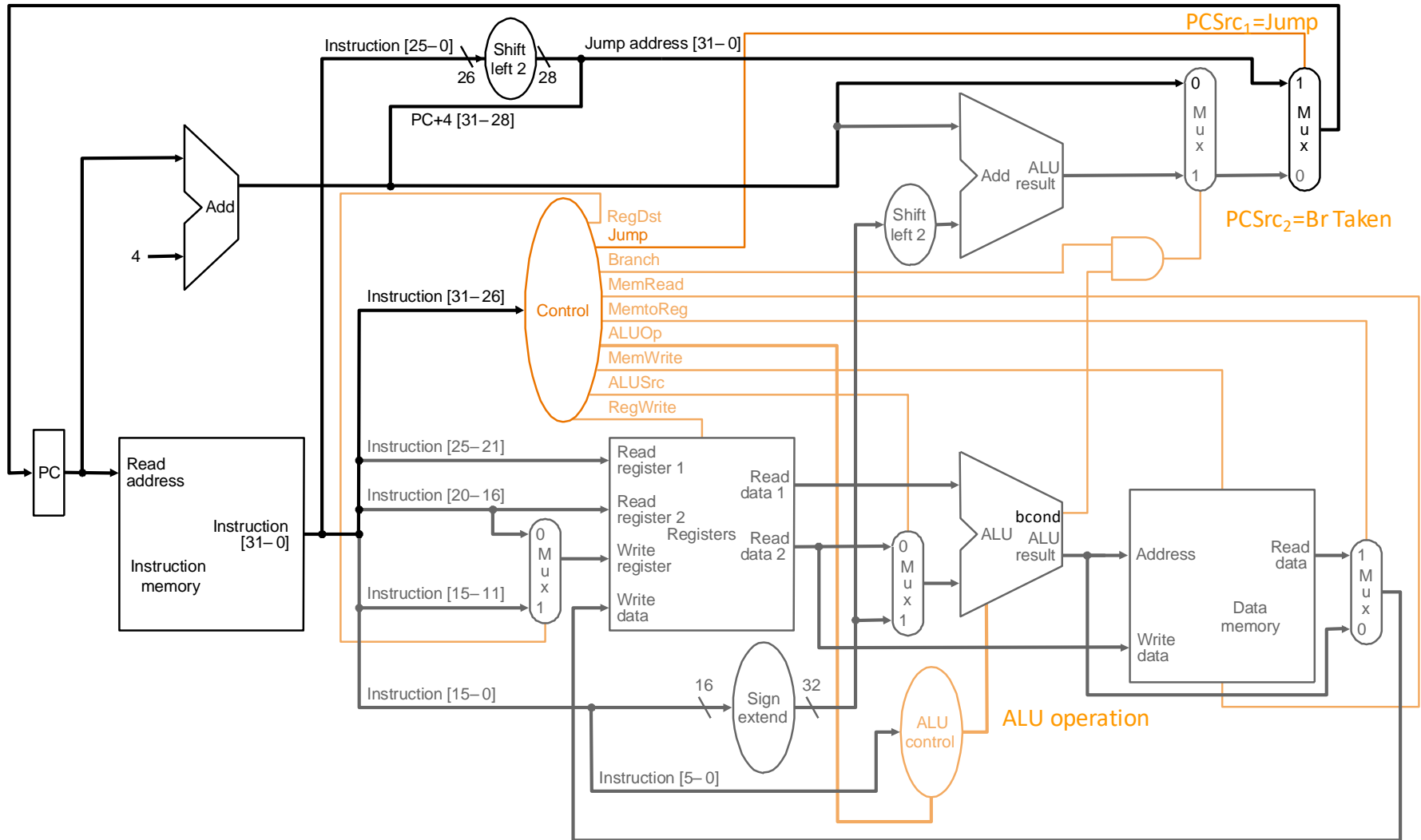


**Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

What is in That Control Box?

- Combinational Logic → **Hardwired Control**
 - Idea: Most control signals generated combinatorially based on bits in instruction encoding
- Sequential Logic → **Sequential Control**
 - Idea: A memory structure contains the control signals associated with an instruction
 - Called **Control Store**
- **Both types of control structure can be used in single-cycle processors**
 - Choice depends on latency of each structure + how much on the critical path control signal generation is, etc.

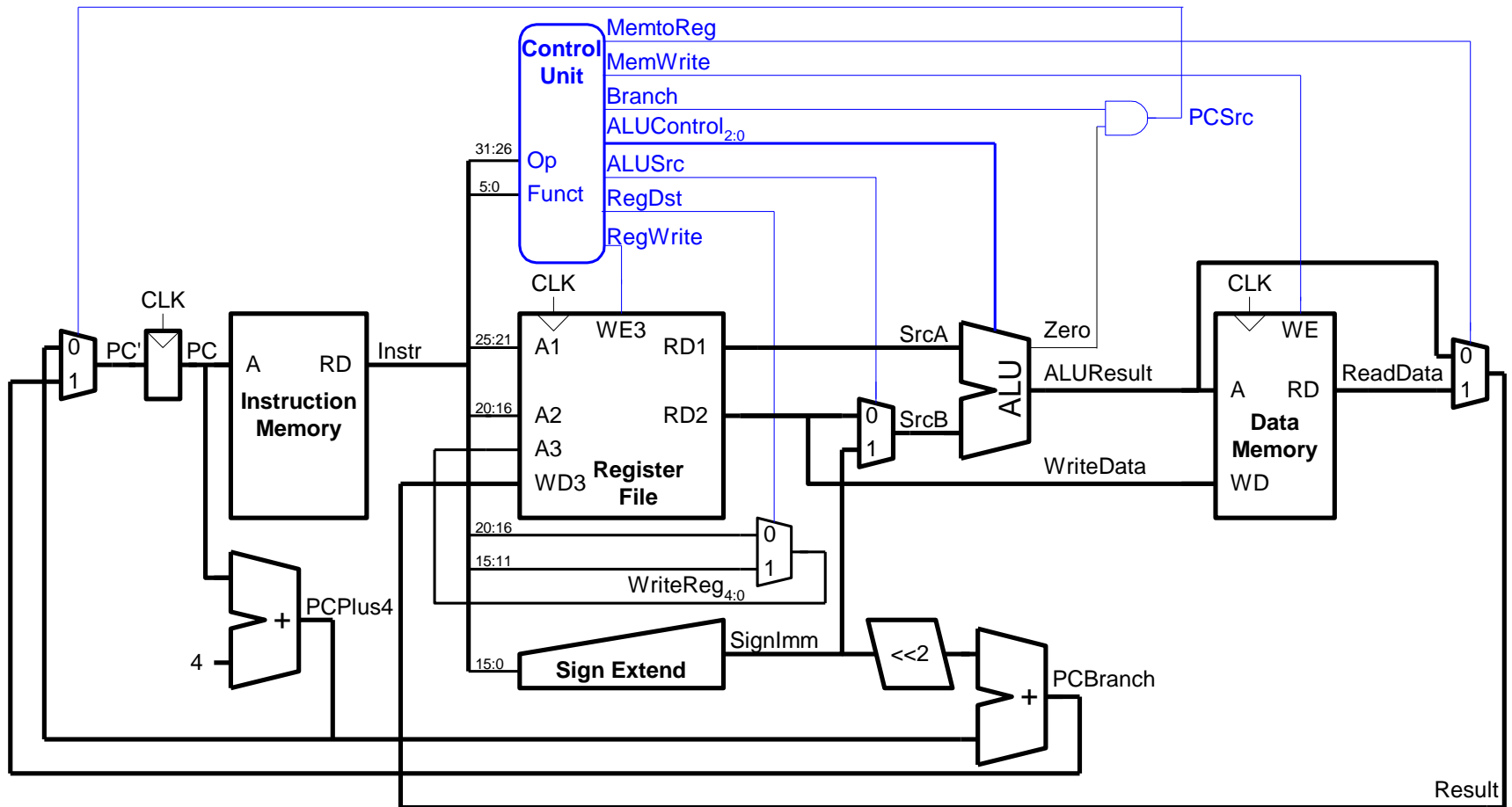
Review: Complete Single-Cycle Processor



Another Single-Cycle MIPS Processor (from H&H)

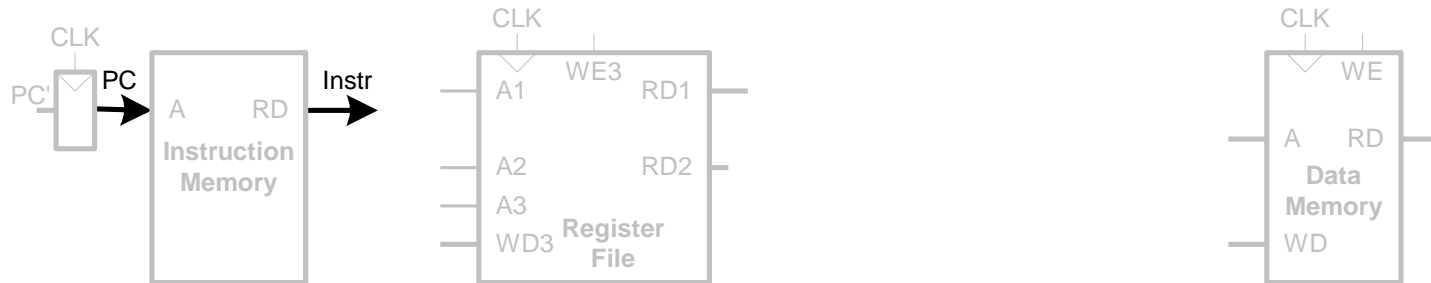
See backup slides to reinforce the concepts we have covered.
They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6

Another Complete Single-Cycle Processor



Example: Single-Cycle Datapath: lw fetch

■ **STEP 1:** Fetch instruction



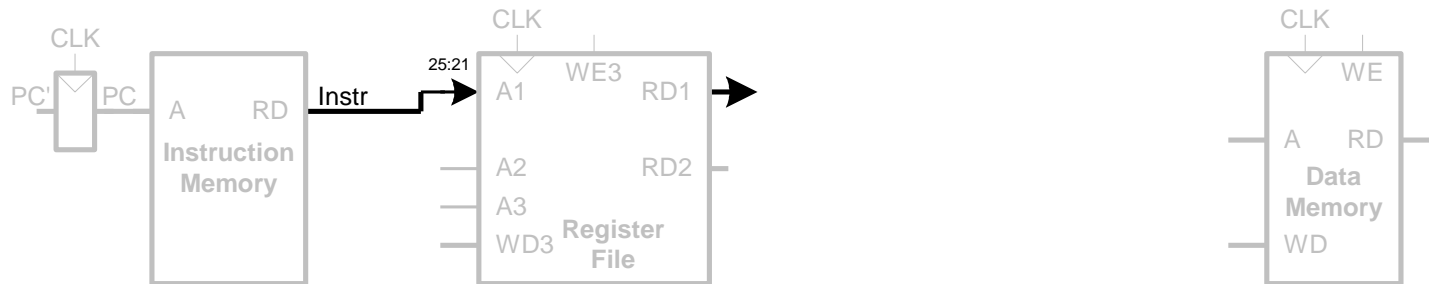
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type



Single-Cycle Datapath: lw register read

- **STEP 2:** Read source operands from register file



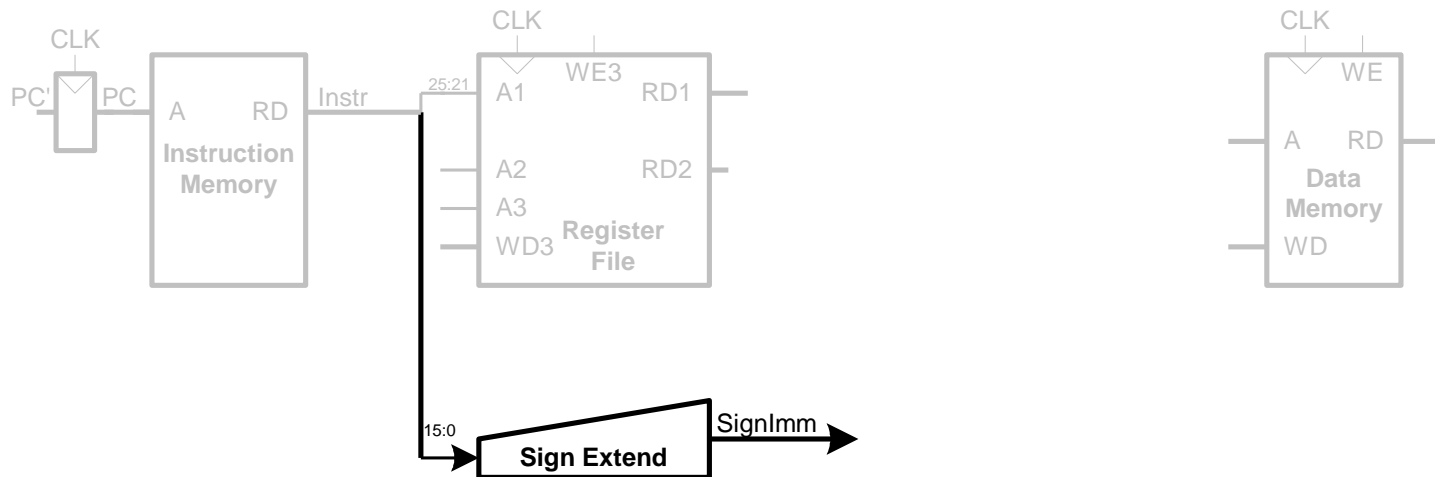
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw immediate

■ **STEP 3:** Sign-extend the immediate



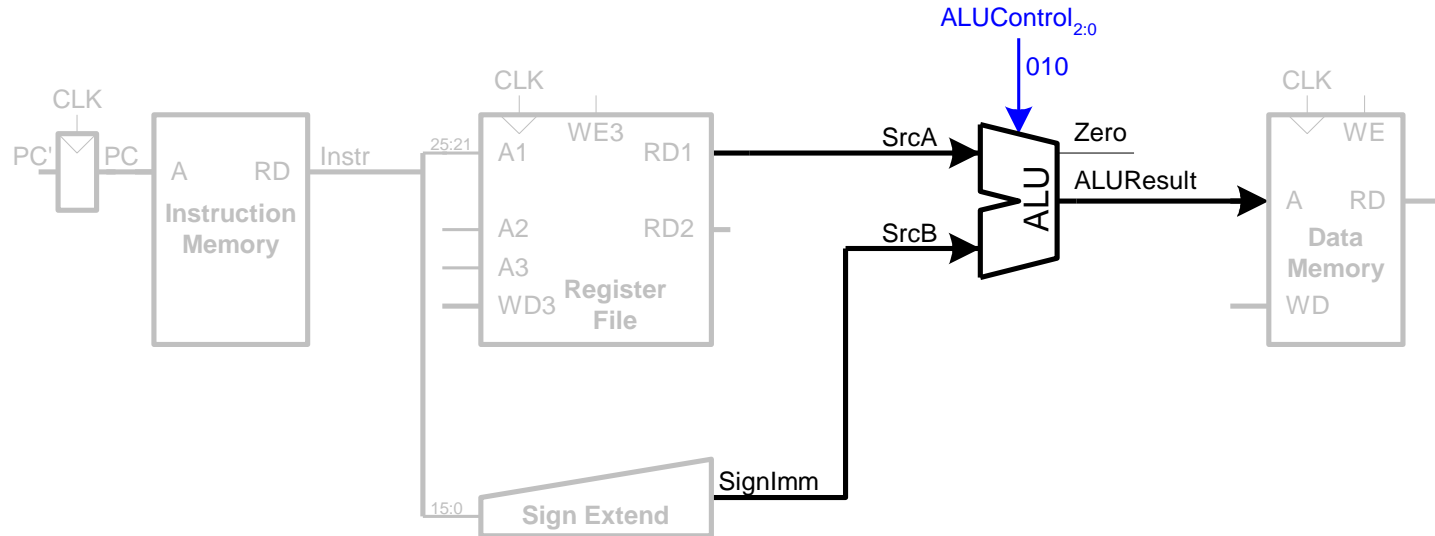
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw address

■ **STEP 4:** Compute the memory address



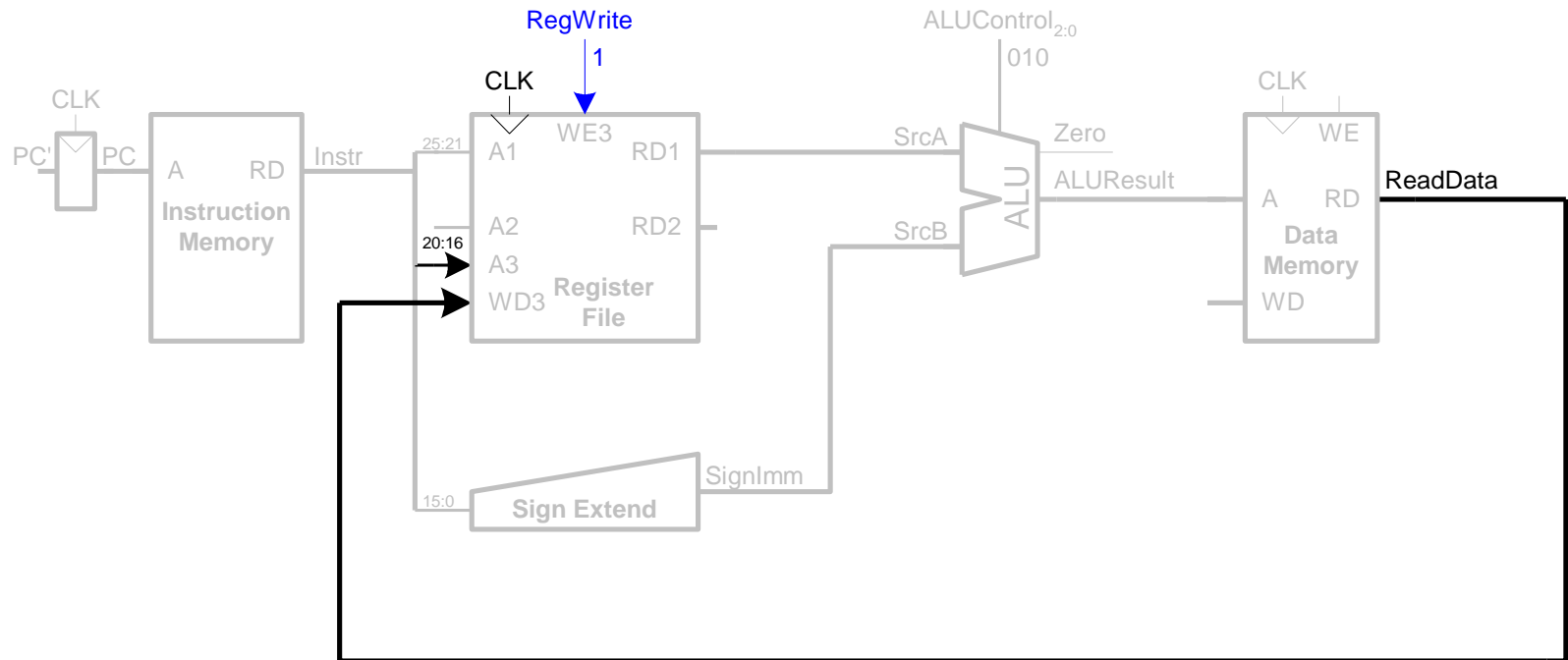
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw memory read

■ **STEP 5:** Read from memory and write back to register file



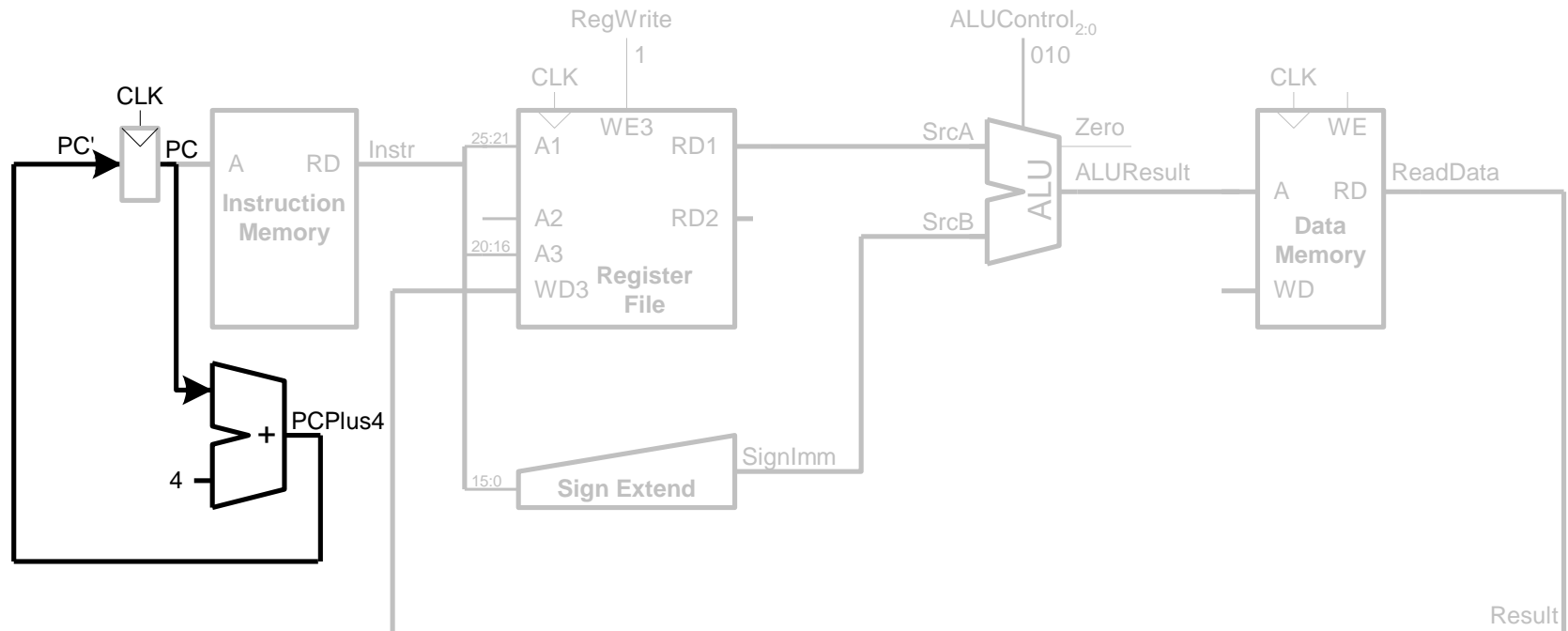
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw PC increment

■ STEP 6: Determine address of next instruction



`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

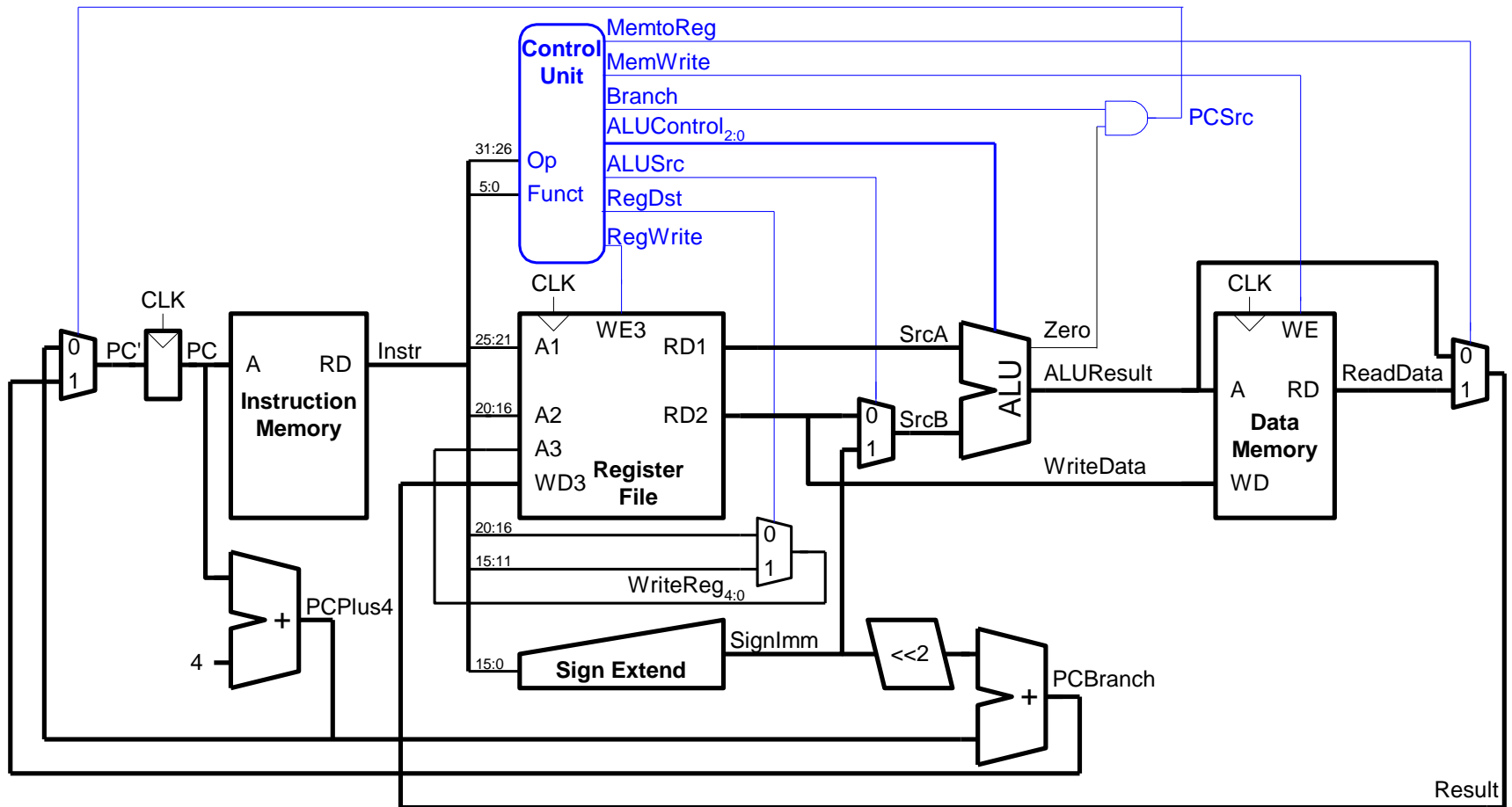
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Similarly, We Need to Design the Control Unit

- **Control signals** are generated by the decoder in control unit

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

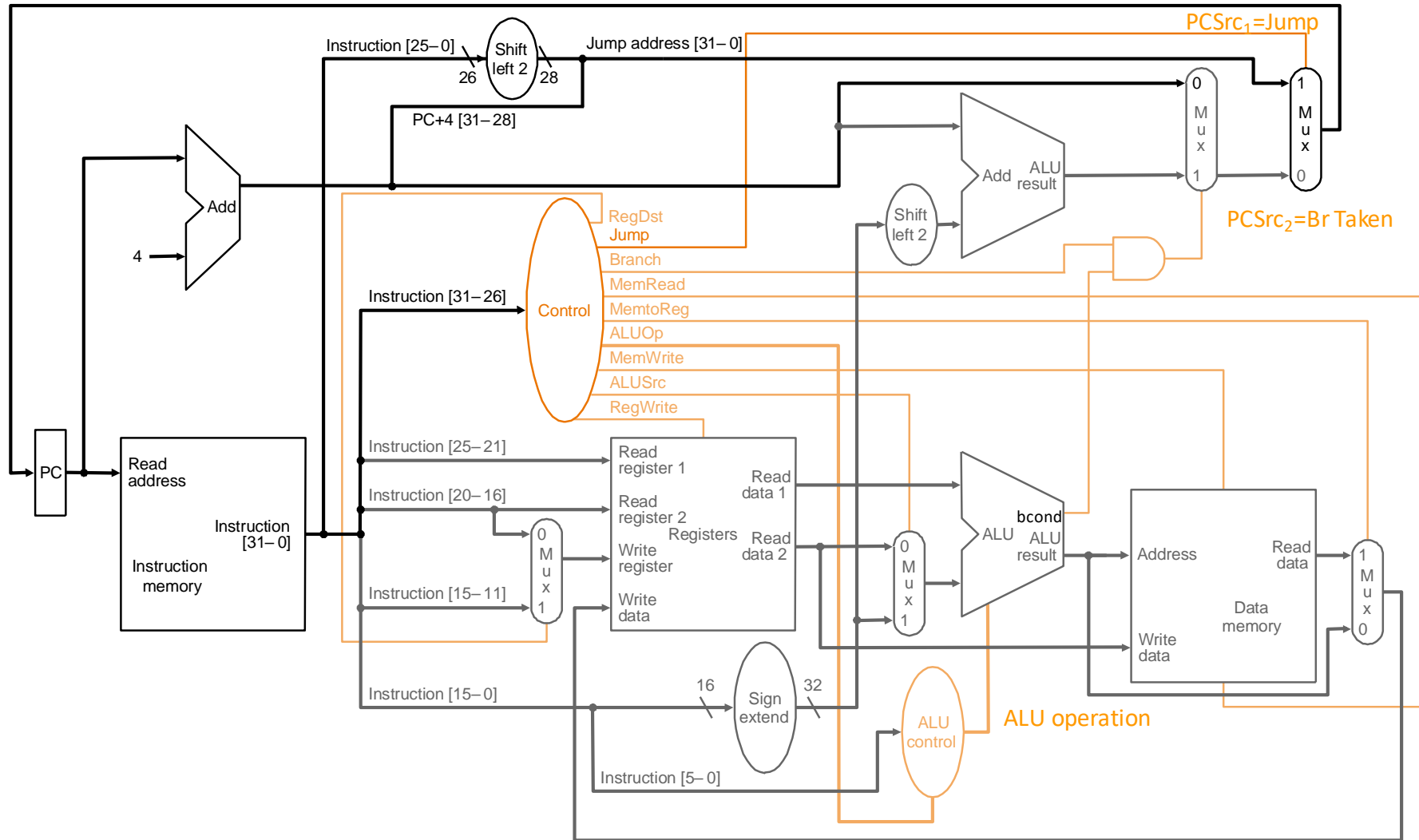
Another Complete Single-Cycle Processor (H&H)



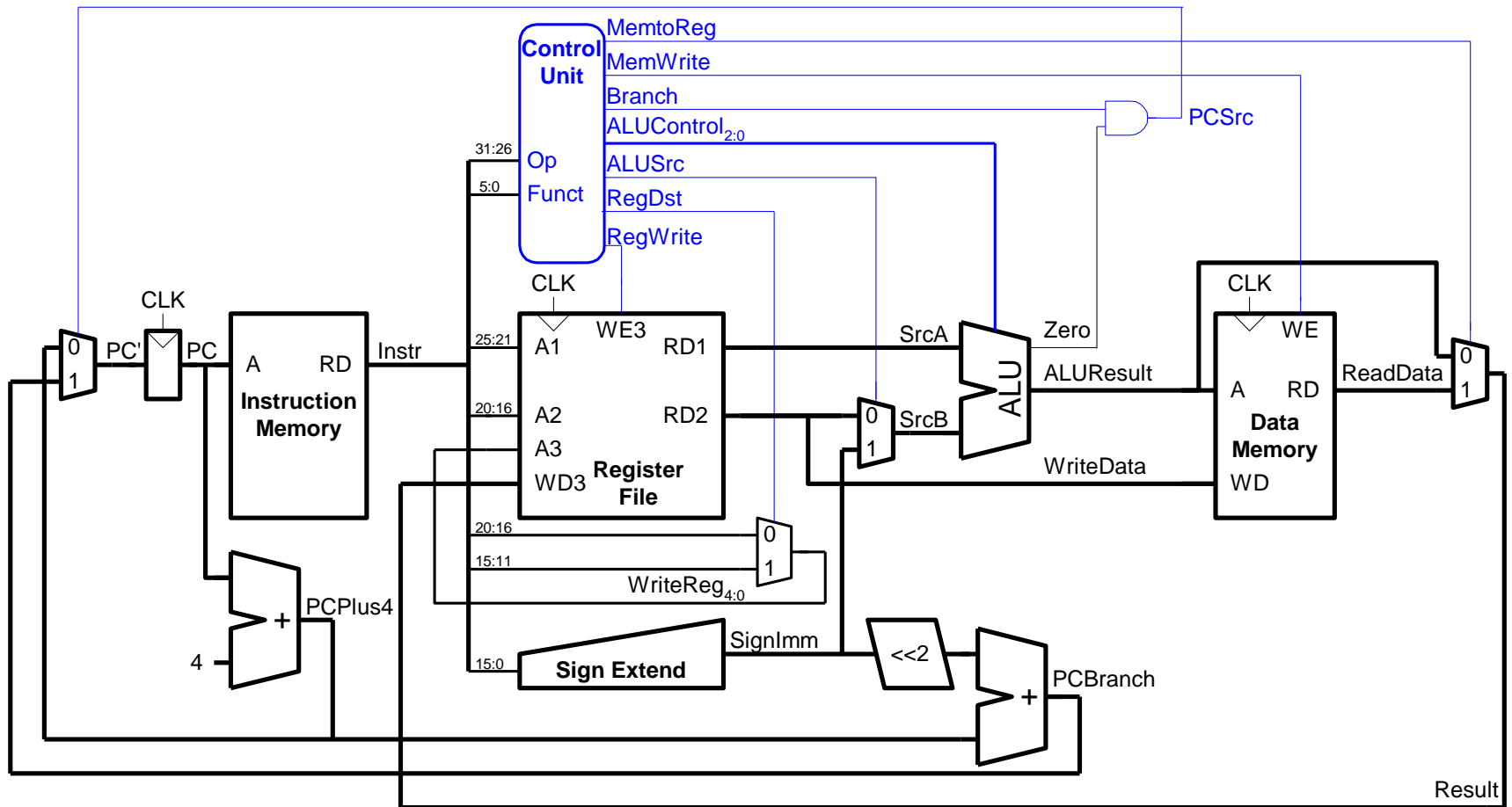
Your Reading Assignment

- Please read the Lecture Slides and the Backup Slides
- Please do your readings from the H&H Book
 - H&H, Chapter 7.1-7.3, 7.6

Single-Cycle Uarch I (We Developed in Lectures)



Single-Cycle Uarch II (In Your Readings)



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?

Performance Analysis Basics

Processor Performance

- **How fast is my program?**
 - Every program consists of a series of instructions
 - Each instruction needs to be executed

Processor Performance

■ How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed

■ So how fast are my instructions ?

- Instructions are realized on the hardware
- They can take one or more clock cycles to complete
- *Cycles per Instruction = CPI*

Processor Performance

■ How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed.

■ So how fast are my instructions ?

- Instructions are realized on the hardware
- They can take one or more clock cycles to complete
- *Cycles per Instruction = CPI*

■ How much time is one clock cycle?

- The critical path determines how much time one cycle requires = *clock period*.
- $1/\text{clock period} = \text{clock frequency}$ = how many cycles can be done each second.

Processor Performance

■ Now as a general formula

- Our program consists of executing **N** instructions
- Our processor needs **CPI** cycles for each instruction
- The maximum clock speed of the processor is **f**,
and the clock period is therefore **$T=1/f$**

Processor Performance

■ Now as a general formula

- Our program consists of executing **N** instructions
- Our processor needs **CPI** cycles for each instruction
- The maximum clock speed of the processor is **f**, and the clock period is therefore **T=1/f**

■ Our program executes in

$$N \times CPI \times (1/f) =$$

$$N \times CPI \times T \text{ seconds}$$

Performance Analysis Basics

- Execution time of a single instruction
 - $\{\text{CPI}\} \times \{\text{clock cycle time}\}$
 - CPI: Number of cycles it takes to execute an instruction
- Execution time of an entire program
 - Sum over all instructions $[\{\text{CPI}\} \times \{\text{clock cycle time}\}]$
 - $\{\# \text{ of instructions}\} \times \{\text{Average CPI}\} \times \{\text{clock cycle time}\}$

Performance Analysis of Our Single-Cycle Design

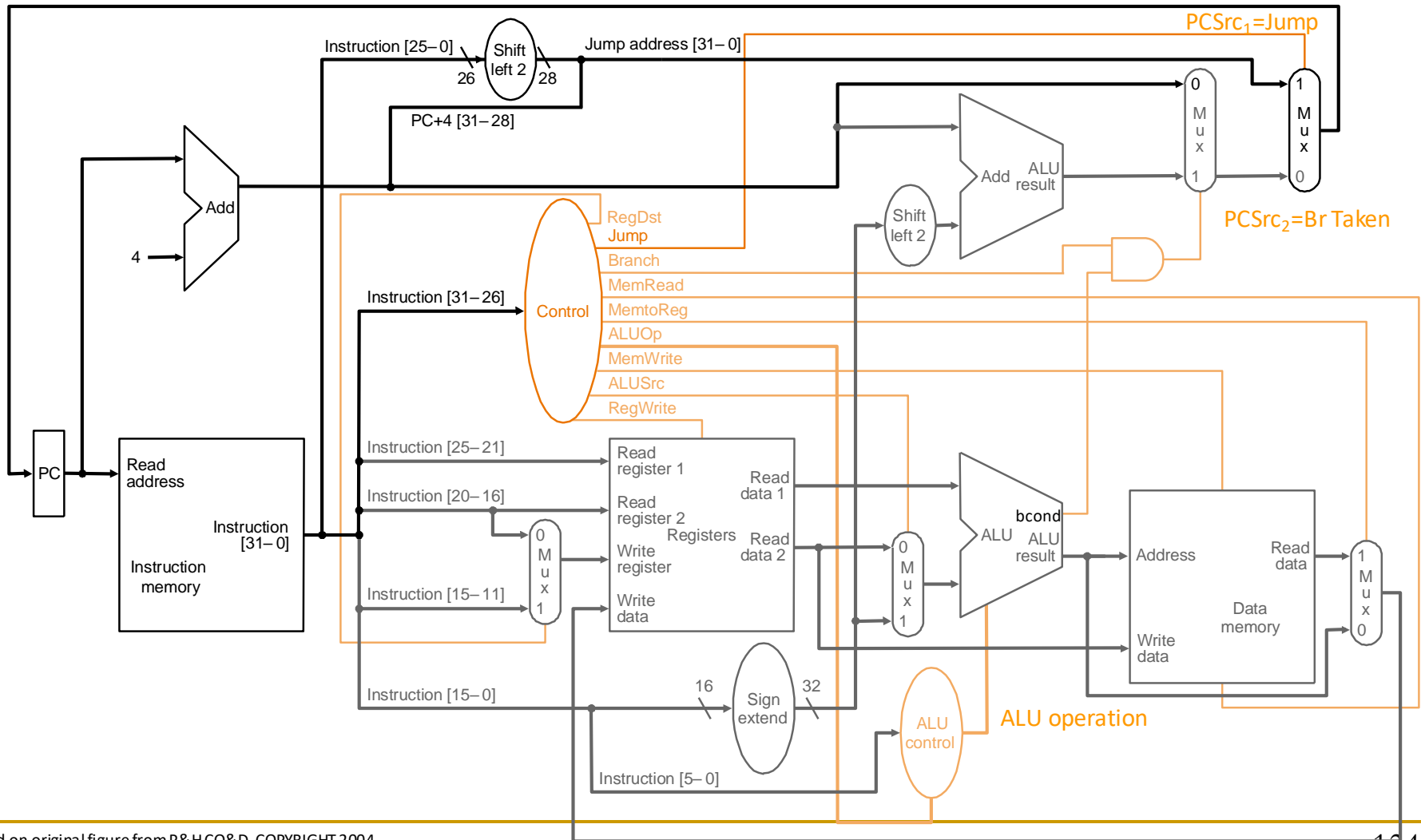
A Single-Cycle Microarchitecture: Analysis

- 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
 - 1. Instruction fetch (IF)
 - ❑ Decode
 - 2. Instruction decode and
 - ❑ Evaluate Address
 - register operand fetch (ID/RF)
 - ❑ Fetch Operands
 - 3. Execute/Evaluate memory address (EX/AG)
 - ❑ Execute
 - 4. Memory operand fetch (MEM)
 - ❑ Store Result
 - 5. Store/writeback result (WB)
- Do each of the above phases take the same time (latency) for all instructions?

Let's Find the Critical Path

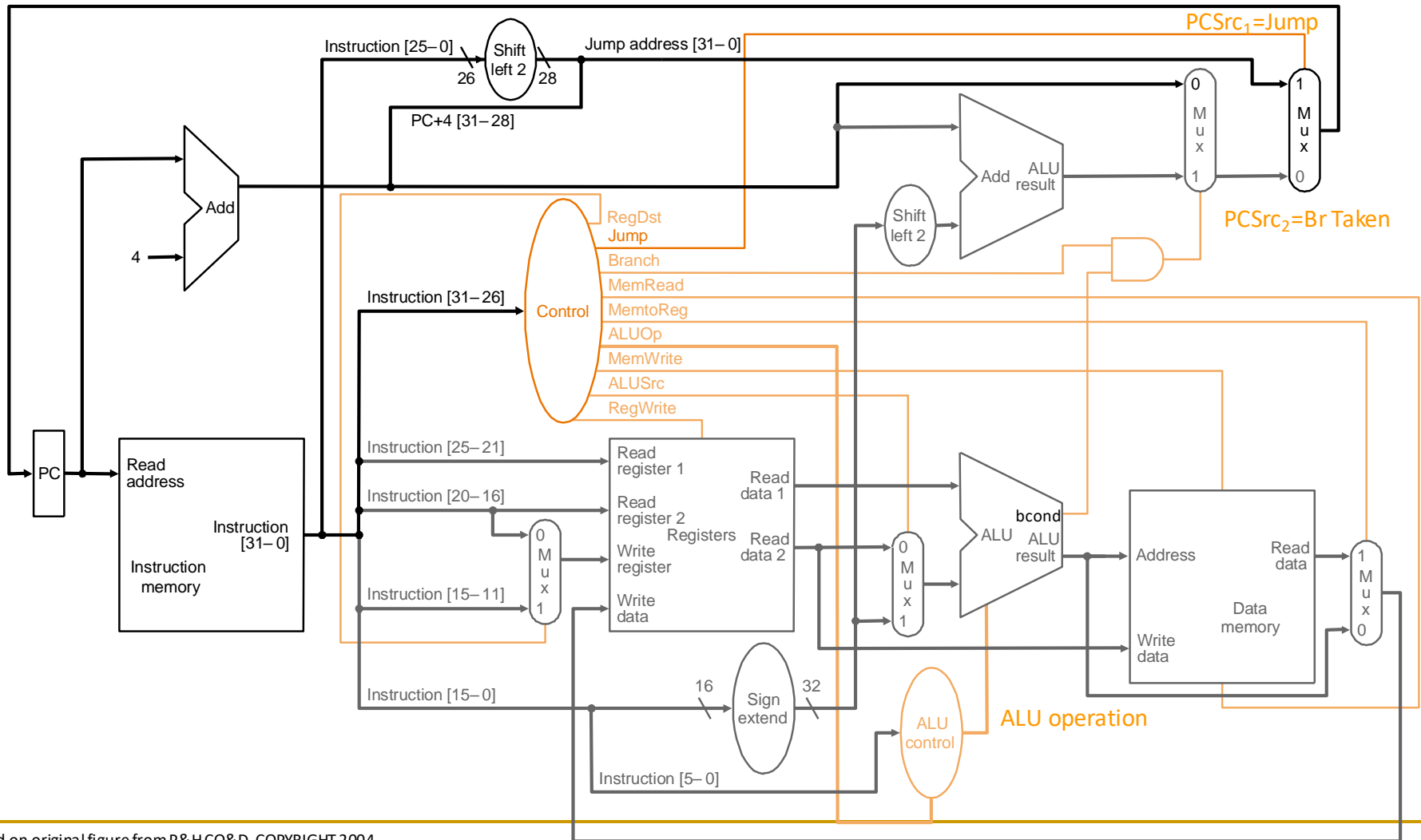


Example Single-Cycle Datapath Analysis

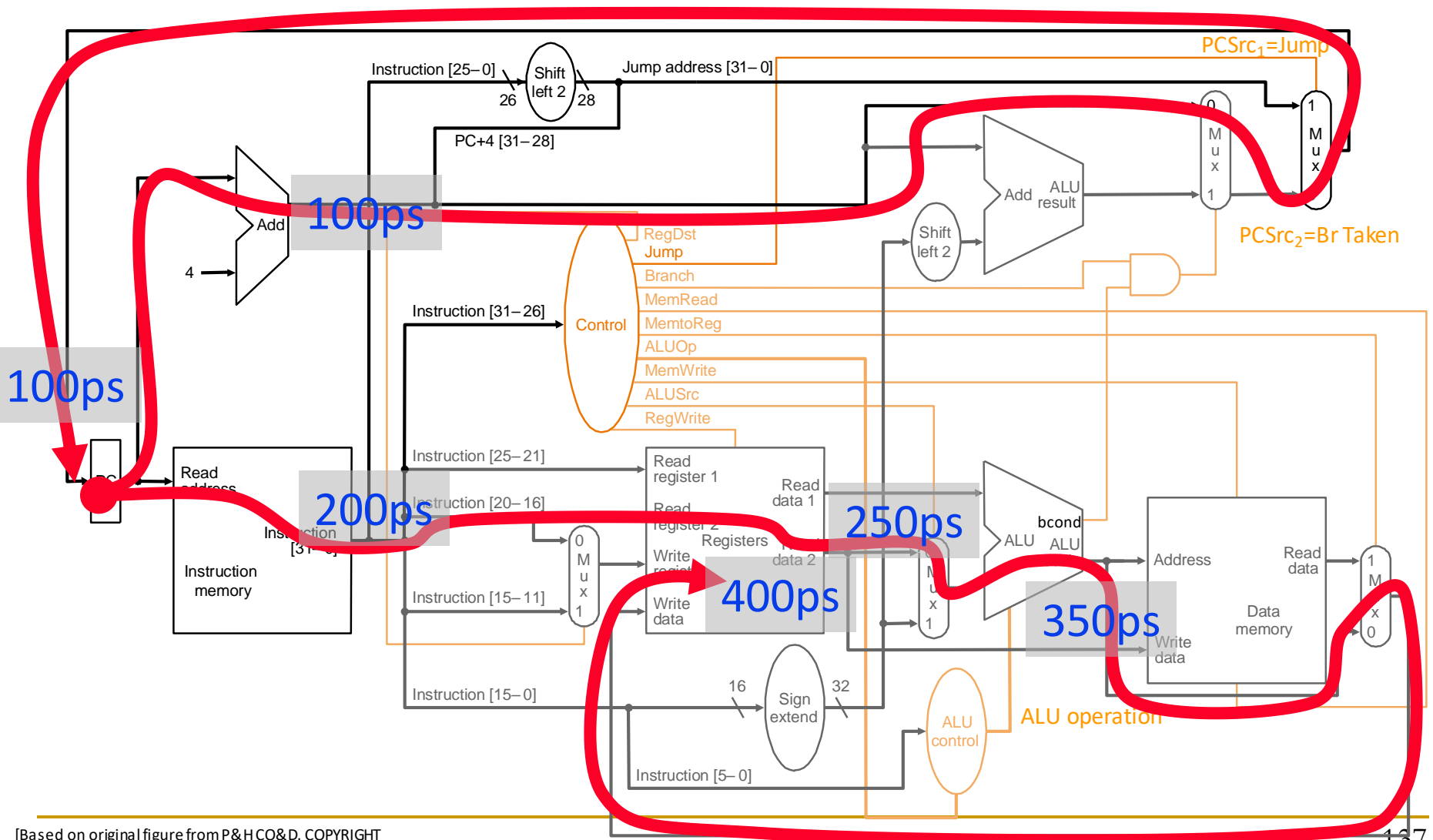
- Assume (for the design in the previous slide)
 - ❑ memory units (read or write): 200 ps
 - ❑ ALU and adders: 100 ps
 - ❑ register file (read or write): 50 ps
 - ❑ other combinational logic: 0 ps

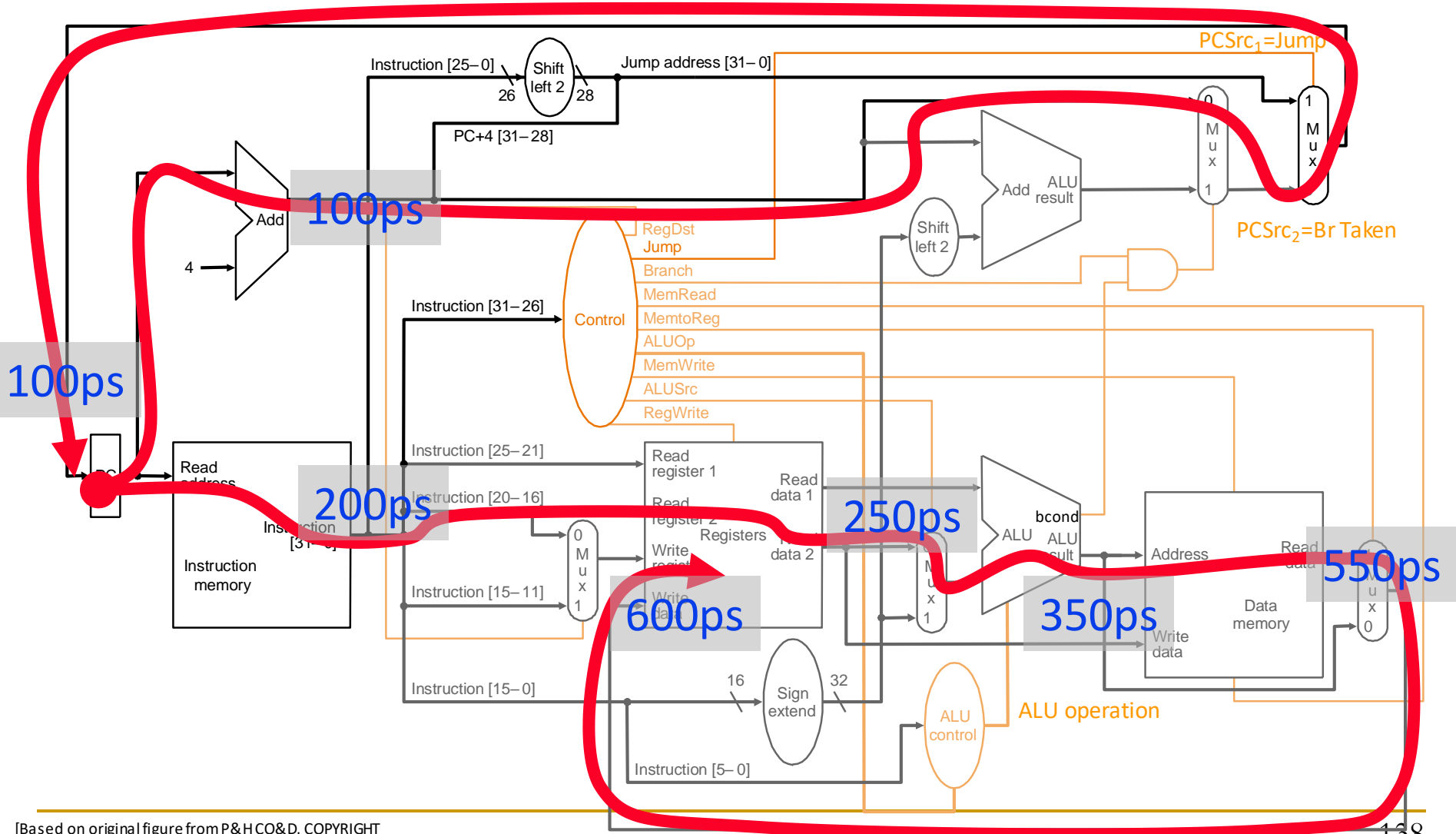
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

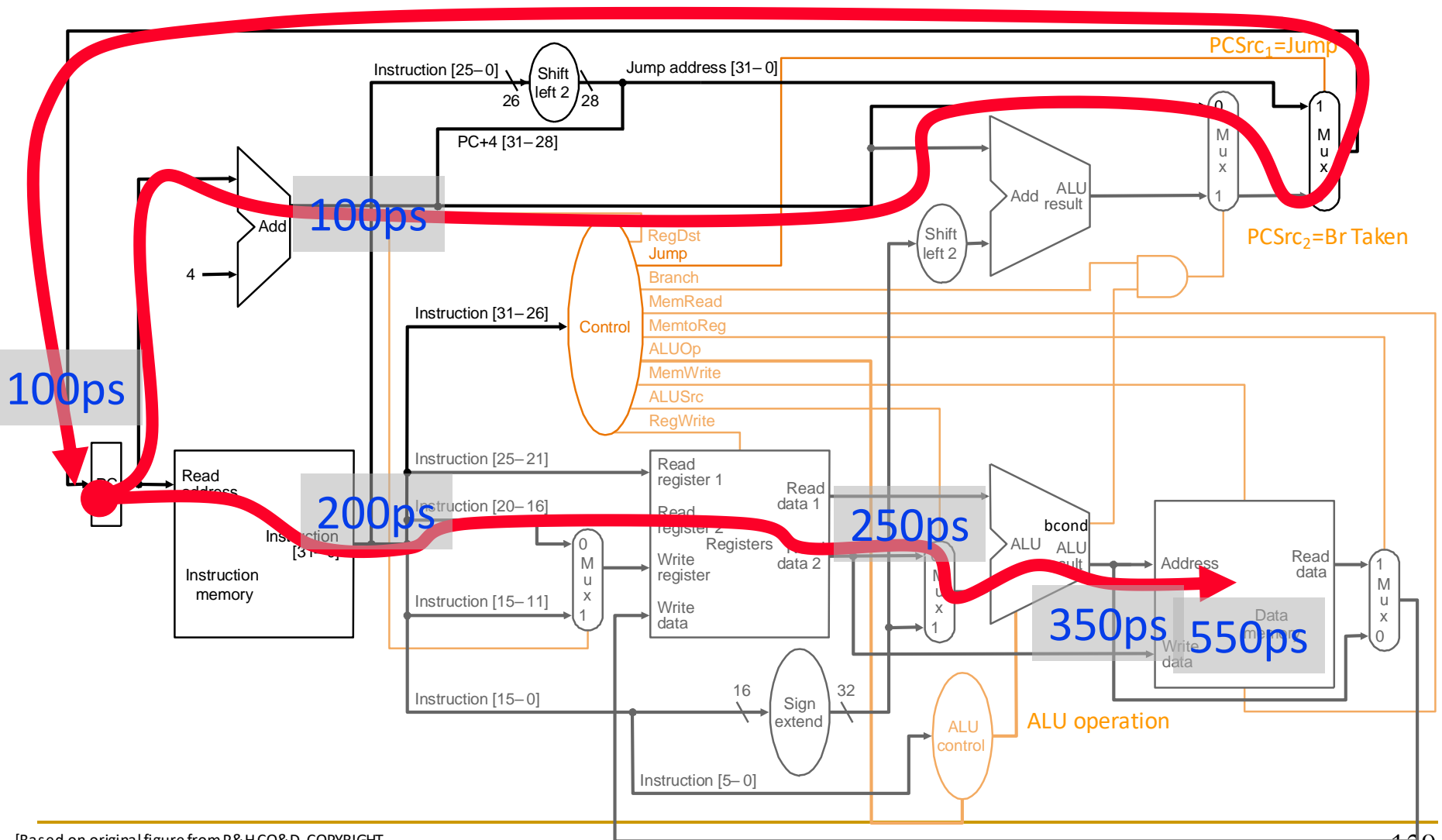
Let's Find the Critical Path



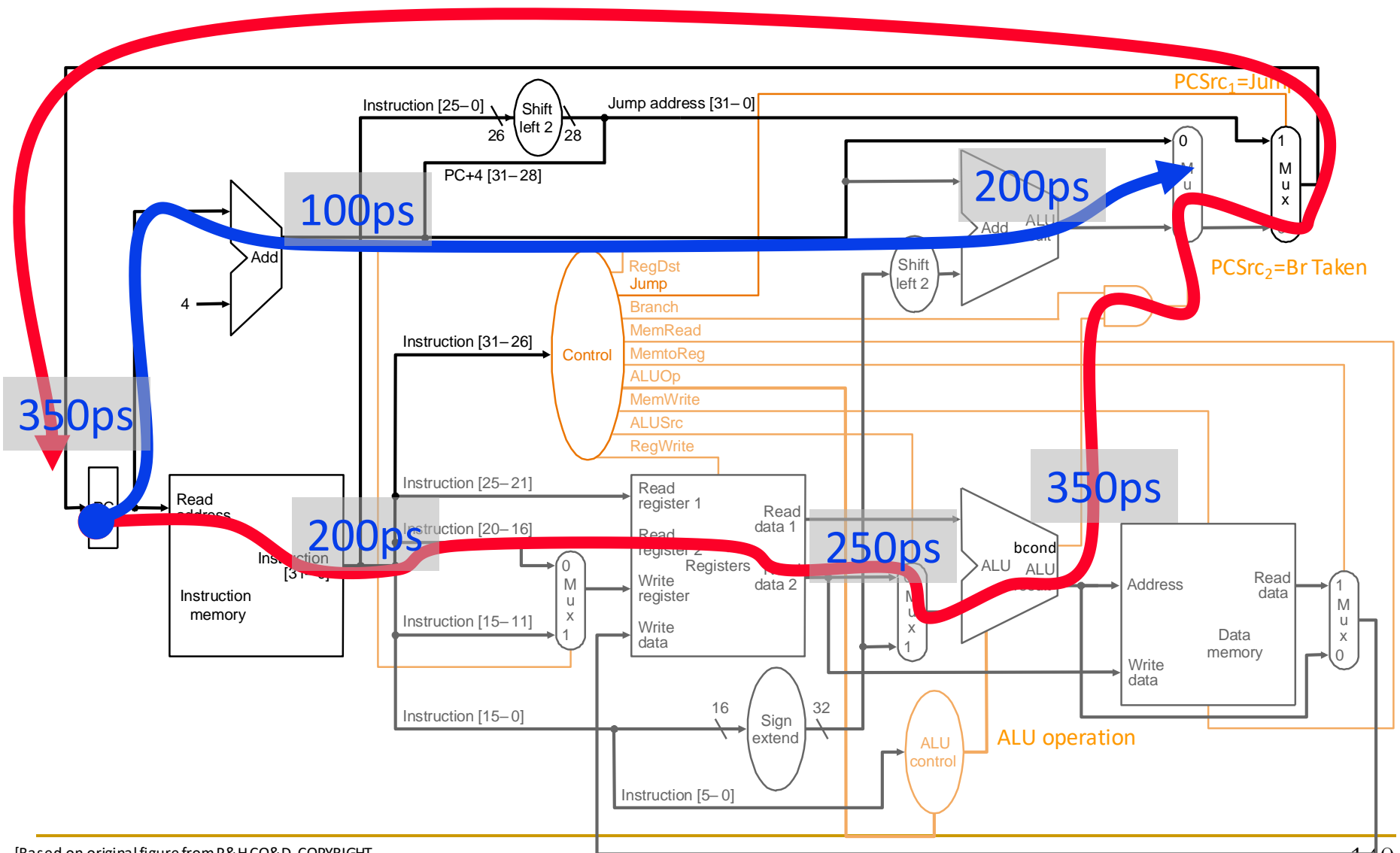
R-Type and I-Type ALU



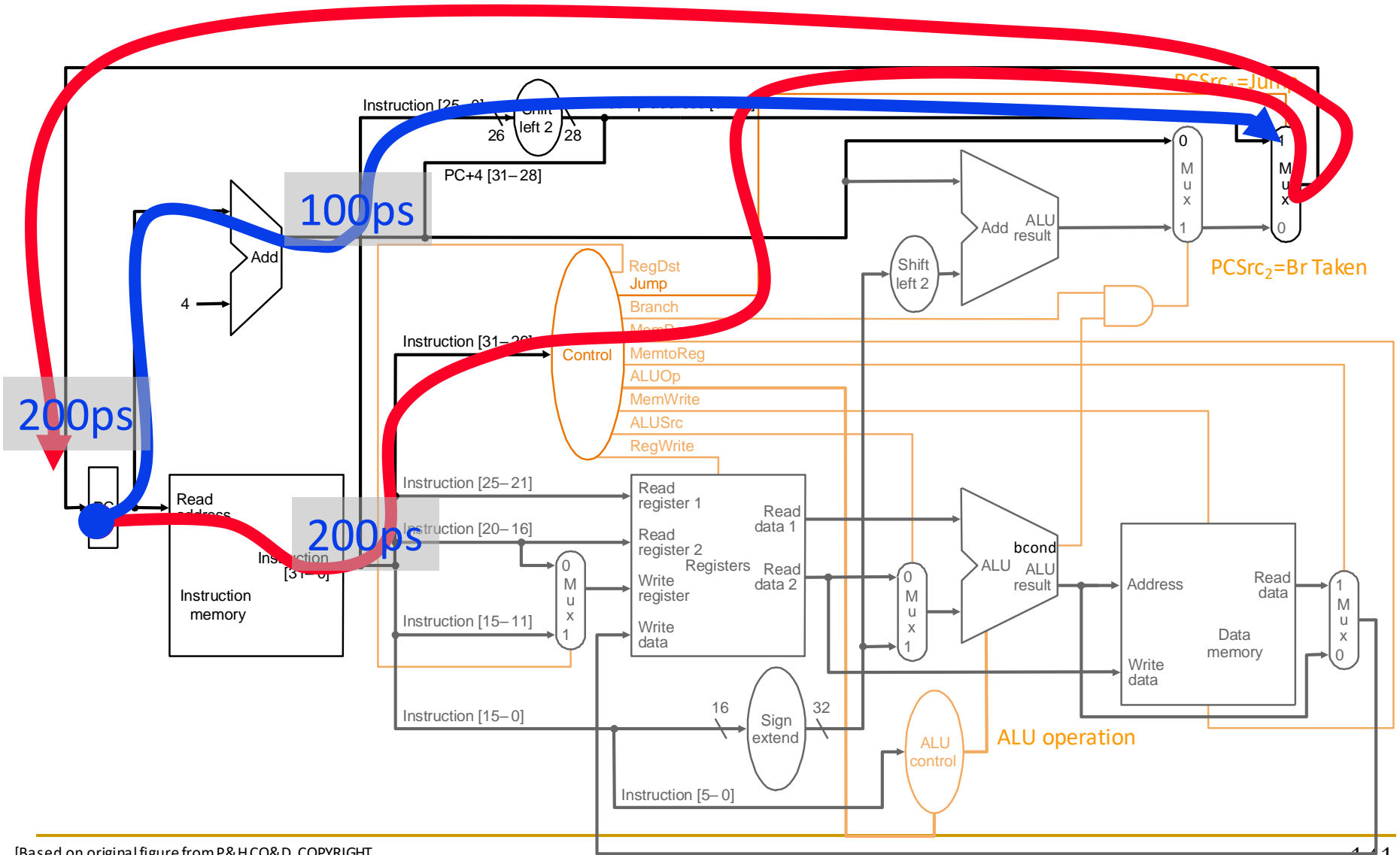




Branch Taken



Jump



What About Control Logic?

- How does that affect the critical path?
- Food for thought for you:
 - Can control logic be on the critical path?
 - Historical example:
 - CDC 5600: control store access too long...

What is the Slowest Instruction to Process?

- Real world: **Memory is slow (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 MOVSB (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 with respect to these principles?

Aside: System Design Principles

- When designing computer systems/architectures, it is important to follow good principles
 - Actually, this is true for *any* system design
 - Real architectures, buildings, bridges, ...
 - Good consumer products
 - ...

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

Aside: From Lecture 2

- “architecture [...] based upon **principle**, and not upon **precedent**”



This



That



Recall: Takeaways

- It all starts from the basic building blocks and design principles
- And, knowledge of how to use, apply, enhance them
- Underlying technology might change (e.g., steel vs. wood)
 - but methods of taking advantage of technology bear resemblance
 - methods used for design depend on the principles employed

Aside: System Design Principles

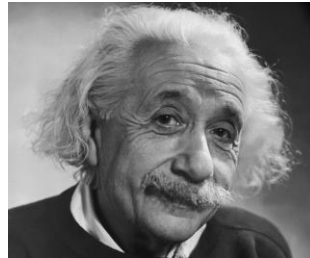
- We will continue to cover key principles in this course
- Here are some references where you can learn more
- Yale Patt, "Requirements, Bottlenecks, and Good Fortune: Agents for Microprocessor Evolution," Proc. of IEEE, 2001. (Levels of transformation, design point, etc)
- Mike Flynn, "Very High-Speed 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>

A Key System Design Principle

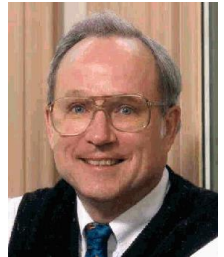
- Keep it simple

- “Everything should be made as simple as possible, but no simpler.”

- Albert Einstein



- And, keep it low cost: “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

Backup Slides on Single-Cycle Uarch for Your Own Study

Please study these to reinforce the concepts
we covered in lectures.

Please do the readings together with these slides:
H&H, Chapter 7.1-7.3, 7.6

Another Single-Cycle MIPS Processor (from H&H)

These are slides for your own study.
They are to complement your reading
H&H, Chapter 7.1-7.3, 7.6

What to do with the Program Counter?

- The PC needs to be incremented by 4 during each cycle (for the time being).
- Initial PC value (after reset) is 0x00400000

```
reg [31:0] PC_p, PC_n;           // Present and next state of PC

// [...]

assign PC_n <= PC_p + 4;         // Increment by 4;

always @ (posedge clk, negedge rst)
begin
    if (rst == '0') PC_p <= 32'h00400000; // default
    else             PC_p <= PC_n;        // when clk
end
```

We Need a Register File

- **Store 32 registers, each 32-bit**
 - $2^5 == 32$, we need 5 bits to address each
- **Every R-type instruction uses 3 register**
 - Two for reading (RS, RT)
 - One for writing (RD)
- **We need a special memory with:**
 - 2 read ports (address x2, data out x2)
 - 1 write port (address, data in)

Register File

```
input [4:0]    a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input                we_rd;
output [31:0] do_rs, do_rt;

    reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description
assign do_rs = R_arr[a_rs];           // Read RS

assign do_rt = R_arr[a_rt];           // Read RT

always @ (posedge clk)
    if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```

Register File

```
input [4:0]    a_rs, a_rt, a_rd;
input [31:0]  di_rd;
input                we_rd;
output [31:0] do_rs, do_rt;

reg [31:0] R_arr [31:0]; // Array that stores regs

// Circuit description; add the trick with $0
assign do_rs = (a_rs != 5'b00000)?    // is address 0?
               R_arr[a_rs] : 0;        // Read RS or 0

assign do_rt = (a_rt != 5'b00000)?    // is address 0?
               R_arr[a_rt] : 0;        // Read RT or 0

always @ (posedge clk)
    if (we_rd) R_arr[a_rd] <= di_rd; // write RD
```


Data Memory Example

- Will be used to store the bulk of data

```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input                we;
output [31:0] do;

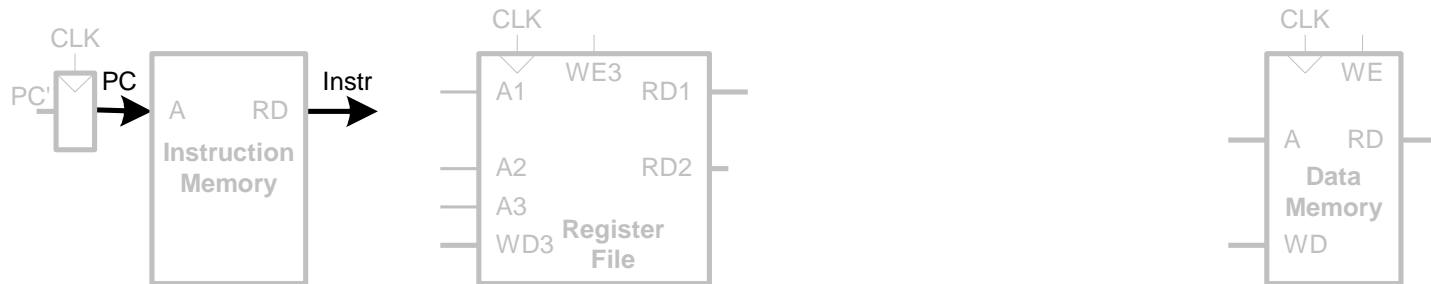
    reg [31:0] M_arr [0:65535];           // Array for Memory

    // Circuit description
    assign do = M_arr[addr];              // Read memory

    always @ (posedge clk)
        if (we) M_arr[addr] <= di;       // write memory
```

Single-Cycle Datapath: lw fetch

■ **STEP 1:** Fetch instruction



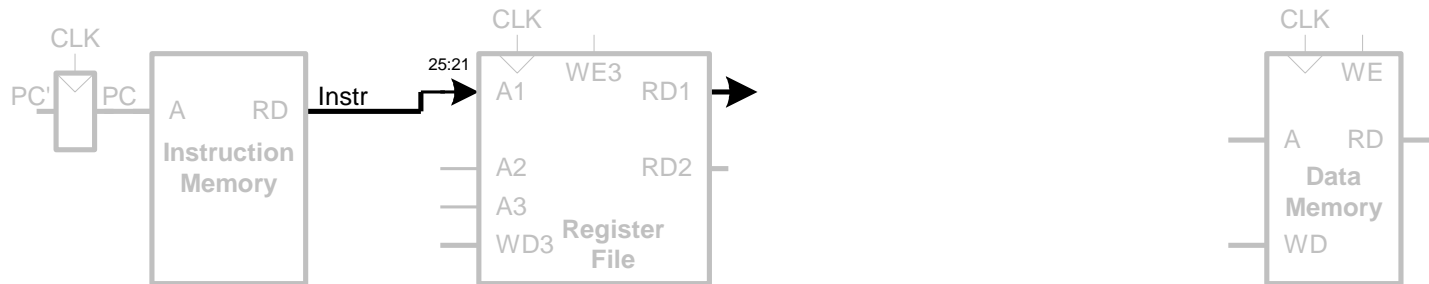
lw \$s3, 1(\$0) # read memory word 1 into \$s3

I-Type



Single-Cycle Datapath: lw register read

- **STEP 2:** Read source operands from register file



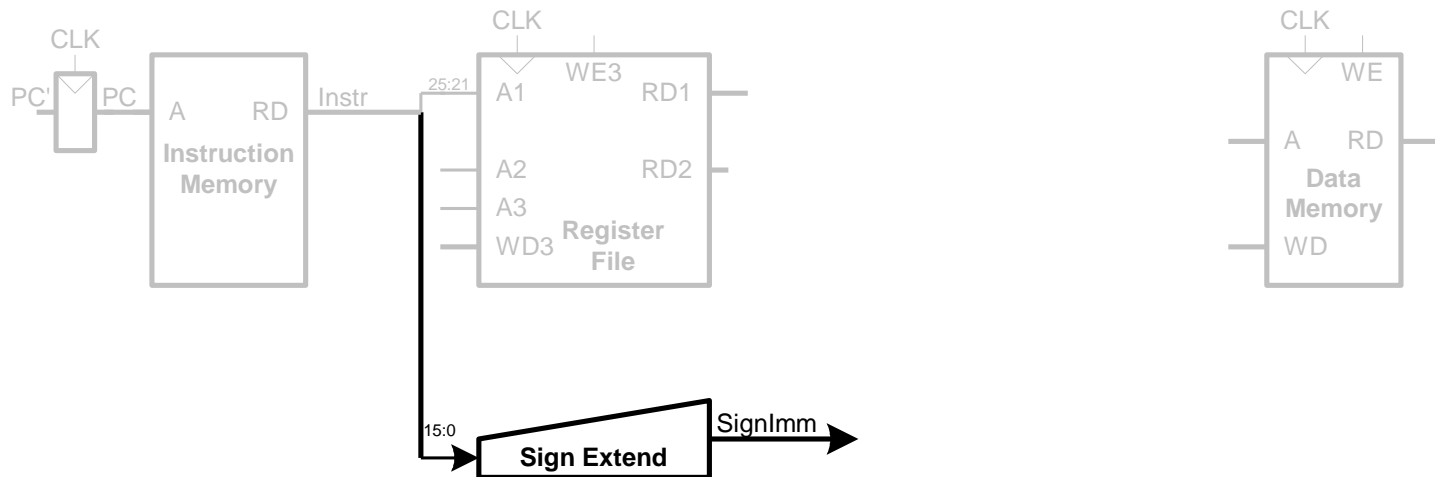
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw immediate

■ STEP 3: Sign-extend the immediate



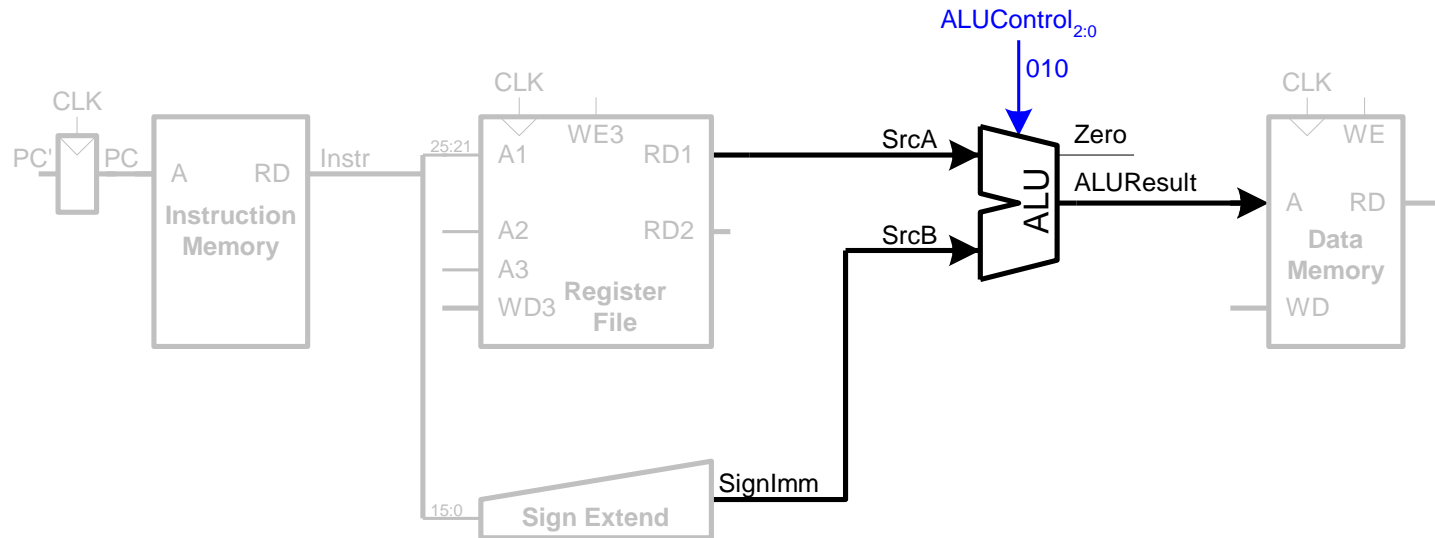
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw address

■ **STEP 4:** Compute the memory address



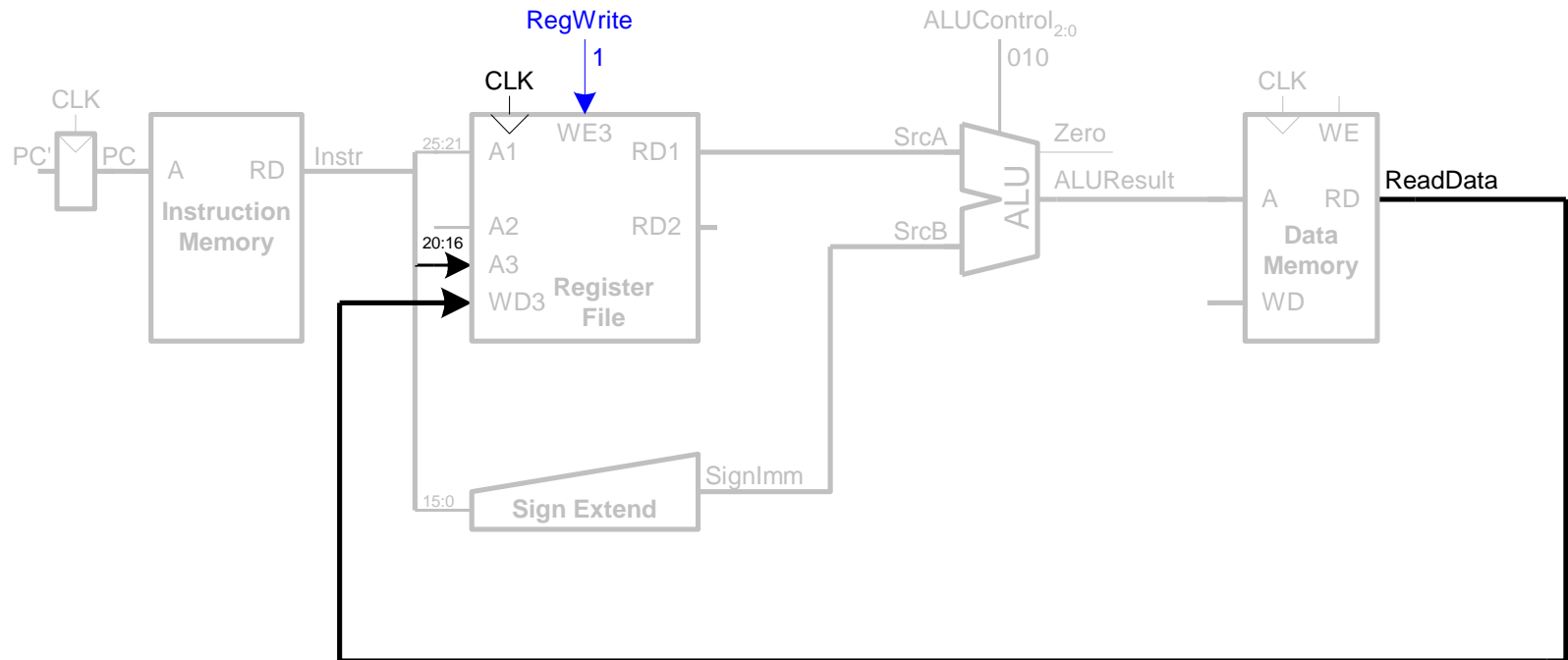
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw memory read

■ **STEP 5:** Read from memory and write back to register file



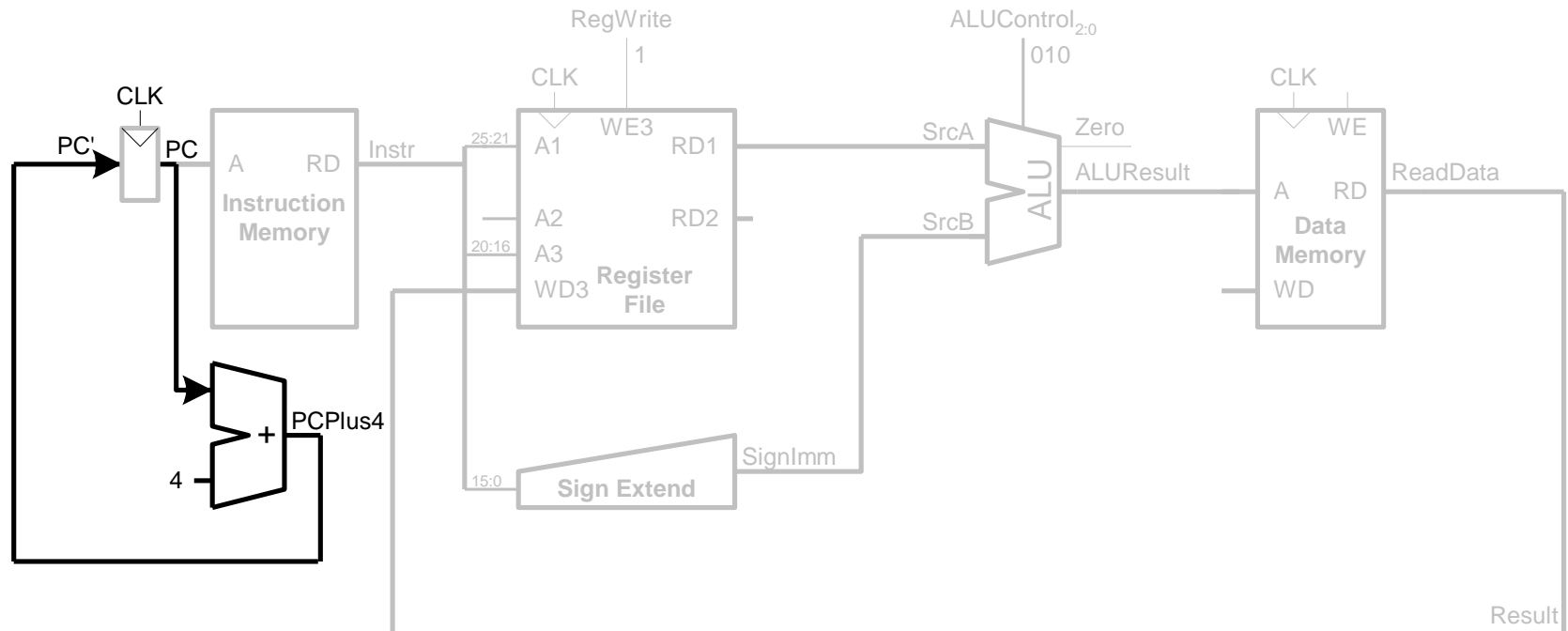
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: lw PC increment

■ **STEP 6:** Determine address of next instruction



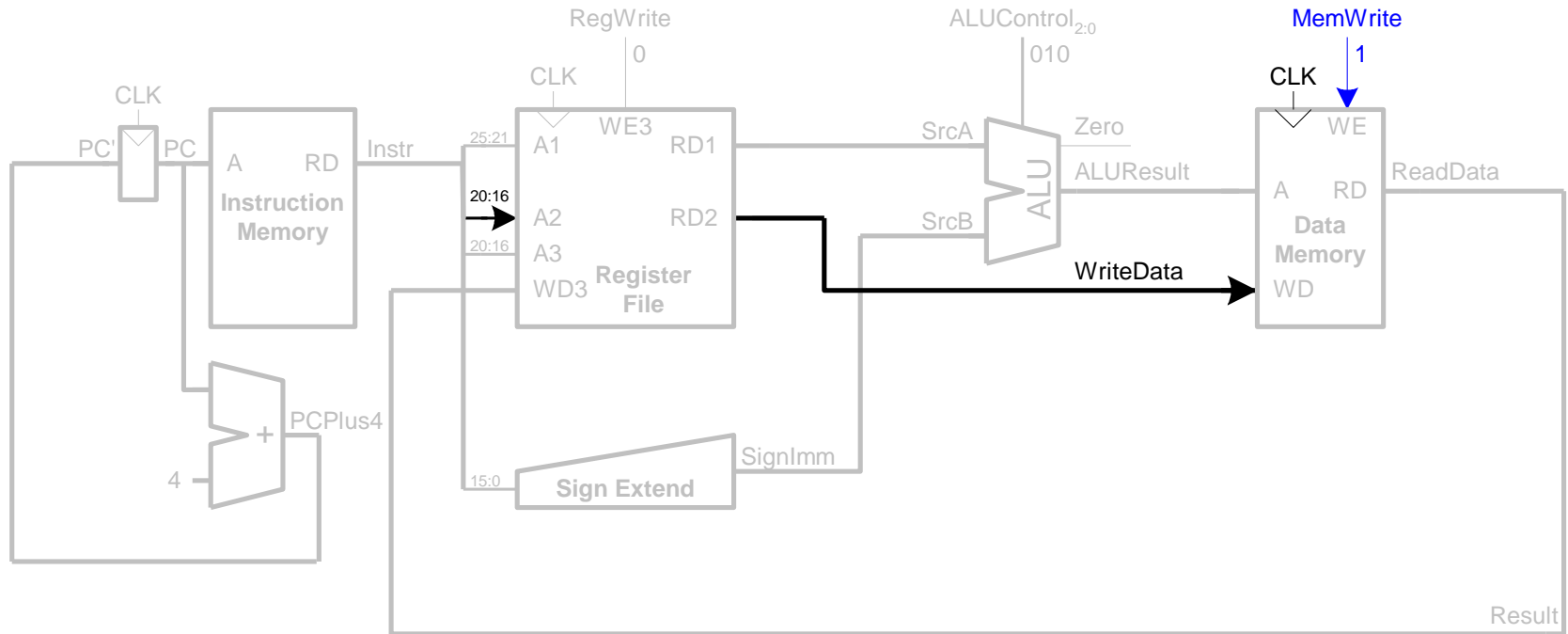
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: sw

■ Write data in rt to memory



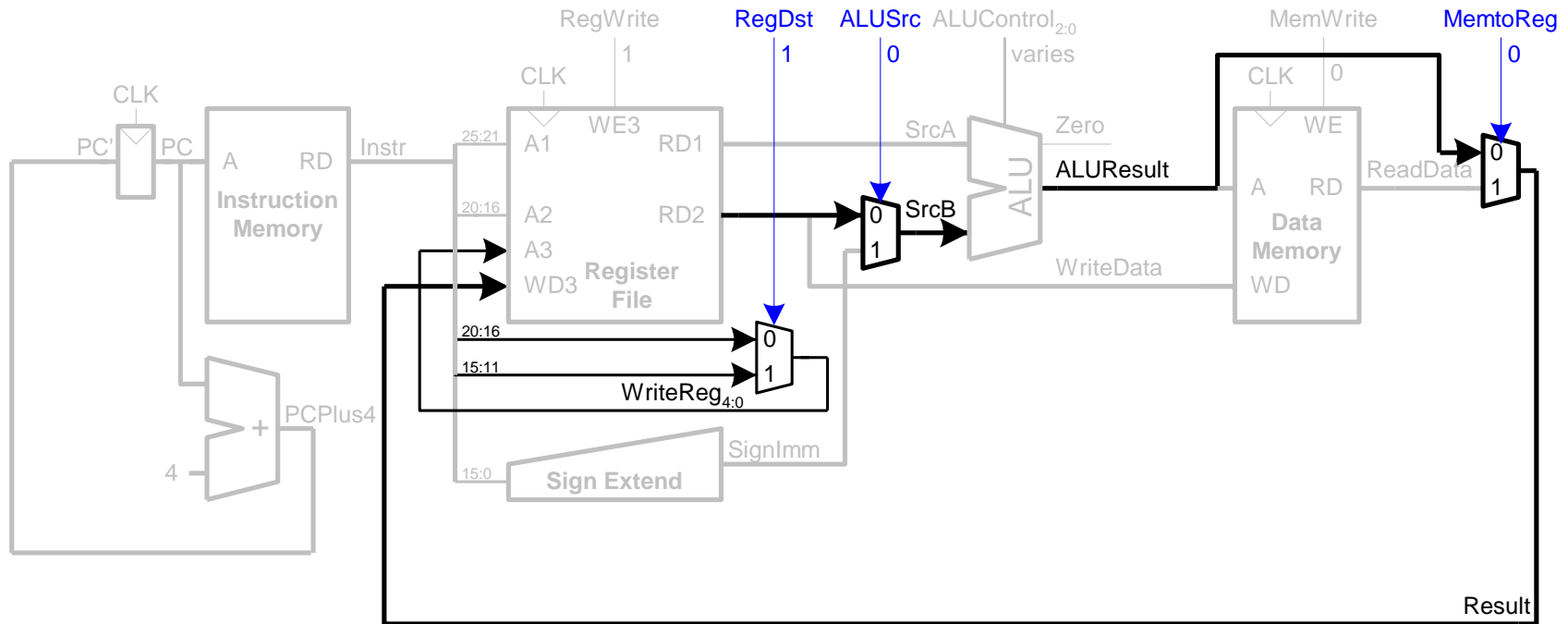
`sw $t7, 44($0) # write t7 into memory address 44`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Single-Cycle Datapath: R-type Instructions

- Read from rs and rt, write ALUResult to register file

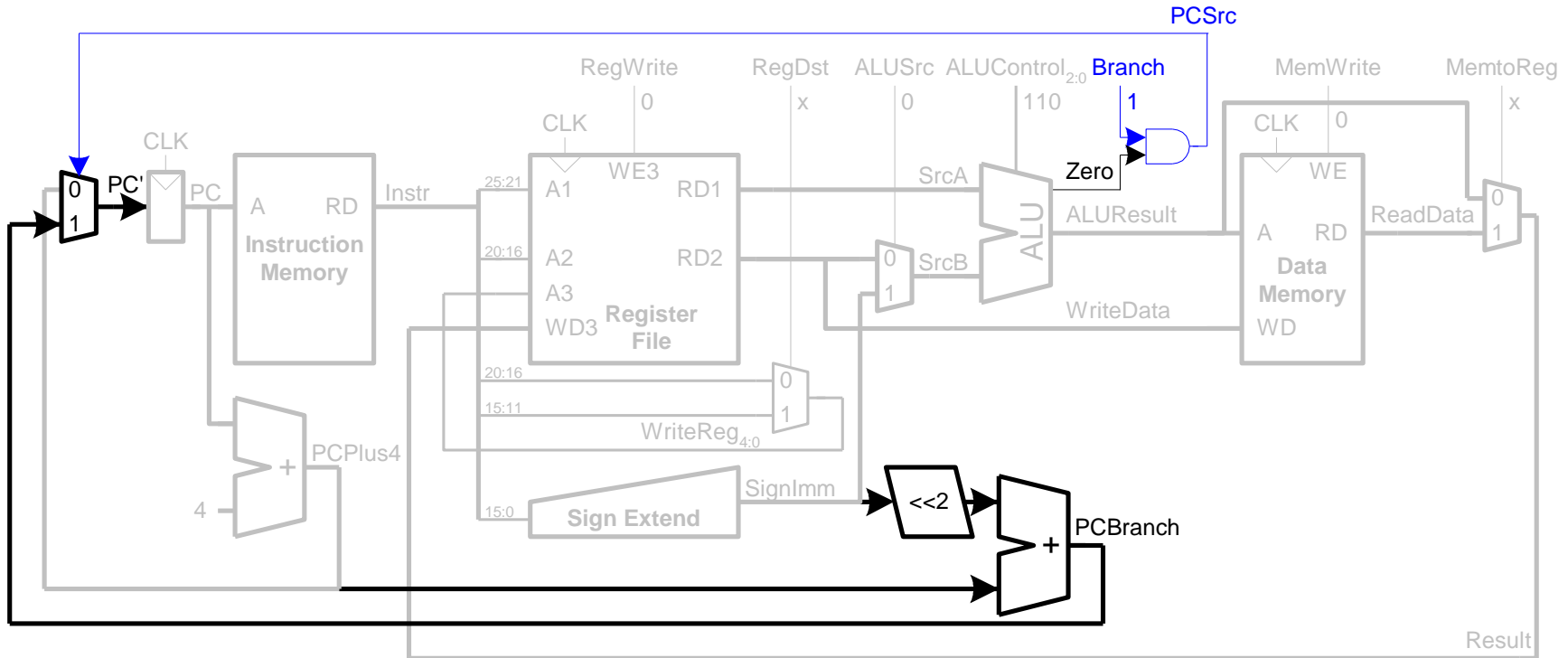


add t, b, c # t = b + c

R-Type

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

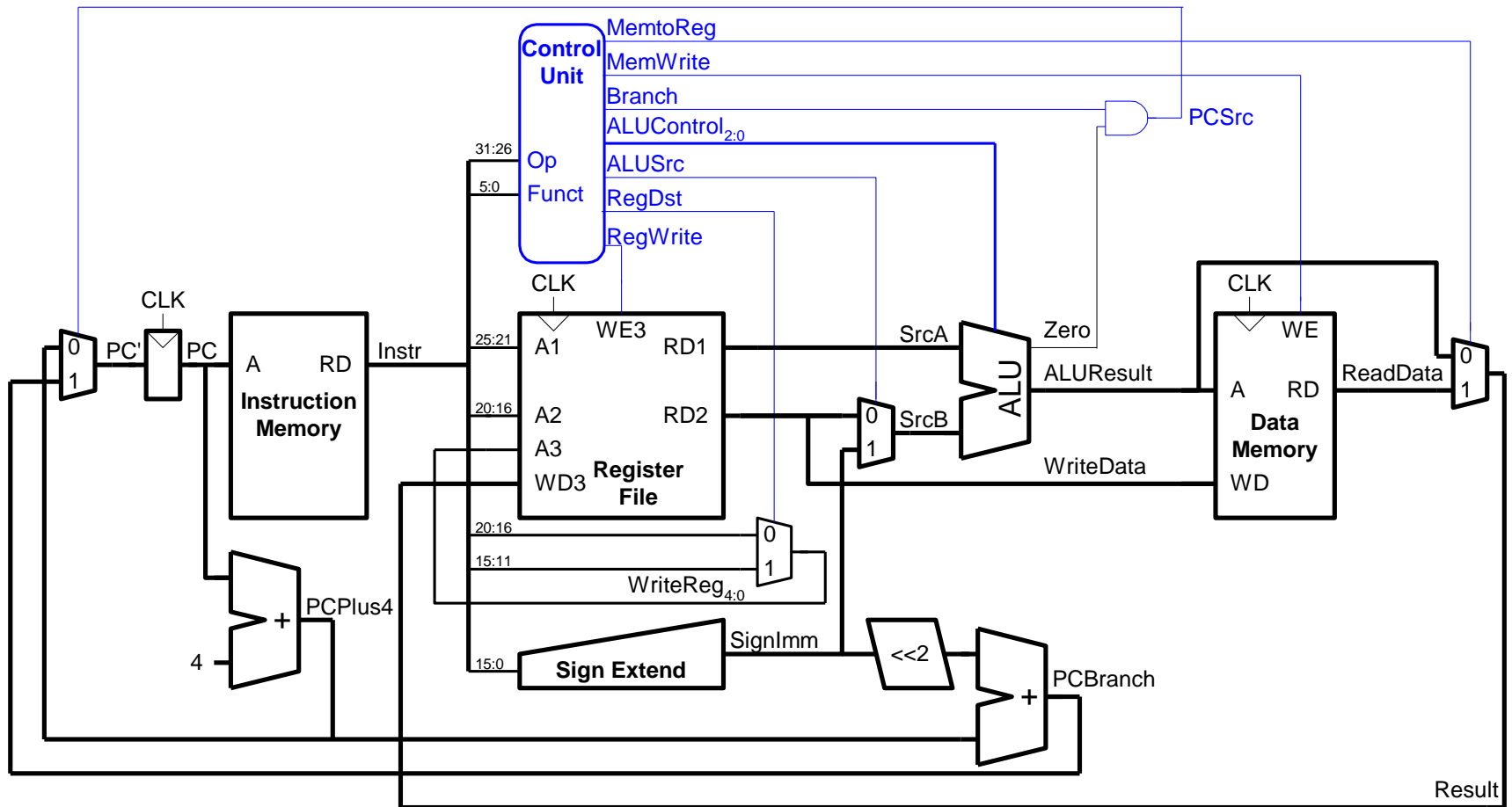
Single-Cycle Datapath: beq



`beq $s0, $s1, target` # branch is taken

- Determine whether values in `rs` and `rt` are equal
Calculate $BTA = (\text{sign-extended immediate} \ll 2) + (PC+4)$

Complete Single-Cycle Processor

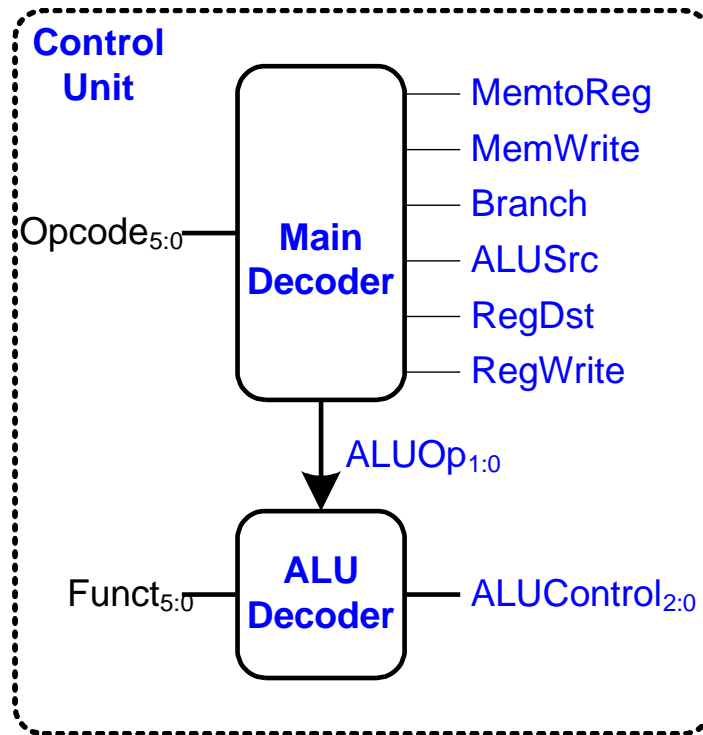


Our MIPS Datapath has Several Options

- **ALU inputs**
 - Either RT or Immediate (*MUX*)
- **Write Address of Register File**
 - Either RD or RT (*MUX*)
- **Write Data In of Register File**
 - Either ALU out or Data Memory Out (*MUX*)
- **Write enable of Register File**
 - Not always a register write (*MUX*)
- **Write enable of Memory**
 - Only when writing to memory (sw) (*MUX*)

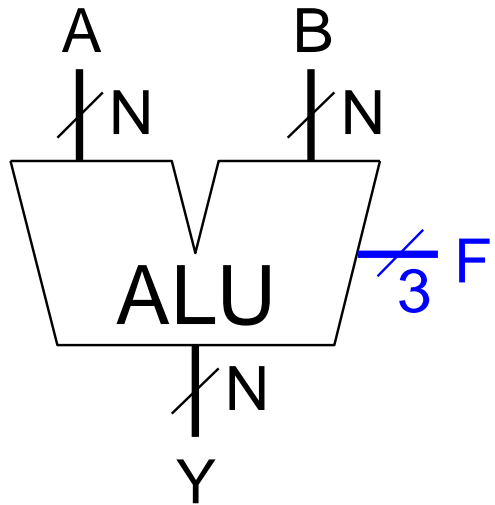
All these options are our control signals

Control Unit



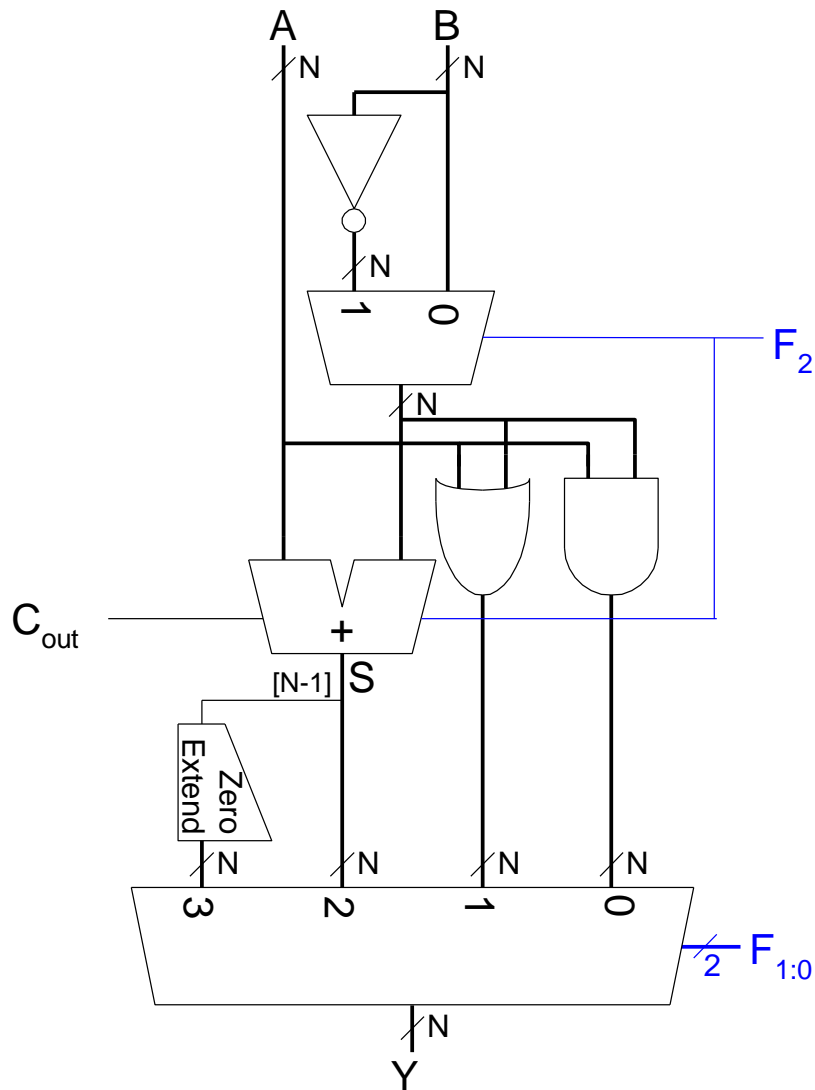
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



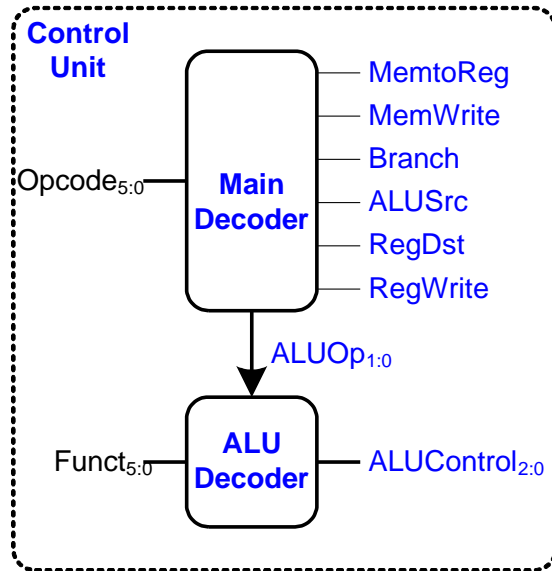
$F_{2:0}$	Function
000	$A \& B$
001	$A B$
010	$A + B$
011	not used
100	$A \& \sim B$
101	$A \sim B$
110	$A - B$
111	SLT

ALU Internals



$F_{2:0}$	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & $\sim B$
101	A $\sim B$
110	A - B
111	SLT

Control Unit: ALU Decoder



ALUOp _{1:0}	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

ALUOp _{1:0}	Funct	ALUControl _{2:0}
00	X	010 (Add)
X1	X	110 (Subtract)
1X	100000 (add)	010 (Add)
1X	100010 (sub)	110 (Subtract)
1X	100100 (and)	000 (And)
1X	100101 (or)	001 (Or)
1X	101010 (slt)	111 (SLT)

Let us Develop our Control Table

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do

Let us Develop our Control Table

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do

Let us Develop our Control Table

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
lw	100011	1	0	1	0	1	add

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do

Let us Develop our Control Table

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	funct
lw	100011	1	0	1	0	1	add
sw	101011	0	X	1	1	X	add

- **RegWrite:** Write enable for the register file
- **RegDst:** Write to register RD or RT
- **AluSrc:** ALU input RT or immediate
- **MemWrite:** Write Enable
- **MemtoReg:** Register data in from Memory or ALU
- **ALUOp:** What operation does ALU do

More Control Signals

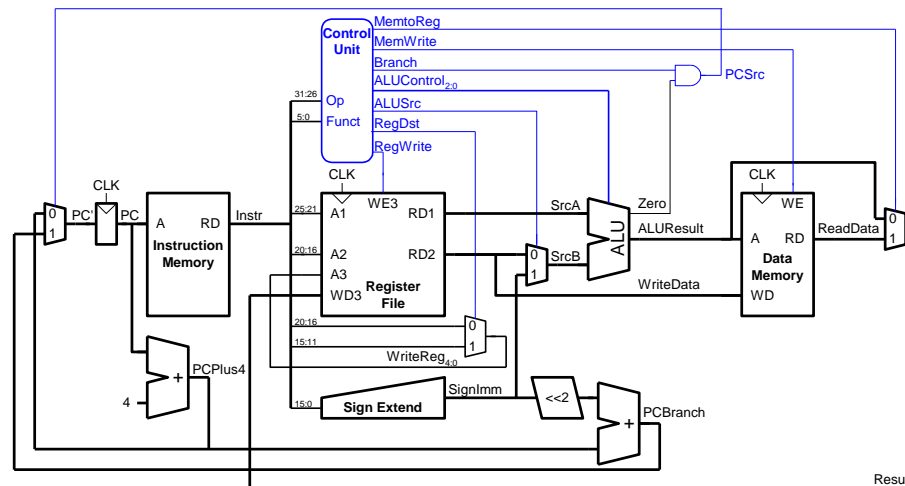
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

■ New Control Signal

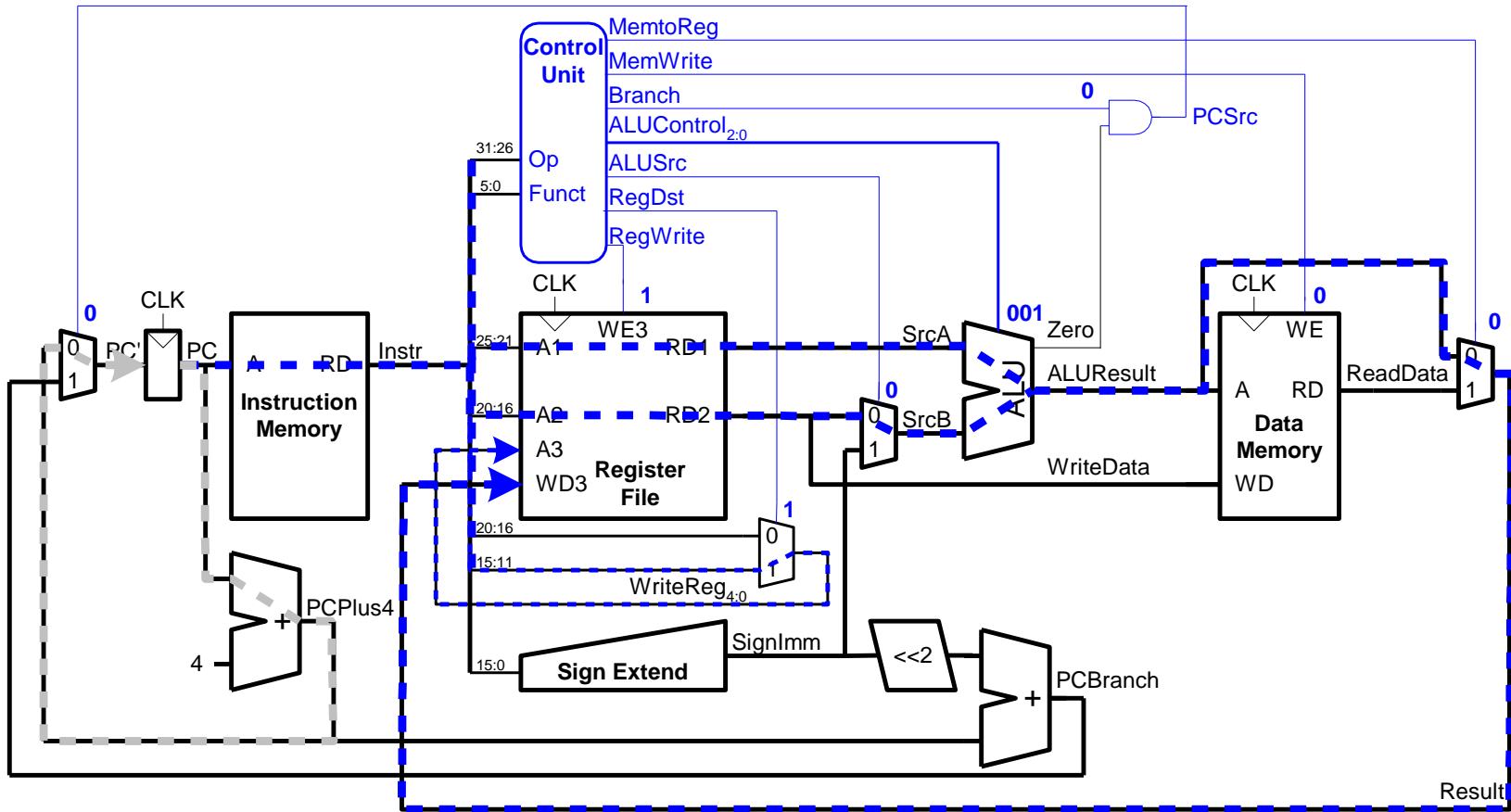
- **Branch:** Are we jumping or not ?

Control Unit: Main Decoder

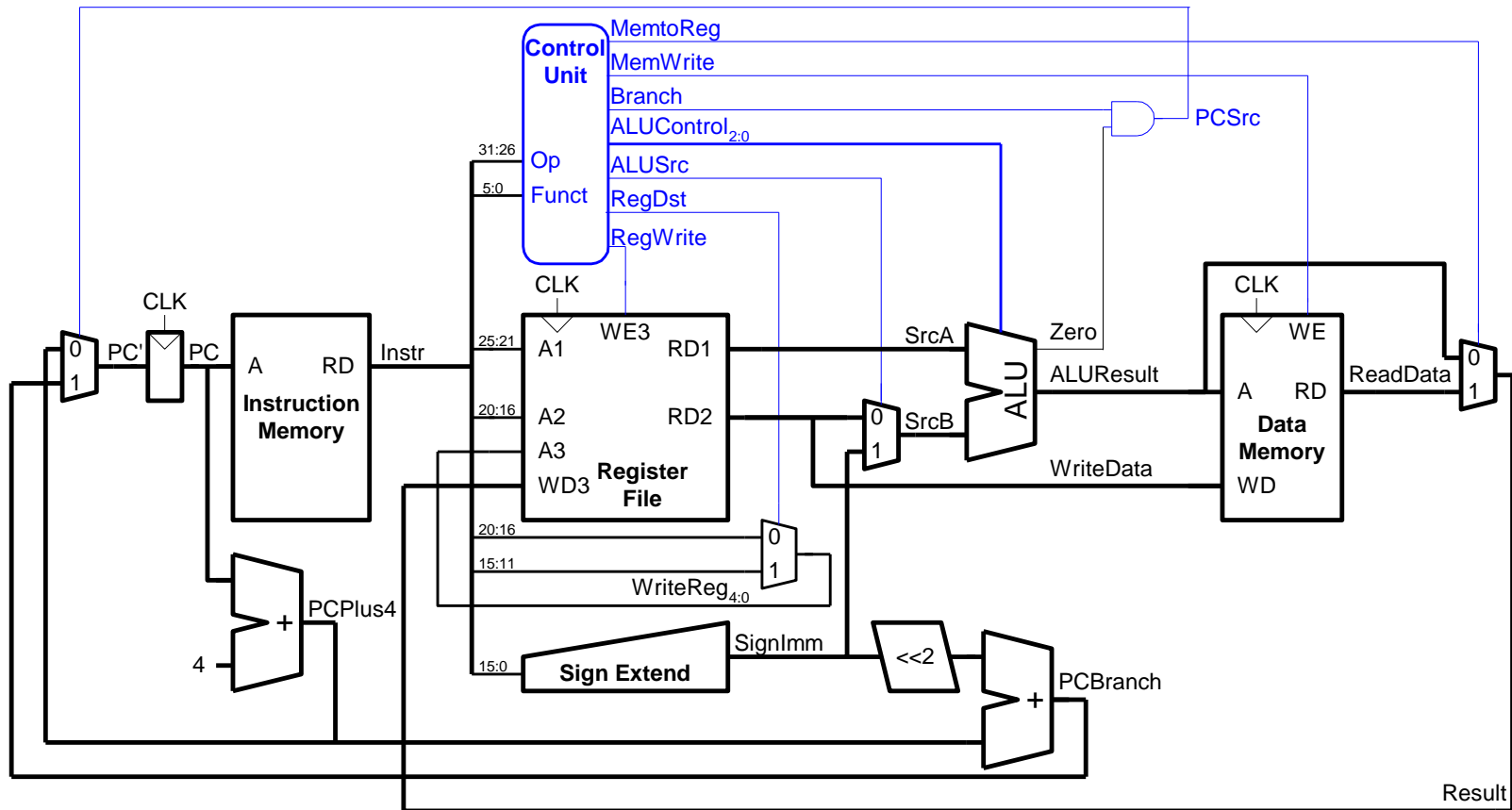
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

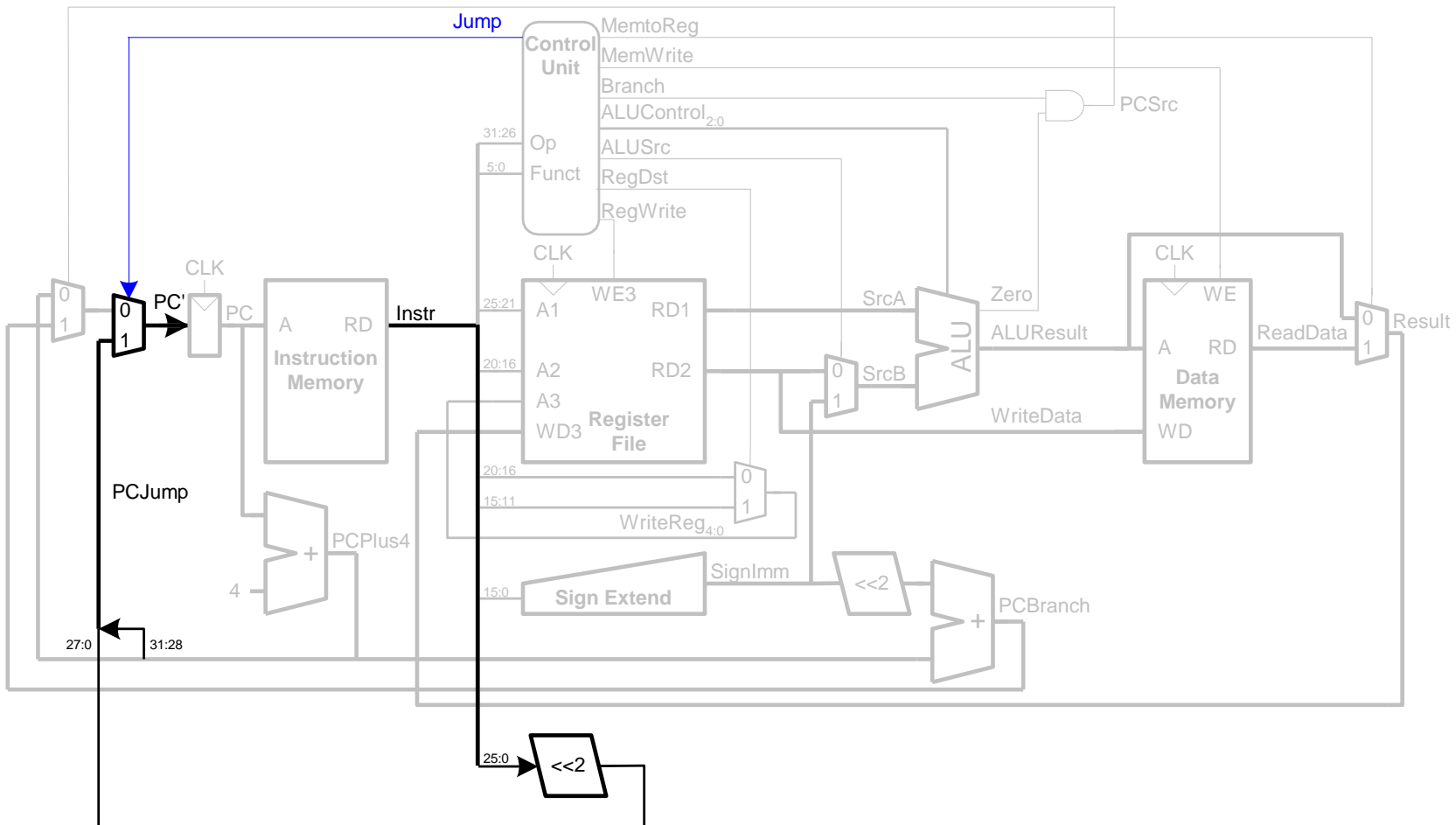


■ No change to datapath

Control Unit: addi

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

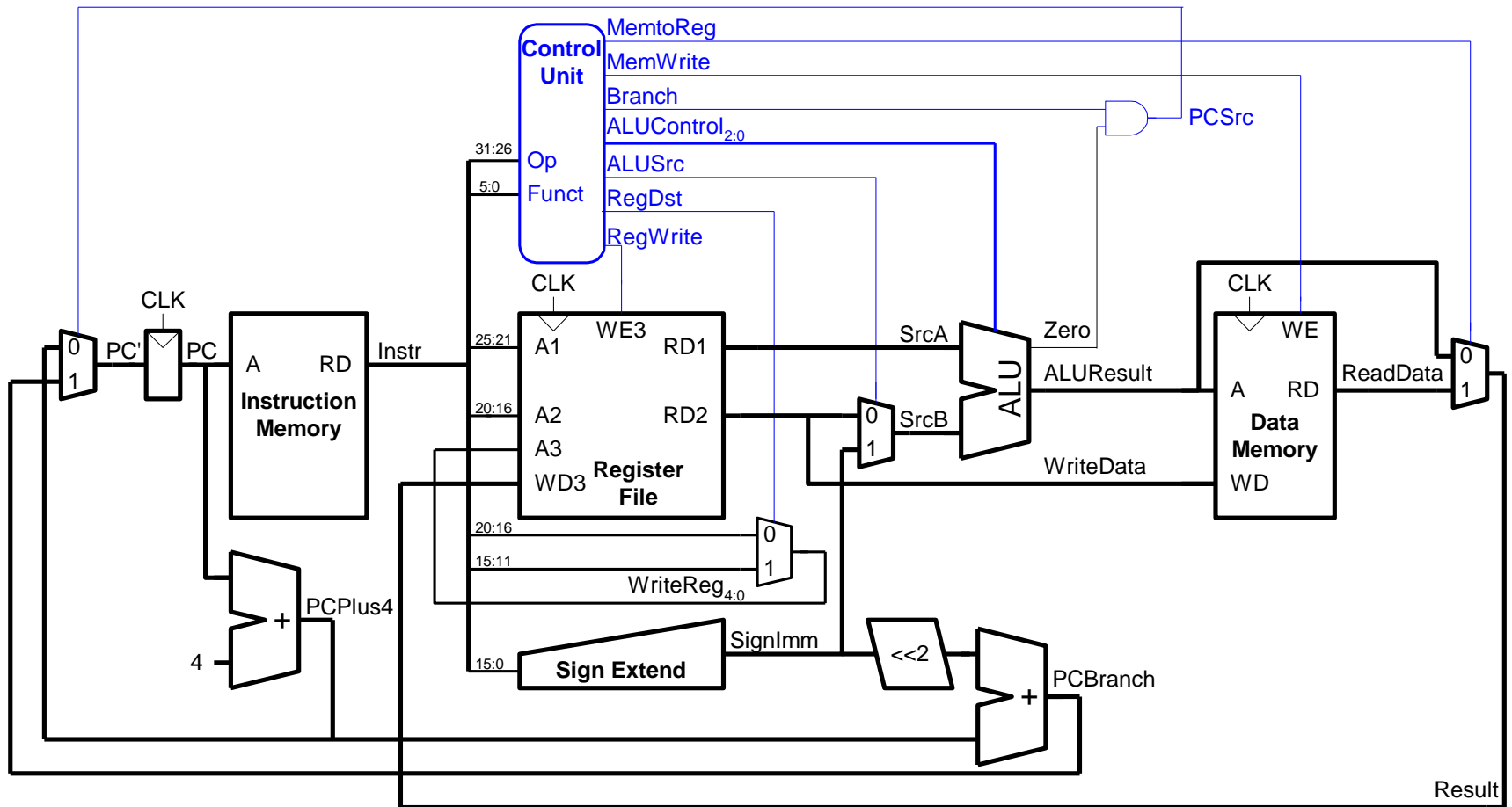
Extended Functionality: j



Control Unit: Main Decoder

Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
j	000100	0	X	X	X	0	X	XX	1

Review: Complete Single-Cycle Processor (H&H)



A Bit More on Performance Analysis

Processor Performance

- **How fast is my program?**
 - Every program consists of a series of instructions
 - Each instruction needs to be executed.

Processor Performance

■ How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed.

■ So how fast are my instructions ?

- Instructions are realized on the hardware
- They can take one or more clock cycles to complete
- *Cycles per Instruction = CPI*

Processor Performance

■ How fast is my program?

- Every program consists of a series of instructions
- Each instruction needs to be executed.

■ So how fast are my instructions ?

- Instructions are realized on the hardware
- They can take one or more clock cycles to complete
- *Cycles per Instruction = CPI*

■ How much time is one clock cycle?

- The critical path determines how much time one cycle requires = *clock period*.
- $1/\text{clock period} = \text{clock frequency}$ = how many cycles can be done each second.

Performance Analysis

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

Processor Performance

■ Now as a general formula

- Our program consists of executing **N** instructions.
- Our processor needs **CPI** cycles for each instruction.
- The maximum clock speed of the processor is **f**,
and the clock period is therefore **$T=1/f$**

Processor Performance

■ Now as a general formula

- Our program consists of executing **N** instructions.
- Our processor needs **CPI** cycles for each instruction.
- The maximum clock speed of the processor is **f**, and the clock period is therefore **T=1/f**

■ Our program will execute in

$$N \times CPI \times (1/f) = N \times CPI \times T \text{ seconds}$$

How can I Make the Program Run Faster?

$$N \times \text{CPI} \times (1/f)$$

How can I Make the Program Run Faster?

$$N \times \text{CPI} \times (1/f)$$

- **Reduce the number of instructions**
 - Make instructions that 'do' more (CISC)
 - Use better compilers

How can I Make the Program Run Faster?

$$N \times \text{CPI} \times (1/f)$$

- **Reduce the number of instructions**
 - Make instructions that 'do' more (CISC)
 - Use better compilers
- **Use less cycles to perform the instruction**
 - Simpler instructions (RISC)
 - Use multiple units/ALUs/cores in parallel

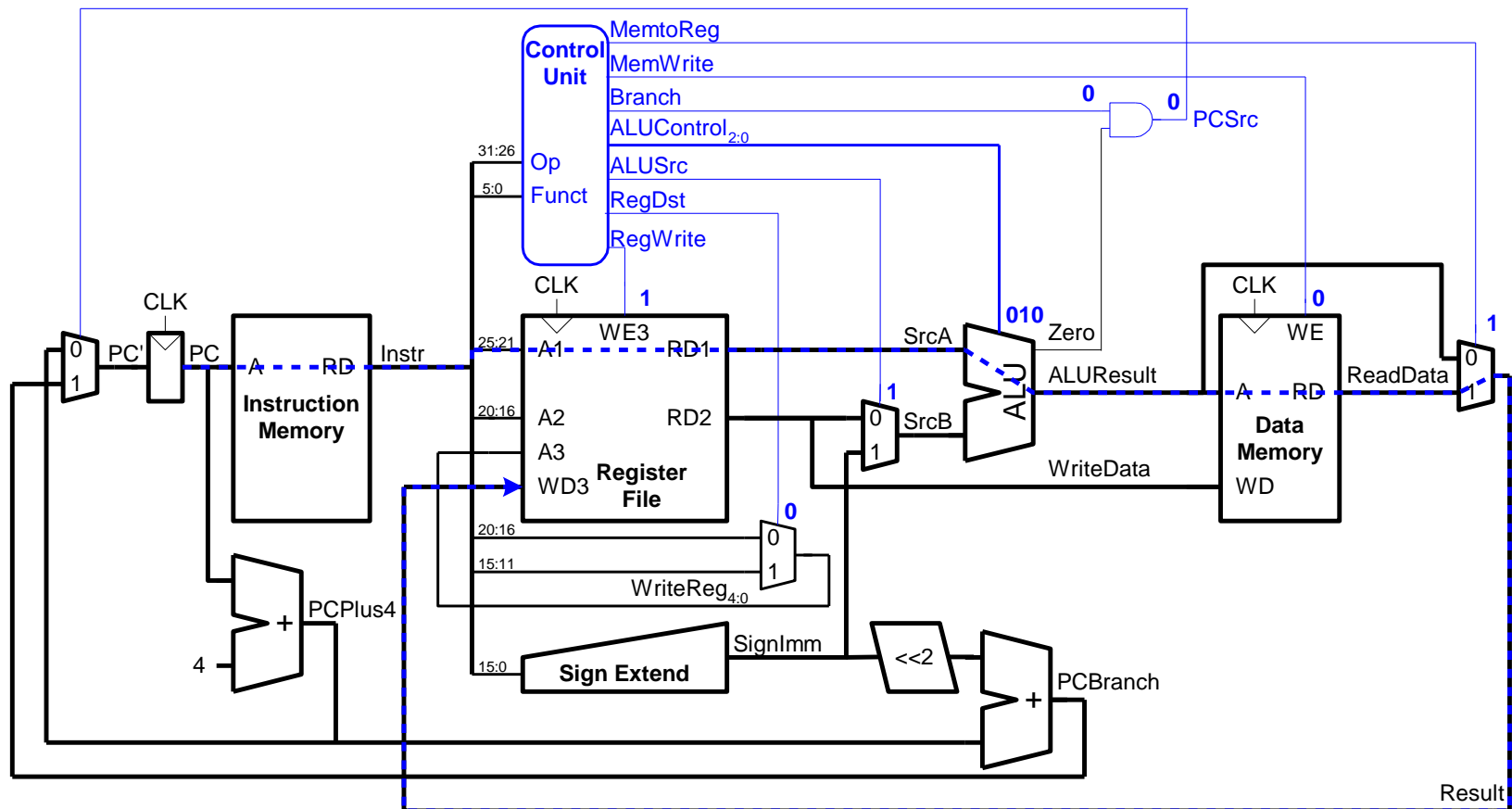
How can I Make the Program Run Faster?

$$N \times \text{CPI} \times (1/f)$$

- **Reduce the number of instructions**
 - Make instructions that 'do' more (CISC)
 - Use better compilers
- **Use less cycles to perform the instruction**
 - Simpler instructions (RISC)
 - Use multiple units/ALUs/cores in parallel
- **Increase the clock frequency**
 - Find a 'newer' technology to manufacture
 - Redesign time critical components
 - Adopt pipelining

Single-Cycle Performance

- T_c is limited by the critical path (1w)



Single-Cycle Performance

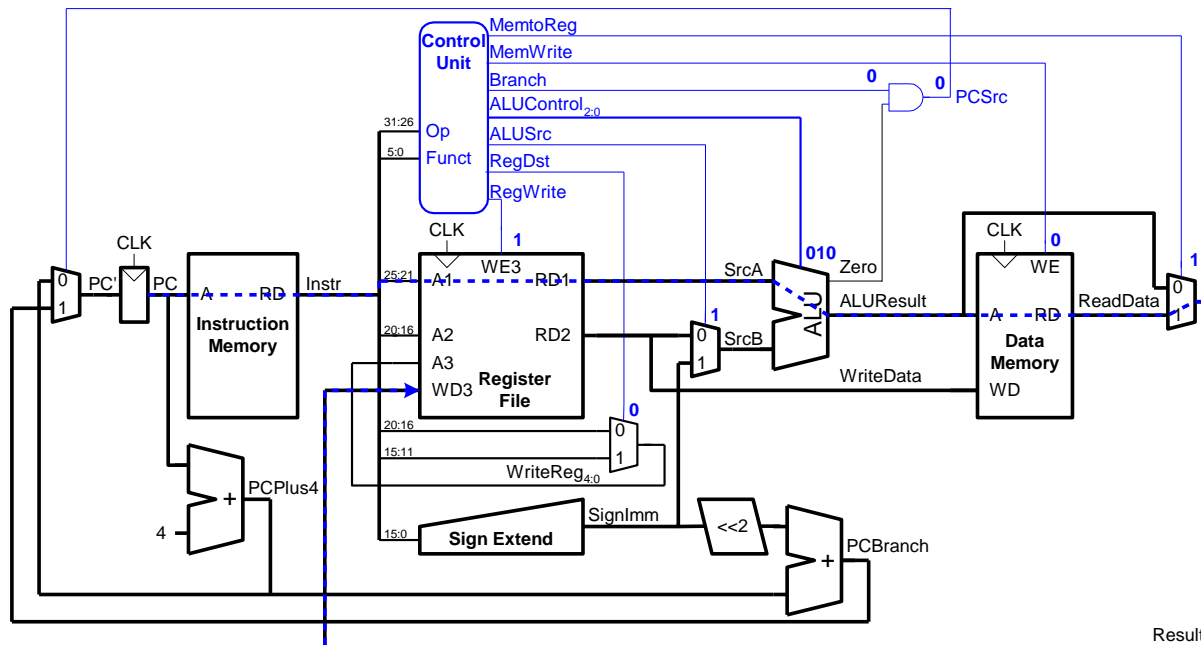
■ Single-cycle critical path:

$$T_c = t_{pcq_PC} + t_{mem} + \max(t_{Rfread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$$

■ In most implementations, limiting paths are:

- memory, ALU, register file.

$$T_c = t_{pcq_PC} + 2t_{mem} + t_{Rfread} + t_{mux} + t_{ALU} + t_{RFsetup}$$



Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$T_c =$$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned}T_c &= t_{pcq_PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup} \\&= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps} \\&= 925 \text{ ps}\end{aligned}$$

Single-Cycle Performance Example

■ Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:

Single-Cycle Performance Example

■ Example:

For a program with 100 billion instructions executing on a single-cycle MIPS processor:

$$\begin{aligned}\text{Execution Time} &= \# \text{ instructions} \times \text{CPI} \times \text{TC} \\ &= (100 \times 10^9)(1)(925 \times 10^{-12} \text{ s}) \\ &= 92.5 \text{ seconds}\end{aligned}$$