

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

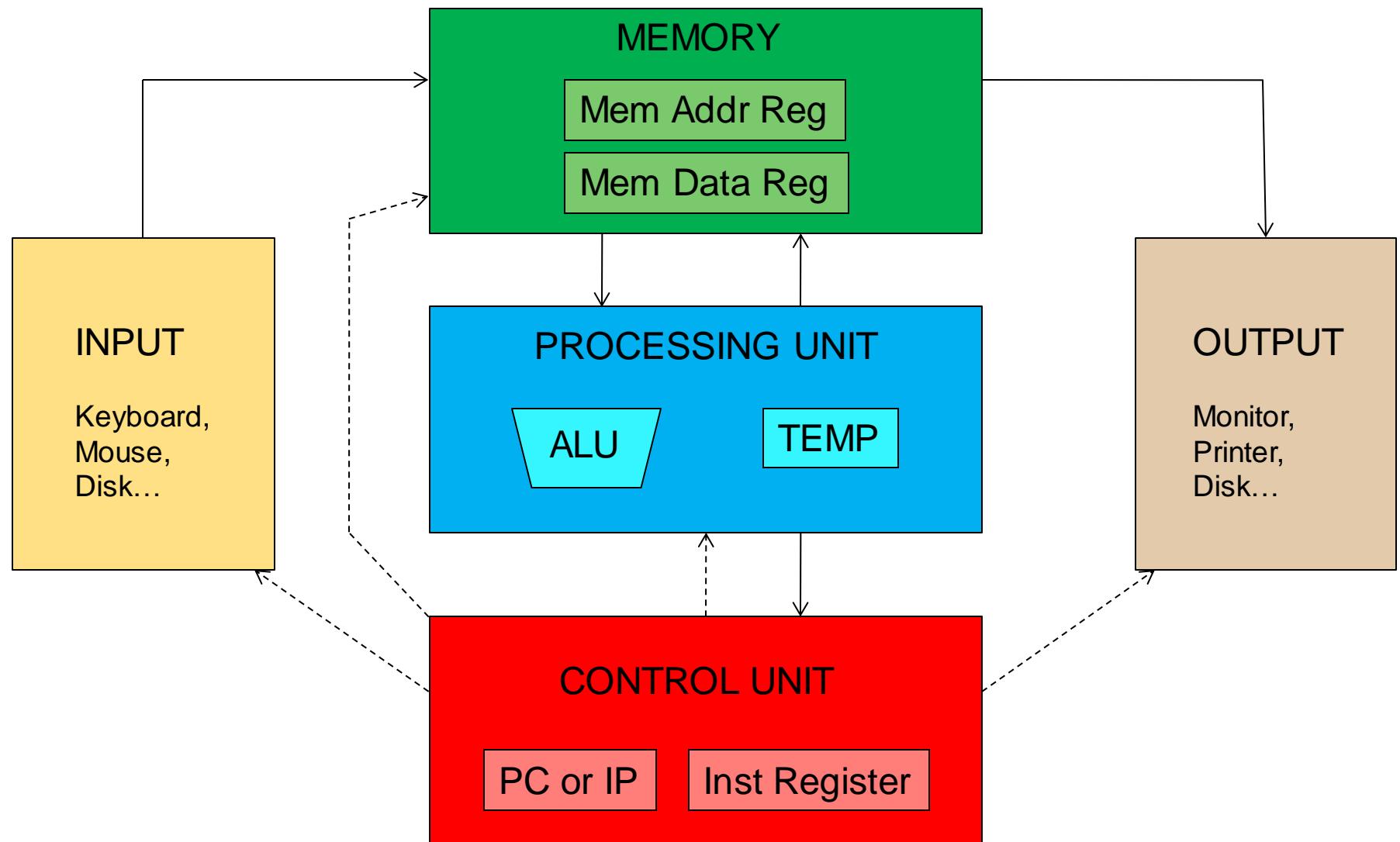
■ Next week

- 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

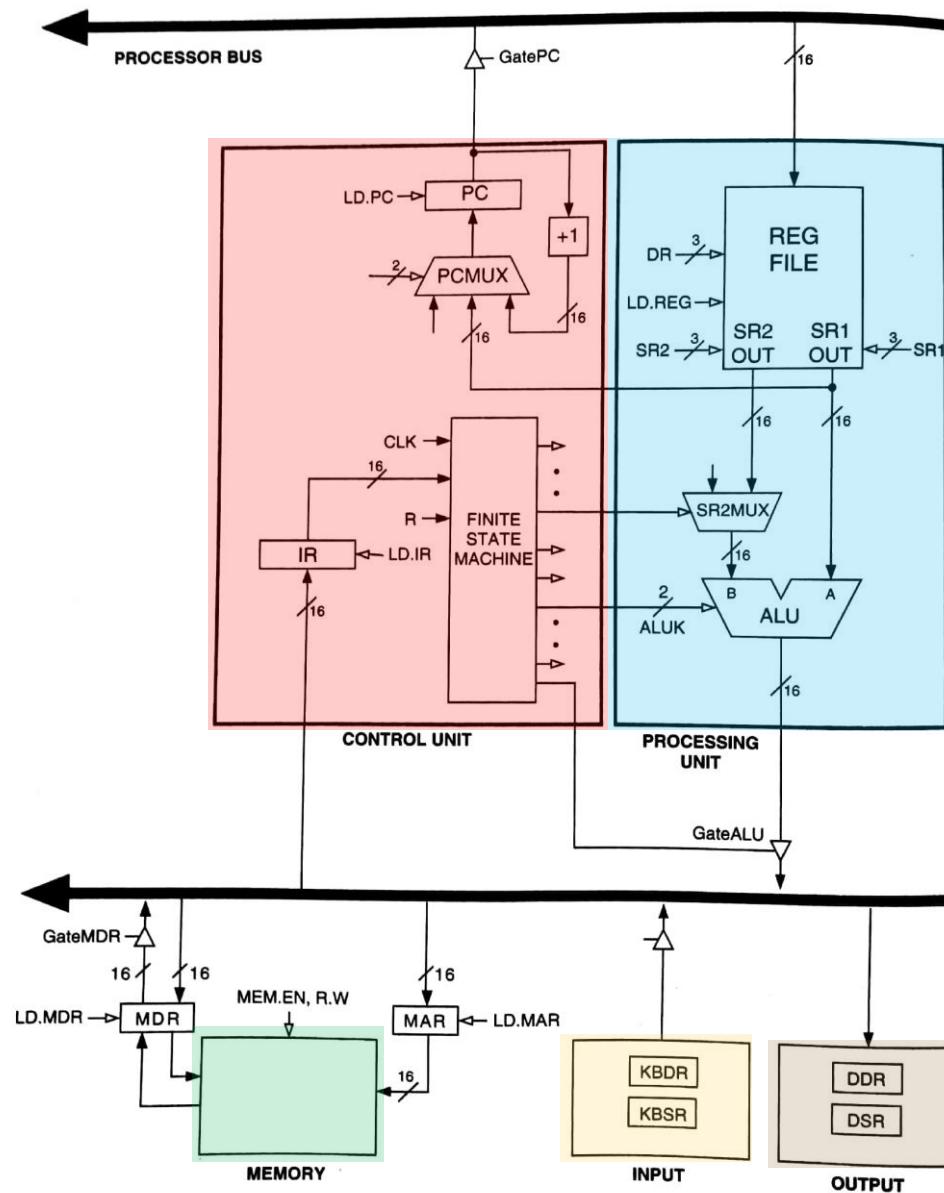
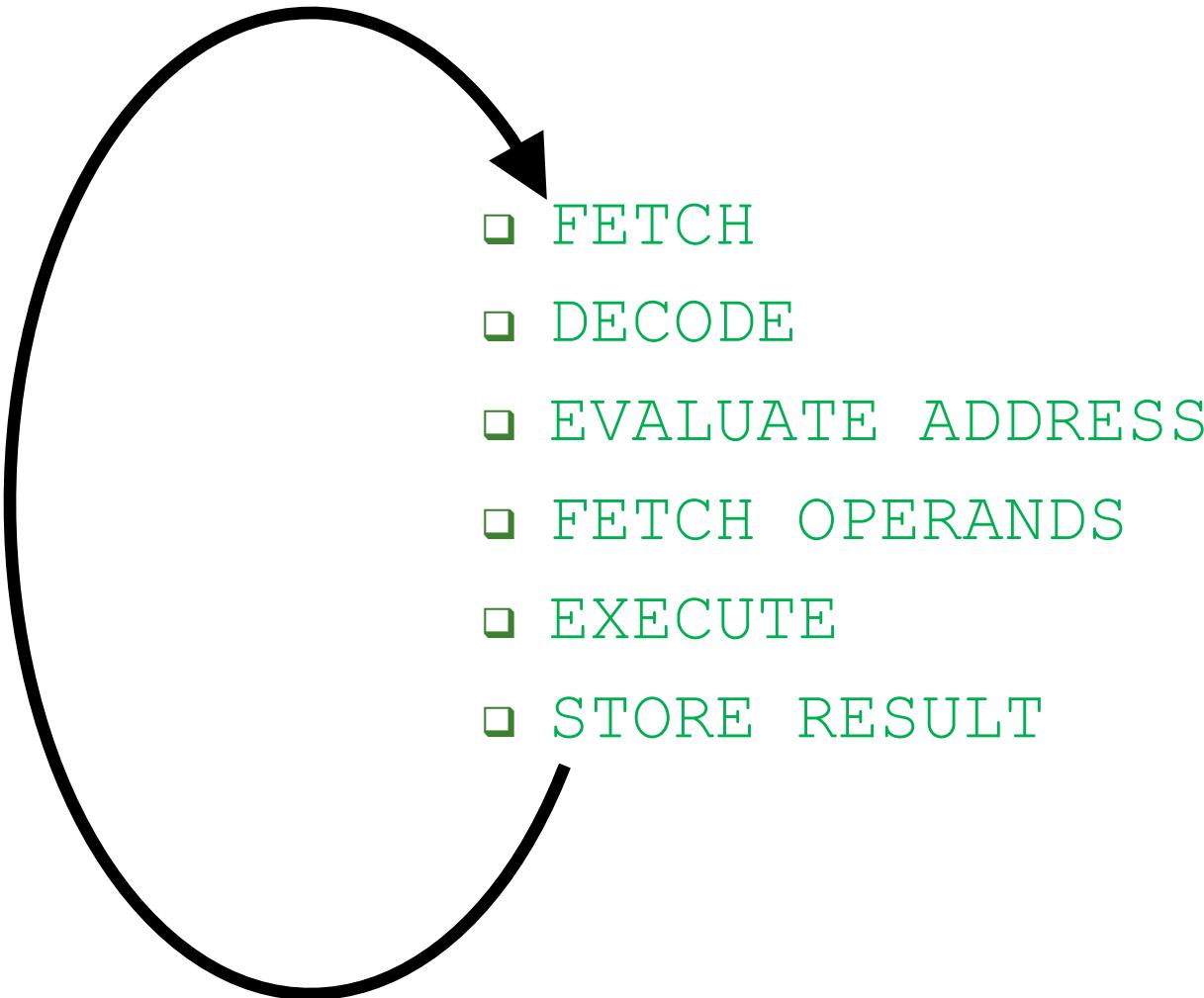


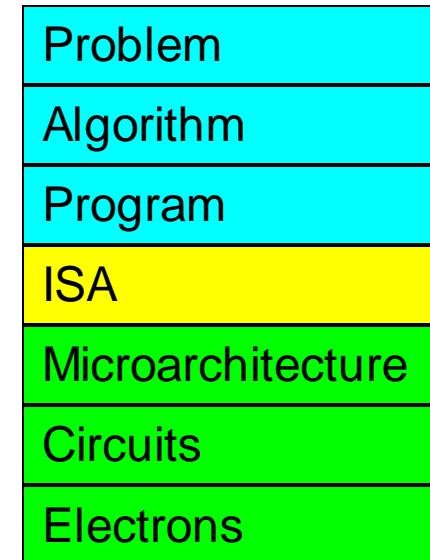
Figure 4.3 The LC-3 as an example of the von Neumann model

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



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, ...
 - ...

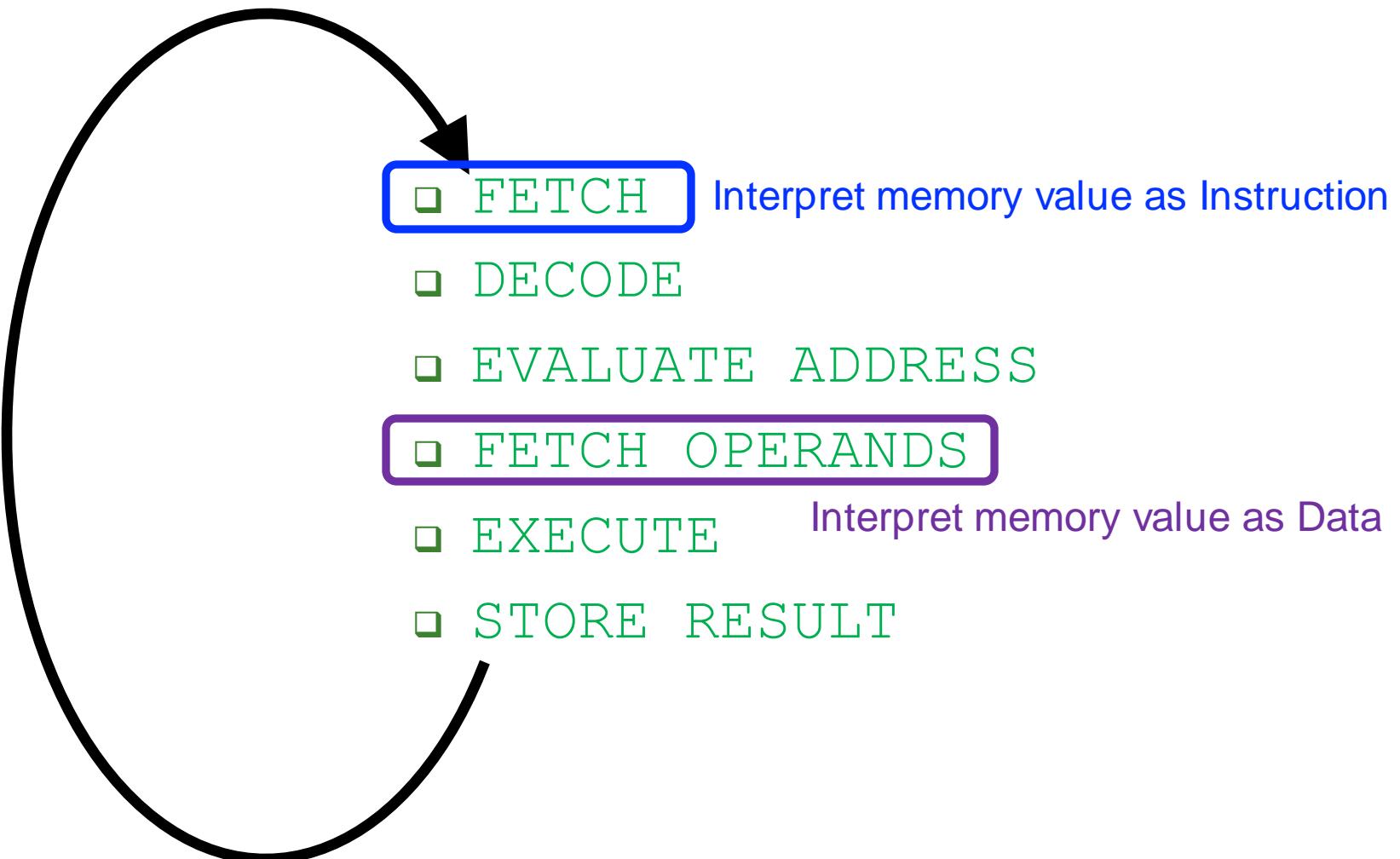
(A Bit More on) ISA Design and Tradeoffs

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**

Harward Architecture ← Stored program
Seperate memory for instructions
 &
 Data

Recall: The Instruction Cycle

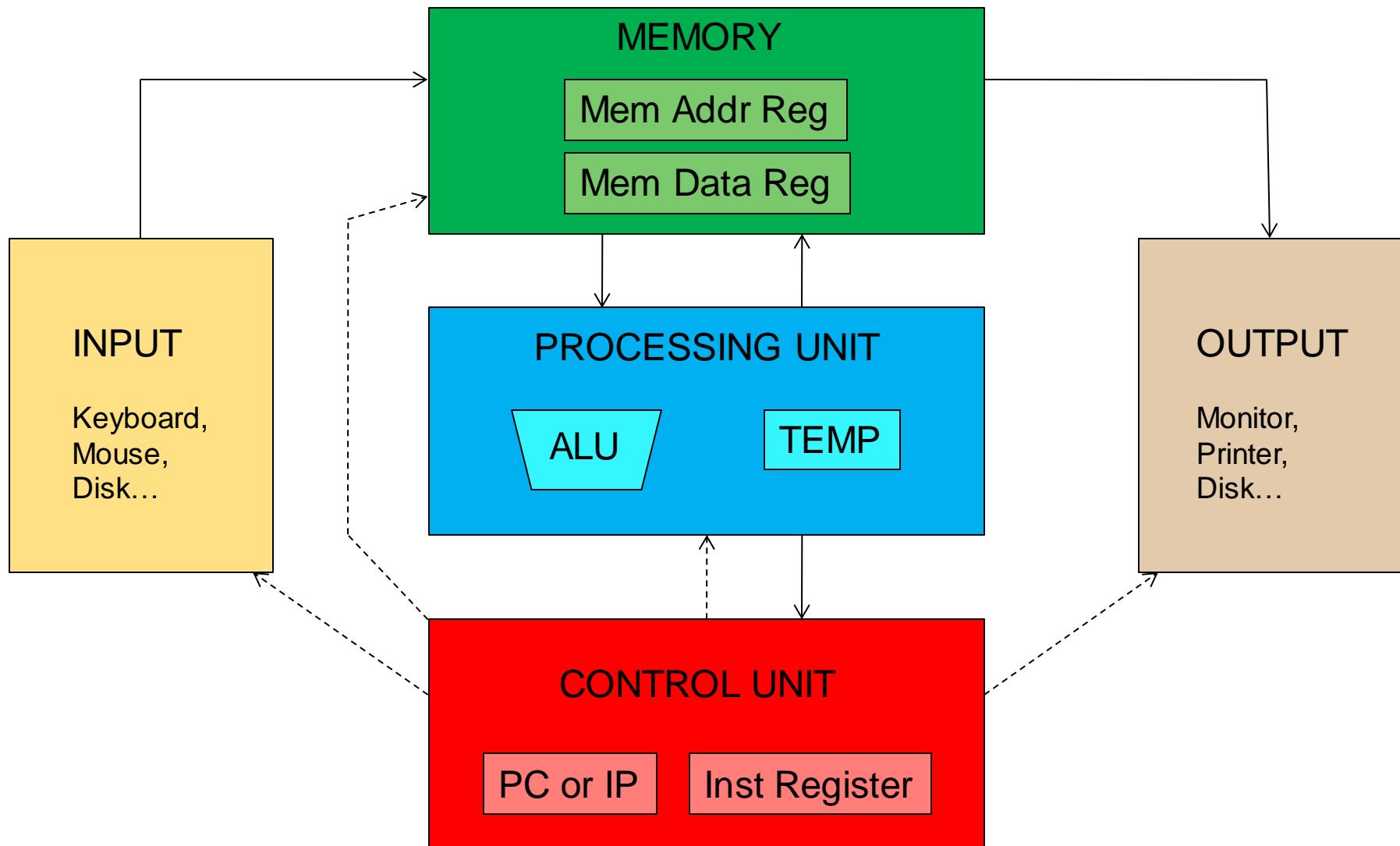


Whether a value fetched from memory is interpreted as an instruction depends on **when** that value is **fetched** in the instruction processing cycle.

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 (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** ← Has an specific flow
 - 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** ← Doesn't matter the order as long as the result is correct.
 - i.e., when its operands are ready ← then we can execute the instruction
 - 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

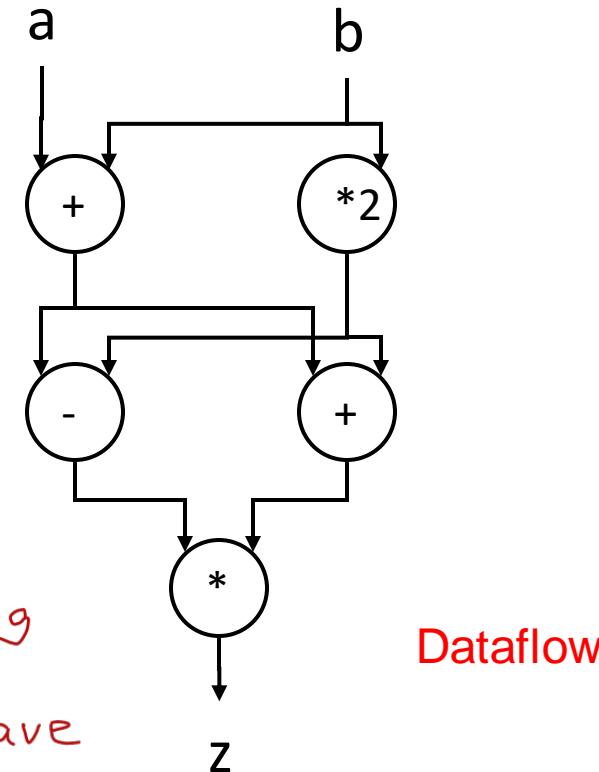
- Consider a Von Neumann program
 - What is the significance of the program order?
 - What is the significance of the storage locations?

$v \leq a + b;$ ← can be done at once
 $w \leq b * 2;$ ←
 $x \leq v - w$ ← dependent on v & w
 $y \leq v + w$
 $z \leq x * y$

Sequential

When implementing it looks like the instructions are running sequentially.

Inside the instruction it doesn't have to be sequential



Dataflow

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

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

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

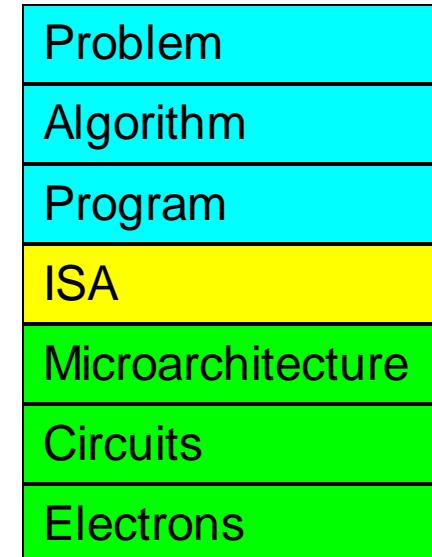
- ❑ Agreed upon interface between software and hardware
 - SW/compiler assumes, HW promises
- ❑ What the software writer needs to know to write and debug system/user programs

■ Microarchitecture

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

■ Microprocessor

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



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?*

ISA

- 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

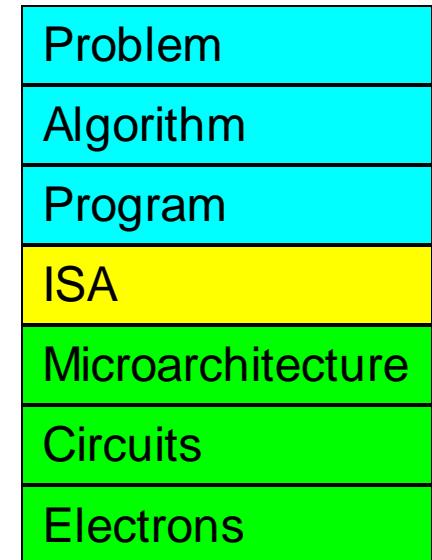
- Implementation of the ISA under specific **design constraints and goals**
- Anything done in hardware without exposure to software
 - Pipelining
 - In-order versus out-of-order instruction execution
 - Memory access scheduling policy
 - Speculative execution
 - Superscalar processing (multiple instruction 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*

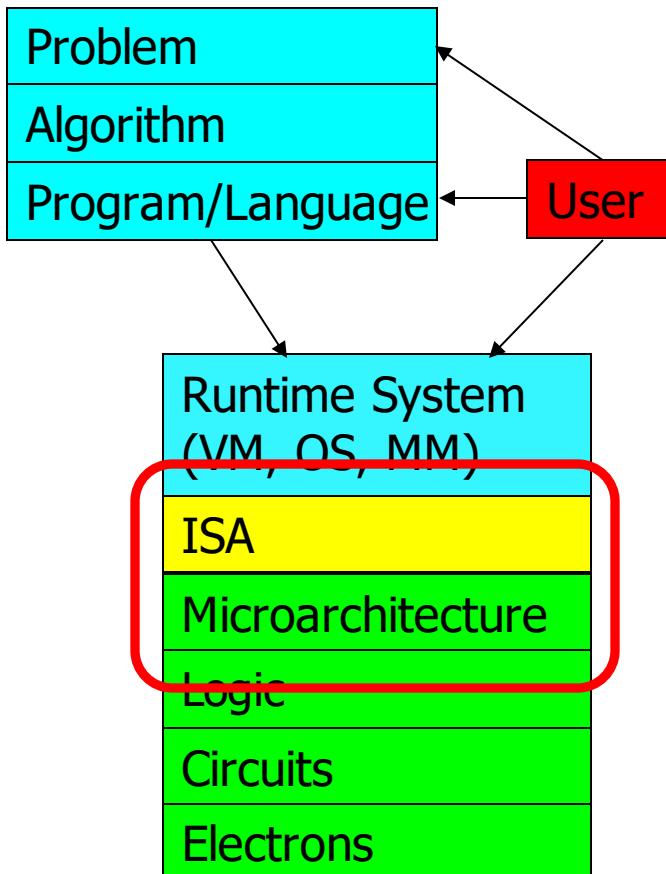


Tradeoffs: Soul of Computer Architecture

- 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)



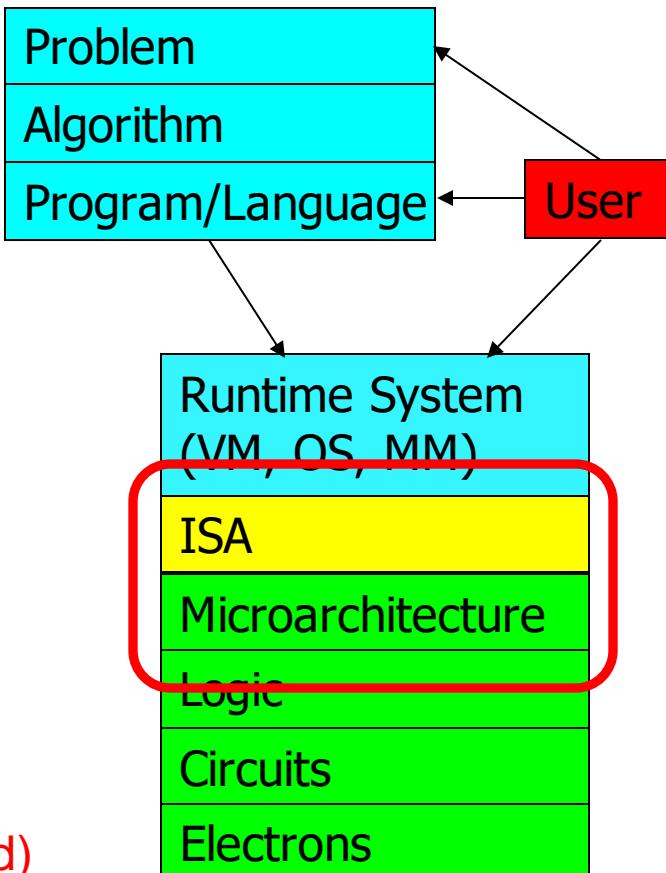
New demands and
personalities of users
(Look Up)

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)!
-

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



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

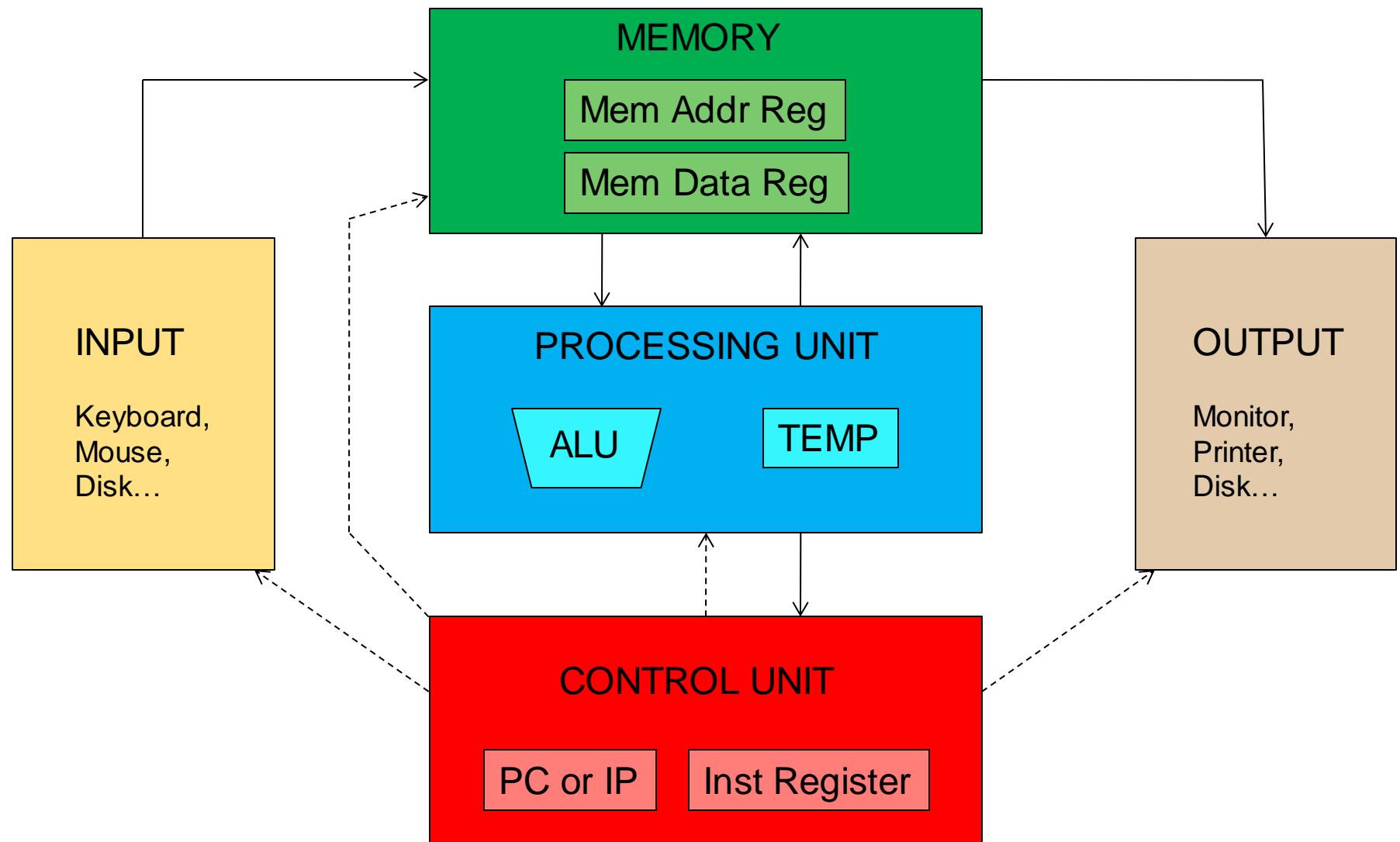
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

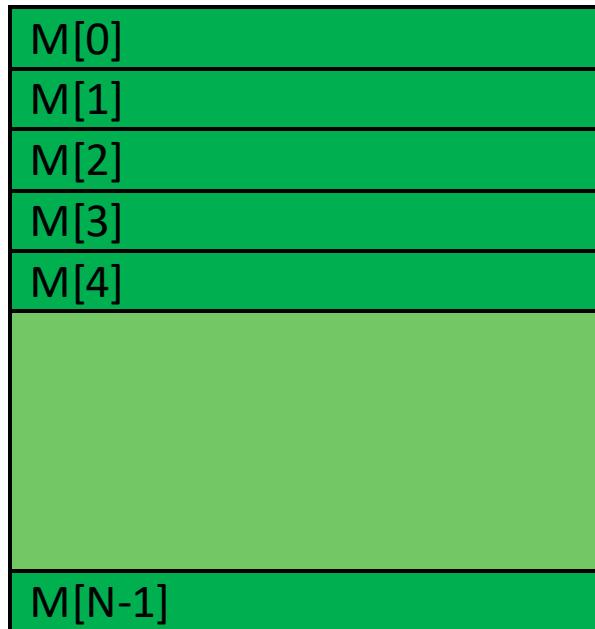
Stored program

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

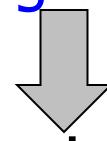
The “Process Instruction” Step

- 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')
-

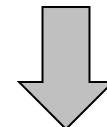
A Very Basic Instruction Processing Engine

- 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



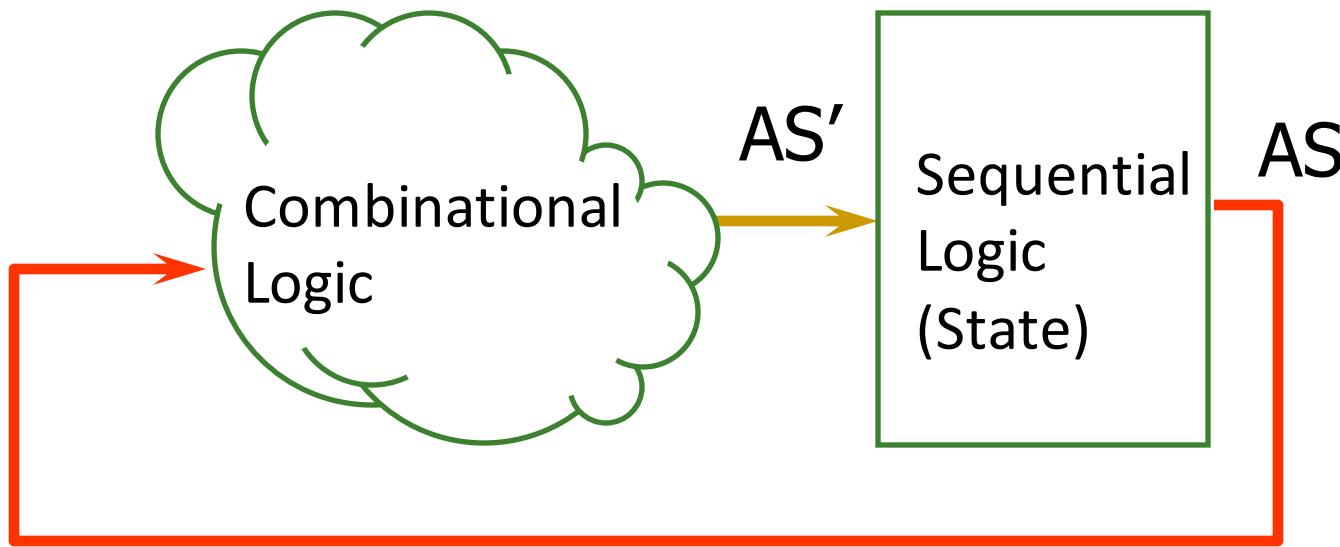
Process instruction in one clock cycle



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

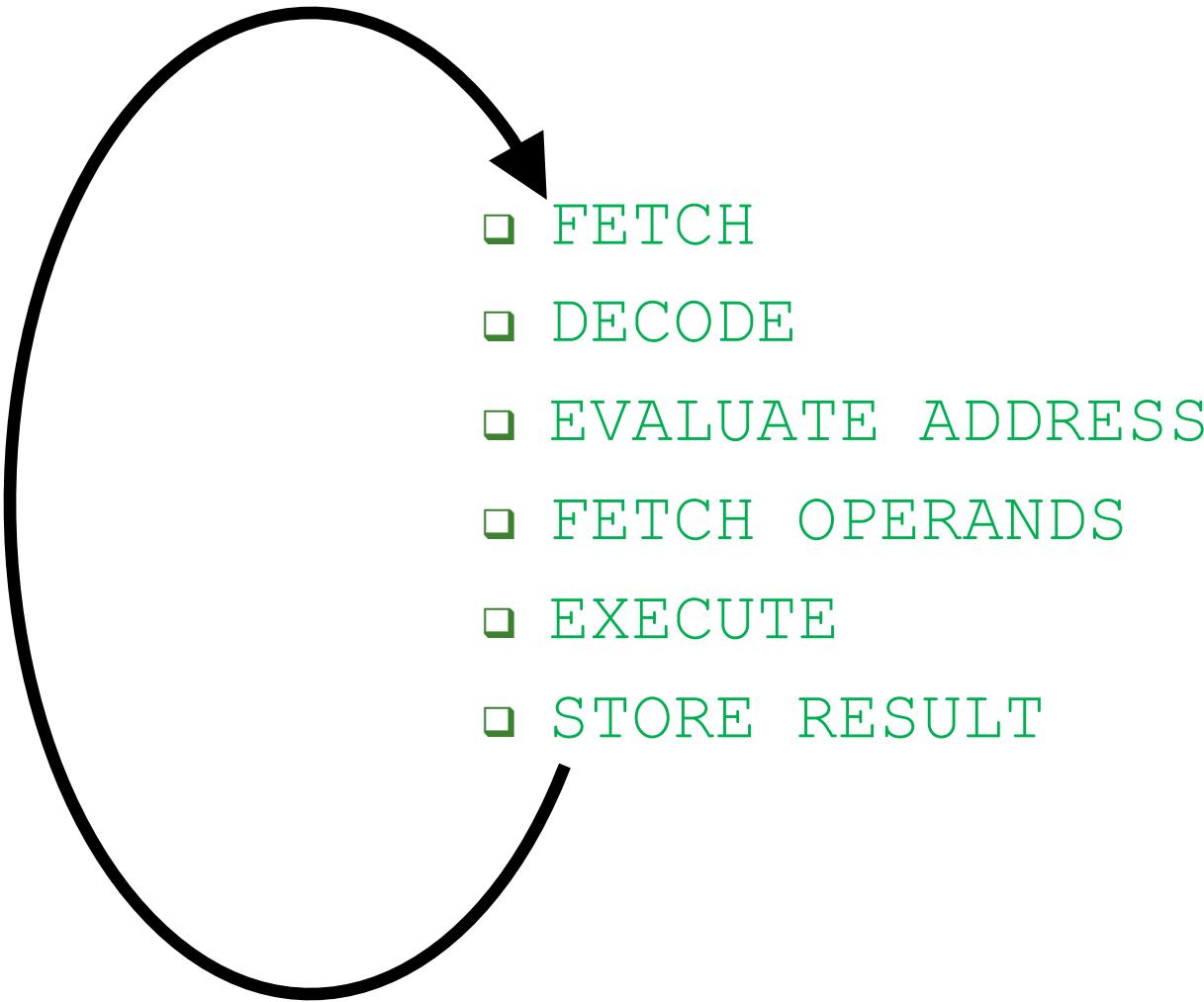
■ Multi-cycle machines

- ❑ 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

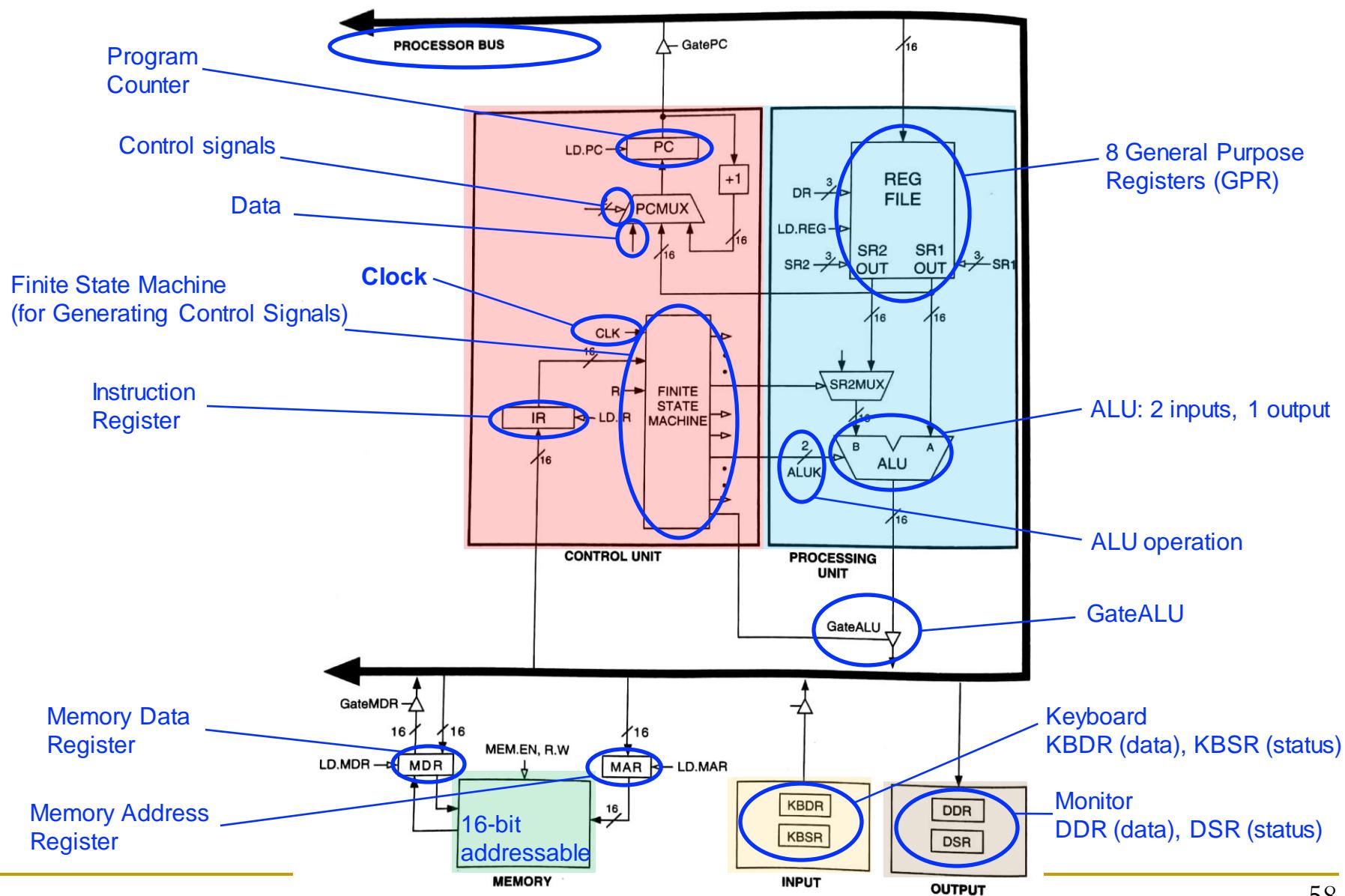


Figure 4.3 The LC-3 as an example of the von Neumann model

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

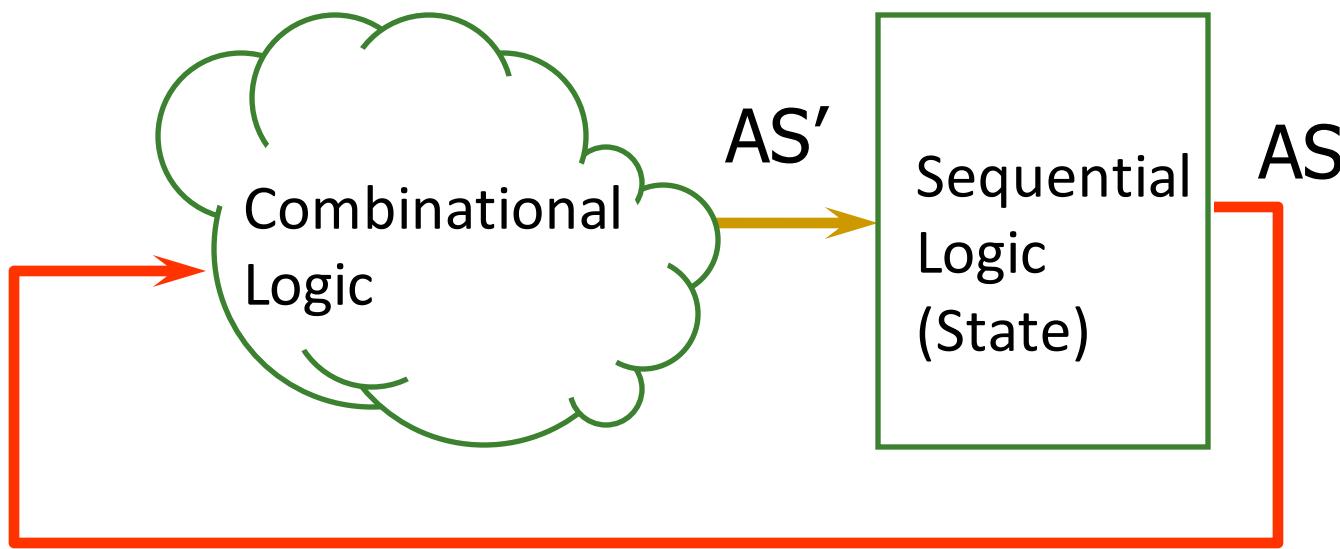
In multi-cycle, we have
two degrees of freedom
to optimize independently

A Single-Cycle Microarchitecture

A Closer Look

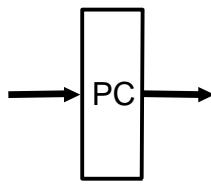
Remember...

- Single-cycle machine

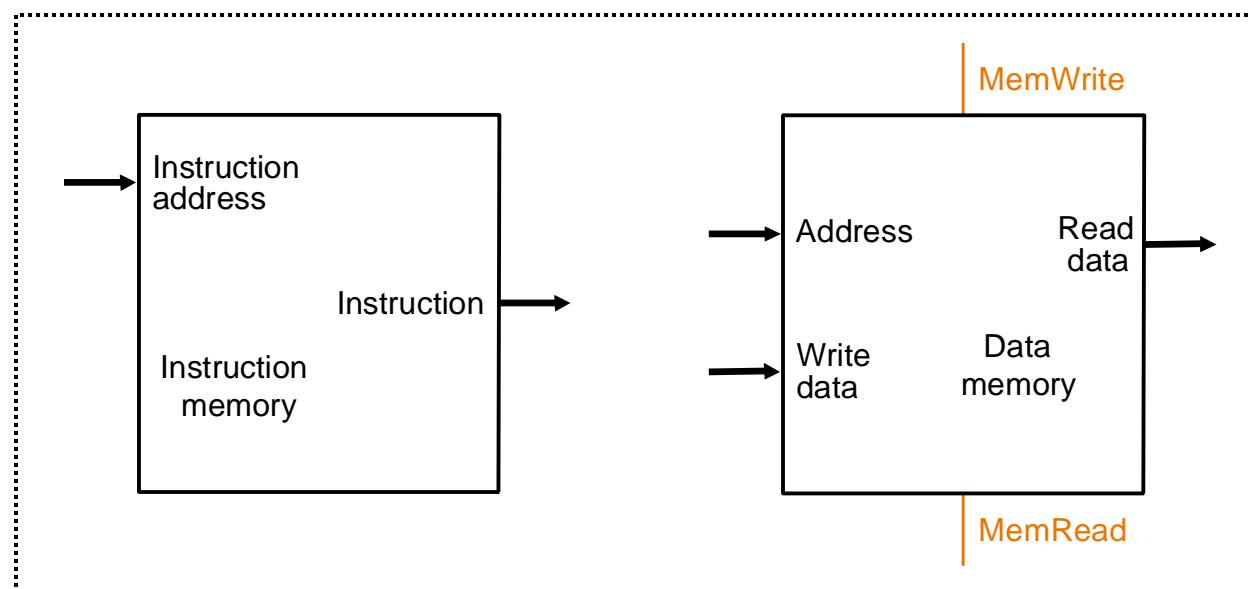
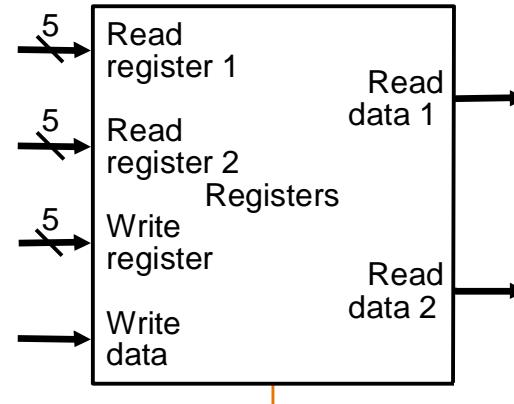


Let's Start with the State Elements

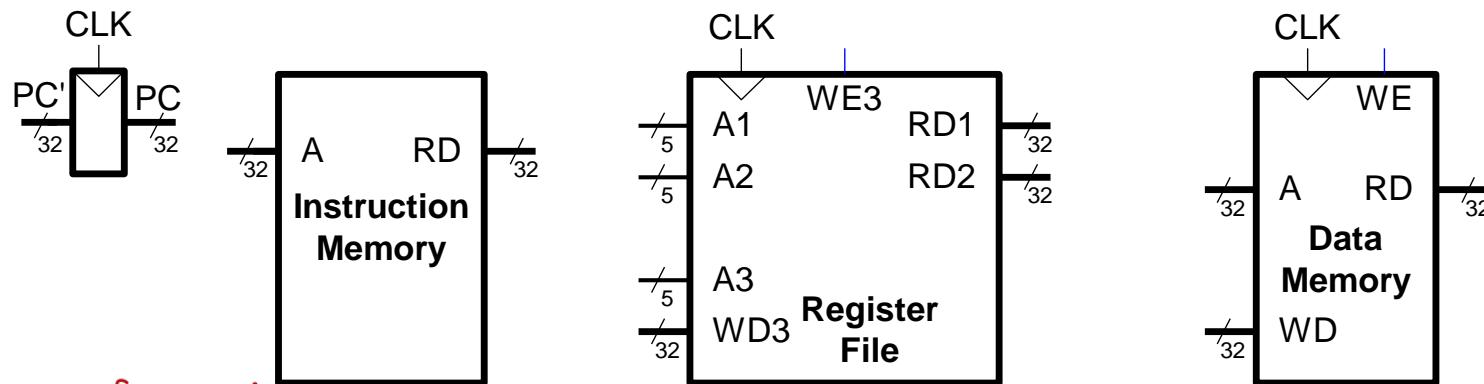
■ Data and control inputs



These are the basic elements to implement first.



MIPS State Elements



Function of each element

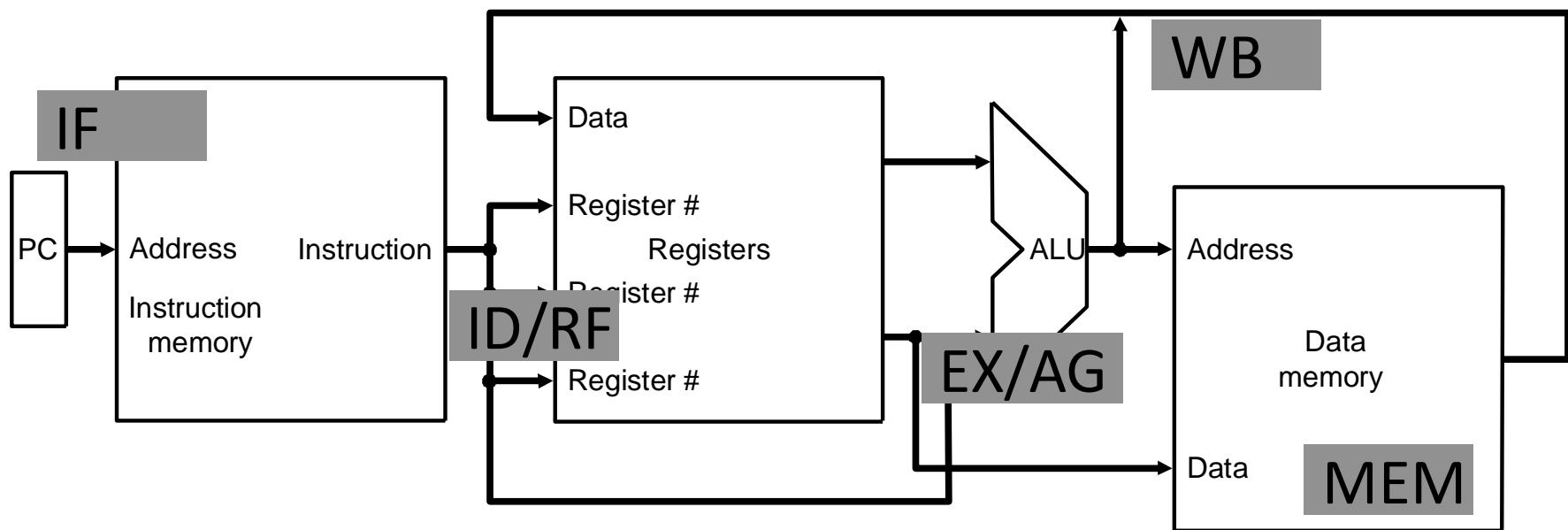
- **Program counter:**
32-bit register
- **Instruction memory:**
Takes input 32-bit address A and reads the 32-bit data (i.e., instruction) from that address to the read data output RD.
- **Register file:**
The 32-element, 32-bit register file has 2 read ports and 1 write port
- **Data memory:**
If the 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

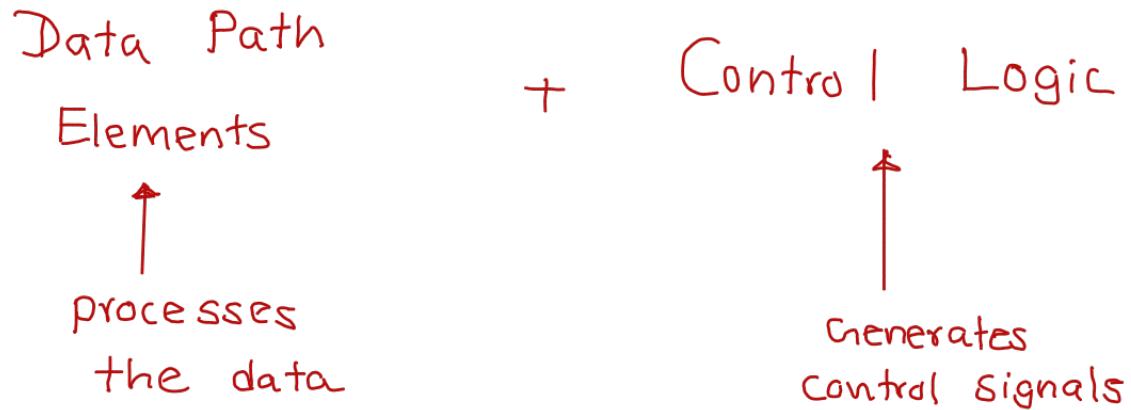
- “Magic” memory and register file
- Combinational read
 - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
 - the selected register is updated on the positive edge clock transition when write enable is asserted
 - 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

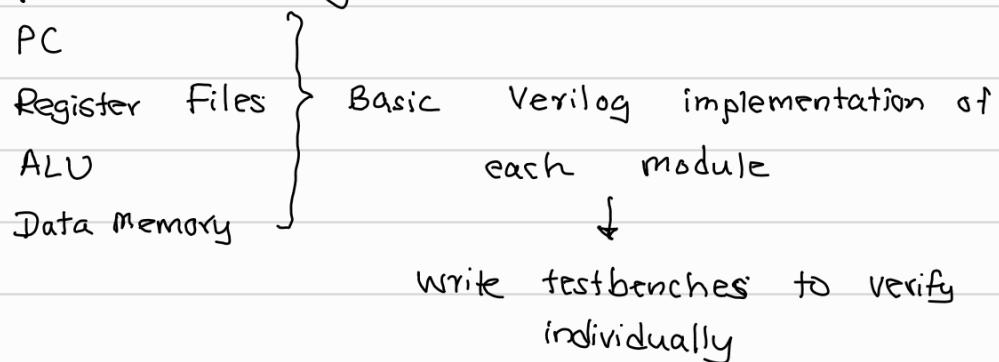


Design Flow

① Know the basics ← • RISC V ISA

- Instruction types and how each instruction behaves

② Implement the basic elements in the data path individually!



③ Test small parts of the data path with manual control.

④ Construct a basic data path for a simple instruction type (R Type)

Test and verify.

Add more elements to the data path for new instruction types

keep testing for each instruction.

⑤ Implement the controller.

Identify the control signals for each instruction.
(note down in a table properly document)

⑥ Construct a program → convert to µ instructions
Run the instructions and test the result
Debug!

Simulation Level

FPGA Implementation

- ⑦ Connect additional modules

UART module \leftarrow to communicate with the computer

Switches

LEDs

↑

give inputs take outputs

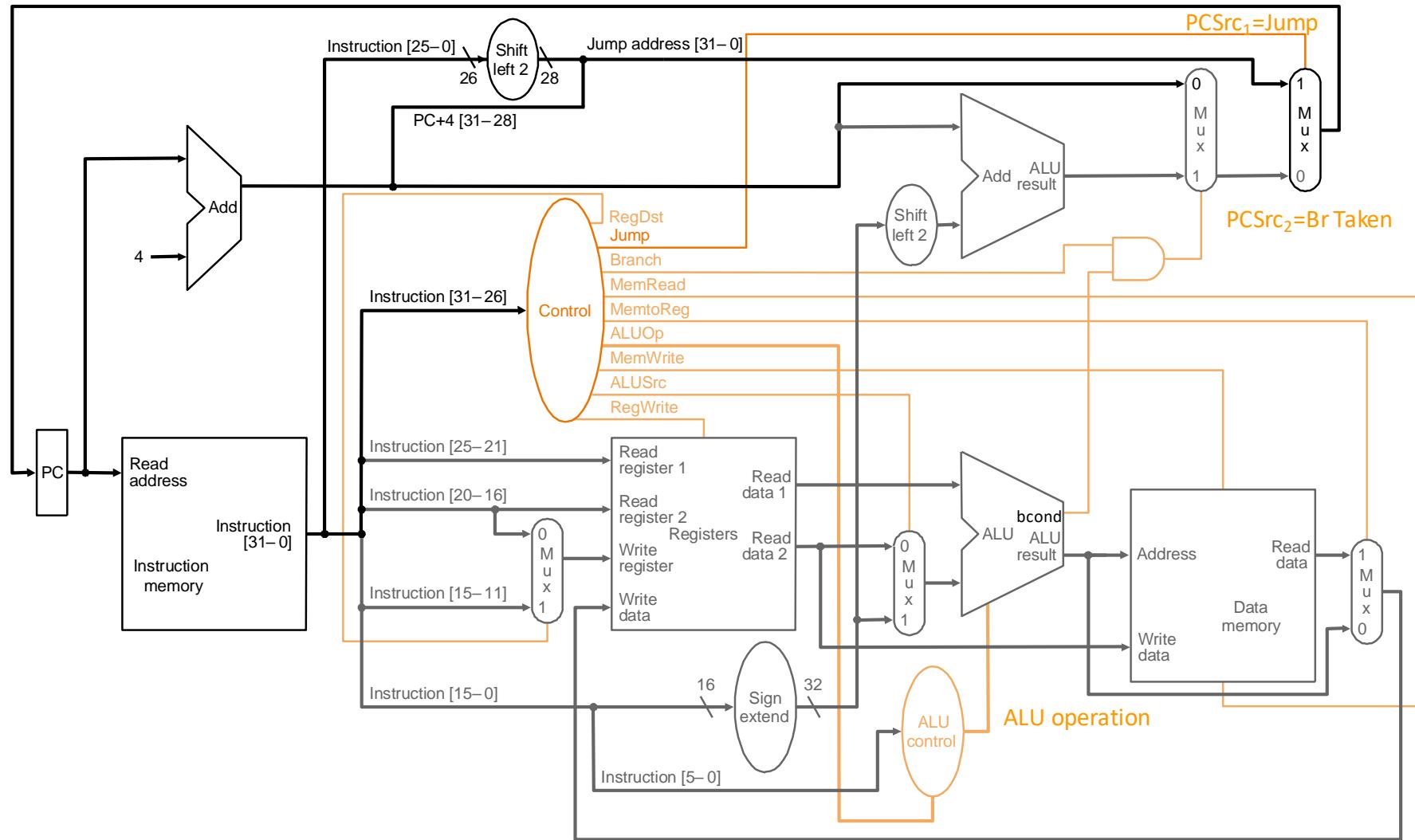
- ⑧ Create memory structures to store program + data

- ⑨ Integrate everything to the design

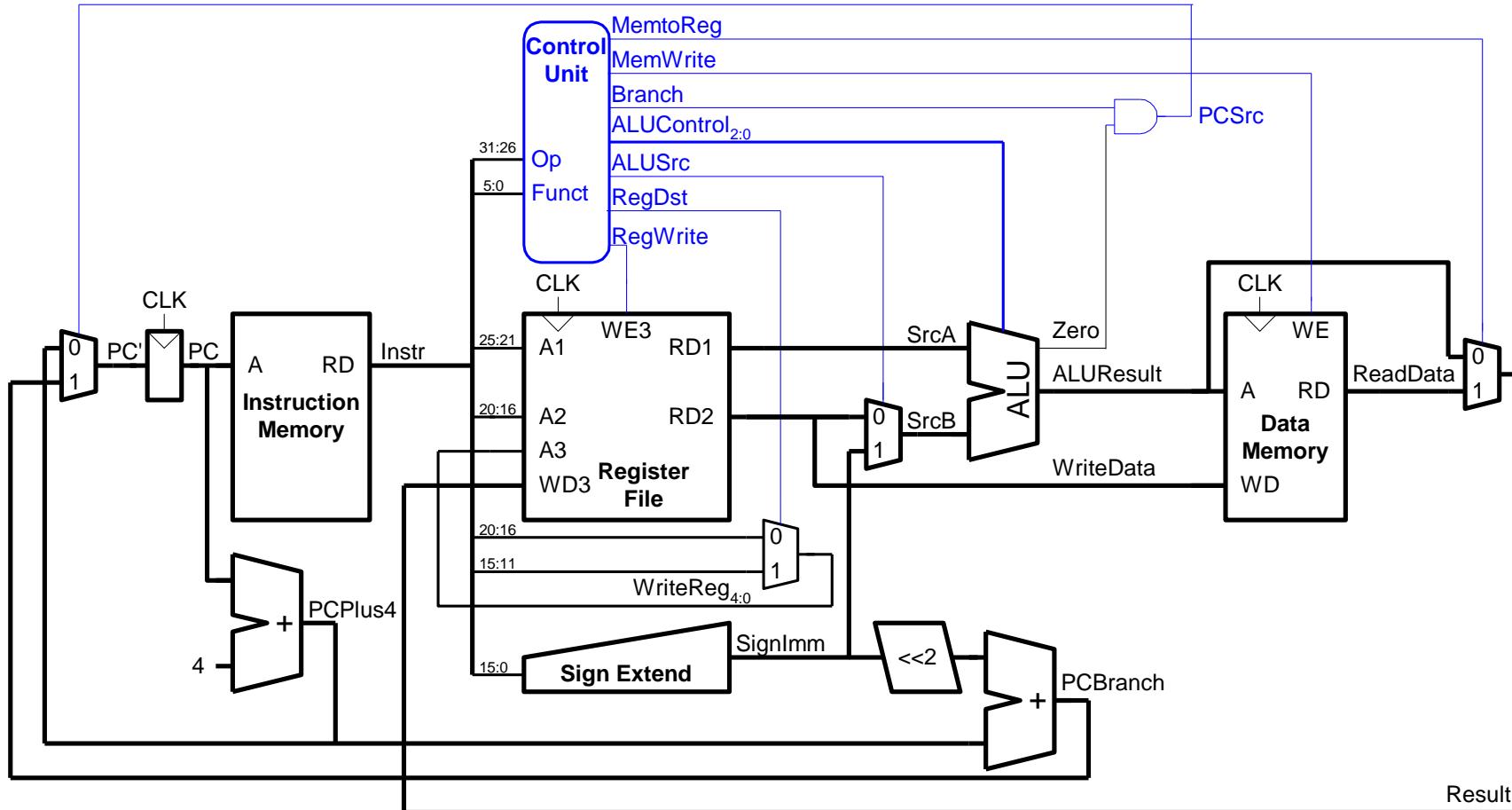
- ⑩ Debug

- ⑪ Deployment.

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
```

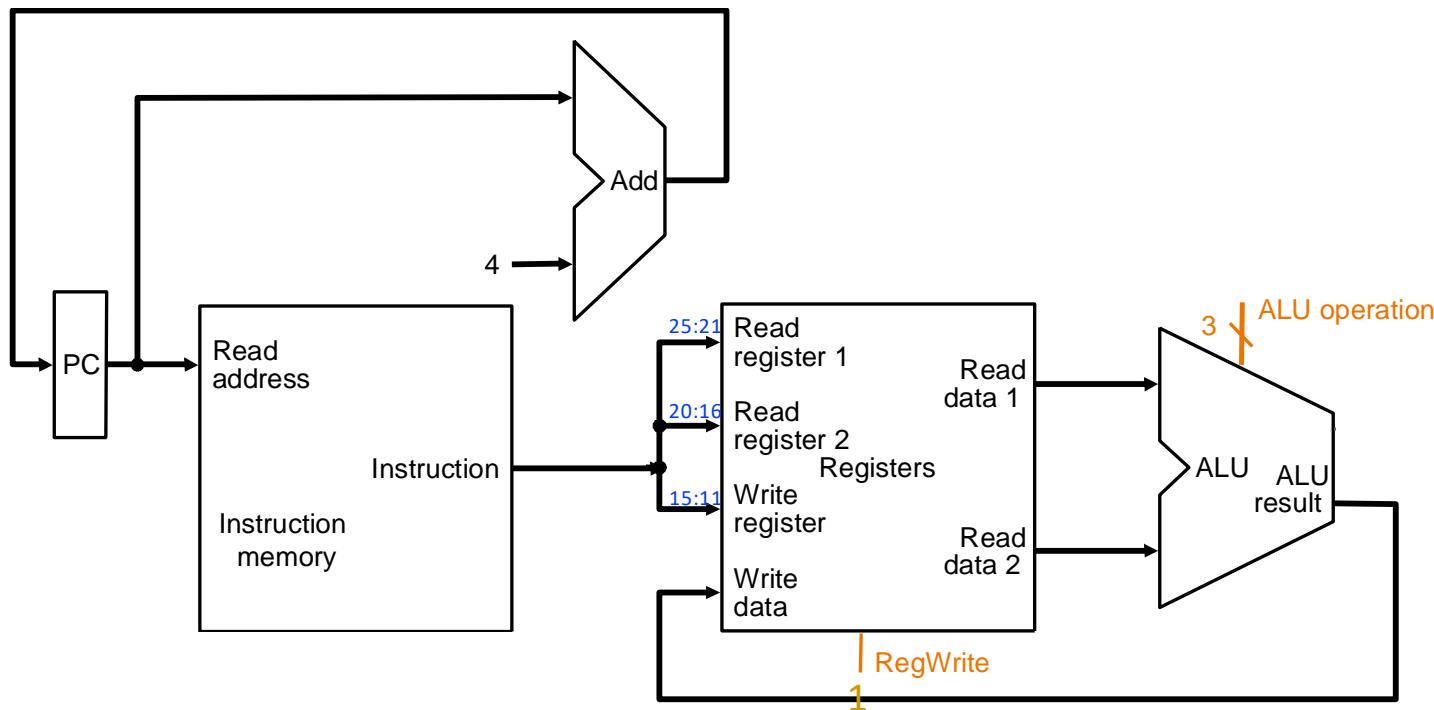
Machine Encoding

0	rs	rt	rd	0	add (32)	R-Type
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

- Semantics

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

(R-Type) ALU Datapath



if $MEM[PC] == ADD\ rd\ rs\ rt$

$GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
 $PC \leftarrow PC + 4$

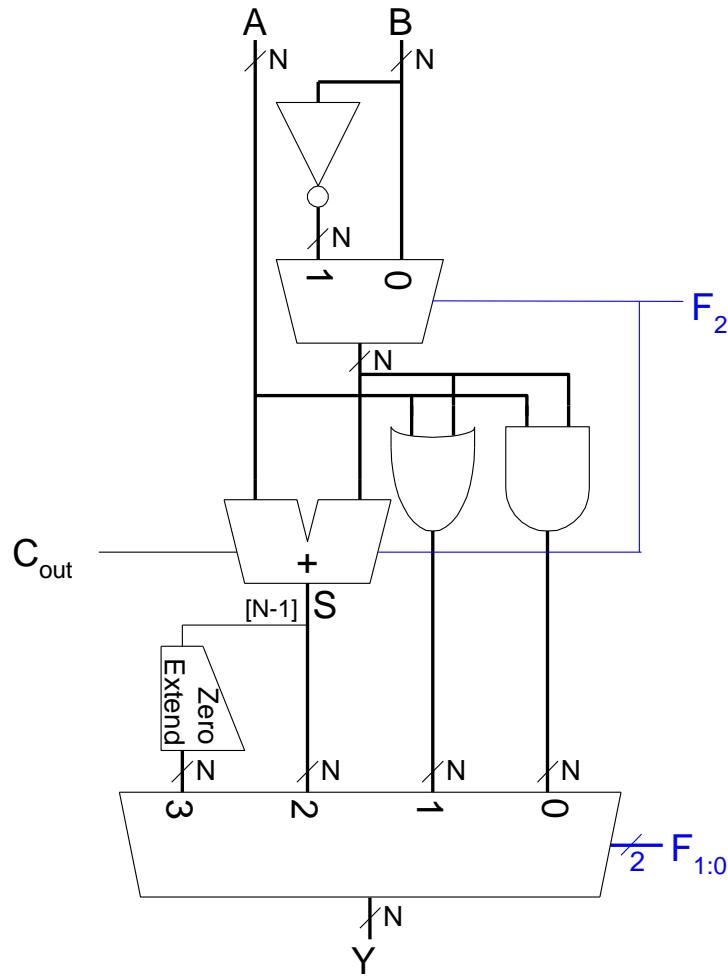
IF ID EX MEM WB



Combinational
state update logic

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	I-Type
6 bits	5 bits	5 bits	16 bits	

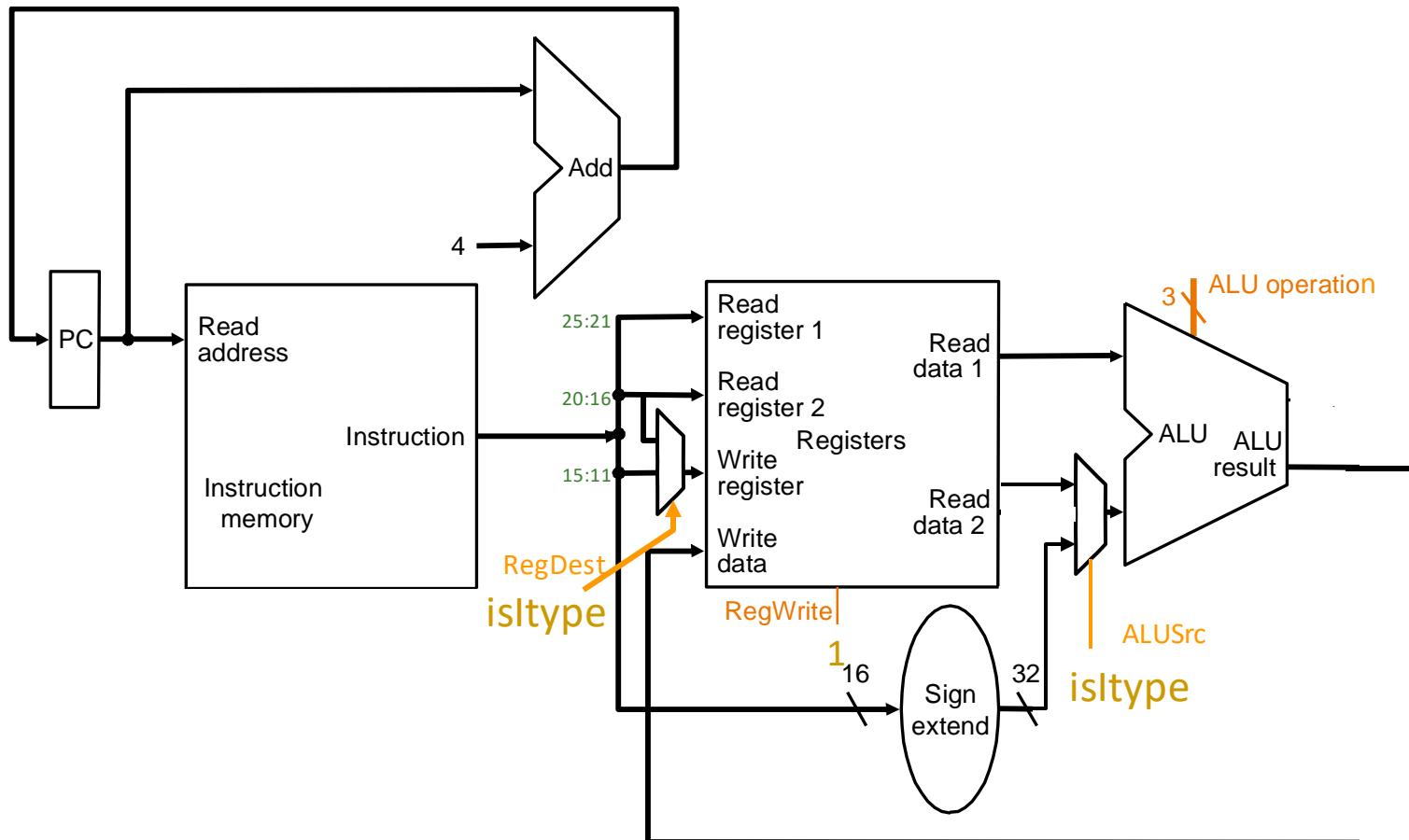
Semantics

if $\text{MEM}[\text{PC}] == \text{addi } \text{rs } \text{rt } \text{immediate}$

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

$\text{GPR}[\text{rt}] \leftarrow \text{GPR}[\text{rs}] + \text{sign-extend}(\text{immediate})$

Datapath for R- and I-Type ALU Insts.



if $MEM[PC] == ADDI\ rt\ rs\ immediate$

$GPR[rt] \leftarrow GPR[rs] + \text{sign-extend (immediate)}$

$PC \leftarrow PC + 4$



Combinational state update logic

Recall: ADD with one Literal in LC-3

■ ADD assembly and machine code

LC-3 assembly

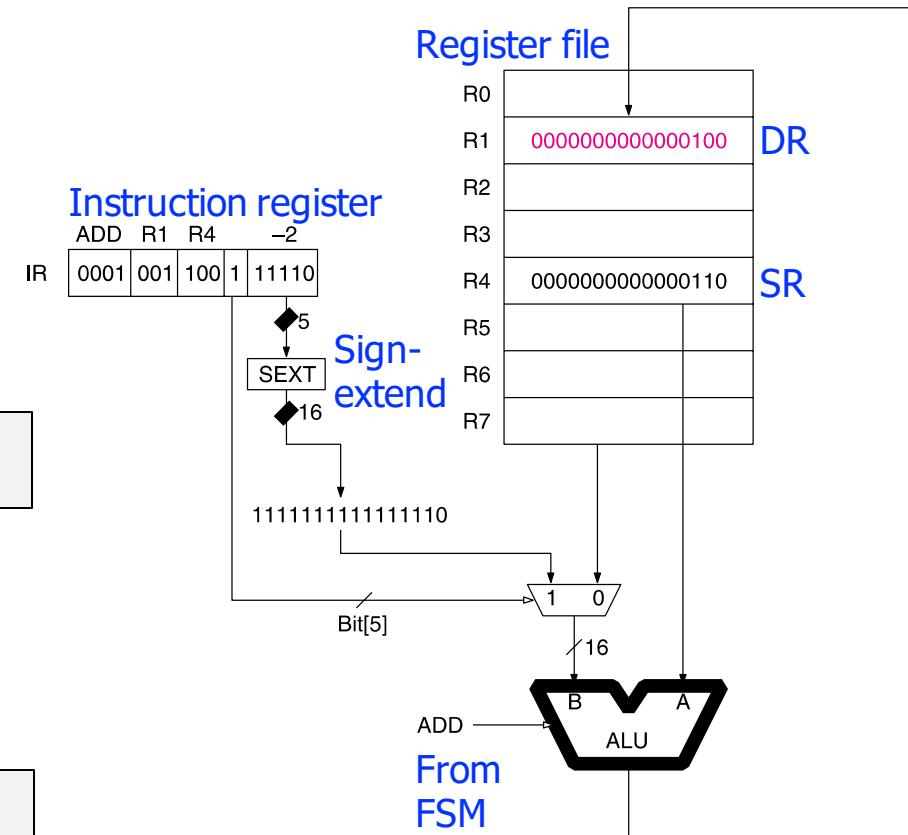
```
ADD R1, R4, #-2
```

Field Values

OP	DR	SR	imm5	
1	1	4	1	-2

Machine Code

OP	DR	SR	imm5	
0 0 0 1	0 0 1	1 0 0	1	1 1 1 1 0
15	12	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

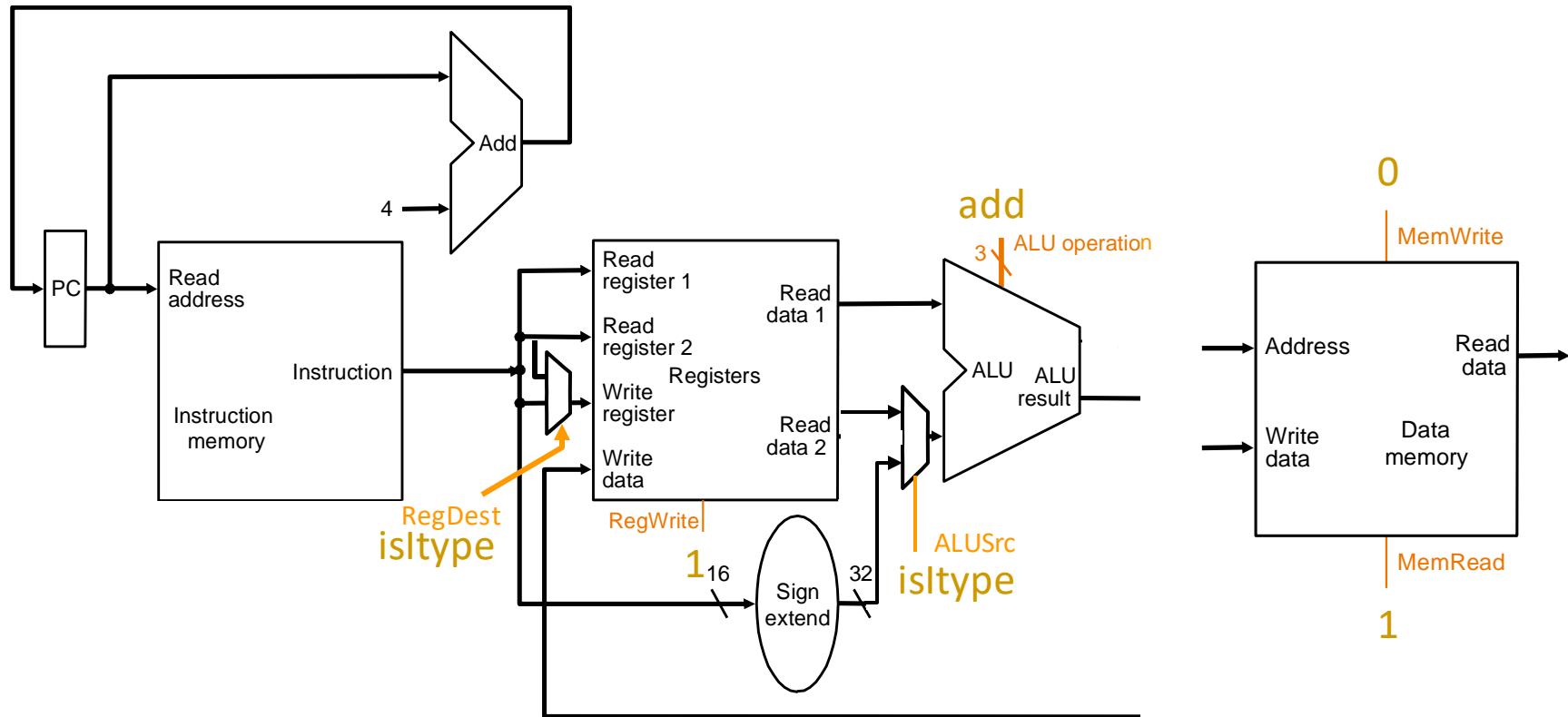
op	rs=base	rt	imm=offset	I-Type
lw (35)	base	rt	offset	

31 26 25 21 20 16 15 0

■ Semantics

```
if MEM[PC] == lw rt offset16 (base)
    PC ← PC + 4
    EA = sign-extend(offset) + GPR(base)
    GPR[rt] ← MEM[ translate(EA) ]
```

LW Datapath



if $MEM[PC] == LW$ rt offset₁₆ (base)
 $EA = \text{sign-extend}(\text{offset}) + GPR[\text{base}]$
 $GPR[\text{rt}] \leftarrow \text{MEM}[\text{translate}(EA)]$
 $PC \leftarrow PC + 4$



Combinational state update logic 80

Store Instructions

- Store 4-byte word

MIPS assembly

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

Machine Encoding

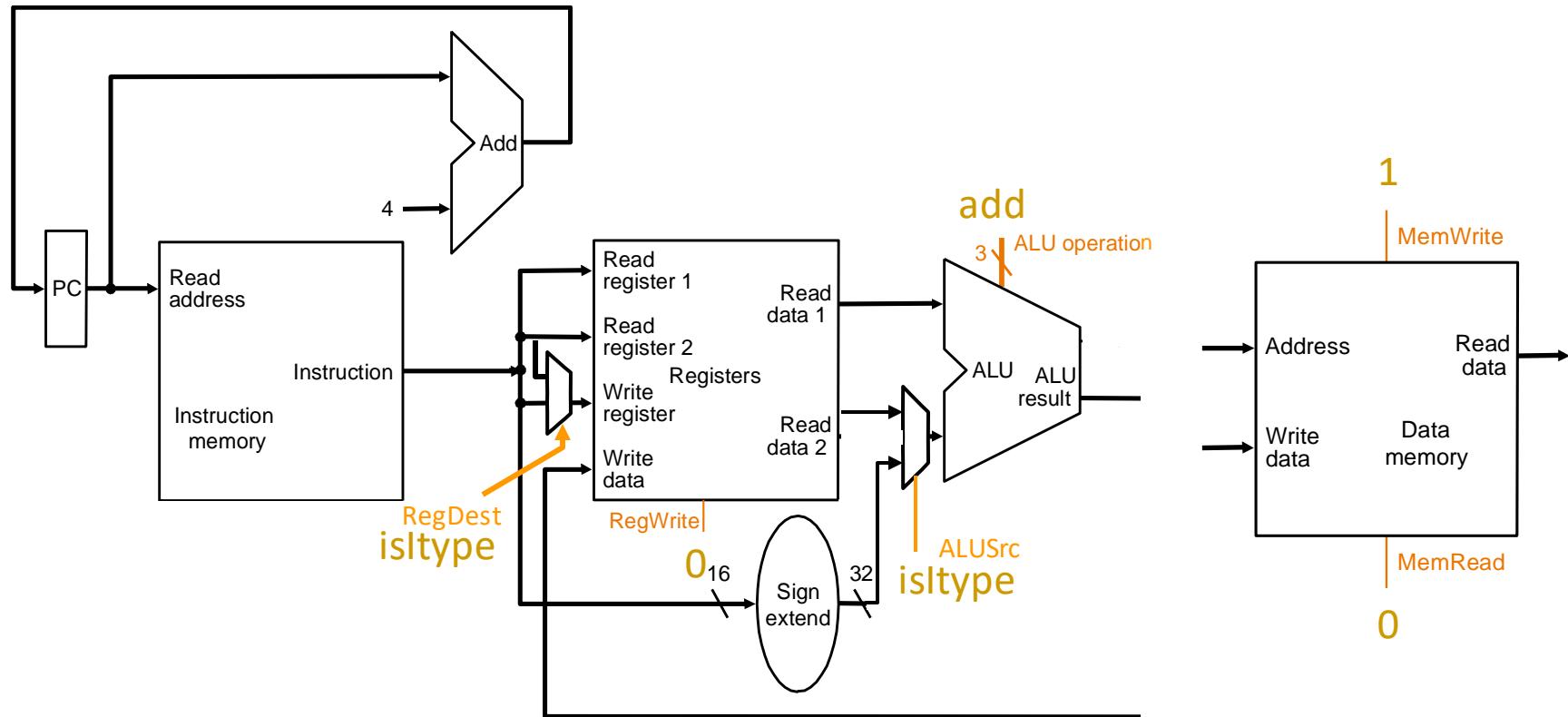
op	rs=base	rt	imm=offset	
sw (43)	base	rt	offset	0

I-Type

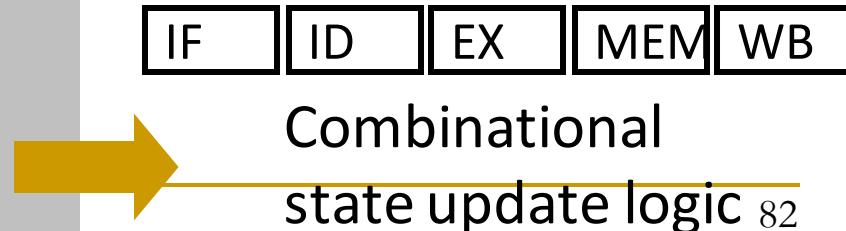
- Semantics

```
if Mem[PC] == sw rt offset16 (base)
  PC ← PC + 4
  EA = sign-extend(offset) + GPR(base)
  MEM[ translate(EA) ] ← GPR[rt]
```

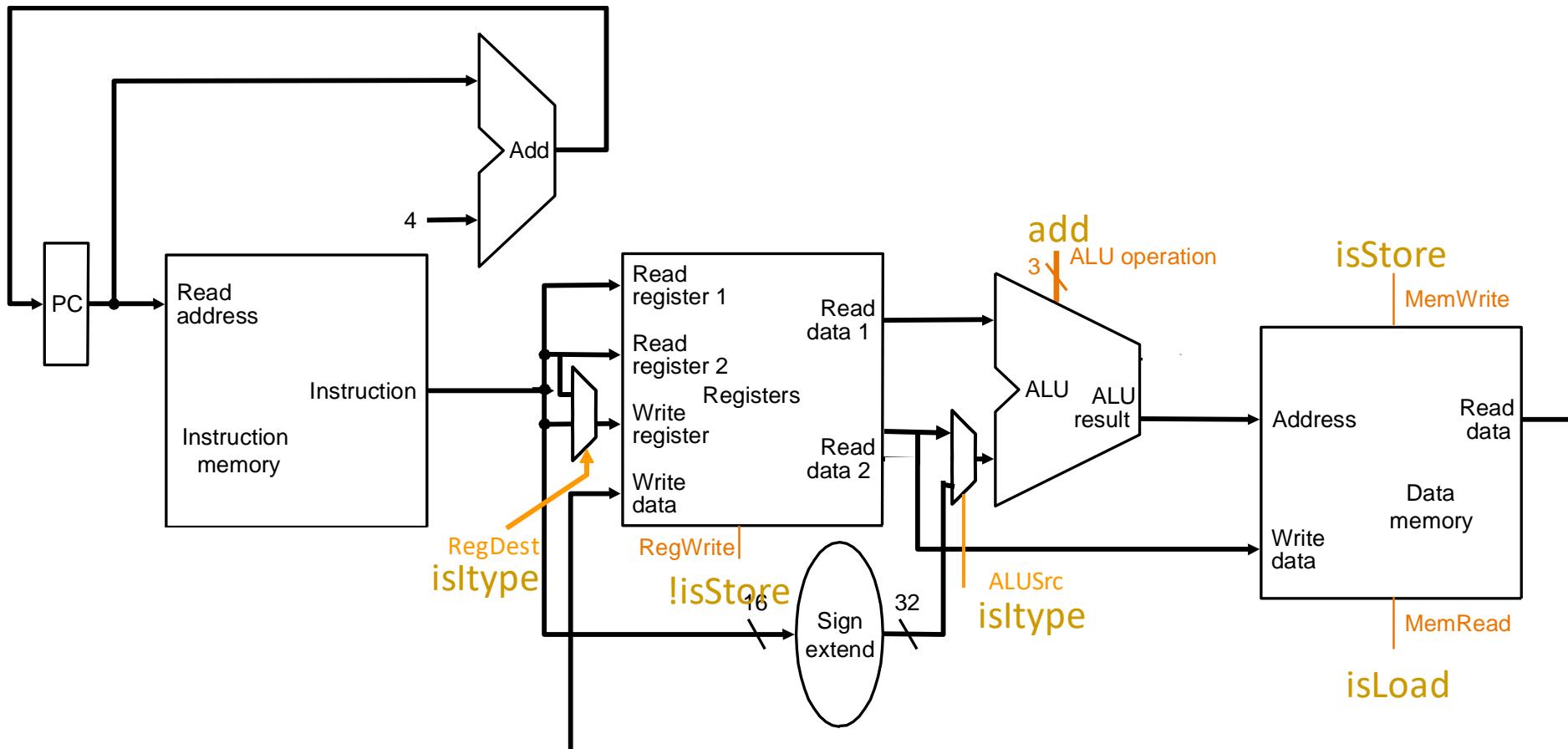
SW Datapath



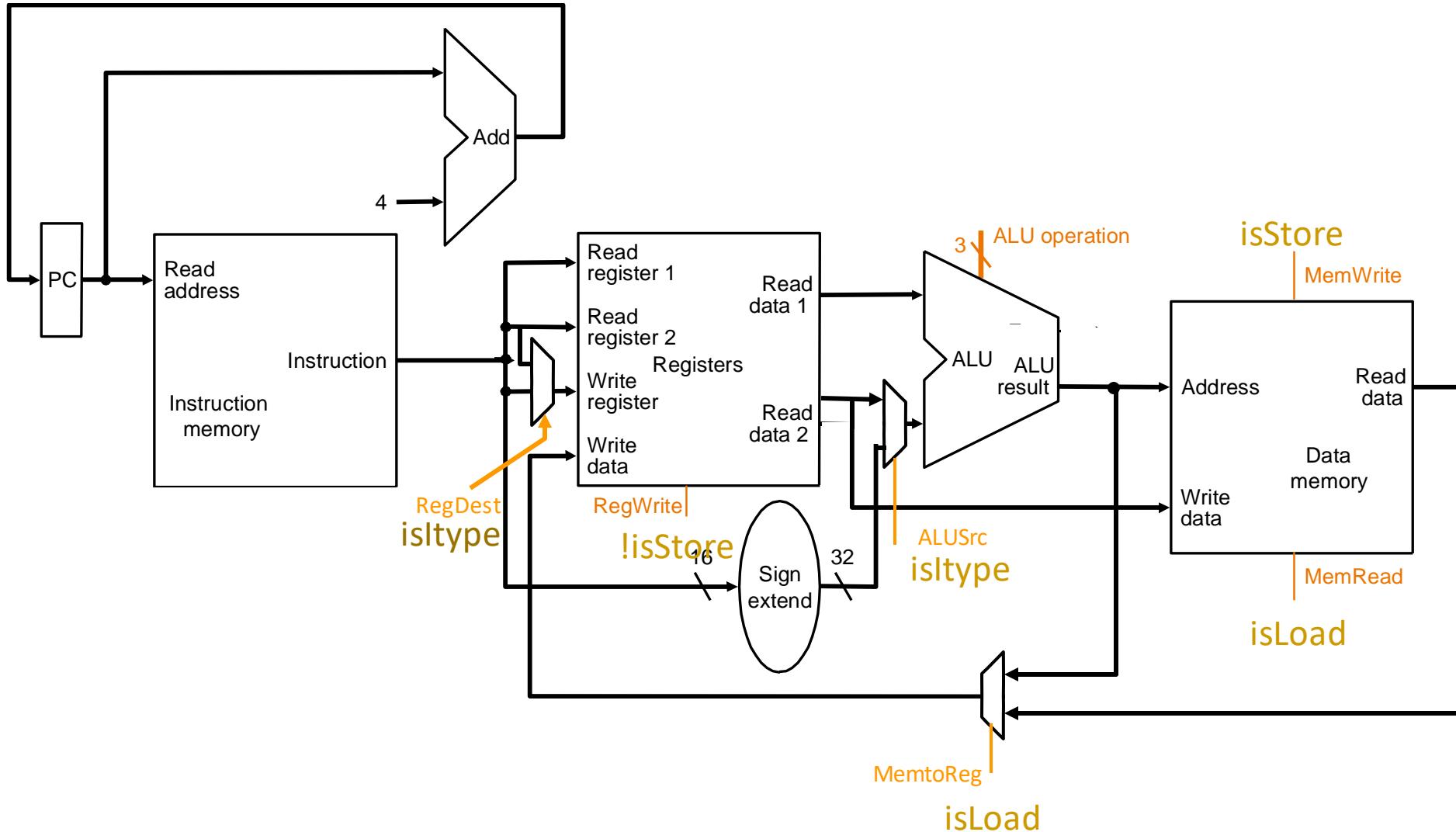
```
if MEM[PC]==SW rt offset16 (base)
    EA = sign-extend(offset) + GPR[base]
    MEM[ translate(EA) ] ← GPR[rt]
    PC ← PC + 4
```



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 (2)	immediate	J-Type
6 bits	26 bits	

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

■ Semantics

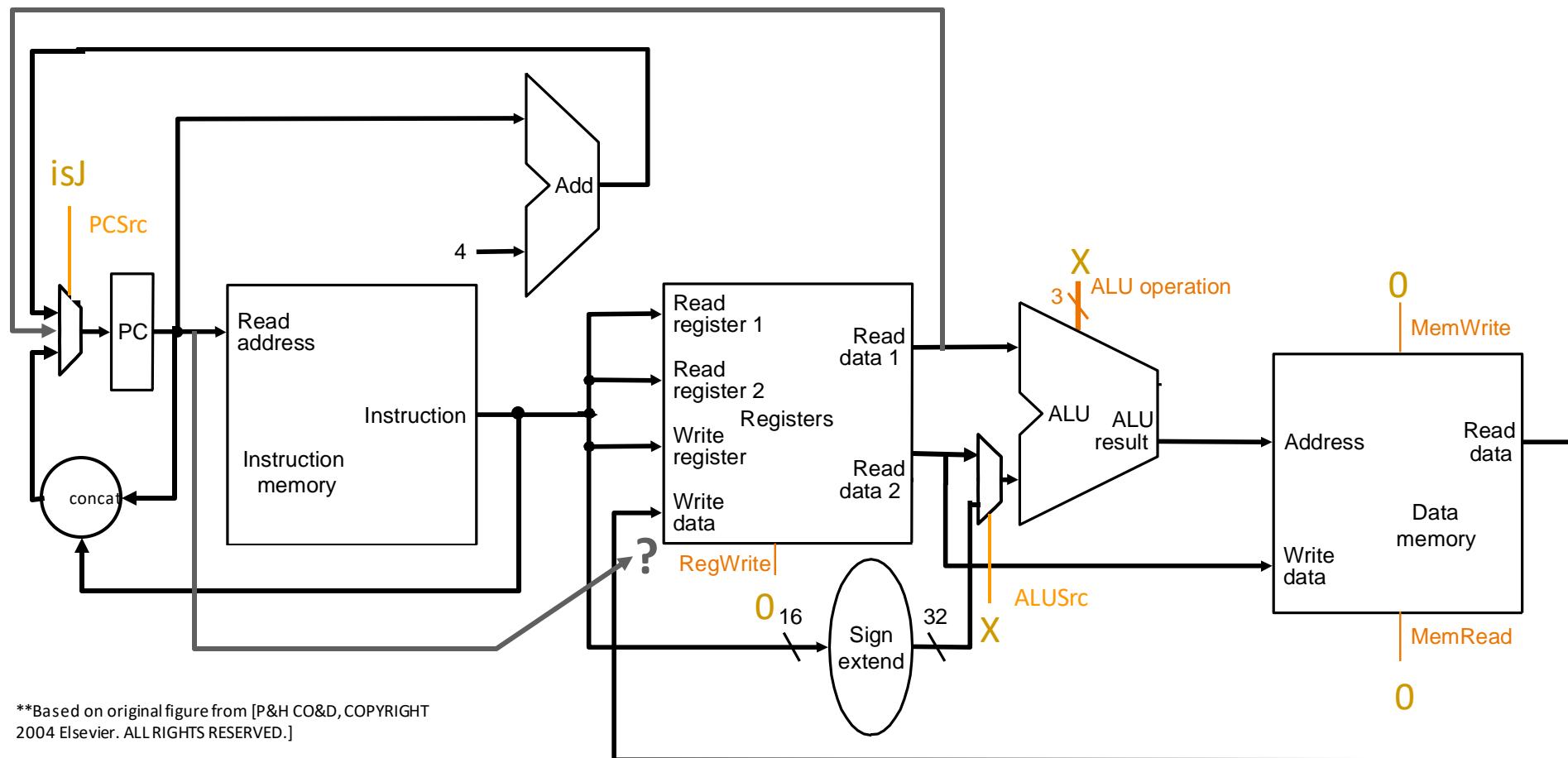
if $\text{MEM}[\text{PC}] == j \text{ immediate}_{26}$

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

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

[†]This is the incremented PC

Unconditional Jump Datapath



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

if $\text{MEM}[\text{PC}] == \text{J}$ immediate26

$\text{PC} = \{ \text{PC}[31:28], \text{immediate26}, 2' \text{ b}00 \}$

What about JR, JAL, JALR?

Other Jumps in MIPS

- jal: jump and link (function calls)

- Semantics

- if $\text{MEM}[\text{PC}] == \text{jal}$ immediate₂₆**

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

- $\text{target} = \{ \text{PC}^+ [31:28], \text{immediate}_{26}, 2' \text{ b}00 \}$

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

- 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

Aside: MIPS Cheat Sheet

■ https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=mips_reference_data.pdf

■ On the course website

MIPS Reference Data			
CORE INSTRUCTION SET			
NAME, MNEMONIC			FOR- MAT
NAME, MNEMONIC			OPERATION (in Verilog)
NAME, MNEMONIC	FOR- MAT	OPERATION (in Verilog)	OPCODE / FMT / FT / FUNCT
Add add	add	$R_1 = R_0 + R_2$	1) 0/20hex
Add Immediate addi	addi	$I_1 = R_0 + R_2 + \text{SignExtImm}$	1,2) 8hex
Add Imm. Unsigned addiu	addiu	$I_1 = R_0 + R_2 + \text{SignExtImm}$	2) 9hex
And and	and	$R_1 = R_0 \& R_2$	0/21hex
And Immediate andi	andi	$I_1 = R_0 \& R_2 + \text{ZeroExtImm}$	3) 9hex
Branch On Equal beq	beq	$I_1 = R_0 == R_2$	4) 4hex
Branch On Not Equal bne	bne	$I_1 = R_0 != R_2$	5) 5hex
Branch On PC True bctrl	bctrl	$I_1 = \text{if}(R_0 == PC) \text{PC} + 4 + \text{BranchAddr}$	4) 11/8/-1
Branch On PC False bctrl	bctrl	$I_1 = \text{if}(R_0 != PC) \text{PC} + 4 + \text{BranchAddr}$	4) 11/8/0
Divide div	div	$R_1 = R_0 / R_2$	11/8/-1
Divide Unsigned divu	divu	$R_1 = R_0 / R_2$	11/10/-0
Divide Add Single add.s	add.s	$R_1 = R_0 + R_2 + \text{SignExtImm}$	11/10/-0
Divide Add Double add.d	add.d	$R_1 = R_0 + R_2 + \text{SignExtImm}$	11/11/-1b
FP Compare Single c.x*	c.x*	$R_1 = \text{FPComp}(F_0, F_1) \& 1$	1) 11/0/-y
FP Compare Double c.x.d*	c.x.d*	$R_1 = \text{FPComp}(F_0, F_1) \& 1$	0/24hex
Double Double	double	$R_1 = R_0 + R_2 + \text{SignExtImm}$	11/11/-y
FP Divide Single div.s	div.s	$R_1 = R_0 / R_2 + \text{SignExtImm}$	11/10/-3
FP Divide Double div.d	div.d	$R_1 = R_0 / R_2 + \text{SignExtImm}$	11/11/-3
FP Multiply Single mul.s	mul.s	$R_1 = R_0 * R_2 + \text{SignExtImm}$	11/10/-2
FP Multiply Double mul.d	mul.d	$R_1 = R_0 * R_2 + \text{SignExtImm}$	11/11/-2
FP Subtract Single sub.s	sub.s	$R_1 = R_0 - R_2 + \text{SignExtImm}$	11/10/-1
FP Subtract Double sub.d	sub.d	$R_1 = R_0 - R_2 + \text{SignExtImm}$	11/11/-1
Load Halfword lhu	lhu	$I_1 = 24\text{b}[M(R_0)]$	2) 24hex
Load Byte Unsigned lbu	lbu	$I_1 = 8\text{b}[M(R_0)] + \text{SignExtImm}(7:0)$	2) 25hex
Load Word lw	lw	$I_1 = M[R_0]$	2,7) 30hex
Load Upper Imm. lui	lui	$I_1 = M[R_0] + \text{SignExtImm}$	(2,7) 30hex
Load Word	lw	$I_1 = M[R_0]$	23hex
Nor nor	nor	$R_1 = \neg(R_0 \& R_2)$	0/27hex
Or or	or	$R_1 = R_0 \mid R_2$	0/25hex
Or Immediate ori	ori	$I_1 = R_0 \mid \text{ZeroExtImm}$	3) dhex
Set Less Than slt	slt	$R_1 = \text{R}_0 < \text{R}_2$	1) 0
Set Less Than Imm. slti	slt	$I_1 = \text{R}_0 < \text{SignExtImm}$	1) 0 (2) 8hex
Set Less Than Unsigned slti	slt	$I_1 = \text{R}_0 < \text{SignExtImm}$	7) 1
Set Less Than Unsigned sltiu	slt	$I_1 = \text{R}_0 < \text{SignExtImm}$	6) 0/2bhex
Shift Left Logical sll	sll	$R_1 = \text{R}_0 <> \text{shamt}$	0/0bhex
Shift Right Logical srl	srl	$R_1 = \text{R}_0 <> \text{shamt}$	0/0bhex
Store Byte sb	sb	$I_1 = M[R_0] + \text{SignExtImm}(7:0)$	2) 28hex
Store Conditional sc	sc	$I_1 = M[R_0] + \text{SignExtImm}(R_1)$	2,6) bhex
Store Halfword sh	sh	$I_1 = M[R_0] + \text{SignExtImm}(15:0)$	2) 29hex
Store Word sw	sw	$I_1 = M[R_0] + \text{SignExtImm}(R_1)$	2) 29hex
Subtract sub	sub	$R_1 = R_0 - R_2$	1) 0/22hex
Subtract Unsigned subu	subu	$R_1 = R_0 - R_2 + \text{R}[rt]$	0/23hex
FOLLOWING INSTRUCTION FORMATS			
FR	opcode	fmt	ft
		21 20	19 18 17 16 15 14 13 12 11 10
FI	opcode	fmt	ft
		21 20	19 18 17 16 15
PSEUDOINSTRUCTION SET			
NAME	MNEMONIC	OPERATION	
Branch Less Than	blt	$I(R_0 < R_1) \text{ PC} = \text{Label}$	
Branch Greater Than	bgt	$I(R_0 > R_1) \text{ PC} = \text{Label}$	
Branch Less Than or Equal	ble	$I(R_0 \leq R_1) \text{ PC} = \text{Label}$	
Branch Greater Than or Equal	bge	$I(R_0 \geq R_1) \text{ PC} = \text{Label}$	
Load Immediate	li	$R_1 = \text{Immediate}$	
Move	move	$R_1 = R_0$	
REGISTER NAME, NUMBER, USE, CALL CONVENTION			
NAME	NUMBER	USE	PREERVED ACROSS CALL?
Zero	0	The Constant Value 0	N.A.
\$a	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$s0-\$s7	4-7	Temporaries	No
\$s0-\$s7	8-15	Saved Temporaries	Yes
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$s0-\$s9	24-25	Temporaries	No
\$s0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes
BASIC INSTRUCTION FORMATS			
R	opcode	rs	rt
		21 20	19 18 17 16 15 14 13 12 11 10 9 8
I	opcode	rs	rt
		21 20	19 18 17 16 15 0
J	opcode	rs	address
		21 20	19 18 17 16 15 0

NAME	NUMBER	USE	PREERVED ACROSS CALL?
Zero	0	The Constant Value 0	N.A.
\$a	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$s0-\$s7	4-7	Temporaries	No
\$s0-\$s7	8-15	Saved Temporaries	Yes
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$s0-\$s9	24-25	Temporaries	No
\$s0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

(1) May cause overflow exception

(2) ZeroExtImm = $\{ 16\text{b}[0] \}$, immediate

(3) BranchAdd = $\{ 14\text{b}[31:15] \}$, immediate, 2b⁰

(4) JumpAdd = $\{ 14\text{b}[31:28] \}$, address, 2b⁰

(5) Operands considered unsigned numbers (vs. 2's comp.)

(7) Atomic test/swap pair, R[rt] = 1 if pair atomic, 0 if not atomic

(1) May cause overflow exception

(2) ZeroExtImm = $\{ 16\text{b}[0] \}$, immediate

(3) BranchAdd = $\{ 14\text{b}[31:15] \}$, immediate, 2b⁰

(4) JumpAdd = $\{ 14\text{b}[31:28] \}$, address, 2b⁰

(5) Operands considered unsigned numbers (vs. 2's comp.)

(7) Atomic test/swap pair, R[rt] = 1 if pair atomic, 0 if not atomic

(1) opcode(31:26) = 0

(2) opcode(31:26) = 17₁₆(1)₁₆; if fmt(25:21) = 16₁₆(10₁₆)f^m = (single);

if fnt(25:21) = 17₁₆(1)₁₆ if (double)

IEEE 754 FLOATING-POINT STANDARD

(1) $1 \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127,

Double Precision Bias = 1023,

IEEE 754 Symbols

S.P. MAX = 255, D.P. MAX = 2047

S.P. MIN = -255, D.P. MIN = -2047

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

S.P. UNDERFLOW = Underflow, D.P. UNDERFLOW = Underflow

S.P. OVERFLOW = Overflow, D.P. OVERFLOW = Overflow

S.P. INTEGRAL = Integral, D.P. INTEGRAL = Integral

S.P. NAN = NaN, D.P. NAN = NaN

S.P. INF = Inf, D.P. INF = Inf

S.P. NEG INF = -Inf, D.P. NEG INF = -Inf

S.P. QUIT = QUIT, D.P. QUIT = QUIT

S.P. DENORM = Denorm, D.P. DENORM = Denorm

S.P. SING = Sing, D.P. SING = Sing

S.P. SUBNORMAL = Subnormal, D.P. SUBNORMAL = Subnormal

Conditional Branch Instructions

■ beq (Branch if Equal)

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

beq (4)	rs	rt	immediate=offset
6 bits	5 bits	5 bits	16 bits

I-Type

■ Semantics (assuming no branch delay slot)

if $\text{MEM}[\text{PC}] == \text{beq } \text{rs } \text{rt } \text{immediate}_{16}$

target = $\text{PC}^+ + \text{sign-extend}(\text{immediate}) \times 4$

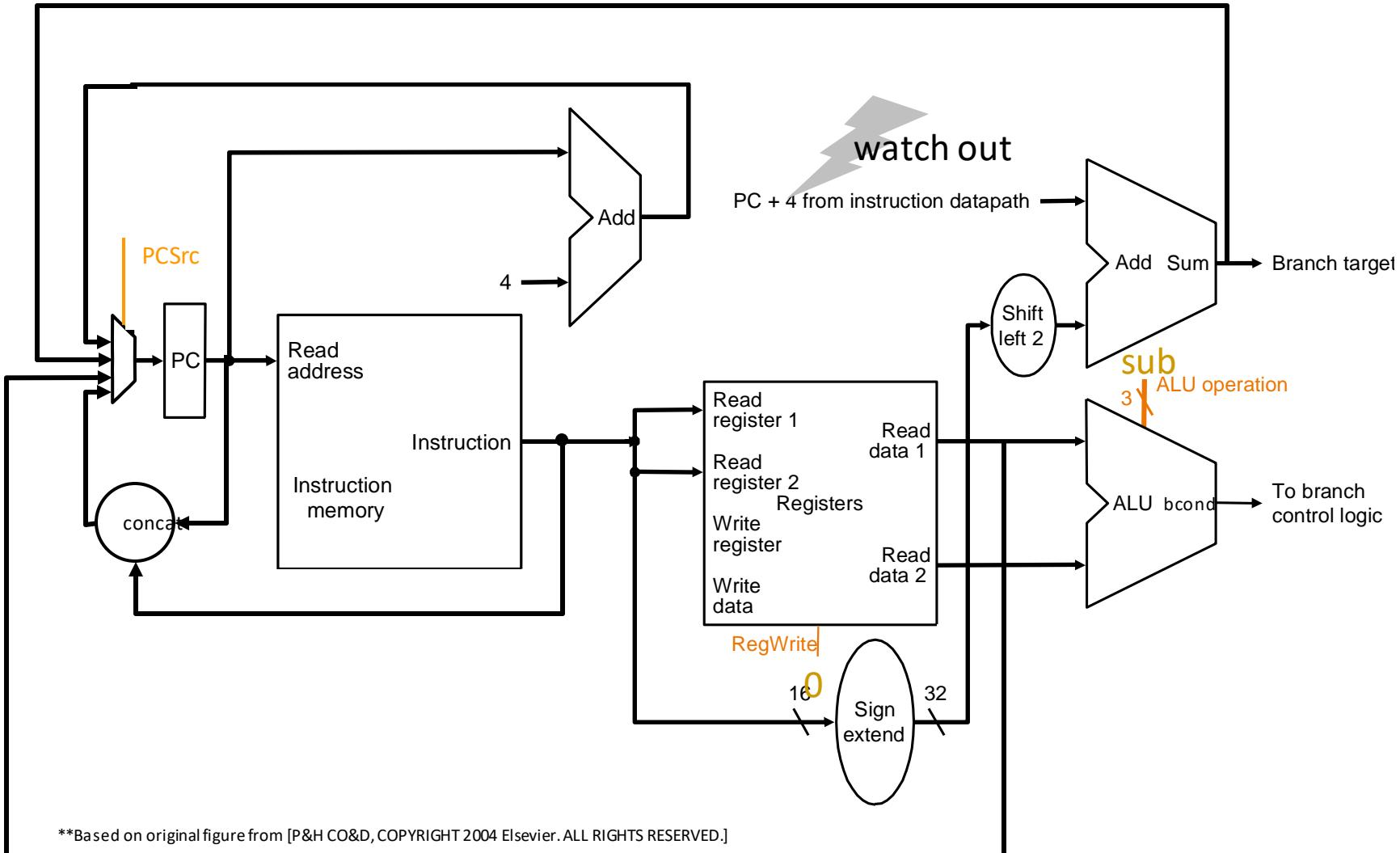
if $\text{GPR}[\text{rs}] == \text{GPR}[\text{rt}]$ then $\text{PC} \leftarrow \text{target}$

else $\text{PC} \leftarrow \text{PC} + 4$

□ Variations: beq, bne, blez, bgtz

[†]This is the incremented PC

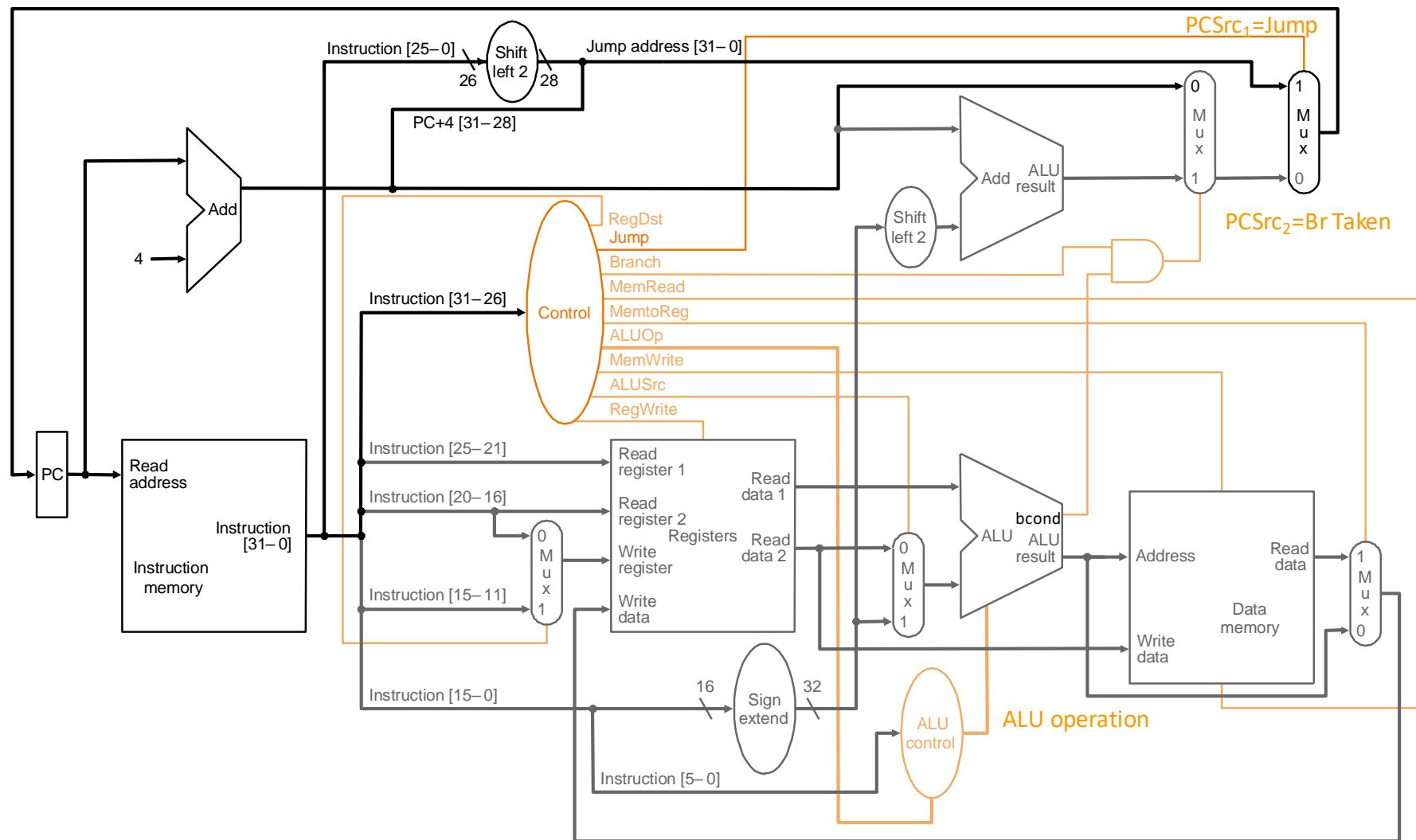
Conditional Branch Datapath (for you to finish)



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

How to uphold the delayed branch semantics?

Putting It All Together



Single-Cycle Control Logic

Single-Cycle Hardwired Control

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

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

R-Type

31	26	25	21	20	16	15	0
opcode	rs	rt				immediate	
6 bits	5 bits	5 bits				16 bits	

I-Type

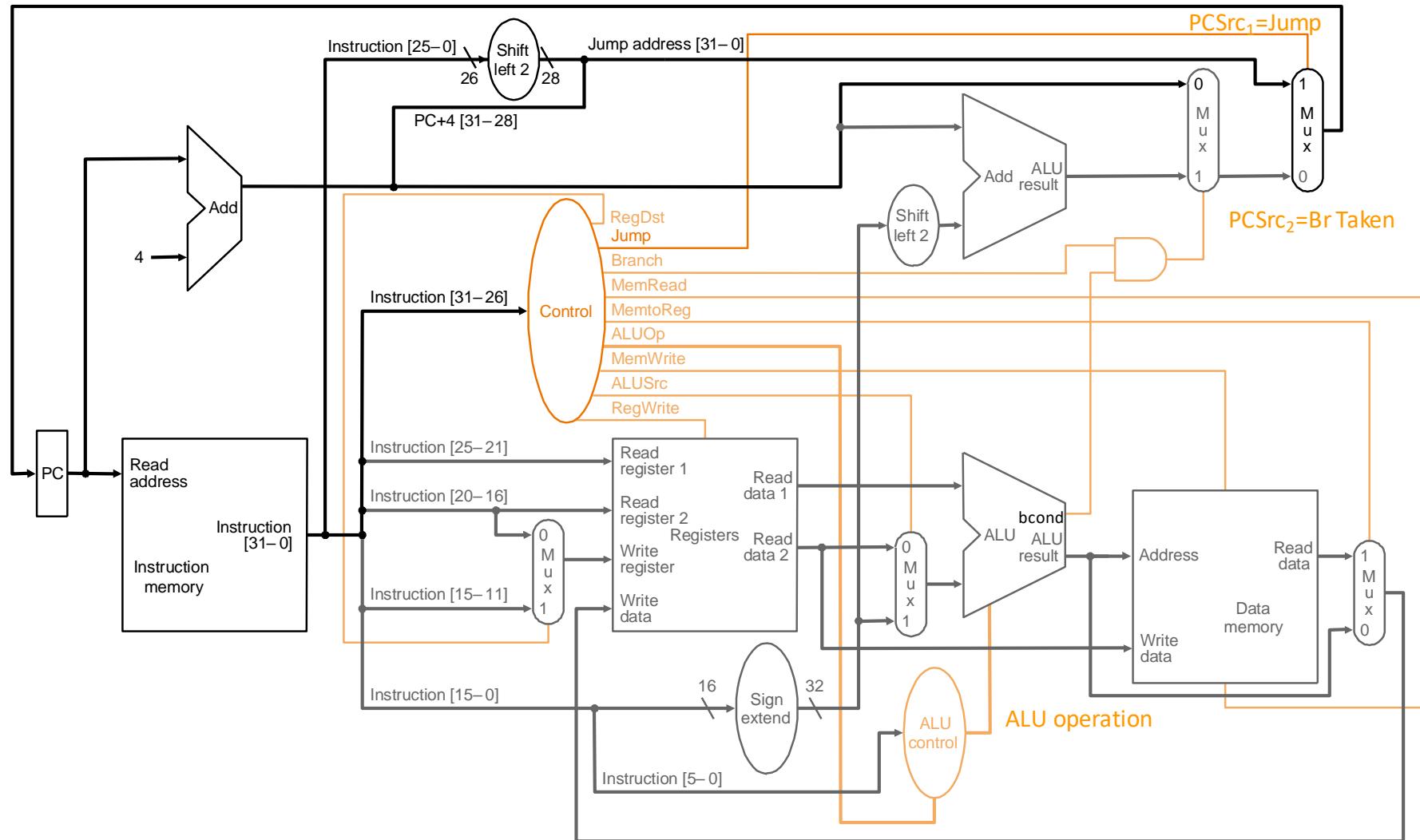
31	26	25	0
opcode			immediate
6 bits			26 bits

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)



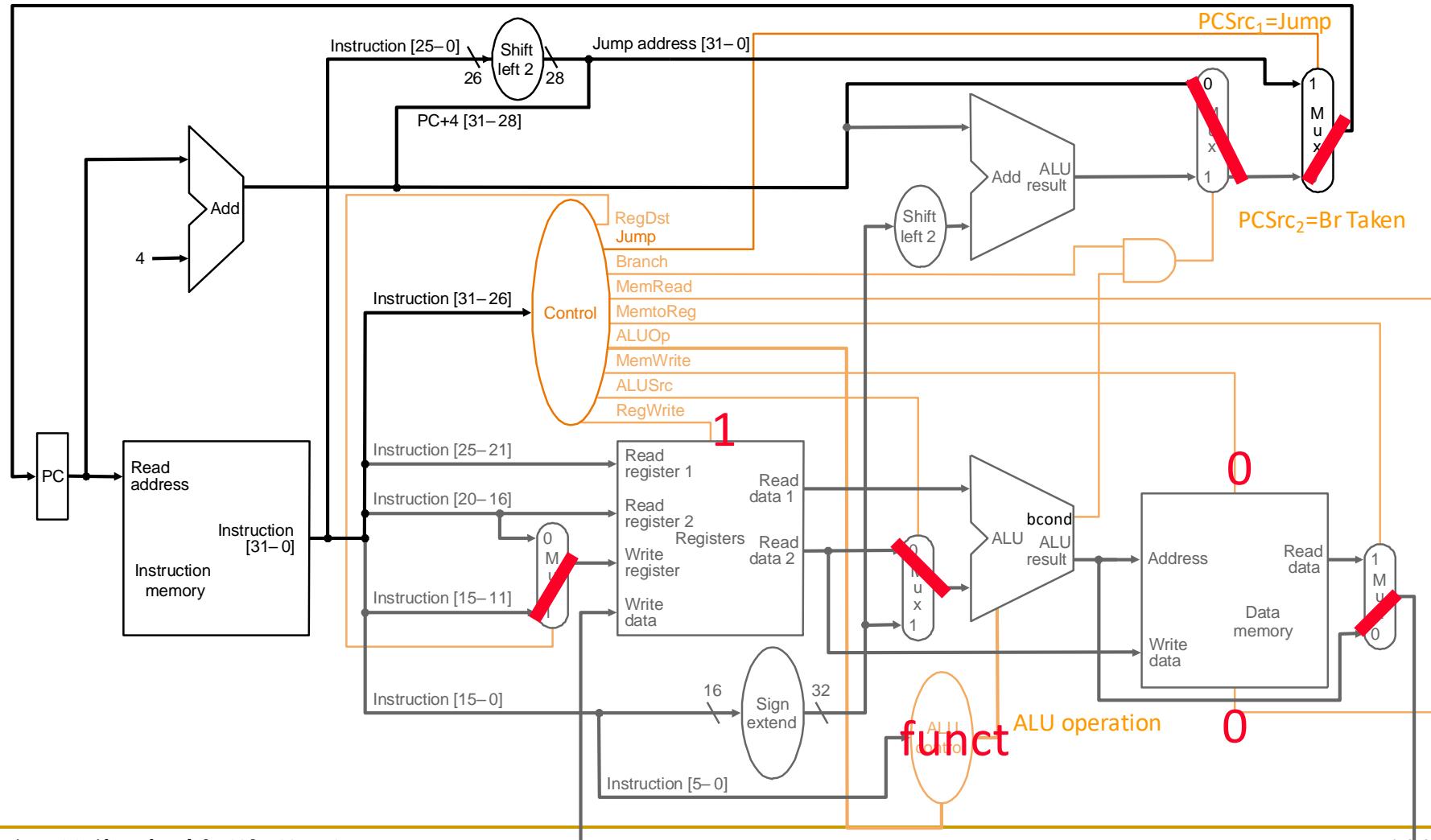
Single-Bit Control Signals (I)

	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	$(opcode != SW) \&\&$ $(opcode != Bxx) \&\&$ $(opcode != J) \&\&$ $(opcode != JR))$

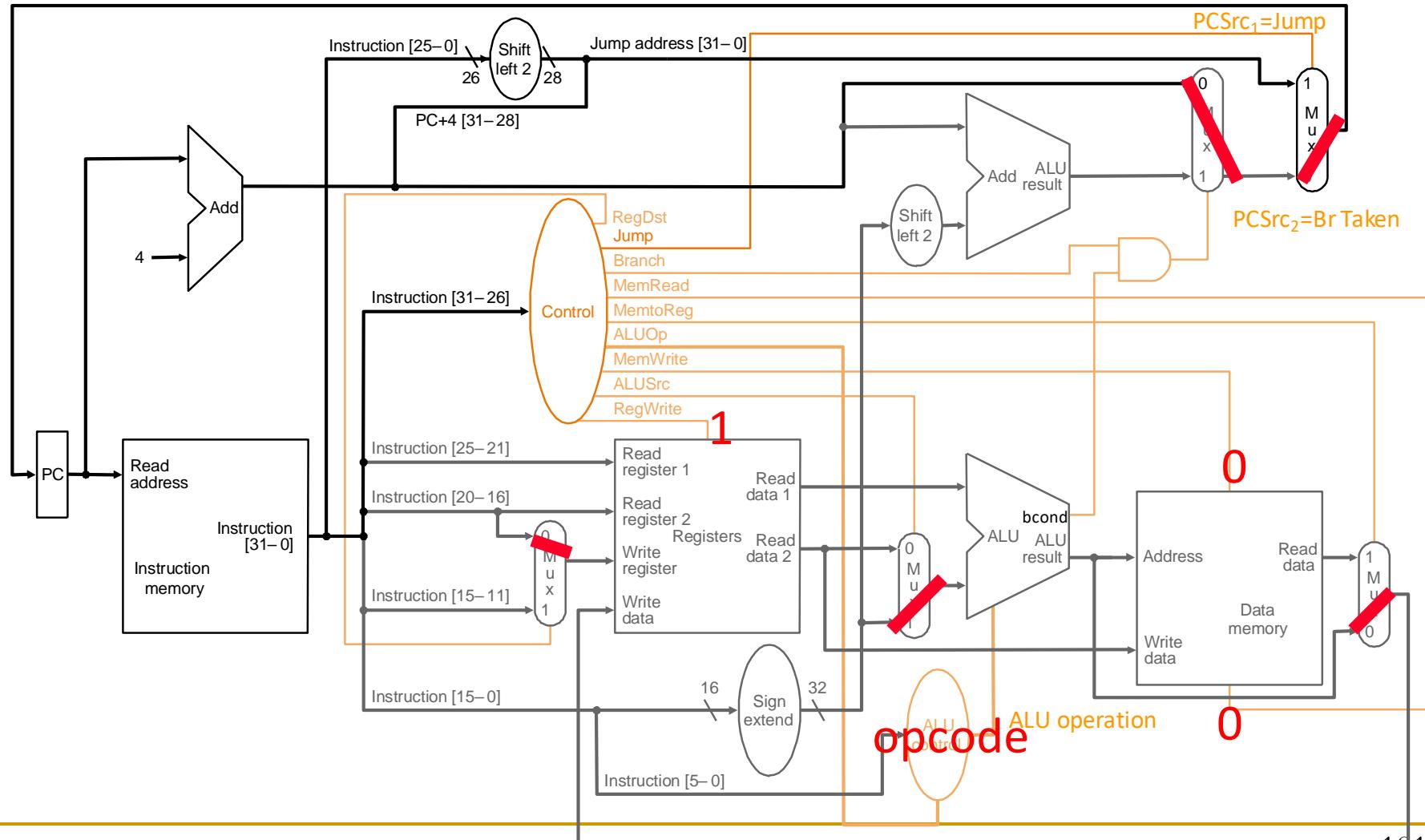
Single-Bit Control Signals (II)

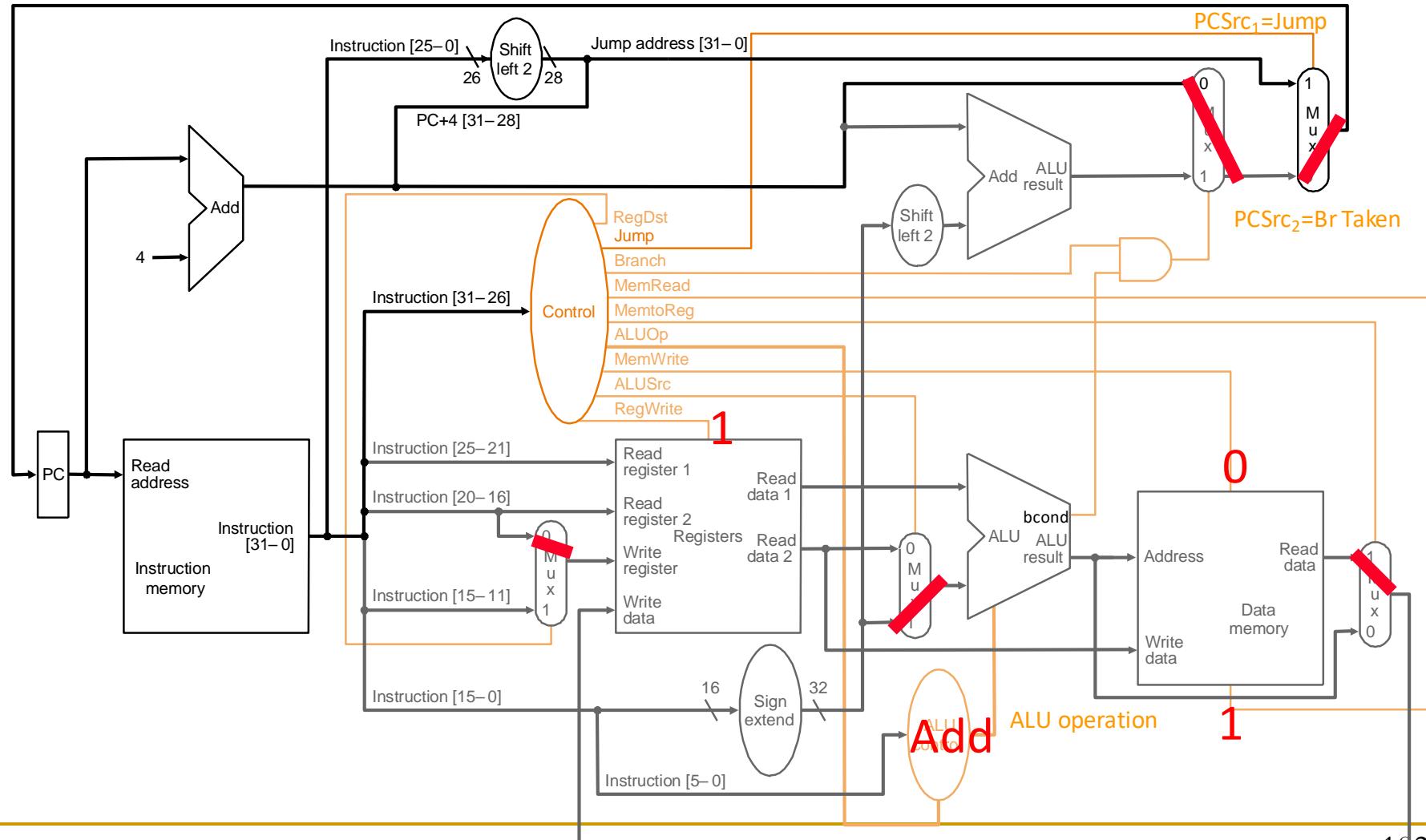
	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	$\text{opcode} == \text{LW}$
MemWrite	Memory write disabled	Memory write enabled	$\text{opcode} == \text{SW}$
PCSrc ₁	According to PCSrc ₂	next PC is based on 26-bit immediate jump target	$(\text{opcode}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

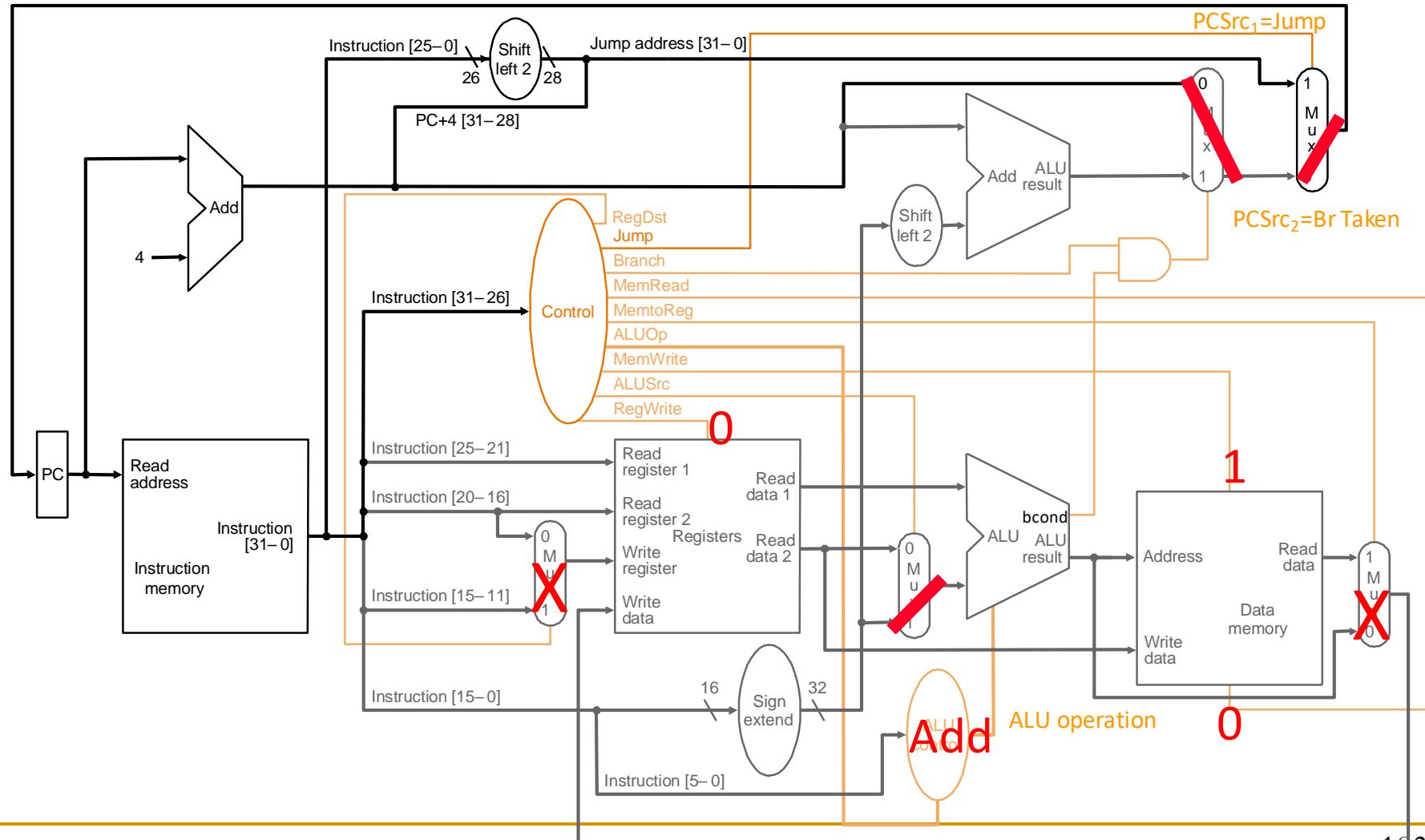
R-Type ALU



I-Type ALU

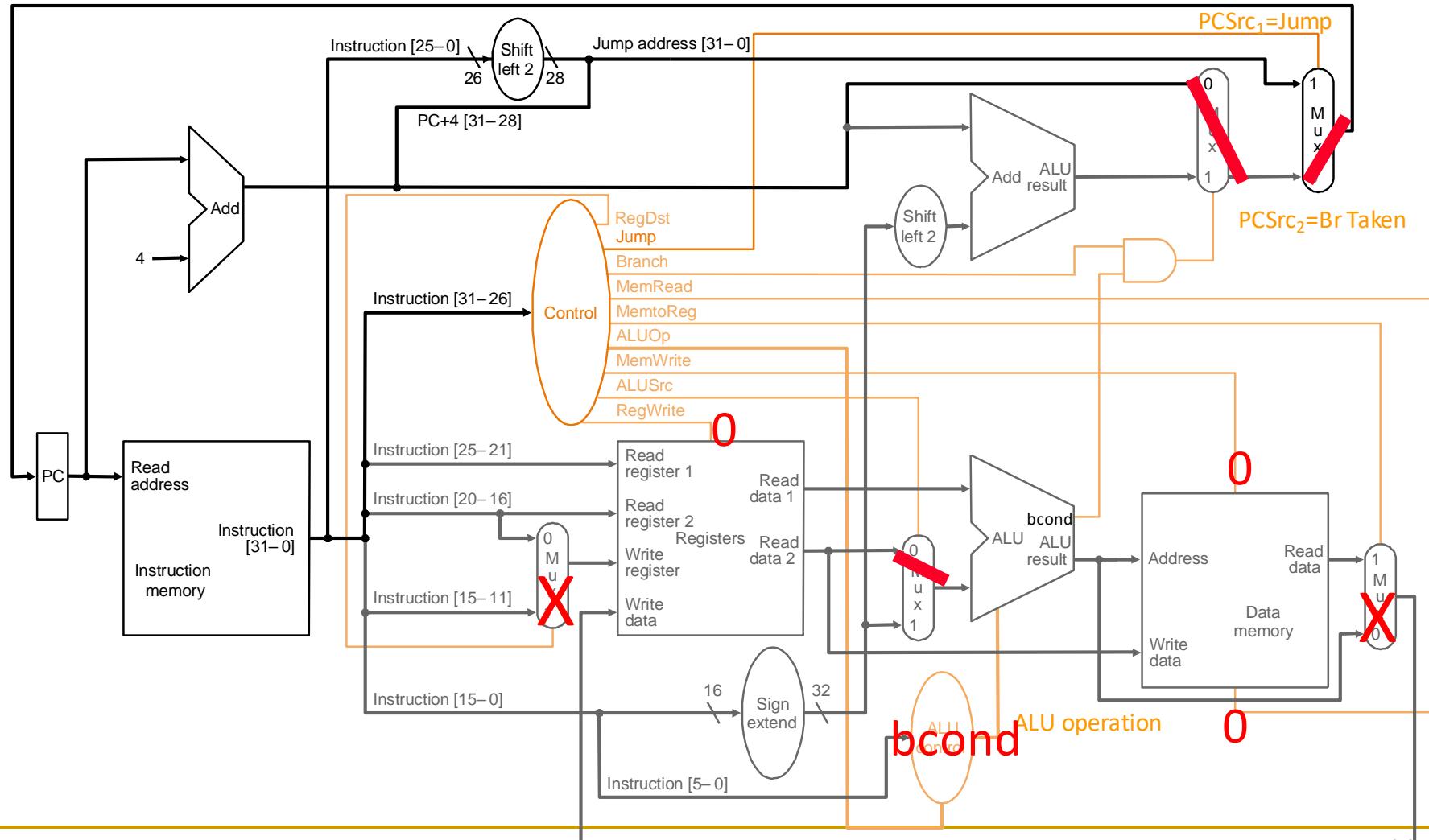






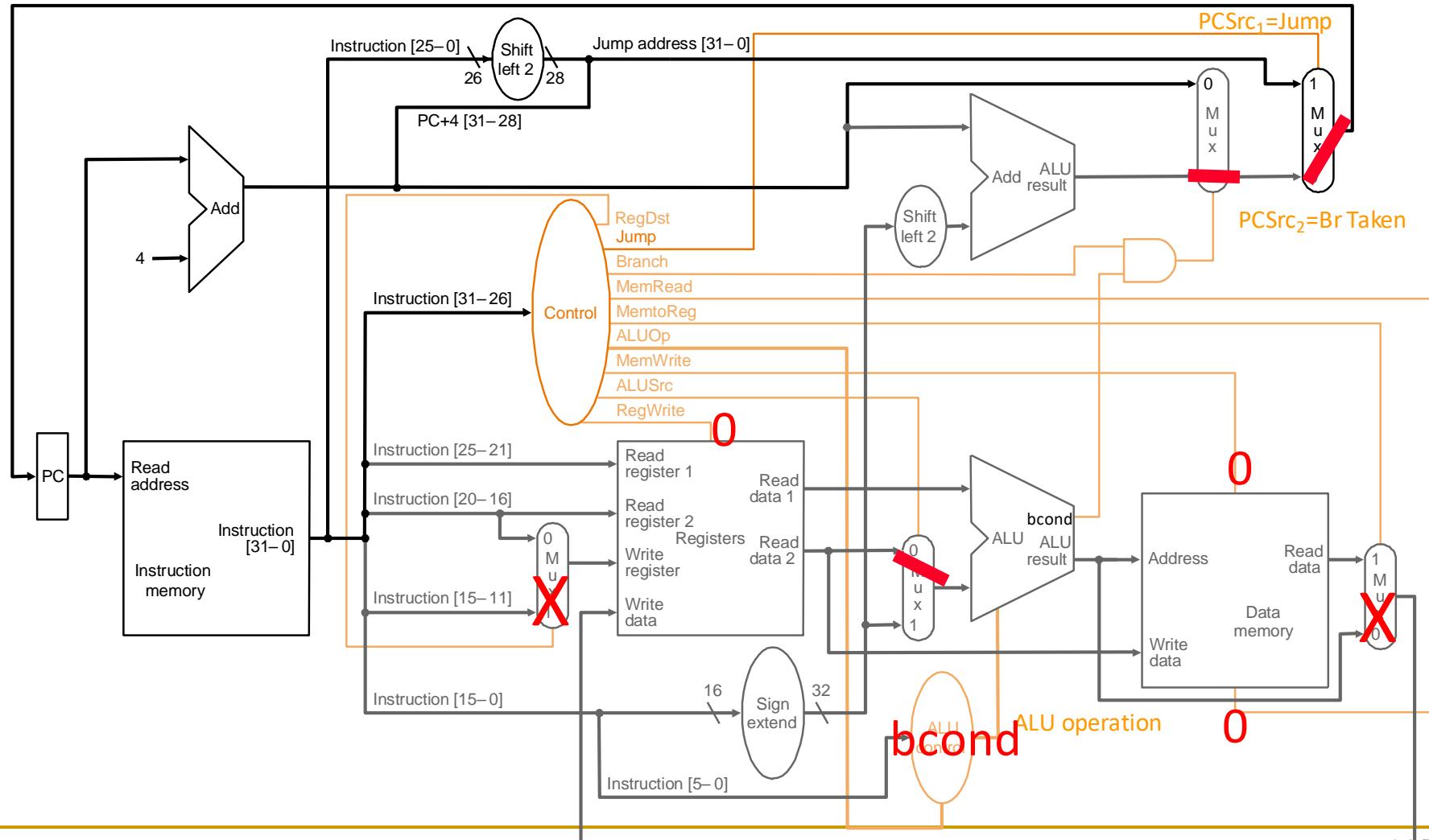
Branch (Not Taken)

Some control signals are dependent on the processing of data

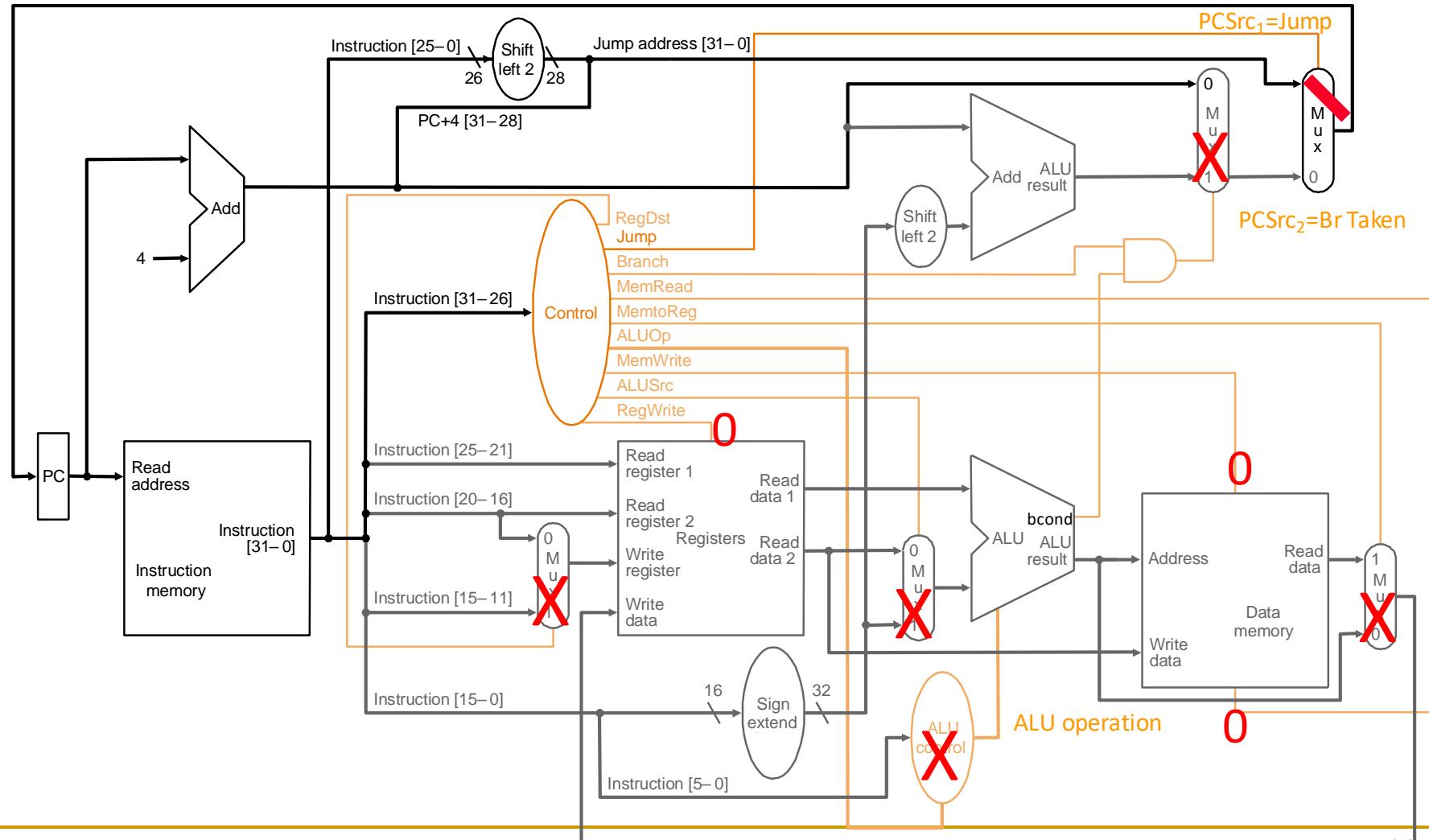


Branch (Taken)

Some control signals are dependent on the processing of data



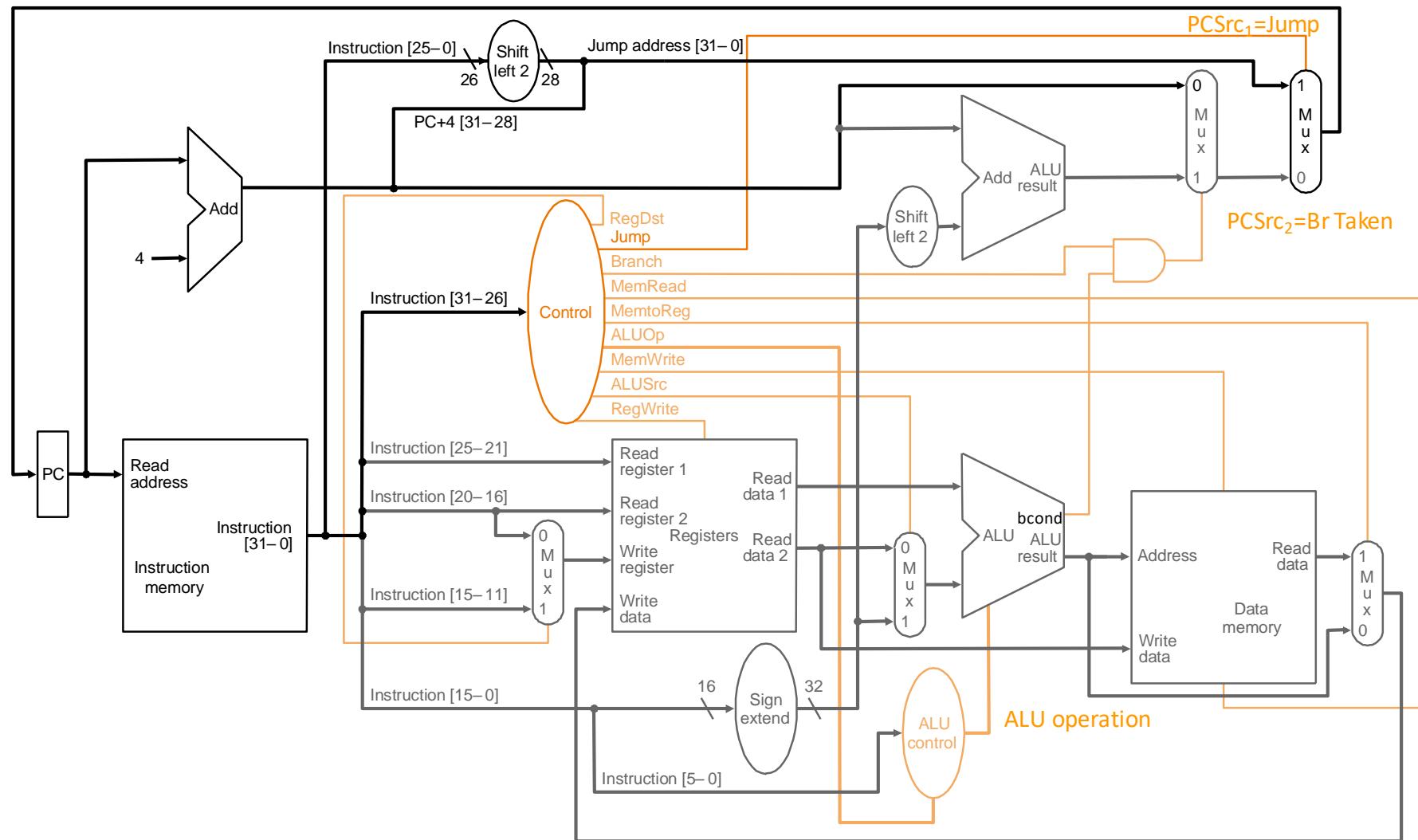
Jump



What is in That Control Box?

- Combinational Logic → Hardwired Control
 - Idea: Most control signals generated组合性地 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.

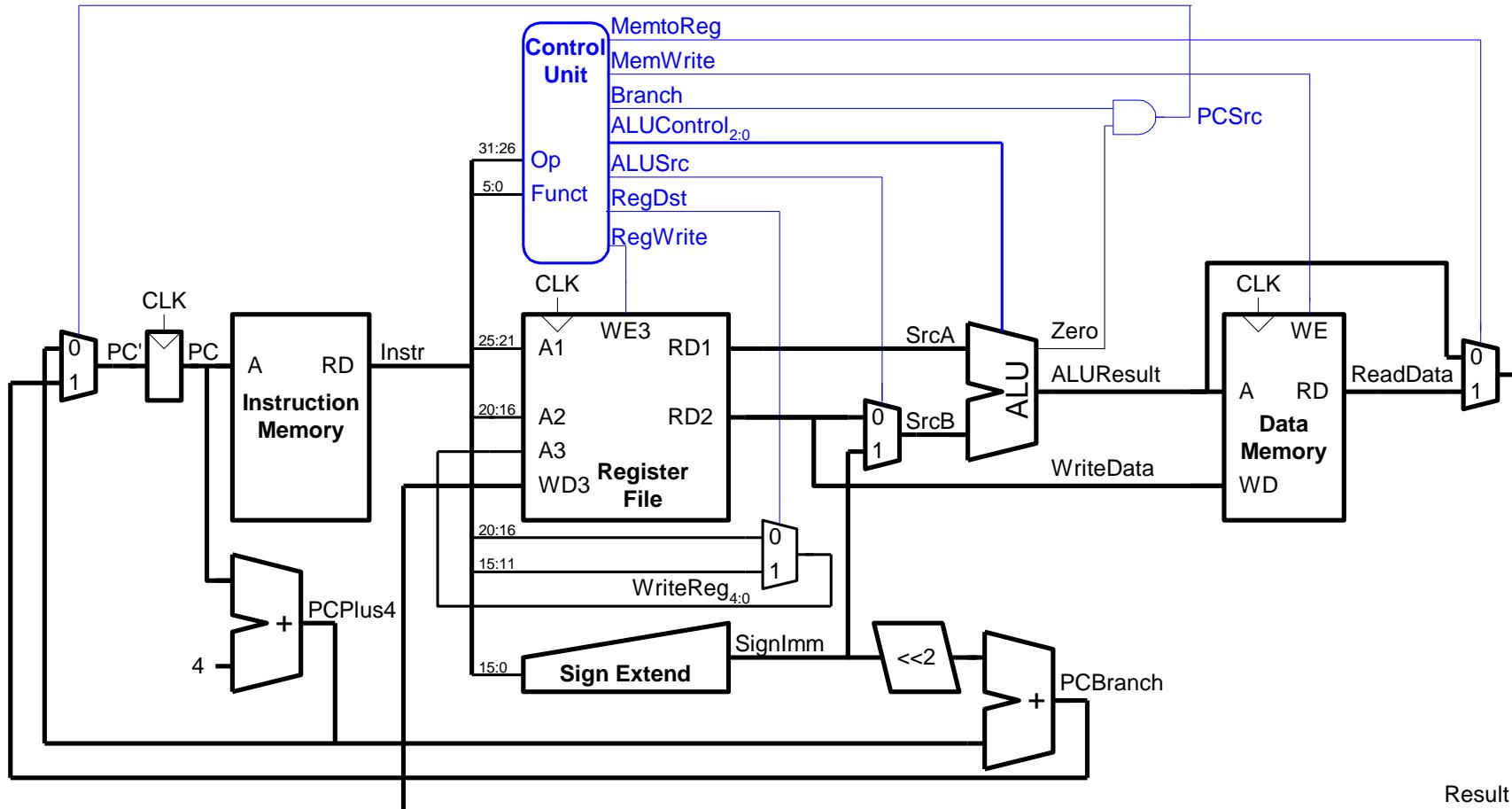
Review: Complete Single-Cycle Processor



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

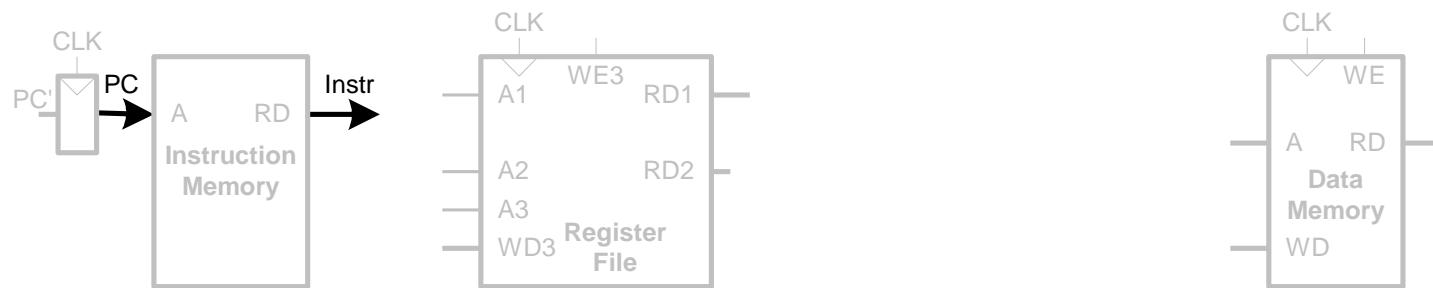
See backup slides to reinforce the concepts we have covered.
They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6

Another Complete Single-Cycle Processor



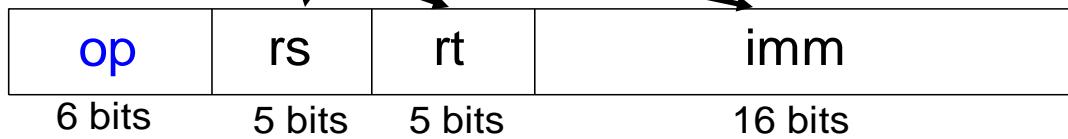
Example: Single-Cycle Datapath: lw fetch

■ STEP 1: Fetch instruction



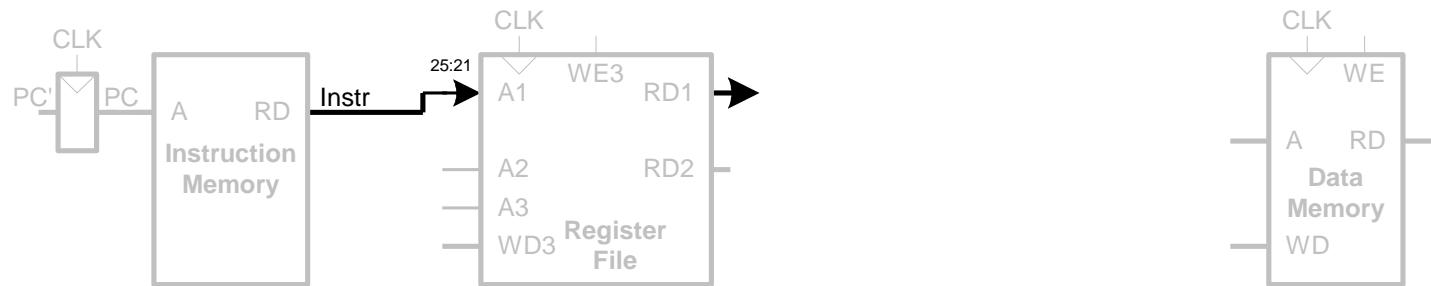
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type



Single-Cycle Datapath: lw register read

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



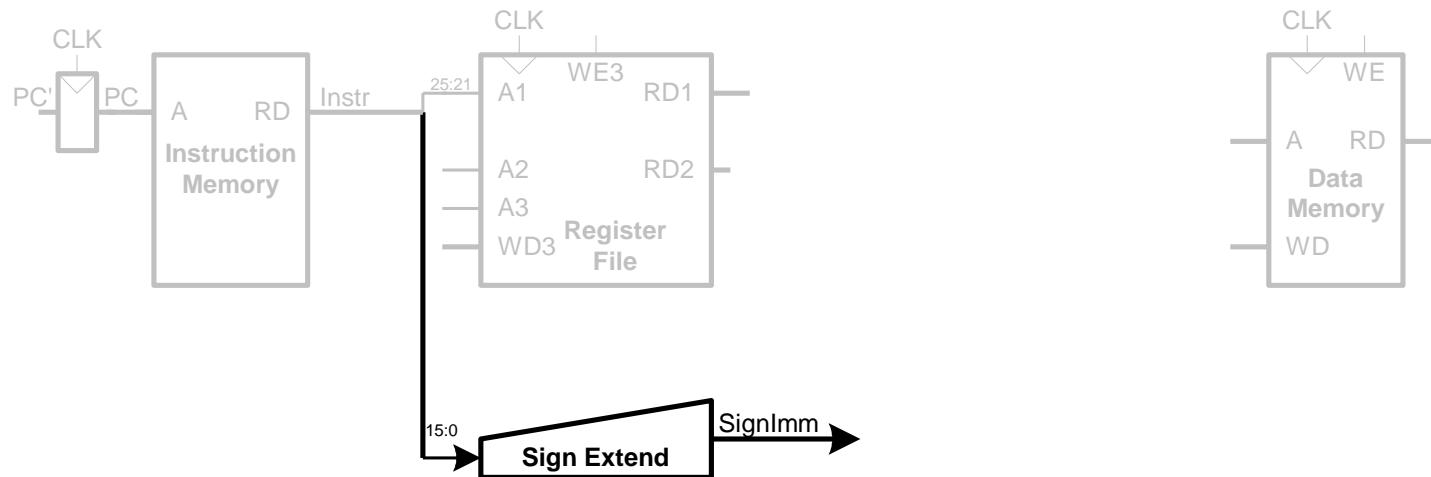
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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**



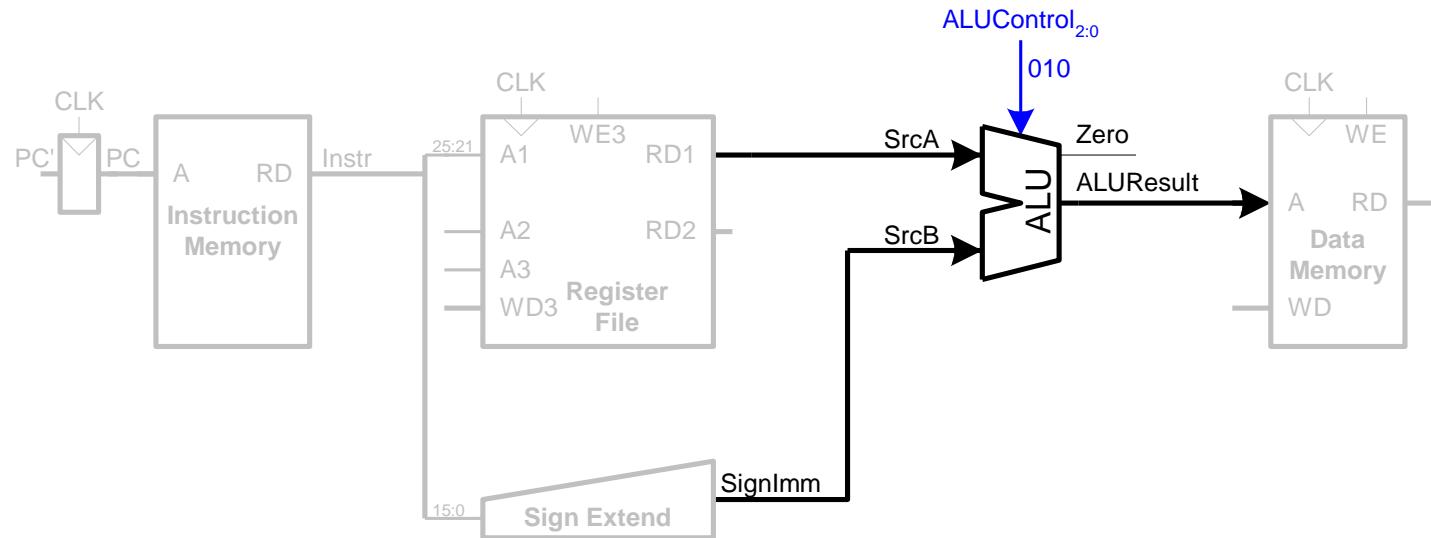
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



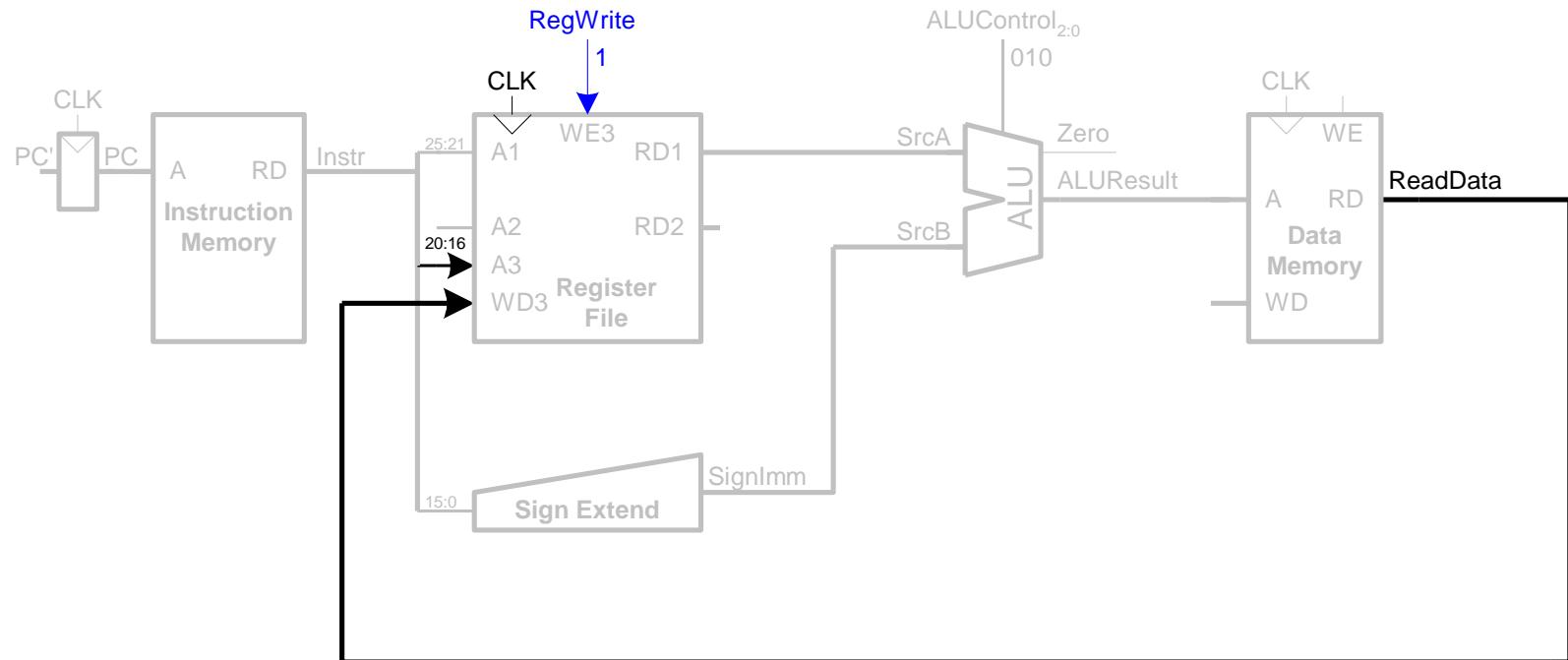
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



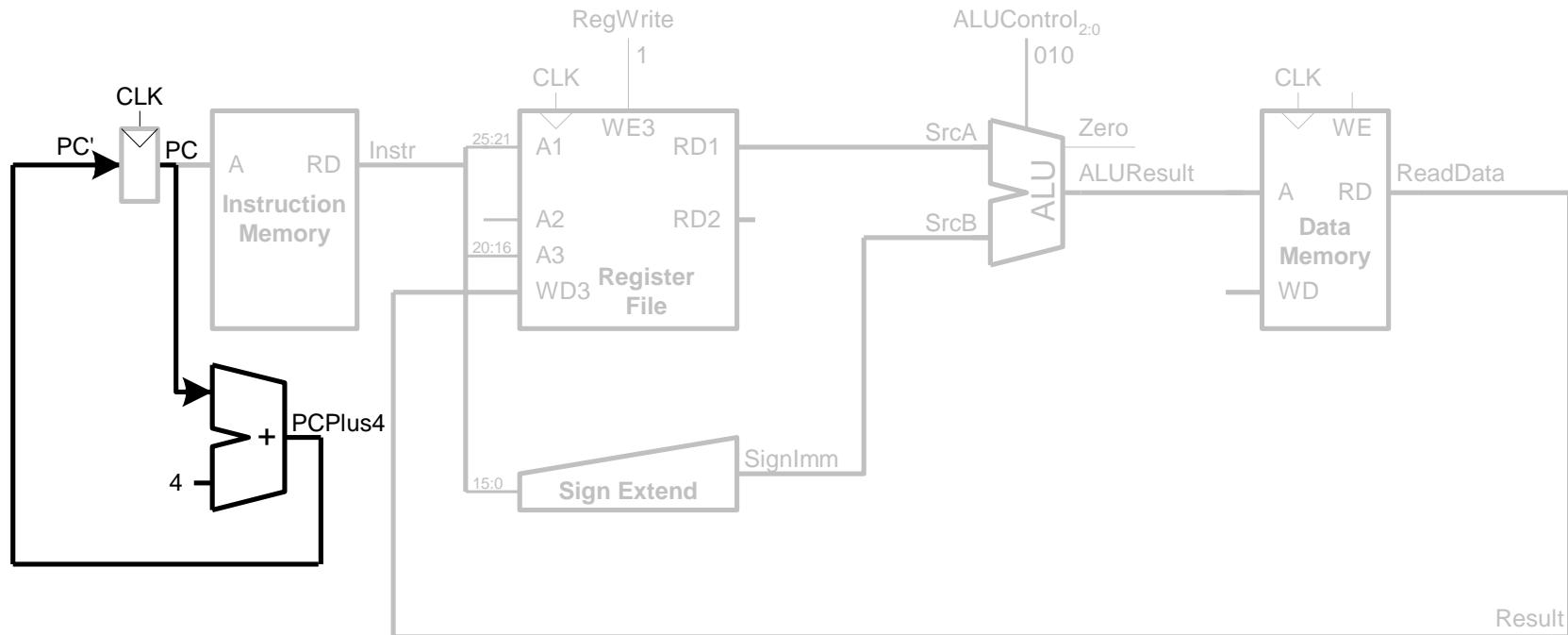
lw \$s3, 1(\$0) # read memory word 1 into \$s3

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

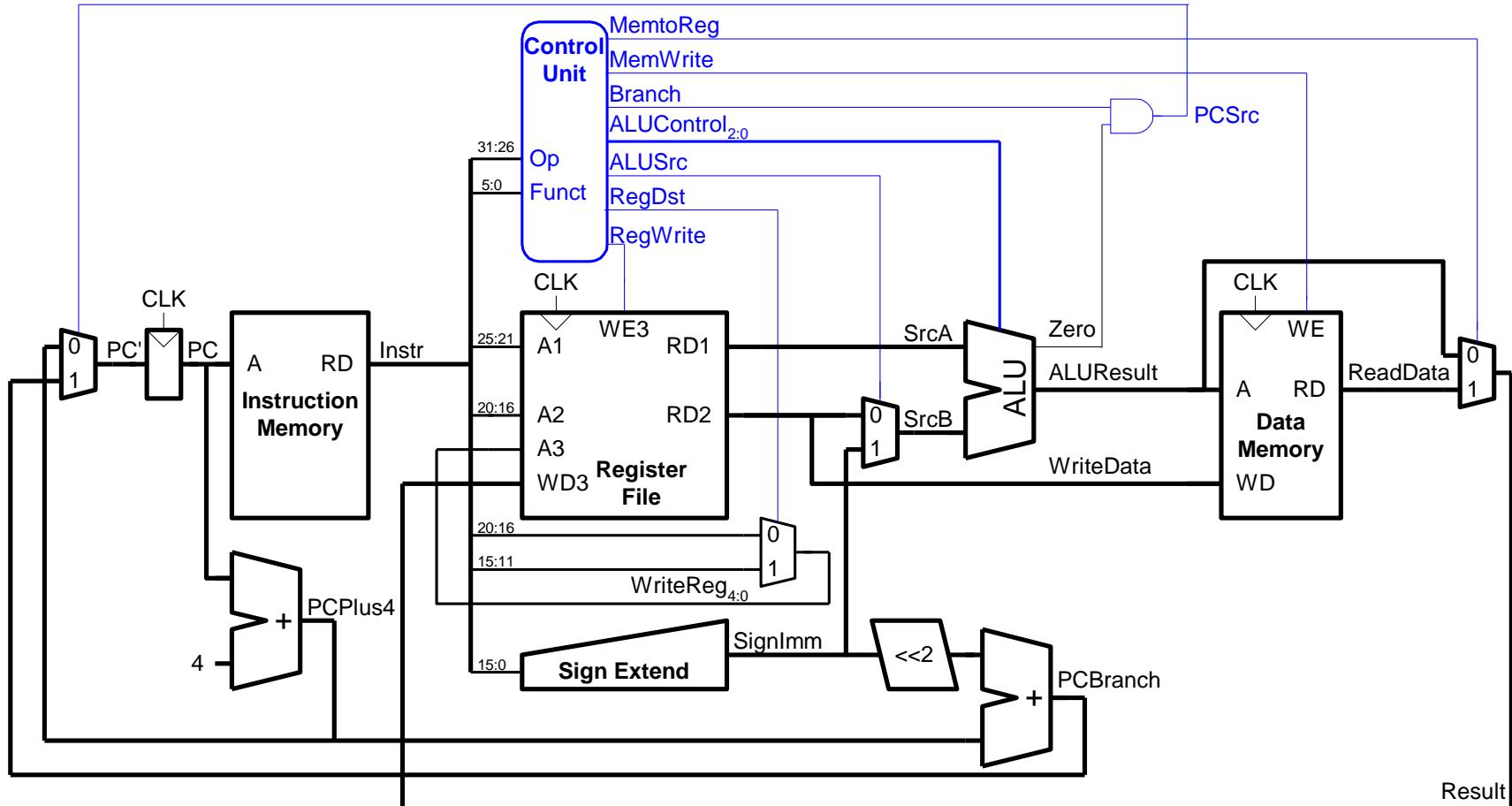
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

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

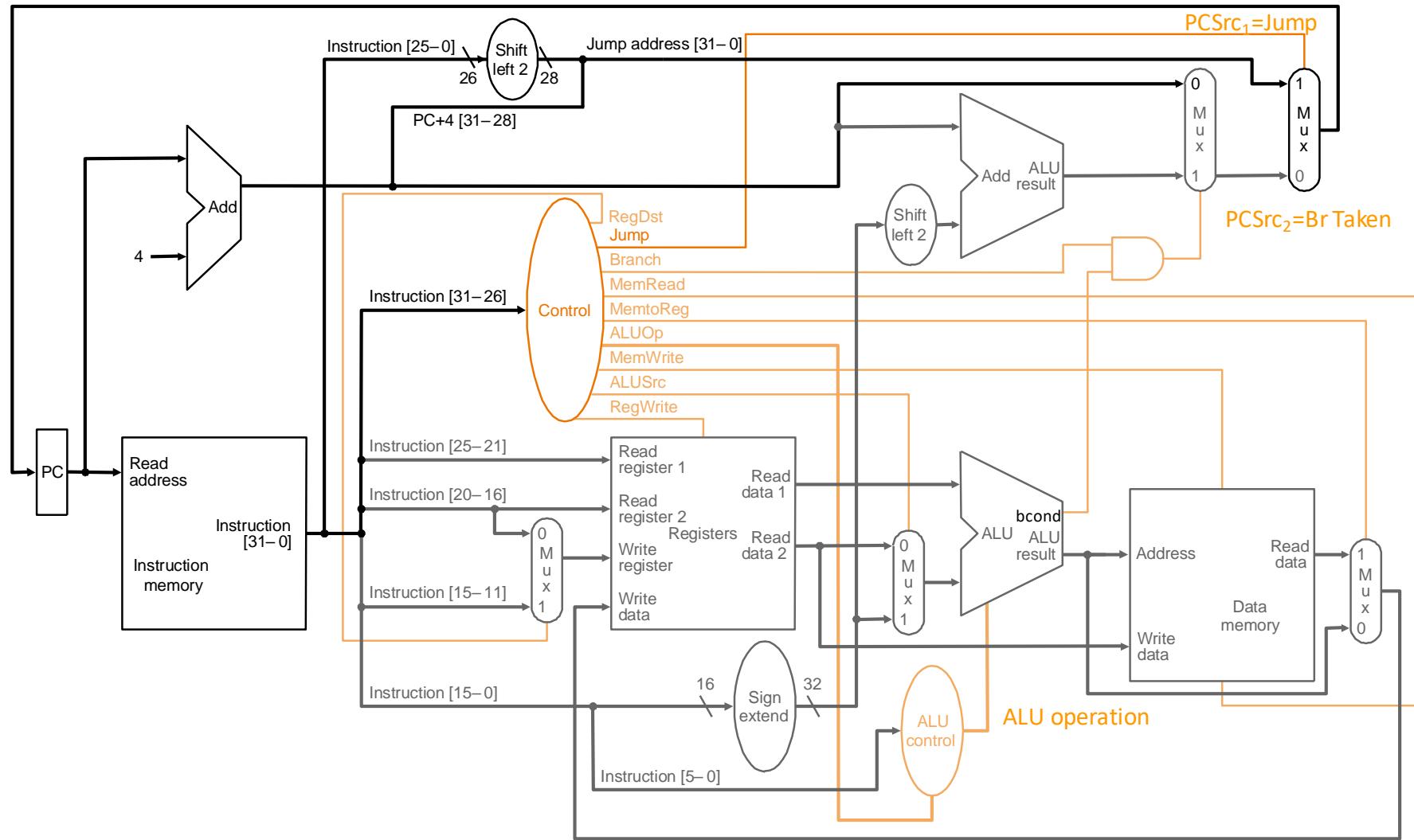
Another Complete Single-Cycle Processor (H&H)



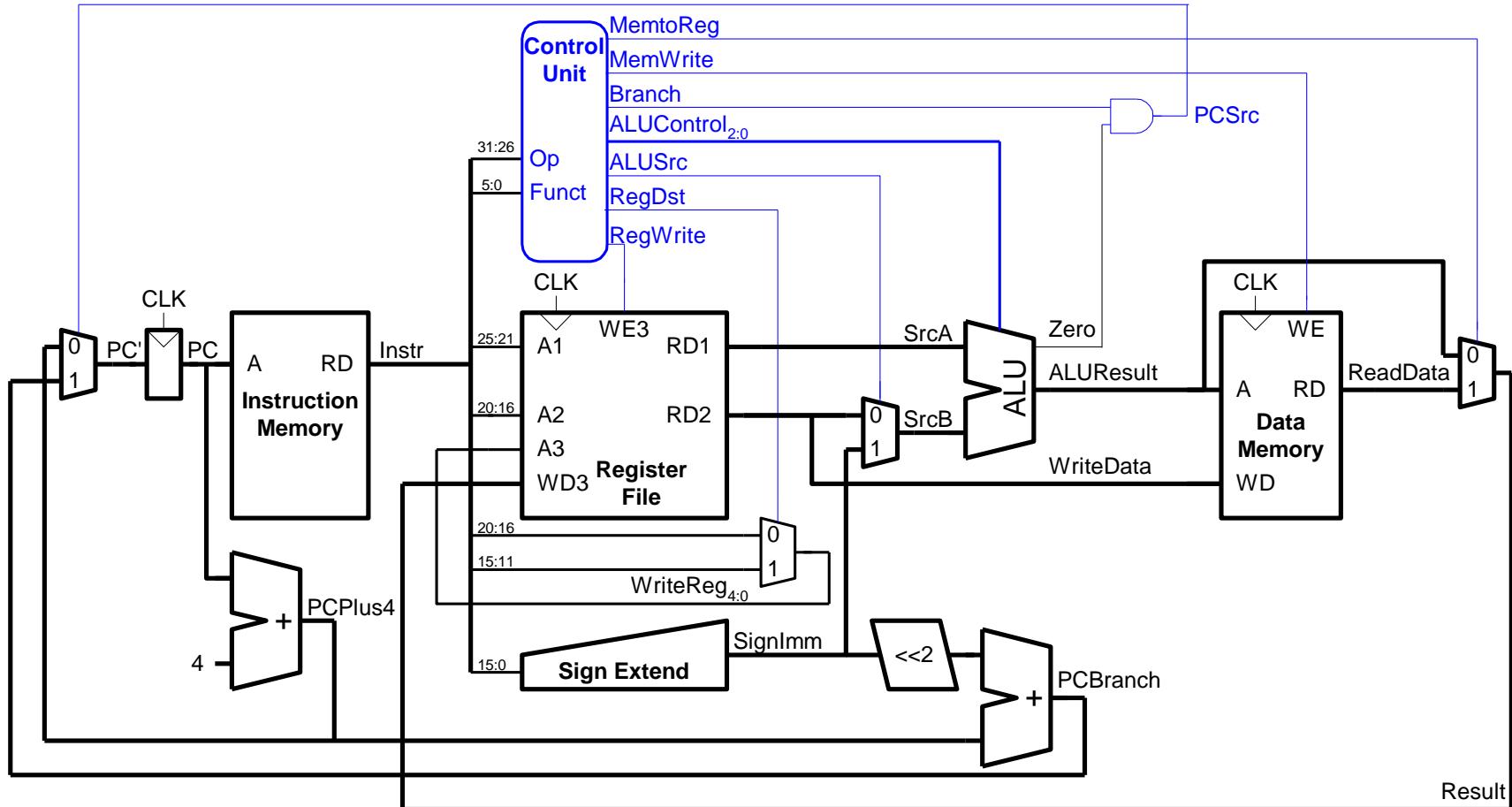
Your Reading Assignment

- 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

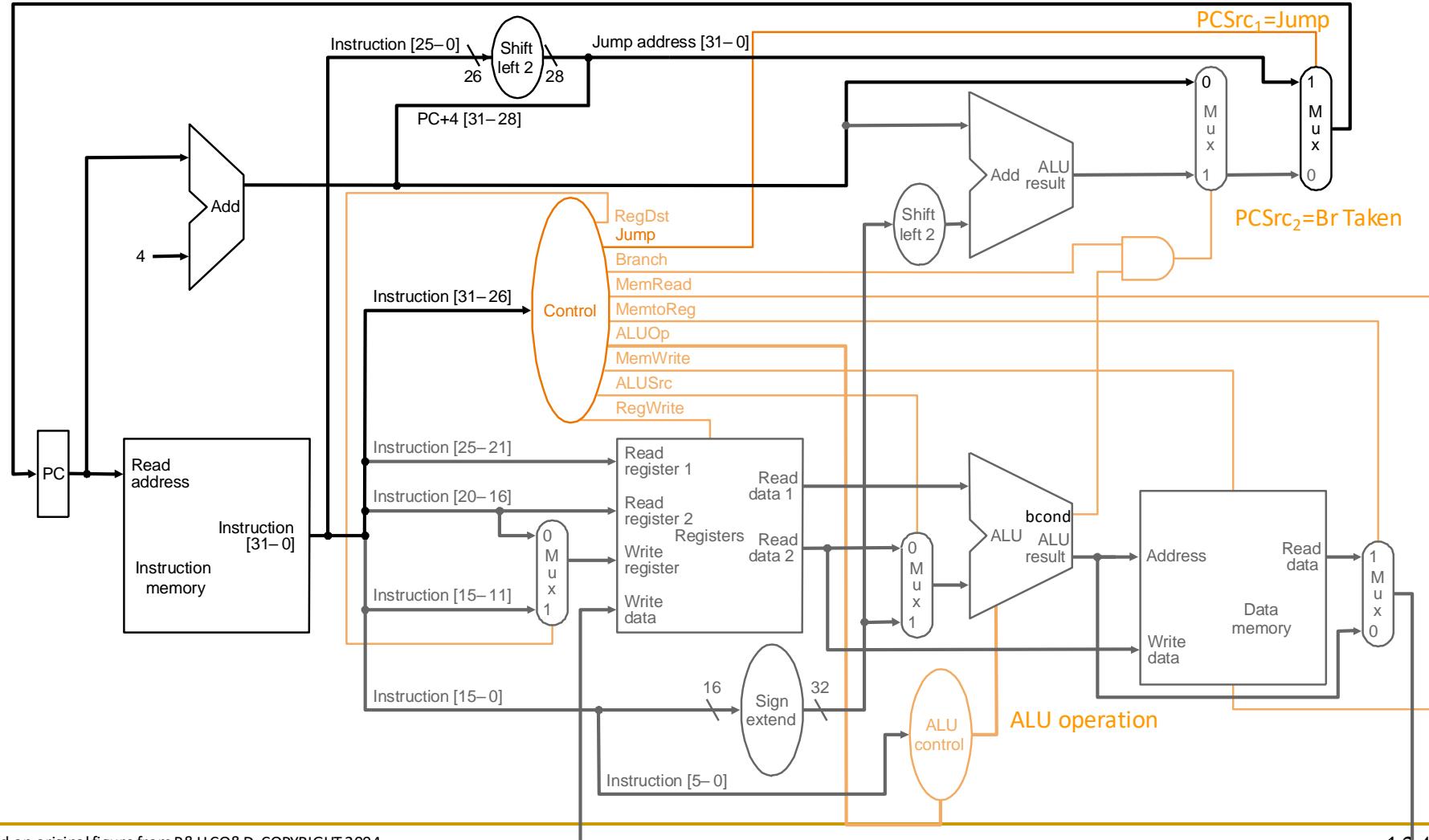
A Single-Cycle Microarchitecture: Analysis

- 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
 - 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?

Let's Find the Critical Path

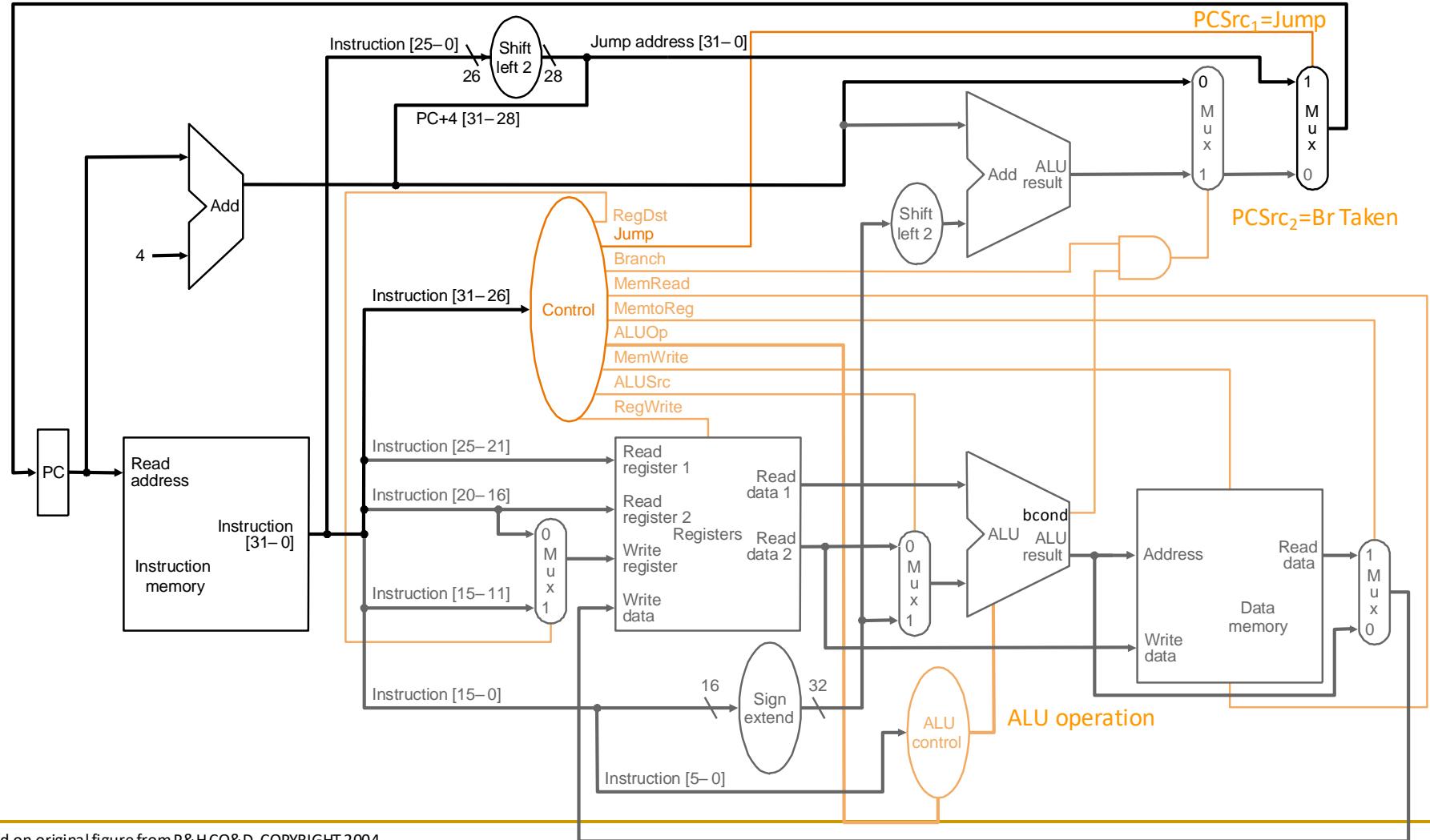


Example Single-Cycle Datapath Analysis

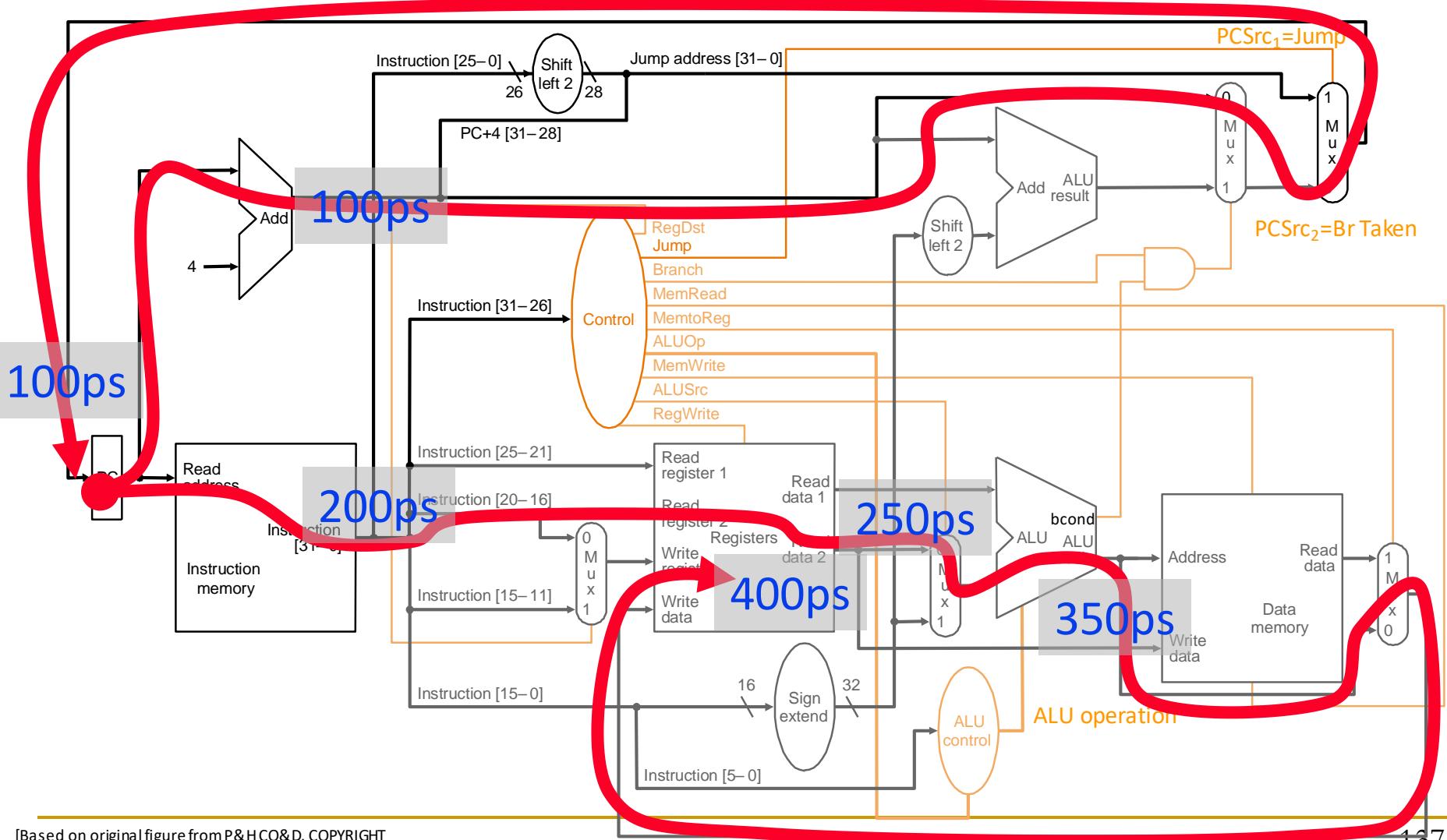
- Assume (for the design in the previous slide)
 - memory units (read or write): 200 ps
 - ALU and adders: 100 ps
 - register file (read or write): 50 ps
 - other combinational logic: 0 ps

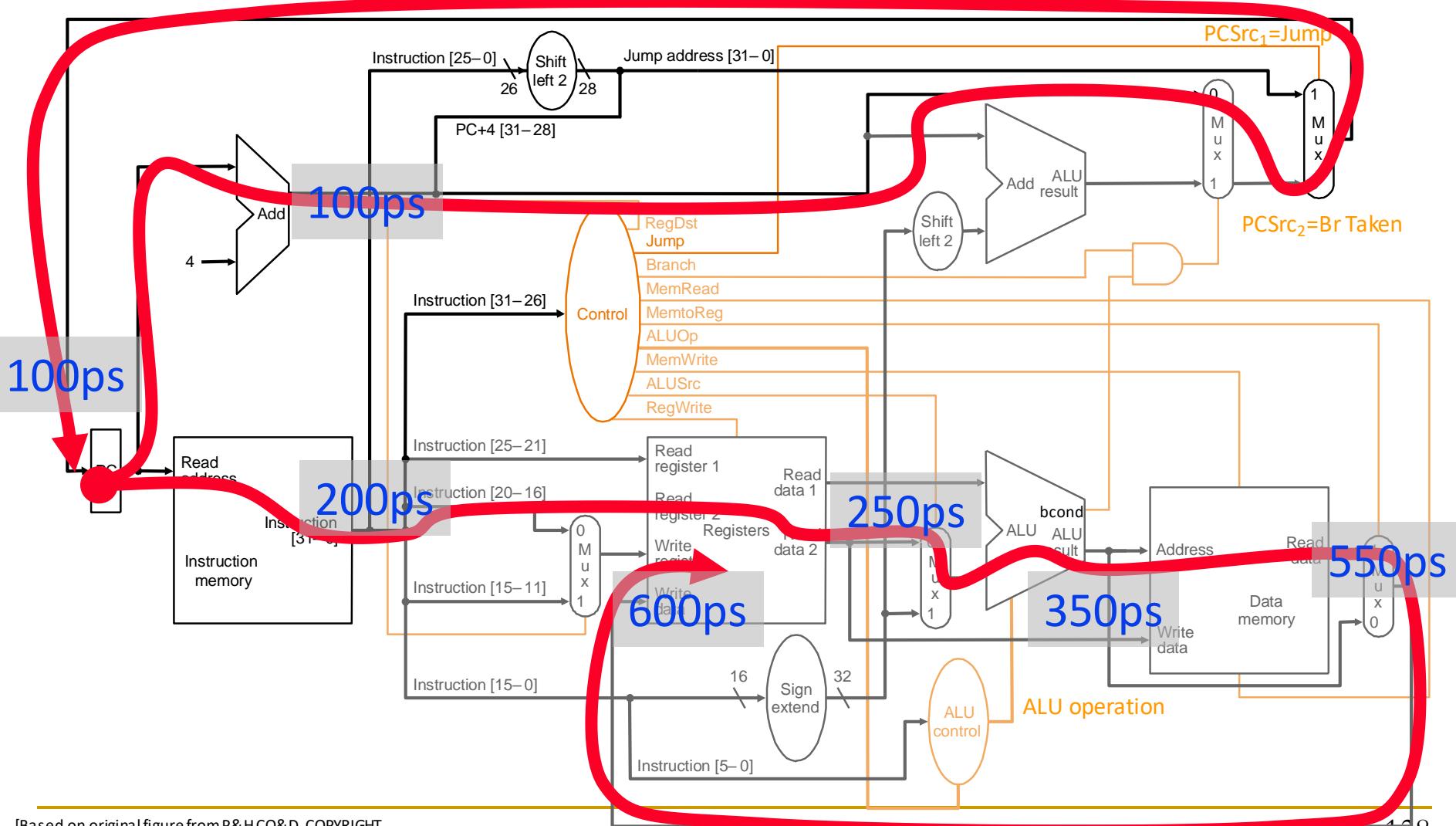
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

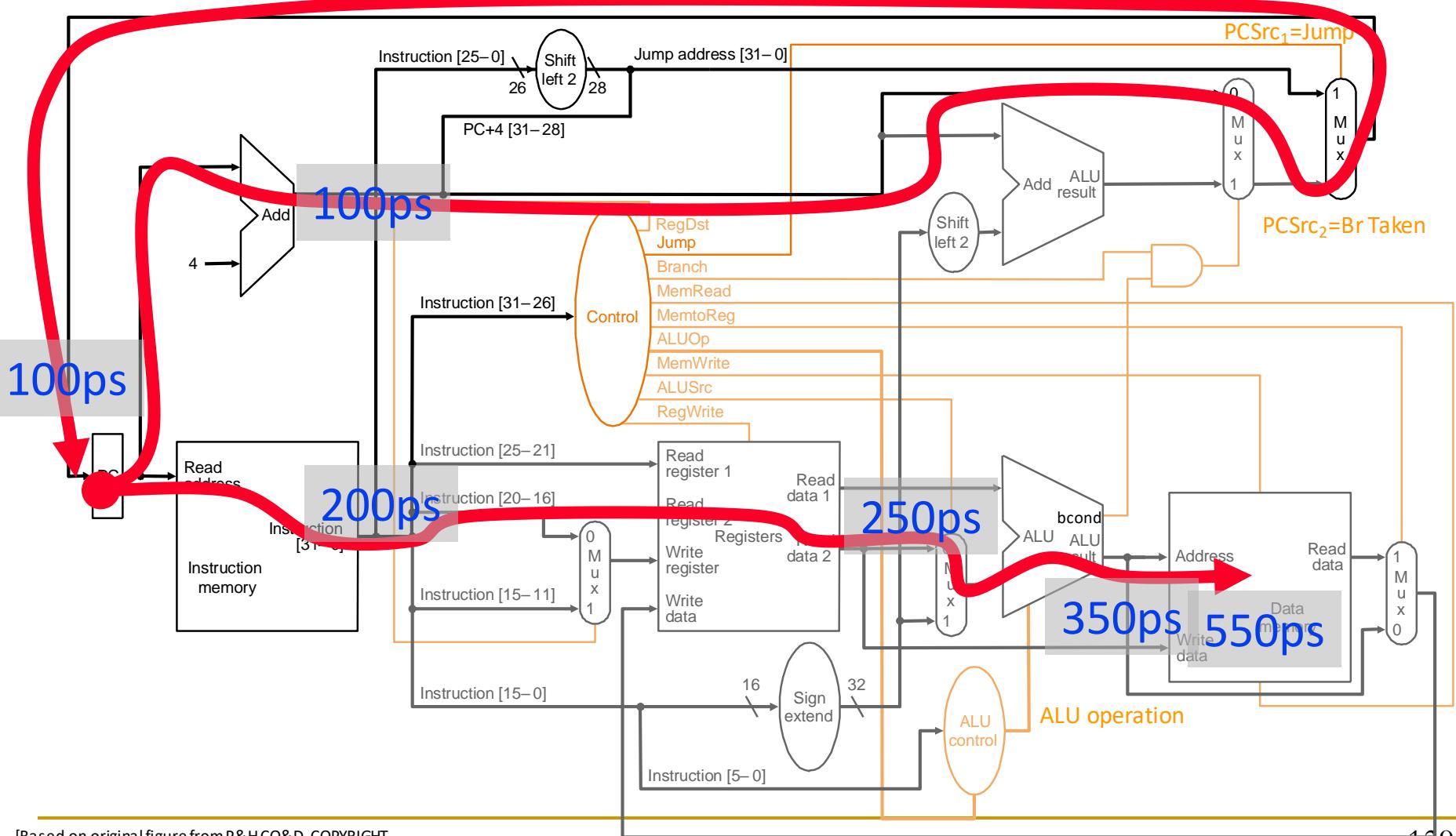
Let's Find the Critical Path



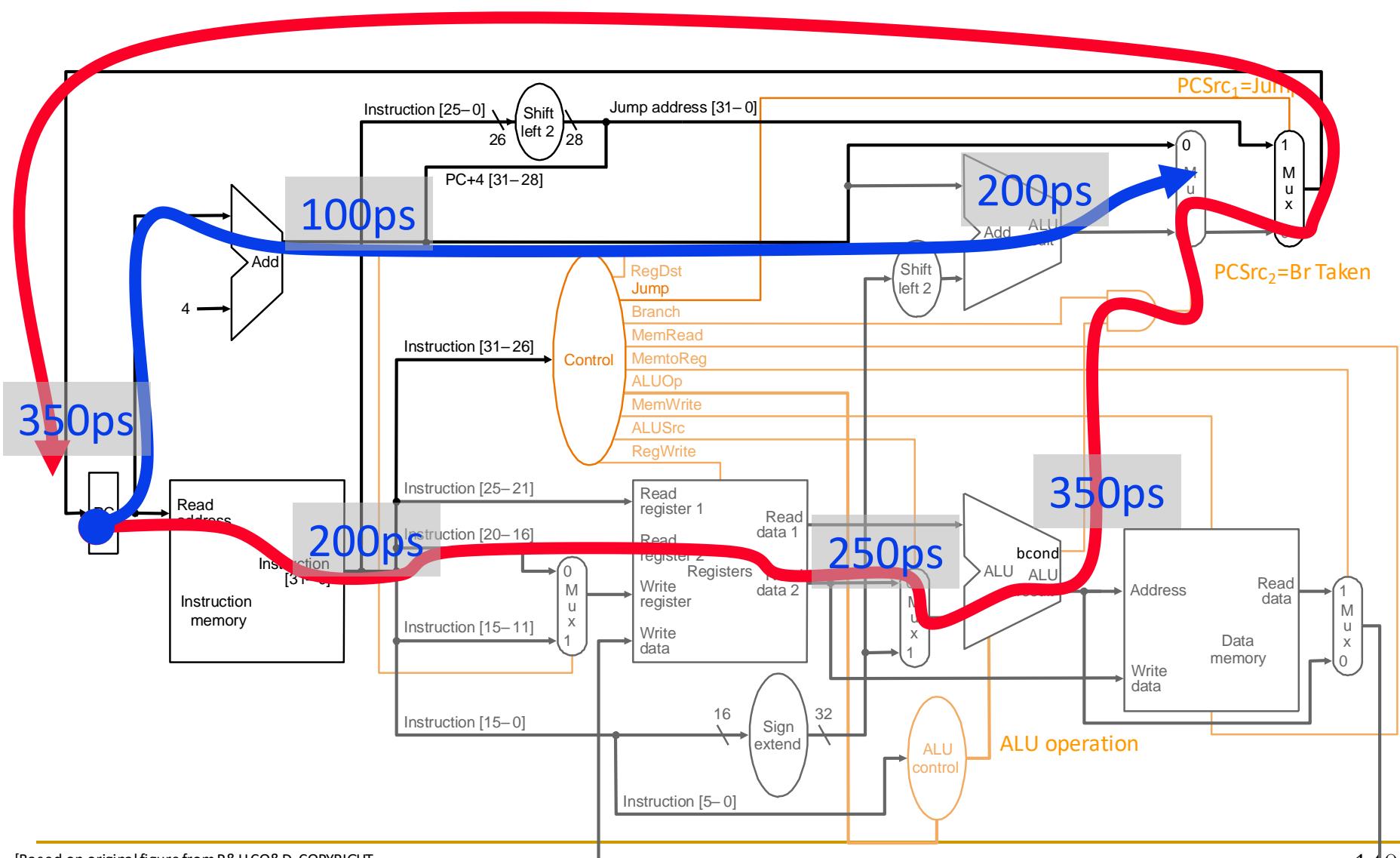
R-Type and I-Type ALU



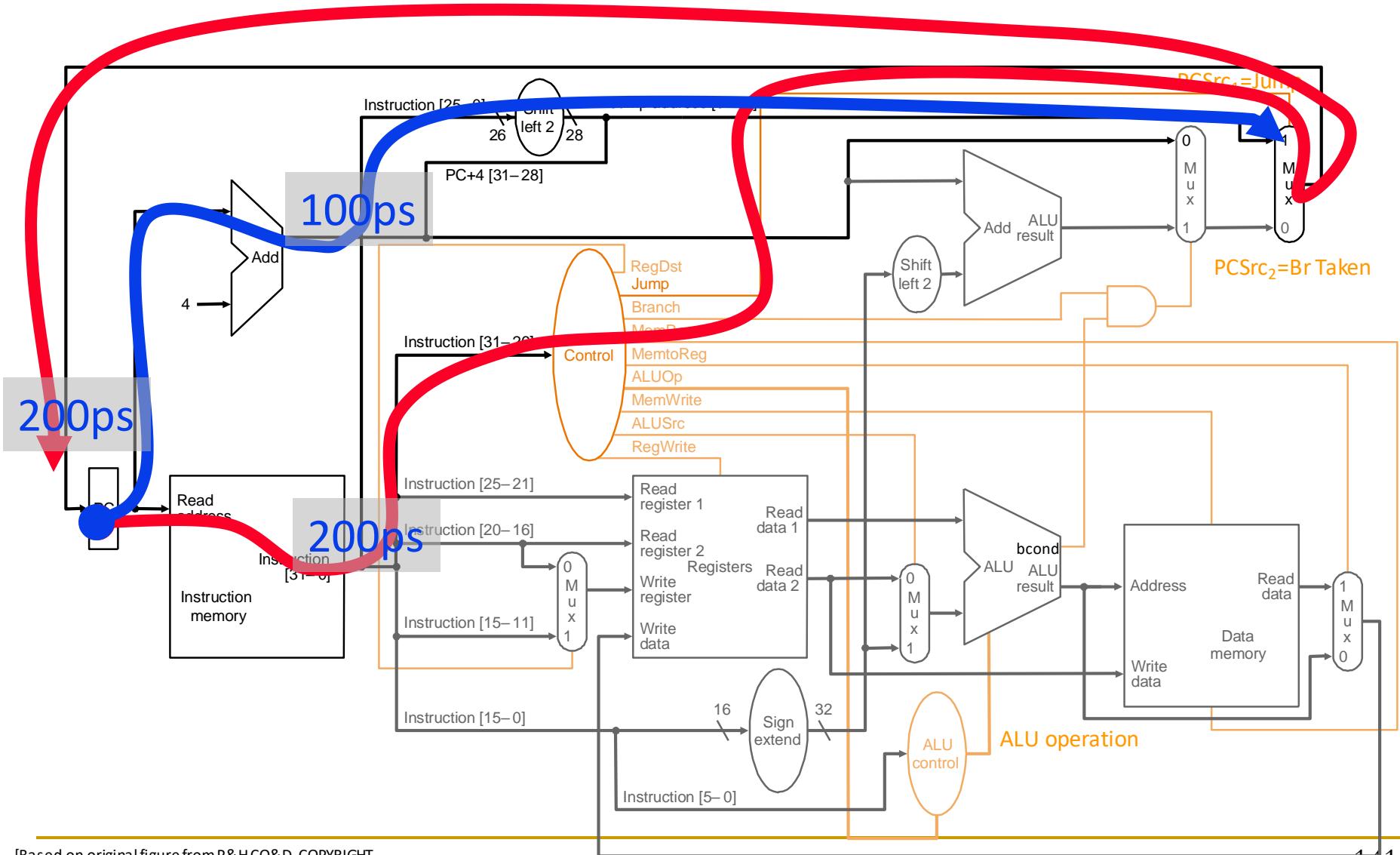




Branch Taken



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 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
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



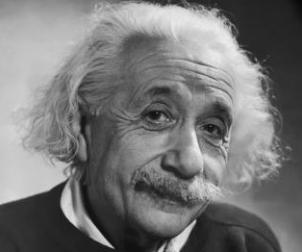
Recall: Takeaways

- 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;

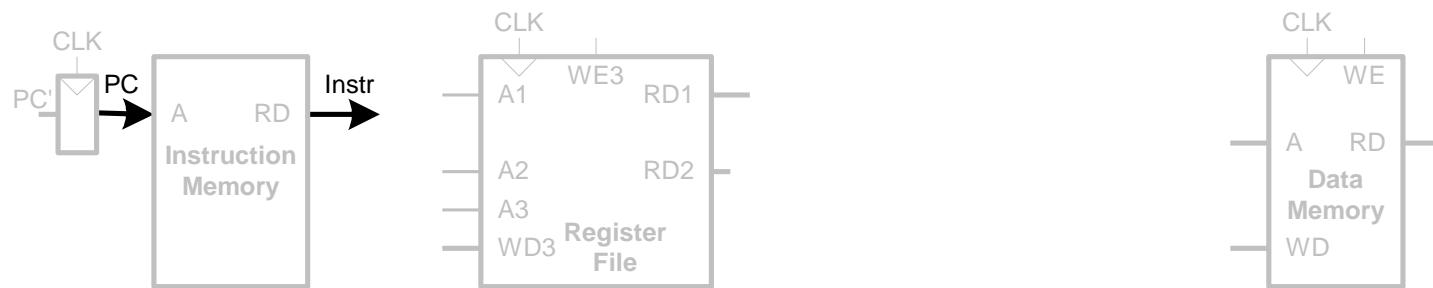
reg [31:0] M_arr [0:65535];           // Array for Memory

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

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

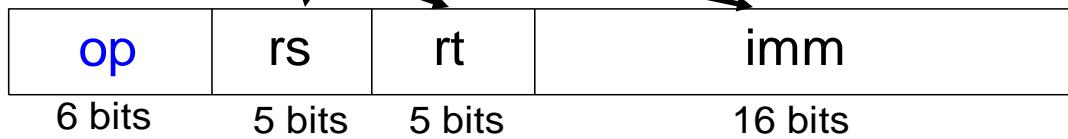
Single-Cycle Datapath: lw fetch

■ STEP 1: Fetch instruction



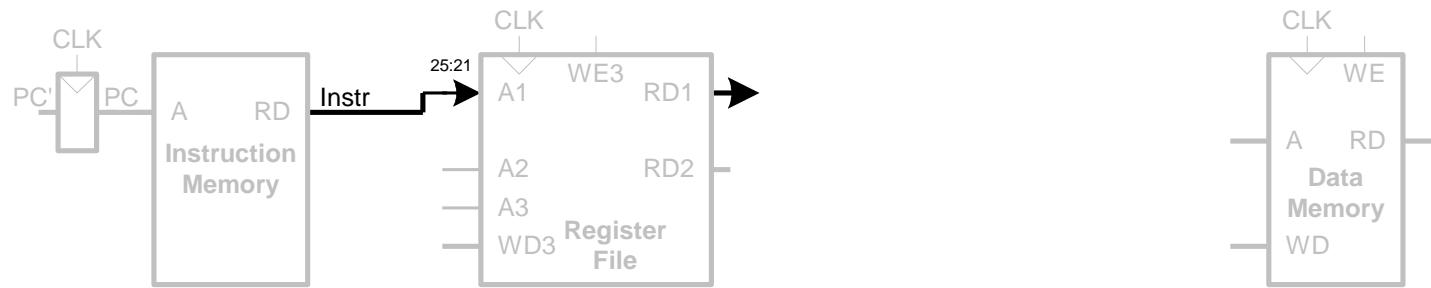
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type



Single-Cycle Datapath: lw register read

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



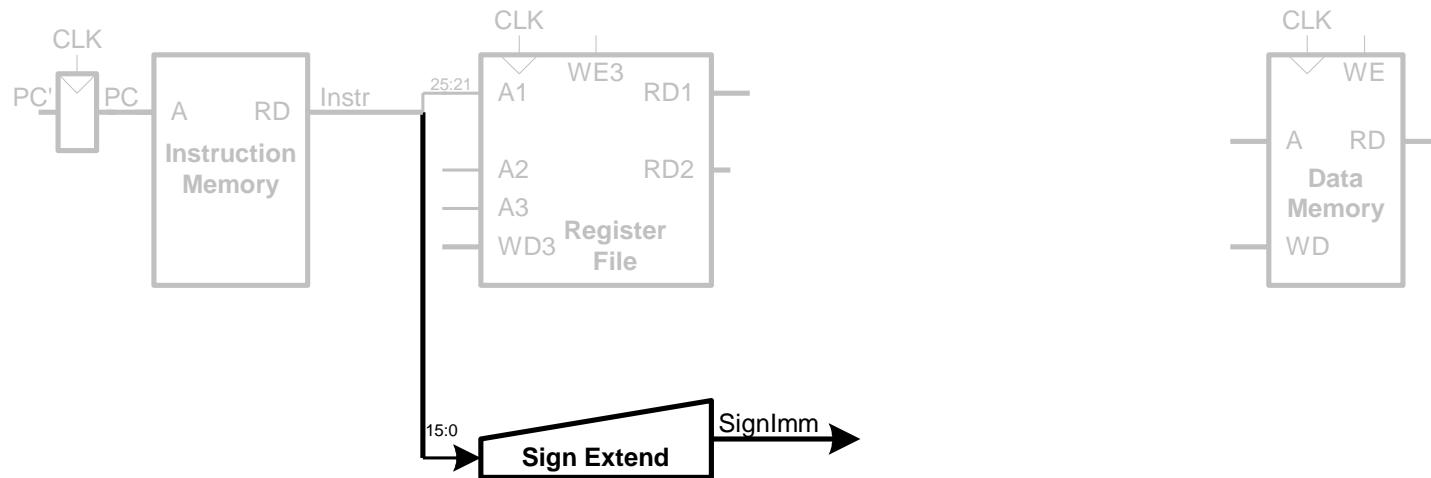
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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**



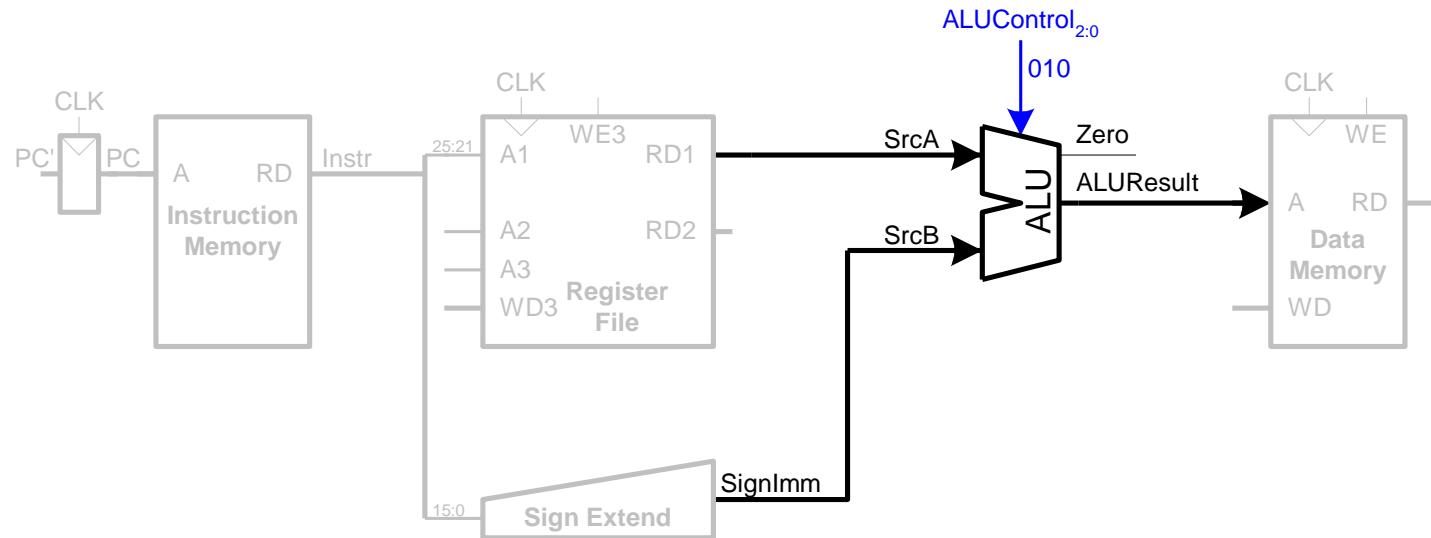
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



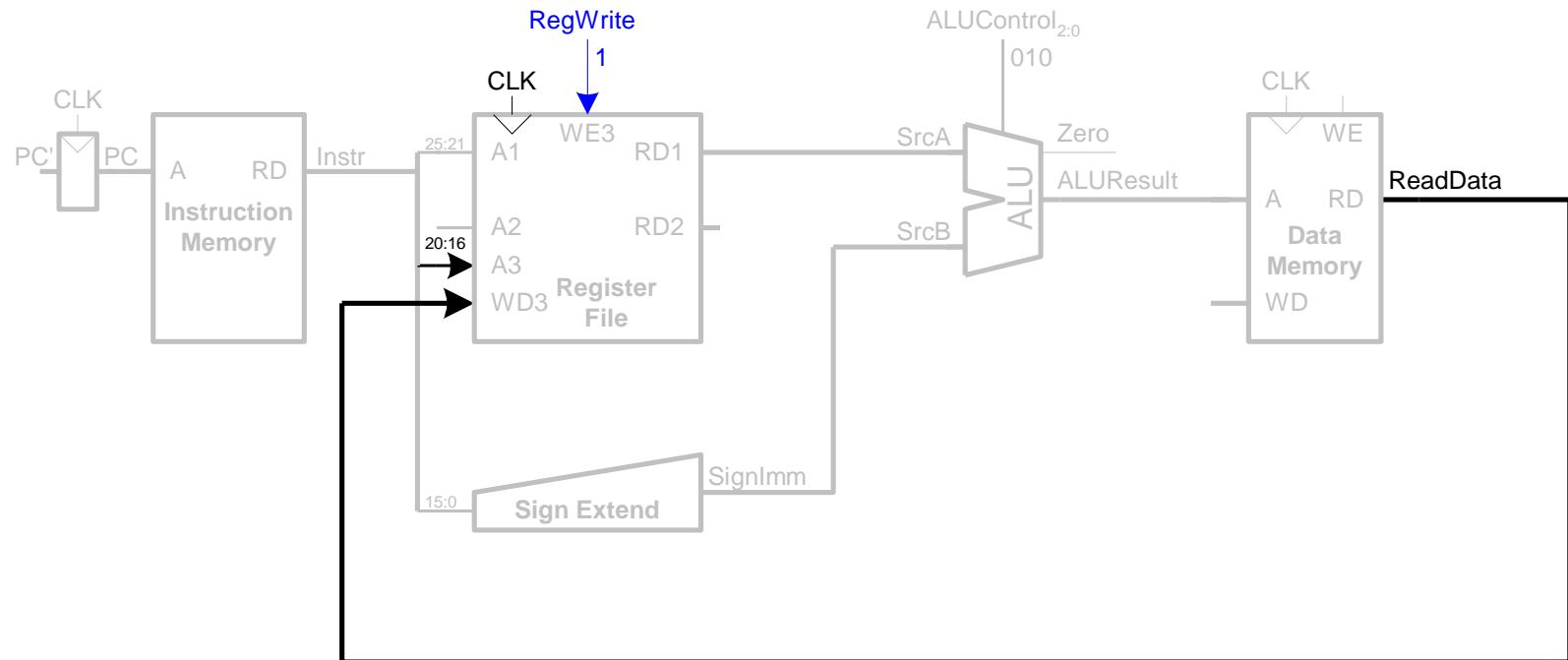
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



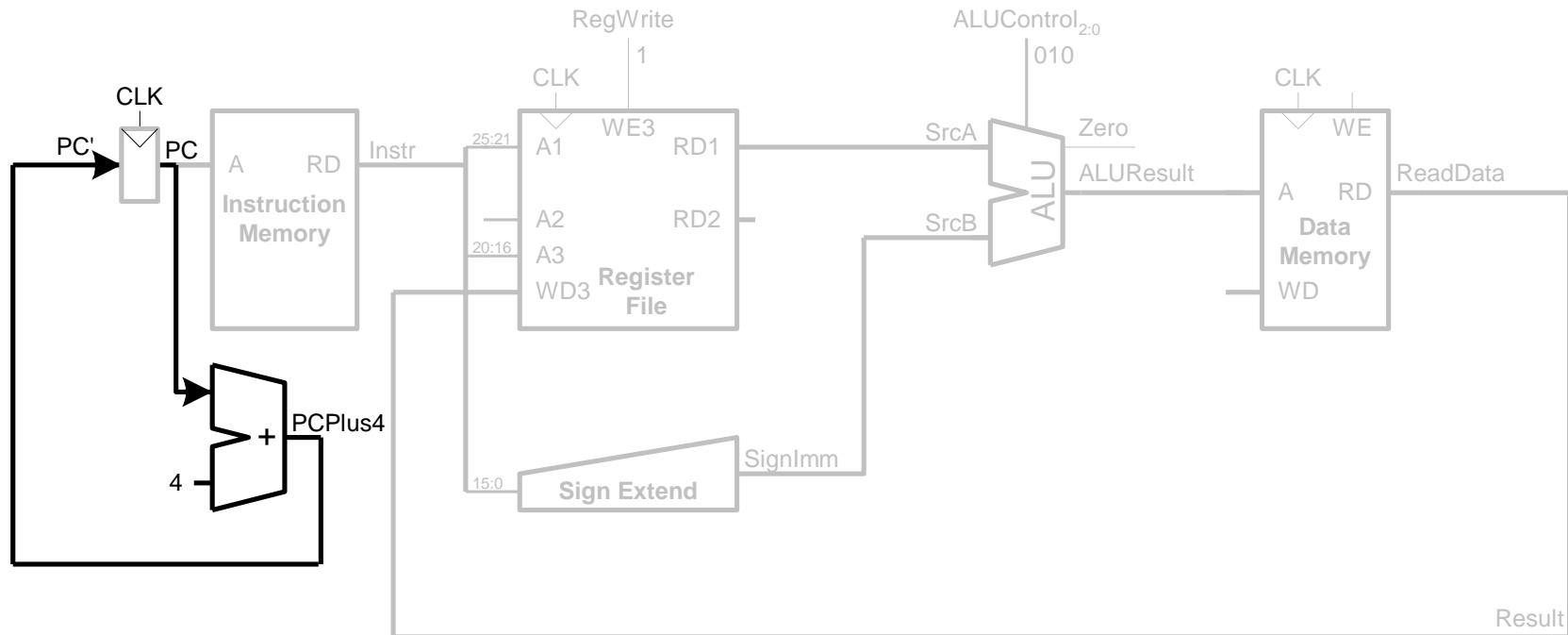
`lw $s3, 1($0) # read memory word 1 into $s3`

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



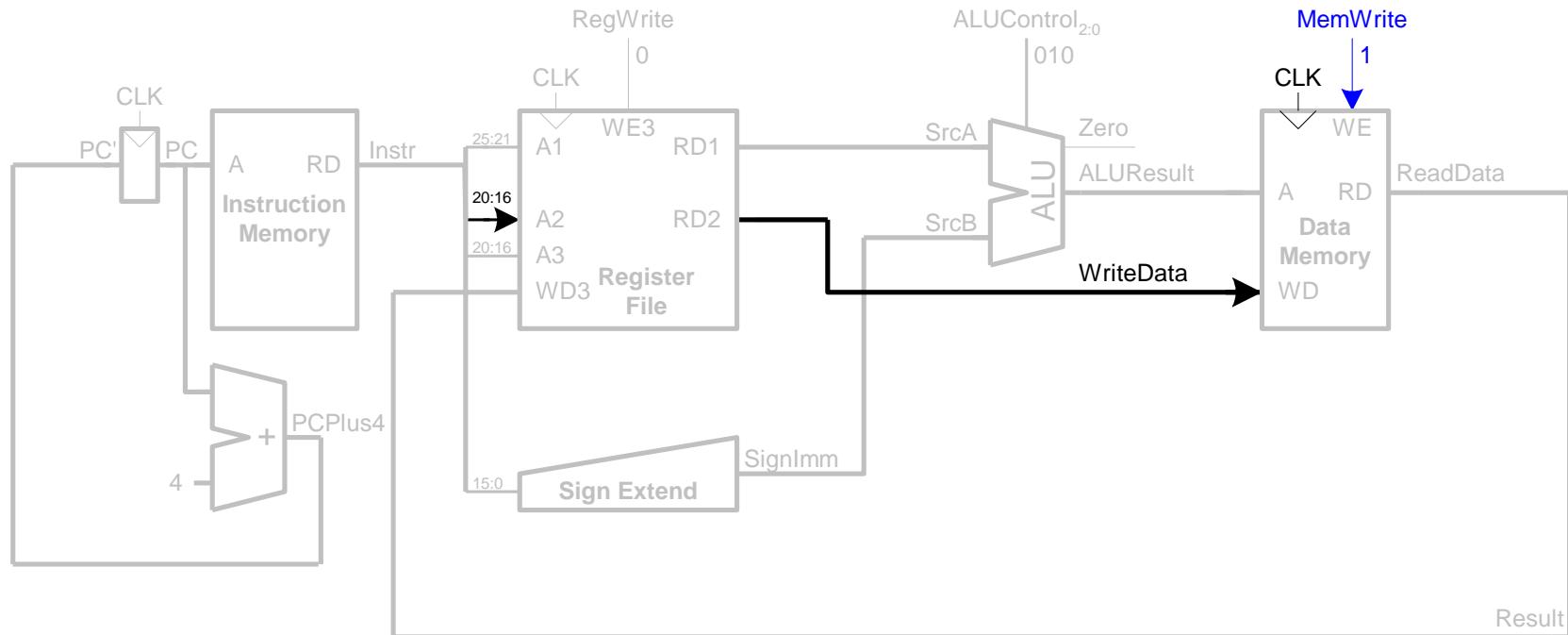
```
lw $s3, 1($0) # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: sw

■ Write data in rt to memory



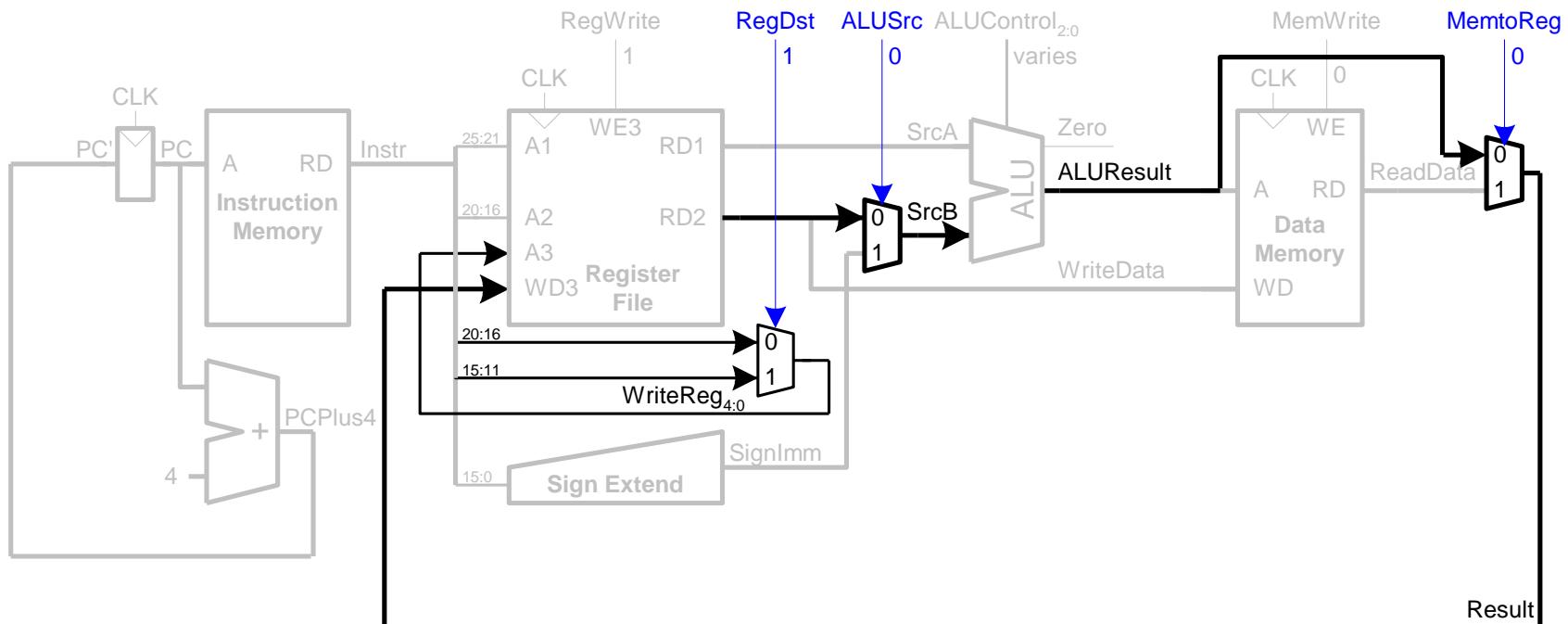
```
sw $t7, 44($0) # write t7 into memory address 44
```

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

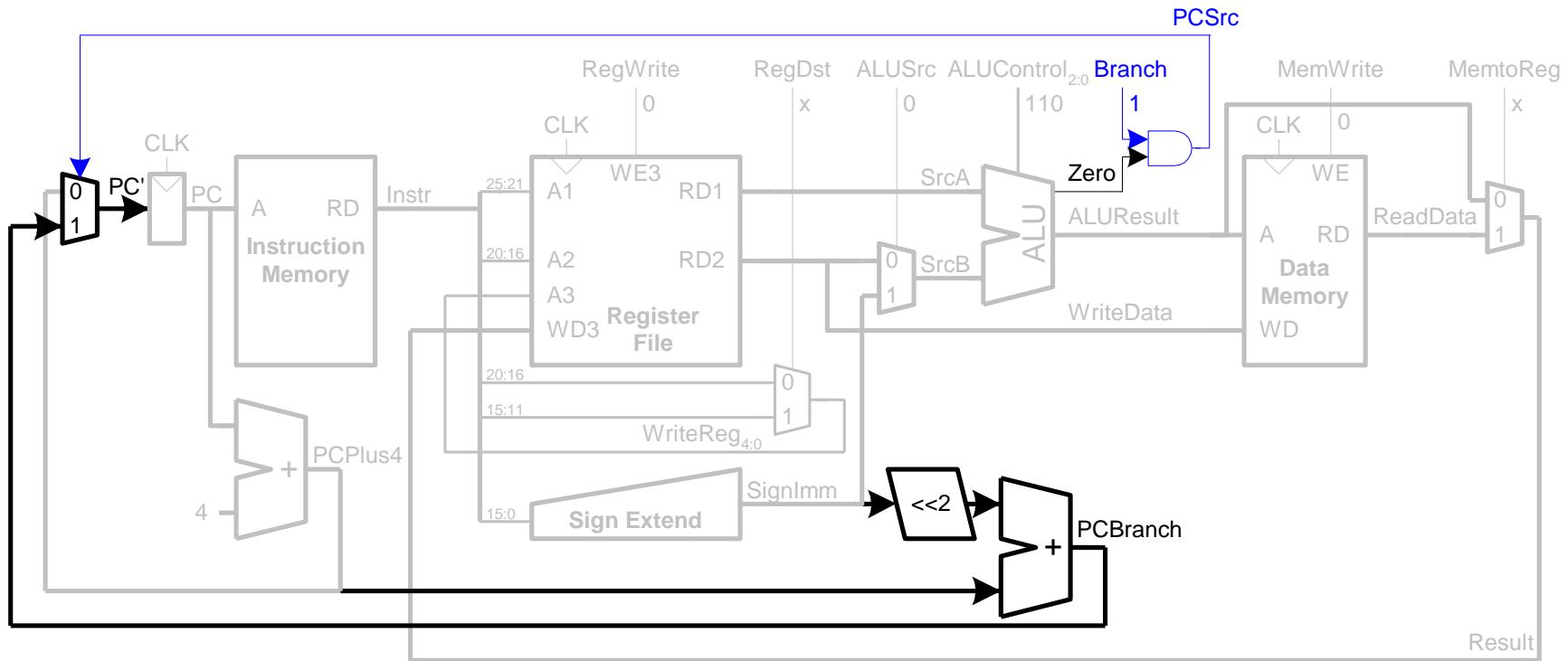


add t, b, c # t = b + c

R-Type

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

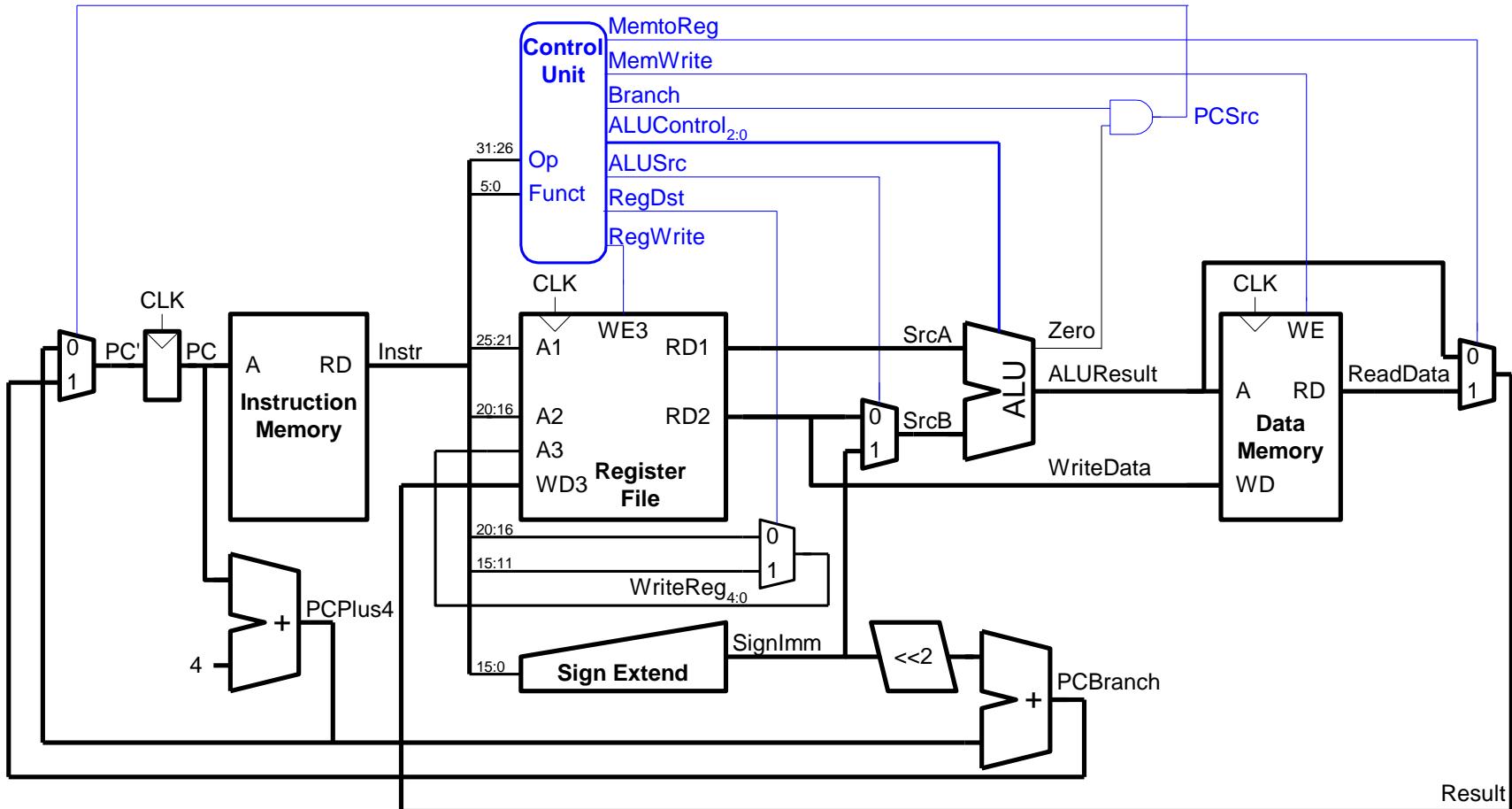
Single-Cycle Datapath: beq



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

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

Complete Single-Cycle Processor

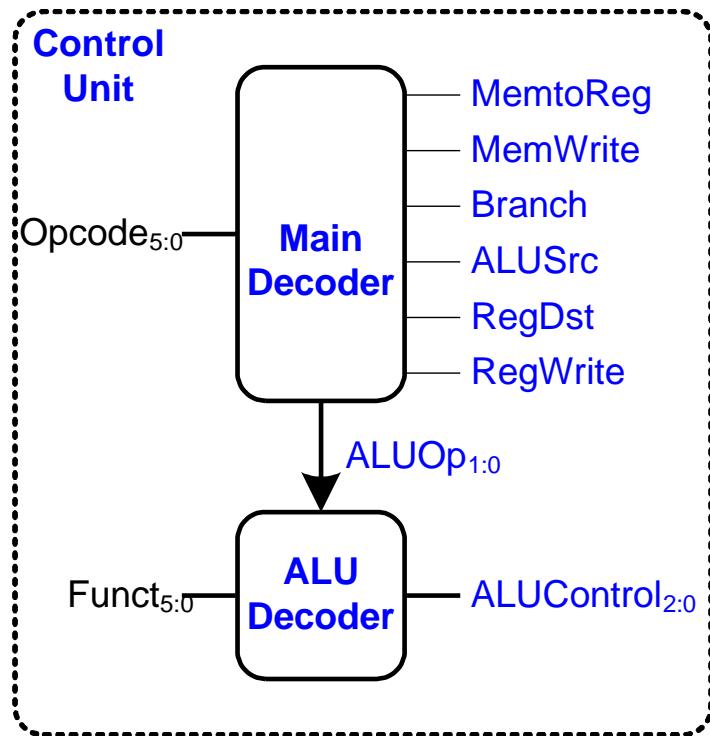


Our MIPS Datapath has Several Options

- **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*)

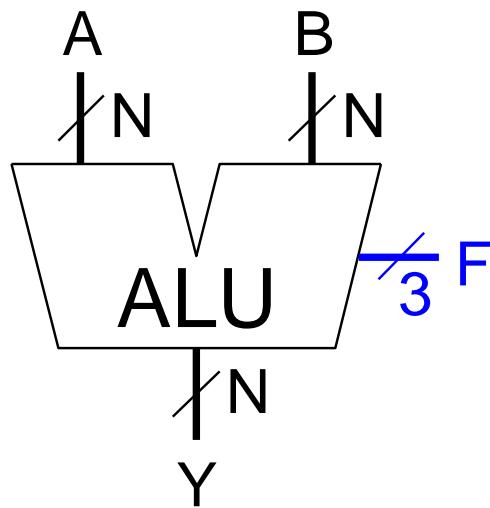
All these options are our control signals

Control Unit



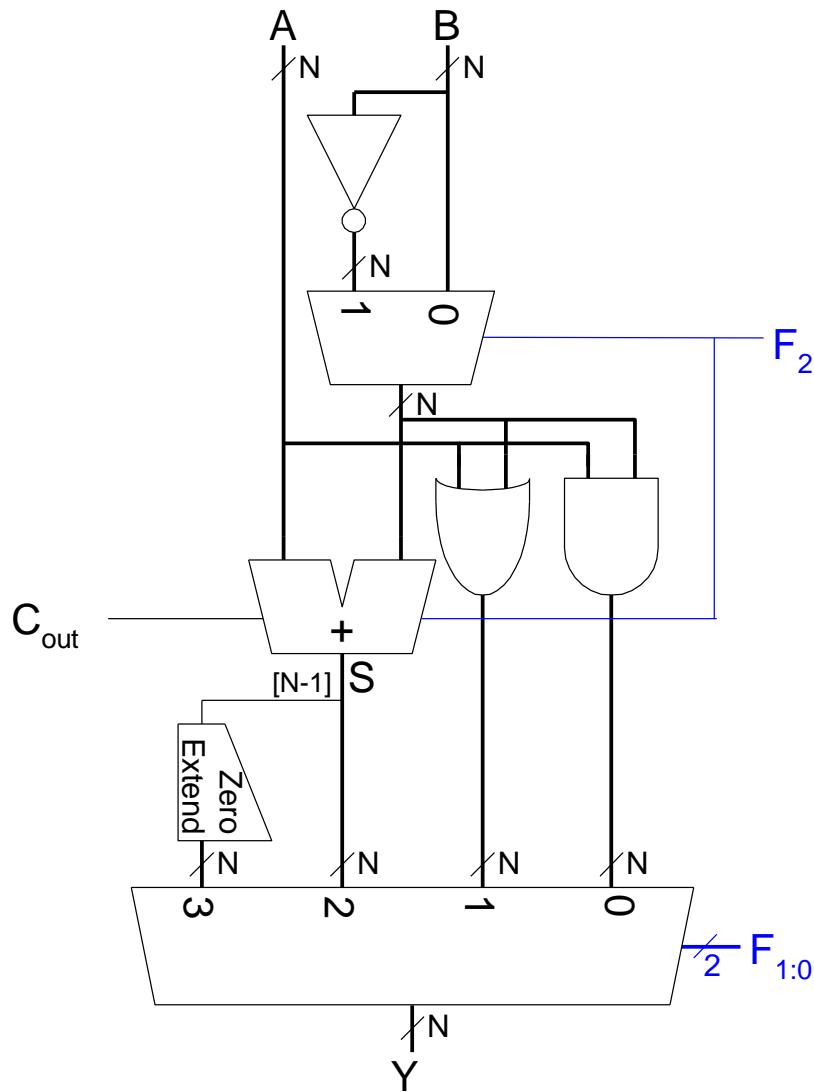
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



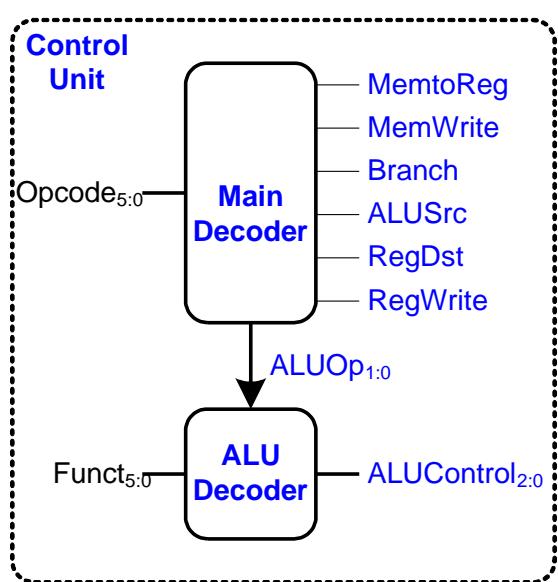
F _{2:0}	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & \sim B
101	A \sim B
110	A - B
111	SLT

ALU Internals



$F_{2:0}$	Function
000	$A \And B$
001	$A \Or B$
010	$A + B$
011	not used
100	$A \And \sim B$
101	$A \Or \sim B$
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

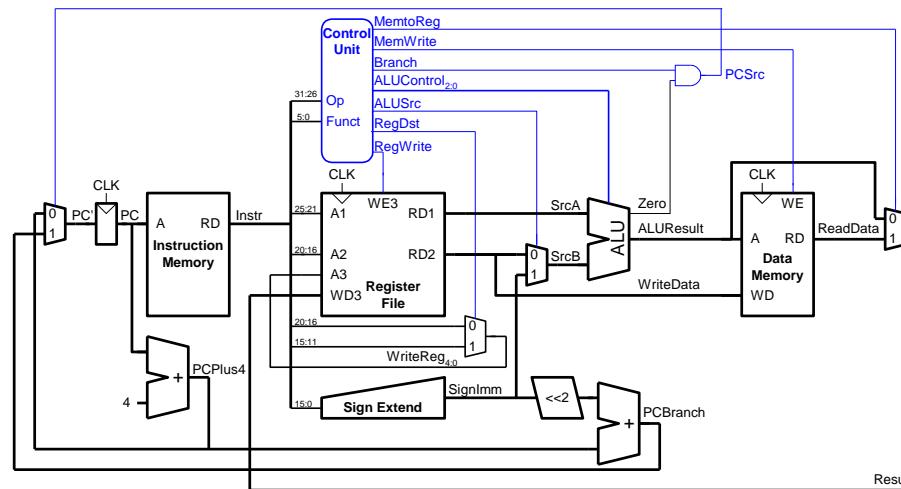
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

■ New Control Signal

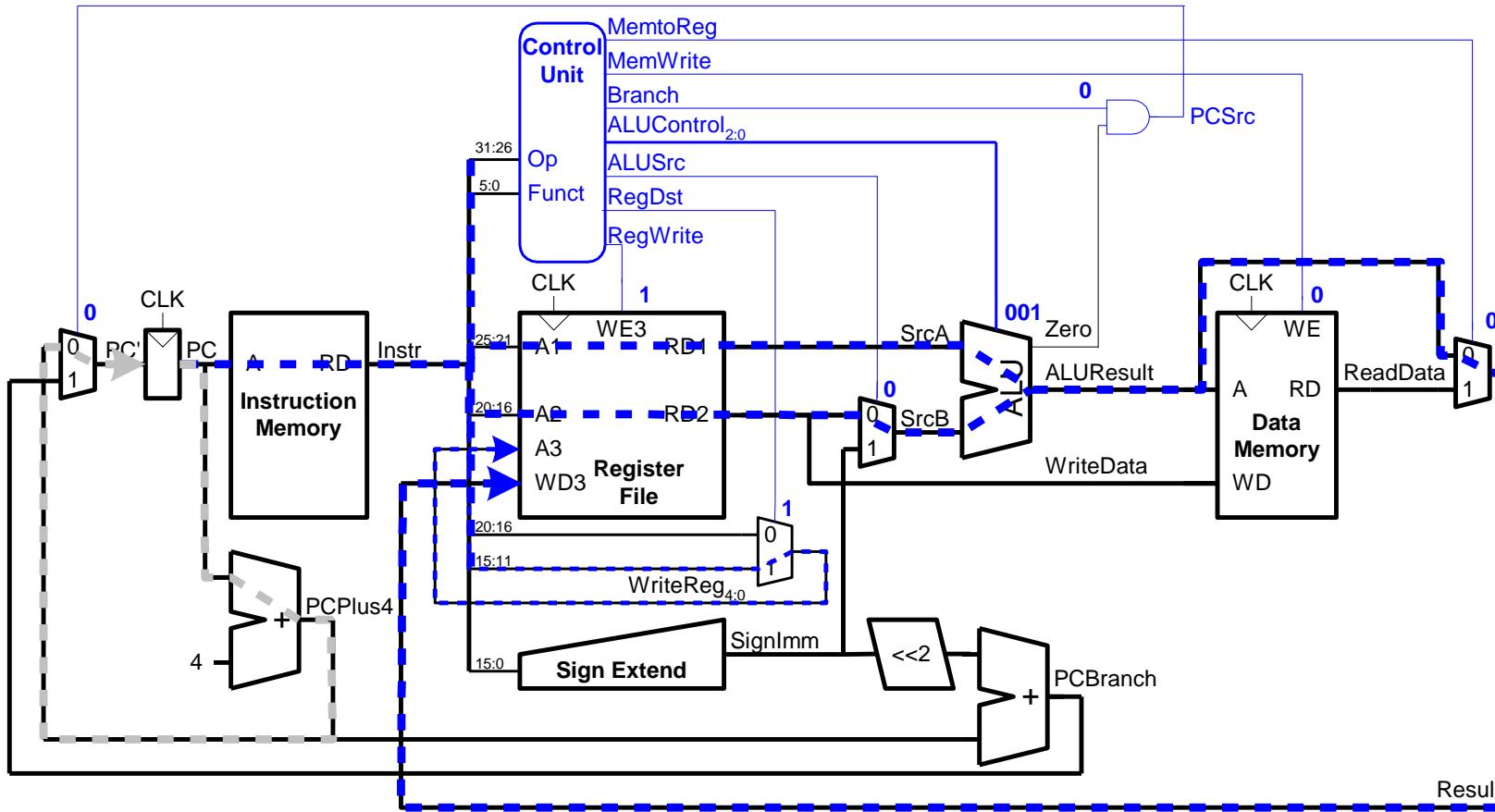
- *Branch*: Are we jumping or not ?

Control Unit: Main Decoder

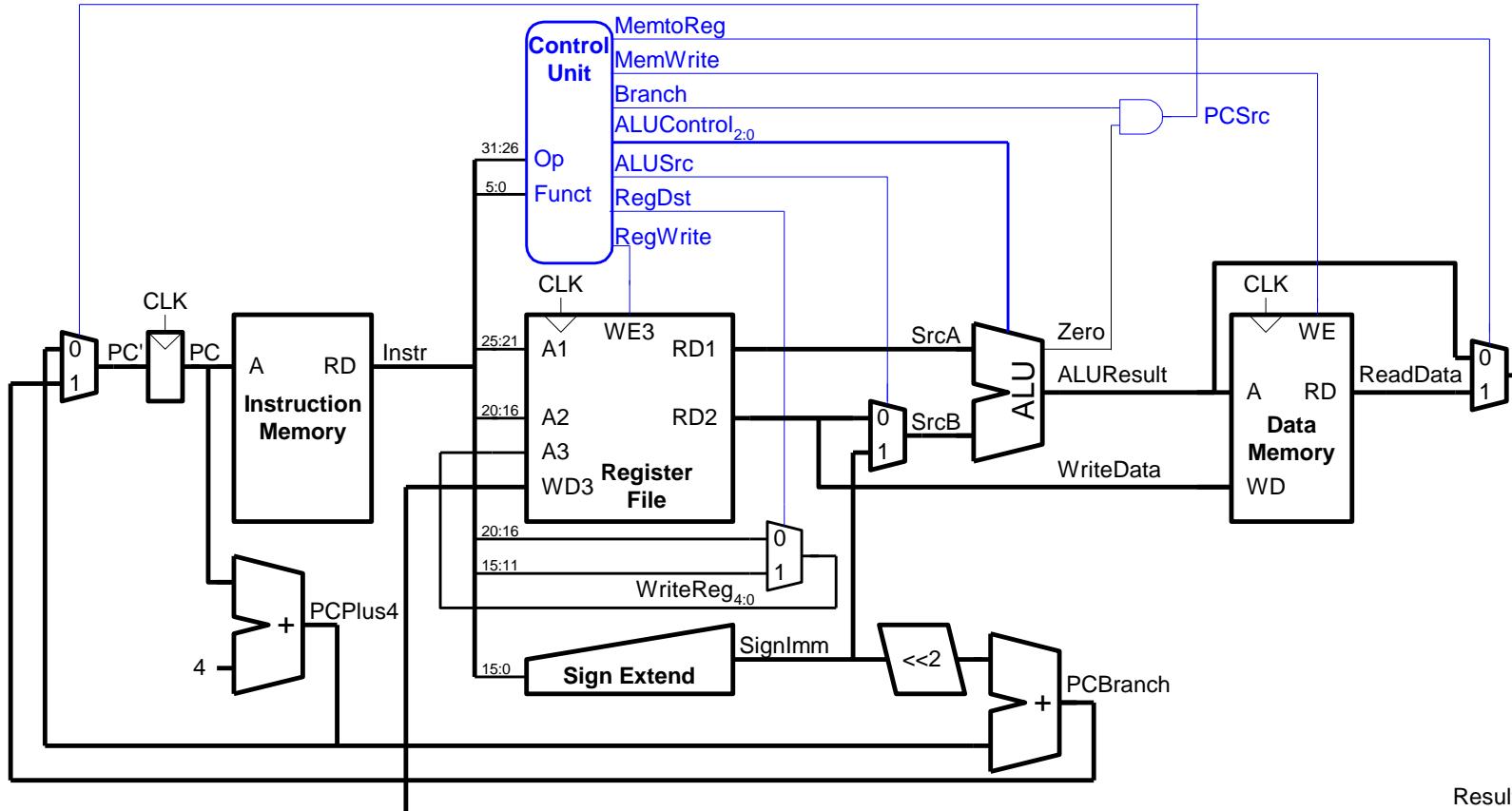
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

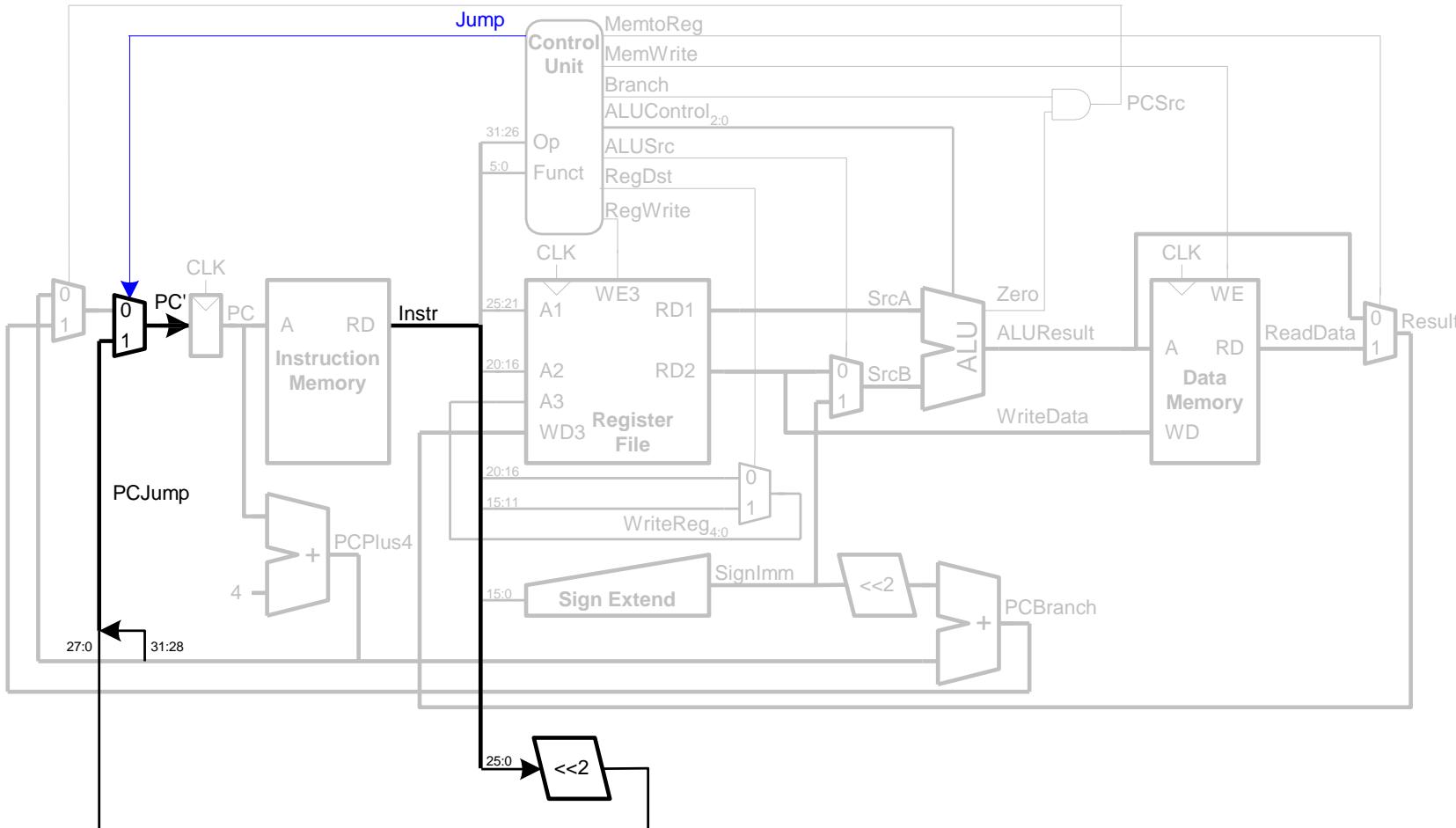


- No change to datapath

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

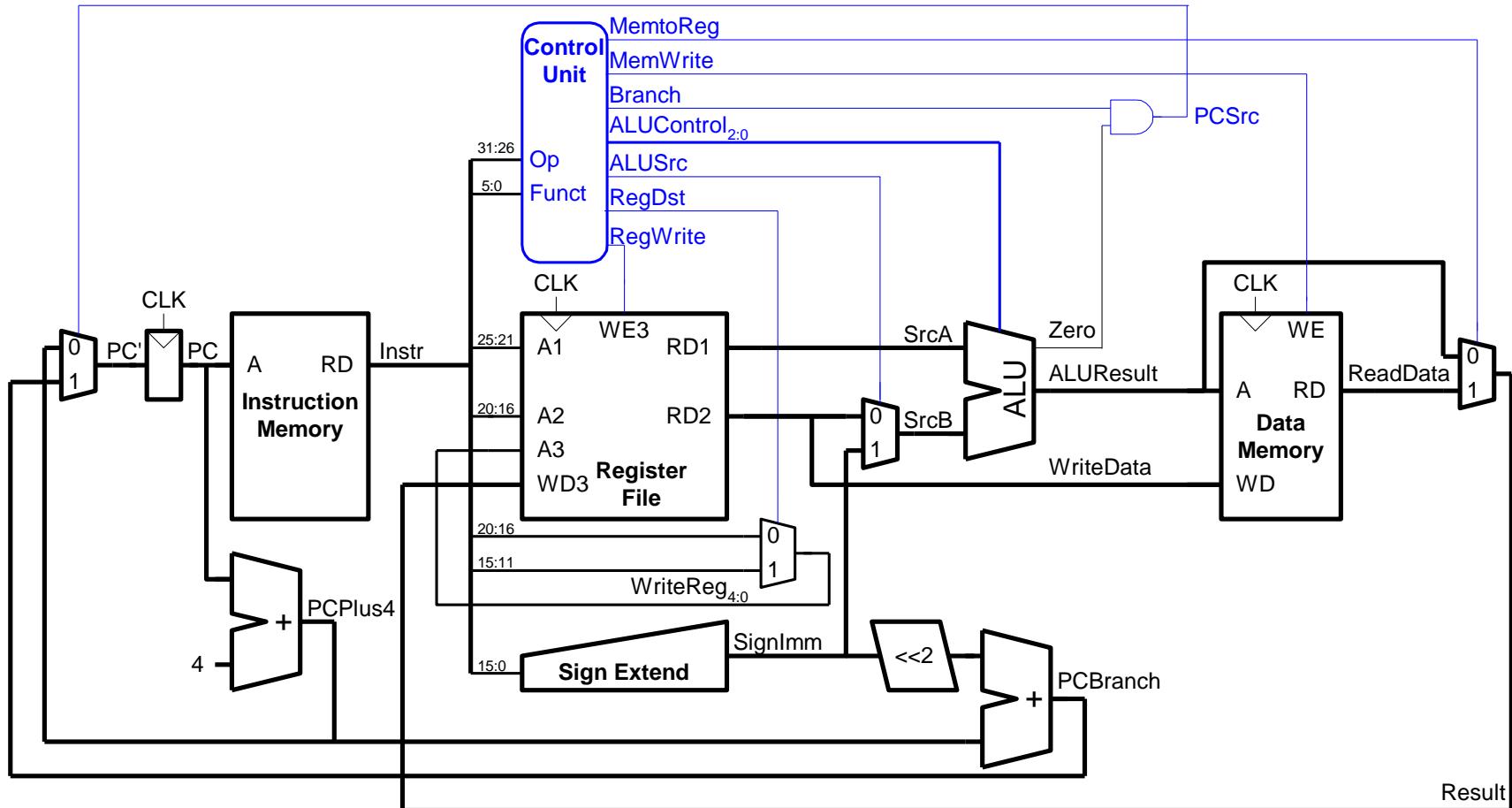
Extended Functionality: j



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

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

How can I Make the Program Run Faster?

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

How can I Make the Program Run Faster?

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

- **Reduce the number of instructions**

- Make instructions that 'do' more (CISC)
 - Use better compilers

How can I Make the Program Run Faster?

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

- **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

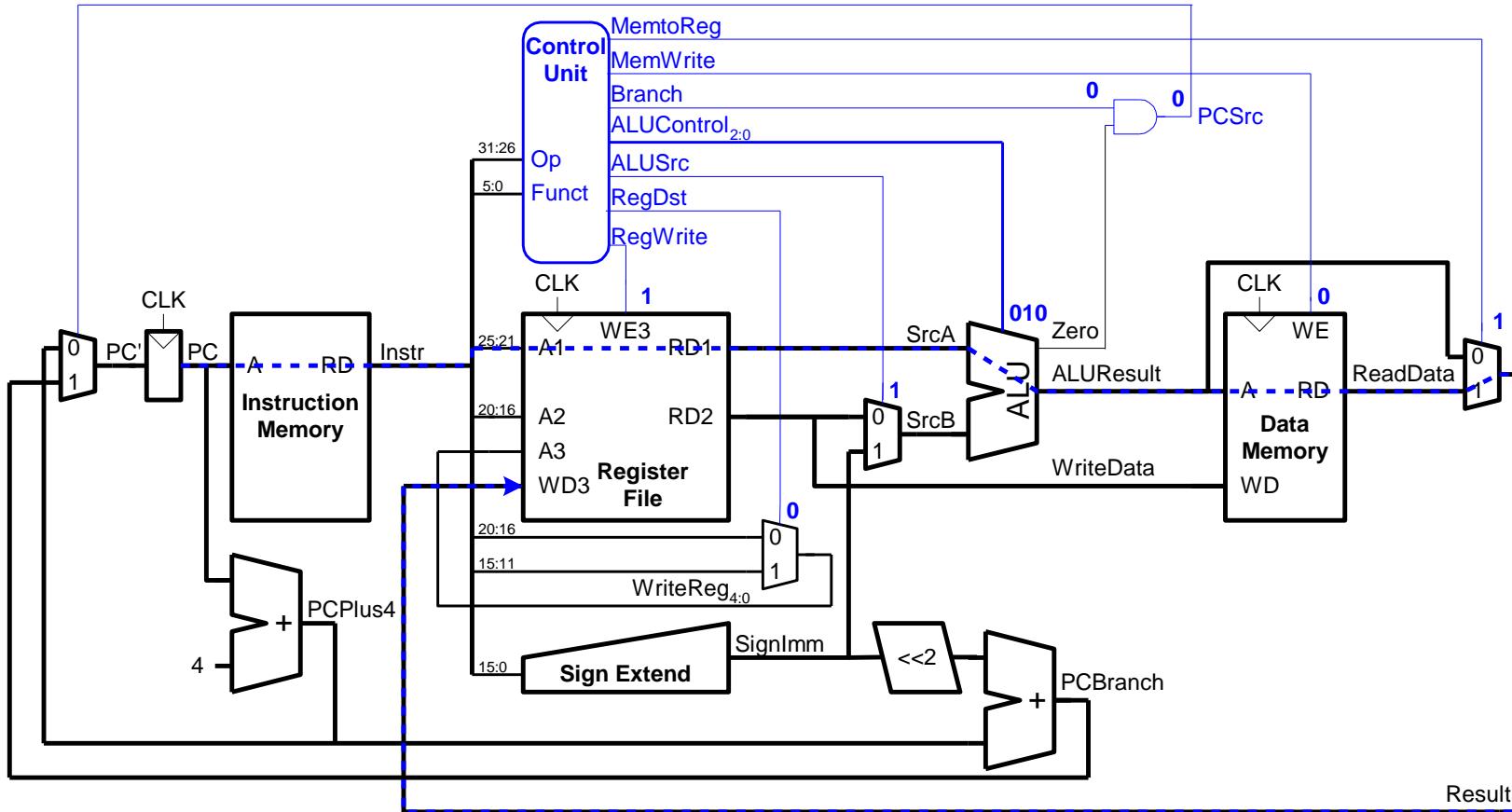
How can I Make the Program Run Faster?

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

- **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 (lw)



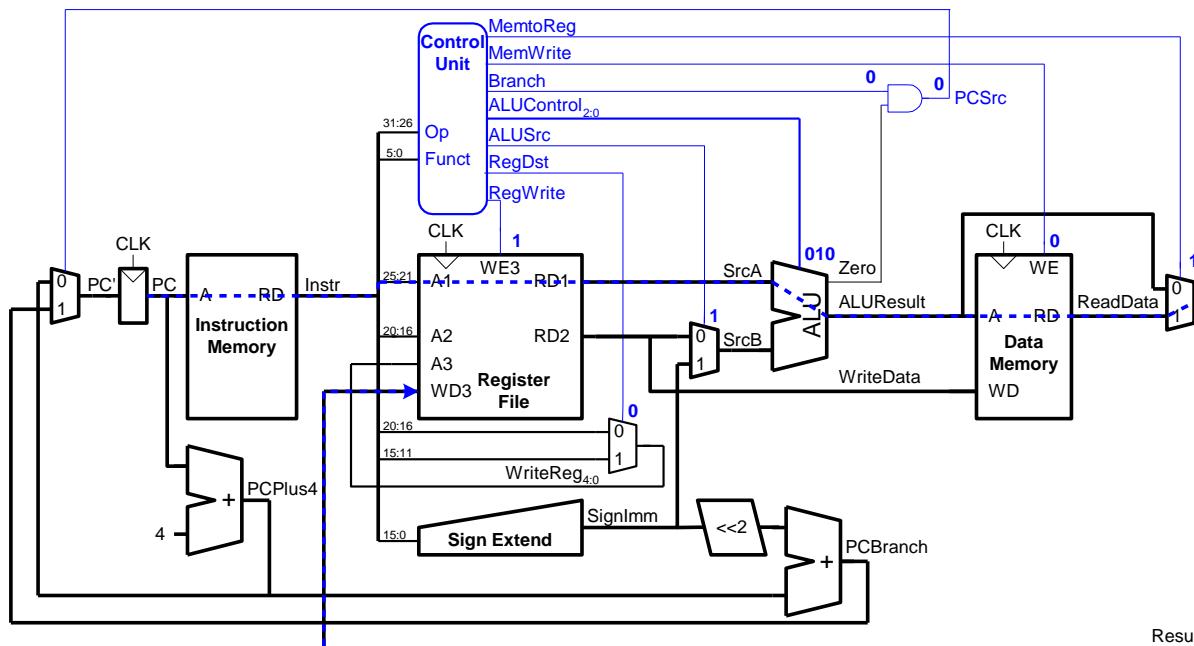
Single-Cycle Performance

■ Single-cycle critical path:

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

■ In most implementations, limiting paths are:

- memory, ALU, register file.
- $T_c = t_{\text{pcq_PC}} + 2t_{\text{mem}} + t_{\text{RFread}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{RFsetup}}$



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:

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

Digital Design & Computer Arch.

Lecture 12: Microarchitecture Fundamentals II

Prof. Onur Mutlu

ETH Zürich
Spring 2021
15 April 2021

Readings

- **Last time and today**
 - 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
 - **Tomorrow and next week**
 - Pipelining
 - H&H, Chapter 7.5
 - Pipelining Issues
 - H&H, Chapter 7.8.1-7.8.3
-

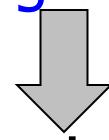
Agenda for Today & Next Few Lectures

- 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

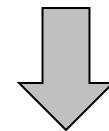
Recall: A Very Basic Instruction Processing Engine

- 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

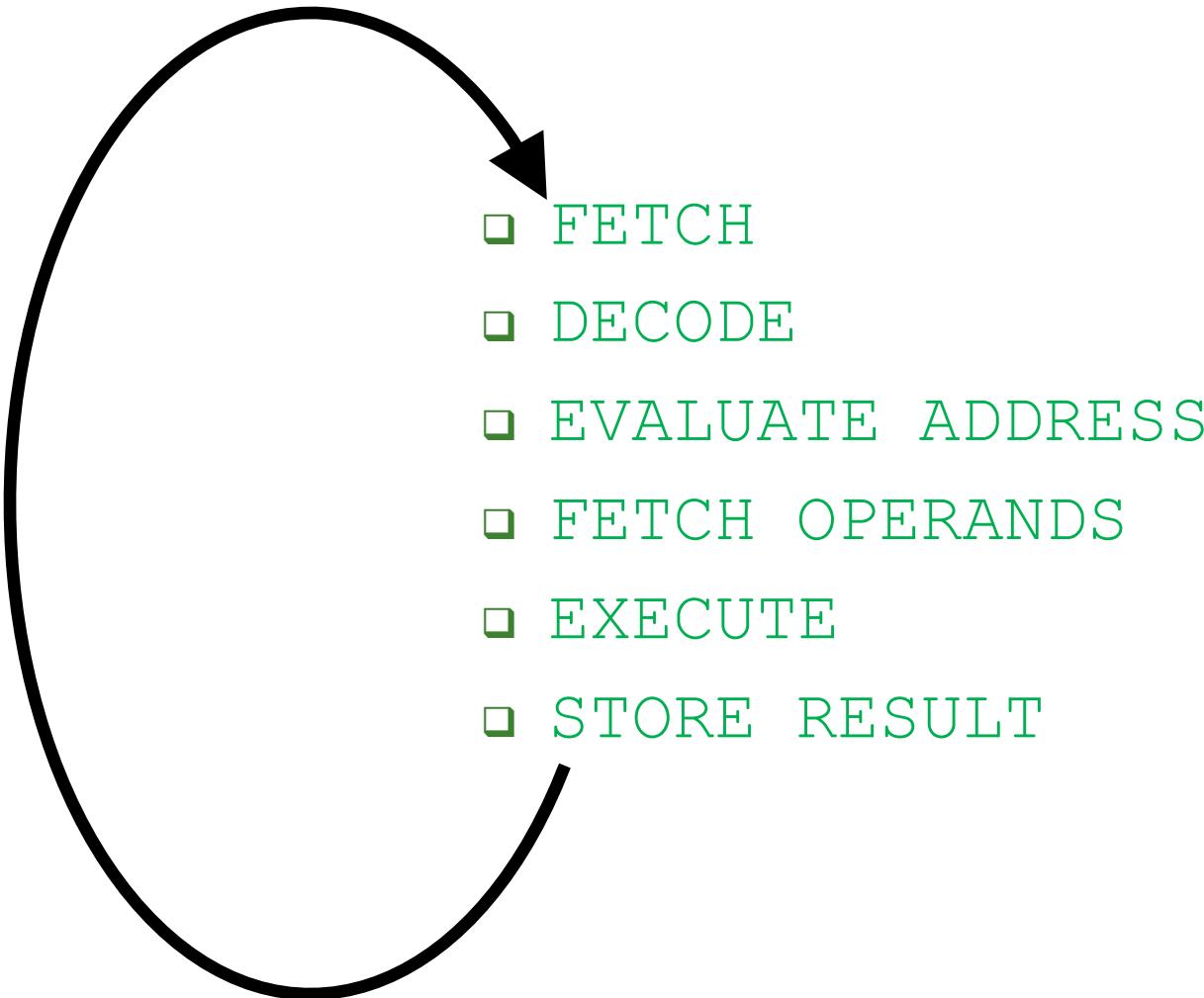


Process instruction in one clock cycle



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

Recall: The Instruction Processing “Cycle”

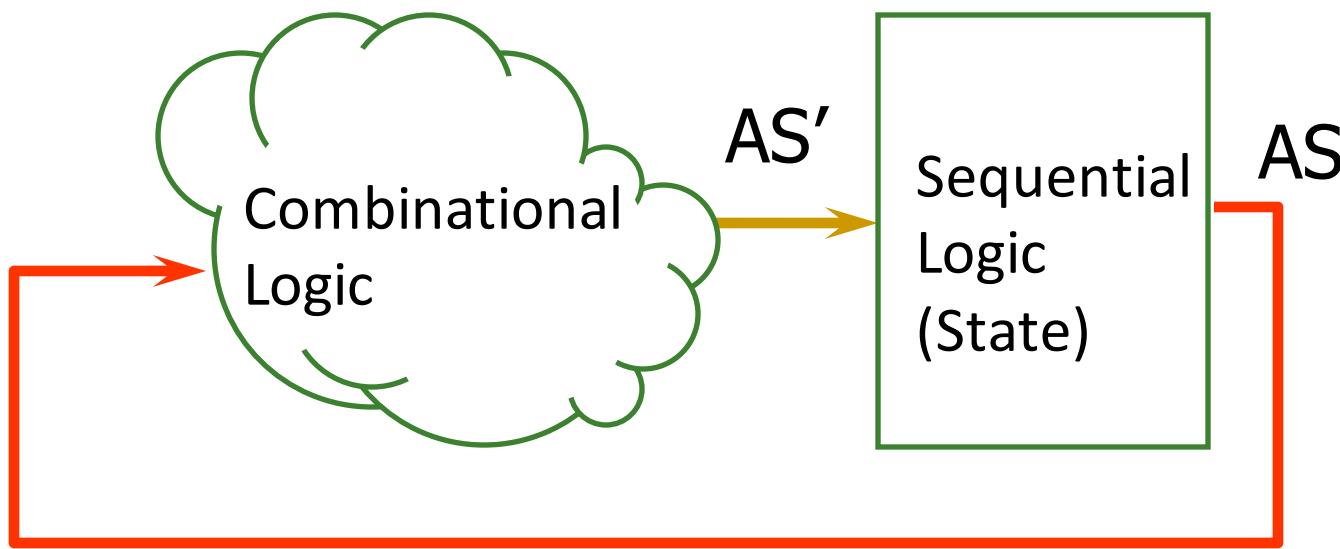


Instruction Processing “Cycle” vs. Machine Clock Cycle

- **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

Recall: Single-Cycle Machine

- Single-cycle machine

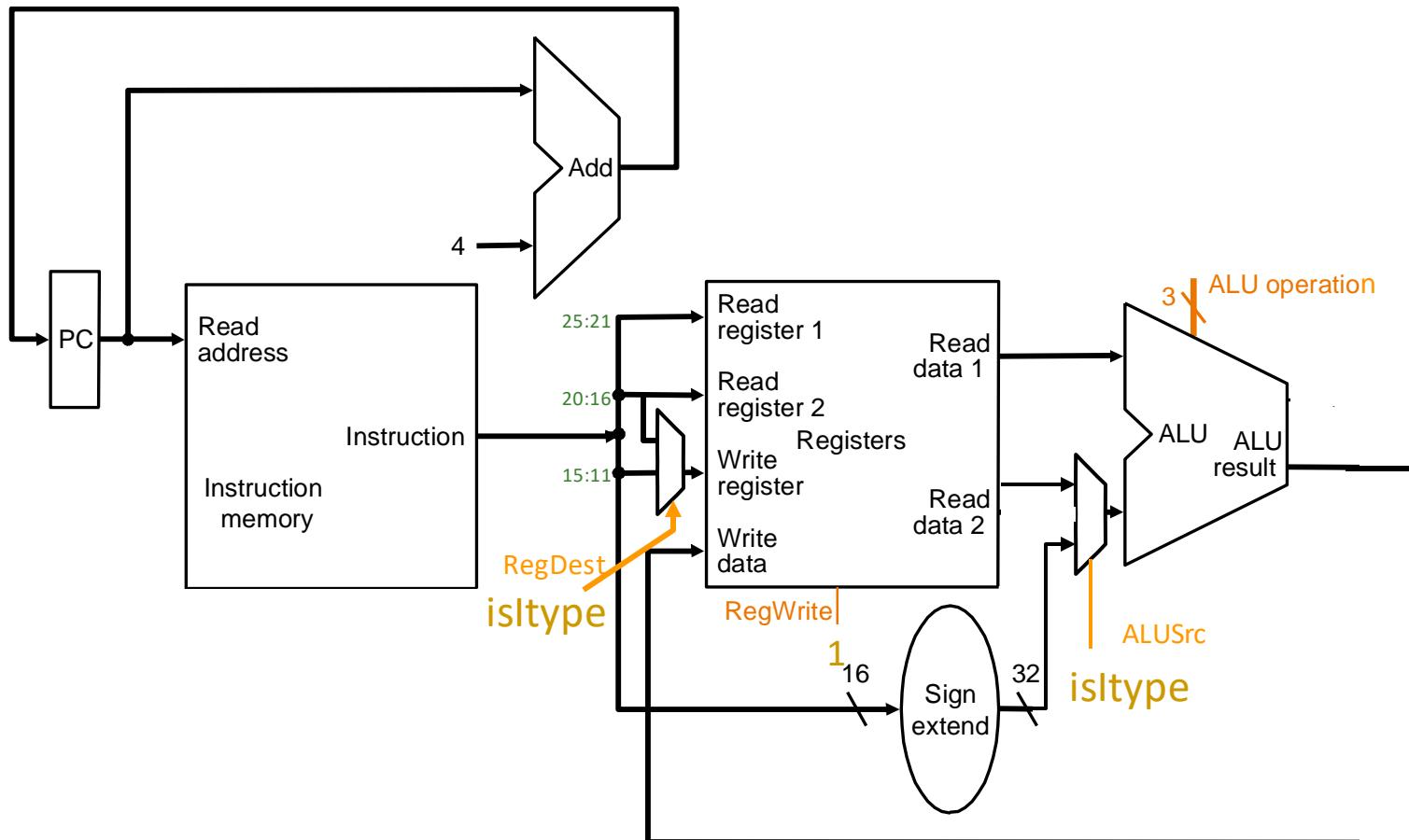


Recall: Datapath and Control Logic

- 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

Single-Cycle Datapath for *Arithmetic and Logical Instructions*

Datapath for R- and I-Type ALU Insts.



if $MEM[PC] == ADDI\ rt\ rs\ immediate$

$GPR[rt] \leftarrow GPR[rs] + \text{sign-extend (immediate)}$

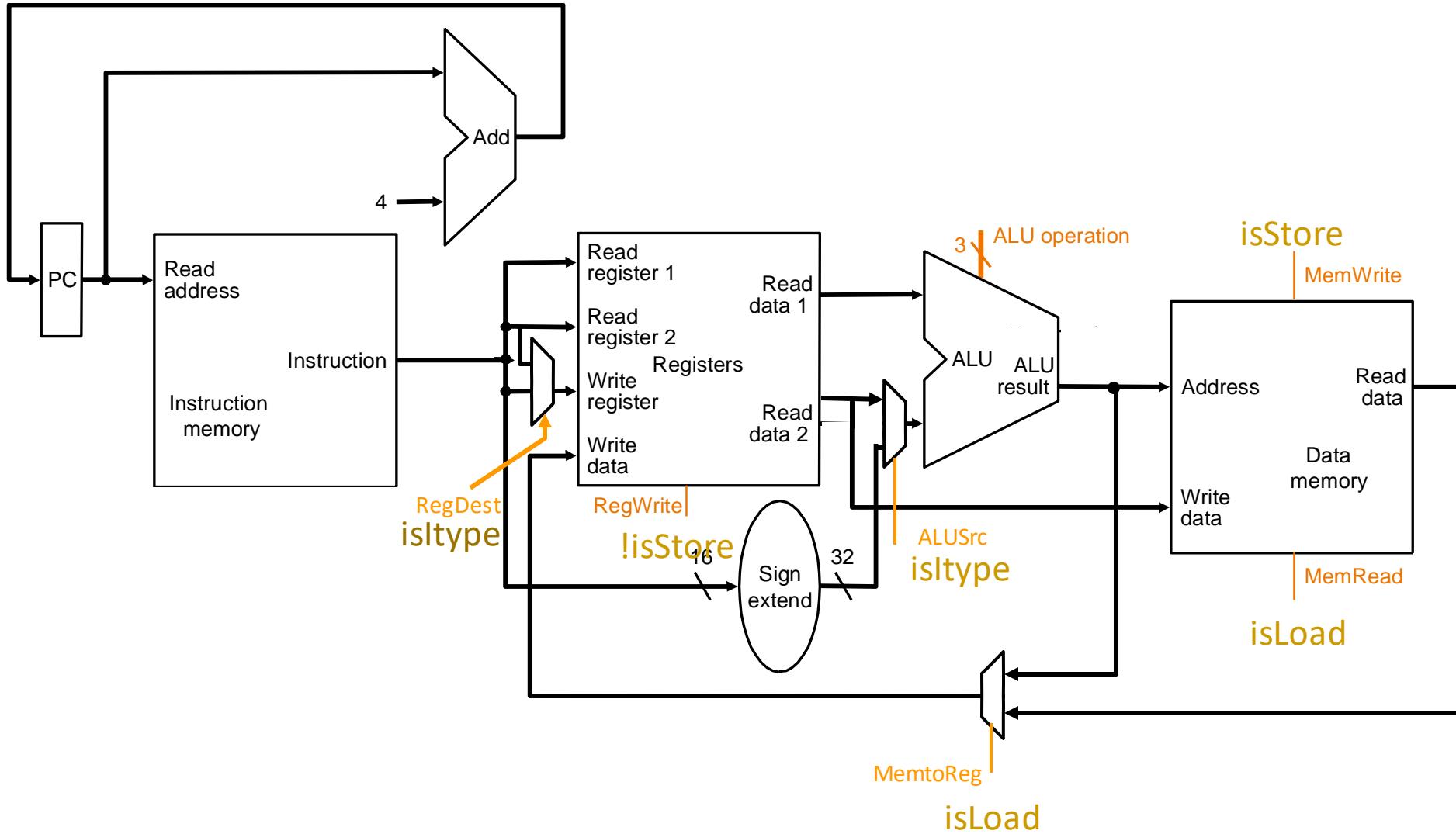
$PC \leftarrow PC + 4$



Combinational state update logic

Single-Cycle Datapath for *Data Movement Instructions*

Datapath for Non-Control-Flow Insts.



Single-Cycle Datapath for *Control Flow Instructions*

Jump Instruction

■ Unconditional branch or jump

j target

j (2)	immediate	J-Type
6 bits	26 bits	

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

■ Semantics

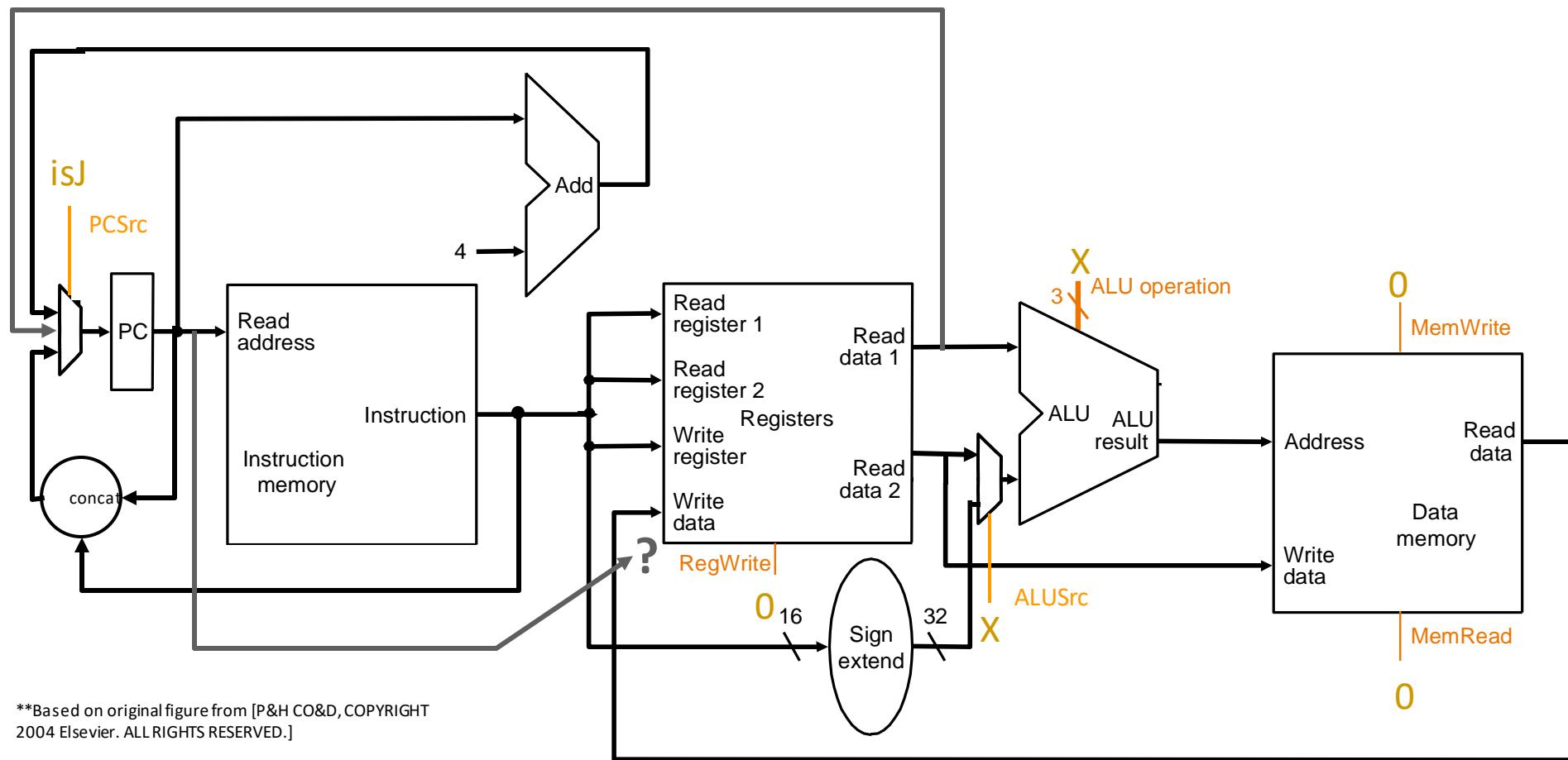
if $\text{MEM}[\text{PC}] == j \text{ immediate}_{26}$

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

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

[†]This is the incremented PC

Unconditional Jump Datapath



if $\text{MEM}[\text{PC}] == \text{J}$ immediate26

$\text{PC} = \{ \text{PC}[31:28], \text{immediate26}, 2' b00 \}$

What about JR, JAL, JALR?

Other Jumps in MIPS

- jal: jump and link (function calls)

- Semantics

- if $\text{MEM}[\text{PC}] == \text{jal}$ immediate₂₆**

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

- $\text{target} = \{ \text{PC}^+ [31:28], \text{immediate}_{26}, 2' \text{ b}00 \}$

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

- 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

Aside: MIPS Cheat Sheet

- https://safari.ethz.ch/digitaltechnik/spring2021/lib/exe/fetch.php?media=mips_reference_data.pdf

■ On the course website

OPCODES, BASE CONVERSION, ASCII SYMBOLS									
(1) MIPS		(1) MIPS		(2) MIPS		ASCII		ASCII	
opcode	funct	opcode	funct	Binary	dec	hex	char	dec	char
(31:26) (5:0)		(31:26) (5:0)		(31:26) (5:0)		(31:26) (5:0)		(31:26) (5:0)	
(1) sll	addf	000000	0	0	0	NUL	64	40	
	subf	000001	1	1	0	SOF	65	41	A
j	sr1	000010	2	2	0	SOH	66	42	
jal	sr2	000011	3	3	0	ETX	67	43	C
beq	sl1v	sqt1	001000	4	4	EOT	68	44	D
	sl2v	sqt2	001001	5	5	ENQ	69	45	E
bles	sr1v	mcnf	001010	6	6	SOH	70	46	F
bgtz	sr2v	nesf	001011	7	7	TEL	71	47	G
addi	jr	001000	8	8	BS	72	48	H	
addi	jalr	001001	9	9	HT	73	49	I	
slti	movz	001010	10	10	FF	74	4A	J	
sltiu	movn	001011	11	11	VT	75	4B	K	
andi	syseval	round,wj	001100	12	C	PF	76	4C	L
ori	break	trunc,wj	001101	13	D	CR	77	4D	M
xori	cell,wj	ceil,wj	001110	14	E	SF	78	4E	N
lui	sync	clz,wj	001111	15	F	SI	79	4F	O
	mfhi		010000	16	10	DLE	80	50	P
(2)	mthi		010001	17	11	DC1	81	51	Q
mflo	movz		010010	18	12	DC2	82	52	R
mtlo	movn		010011	19	13	IC3	83	53	S
			010100	20	14	DC4	84	54	T
			010101	21	15	NAK	85	55	U
			010110	22	16	SYN	86	56	V
			010111	23	17	ETB	87	57	W
mult			011000	24	18	CAN	88	58	X
multu			011001	25	19	EM	89	59	Y
div			011010	26	1a	SUB	90	5A	Z
divu			011011	27	1b	SD	91	5B	
			011100	28	1c	FS	92	5C	
			011101	29	1d	GS	93	5D	J
			011110	30	1e	RE	94	5E	
			011111	31	1f	US	95	5F	
lb	add	cvt,af	100000	32	20	WSpace	96	60	
lh	addu	cvt,af	100001	33	21	97	61	Space	
lw	sub		001010	34	22	*	98	62	b
lw	subu		001011	35	23	*	99	63	c
		cvt,af	010000	36	24	*	100	64	
lbu	or		010101	37	25	*	101	65	e
lwr	xor		010110	38	26	*	102	66	f
			010111	39	27	*	103	67	g
sb			101000	40	28	(104	68	h
sh			101001	41	29)	105	69	i
swl	slt		101010	42	2a	*	106	6a	j
sw	sltu		101011	43	2b	*	107	6b	k
			101100	44	2c	*	108	6c	
			101101	45	2d	*	109	6d	m
			101110	46	2e	*	110	6e	n
cache			101111	47	2f	*	111	6f	o
ll	teq	c-eif	110000	48	20	70	112	70	p
lcl	teqc	c-umf	110001	49	31	1	113	71	q
lw2c	tit	c-eqf	110010	50	32	2	114	72	r
pref	titu	c-uesf	110011	51	33	3	115	73	s
		ted	110100	52	34	4	116	74	t
ldcl	c-olif	c-olif	110101	53	35	5	117	75	u
ld2c	c-olif	c-olif	110110	54	36	6	118	76	v
		c-olif	110111	55	37	7	119	77	w
sc	c-eif		111000	56	38	8	120	78	x
swcl	c-eif		111001	57	39	9	121	79	y
swc2	c-eif		111010	58	3a	10	122	7a	z
	c-eif		111011	59	3b	11	123	7b	
sdcl	c-ngef		111100	60	3c	124	7c		
	c-ngef		111101	61	3d	125	7d		
	c-ngef		111110	62	3e	126	7e		
	c-ngef		111111	63	3f	127	7f	DEL	

IEEE 754 Floating-Point STANDARD		IEEE 754 Symbols																																																										
$(-1)^S \times (1 + (Fraction) \times 2^E) \times 2^{(Exponent - Bias)}$																																																												
where Single Precision Bias = 127, Double Precision Bias = 1023.																																																												
IEEE Single Precision and Double Precision Formats:																																																												
<table border="1"> <tr><td>S</td><td>Exponent</td></tr> <tr><td>31</td><td>30</td><td>23</td><td>22</td></tr> <tr><td>S</td><td>Exponent</td></tr> <tr><td>63</td><td>62</td><td>52</td><td>51</td></tr> </table>		S	Exponent	31	30	23	22	S	Exponent	63	62	52	51	<table border="1"> <tr><td>Exponent</td><td>Fraction</td><td>Object</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>Denormal</td></tr> <tr><td>1 to MAX - 1</td><td>anything</td><td>F1, F2, ... FN</td></tr> <tr><td>MAX</td><td>0</td><td>± infinity</td></tr> <tr><td>MAX</td><td>0</td><td>NaN</td></tr> <tr><td>S.P. MAX = 255, D.P. MAX = 2047</td><td></td><td></td></tr> </table>		Exponent	Fraction	Object	0	0	0	0	0	Denormal	1 to MAX - 1	anything	F1, F2, ... FN	MAX	0	± infinity	MAX	0	NaN	S.P. MAX = 255, D.P. MAX = 2047																										
S	Exponent																																																											
31	30	23	22																																																									
S	Exponent																																																											
63	62	52	51																																																									
Exponent	Fraction	Object																																																										
0	0	0																																																										
0	0	Denormal																																																										
1 to MAX - 1	anything	F1, F2, ... FN																																																										
MAX	0	± infinity																																																										
MAX	0	NaN																																																										
S.P. MAX = 255, D.P. MAX = 2047																																																												
MEMORY ALLOCATION		Fraction																																																										
$\$sp \rightarrow 7\text{fff ftt}_{\text{hex}}$ $\$gp \rightarrow 1000\ 8000_{\text{hex}}$ $1000\ 0000_{\text{hex}}$ $pc \rightarrow 0040\ 0000_{\text{hex}}$ 0_{hex}		<table border="1"> <tr><td>Stack</td></tr> <tr><td>↓</td></tr> <tr><td>Dynamic Data</td></tr> <tr><td>↓</td></tr> <tr><td>Static Data</td></tr> <tr><td>↓</td></tr> <tr><td>Text</td></tr> <tr><td>↓</td></tr> <tr><td>Reserved</td></tr> </table>		Stack	↓	Dynamic Data	↓	Static Data	↓	Text	↓	Reserved																																																
Stack																																																												
↓																																																												
Dynamic Data																																																												
↓																																																												
Static Data																																																												
↓																																																												
Text																																																												
↓																																																												
Reserved																																																												
		STACK FRAME <table border="1"> <tr><td>Higher Memory Address</td></tr> <tr><td>Argument 6</td></tr> <tr><td>Argument 5</td></tr> <tr><td>↓</td></tr> <tr><td>Saved Registers</td></tr> <tr><td>↓</td></tr> <tr><td>Local Variables</td></tr> <tr><td>↓</td></tr> <tr><td>Stack Grows</td></tr> <tr><td>↓</td></tr> <tr><td>Lower Memory Address</td></tr> </table>		Higher Memory Address	Argument 6	Argument 5	↓	Saved Registers	↓	Local Variables	↓	Stack Grows	↓	Lower Memory Address																																														
Higher Memory Address																																																												
Argument 6																																																												
Argument 5																																																												
↓																																																												
Saved Registers																																																												
↓																																																												
Local Variables																																																												
↓																																																												
Stack Grows																																																												
↓																																																												
Lower Memory Address																																																												
DATA ALIGNMENT																																																												
<table border="1"> <tr><td colspan="4">Double Word</td></tr> <tr><td colspan="4">Word</td></tr> <tr><td colspan="2">Halfword</td><td colspan="2">Word</td></tr> <tr><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> </table>		Double Word				Word				Halfword		Word		Byte	Byte	Byte	Byte	0	1	2	3	<table border="1"> <tr><td colspan="4">Double Word</td></tr> <tr><td colspan="4">Word</td></tr> <tr><td colspan="2">Halfword</td><td colspan="2">Word</td></tr> <tr><td>Byte</td><td>Byte</td><td>Byte</td><td>Byte</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> </table>		Double Word				Word				Halfword		Word		Byte	Byte	Byte	Byte	0	1	2	3																	
Double Word																																																												
Word																																																												
Halfword		Word																																																										
Byte	Byte	Byte	Byte																																																									
0	1	2	3																																																									
Double Word																																																												
Word																																																												
Halfword		Word																																																										
Byte	Byte	Byte	Byte																																																									
0	1	2	3																																																									
Value of three least significant bits of byte address (Big Endian)																																																												
EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS																																																												
<table border="1"> <tr><td>B</td><td>D</td><td>13</td><td>15</td><td>9</td><td>8</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr> <tr><td colspan="4">Interrupt Mask</td><td colspan="4">Exception Code</td><td colspan="8"></td></tr> <tr><td colspan="4">Pending Interrupt</td><td colspan="4">U</td><td colspan="4">E</td><td colspan="2">I</td></tr> <tr><td colspan="4">Pending Interrupt</td><td colspan="4">M</td><td colspan="4">I</td><td colspan="2">E</td></tr> </table>		B	D	13	15	9	8	6	5	4	3	2	1	0	Interrupt Mask				Exception Code												Pending Interrupt				U				E				I		Pending Interrupt				M				I				E			
B	D	13	15	9	8	6	5	4	3	2	1	0																																																
Interrupt Mask				Exception Code																																																								
Pending Interrupt				U				E				I																																																
Pending Interrupt				M				I				E																																																
BI = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable																																																												
EXCEPTION CODES																																																												
Number	Cause of Exception	Number	Cause of Exception																																																									
0	Int. Interrupt (hardware)	9	Bp Breakpoint	Exception																																																								
4	Address Error Exception (load or instruction fetch)	10	Reserved Instruction																																																									
5	AIES Address Error Exception (store)	11	CpU Coprocessor	Unimplemented																																																								
6	IBE Bus Error on Instruction Fetch	12	Ov Arithmetic Overflow	Exception																																																								
7	DBE Bus Error on Load or Store	13	Tr Trap																																																									
8	E	FPU FPN	D	PC																																																								

SIZE PREFIXES (10 ³ for Disk, Communication; 2 ¹⁰ for Memory)					
	PRE- FIX	PRE- FIX	PRE- FIX	PRE- FIX	PRE- FIX
SIZE	SIZE	SIZE	SIZE	SIZE	SIZE
10 ³	Kilo	10 ¹⁵	2 ¹⁰	Peta	10 ¹⁵
10 ⁶	Mega	10 ¹²	2 ¹⁰	Exa	10 ¹⁸
10 ⁹	Giga	10 ⁹	2 ¹⁰	Zetta	10 ²¹
10 ¹²	Tera	10 ⁶	2 ¹⁰	Yotta	10 ²⁴
The symbol for each prefix is just its first letter, except µ is used for micro.					

Conditional Branch Instructions

- **beq (Branch if Equal)**

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

beq (4)	rs	rt	immediate=offset
6 bits	5 bits	5 bits	16 bits

I-Type

- Semantics (assuming no branch delay slot)

if $\text{MEM}[\text{PC}] == \text{beq } \text{rs } \text{rt } \text{immediate}_{16}$

target = $\text{PC}^+ + \text{sign-extend}(\text{immediate}) \times 4$

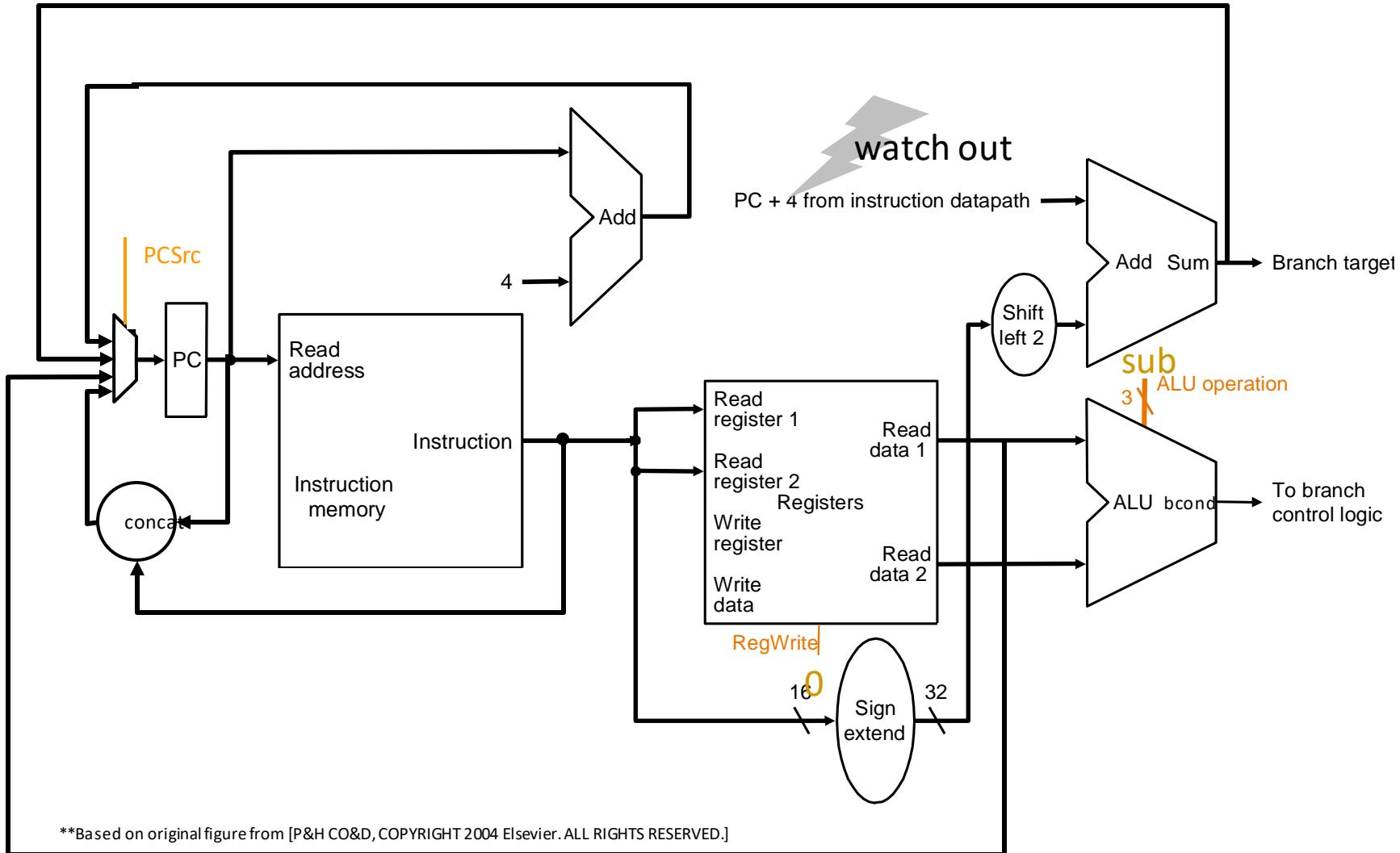
if $\text{GPR}[\text{rs}] == \text{GPR}[\text{rt}]$ then $\text{PC} \leftarrow \text{target}$

else $\text{PC} \leftarrow \text{PC} + 4$

- Variations: beq, bne, blez, bgtz

[†]This is the incremented PC

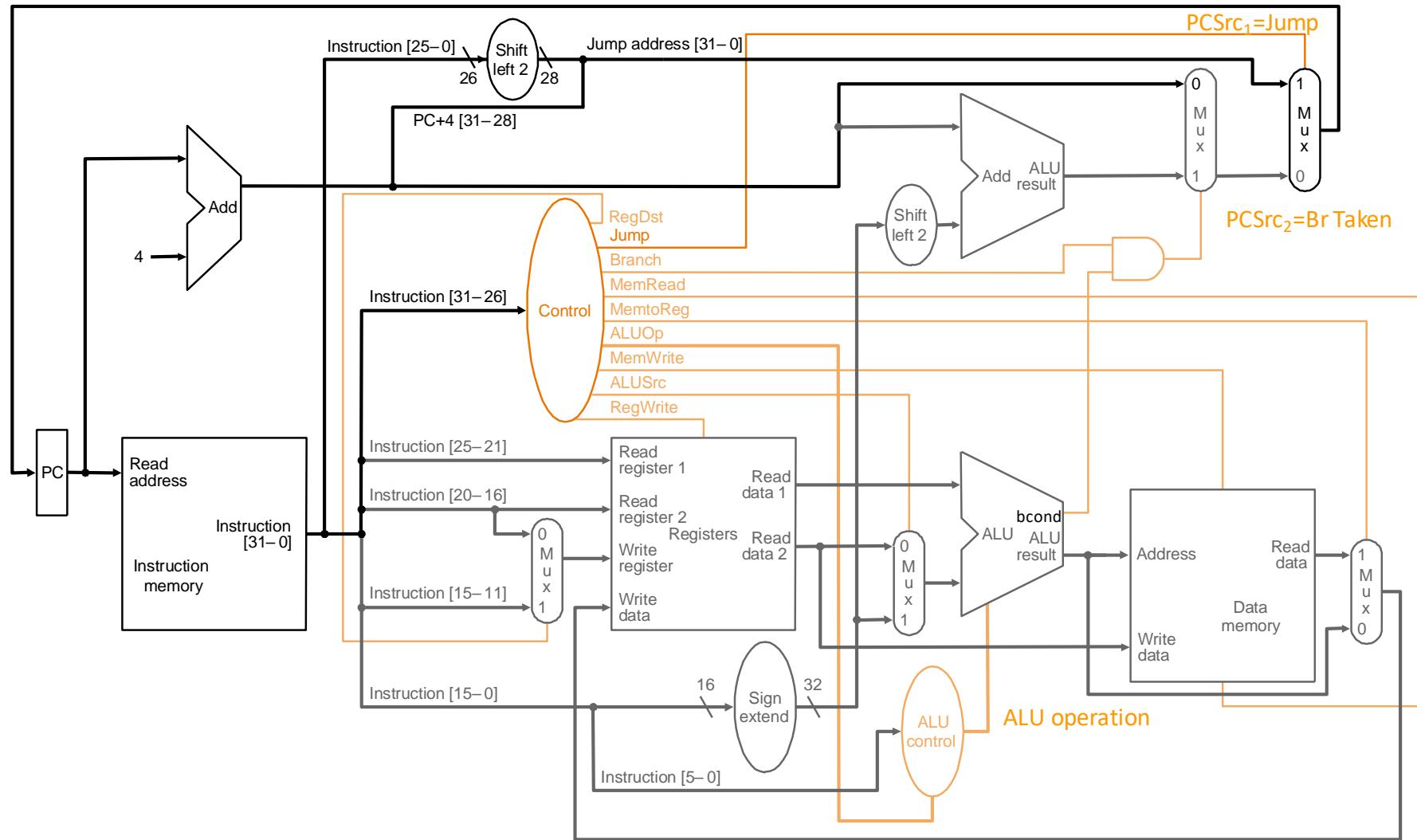
Conditional Branch Datapath (for you to finish)



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

How to uphold the delayed branch semantics?¹⁸

Putting It All Together



Single-Cycle Control Logic

Single-Cycle Hardwired Control

- As combinational function of $\text{Inst} = \text{MEM}[\text{PC}]$

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

R-Type

31	26	25	21	20	16	15	0
opcode		rs		rt			immediate
6 bits	5 bits	5 bits					16 bits

I-Type

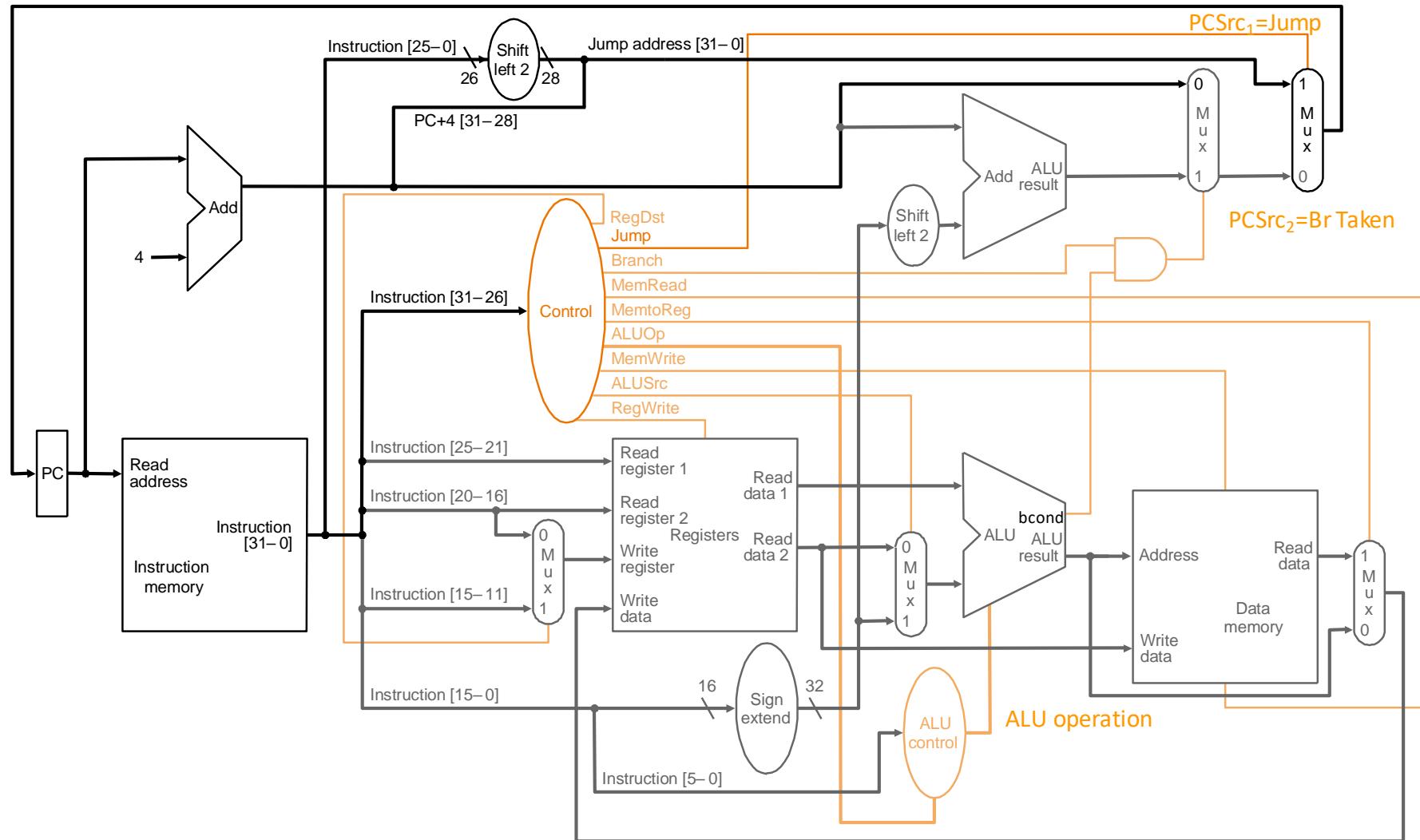
31	26	25	0
opcode			immediate
6 bits			26 bits

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)



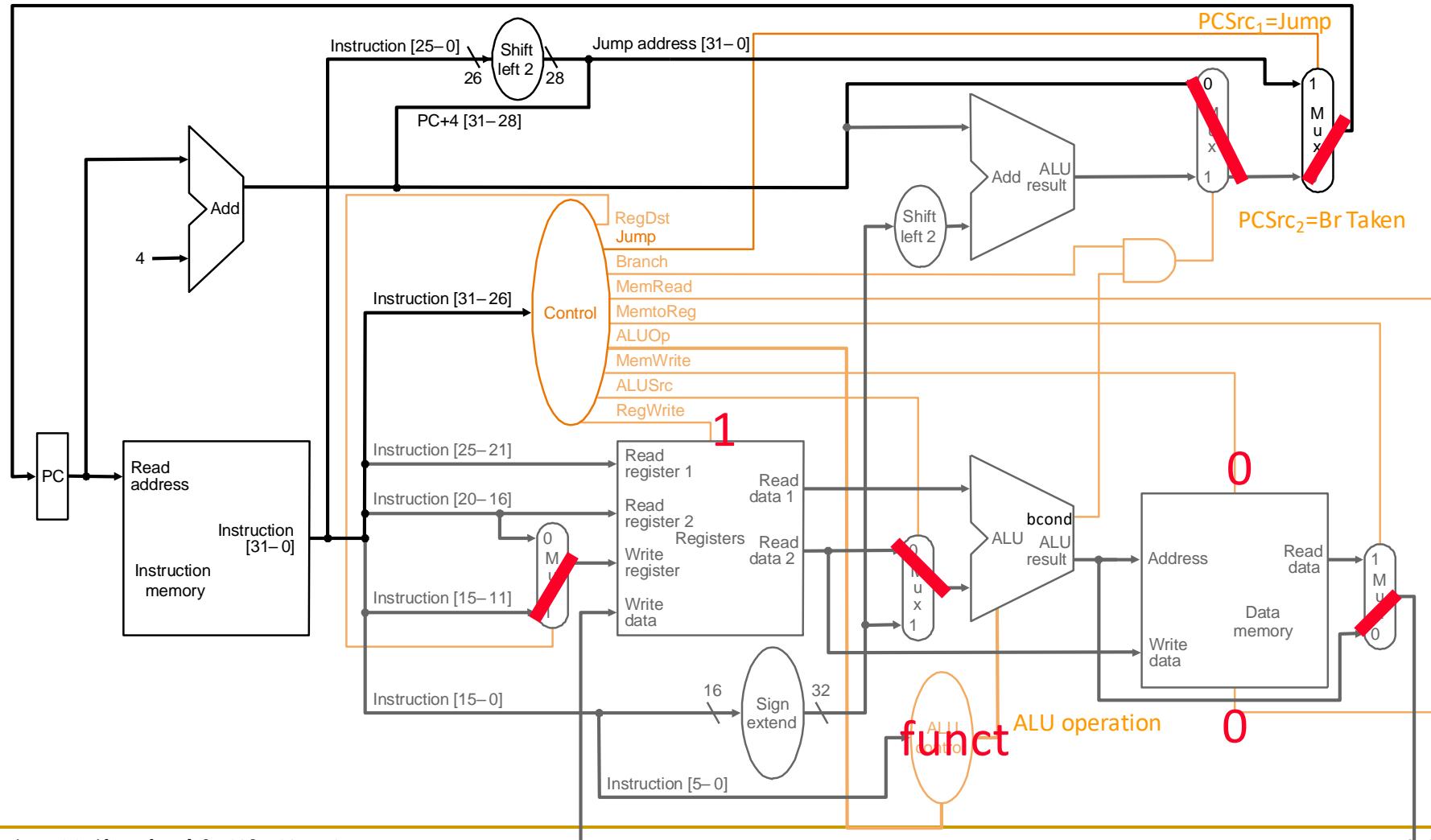
Single-Bit Control Signals (I)

	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 output to GPR write port	$opcode == LW$
RegWrite	GPR write disabled	GPR write enabled	$(opcode != SW) \&\&$ $(opcode != Bxx) \&\&$ $(opcode != J) \&\&$ $(opcode != JR))$

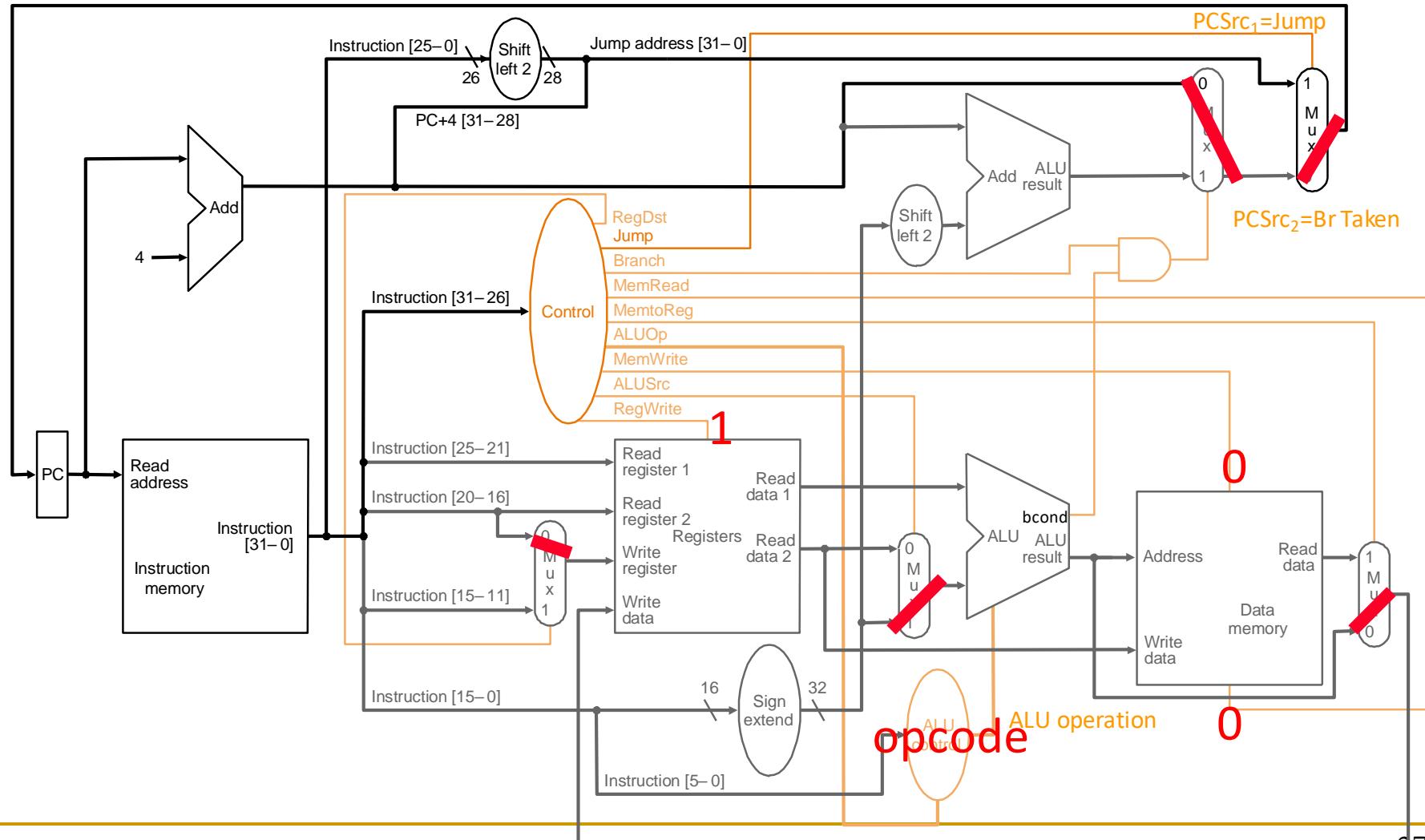
Single-Bit Control Signals (II)

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port returns 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}) \ $ $(\text{opcode} == \text{JAL})$
PCSrc ₂	next PC = PC + 4	next PC is based on 16-bit immediate branch target	$(\text{opcode} == \text{Bxx}) \ \&\&$ “bcond is satisfied”

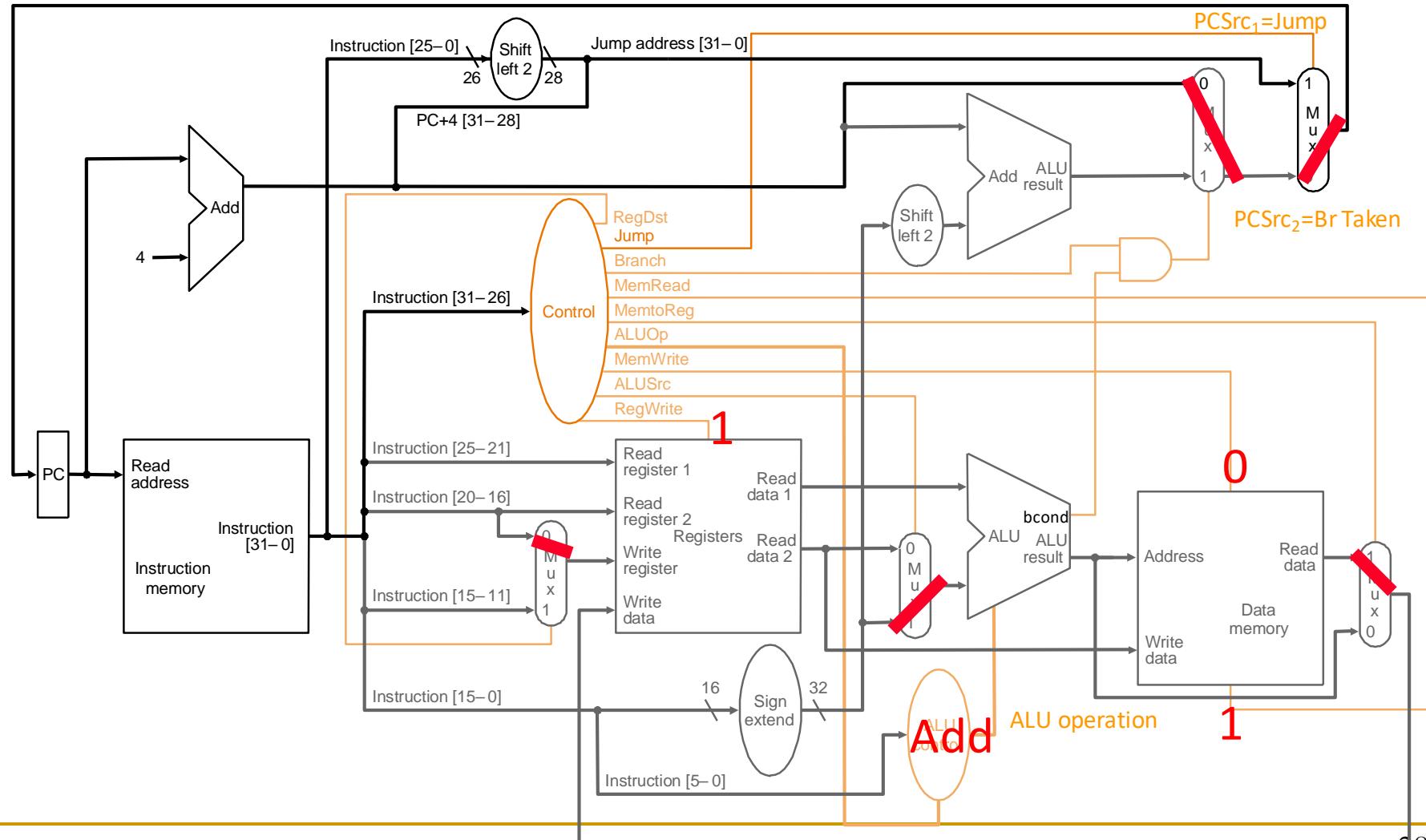
R-Type ALU

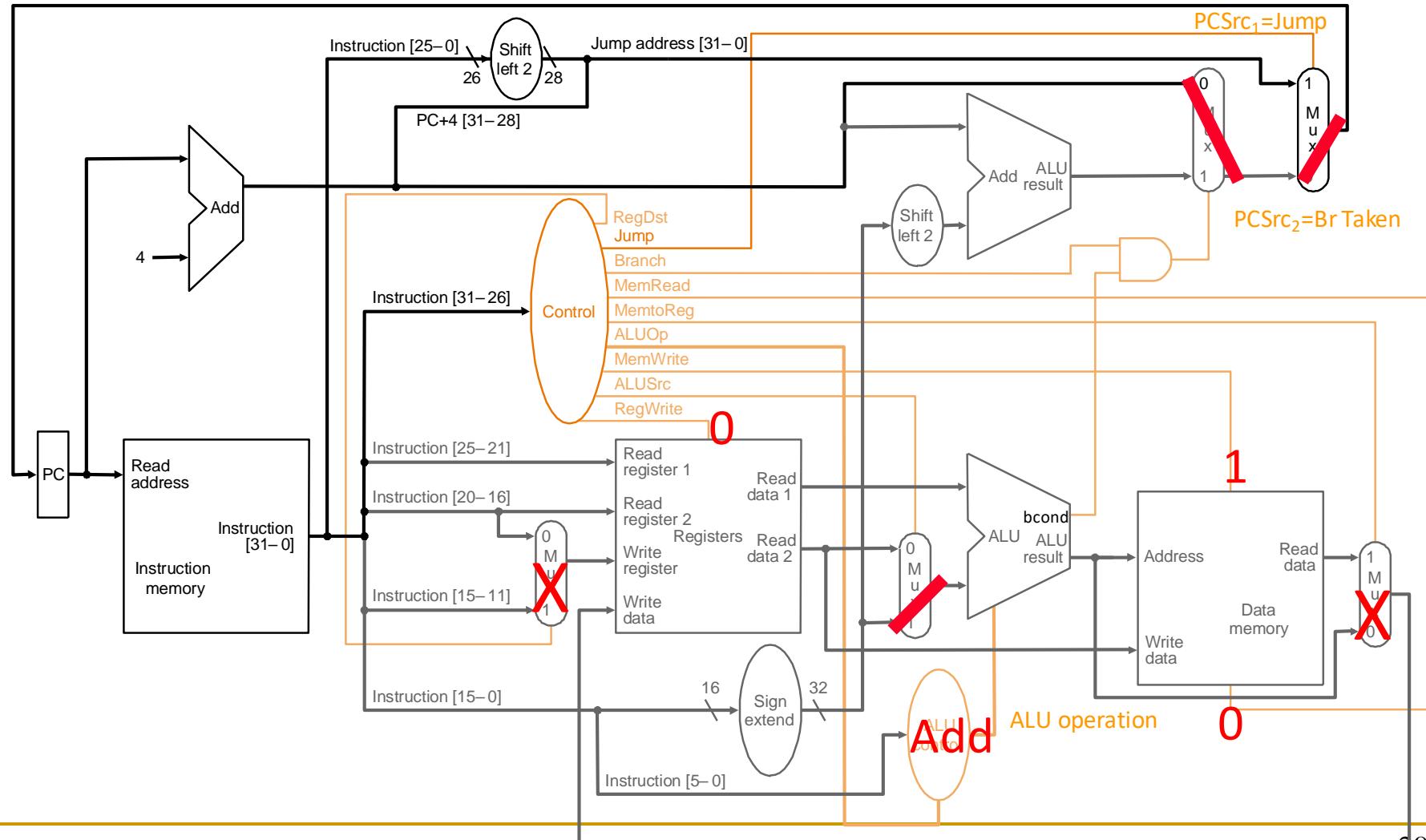


I-Type ALU



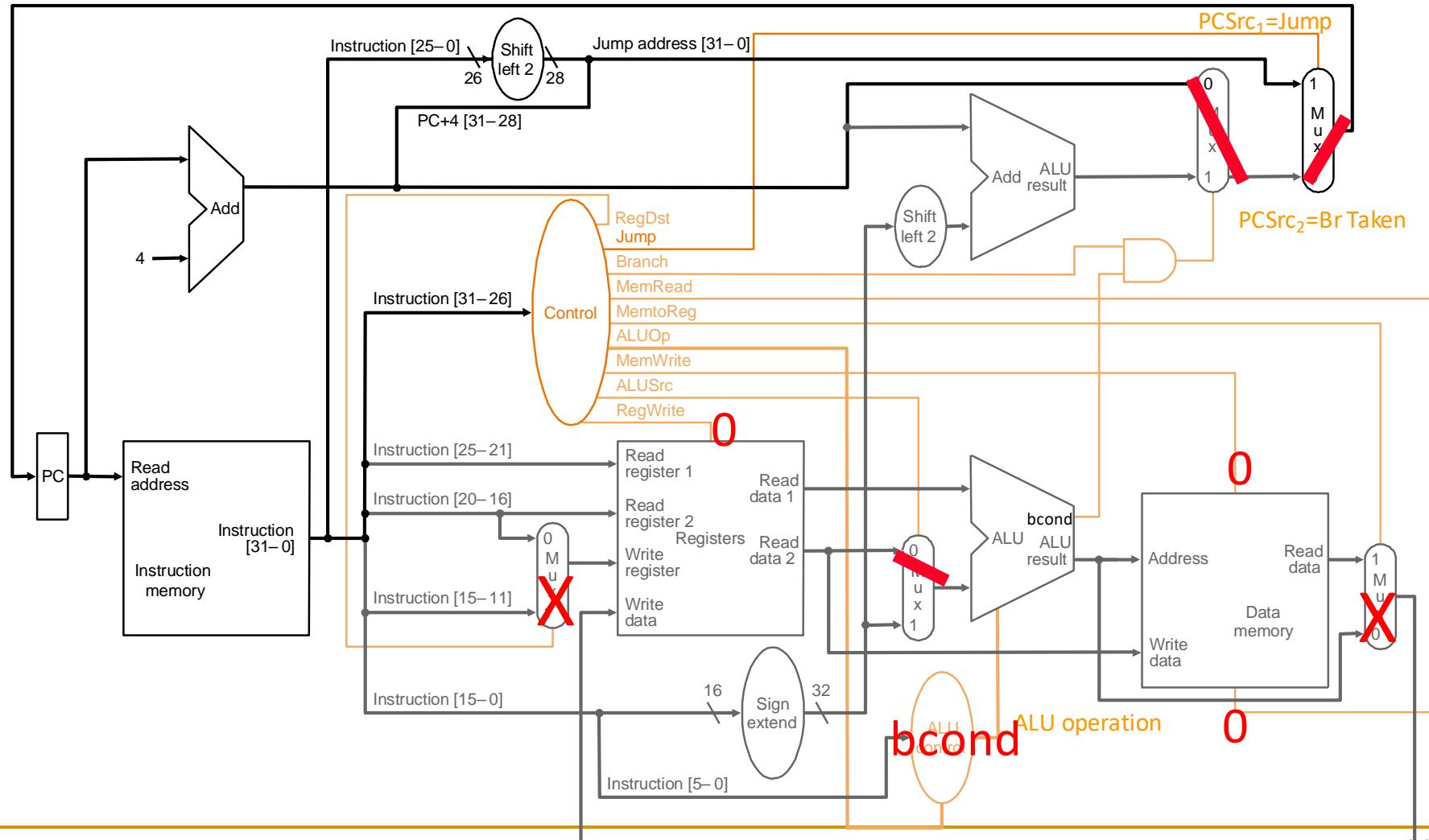
LW





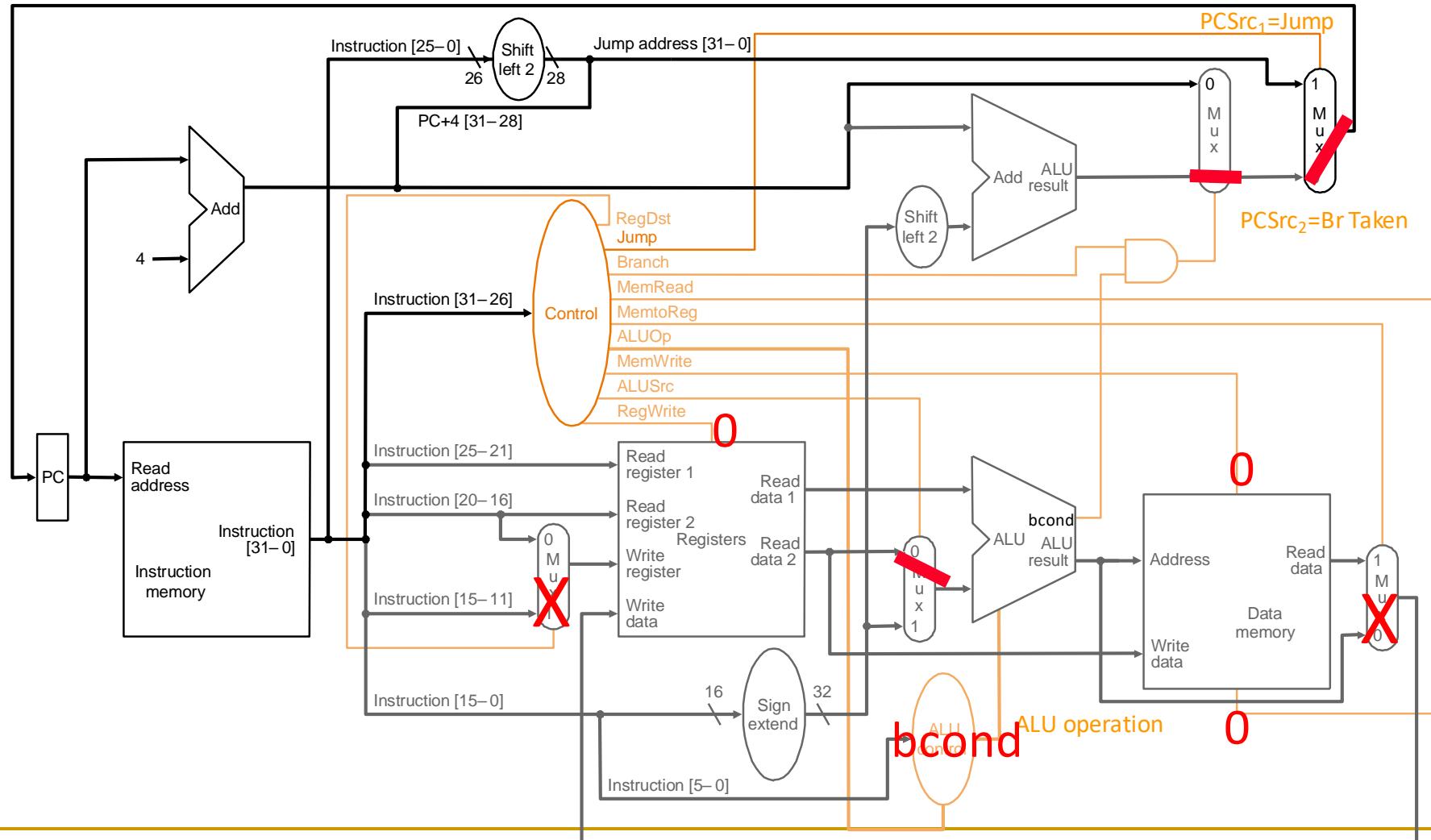
Branch (Not Taken)

Some control signals are dependent on the processing of data

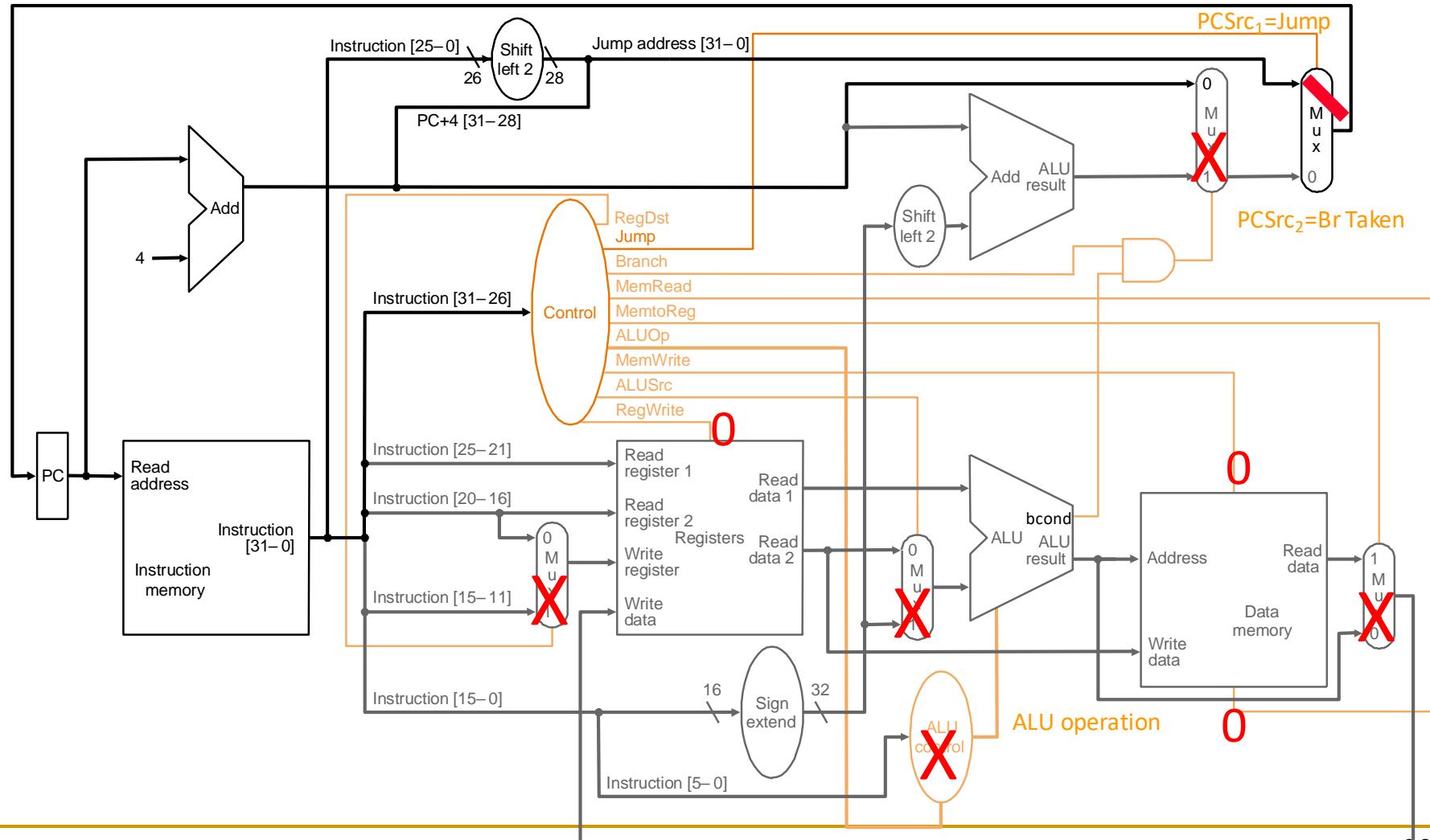


Branch (Taken)

Some control signals are dependent on the processing of data



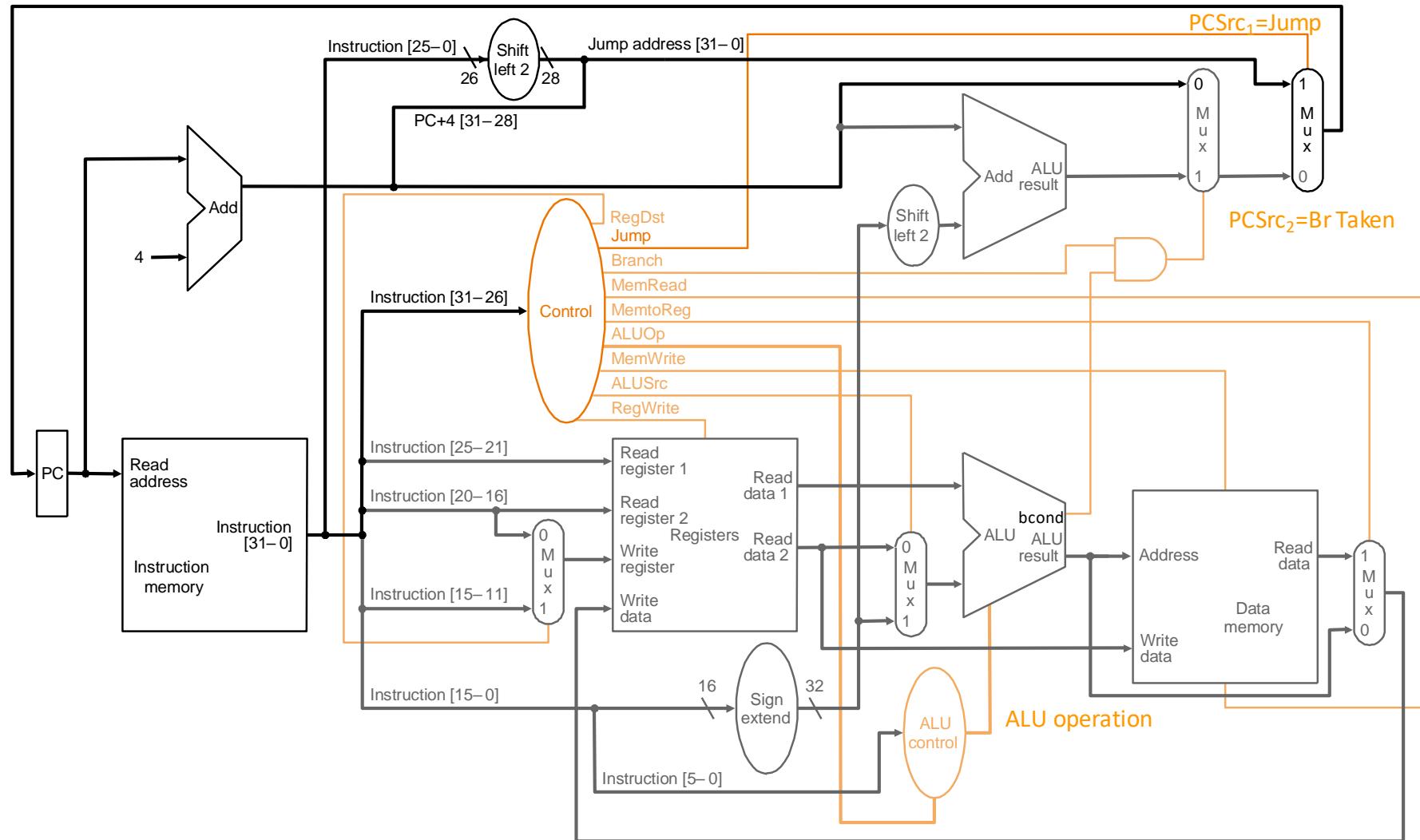
Jump



What is in That Control Box?

- Combinational Logic → Hardwired Control
 - Idea: Control signals generated combinationally 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.

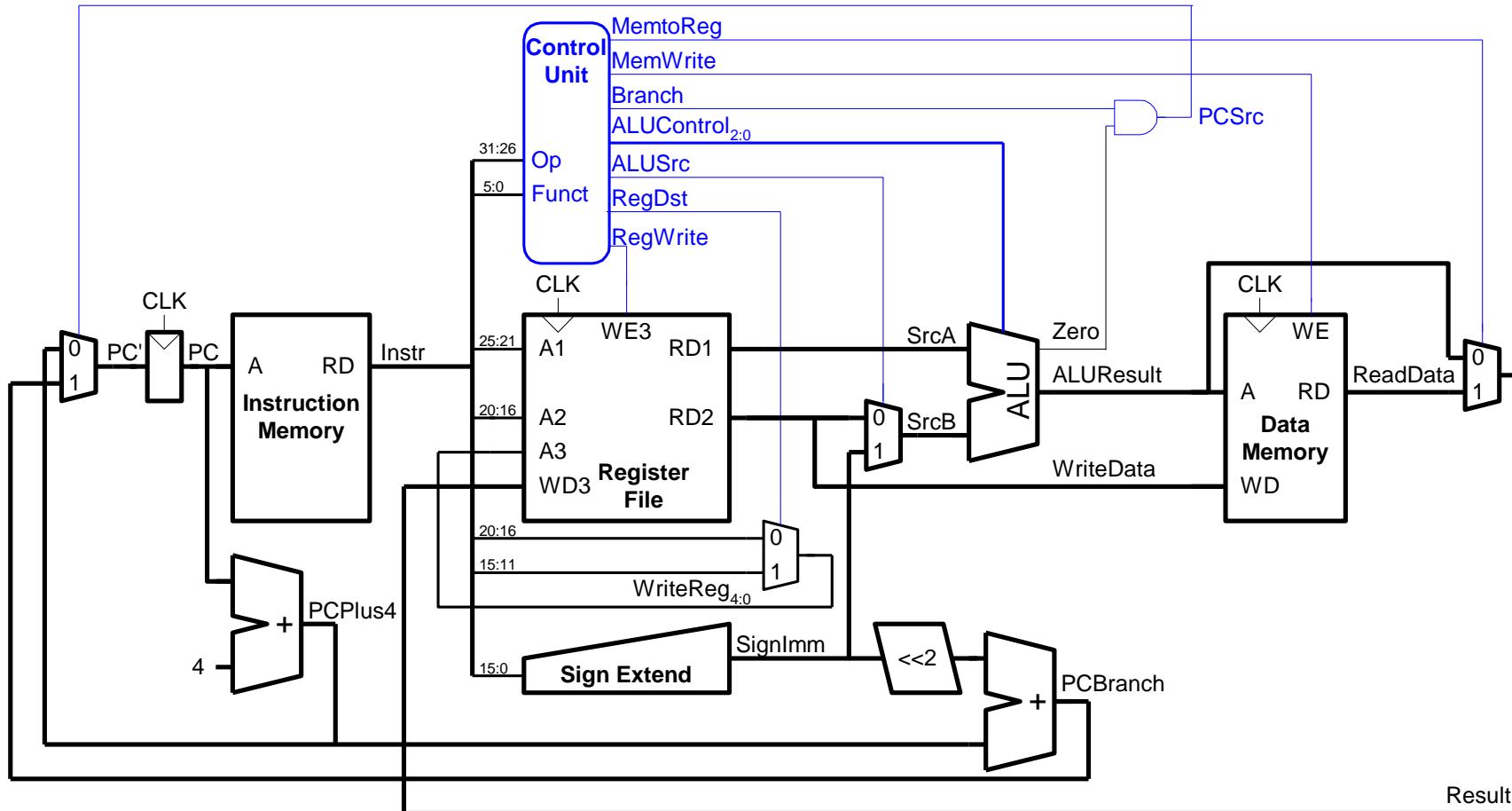
Review: Complete Single-Cycle Processor



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

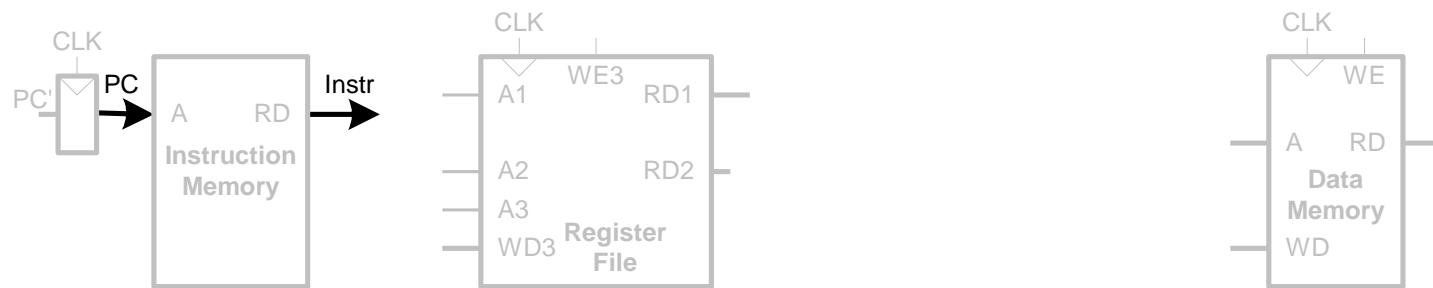
See backup slides to reinforce the concepts we have covered.
They are to complement your reading:
H&H, Chapter 7.1-7.3, 7.6

Another Complete Single-Cycle Processor



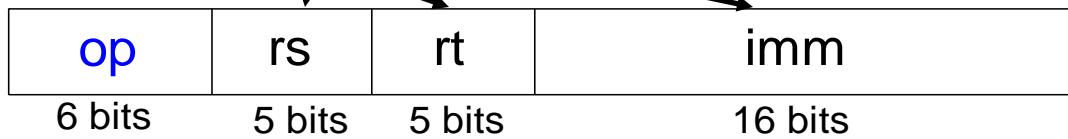
Example: Single-Cycle Datapath: lw fetch

■ STEP 1: Fetch instruction



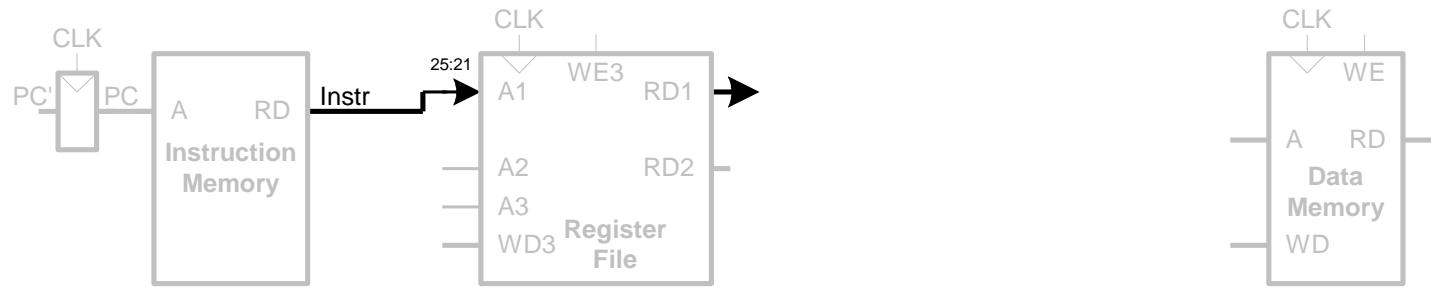
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type



Single-Cycle Datapath: lw register read

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



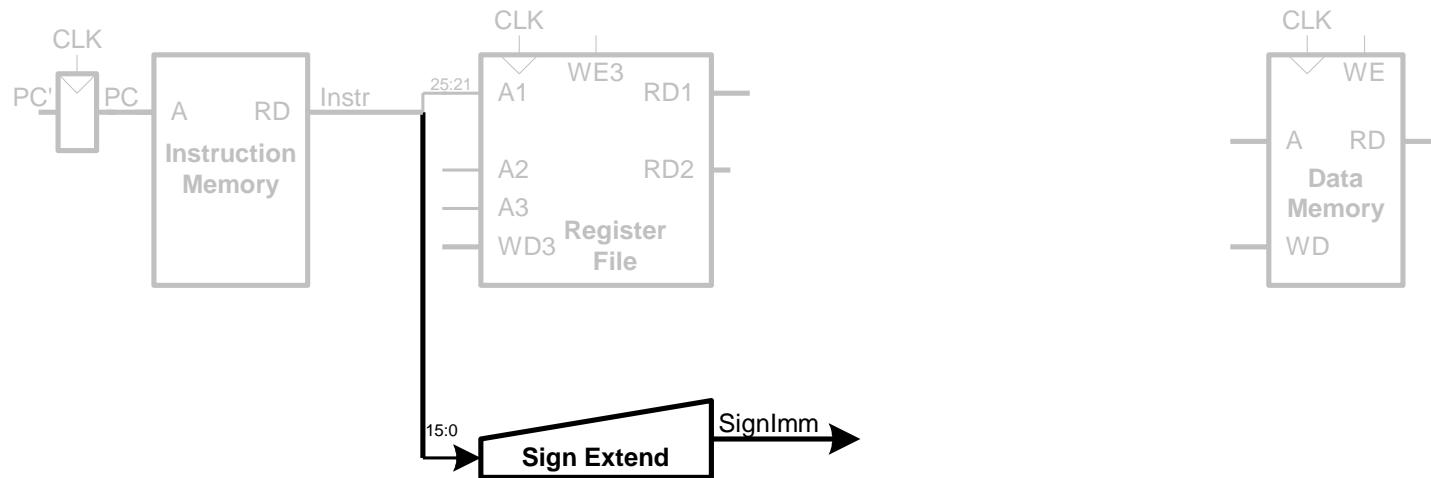
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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*



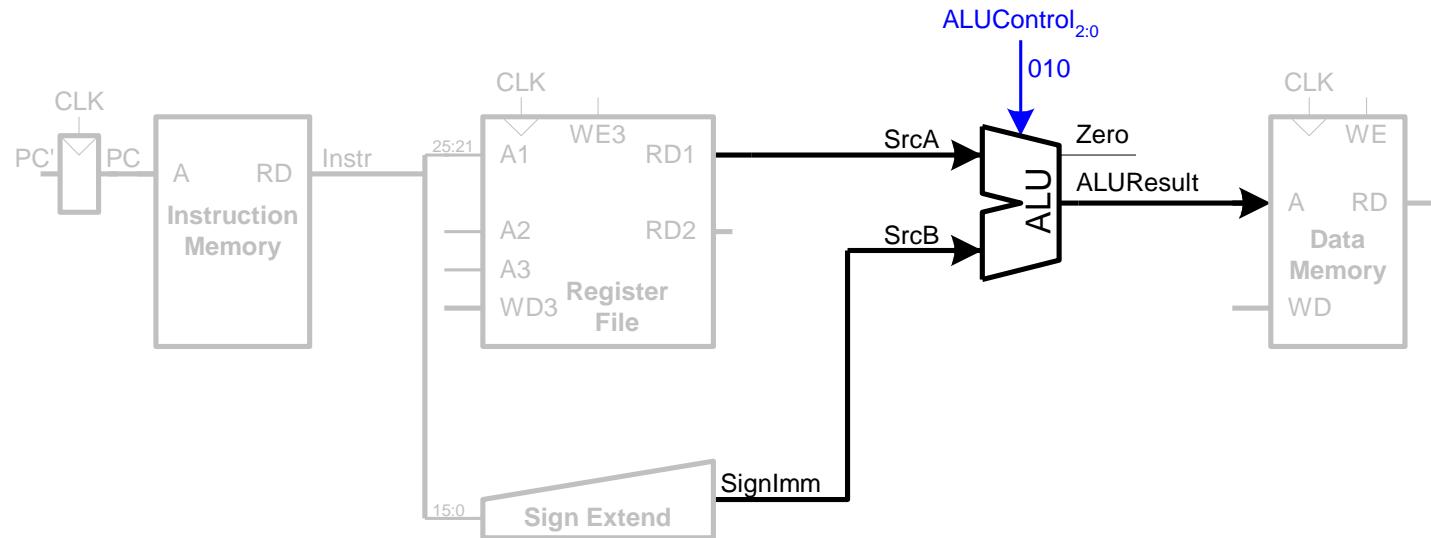
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



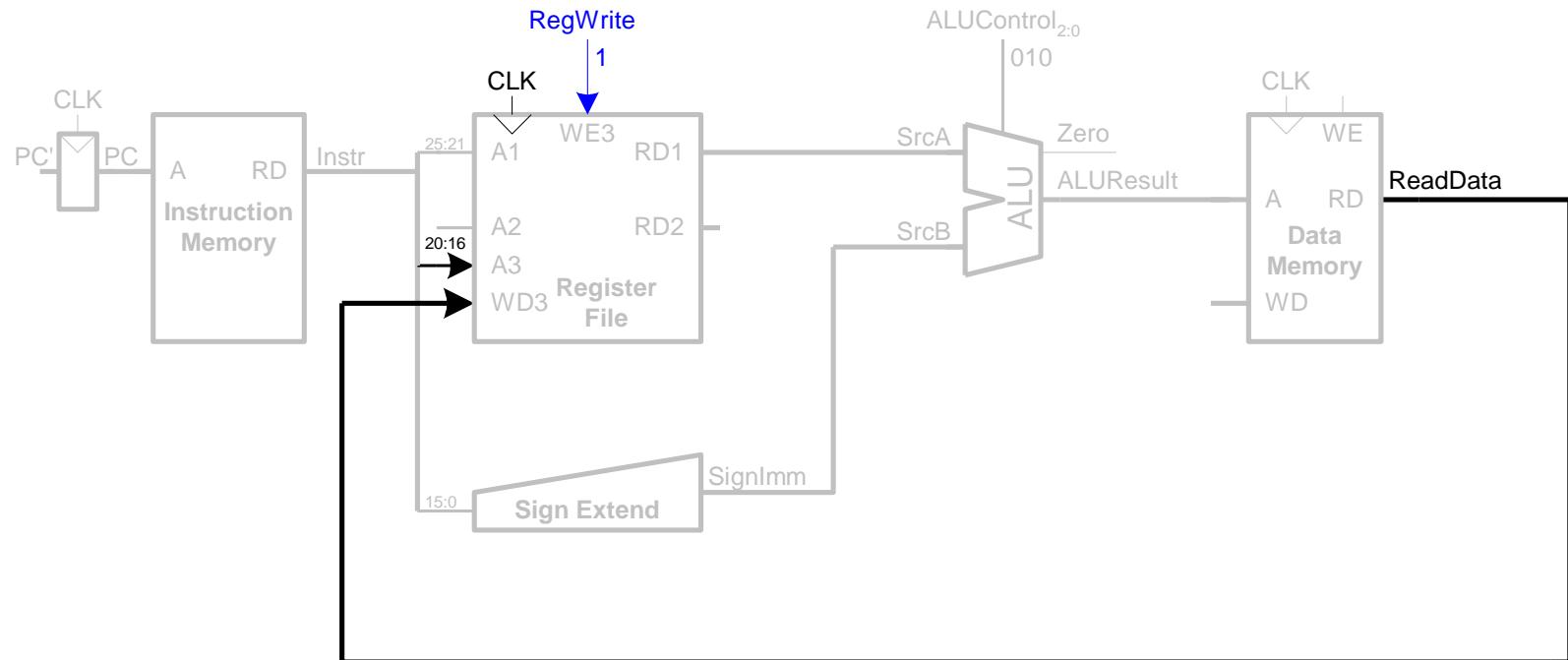
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



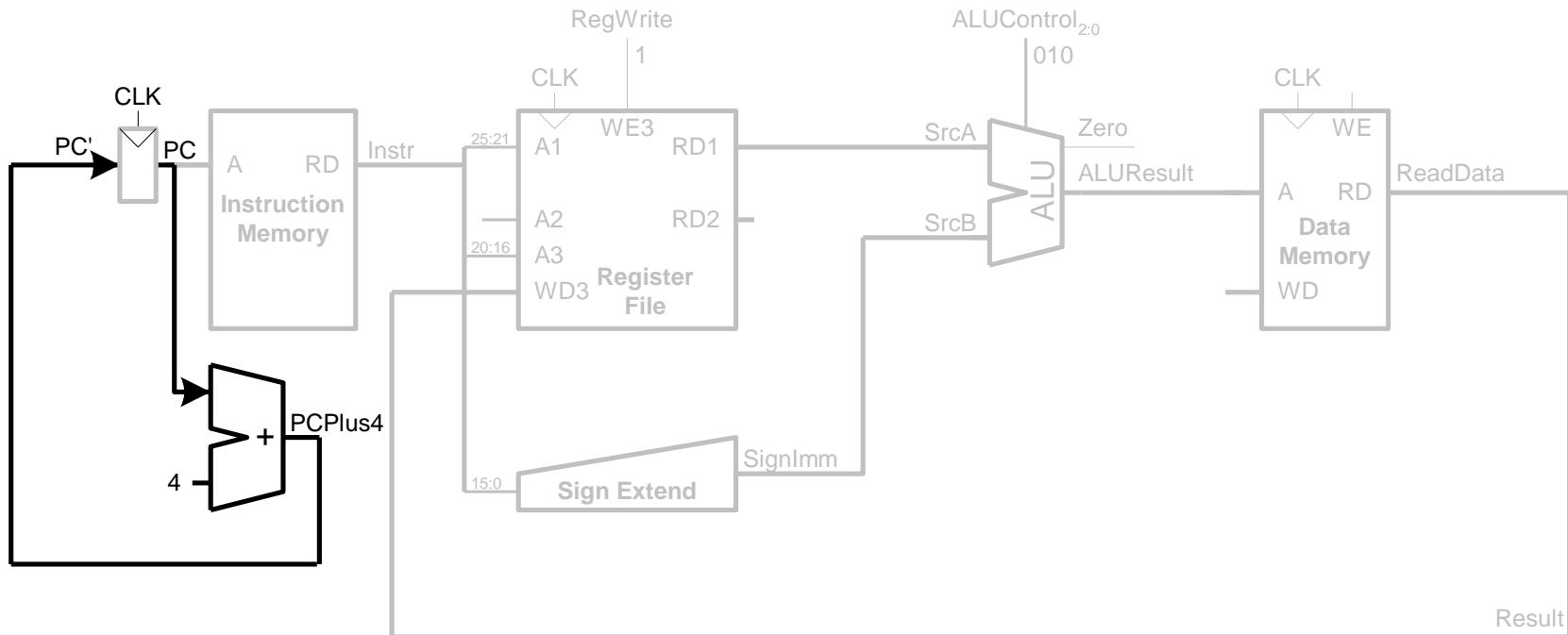
`lw $s3, 1($0) # read memory word 1 into $s3`

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

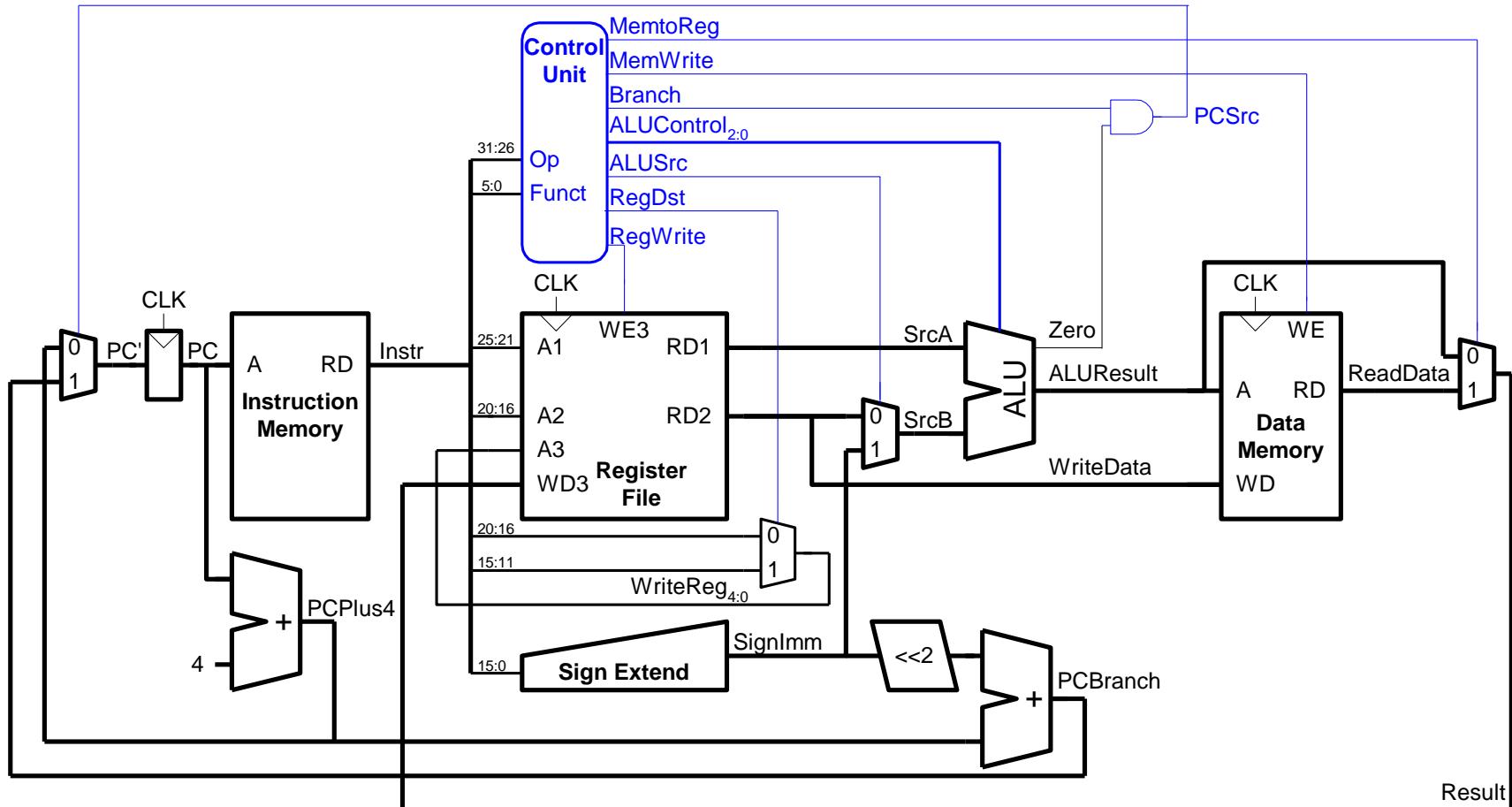
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

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

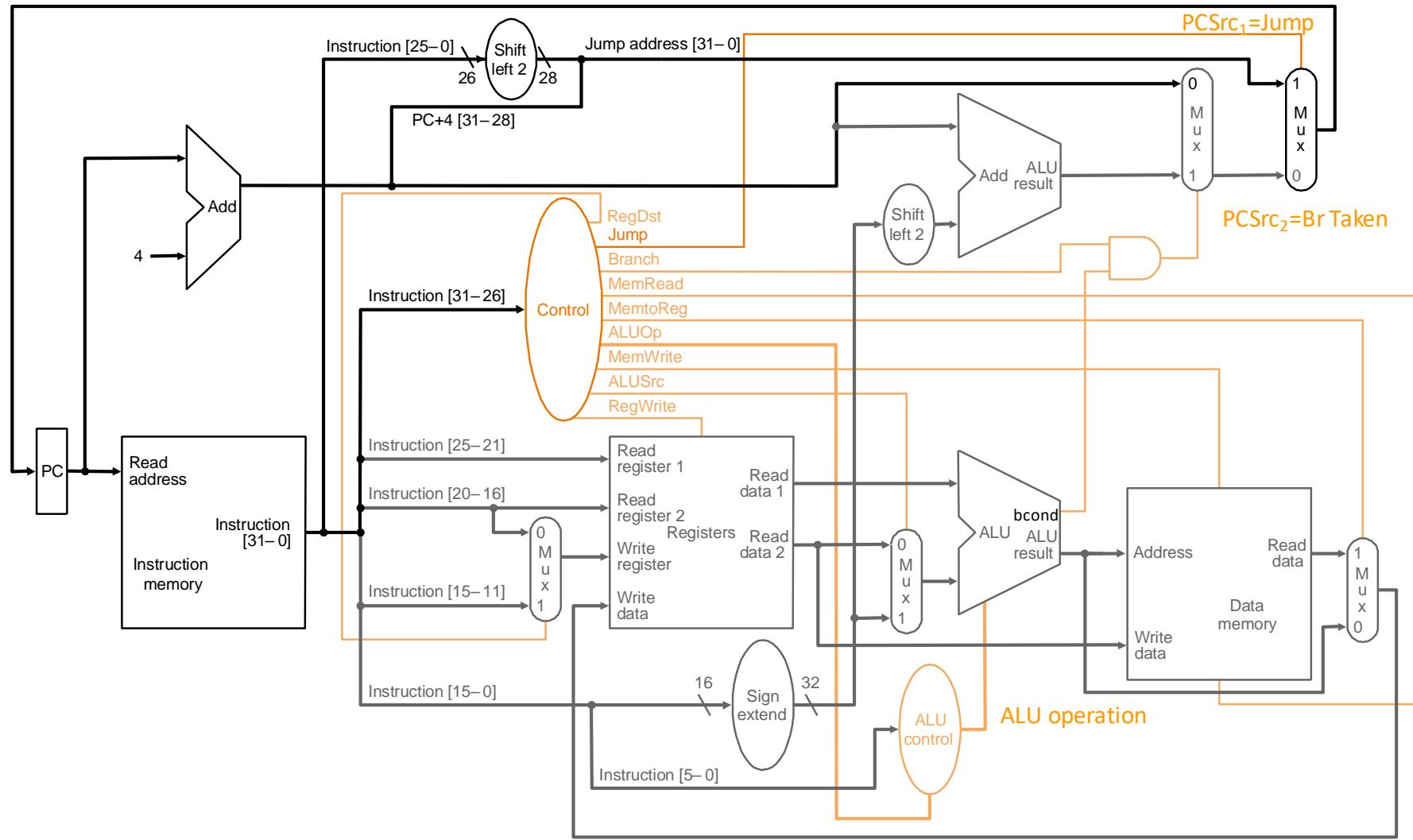
Another Complete Single-Cycle Processor (H&H)



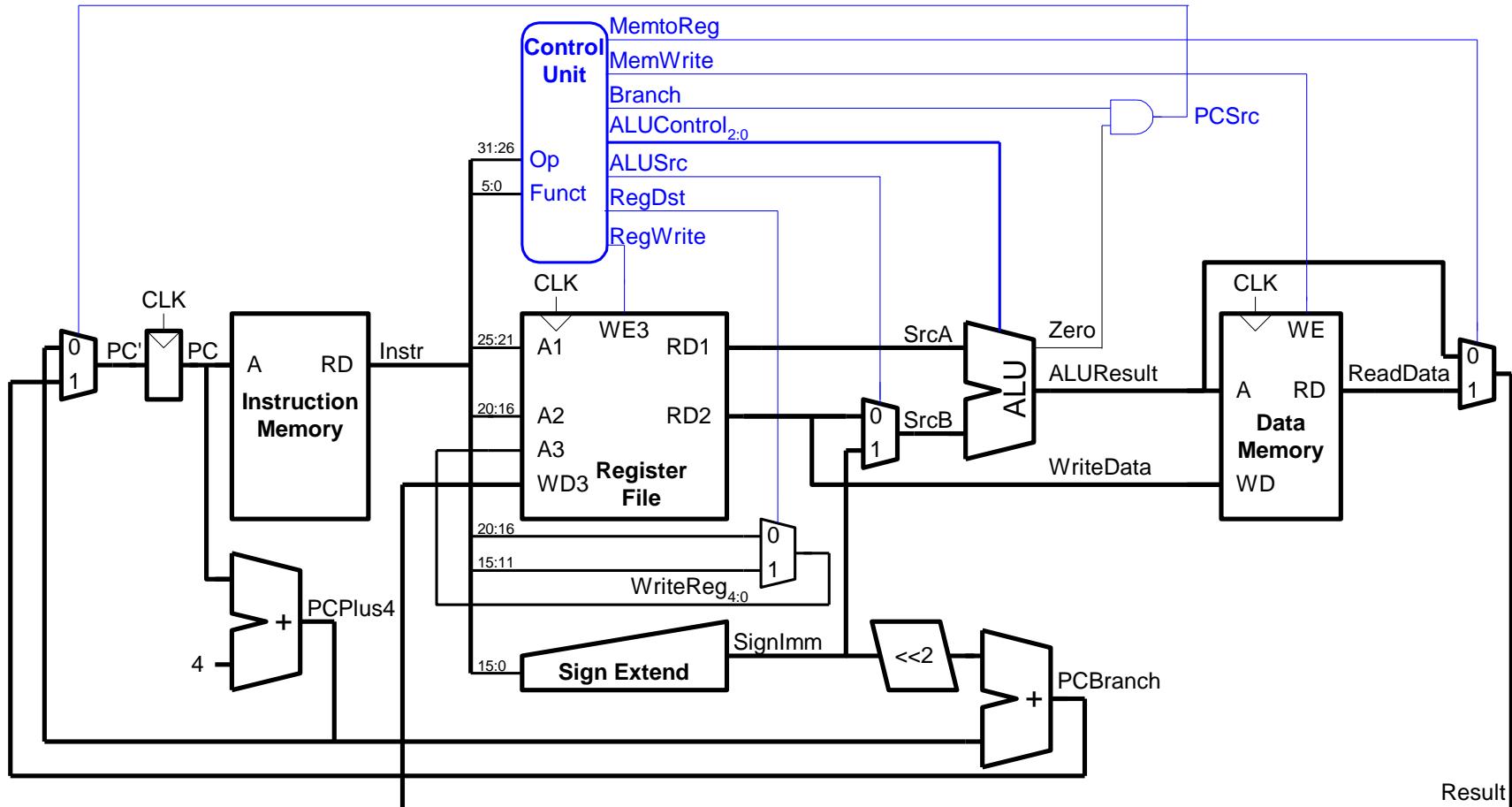
Your Reading Assignment

- Please read the Lecture Slides & 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

Recall: Performance Analysis Basics

- Execution time of a single instruction
 - **{CPI} x {clock cycle time}**
 - CPI: Number of cycles it takes to execute an 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}**

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

■ How fast are my instructions?

- Instructions are realized on the hardware
- Each instruction 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

■ How fast are my instructions?

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

■ How long 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

■ As a general formula

- Our program consists of executing **N** instructions
- Our processor needs **CPI** cycles (on average) for each instruction
- The clock frequency of the processor is **f**
 - the clock period is therefore **T=1/f**

Processor Performance

■ As a general formula

- Our program consists of executing **N** instructions
- Our processor needs **CPI** cycles (on average) for each instruction
- The clock frequency of the processor is **f**
→ 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 of Our Single-Cycle Design

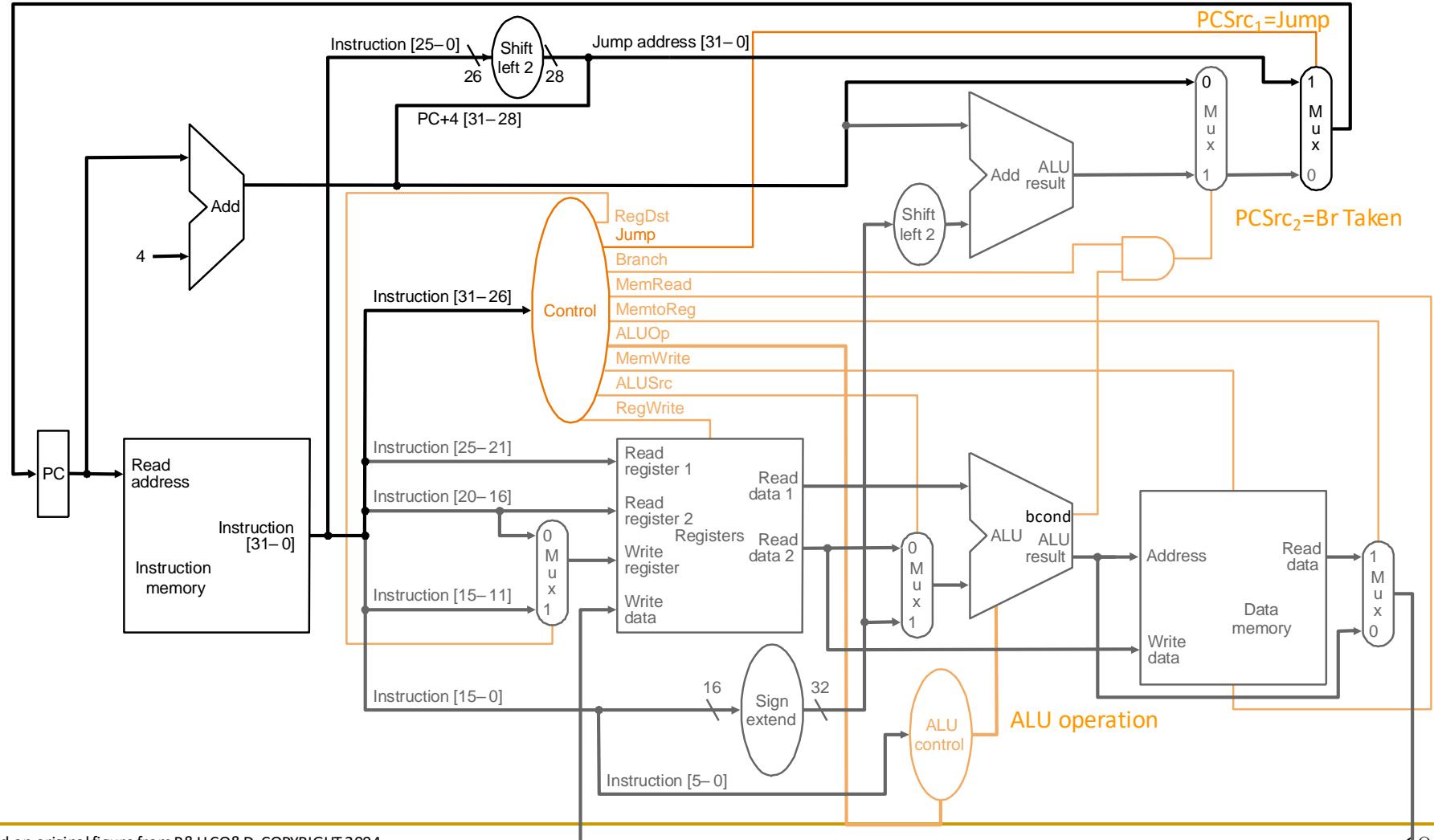
A Single-Cycle Microarchitecture: Analysis

- 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
 - 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?

Let's Find the Critical Path

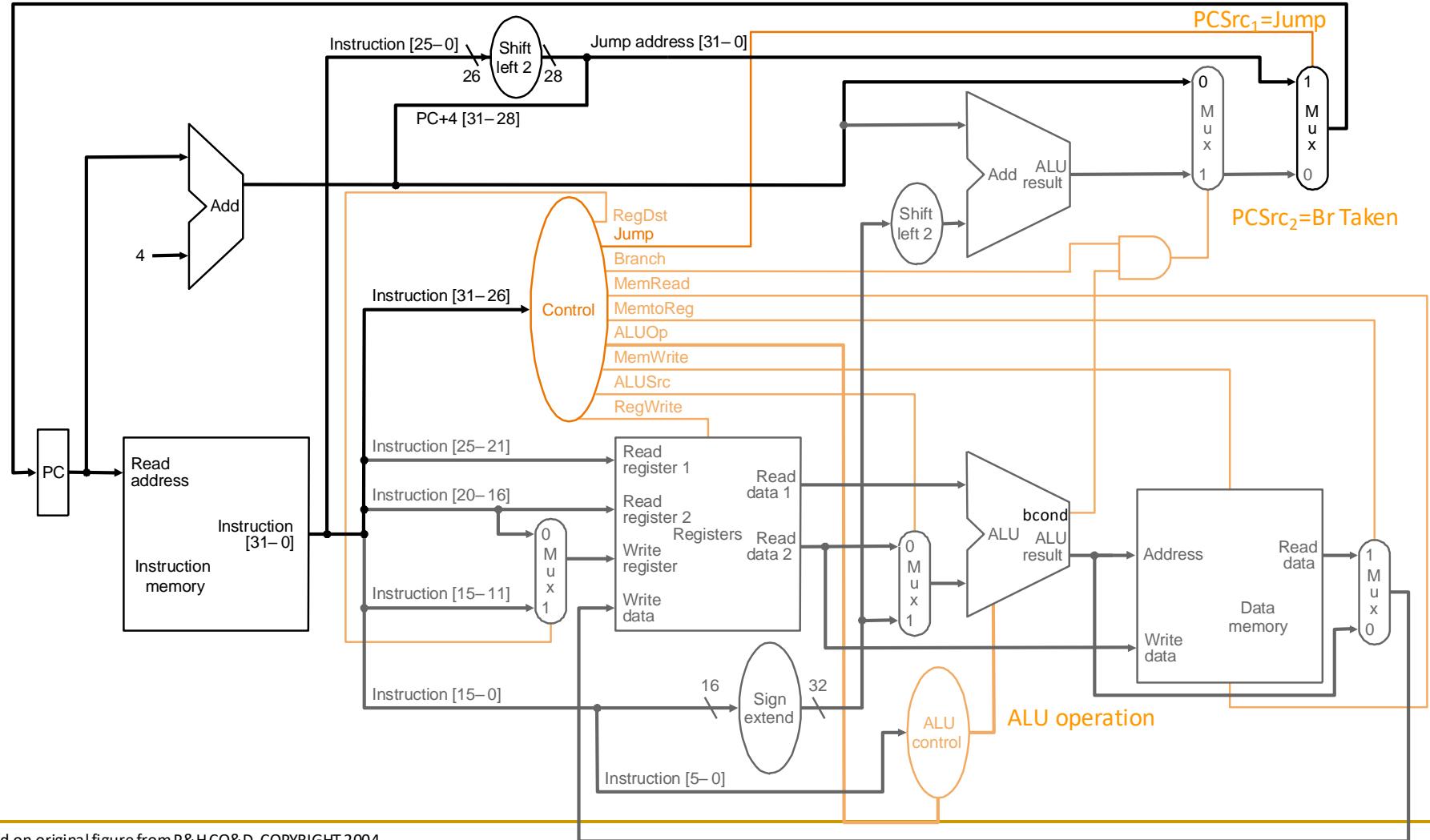


Example Single-Cycle Datapath Analysis

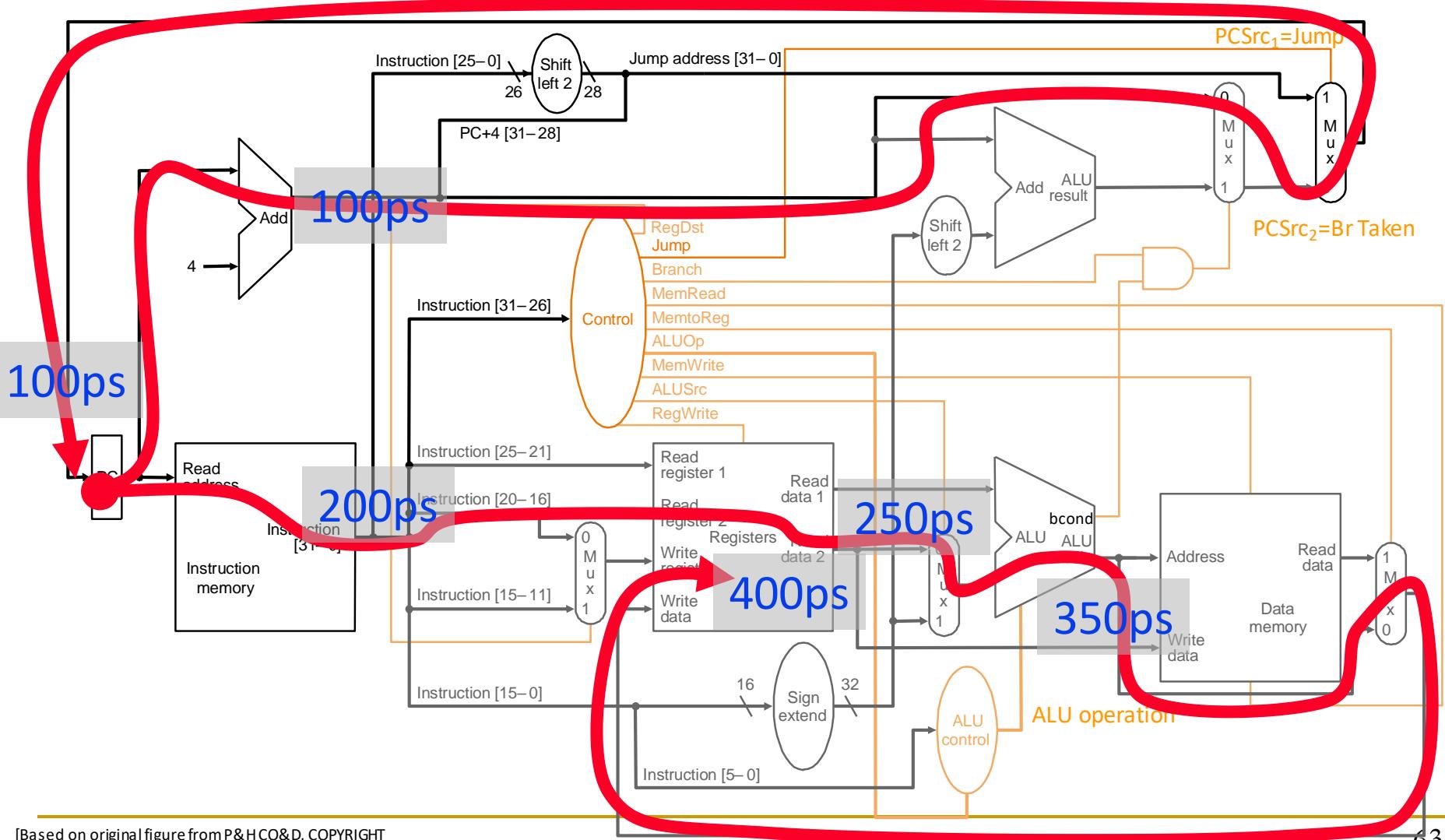
- Assume (for the design in the previous slide)
 - memory units (read or write): 200 ps
 - ALU and adders: 100 ps
 - register file (read or write): 50 ps
 - other combinational logic: 0 ps

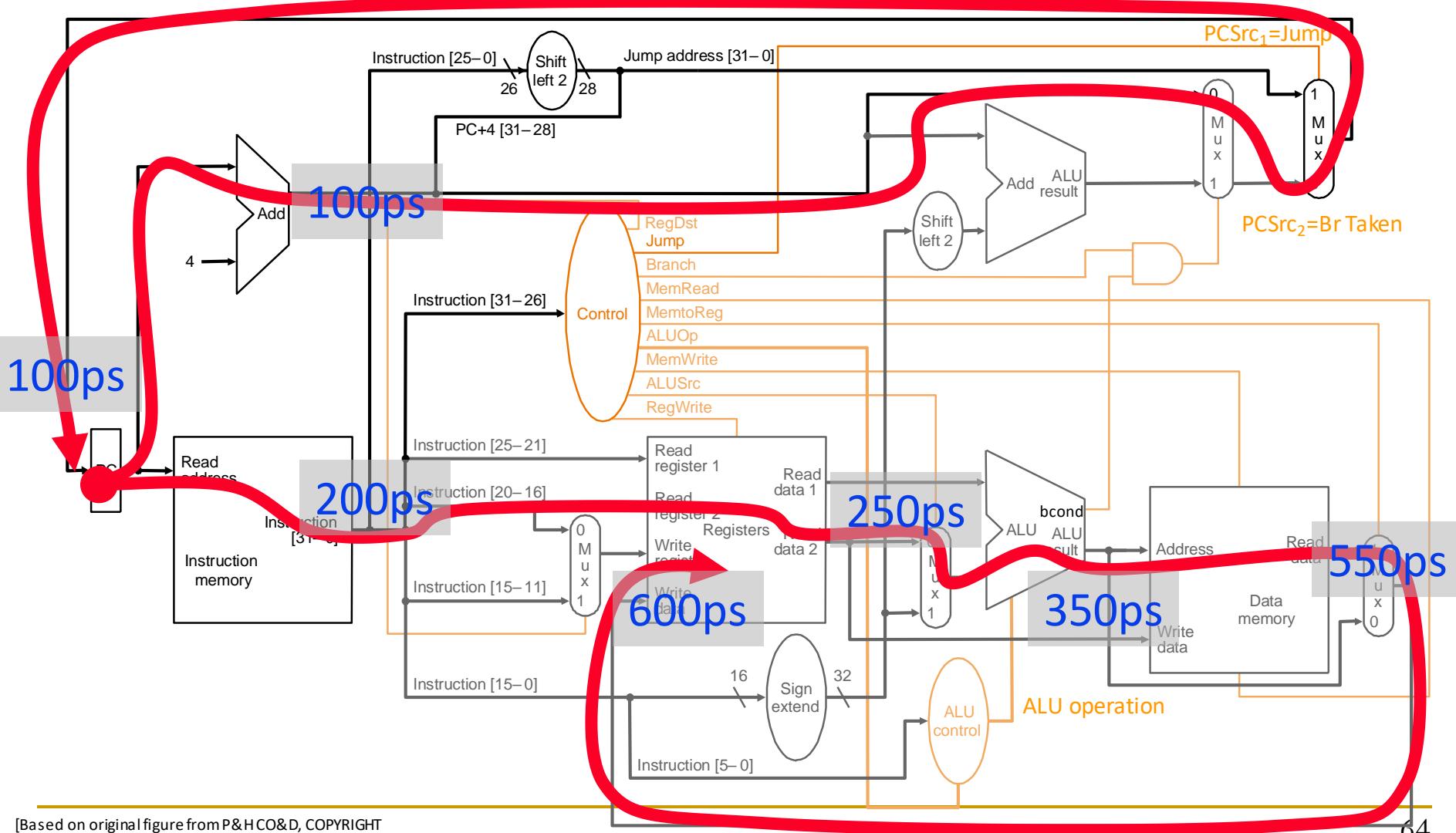
steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

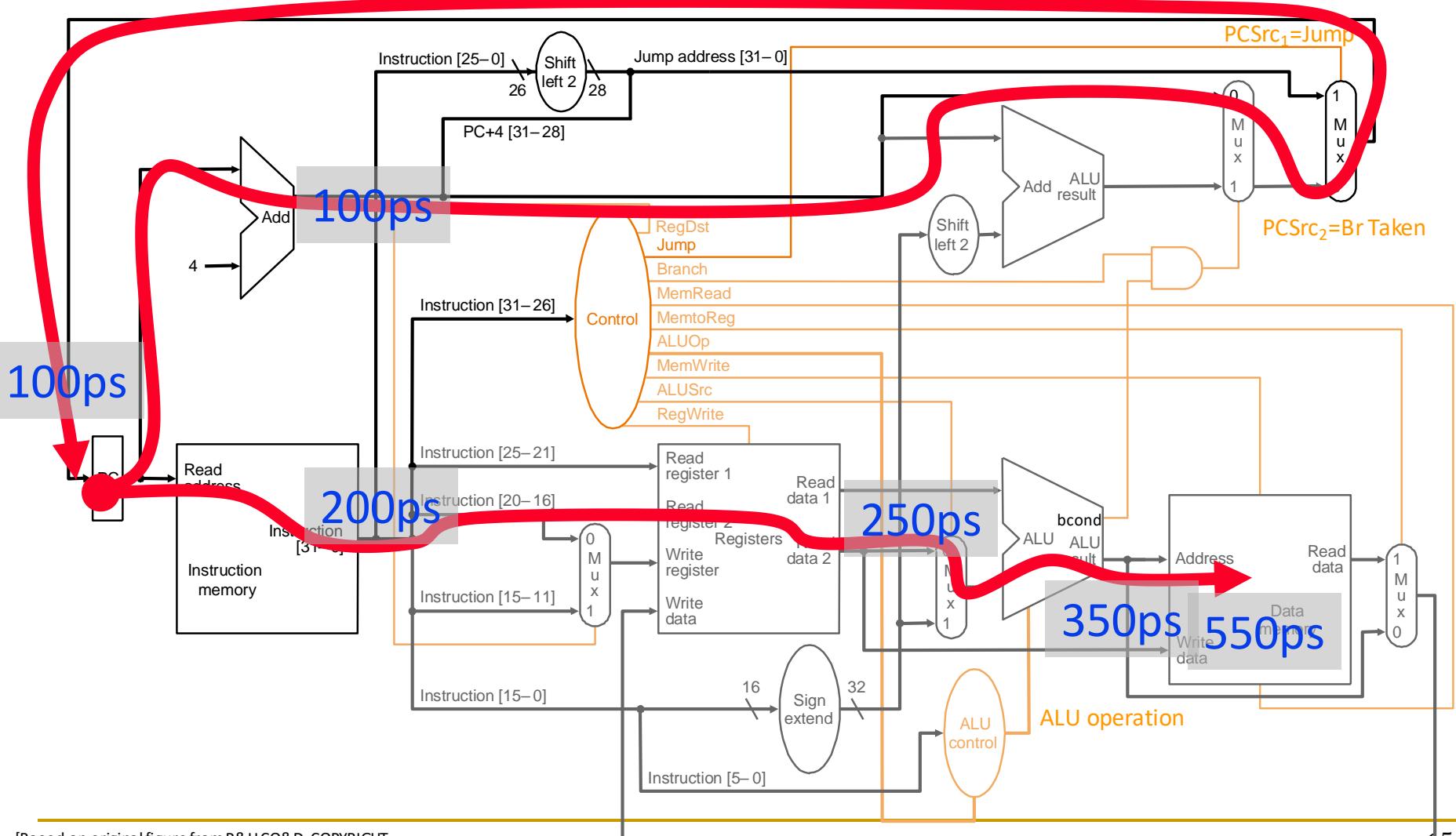
Let's Find the Critical Path



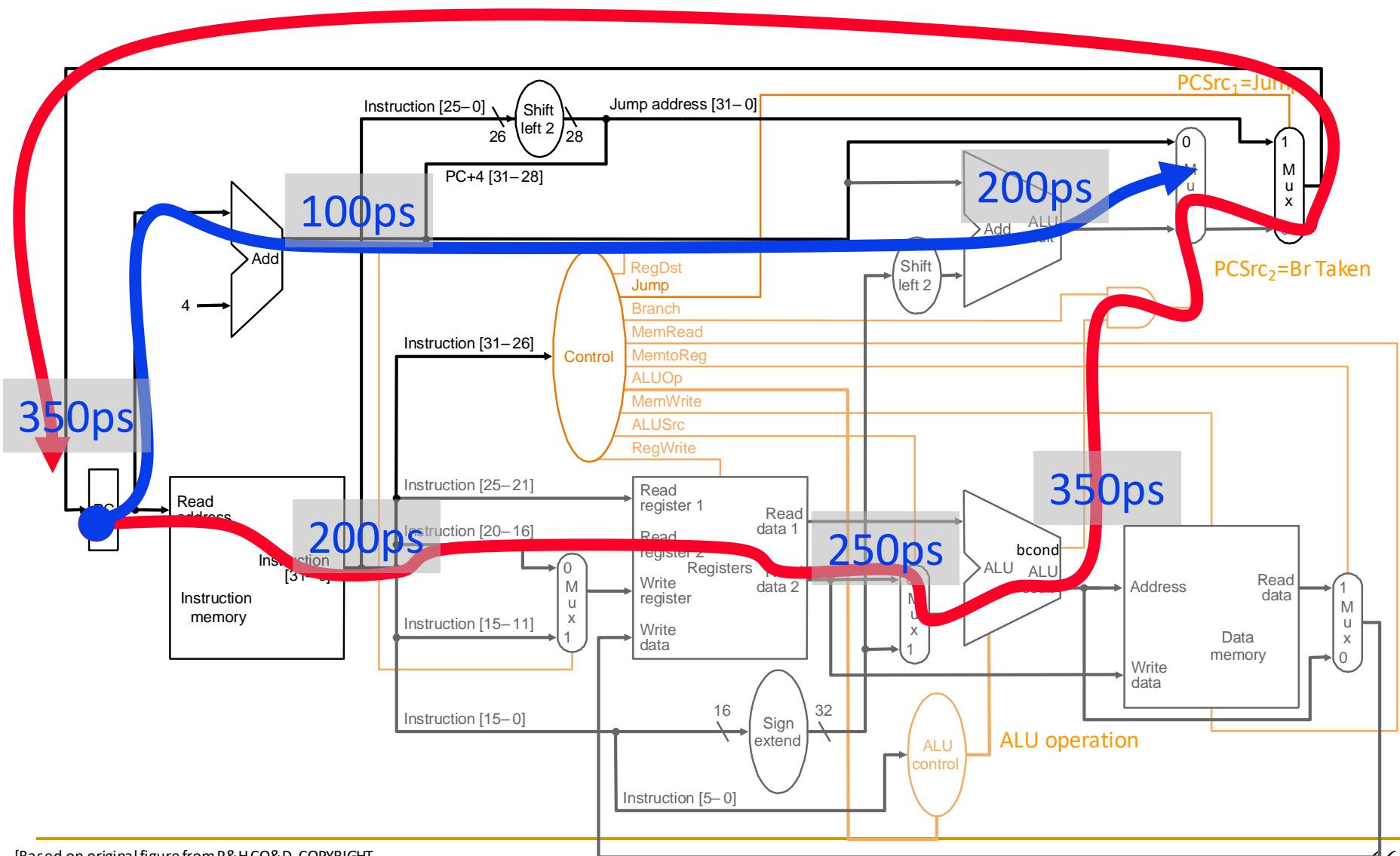
R-Type and I-Type ALU



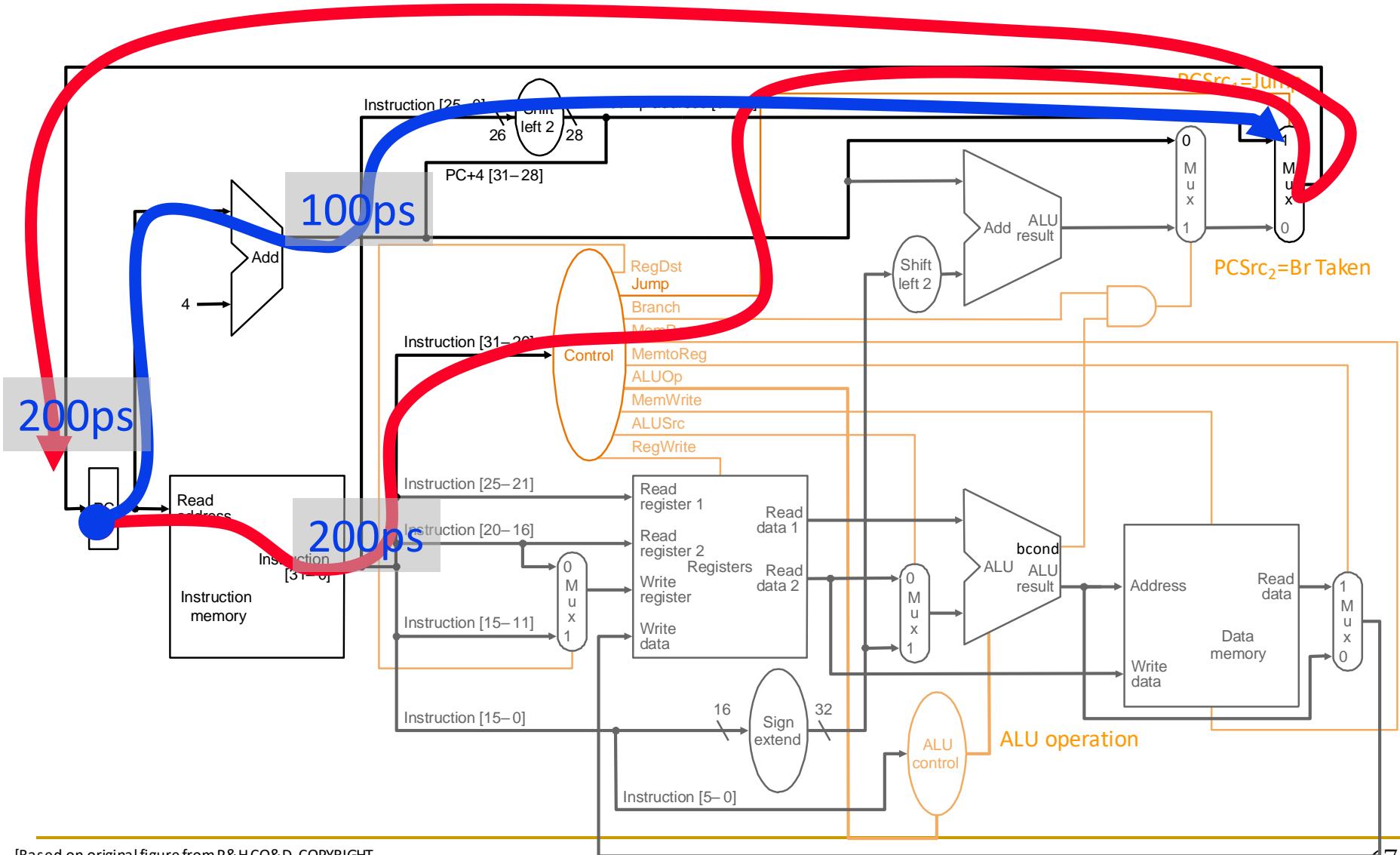




Branch Taken



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 MOVS (x86) or INDEX (VAX)?
 - Not easy to optimize/improve performance
 - Optimizing the common case (frequent instructions) does not work
 - 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
 - Mechanisms for security/safety-critical systems
 - ...
- 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



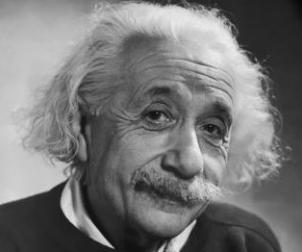
Recall: Takeaways

- 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.

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>



Can We Do Better?

Multi-Cycle Microarchitectures

Multi-Cycle Microarchitectures

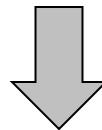
- Goal: Let each instruction take (close to) only as much time it really needs
- Idea
 - Determine clock cycle time independently of instruction processing time
 - Each instruction takes as many clock cycles as it needs to take
 - Multiple state transitions per instruction
 - The states followed by each instruction is different

Recall: The “Process Instruction” Step

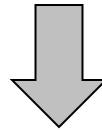
- ISA specifies abstractly what AS' should be, given an instruction and AS
 - It defines an abstract finite state machine where
 - State = programmer-visible state
 - Next-state logic = instruction execution specification
 - From ISA point of view, there are no “intermediate states” between AS and AS'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

Multi-Cycle Microarchitecture

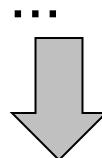
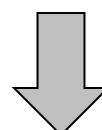
AS = Architectural (programmer visible) state
at the beginning of an instruction



Step 1: Process part of instruction in one clock cycle



Step 2: Process part of instruction in the next clock cycle



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

Benefits of Multi-Cycle Design

- **Critical path design**
 - Can keep reducing the critical path independently of the worst-case processing time of any instruction
- **Bread and butter (common case) design**
 - Can optimize the number of states it takes to execute “important” instructions that make up much of the execution time
- **Balanced design**
 - No need to provide more capability or resources than really needed
 - An instruction that needs resource X multiple times does not require multiple X’s to be implemented
 - Leads to more efficient hardware: Can reuse hardware components needed multiple times for an instruction

Downsides of Multi-Cycle Design

- Need to store the intermediate results at the end of each clock cycle
 - Hardware overhead for registers
 - Register setup/hold overhead paid multiple times for an instruction

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

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

A Multi-Cycle Microarchitecture

A Closer Look

How Do We Implement This?

- Maurice Wilkes, "[The Best Way to Design an Automatic Calculating Machine](#)," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.



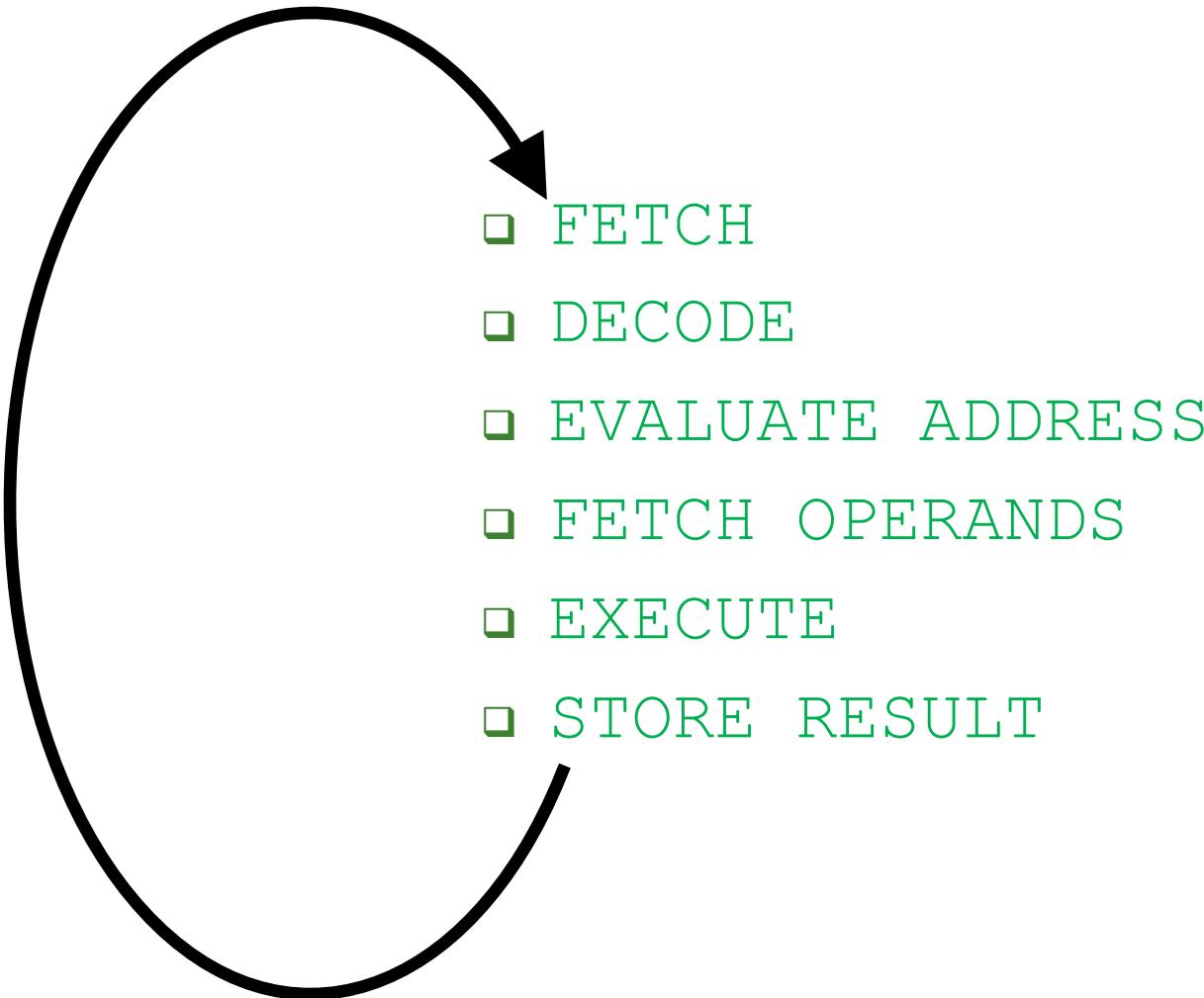
- An elegant implementation:
 - [The concept of microcoded/microprogrammed machines](#)

Multi-Cycle Microarchitectures

■ Key Idea for Realization

- ❑ One can implement the “process instruction” step as a finite state machine that sequences between states and eventually returns back to the “fetch instruction” state
- ❑ A state is defined by the control signals asserted in it
- ❑ Control signals for the next state are determined in current state

Recall: The Instruction Processing “Cycle”

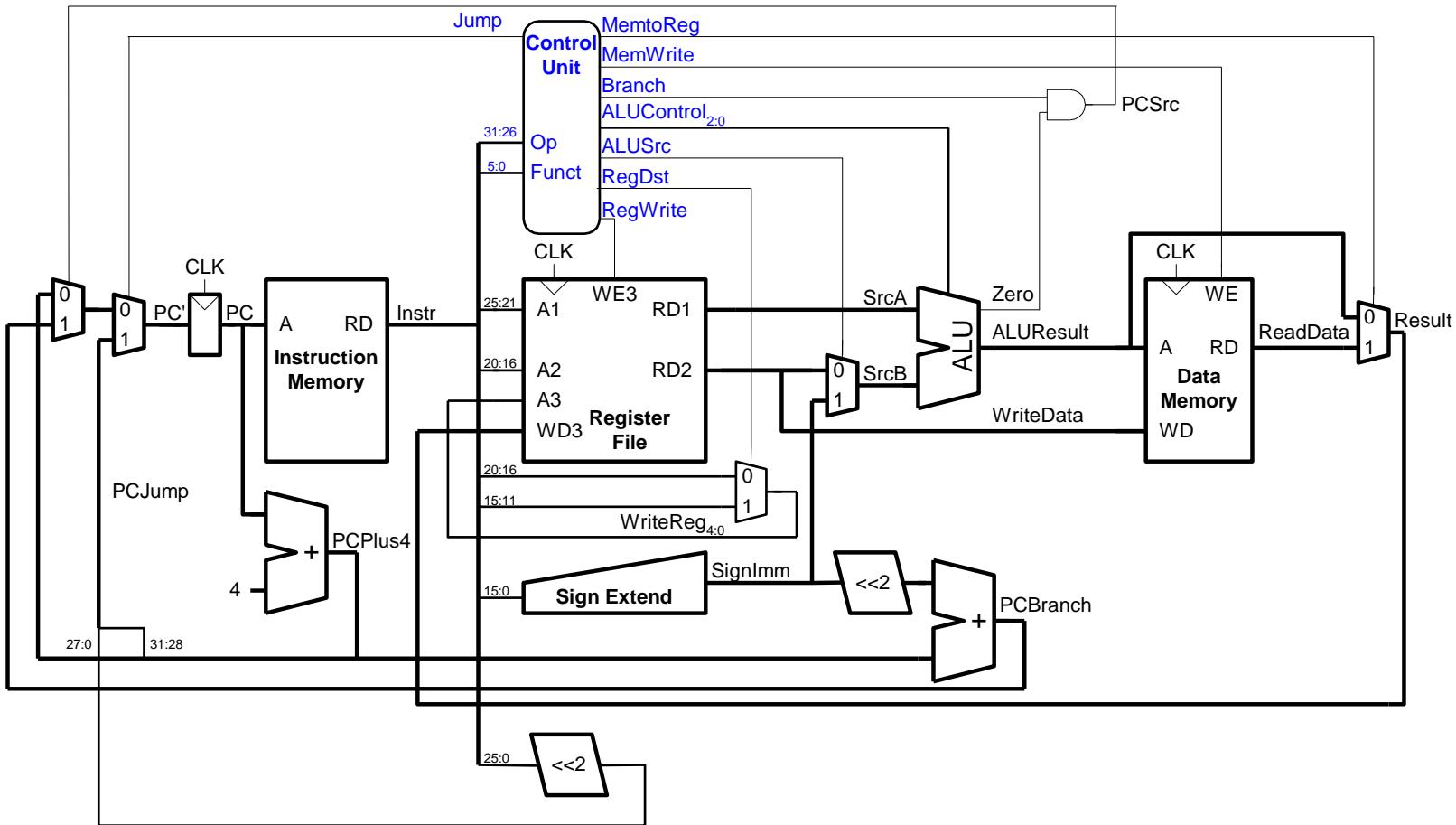


A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into “states”
 - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
 - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control two things:
 - How the datapath should process the data
 - How to generate the control signals for the (next) clock cycle

One Example Multi-Cycle Microarchitecture

Remember: Single-Cycle MIPS Processor



Multi-Cycle MIPS Processor

- **Single-cycle microarchitecture:**
 - cycle time limited by longest instruction (lw) → **low clock frequency**
 - three adders/ALUs and two memories → **high hardware cost**
- **Multi-cycle microarchitecture:**
 - + higher clock frequency
 - + simpler instructions take few clock cycles
 - + reuse expensive hardware across multiple cycles
 - sequencing overhead paid many times
 - hardware overhead for storing intermediate results
- **Multi-cycle requires the same design steps as single cycle:**
 - datapath
 - control logic

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
 - One memory stores instructions, the other data
 - We want to use a single memory (lower cost)

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
 - One memory stores instructions, the other data
 - We want to use a single memory (lower cost)
- **Single-cycle microarchitecture needs three adders**
 - ALU, PC, Branch address calculation
 - We want to use only one ALU for all operations (lower cost)

What Do We Want To Optimize?

- **Single-cycle microarchitecture uses two memories**
 - One memory stores instructions, the other data
 - We want to use a single memory (lower cost)
- **Single-cycle microarchitecture needs three adders**
 - ALU, PC, Branch address calculation
 - We want to use only one ALU for all operations (lower cost)
- **Single-cycle microarchitecture: each instruction takes one cycle**
 - The slowest instruction slows down every single instruction
 - We want to determine clock cycle time independently of instruction processing time
 - Divide each instruction into multiple clock cycles
 - Simpler instructions can be very fast (compared to the slowest)

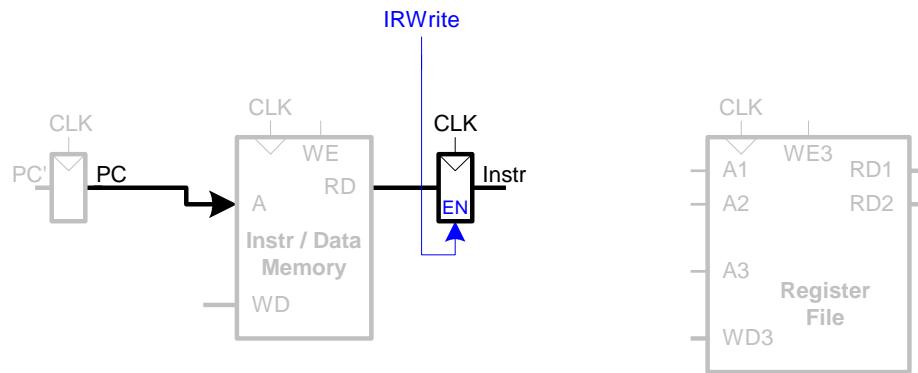
Let's Construct the Multi-Cycle Datapath

Consider the lw Instruction

- For an instruction such as: **lw \$t0, 0x20(\$t1)**
- We need to:
 - Read the instruction from memory
 - Then read **\$t1** from register array
 - Add the immediate value (**0x20**) to calculate the memory address
 - Read the content of this address
 - Write to the register **\$t0** this content

Multi-Cycle Datapath: Instruction Fetch

- We will consider **lw**, but fetch is the same for all instructions
 - STEP 1: Fetch instruction

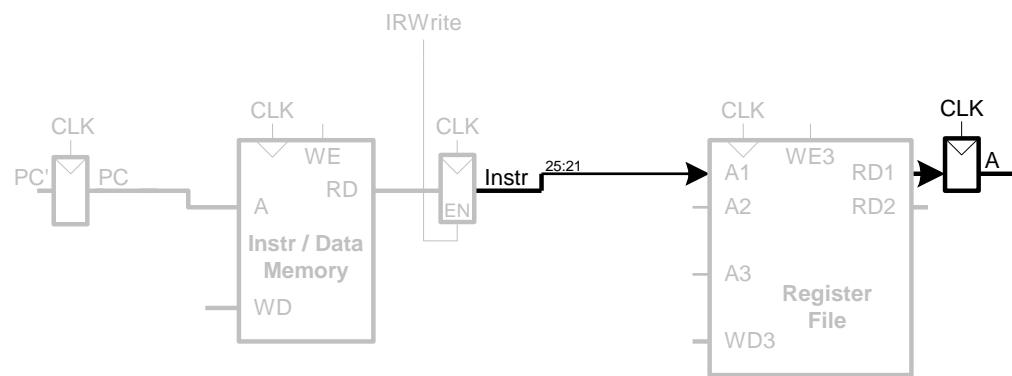


read from the memory location $[rs] + imm$ to location $[rt]$

I-Type

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

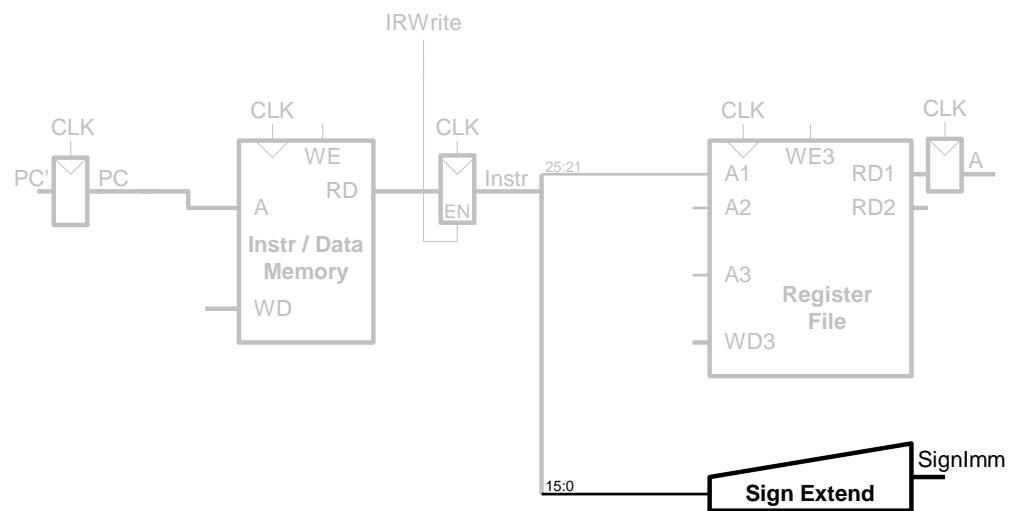
Multi-Cycle Datapath: lw register read



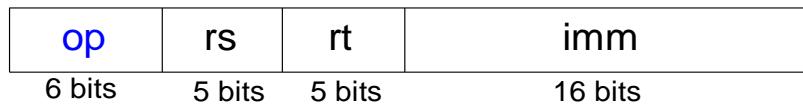
I-Type

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

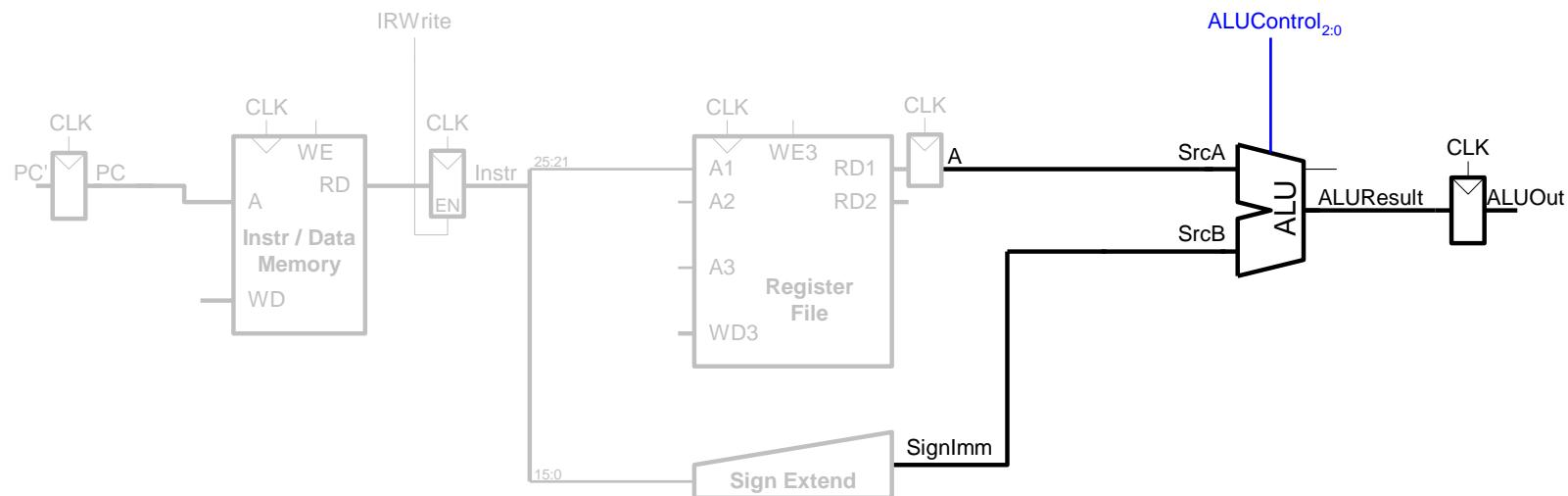
Multi-Cycle Datapath: lw immediate



I-Type



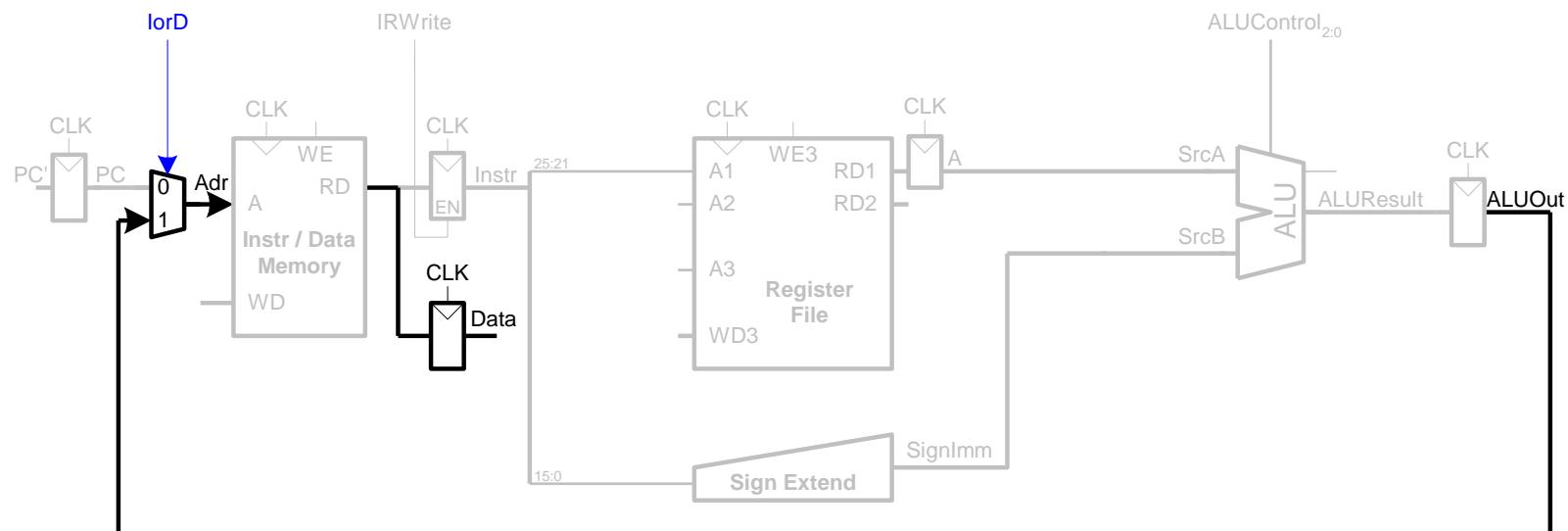
Multi-Cycle Datapath: lw address



I-Type

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

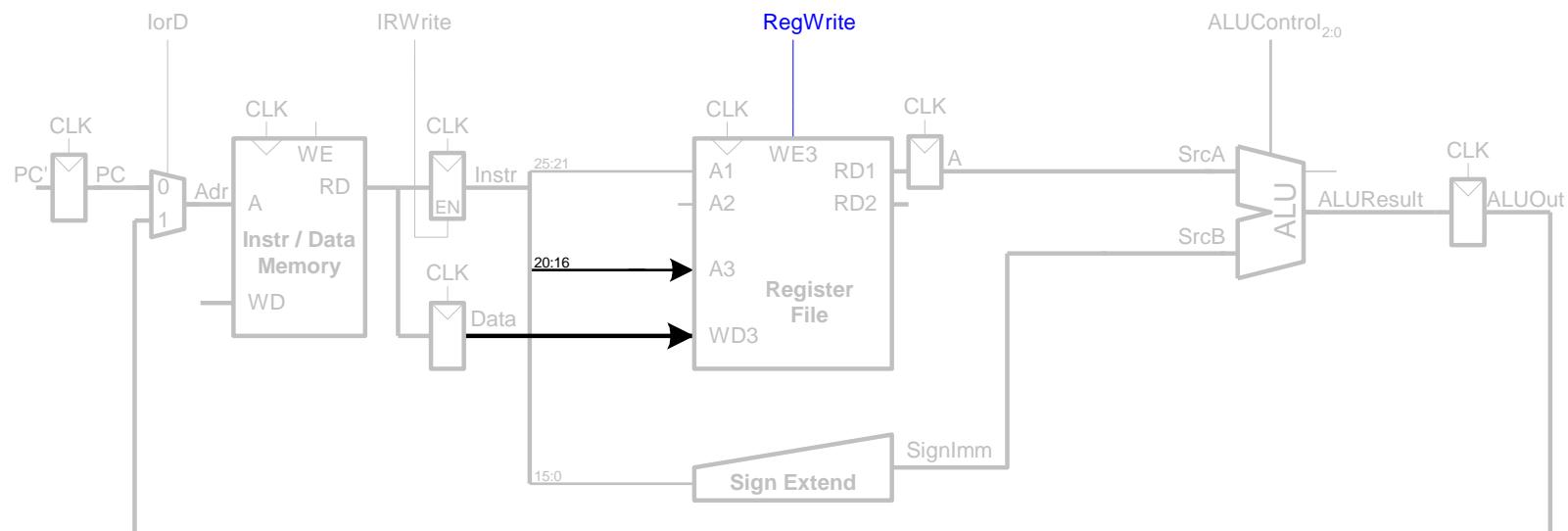
Multi-Cycle Datapath: lw memory read



I-Type

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

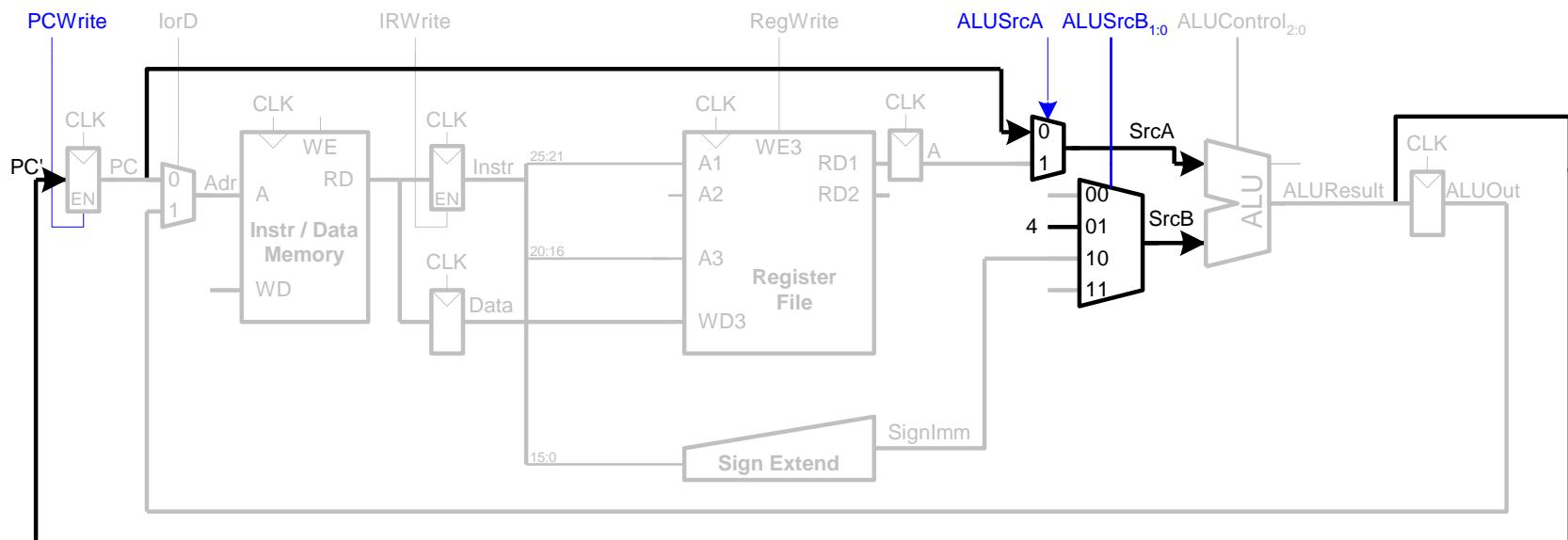
Multi-Cycle Datapath: lw write register



I-Type

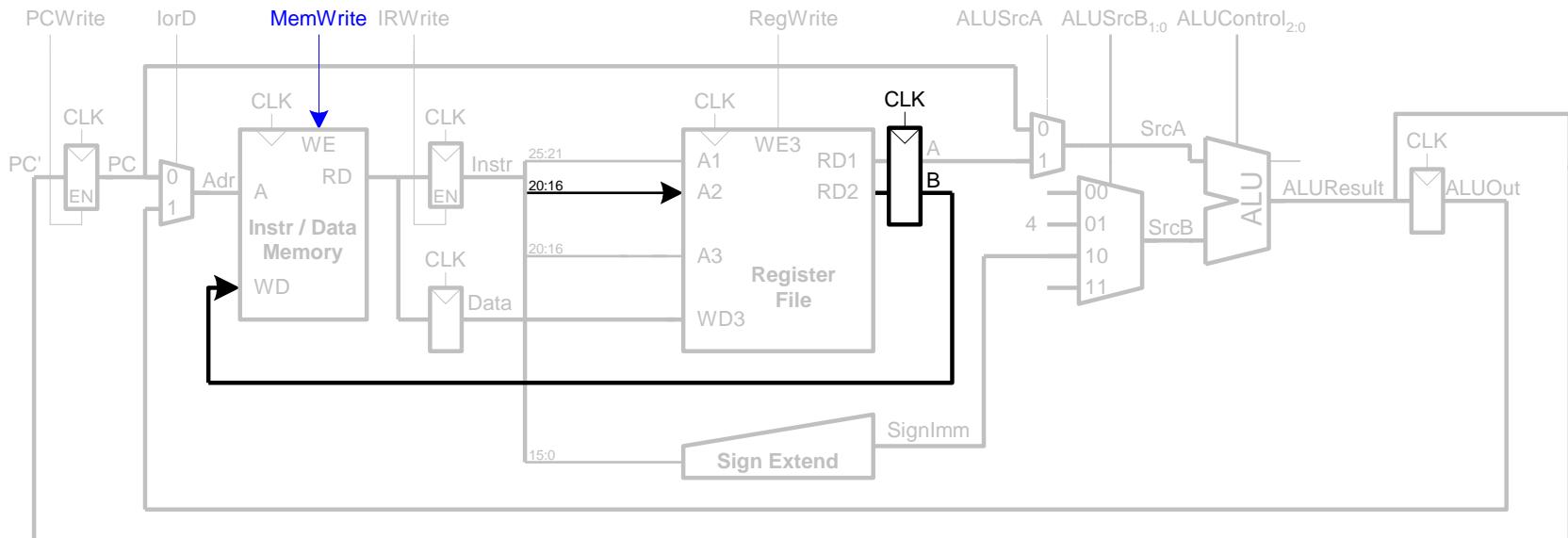
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-Cycle Datapath: increment PC



Multi-Cycle Datapath: sw

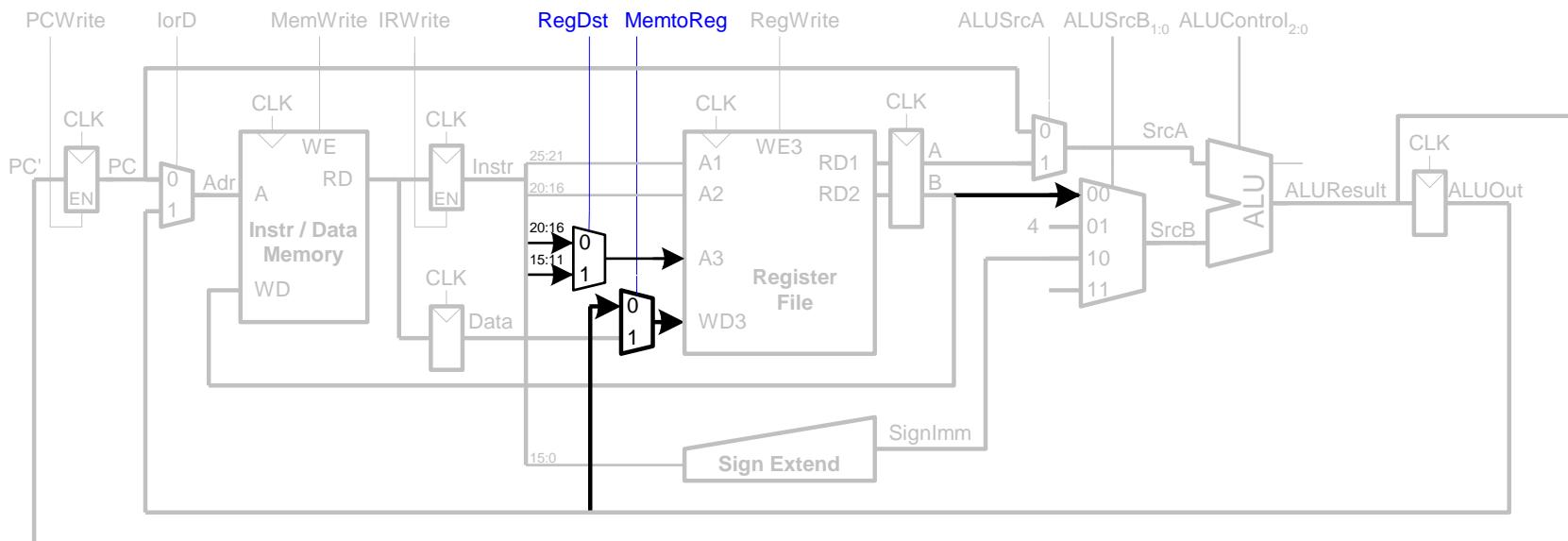
■ Write data in rt to memory



Multi-Cycle Datapath: R-type Instructions

■ Read from rs and rt

- Write ALUResult to register file
- Write to rd (instead of rt)

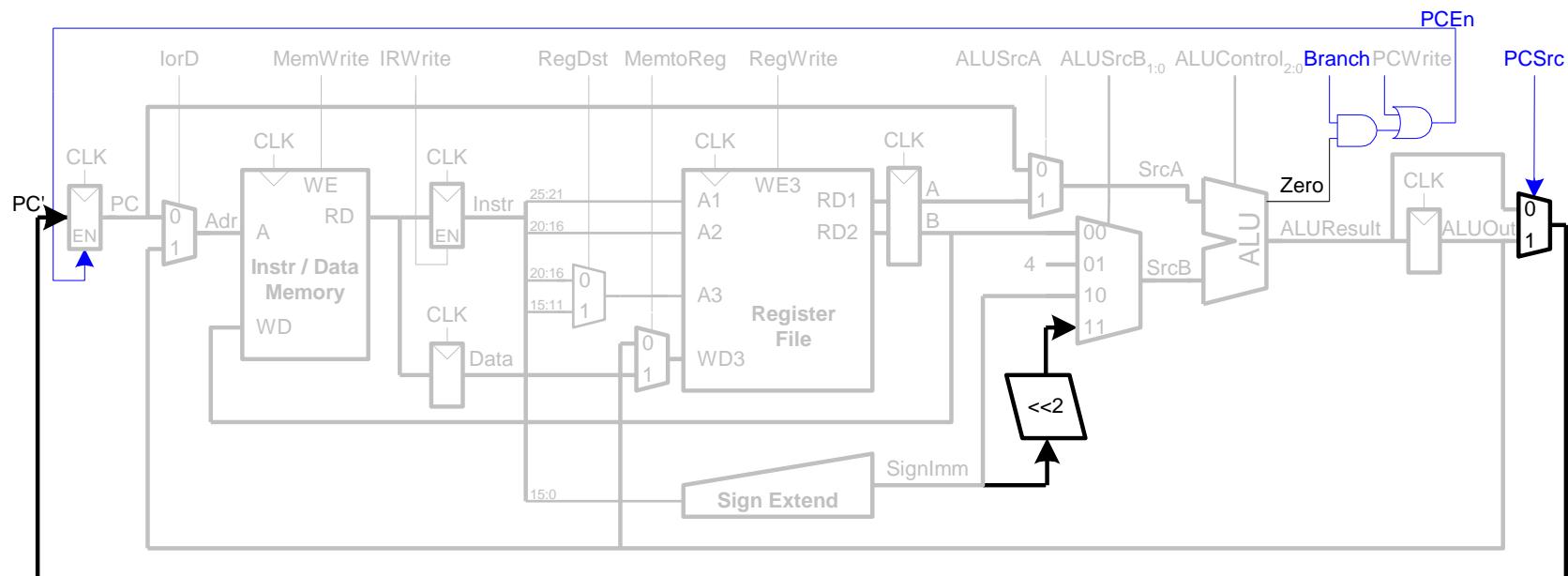


Multi-Cycle Datapath: beq

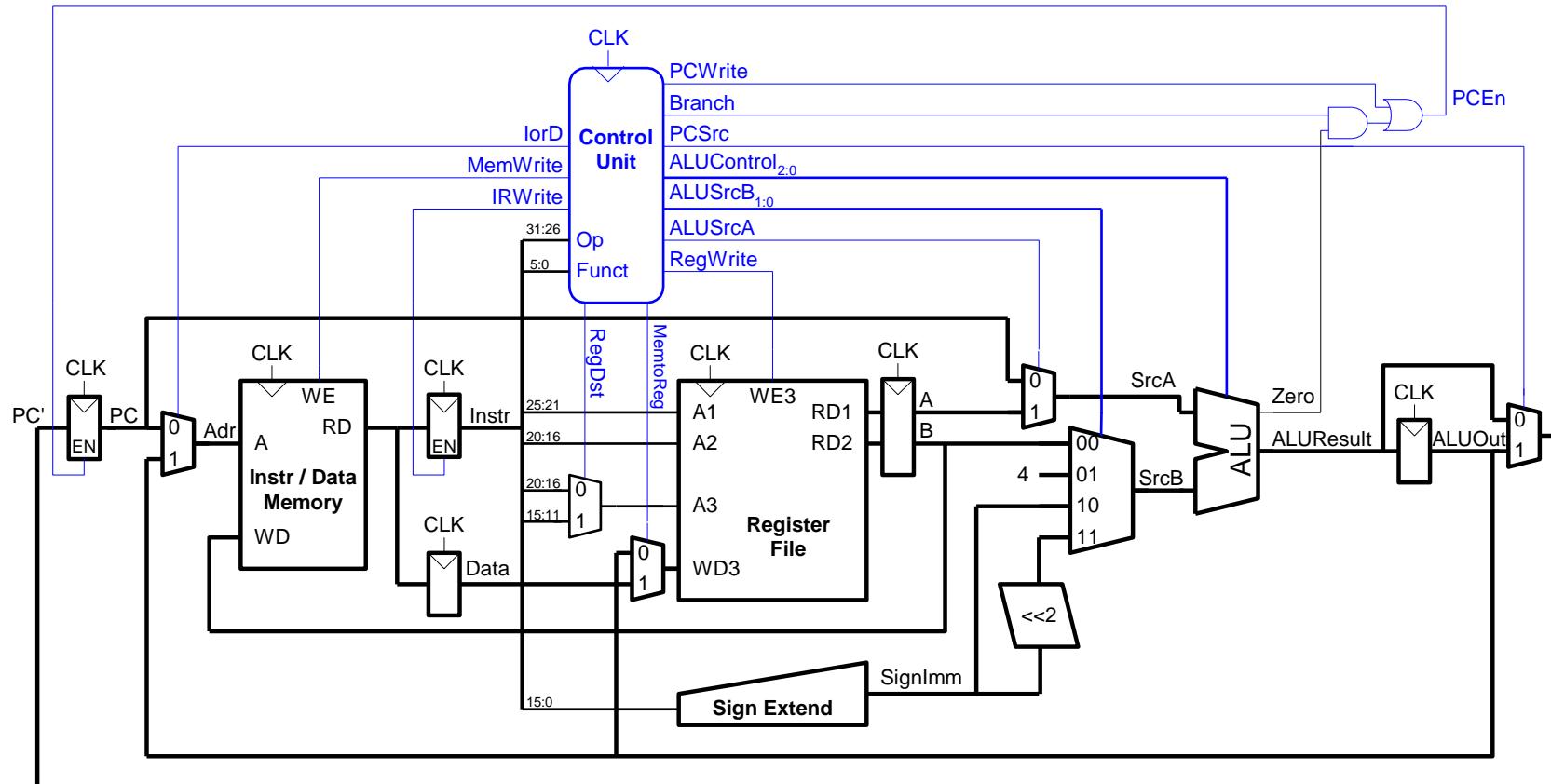
- Determine whether values in rs and rt are equal

- Calculate branch target address:

$$\text{Target Address} = (\text{sign-extended immediate} \ll 2) + (\text{PC} + 4)$$

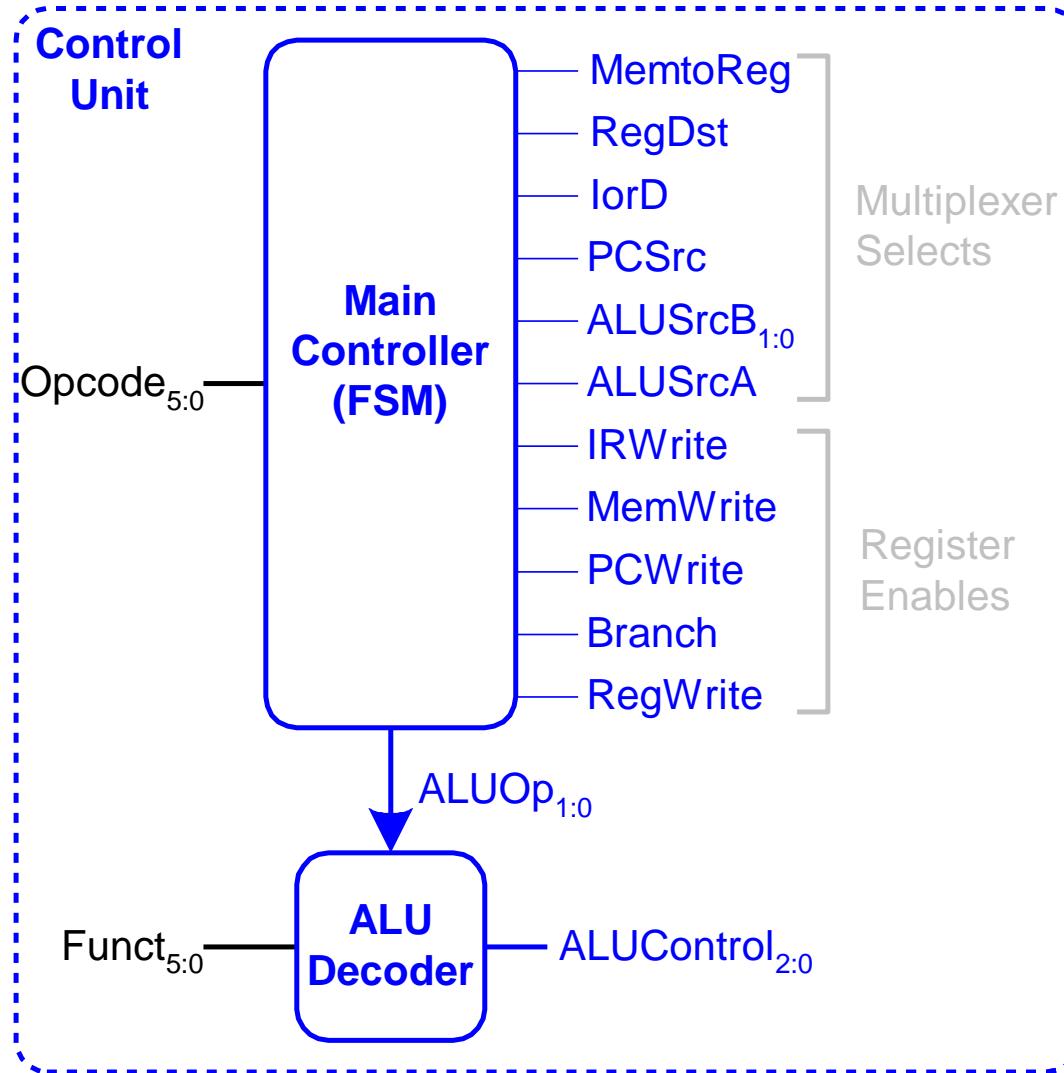


Complete Multi-Cycle Processor

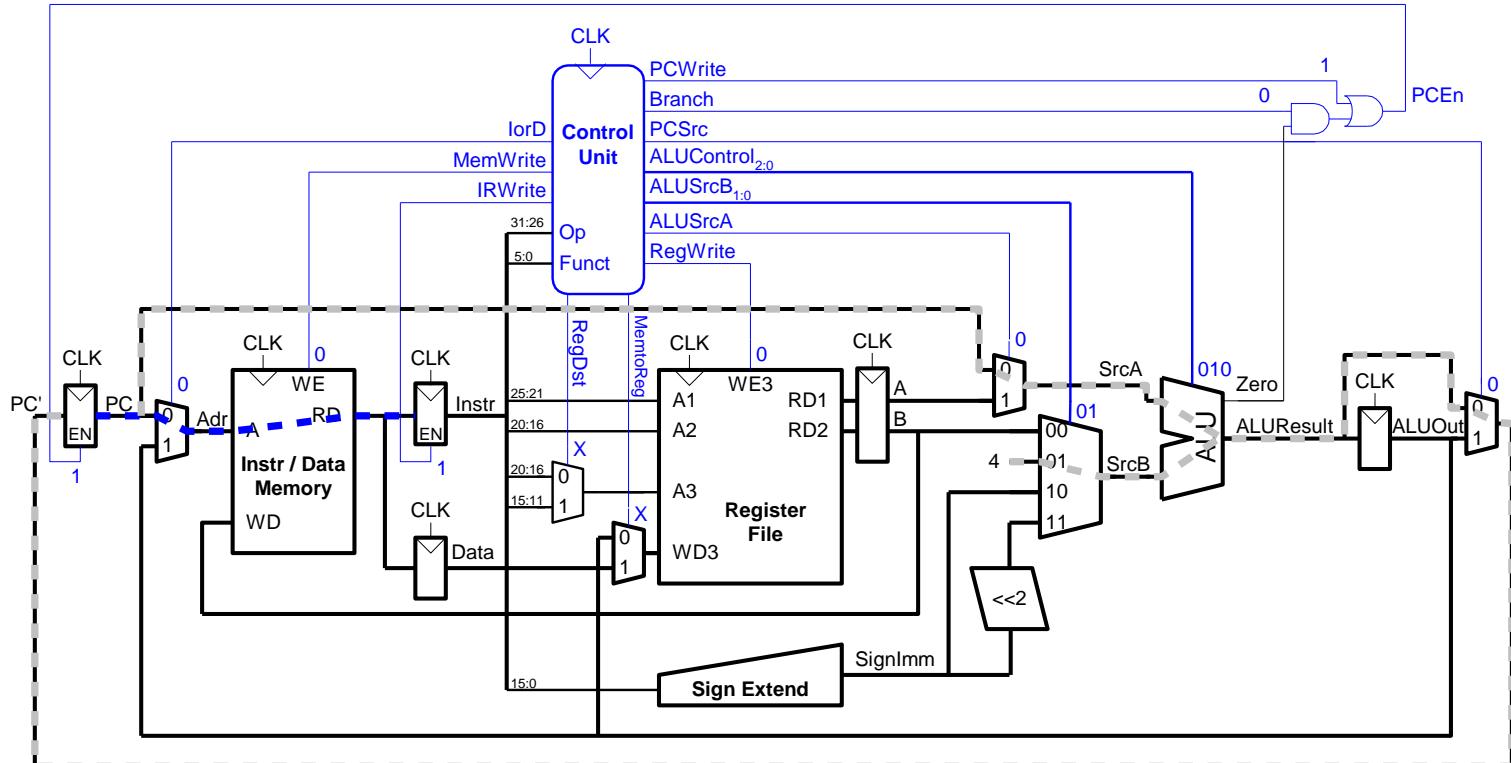
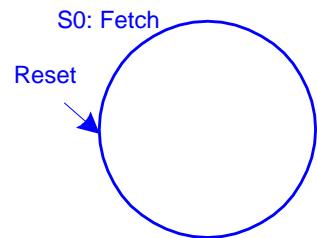


Let's Construct the Multi-Cycle Control Logic

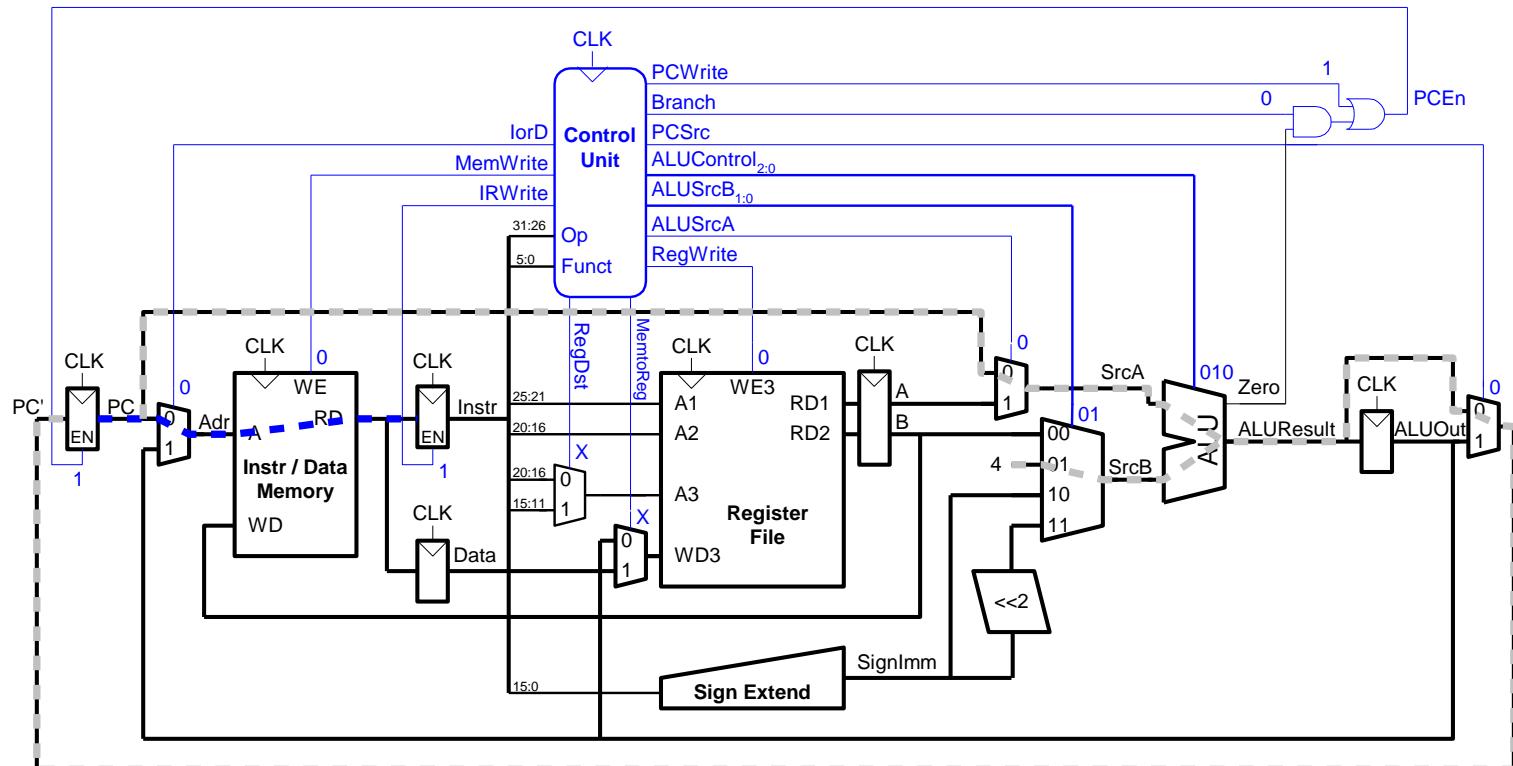
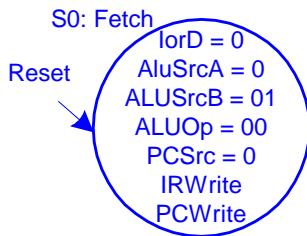
Control Unit



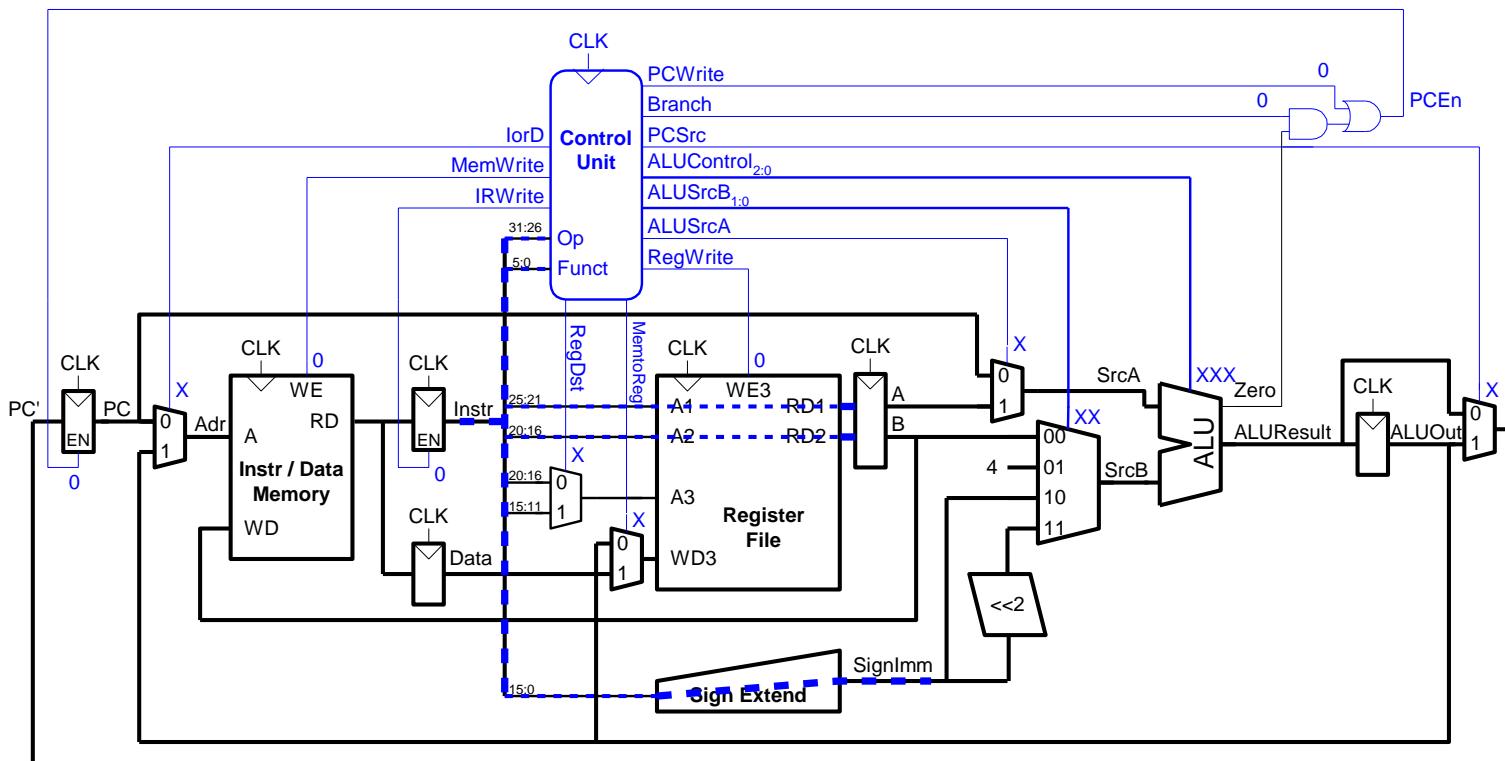
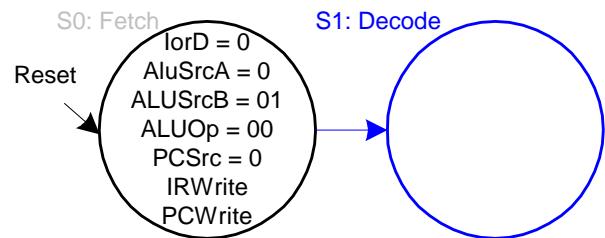
Main Controller FSM: Fetch



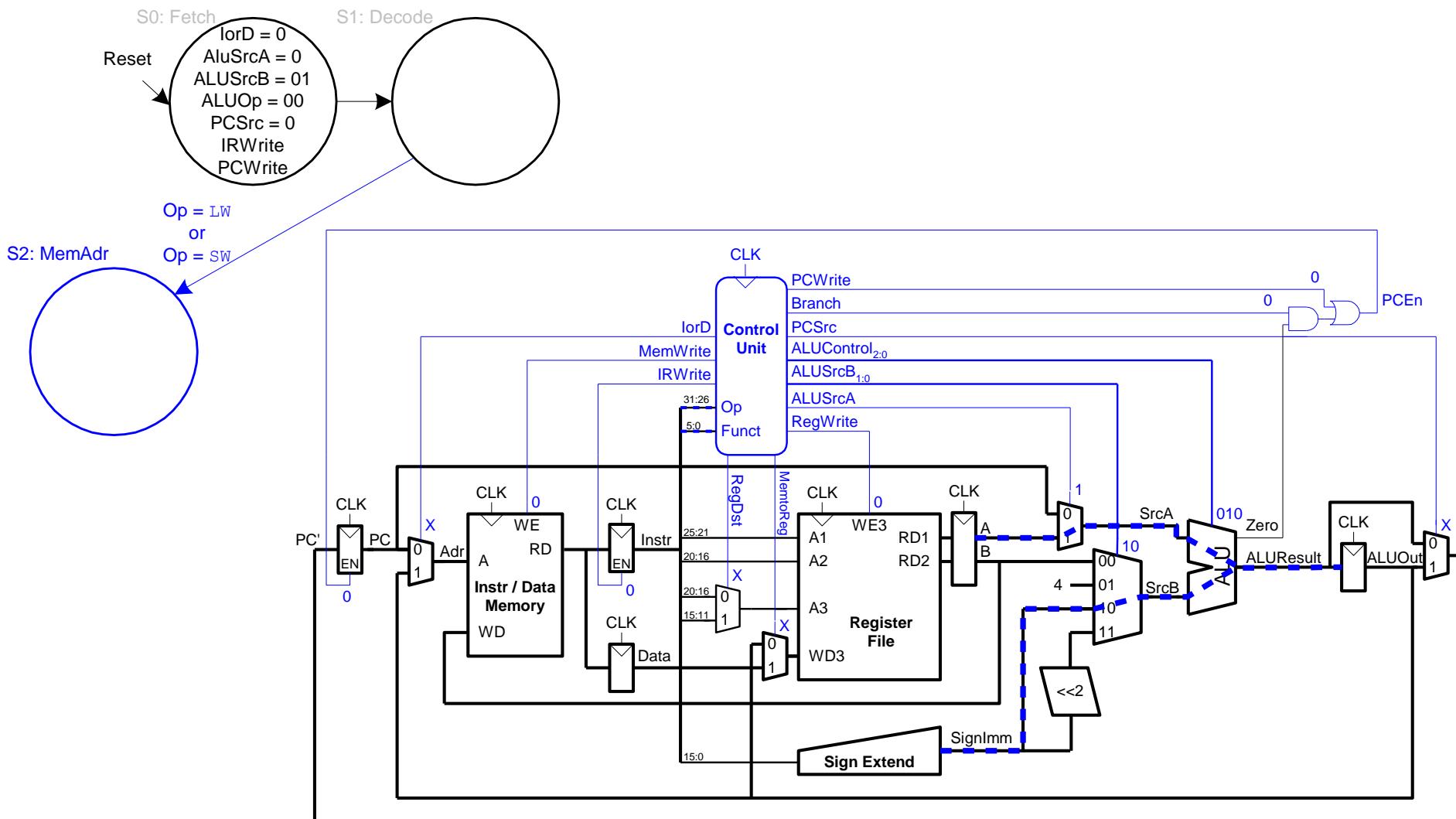
Main Controller FSM: Fetch



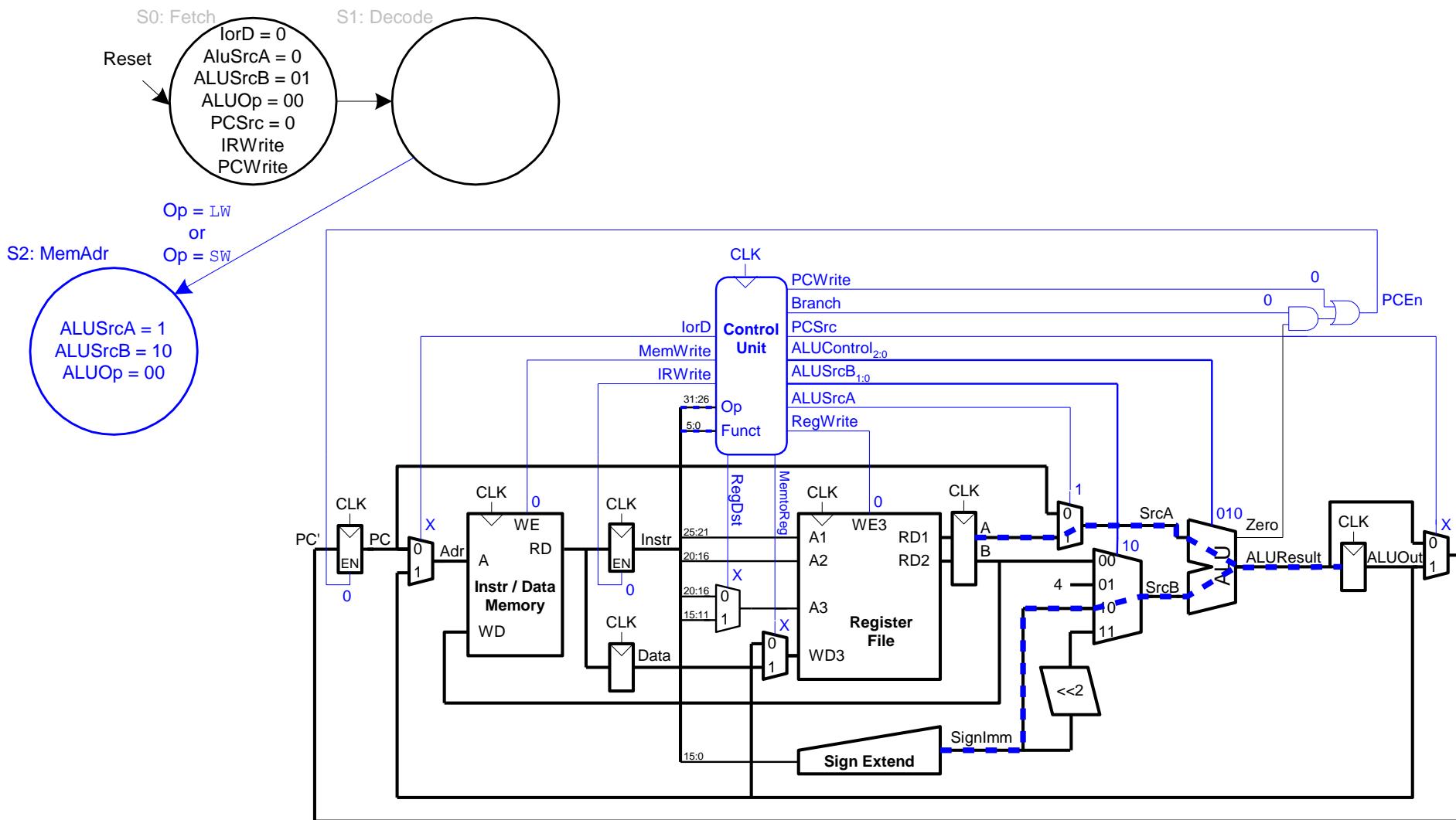
Main Controller FSM: Decode



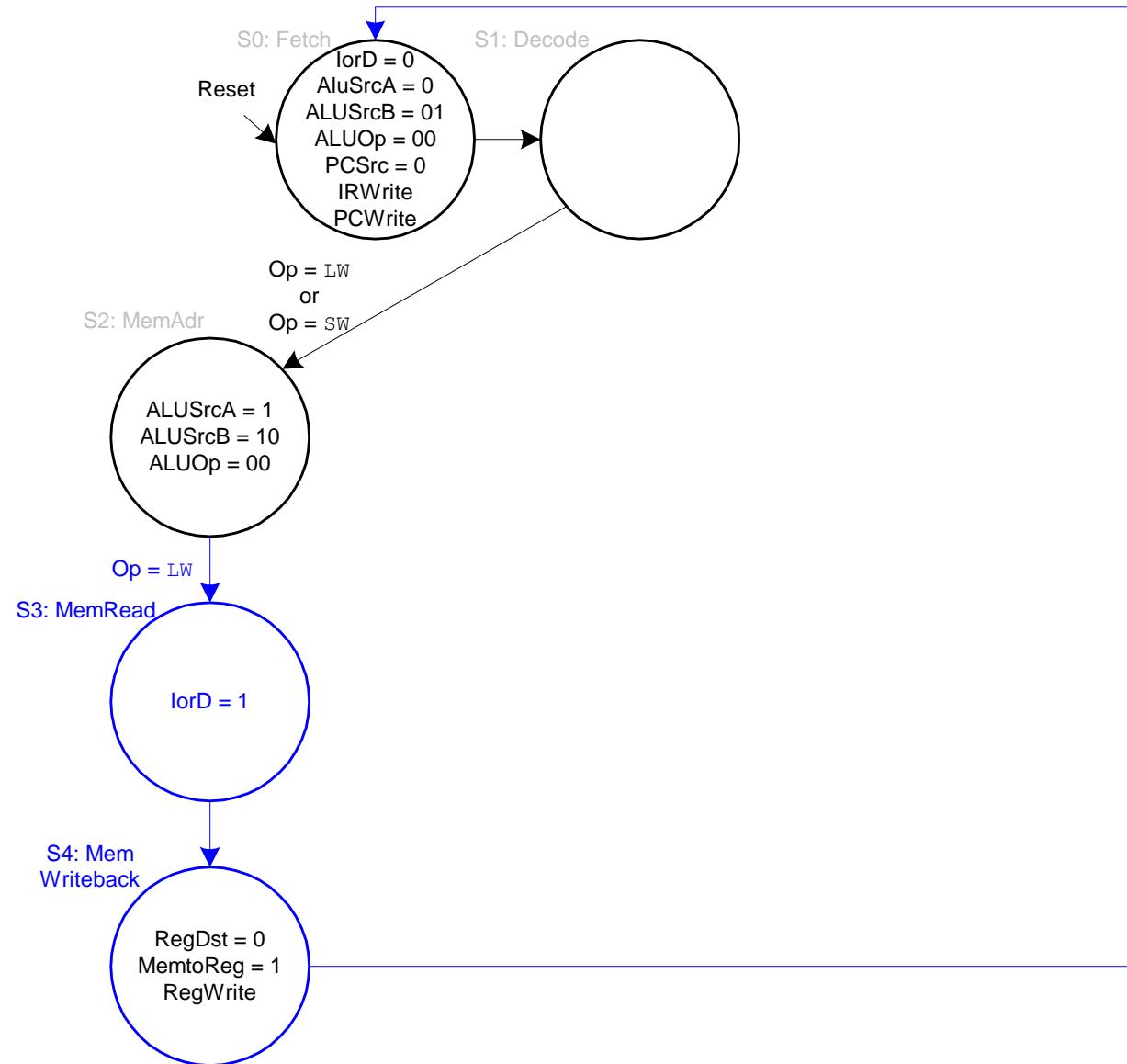
Main Controller FSM: Address Calculation



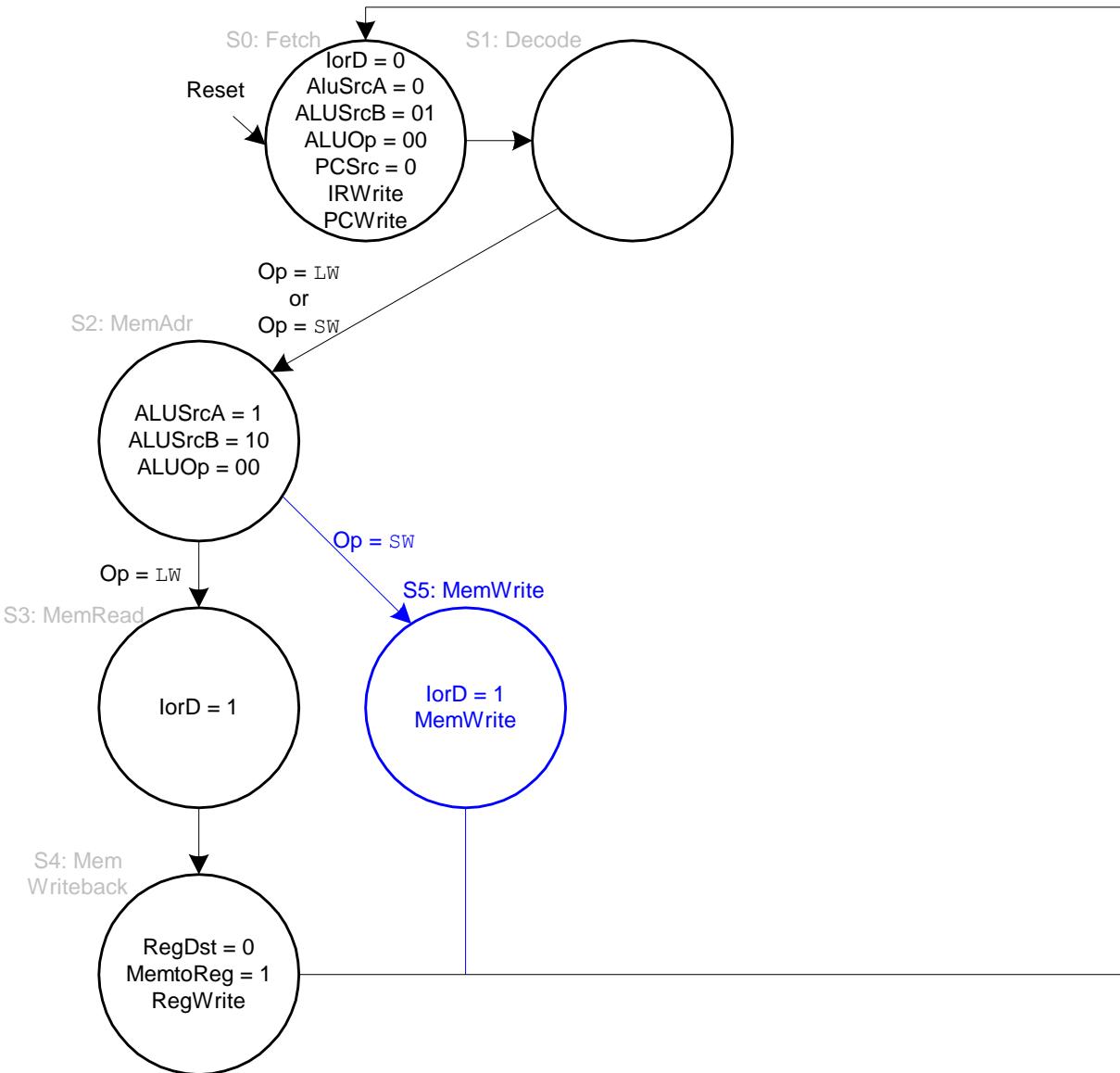
Main Controller FSM: Address Calculation



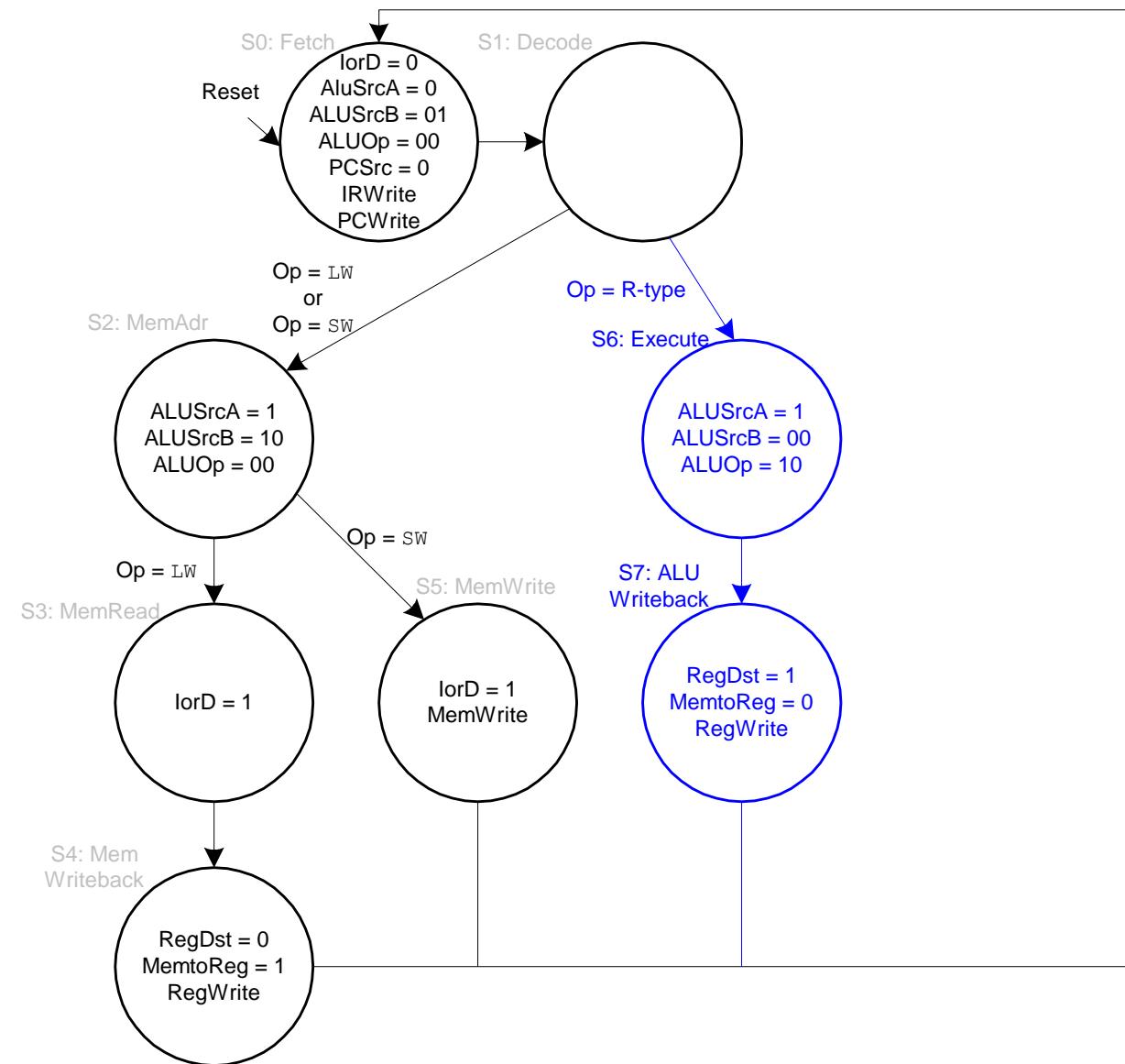
Main Controller FSM: lw



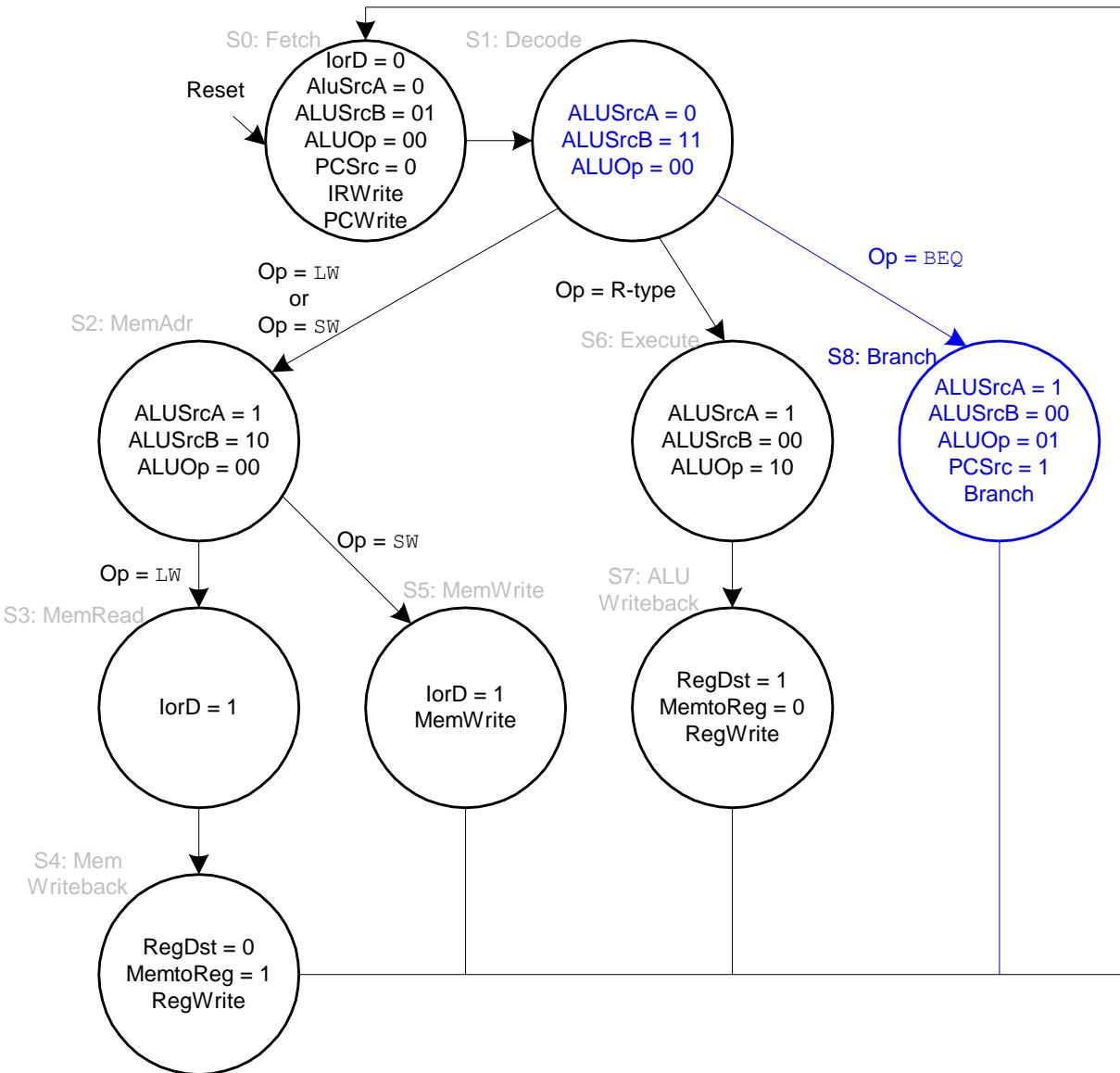
Main Controller FSM: SW



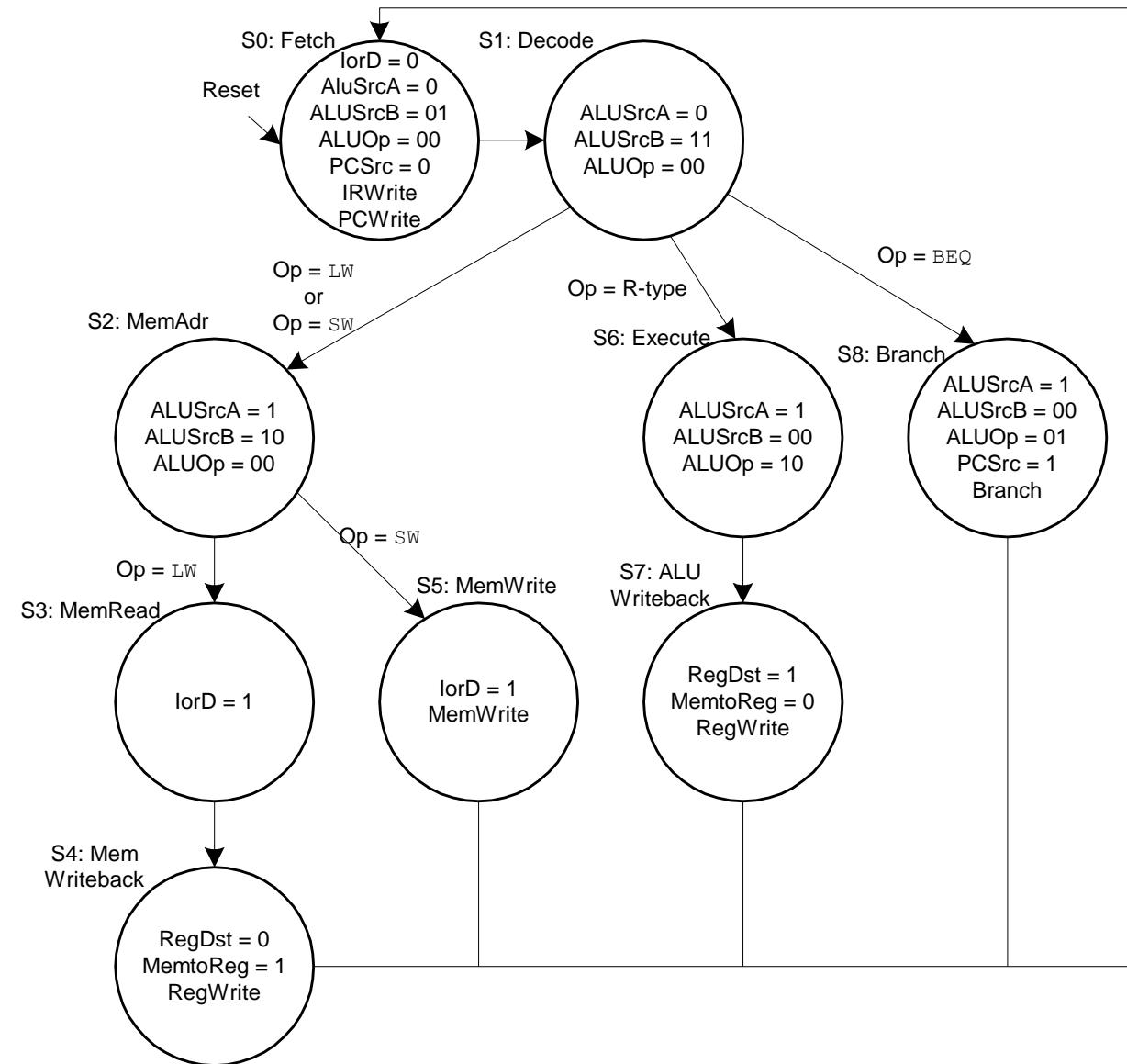
Main Controller FSM: R-Type



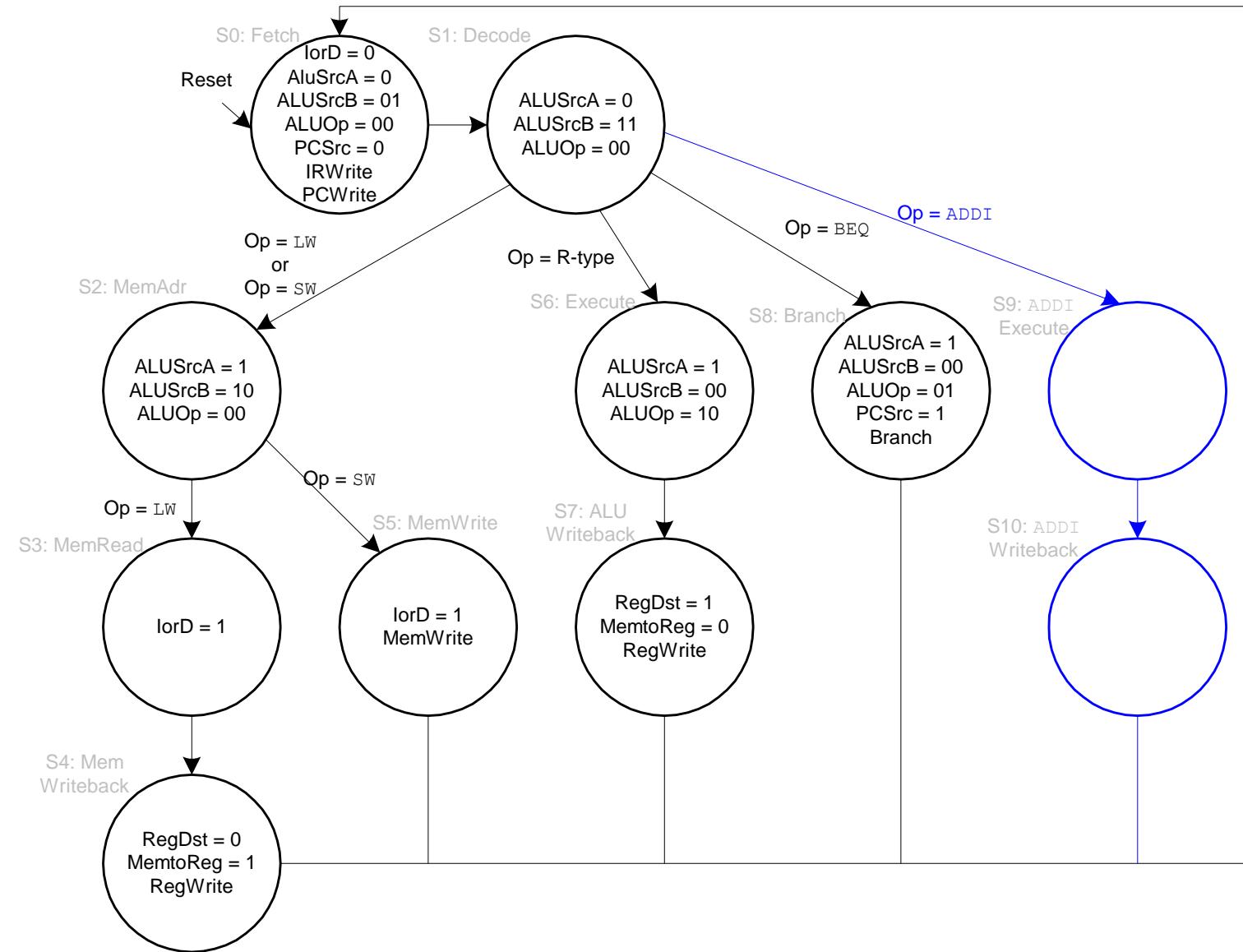
Main Controller FSM: beq



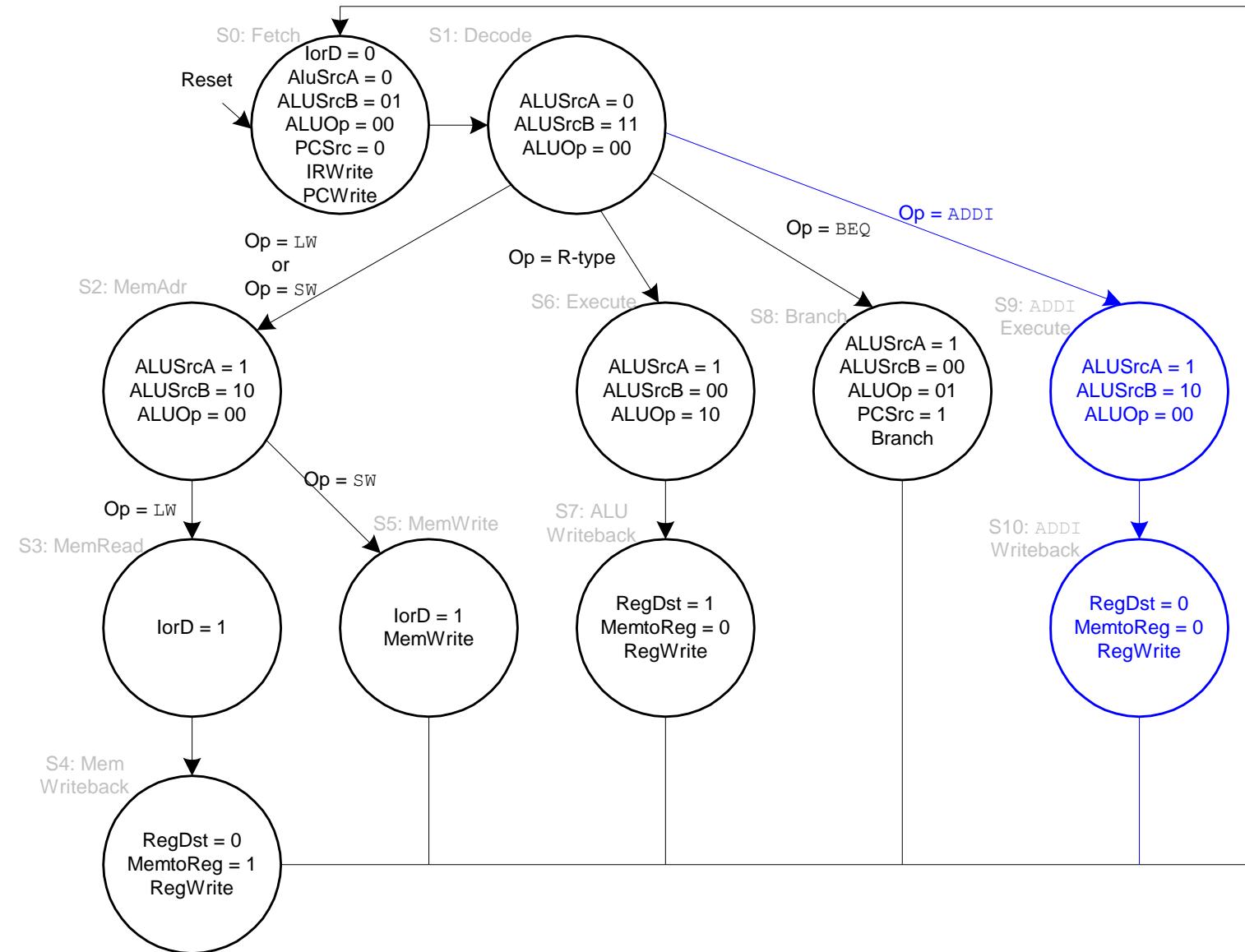
Complete Multi-Cycle Controller FSM



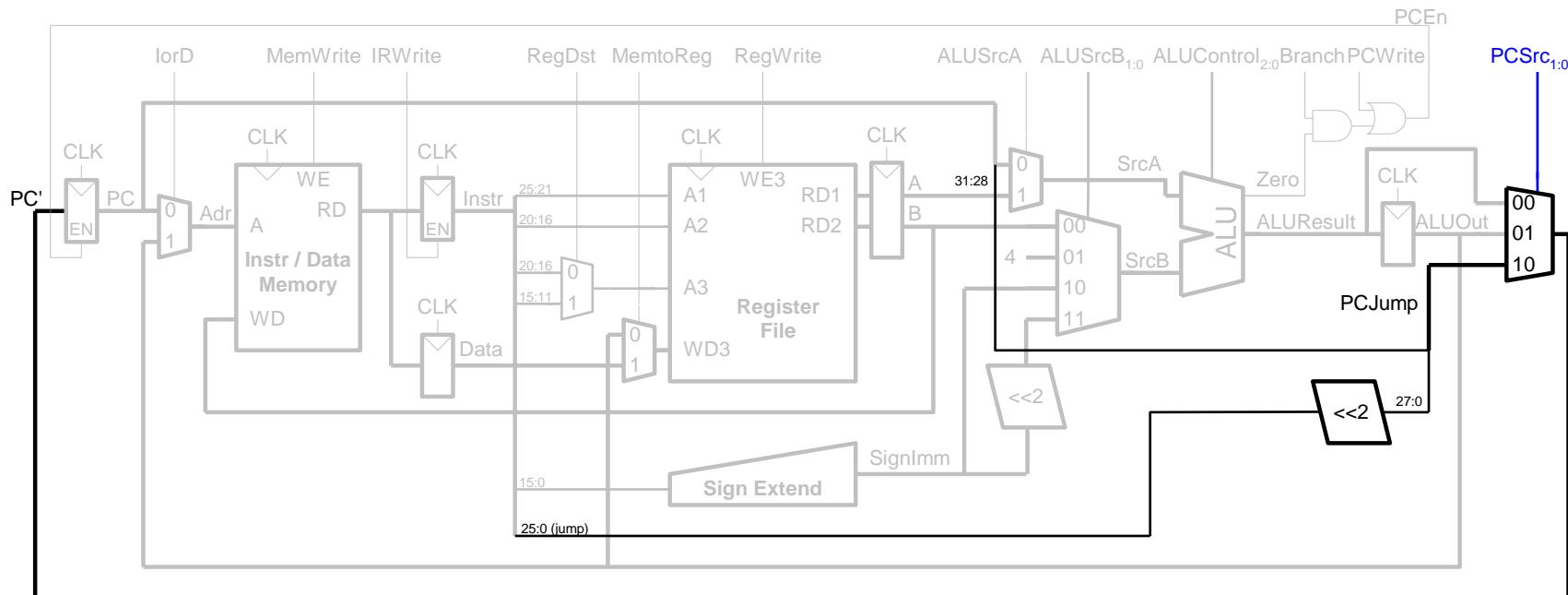
Main Controller FSM: addi



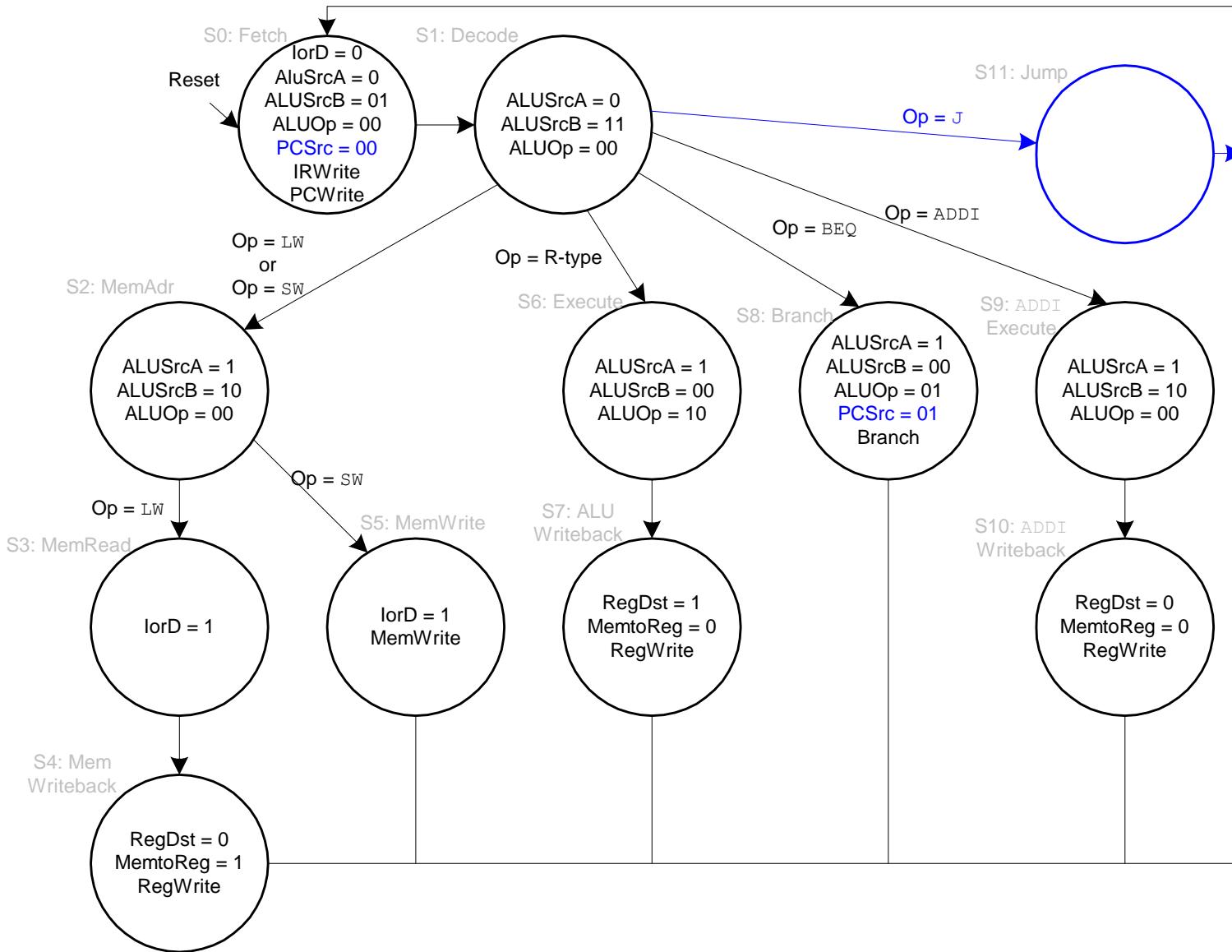
Main Controller FSM: addi



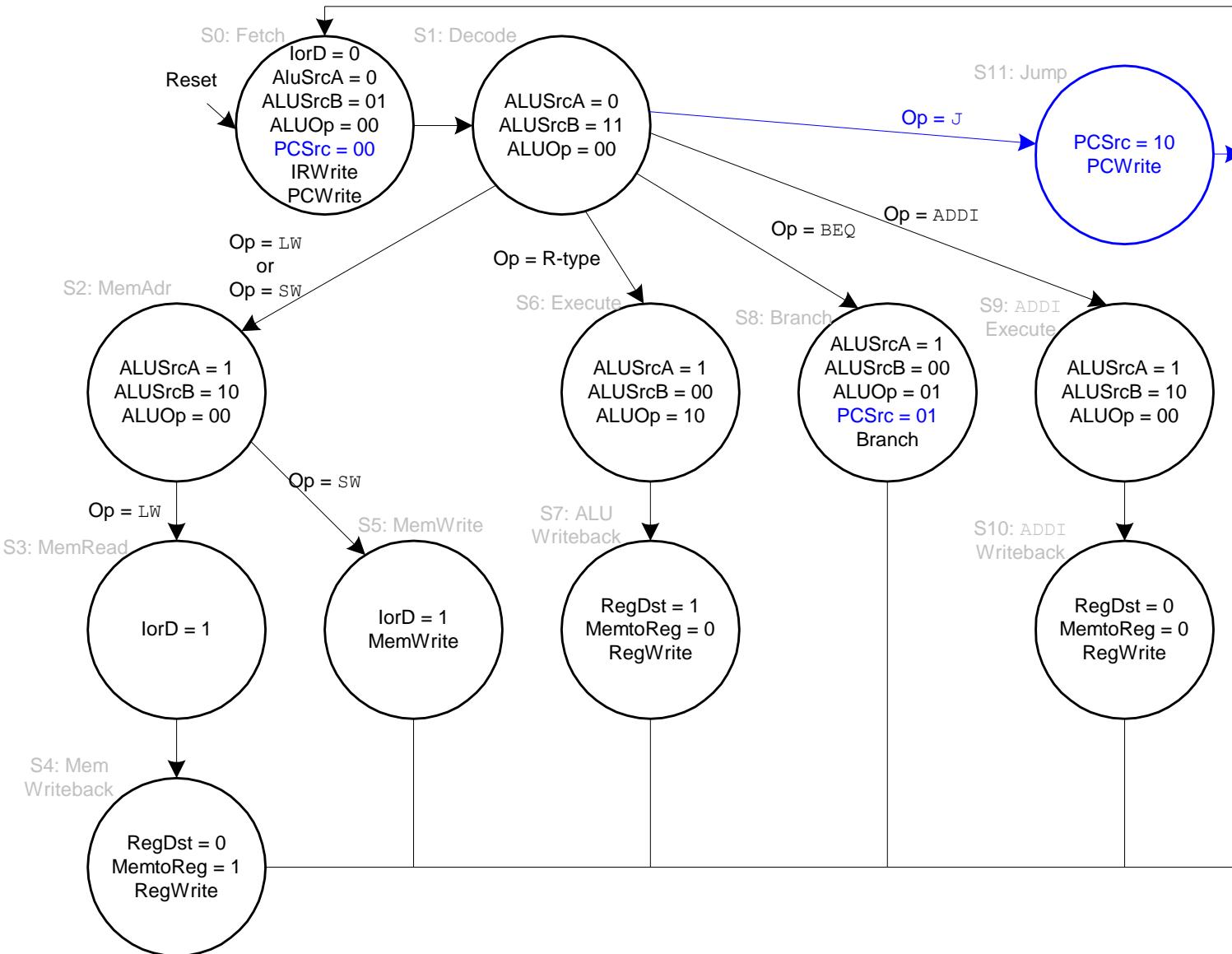
Extended Functionality: j



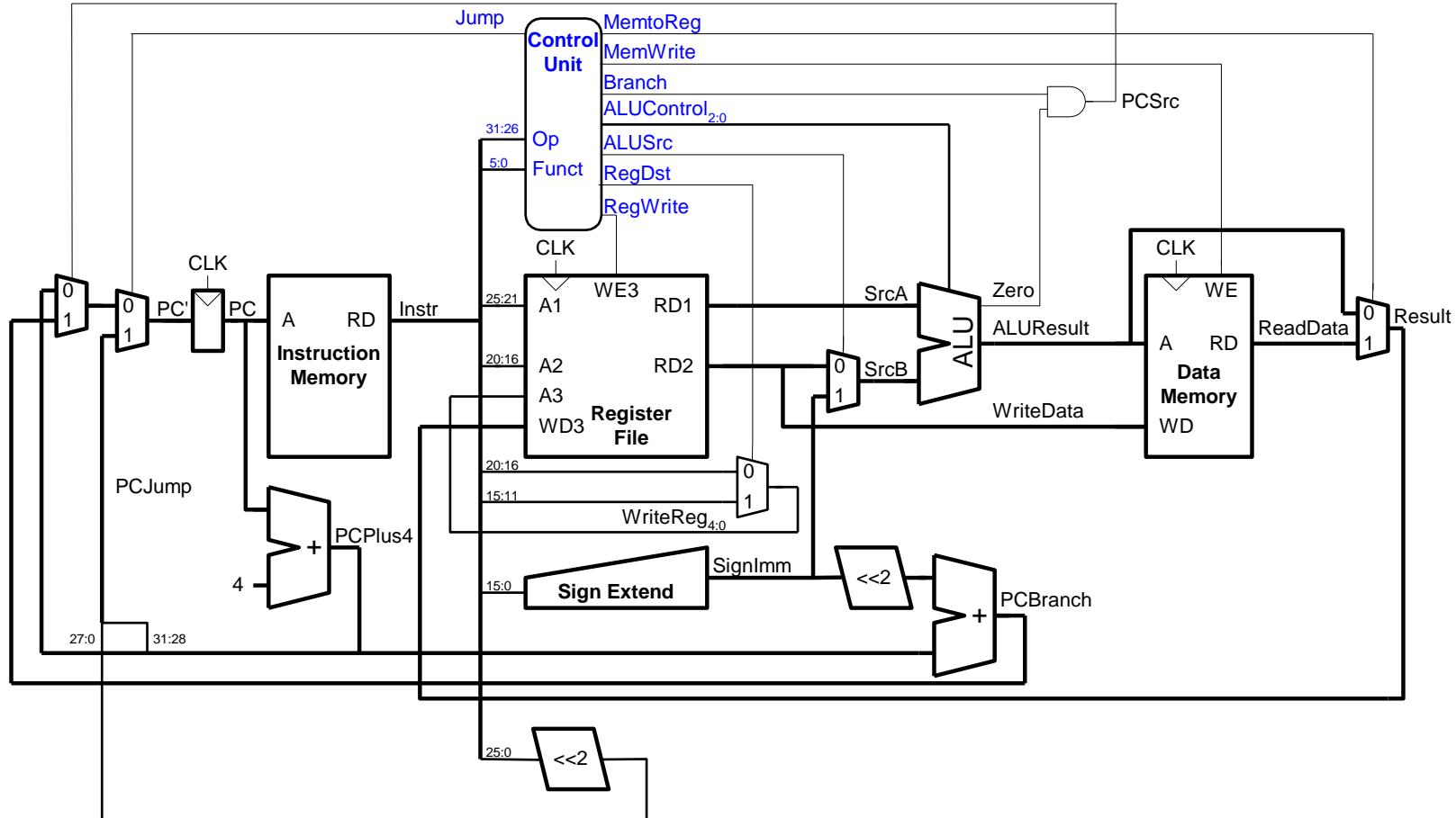
Control FSM: j



Control FSM: j

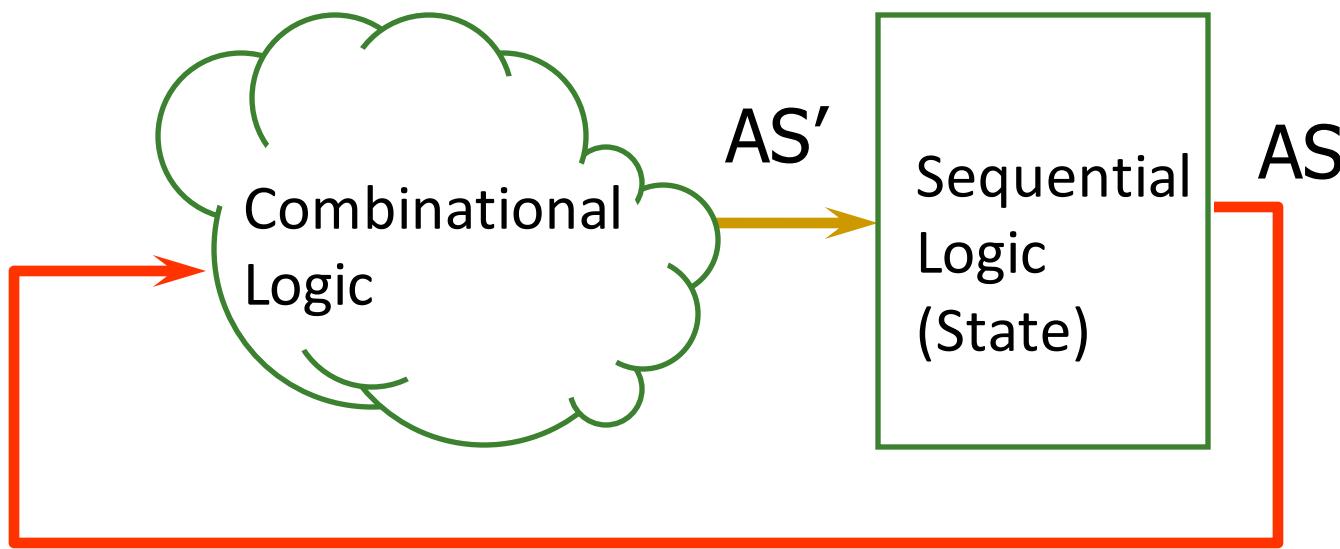


Review: Single-Cycle MIPS Processor

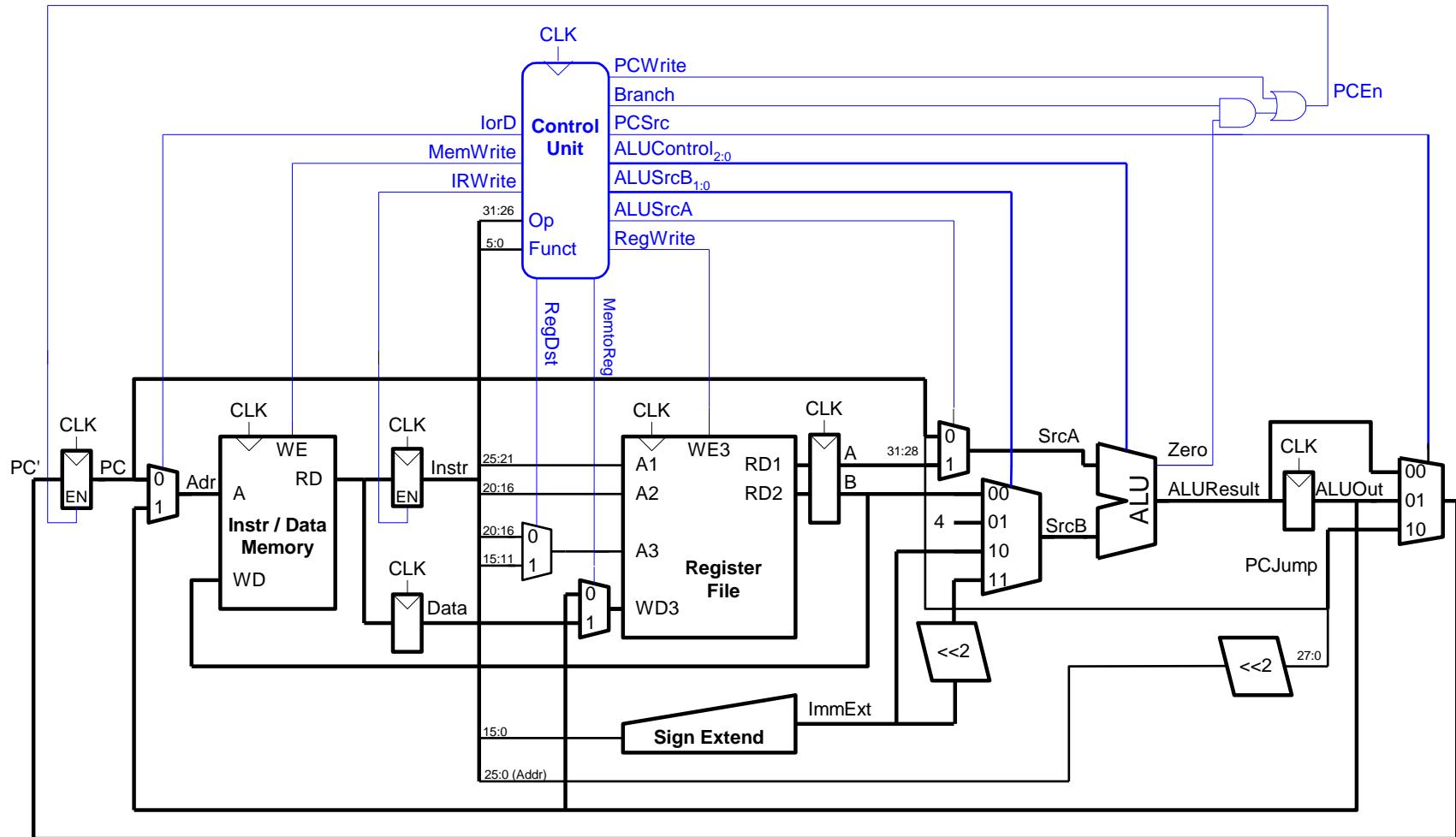


Review: Single-Cycle MIPS FSM

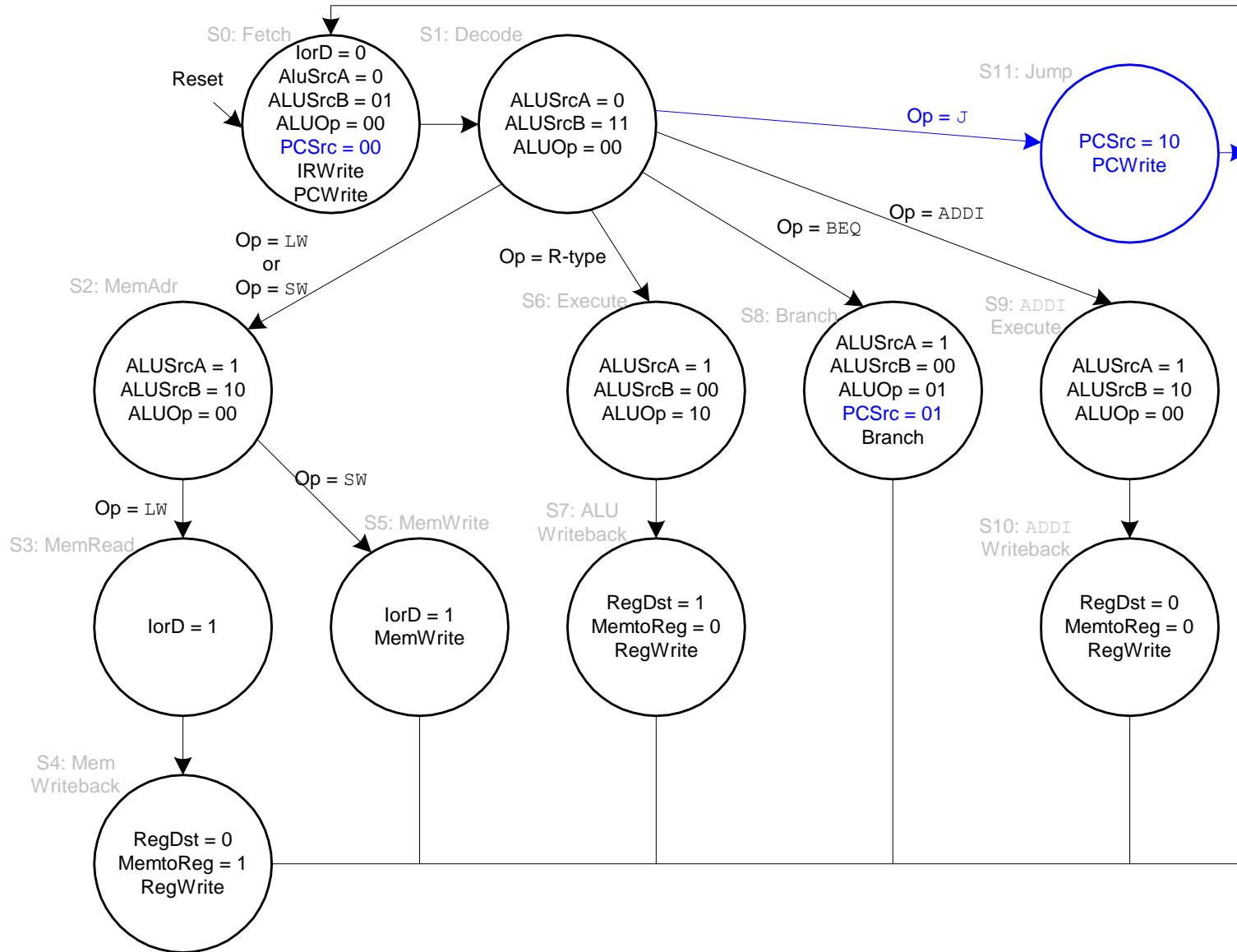
- Single-cycle machine



Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state

Another Example: **Microprogrammed Multi-Cycle Microarchitecture**

Recall: How Do We Implement This?

- Maurice Wilkes, "[The Best Way to Design an Automatic Calculating Machine](#)," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

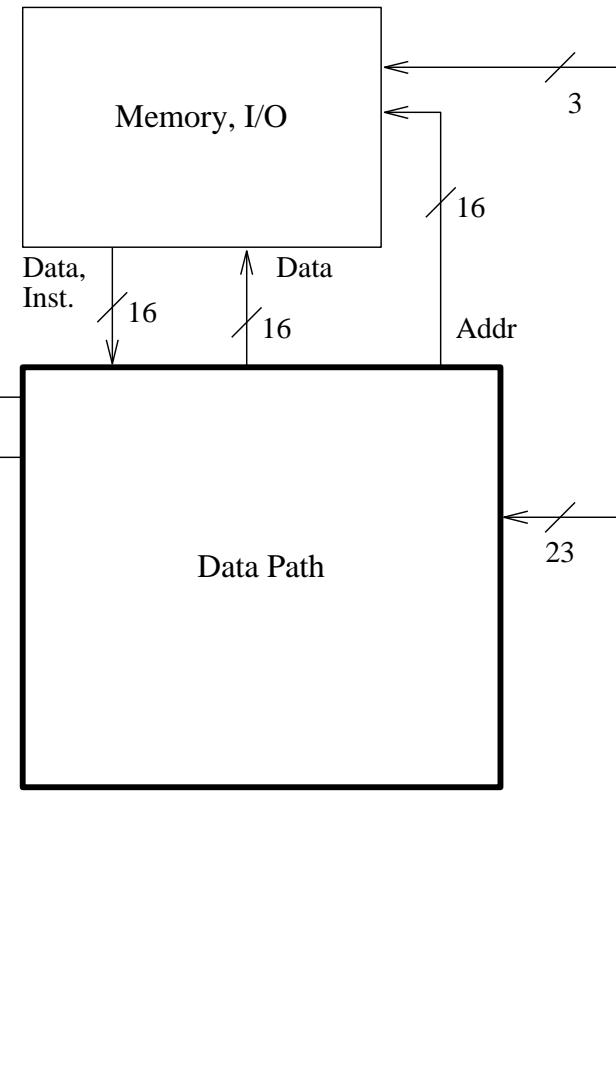
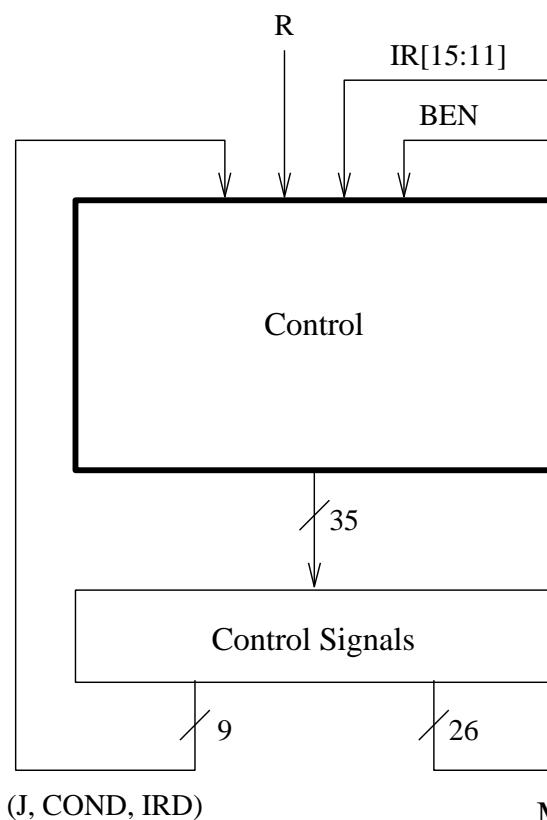
By M. V. Wilkes, M.A., Ph.D., F.R.A.S.



- An elegant implementation:
 - [The concept of microcoded/microprogrammed machines](#)

Example uProgrammed Control & Datapath

For your own study
P&P Revised Appendix C
On website
+ In Backup Slides



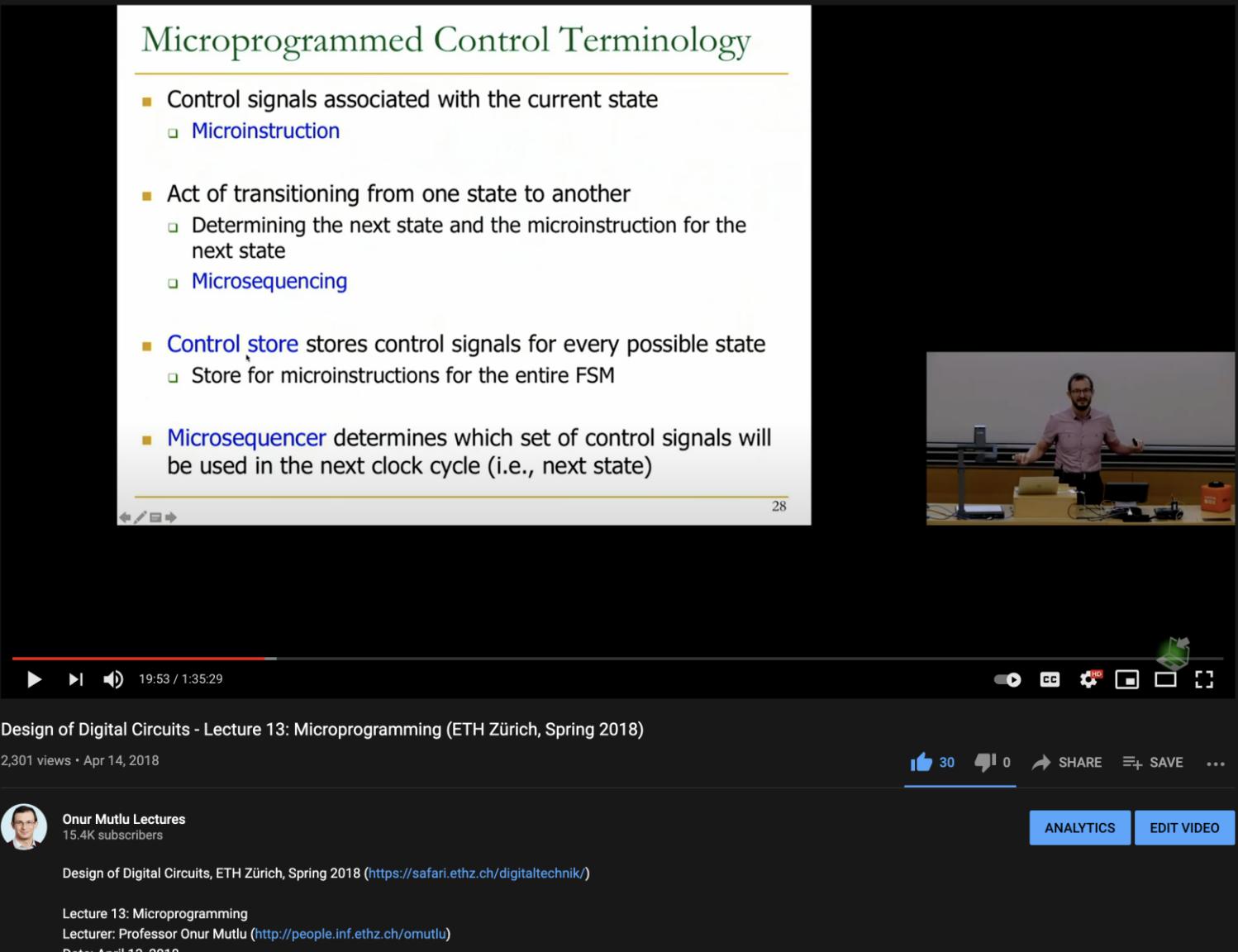
Microarchitecture of the LC-3b, major components

For More on Microprogrammed Designs

Micropogrammed Control Terminology

- Control signals associated with the current state
 - **Microinstruction**
- Act of transitioning from one state to another
 - Determining the next state and the microinstruction for the next state
 - **Microsequencing**
- **Control store** stores control signals for every possible state
 - Store for microinstructions for the entire FSM
- **Microsequencer** determines which set of control signals will be used in the next clock cycle (i.e., next state)

28



Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures
15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (<https://safari.ethz.ch/digitaltechnik/>)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (<http://people.inf.ethz.ch/omutlu>)
Date: April 13, 2018

Detailed Lectures on Microprogramming

- Design of Digital Circuits, Spring 2018, Lecture 13
 - Microprogramming (ETH Zürich, Spring 2018)
 - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7I&index=13
- Computer Architecture, Spring 2013, Lecture 7
 - Microprogramming (CMU, Spring 2013)
 - https://www.youtube.com/watch?v=_igvSI5h8cs&list=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6IJ&index=7

Digital Design & Computer Arch.

Lecture 12: Microarchitecture Fundamentals II

Prof. Onur Mutlu

ETH Zürich
Spring 2021
15 April 2021

We did not cover the following slides.
They are for your benefit.

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;

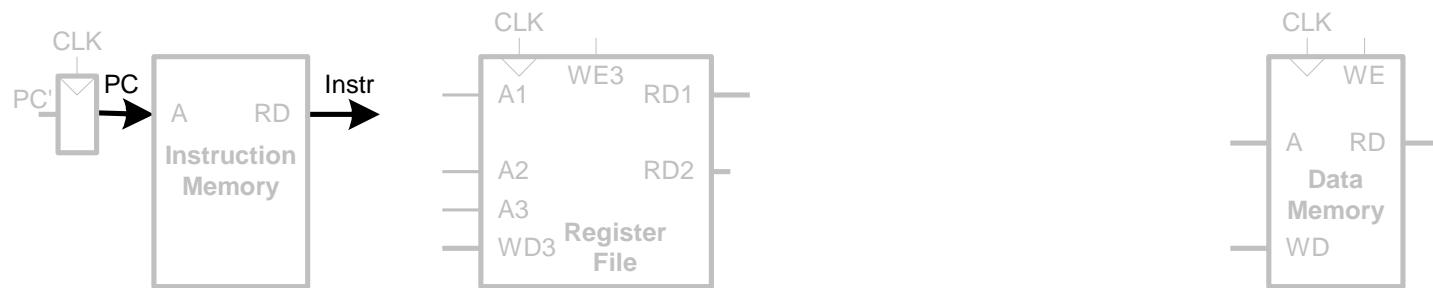
reg [31:0] M_arr [0:65535];           // Array for Memory

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

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

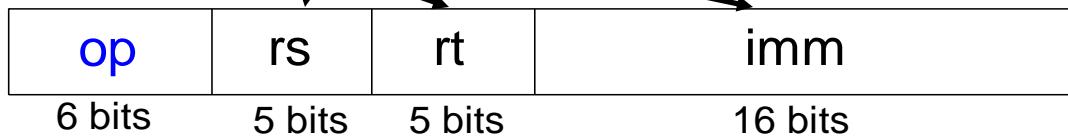
Single-Cycle Datapath: lw fetch

■ STEP 1: Fetch instruction



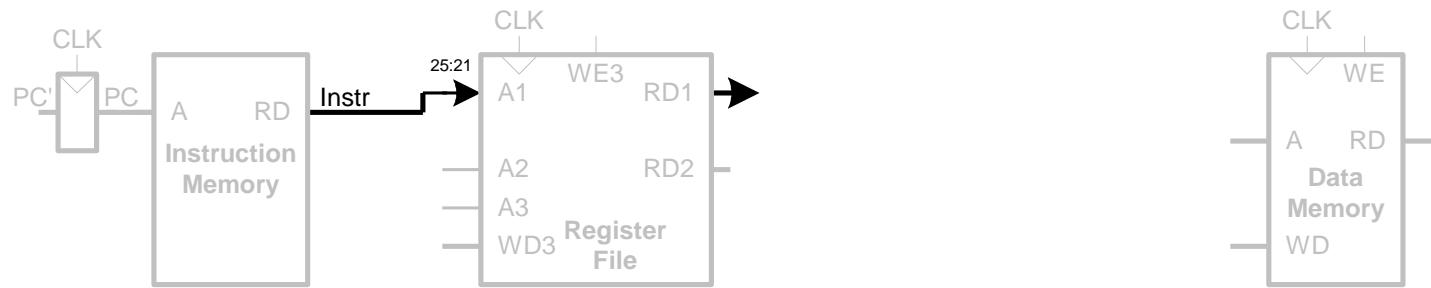
`lw $s3, 1($0) # read memory word 1 into $s3`

I-Type



Single-Cycle Datapath: lw register read

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



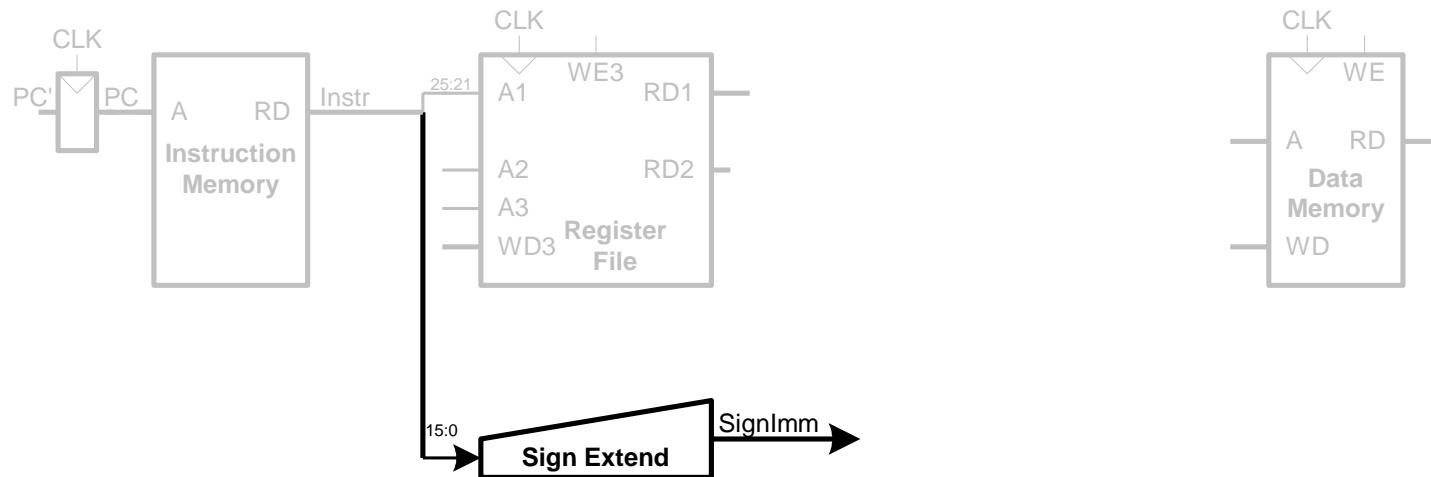
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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**



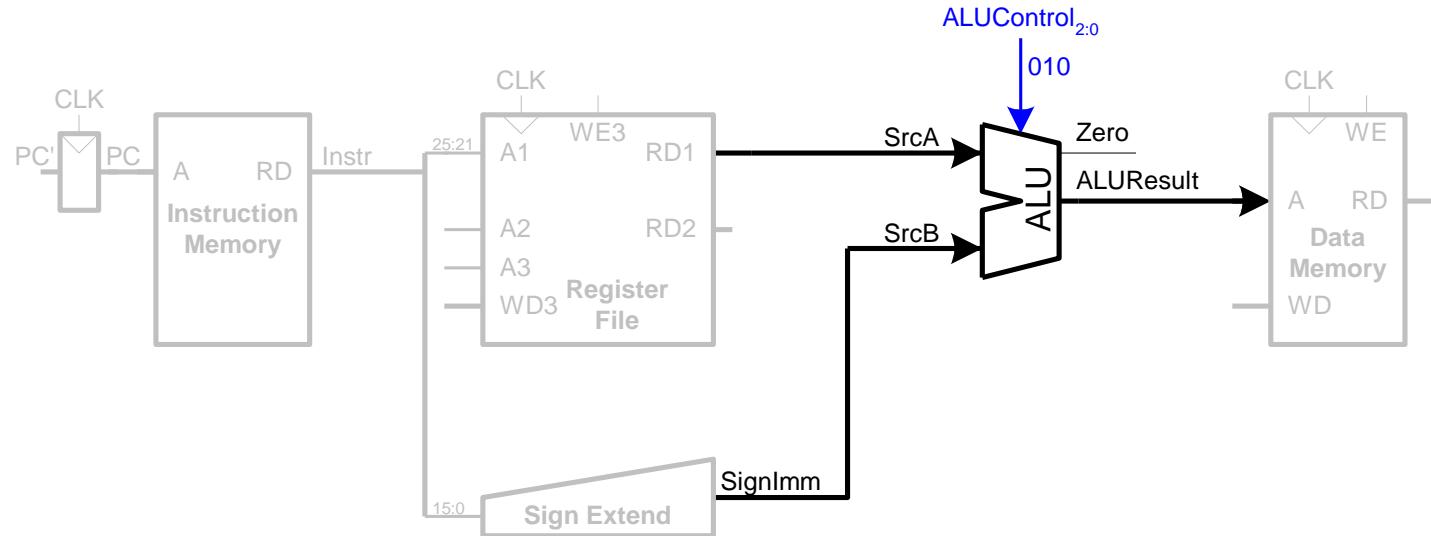
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



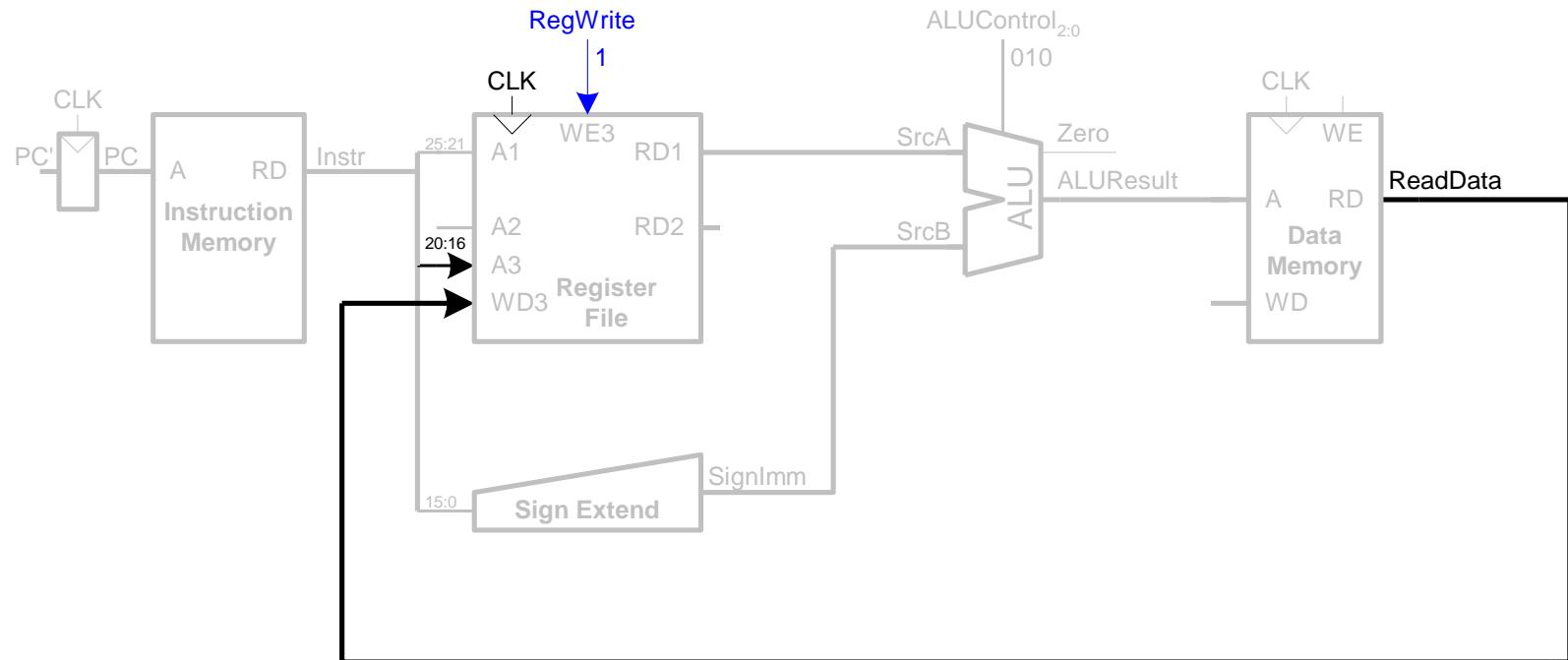
```
lw $s3, 1($0) # read memory word 1 into $s3
```

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



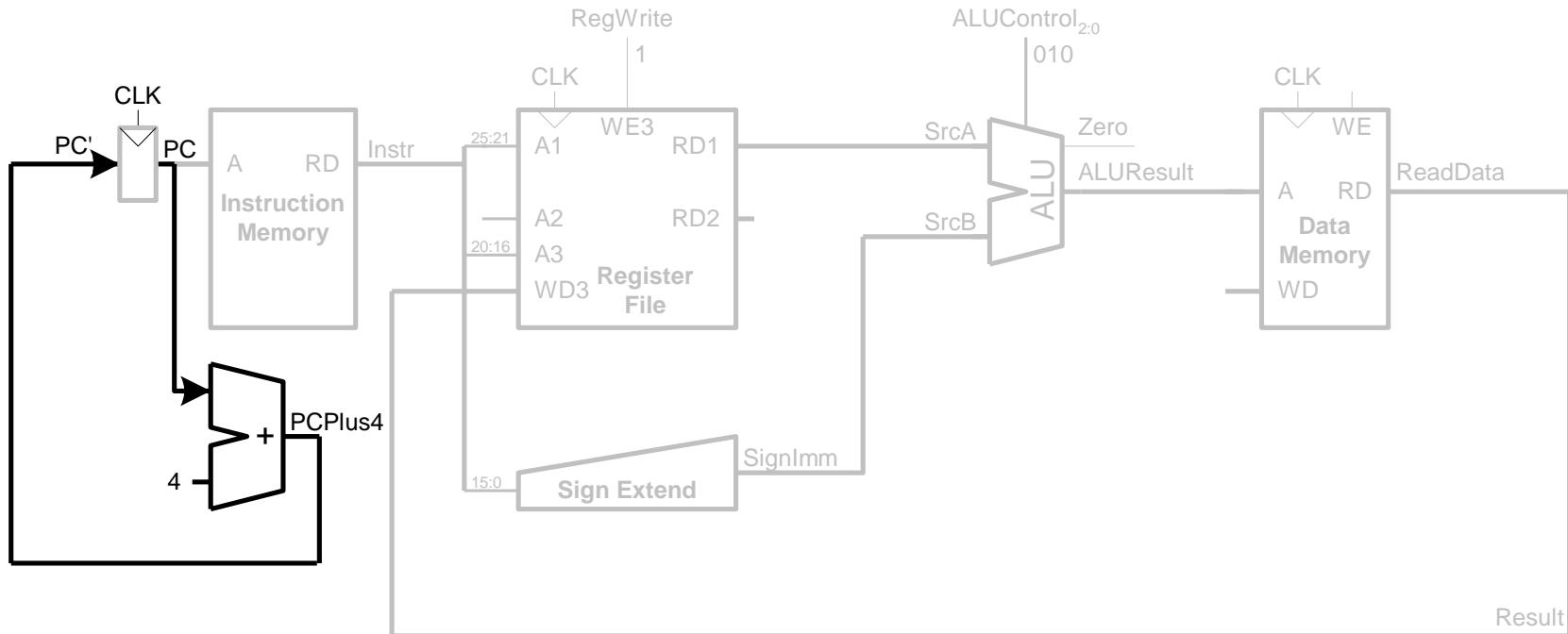
lw \$s3, 1(\$0) # read memory word 1 into \$s3

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



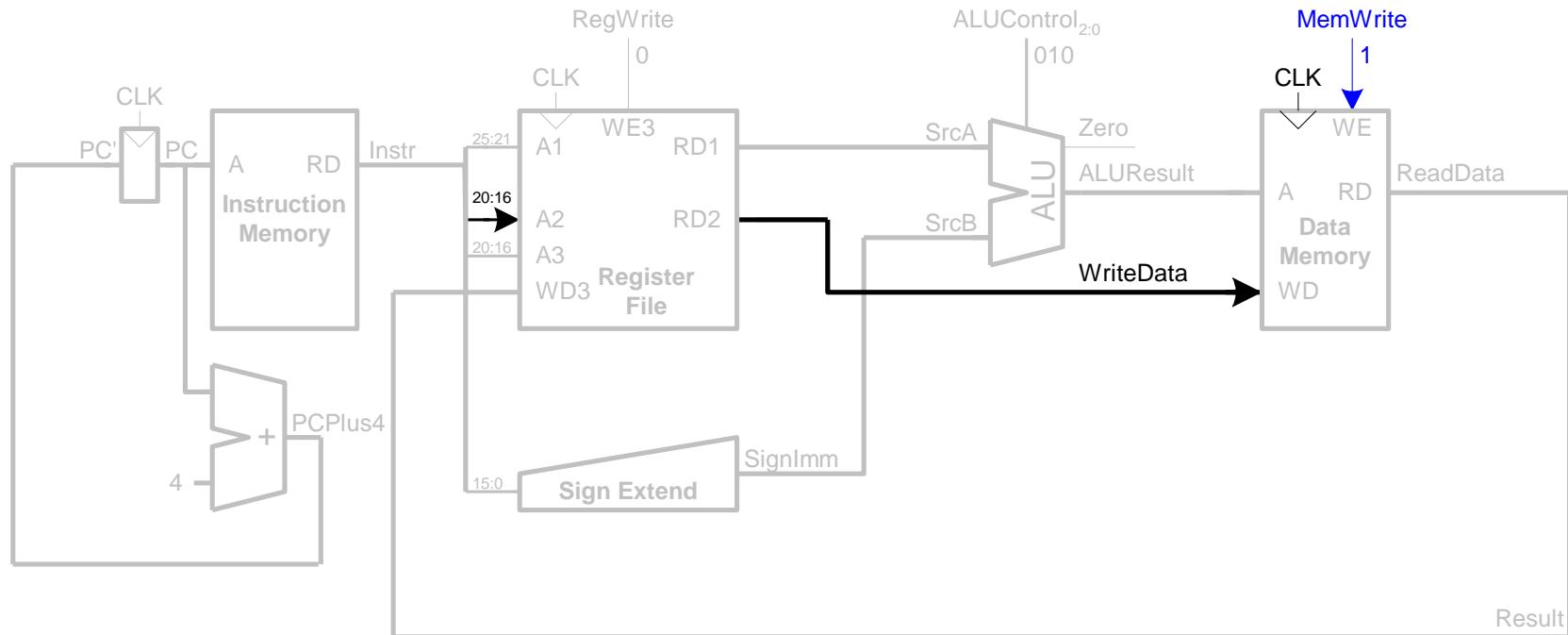
```
lw $s3, 1($0) # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: sw

■ Write data in rt to memory



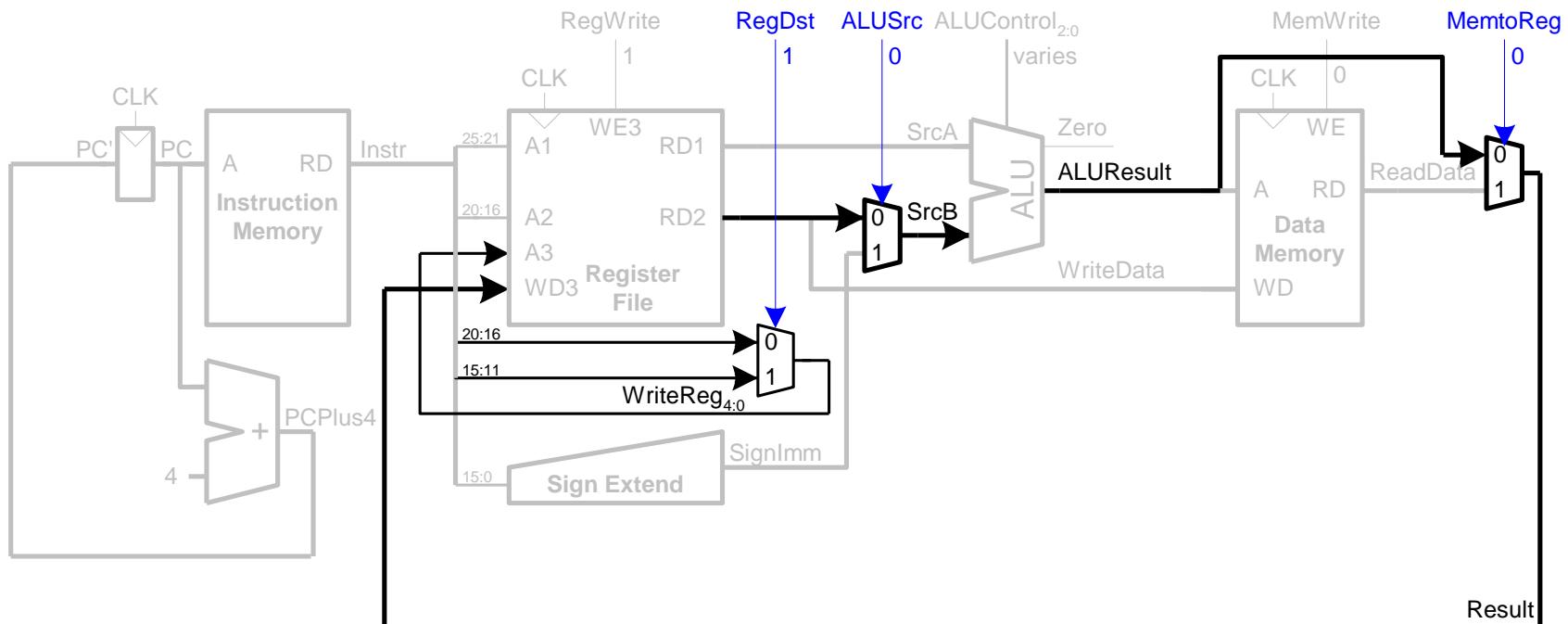
```
sw $t7, 44($0) # write t7 into memory address 44
```

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

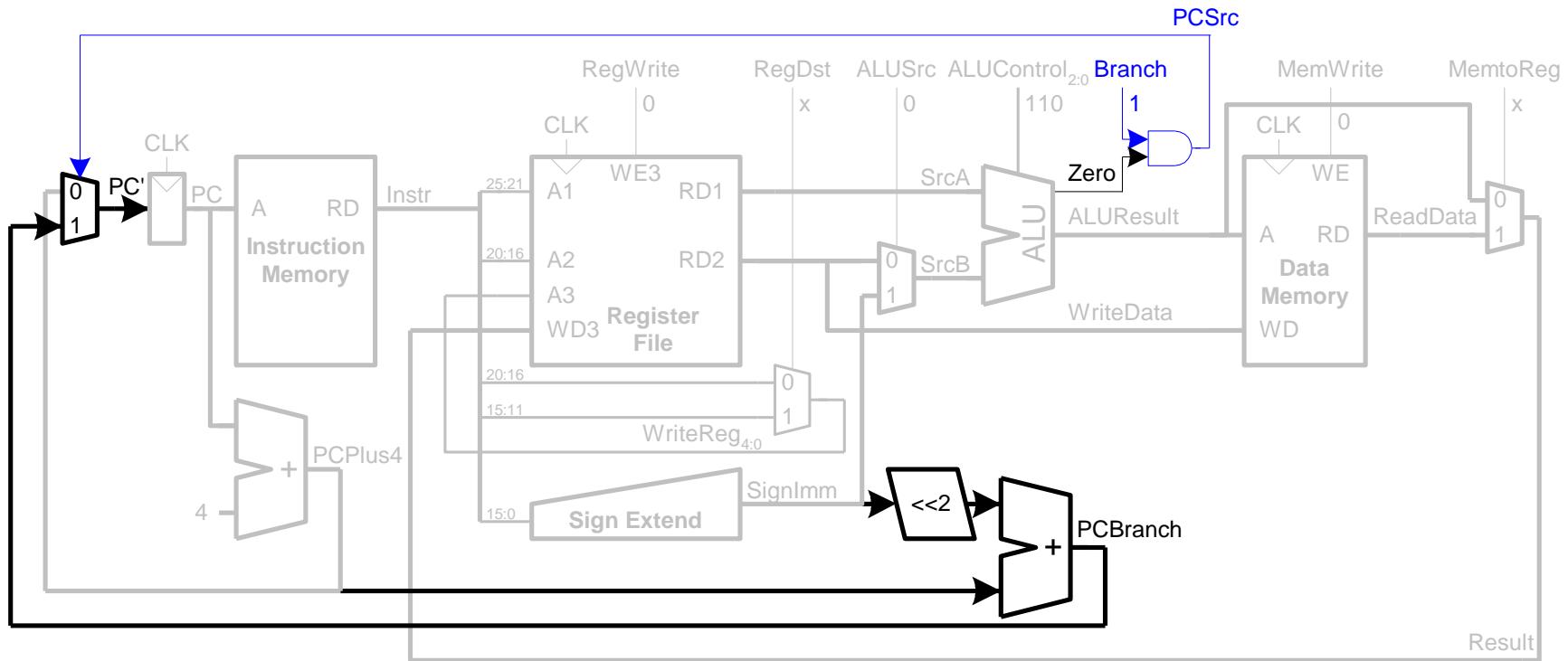


add t, b, c # $t = b + c$

R-Type

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

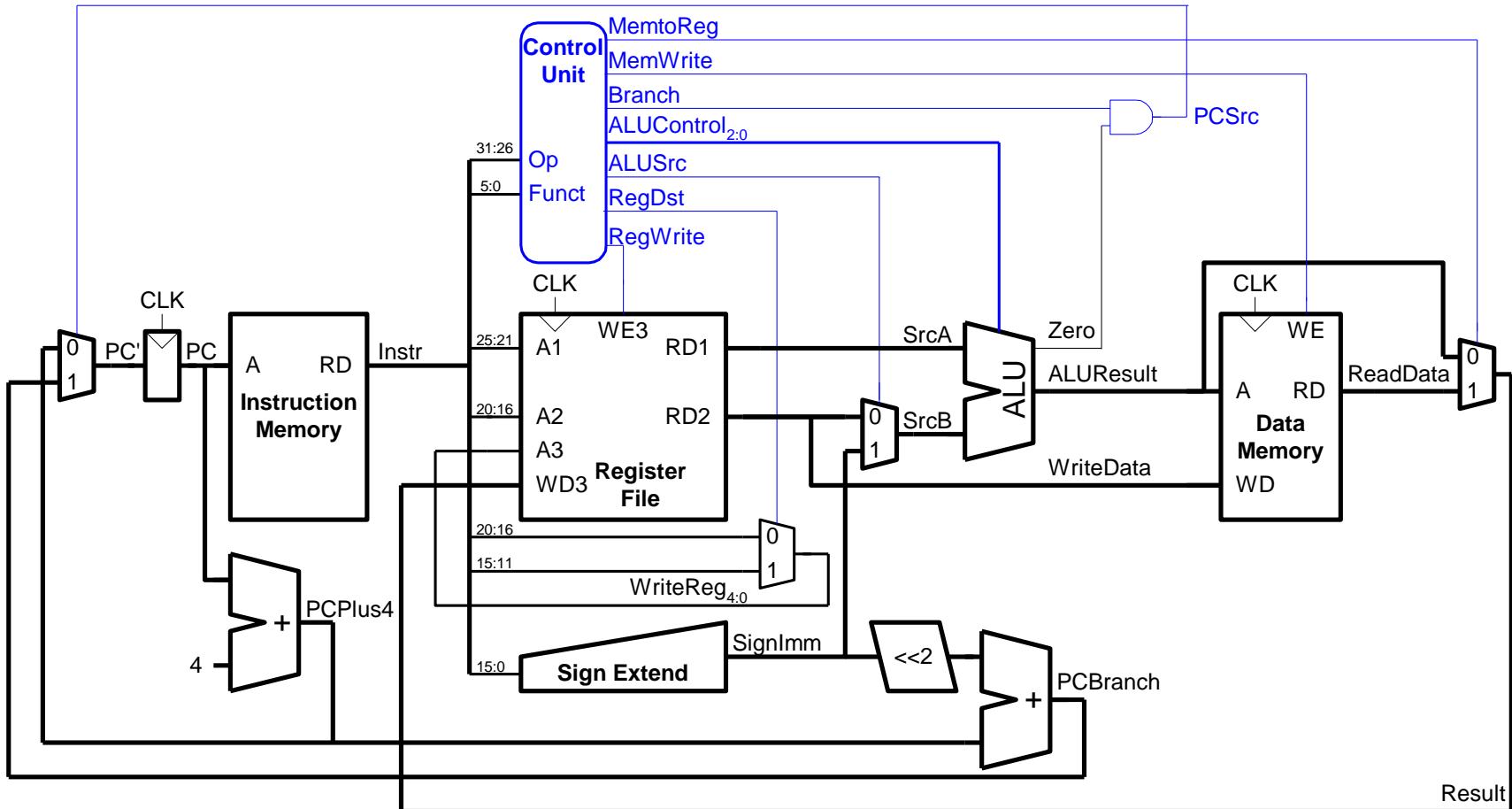
Single-Cycle Datapath: beq



`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

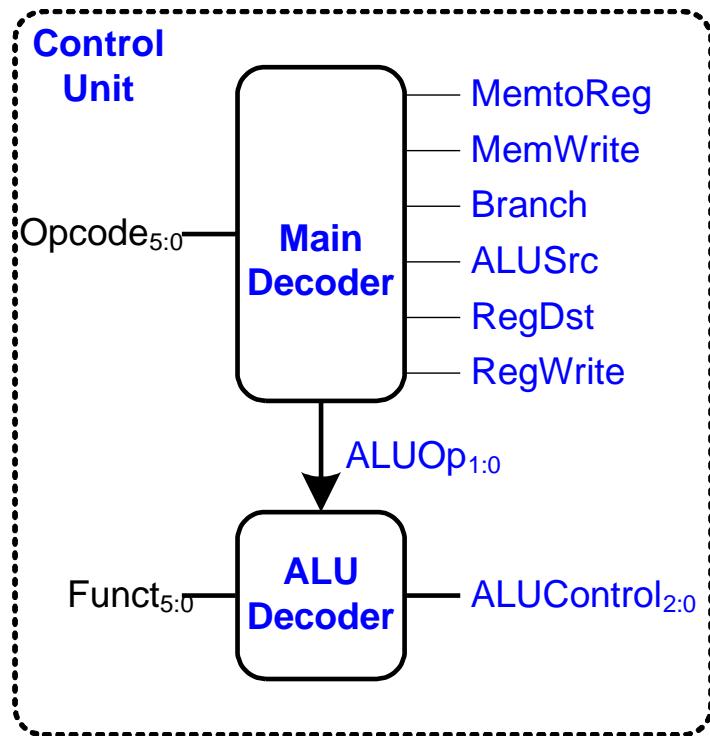


Our MIPS Datapath has Several Options

- **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*)

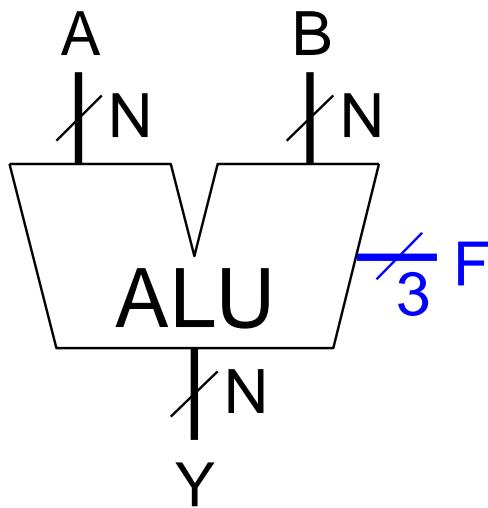
All these options are our control signals

Control Unit



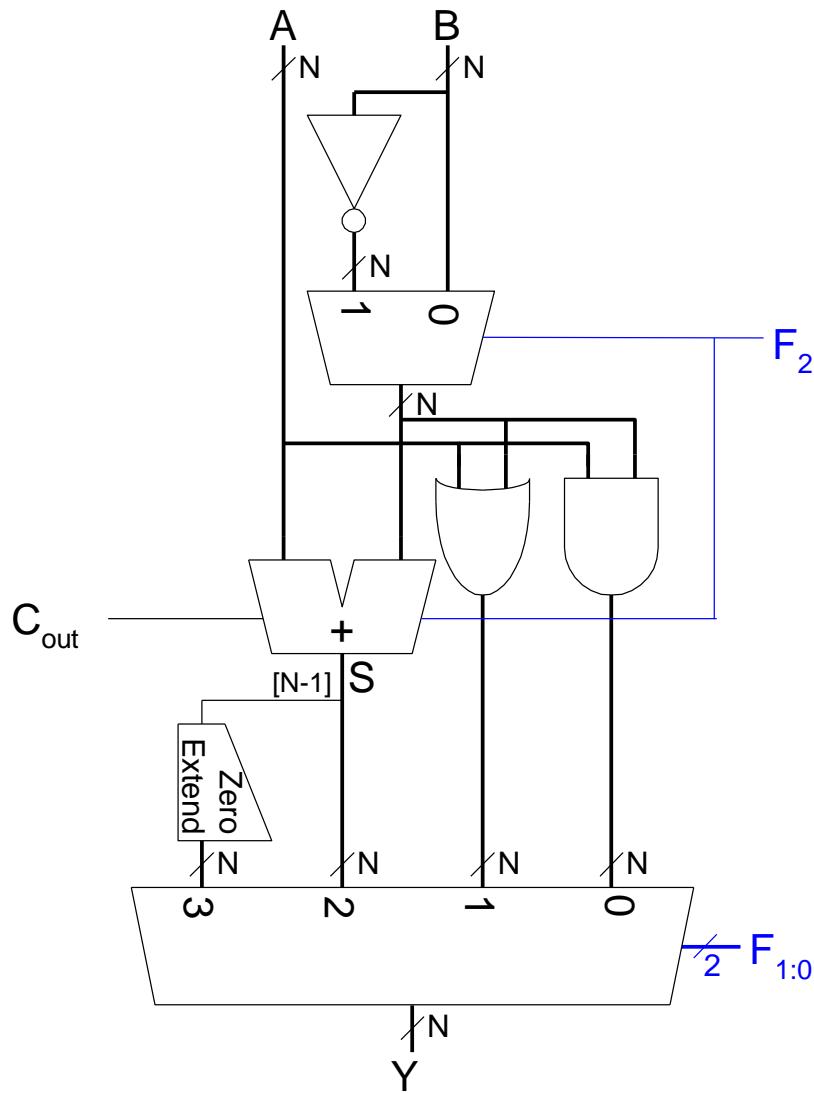
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



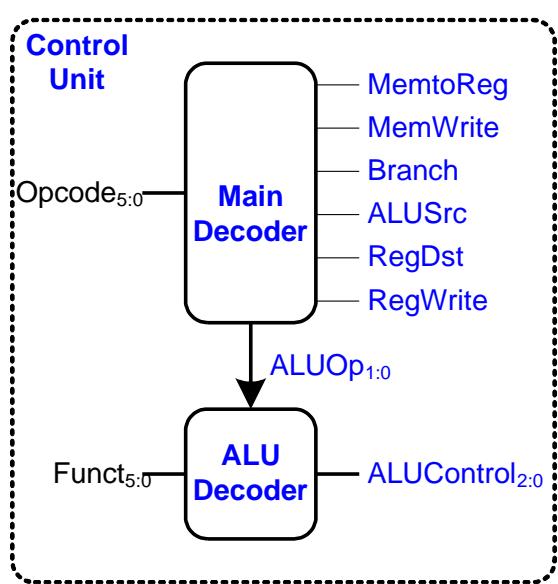
$F_{2:0}$	Function
000	A & B
001	A B
010	A + B
011	not used
100	A & \sim B
101	A \sim B
110	A - B
111	SLT

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

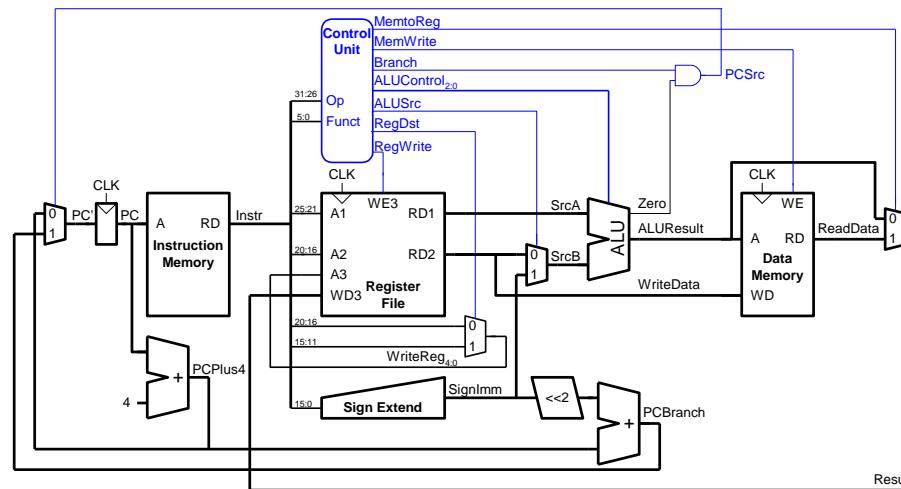
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

■ New Control Signal

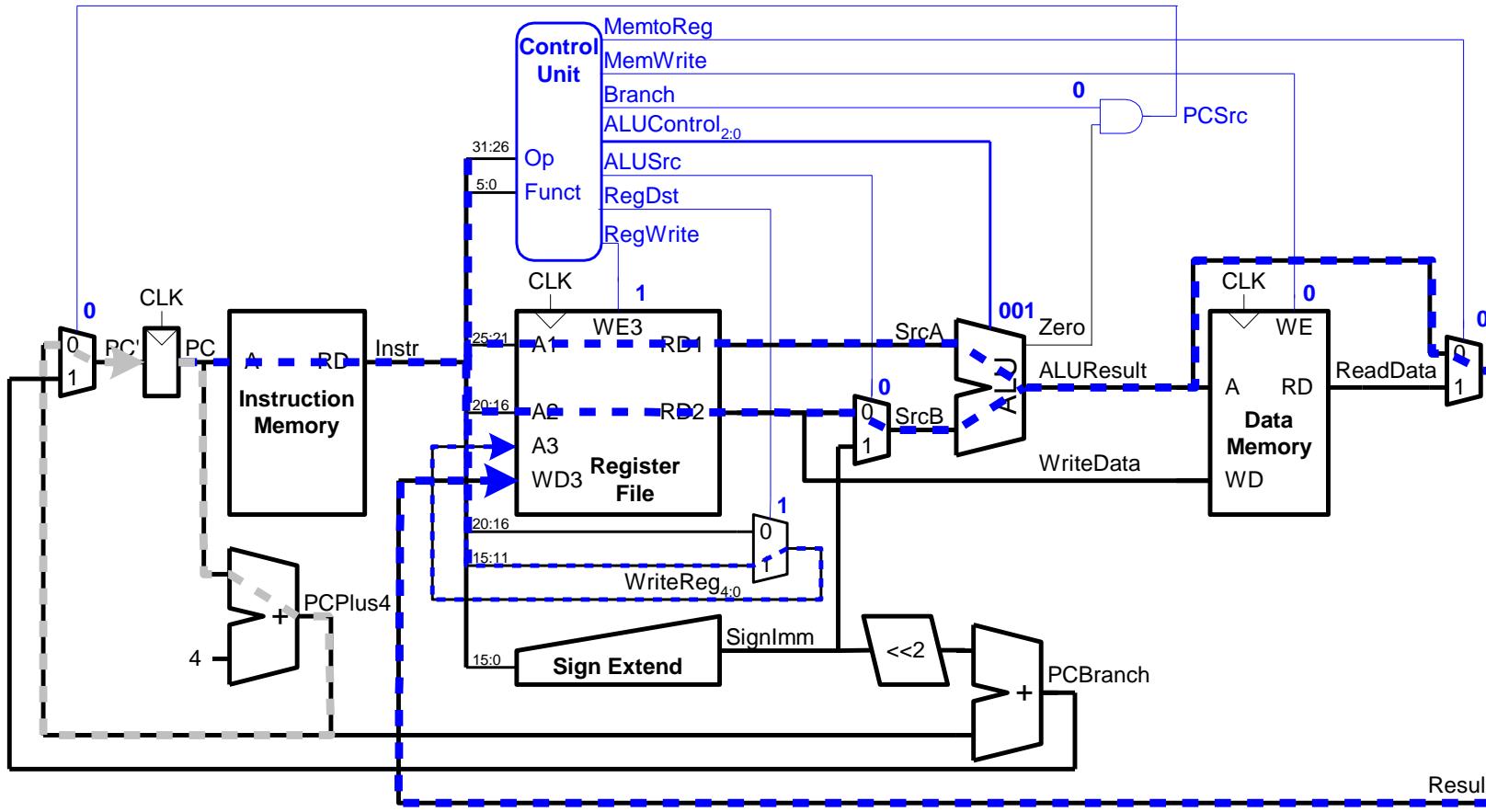
- *Branch*: Are we jumping or not ?

Control Unit: Main Decoder

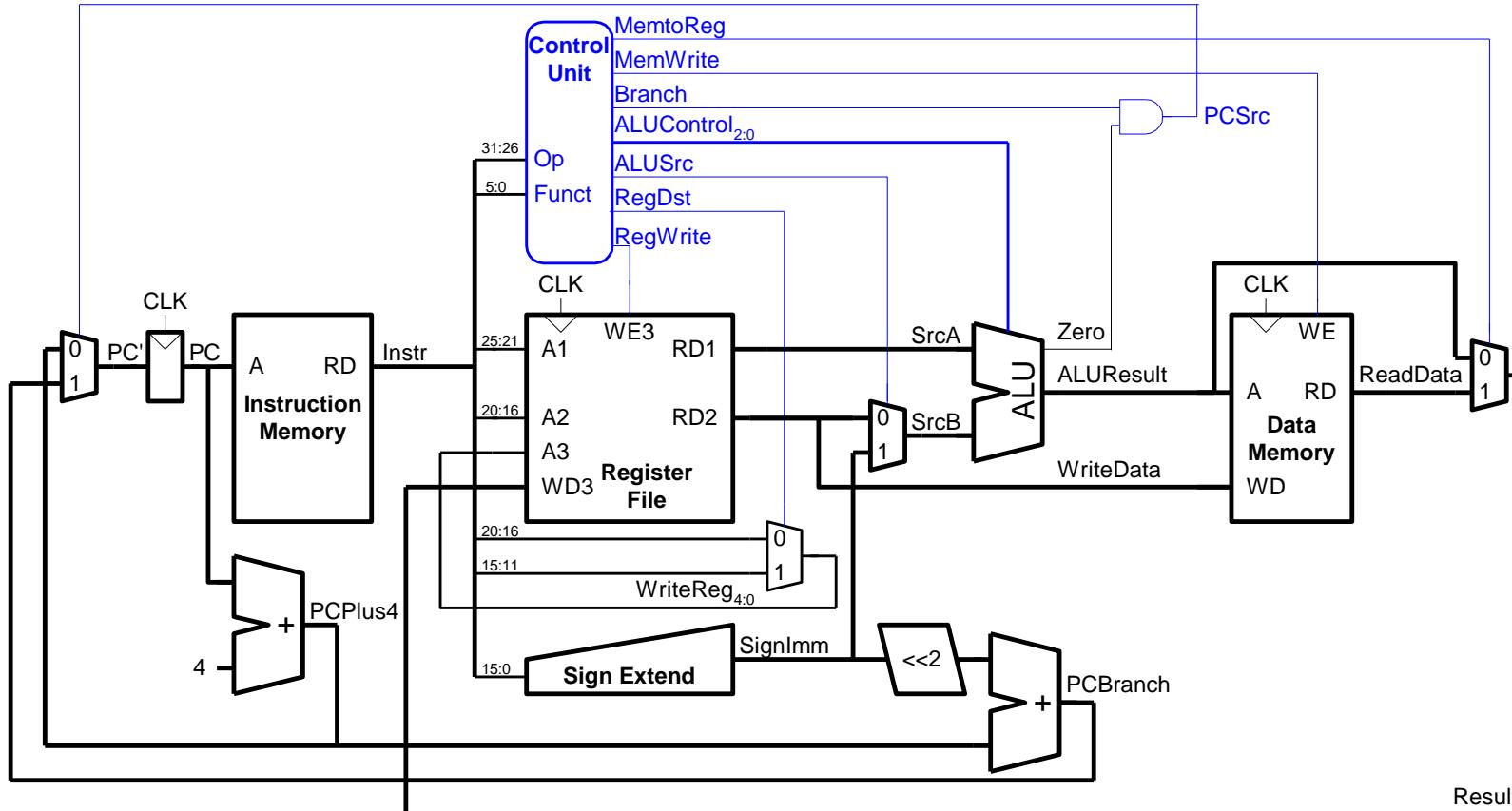
Instruction	Op _{5:0}	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp _{1:0}
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

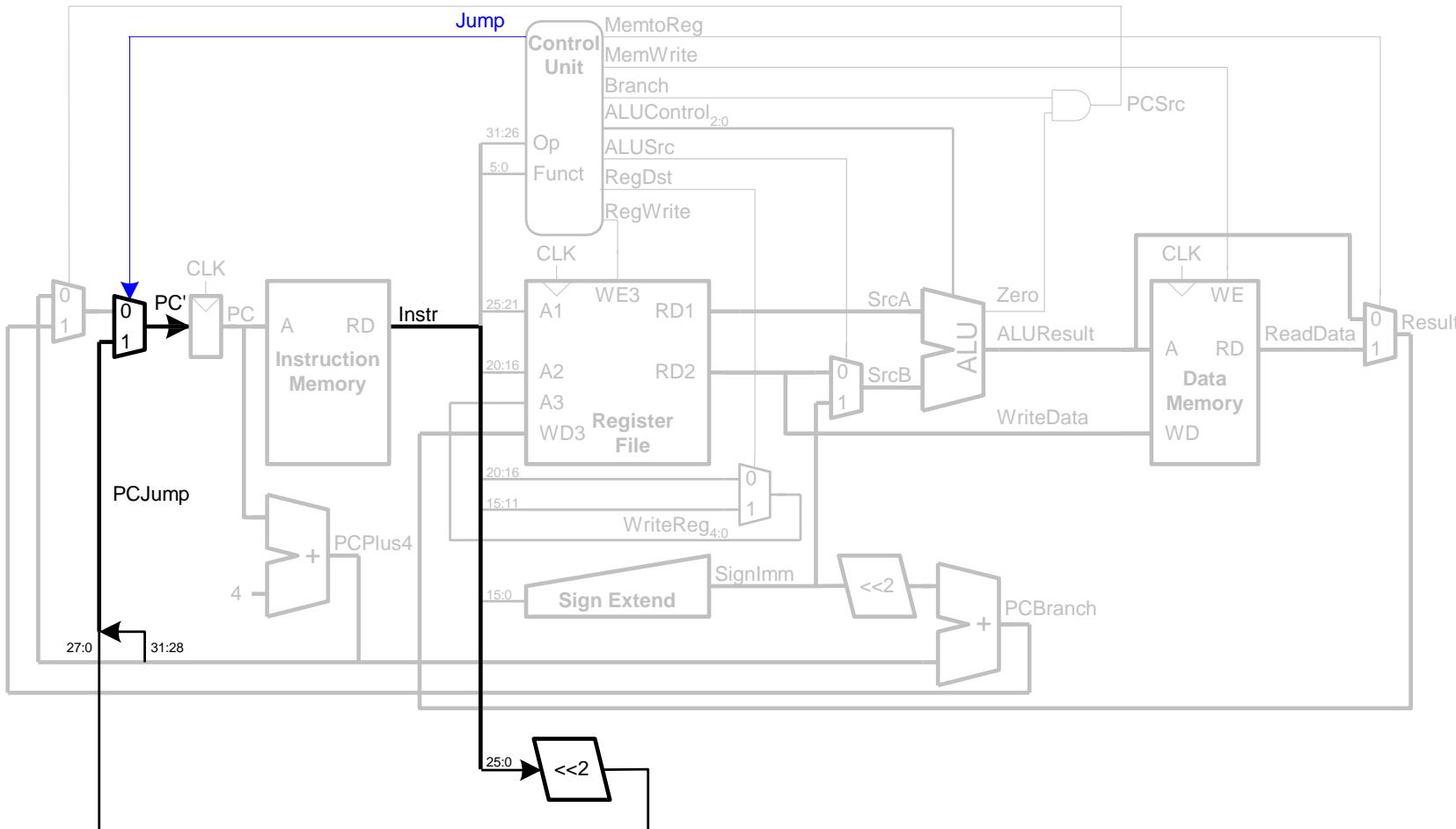


- No change to datapath

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

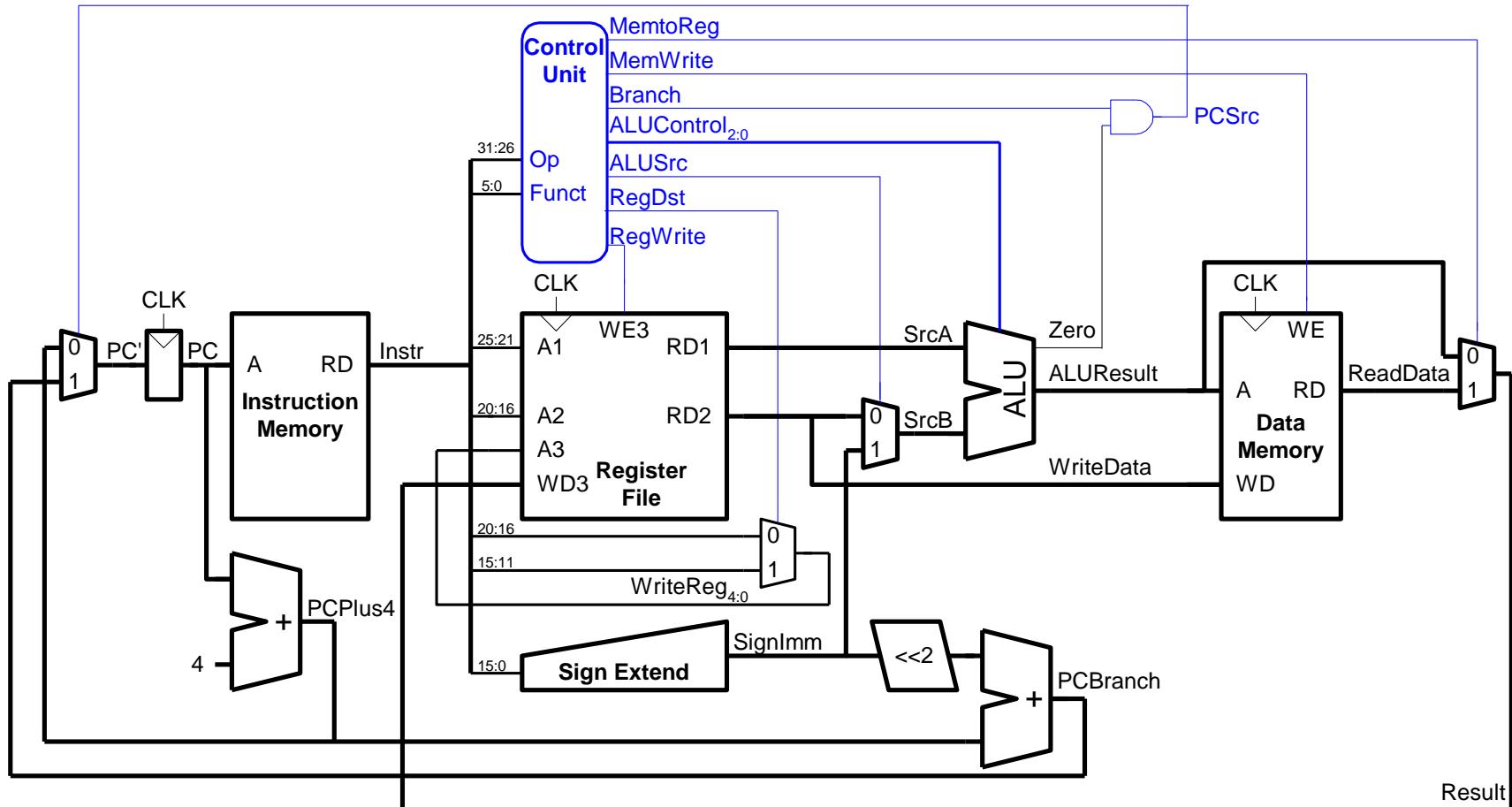
Extended Functionality: j



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

How can I Make the Program Run Faster?

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

How can I Make the Program Run Faster?

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

- **Reduce the number of instructions**

- Make instructions that 'do' more (CISC – complex instruction sets)
 - Use better compilers

How can I Make the Program Run Faster?

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

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

How can I Make the Program Run Faster?

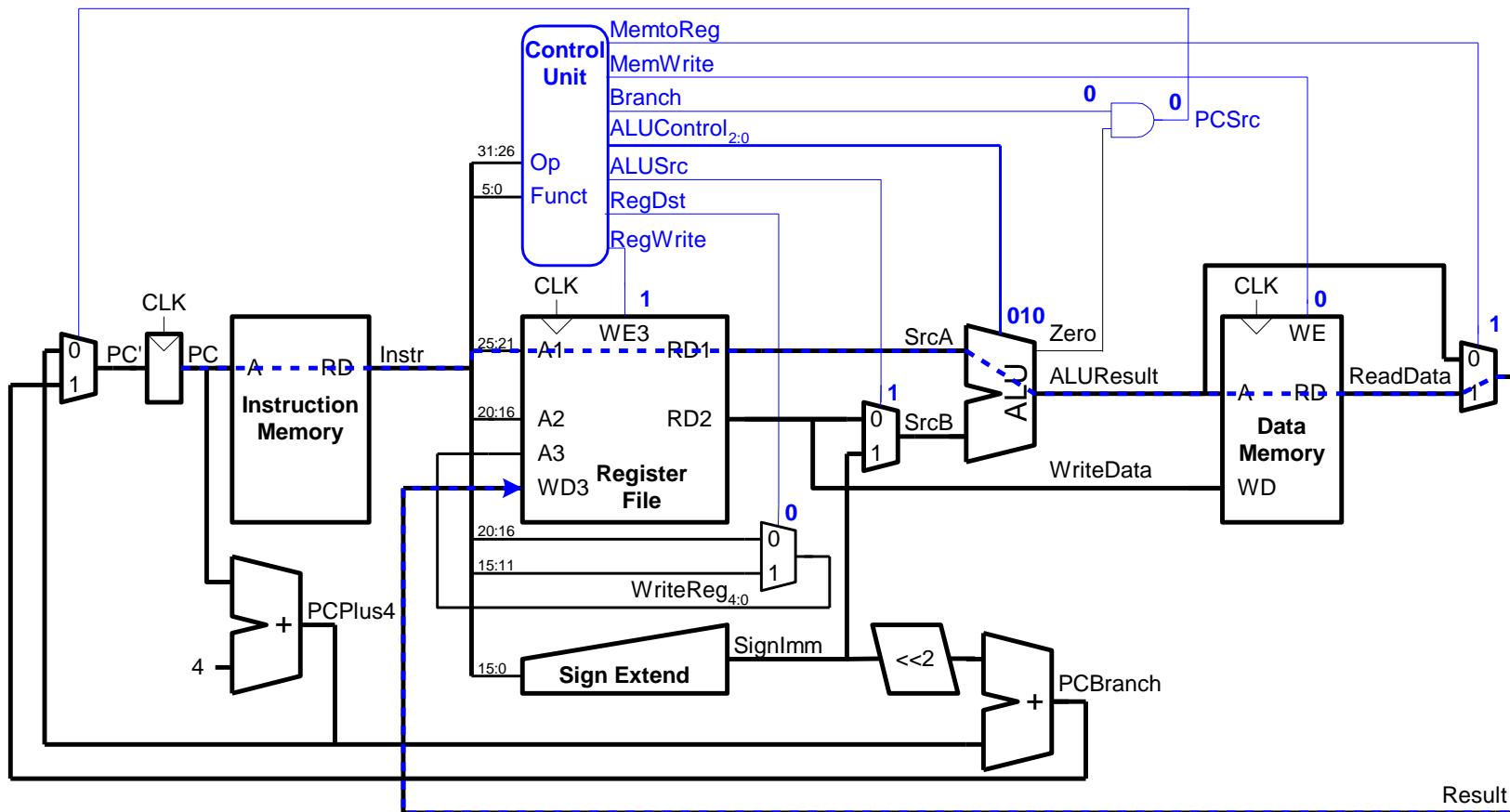
$$N \times CPI \times (1/f)$$

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

Performance Analysis of Single-Cycle vs. Multi-Cycle Designs

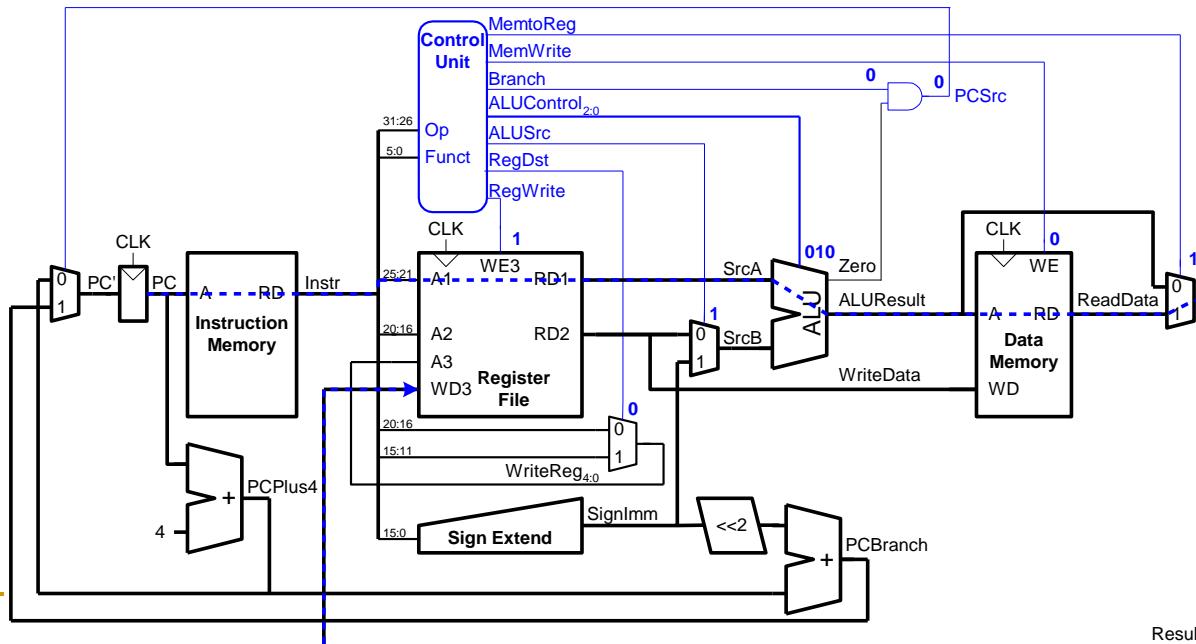
Single-Cycle Performance

- 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:

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

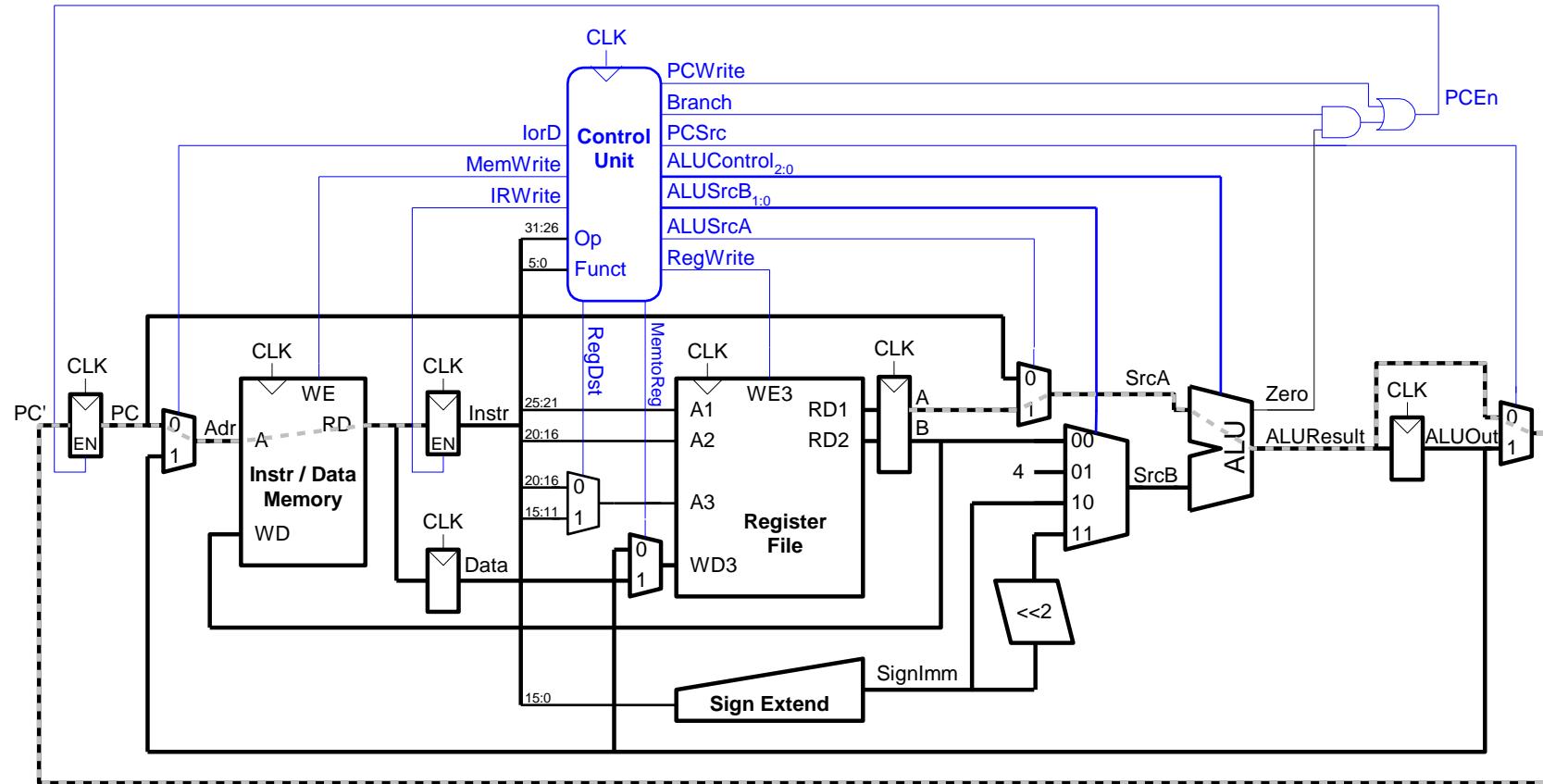
Multi-Cycle Performance: CPI

- Instructions take different number of cycles:
 - 3 cycles: beq, j
 - 4 cycles: R-Type, sw, addi
 - 5 cycles: lw **Realistic?**
- CPI is weighted average, e.g. SPECINT2000 benchmark:
 - 25% loads
 - 10% stores
 - 11% branches
 - 2% jumps
 - 52% R-type
- *Average CPI* = $(0.11 + 0.02) 3 + (0.52 + 0.10) 4 + (0.25) 5$
= 4.12

Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

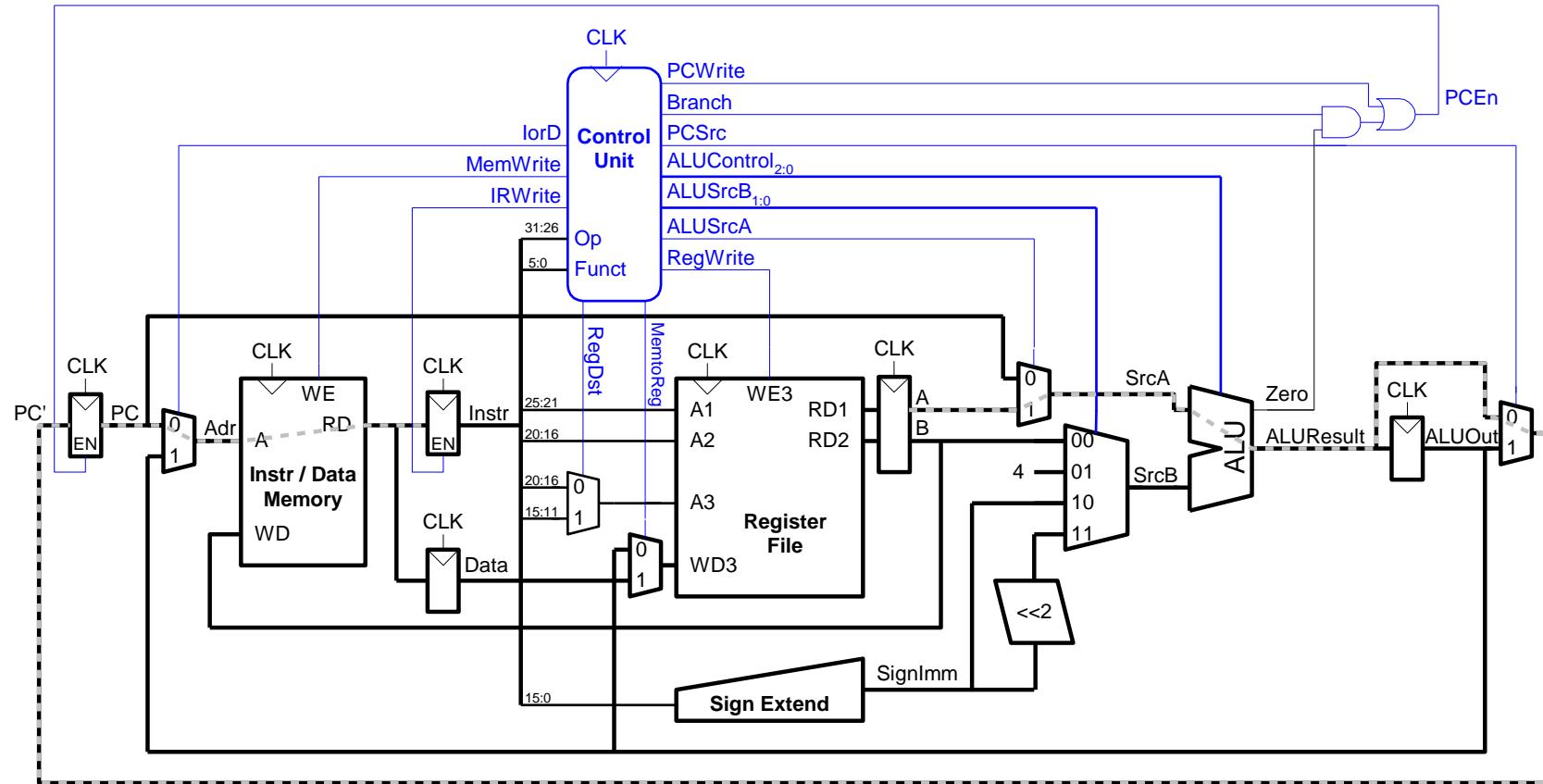
$$T_c =$$



Multi-Cycle Performance: Cycle Time

- Multi-cycle critical path:

$$T_c = t_{pcq} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup}$$



Multi-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

$$T_c =$$

Multi-Cycle Performance Example

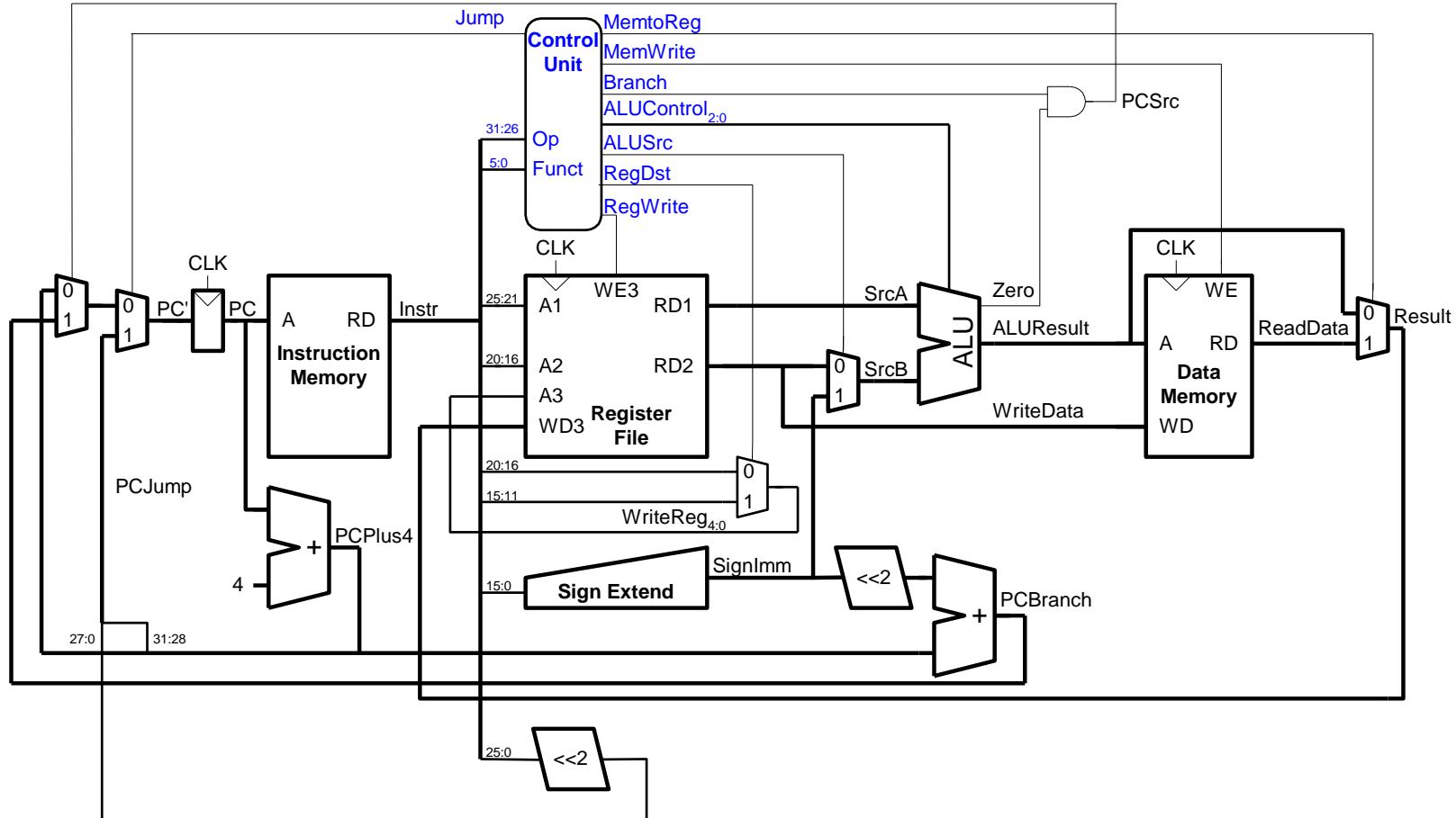
Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	30
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20

$$\begin{aligned} T_c &= t_{pcq_PC} + t_{mux} + \max(t_{ALU} + t_{mux}, t_{mem}) + t_{setup} \\ &= [30 + 25 + 250 + 20] \text{ ps} \\ &= 325 \text{ ps} \end{aligned}$$

Multi-Cycle Performance Example

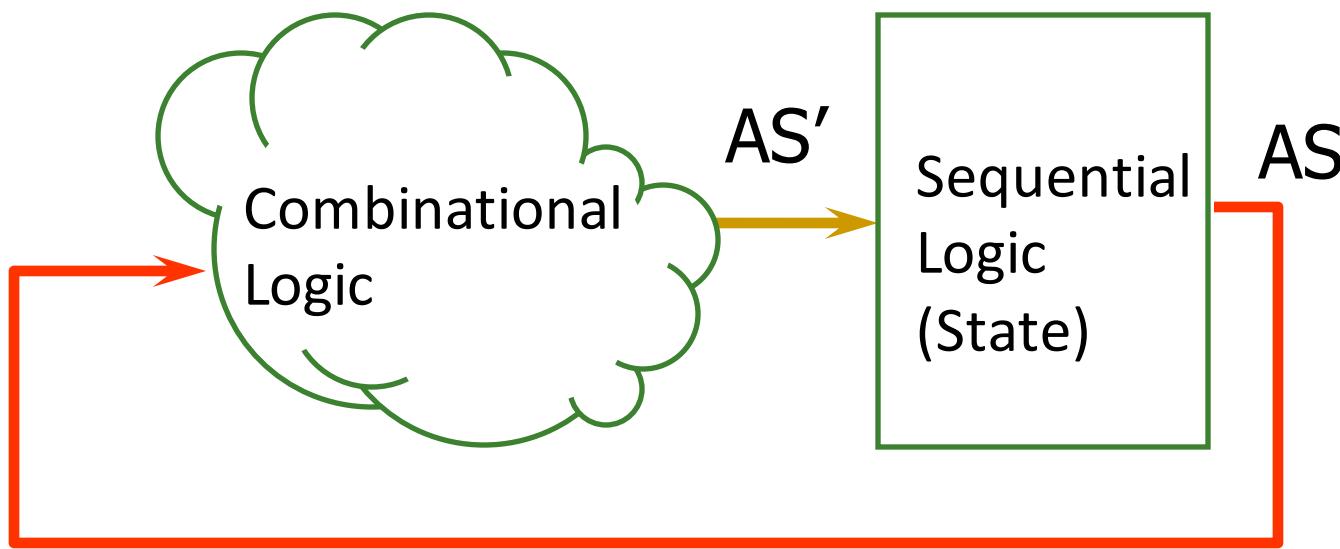
- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
 - CPI = 4.12
 - $T_c = 325 \text{ ps}$
- *Execution Time* $= (\# \text{ instructions}) \times \text{CPI} \times T_c$ $= (100 \times 10^9)(4.12)(325 \times 10^{-12})$ $= 133.9 \text{ seconds}$
- This is slower than the single-cycle processor (92.5 seconds). Why?
- Did we break the stages in a balanced manner?
- Overhead of register setup/hold paid many times
- How would the results change with different assumptions on memory latency and instruction mix?

Review: Single-Cycle MIPS Processor

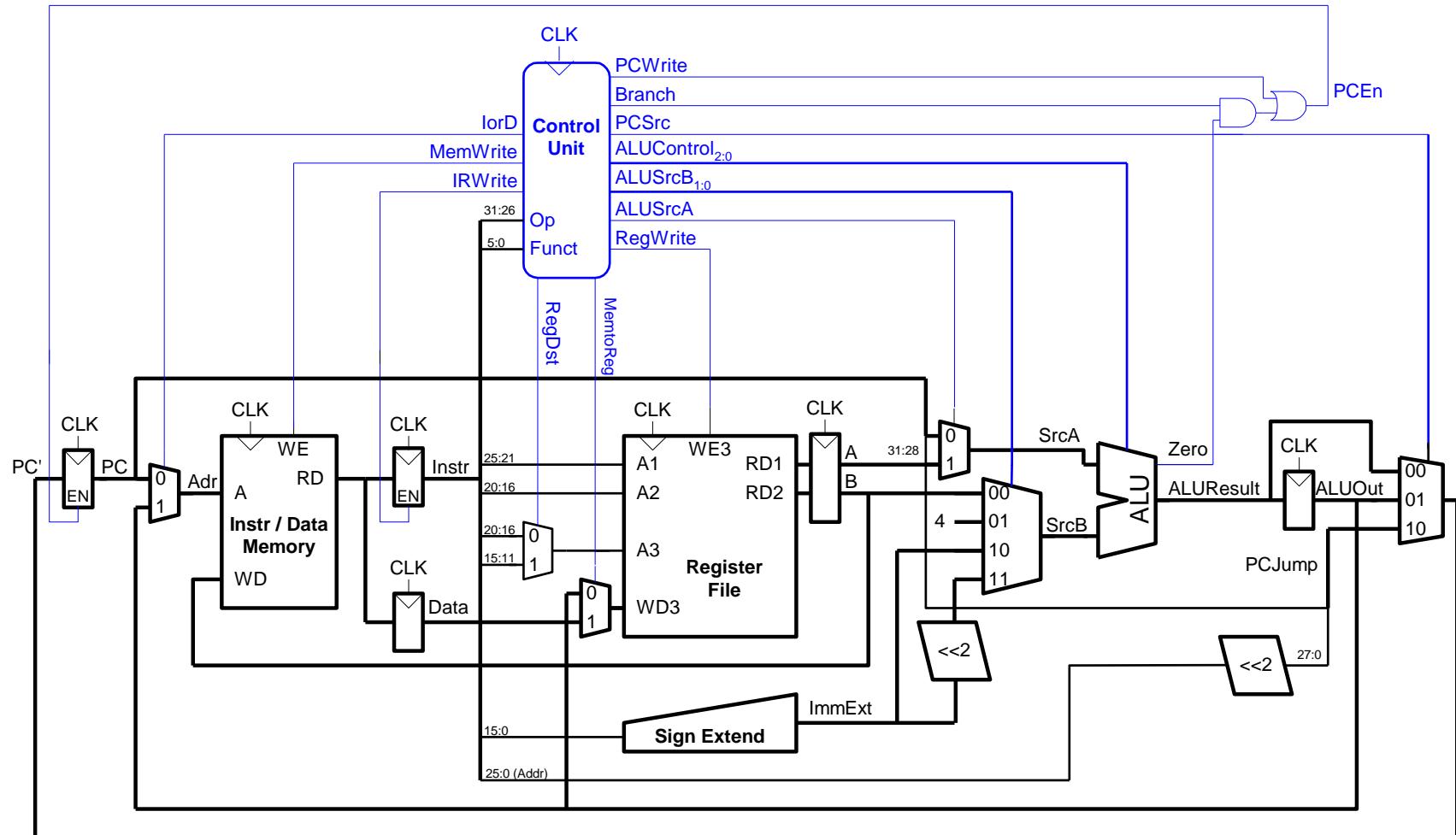


Review: Single-Cycle MIPS FSM

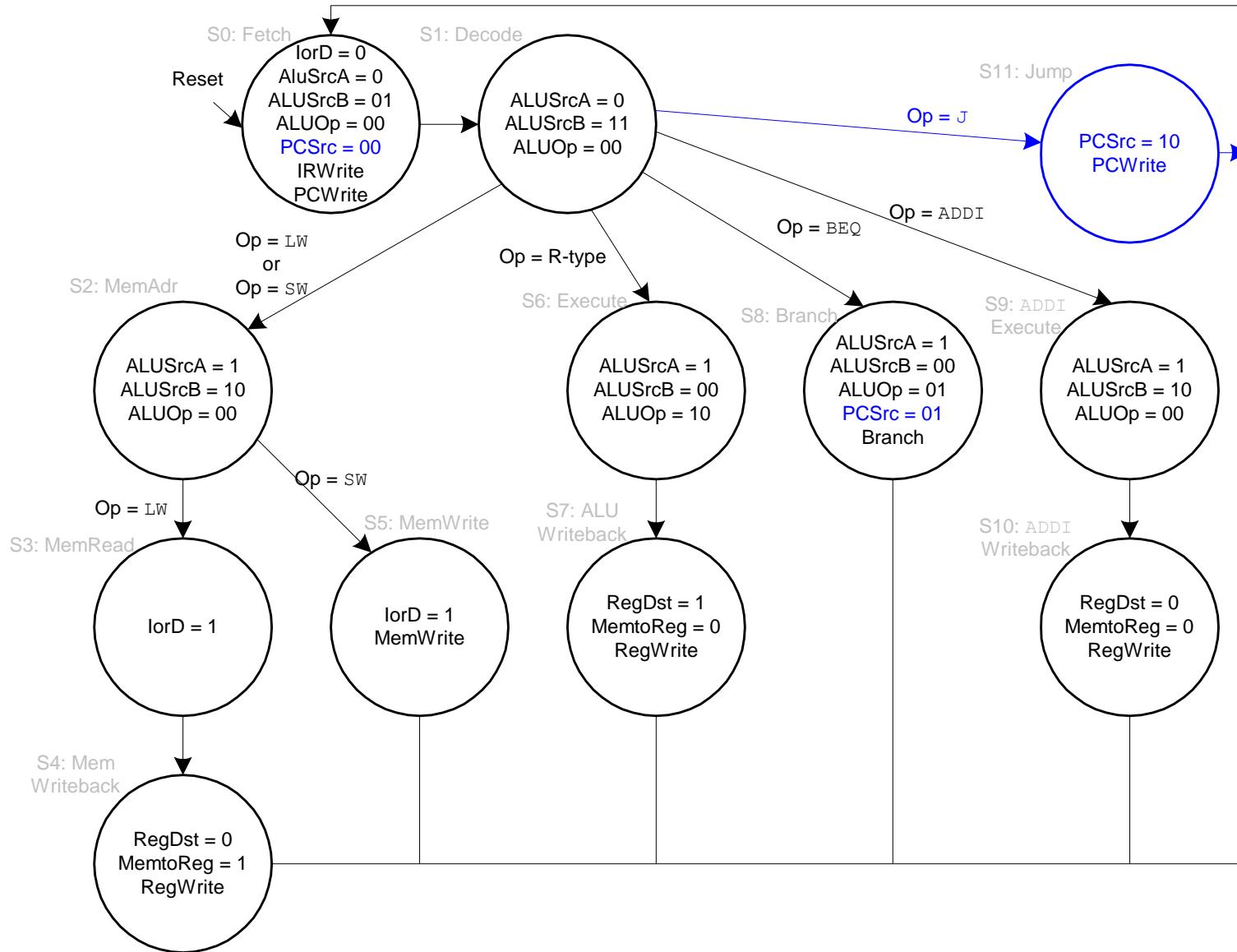
- Single-cycle machine



Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

What If Memory Takes > One Cycle?

- Stay in the same “memory access” state until memory returns the data
- “Memory Ready?” bit is an input to the control logic that determines the next state

Backup Slides on **Microprogrammed Multi-Cycle Microarchitectures**

These Slides Are Covered in A Past Lecture

Micropogrammed Control Terminology

- Control signals associated with the current state
 - Microinstruction
- Act of transitioning from one state to another
 - Determining the next state and the microinstruction for the next state
 - Microsequencing
- **Control store** stores control signals for every possible state
 - Store for microinstructions for the entire FSM
- **Microsequencer** determines which set of control signals will be used in the next clock cycle (i.e., next state)

28



▶ ▶ ⏪ 19:53 / 1:35:29

Design of Digital Circuits - Lecture 13: Microprogramming (ETH Zürich, Spring 2018)

2,301 views • Apr 14, 2018

Onur Mutlu Lectures 15.4K subscribers

Design of Digital Circuits, ETH Zürich, Spring 2018 (<https://safari.ethz.ch/digitaltechnik/>)

Lecture 13: Microprogramming
Lecturer: Professor Onur Mutlu (<http://people.inf.ethz.ch/omutlu>)
Date: April 13, 2018

199

Lectures on Microprogrammed Designs

- Design of Digital Circuits, Spring 2018, Lecture 13
 - Microprogramming (ETH Zürich, Spring 2018)
 - https://www.youtube.com/watch?v=u4GhShuBP3Y&list=PL5Q2soXY2Zi_QedyPWtRmFUJ2F8DdYP7l&index=13
- Computer Architecture, Spring 2013, Lecture 7
 - Microprogramming (CMU, Spring 2013)
 - https://www.youtube.com/watch?v=_igvSI5h8cs&list=PL5PHm2jkkXmidJ0d59REog9jDnPDTG6IJ&index=7

Another Example: **Microprogrammed Multi-Cycle Microarchitecture**

How Do We Implement This?

- Maurice Wilkes, "[The Best Way to Design an Automatic Calculating Machine](#)," Manchester Univ. Computer Inaugural Conf., 1951.

THE BEST WAY TO DESIGN AN AUTOMATIC CALCULATING MACHINE

By M. V. Wilkes, M.A., Ph.D., F.R.A.S.



- An elegant implementation:
 - The concept of microcoded/microprogrammed machines

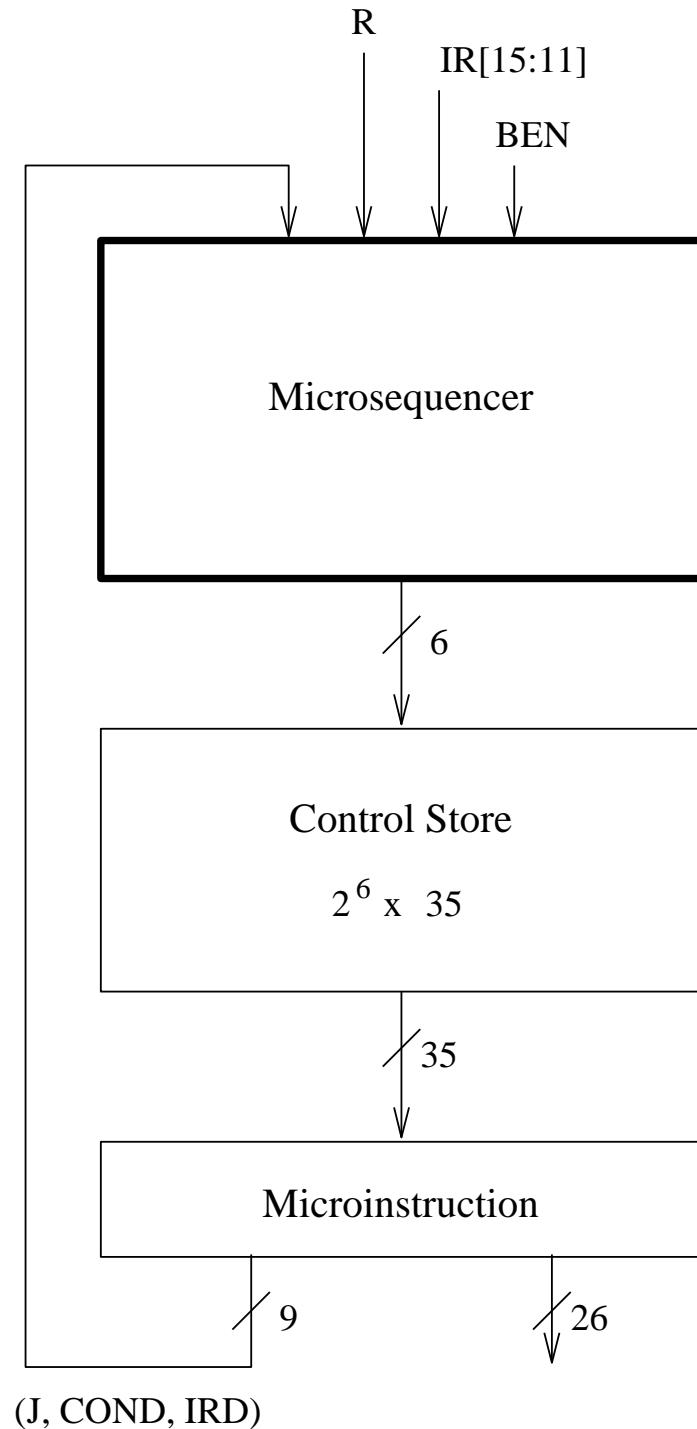
Recall: A Basic Multi-Cycle Microarchitecture

- Instruction processing cycle divided into “states”
 - A stage in the instruction processing cycle can take multiple states
- A multi-cycle microarchitecture sequences from state to state to process an instruction
 - The behavior of the machine in a state is completely determined by control signals in that state
- The behavior of the entire processor is specified fully by a *finite state machine*
- In a state (clock cycle), control signals control two things:
 - How the datapath should process the data
 - How to generate the control signals for the (next) clock cycle

Microprogrammed Control Terminology

- Control signals associated with the current state
 - **Microinstruction**
 - Act of transitioning from one state to another
 - Determining the next state and the microinstruction for the next state
 - **Microsequencing**
 - **Control store** stores control signals for every possible state
 - Store for microinstructions for the entire FSM
 - **Microsequencer** determines which set of control signals will be used in the next clock cycle (i.e., next state)
-

Example Control Structure



Simple Design
of the Control Structure

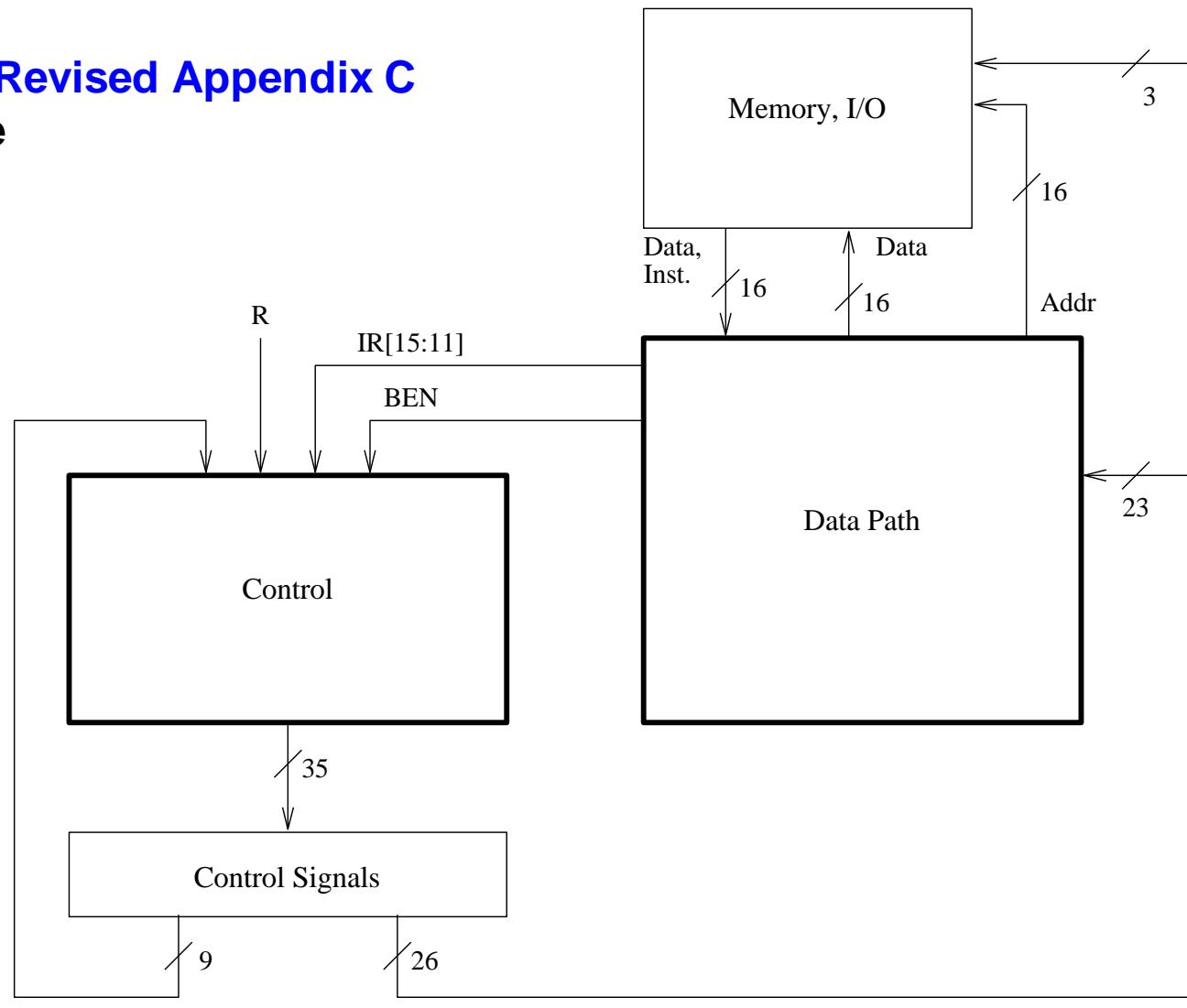
What Happens In A Clock Cycle?

- The control signals (microinstruction) for the current state control two things:
 - Processing in the data path
 - Generation of control signals (microinstruction) for the next cycle
 - *See Supplemental Figure 1 (next-next slide)*
- Datapath and microsequencer operate concurrently
- Question: why not generate control signals for the current cycle in the current cycle?
 - This could lengthen the clock cycle
 - Why could it lengthen the clock cycle?
 - *See Supplemental Figure 2*

Example uProgrammed Control & Datapath

Read P&P Revised Appendix C

On website

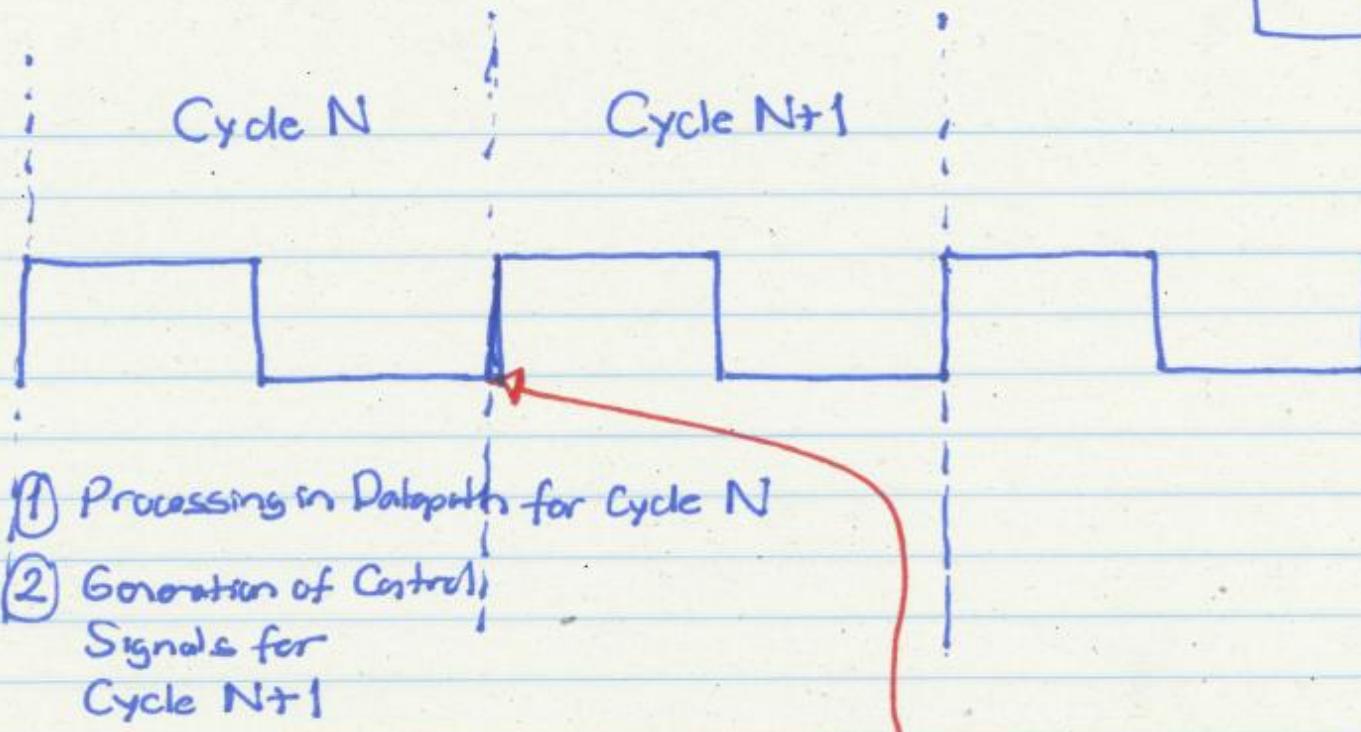


(J, COND, IRD)

Microarchitecture of the LC-3b, major components

A Clock Cycle

Supplemental
Figures



Latch

1) Results of current cycle N

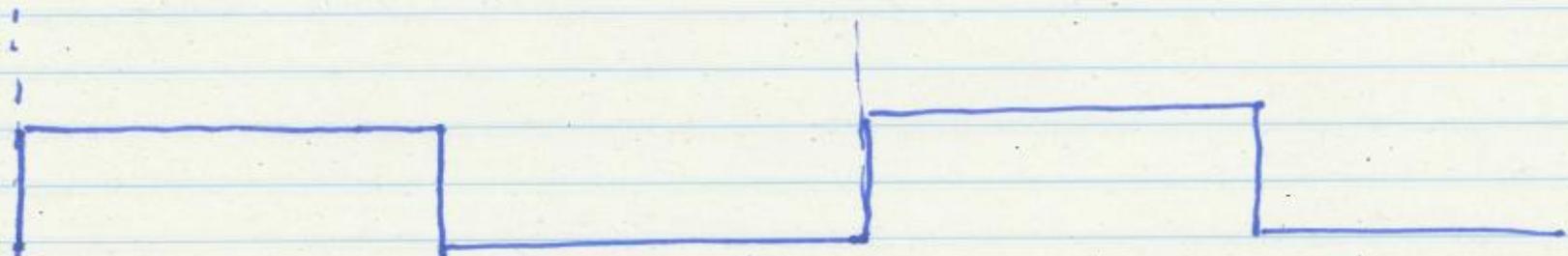
2) Control signals needed for the next cycle N+1

Fig 1

A Bad Clock Cycle!

Alternative - A BAD ONE!

Bad cycle !



① Generation of Control Signals for Cycle N

② Processing for Datapath for Cycle N

Step ② is dependent on Step ①

If Step ① takes non-zero time (it does!), clock cycle increases unnecessarily

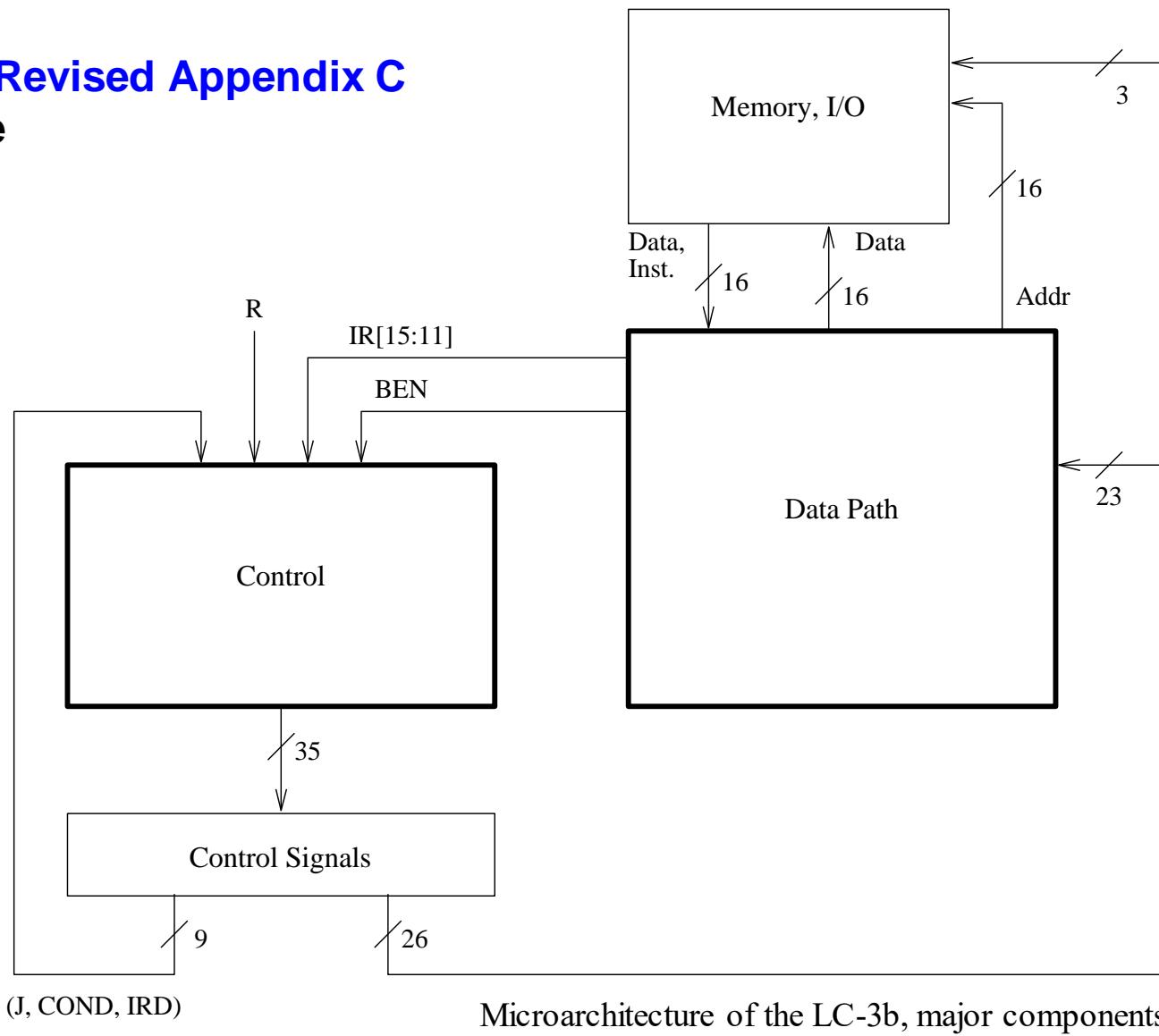
→ Violates the "Critical Path Design" principle

Fig 2

A Simple LC-3b Control and Datapath

Read P&P Revised Appendix C

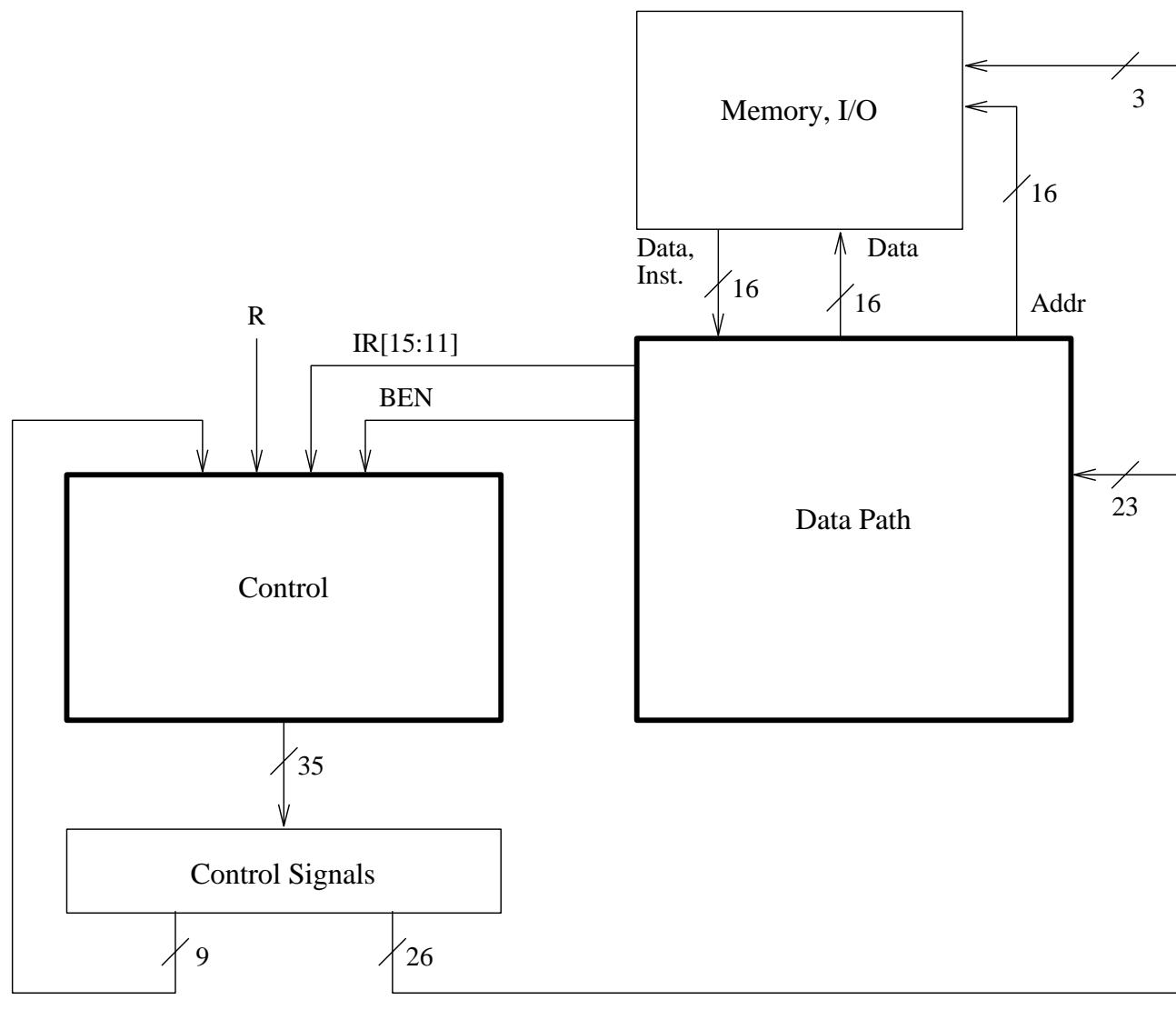
On website



What Determines Next-State Control Signals?

- What is happening in the current clock cycle
 - See the 9 control signals coming from “Control” block
 - What are these for?
- The instruction that is being executed
 - IR[15:11] coming from the Data Path
- Whether the condition of a branch is met, if the instruction being processed is a branch
 - BEN bit coming from the datapath
- Whether the memory operation is completing in the current cycle, if one is in progress
 - R bit coming from memory

A Simple LC-3b Control and Datapath

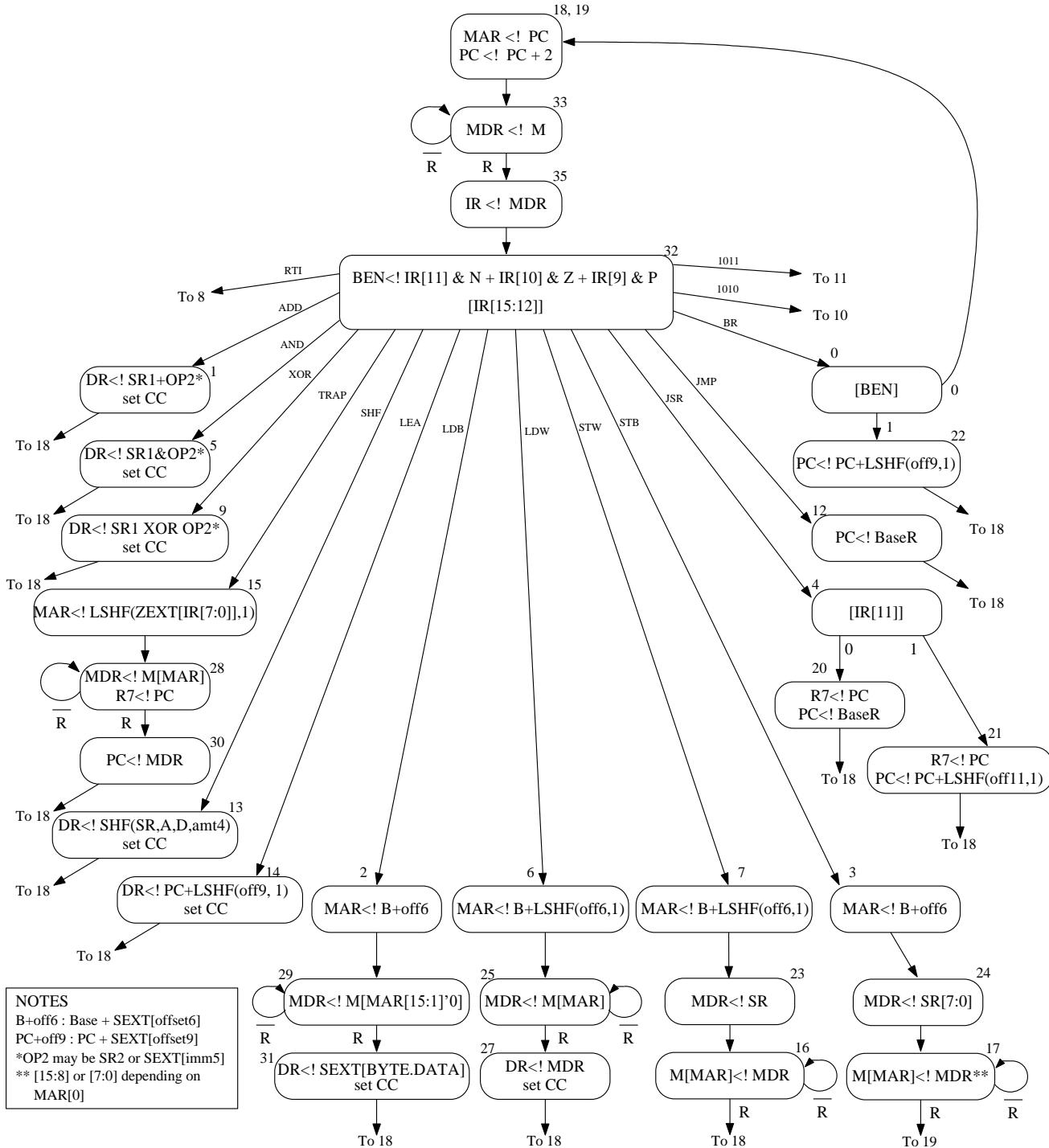


The State Machine for Multi-Cycle Processing

- The behavior of the LC-3b uarch is completely determined by
 - the 35 control signals and
 - additional 7 bits that go into the control logic from the datapath
- 35 control signals completely describe the state of the control structure
- We can completely describe the behavior of the LC-3b as a state machine, i.e. a directed graph of
 - Nodes (one corresponding to each state)
 - Arcs (showing flow from each state to the next state(s))

An LC-3b State Machine

- Patt and Patel, Revised Appendix C, Figure C.2
- Each state must be uniquely specified
 - Done by means of *state variables*
- 31 distinct states in this LC-3b state machine
 - Encoded with 6 state variables
- Examples
 - State 18,19 correspond to the beginning of the instruction processing cycle
 - Fetch phase: state 18, 19 → state 33 → state 35
 - Decode phase: state 32



The FSM Implements the LC-3b ISA

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺	0001		DR		SR1		A		op.spec							
AND ⁺	0101		DR		SR1		A		op.spec							
BR	0000	n	z	p				PCoffset9								
JMP	1100		000		BaseR			000000								
JSR(R)	0100	A			operand.specifier											
LDB ⁺	0010		DR		BaseR			boffset6								
LDW ⁺	0110		DR		BaseR			offset6								
LEA ⁺	1110		DR			PCoffset9										
RTI	1000				00000000000000											
SHF ⁺	1101		DR		SR	A	D	amount4								
STB	0011		SR		BaseR			boffset6								
STW	0111		SR		BaseR			offset6								
TRAP	1111		0000			trapvect8										
XOR ⁺	1001		DR		SR1	A	op.spec									
not used	1010															
not used	1011															

■ P&P Appendix A (revised):

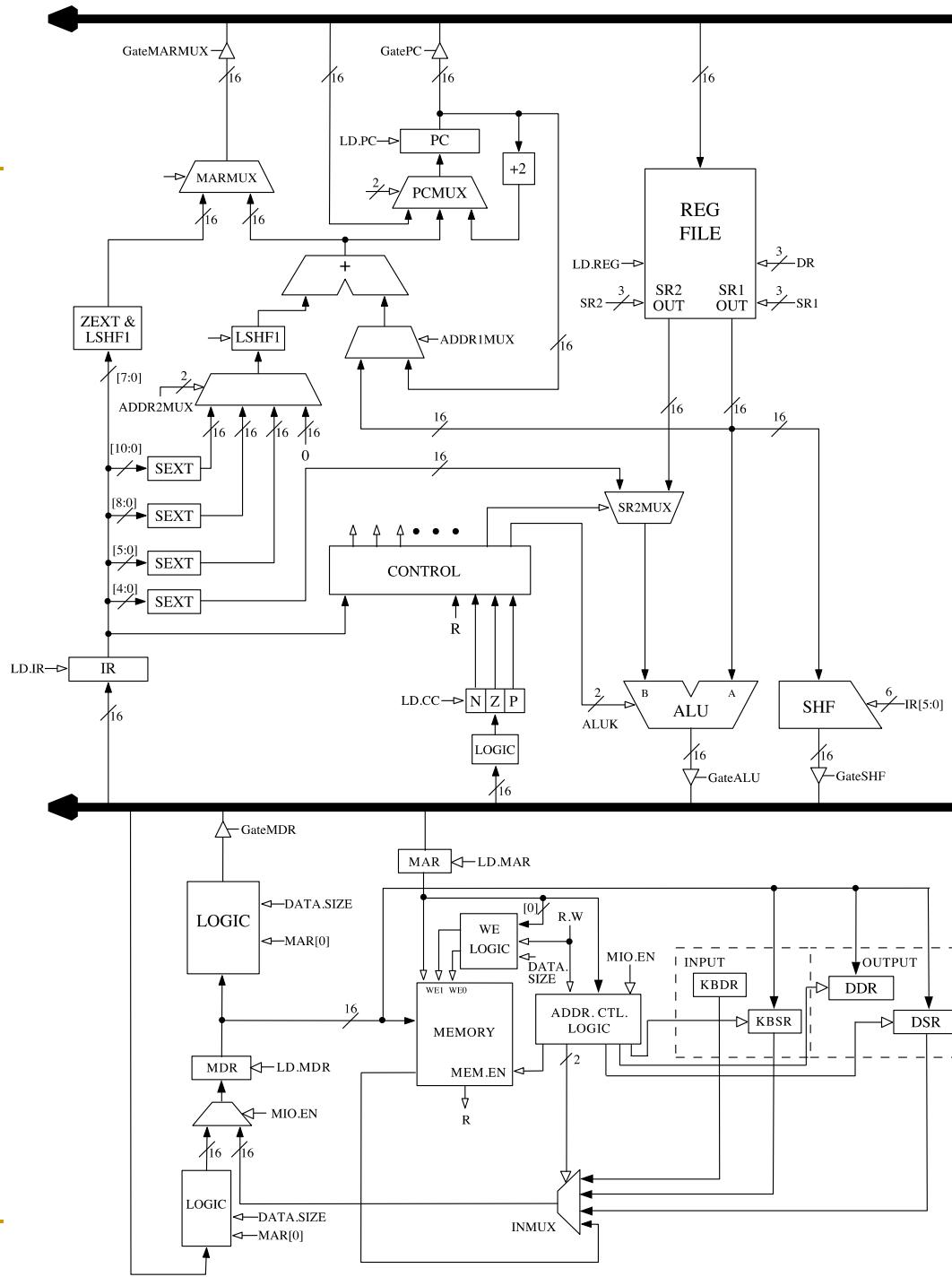
- <https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=pp-appendixa.pdf>

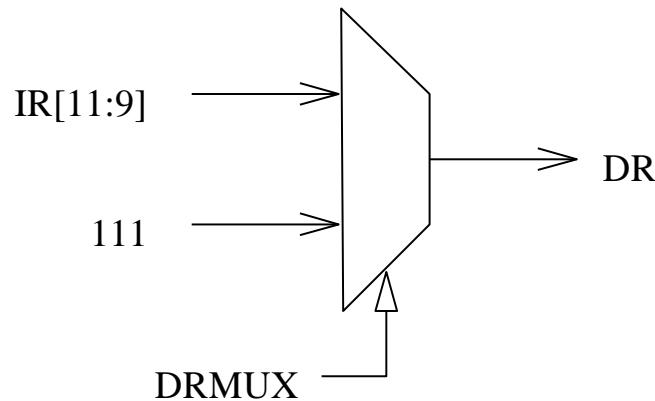
LC-3b State Machine: Some Questions

- How many cycles does the fastest instruction take?
- How many cycles does the slowest instruction take?
- Why does the BR take as long as it takes in the FSM?
- What determines the clock cycle time?

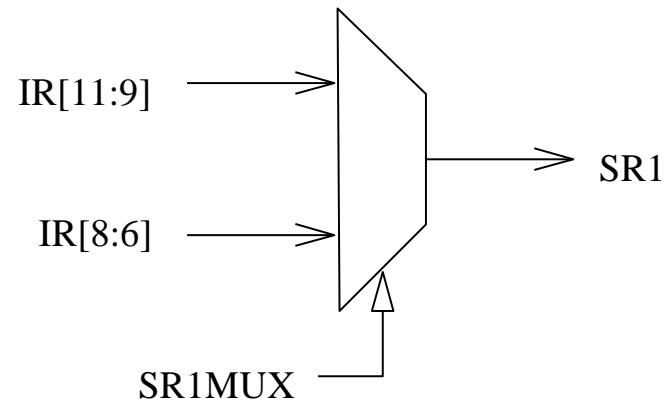
LC-3b Datapath

- Patt and Patel, Revised Appendix C, Figure C.3
- Single-bus datapath design
 - At any point only one value can be “gated” on the bus (i.e., can be driving the bus)
 - **Advantage:** Low hardware cost: one bus
 - **Disadvantage:** Reduced concurrency – if instruction needs the bus twice for two different things, these need to happen in different states
- Control signals (26 of them) determine what happens in the datapath in one clock cycle
 - Patt and Patel, Revised Appendix C, Table C.1



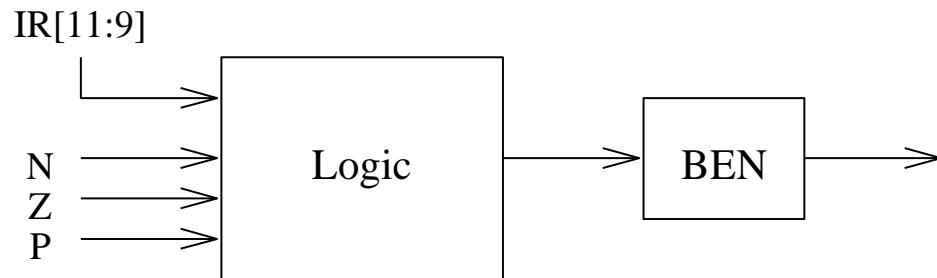


(a)



(b)

Remember the MIPS datapath



(c)

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

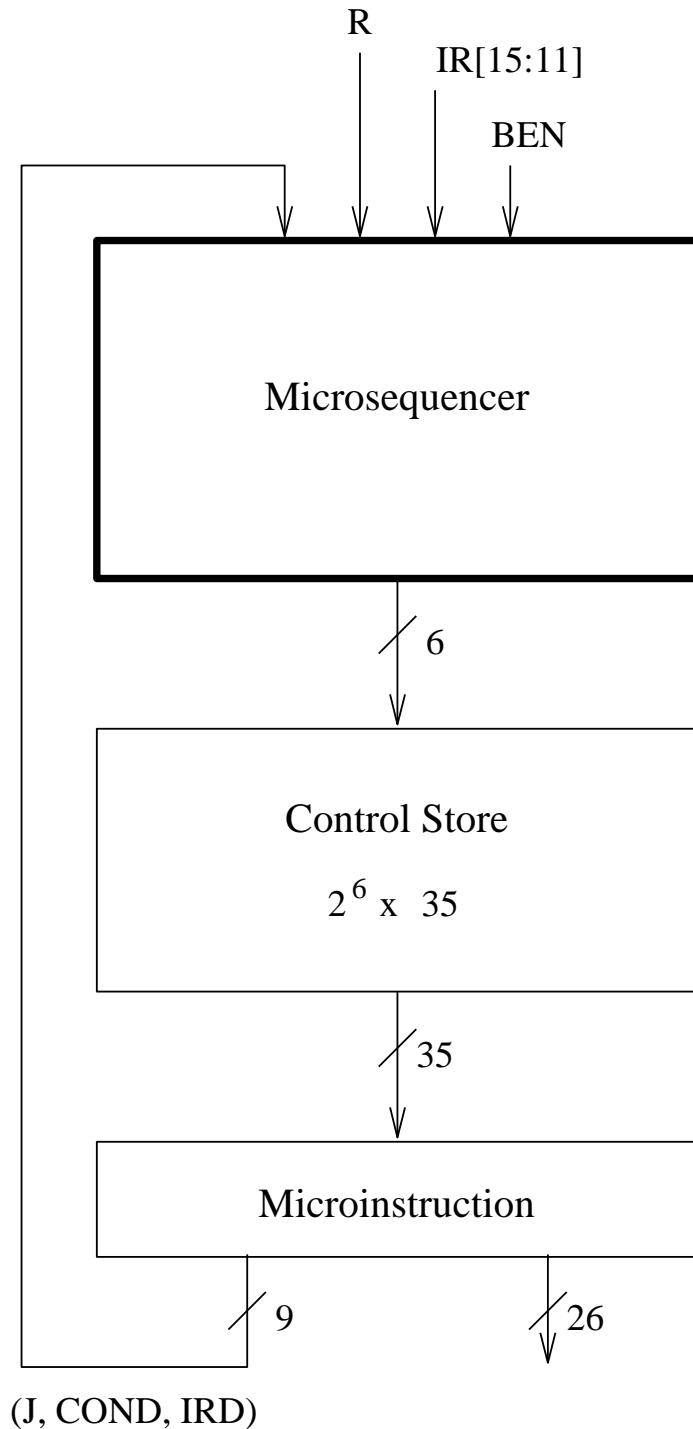
Table C.1: Data path control signals

LC-3b Datapath: Some Questions

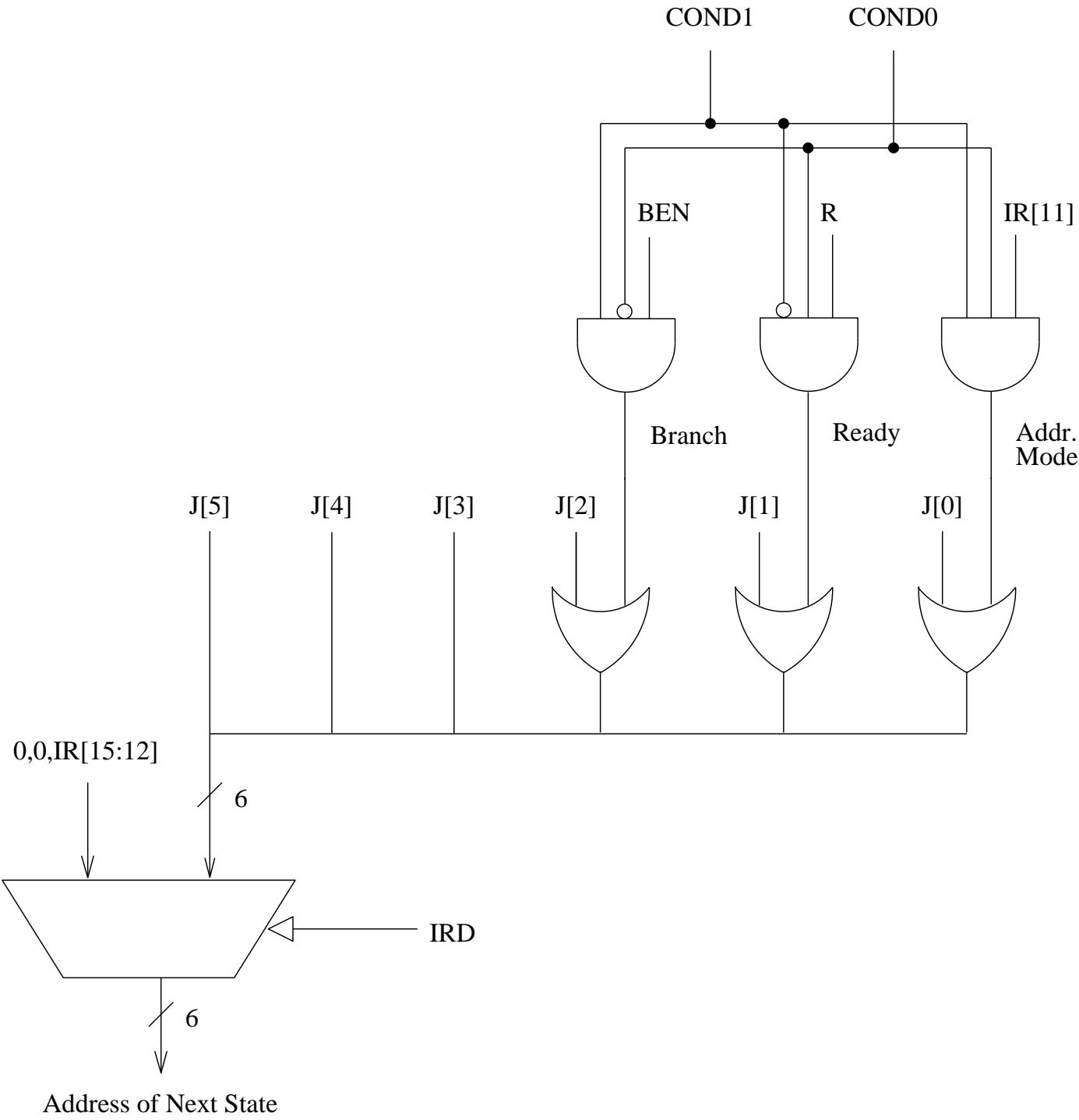
- How does instruction fetch happen in this datapath according to the state machine?
- What is the difference between gating and loading?
 - Gating: Enable/disable an input to be connected to the bus
 - Combinational: during a clock cycle
 - Loading: Enable/disable an input to be written to a register
 - Sequential: e.g., at a clock edge (assume at the end of cycle)
- Is this the smallest hardware you can design?

LC-3b Microprogrammed Control Structure

- Patt and Patel, Appendix C, Figure C.4
- Three components:
 - Microinstruction, control store, microsequencer
- **Microinstruction**: control signals that control the datapath (26 of them) and help determine the next state (9 of them)
- Each microinstruction is stored in a *unique location* in the **control store** (a special memory structure)
- *Unique location*: address of the state corresponding to the microinstruction
 - Remember each state corresponds to one microinstruction
- **Microsequencer** determines the address of the next microinstruction (i.e., next state)



Simple Design of the Control Structure

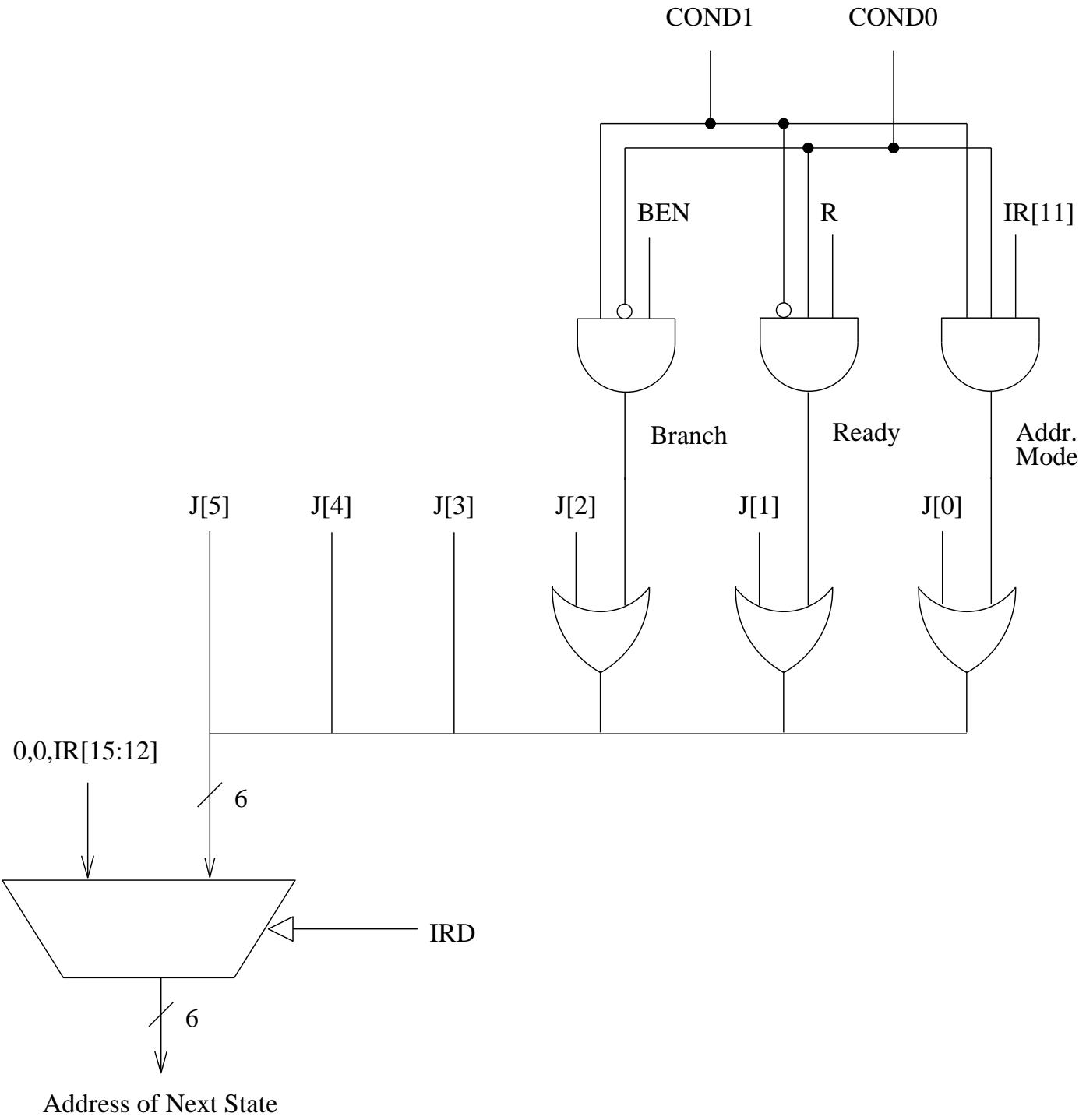


LC-3b Microsequencer

- Patt and Patel, Appendix C, Figure C.5
- The purpose of the microsequencer is to determine the address of the next microinstruction (i.e., next state)
 - Next state could be conditional or unconditional
- Next state address depends on 9 control signals (plus 7 data signals)

Signal Name	Signal Values
J/6:	
COND/2:	COND ₀ ;Unconditional
	COND ₁ ;Memory Ready
	COND ₂ ;Branch
	COND ₃ ;Addressing Mode
IRD/1:	NO, YES

Table C.2: Microsequencer control signals



The Microsequencer: Some Questions

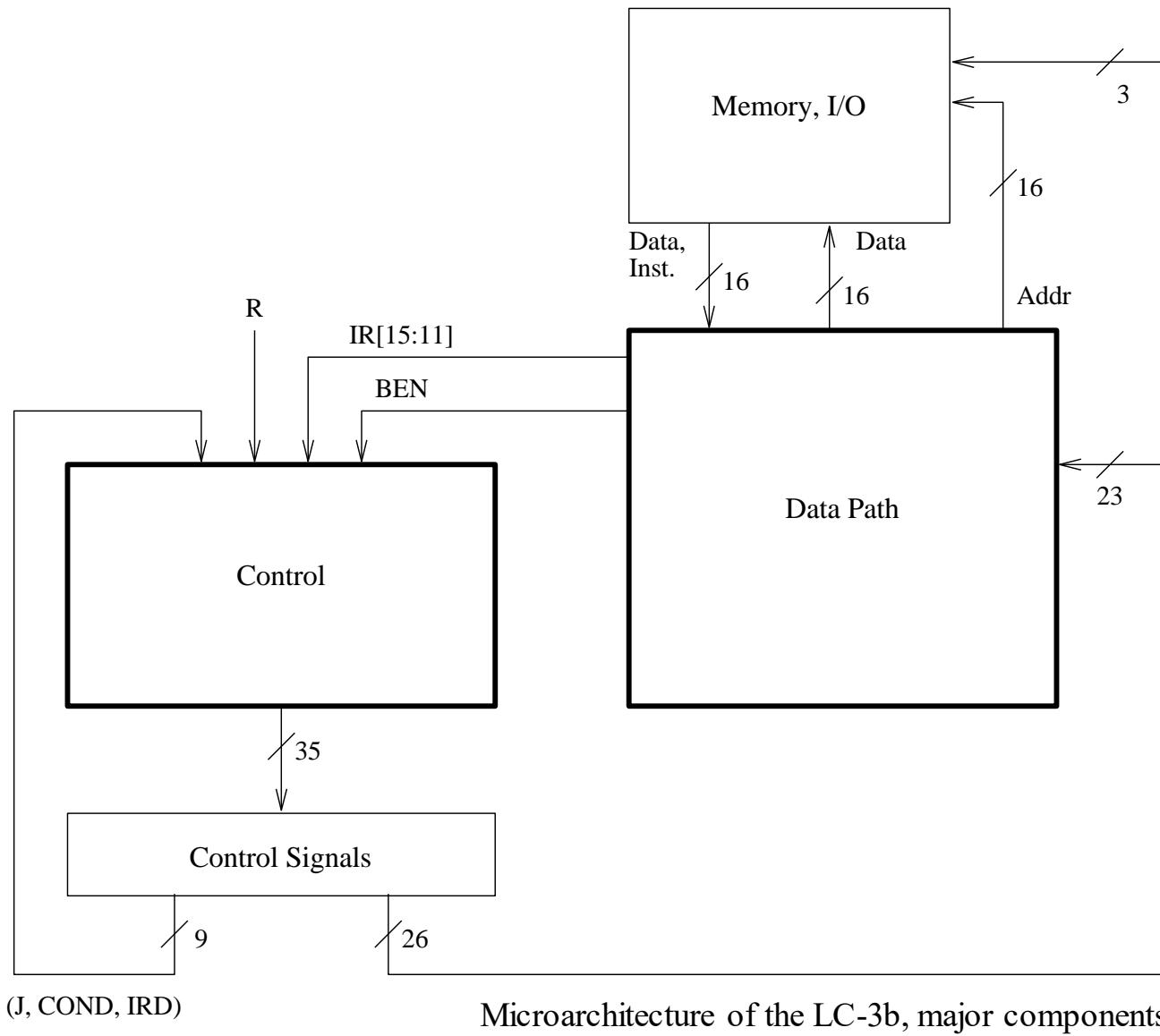
- When is the IRD signal asserted?
- What happens if an illegal instruction is decoded?
- What are condition (COND) bits for?
- How is variable latency memory handled?
- How do you do the state encoding?
 - Minimize number of state variables (~ control store size)
 - Start with the 16-way branch
 - Then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

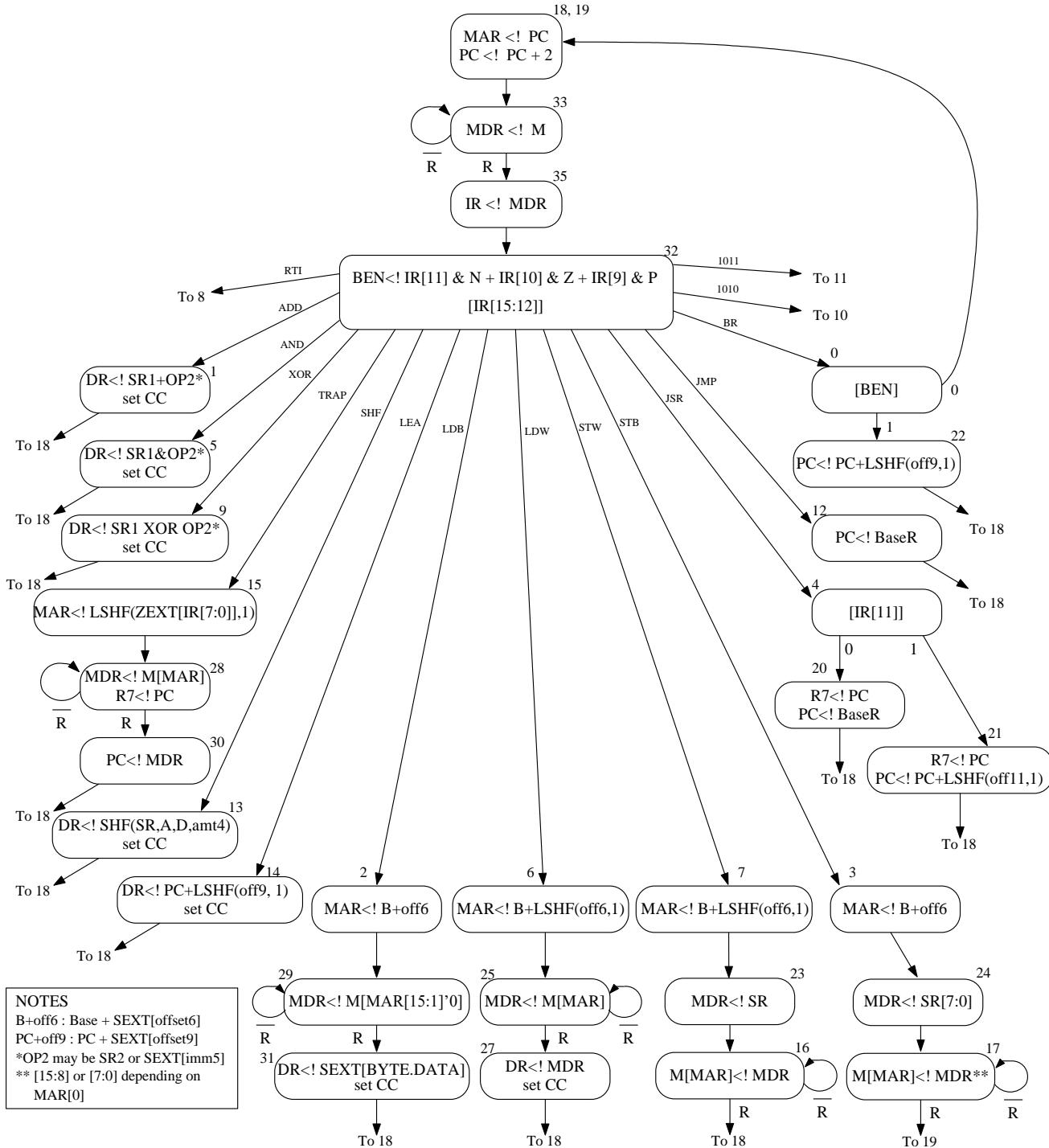
An Exercise in Microprogramming

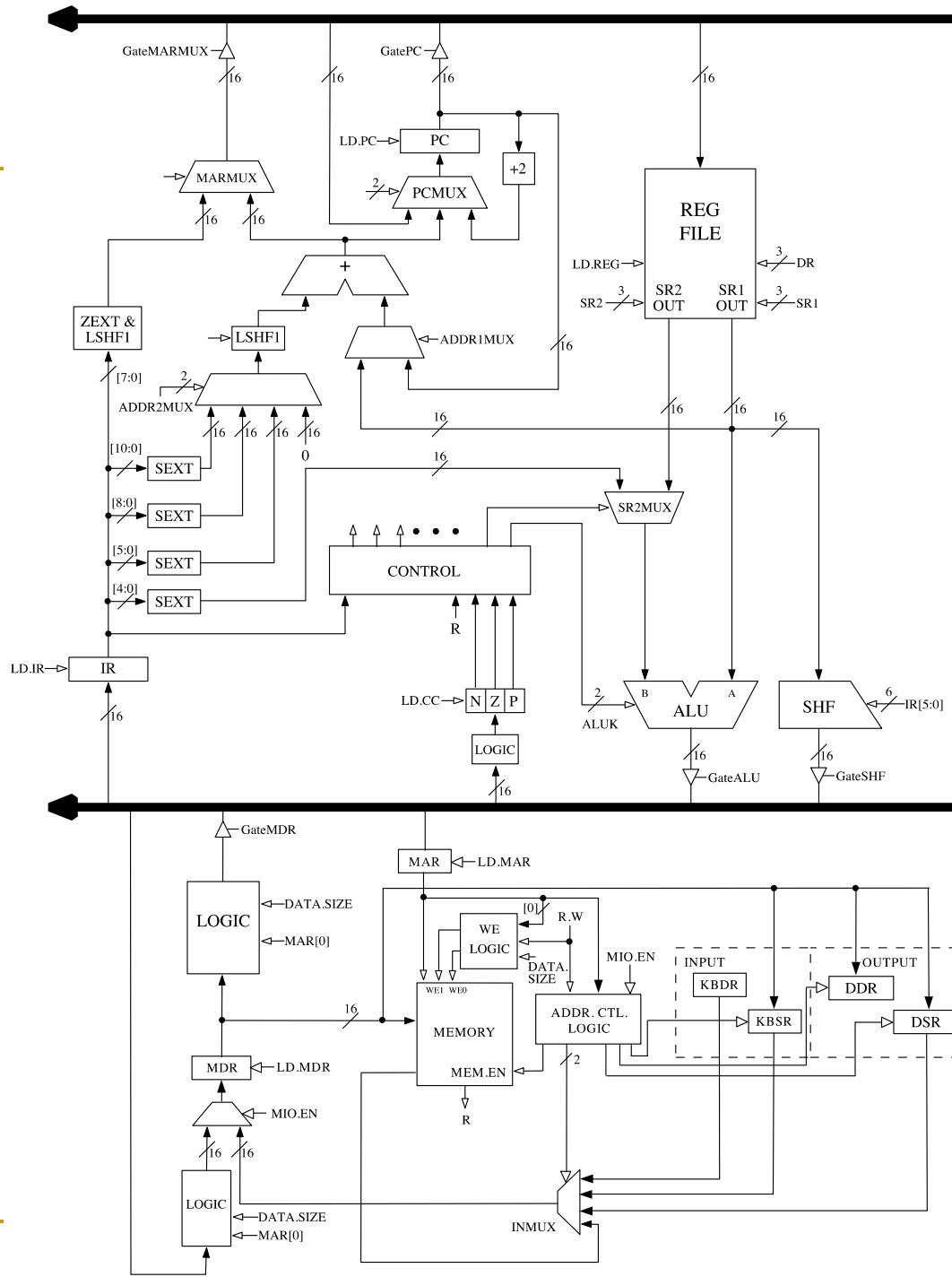
Handouts

- 7 pages of Microprogrammed LC-3b design
- <https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=lc3b-figures.pdf>

A Simple LC-3b Control and Datapath

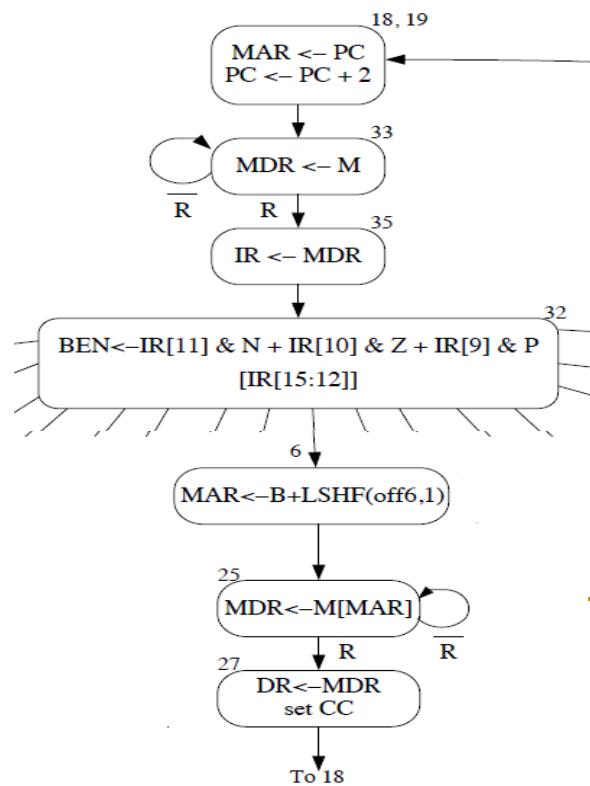




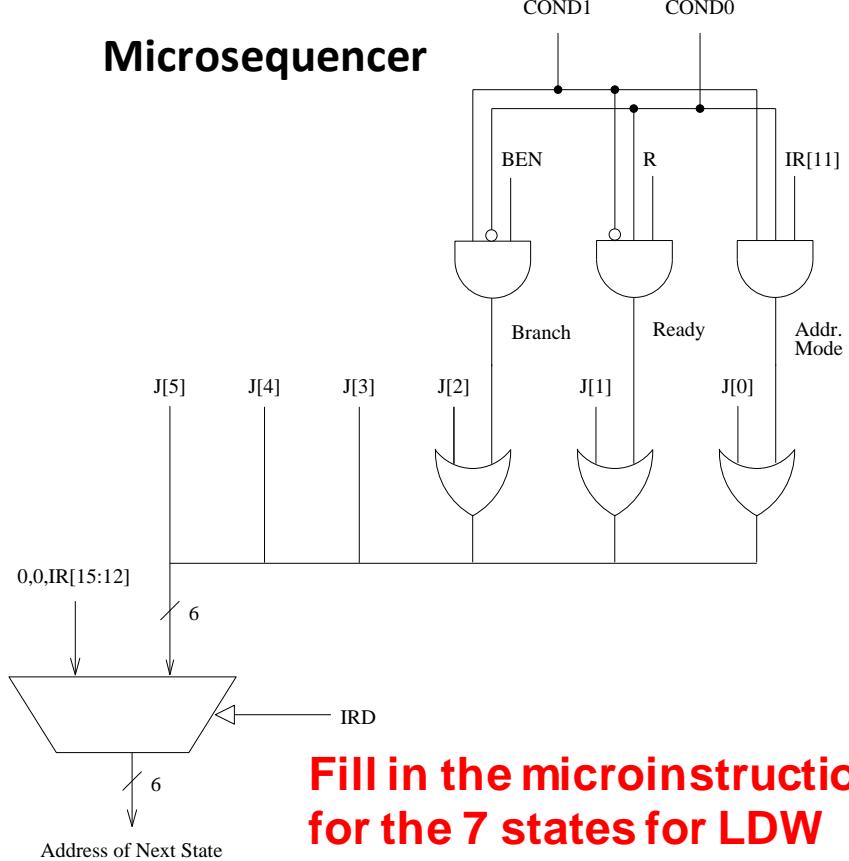


A Simple Datapath
Can Become
Very Powerful

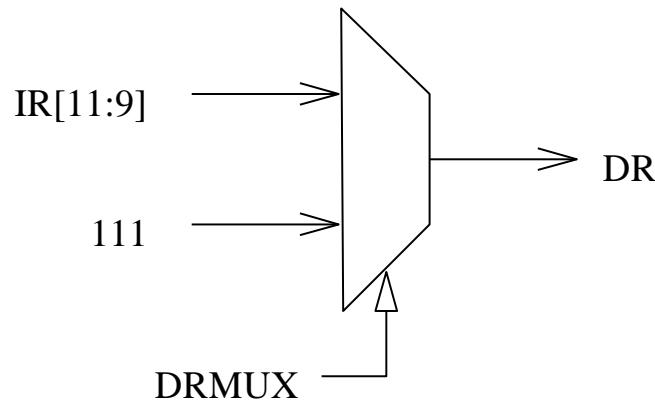
State Machine for LDW



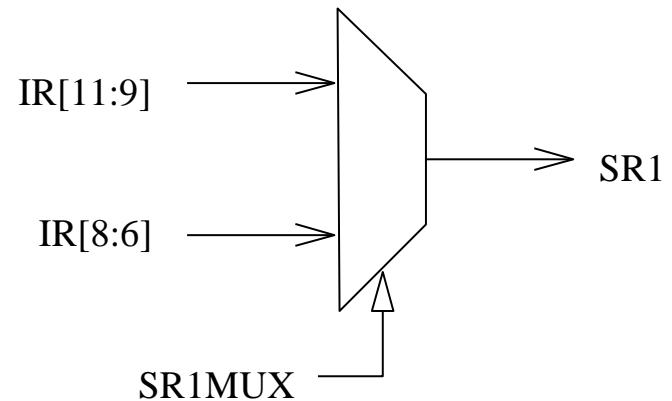
Microsequencer



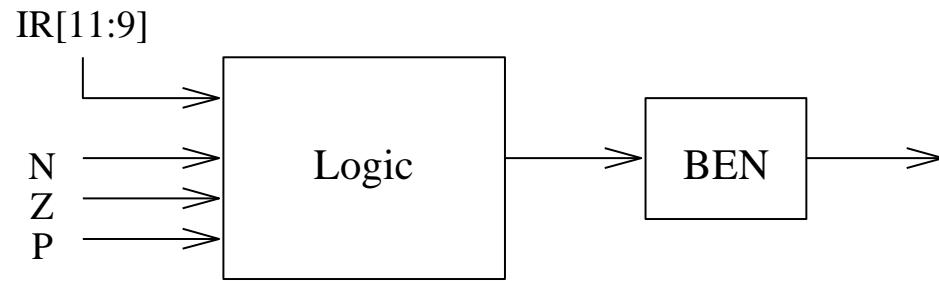
Fill in the microinstructions for the 7 states for LDW



(a)



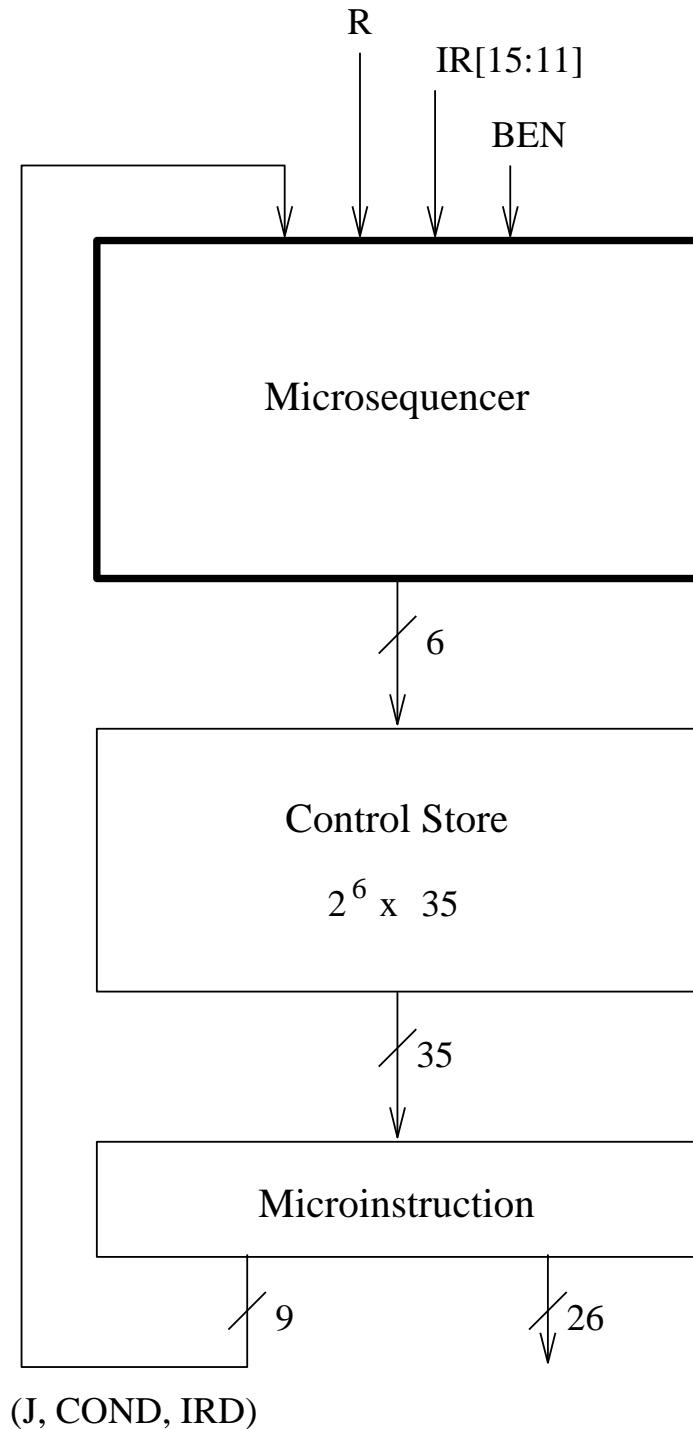
(b)



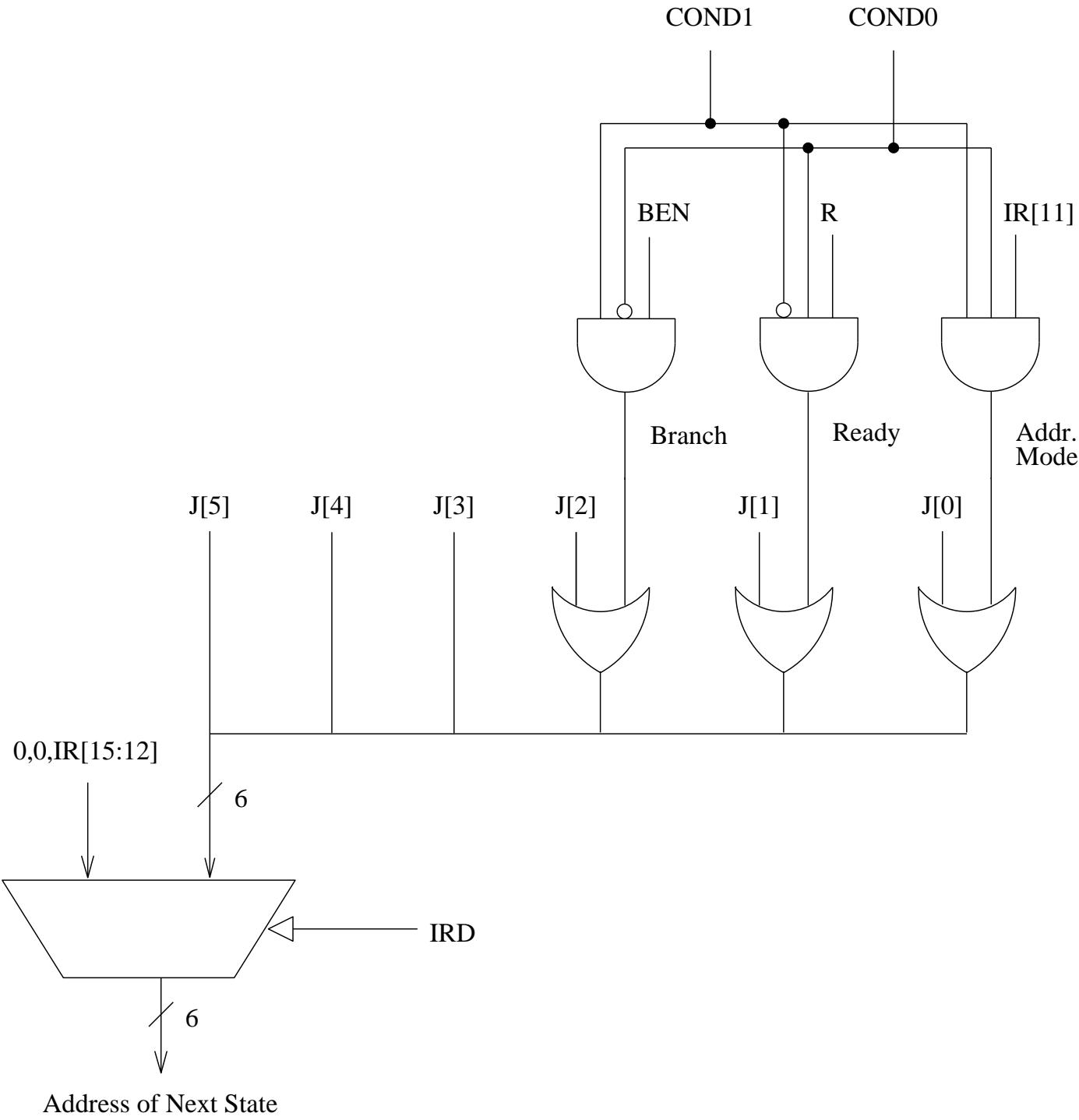
(c)

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

Table C.1: Data path control signals



Simple Design
of the Control Structure



End of the Exercise in Microprogramming

Variable-Latency Memory

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

The Microsequencer: Advanced Questions

- What happens if the machine is interrupted?
- What if an instruction generates an exception?
- How can you implement a complex instruction using this control structure?
 - Think REP MOVS instruction in x86
 - string copy of N elements starting from address A to address B

The Power of Abstraction

- The concept of a control store of microinstructions enables the hardware designer with a new abstraction: **microprogramming**
- The designer can translate any desired operation to a sequence of microinstructions
- All the designer needs to provide is
 - **The sequence of microinstructions** needed to implement the desired operation
 - **The ability for the control logic to correctly sequence** through the microinstructions
 - **Any additional datapath elements and control signals** needed (no need if the operation can be “translated” into existing control signals)

Let's Do Some More Microprogramming

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

x86 REP MOVS (String Copy) Instruction

REP MOVS (DEST SRC)

```
IF AddressSize = 16
  THEN
    Use CX for CountReg;
  ELSE IF AddressSize = 64 and REX.W used
    THEN Use RCX for CountReg; Fl;
  ELSE
    Use ECX for CountReg;
  Fl;
WHILE CountReg ≠ 0
  DO
    Service pending interrupts (if any);
    Execute associated string instruction;
    CountReg ← (CountReg - 1);
    IF CountReg = 0
      THEN exit WHILE loop; Fl;
    IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
    or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
      THEN exit WHILE loop; Fl;
  OD;
```

```
DEST ← SRC;
IF (Byte move)
  THEN IF DF = 0
    THEN
      (R|E)SI ← (R|E)SI + 1;
      (R|E)DI ← (R|E)DI + 1;
    ELSE
      (R|E)SI ← (R|E)SI - 1;
      (R|E)DI ← (R|E)DI - 1;
    Fl;
  ELSE IF (Word move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 2;
      (R|E)DI ← (R|E)DI + 2;
    ELSE
      (R|E)SI ← (R|E)SI - 2;
      (R|E)DI ← (R|E)DI - 2;
    Fl;
  ELSE IF (Doubleword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 4;
      (R|E)DI ← (R|E)DI + 4;
    ELSE
      (R|E)SI ← (R|E)SI - 4;
      (R|E)DI ← (R|E)DI - 4;
    Fl;
  ELSE IF (Quadword move)
    THEN IF DF = 0
      (R|E)SI ← (R|E)SI + 8;
      (R|E)DI ← (R|E)DI + 8;
    ELSE
      (R|E)SI ← (R|E)SI - 8;
      (R|E)DI ← (R|E)DI - 8;
    Fl;
```

How many instructions does this take in MIPS ISA?

How many microinstructions does this take to add to the LC-3b microarchitecture? 246

Aside: Alignment Correction in Memory

- Unaligned accesses
- LC-3b has byte load and byte store instructions that move data not aligned at the word-address boundary
 - Convenience to the programmer/compiler
- How does the hardware ensure this works correctly?
 - Take a look at state 29 for LDB
 - States 24 and 17 for STB
 - Additional logic to handle unaligned accesses
- P&P, Revised Appendix C.5

Aside: Memory Mapped I/O

- Address control logic determines whether the specified address of LDW and STW are to memory or I/O devices
- Correspondingly enables memory or I/O devices and sets up muxes
- An instance where the final control signals of some datapath elements (e.g., MEM.EN or INMUX/2) **cannot** be stored in the control store
 - These signals are dependent on memory address
- P&P, Revised Appendix C.6

Advantages of Microprogrammed Control

- Allows a very simple design to do powerful computation by controlling the datapath (using a sequencer)
 - High-level ISA translated into microcode (sequence of u-instructions)
 - Microcode (u-code) enables a minimal datapath to emulate an ISA
 - Microinstructions can be thought of as a **user-invisible ISA (u-ISA)**
- Enables easy extensibility of the ISA
 - Can support a new instruction by changing the microcode
 - Can support complex instructions as a sequence of simple microinstructions (e.g., REP MOVS, INC [MEM])
- Enables update of machine behavior
 - A buggy implementation of an instruction can be fixed by changing the microcode in the field
 - Easier if datapath provides ability to do the same thing in different ways

Update of Machine Behavior

- The ability to update/patch microcode in the field (after a processor is shipped) enables
 - Ability to add new instructions without changing the processor!
 - Ability to “fix” buggy hardware implementations
- Examples
 - IBM 370 Model 145: microcode stored in main memory, can be updated after a reboot
 - IBM System z: Similar to 370/145.
 - Heller and Farrell, “[Millicode in an IBM zSeries processor](#),” IBM JR&D, May/Jul 2004.
 - B1700 microcode can be updated while the processor is running
 - User-microprogrammable machine!
 - Wilner, “Microprogramming environment on the Burroughs B1700”, CompCon 1972.

Multi-Cycle vs. Single-Cycle uArch

- Advantages
- Disadvantages
- For you to fill in

