Design of Digital Circuits
Lecture 11: Microarchitecture

Prof. Onur Mutlu
ETH Zurich
Spring 2018
28 March 2019

Readings

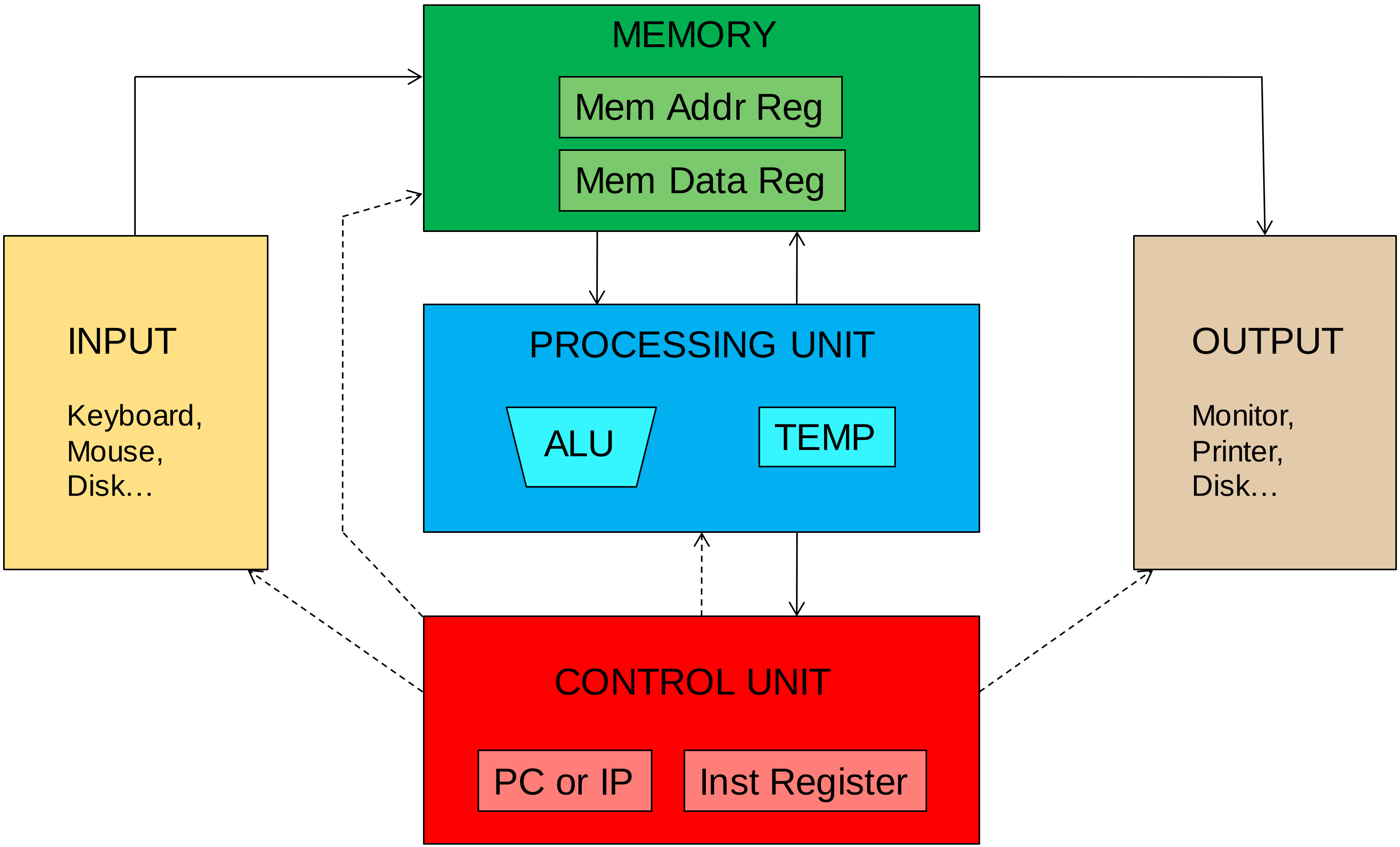
- This week
  - Introduction to microarchitecture and single-cycle microarchitecture
    - H&H, Chapter 7.1-7.3
    - P&P, Appendices A and C
  - Multi-cycle microarchitecture
    - H&H, Chapter 7.4
    - P&P, Appendices A and C

- Next week
  - Pipelining
    - H&H, Chapter 7.5
  - Pipelining Issues
    - H&H, Chapter 7.8.1-7.8.3

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

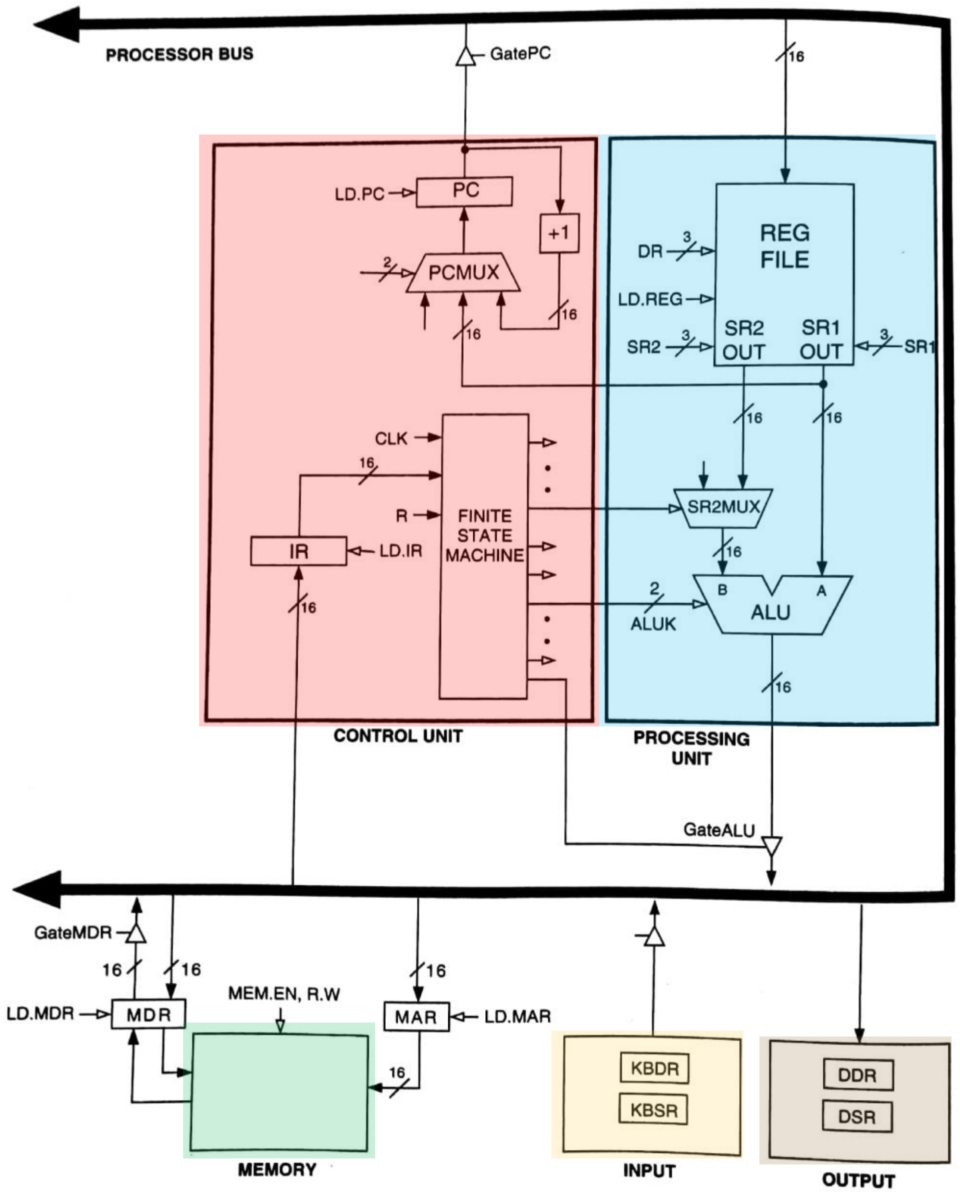


Figure 4.3 The LC-3 as an example of the von Neumann model

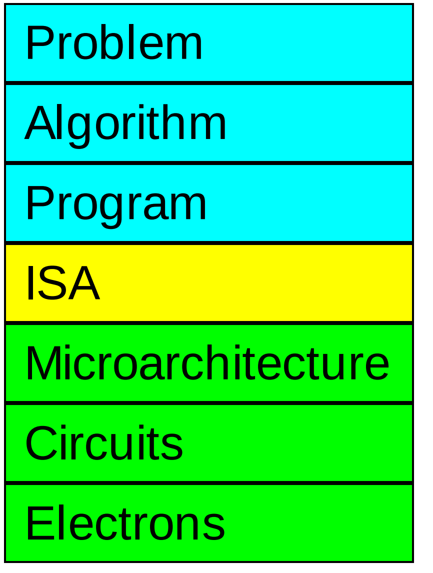
Recall: The Instruction Cycle

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

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: 32 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, …
  - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, … AMD K5, K7, K9, Bulldozer, BobCat, …

(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

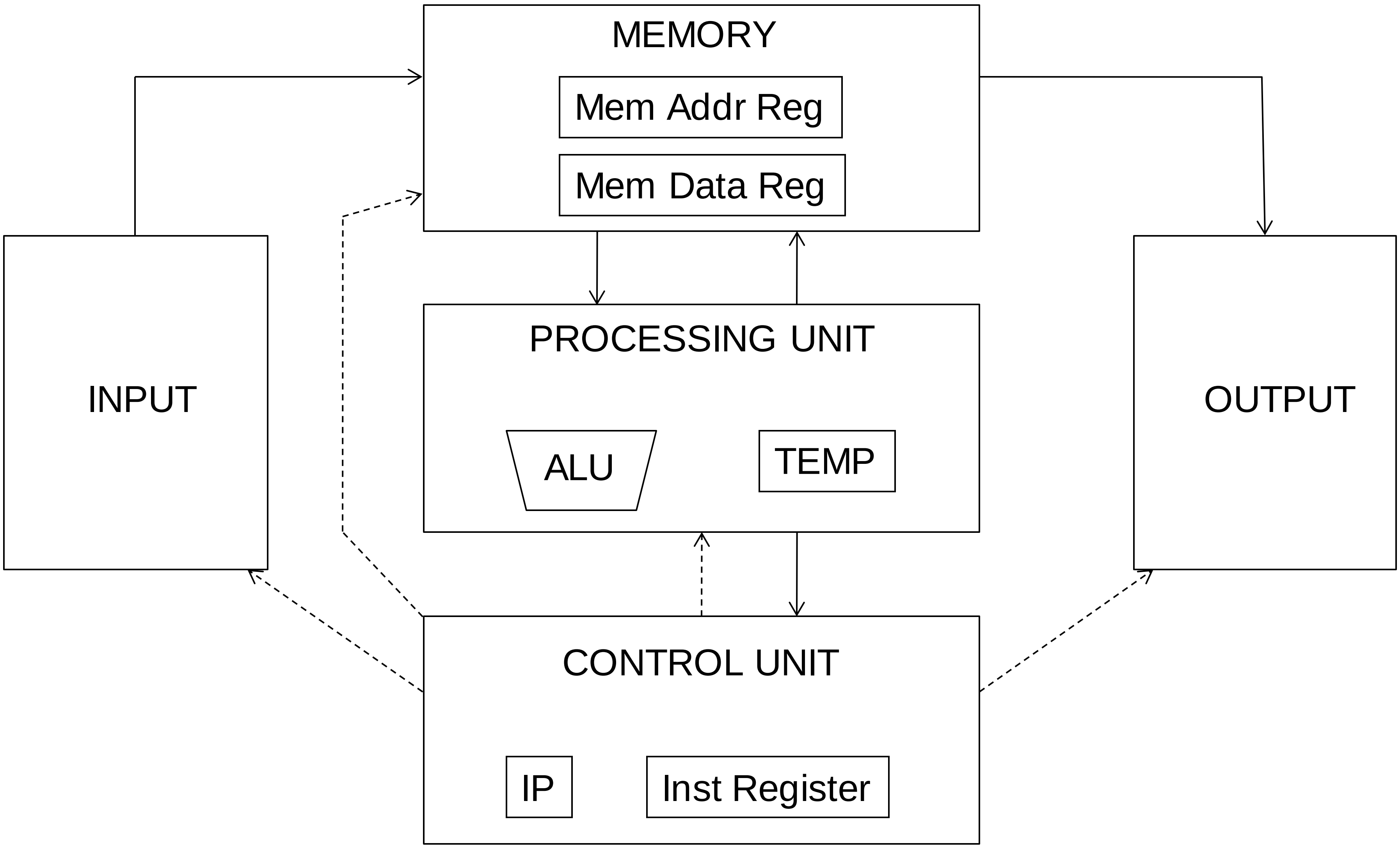
When is a value interpreted as an instruction?

- Sequential instruction processing
  - One instruction processed (fetched, executed, completed) at a time
  - Program counter (instruction pointer) identifies the current instruction
  - Program counter is advanced sequentially except for control transfer instructions

The Von Neumann Model/Architecture

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

The Von Neumann Model (of a Computer)



The Von Neumann Model (of a Computer)

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

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

The Dataflow Model (of a Computer)

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

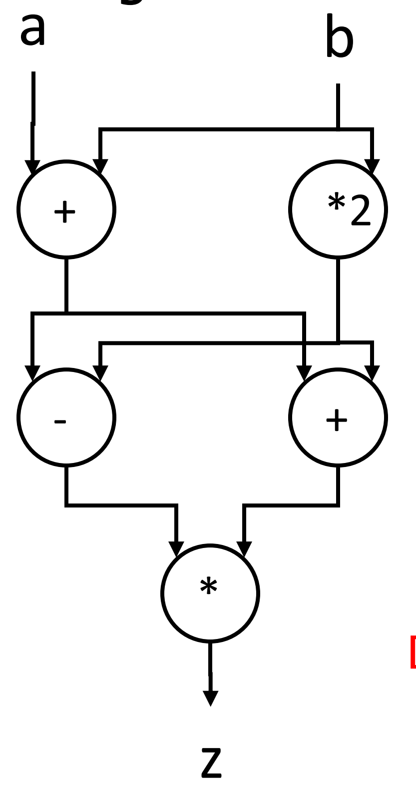
- Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies "who" should receive the result
    - An instruction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

Von Neumann vs Dataflow

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

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

Sequential



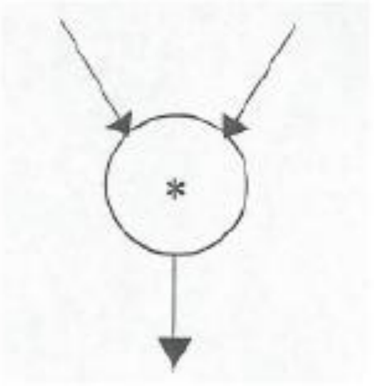
Dataflow

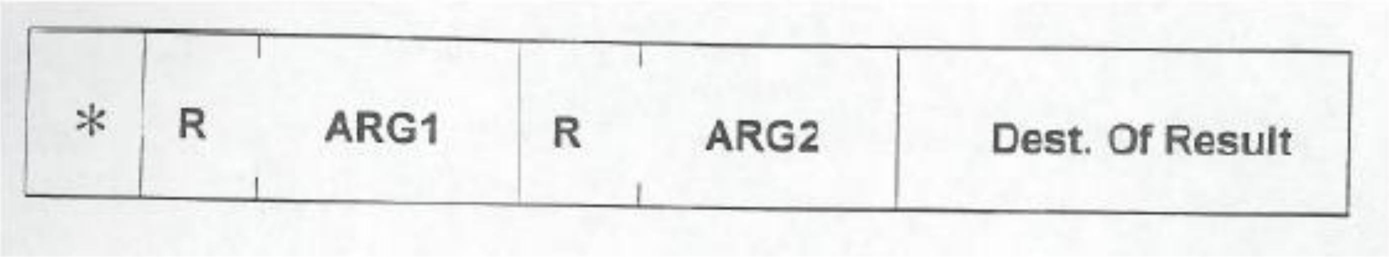
- Which model is more natural to you as a programmer?

More on Data Flow

- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all it inputs are ready
    - i.e. when all inputs have tokens

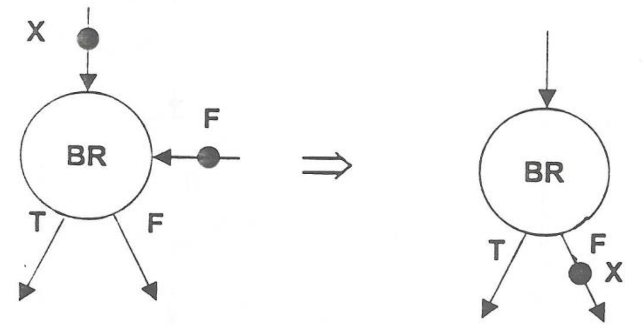
- Data flow node and its ISA representation



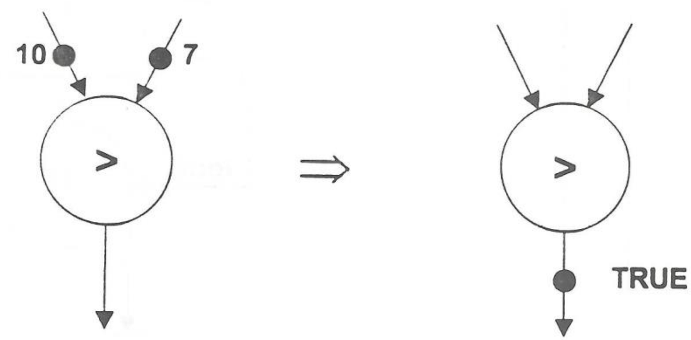


Data Flow Nodes

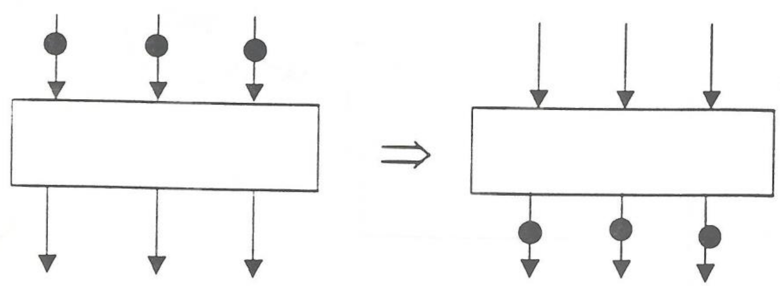
*Conditional



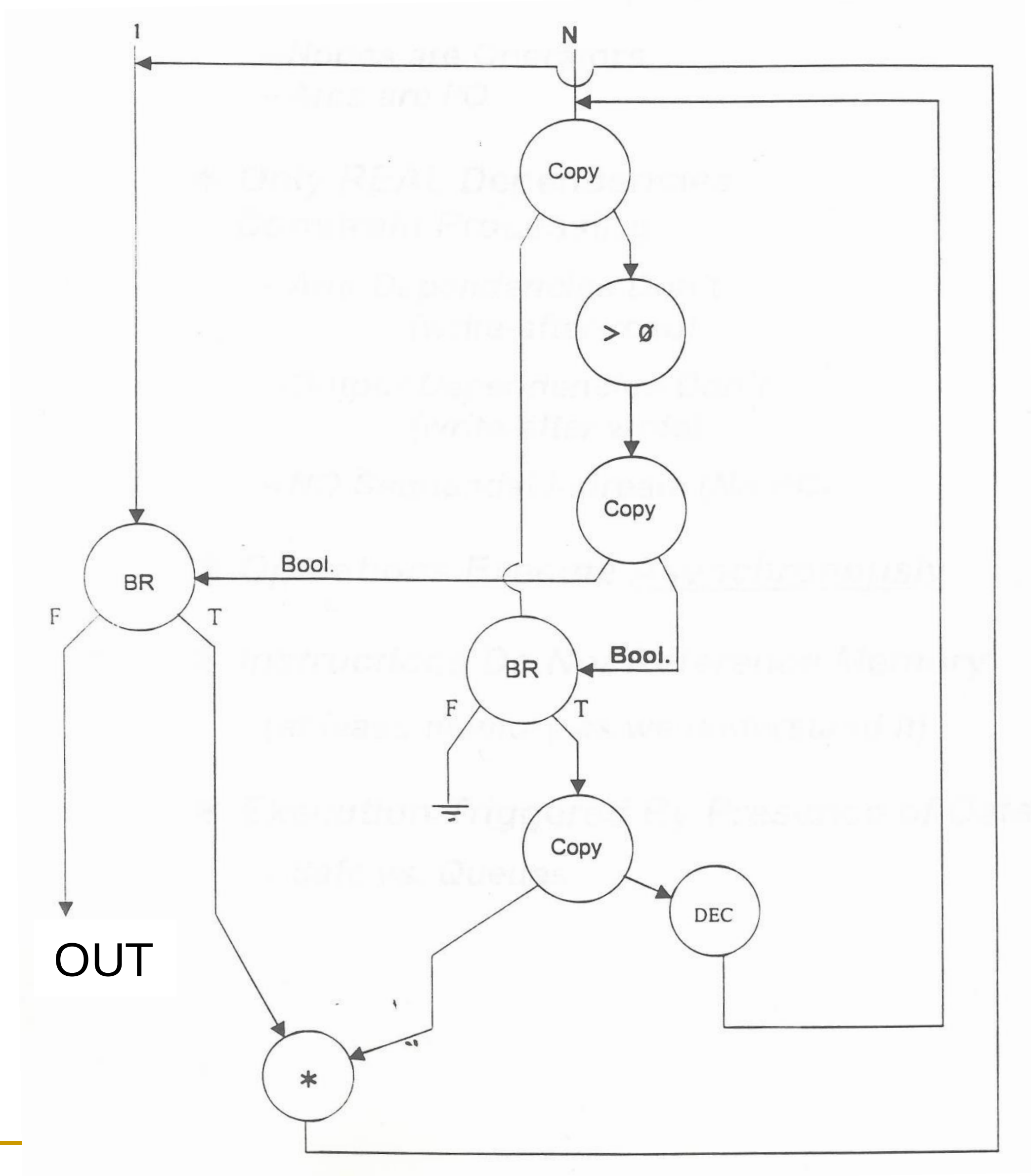
*Relational



*Barrier Synch



An Example Data Flow Program



ISA-level Tradeoff: Instruction Pointer

- Do we need an instruction pointer in the ISA?
  - Yes: Control-driven, sequential execution
    - An instruction is executed when the IP points to it
    - IP automatically changes sequentially (except for control flow instructions)
  - No: Data-driven, parallel execution
    - An instruction is executed when all its operand values are available (data flow)

- 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 data-flow execution order

- Microarchitecture: How the **underlying implementation actually executes** instructions
  - Microarchitecture can execute instructions in any order as long as it obeys the semantics specified by the ISA when making the instruction results visible to software
    - Programmer should see the order specified by the ISA

Let's Get Back to the Von Neumann Model

- But, if you want to learn more about dataflow…

- Dennis and Misunas, "A preliminary architecture for a basic data-flow processor," ISCA 1974.
- Gurd et al., "The Manchester prototype dataflow computer," CACM 1985.
- A later lecture

- If you are really impatient:
  - http://www.youtube.com/watch?v=D2uue7izU2c
  - http://www.ece.cmu.edu/~ece740/f13/lib/exe/fetch.php?media=onur-740-fall13-module5.2.1-dataflow-part1.ppt

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

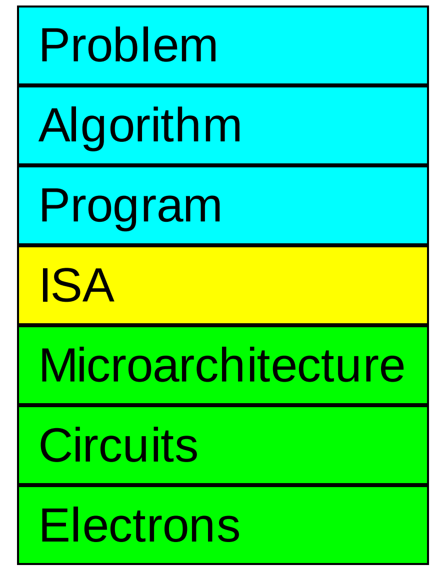
What is Computer Architecture?

- **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: 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core, Kaby Lake, Coffee Lake, …

- 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®

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?

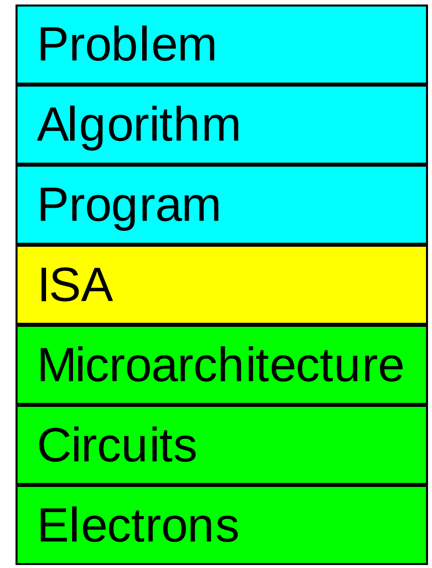
Property of ISA vs. Uarch?

- 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*

Application Space

Dream, and they will appear…

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

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

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

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

Many other workloads:

Genome analysis

Machine learning

Robotics

Web search

Graph analytics

…

Increasingly Demanding Applications

Dream

and, they will come

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

Tradeoffs: Soul of Computer Architecture

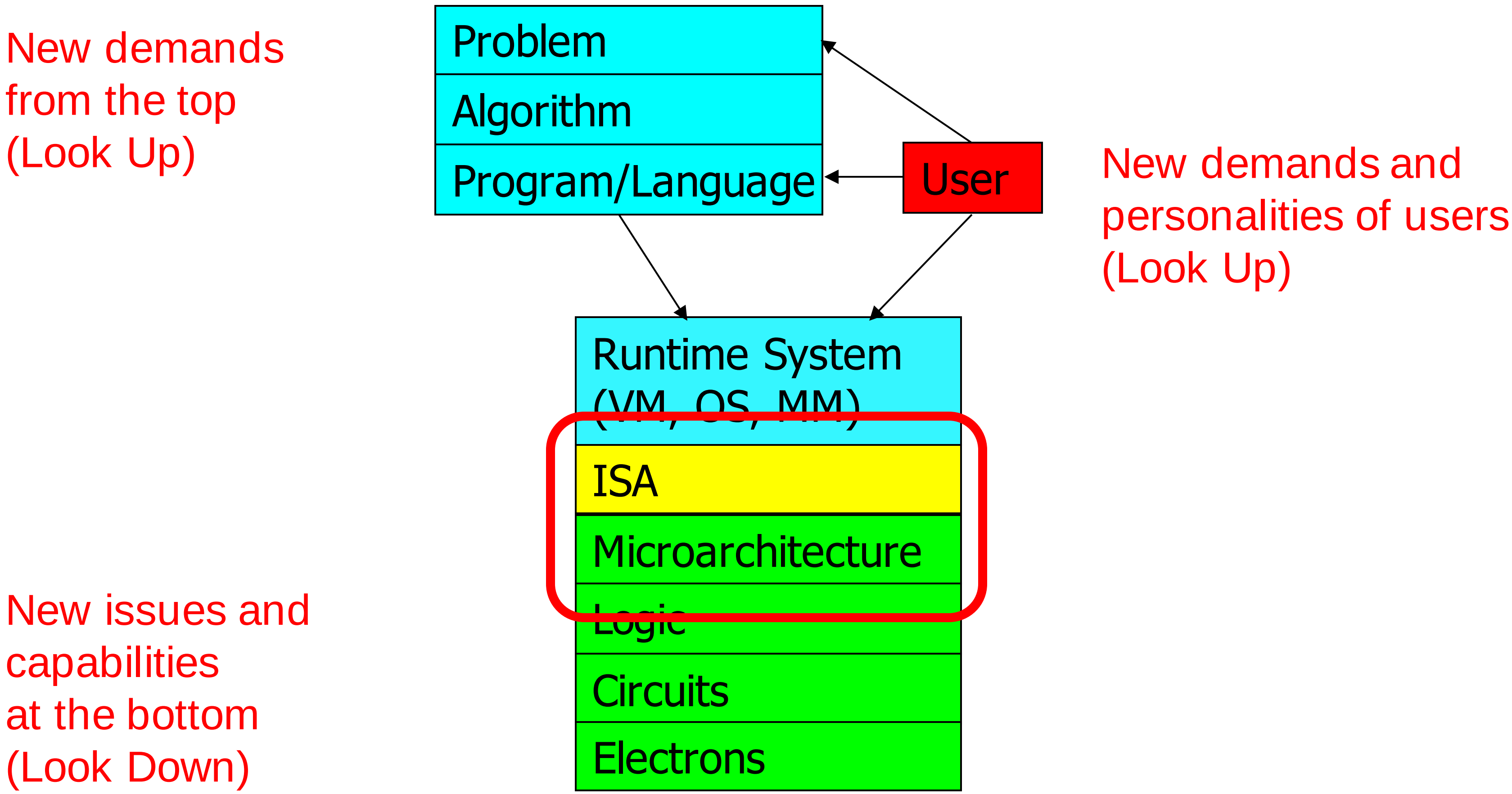
- ISA-level tradeoffs

- Microarchitecture-level tradeoffs

- System and Task-level tradeoffs
  - How to divide the labor between hardware and software

- *Computer architecture is the science and art of making the appropriate trade-offs to meet a design point*
  - *Why **art**?*

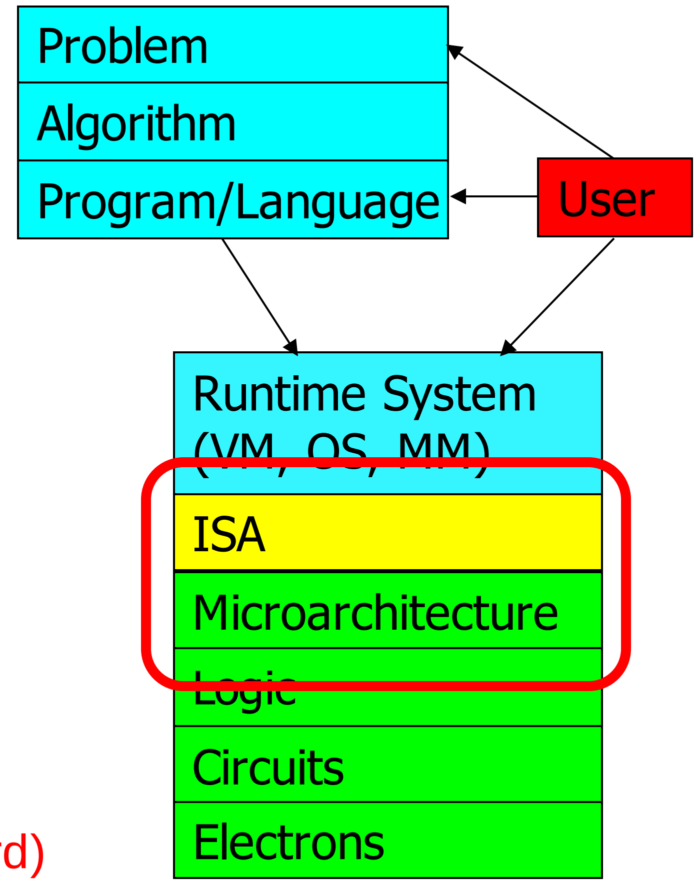
Why Is It (Somewhat) Art?



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

Why Is It (Somewhat) Art?

Changing demands at the top (Look Up and Forward)



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

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

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

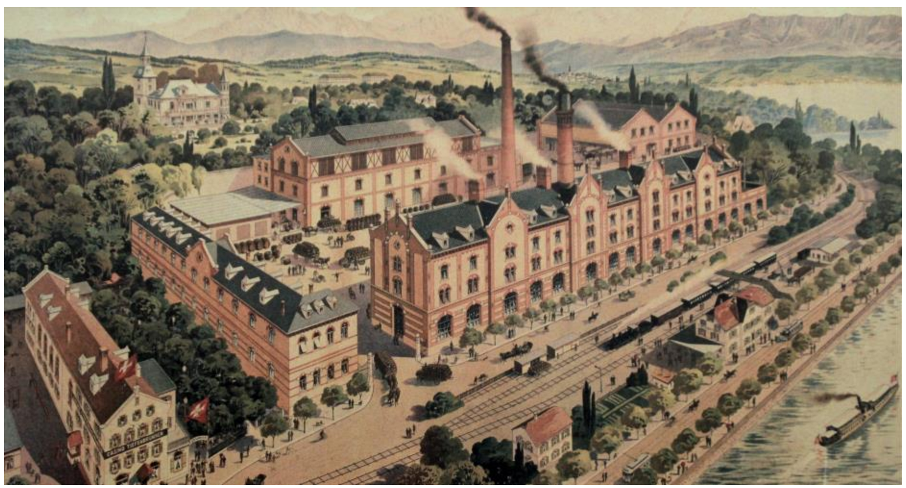
Analogue from Macro-Architecture

- Future is not constant in macro-architecture, either

- Example: Can a mill be later used as a theater + restaurant + conference room?

Mühle Tiefenbrunnen

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



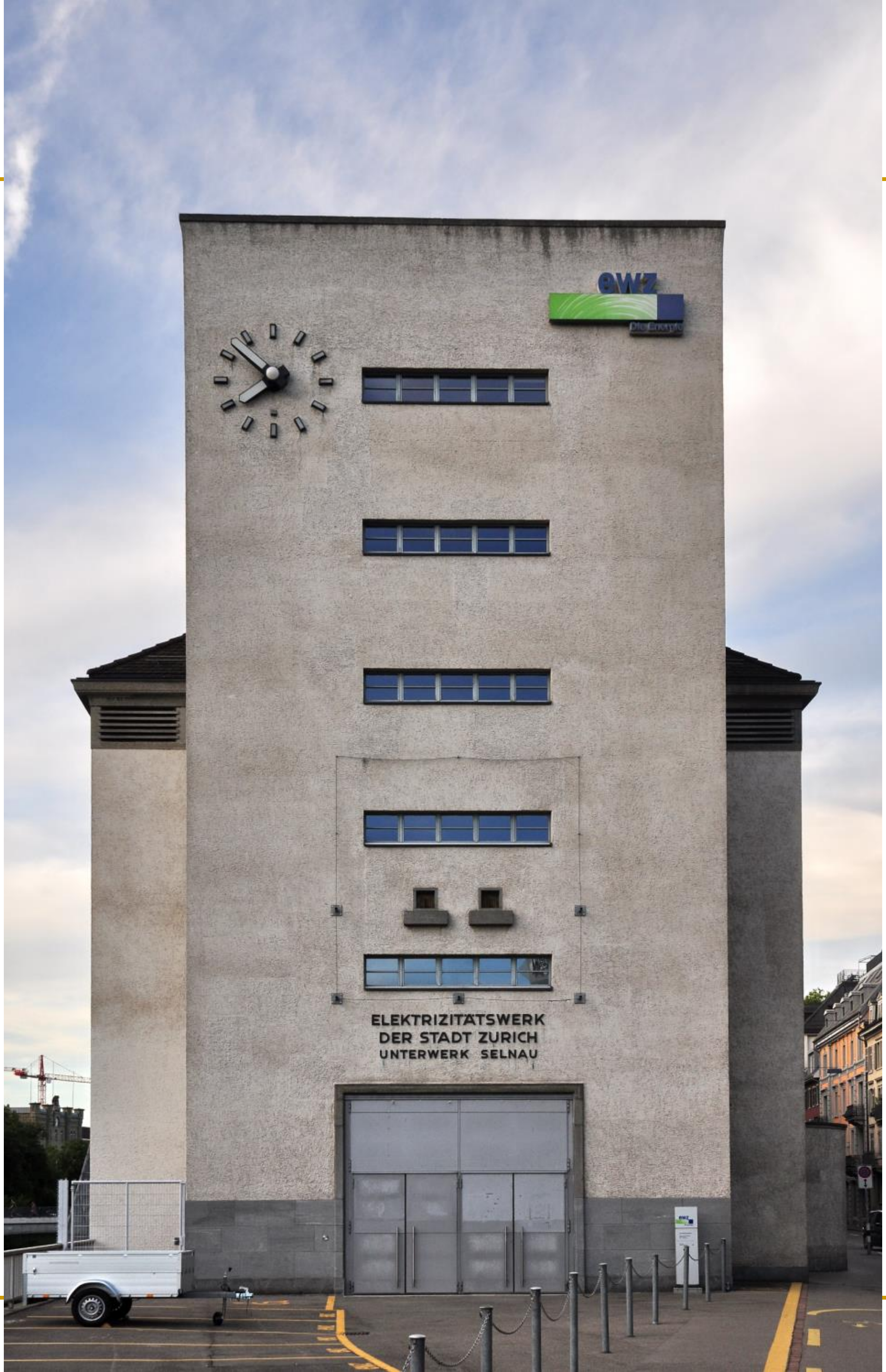
Brewery in 1900

Another Example (I)

Photo credit: Prof. Can Alkan

Another Example (II)

Photo credit: Prof. Can Alkan


ewz
ELEKTRIZITÄTSWERK
DER STADT ZURICH
UNTERWERK SELNAU

Implementing the ISA: Microarchitecture Basics

Now That We Have an ISA

- How do we implement it?

- i.e., how do we design a system that obeys the hardware/software interface?

- Aside: "System" can be solely hardware or a combination of hardware and software
  - "Translation of ISAs"
  - A **virtual ISA** can be converted by "software" into an **implementation ISA**

- We will assume "hardware" implementation for most lectures

How Does a Machine Process Instructions?

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

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

↓

Process instruction

↓

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

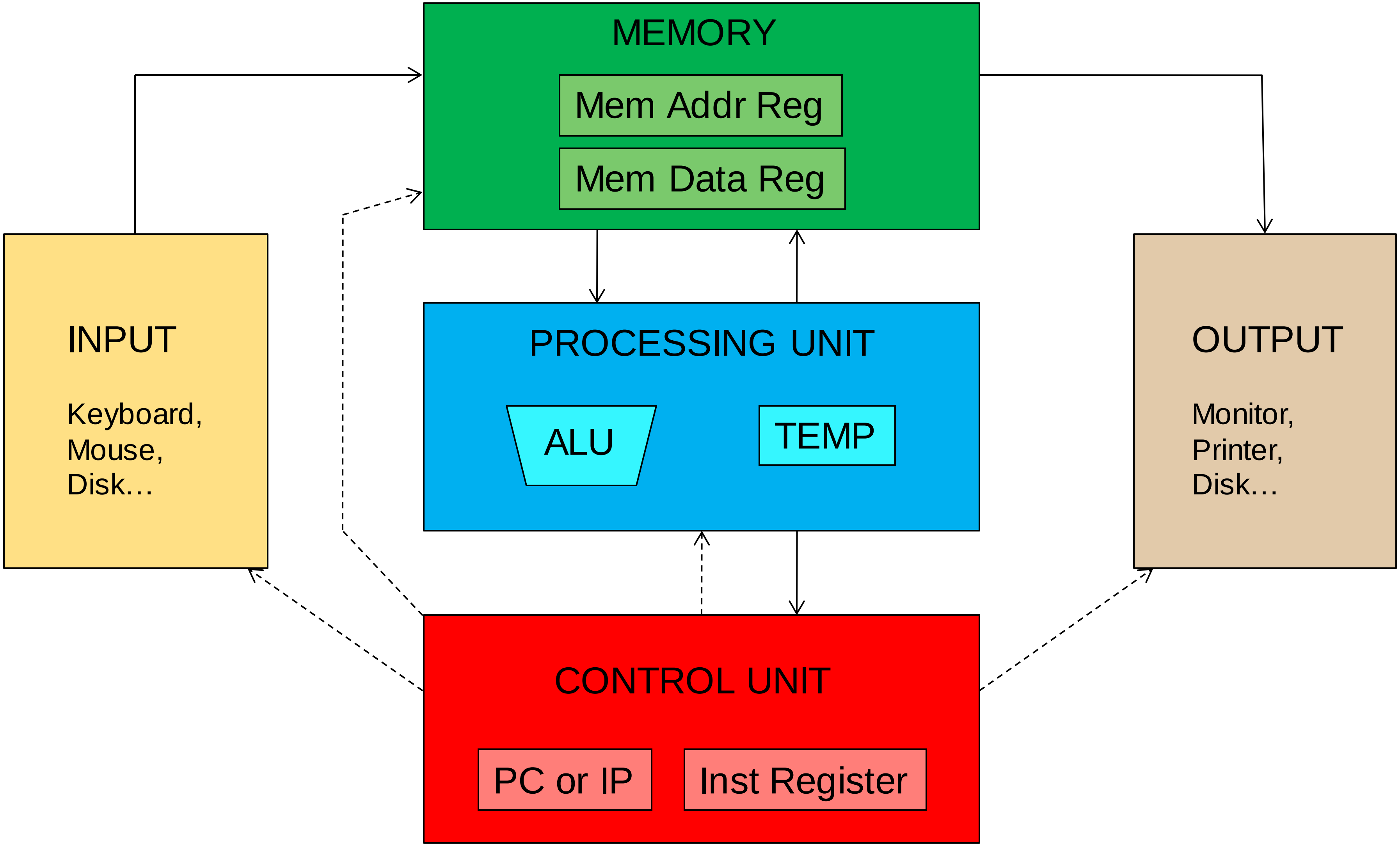
- Processing an instruction: Transforming AS to AS' according to the ISA specification of the instruction

The Von Neumann Model/Architecture

Stored program

Sequential instruction processing

Recall: The Von Neumann Model



The "Process Instruction" Step

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

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

A Very Basic Instruction Processing Engine

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

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

↓

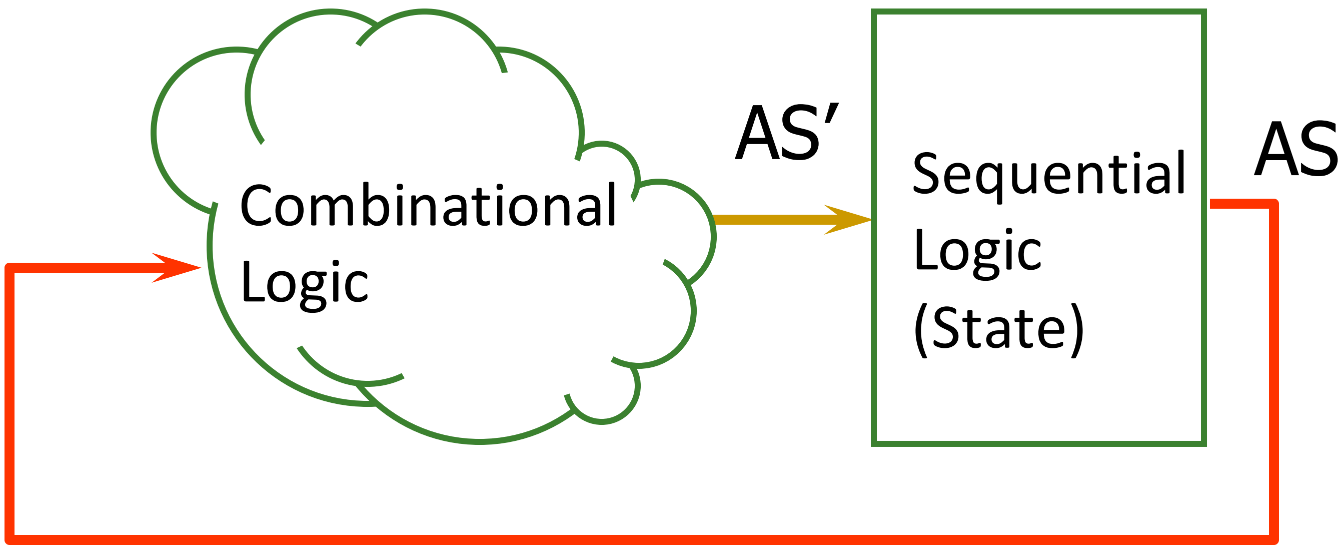
Process instruction in one clock cycle

↓

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

A Very Basic Instruction Processing Engine

- Single-cycle machine



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

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

memory address
of the current instruction

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

Single-cycle vs. Multi-cycle Machines

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

- Multi-cycle machines
  - Instruction processing broken into multiple cycles/stages
  - State updates can be made during an instruction's execution
  - Architectural state updates made 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"

- FETCH
- DECODE
- EVALUATE ADDRESS
- FETCH OPERANDS
- EXECUTE
- STORE RESULT

Instruction Processing "Cycle" vs. Machine Clock Cycle

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

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

Instruction Processing Viewed Another Way

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

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

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

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

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

- 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 data path and control logic

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

- Control signals and structure depend on the datapath design

Flash-Forward: Performance Analysis

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

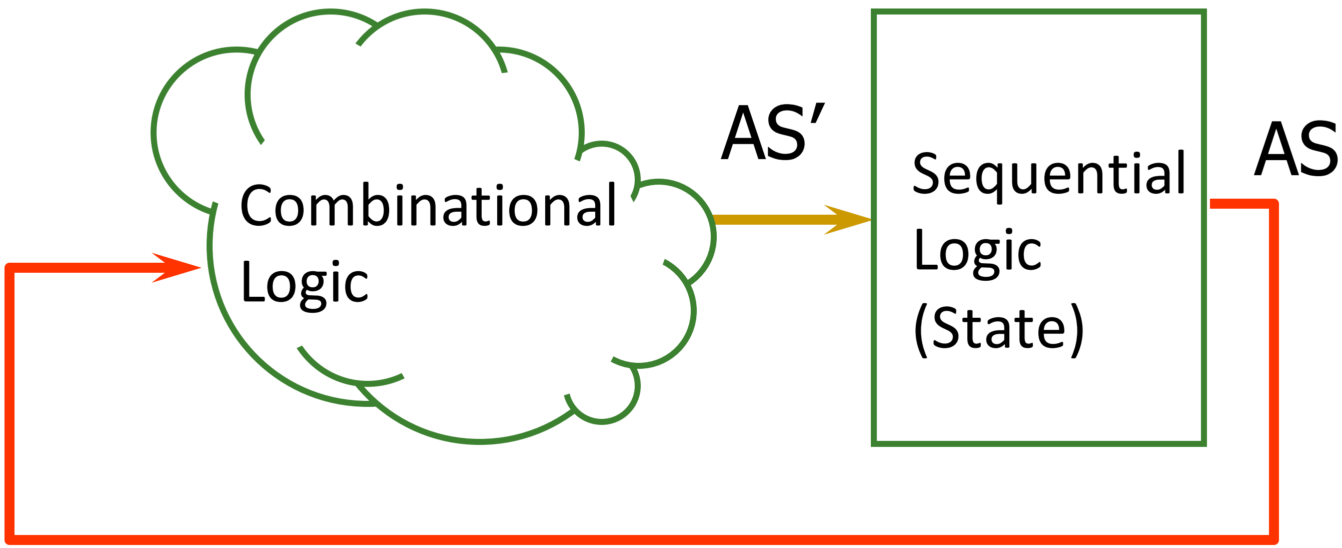
- Single-cycle microarchitecture performance
  - CPI = 1
  - Clock cycle time = long
- Multi-cycle microarchitecture performance
  - CPI = different for each instruction
    - Average CPI → hopefully small
  - Clock cycle time = short

Here, we have two degrees of freedom to optimize independently

A Single-Cycle Microarchitecture
A Closer Look

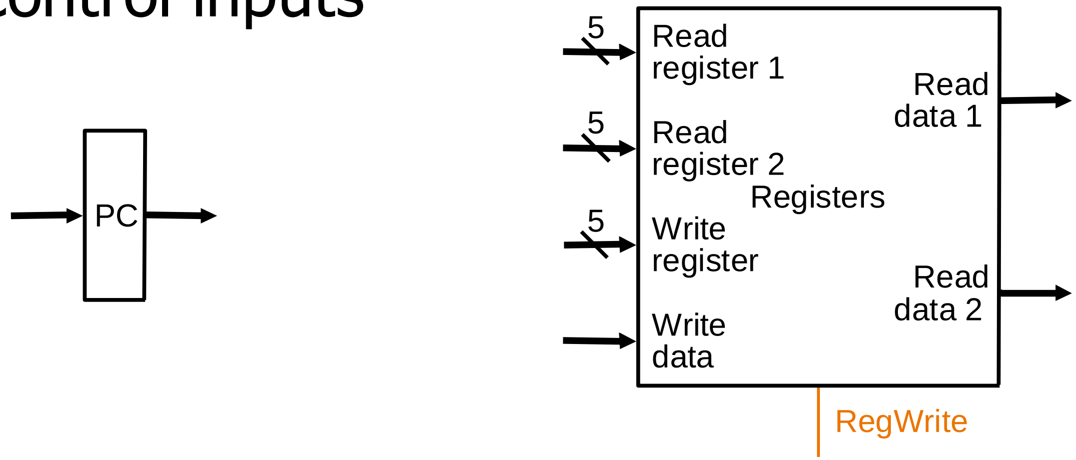
Remember…

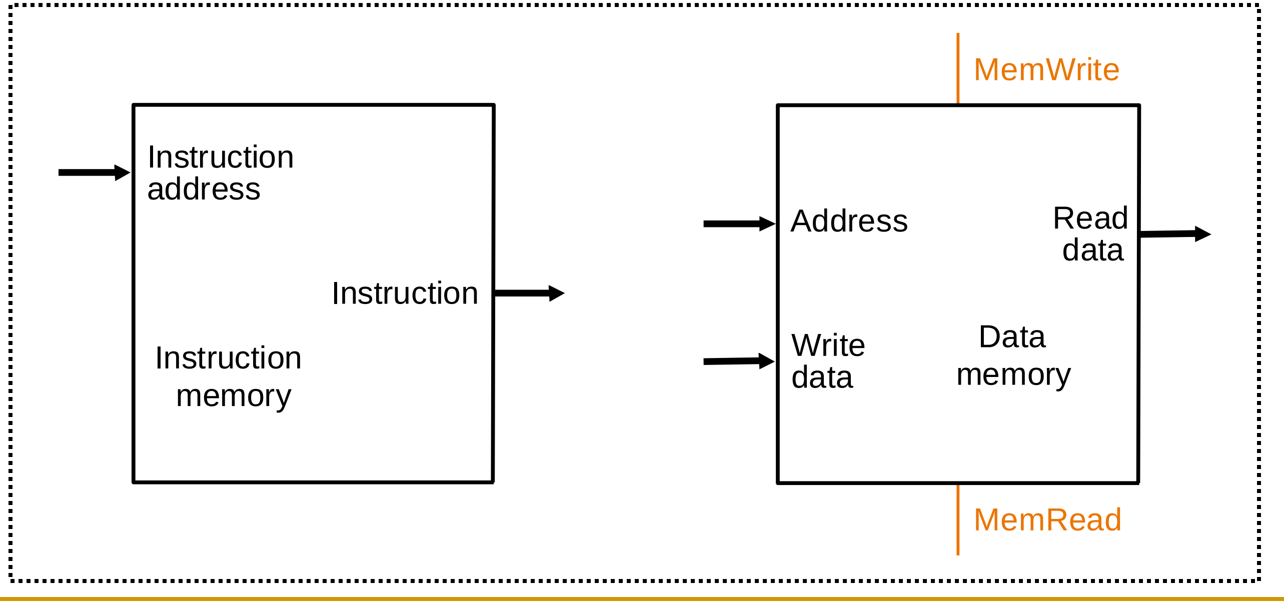
- Single-cycle machine



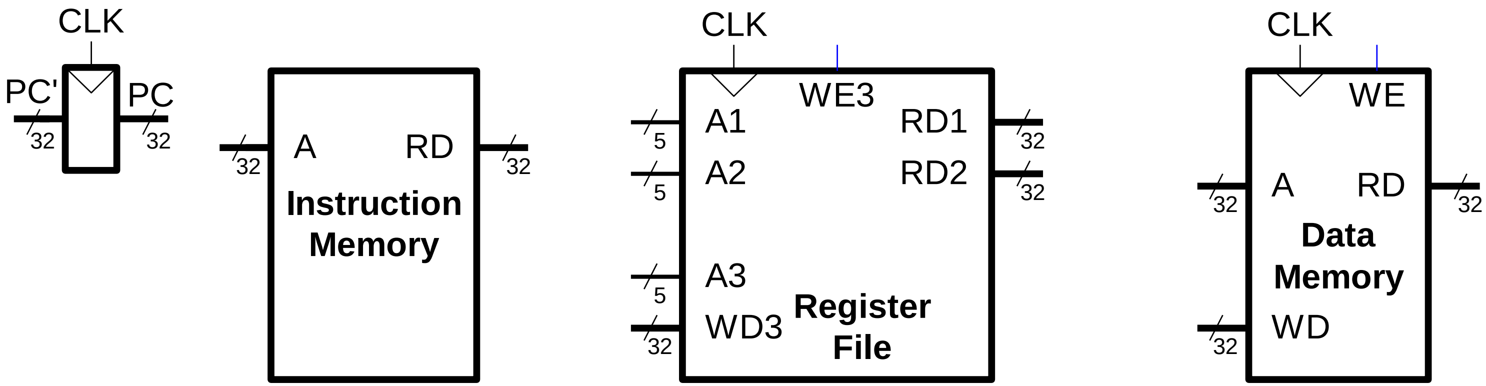
Let's Start with the State Elements

- Data and control inputs





MIPS State Elements



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

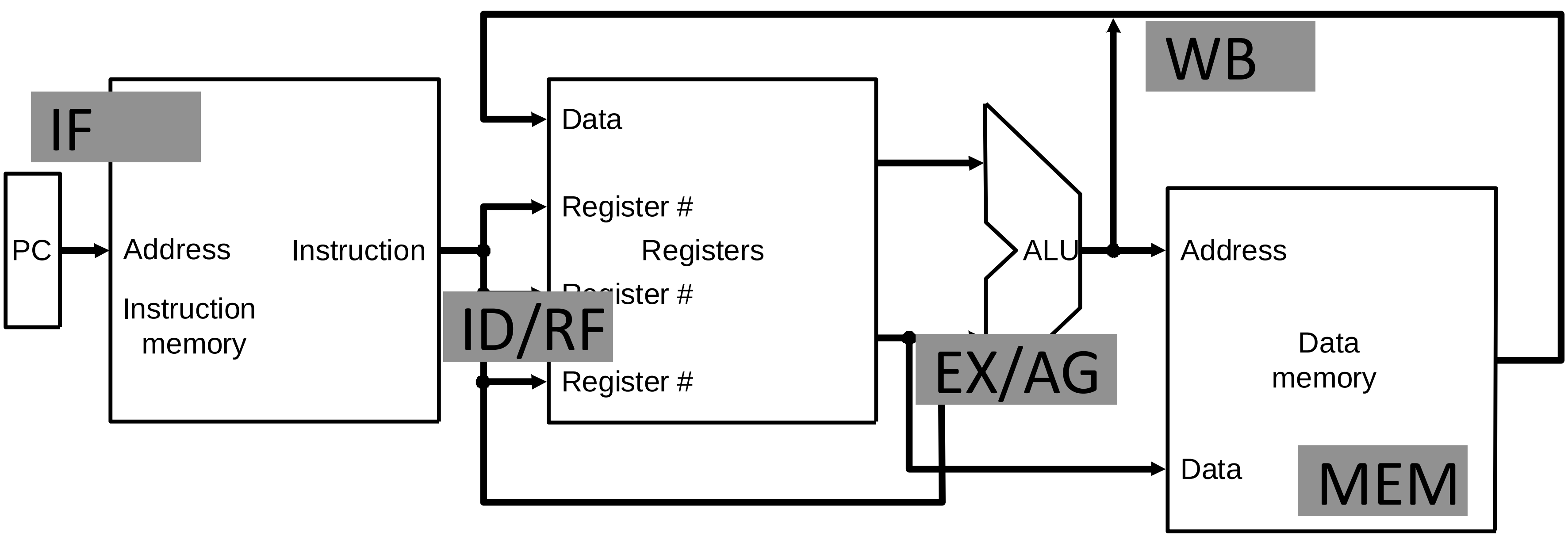
This notation is used in H&H single-cycle MIPS implementation (H&H Chapter 7.3)

For Now, We Will Assume

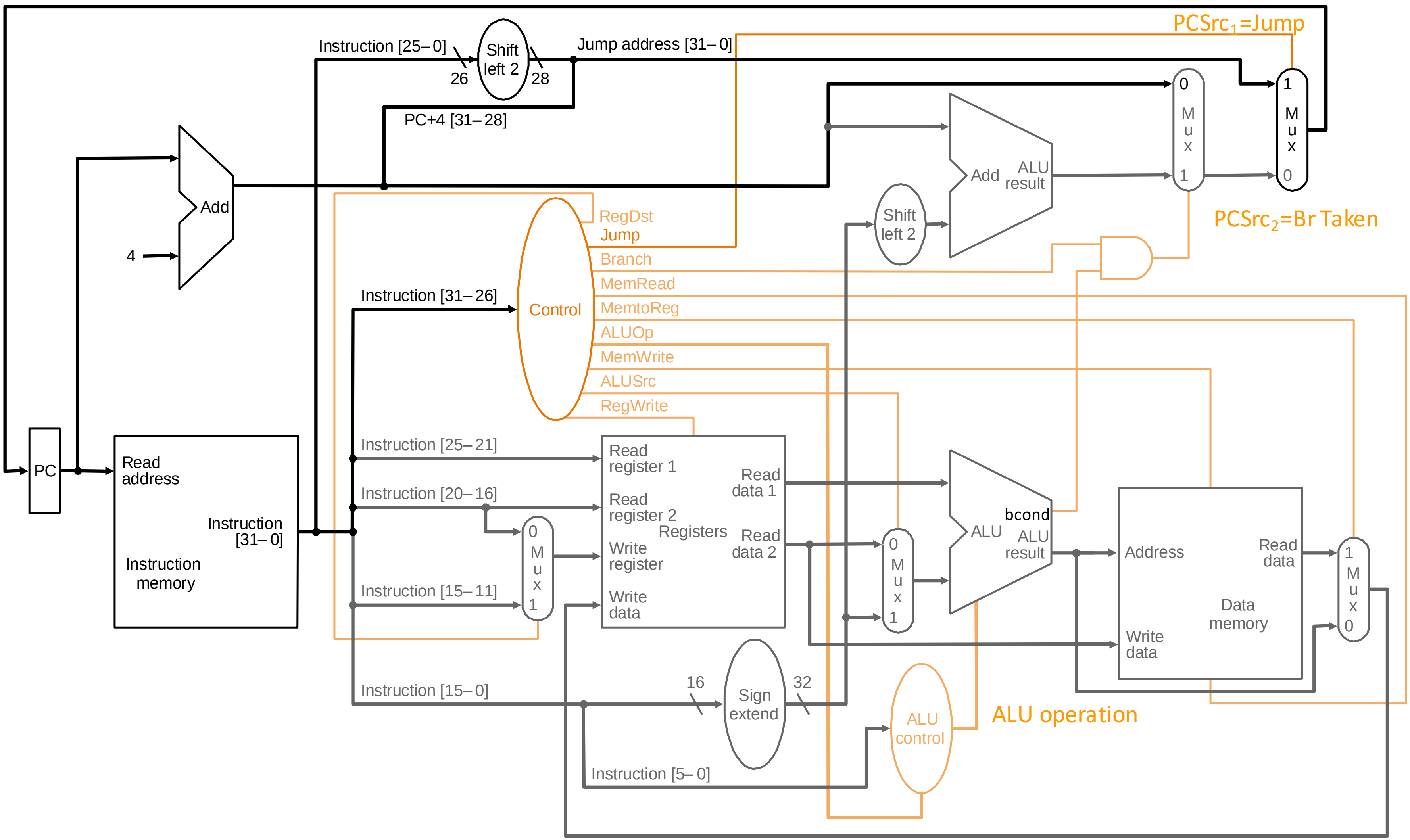
- "Magic" memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is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



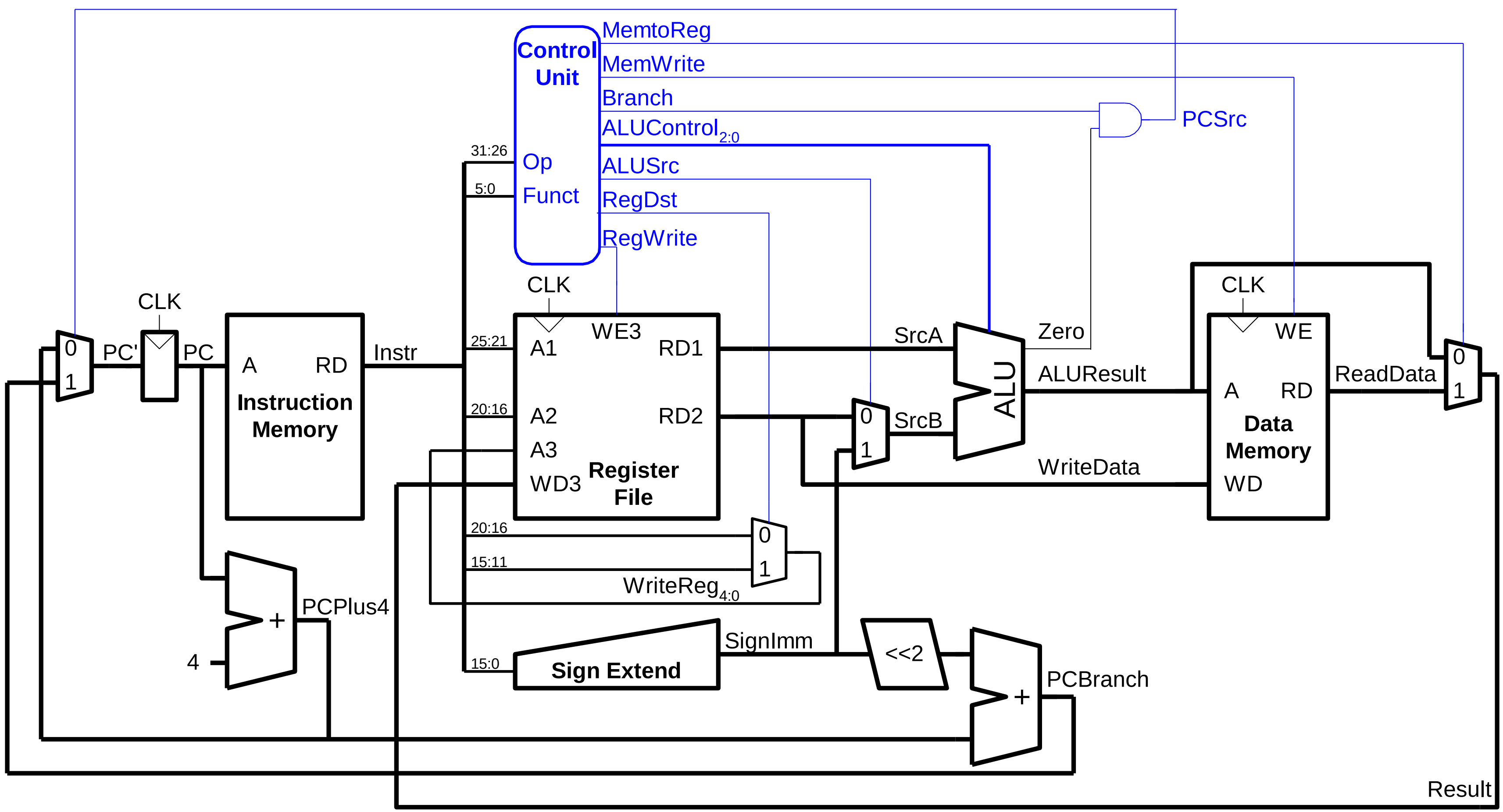
What Is To Come: The Full MIPS Datapath



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

JAL, JR, JALR omitted

Another Complete Single-Cycle Processor



Single-Cycle Datapath for *Arithmetic and Logical Instructions*

R-Type ALU Instructions

- R-type: 3 register operands

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

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

Machine Encoding

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

R-Type

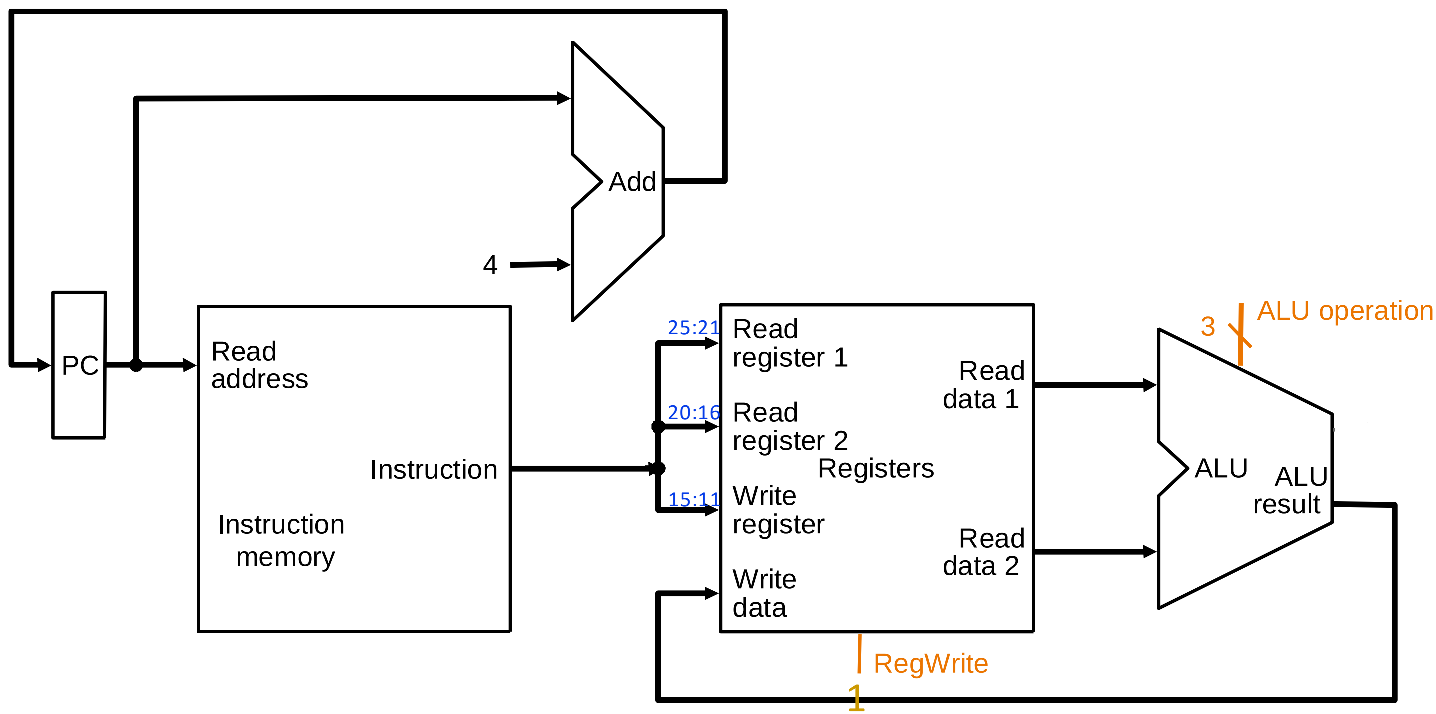
- Semantics

if MEM[PC] == add rd rs rt

GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4

(R-Type) ALU Datapath



if MEM[PC] == ADD rd rs rt

GPR[rd] ← GPR[rs] + GPR[rt]

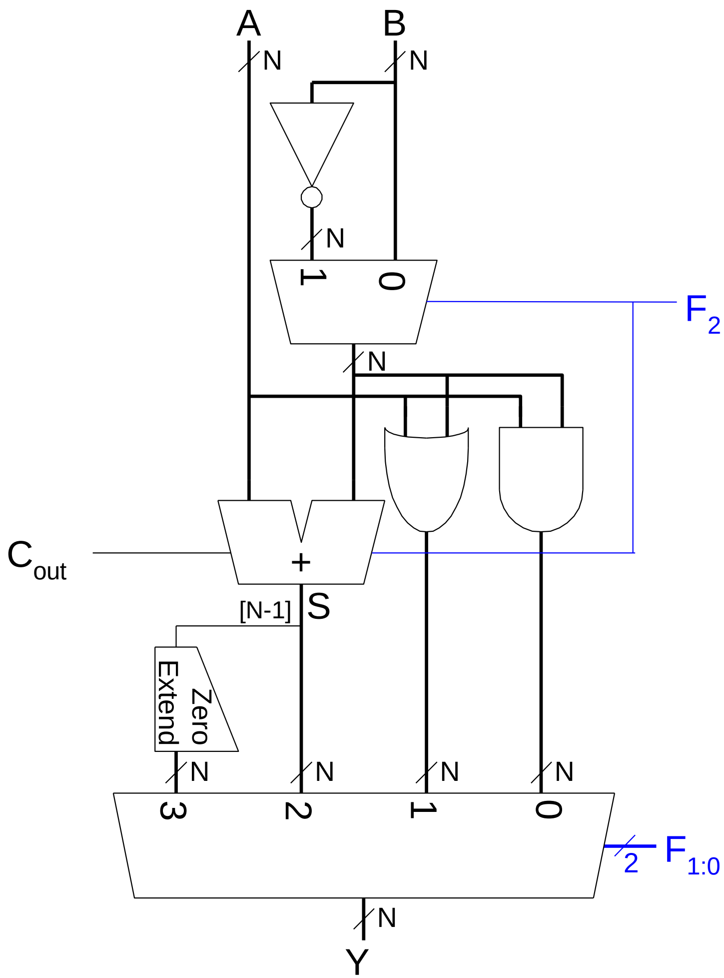
PC ← PC + 4

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 & ~B
101	A | ~B
110	A - B
111	SLT

I-Type ALU Instructions

- I-type: 2 register operands and 1 immediate

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

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

Machine Encoding

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

I-Type

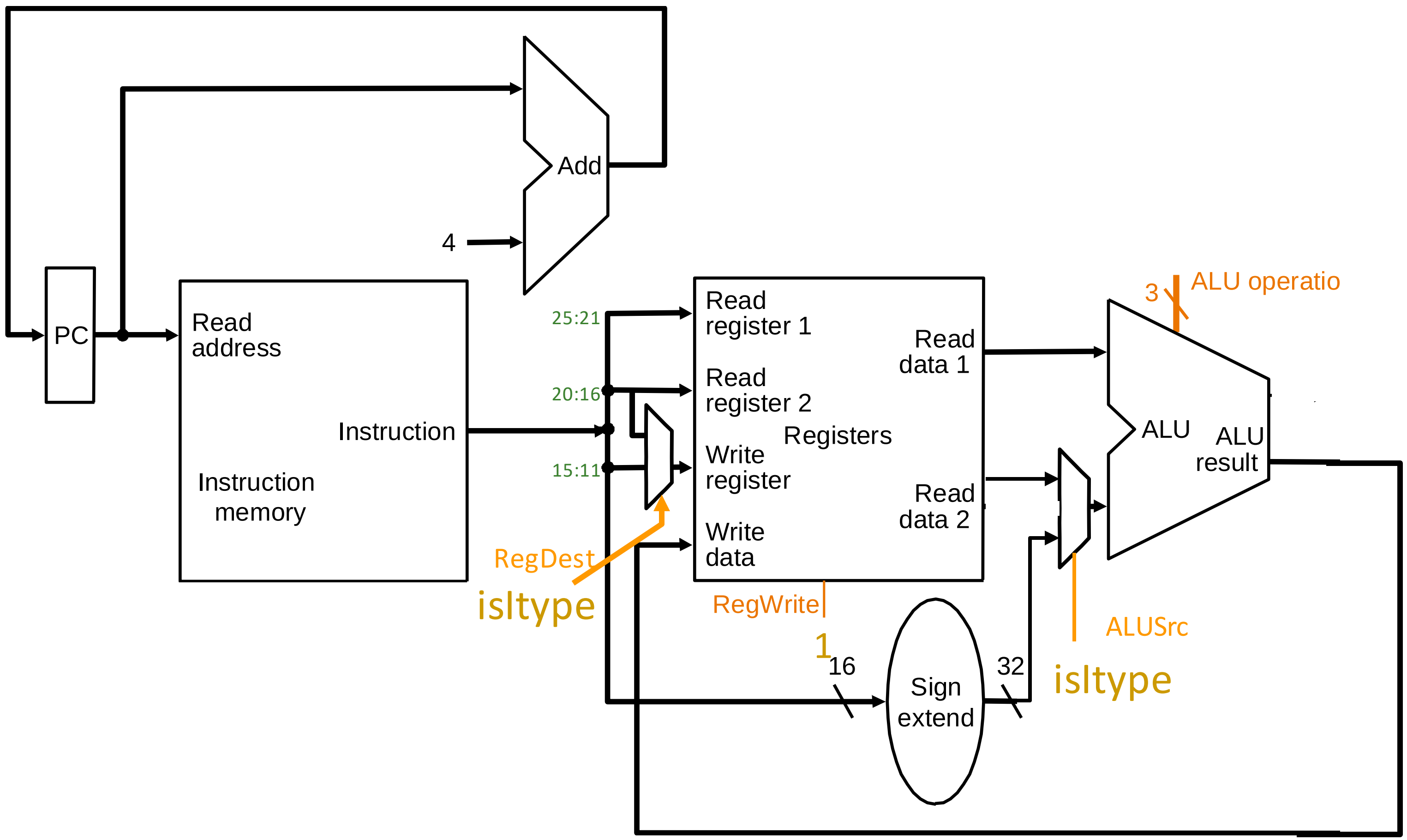
- Semantics

if MEM[PC] == addi rs rt immediate

PC $\leftarrow$ PC + 4

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

Datapath for R and I-Type ALU Insts.



IF	ID	EX	MEM	WB

if MEM[PC] == ADDI rt rs immediate

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

PC ← PC + 4

Combinational state update logic

Recall: ADD with one Literal in LC-3

- ADD assembly and machine code

LC-3 assembly

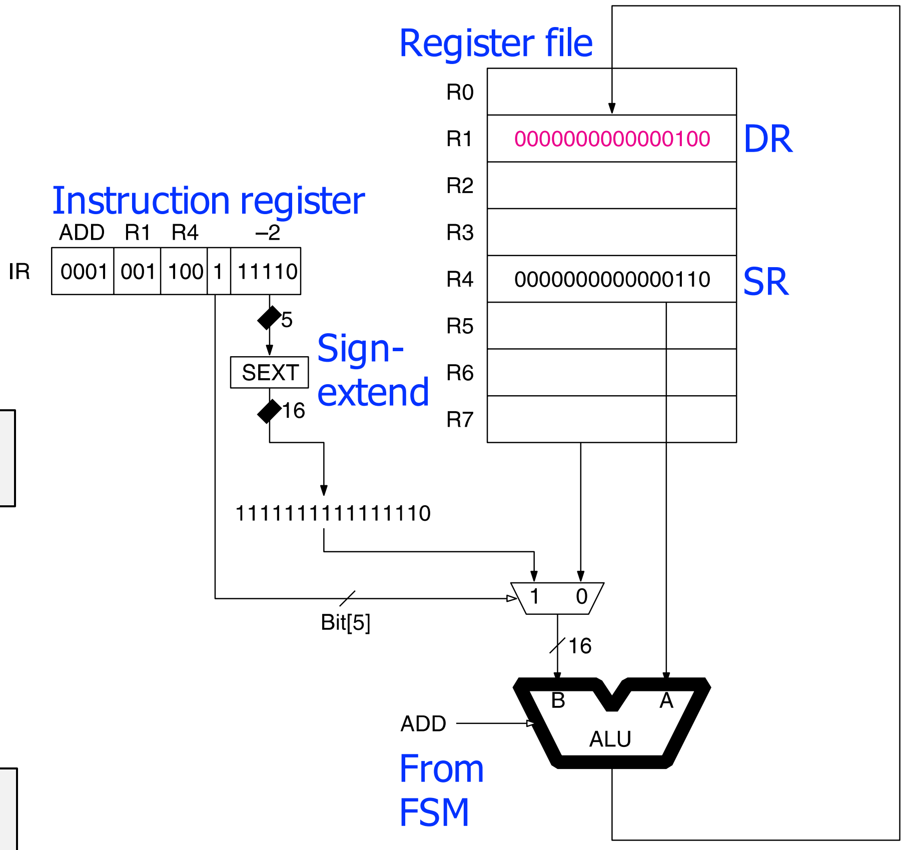
```
ADD R1, R4, #-2
```

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	11 9	8 6	5	4 0



Single-Cycle Datapath for *Data Movement Instructions*

Load Instructions

- Load 4-byte word

MIPS assembly

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

Machine Encoding

op	rs=base	rt	imm=offset
lw (35)	base	rt	offset
31 26	25 21	20 16	15 0

I-Type

- Semantics

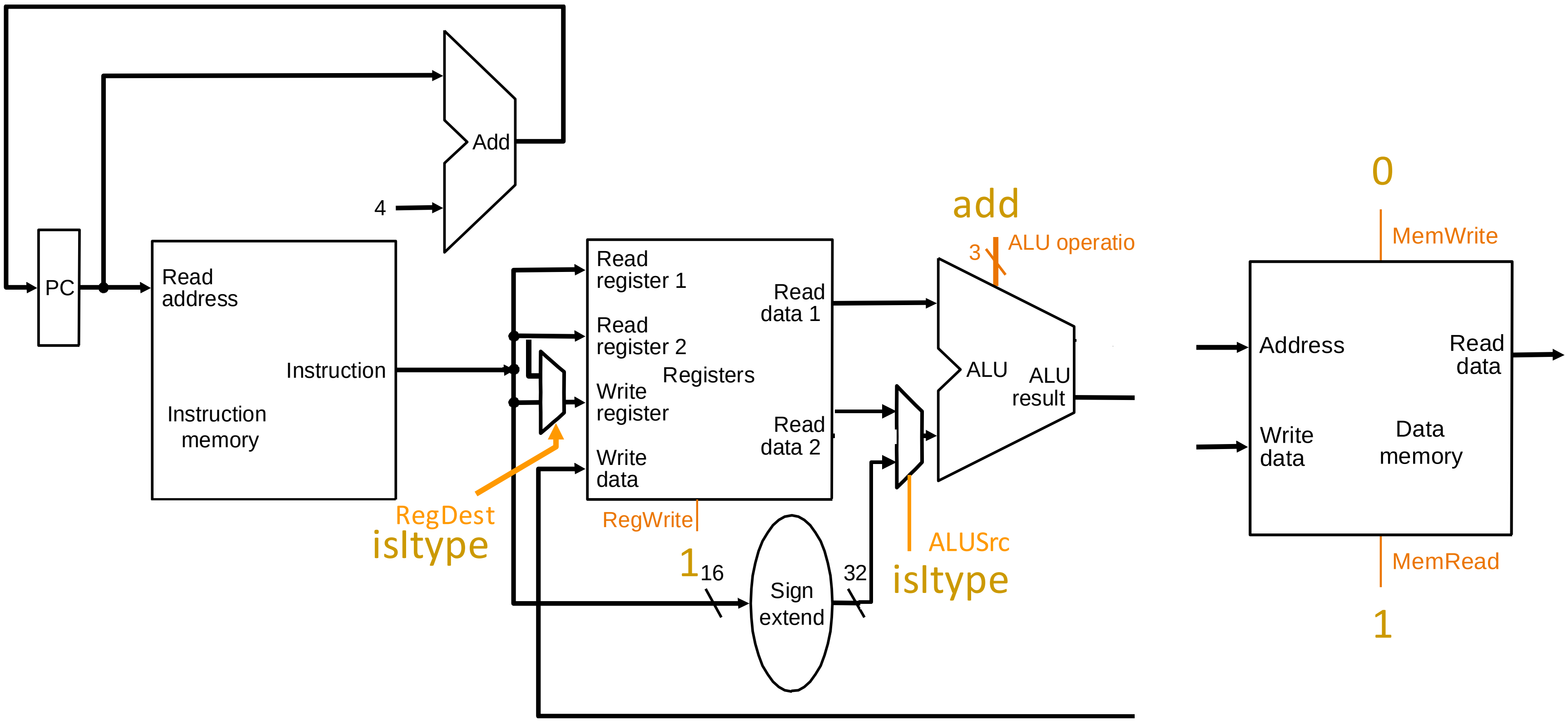
if MEM[PC] == lw rt $offset_{16}$ (base)

PC ← PC + 4

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

GPR[rt] ← MEM[translate(EA)]

LW Datapath



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

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

GPR[rt] ← MEM[translate(EA)]

PC ← PC + 4

IF	ID	EX	MEM	WB

Combinational state update logic

Store Instructions

- 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
31 26	25 21	20 16	15 0

I-Type

- Semantics

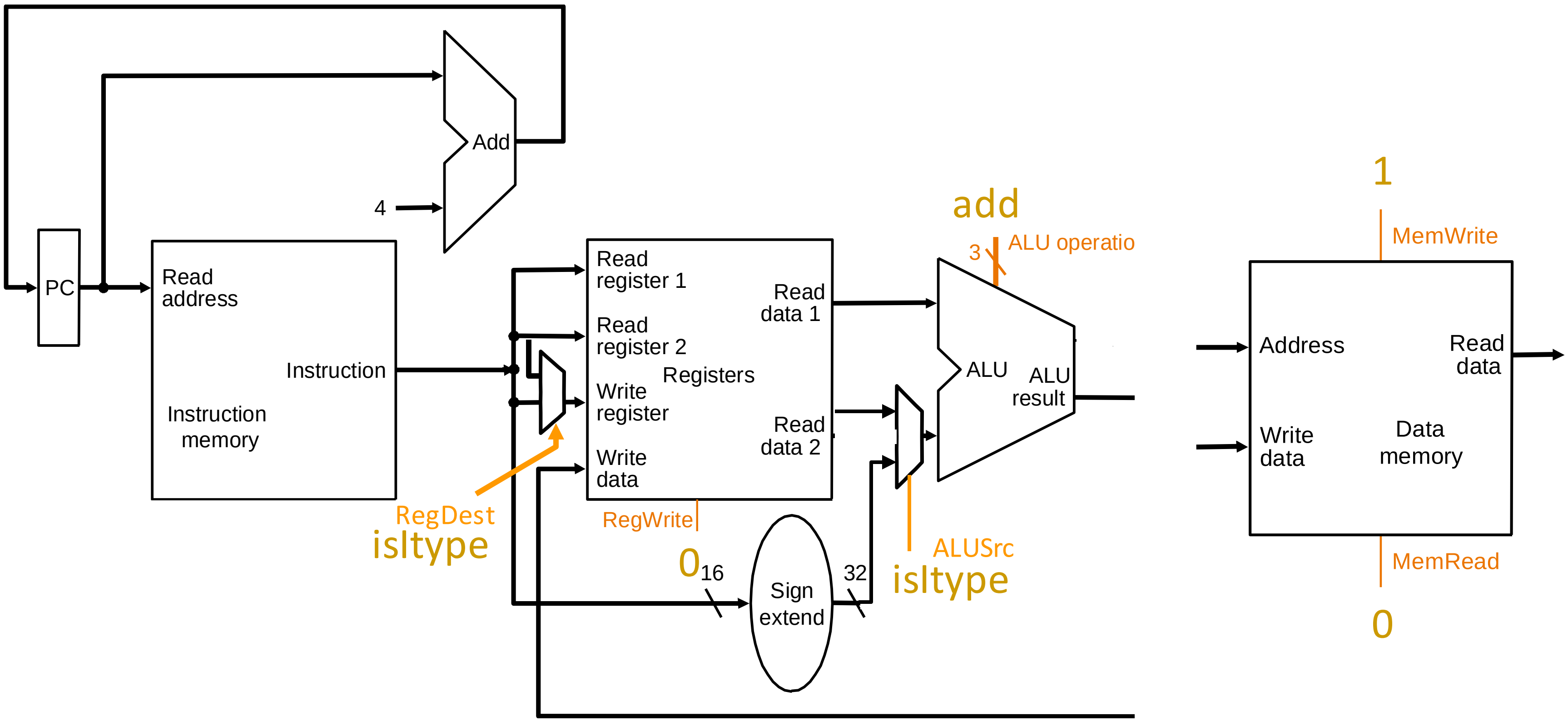
if Mem[PC] == sw rt offset$_{16}$ (base)

PC $\leftarrow$ PC + 4

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

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

SW Datapath



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

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

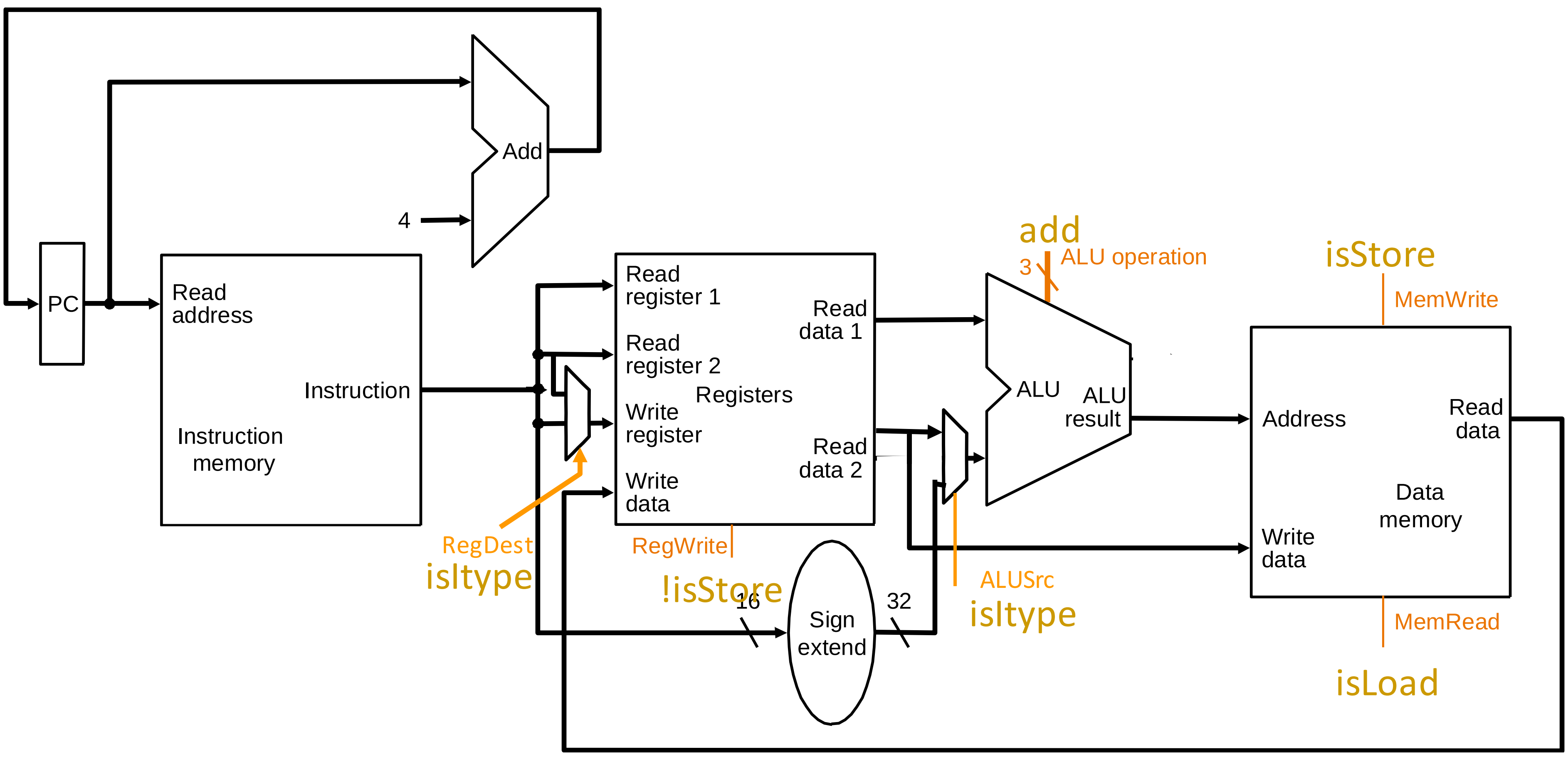
MEM[translate(EA)] ← GPR[rt]

PC ← PC + 4

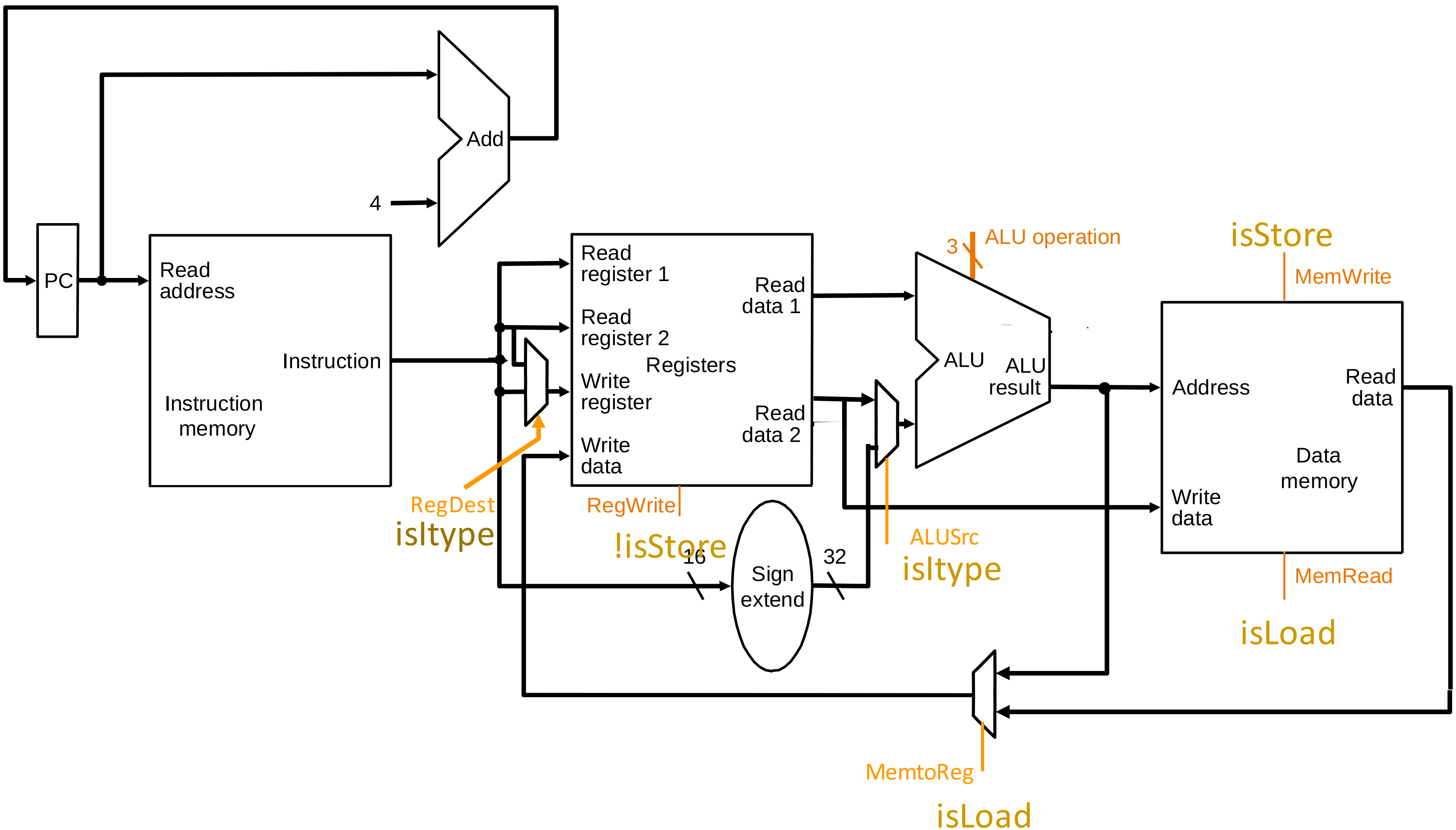
IF	ID	EX	MEM	WB

Combinational state update logic

Load-Store Datapath



Datapath for Non-Control-Flow Insts.



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

Single-Cycle Datapath for *Control Flow Instructions*

Jump Instruction

- Unconditional branch or jump

```
j   target
```

j (2)	immediate
6 bits	26 bits

J-Type

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

- Semantics

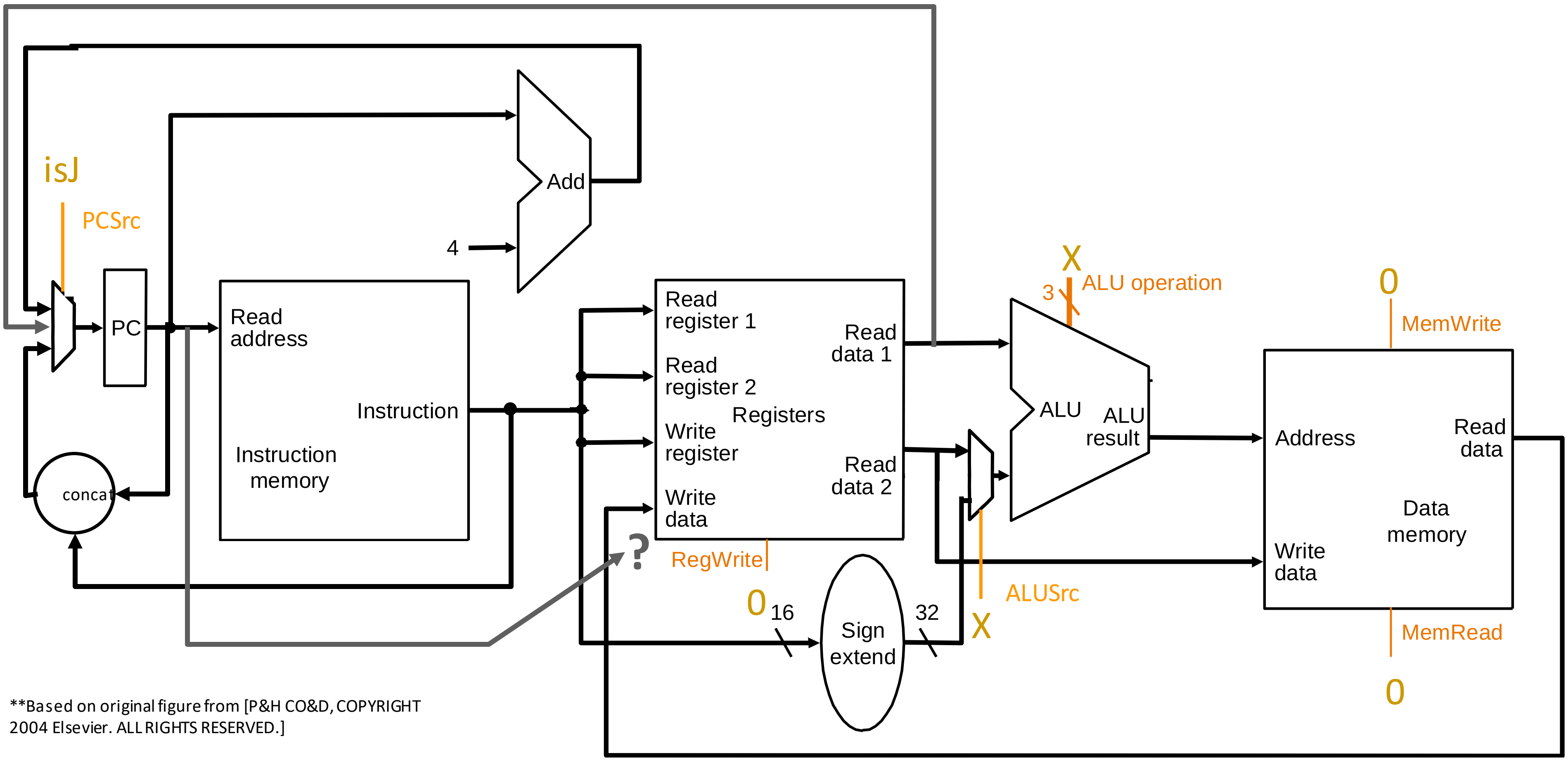
if MEM[PC]== j immediate_{26}

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

PC ← 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 MEM[PC]==J immediate26
PC = { PC[31:28], immediate26, 2ʼ b00 }

What about JR, JAL, JALR?

Other Jumps in MIPS

- jal: jump and link (function calls)
  - Semantics

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

$ra $\leftarrow$ PC + 4

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

PC $\leftarrow$ target

- jr: jump register
  - Semantics

if MEM[PC]== jr rs

PC $\leftarrow$ GPR(rs)

- jalr: jump and link register
  - Semantics

if MEM[PC]== jalr rs

$ra $\leftarrow$ PC + 4

PC $\leftarrow$ GPR(rs)

[†] This is the incremented PC

Aside: MIPS Cheat Sheet

- https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=mips_reference_data.pdf

- On the course website

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

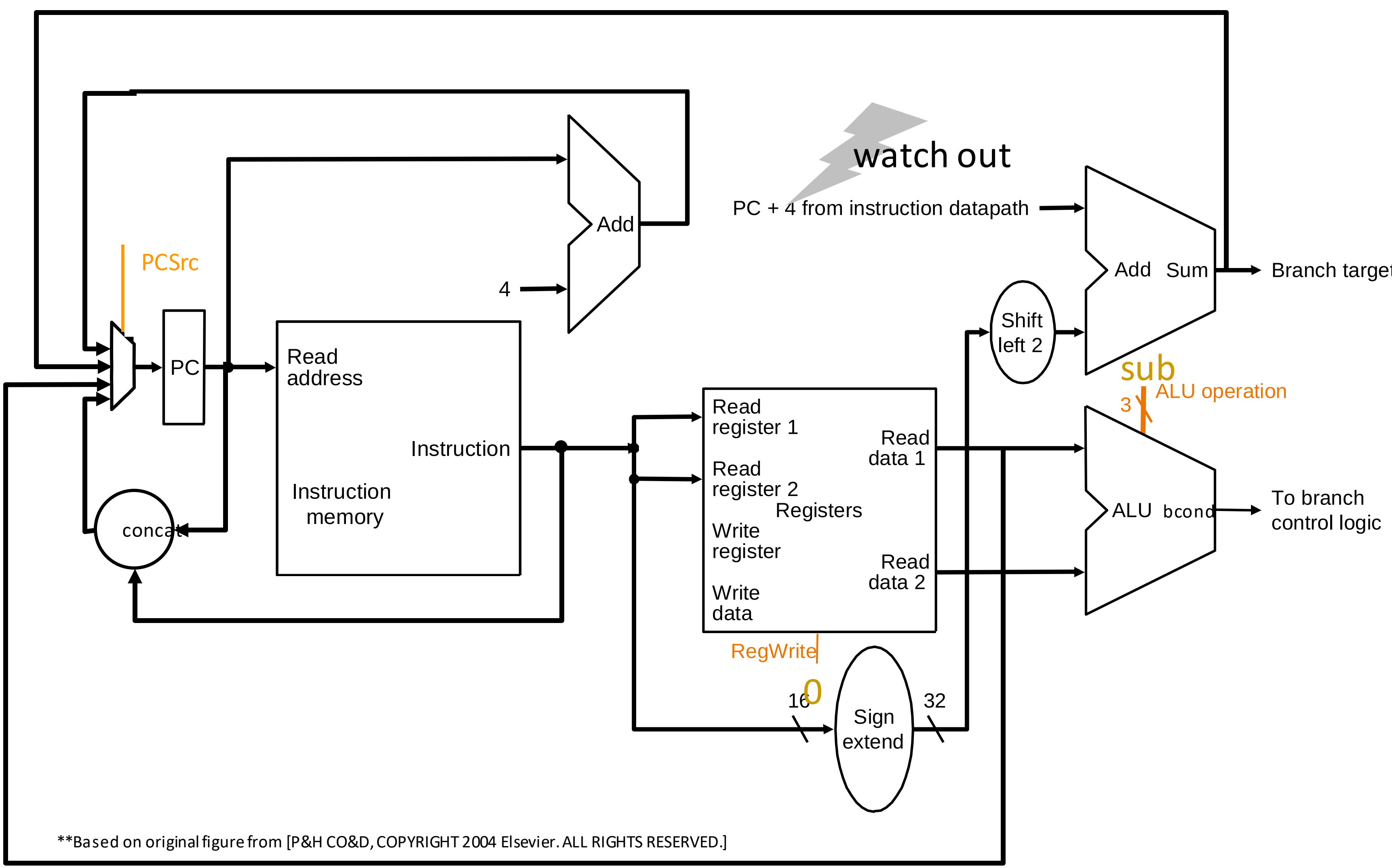
- Semantics (assuming no branch delay slot)

if MEM[PC] == beq rs rt $immediate_{16}$
  target = PC[†] + sign-extend(immediate) x 4
  if GPR[rs]==GPR[rt] then PC ← target
  else PC ← PC + 4

  - Variations: beq, bne, blez, bgtz

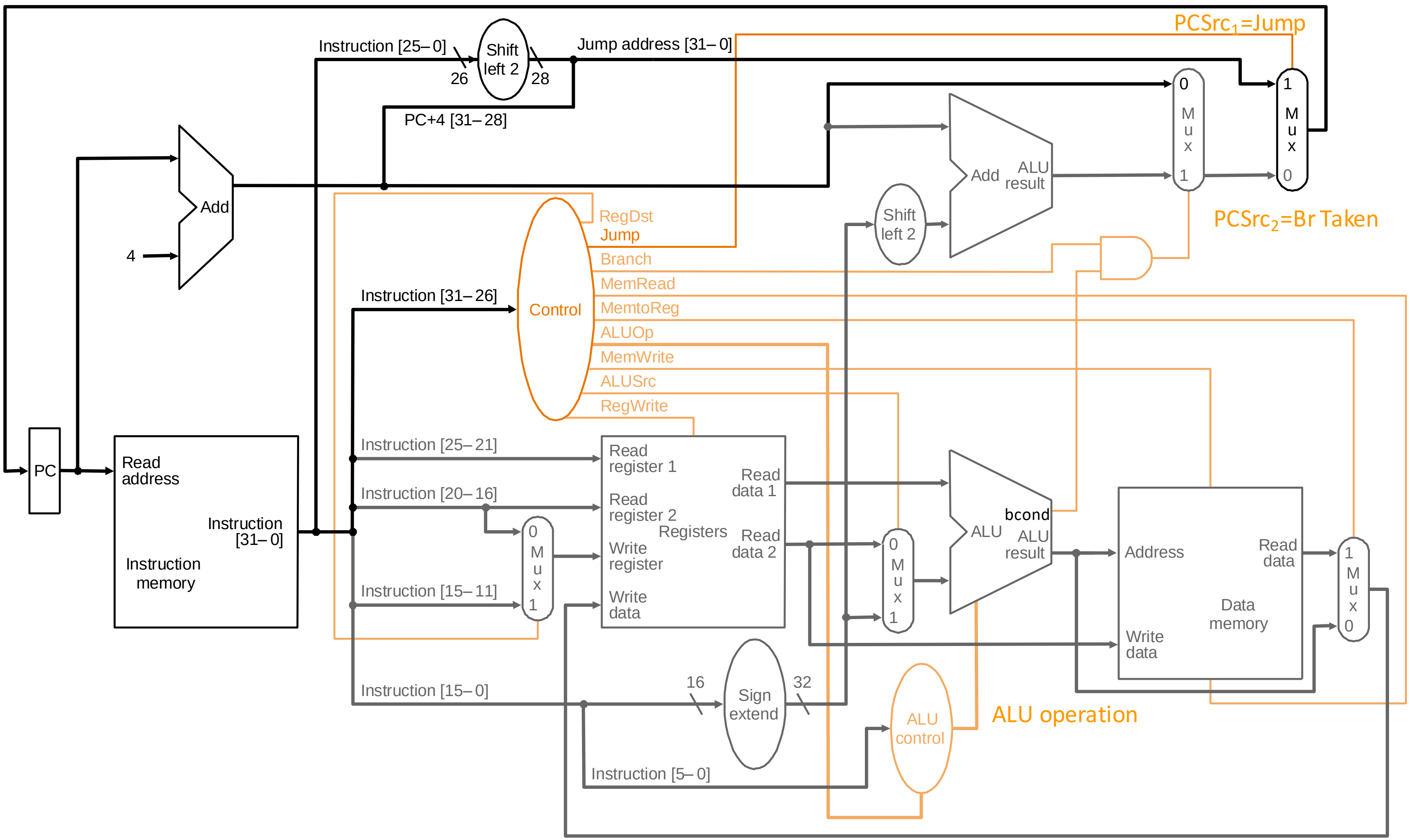
† This is the incremented PC

Conditional Branch Datapath (for you to finish)



How to uphold the delayed branch semantics?

Putting It All Together



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

JAL, JR, JALR omitted

Single-Cycle Control Logic

Single-Cycle Hardwired Control

- As combinational function of Inst=MEM[PC]

R-Type

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

I-Type

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

J-Type

31–26	25–0
opcode	immediate
6 bits	26 bits

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

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	2nd ALU input from 2nd GPR read port	2nd ALU input from sign-extended 16-bit immediate	(opcode!=0) && (opcode!=BEQ) **&&** (opcode!=BNE)
MemtoReg	Steer ALU result to GPR write port	steer memory load to GPR write port	opcode==LW
RegWrite	GPR write disabled	GPR write enabled	(opcode!=SW) && (opcode!=Bxx) && (opcode!=J) && (opcode!=JR))

JAL and JALR require additional RegDest and MemtoReg options

Single-Bit Control Signals (II)

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

JR and JALR require additional PCSrc options

ALU Control

- case opcode

  ‘0’ ⇒ select operation according to funct

  ‘ALUi’ ⇒ selection operation according to opcode

  ‘LW’ ⇒ select addition

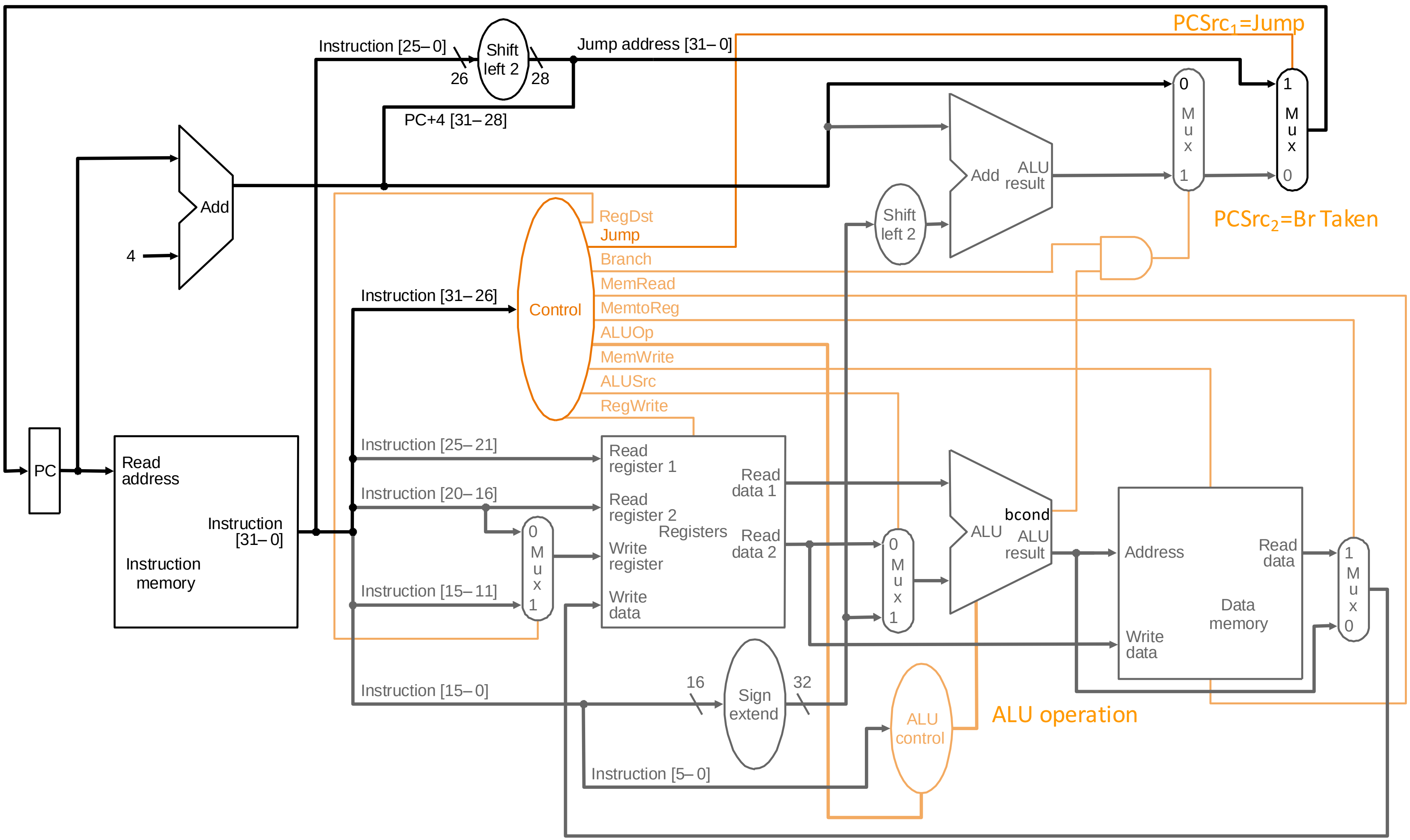
  ‘SW’ ⇒ select addition

  ‘Bxx’ ⇒ select bcond generation function

  __ ⇒ don’t care

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

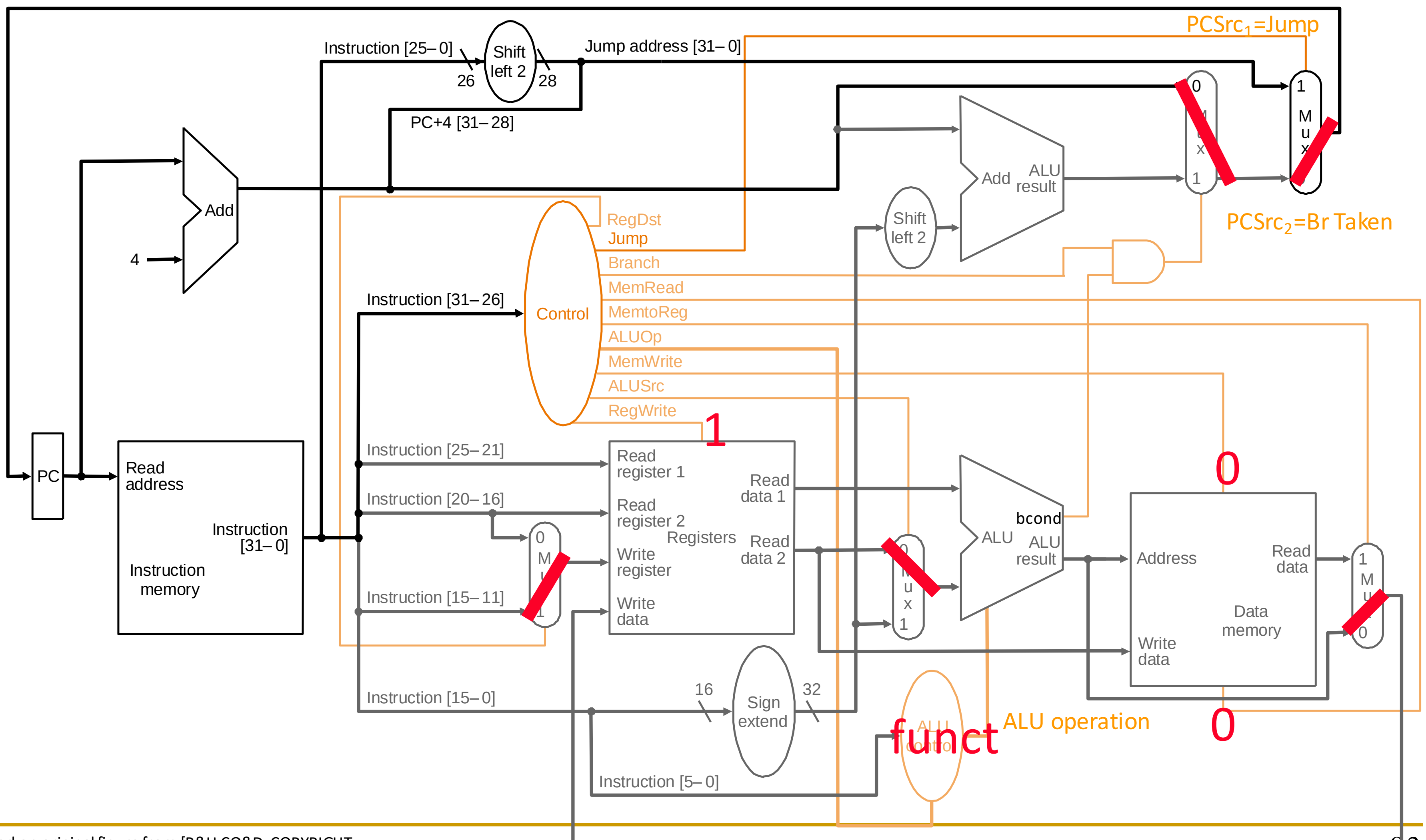
Let's Control The Single-Cycle MIPS Datapath



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

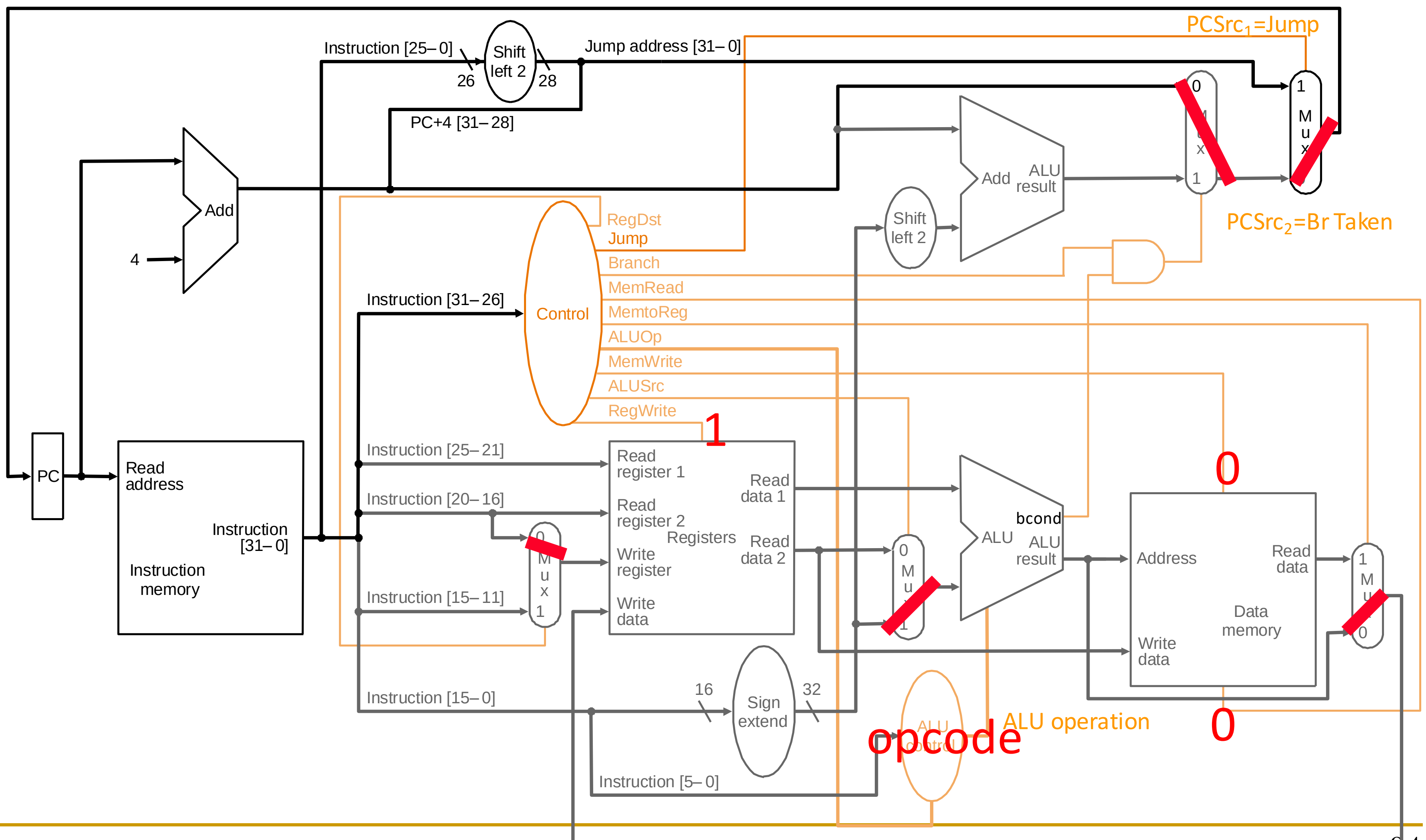
JAL, JR, JALR omitted

R-Type ALU



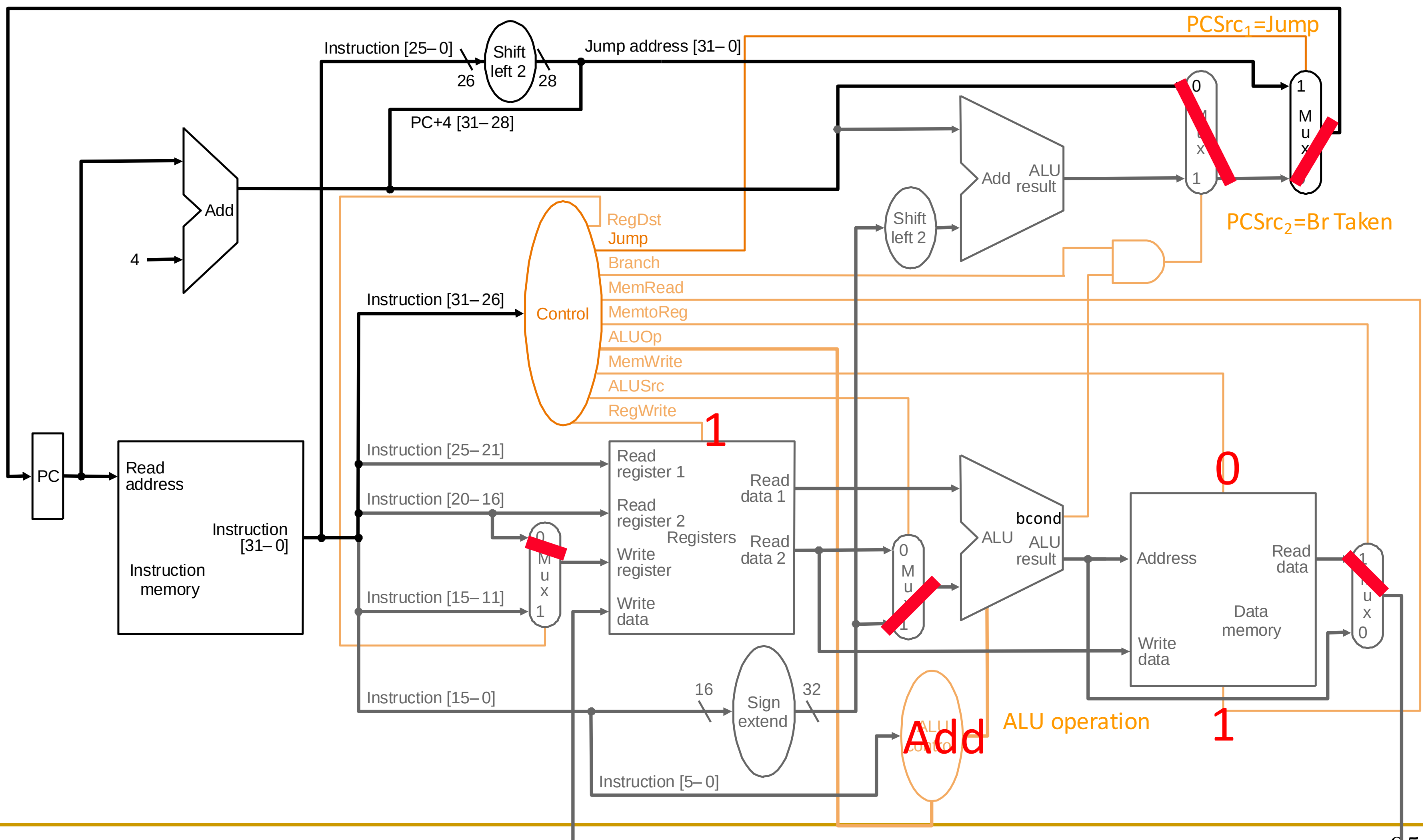
PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
0
M
u
x
1
1
M
u
x
0
Add
ALU result
Add
4
PCSrc2=Br Taken
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31–26]
Control
1
Instruction [25–21]
Instruction [20–16]
Instruction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

I-Type ALU

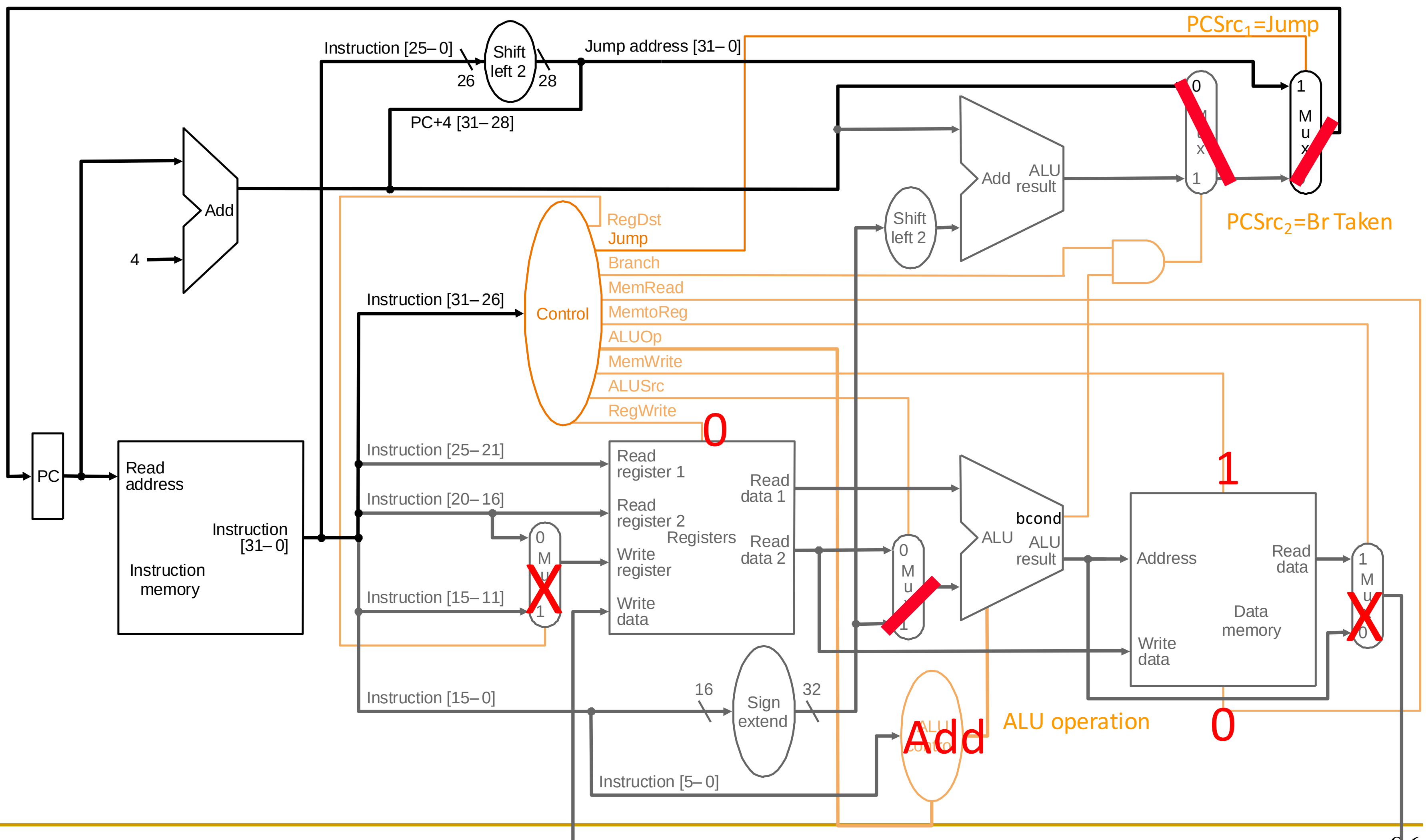


PCSrc1=Jump
Instruction [25–0]
26
Shift left 2
28
Jump address [31–0]
PC+4 [31–28]
Add
4
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [31–26]
Control
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
1
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction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

LW


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

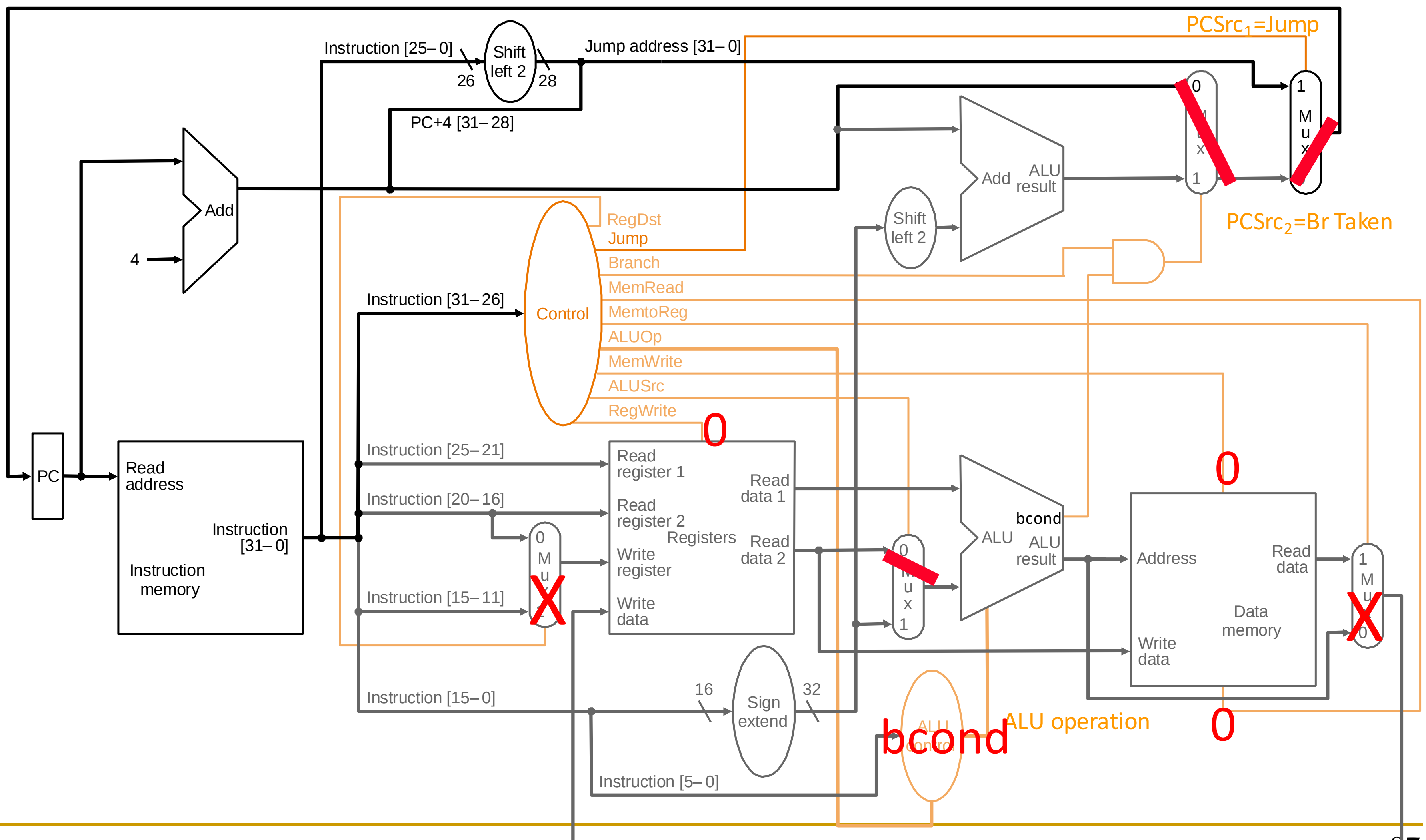
SW



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

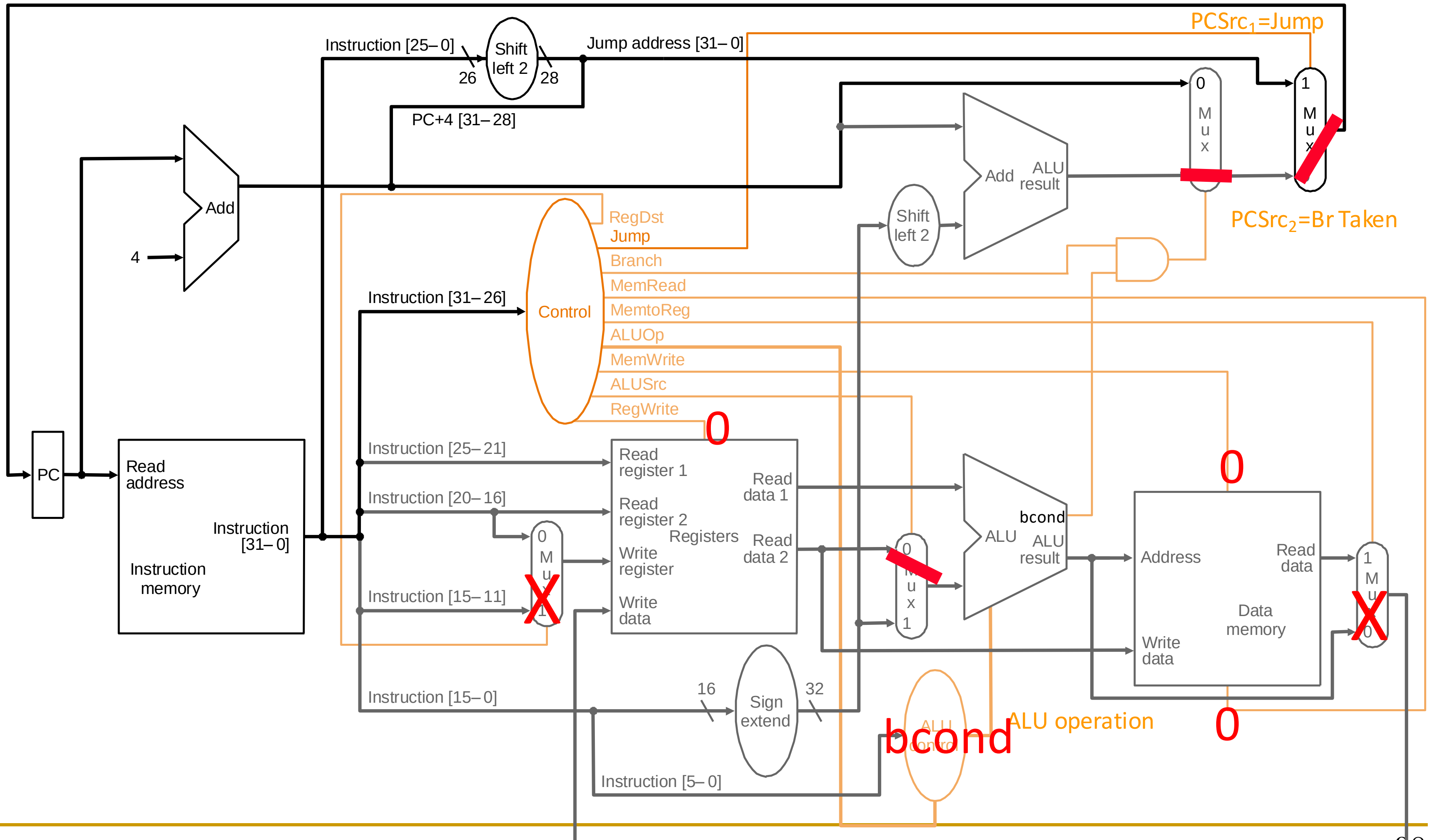
Branch (Not Taken)

Some control signals are dependent on the processing of data

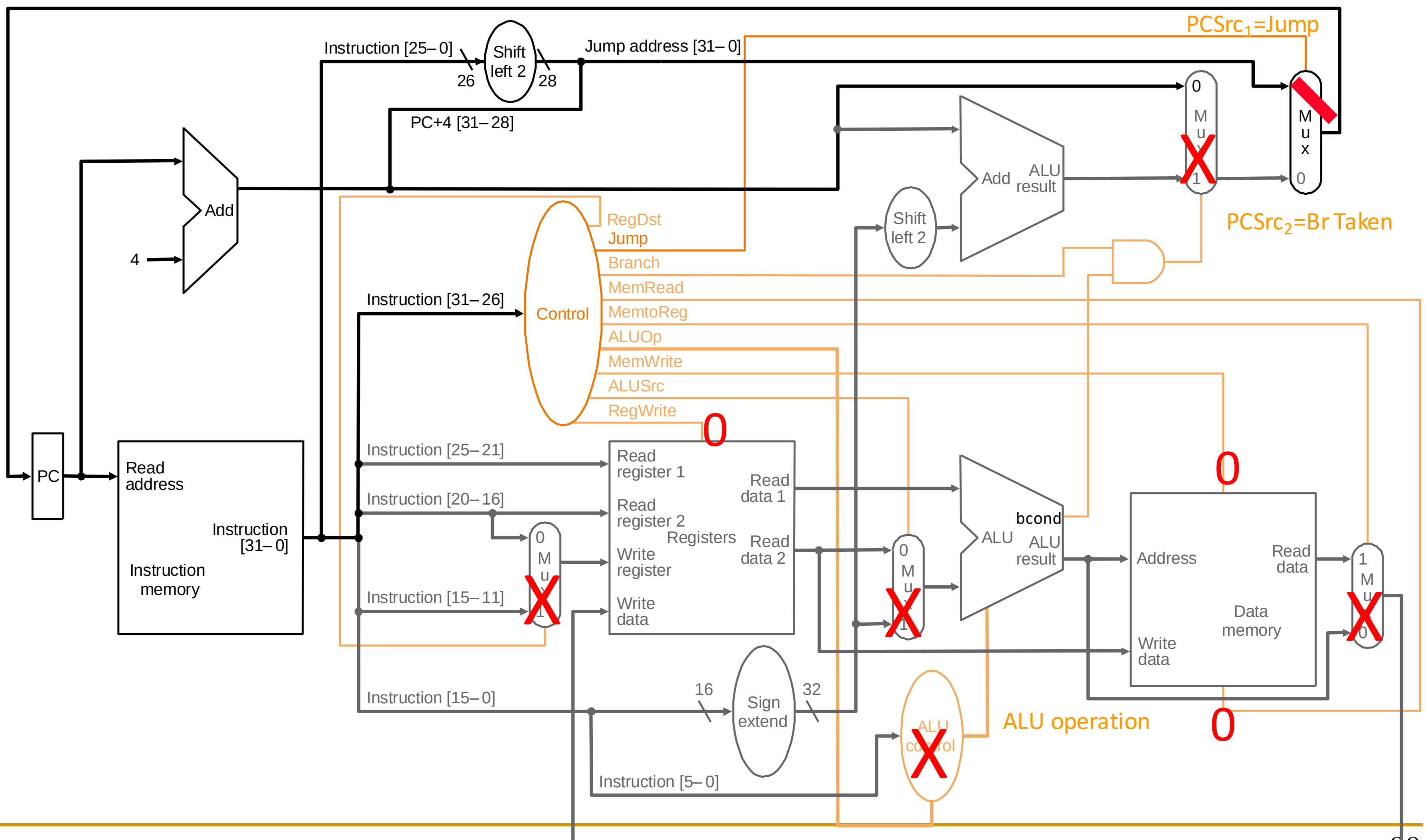


Branch (Taken)

Some control signals are dependent on the processing of data



Jump



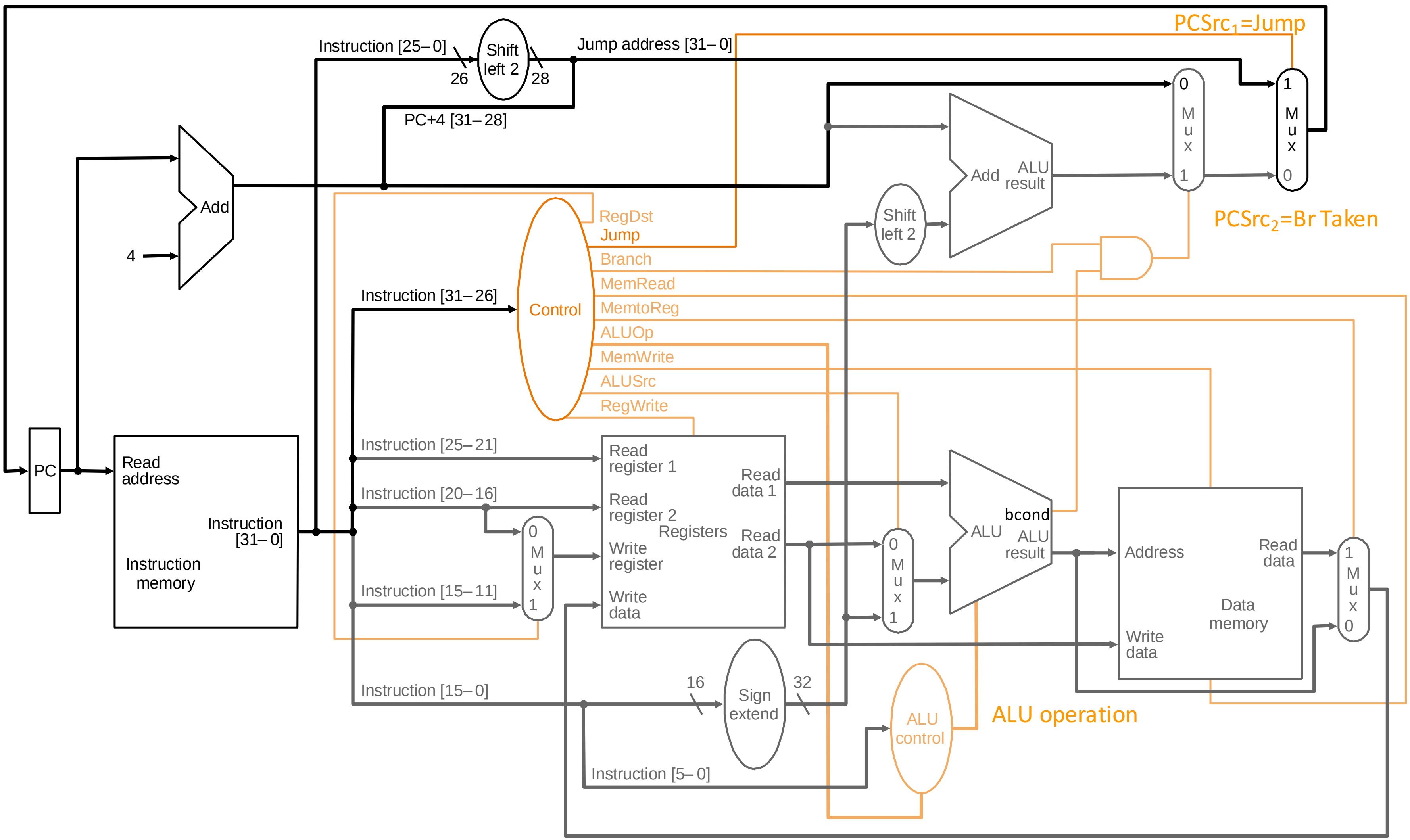
PCSrc1=Jump
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
0
M u x
1
Add
ALU result
4
Shift left 2
PCSrc2=Br Taken
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
0
PC
Read address
Instruction [31–0]
Instruction memory
Instruction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

What is in That Control Box?

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

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

Review: Complete Single-Cycle Processor



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

JAL, JR, JALR omitted

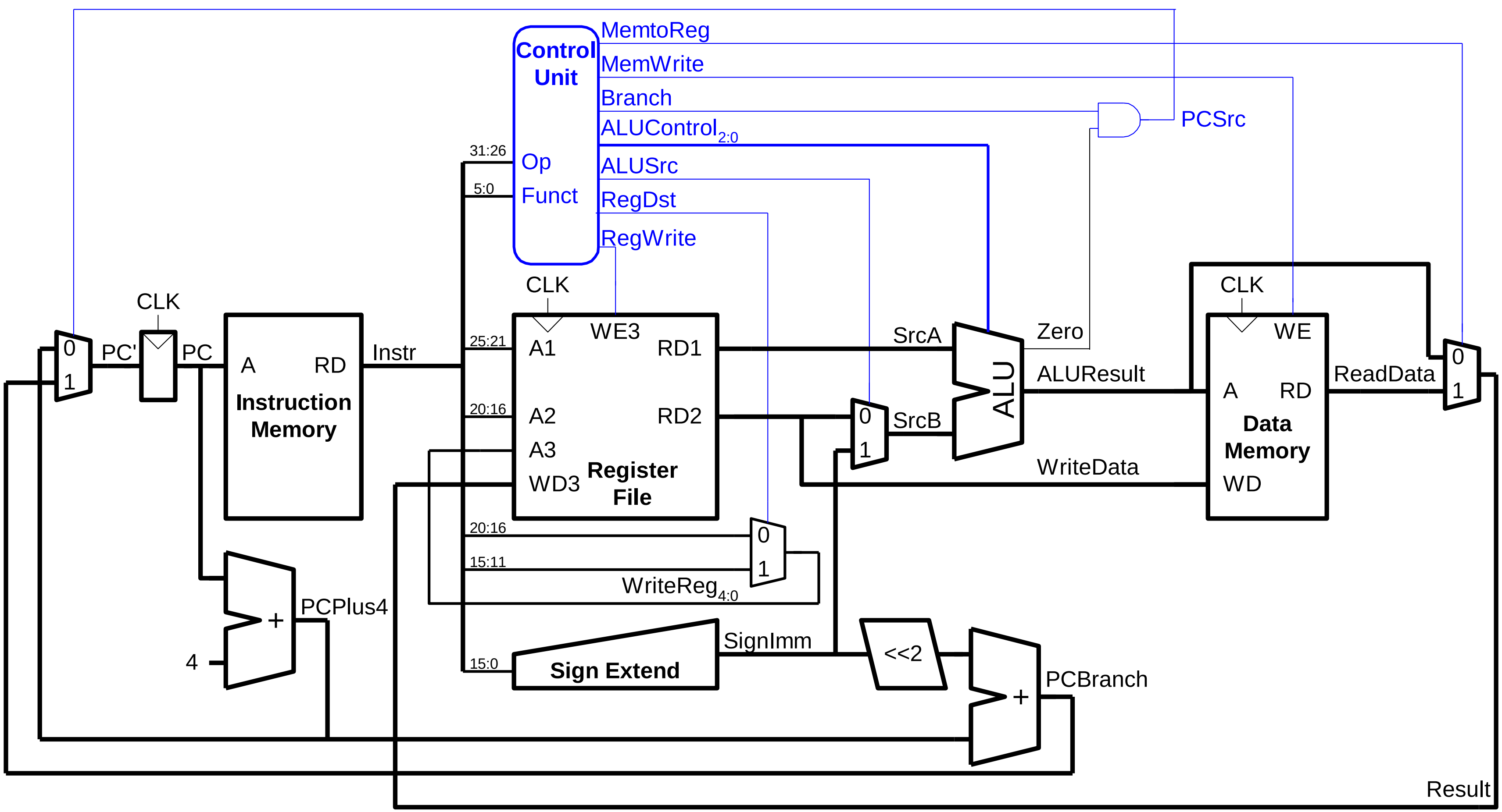
Another Single-Cycle MIPS Processor (from H&H)

See backup slides to reinforce the concepts we have covered.

They are to complement your reading:

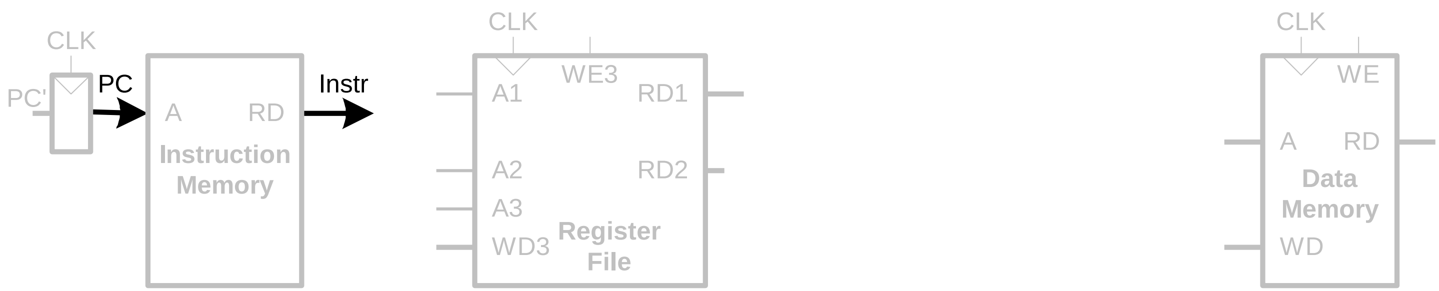
H&H, Chapter 7.1-7.3, 7.6

Another Complete Single-Cycle Processor



Example: Single-Cycle Datapath: `lw` fetch

- *STEP 1:* Fetch instruction



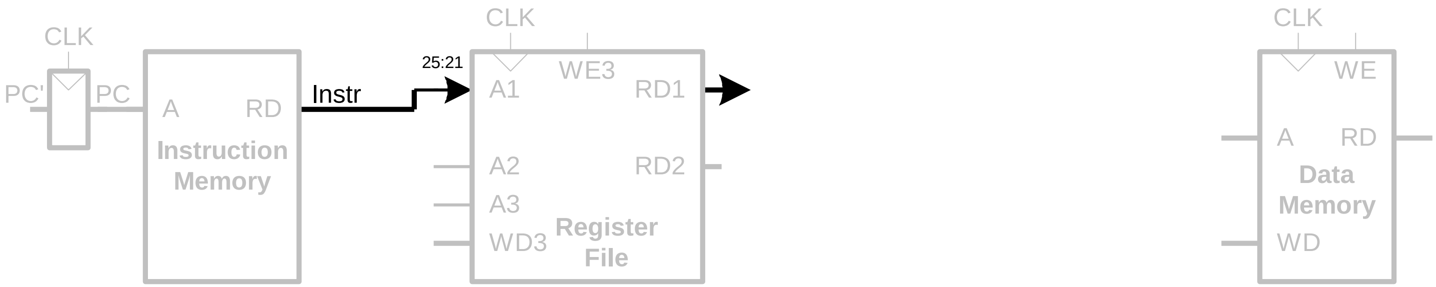

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: `lw` register read

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



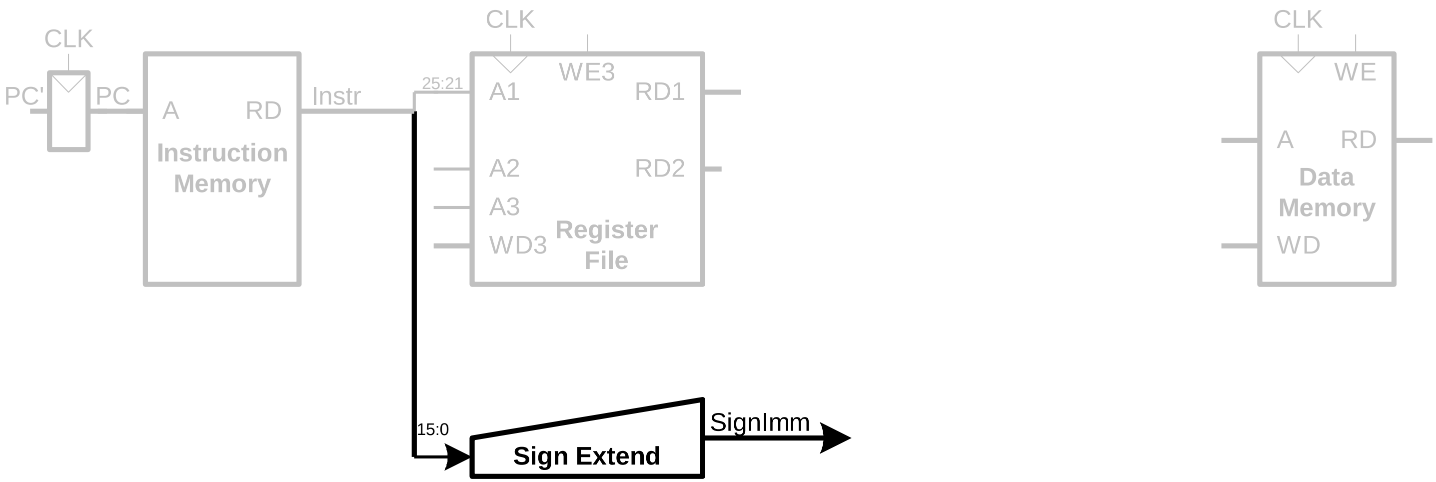

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



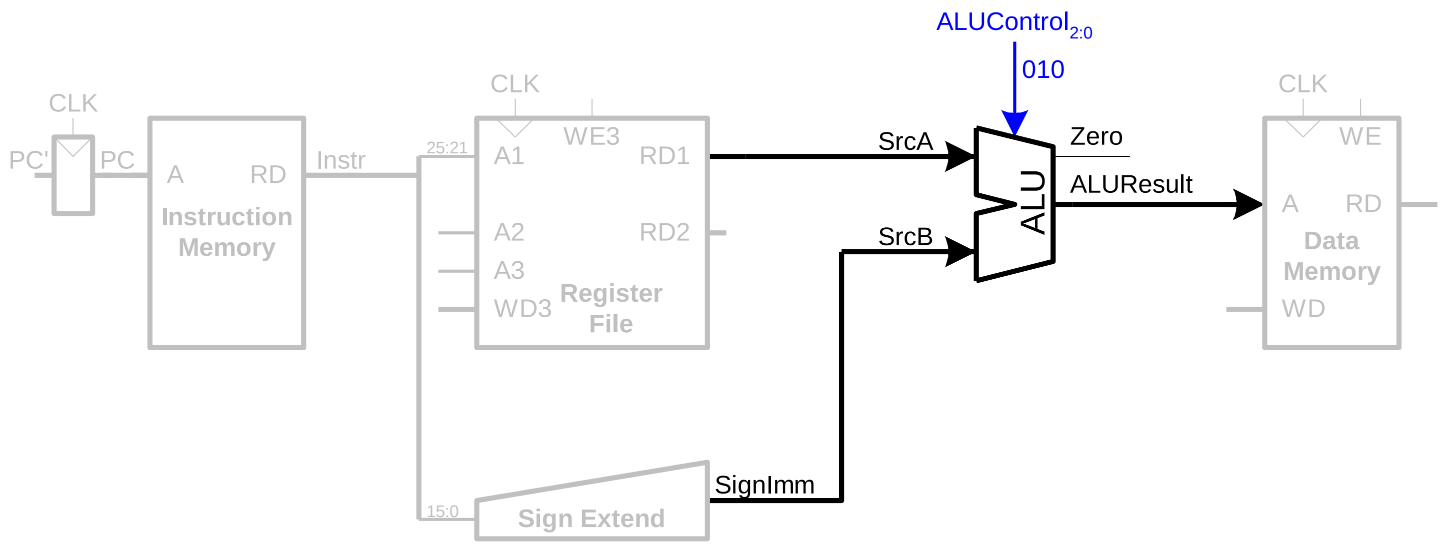

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

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**



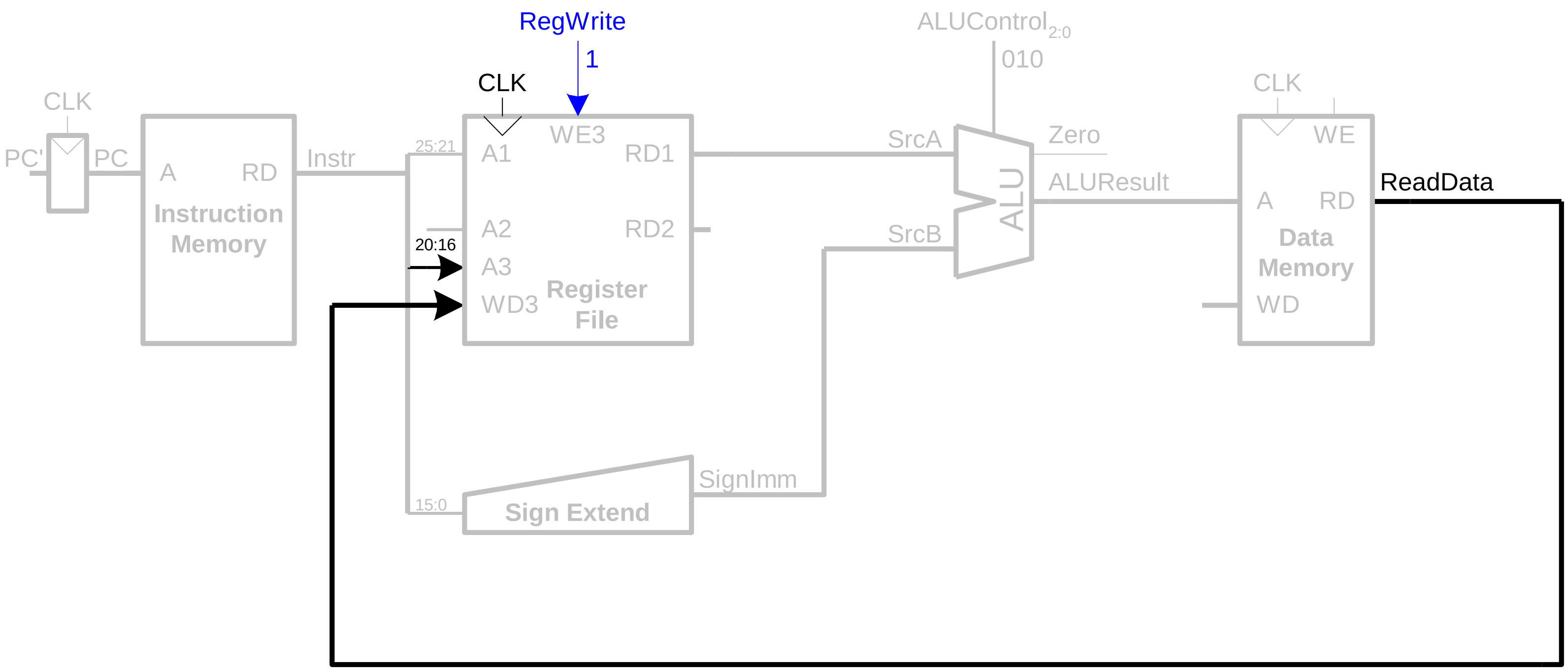

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



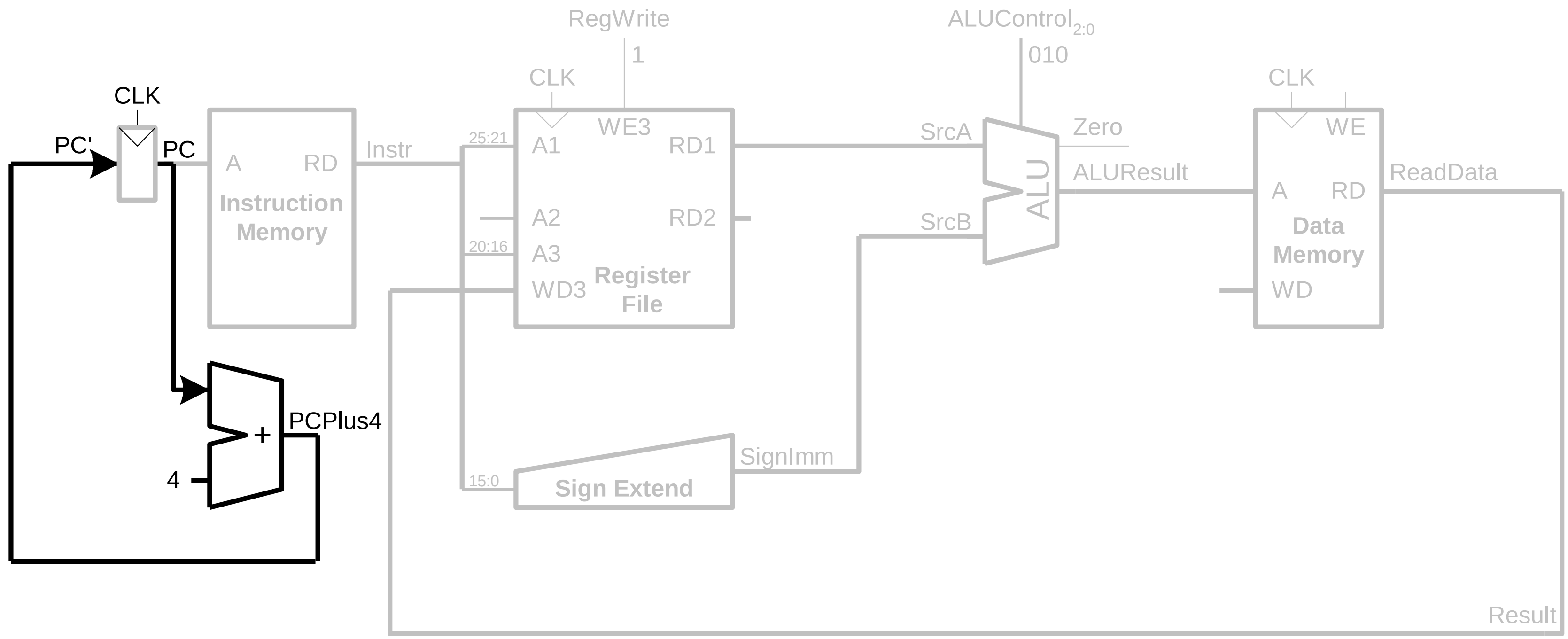
`lw $s3, 1($0)   # read memory word 1 into $s3`

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

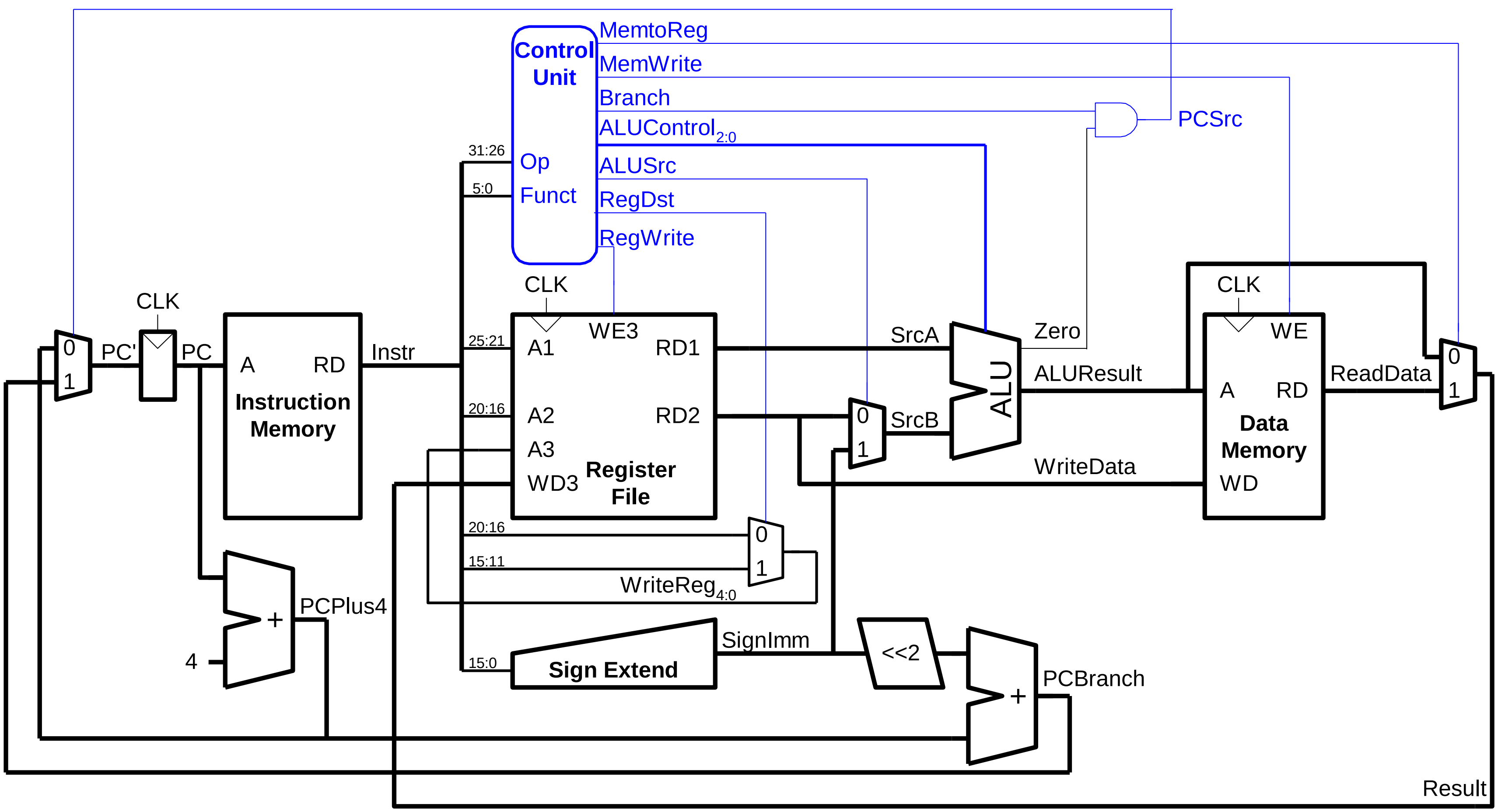
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Similarly, We Need to Design the Control Unit

- Control signals 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)

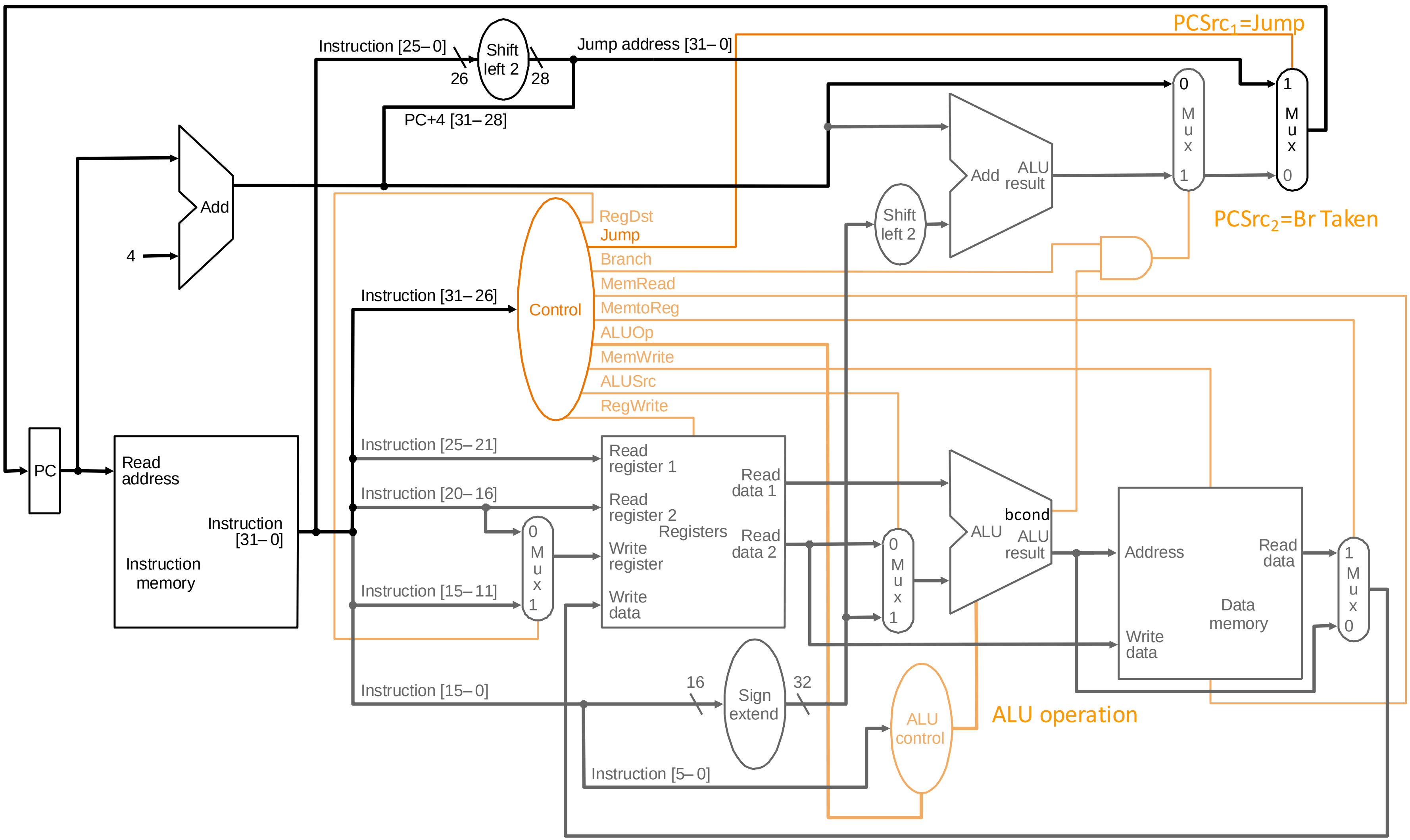


Your Assignment

- Please read the Backup Slides

- Please do your readings from the H&H Book
  - H&H, Chapter 7.1-7.3, 7.6

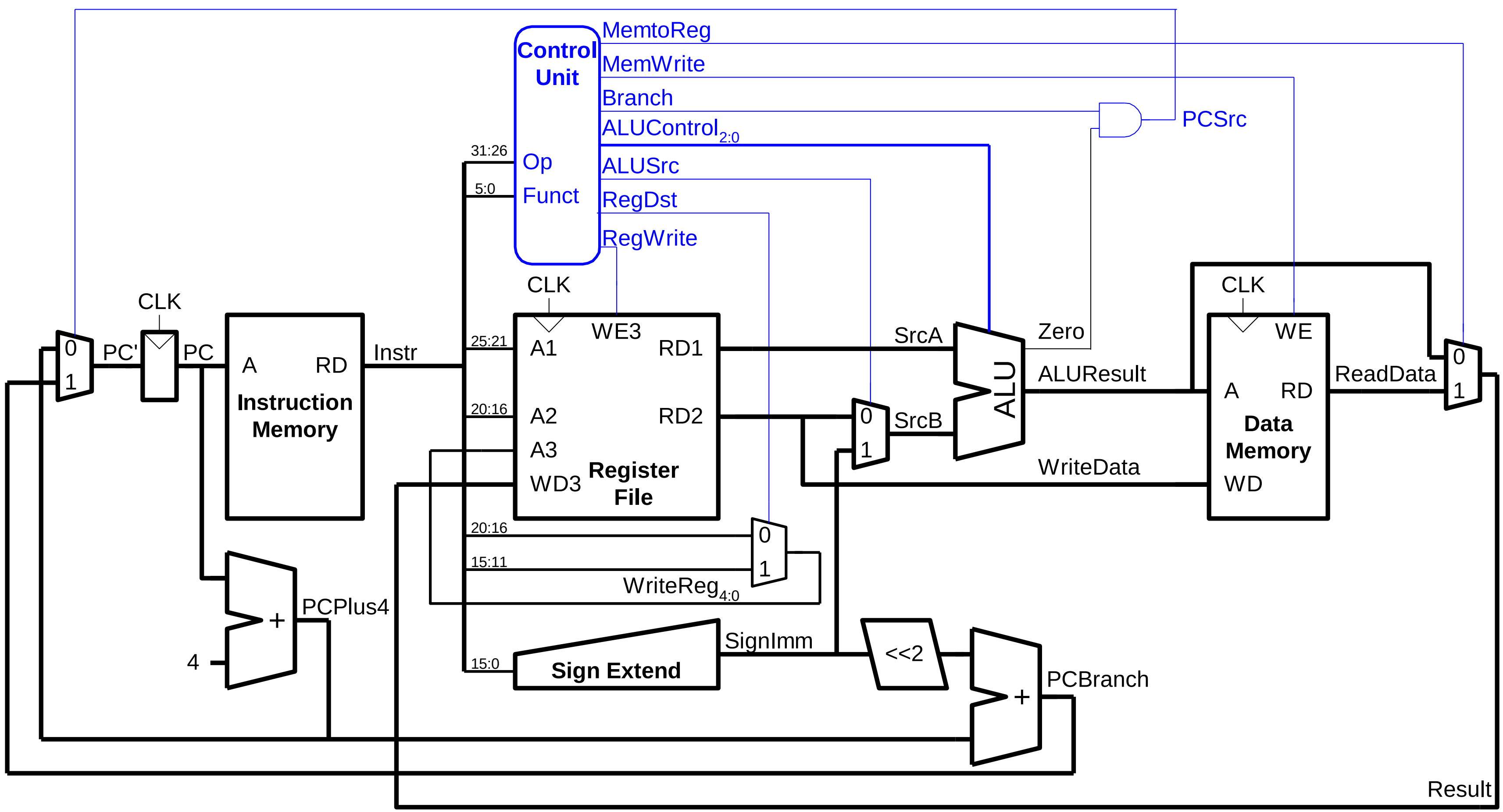
Single-Cycle Uarch I (We Developed in Lectures)



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

JAL, JR, JALR omitted

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/clock period = ***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 an instruction
  - {CPI} x {clock cycle time}
    - CPI: Number of cycles it takes to execute an instruction

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

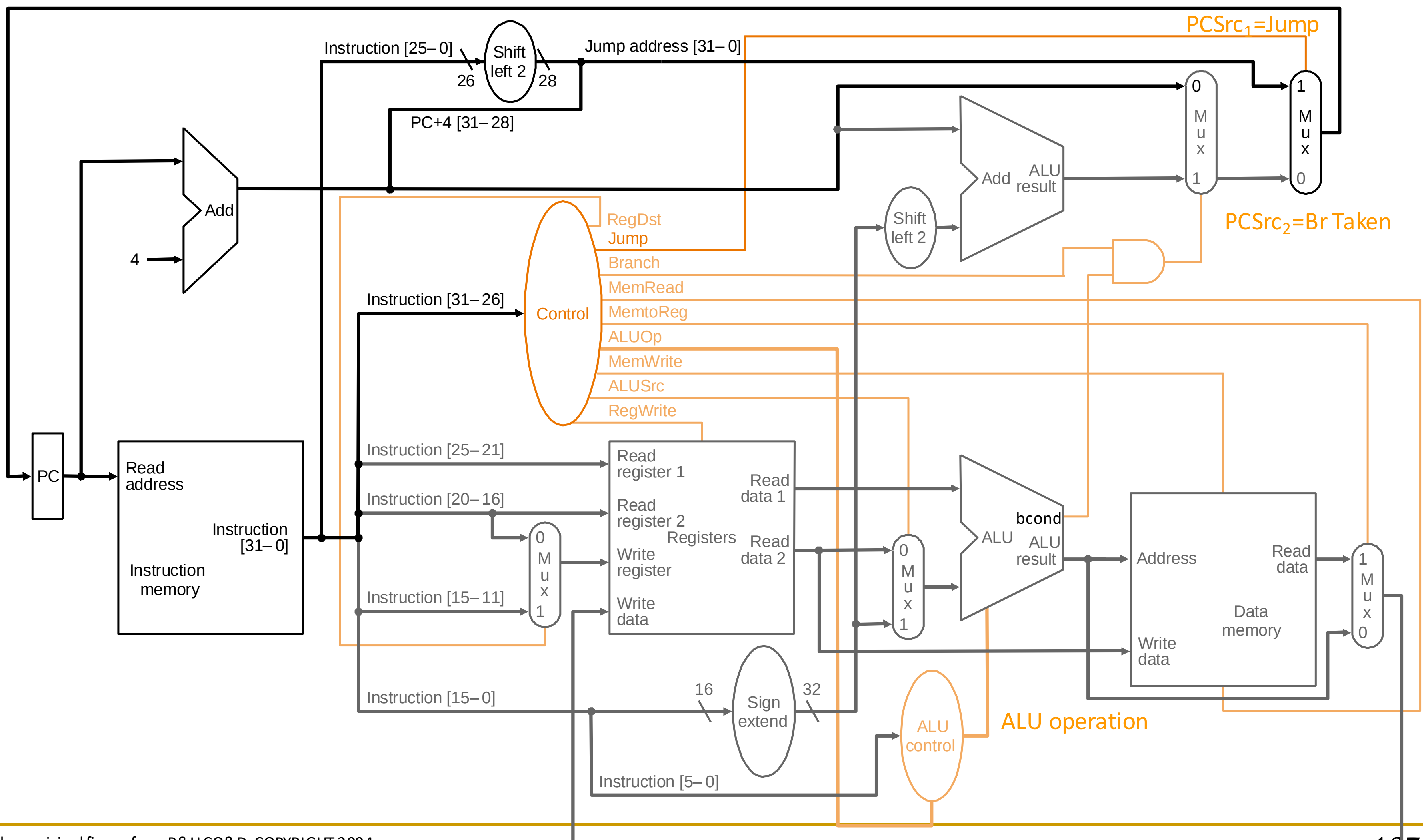
- Let's go back to the basics

- 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

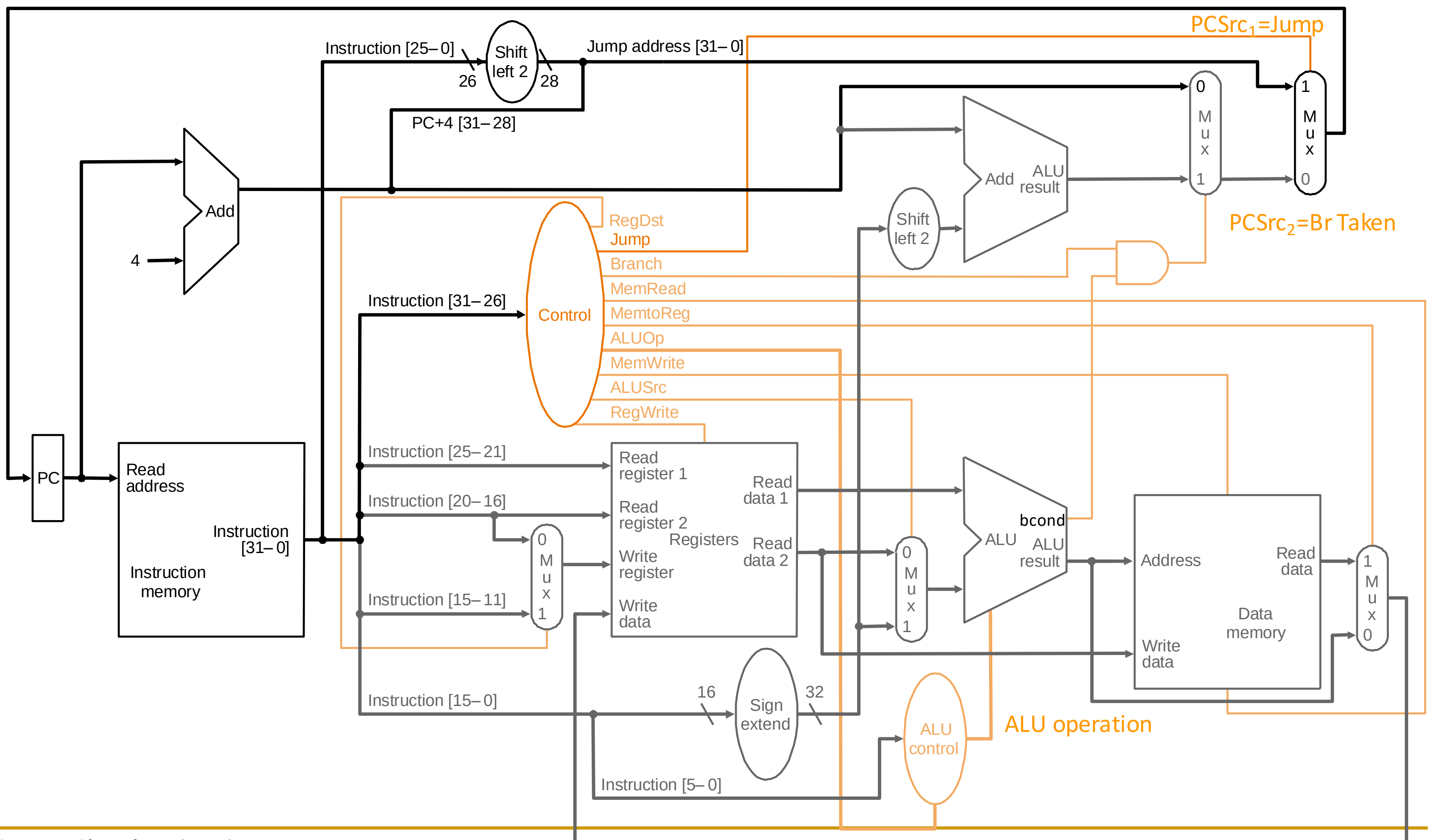


Example Single-Cycle Datapath Analysis

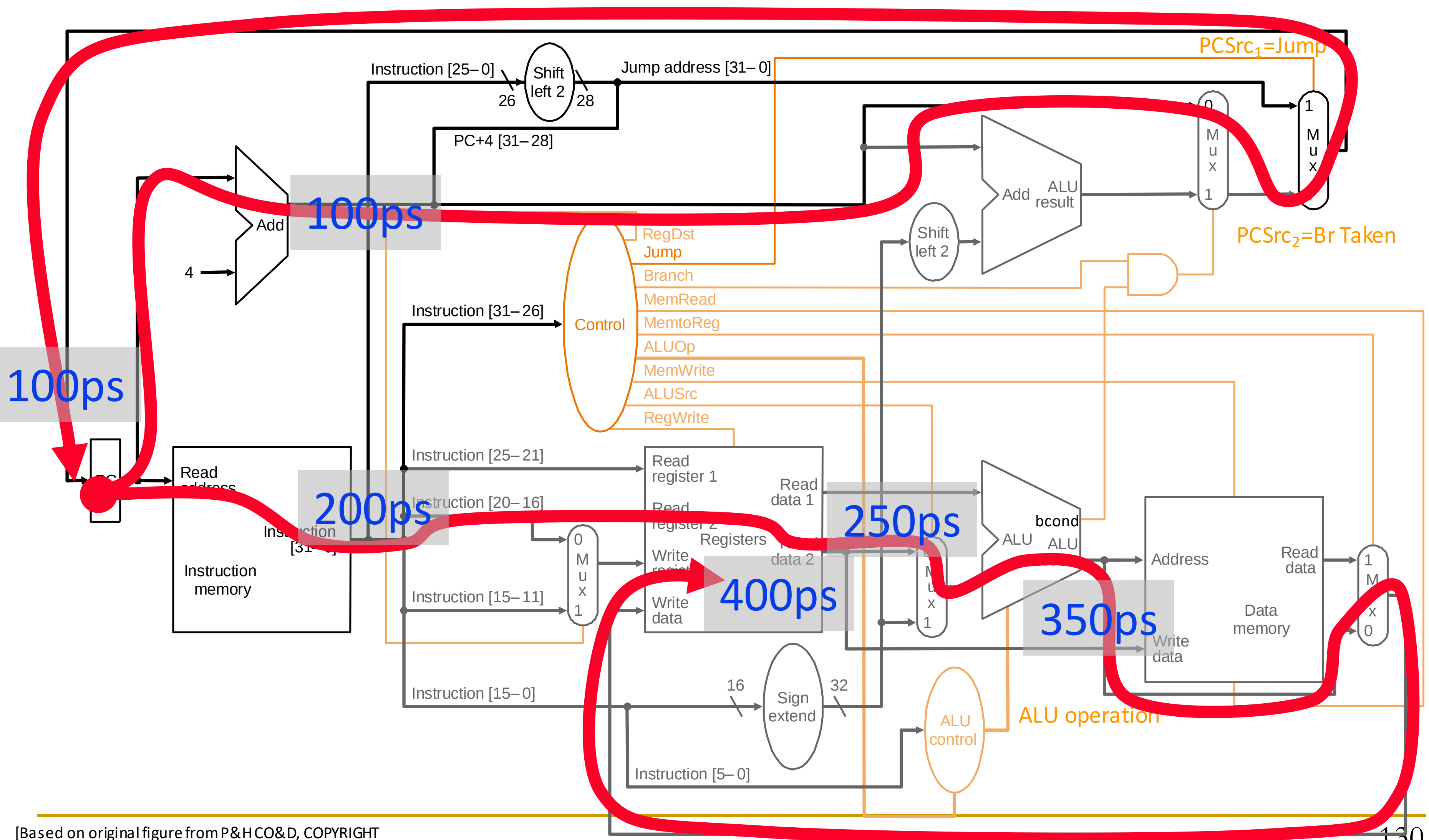
- Assume (for the design in the previous slide)
  - memory units (read or write): 200 ps
  - ALU and adders: 100 ps
  - register file (read or write): 50 ps
  - other combinational logic: 0 ps

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

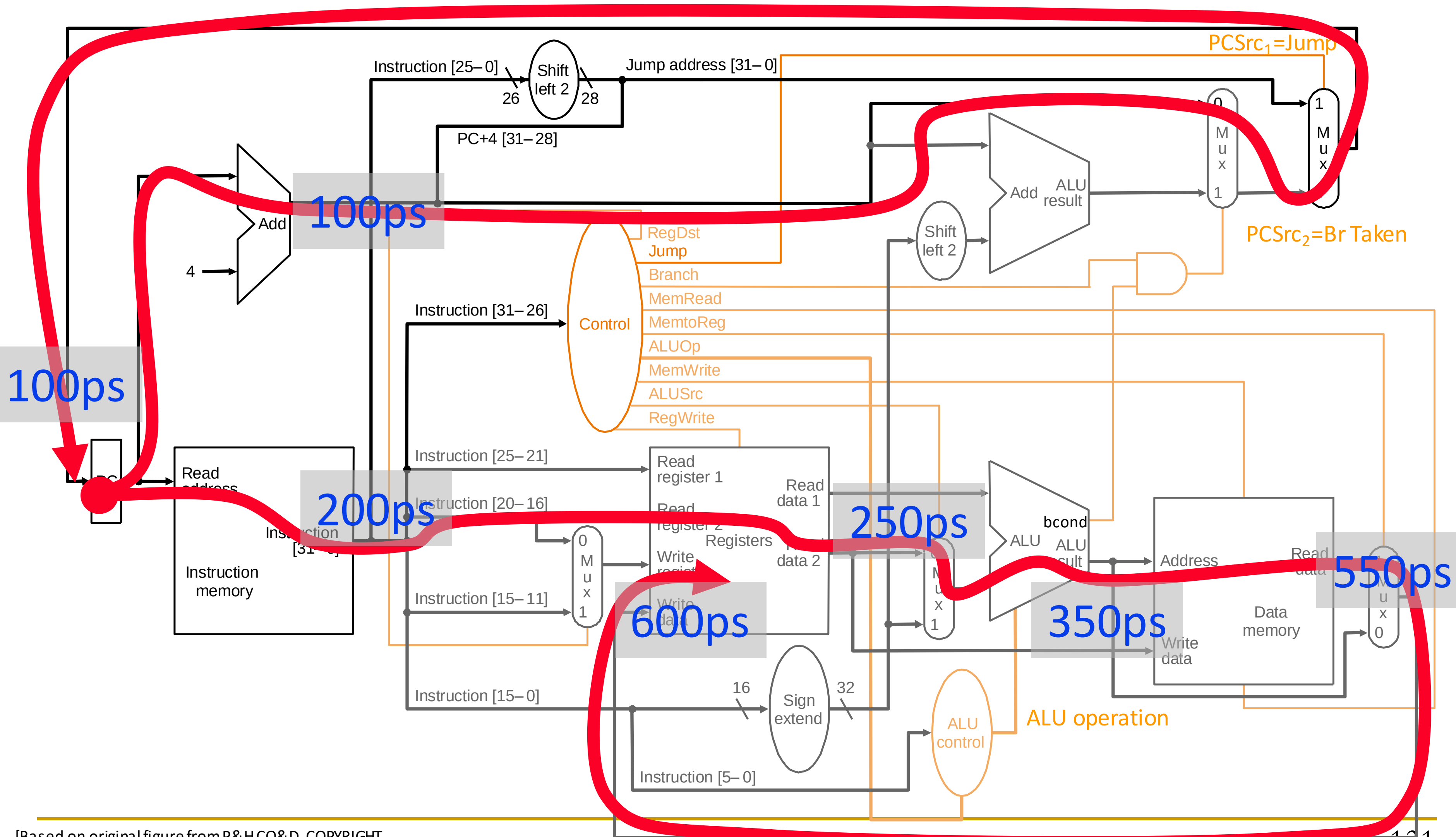
Let’s Find the Critical Path



R-Type and I-Type ALU

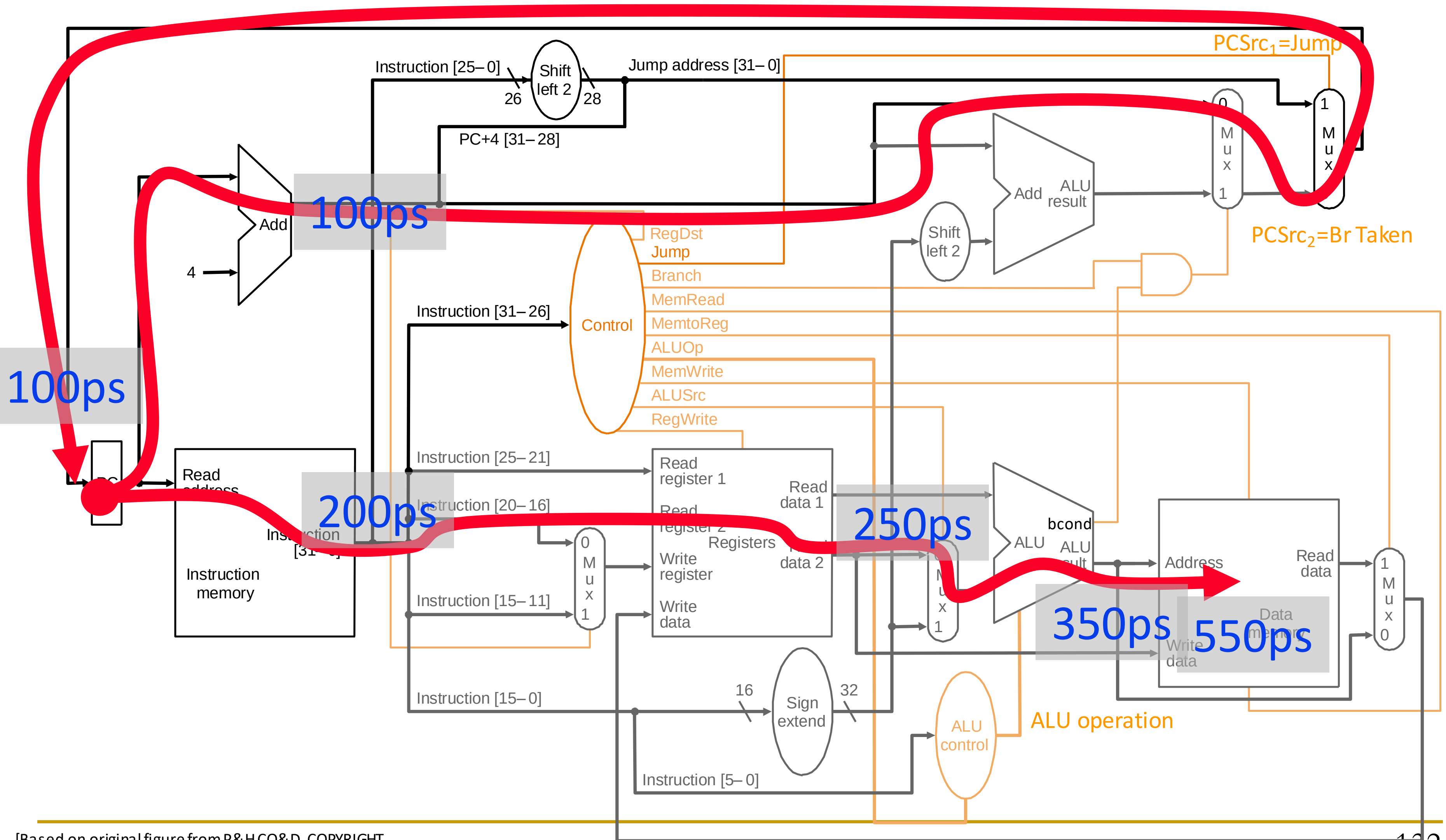


LW



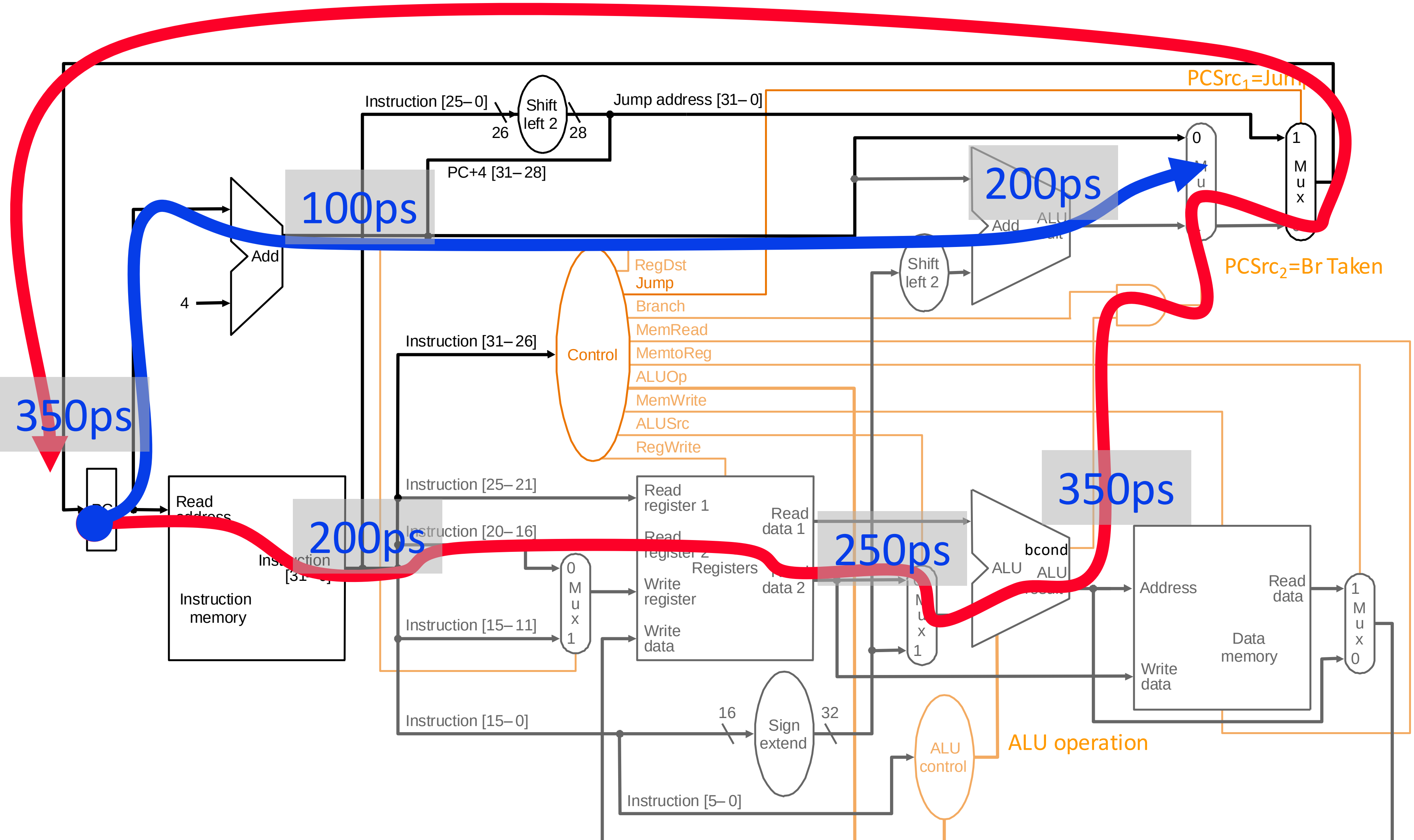
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PCSrc1=Jump
PC+4 [31–28]
Add
4
100ps
100ps
Add ALU result
Shift left 2
PCSrc2=Br Taken
M u x
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]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

SW

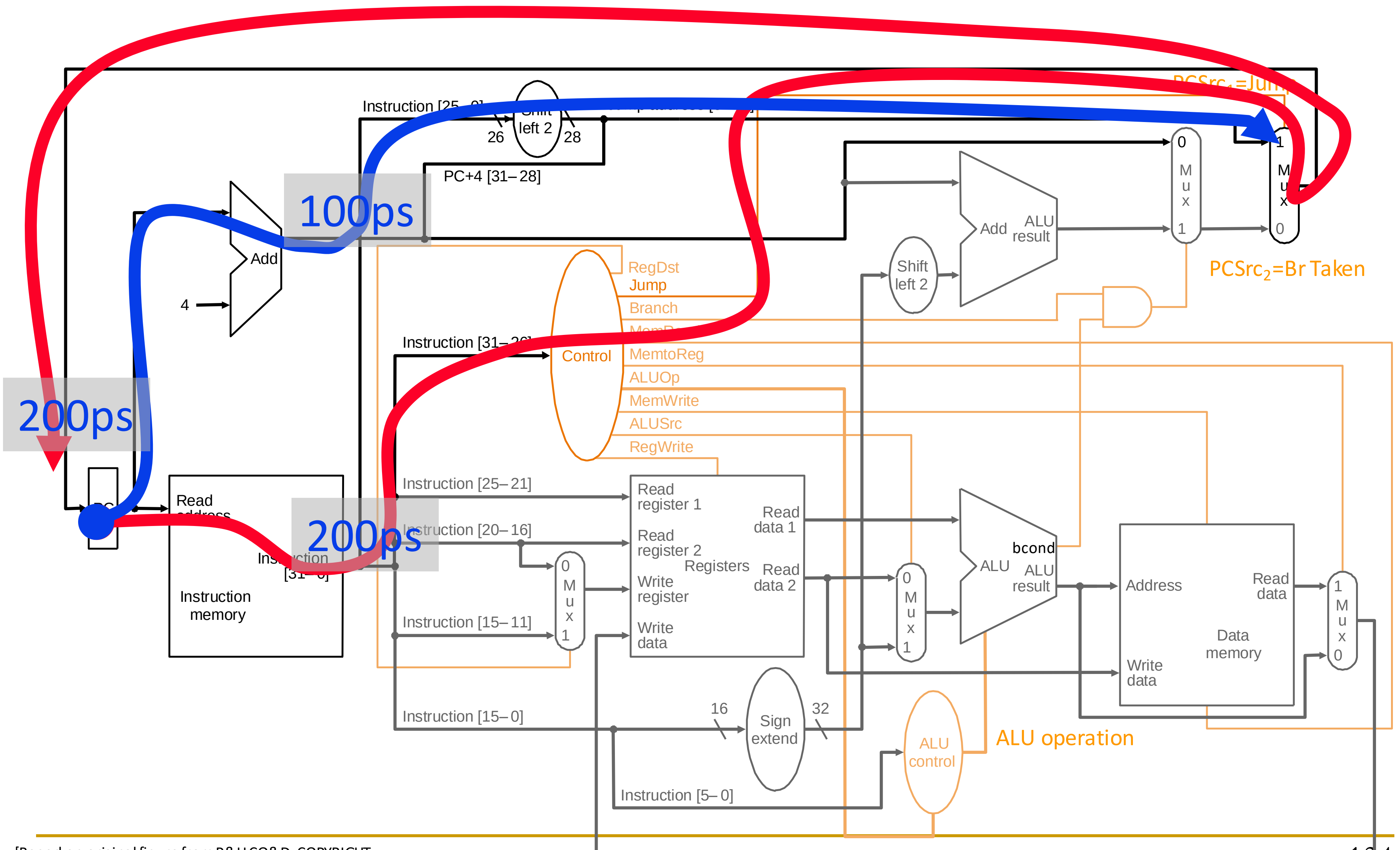


PCSrc1=Jump
PCSrc2=Br Taken
Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
100ps
100ps
Add
ALU result
Shift left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction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

Branch Taken



Jump



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

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?

- Memory is not magic
- What if memory *sometimes* takes 100ms to access?
- Does it make sense to have a simple register to register add or jump to take {100ms+all else to do a memory operation}?
- And, what if you need to access memory more than once to process an instruction?
  - Which instructions need this?
  - Do you provide multiple ports to memory?

Single Cycle uArch: Complexity

- Contrived
  - All instructions run as slow as the slowest instruction
- Inefficient
  - All instructions run as slow as the slowest instruction
  - Must provide worst-case combinational resources in parallel as required by any instruction
  - Need to replicate a resource if it is needed more than once by an instruction during different parts of the instruction processing cycle
- Not necessarily the simplest way to implement an ISA
  - Single-cycle implementation of REP MOVS (x86) or INDEX (VAX)?
- Not easy to optimize/improve performance
  - Optimizing the common case does not work (e.g. common instructions)
  - Need to optimize the worst case all the time

(Micro)architecture Design Principles

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

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

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

Single-Cycle Design vs. Design Principles

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

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

Aside: System Design Principles

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

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

Aside: From Lecture 1

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

Aside: System Design Principles

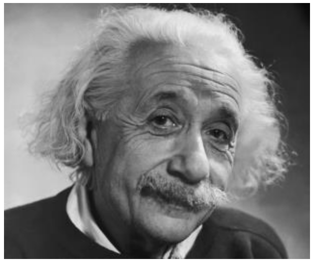
- We will continue to cover key principles in this course
- Here are some references where you can learn more

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

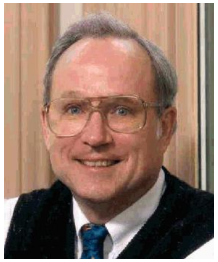
A Key System Design Principle

- 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

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

Remember: The "Process instruction" Step

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

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

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

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

Not easy to optimize design

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 uArch

- 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

The Instruction Processing Cycle

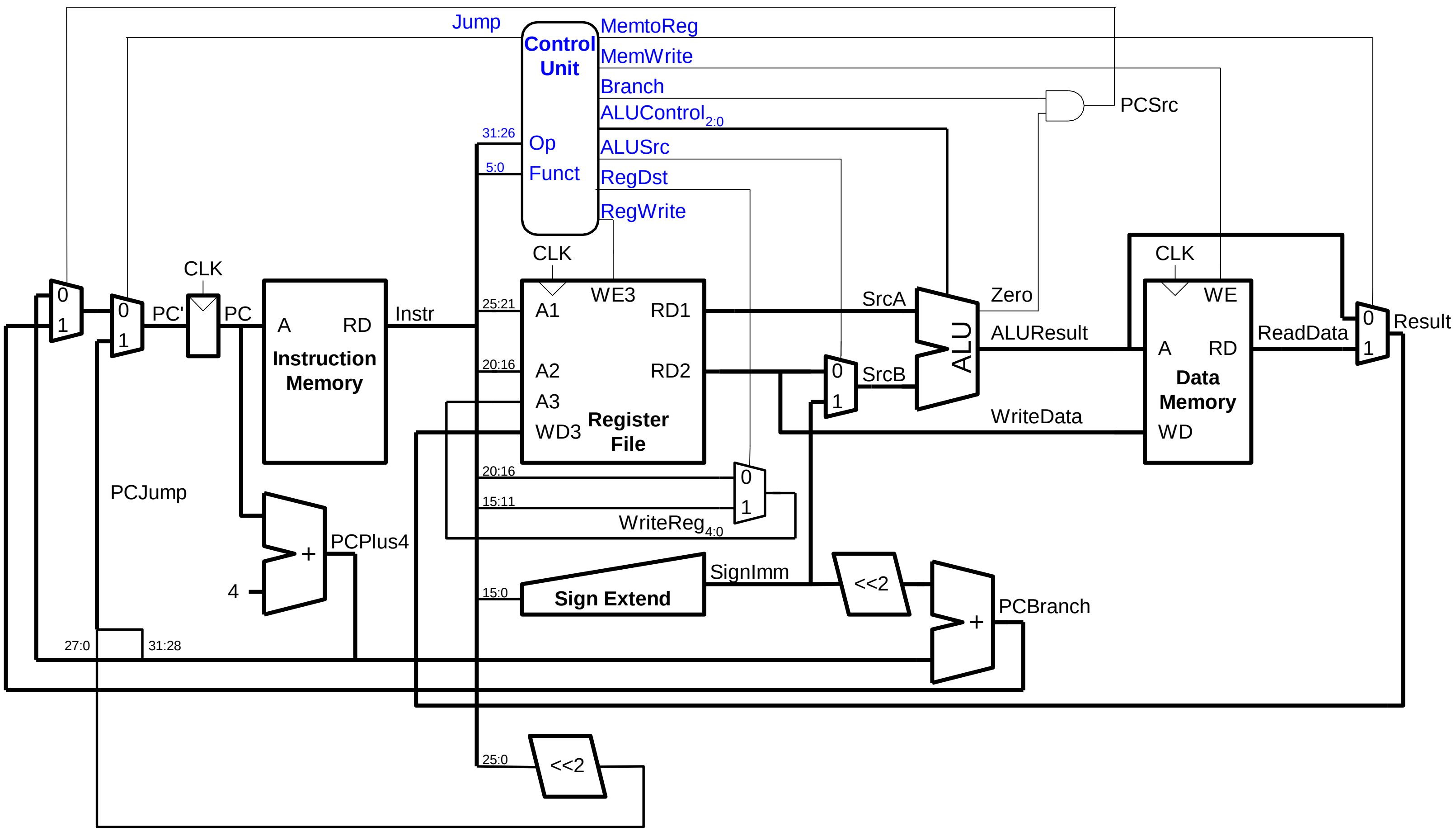
- Fetch
- Decode
- Evaluate Address
- Fetch Operands
- Execute
- Store Result

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:**
  - \+ simple
  - \- cycle time limited by longest instruction (`lw`)
  - \- three adders/ALUs and two memories

- **Multi-cycle microarchitecture:**
  - \+ higher clock speed
  - \+ simpler instructions run faster
  - \+ reuse expensive hardware on multiple cycles
  - \- sequencing overhead paid many times
  - \- hardware overhead for storing intermediate results

- **Same design steps: datapath & control**

What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

What Do We Want To Optimize

- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)

- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)

What Do We Want To Optimize

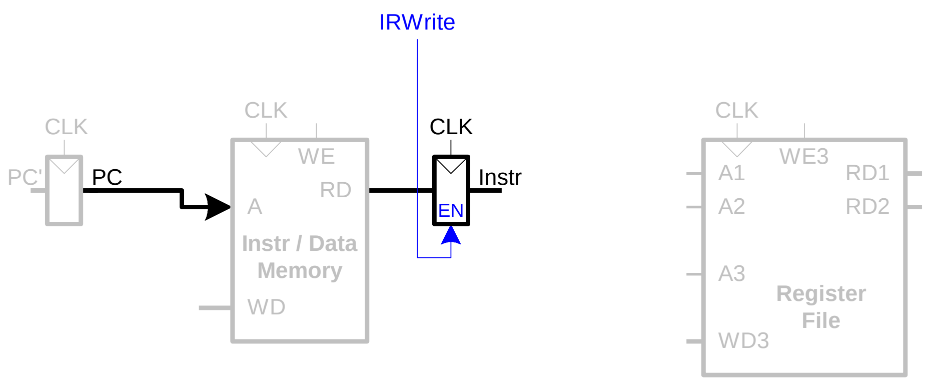
- **Single Cycle Architecture uses two memories**
  - One memory stores instructions, the other data
  - We want to use a single memory (Smaller size)
- **Single Cycle Architecture needs three adders**
  - ALU, PC, Branch address calculation
  - We want to use the ALU for all operations (smaller size)
- **In Single Cycle Architecture all instructions take one cycle**
  - The most complex operation slows down everything!
  - Divide all instructions into multiple steps
  - Simpler instructions can take fewer cycles (average case may be faster)

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

- **First consider executing lw**
  - STEP 1: Fetch instruction

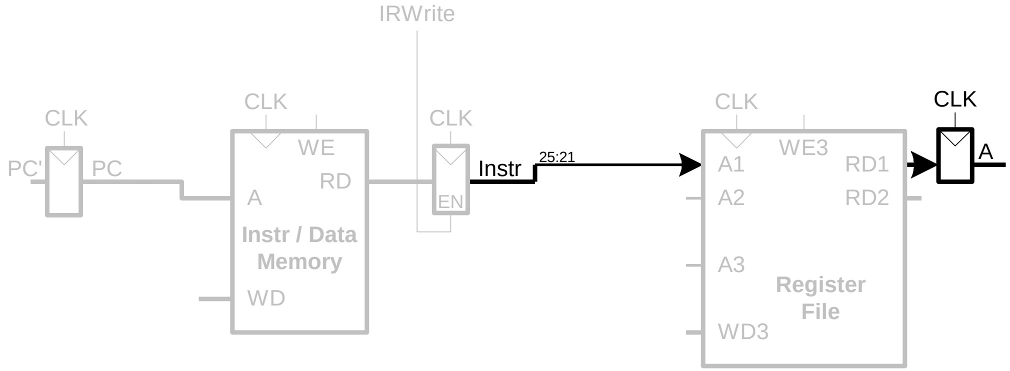


`read from the memory location [rs]+imm to location [rt]`

I-Type

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

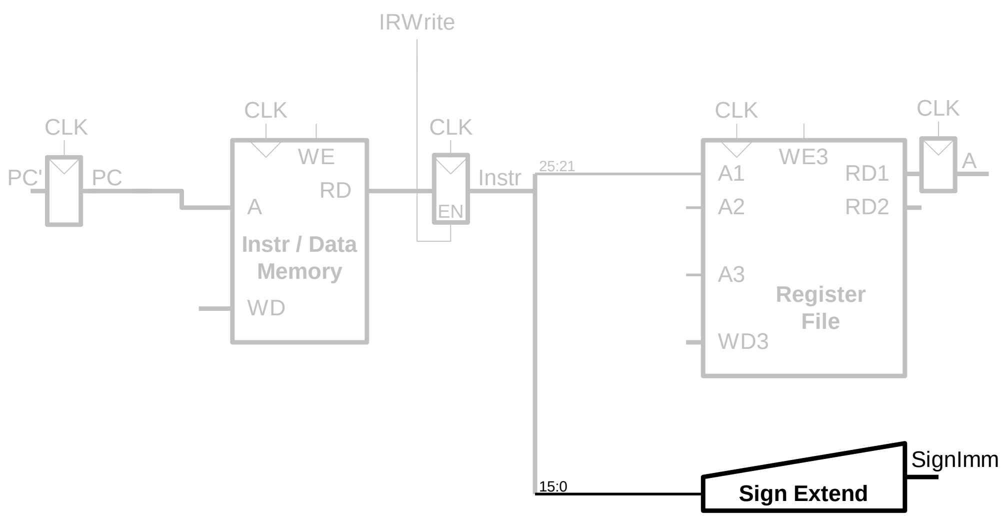
Multi-cycle Datapath: `lw` register read



I-Type

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

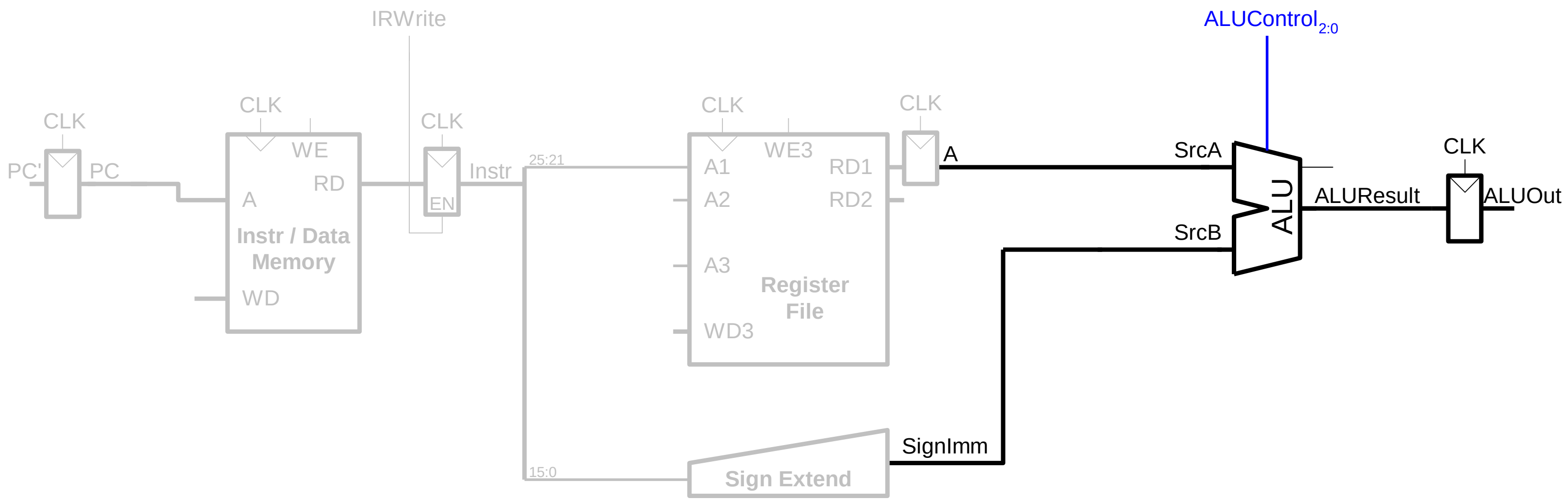
Multi-cycle Datapath: `lw` immediate



I-Type

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

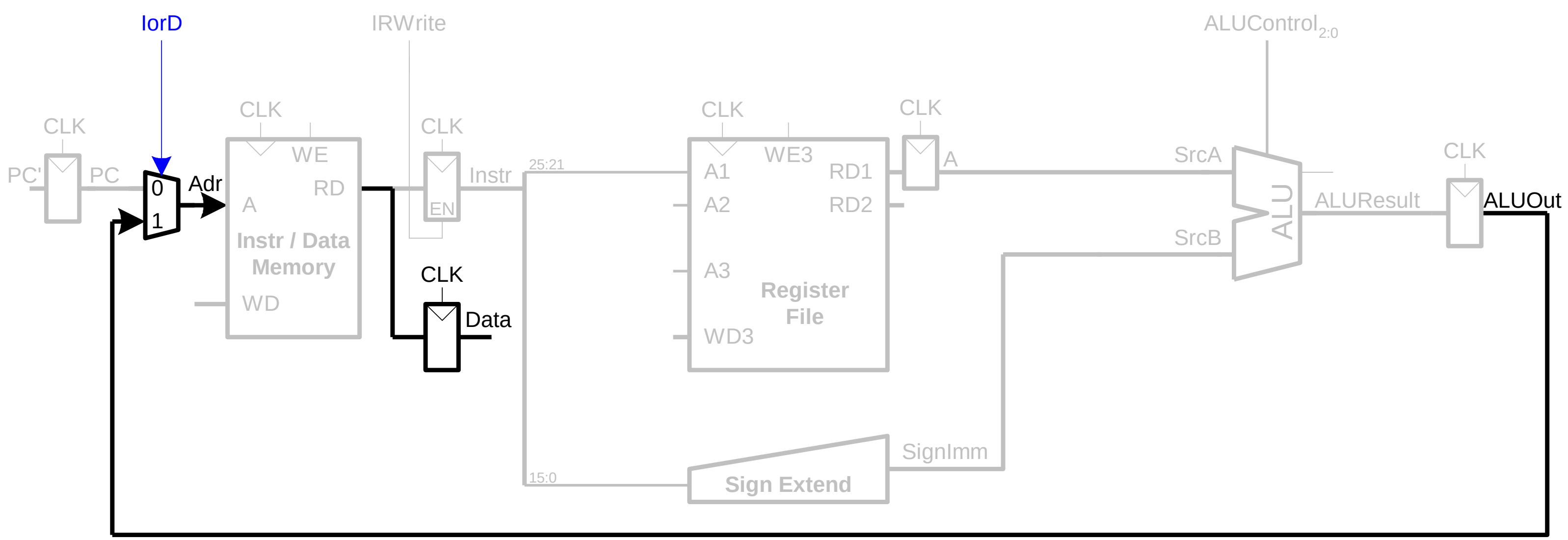
Multi-cycle Datapath: `lw` address



I-Type

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

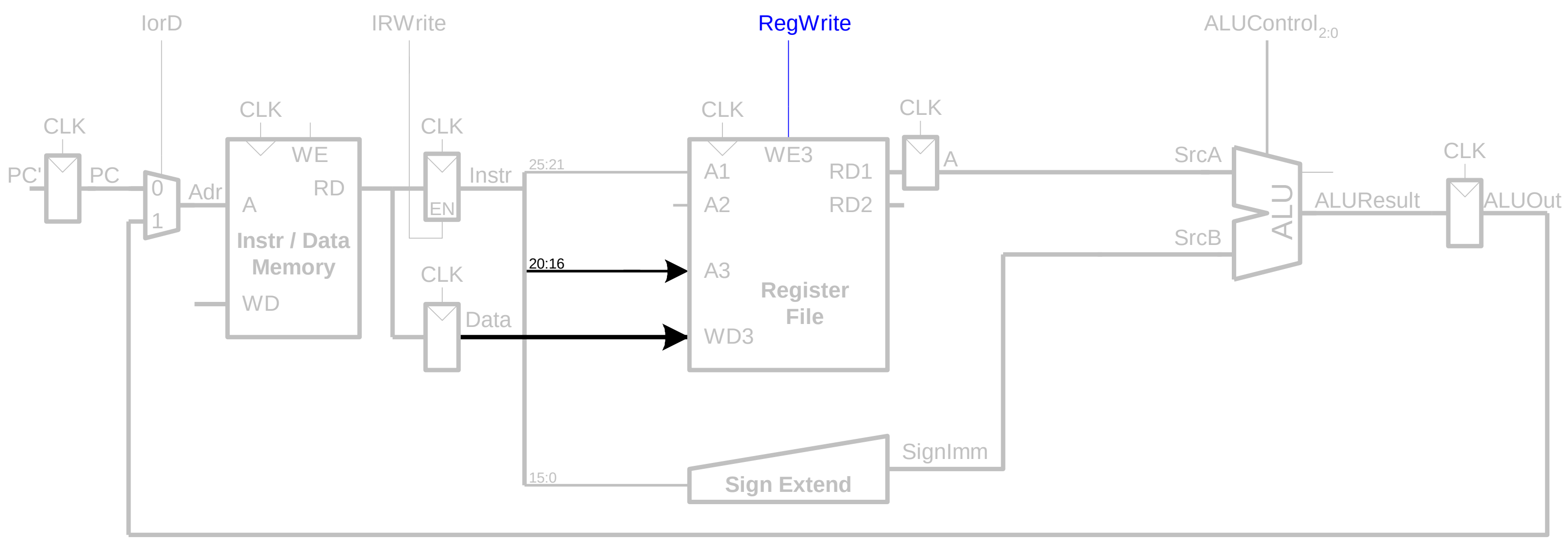
Multi-cycle Datapath: `lw` memory read



I-Type

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

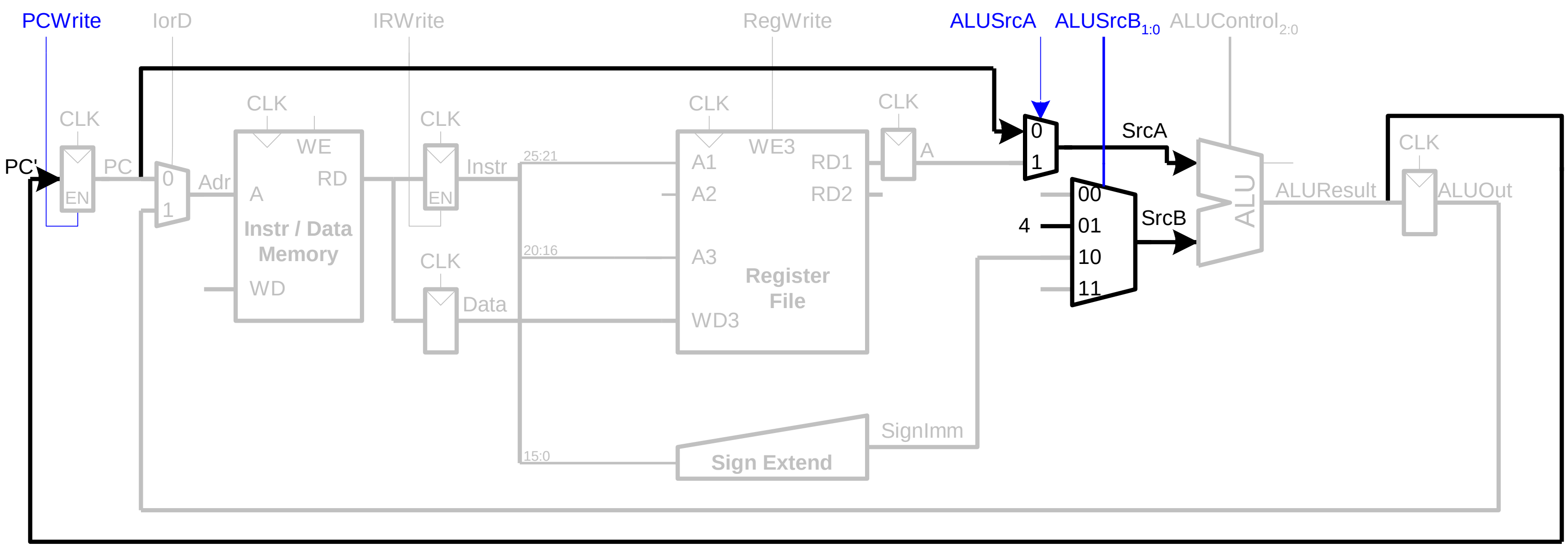
Multi-cycle Datapath: lw write register



I-Type

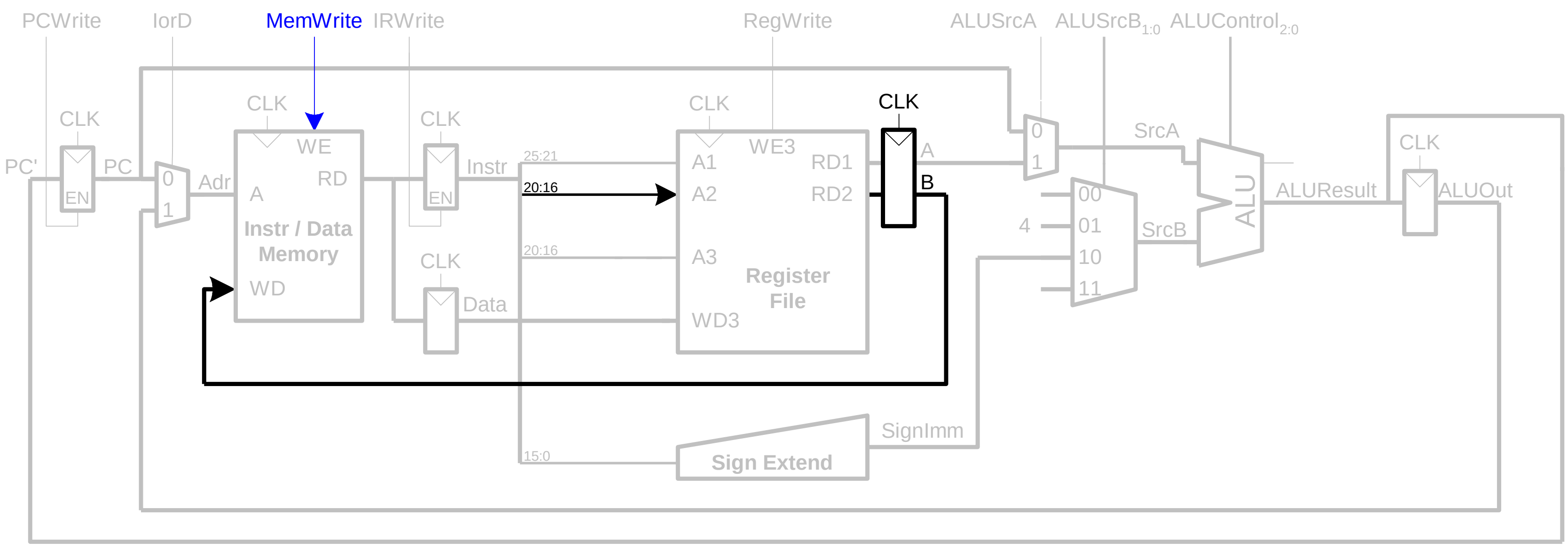
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Multi-cycle Datapath: increment PC



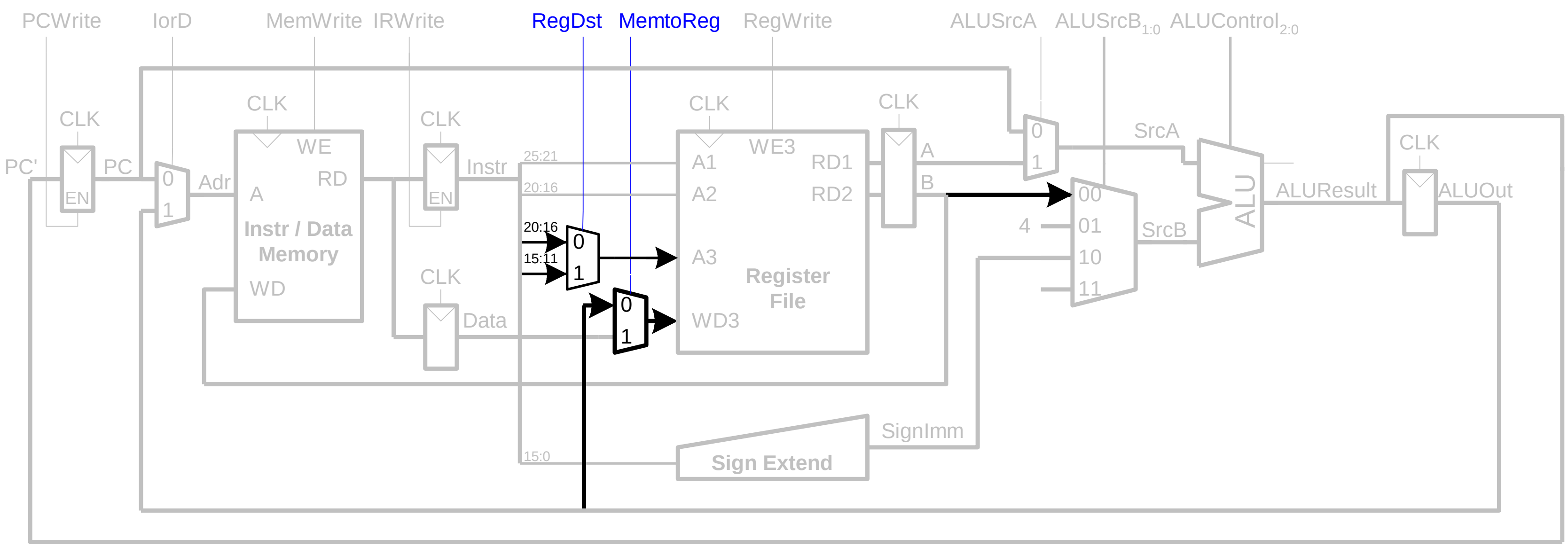
Multi-cycle Datapath: sw

- Write data in rt to memory



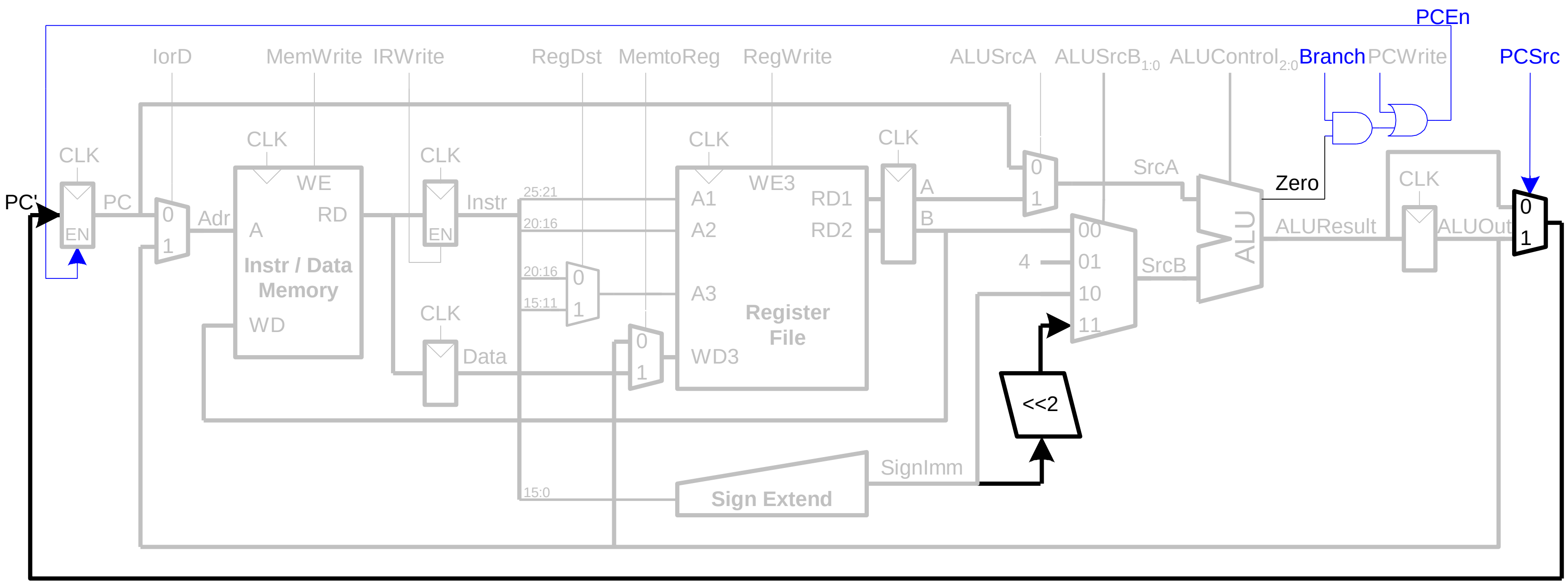
Multi-cycle Datapath: R-type Instructions

- **Read from rs and rt**
  - Write ALUResult to register file
  - Write to rd (instead of rt)

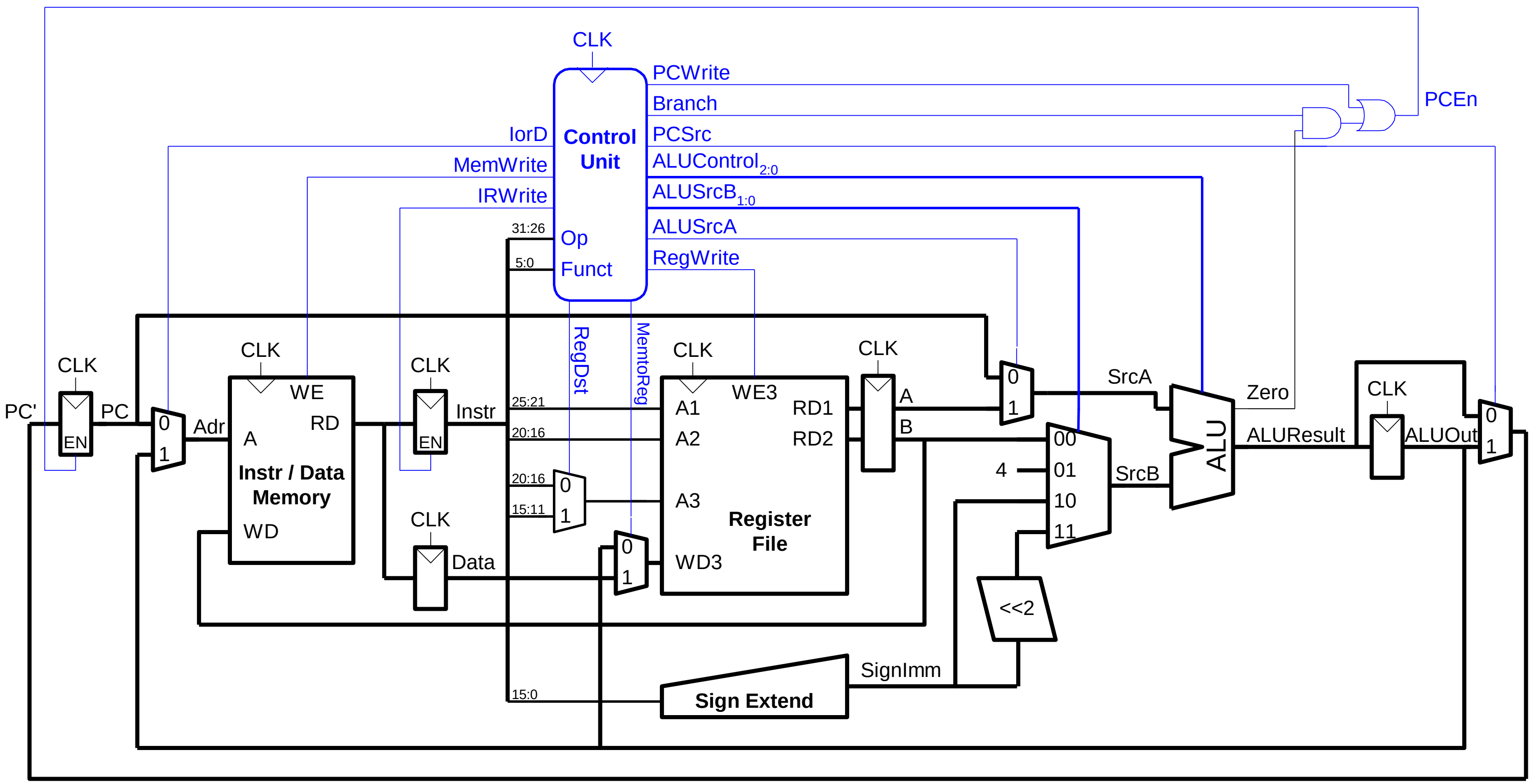


Multi-cycle Datapath: beq

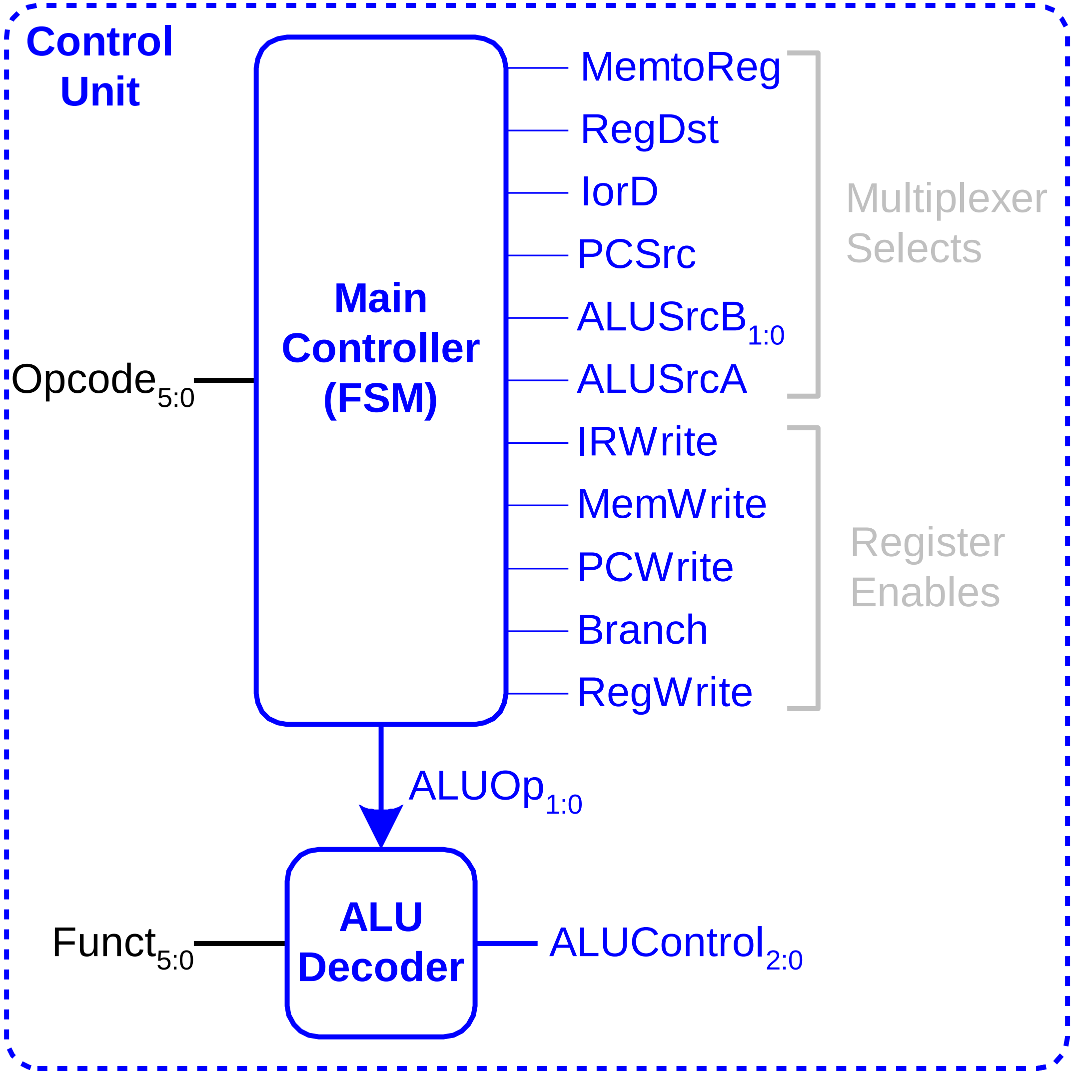
- **Determine whether values in rs and rt are equal**
  - Calculate branch target address:
    BTA = (sign-extended immediate << 2) + (PC+4)



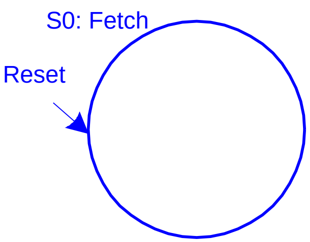
Complete Multi-cycle Processor

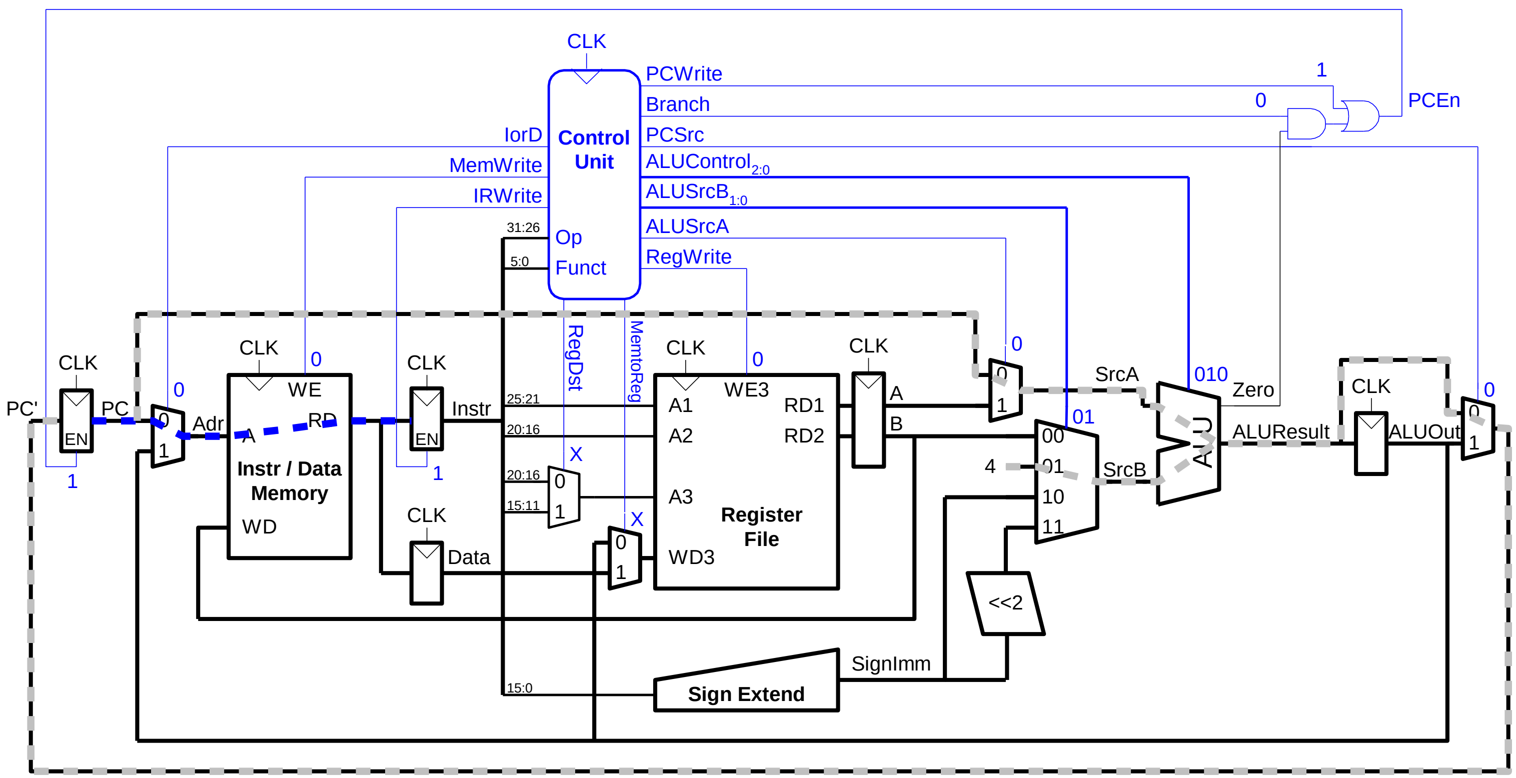


Control Unit

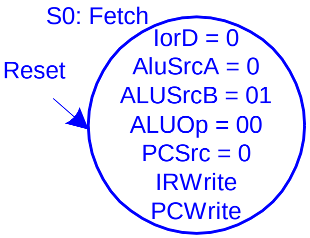


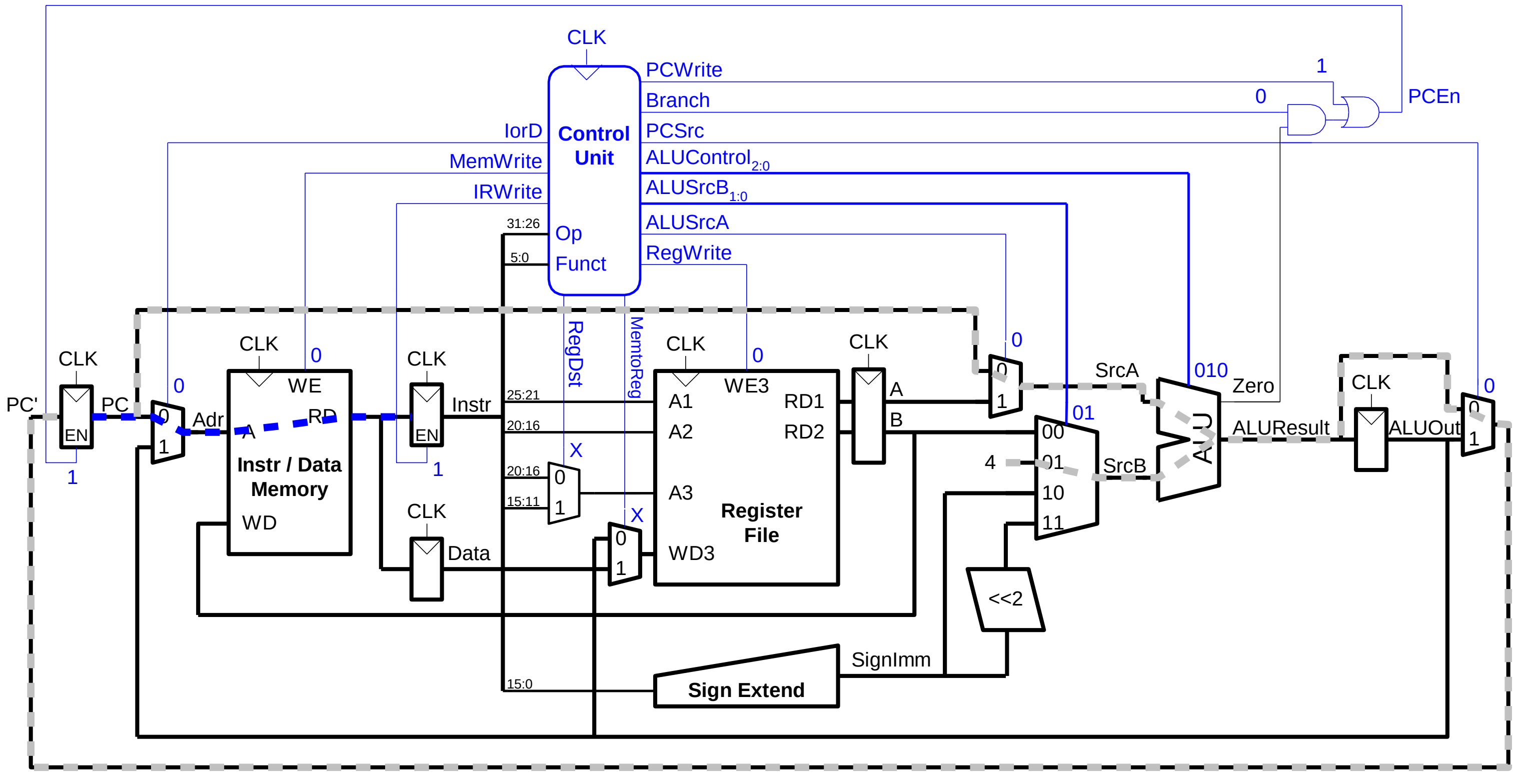
Main Controller FSM: Fetch



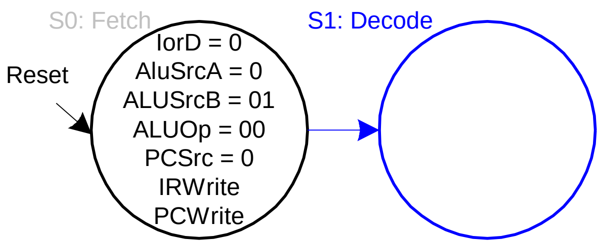


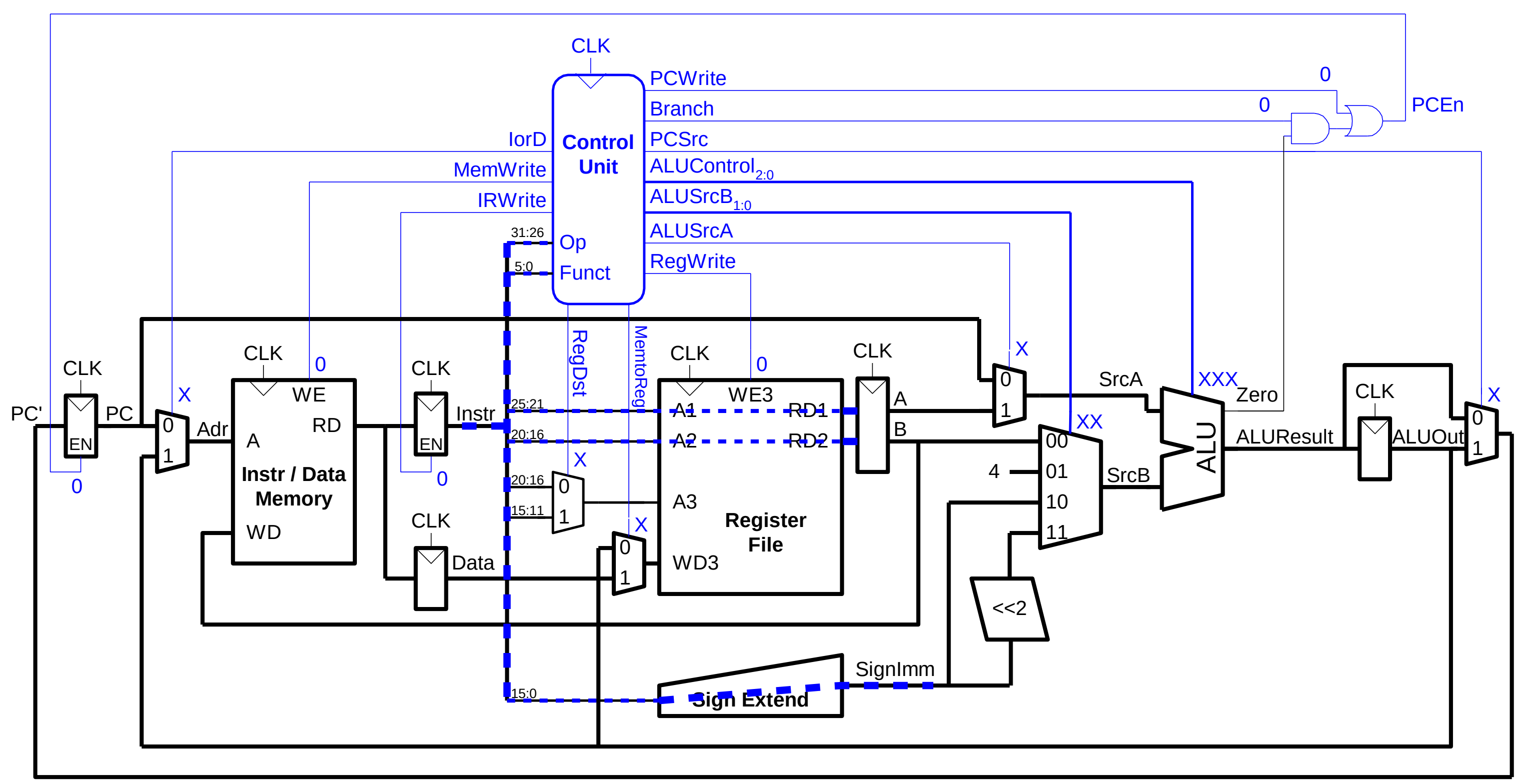
Main Controller FSM: Fetch



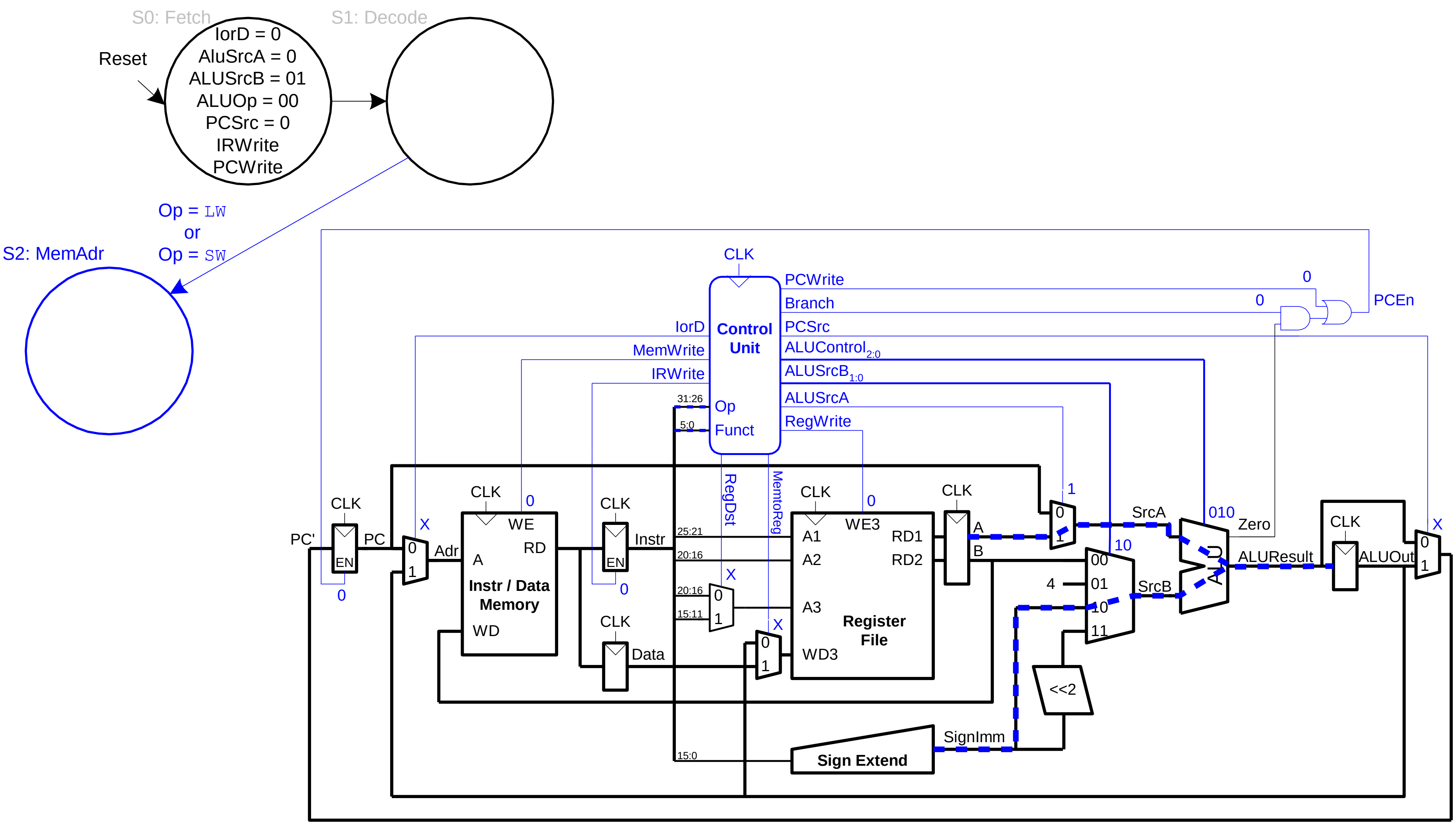


Main Controller FSM: Decode

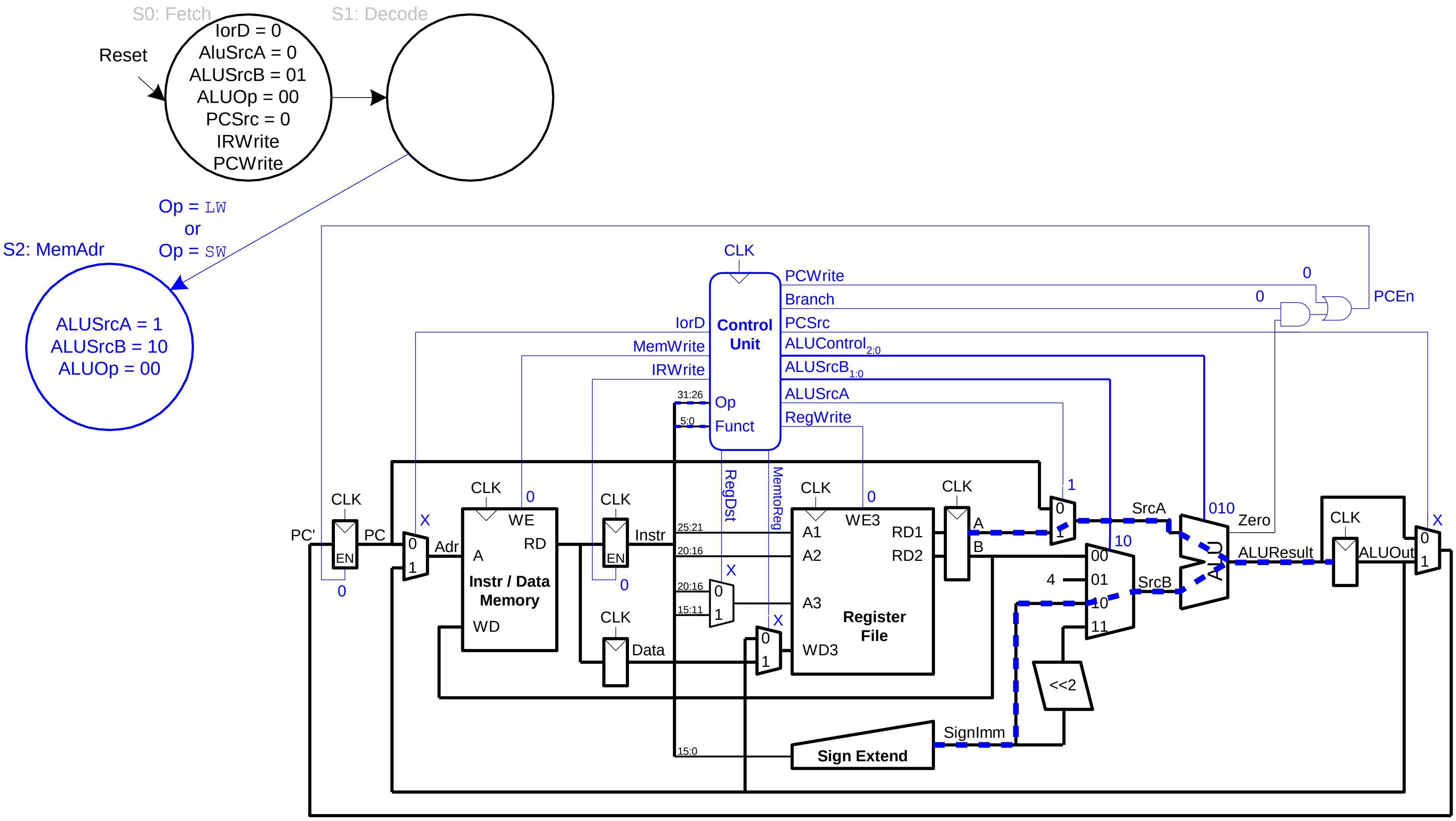




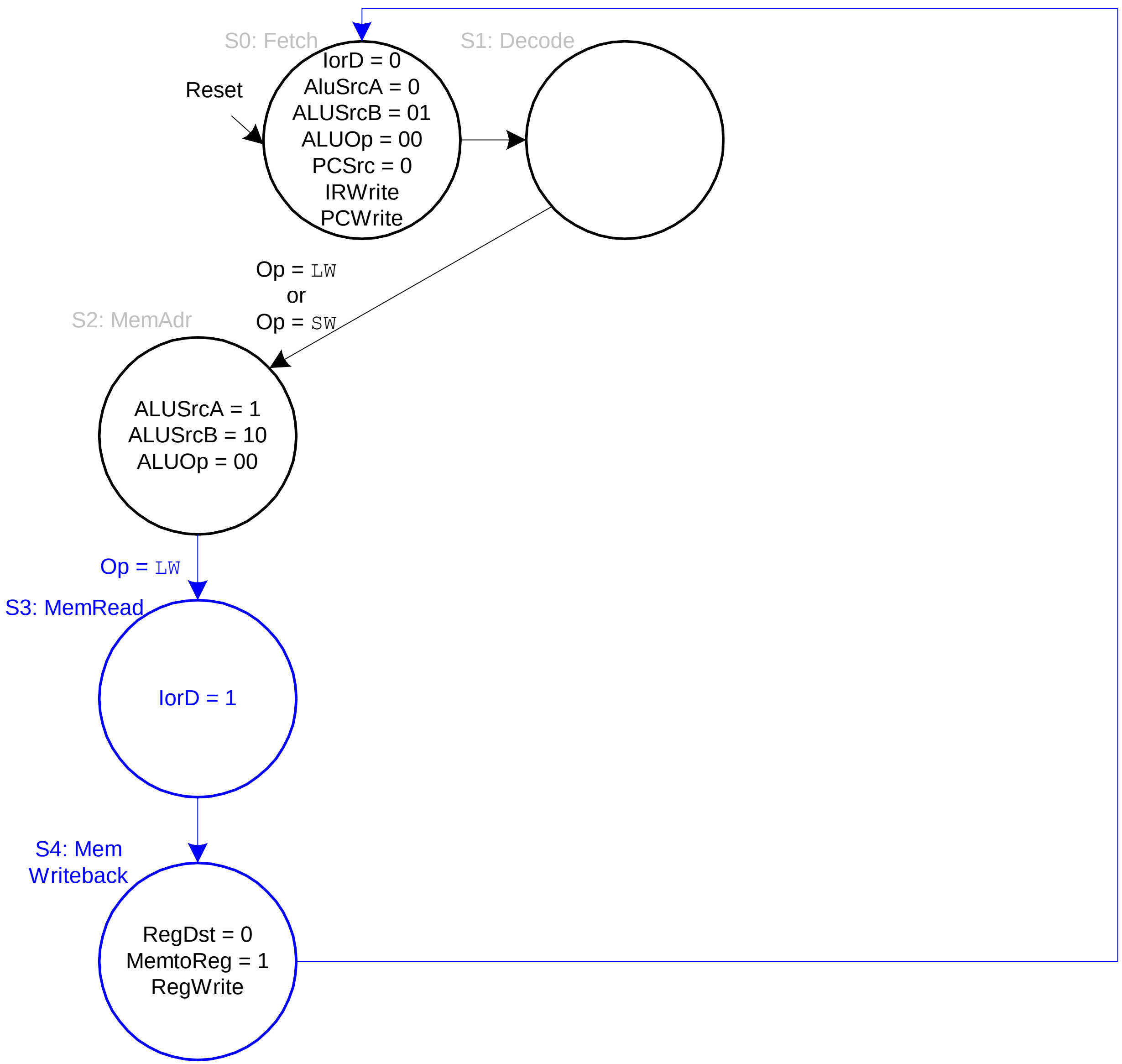
Main Controller FSM: Address Calculation



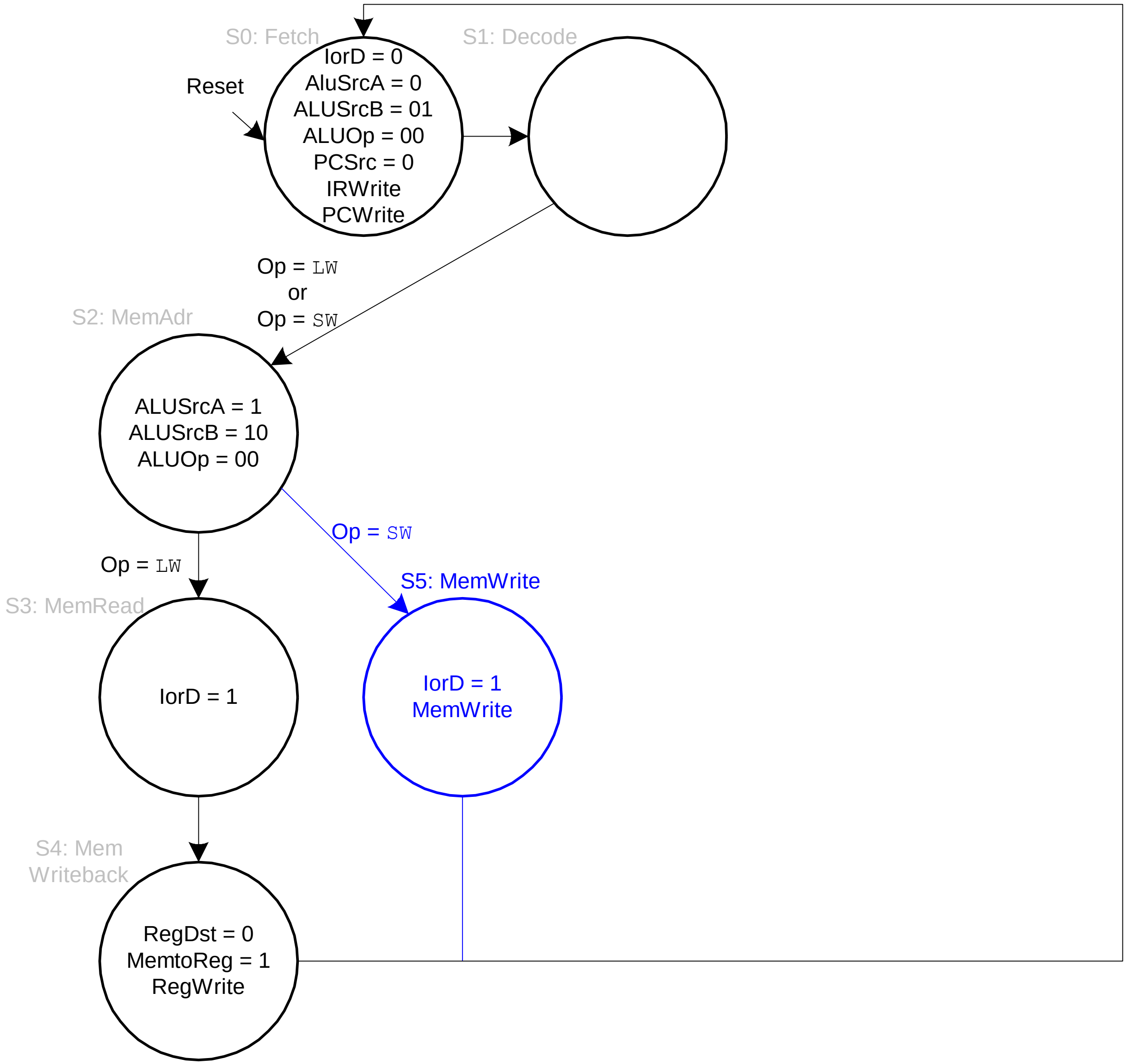
Main Controller FSM: Address Calculation



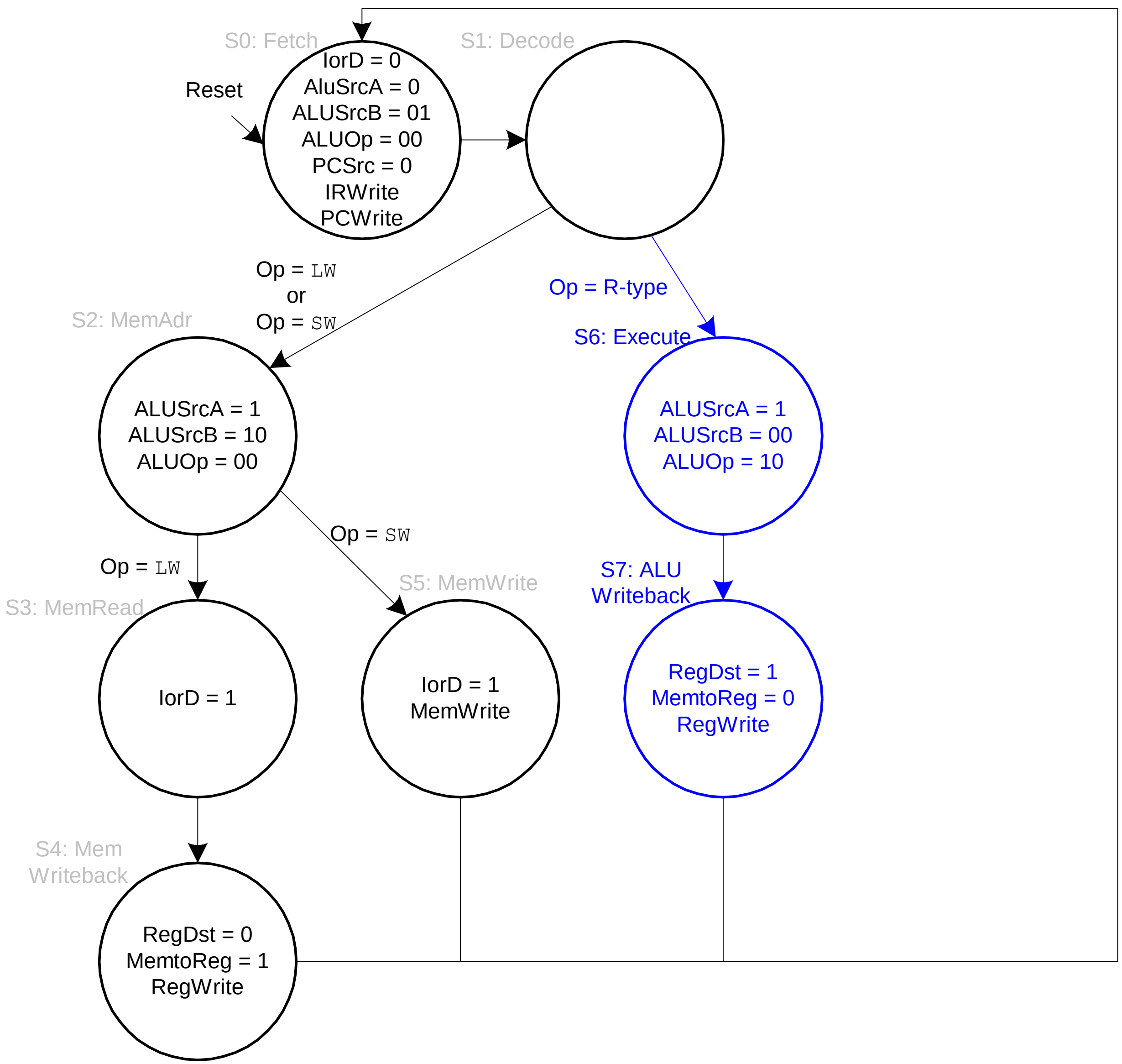
Main Controller FSM: lw



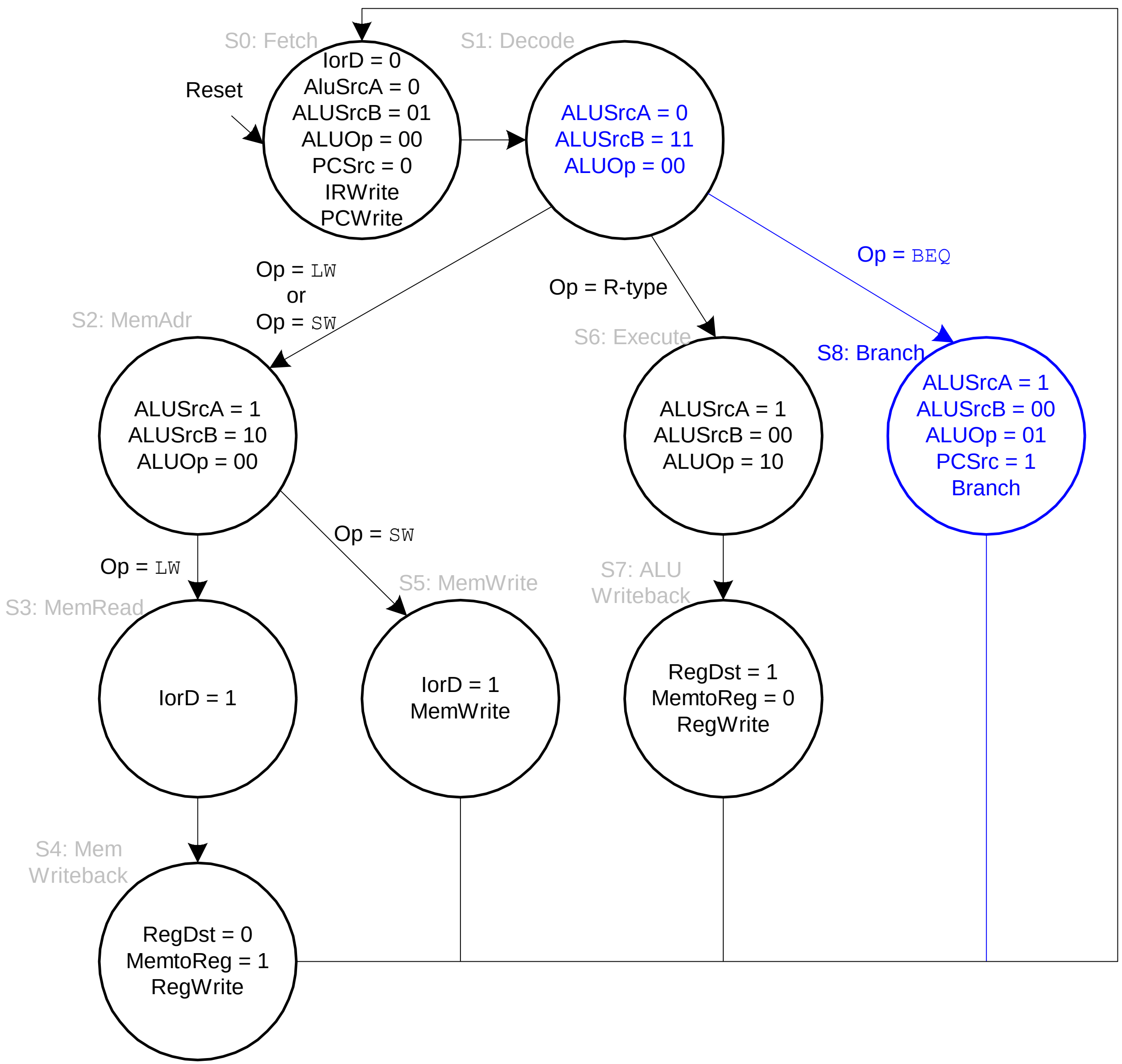
Main Controller FSM: `sw`



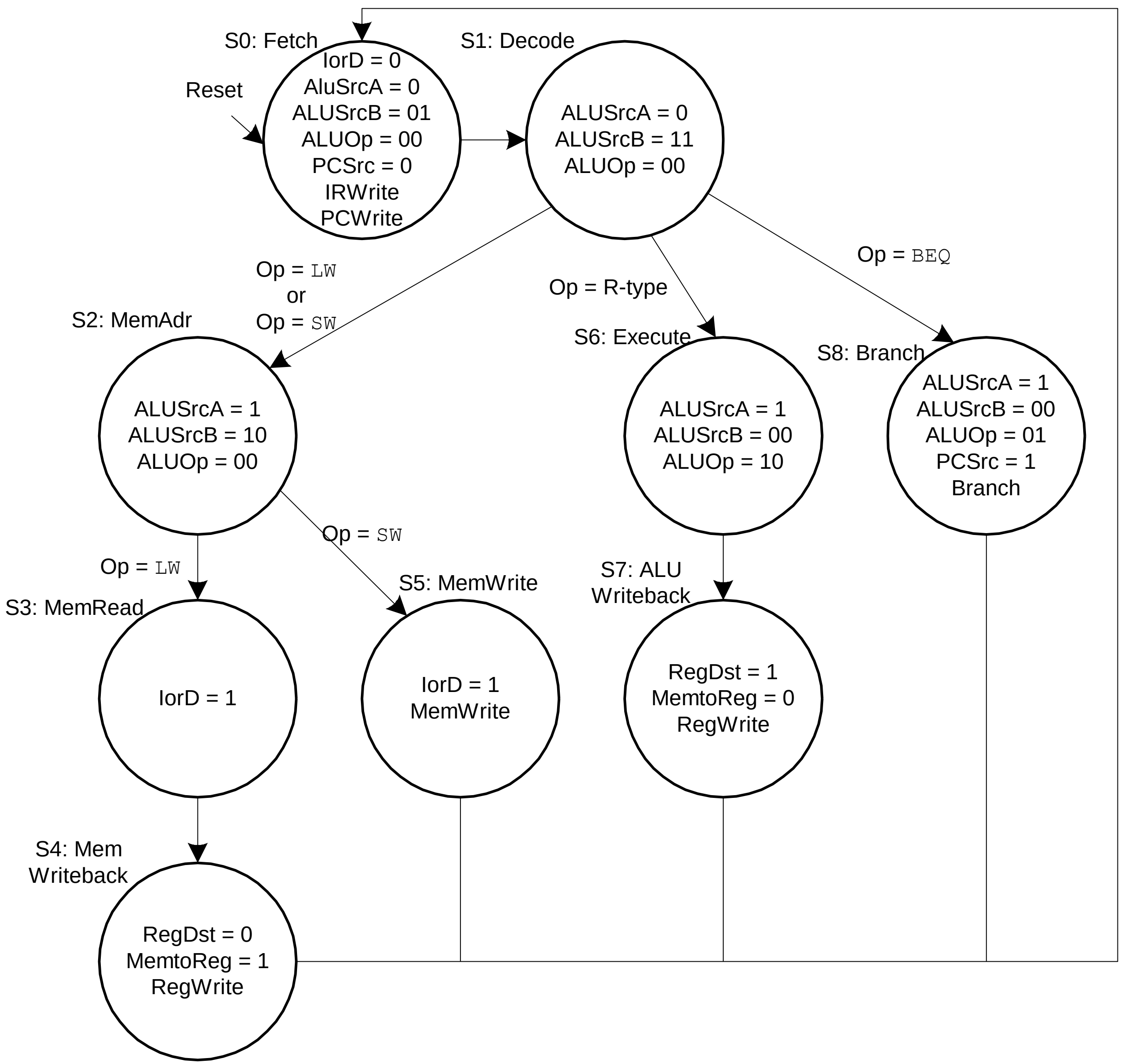
Main Controller FSM: R-Type



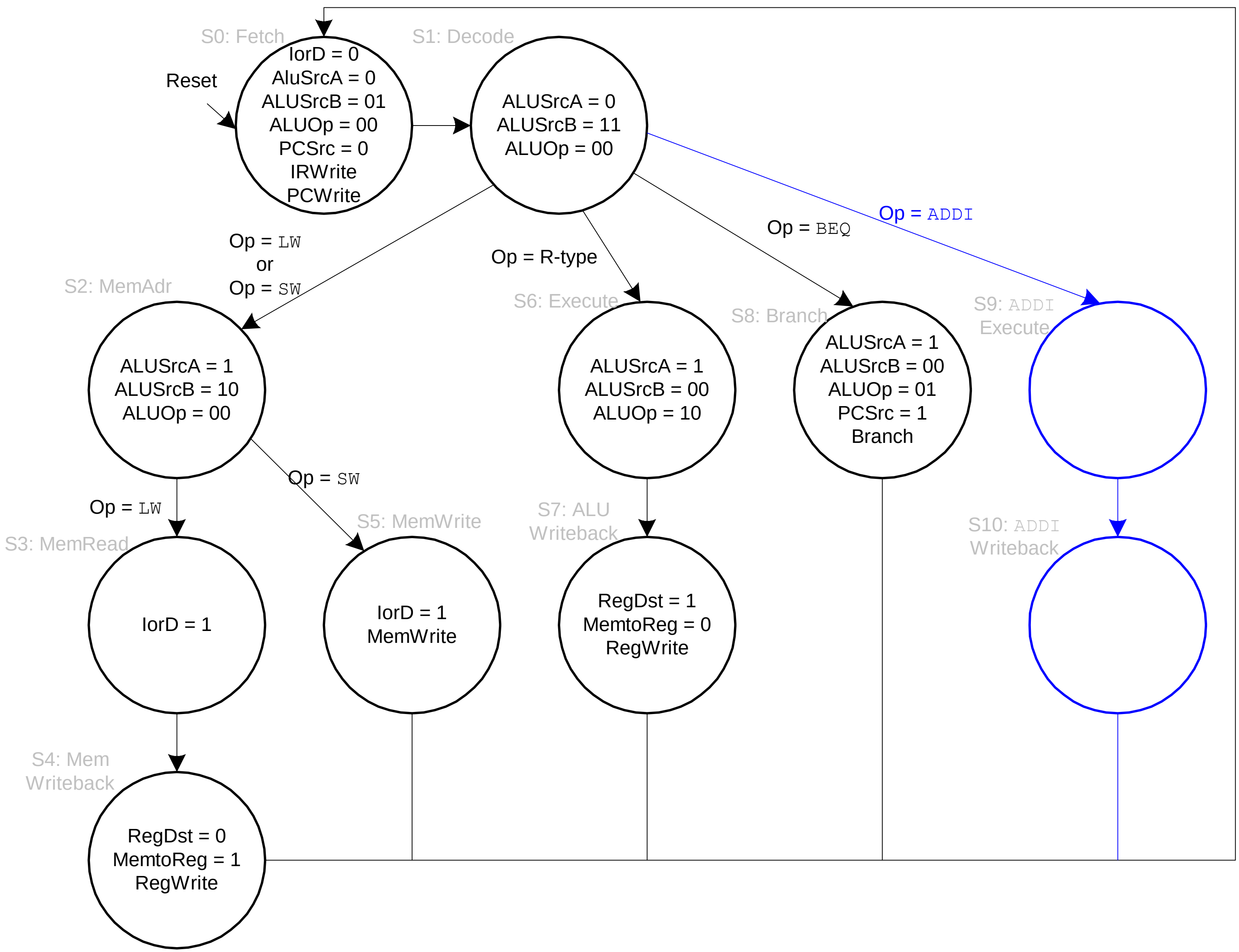
Main Controller FSM: beq



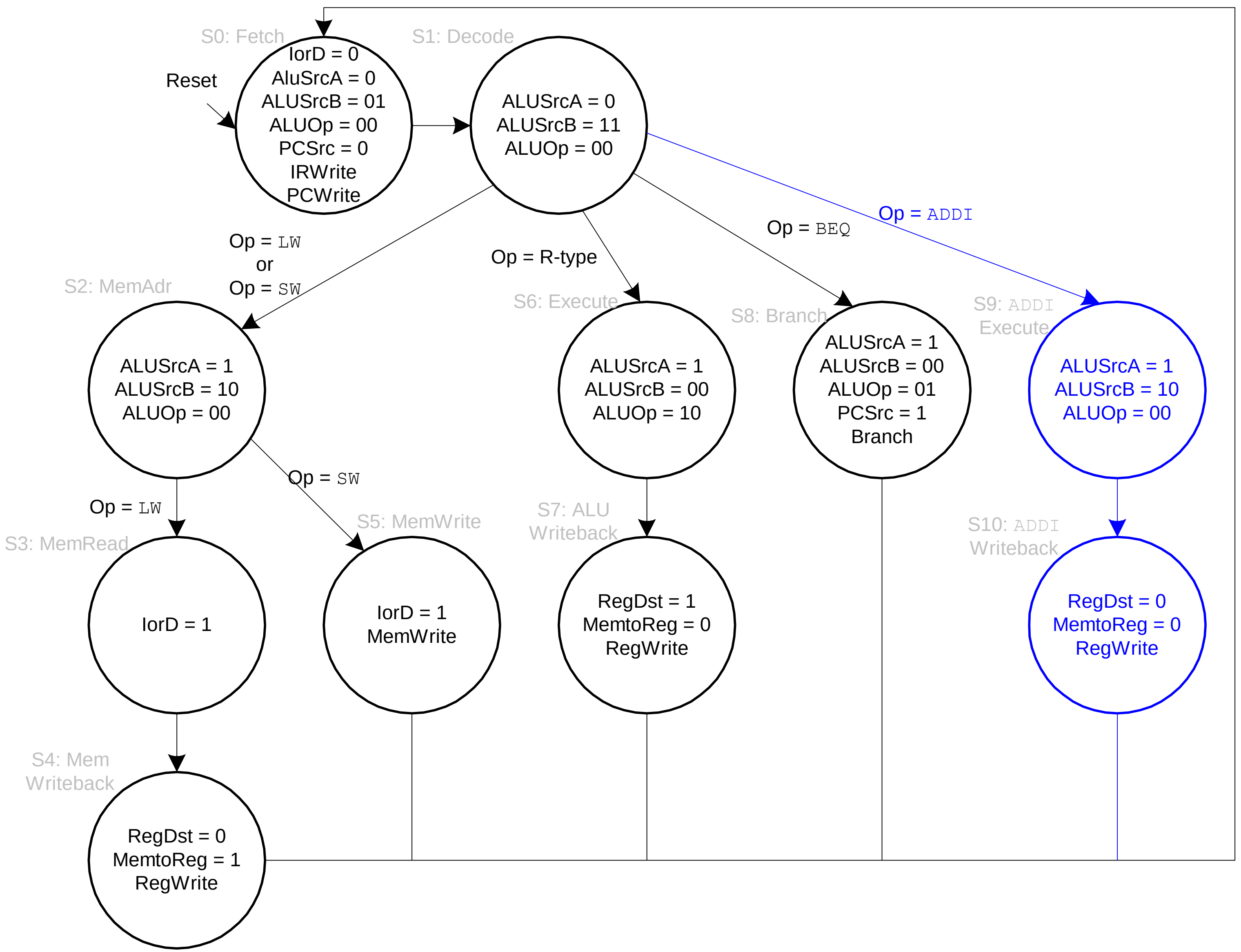
Complete Multi-cycle Controller FSM



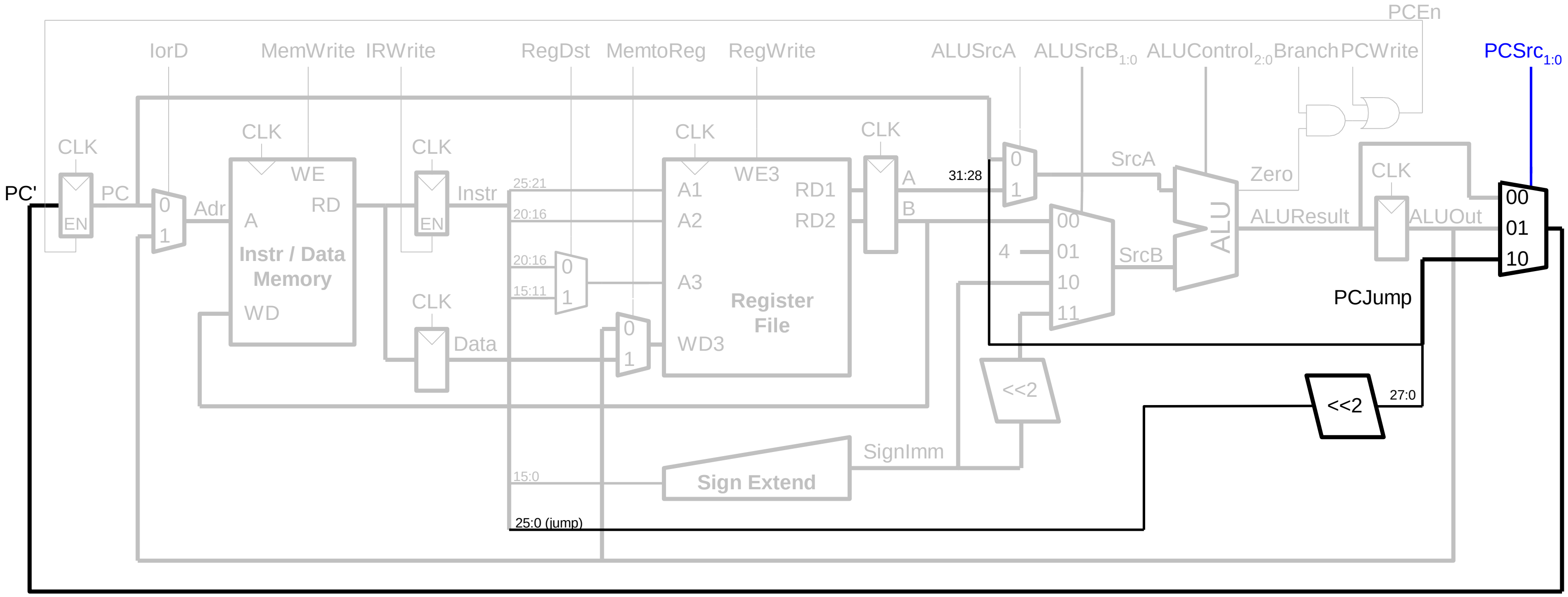
Main Controller FSM: addi



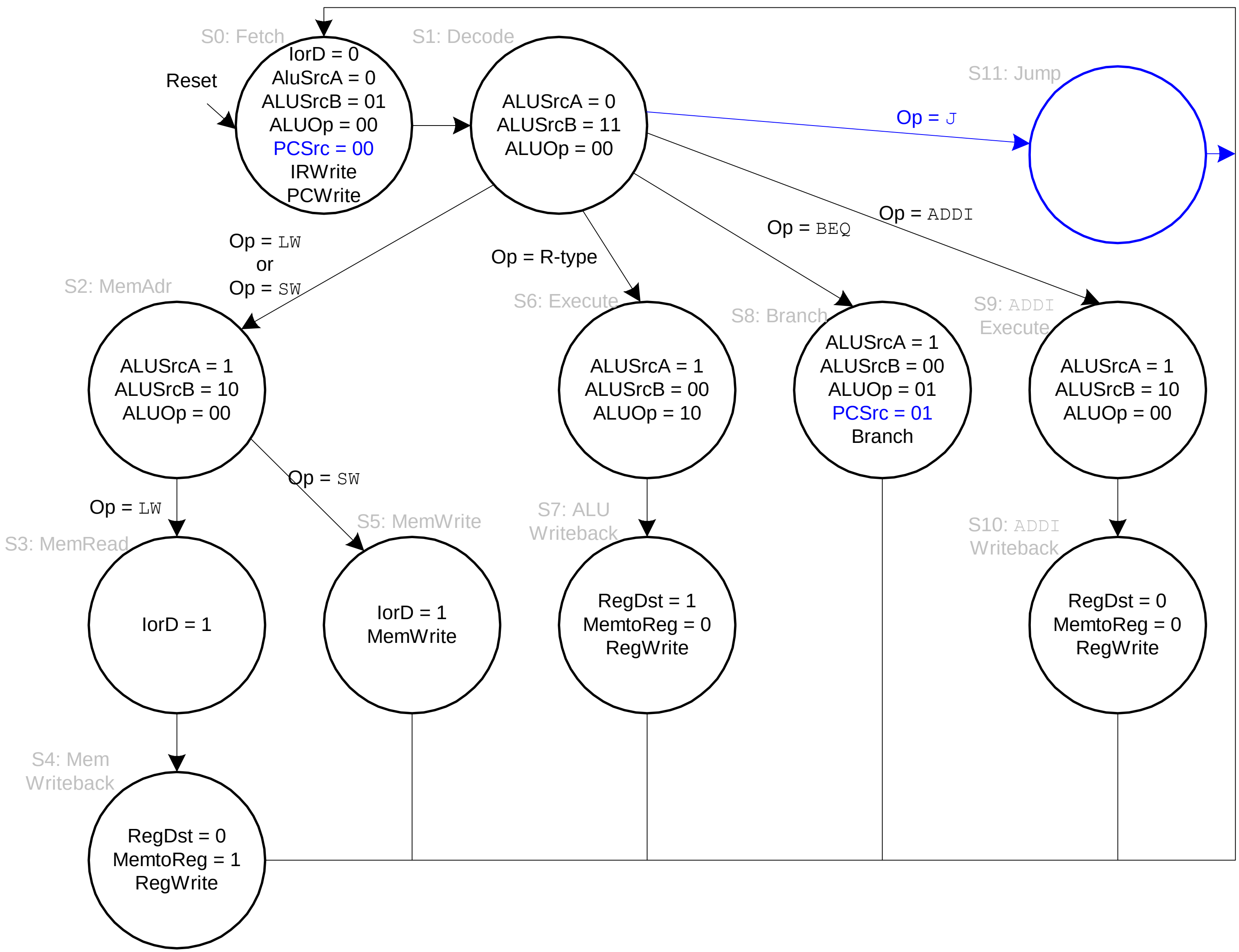
Main Controller FSM: addi



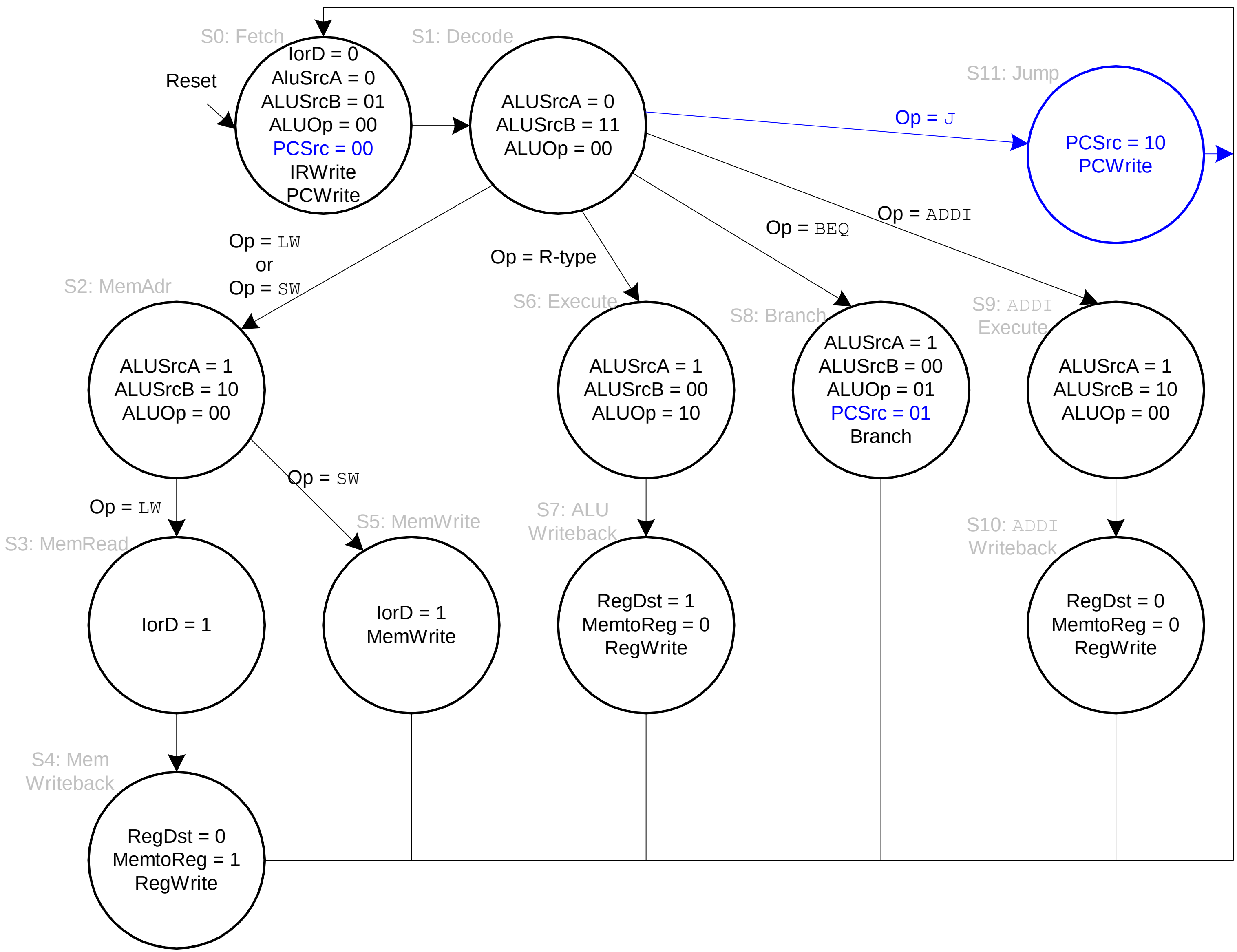
Extended Functionality: j



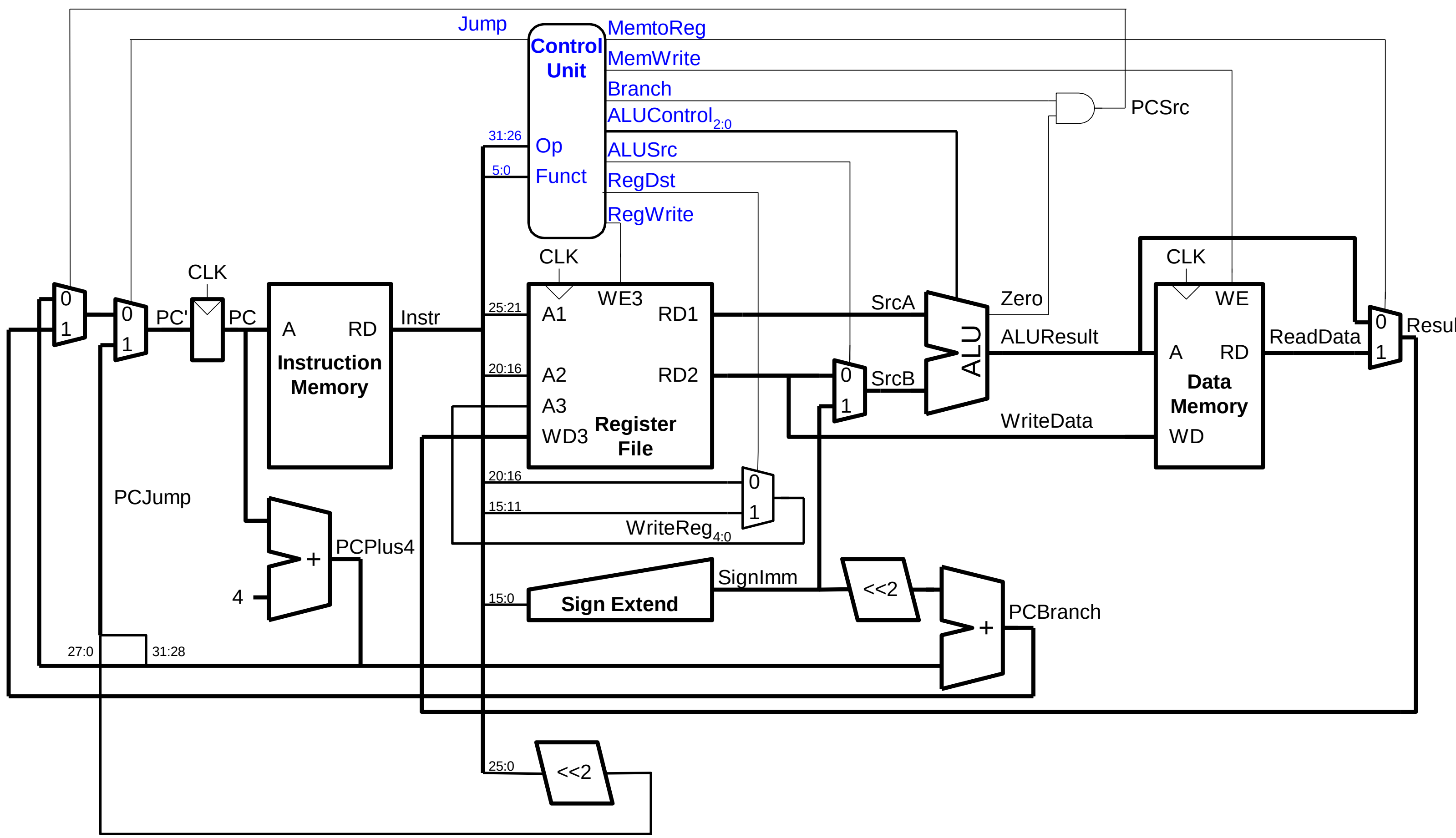
Control FSM: j



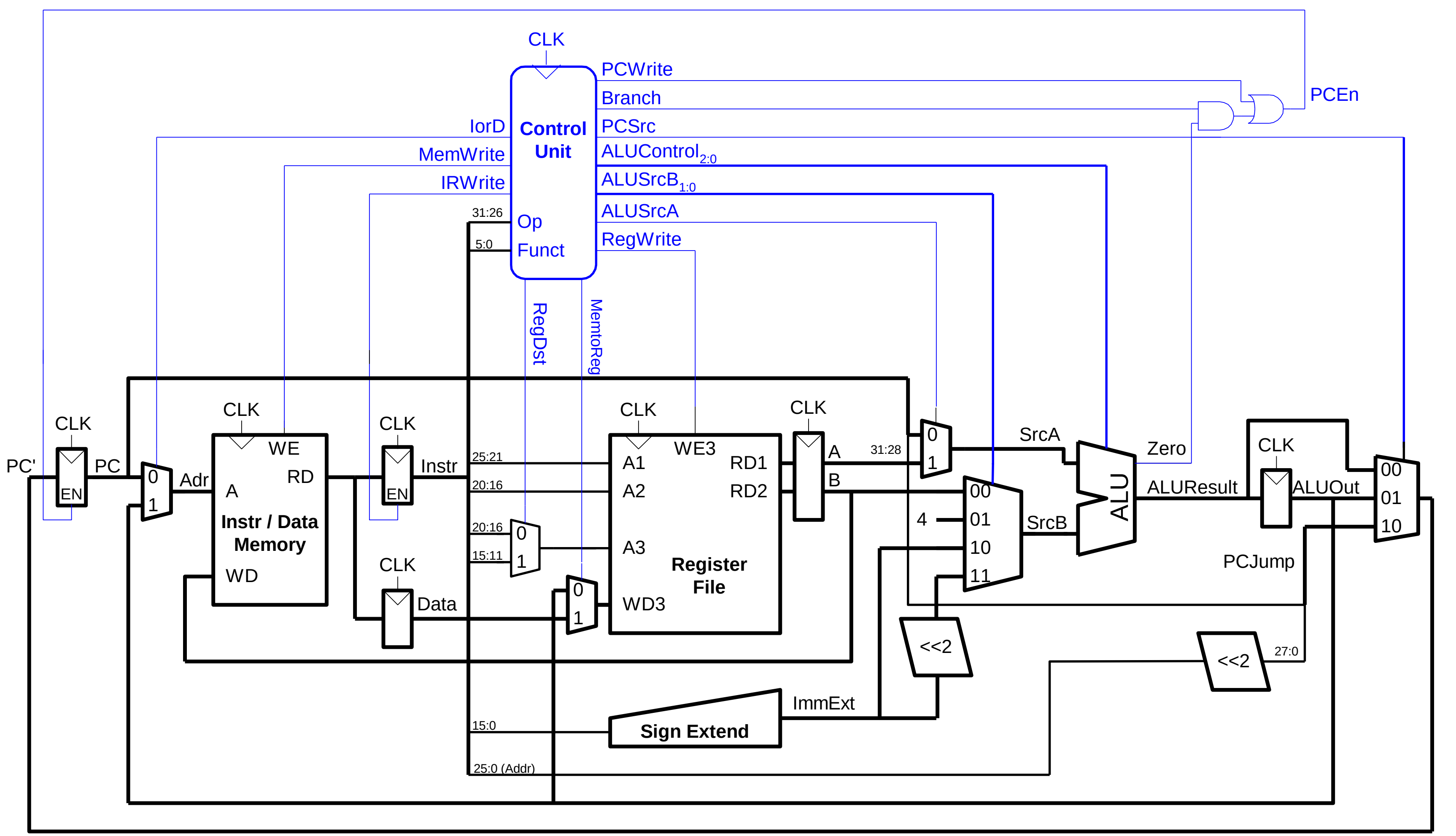
Control FSM: j



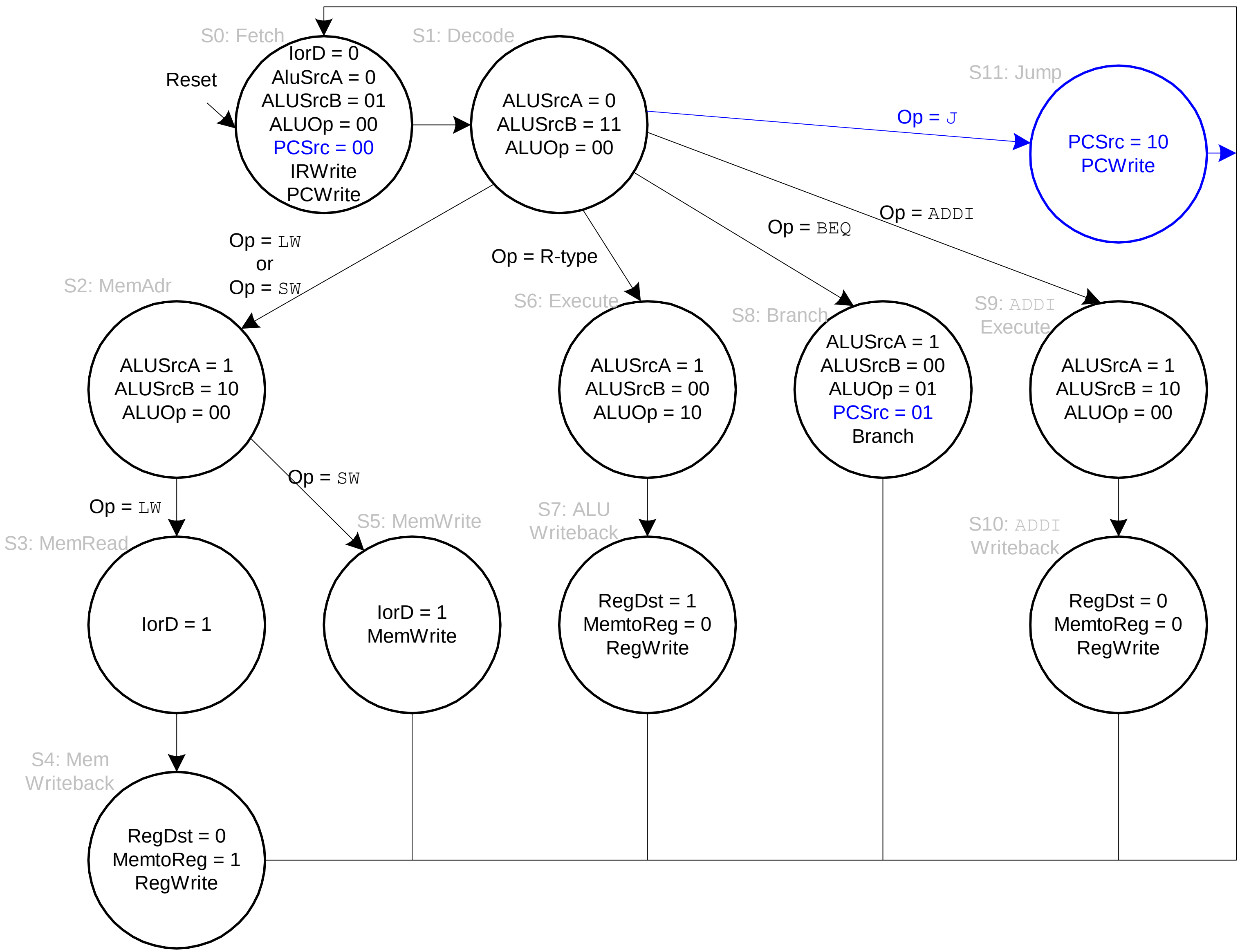
Review: Single-Cycle MIPS Processor



Review: Multi-Cycle MIPS Processor



Review: Multi-Cycle MIPS FSM



What is the shortcoming of this design?

What does this design assume about memory?

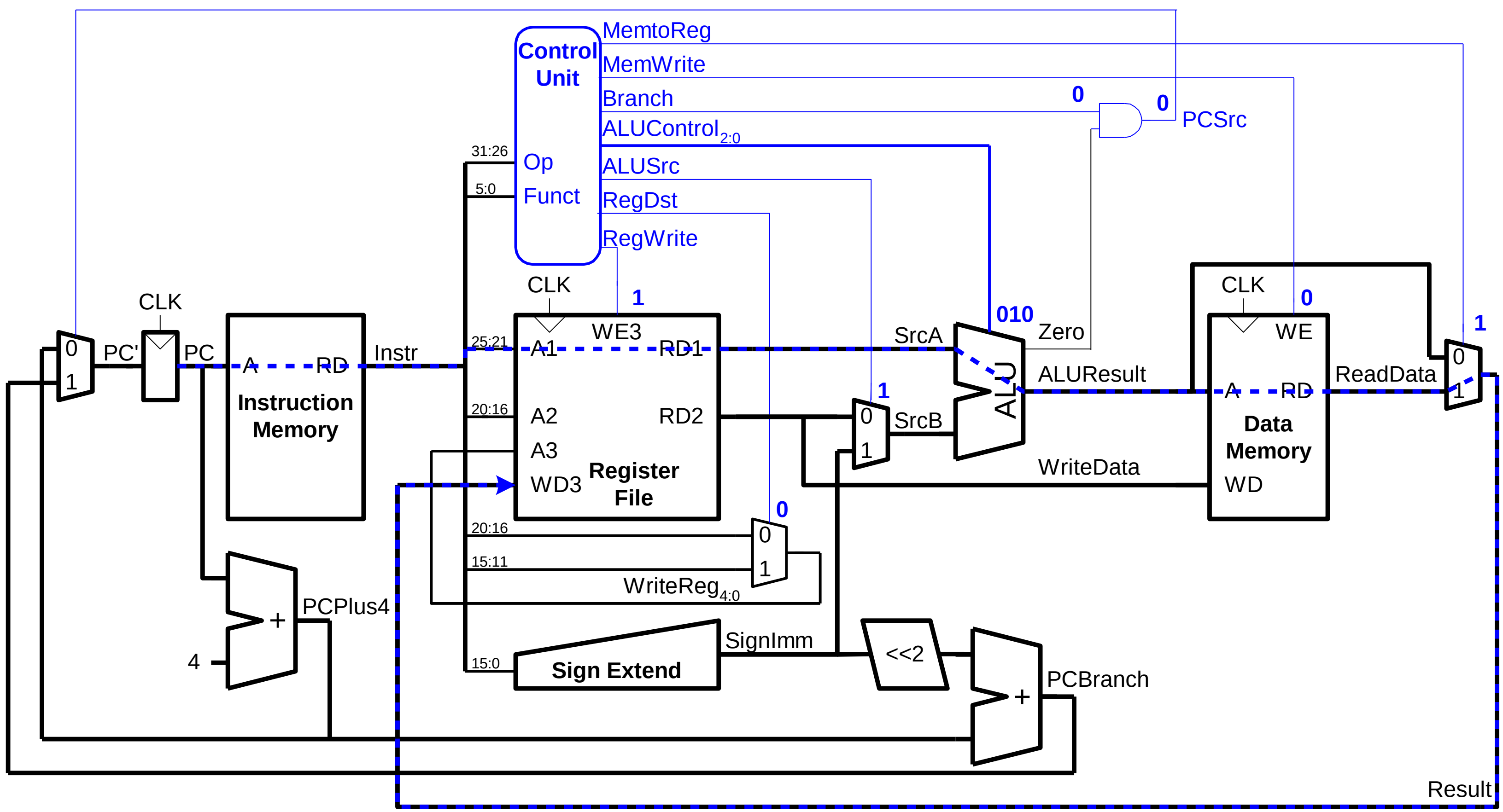
What If Memory Takes > One Cycle?

- 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

More on Performance Analysis

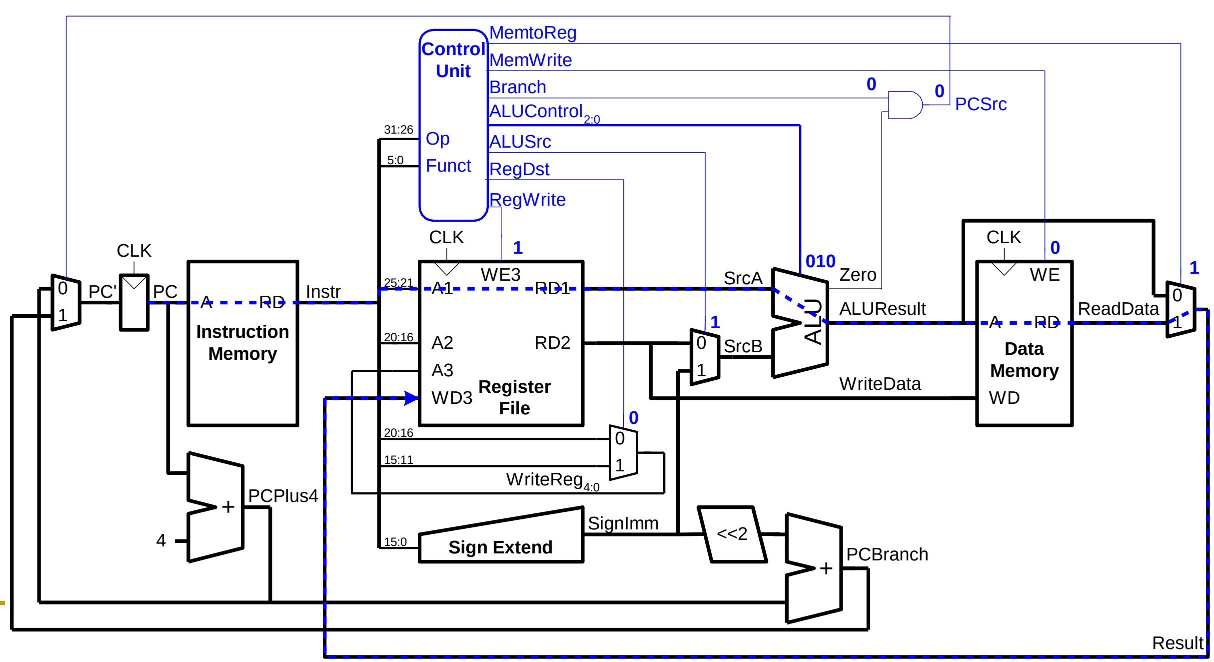
Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



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

$$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$$

$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$

$$= 925 \text{ ps}$$

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:

Execution Time $= \# \text{ instructions} \times \text{CPI} \times T_c$

$= (100 \times 10^9)(1)(925 \times 10^{-12}\text{ s})$

$= 92.5$ seconds

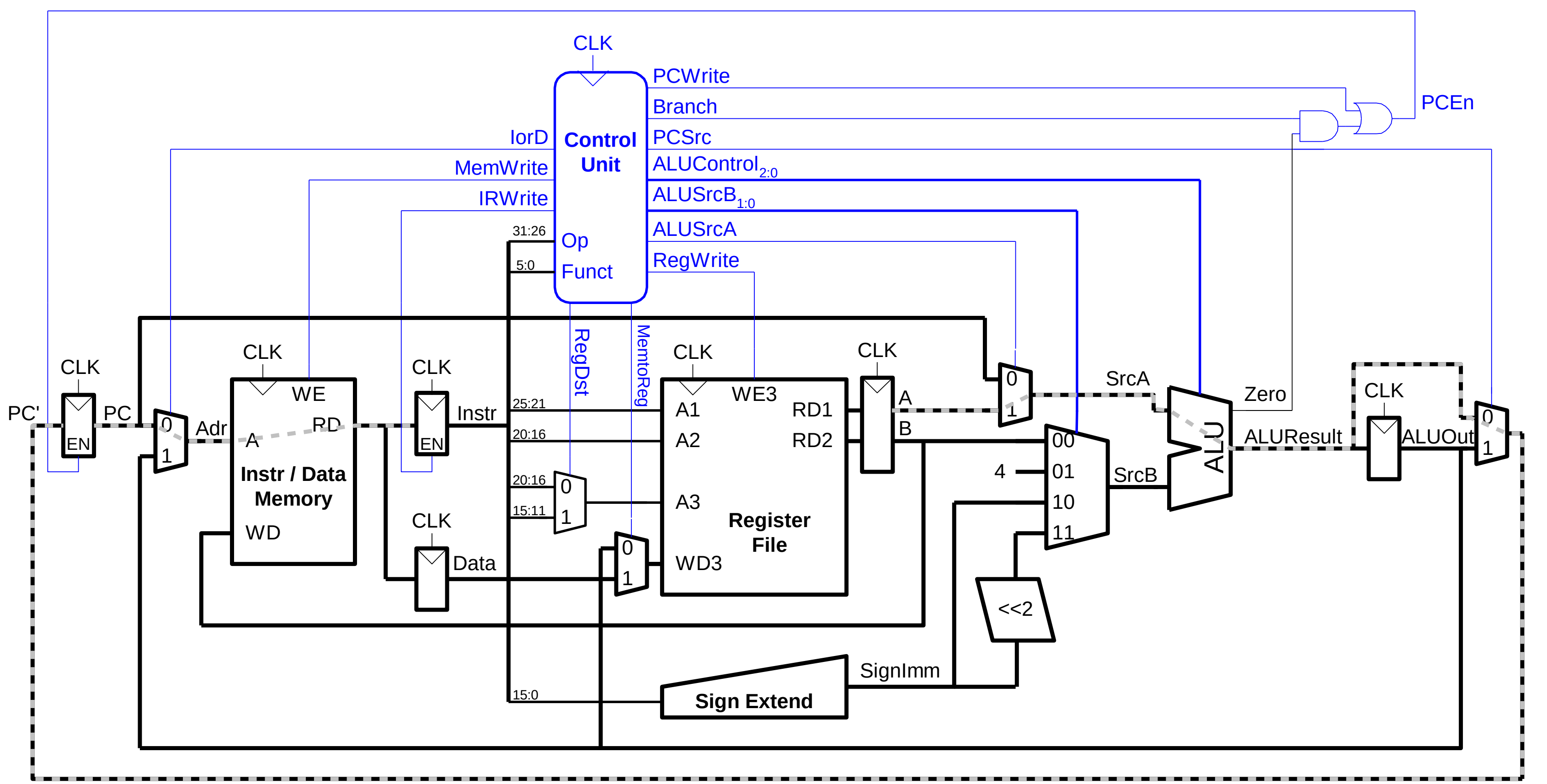
Multi-Cycle Performance: CPI

- 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:

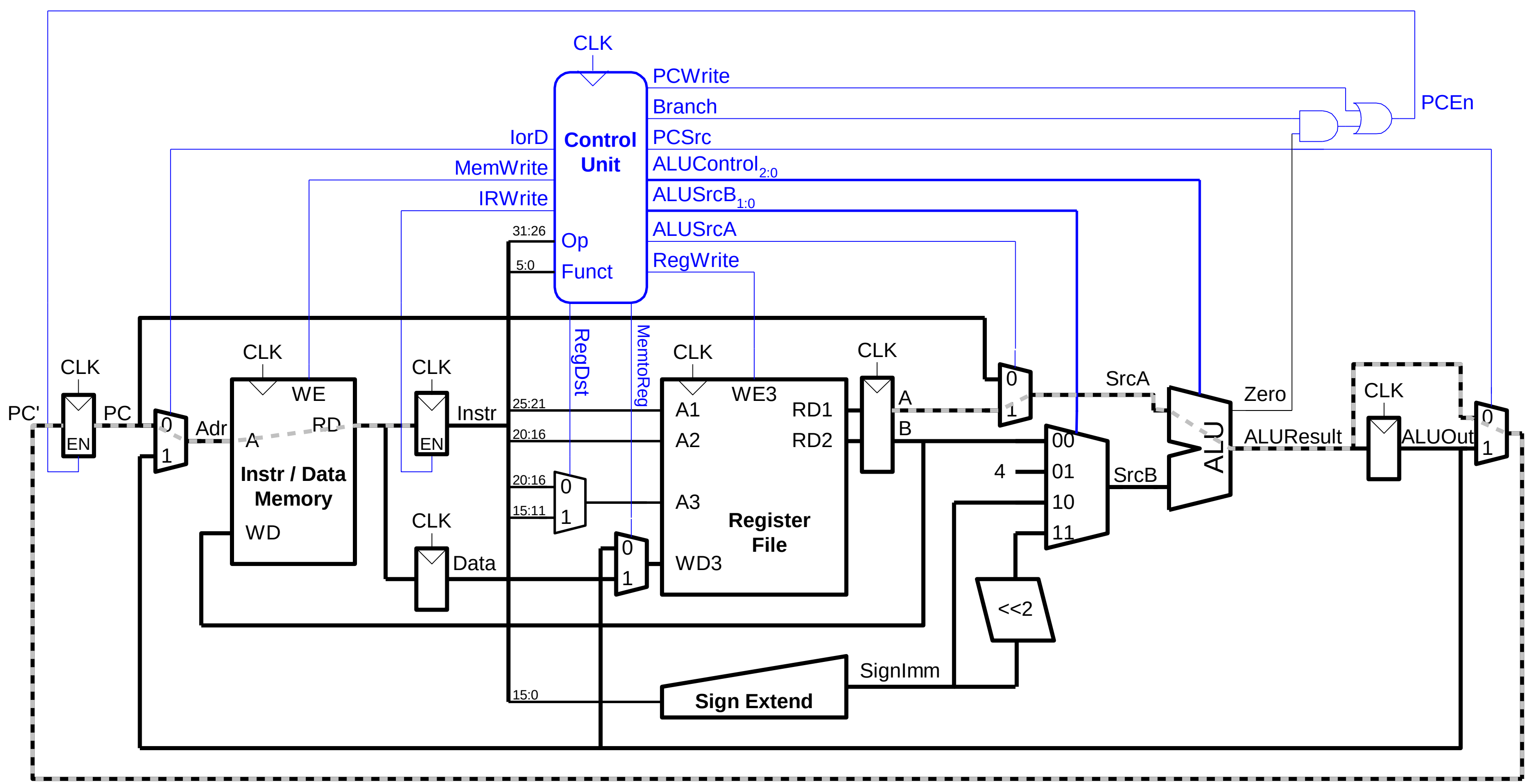
  $T_c =$



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

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

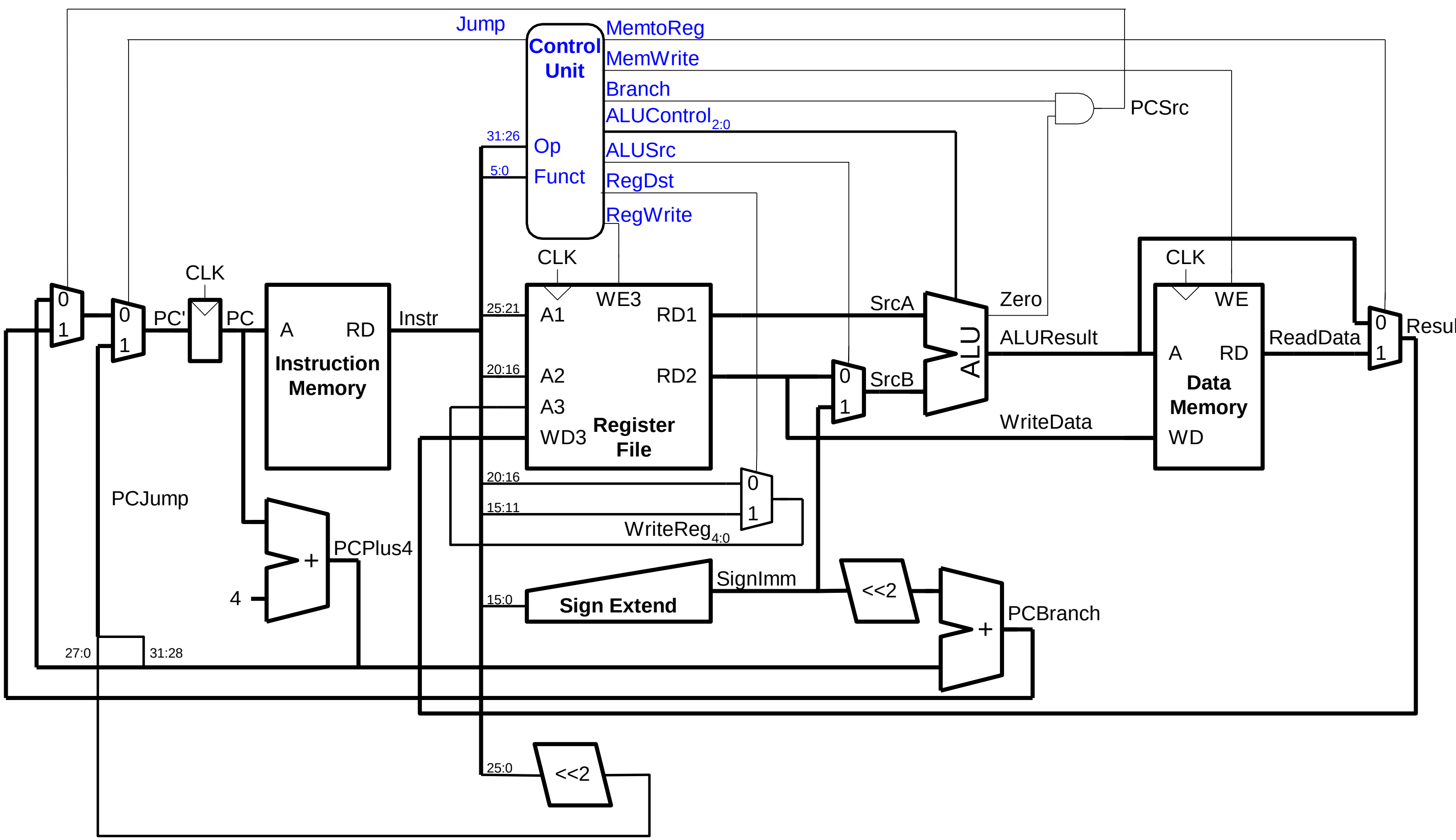
$$= [30 + 25 + 250 + 20] \text{ ps}$$

$$= 325 \text{ ps}$$

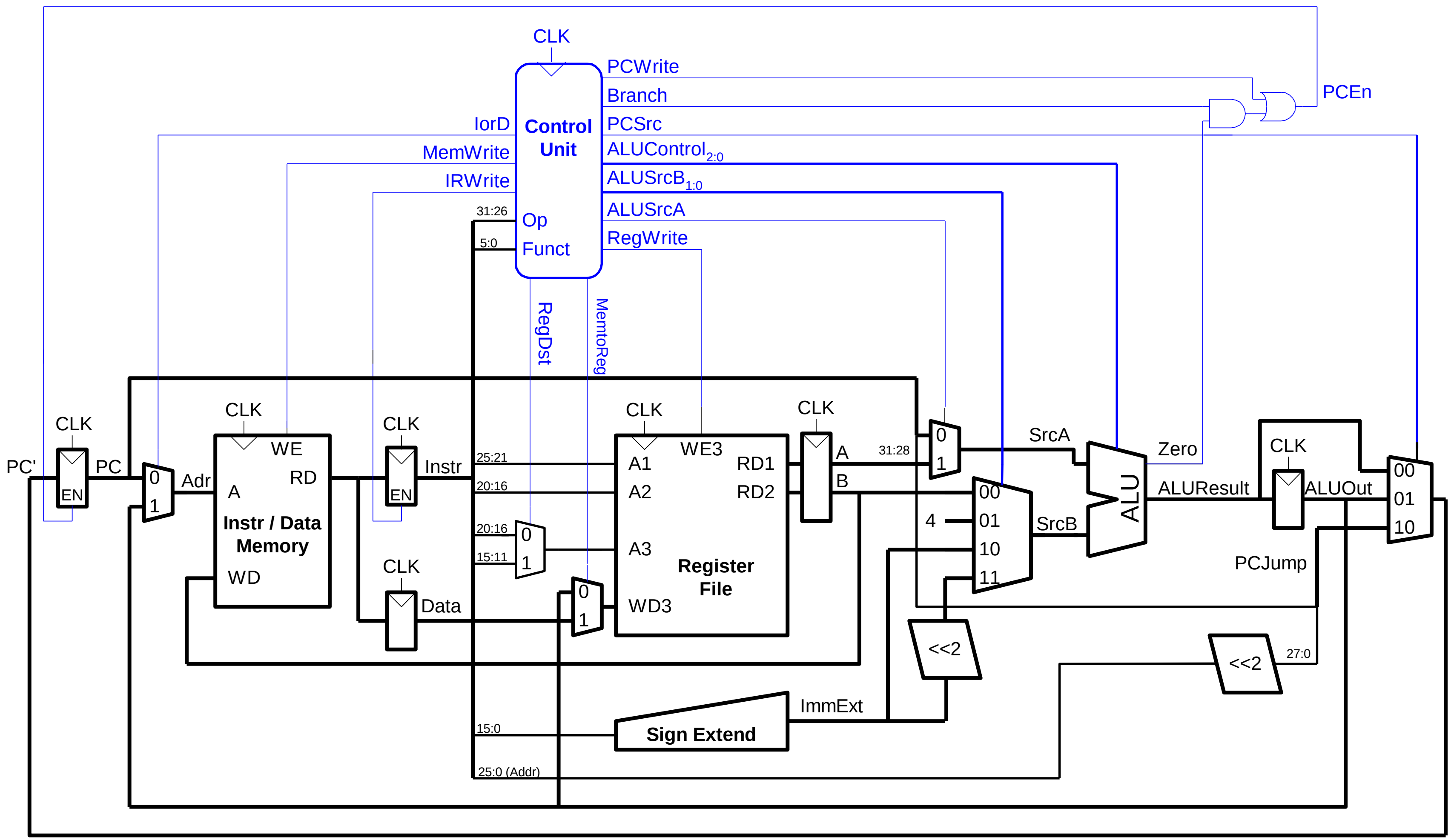
Multi-Cycle Performance Example

- For a program with 100 billion instructions executing on a multi-cycle MIPS processor
  - CPI = 4.12
  - T_c = 325 ps
- *Execution Time* $= (\text{\# instructions}) \times \text{CPI} \times T_c$
  $= (100 \times 10^9)(4.12)(325 \times 10^{-12})$
  $= 133.9$ 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?

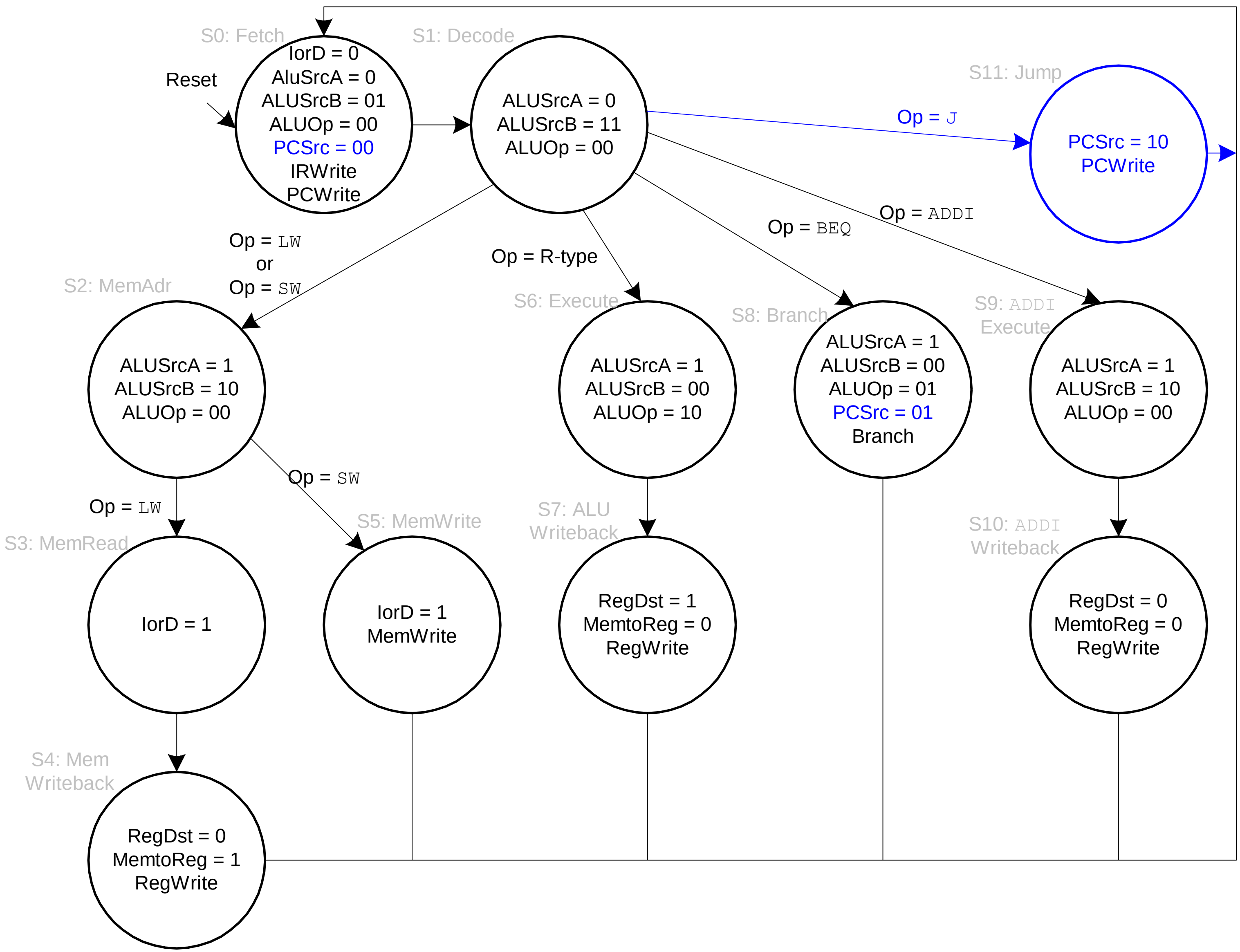
Review: Single-Cycle MIPS Processor



Review: 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

Design of Digital Circuits
Lecture 11: Microarchitecture

Prof. Onur Mutlu
ETH Zurich
Spring 2018
28 March 2019

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
```

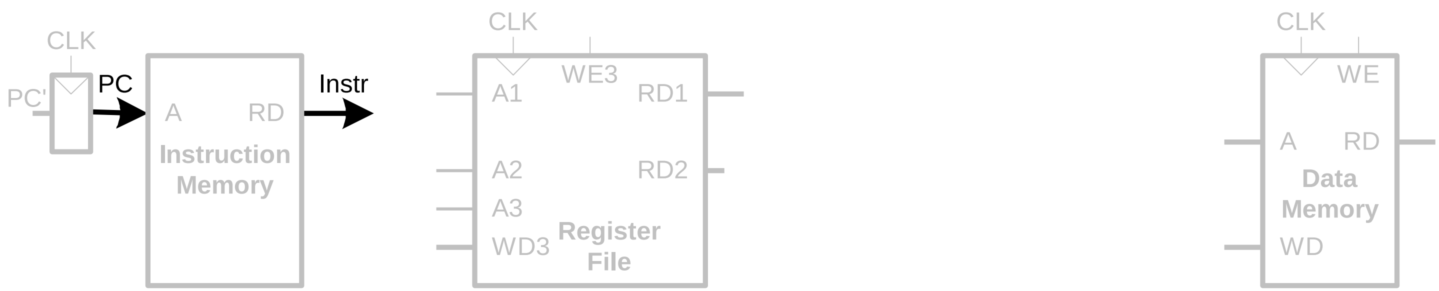
Data Memory Example

- **Will be used to store the bulk of data**

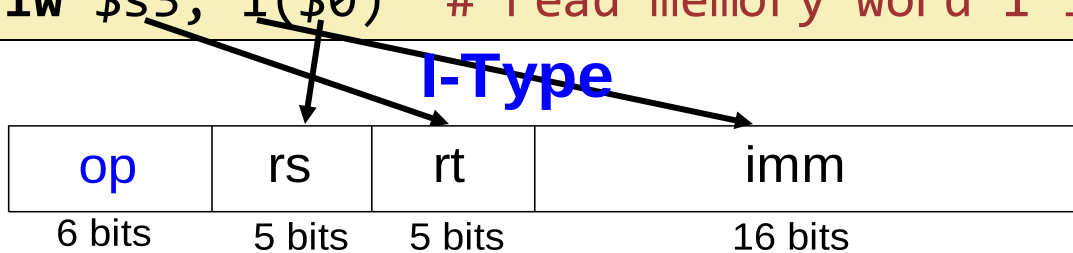
```
input [15:0]  addr; // Only 16 bits in this example
input [31:0]  di;
input         we;
output [31:0] do;

  reg [65535:0] M_arr [31:0];          // 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**

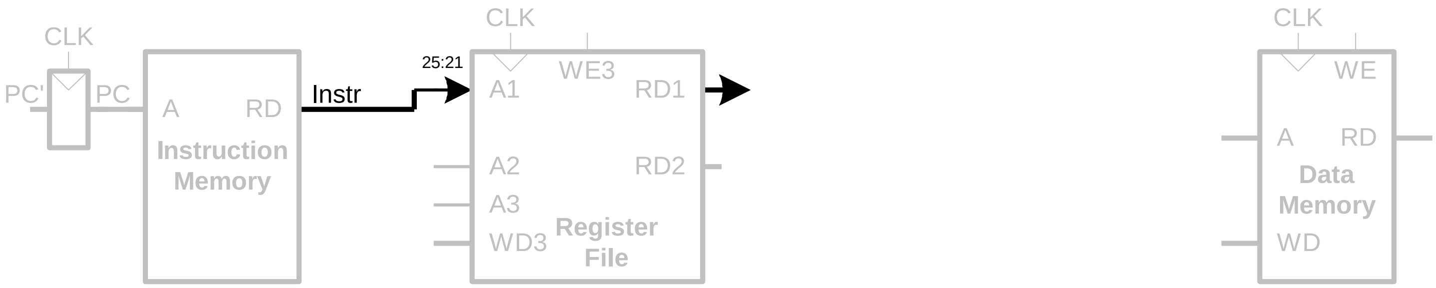



```
lw $s3, 1($0)   # read memory word 1 into $s3
```



Single-Cycle Datapath: `lw` register read

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



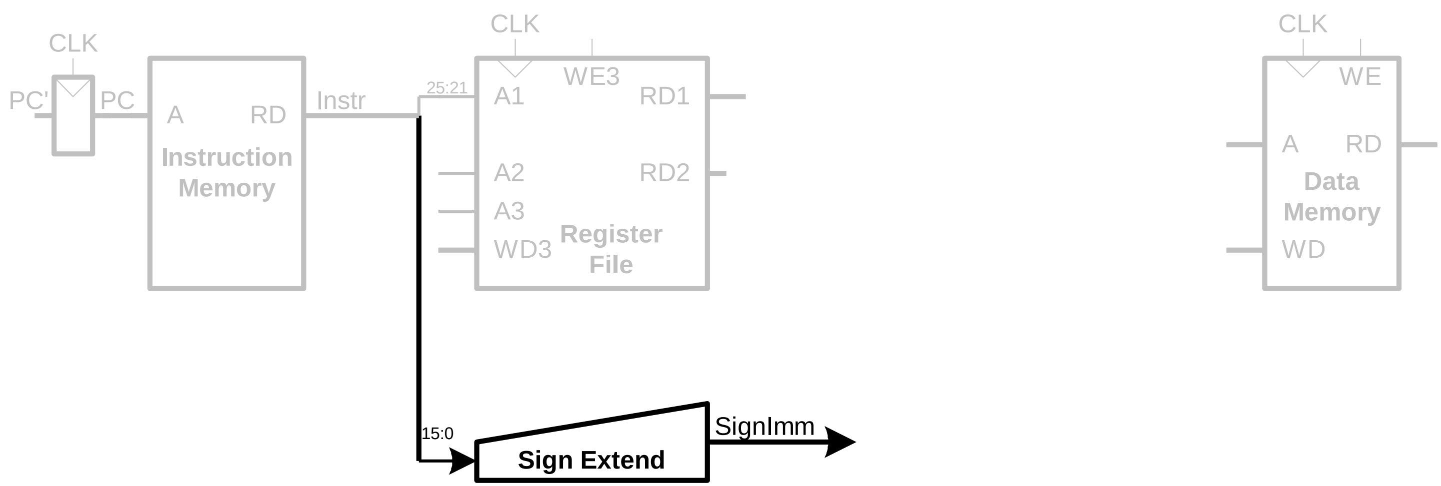

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

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



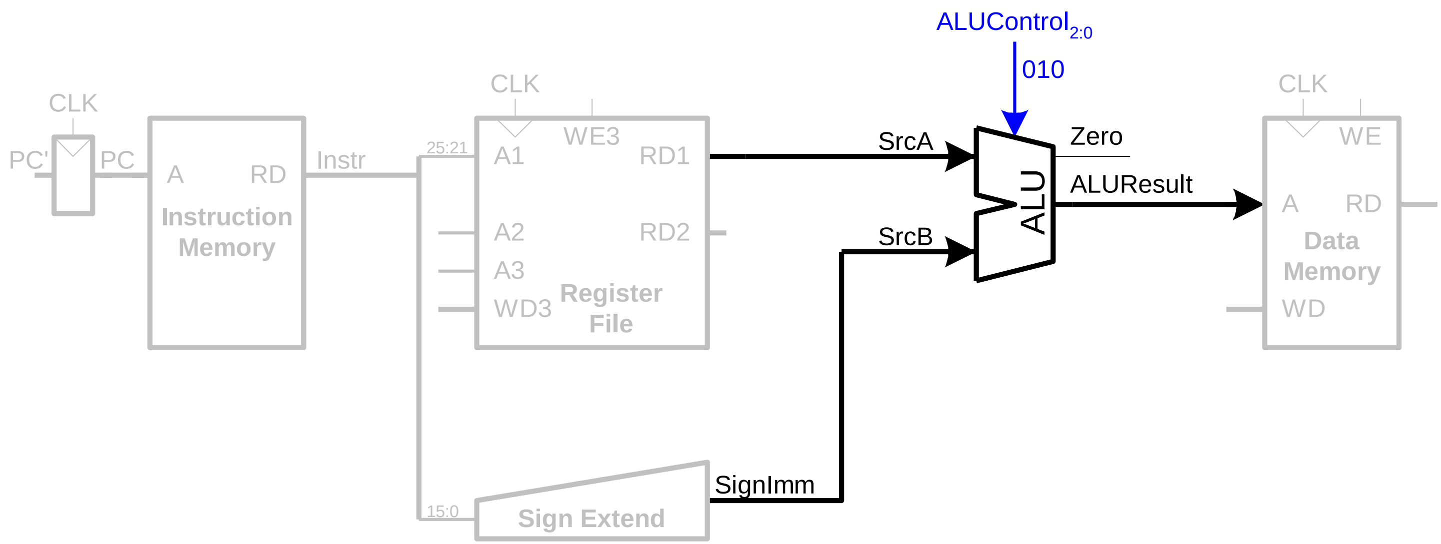

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



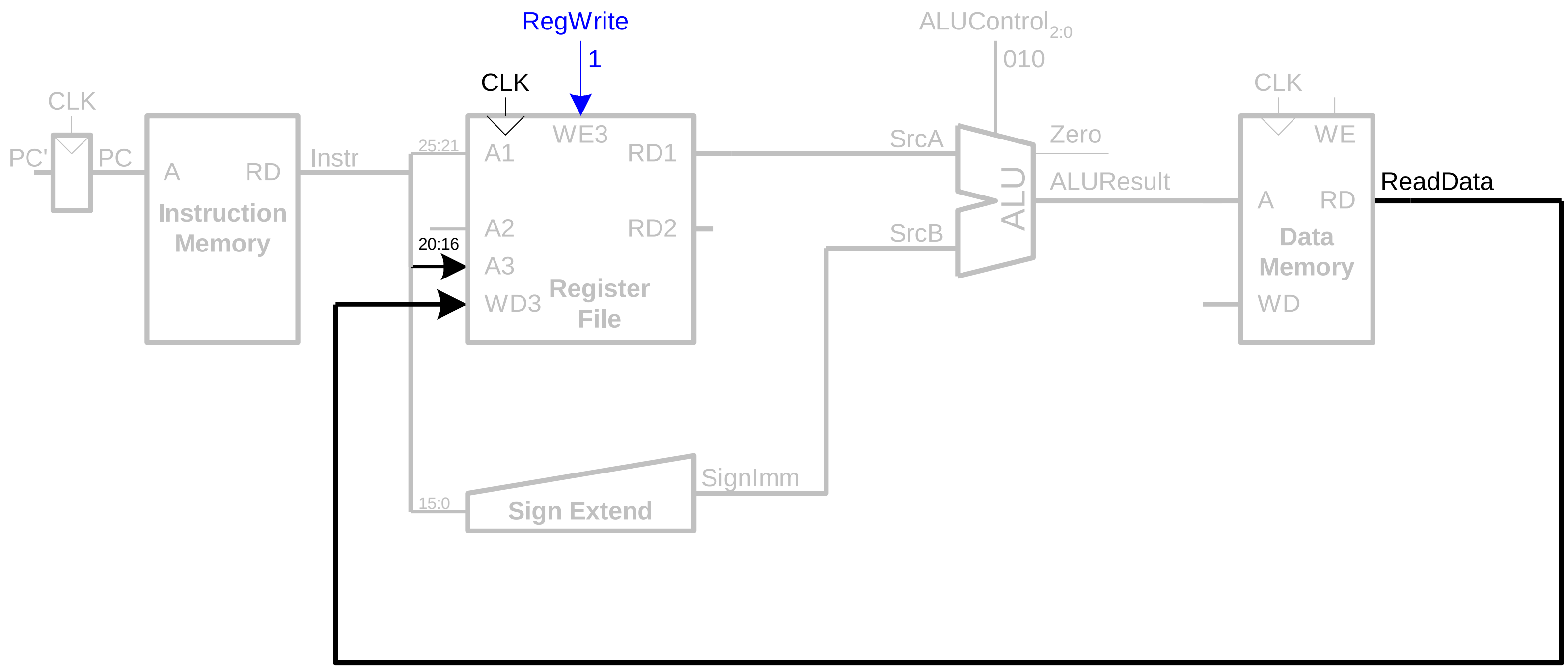

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



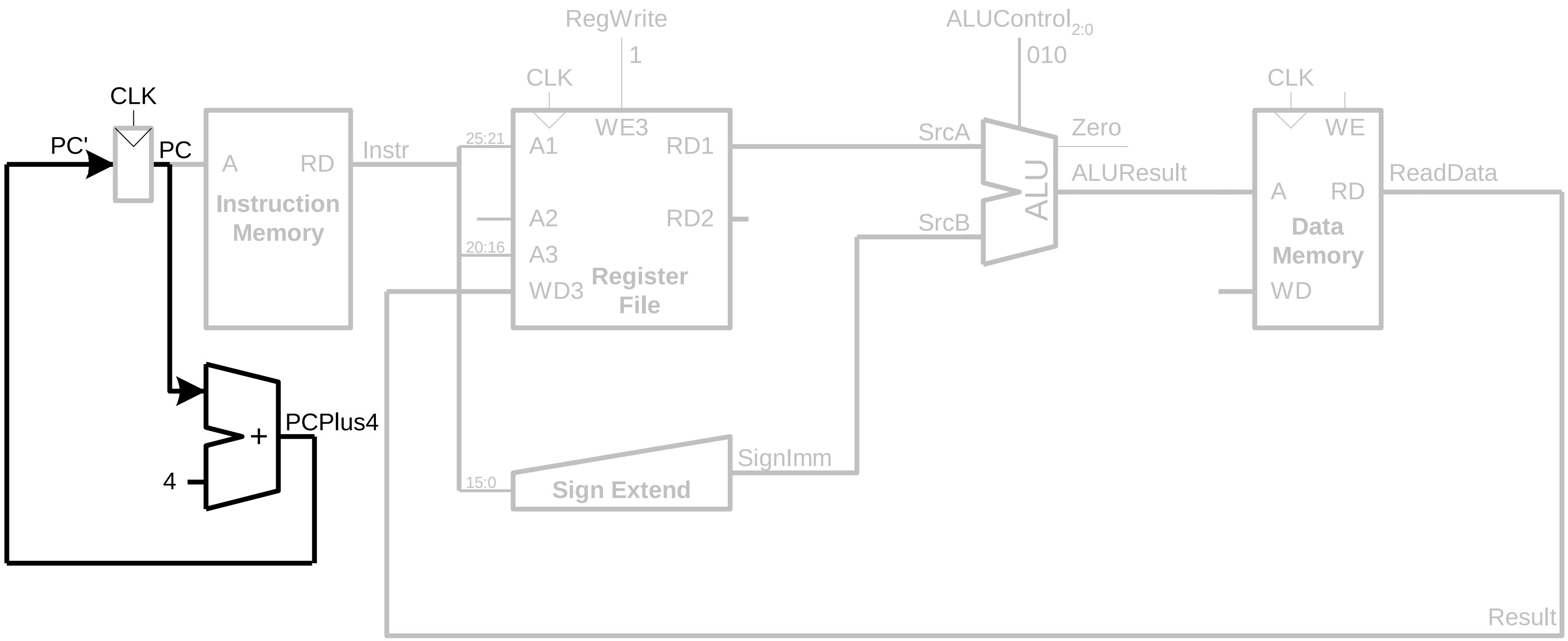

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

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**



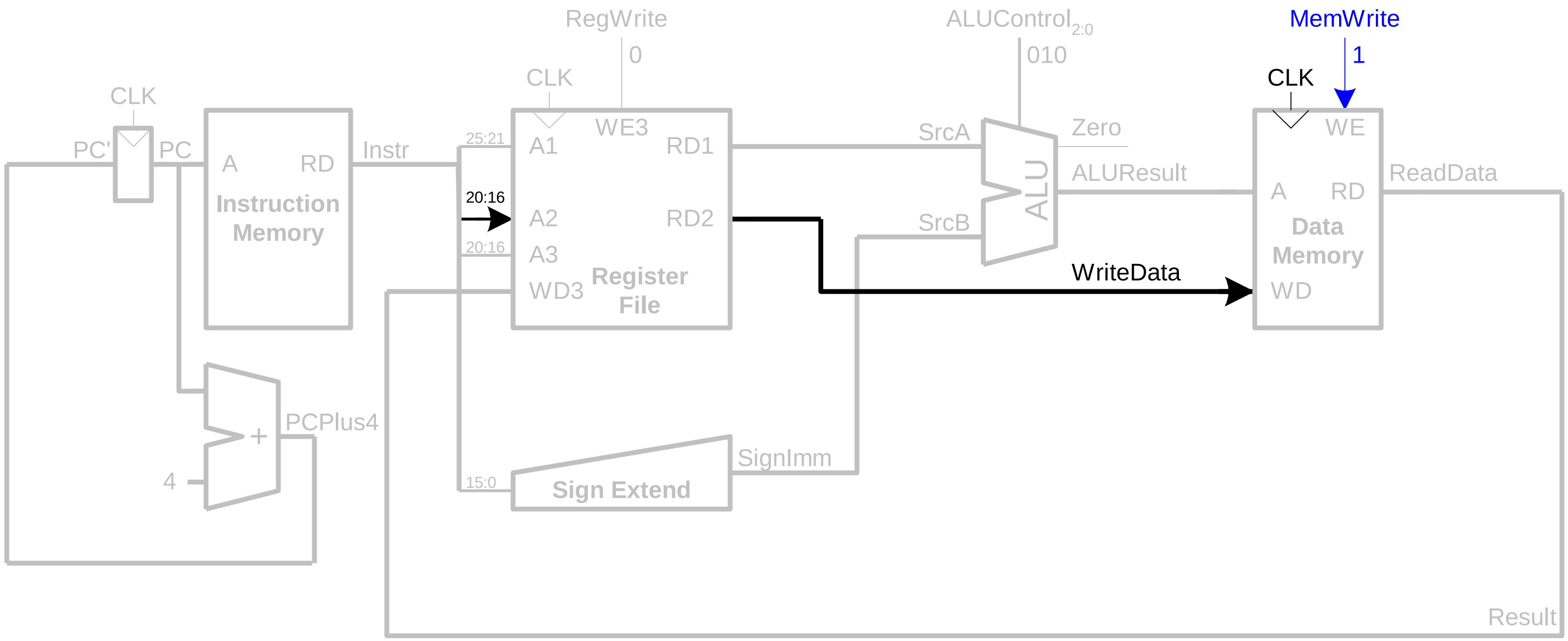

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

I-Type

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

Single-Cycle Datapath: sw

- **Write data in `rt` to memory**



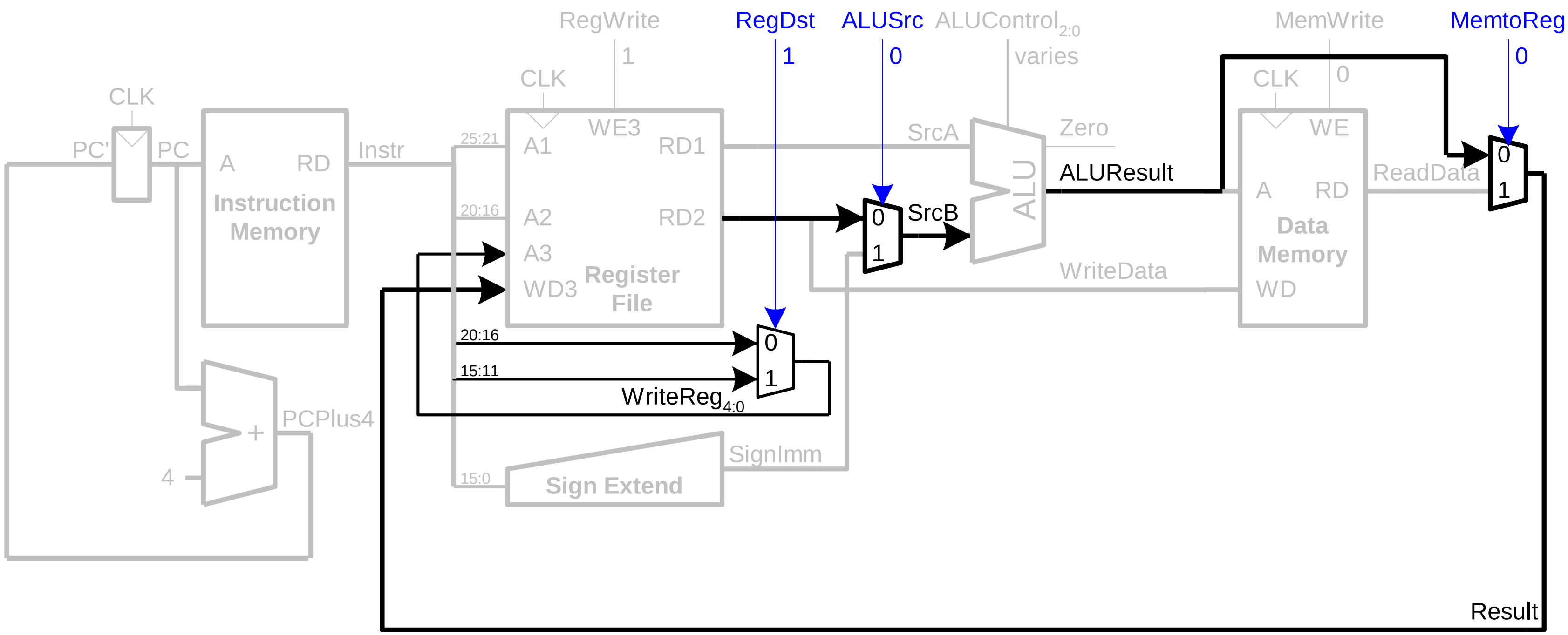

```
sw $t7, 44($0)  # write t7 into memory address 44
```

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**

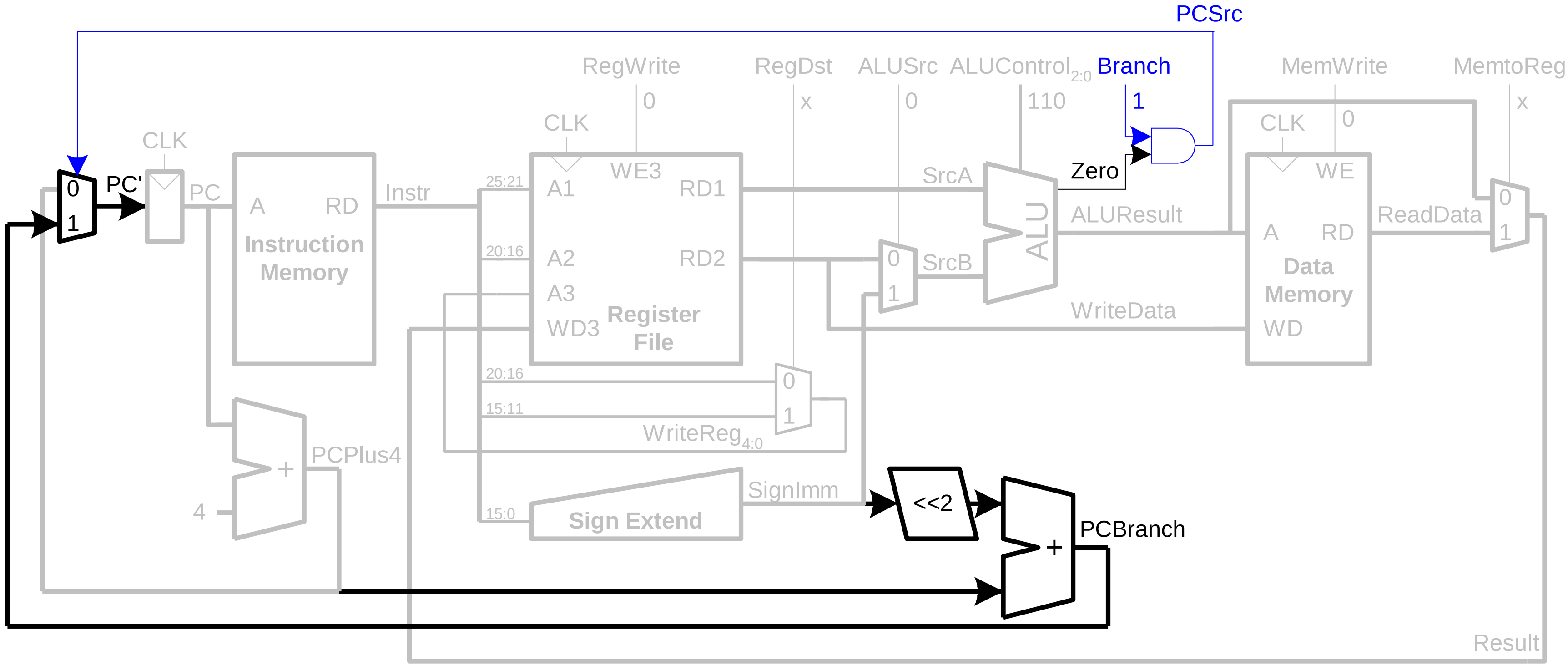



```
add t, b, c  # t = b + c
```

R-Type

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

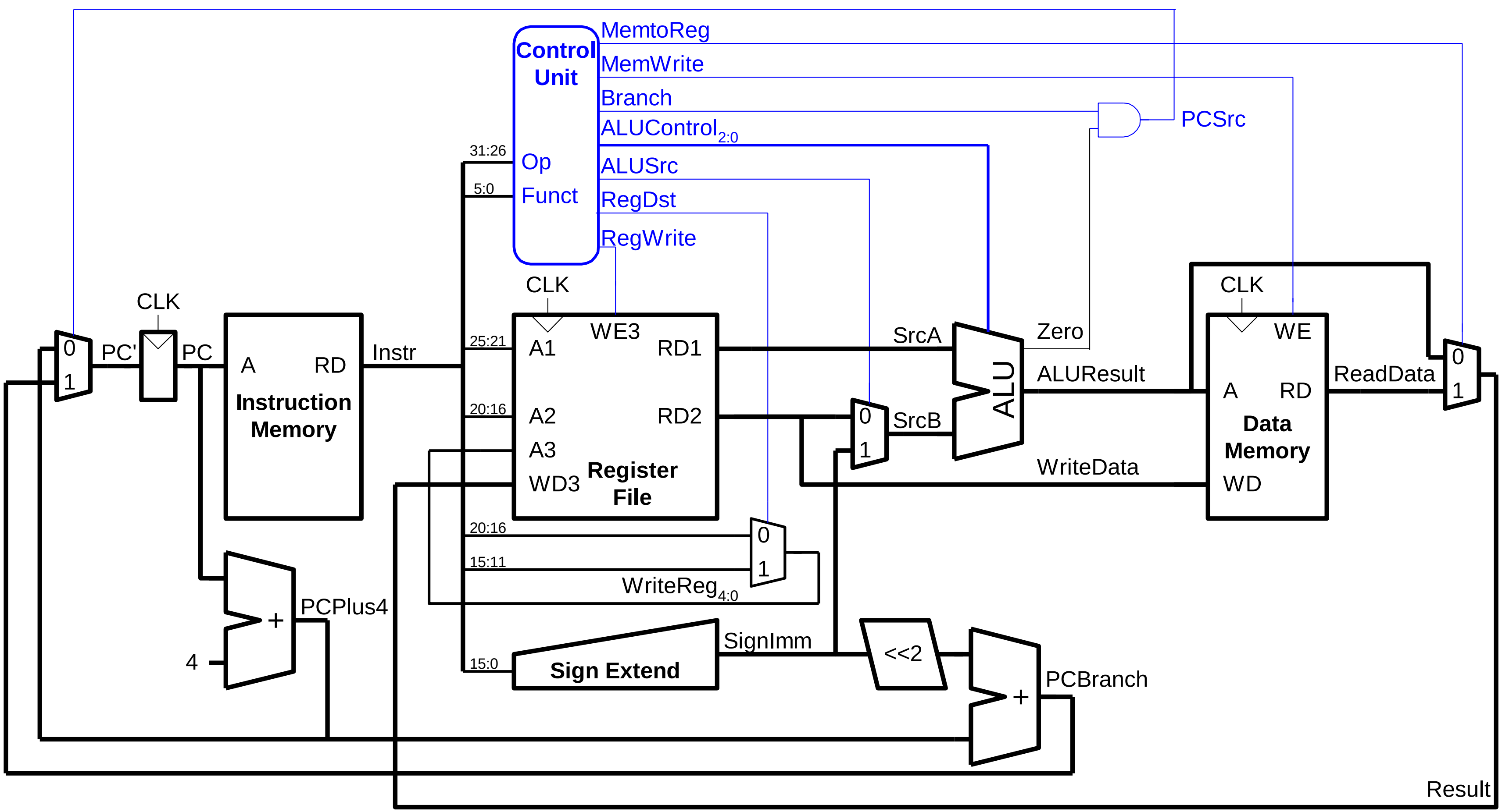
Single-Cycle Datapath: beq




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

- **Determine whether values in rs and rt are equal**
  Calculate BTA = (sign-extended immediate << 2) + (PC+4)

Complete Single-Cycle Processor

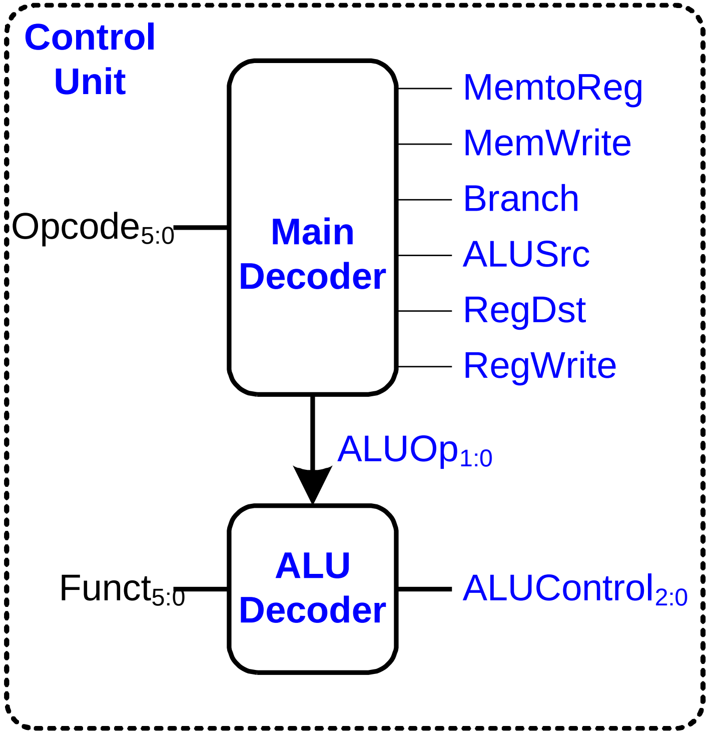


Our MIPS Datapath has Several Options

- **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)***

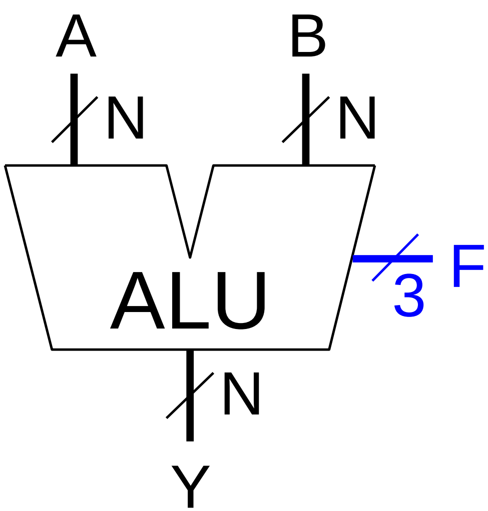
All these options are our control signals

Control Unit



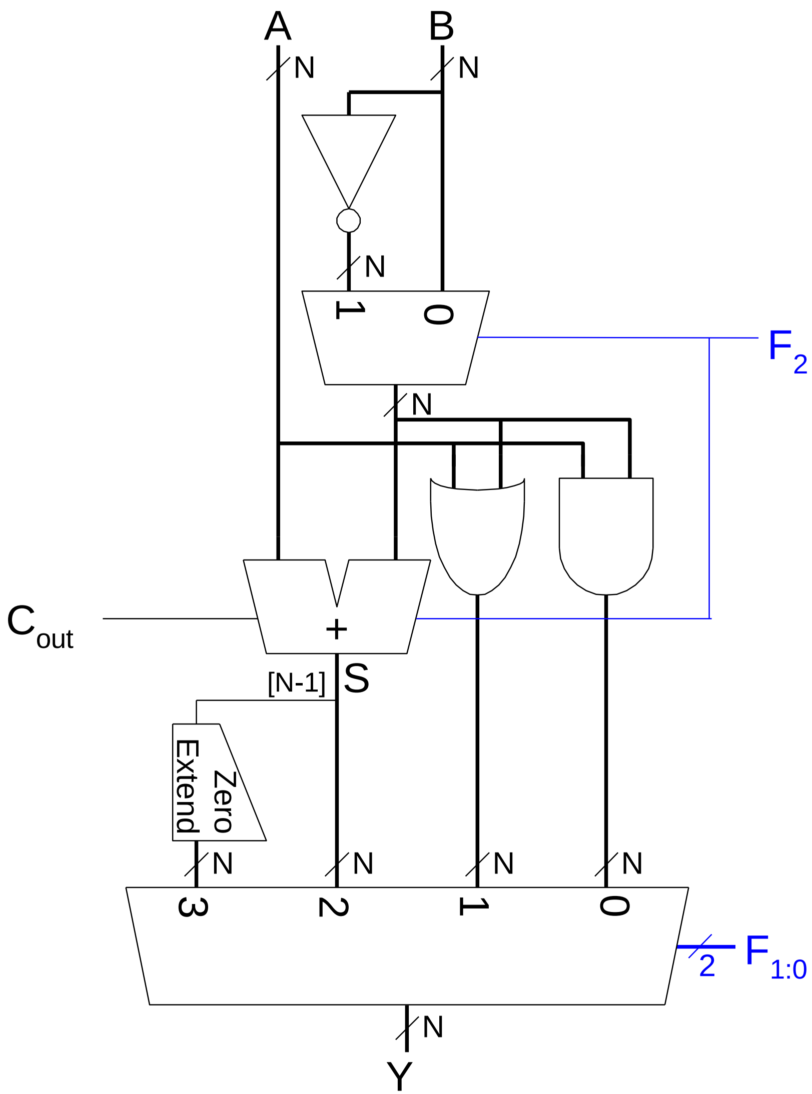
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

ALU Does the Real Work in a Processor



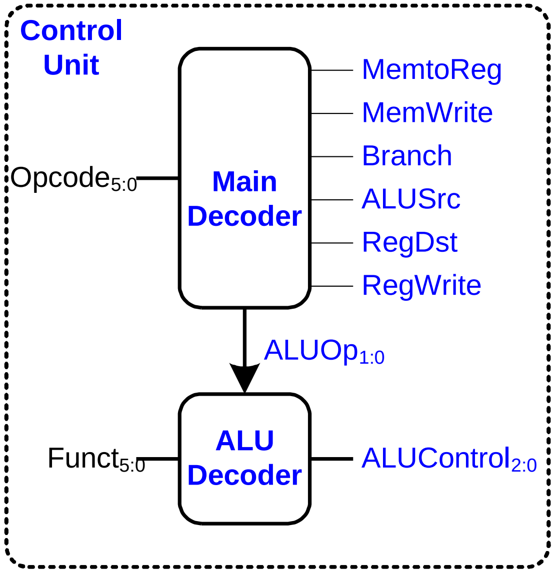
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Internals



$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~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

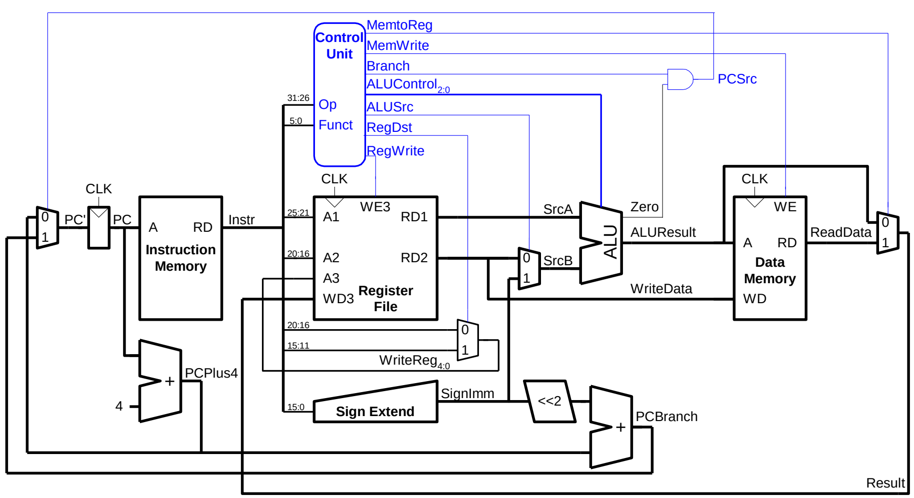
More Control Signals

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

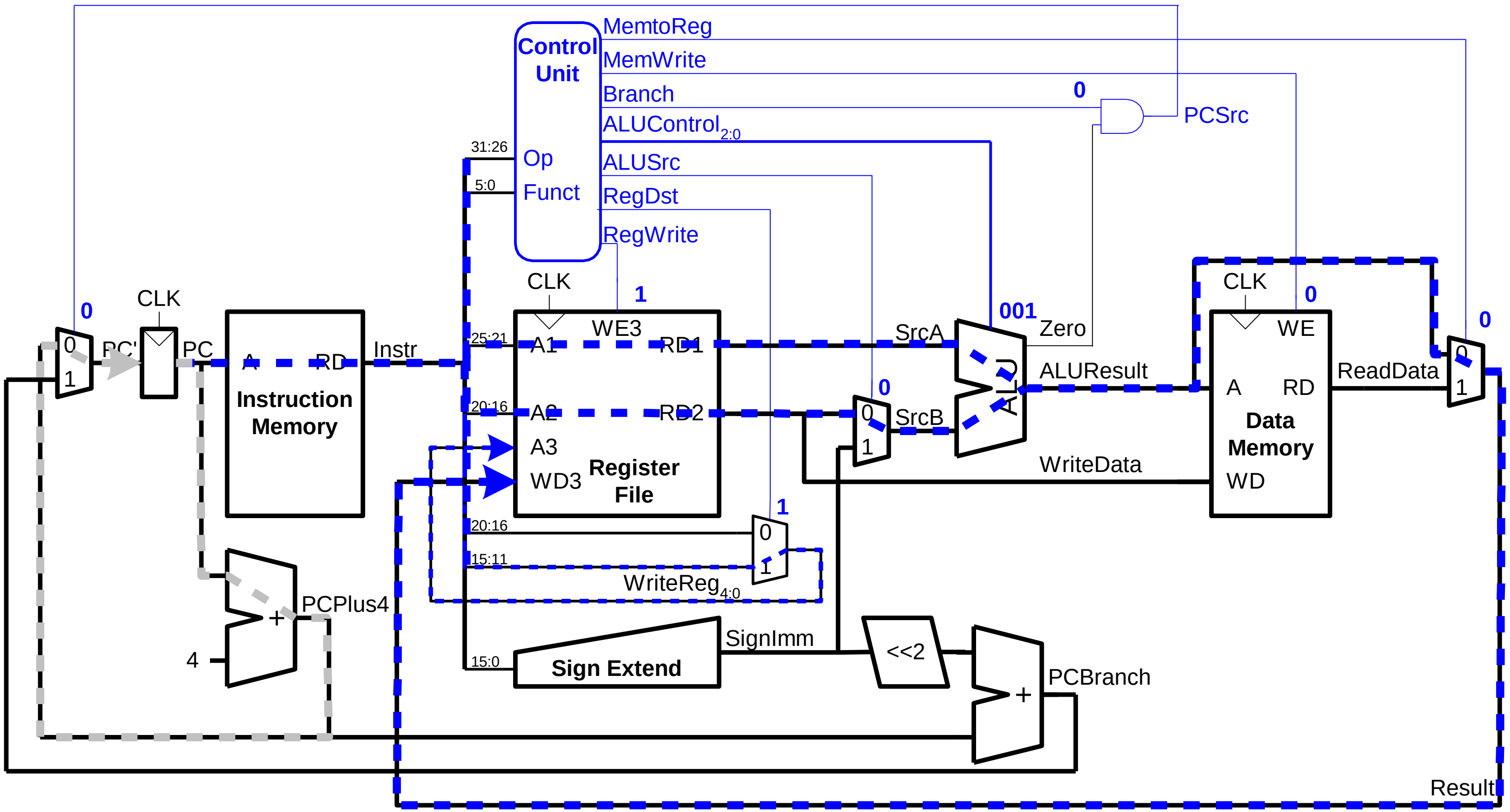
- **New Control Signal**
  - ***Branch***: Are we jumping or not ?

Control Unit: Main Decoder

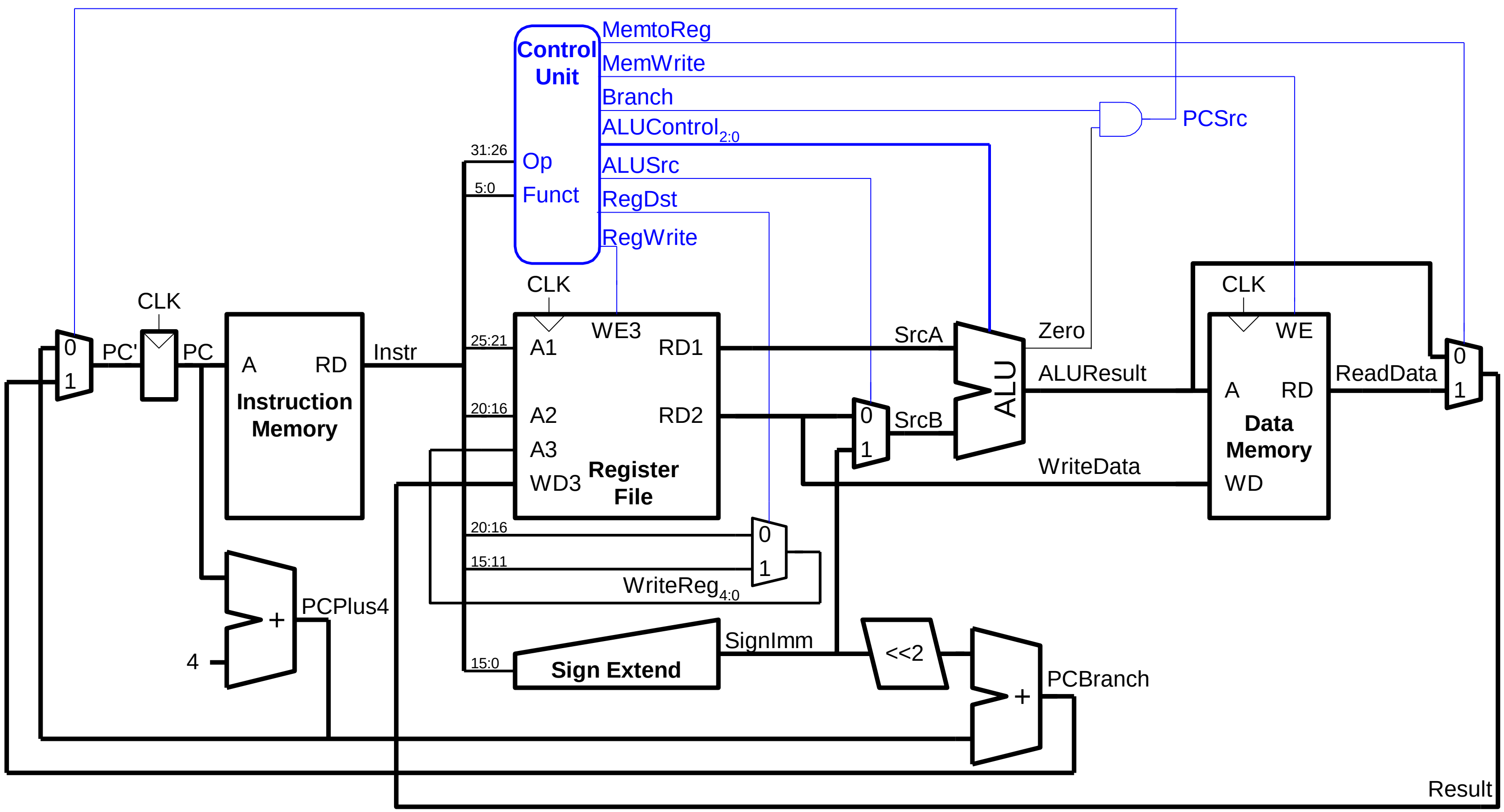
Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



Single-Cycle Datapath Example: or



Extended Functionality: addi

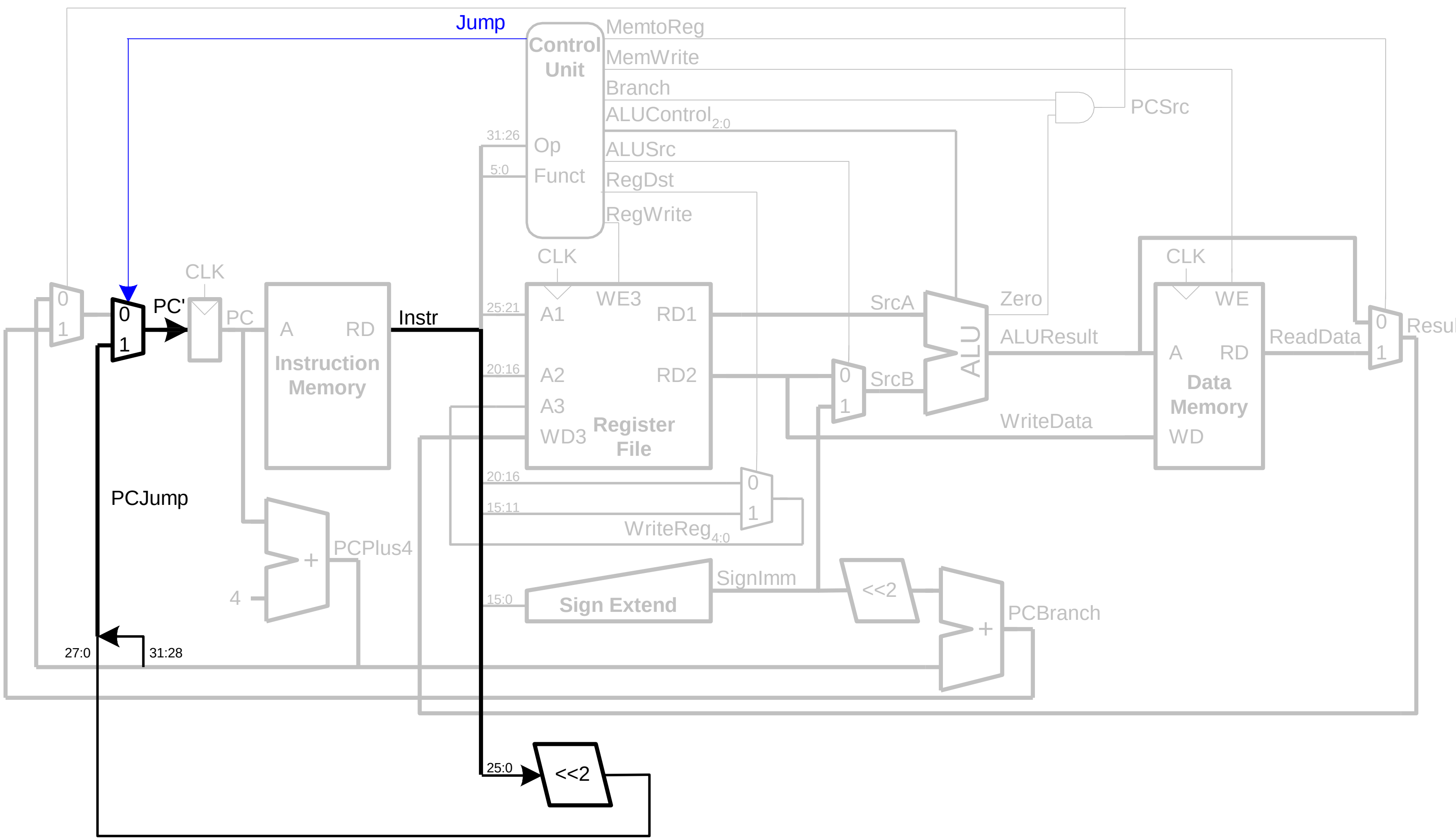


- No change to datapath

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

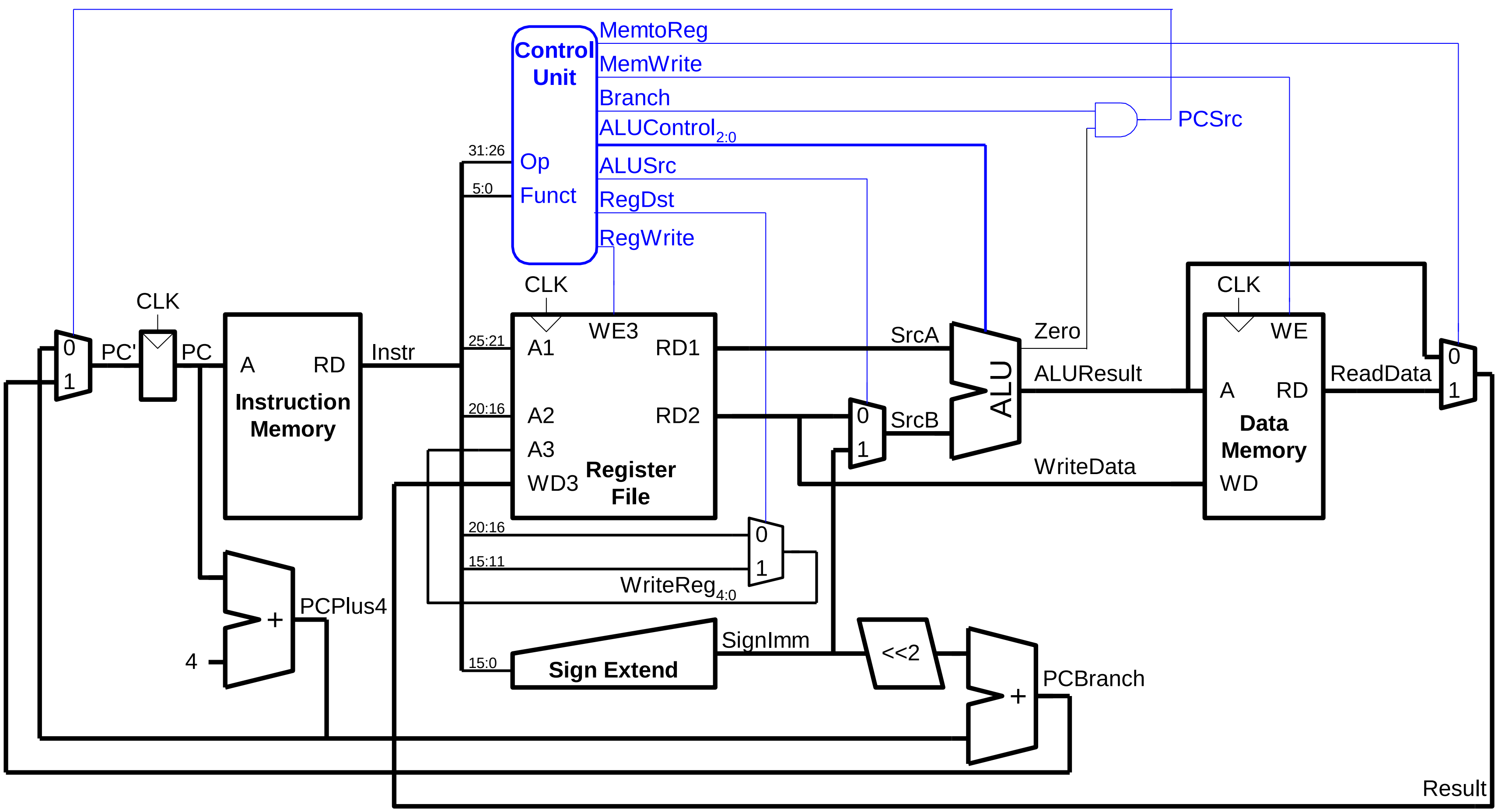
Extended Functionality: j



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/clock period = ***clock frequency*** = how many cycles can be done each second.

Performance Analysis

- Execution time of an instruction
  - {CPI} x {clock cycle time}

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

Processor Performance

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

Processor Performance

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

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

How can I Make the Program Run Faster?

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

How can I Make the Program Run Faster?

N x CPI x (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 x CPI x (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

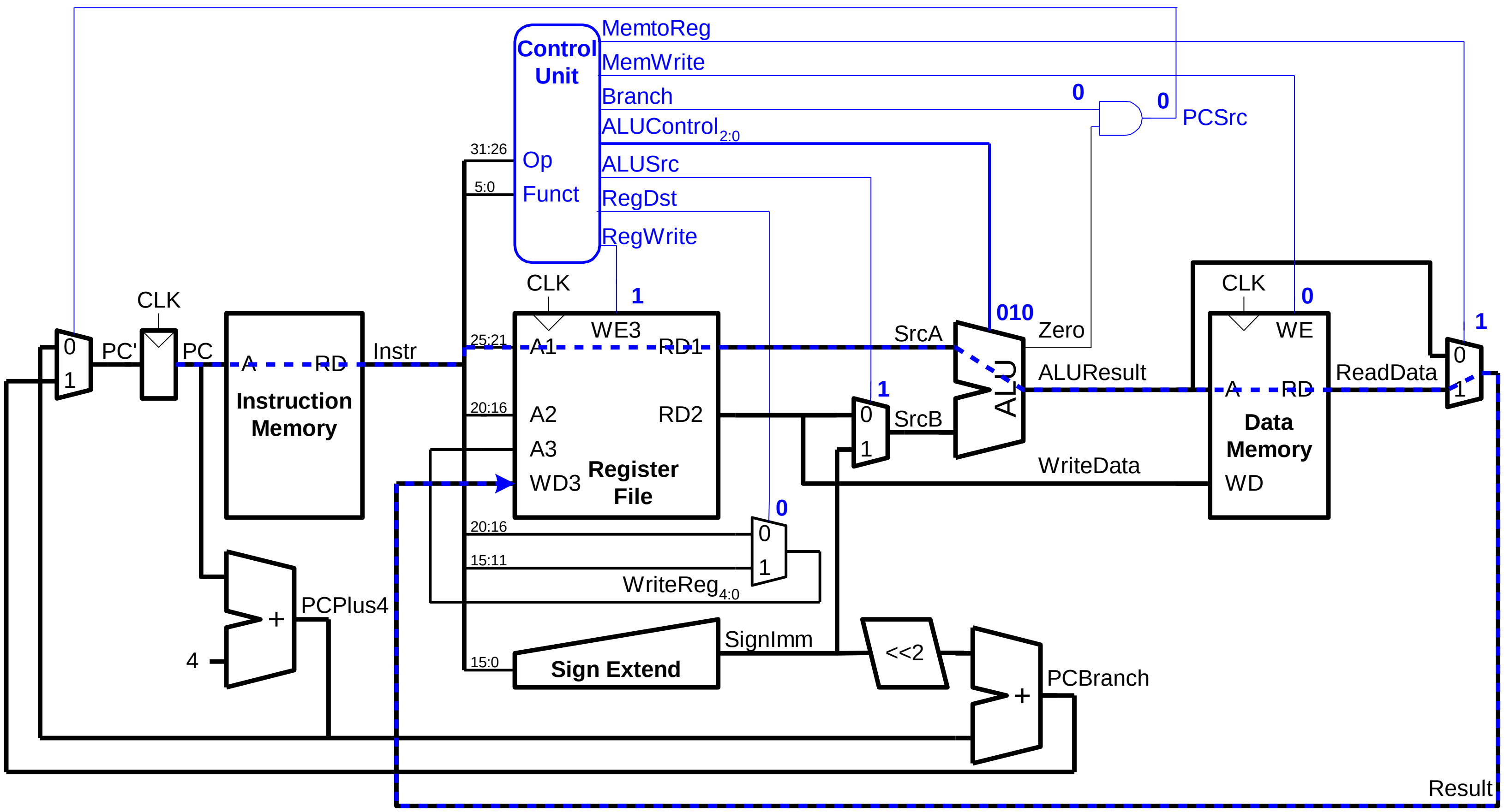
How can I Make the Program Run Faster?

N x CPI x (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

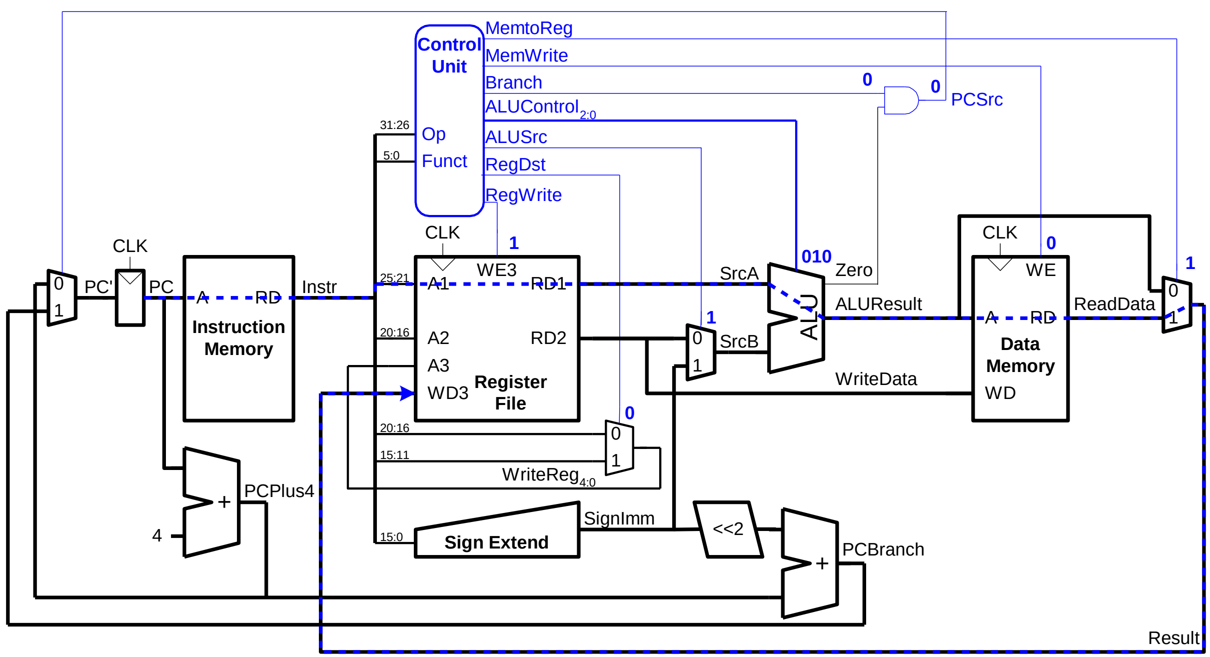
Single-Cycle Performance

- T_C is limited by the critical path (`lw`)



Single-Cycle Performance

- **Single-cycle critical path:**
  - $T_c = t_{pcq\_PC} + t_{mem} + \max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} + t_{mem} + t_{mux} + t_{RFsetup}$
- **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

$$T_c = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{RF\text{setup}}$$
$$= [30 + 2(250) + 150 + 25 + 200 + 20] \text{ ps}$$
$$= 925 \text{ ps}$$

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:

Execution Time = # instructions x CPI x TC

$= (100 \times 10^{9})(1)(925 \times 10^{-12}\ s)$

= 92.5 seconds