

# Design of Digital Circuits

## Lecture 11: Microarchitecture

Prof. Onur Mutlu

ETH Zurich

Spring 2018

29 March 2018

# Readings

---

## ■ This week

- Introduction to microarchitecture and single-cycle microarchitecture
  - P&P, Appendices A and C
  - H&H, Chapter 7.1-7.3, 7.6

## ■ Next week

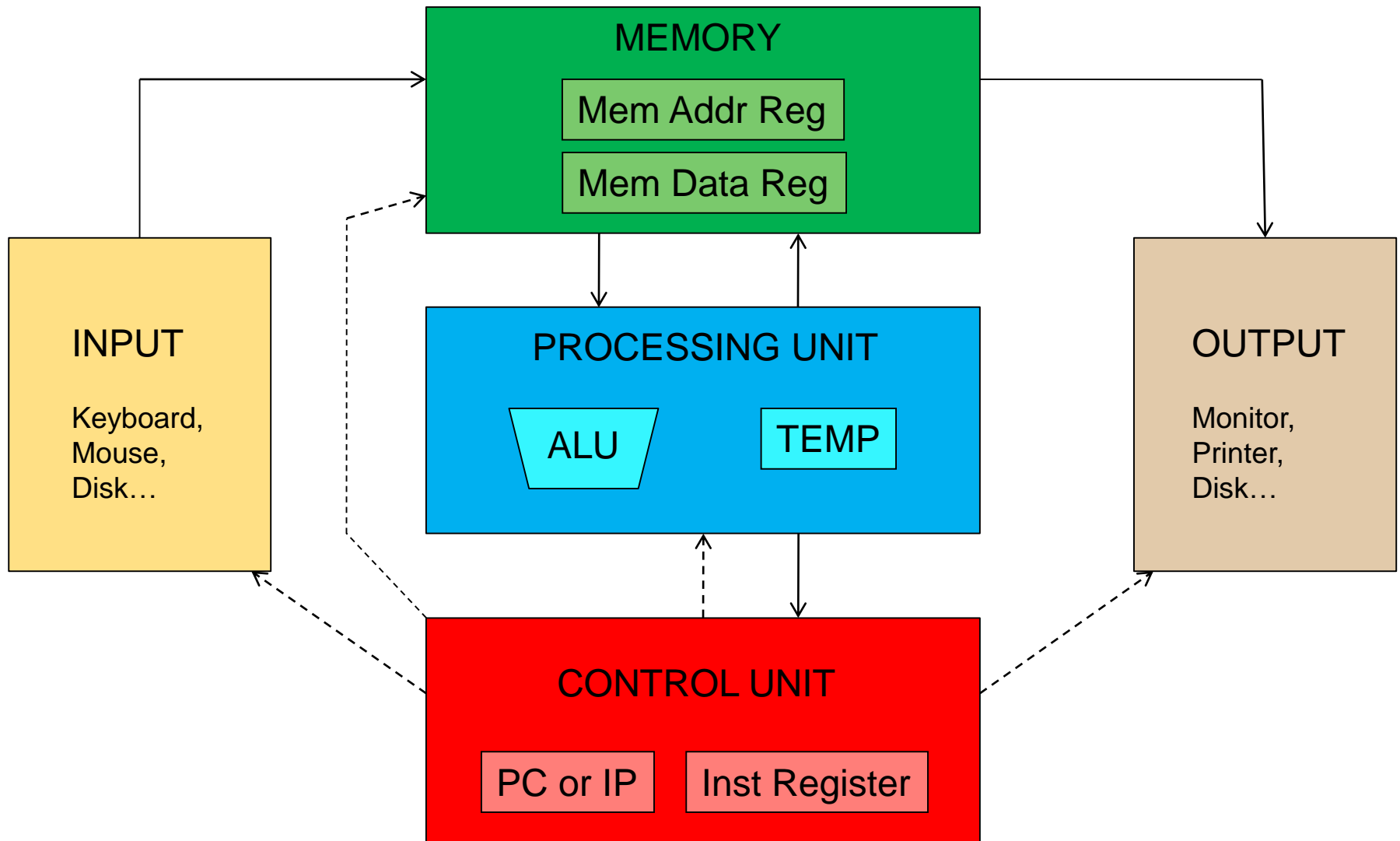
- Multi-cycle microarchitecture
  - P&P, Appendices A and C
  - H&H, Chapter 7.4
- Microprogramming
  - P&P, Appendices A and C
- Pipelining
  - H&H, Chapter 7.5

# 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
- Microprogramming
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution

# Recall: The Von Neumann Model



# Recall: LC-3: A Von Neumann Machine

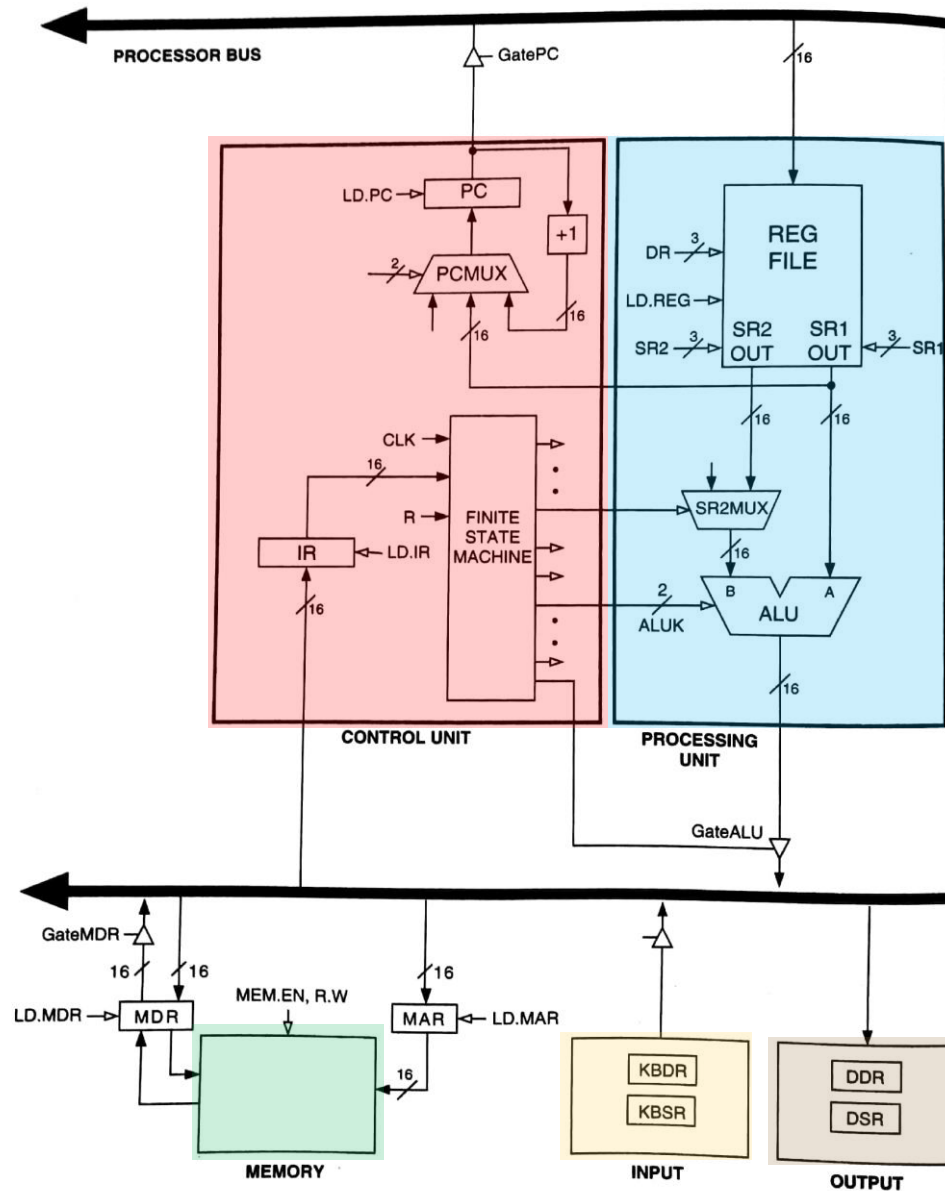
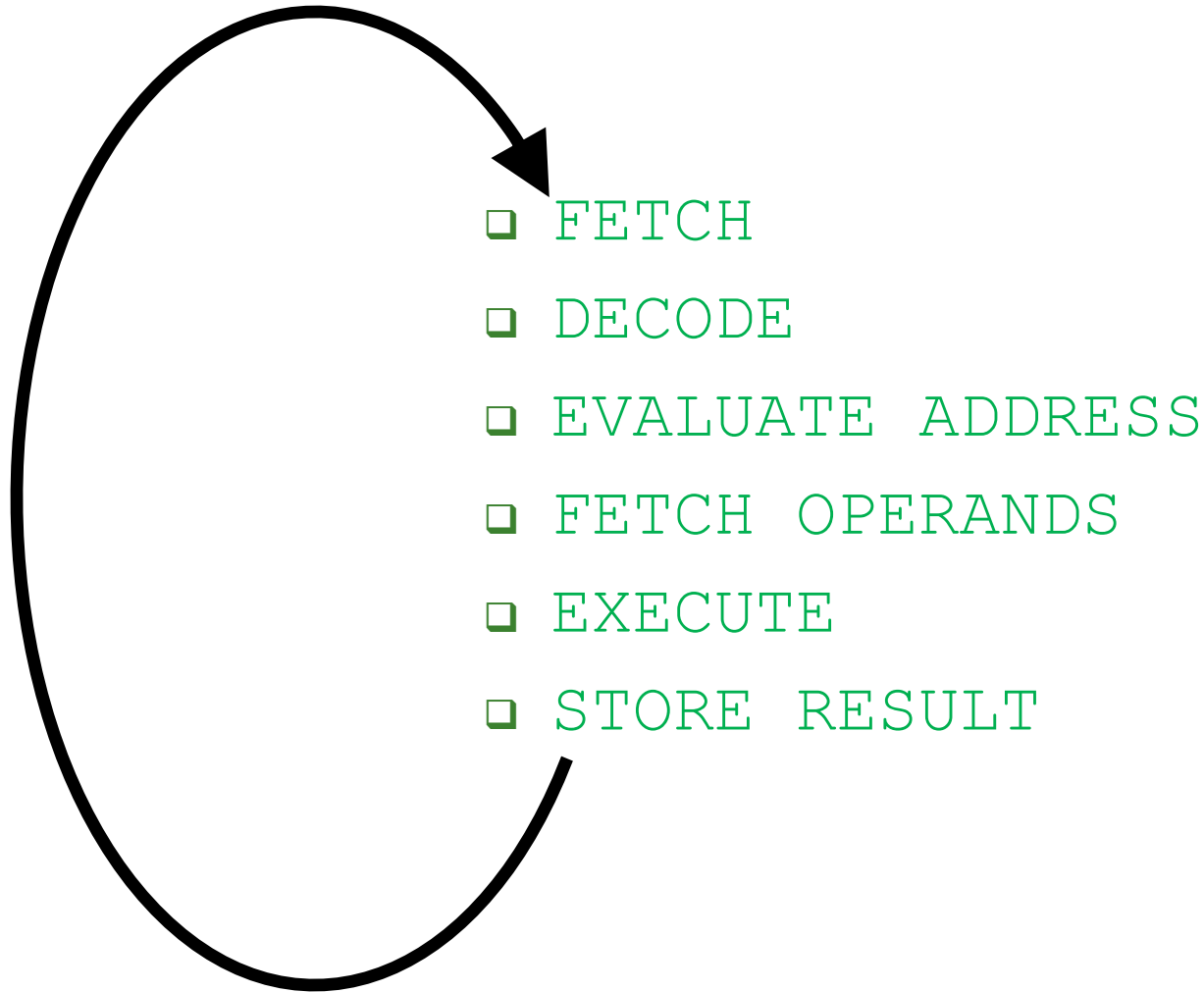


Figure 4.3 The LC-3 as an example of the von Neumann model

# Recall: The Instruction Cycle

---



# Recall: The Instruction Set Architecture

---

- The ISA is the **interface between** what the **software** commands and what the **hardware** carries out
- The ISA specifies
  - The **memory organization**
    - Address space (LC-3:  $2^{16}$ , MIPS:  $2^{32}$ )
    - Addressability (LC-3: 16 bits, MIPS: 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

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

# Microarchitecture

---

- An **implementation** of the ISA
- How do we implement the ISA?
  - We will discuss this for many lectures
- There can be many implementations of the same ISA
  - MIPS R2000, R10000, ...
  - Intel 80486, Pentium, Pentium Pro, Pentium 4, Kaby Lake, Coffee Lake, ...



# (A Bit More on) ISA Design and Tradeoffs

# The Von Neumann Model/Architecture

---

- Also called *stored program computer* (instructions in memory). 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
- Sequential instruction processing
  - One instruction processed (fetched, executed, and completed) at a time
  - Program counter (instruction pointer) identifies the current instr.
  - Program counter is advanced sequentially except for control transfer instructions

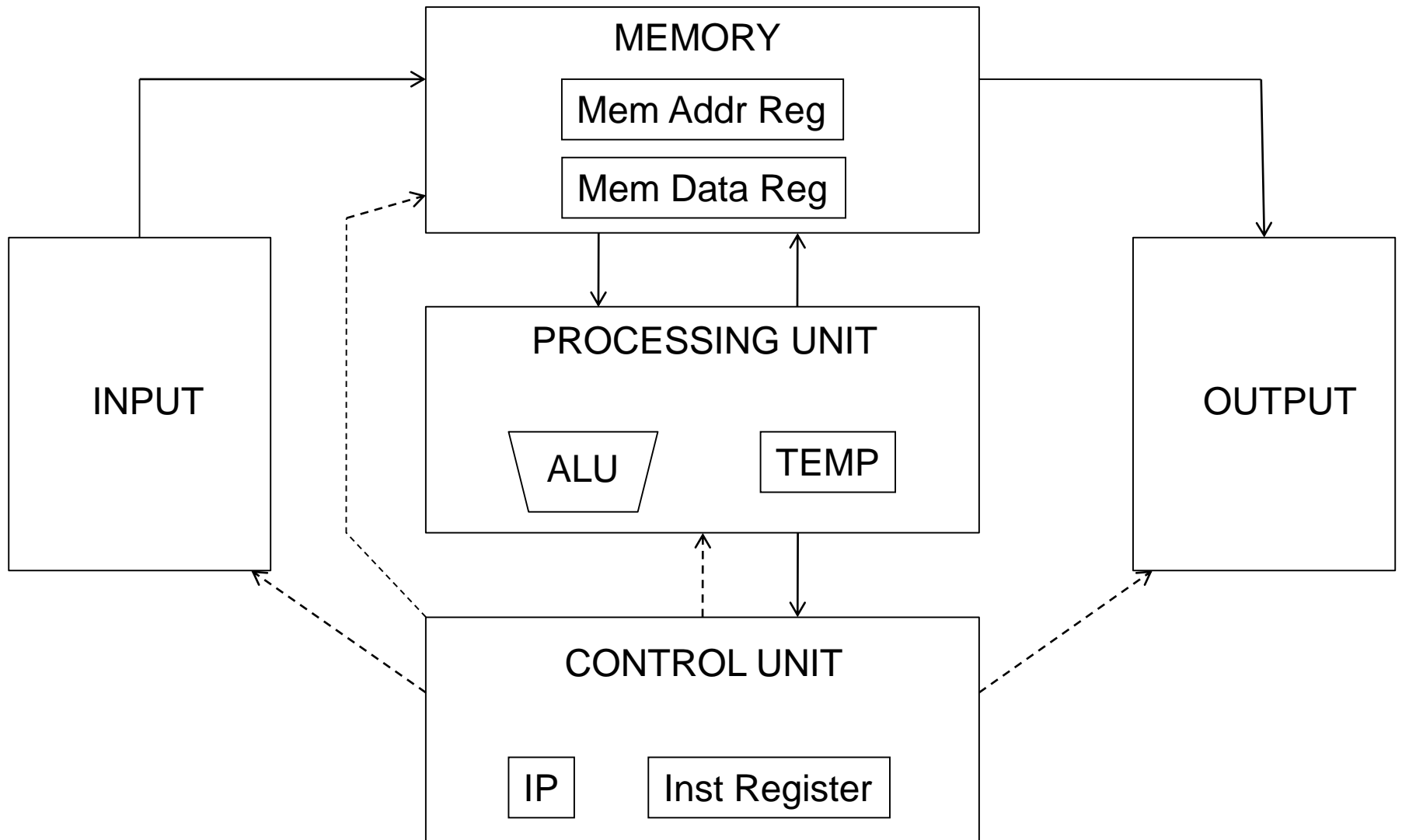
# The Von Neumann Model/Architecture

---

- 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)

---

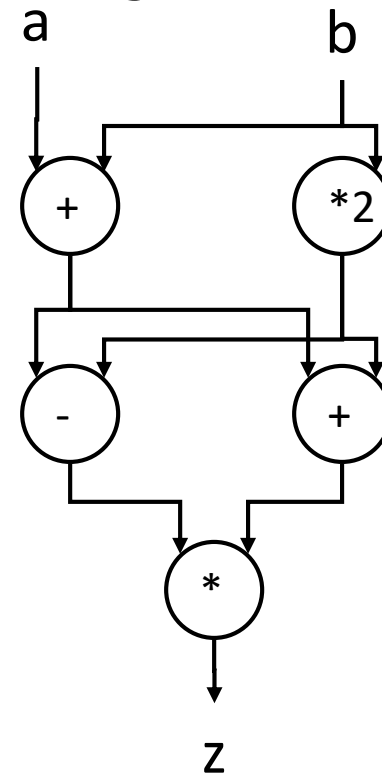
- Von Neumann model: An instruction is fetched and executed in **control flow order**
  - As specified by the **instruction pointer**
  - Sequential unless explicit control flow instruction
  
- Dataflow model: An instruction is fetched and executed in **data flow order**
  - i.e., when its operands 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



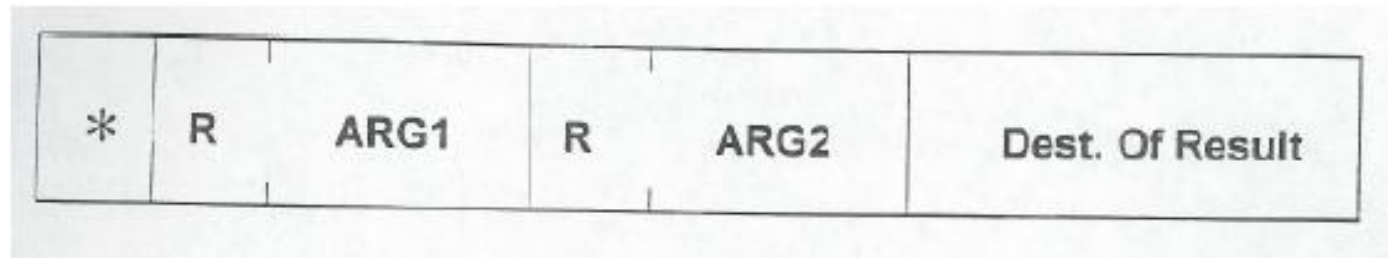
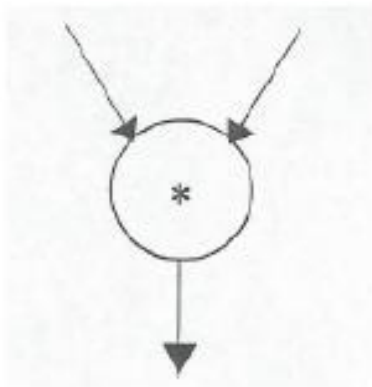
Dataflow

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

# More on Data Flow

---

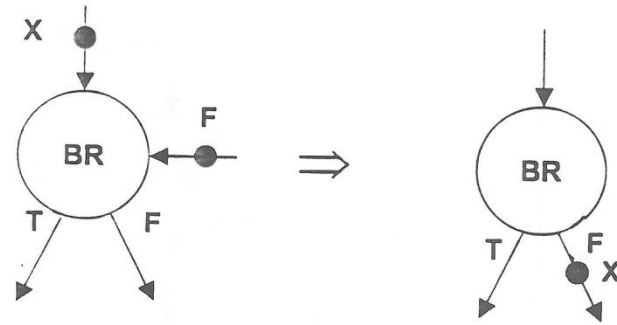
- In a data flow machine, a program consists of data flow nodes
  - A data flow node fires (fetched and executed) when all its inputs are ready
    - i.e. when all inputs have tokens
- Data flow node and its ISA representation



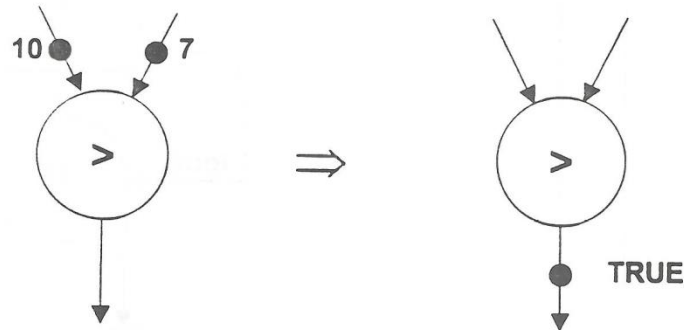


# Data Flow Nodes

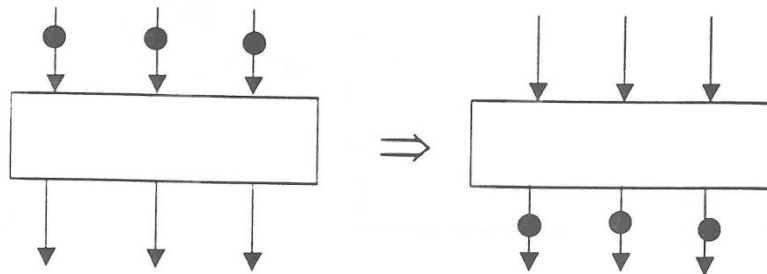
## *\*Conditional*



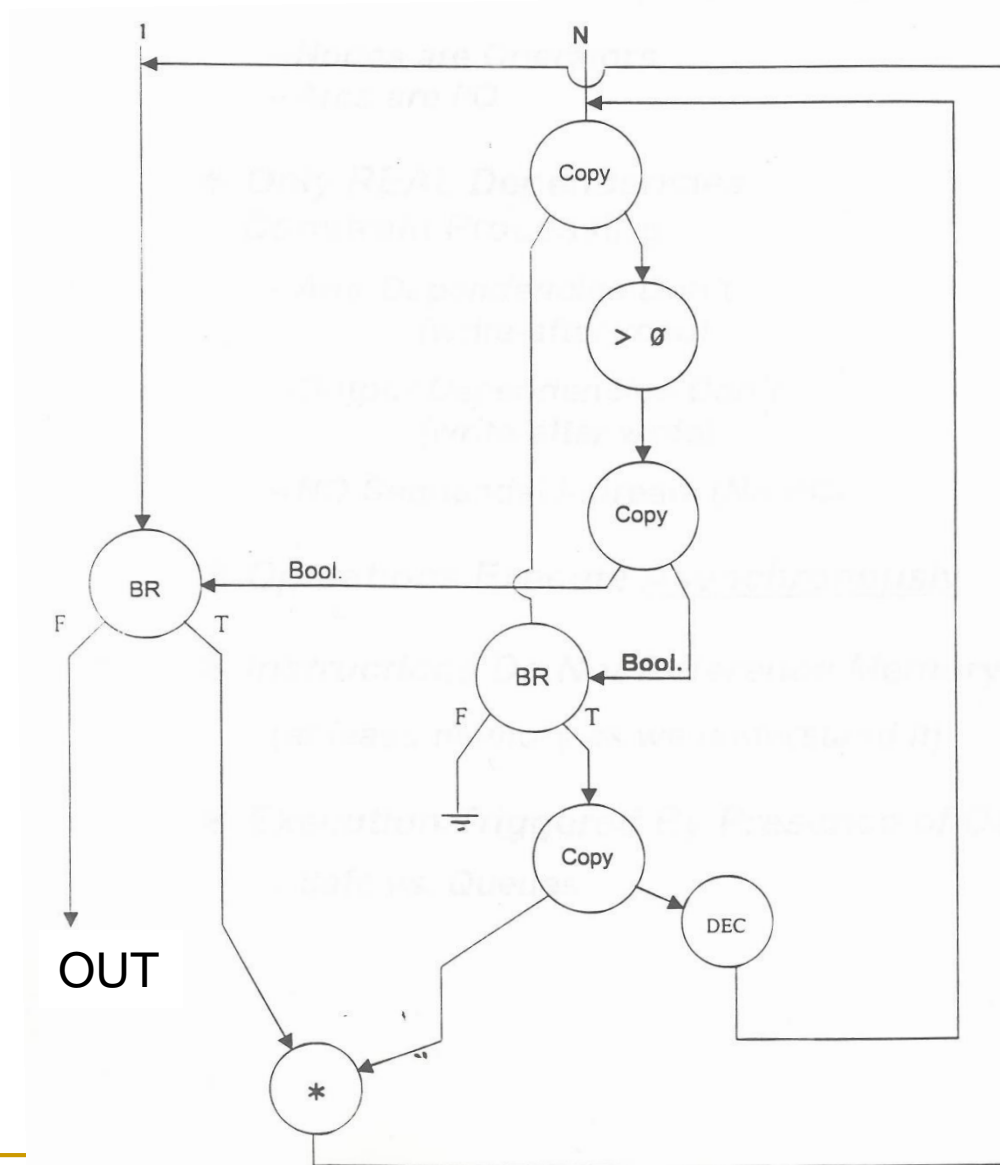
## *\*Relational*



## *\*Barrier Synchron*



# 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
- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution: *Intel 80486 uarch*
  - Multiple instructions at a time: *Intel Pentium uarch*
  - Out-of-order execution: *Intel Pentium Pro uarch*
  - Separate instruction and data caches
- But, what happens underneath that is **not consistent** with the von Neumann model is **not exposed** to software
  - Difference between ISA and microarchitecture

# What is Computer Architecture?

---

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

# ISA vs. Microarchitecture

---

## ■ ISA

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

## ■ Microarchitecture

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

## ■ Microprocessor

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

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons



# ISA vs. Microarchitecture

---

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implement “acceleration”
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Add instruction vs. Adder implementation
    - Bit serial, ripple carry, carry lookahead adders are all part of microarchitecture **(see H&H Chapter 5.2.1)**
  - x86 ISA has many implementations: 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) but many uarchs
  - *Why?*

## ■ Instructions

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

## ■ Memory

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

## ■ Call, Interrupt/Exception Handling

## ■ Access Control, Priority/Privilege

## ■ I/O: memory-mapped vs. instr.

## ■ Task/thread Management

## ■ Power and Thermal Management

## ■ Multi-threading support, Multiprocessor support

## ■ ...



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

Volume 1:  
Basic Architecture

# Microarchitecture

---

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

# Property of ISA vs. Uarch?

---

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

# Design Point

---

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

Problem
Algorithm
Program
ISA
Microarchitecture
Circuits
Electrons

# Application Space

## Dream, and they will appear...

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

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

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

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

### Many other workloads:

Genome analysis

Machine learning

Robotics

Web search

Graph analytics

...

# Increasingly Demanding Applications

---

Dream

and, they will come

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

# Tradeoffs: Soul of Computer Architecture

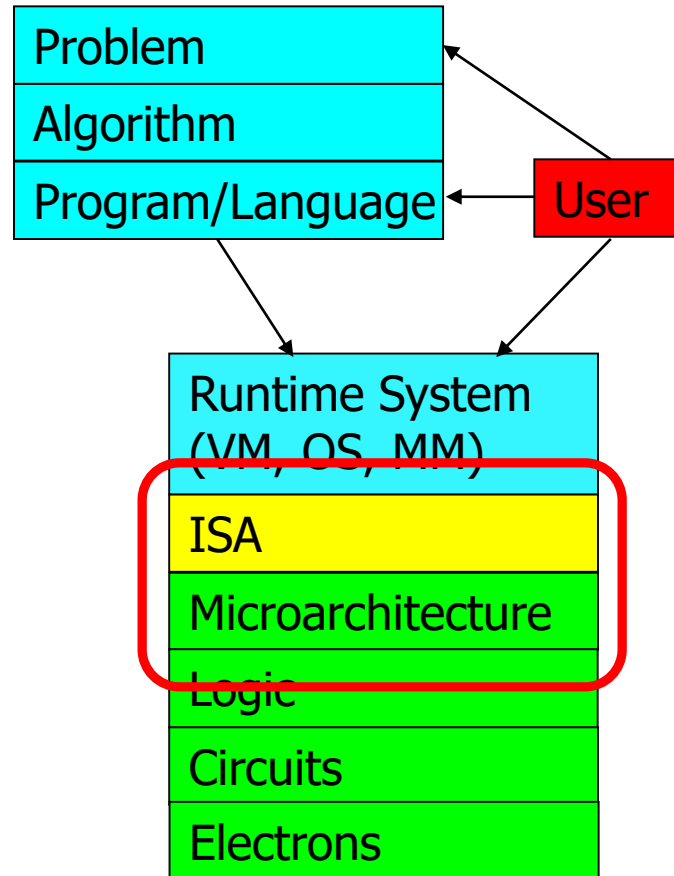
---

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



# Why Is It (Somewhat) Art?

New demands  
from the top  
(Look Up)



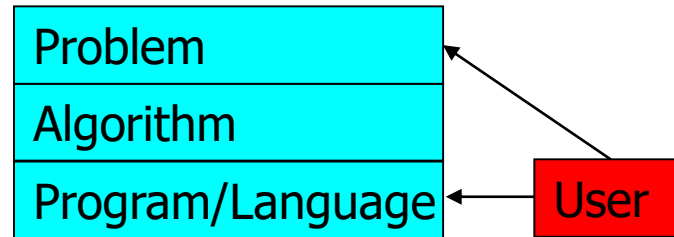
New demands and  
personalities of users  
(Look Up)

New issues and  
capabilities  
at the bottom  
(Look Down)

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

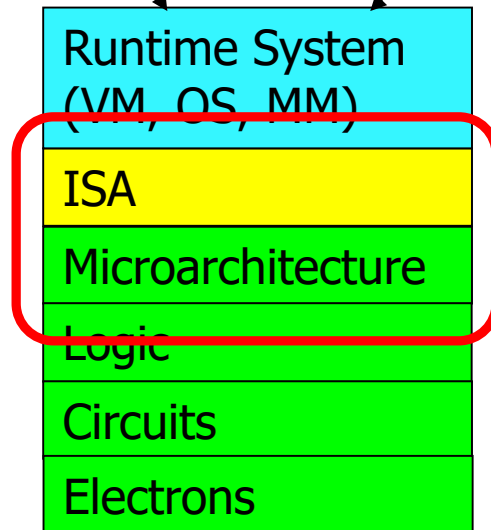
# Why Is It (Somewhat) Art?

Changing demands  
at the top  
(Look Up and Forward)



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

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



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

# Analogue from Macro-Architecture

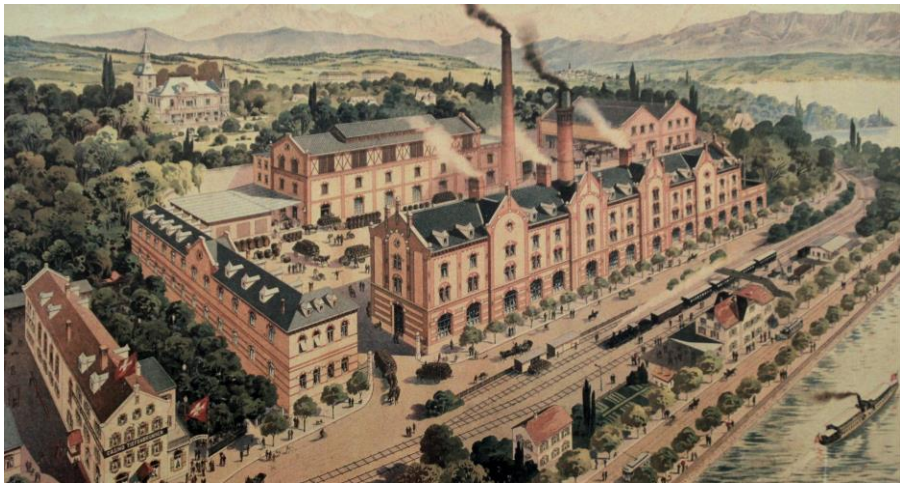
---

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

# Mühle Tiefenbrunnen

---

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



Brewery in 1900





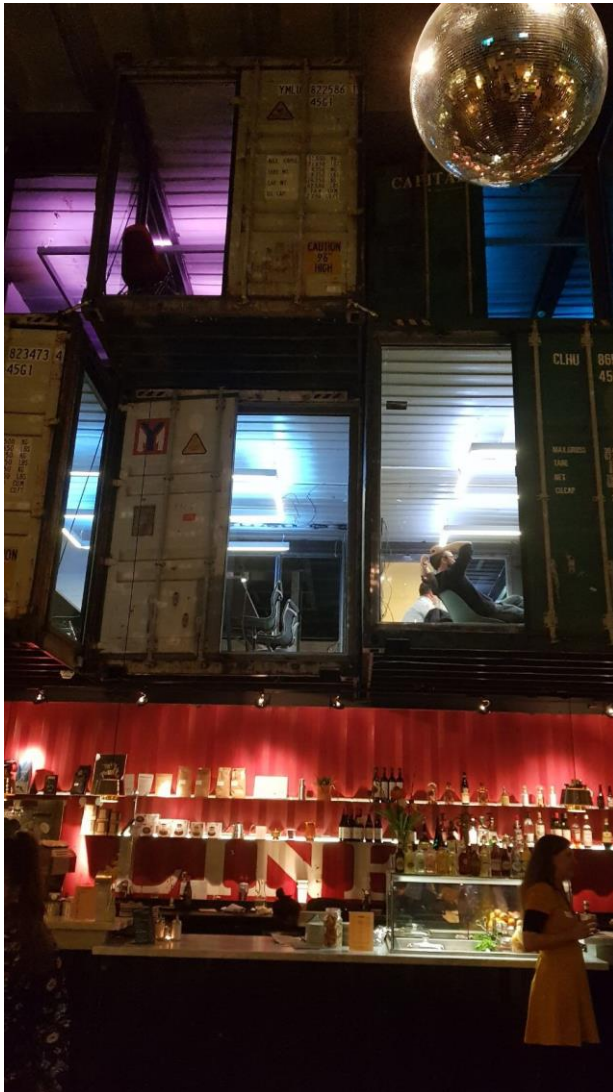
# Another Example (I)

---

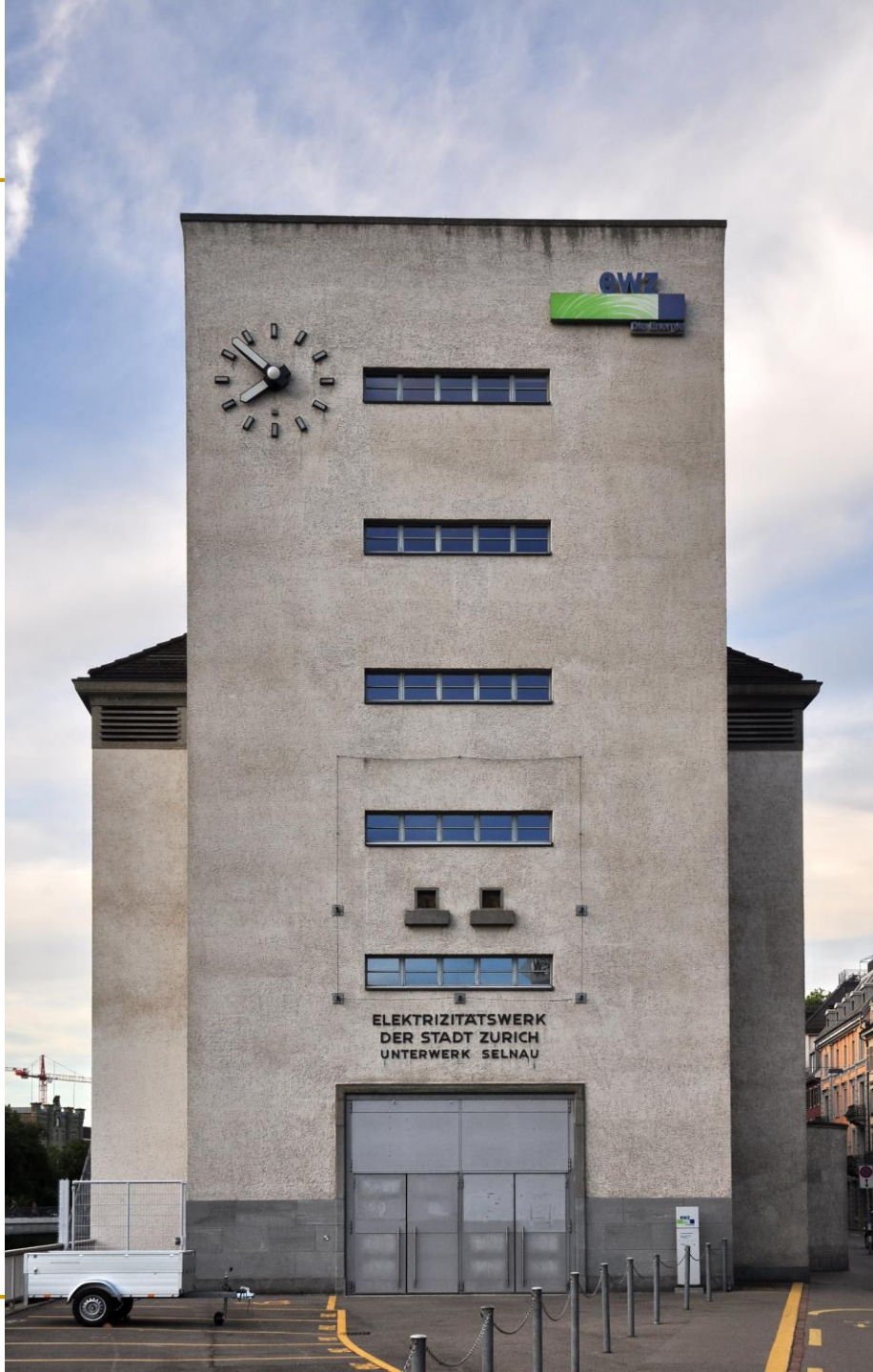


# Another Example (II)

---







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

# Implementing the ISA: Microarchitecture Basics



# Now That We Have an ISA

---

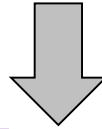
- How do we implement it?
- i.e., how do we design a system that obeys the hardware/software interface?
- Aside: “System” can be solely hardware or a combination of hardware and software
  - Remember “Translation of ISAs” (Transmeta example in Lec. 2)
  - A **virtual ISA** can be converted by “software” into an **implementation ISA**
- We will assume “hardware” for most lectures

# How Does a Machine Process Instructions?

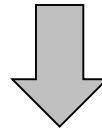
---

- 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

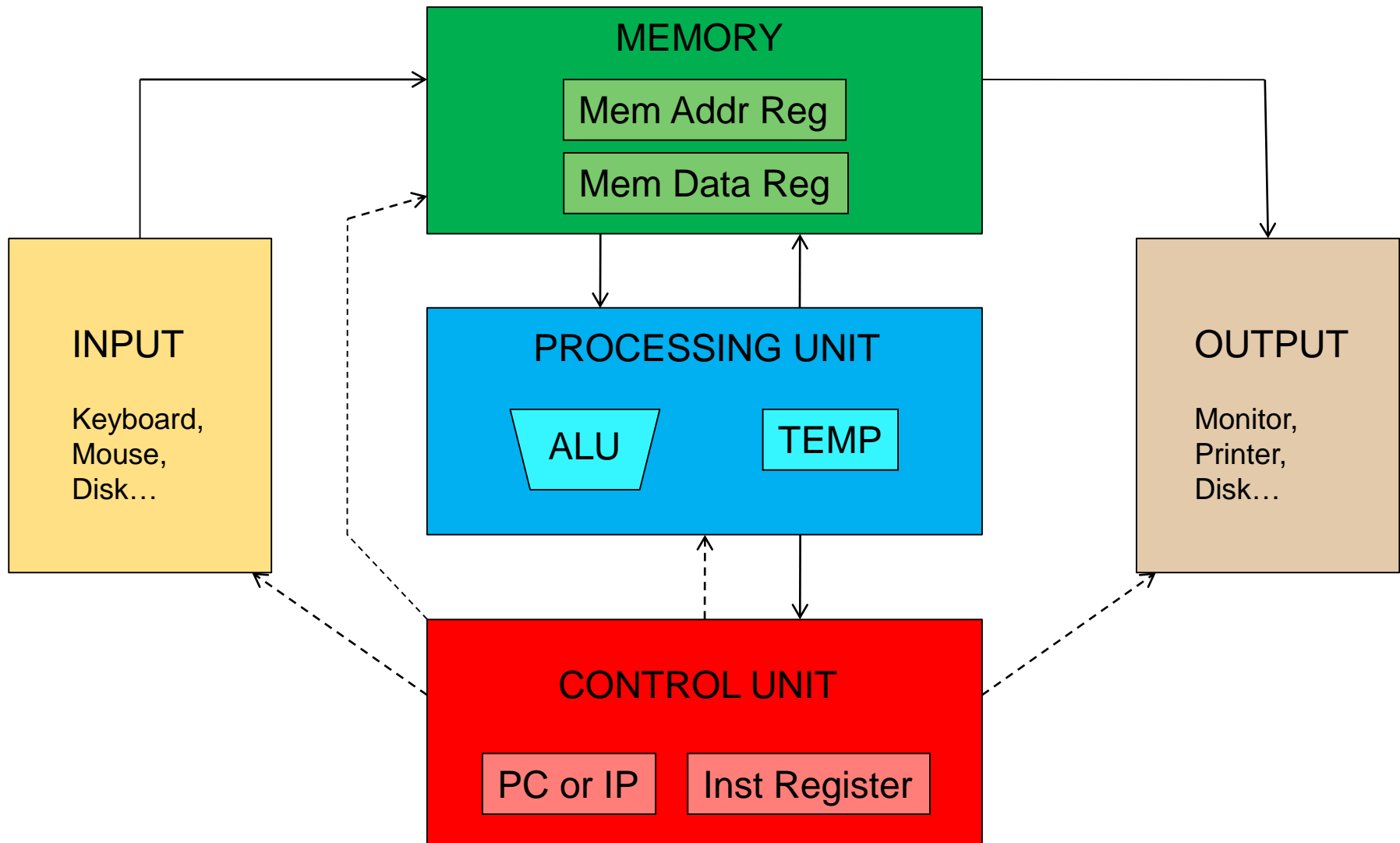
---

**Stored program**

**Sequential instruction processing**

# Recall: The Von Neumann Model

---



# The “Process Instruction” Step

---

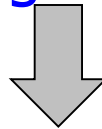
- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no “intermediate states” between AS 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:  $AS \rightarrow AS'$  (transform AS to AS' in a single clock cycle)
    - Choice 2:  $AS \rightarrow AS+MS1 \rightarrow AS+MS2 \rightarrow AS+MS3 \rightarrow AS'$  (take multiple clock cycles to transform AS to AS')

# A Very Basic Instruction Processing Engine

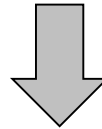
---

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

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



Process instruction in one clock cycle

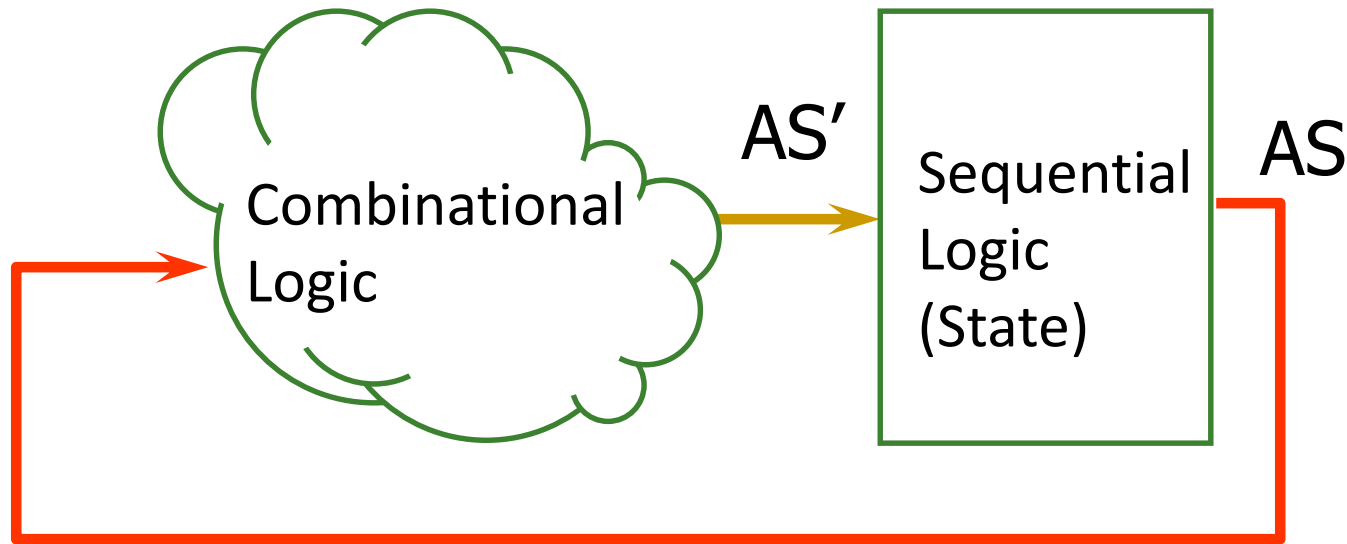


AS' = Architectural (programmer visible) state  
at the end of a clock cycle

# A Very Basic Instruction Processing Engine

---

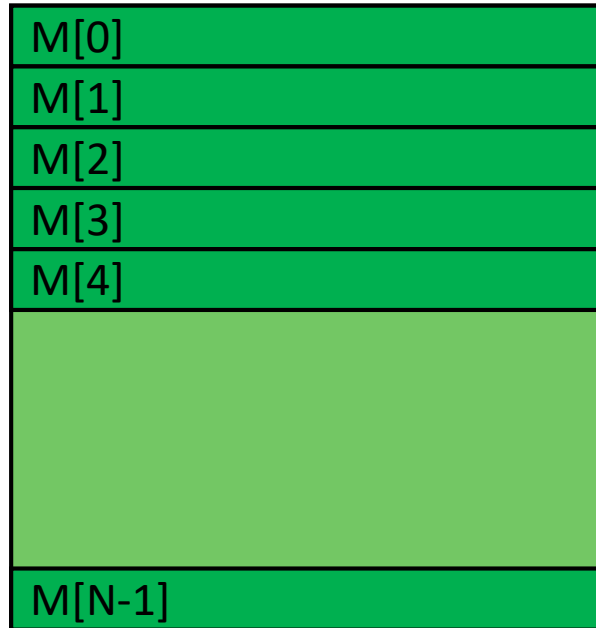
- Single-cycle machine



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

Program Counter

memory address  
of the current instruction

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



# Single-cycle vs. Multi-cycle Machines

---

## ■ Single-cycle machines

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

## ■ Multi-cycle machines

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

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

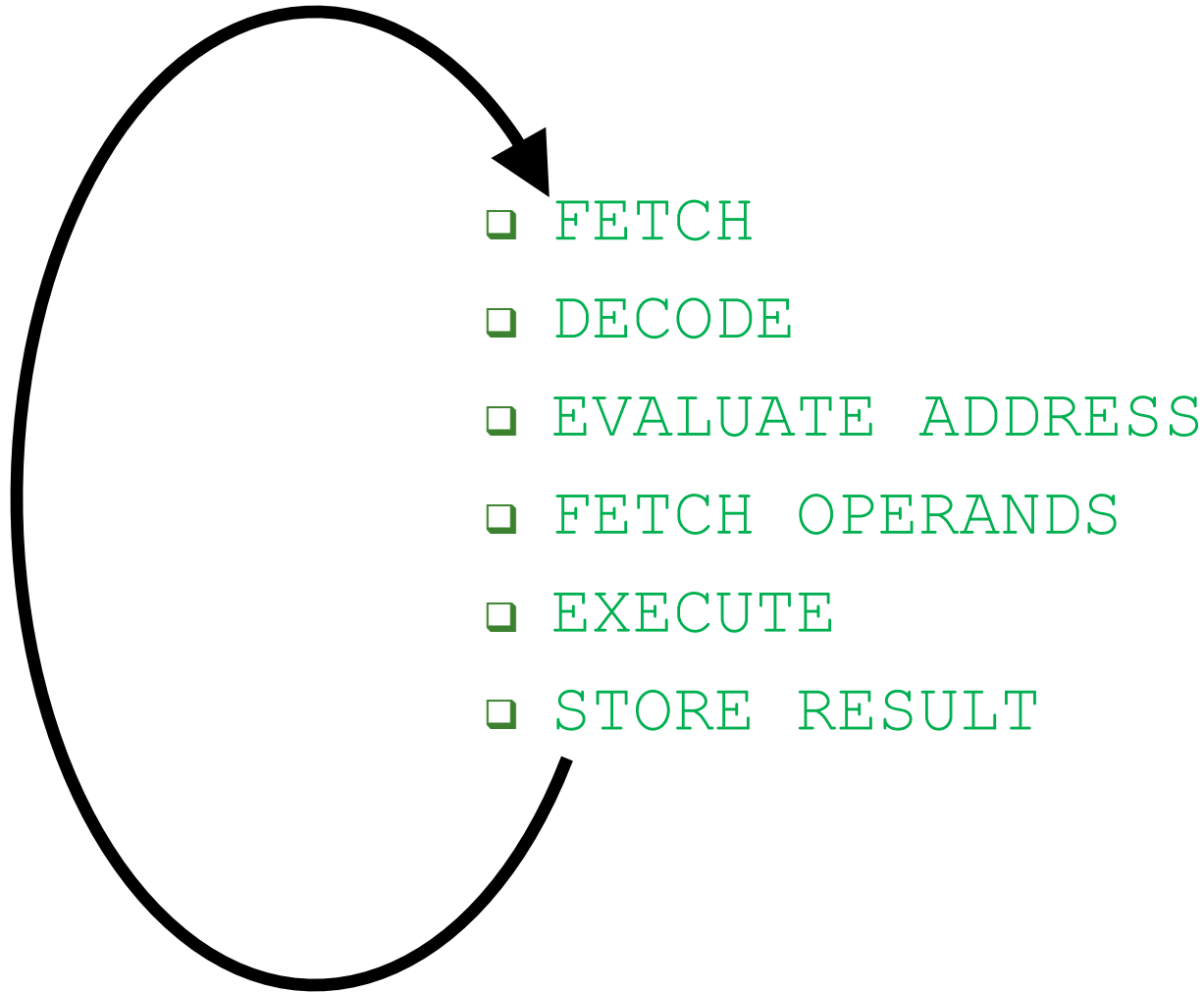
# Instruction Processing “Cycle”

---

- Instructions are processed under the direction of a “control unit” step by step.
- Instruction cycle: Sequence of steps to process an instruction
- Fundamentally, there are six steps:
  - Fetch
  - Decode
  - Evaluate Address
  - Fetch Operands
  - Execute
  - Store Result
- Not all instructions require all six steps (see P&P Ch. 4)

# Recall: The Instruction Processing “Cycle”

---



# Instruction Processing “Cycle” vs. Machine Clock Cycle

---

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

# Instruction Processing Viewed Another Way

---

- Instructions transform Data (AS) to Data' (AS')
- This transformation is done by functional units
  - Units that “operate” on data
- These units need to be told what to do to the data
- An instruction processing engine consists of two components
  - **Datapath**: Consists of hardware elements that deal with and transform data signals
    - functional units that operate on data
    - hardware structures (e.g. wires and muxes) that enable the flow of data into the functional units and registers
    - storage units that store data (e.g., registers)
  - **Control logic**: Consists of hardware elements that determine control signals, i.e., signals that specify what the datapath elements should do to the data

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

---

- Single-cycle machine:
  - Control signals are generated in the same clock cycle as the one during which data signals are operated on
  - Everything related to an instruction happens in one clock cycle (serialized processing)
- Multi-cycle machine:
  - Control signals needed in the next cycle can be generated in the current cycle
  - Latency of control processing can be overlapped with latency of datapath operation (more parallelism)
- We will see the difference clearly in *microprogrammed multi-cycle microarchitectures*

# Many Ways of Datapath and Control Design

---

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

Here, we have  
two degrees of freedom  
to optimize independently



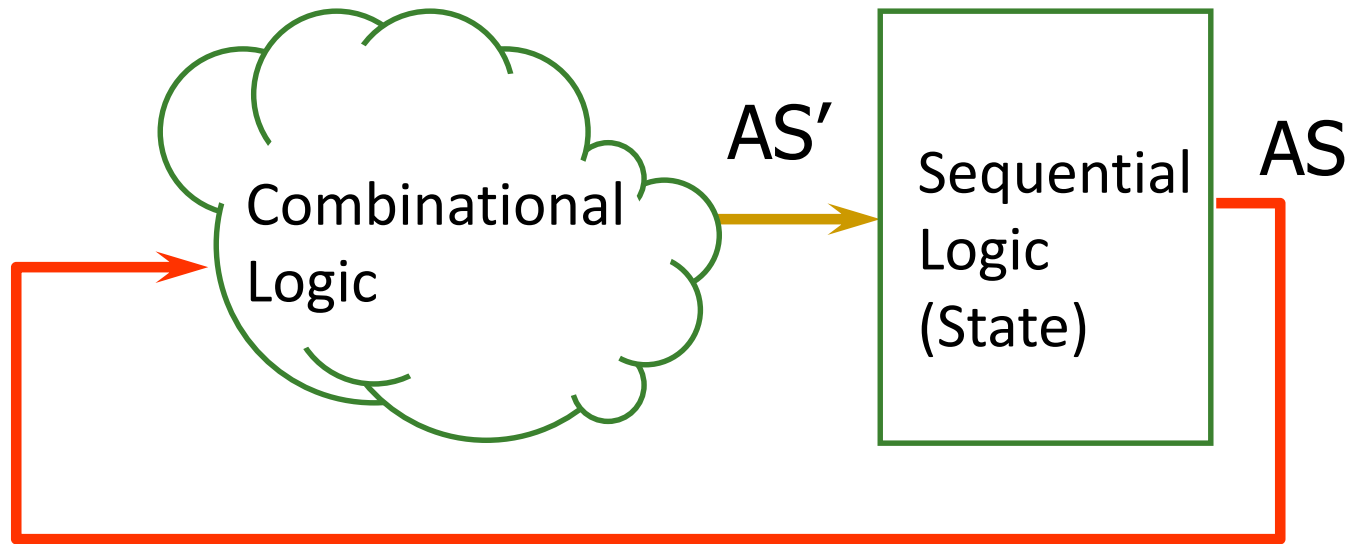
# A Single-Cycle Microarchitecture

## *A Closer Look*

# Remember...

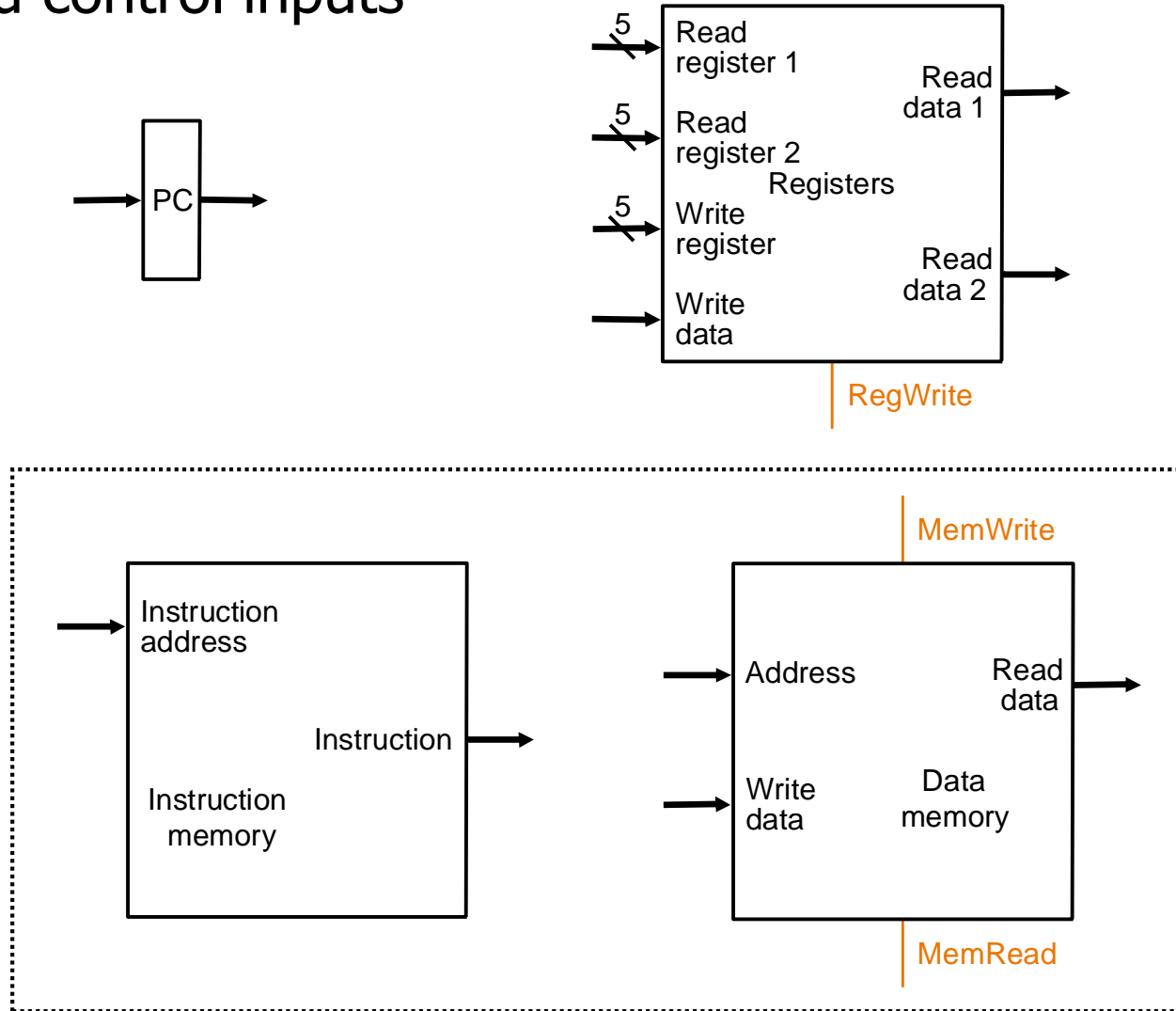
---

- Single-cycle machine

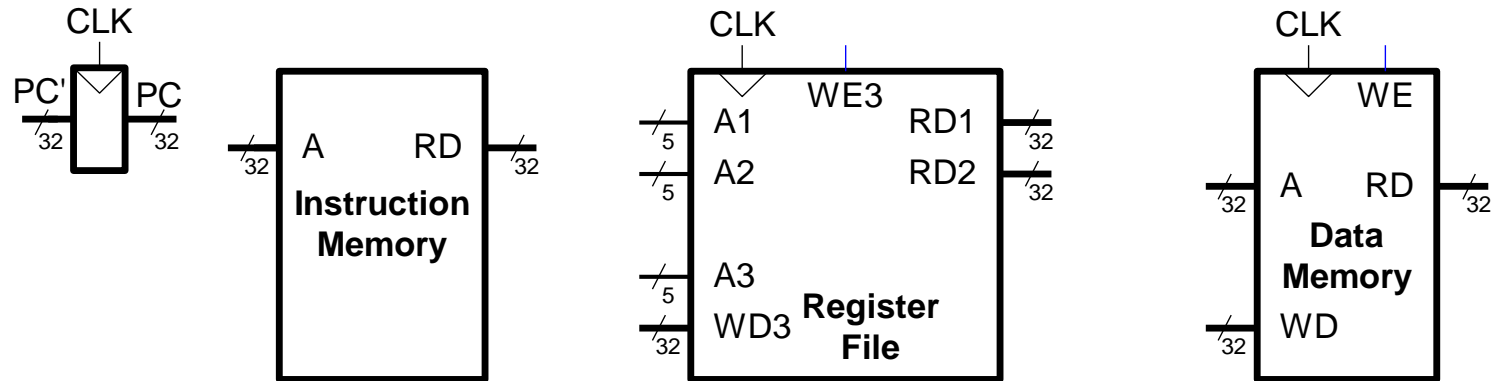


# Let's Start with the State Elements

## ■ 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.

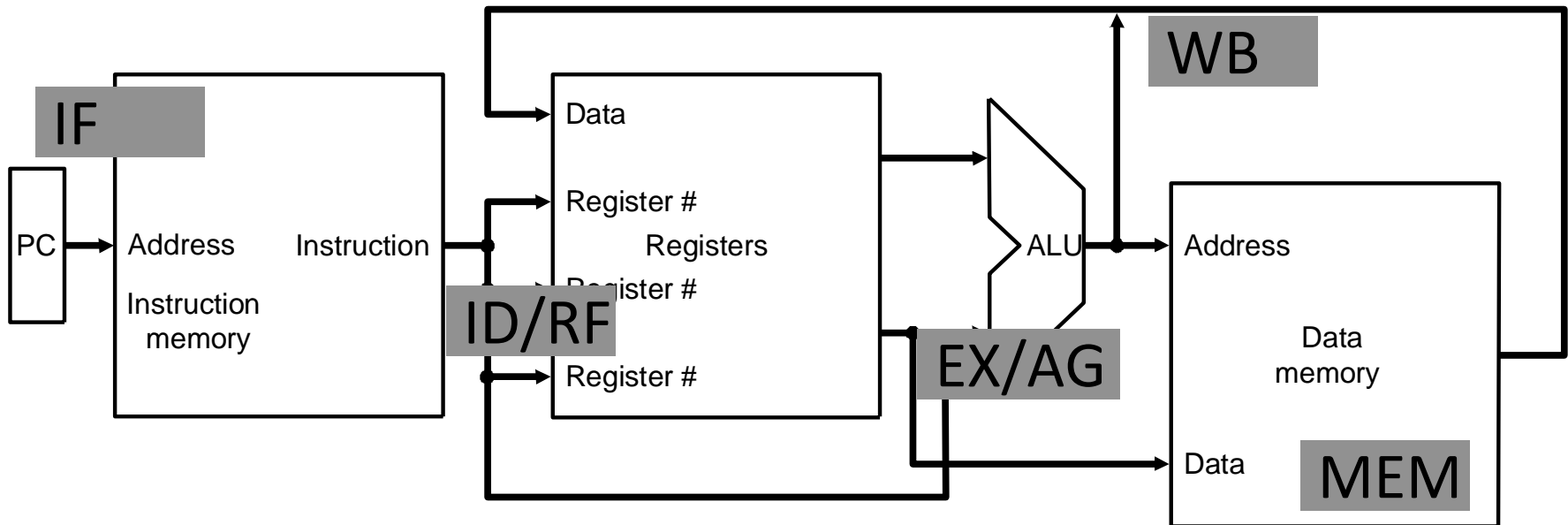
# For Now, We Will Assume

---

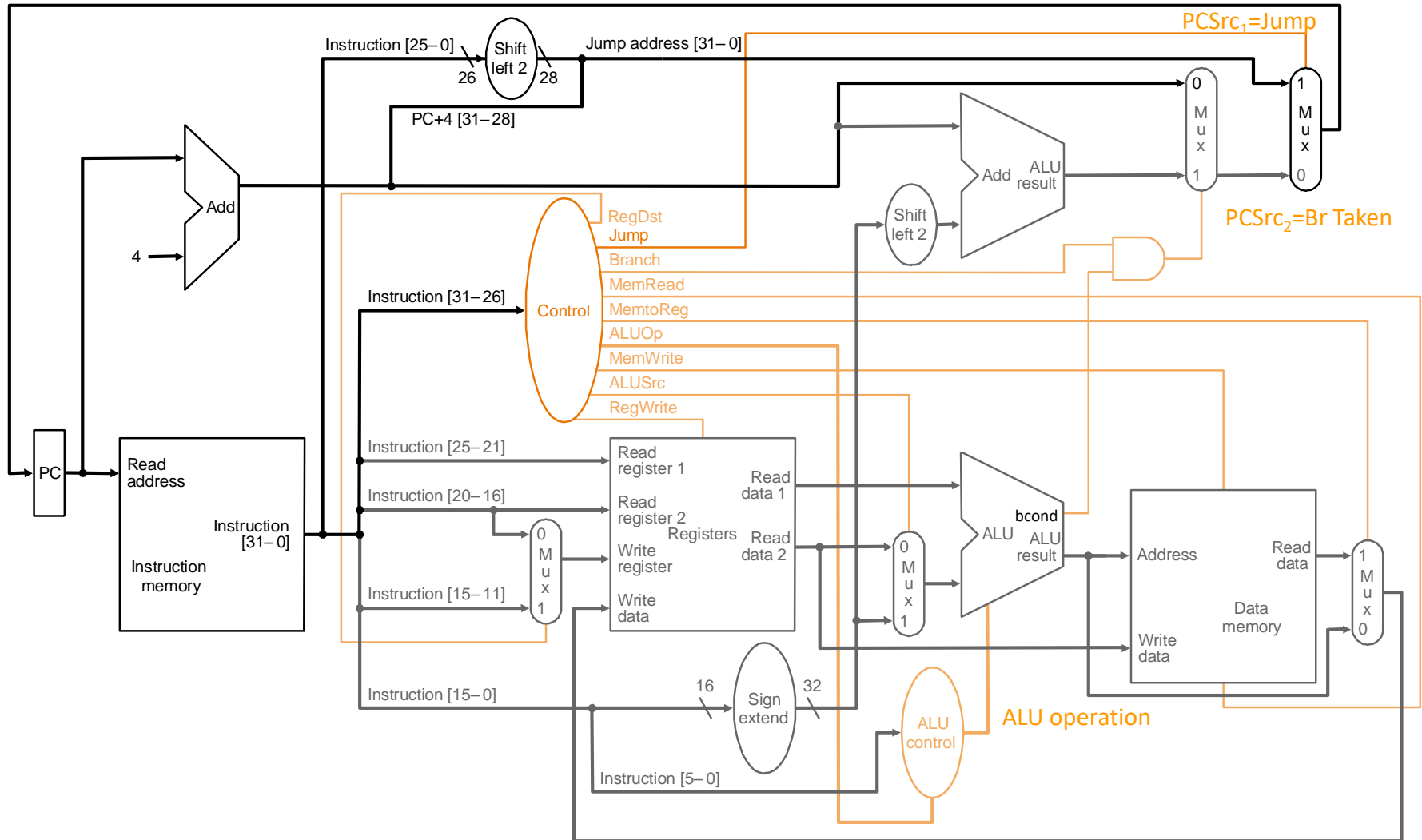
- “Magic” memory and register file
- Combinational read
  - output of the read data port is a combinational function of the register file contents and the corresponding read select port
- Synchronous write
  - the selected register is updated on the positive edge clock transition when write enable is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

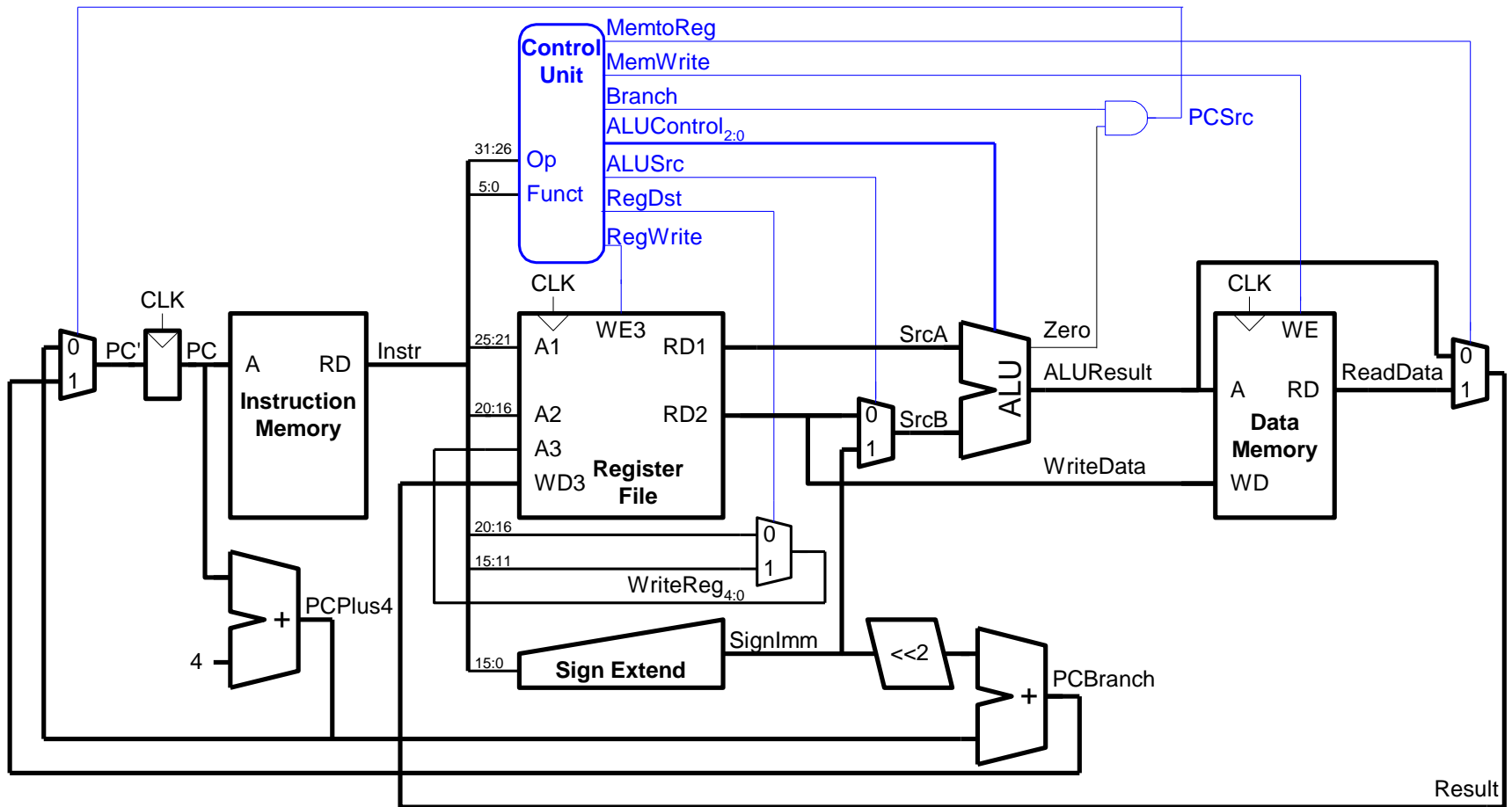


# What Is To Come: The Full MIPS Datapath



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

# 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

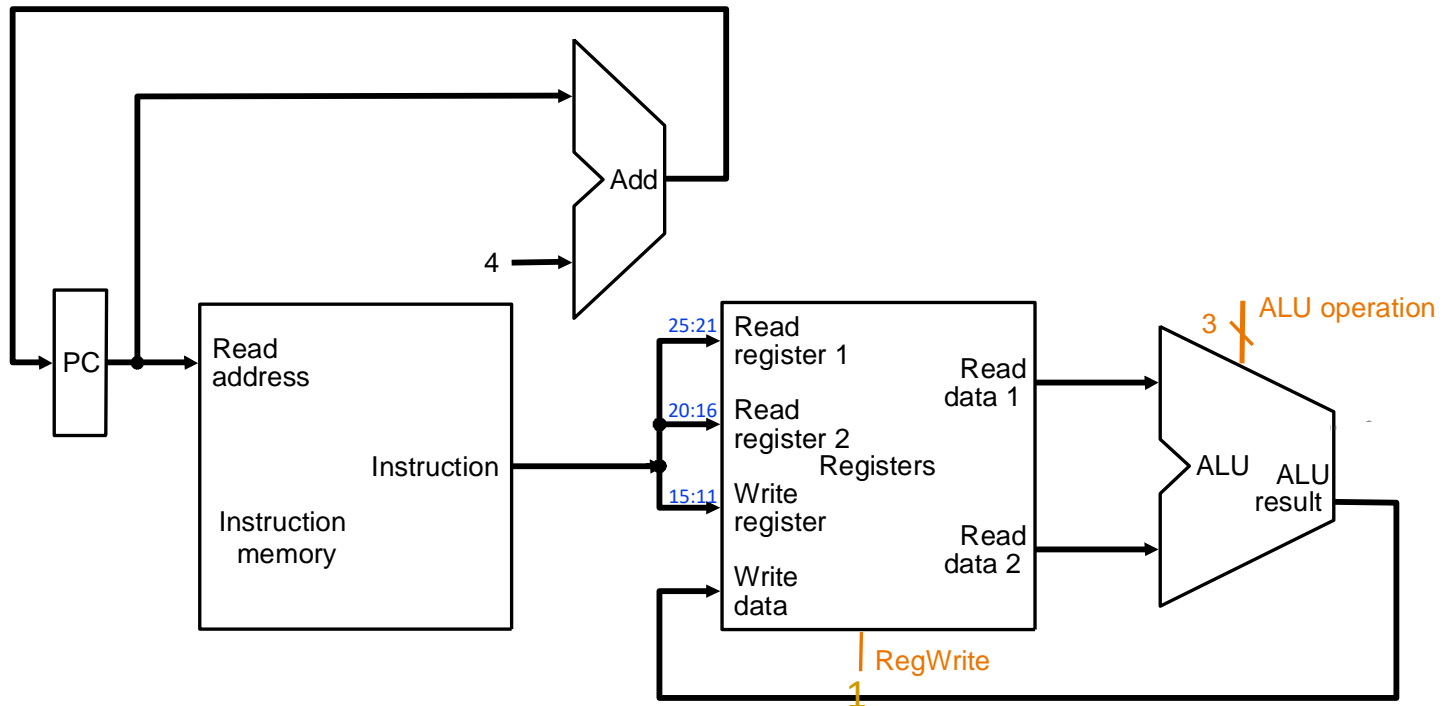
0	rs	rt	rd	0	add (32)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

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

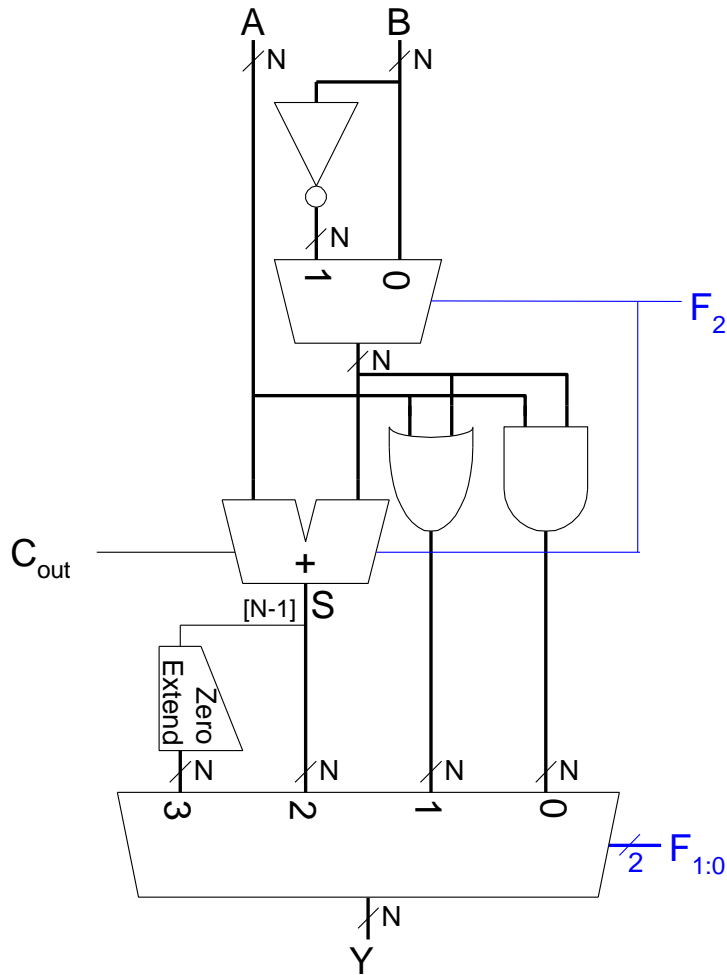


IF	ID	EX	MEM	WB
----	----	----	-----	----

Combinational  
state update logic

# Example: ALU Design

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



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

# I-Type ALU Instructions

- I-type: 2 register operands and 1 immediate

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

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

Machine Encoding

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

I-Type

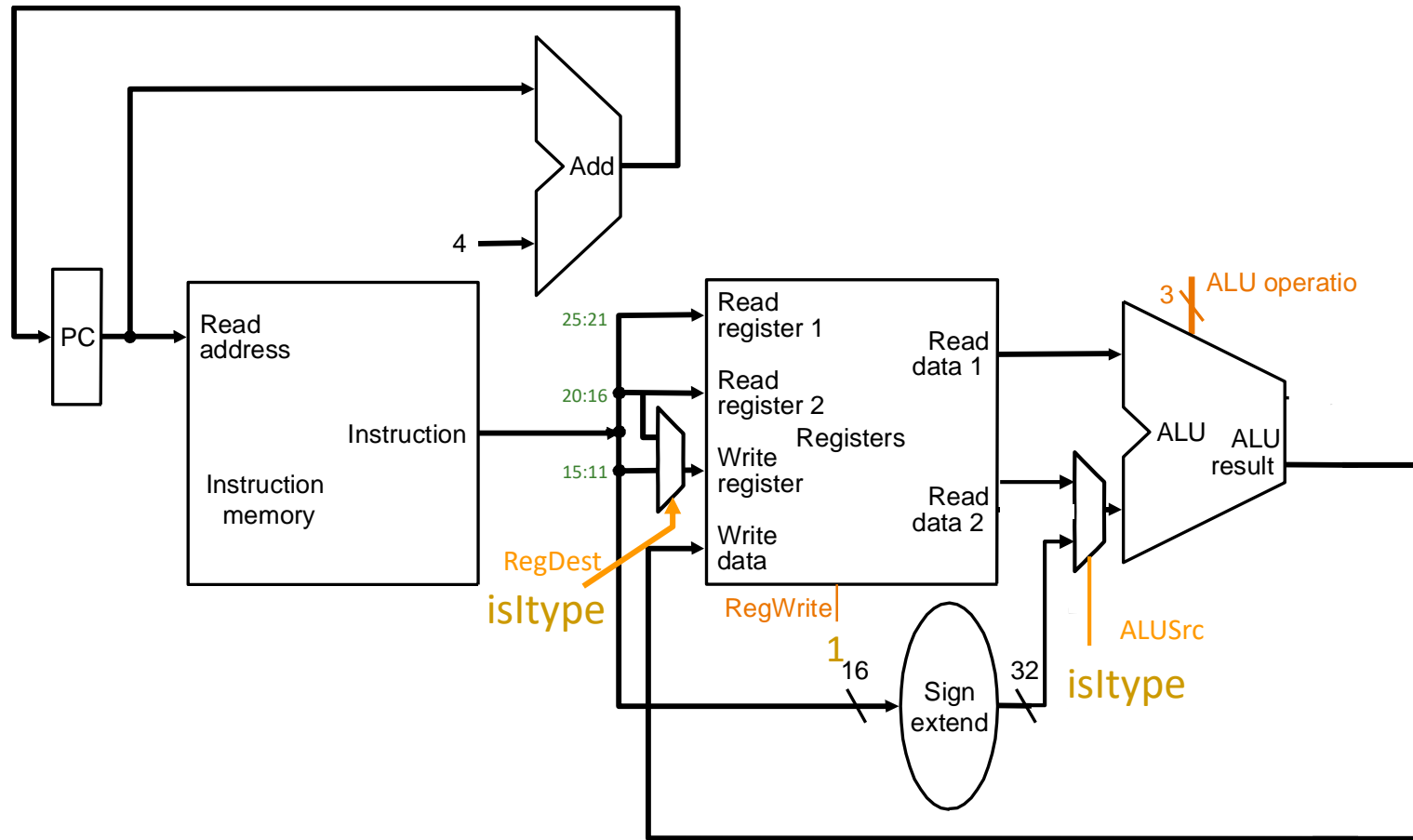
- Semantics

if MEM[PC] == addi rs rt immediate

PC  $\leftarrow$  PC + 4

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

# Datapath for R and I-Type ALU Insts.



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



Combinational  
state update logic

# Recall: ADD with one Literal in LC-3

## ■ ADD assembly and machine code

LC-3 assembly

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

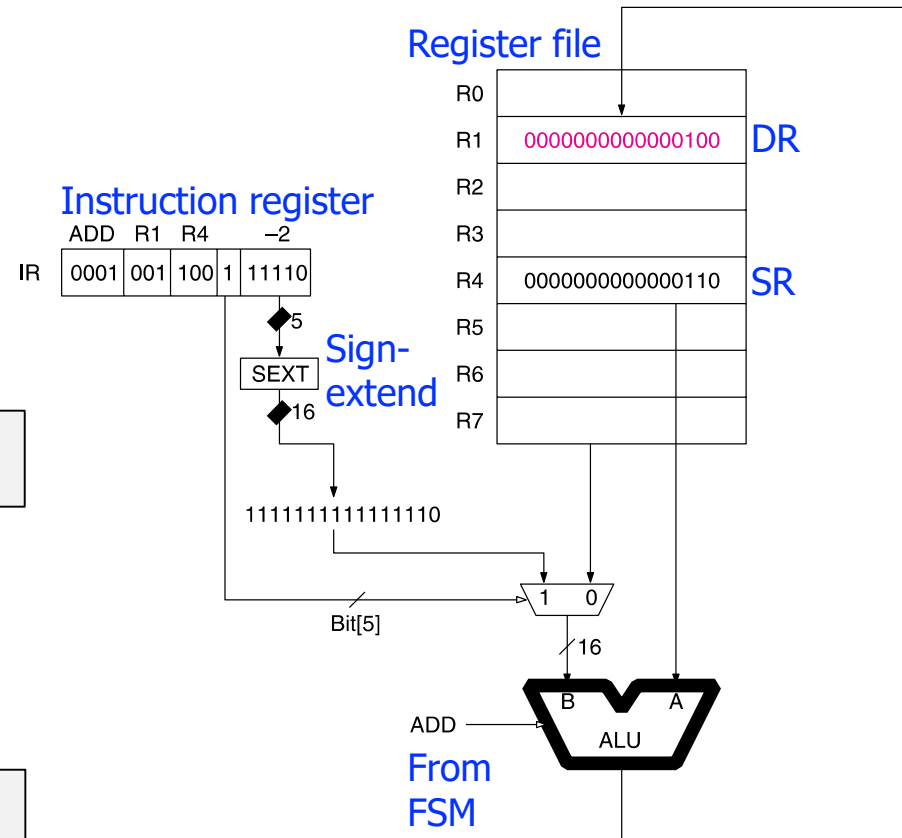
Field Values

OP	DR	SR		imm5
1	1	4	1	-2

Machine Code

OP	DR	SR		imm5
0001	001	100	1	11110

15 12 11 9 8 6 5 4 0



# Single-Cycle Datapath for *Data Movement Instructions*



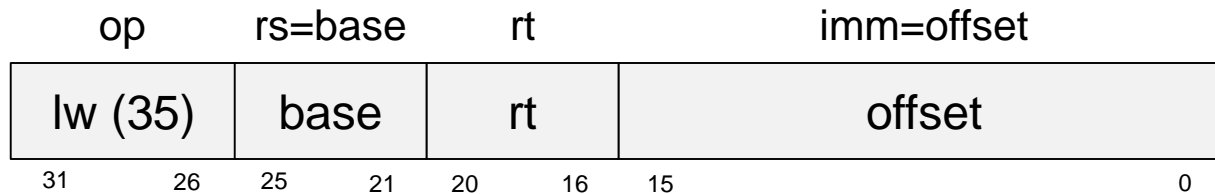
# Load Instructions

- Load 4-byte word

MIPS assembly

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

Machine Encoding

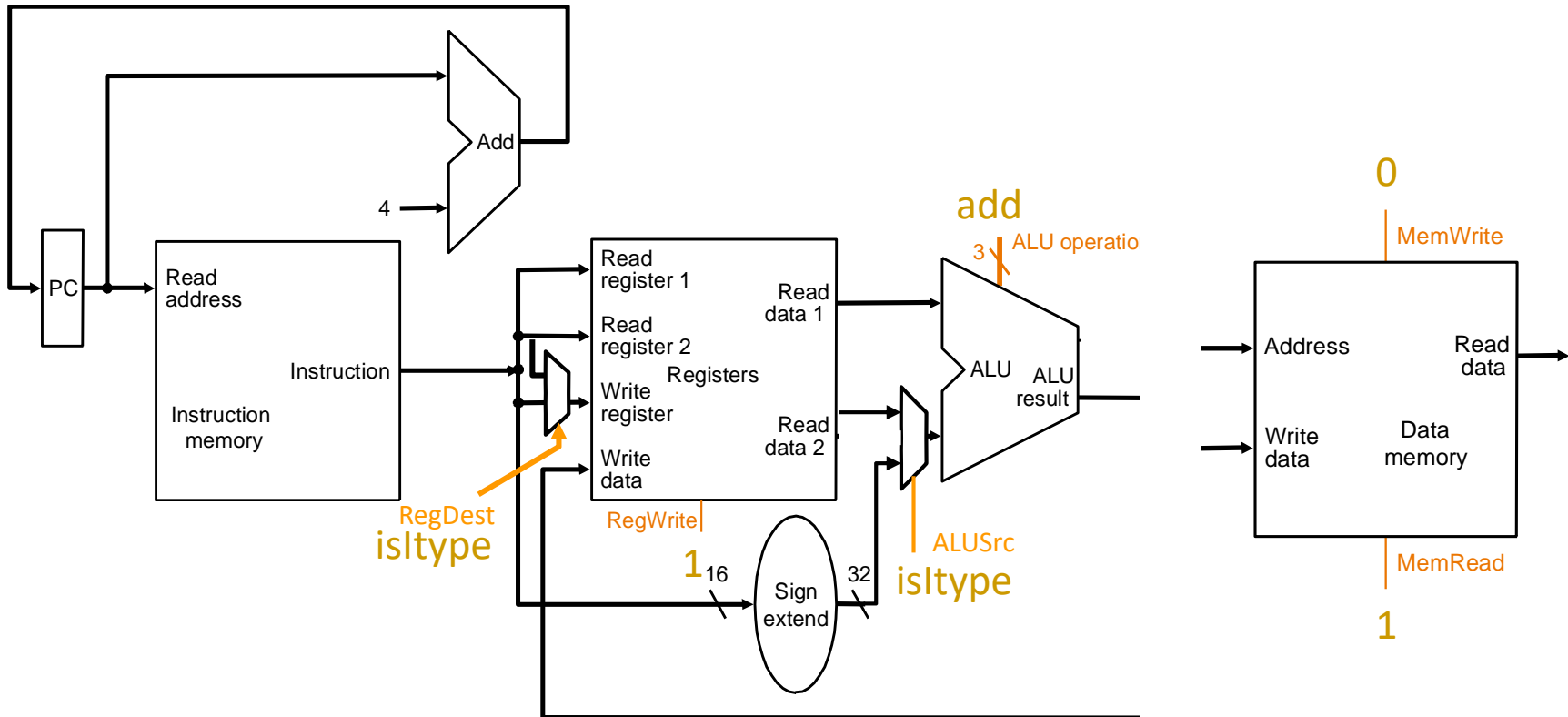


I-Type

- Semantics

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

# LW Datapath



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

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

Combinational  
state update logic

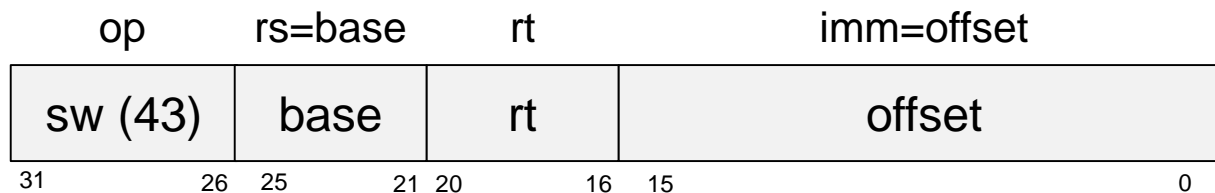
# Store Instructions

## ■ Store 4-byte word

MIPS assembly

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

Machine Encoding



I-Type

## ■ Semantics

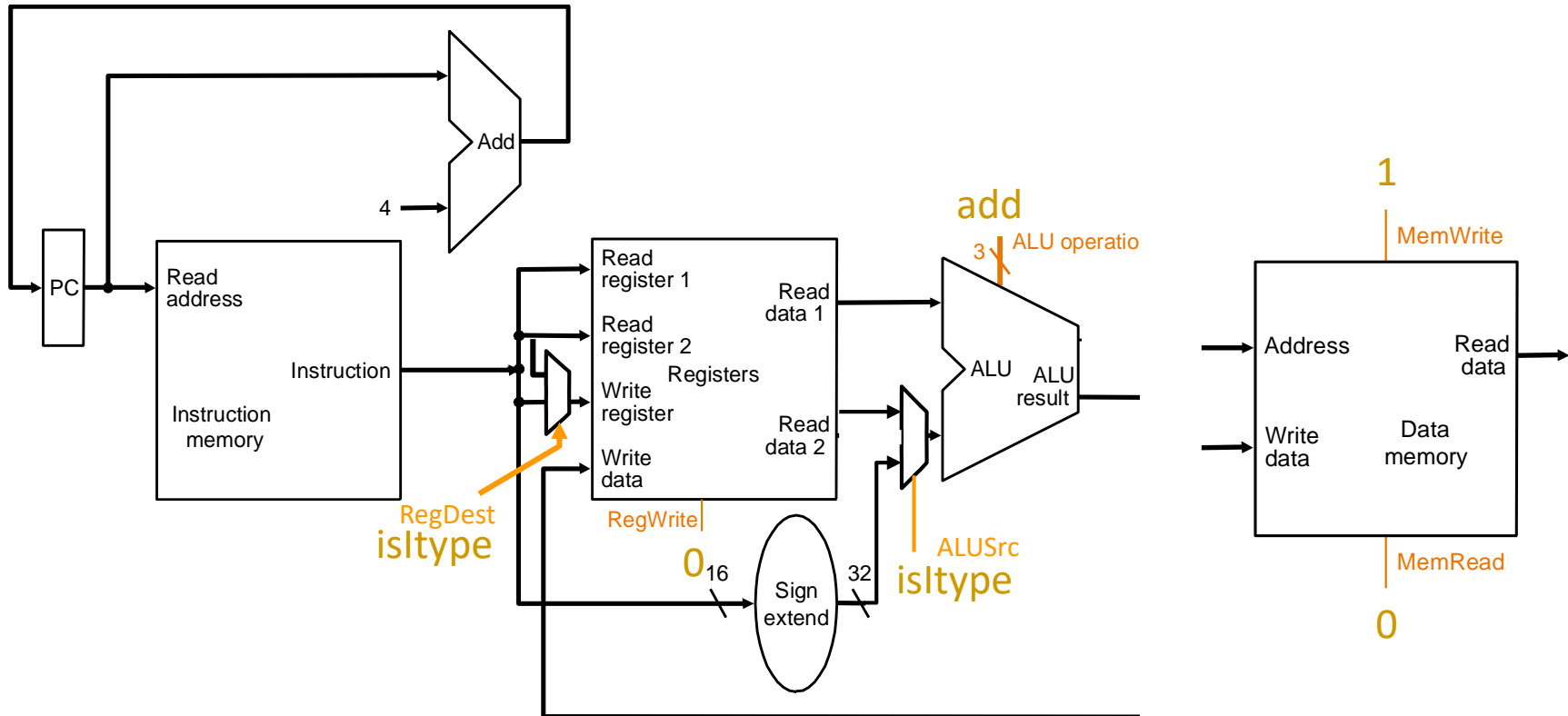
if  $\text{Mem}[\text{PC}] == \text{sw rt offset}_{16}(\text{base})$

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

$\text{EA} = \text{sign-extend}(\text{offset}) + \text{GPR}(\text{base})$

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

# SW Datapath



```

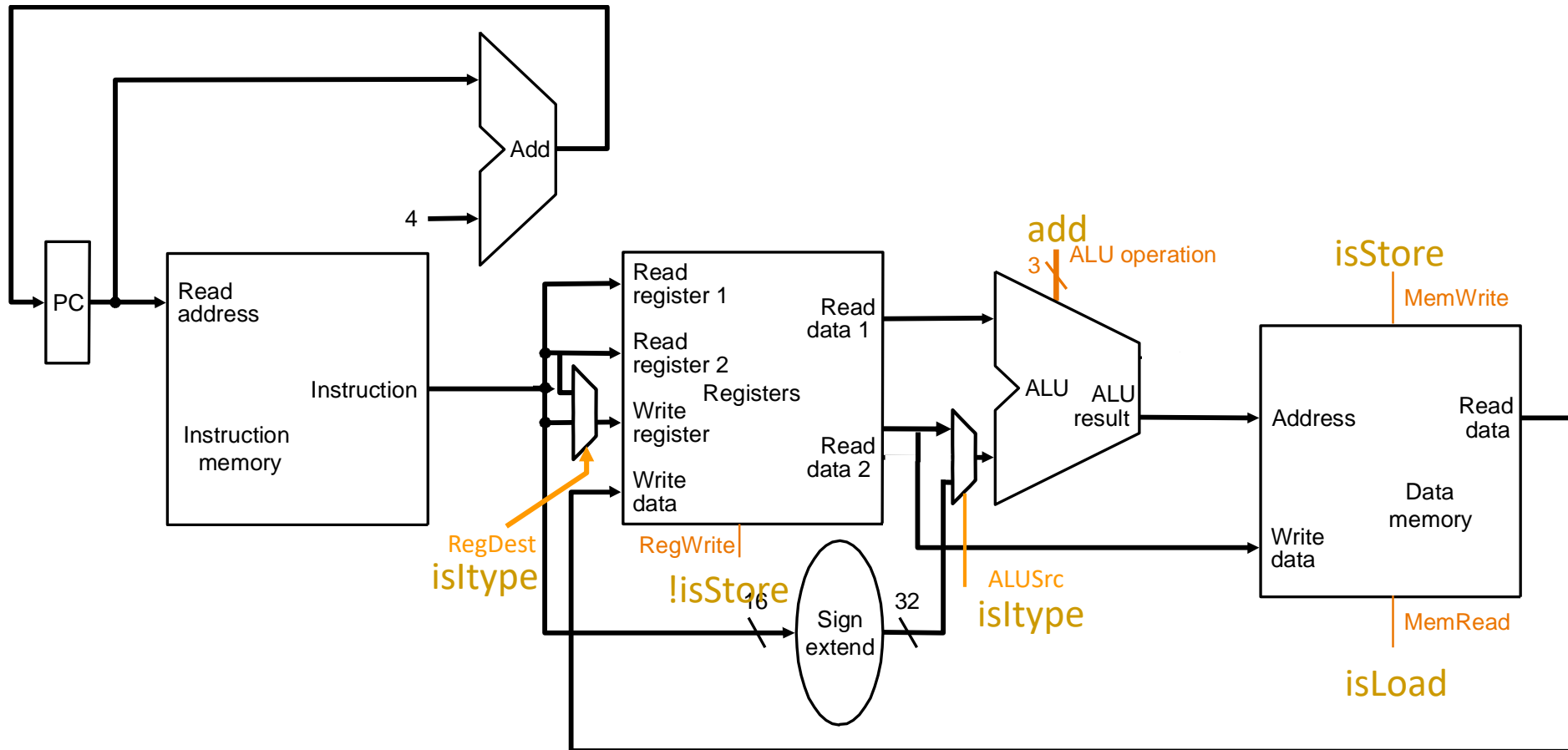
if MEM[PC]==SW rt offset16 (base)
    EA = sign-extend(offset) + GPR[base]
    MEM[ translate(EA) ] ← GPR[rt]
    PC ← PC + 4

```

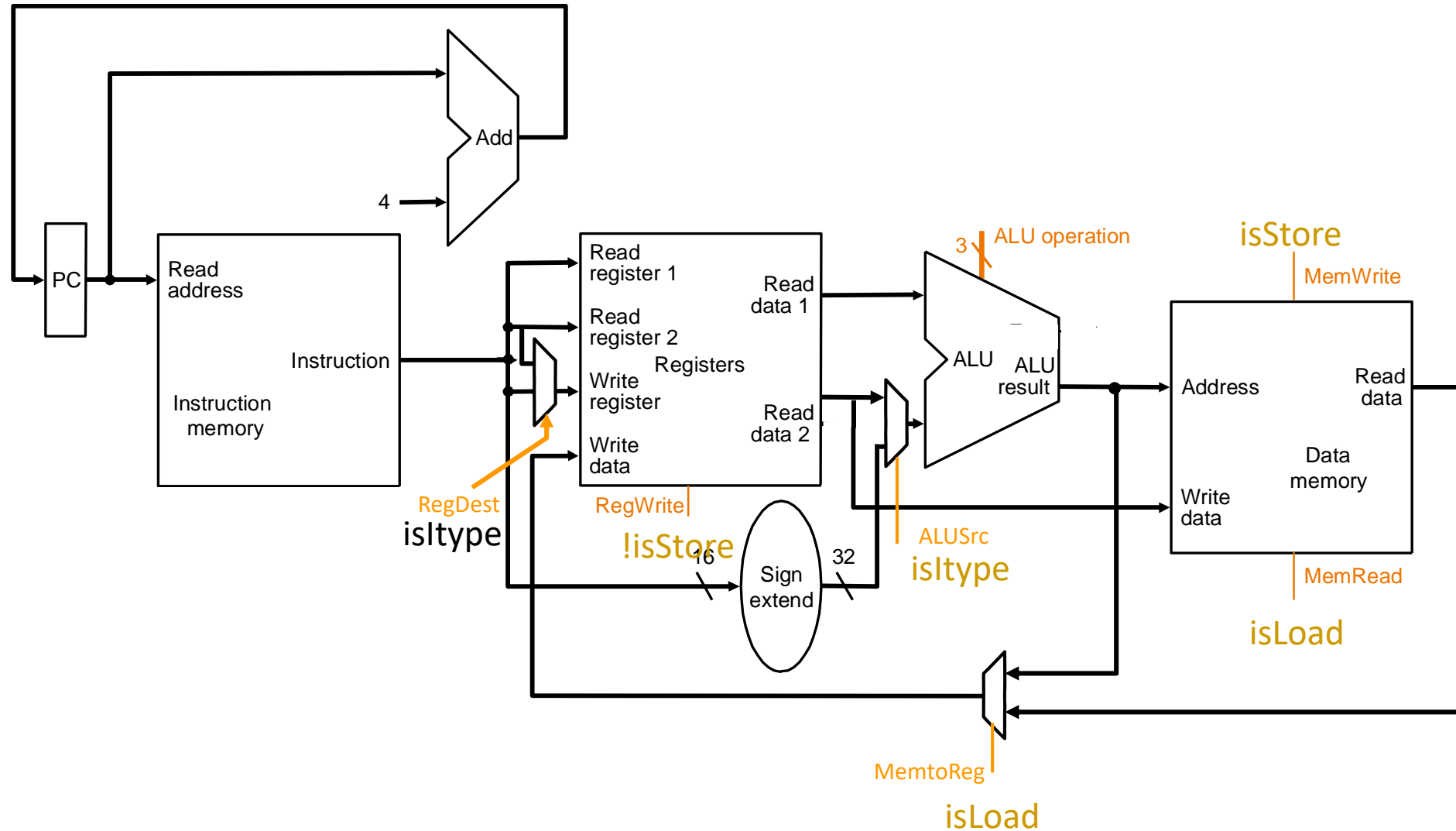


## Combinational state update logic <sup>76</sup>

# Load-Store Datapath



## Datapath for Non-Control-Flow Insts.

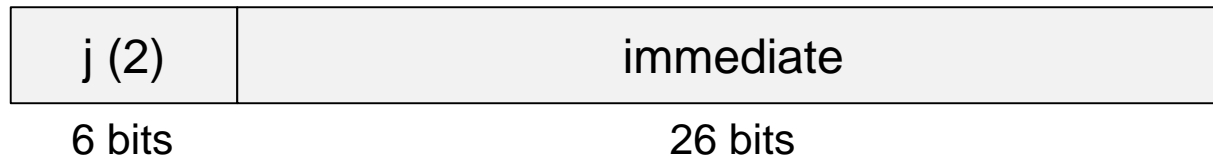


# Single-Cycle Datapath for *Control Flow Instructions*

# Jump Instruction

- Unconditional branch or jump

j    target



J-Type

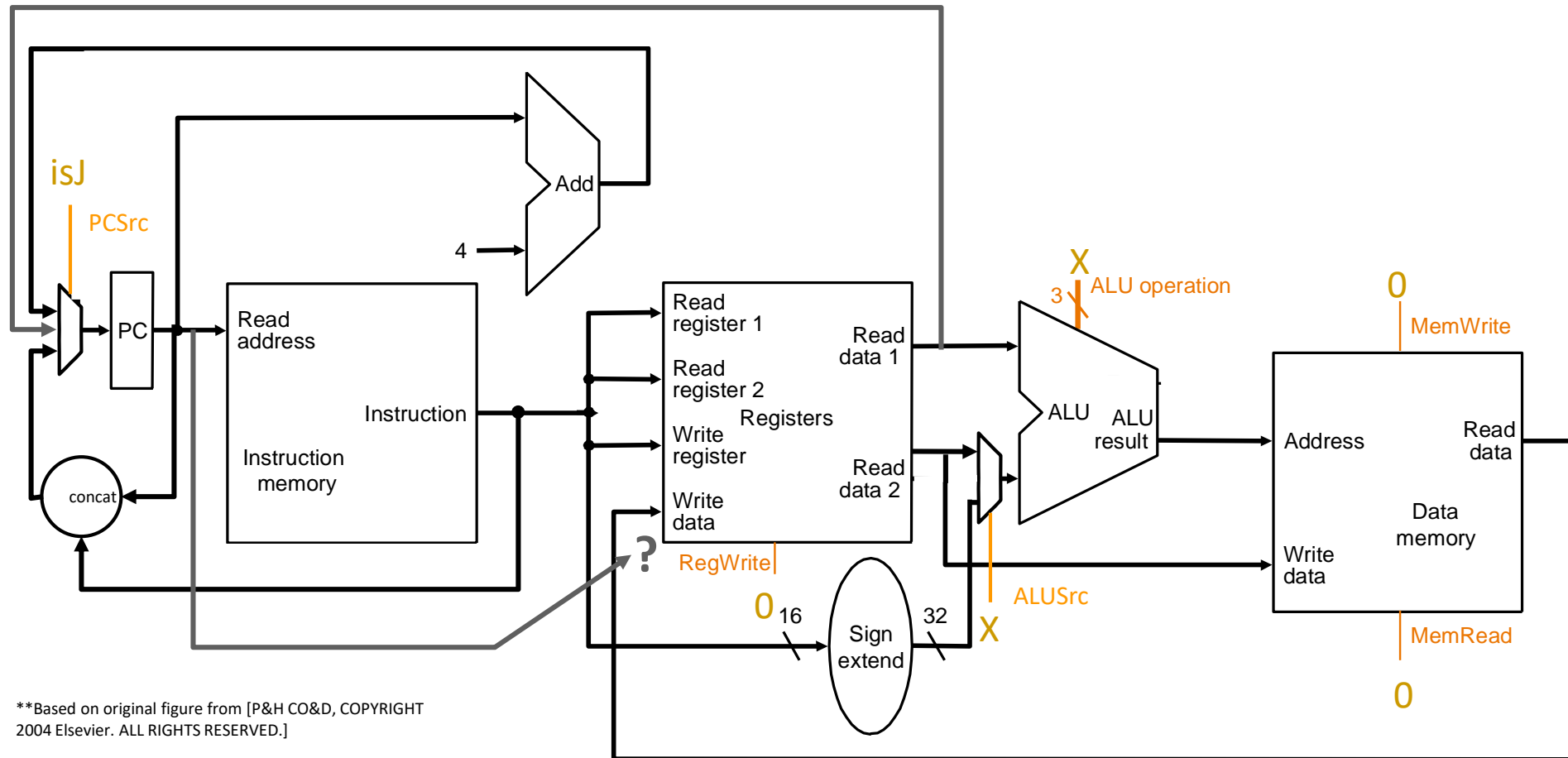
- 2 = opcode
- immediate (target) = target address

- Semantics

if  $\text{MEM}[\text{PC}] == j \text{ immediate}_{26}$   
     $\text{target} = \{ \text{PC} + [31:28], \text{immediate}_{26}, 2' \text{ b}00 \}$   
     $\text{PC} \leftarrow \text{target}$



# Unconditional Jump Datapath



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

What about JR, JAL, JALR?

# Other Jumps in MIPS

---

- ❑ jal: jump and link (function calls)
  - Semantics
  - if  $\text{MEM}[\text{PC}] == \text{jal immediate}_{26}$ 
    - $\$ra \leftarrow \text{PC} + 4$
    - $\text{target} = \{ \text{PC}^\dagger[31:28], \text{immediate}_{26}, 2'b00 \}$
    - $\text{PC} \leftarrow \text{target}$
  
- ❑ jr: jump register
  - Semantics
  - if  $\text{MEM}[\text{PC}] == \text{jr rs}$ 
    - $\text{PC} \leftarrow \text{GPR}(rs)$
  
- ❑ jalr: jump and link register
  - Semantics
  - if  $\text{MEM}[\text{PC}] == \text{jalr rs}$ 
    - $\$ra \leftarrow \text{PC} + 4$
    - $\text{PC} \leftarrow \text{GPR}(rs)$

---

<sup>†</sup> This is the incremented PC

# Aside: MIPS Cheat Sheet

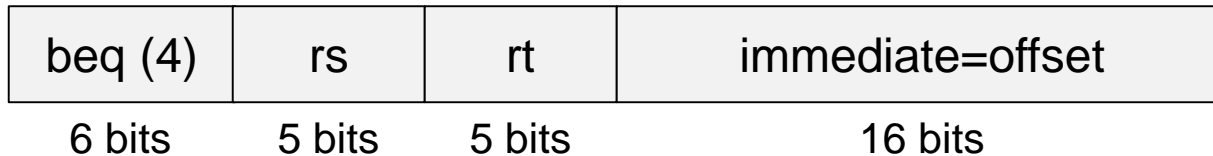
---

- [https://safari.ethz.ch/digitaltechnik/spring2018/lib/exe/fetch.php?media=mips\\_reference\\_data.pdf](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
```



I-Type

- Semantics (assuming no branch delay slot)

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

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

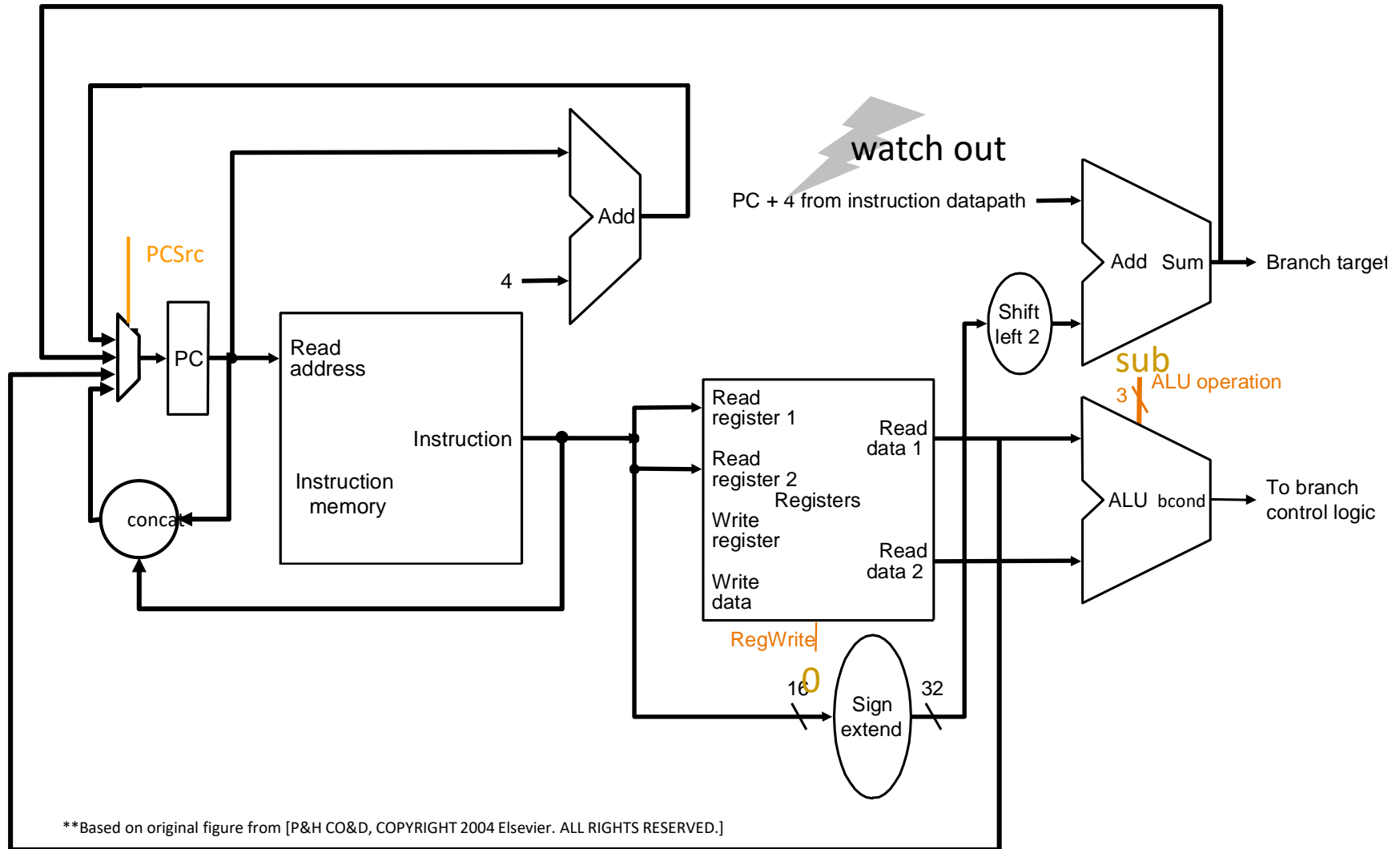
if  $\text{GPR}[rs] == \text{GPR}[rt]$  then  $\text{PC} \leftarrow \text{target}$

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

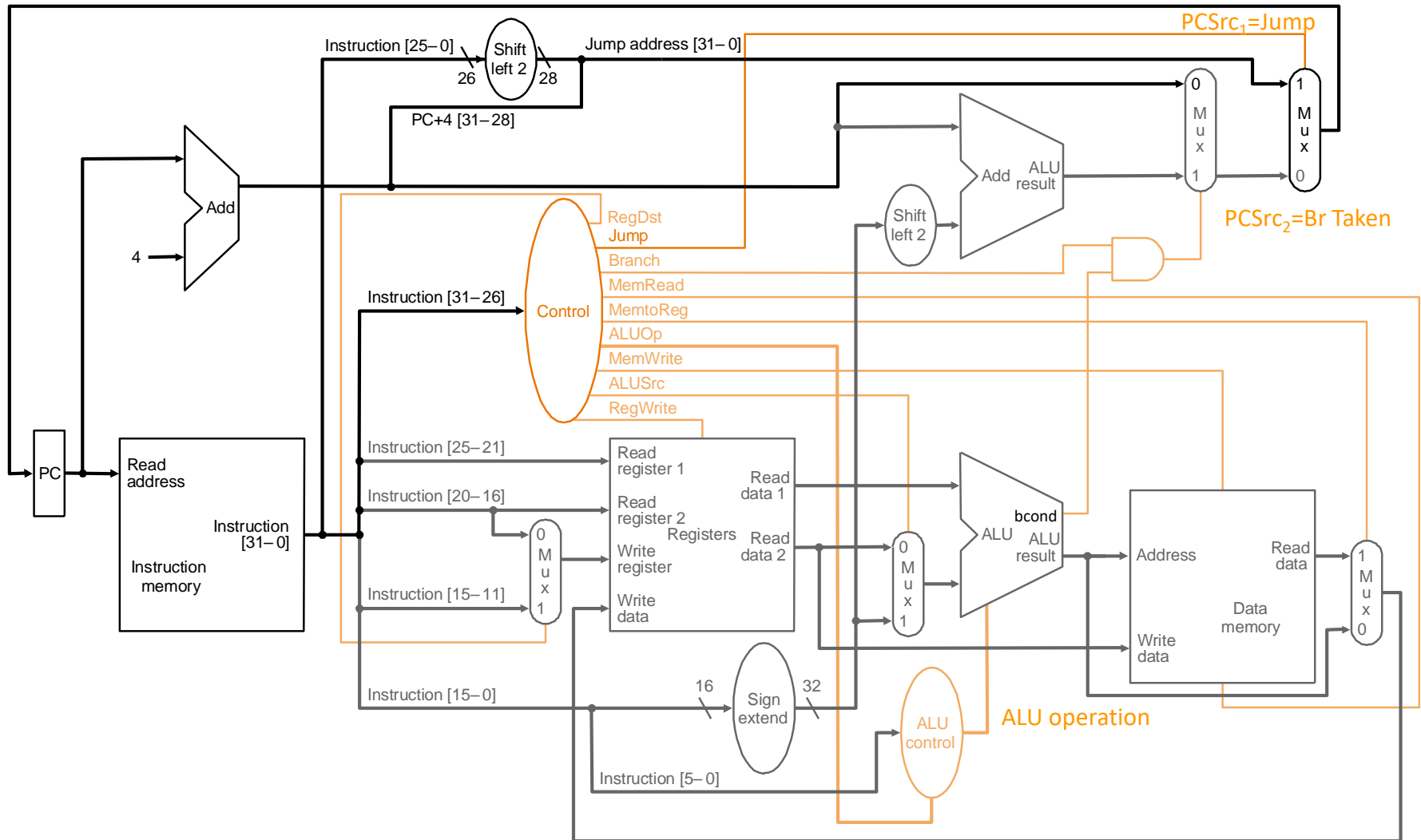
- Variations: beq, bne, blez, bgtz

<sup>†</sup> This is the incremented PC

# Conditional Branch Datapath (for you to finish)



# Putting It All Together



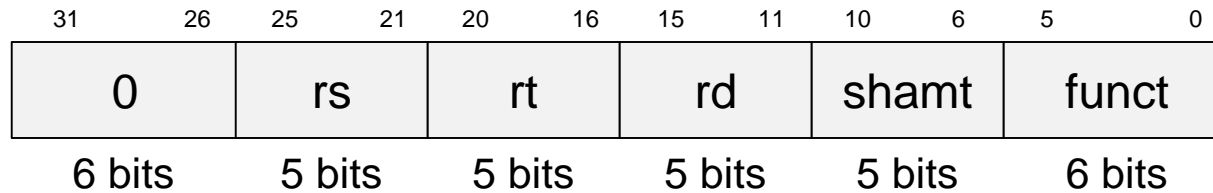
We did not cover the following slides in lecture.  
These are for your preparation for the next lecture

# Single-Cycle Control Logic

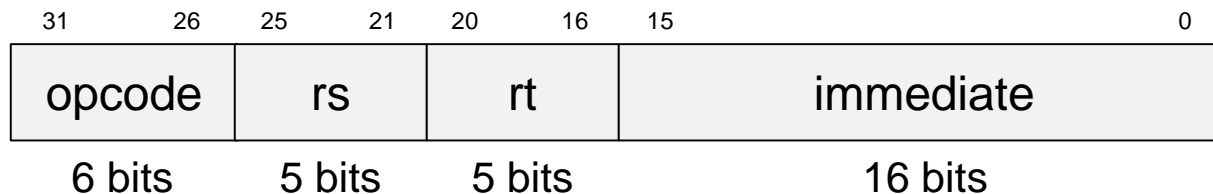


# Single-Cycle Hardwired Control

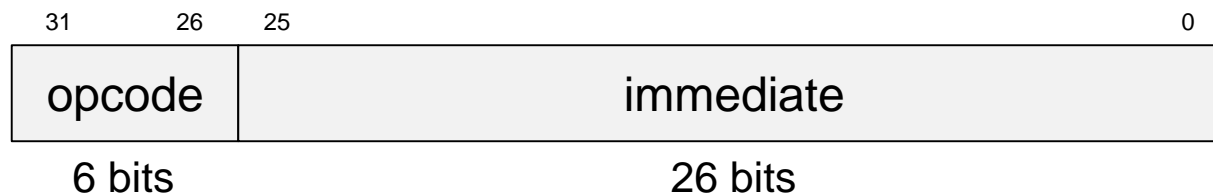
- As combinational function of **Inst=MEM[PC]**



R-Type



I-Type



J-Type

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

# Single-Bit Control Signals

---

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

# Single-Bit Control Signals

---

	When De-asserted	When asserted	Equation
MemRead	Memory read disabled	Memory read port return load value	<code>opcode==LW</code>
MemWrite	Memory write disabled	Memory write enabled	<code>opcode==SW</code>
PCSrc <sub>1</sub>	According to PCSrc <sub>2</sub>	next PC is based on 26-bit immediate jump target	<code>(opcode==J)    (opcode==JAL)</code>
PCSrc <sub>2</sub>	next PC = PC + 4	next PC is based on 16-bit immediate branch target	<code>(opcode==Bxx) &amp;&amp; "bcond is satisfied"</code>

# ALU Control

---

- case **opcode**

- ‘0’  $\Rightarrow$  select operation according to **funct**

- ‘ALUi’  $\Rightarrow$  selection operation according to **opcode**

- ‘LW’  $\Rightarrow$  select addition

- ‘SW’  $\Rightarrow$  select addition

- ‘Bxx’  $\Rightarrow$  select bcond generation function

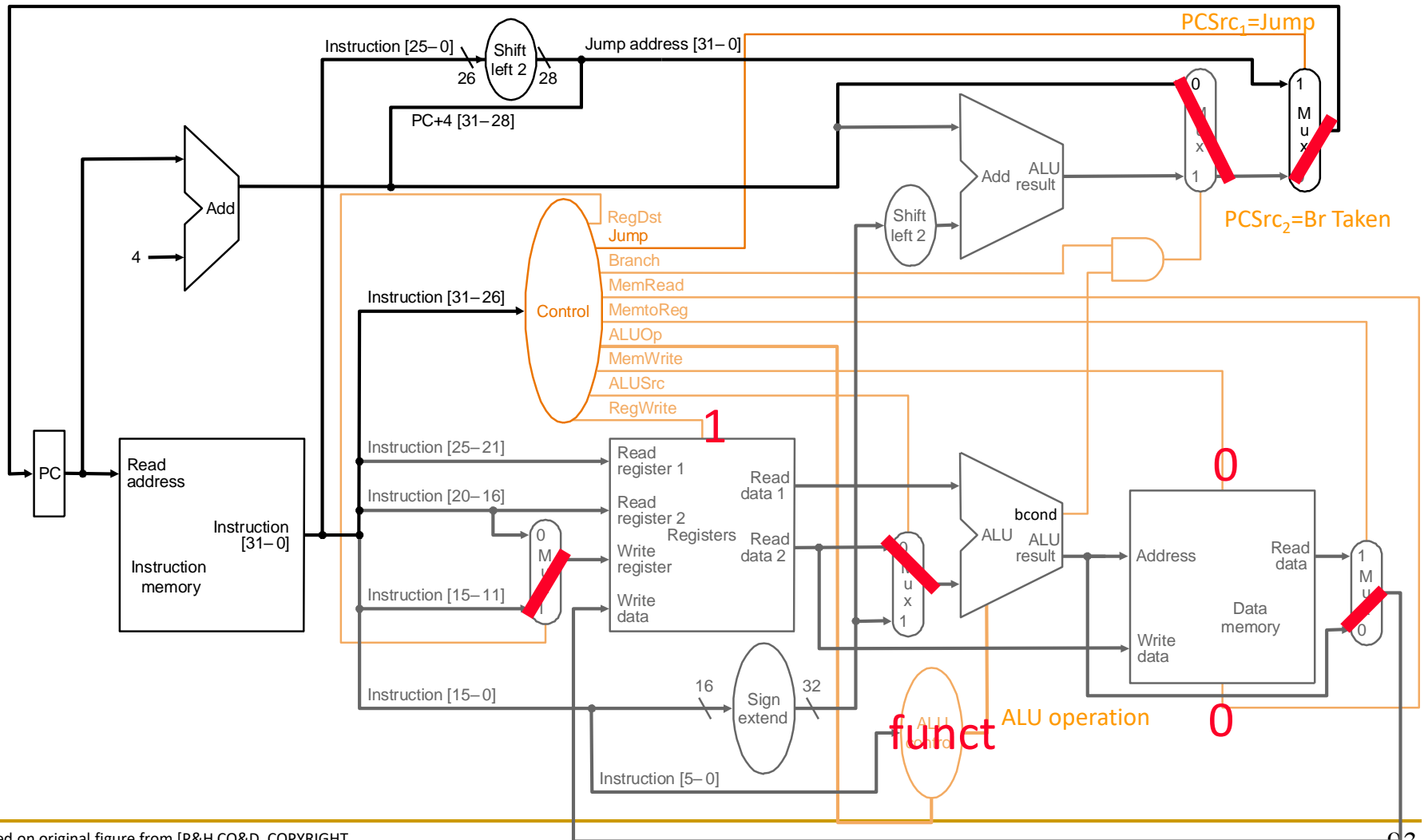
- \_\_\_  $\Rightarrow$  don't care

- Example ALU operations

- ❑ ADD, SUB, AND, OR, XOR, NOR, etc.

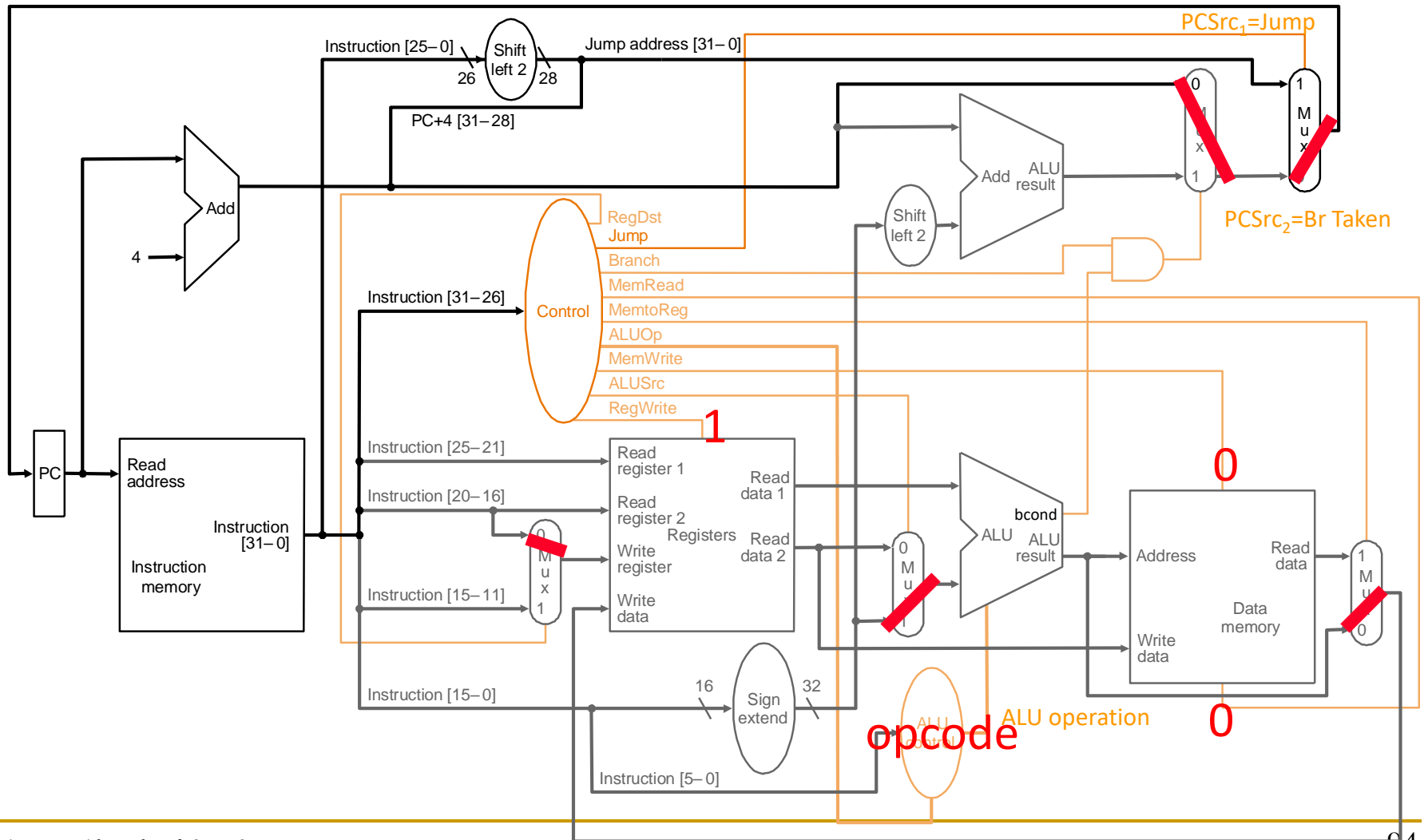
- ❑ bcond on equal, not equal, LE zero, GT zero, etc.

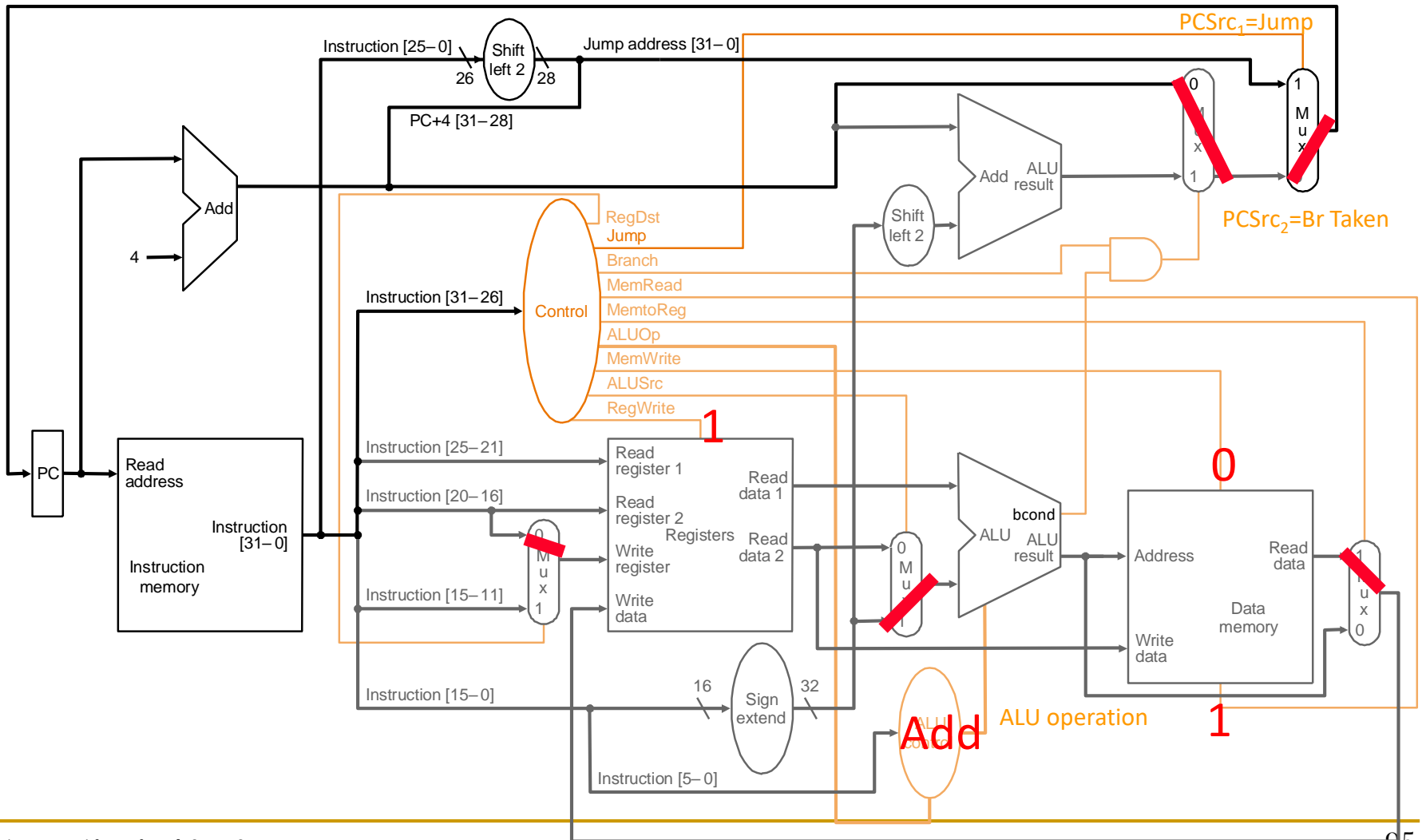
# R-Type ALU

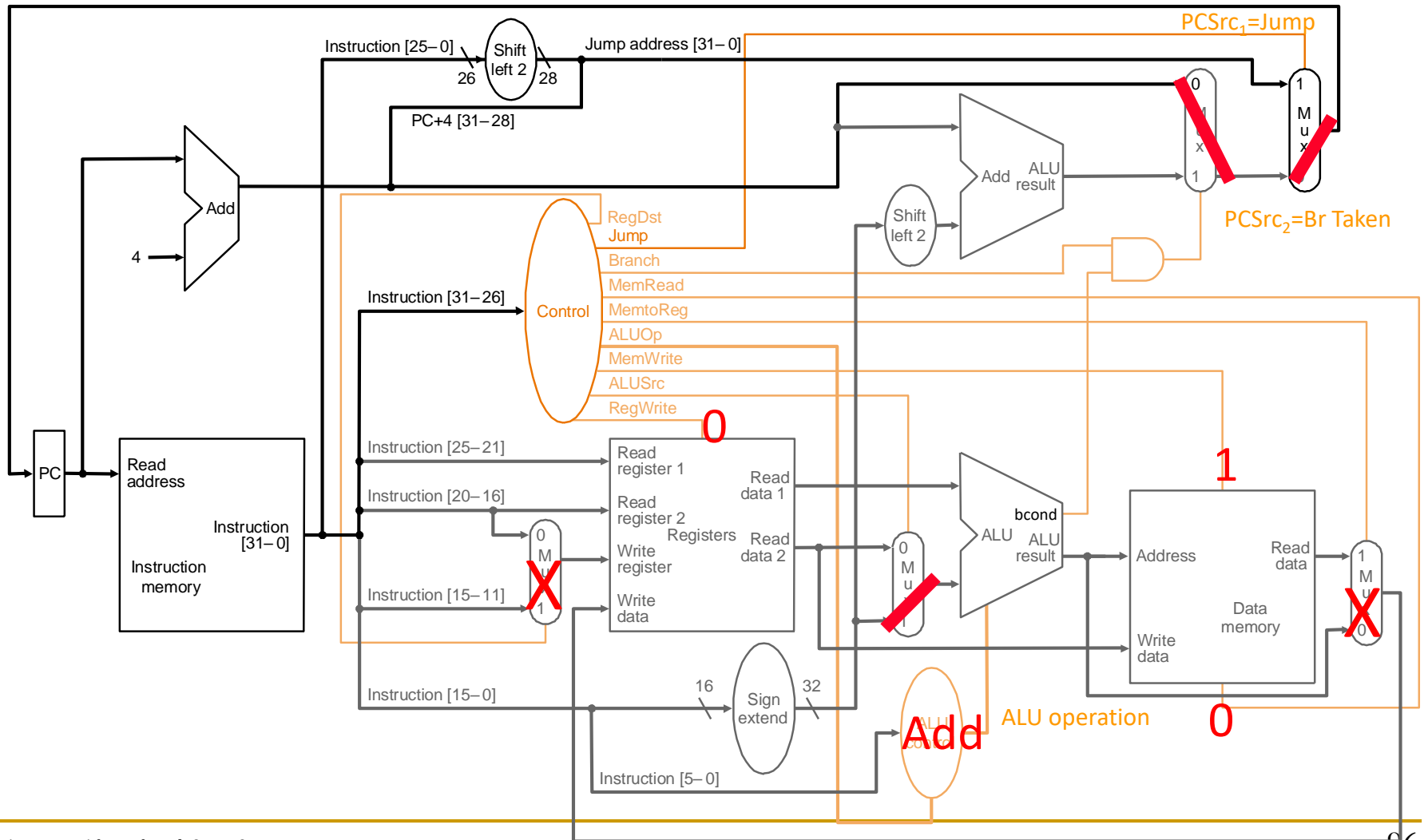


\*\*Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# I-Type ALU



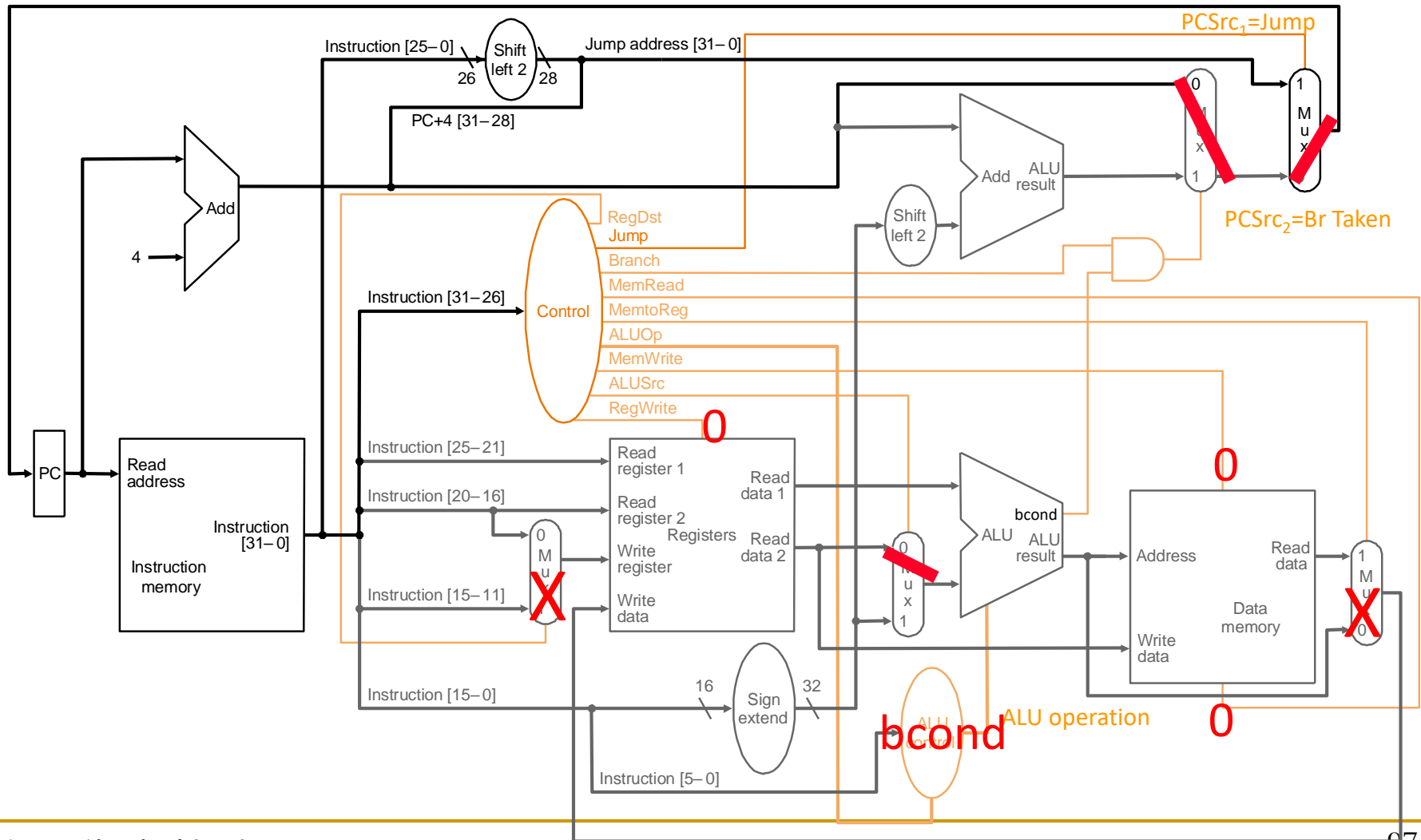






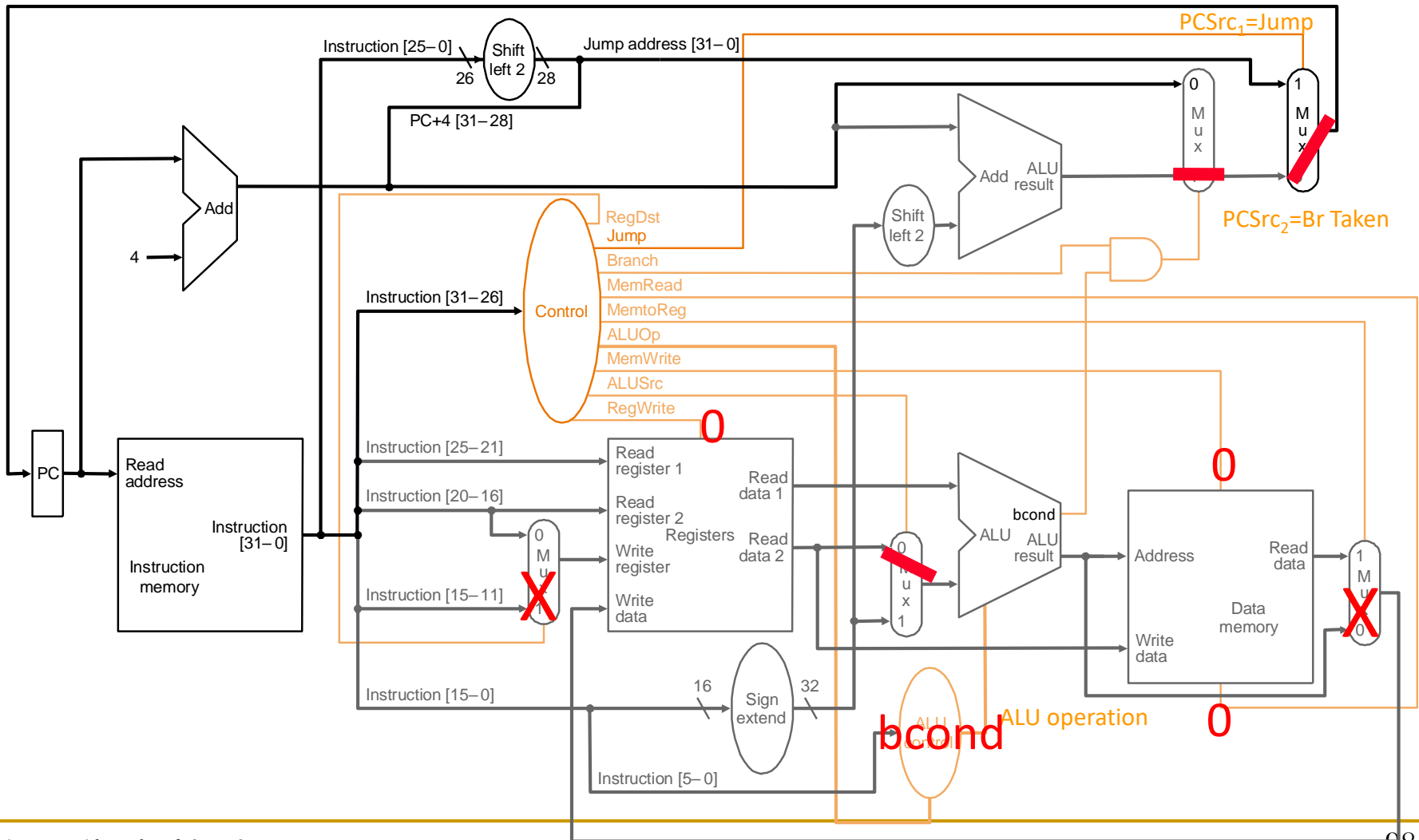
# Branch (Not Taken)

Some control signals are dependent on the processing of data

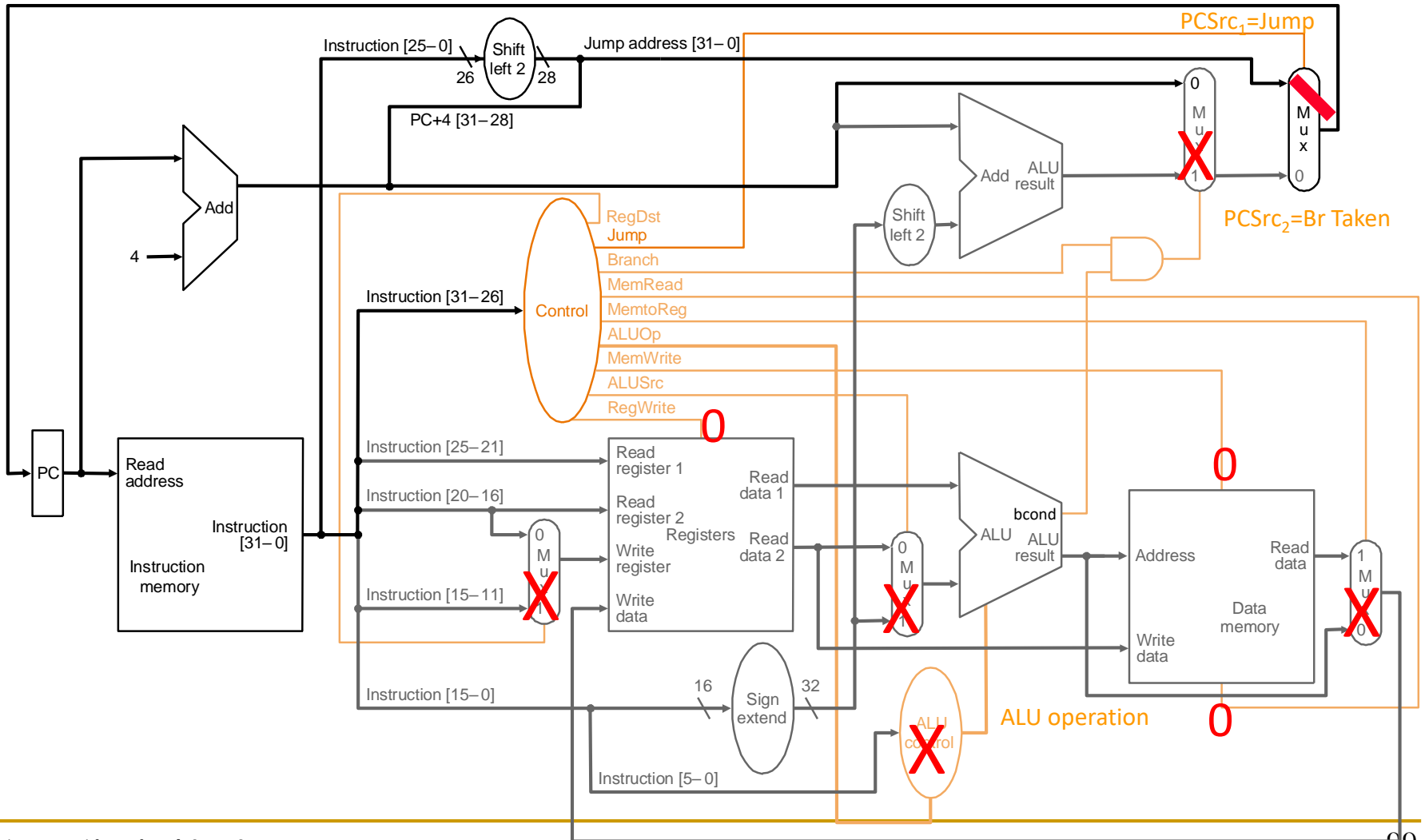


# Branch (Taken)

Some control signals are dependent on the processing of data



# Jump



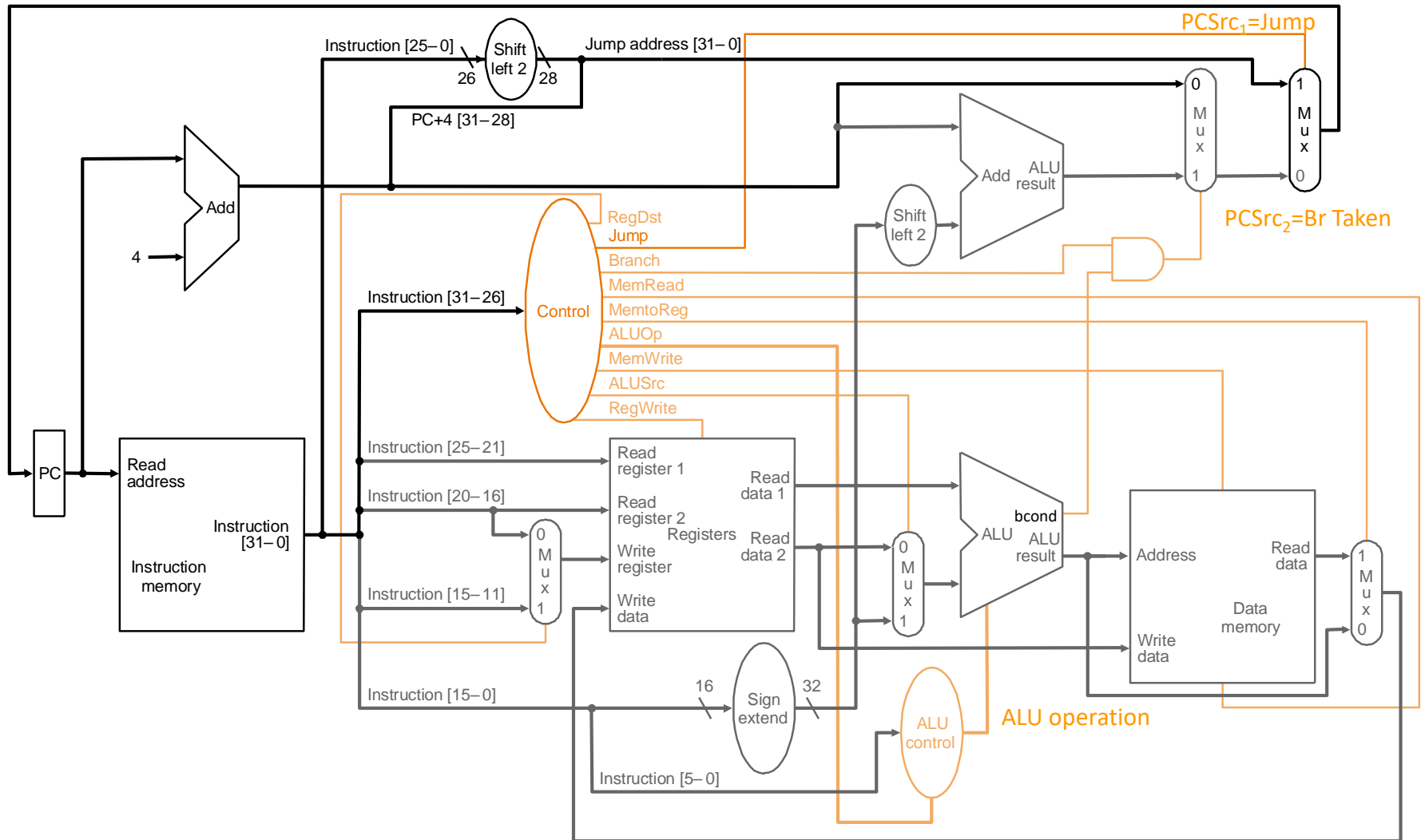
\*\*Based on original figure from [P&H CO&D, COPYRIGHT 2004 Elsevier. ALL RIGHTS RESERVED.]

# What is in That Control Box?

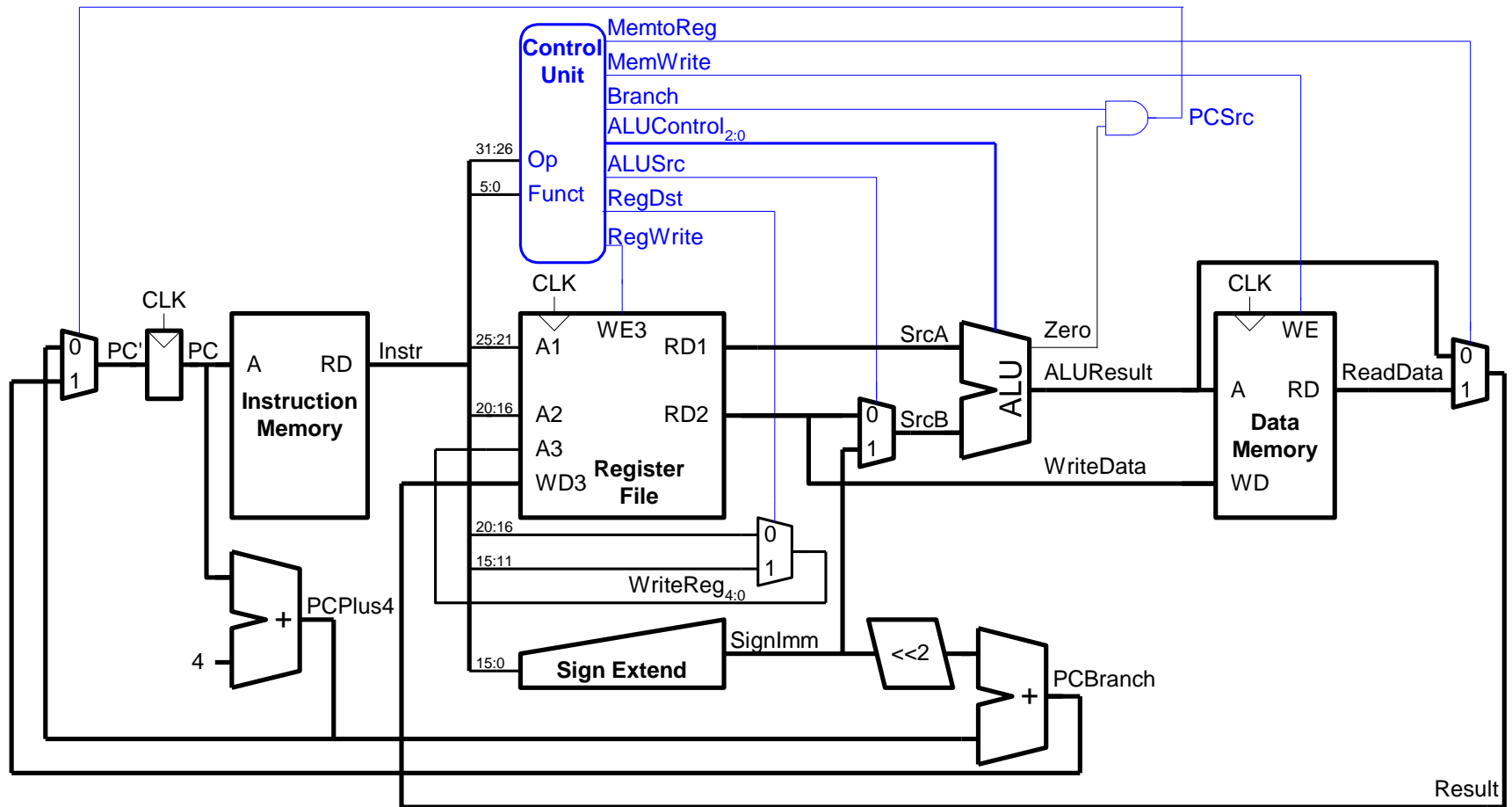
---

- Combinational Logic → **Hardwired Control**
  - ❑ Idea: Control signals generated combinatorially 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

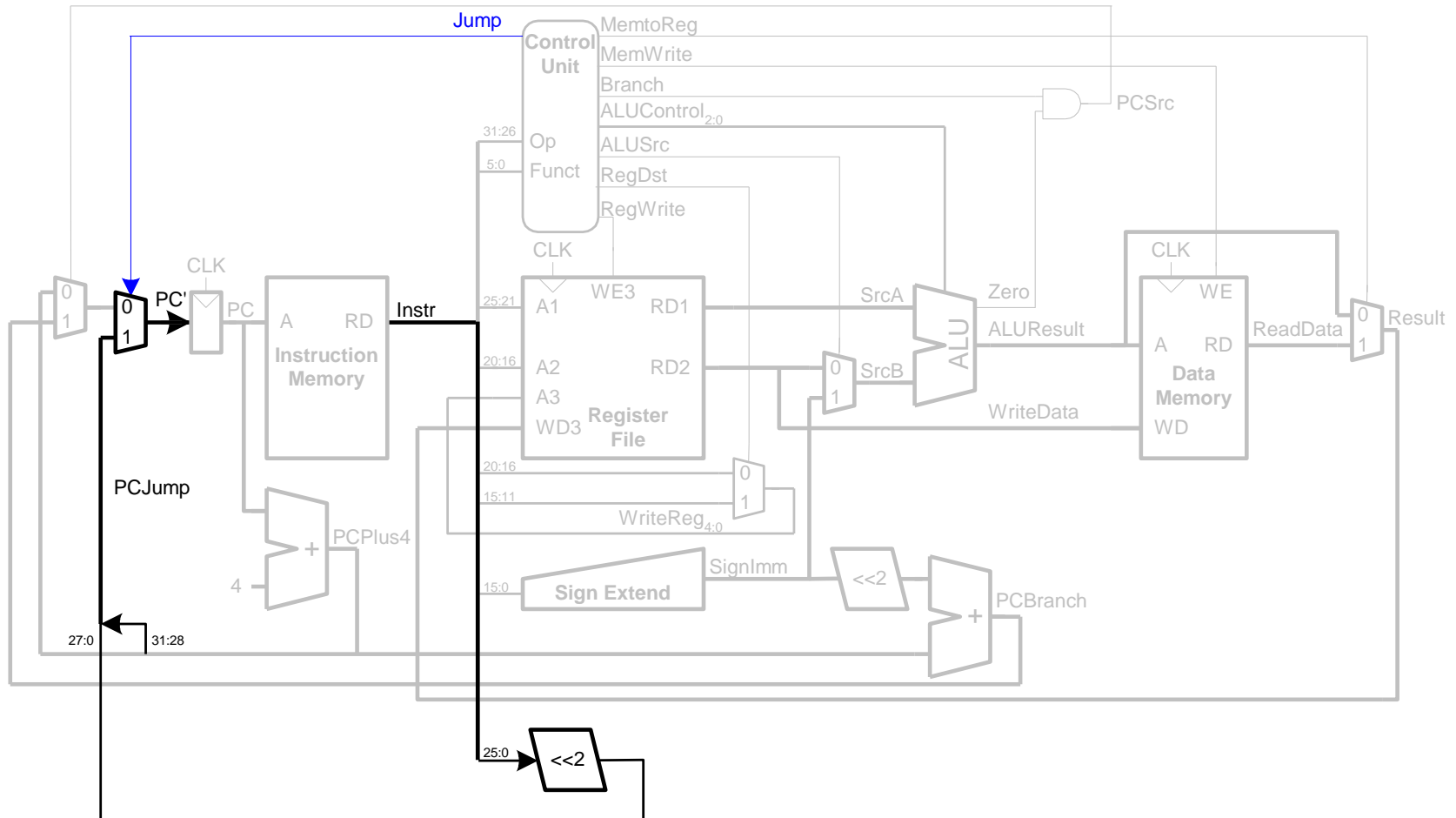
# Review: Complete Single-Cycle Processor



# Another Complete Single-Cycle Processor



# Extended Functionality: j



# Control Unit

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

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>	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 Single-Cycle MIPS Processor (from H&H)

# 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;

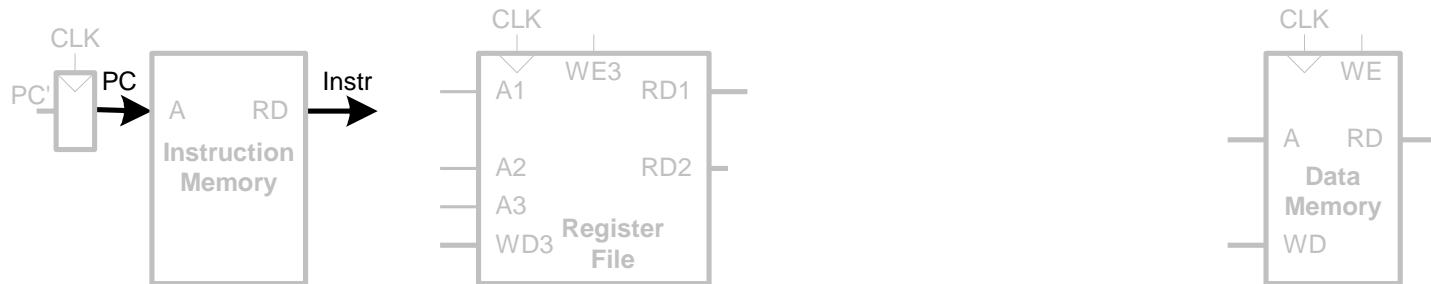
    reg [65535:0] M_arr [31:0];                // Array for Memory

    // 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



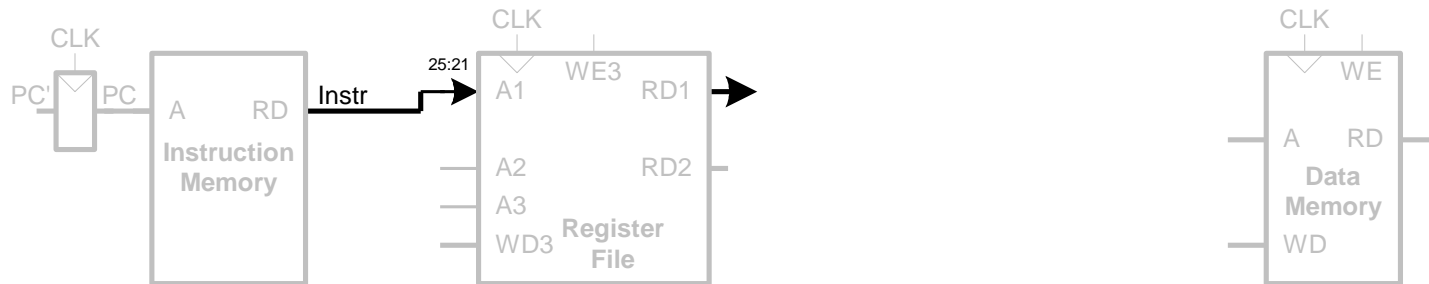
`lw $s3, 1($0) # read memory word 1 into $s3`

**I-Type**



# Single-Cycle Datapath: lw register read

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



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

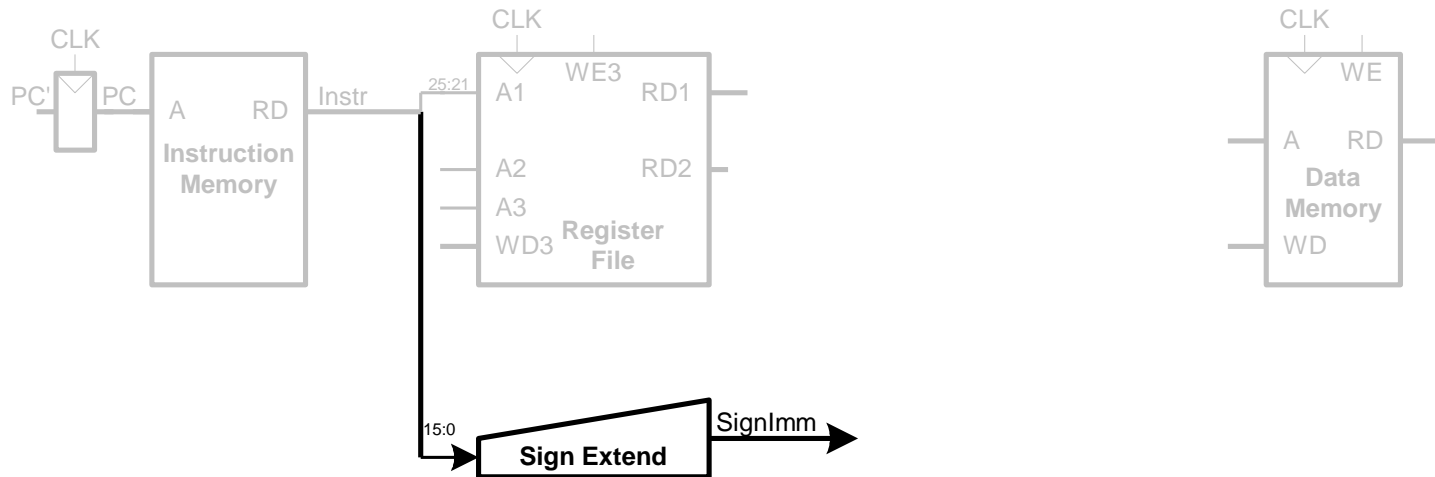
**I-Type**

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



# Single-Cycle Datapath: lw immediate

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



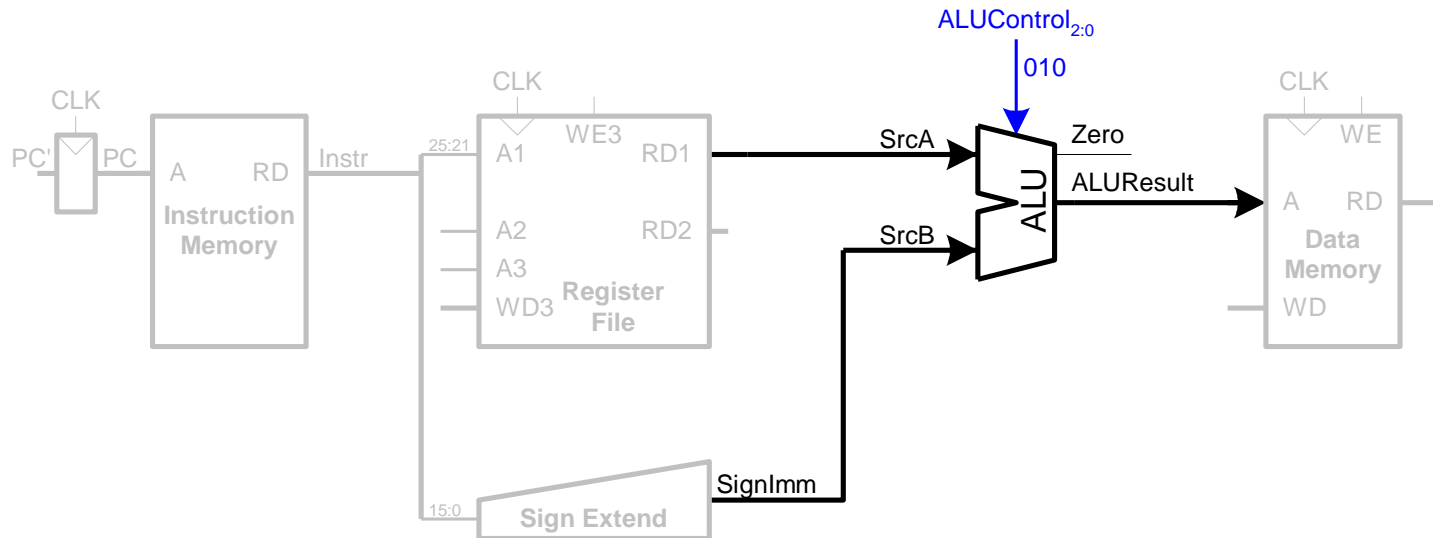
`lw $s3, 1($0) # read memory word 1 into $s3`

**I-Type**

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

# Single-Cycle Datapath: lw address

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



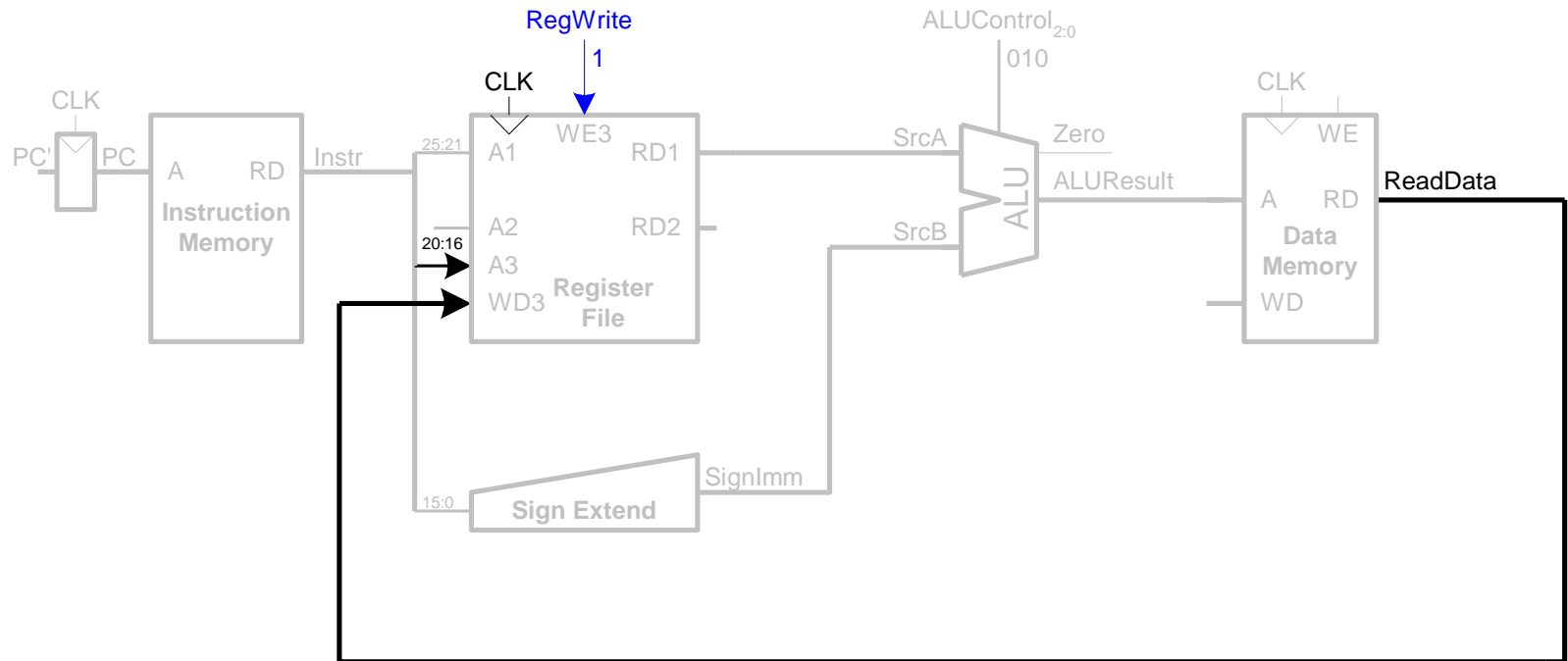
`lw $s3, 1($0) # read memory word 1 into $s3`

**I-Type**

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

# Single-Cycle Datapath: lw memory read

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



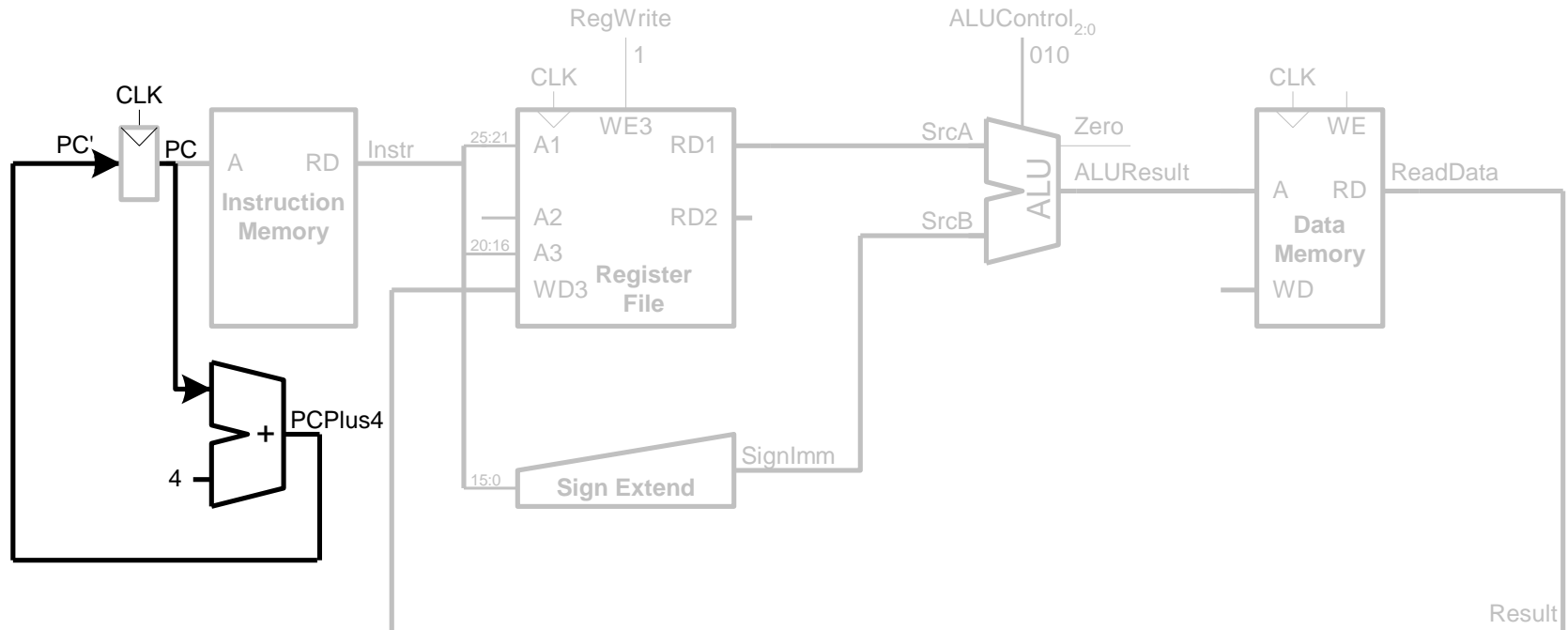
`lw $s3, 1($0) # read memory word 1 into $s3`

**I-Type**

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

# Single-Cycle Datapath: lw PC increment

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



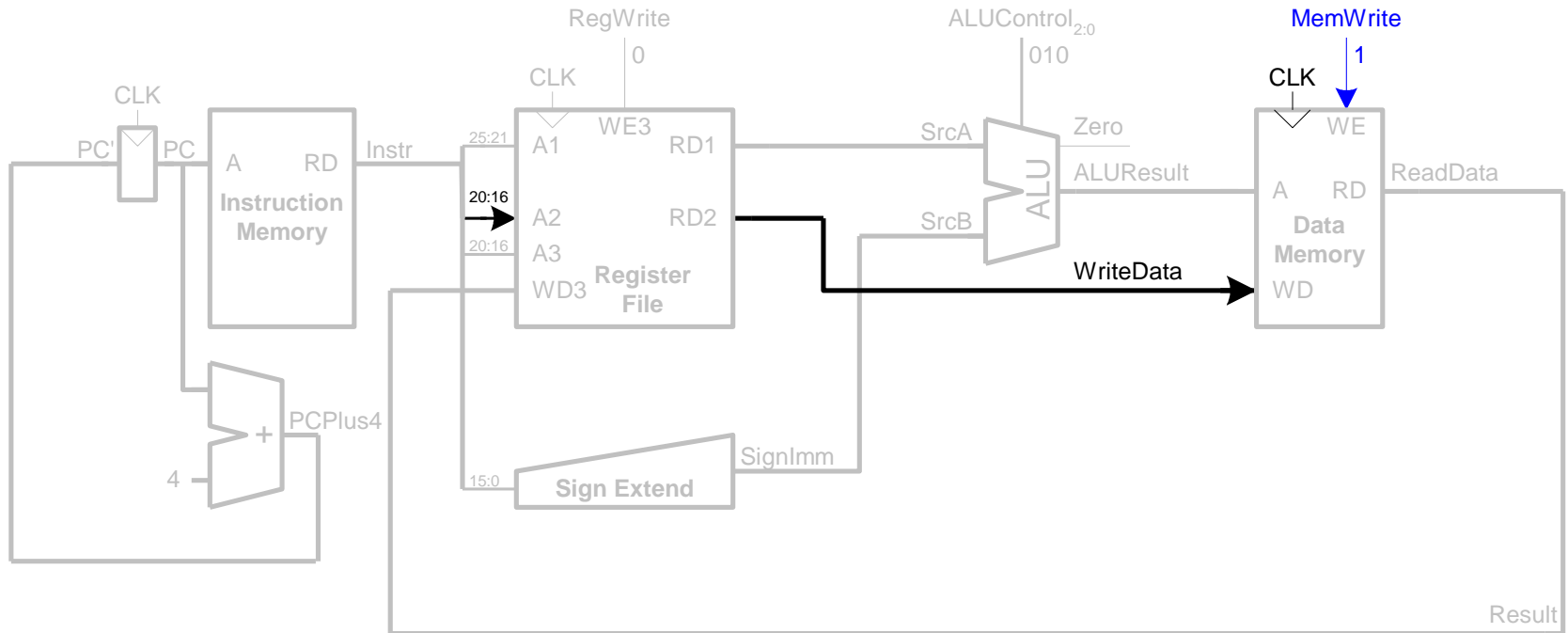
`lw $s3, 1($0) # read memory word 1 into $s3`

**I-Type**

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

# Single-Cycle Datapath: sw

## ■ Write data in rt to memory



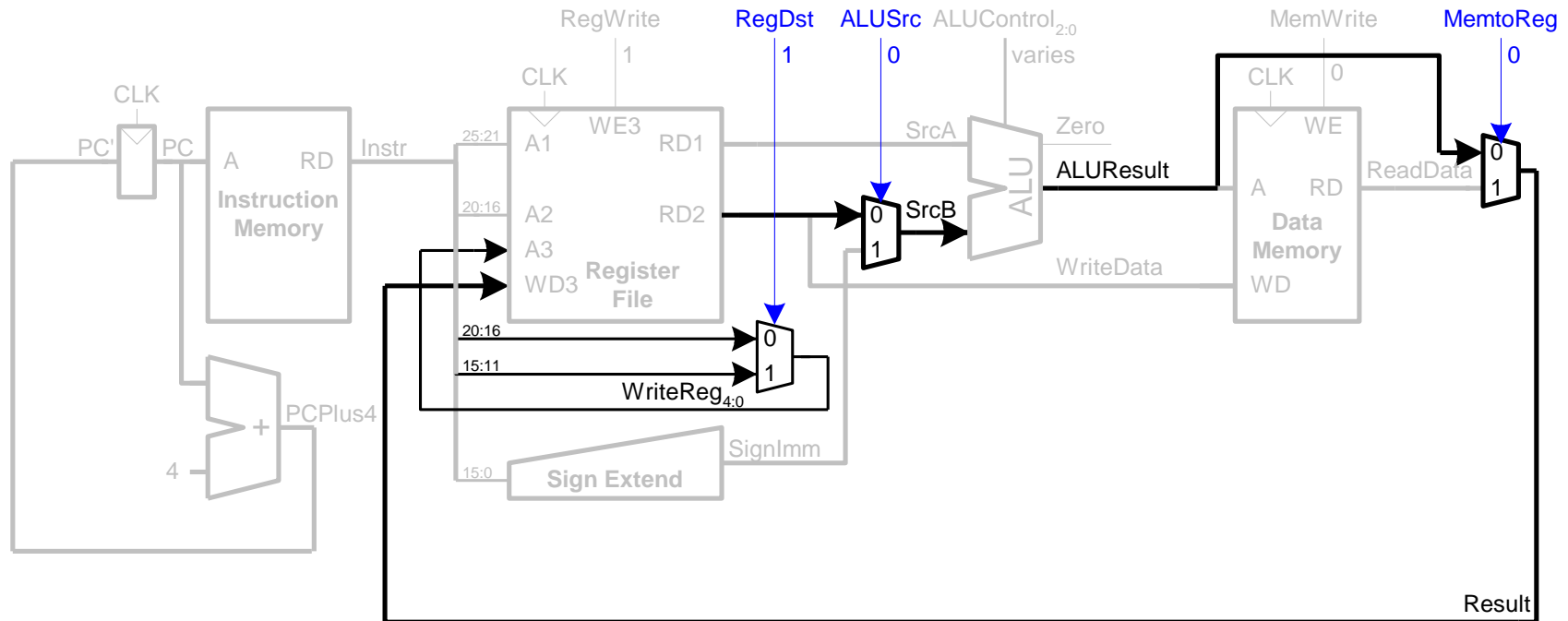
`sw $t7, 44($0) # write t7 into memory address 44`

**I-Type**

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

# Single-Cycle Datapath: R-type Instructions

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

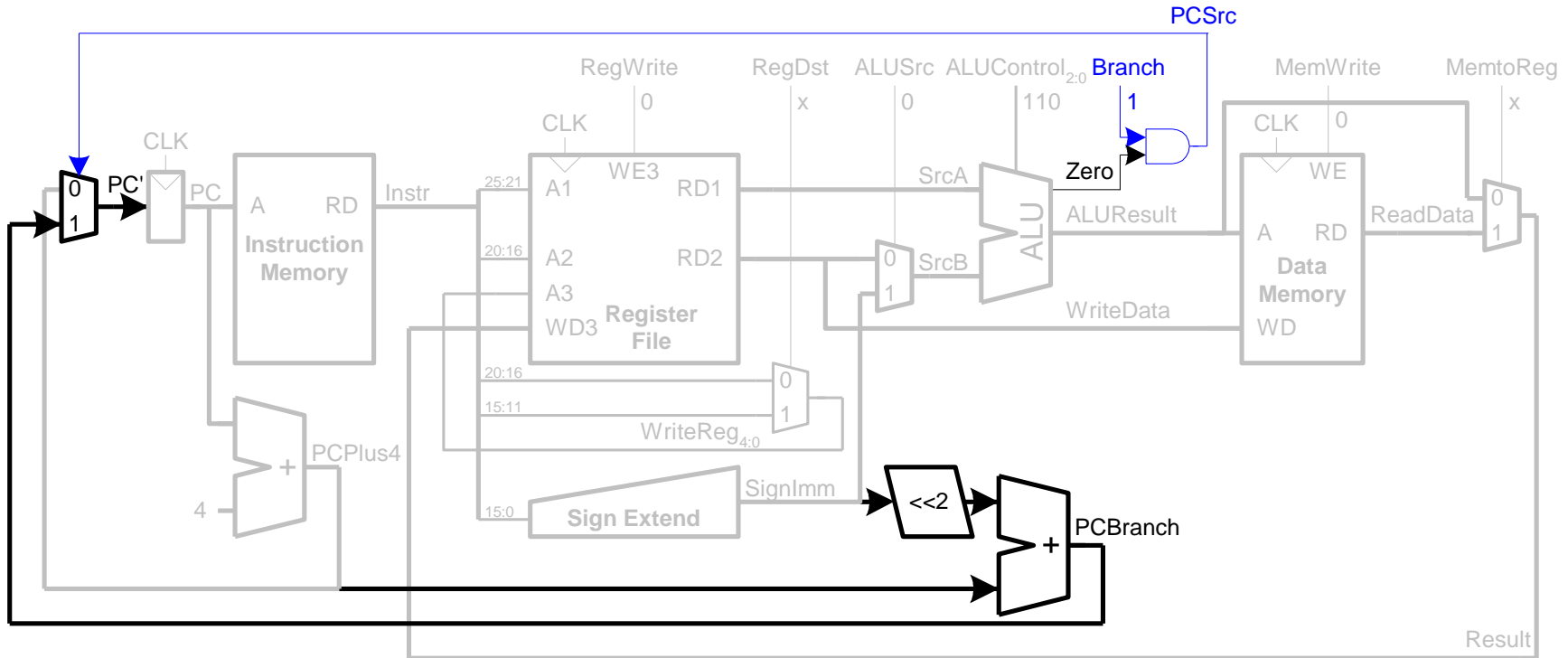


add t, b, c # t = b + c

**R-Type**

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

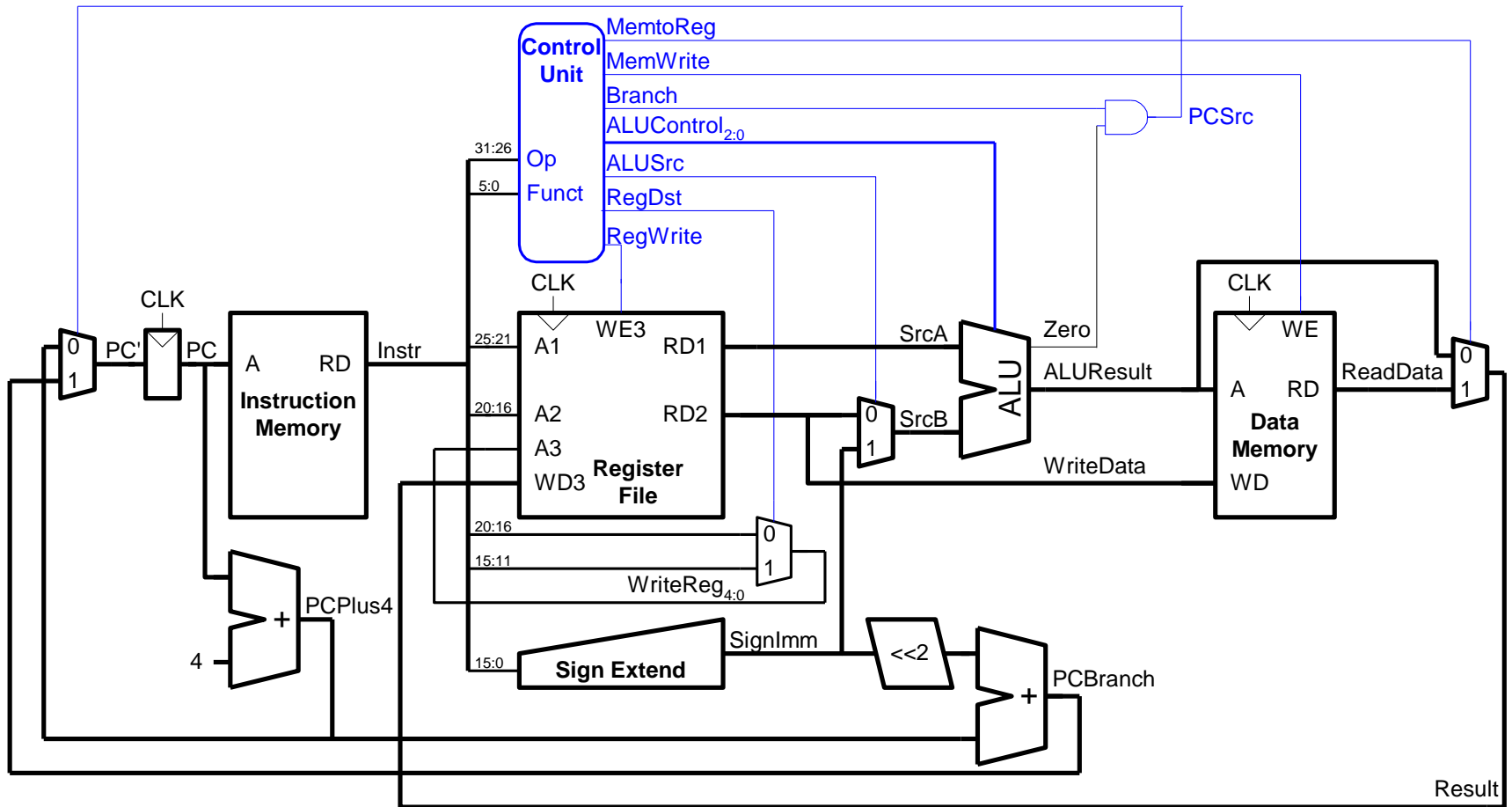
# Single-Cycle Datapath: beq



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

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

# Complete Single-Cycle Processor





# Our MIPS Datapath has Several Options

- **ALU inputs**

- Either RT or Immediate (*MUX*)

- **Write Address of Register File**

- Either RD or RT (*MUX*)

- **Write Data In of Register File**

- Either ALU out or Data Memory Out (*MUX*)

- **Write enable of Register File**

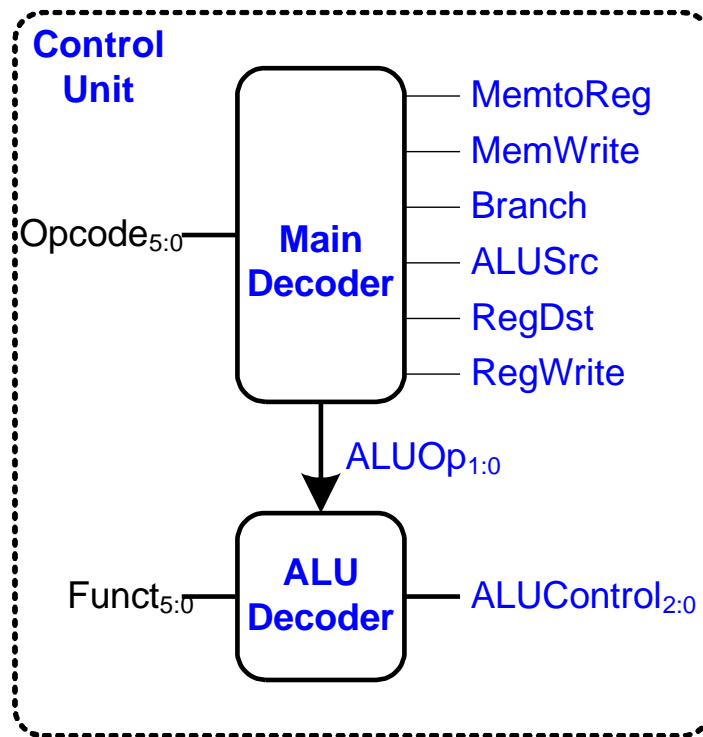
- Not always a register write (*MUX*)

- **Write enable of Memory**

- Only when writing to memory (sw) (*MUX*)

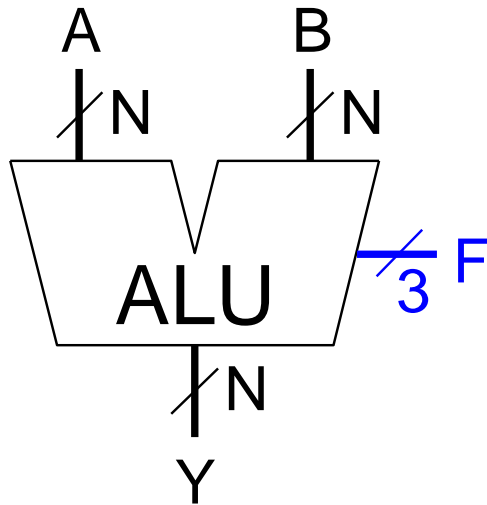
*All these options are our control signals*

# Control Unit



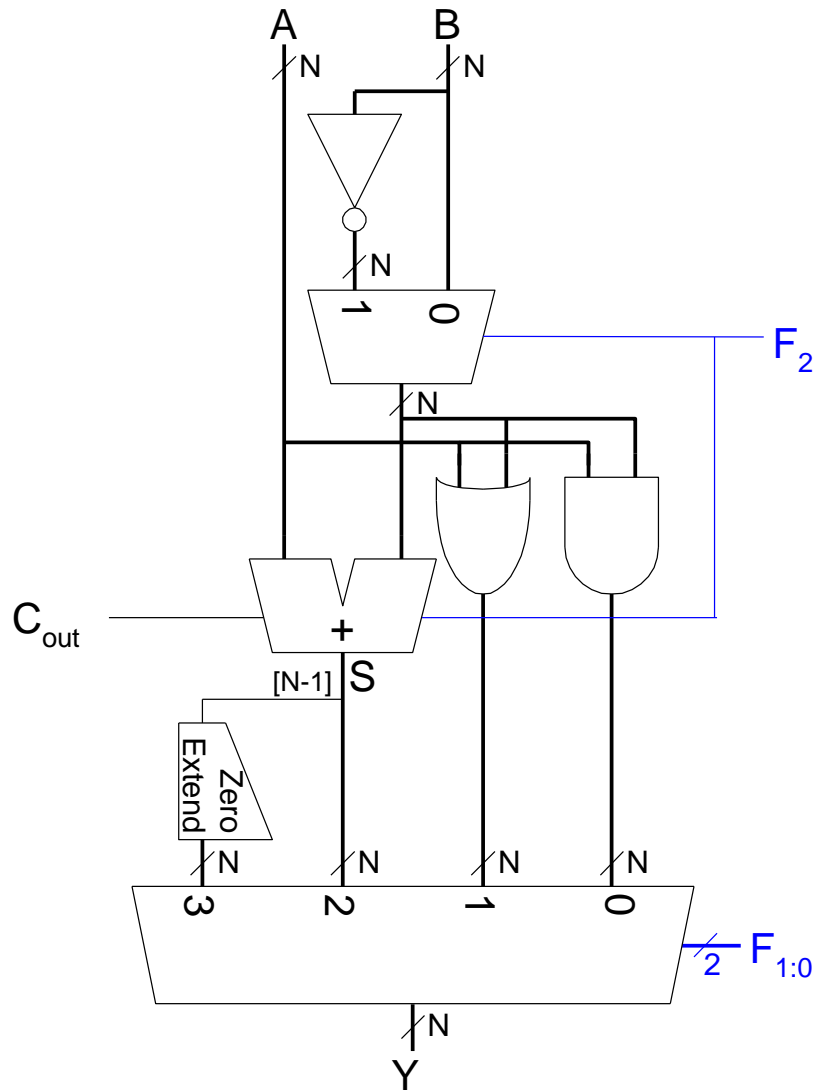
ALUOp	Meaning
00	add
01	subtract
10	look at funct field
11	n/a

# ALU Does the Real Work in a Processor



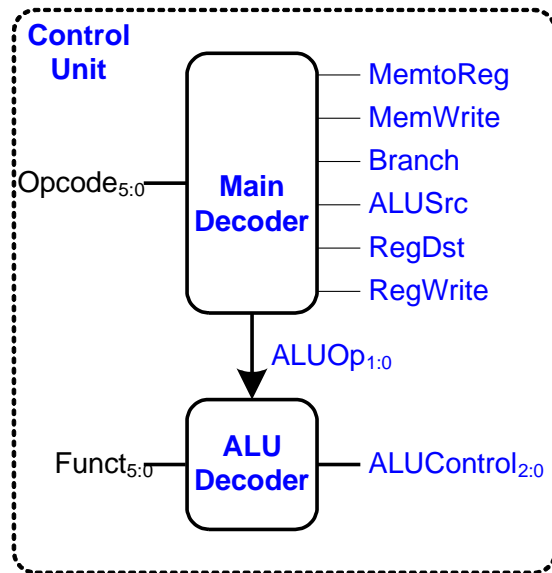
F <sub>2:0</sub>	Function
000	A & B
001	A   B
010	A + B
011	not used
100	A & ~B
101	A   ~B
110	A - B
111	SLT

# ALU Internals



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

# Control Unit: ALU Decoder



ALUOp <sub>1:0</sub>	Meaning
00	Add
01	Subtract
10	Look at Funct
11	Not Used

ALUOp <sub>1:0</sub>	Funct	ALUControl <sub>2:0</sub>
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 <sub>5:0</sub>	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 <sub>5:0</sub>	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 <sub>5:0</sub>	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 <sub>5:0</sub>	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

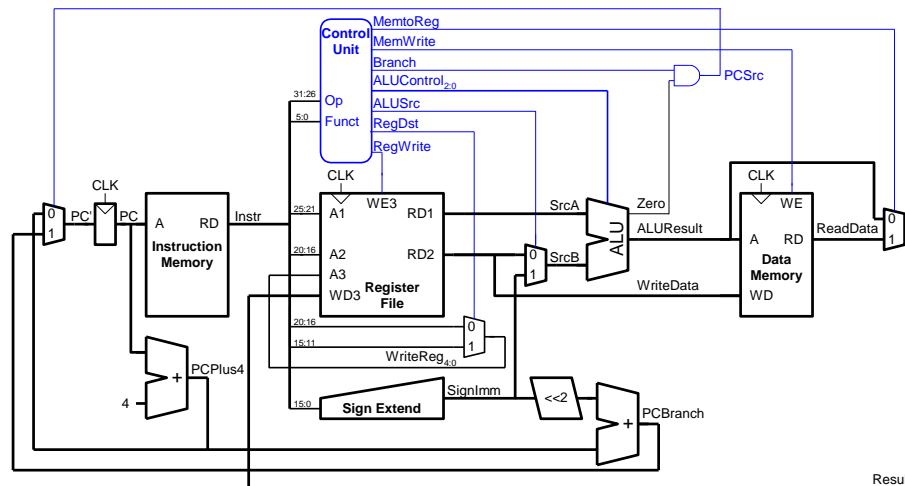
Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	funct
lw	100011	1	0	1	0	0	1	add
sw	101011	0	X	1	0	1	X	add
beq	000100	0	X	0	1	0	X	sub

## ■ New Control Signal

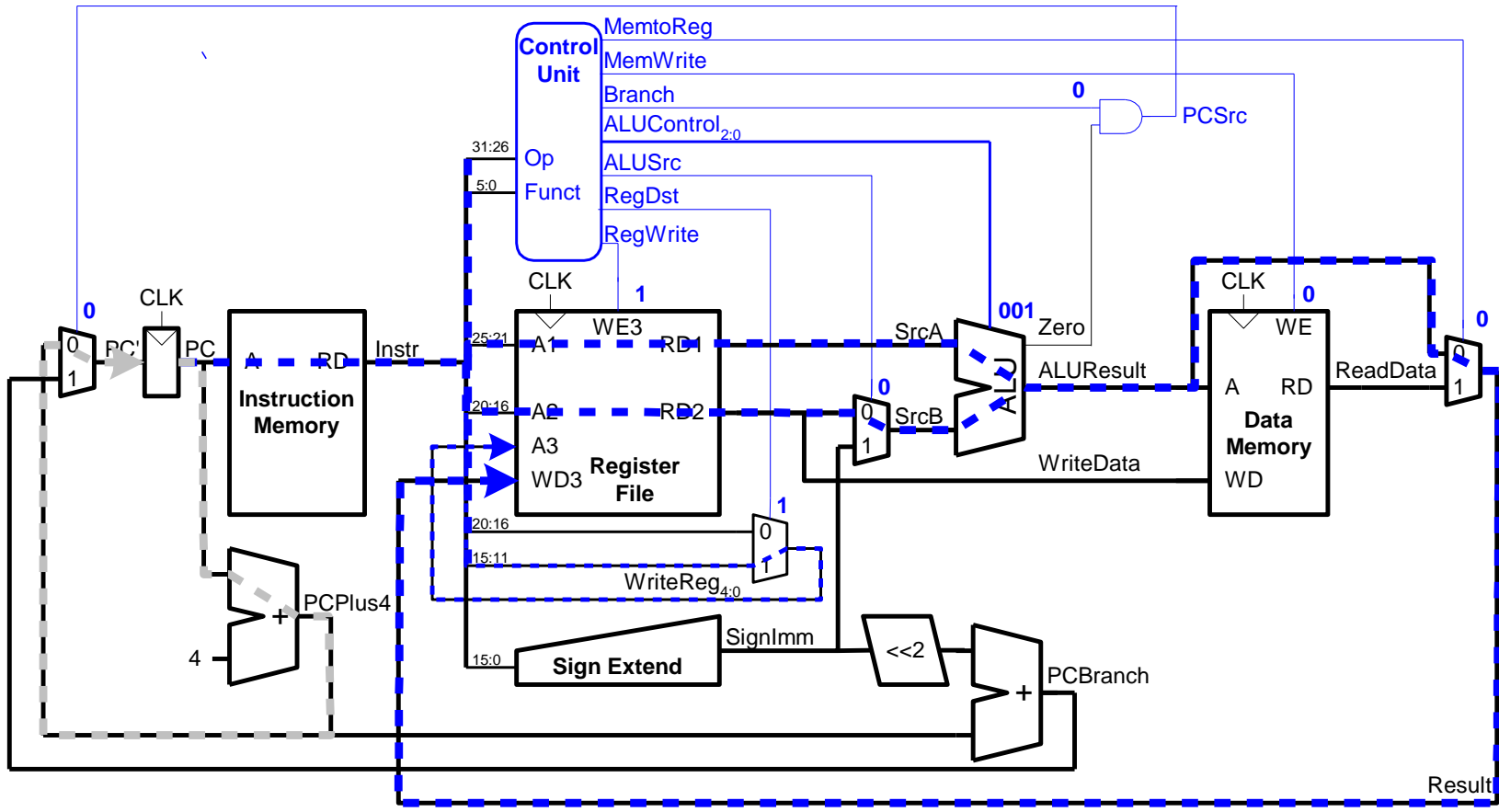
- **Branch:** Are we jumping or not ?

# Control Unit: Main Decoder

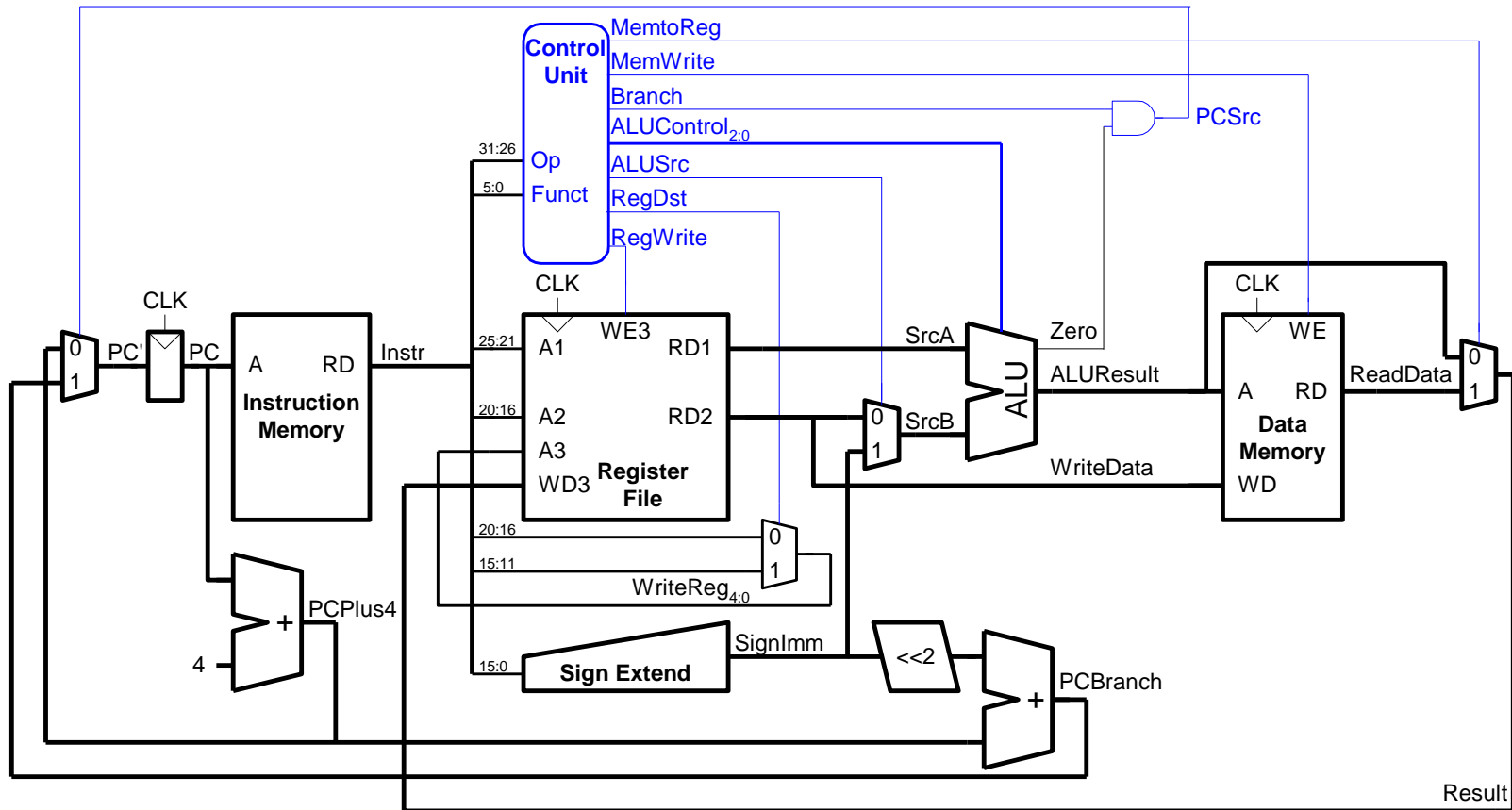
Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



# Single-Cycle Datapath Example: or



# Extended Functionality: addi

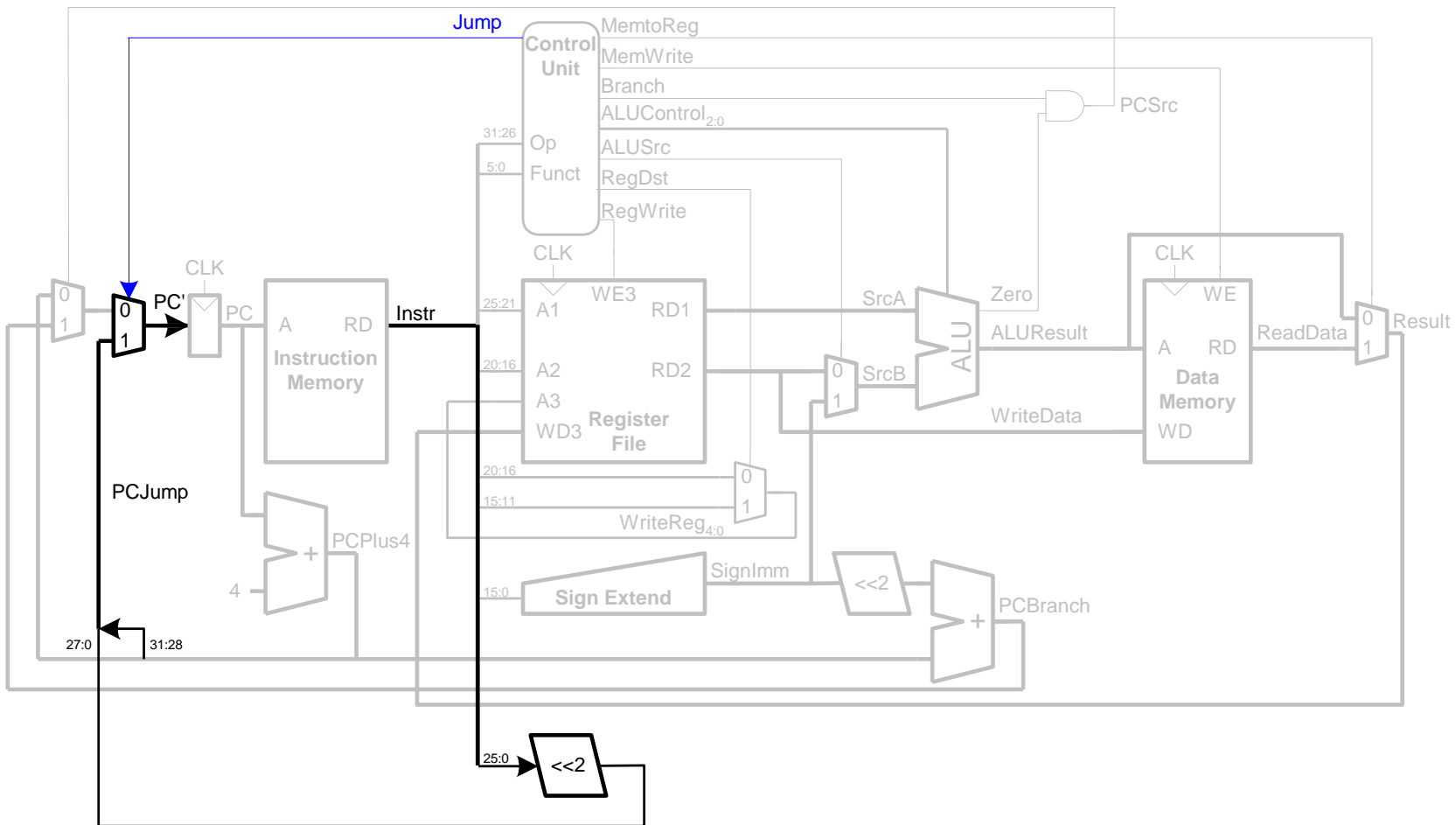


■ No change to datapath

# Control Unit: addi

Instruction	Op <sub>5:0</sub>	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp <sub>1:0</sub>
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
<b>addi</b>	<b>001000</b>	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>0</b>	<b>00</b>

# Extended Functionality: j

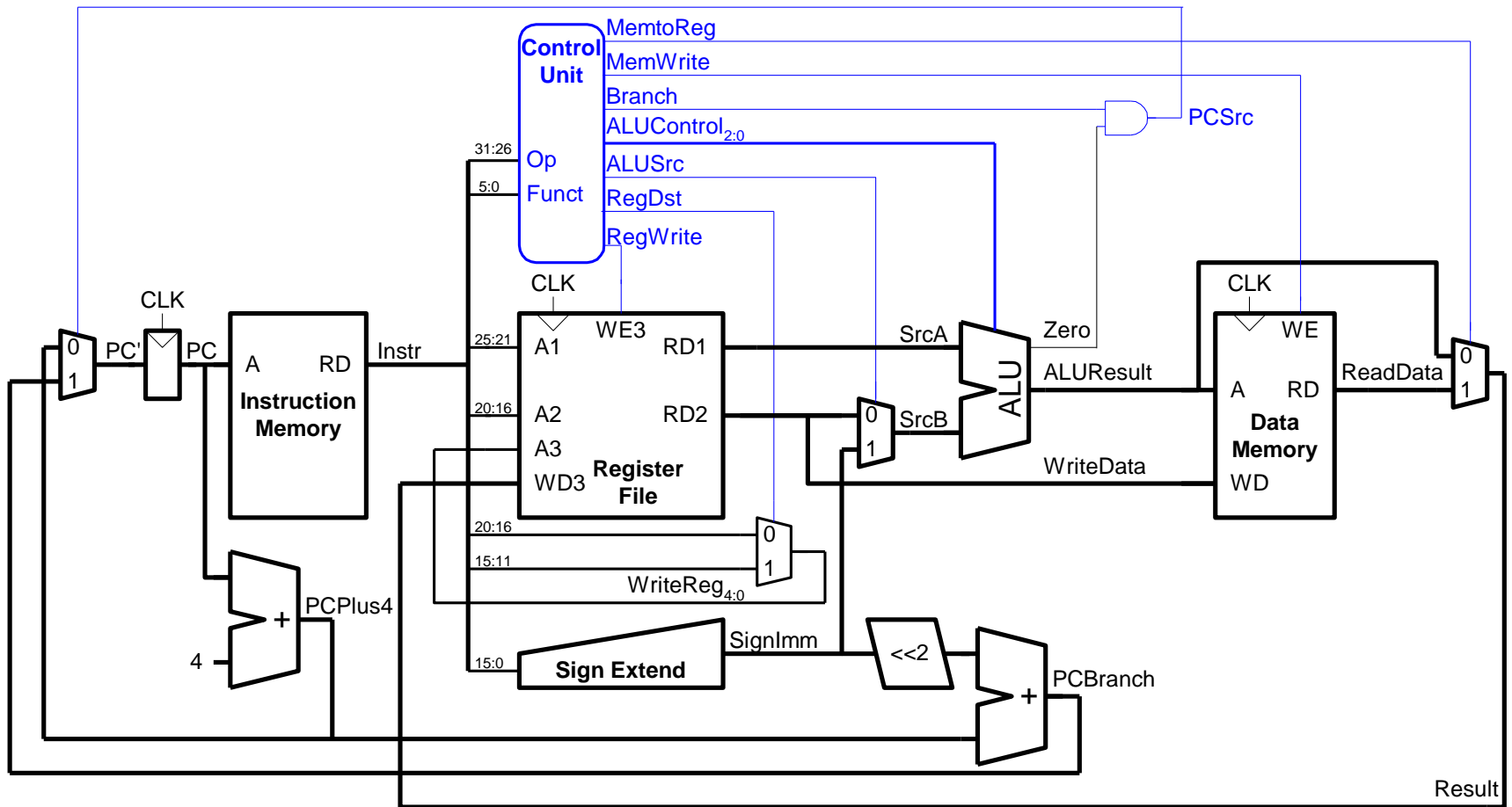


# Control Unit: Main Decoder

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



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



# A Bit More on Performance Analysis

# Processor Performance

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

# Processor Performance

## ■ How fast is my program?

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

## ■ So how fast are my instructions ?

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

# Processor Performance

## ■ How fast is my program?

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

## ■ So how fast are my instructions ?

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

## ■ How much time is one clock cycle?

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

# Performance Analysis

---

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

# Processor Performance

## ■ Now as a general formula

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

# Processor Performance

## ■ Now as a general formula

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

## ■ Our program will execute in

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



# How can I Make the Program Run Faster?

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

# How can I Make the Program Run Faster?

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

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

# How can I Make the Program Run Faster?

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

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

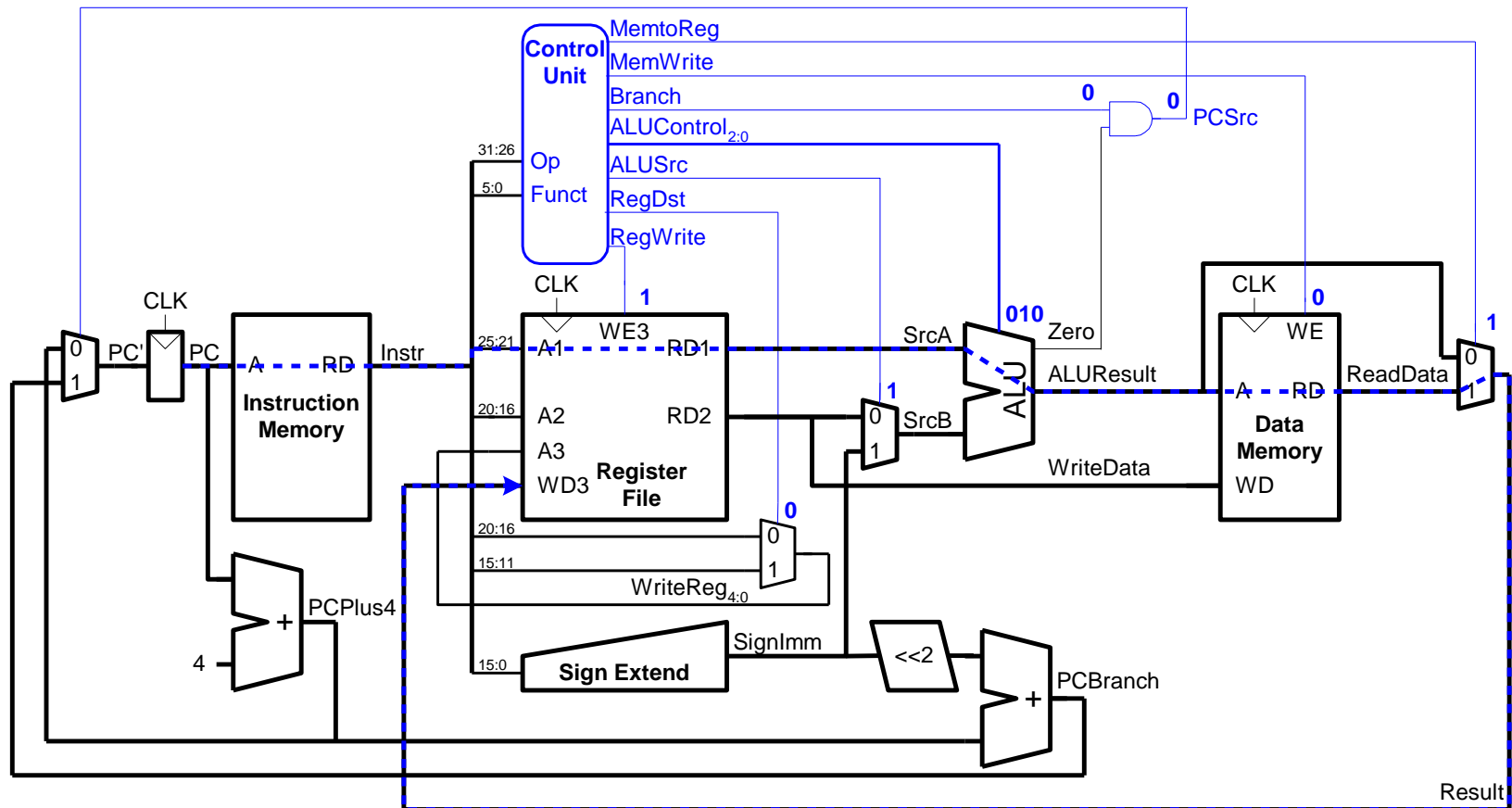
# How can I Make the Program Run Faster?

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

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

# Single-Cycle Performance

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



# Single-Cycle Performance

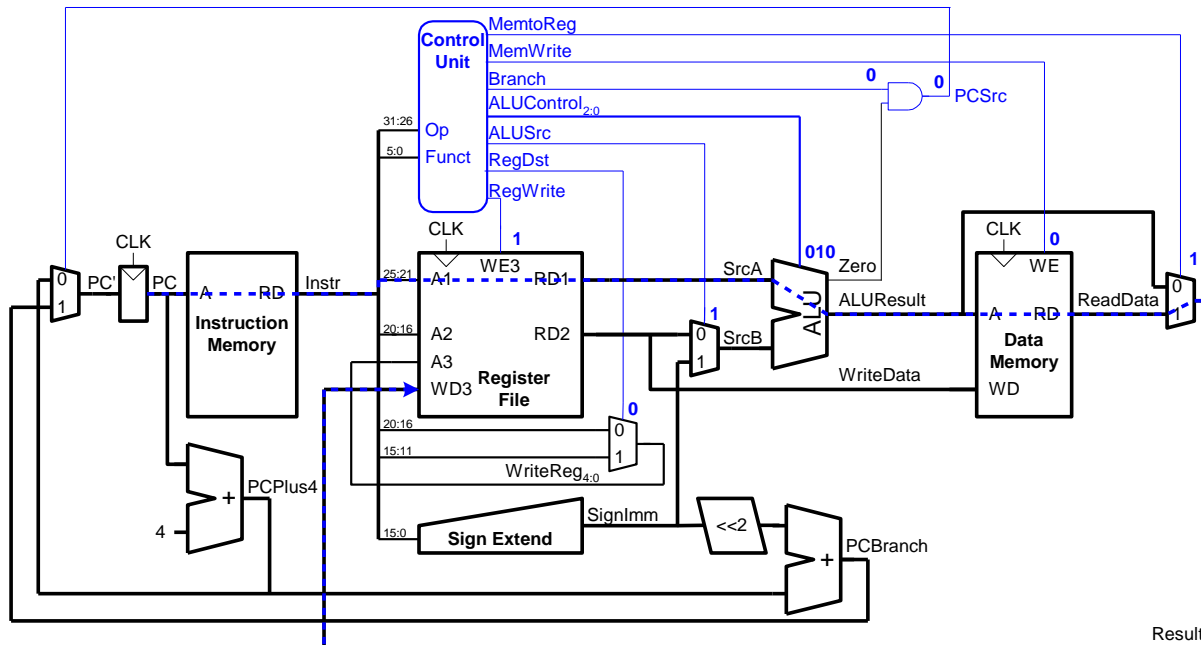
## ■ Single-cycle critical path:

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

## ■ In most implementations, limiting paths are:

- memory, ALU, register file.

$$T_c = t_{pcq\_PC} + 2t_{mem} + t_{Rfread} + t_{mux} + t_{ALU} + t_{RFsetup}$$



# Single-Cycle Performance Example

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

$$T_c =$$

# Single-Cycle Performance Example

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

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



# Single-Cycle Performance Example

## ■ Example:

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

# Single-Cycle Performance Example

## ■ Example:

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

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

# Design of Digital Circuits

## Lecture 11: Microarchitecture

Prof. Onur Mutlu

ETH Zurich

Spring 2018

29 March 2018