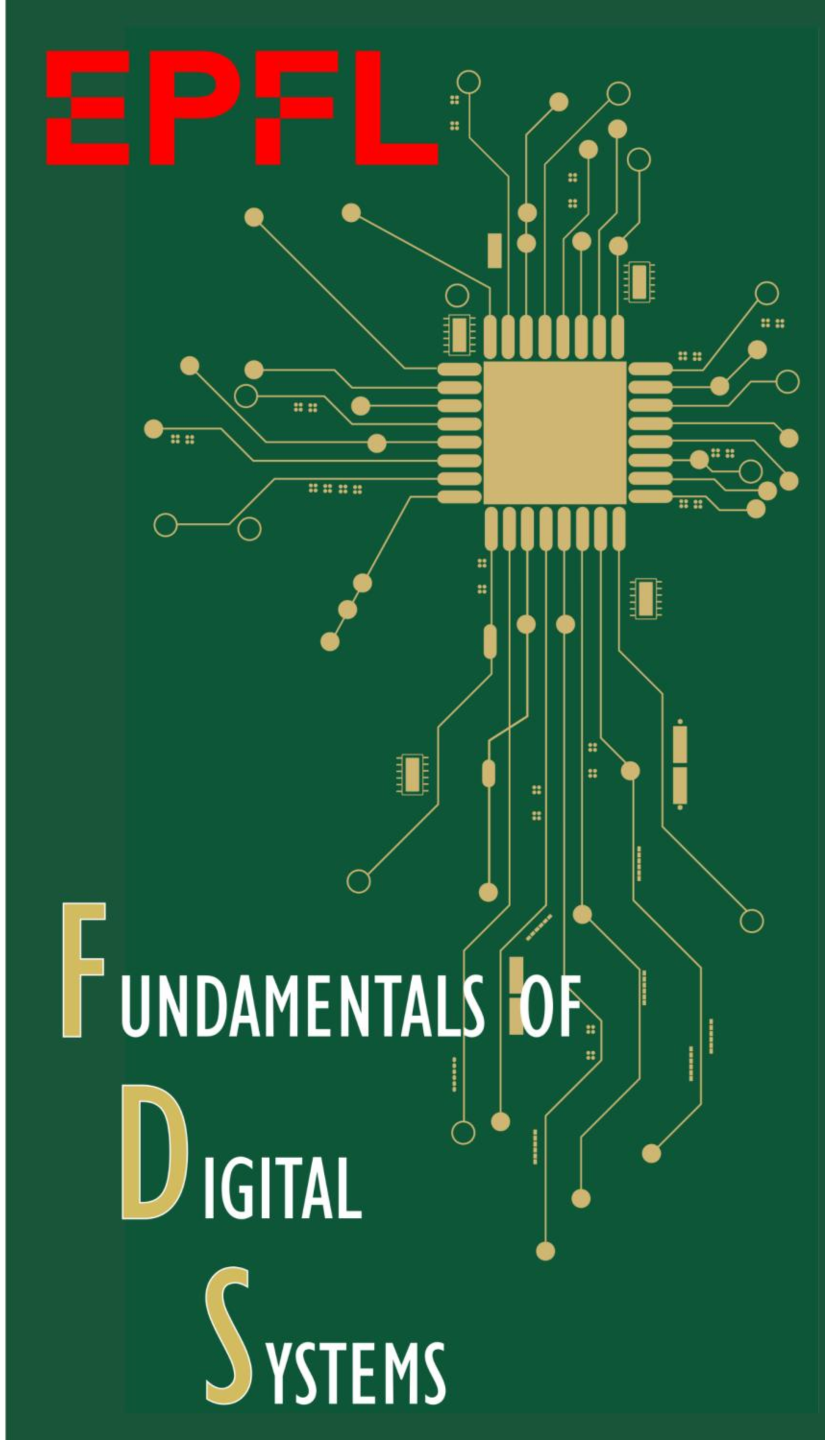


Computer Architecture

Multicycle CPU FSM

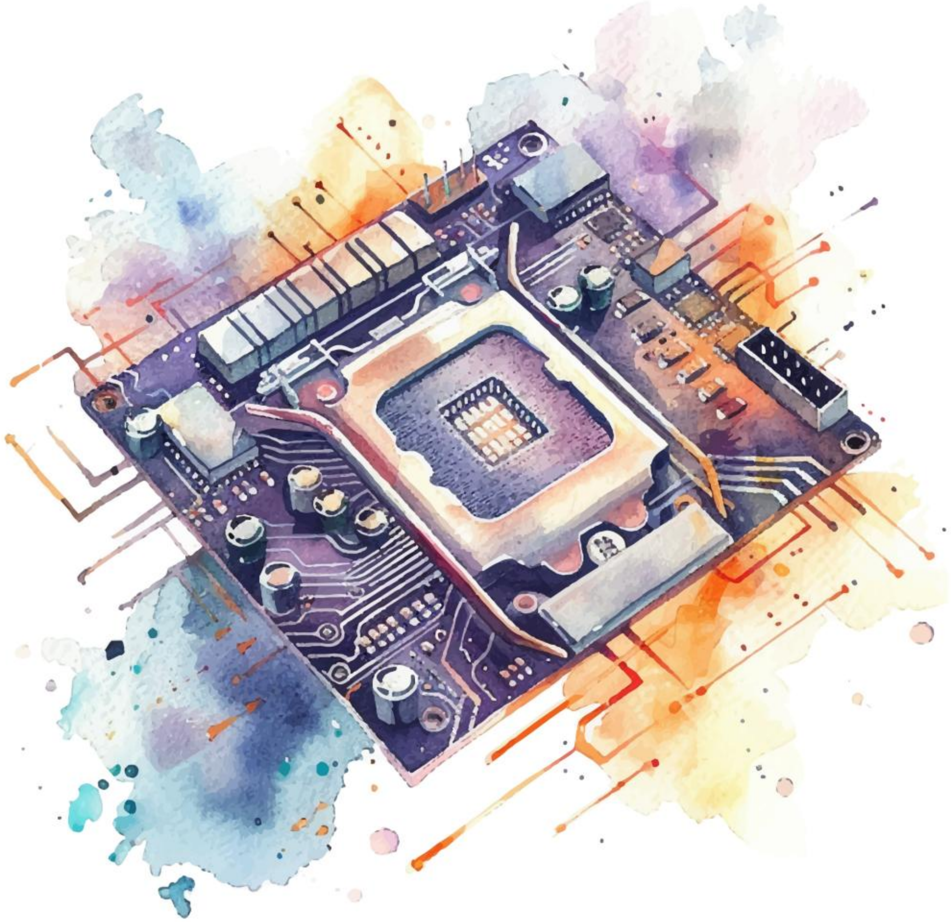
CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025

Previously on FDS

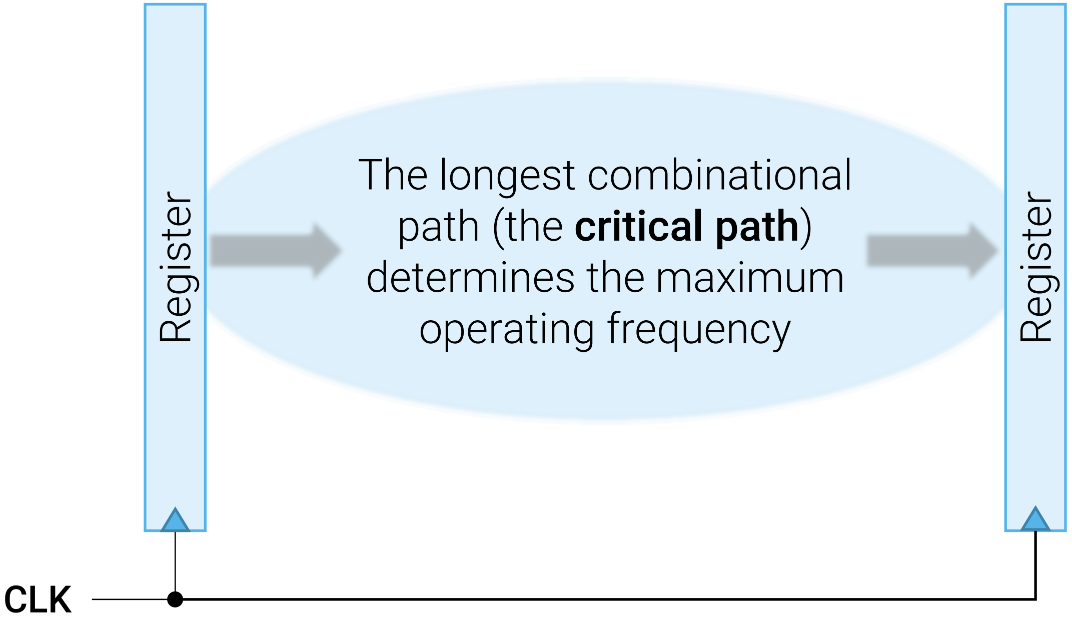
- Processor Implementations
  - Single-cycle vs. multicycle CPU
  - Multicycle CPU



© EnelEva / Adobe Stock

Recall: Single-Cycle vs. Multicycle CPU

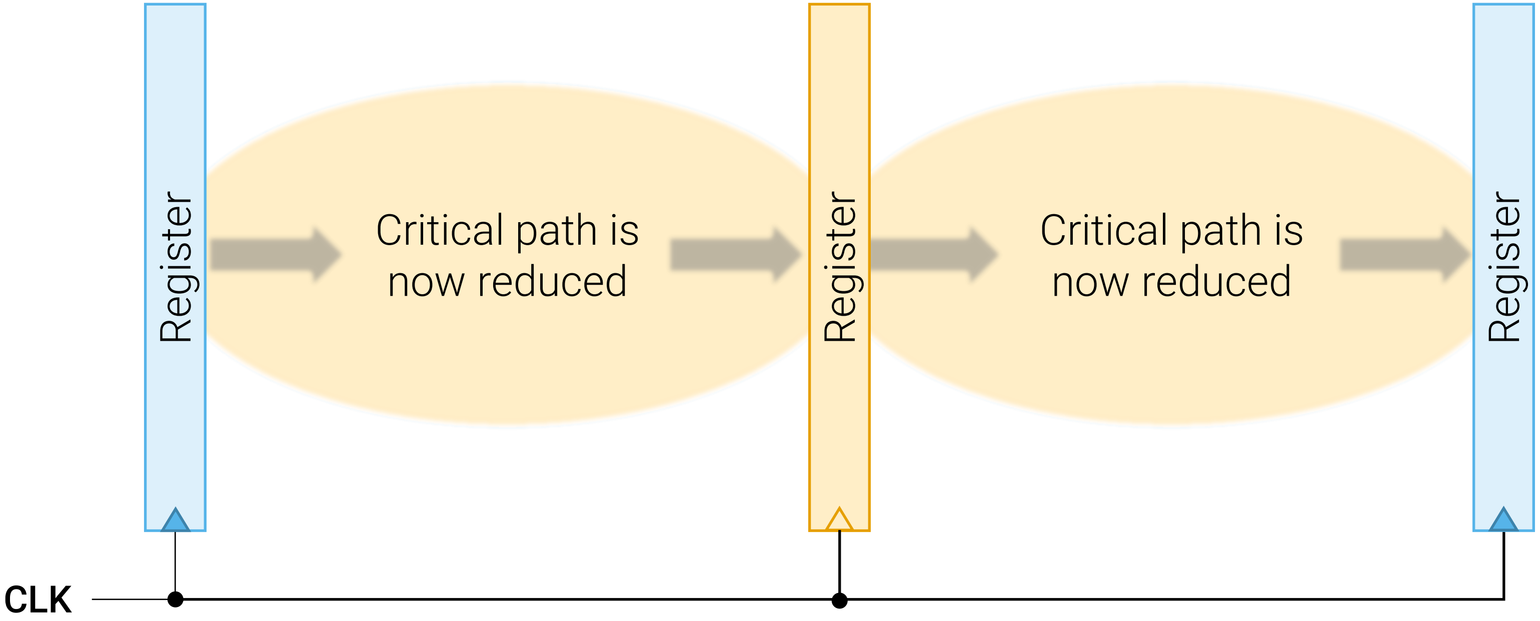
- If all instruction steps are performed in a single clock cycle, we have a **single-cycle** CPU implementation


Register
The longest combinational path (the **critical path**) determines the maximum operating frequency
Register
CLK

Note: These two registers could be the same one (e.g., PC)

Recall: Single-Cycle vs. Multicycle CPU

- Alternative implementation is a **multicycle** CPU
  - In a multi-cycle implementation, one or more instruction **steps** take one clock cycle, and consequently, some instructions take multiple clock cycles



Note: These two registers could be the same one (e.g., PC)

Note: This is an example; Other multi-cycle implementations are also possible

Recall: Is Single-Cycle CPU More Efficient?

- **No.** Every instruction takes one cycle. f_{max} is limited by the longest of all paths that instructions take (the critical path).
- In a multi-cycle CPU, one or more instruction steps take one cycle.
  f_{max} increases as the critical path is shorter now.
  Instructions that require fewer steps will likely be executed faster, reducing the program's overall execution time.

Recall: Multicycle CPU

vs. Single-Cycle CPU

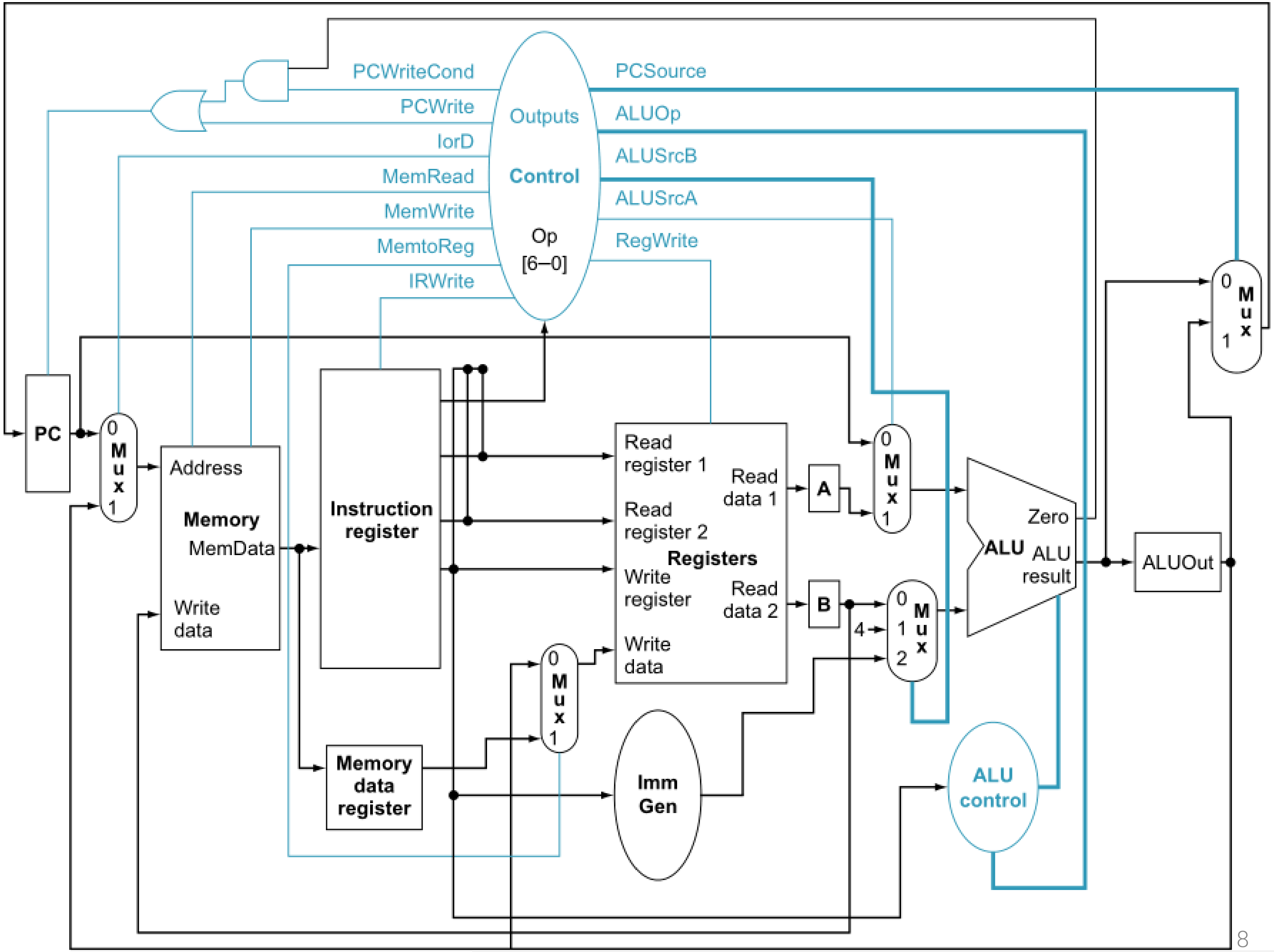
- A **single memory** unit for **both** instructions and data
  - **Why?** Having more than one cycle available (more time to read instructions, read/write data) allows memory sharing
- A **single ALU** instead of an ALU and two adders
  - **Why?** The same ALU can be used in different clock cycles
- **Additional registers** to hold the outputs of the functional units until the value is used (consumed) in a subsequent clock cycle
  - **Why?** Ensure that the value to be used is “stable” for the entire cycle

Recall: A Simple Multicycle CPU

- *Recall:* Let us build a simple CPU supporting the following **subset** of RISC-V instructions for simplicity
  - **R-type arithmetic-logical instructions**
    - add, sub, and, or
  - **Memory instructions**
    - load and store word
  - **Control flow**
    - branch if equal

Recall:

A Simple Multicycle CPU



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

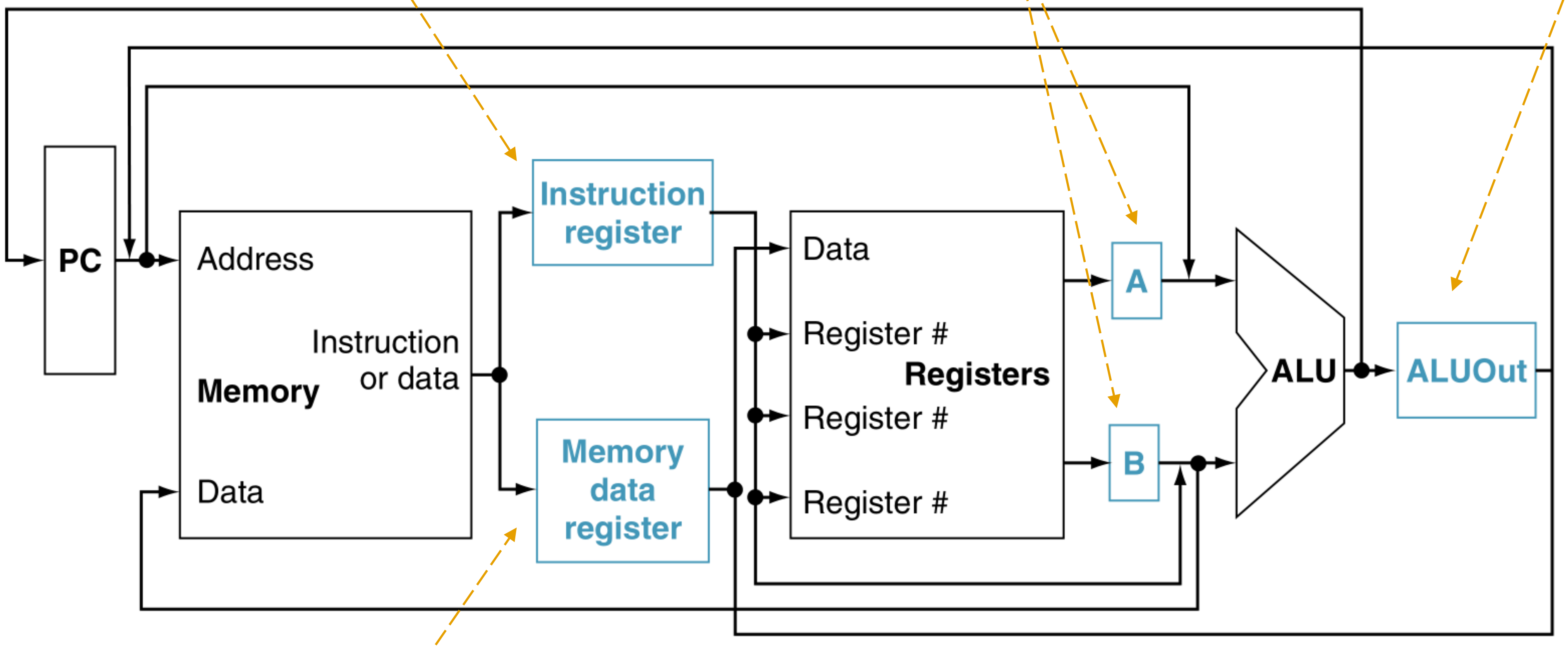
Recall: Additional Registers

Multicycle CPU Datapath, High-level View

*Instruction register (**IR**) – for the 32-bit instruction word read from memory*

One register per output of the register file, to hold the operands for the ALU

A register for ALU output



*Memory data register (**MDR**)– for the 32-bit data word read from memory*

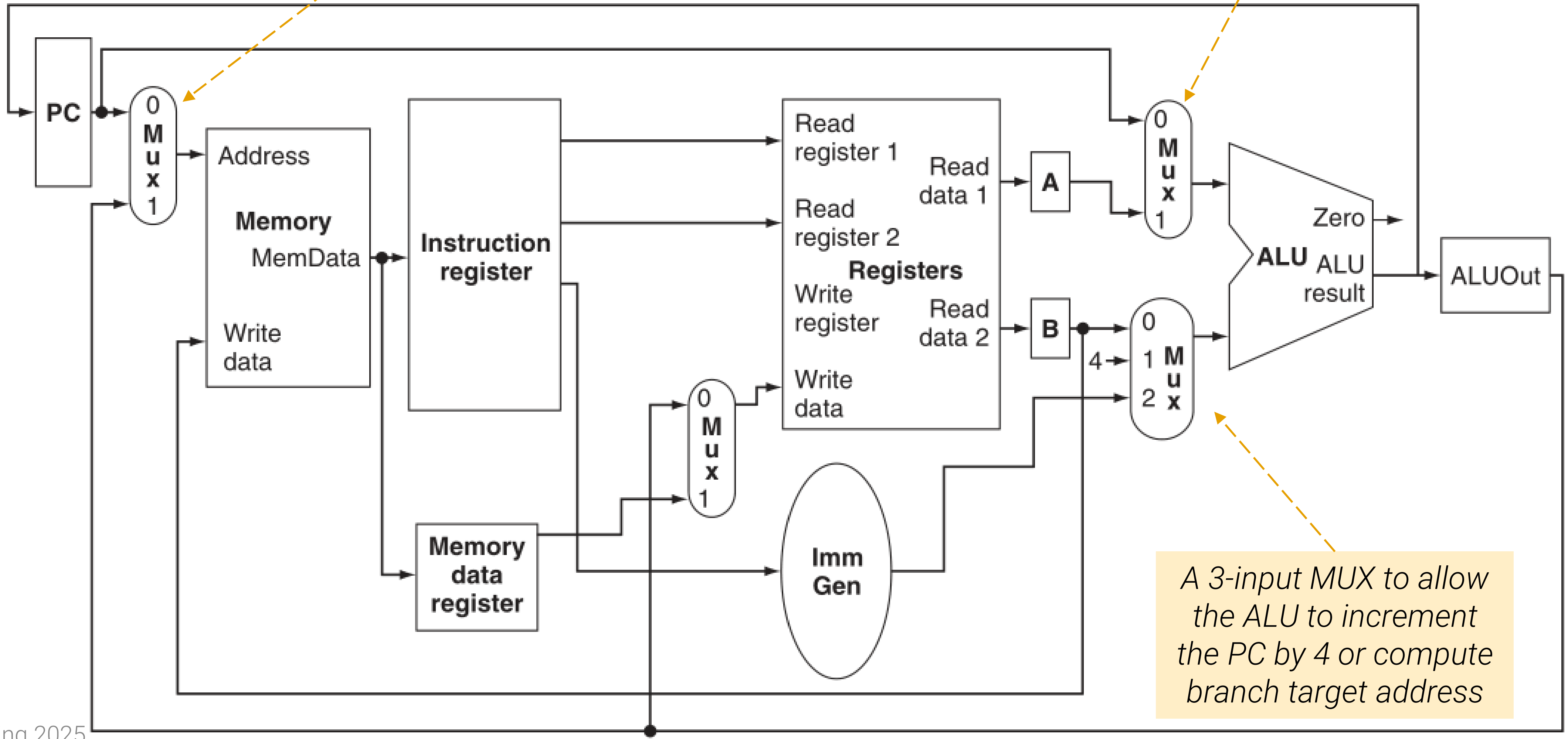
Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Recall: Additional Multiplexers

Sharing Functional Units

A MUX to select between PC and ALU output for the next memory address

A MUX to select between the PC and a register from the register file

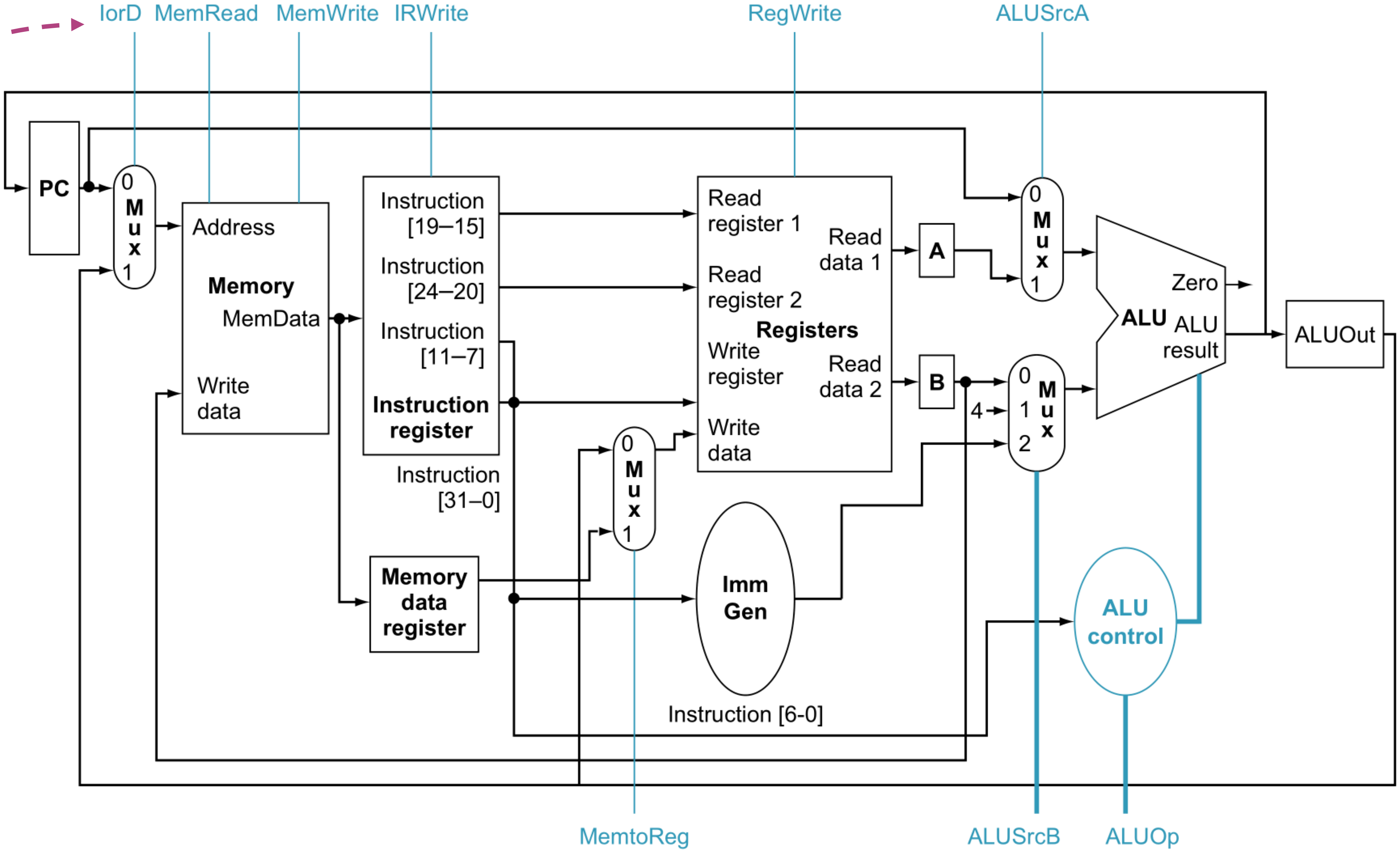


A 3-input MUX to allow the ALU to increment the PC by 4 or compute branch target address

Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Recall: A Multicycle CPU

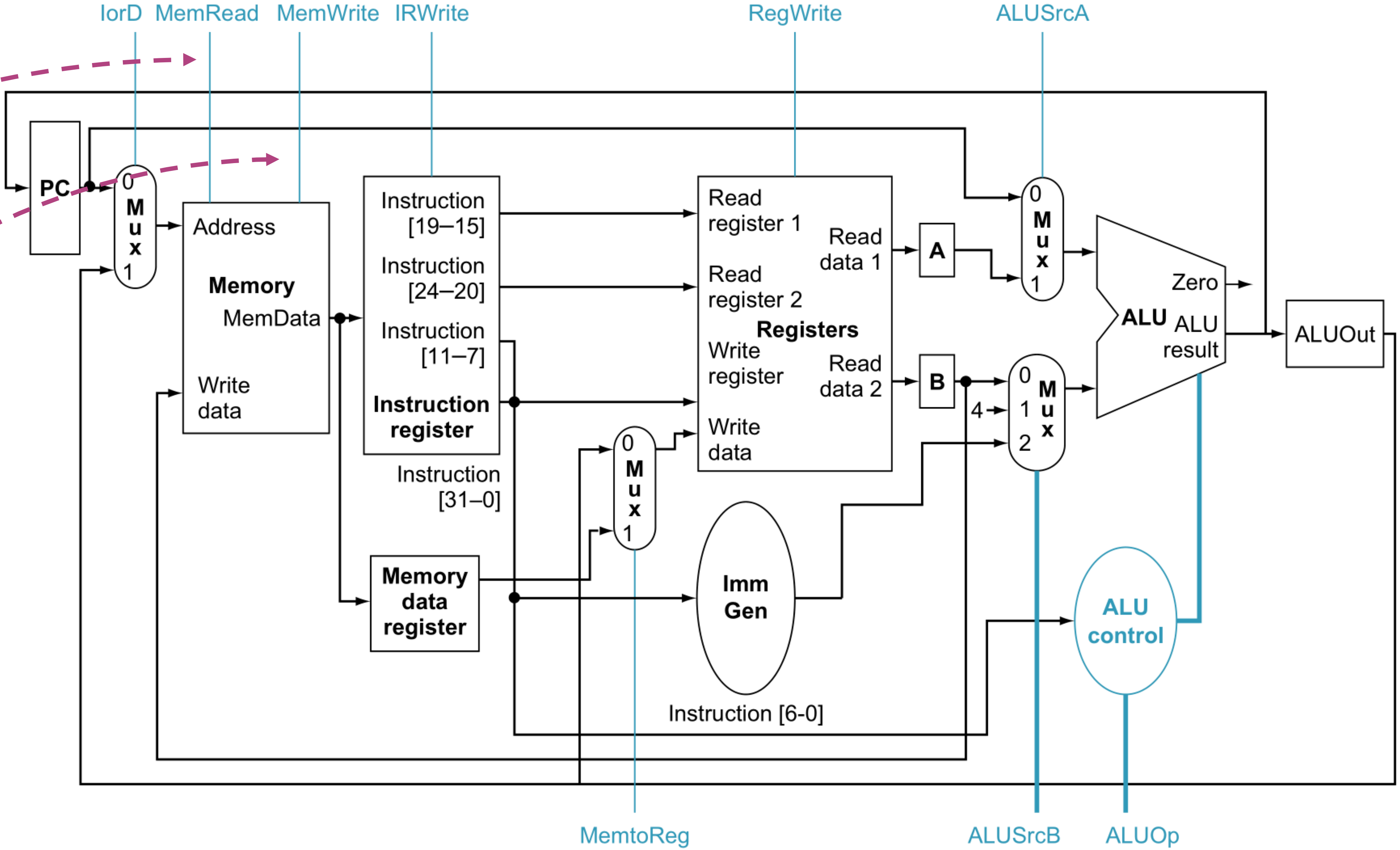
With Some Control Lines Shown



Determines if the address to the memory is supplied from ALUOut register or the PC

Recall: A Multicycle CPU

With Some Control Lines Shown



If asserted, memory contents designated by the address input are put on the output

If asserted, memory contents designated by the address input are replaced by the value on the Write data input

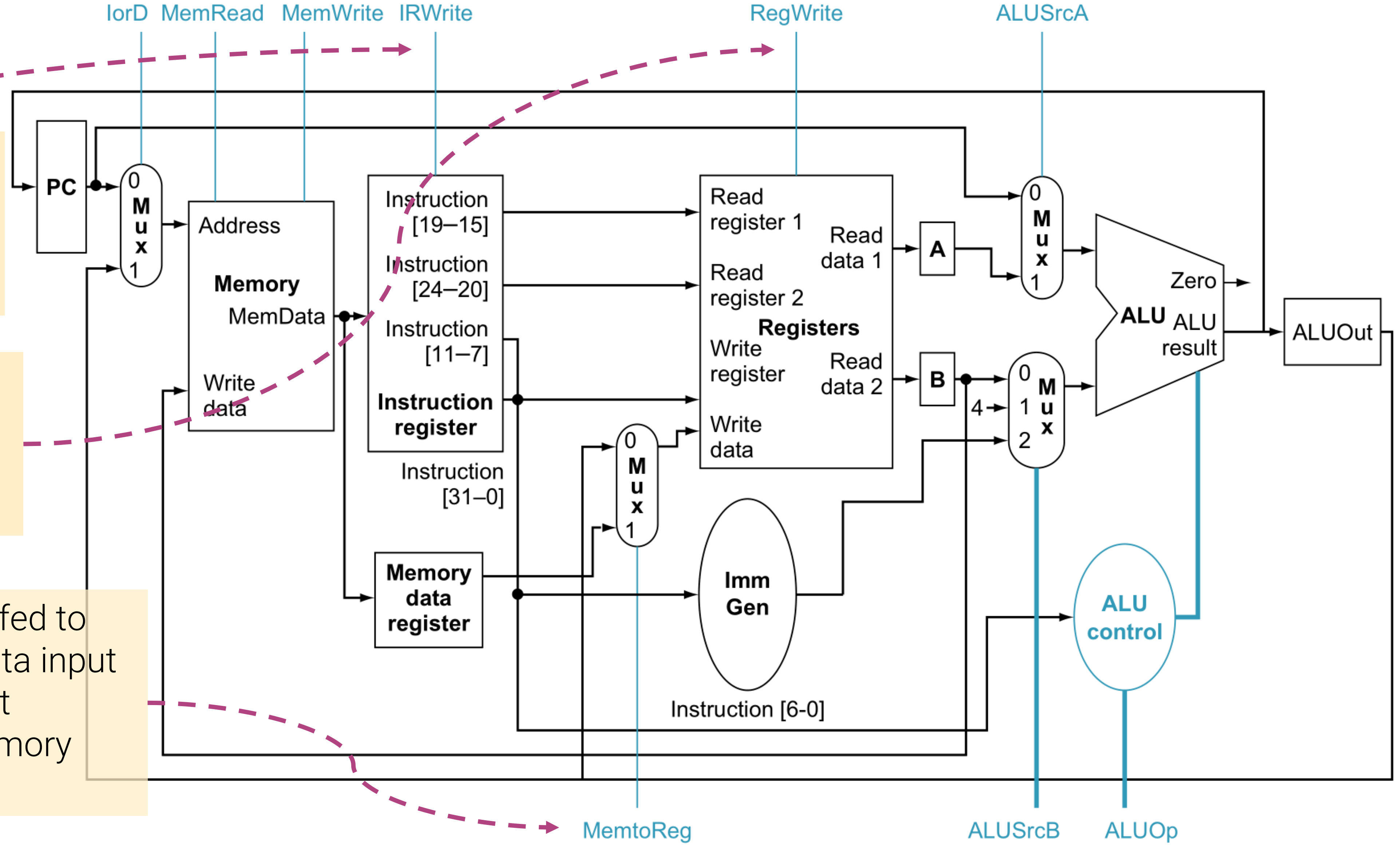
Recall: A Multicycle CPU

With Some Control Lines Shown

The output of the memory is written into the instruction register (IR)

If asserted, the register on the Write reg. input is written with the value on the Write data input

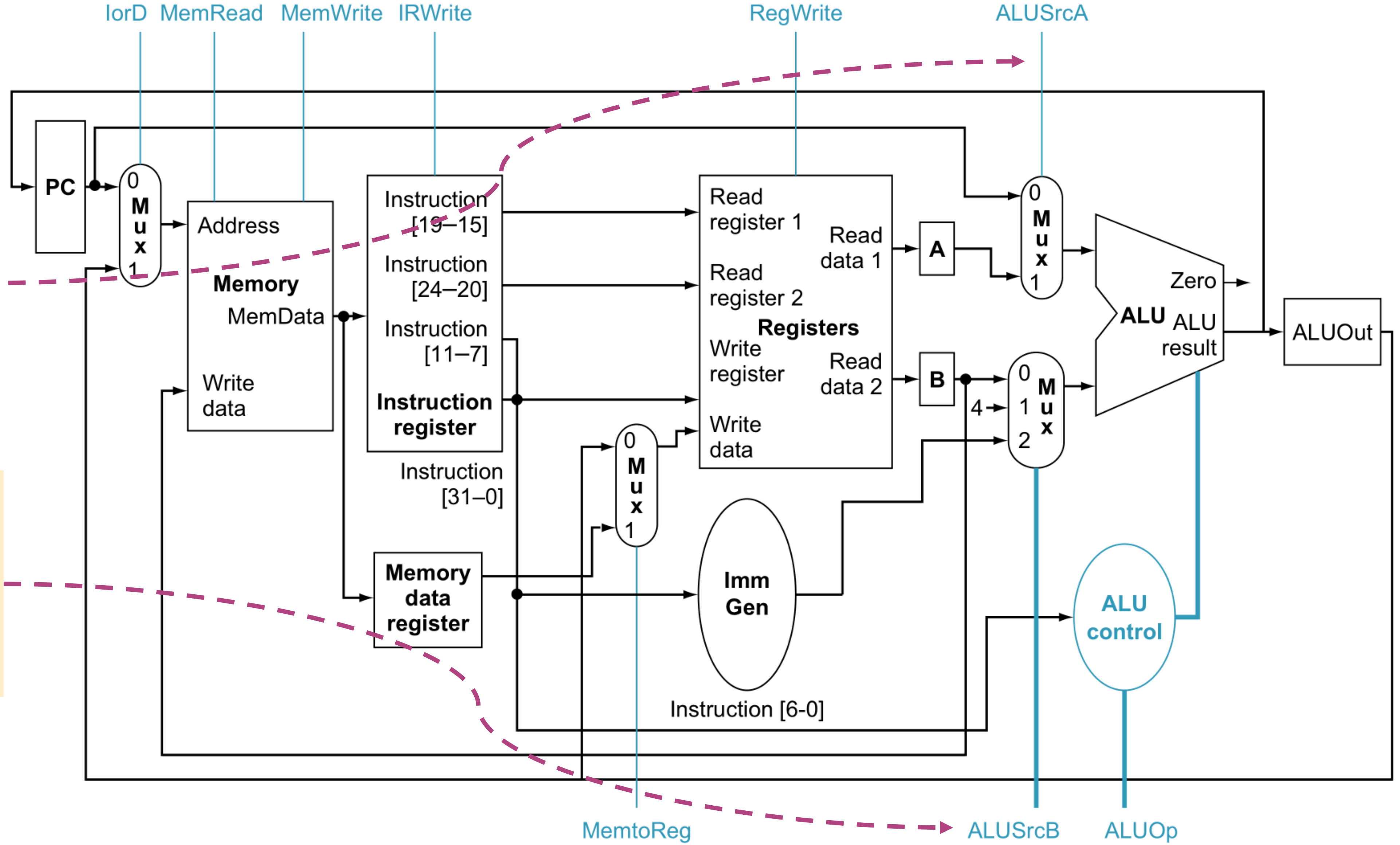
Determines if the value fed to the register file Write data input comes from the ALUOut register or from the memory data register (MDR)



Recall: A Multicycle CPU

With Some Control Lines Shown

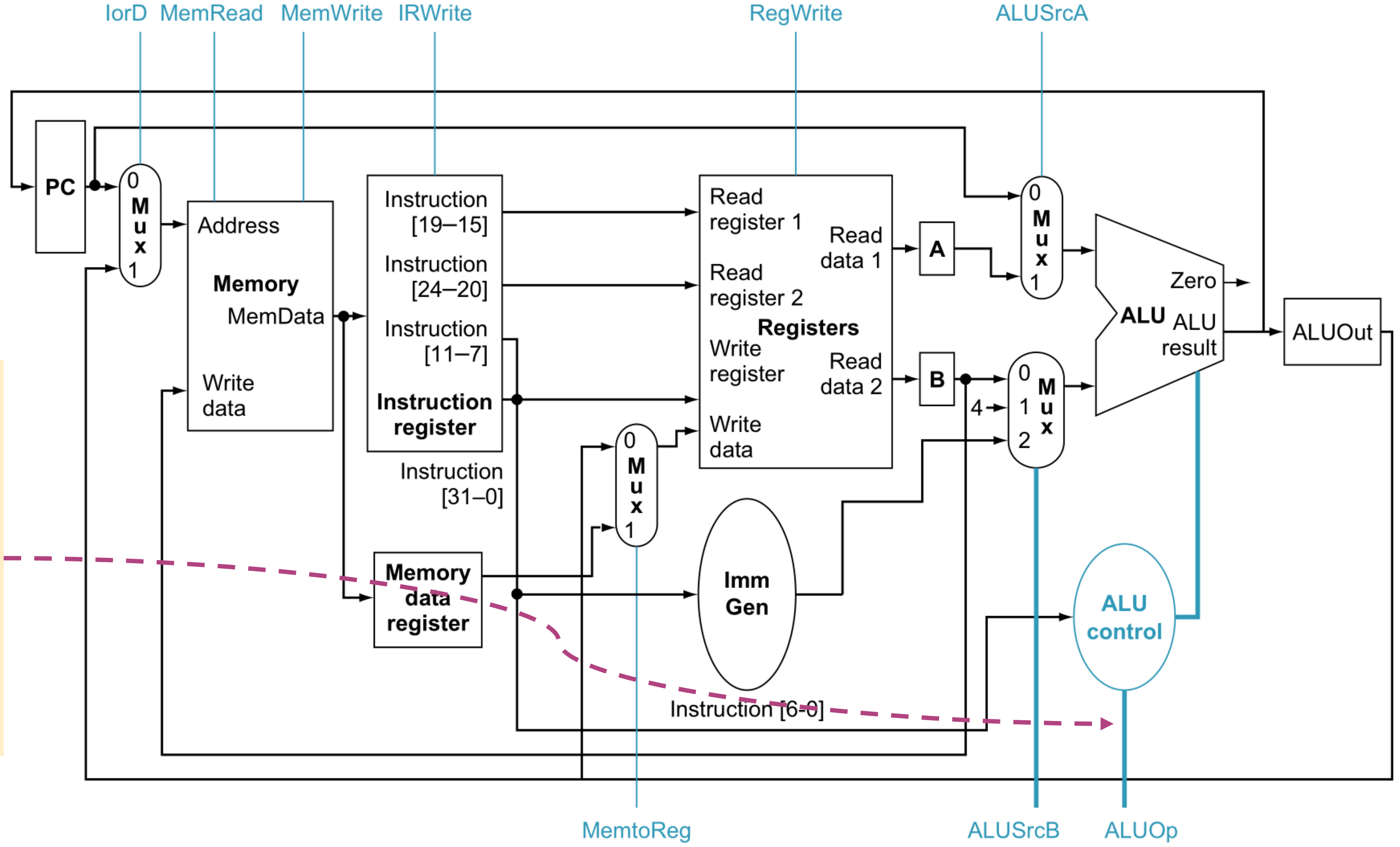
Determines whether the first ALU operand is register A or the PC



Determines whether the second ALU operand is register B, constant 4, or the sign-extended immediate

Recall: A Multicycle CPU

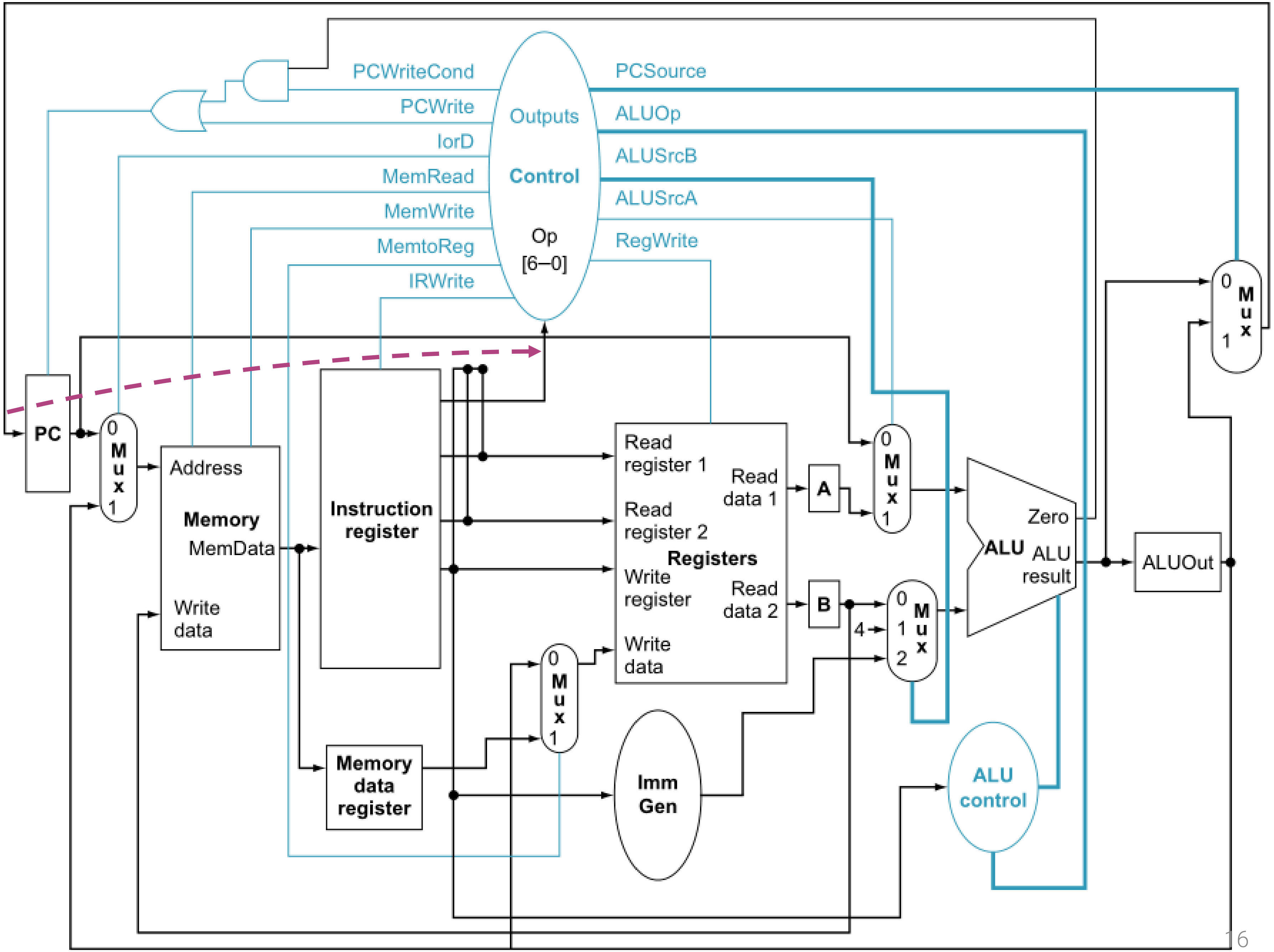
With Some Control Lines Shown



Determines if ALU performs addition (PC = PC+4, or branch target address), subtraction (comparing two registers for a branch if equal instruction), or another operation

Recall:

Multicycle CPU Control



The opcode field of the instruction (register IR) determines the operation of the ALU via **ALUOp** (if not an R-type instruction, but memory access or branch)

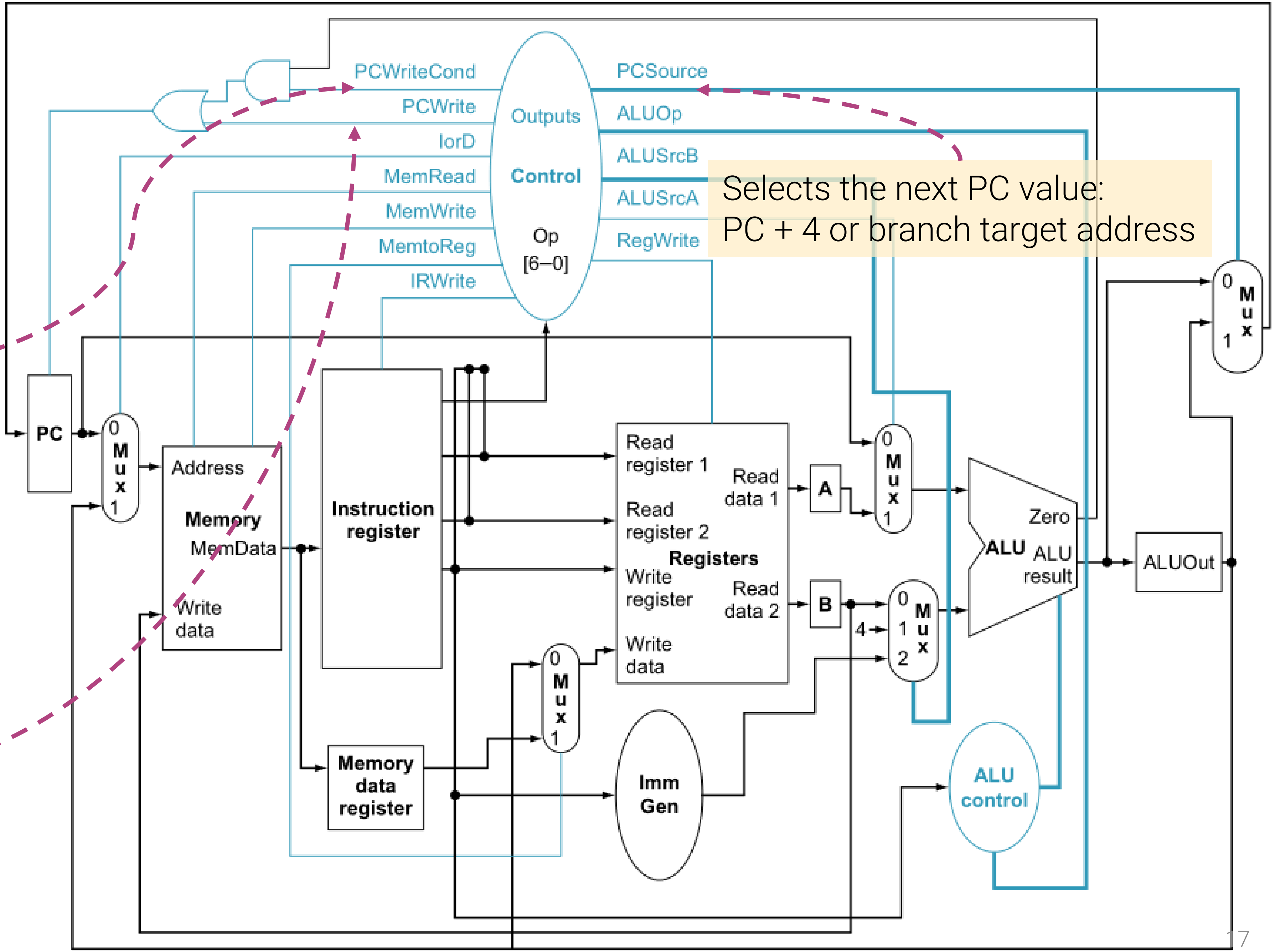
Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Recall:

Multicycle CPU Control

Conditional PC write:
PCWriteCond causes a write of the PC if the branch condition is also true (if the Zero output from the ALU is also active).

Unconditional PC write:
PCwrite causes an unconditional write of the PC, during normal increment (PC = PC + 4)



Selects the next PC value:
PC + 4 or branch target address

Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Recall: Actions of the 1-bit Control Signals

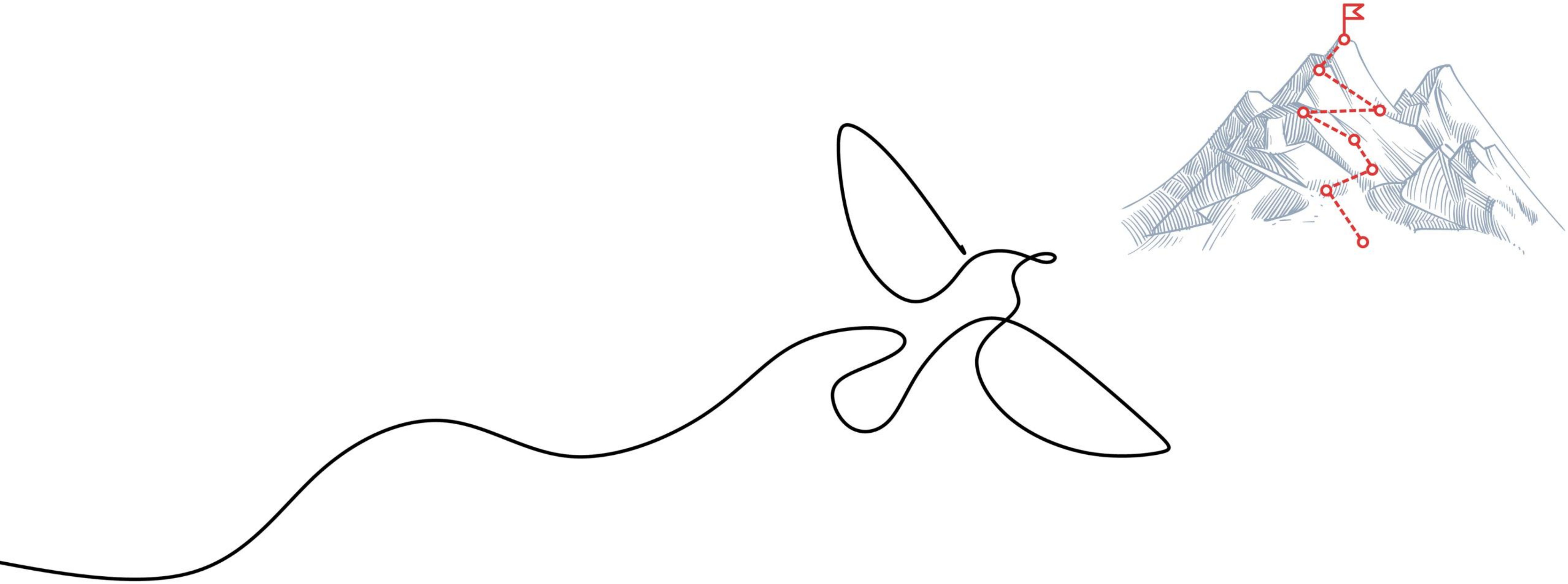
Summary

Signal name	Effect
RegWrite	If asserted, the register on the Write reg. input is written with the value on the Write data input
ALUSrcA	Determines whether the first ALU operand is register A or the PC
MemRead	If asserted, memory contents designated by the address input are put on the output
MemWrite	If asserted, memory contents designated by the address input are replaced by the value on the Write data input
MemtoReg	Determines if the value fed to the register file Write data input comes from the ALUOut register or from the memory data register (MDR)
IorD	Determines if the address to the memory is supplied from ALUOut register or the PC
IRWrite	The output of the memory is written into the instruction register (IR)
PCWrite	The PC is written; the source is controlled by **PCSource**
PCWriteCond	The PC is written if the Zero output from the ALU is also active

Recall: Actions of the 2-bit Control Signals

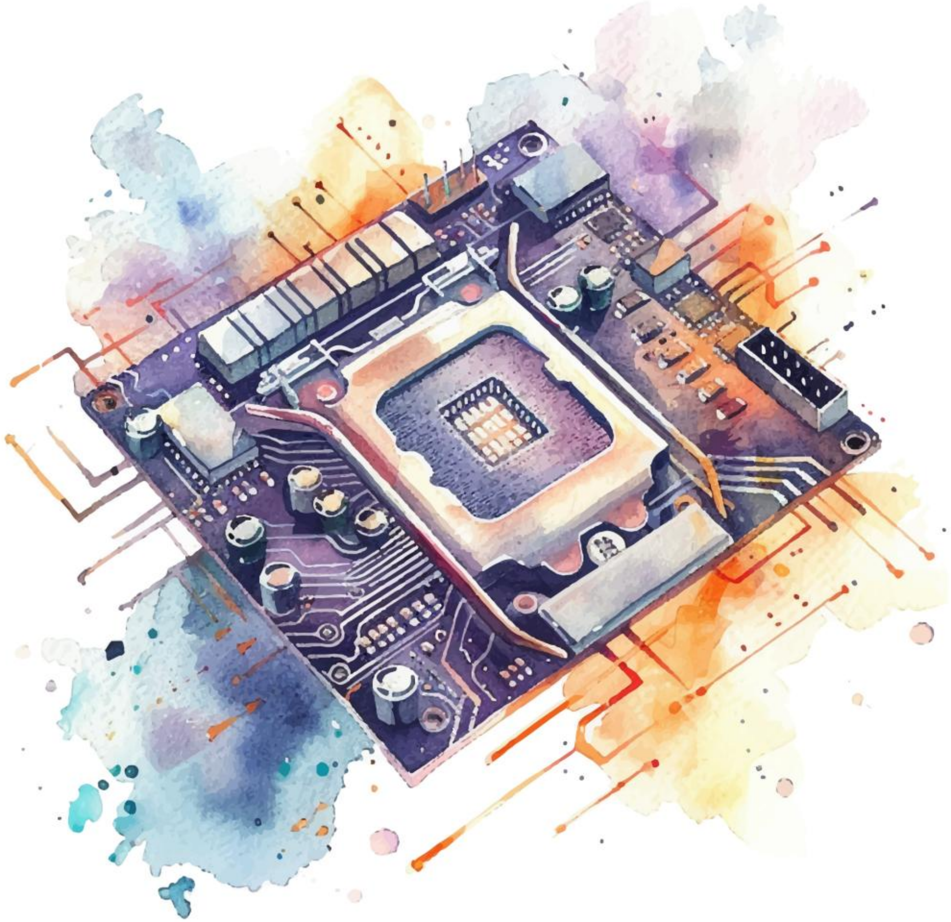
Summary

Signal name	Value	Effect
ALUOp	00	addition
	01	subtraction
	10	The funct field of the instruction determines the operation of the ALU (distinction between `add`, `sub`, `and`, and `or`; they all share the same opcode)
ALUSrcB	00	The second input to the ALU comes from the register B
	01	The second input to the ALU is the constant 4
	10	The second input to the ALU is the immediate generated from the instruction register (IR)
PCSource	00	Output of the ALU (PC+4) is sent to the PC for writing
	01	The contents of the ALUOut register (the branch target address) are sent to the PC for writing
	10	Additional functionality *(not covered in this example, ignore)*



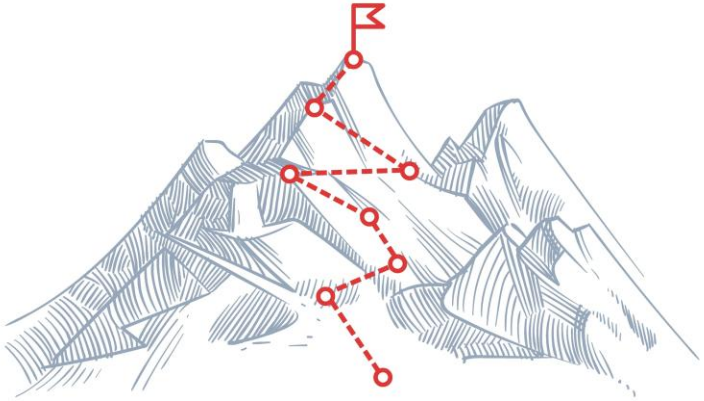
Let's Talk About

- Breaking the instruction execution into cycles
- Multicycle CPU FSM



© EnelEva / Adobe Stock

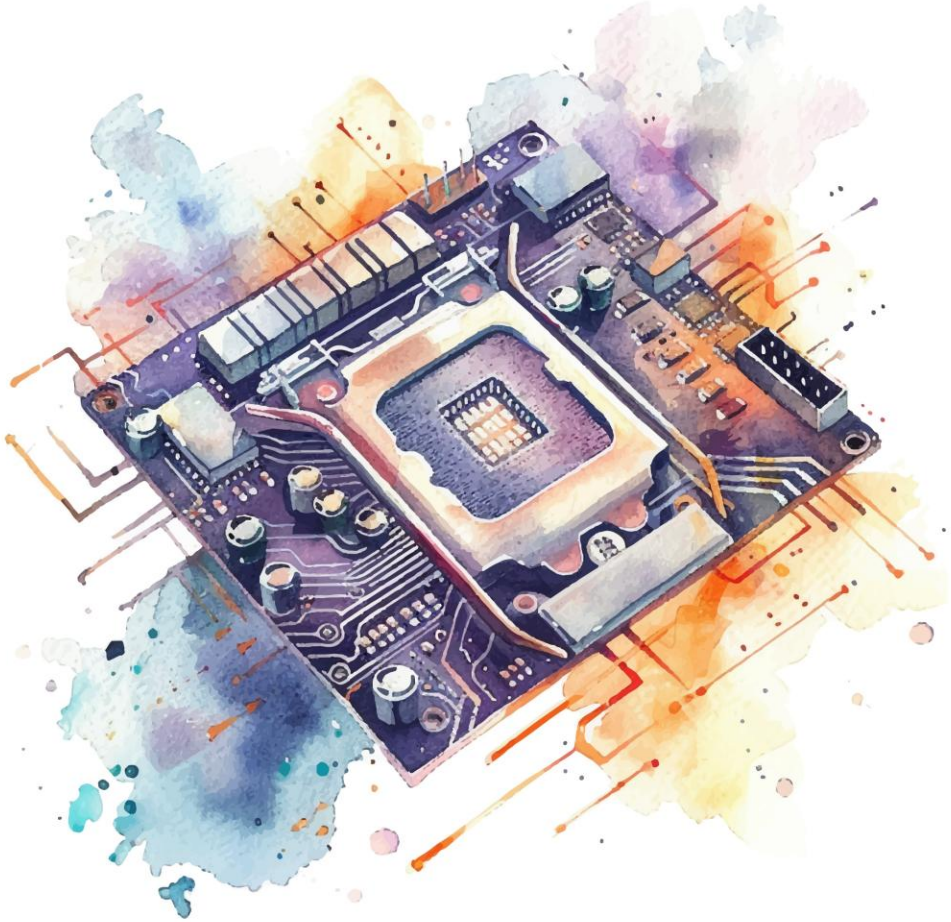
Learning Outcomes



- List the instruction steps and explain them
- Draw and explain the multicycle CPU FSM
- Quantify and compare CPU performance

Quick Outline

- Instruction execution steps
  - Instruction fetch
  - Instruction decode and operand fetch
  - Execution, memory address computation, or branch completion
  - Memory access or R-type instruction completion
  - Memory read completion step
- Multicycle CPU FSM
- Complete FSM
- Example: Compute CPI in our multicycle CPU



© EnelEva / Adobe Stock

Instruction Execution Across Cycles

- To determine which control signals are needed and their setting, we need to look at what should happen in each CPU cycle
- When deciding how to break instruction execution into cycles, the goal is performance
- We break execution into a series of steps, each taking one cycle, attempting to keep the amount of work per cycle roughly equal

Instruction Execution Across Cycles

Contd.

- Let us restrict each step to contain **at most**
  - one memory access
  - one register file access
  - an ALU operation
- The CPU cycle could be **as short as the longest** of the above
- At the end of every CPU cycle, any data needed on a subsequent cycle must be stored in a register
- Edge-triggered design: We can continue to read the current value of a register; the new value does not appear until the next cycle

Instruction Execution Across Cycles

Contd.

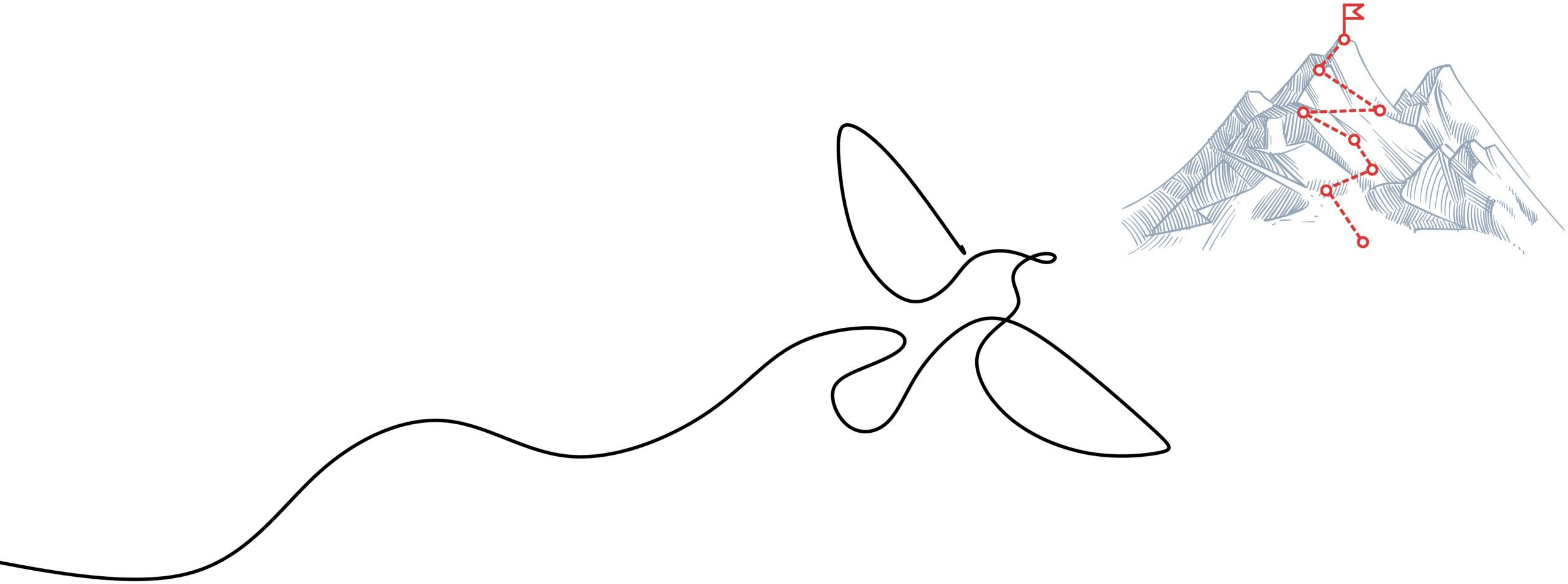
- Values required by subsequent cycles must be kept constant for the duration of at least the subsequent cycle:
  - Major state elements
    - Program counter: **PC**
    - Register file, memory
  - Temporary registers that are written on every clock cycle
    - At the output of the register file: **A** and **B**
    - At the memory output, memory data register: **MDR**
    - At the output of the ALU: **ALUOut**
  - Temporary register with write control
    - Instruction register: **IR**

In a Single-Cycle CPU Datapath…

- Each instruction uses a set of datapath elements to carry out its execution
- Many of the datapath elements operate in series, using the output of another element as an input
- Some datapath elements operate in parallel:
  - E.g., PC is incremented and the instruction is read at the same time

In a Multicycle CPU Datapath…

- All operations in one instruction step occur in parallel within one clock cycle
- Successive instruction steps operate in series in different clock cycles
- The limitation of one ALU operation, one memory access, and one register file access determines what can "fit" in one instruction step (one cycle)



Breaking the Instruction Execution…

…into Clock (CPU) Cycles

Five steps

- Instruction fetch
- Instruction decode and register (operand) fetch
- Execution, memory address computation, or branch completion
- Memory access or R-type instruction completion
- Memory read completion

Step 1

Instruction Fetch

- Fetch the instruction from memory and compute the address of the next instruction in the program sequence
  - `IR <= Memory [PC]`
  - `PC <= PC + 4`

Note: deliberate use of nonblocking Verilog operator symbol <=; it indicates that right-hand sides are evaluated and then all assignments are made, which is effectively how the hardware executes during the cycle

- Operation
  - Send the PC to the memory as the address, perform a memory read, and write the fetched instruction to the Instruction Register (IR)
  - Increment the PC by four to prepare for the subsequent instruction
  - Save the incremented instruction address in the PC

Step 1

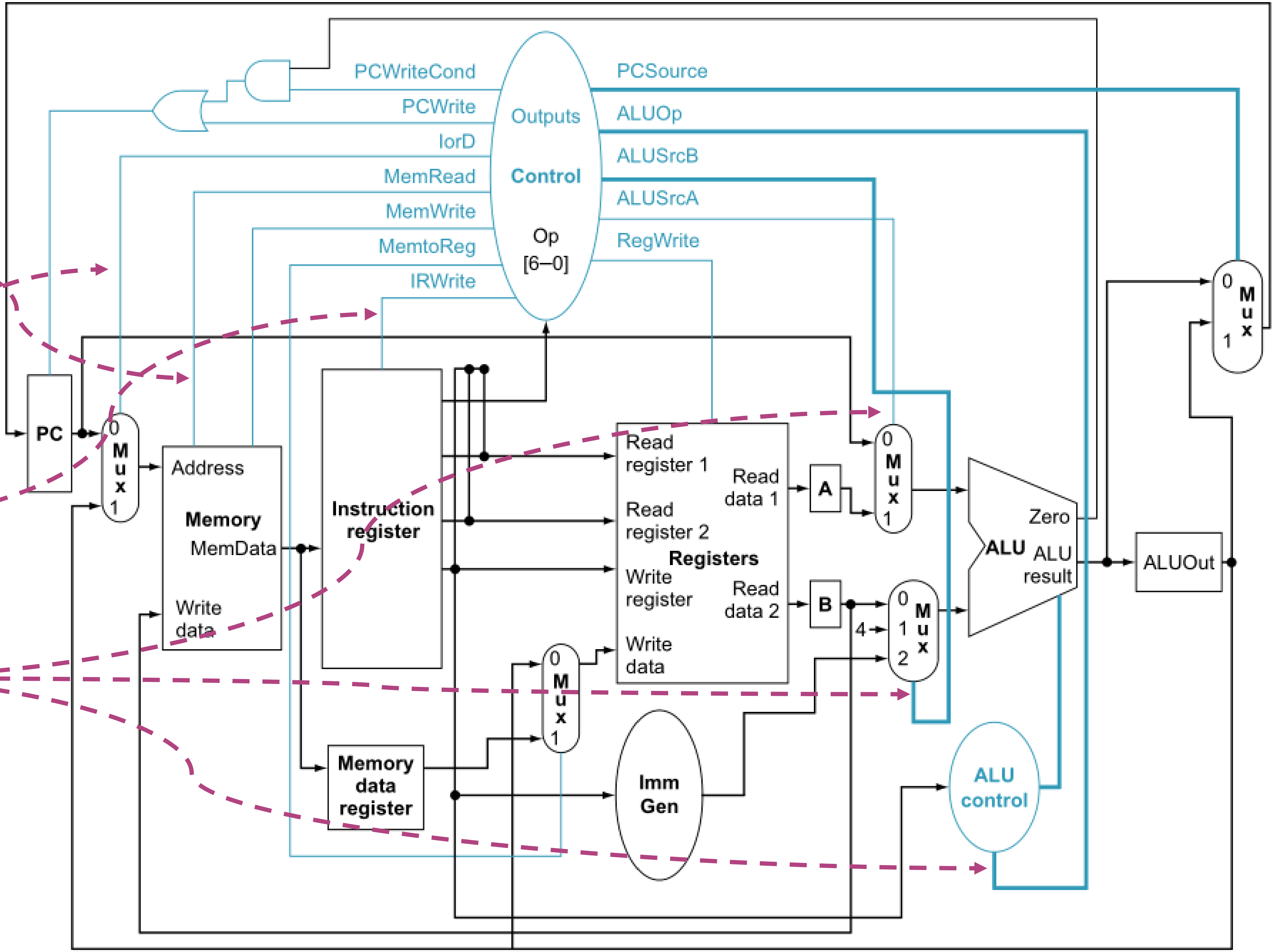
Instruction Fetch

Assert **MemRead**

Set **IorD** to zero, to select the PC as the source of the memory address

Assert **IRWrite**

Set **ALUSrcA**, **ALUSrcB**, and **ALUOp** so that the ALU computes PC = PC + 4

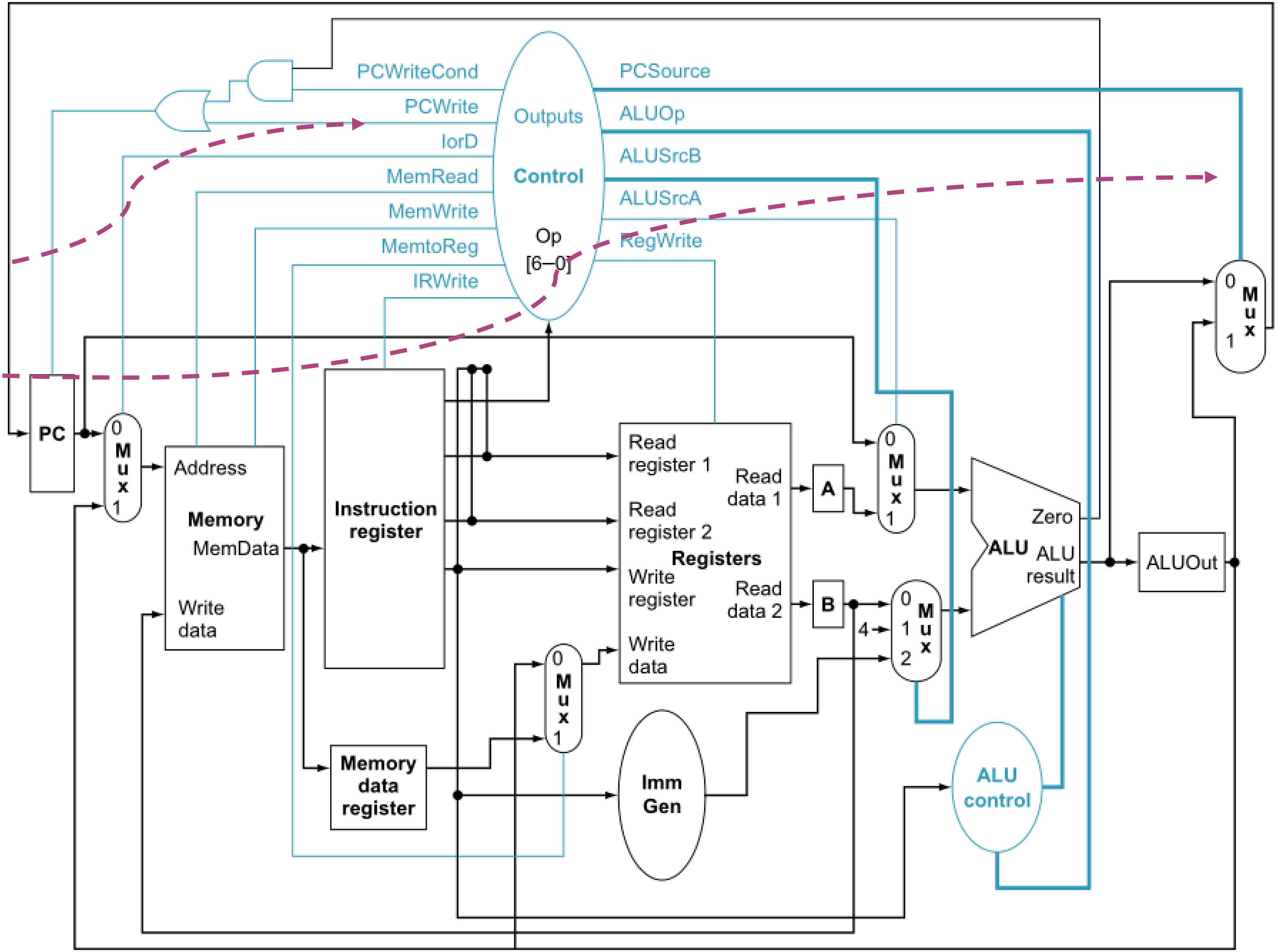


Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2^{nd} Ed.

Step 1

Instruction Fetch

Update PC:
- set PCWrite
- set PCSource to zero



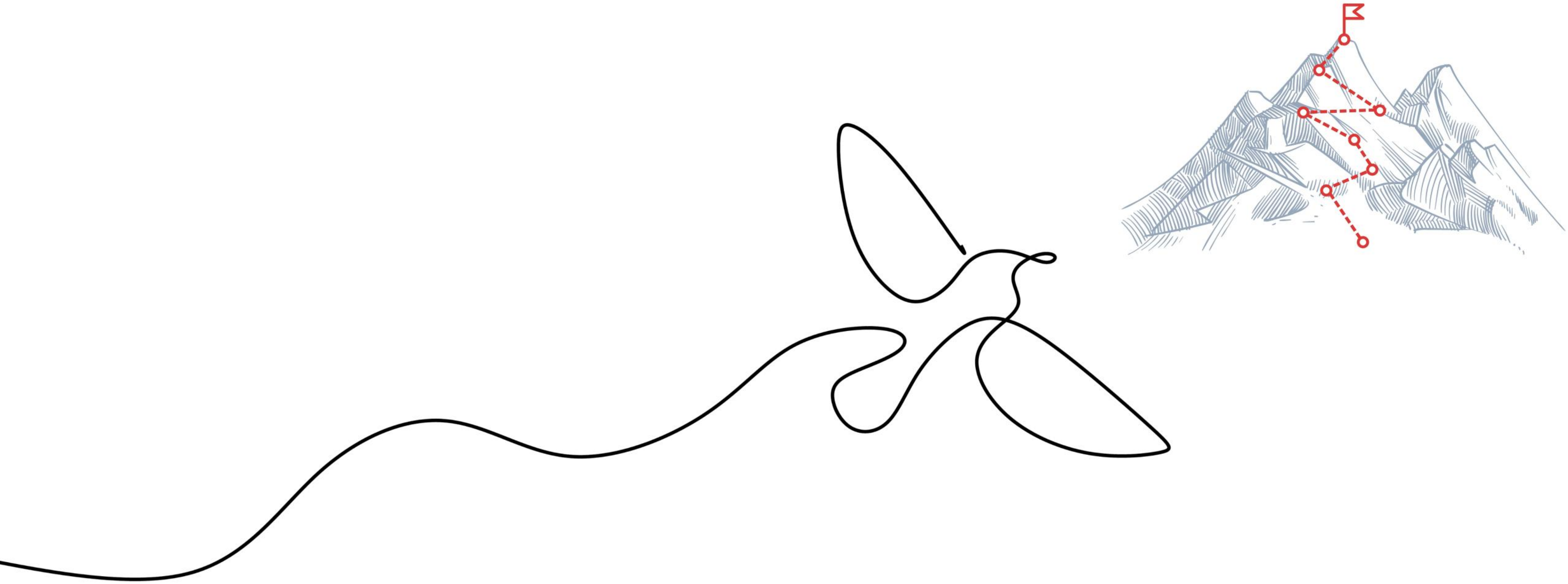
PC increment and instruction memory access occur in parallel. The new PC value is not visible until the next cycle. Incremented PC will also be saved in ALUOut, but this action is harmless.

Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 1: Summary

Instruction Fetch

- Operations and control signals involved
  - **IorD:** Select PC as the source address
  - **MemRead:** Perform a memory read
  - **IRWrite:** Write the instruction from the memory into the Instruction Register
  - Increment the PC by four
    - **ALUSrcA:** Send the PC to the first input of the ALU
    - **ALUSrcB:** Send 4 to the second input of the ALU
    - **ALUOp:** Instruct ALU to perform addition
  - Save the incremented instruction address in the PC
    - **PCSource:** Send ALU output to the PC
    - **PCWrite:** Write to the PC



Step 2

Instruction Decode and Register (Operand) Fetch

- In the previous and this step, we do not yet know what the instruction is
- We can only perform actions that are either
  - Applicable to all instructions (e.g., fetching the instruction in step 1) or
  - Not harmful, in case the instruction isn't what we think it might be

- What can we do?
- (1) Can read `rs1` and `rs2` registers
  - It's not harmful to read them, even if not necessary
  - Those values may be needed later, so we keep them in temporary registers A and B

Step 2

Instruction Decode and Register (Operand) Fetch

- In the previous and this step, we do not yet know what the instruction is
- We can only perform actions that are either
  - Applicable to all instructions (e.g., fetching the instruction in step 1) or
  - Not harmful, in case the instruction isn't what we think it might be

- What else can we do?
- (2) Can compute branch target address with the ALU
  - It's not harmful because we can ignore this value if the instruction turned out not to be a branch
  - The value may be needed later, so we keep it in register ALUOut

Step 2

Instruction Decode and Register (Operand) Fetch

- **Q:** Why do these optimistic actions?
- **A:** Performing "optimistic" actions early helps decrease the number of cycles needed to execute an instruction

- **Q:** What makes doing these optimistic actions possible?
- **A:** The regularity of the instruction formats
  - For example, if the instruction has two register operands, they are always in the `rs1` and `rs2` fields

Step 2

Instruction Decode and Register (Operand) Fetch

- *Recall:* Step 2 performs a few "optimistic" actions, as they do not hurt, but may prove helpful later when it is known what the instruction is
  - `A <= RF[Instruction Register[19:15]]`
  - `B <= RF[Instruction Register[24:20]]`
  - `ALUOut <= PC + offset`

- Operations
  - Access register file (RF) and read registers **`rs1`** and **`rs2`**
  - Write to registers A and B; they are overwritten every clock cycle
  - Compute the branch target address and place it in the ALUOut register, from where it will be read in the next clock cycle if the instruction is a branch

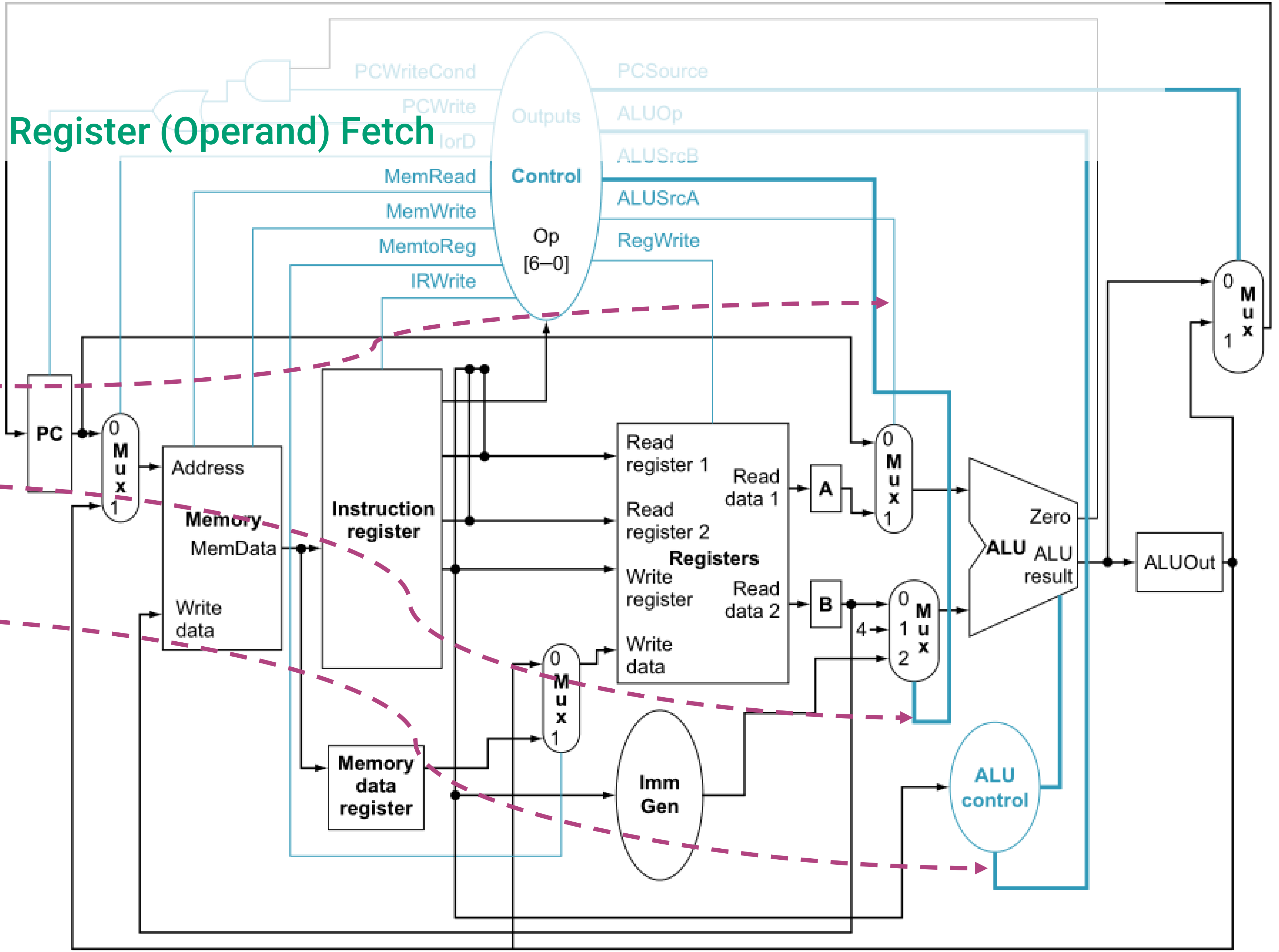
Step 2

Instruction Decode and Register (Operand) Fetch

Set **ALUSrcA** to zero (PC sent to ALU)

Set **ALUSrcB** so that the offset is sent to the ALU

Set **ALUOp** so that the ALU adds



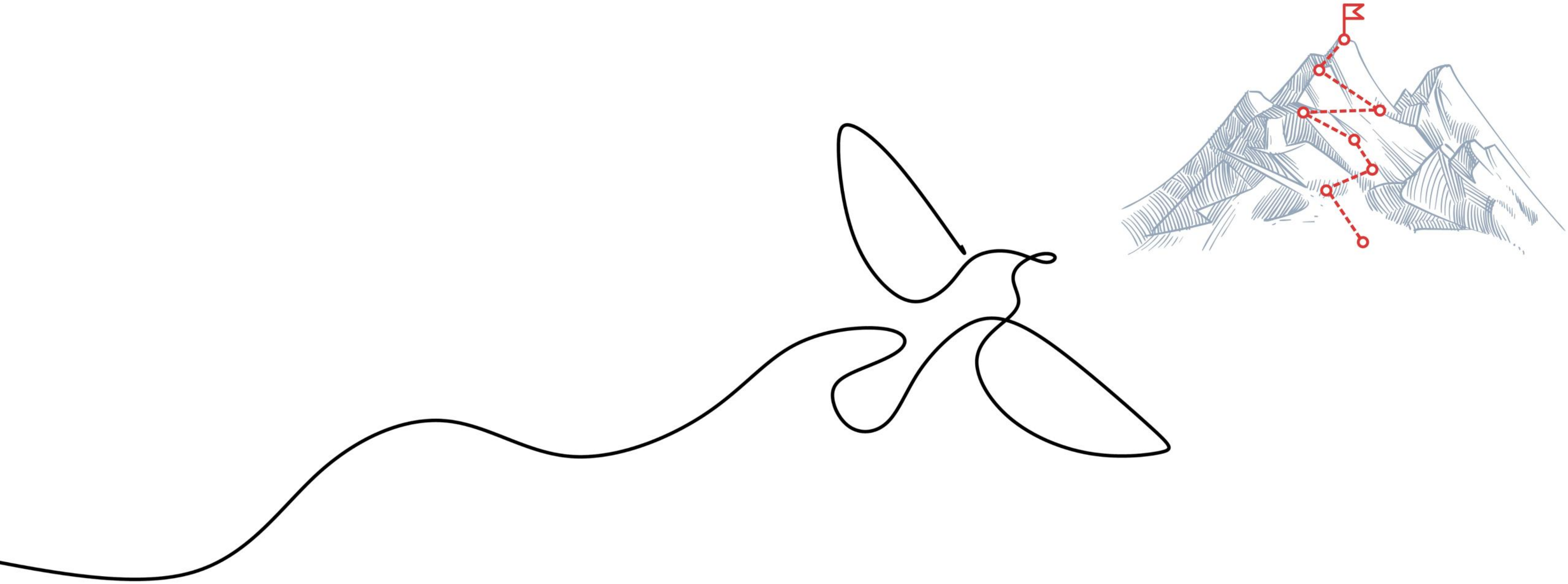
Register file access and computation of branch target occur in parallel

Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 2: Summary

Instruction Decode and Register (Operand) Fetch

- Operations and control signals involved
  - Access the register file and read registers `rs1` and `rs2`
  - Write to the registers A and B; they are overwritten every clock cycle
  - Compute the branch target address and place it in the ALUOut register, from where it will be read on the next clock cycle if the instruction is a branch
    - **ALUSrcA:** PC sent to the ALU
    - **ALUSrcB:** PC offset (for computing branch target address) sent to the ALU
    - **ALUOp:** ALU instructed to perform addition
  - The register file access and branch target address computation occur in parallel



Step 3

Execution, Memory Address Computation, or Branch Completion

- ALU operates on the operands prepared in the previous cycle, performing a function depending on the instruction class
  - Memory address computation (load, store) or
  - Arithmetic-logical instruction (R-type) or
  - Branch if equal

Step 3

Memory Address Computation

- *Recall:* ALU operates on the operands prepared in the previous cycle, performing a function depending on the instruction class
- **Memory address computation**
  - ALU adding the operands to form the memory address

Step 3

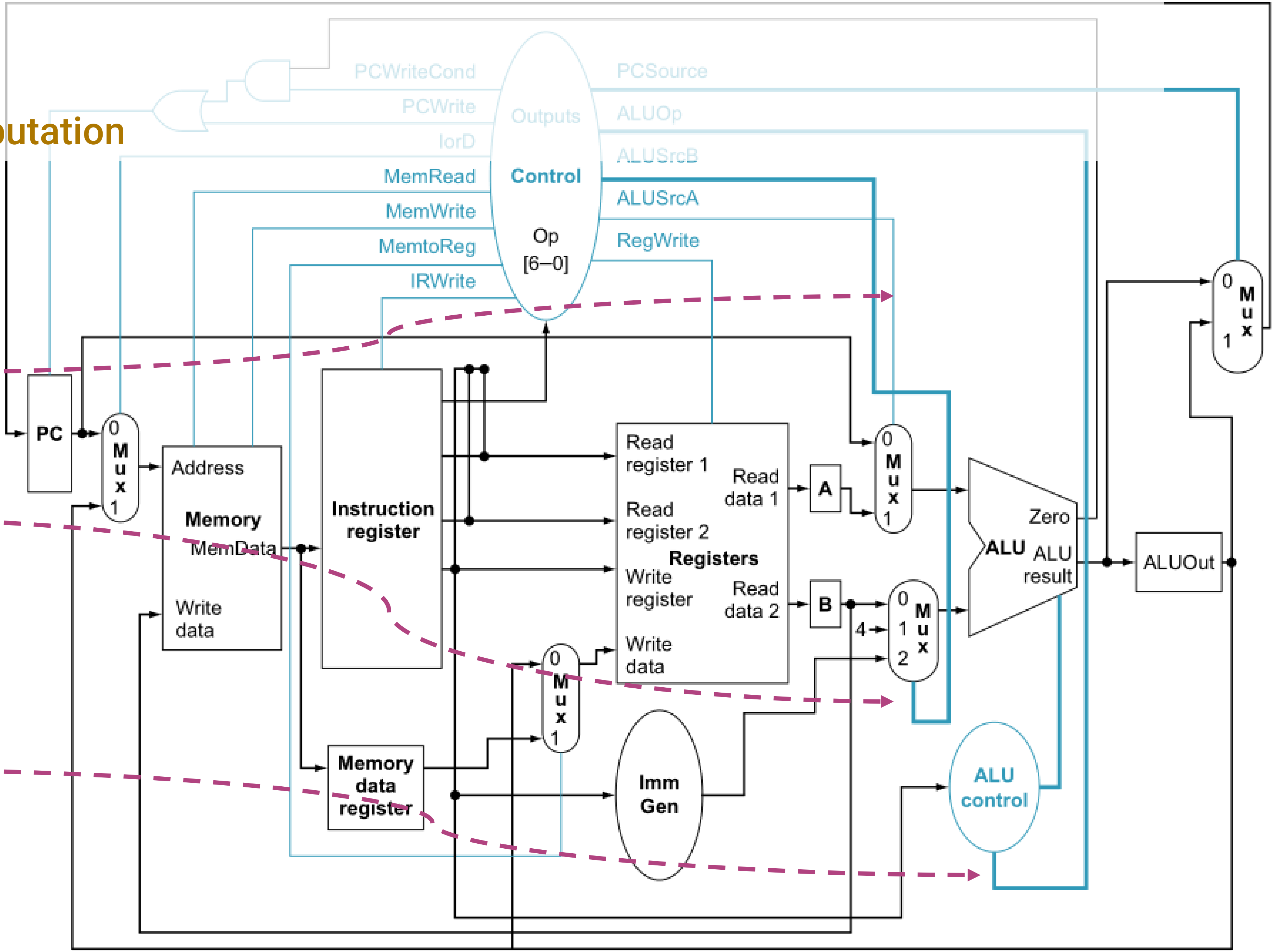
Memory Address Computation

ALU is adding the operands to form the memory address

Set **ALUSrcA** to 1, so that register A is the first ALU input

Set **ALUSrcB** so that the output of the offset generation unit is the second ALU input

Set **ALUOp** so that the ALU adds



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 3: Summary

Memory Address Computation

Memory address computation—operations and the control signals involved

- ALU adding the operands to form the memory address
- **ALUSrcA:** First ALU input is register A
- **ALUSrcB:** Second ALU input is the offset
- **ALUOp**: ALU instructed to perform addition

Step 3

R-type instruction execution

- *Recall:* ALU operates on the operands prepared in the previous cycle, performing a function depending on the instruction class
- **Arithmetic-logical instruction (R-type)**
  - `ALUOut <= A op B`
  - ALU performing the operation specified by the opcode
- *Recall:* Distinction between **`add`**, **`sub`**, **`and`**, and **`or`** cannot be made based on the **`opcode`** (it is the same!); the `funct` field serves the purpose

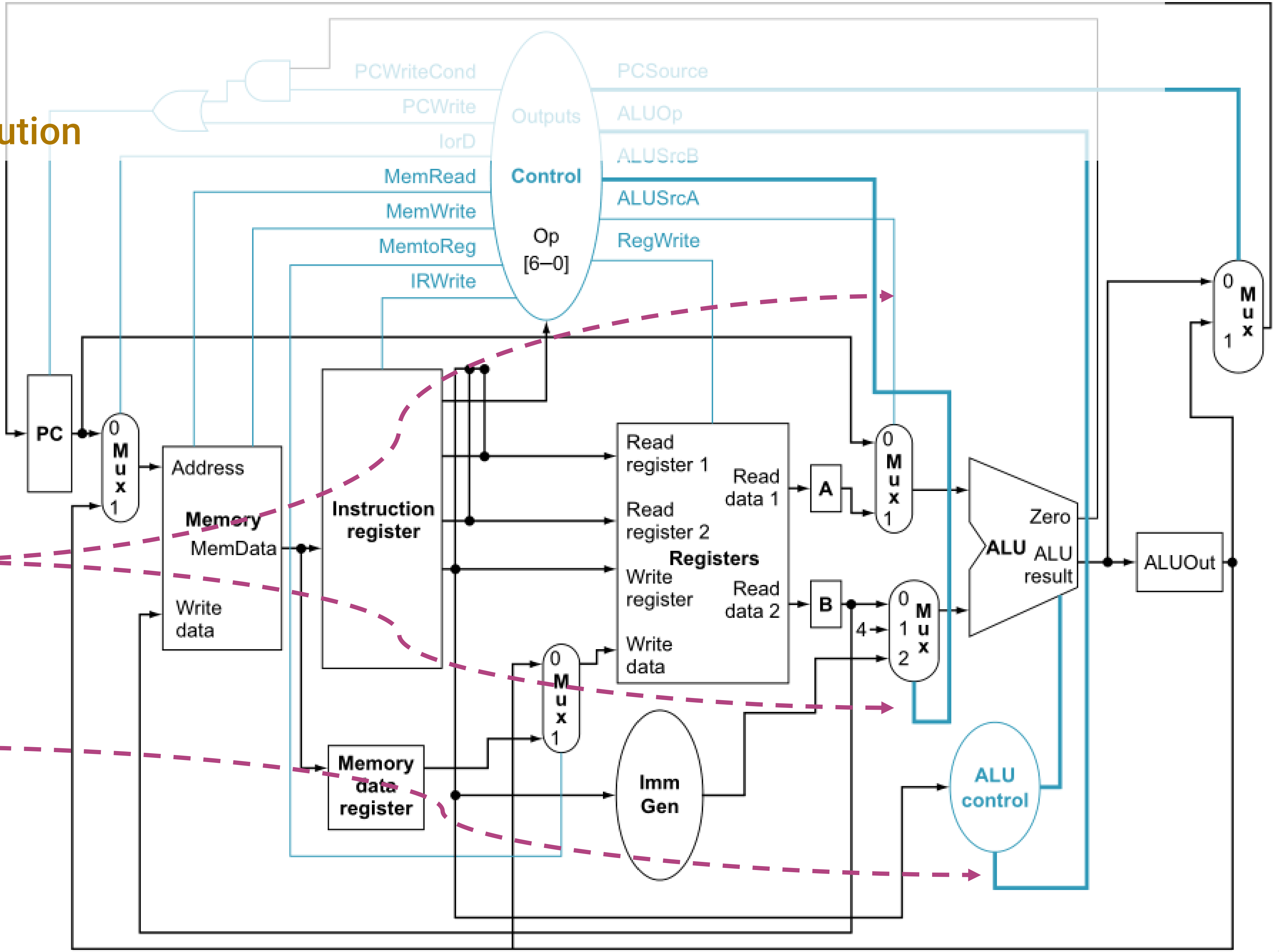
Step 3

R-type instruction execution

ALU is performing the operation on two values read from the register file in the previous cycle

Set **ALUSrcA** to 1 and **ALUSrcB** so that both ALU operands are from the register file

Set **ALUOp** so that `funct` field from the IR is used to determine the ALU operation (`add`, `sub`, `and`, `or`)



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 3: Summary

R-type instruction execution

- **Arithmetic-logical instruction (R-type)**—operations and the control signals
  - `ALUOut <= A op B`
  - ALU performing the operation specified by the `funct` field
  - **ALUSrcA:** First ALU input is register A
  - **ALUSrcB:** Second ALU input is register B
  - **ALUOp:** ALU operation determined by the `funct` field

Step 3

Branch Completion

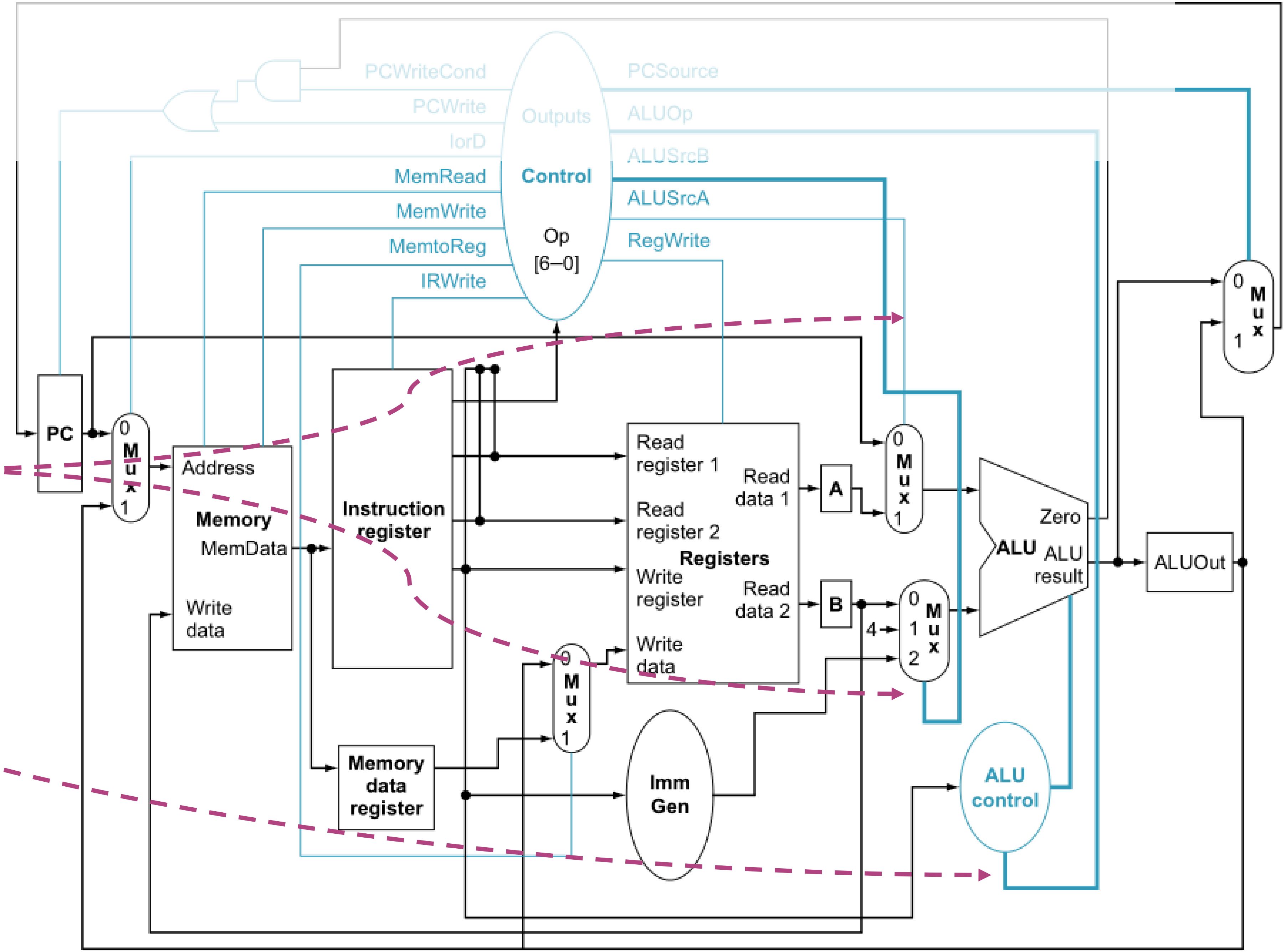
- *Recall:* ALU operates on the operands prepared in the previous cycle, performing a function depending on the instruction class
- **Branch if equal**
  - `if (A == B) PC <= ALUOut`
  - ALU subtracts registers A and B;
  - Zero output is asserted if A equals B;
  - If Zero output is asserted, and the instruction is **`beq`**, PC is updated with the value coming from the ALUOut register

Step 3

Branch Completion

Set ALUSrcA to 1 and ALUSrcB so that both ALU operands are from the register file

Set ALUOp so that ALU subtracts



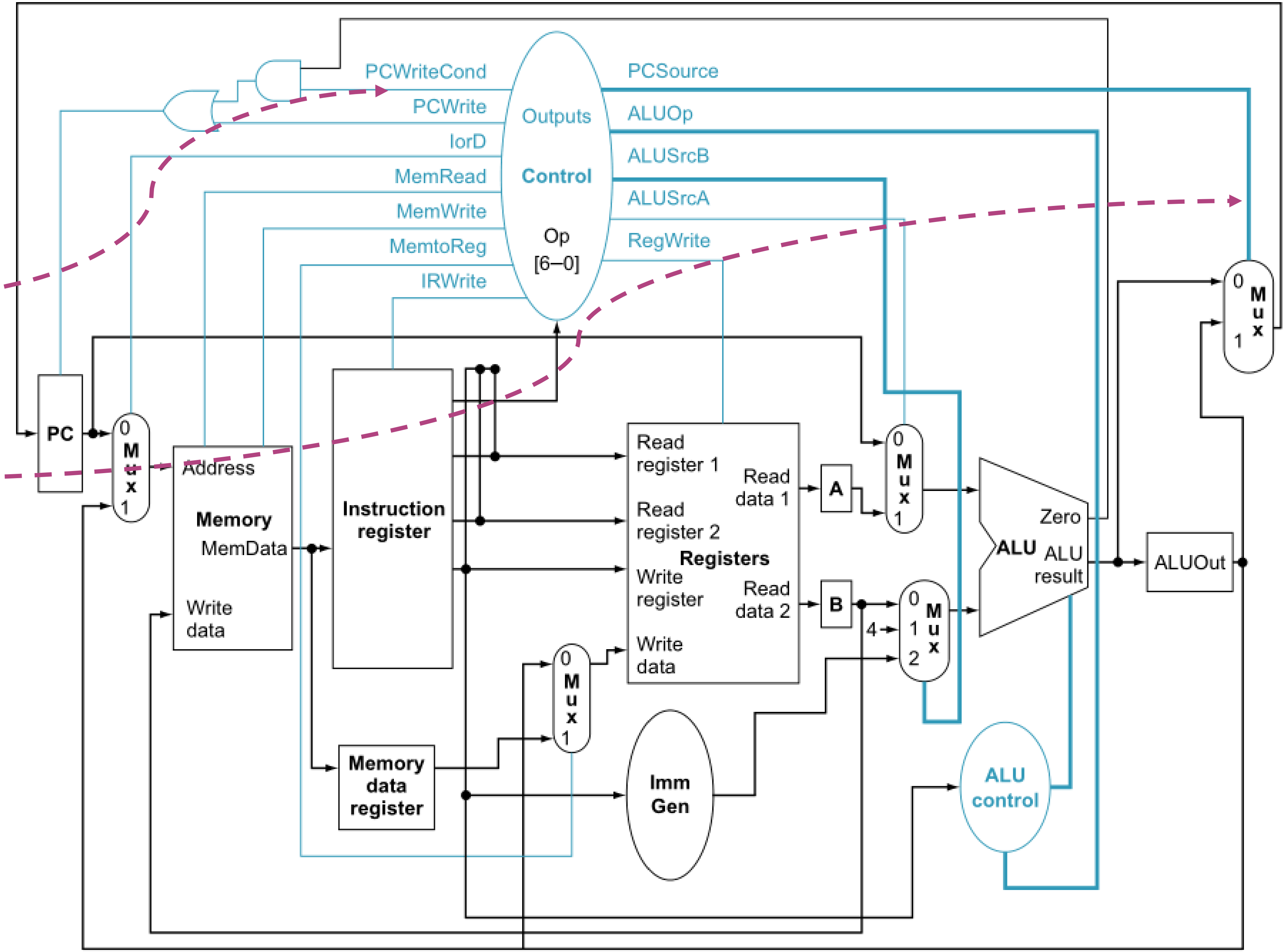
Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 3

Branch Completion

Assert **PCWriteCond** to update the PC if the Zero output of the ALU is asserted

Set **PCSource** to 1 so the value written into PC comes from ALUOut, which holds the branch target address computed in the **previous** cycle

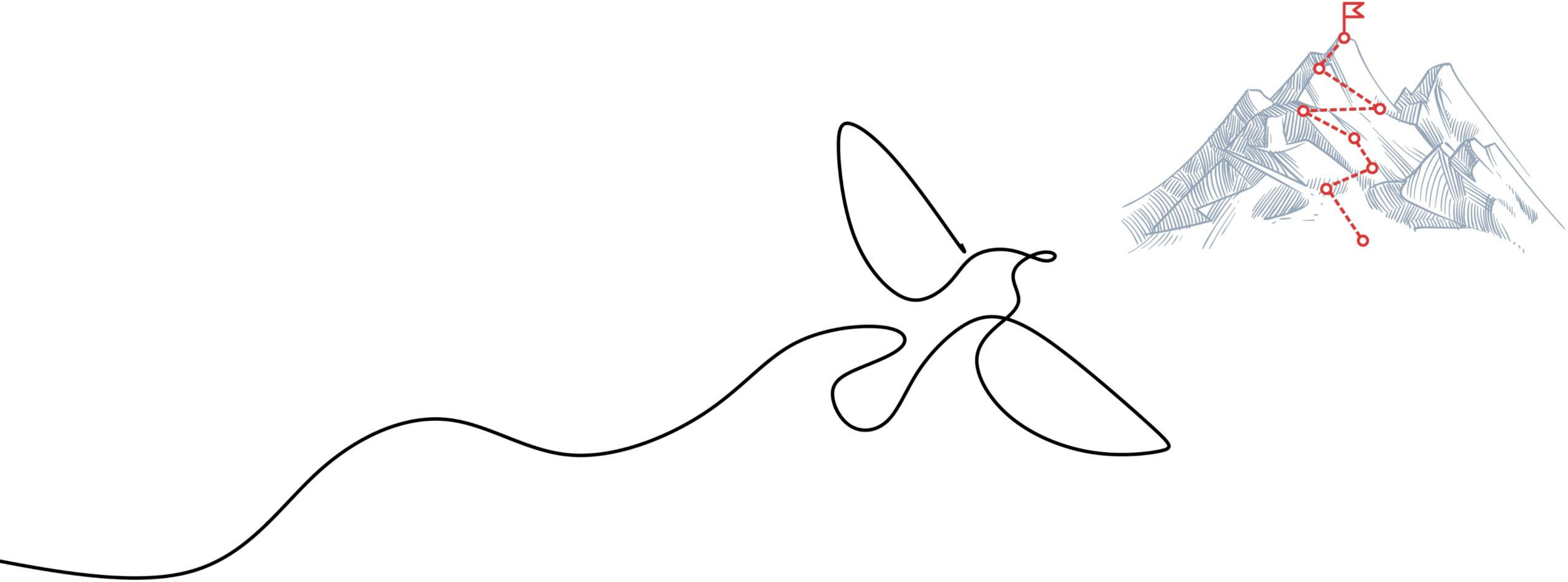


Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 3: Summary

Branch Completion

- **Branch if equal**—operations and the control signals involved
  - ALU is used to compare two registers;
    if they are equal, the branch is taken; otherwise, the branch is not taken
  - **ALUSrcA:** First ALU input is register A
  - **ALUSrcB:** Second ALU input is register B
  - **ALUOp:** ALU instructed to perform subtraction
  - Zero output asserted if A equals B
  - If Zero output is asserted
    - **PCWriteCond:** Update the PC
    - **PCSource:** The input of the PC is the output of the ALUOut register



Step 4

Memory Access or R-type Instruction Completion

- A load or store instruction accesses memory, or an arithmetic-logical instruction writes its result to the register file
  - Memory load
  - Memory store
  - R-type

Step 4

Memory Access

- *Recall:* A load or store instruction accesses memory, or an arithmetic-logical instruction writes its result to the register file
- **Memory load**
  - `Memory Data Register = Memory[ALUOut]`
  - Memory address comes from the ALUOut; memory read
- **Memory store**
  - `Memory[ALUOut] <= B`
  - Memory address comes from the ALUOut; memory write

Step 4

Memory Access

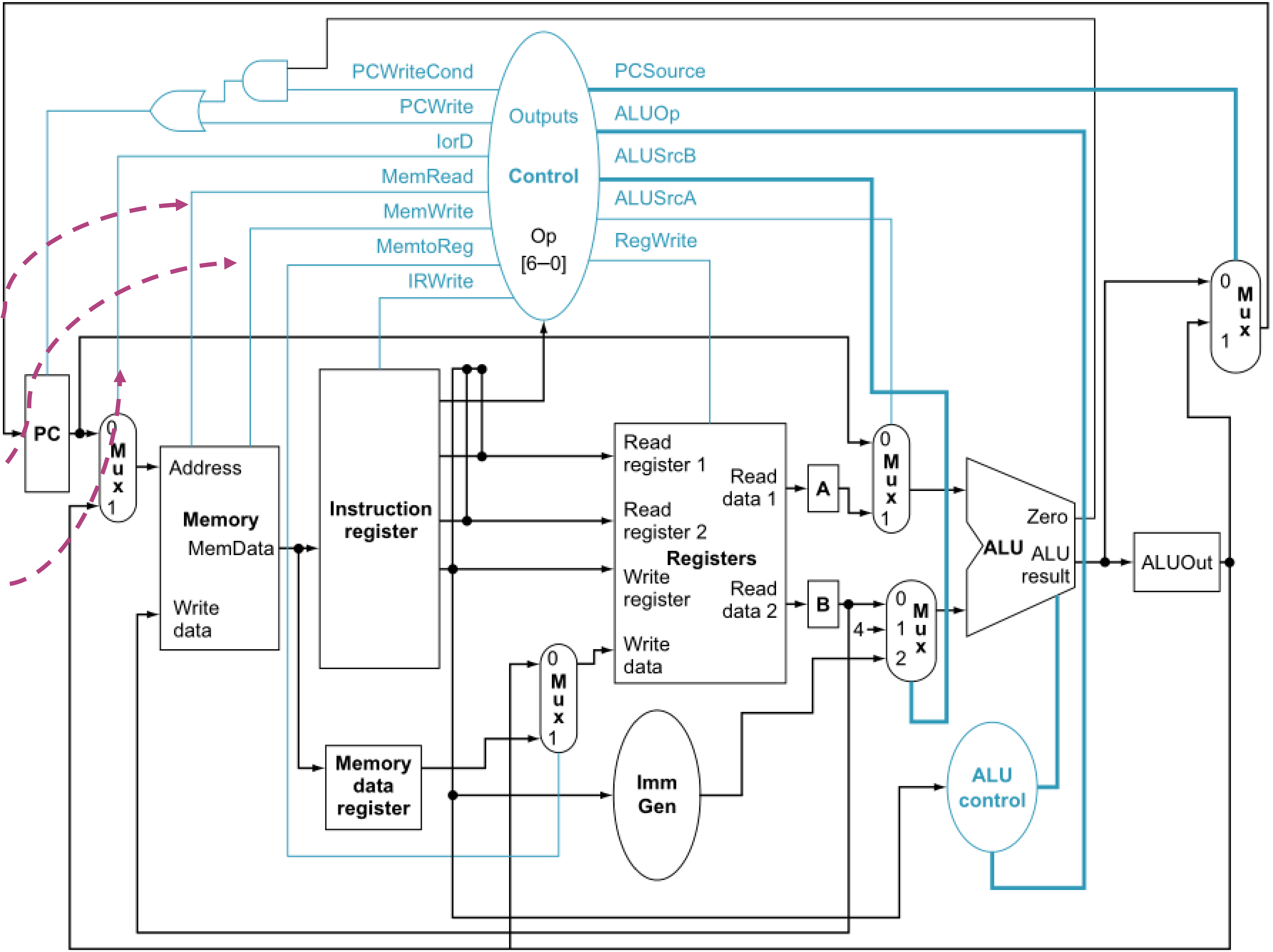
The address was computed during the previous cycle and stored in ALUOut

For memory load, assert MemRead

For memory store, assert MemWrite

Set IorD to 1 to force the address to come from the ALUout rather than from the PC

MDR is overwritten in every clock cycle (no harm in that)



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 4: Summary

Memory Access

- **Memory load**—operations and control signals involved
  - `Memory Data Register = Memory[ALUOut]`
  - IorD: Memory address comes from the ALUOut register rather than the PC
  - MemRead: Reading from the memory

- **Memory store**—operations and control signals involved
  - `Memory[ALUOut] <= B`
  - IorD: Memory address comes from the ALUOut register rather than the PC
  - MemWrite: Writing to the memory

Step 4

R-type Instruction Completion

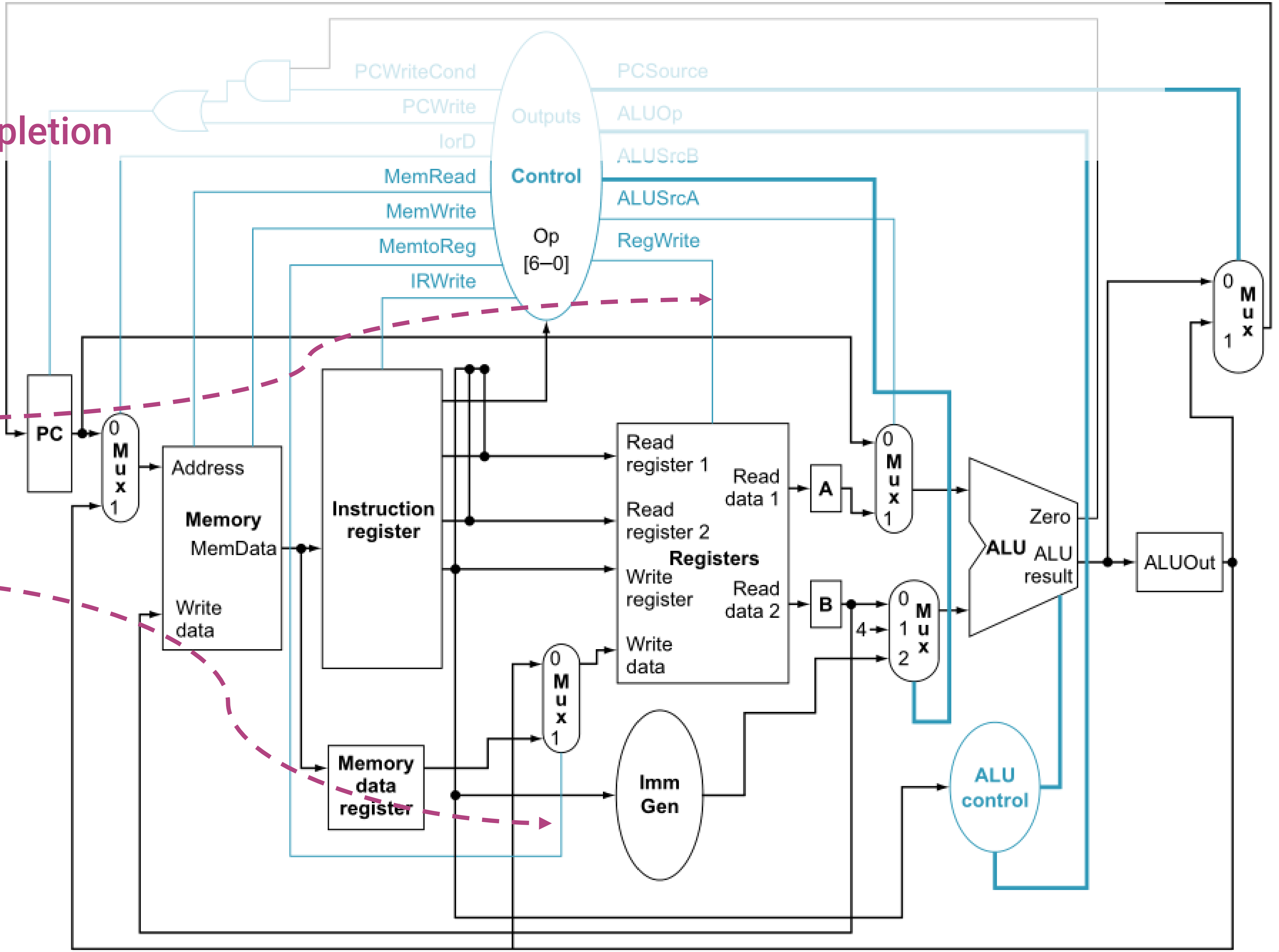
- *Recall:* A load or store instruction accesses memory, or an arithmetic-logical instruction writes its result to the register file
- **R-type**
  - `Reg[Instruction Register[11:7]] <= ALUOut`
  - Copy ALUOut to the register file

Step 4

R-type Instruction Completion

Assert RegWrite to write to the register file

Set MemtoReg to zero, so that the output of the ALUOut is written into the register file, as opposed to the MDR

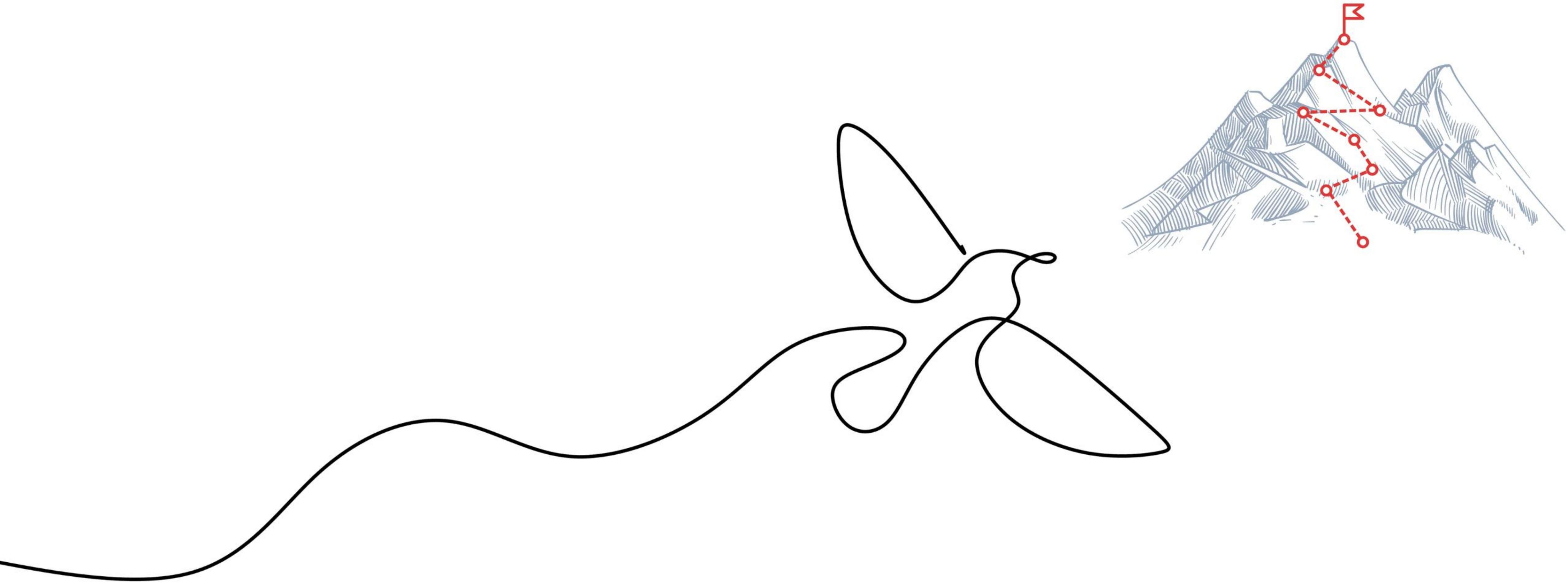


Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 4: Summary

R-type Instruction Completion

- **R-type**—operations and control signals involved
  - `Reg[Instruction Register[11:7]] <= ALUOut`
  - Use Instruction Register [11:7] as the index of the register (in the register file) to write to
  - **RegWrite:** Write to the register file
  - **MemtoReg:** The value from the ALUOut register and not the value from the Memory Data Register is to be written to the register file



Step 5

Memory Read Completion Step

- **Memory Load** (read) completes by writing the value from the memory data register to the register file
  - `Reg[Instruction Register[11:7]] <= Memory Data Register`
  - Write the data, which was placed in the Memory Data Register in the previous cycle, into the register file

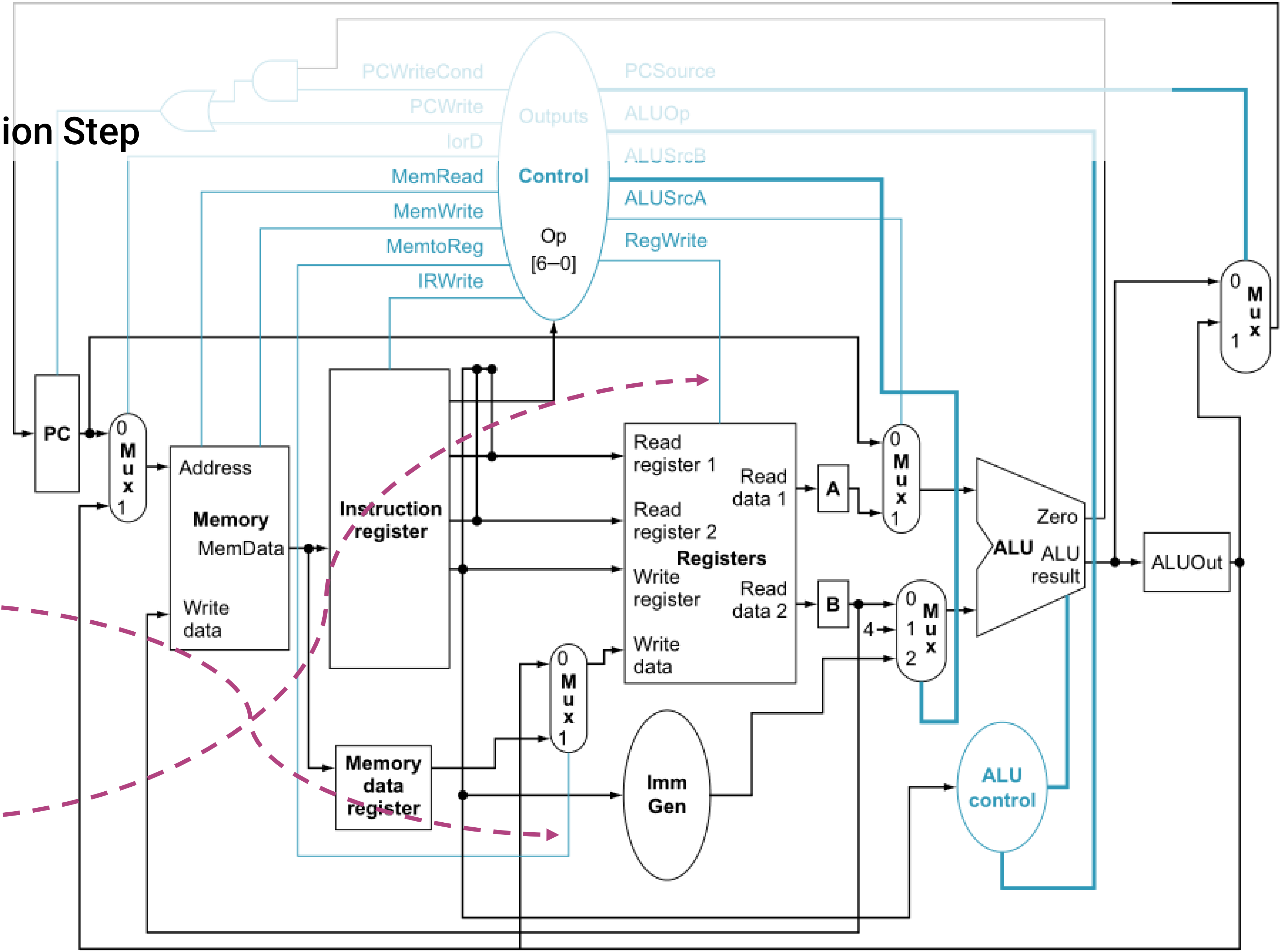
Step 5

Memory Read Completion Step

Write the load data, which was stored in MDR in the previous cycle, into the register file

Set **MemtoReg** to 1, to write the data loaded from memory instead of from the ALUOut

Assert **RegWrite** to cause a write to the register file

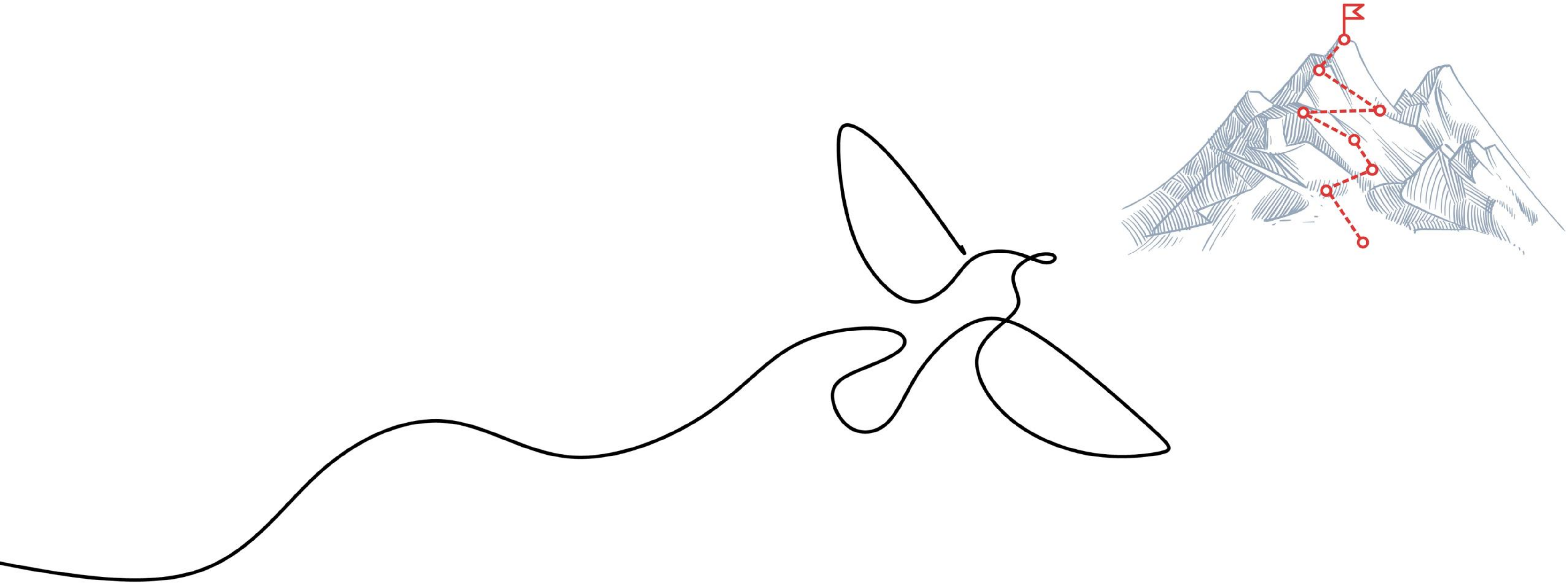


Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Step 5: Summary

Memory Read Completion Step

- Operations and control signals involved:
  - **RegWrite:** Write to the register file
  - **MemtoReg:** The data to write is the value from the Memory Data Register and not the value from the ALUOut register



Done Breaking the Instruction Execution…

…into Clock (CPU) Cycles

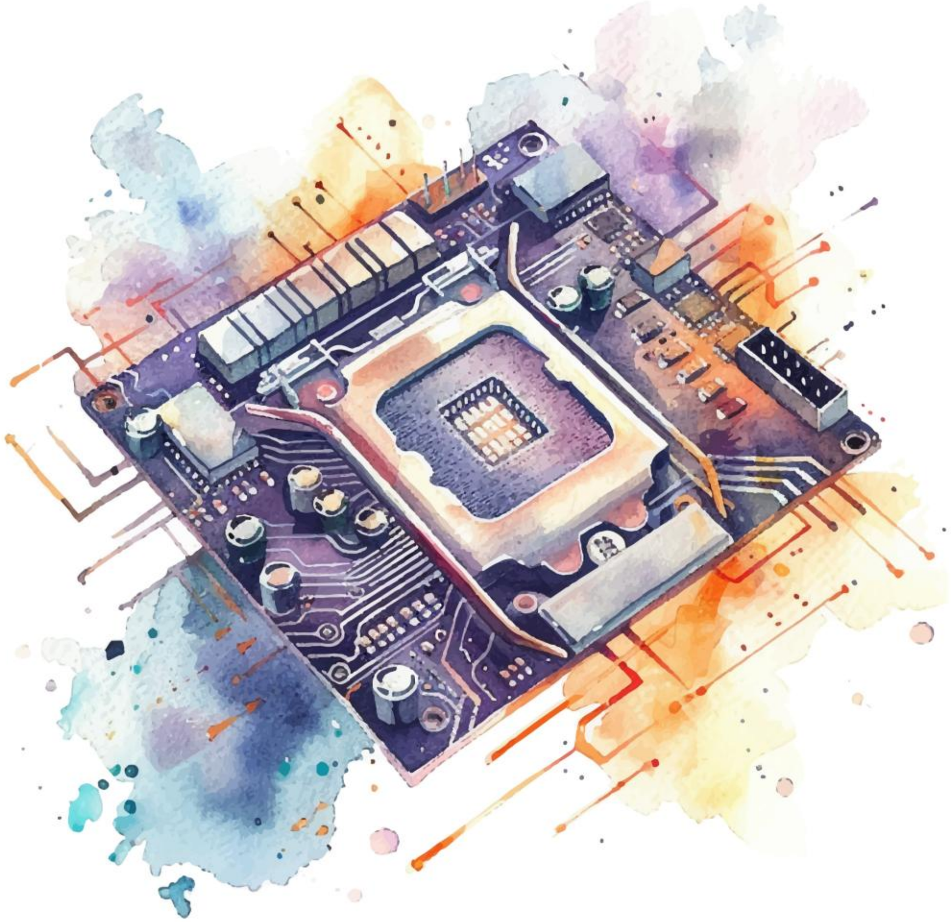
Five steps

- Instruction fetch
- Instruction decode and register (operand) fetch
- Execution, memory address computation, or branch completion
- Memory access or R-type instruction completion
- Memory read completion

What is next?

Defining the Control: Finite State Machine

Multicycle CPU



© EnelEva / Adobe Stock

Recall: Breaking the Instruction Execution…

…into Clock Cycles

- Now that we have determined what the control signals are and when they must be asserted, we can implement the control unit
- The control for the multicycle CPU must specify both the signals to be set in any step (cycle) and the next step in the sequence

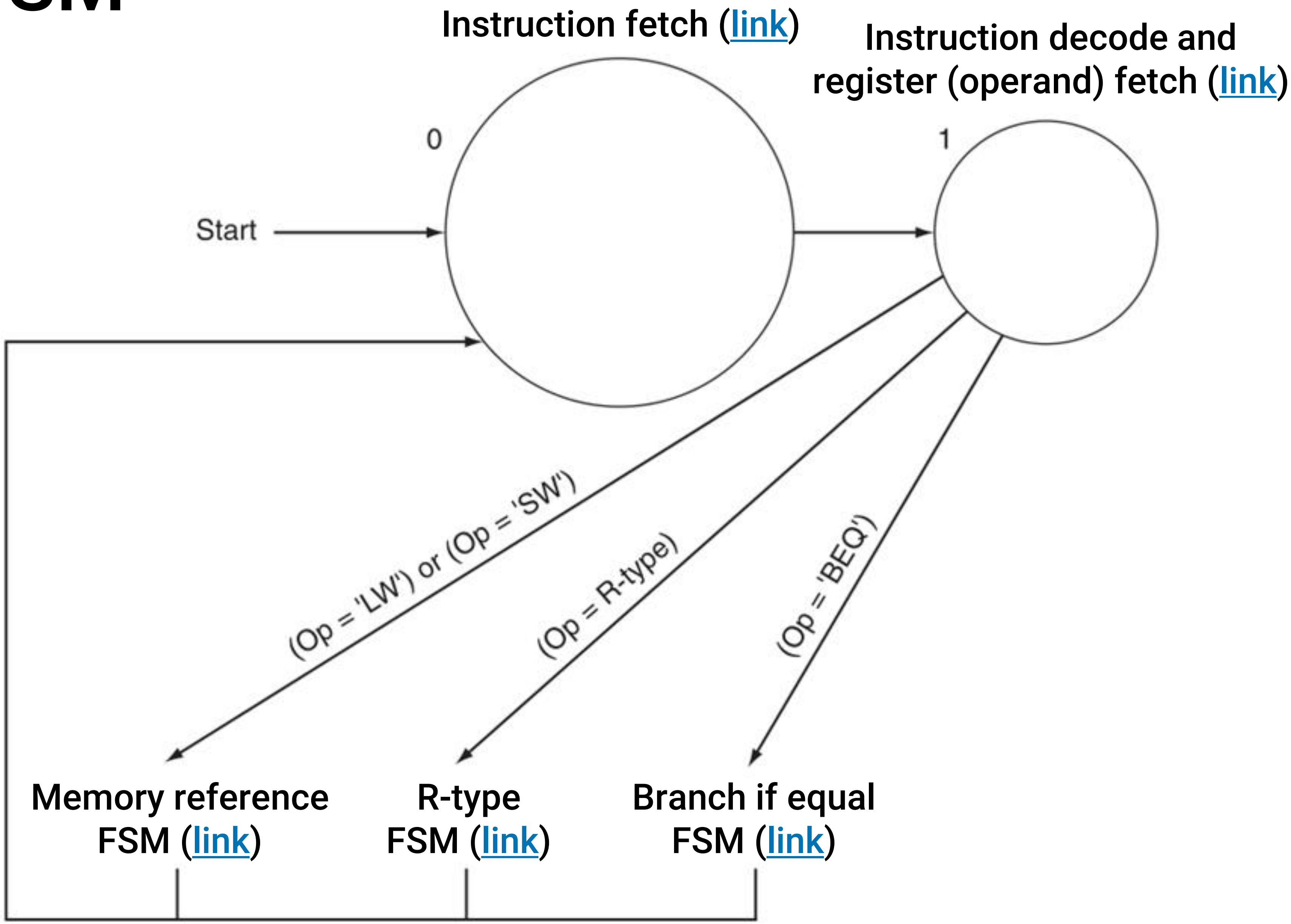
Control FSM

High-level view

Labels on the arcs are conditions tested to determine which state is the next state.

When the next state is unconditional, no label is given.

Labels inside nodes indicate the output signals asserted during that state.



Instruction fetch (link)

Instruction decode and register (operand) fetch (link)

Memory reference FSM (link)

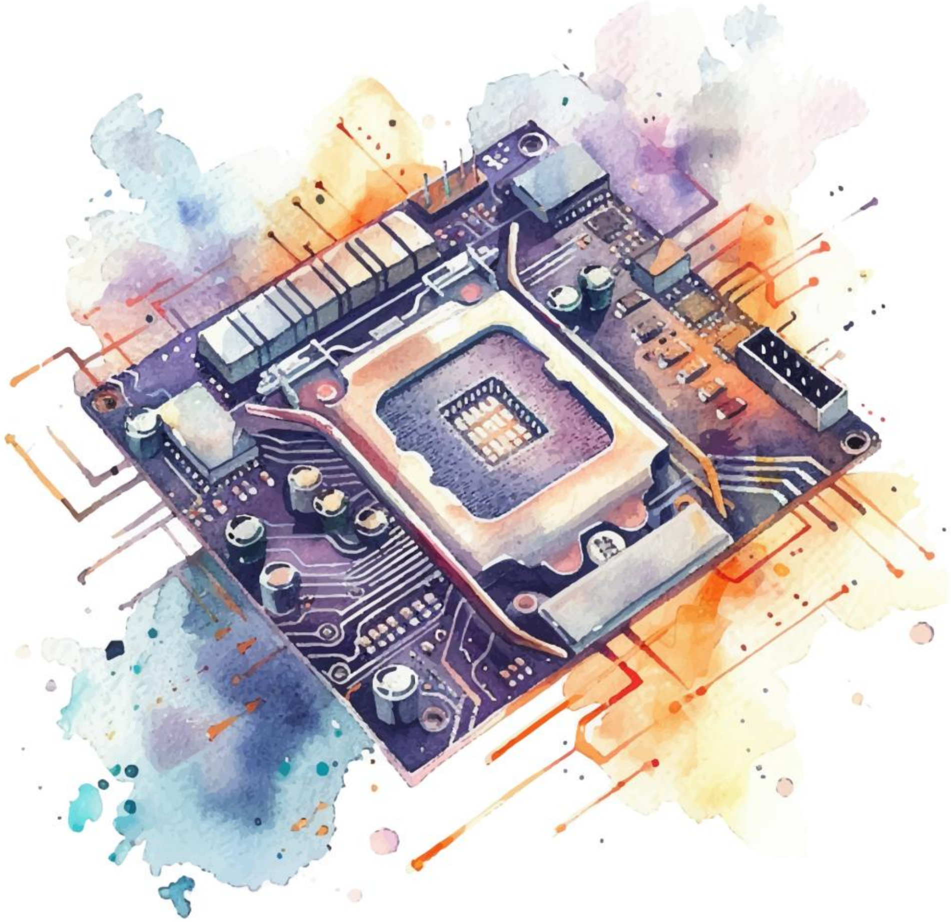
R-type FSM (link)

Branch if equal FSM (link)

Control FSM

Outline

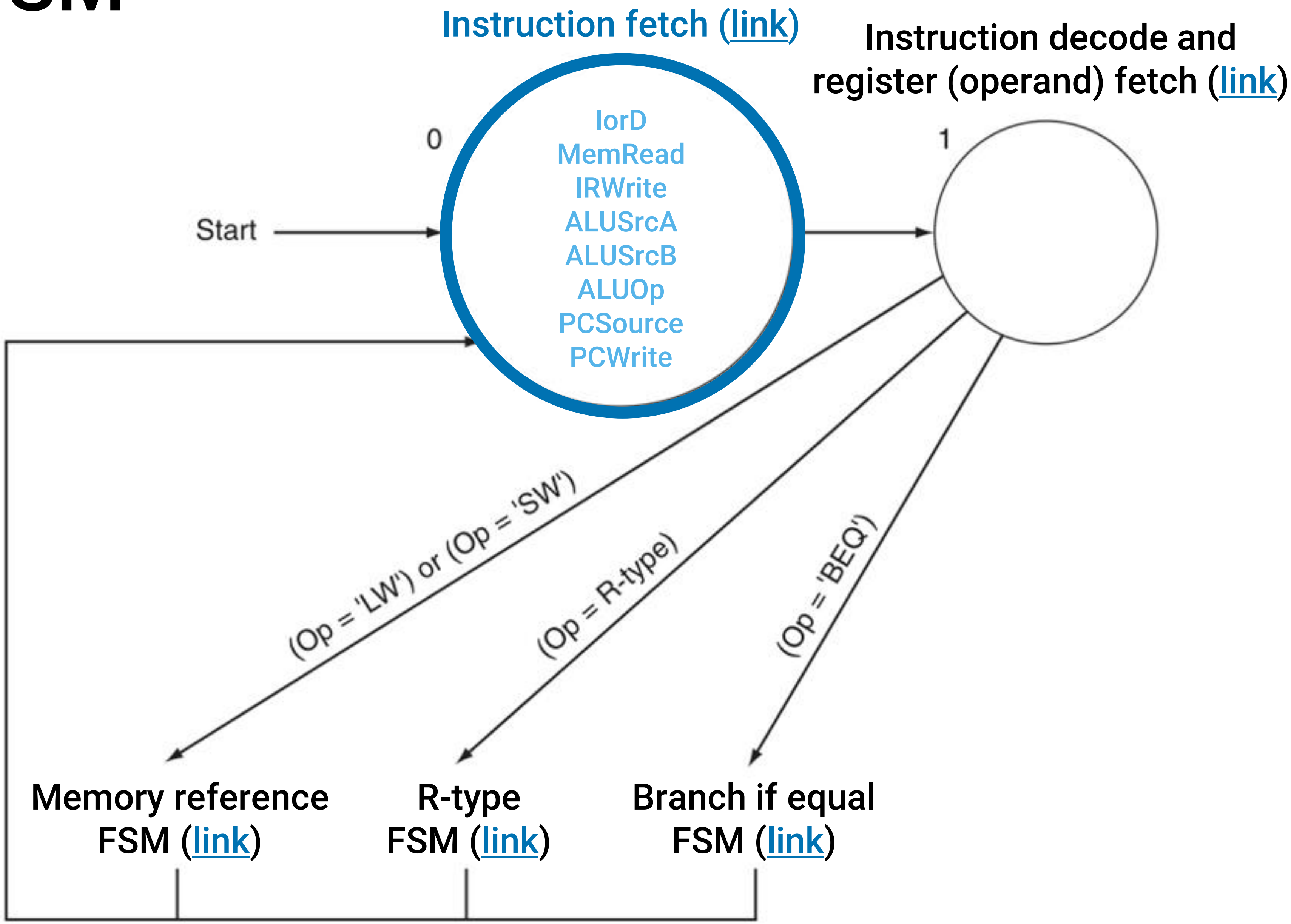
- **Instruction fetch**
    - Identical for all instructions
- Instruction decode and operand fetch
- Memory reference FSM
- R-type FSM
- Branch if equal FSM



© EnelEva / Adobe Stock

Control FSM

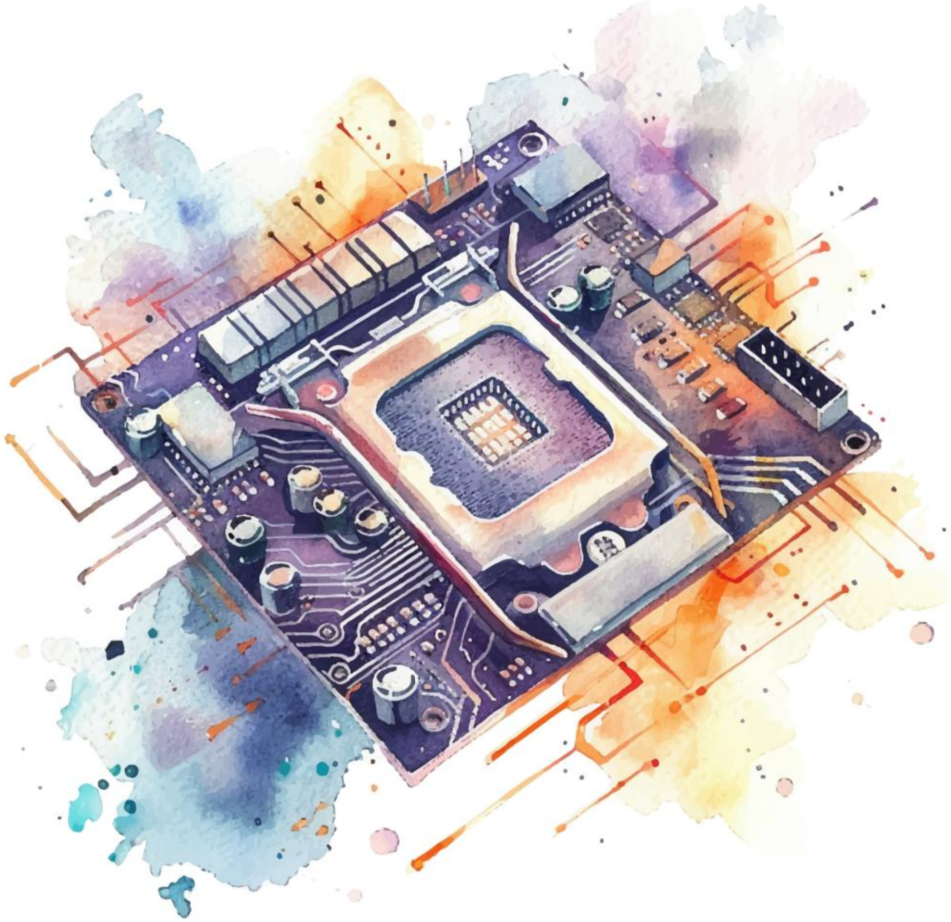
Instruction Fetch


Instruction fetch (link)
Instruction decode and register (operand) fetch (link)
0
IorD
MemRead
IRWrite
ALUSrcA
ALUSrcB
ALUOp
PCSource
PCWrite
1
Start
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
Memory reference FSM (link)
R-type FSM (link)
Branch if equal FSM (link)

Control FSM

Outline

- Instruction fetch
- **Instruction decode and operand fetch**
  - Identical for all instructions
- Memory reference FSM
- R-type FSM
- Branch if equal FSM

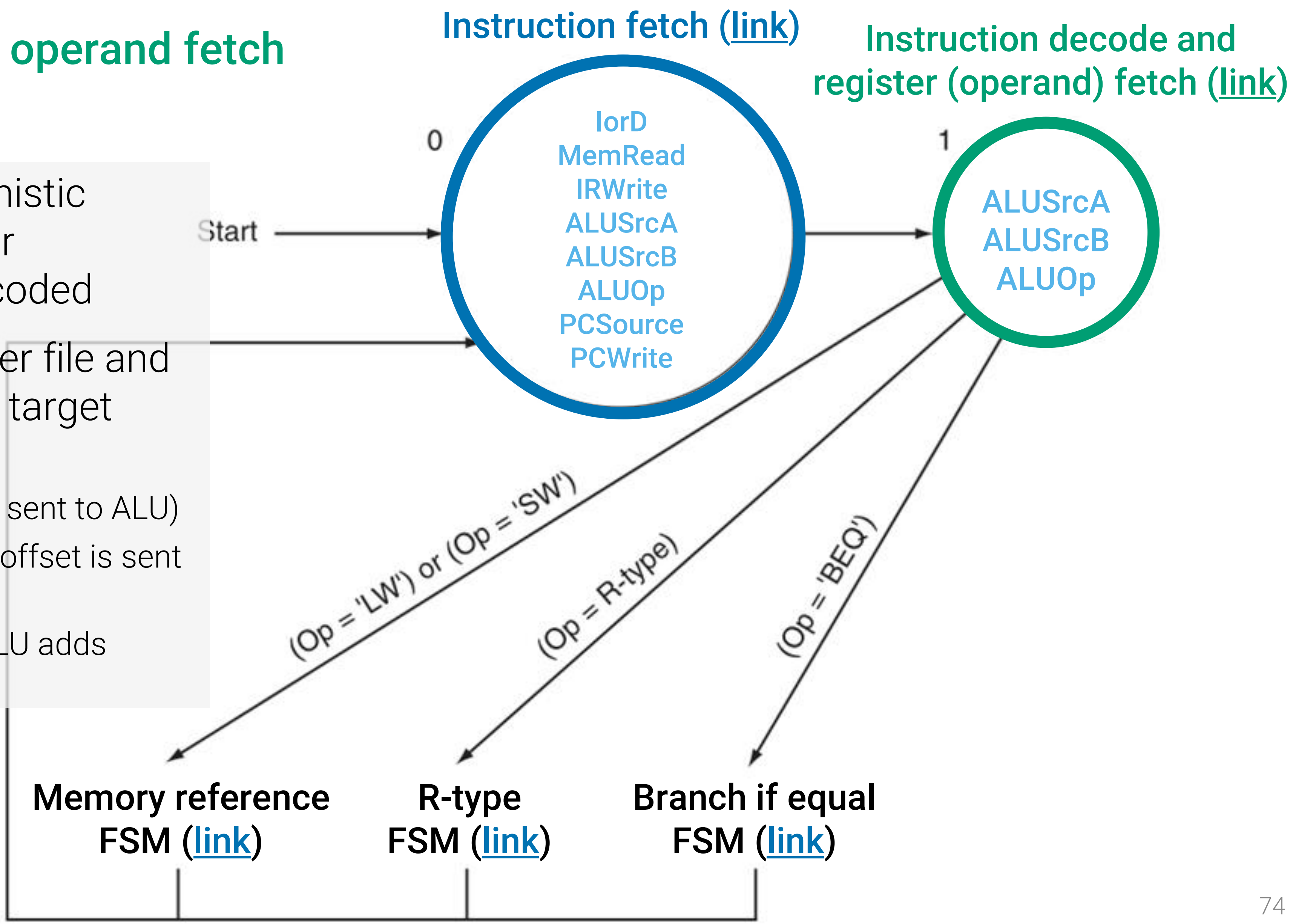


© EnelEva / Adobe Stock

Control FSM

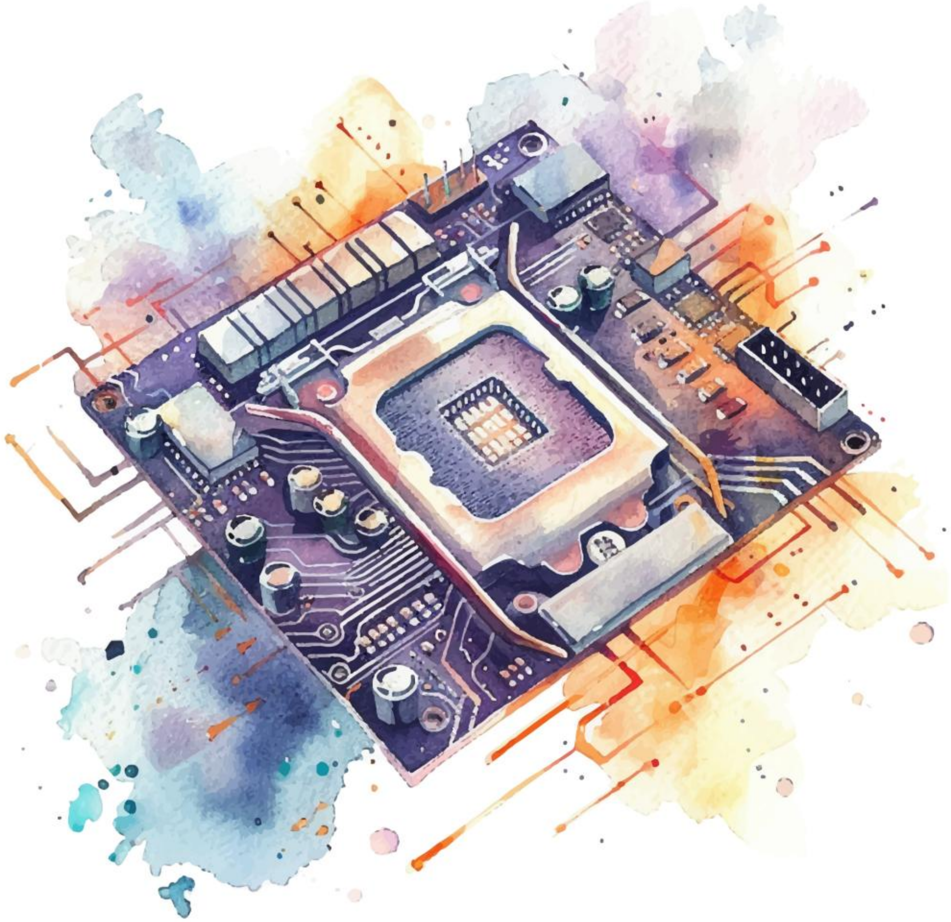
Instruction decode and operand fetch

- *Recall:* Performing optimistic actions, while waiting for the instruction to be decoded
- (1) Read from the register file and (2) Compute the branch target address
  - Set **ALUSrcA** to zero (PC sent to ALU)
  - Set **ALUSrcB** so that the offset is sent to the ALU
  - Set **ALUOp** so that the ALU adds



Control FSM
Outline

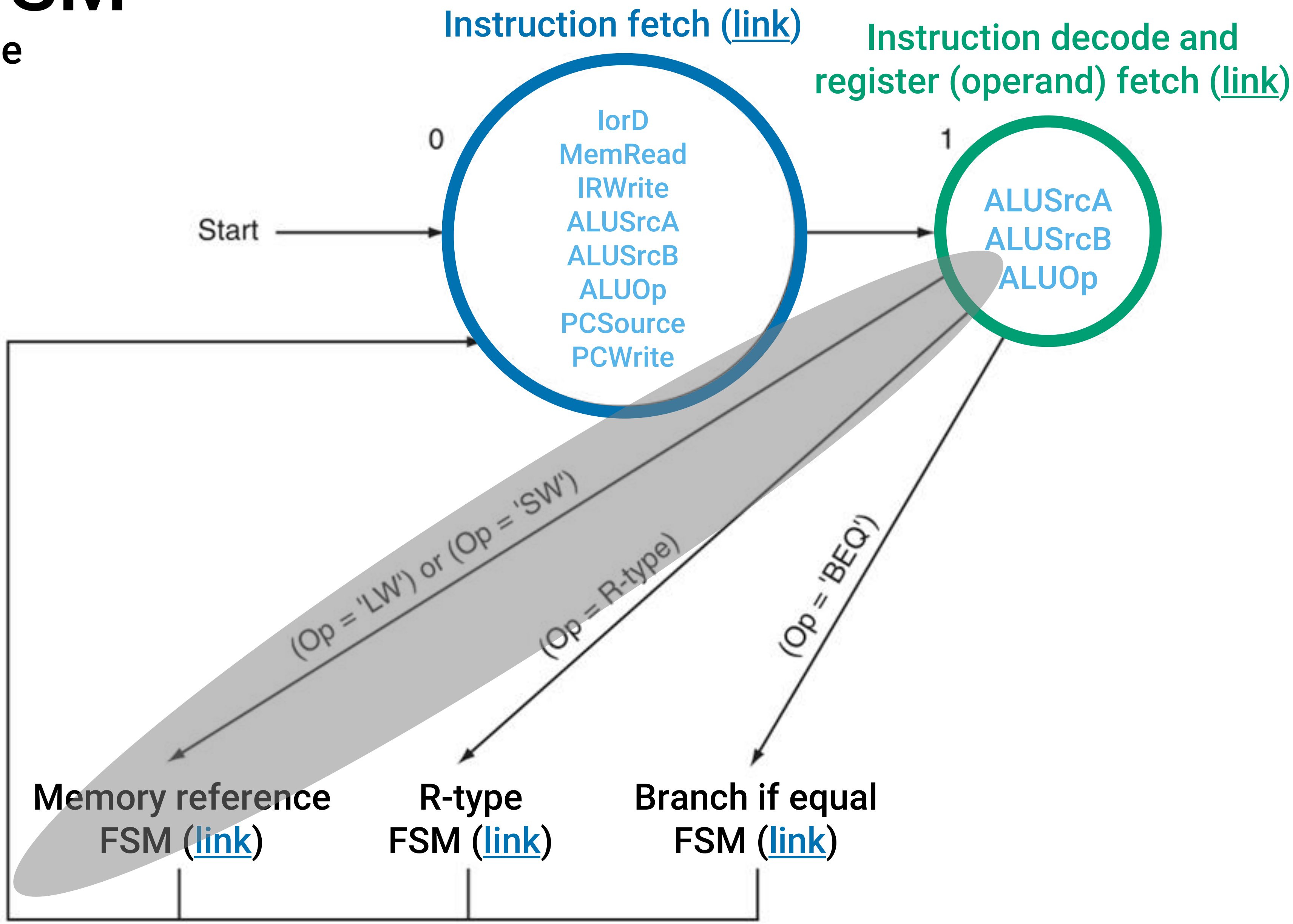
- Instruction fetch
- Instruction decode and register fetch
- **Memory reference FSM**
- R-type FSM
- Branch if equal FSM



© EnelEva / Adobe Stock

Control FSM

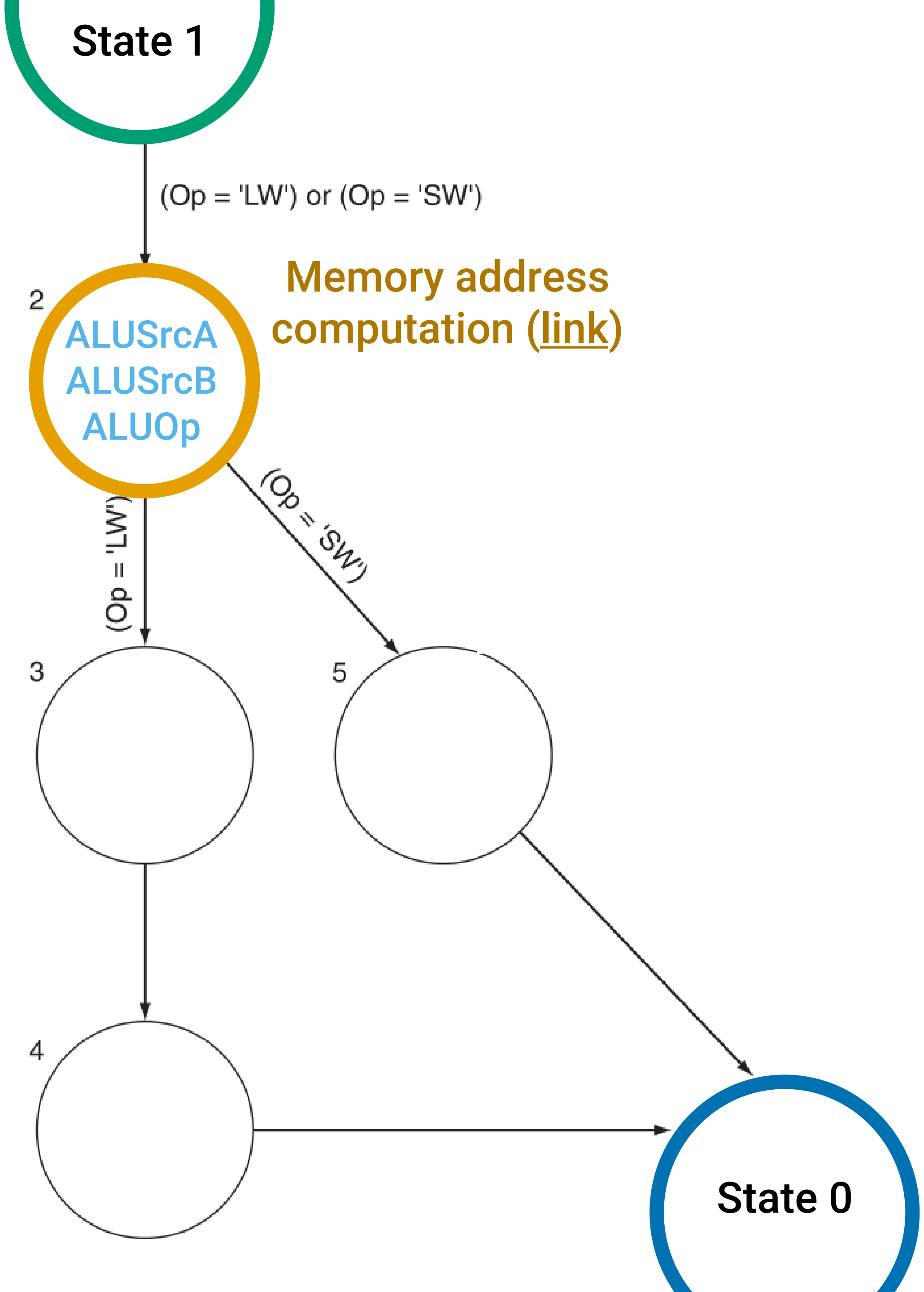
Memory Reference



Memory Reference FSM

Address computation

- *Recall:* ALU adding the operands to form the memory address
  - Set **ALUSrcA** to 1, so that register A is the first ALU input
  - Set **ALUSrcB** so that the output of the offset generation unit is the second ALU input
  - Set **ALUOp** so that the ALU adds

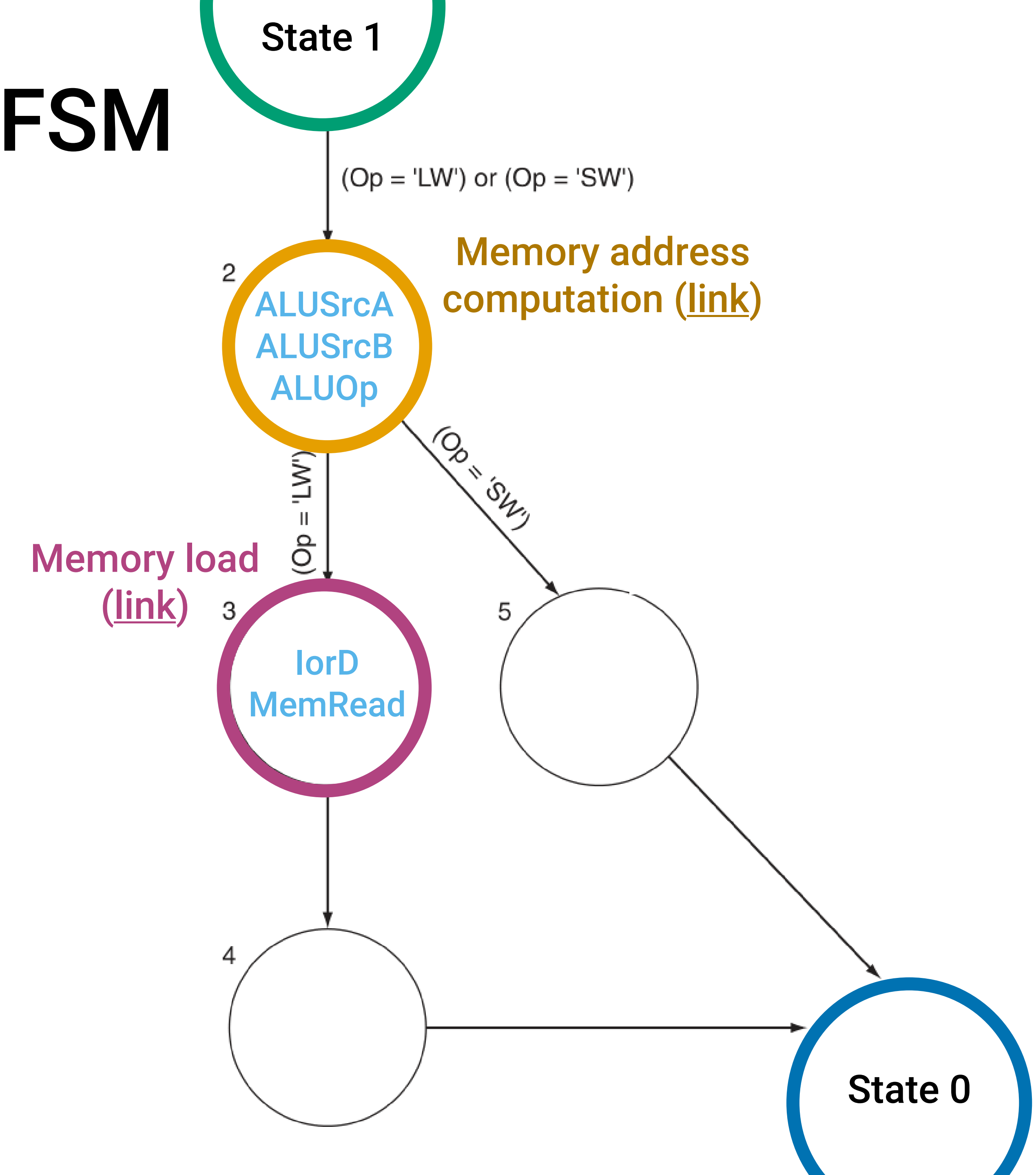


Memory address computation (link)

Memory Reference FSM

Load or Store

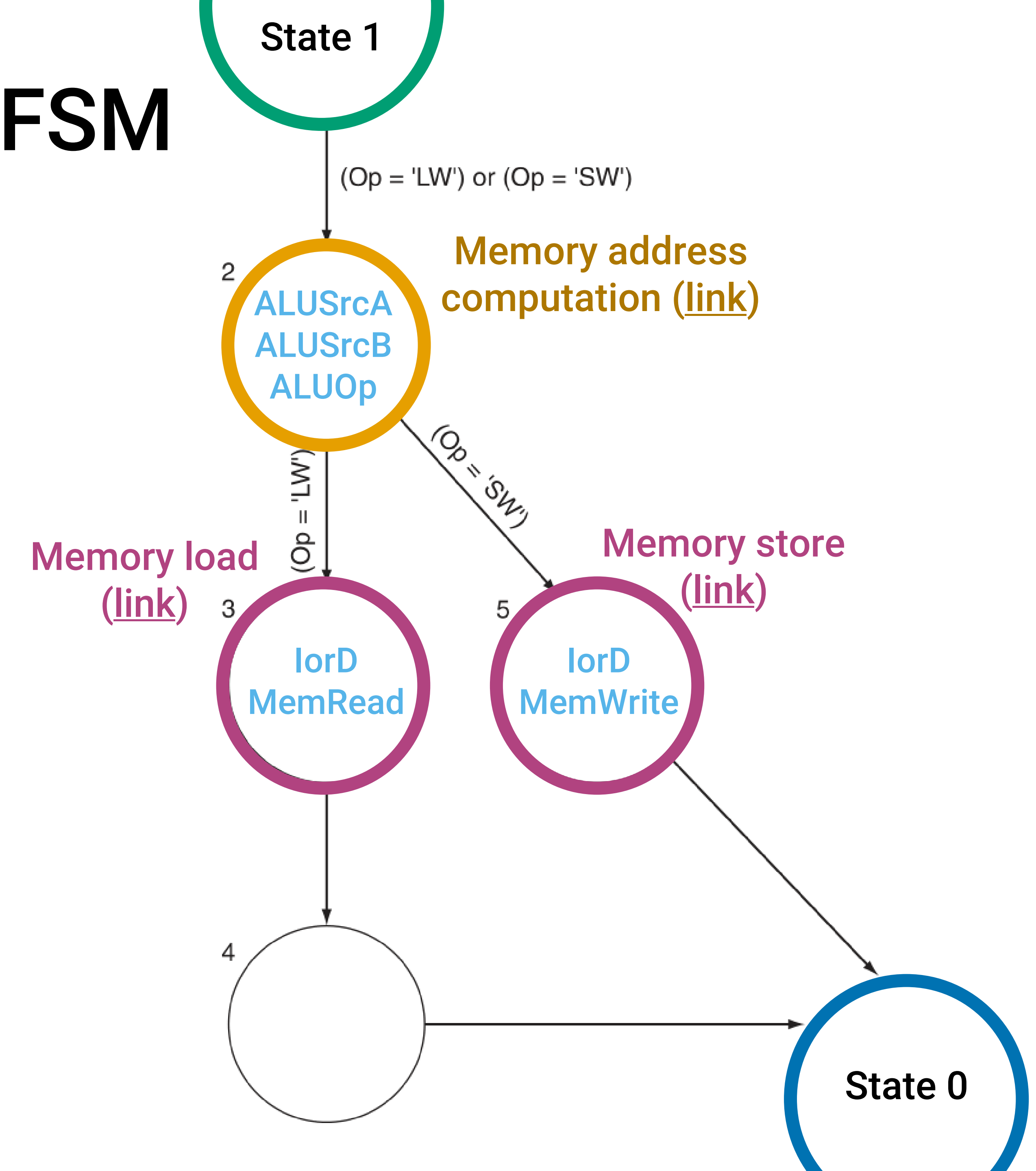
- *Recall:* Memory address was computed in the previous cycle and saved in ALUOut
  - For memory load, assert MemRead
  - Set IorD to 1 to force the address to come from the ALUout rather than from the PC



Memory Reference FSM

Load or Store

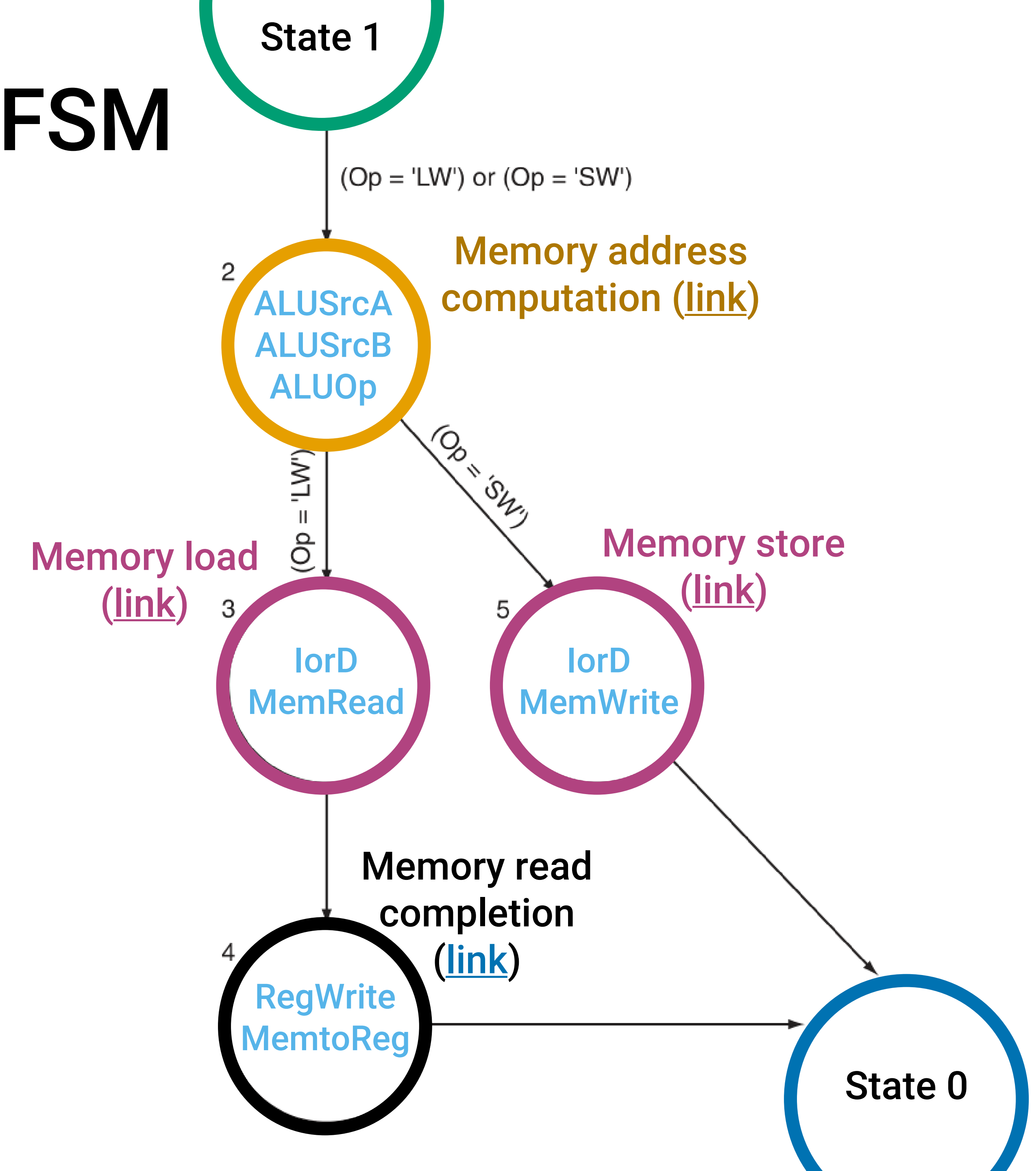
- *Recall:* Memory address was computed in the previous cycle and stored in ALUOut
    - For memory store, assert MemWrite
    - Set IorD to 1 to force the address to come from the ALUout rather than from the PC



Memory Reference FSM

Memory load completion

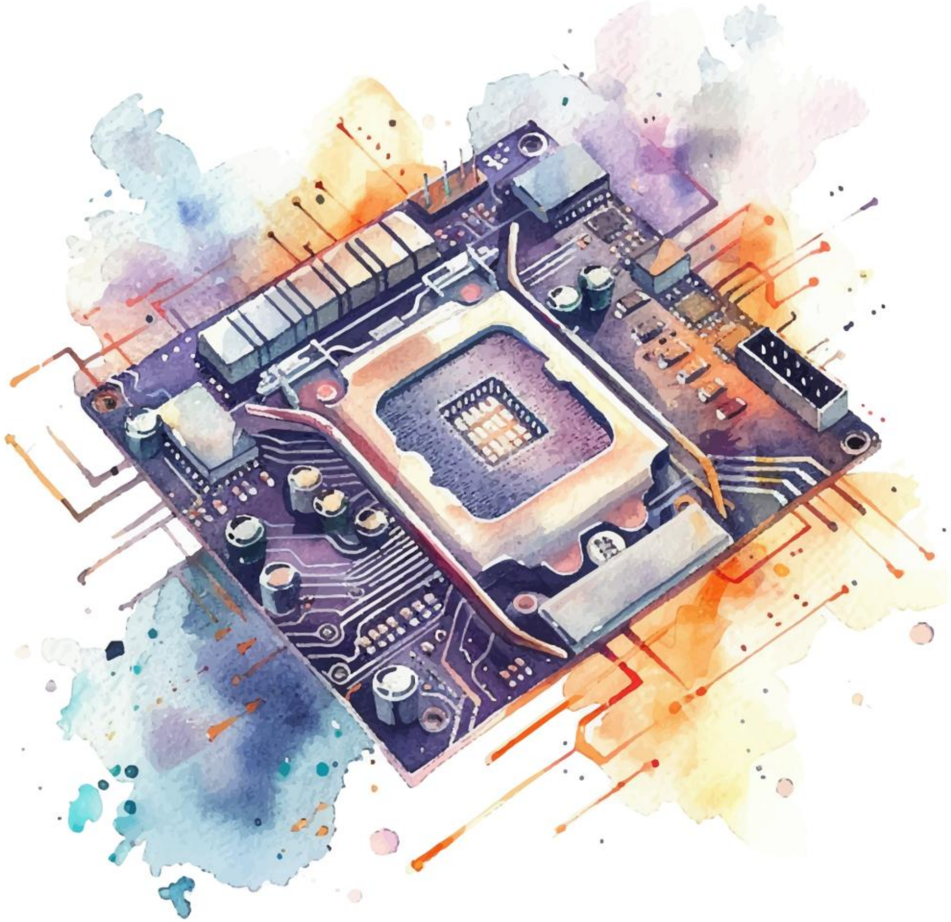
- *Recall:* Write the load data, which was stored in MDR in the previous cycle, into the register file
  - Set MemtoReg to 1, to write the data loaded from memory instead of from the ALUOut
  - Assert RegWrite to cause a write to the register file



Control FSM

Outline

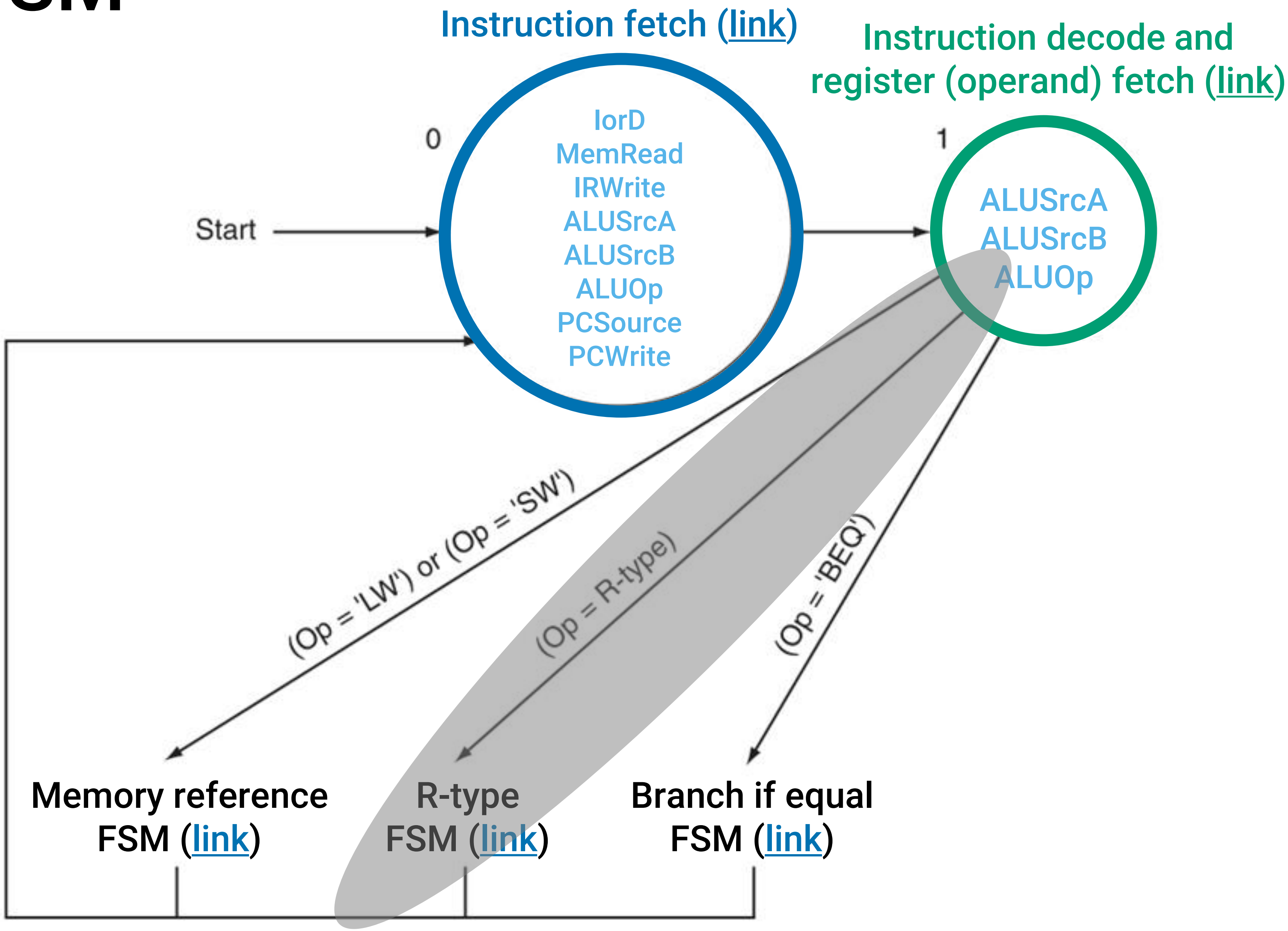
- Instruction fetch
- Instruction decode and register fetch
- Memory reference FSM
- **R-type FSM**
- Branch if equal FSM



© EnelEva / Adobe Stock

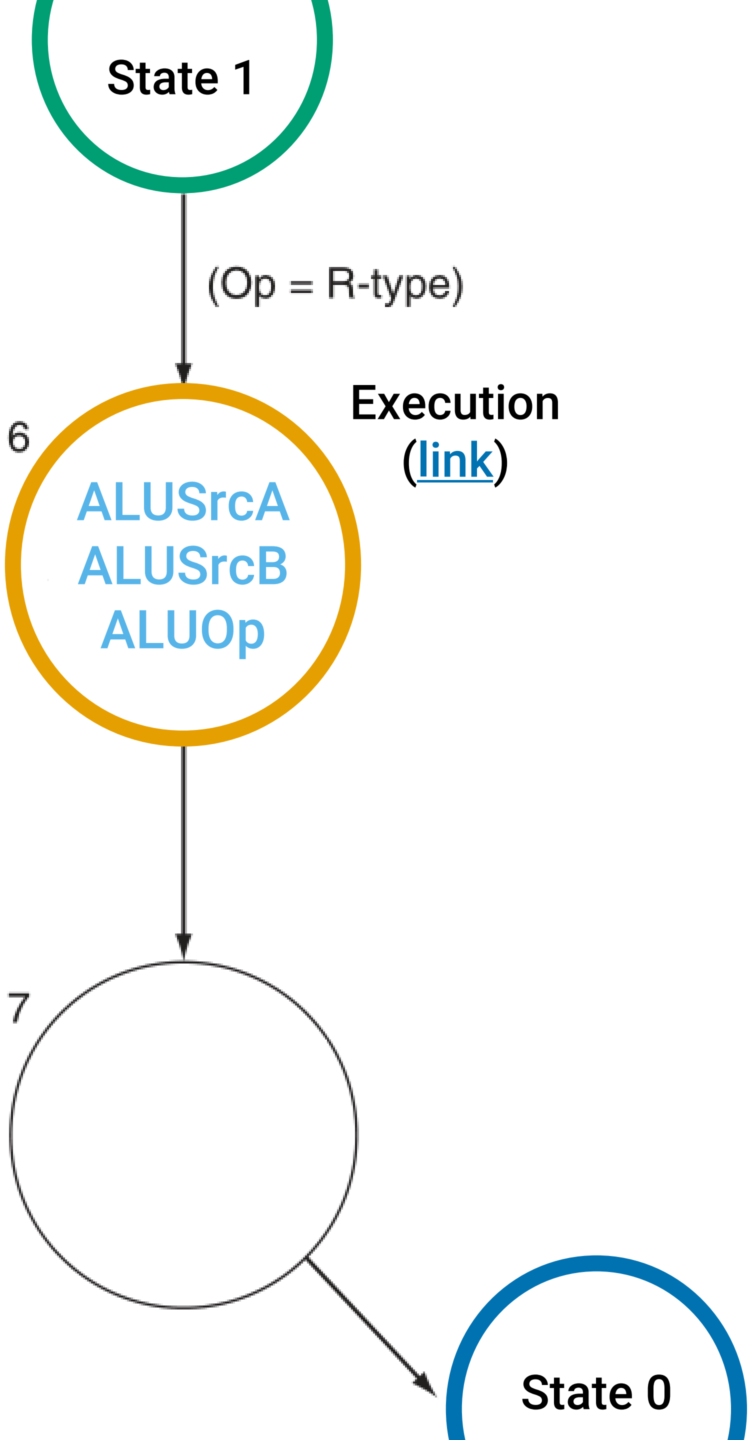
Control FSM

R-type



R-type FSM

- *Recall:* ALU is performing the operation on two values read from the register file in the previous cycle
  - Set ALUSrcA to 1 and ALUSrcB so that both ALU operands are from the register file
  - Set ALUOp so that the `funct` field from the instruction register is used to determine the ALU operation (`add, sub, and, or`)

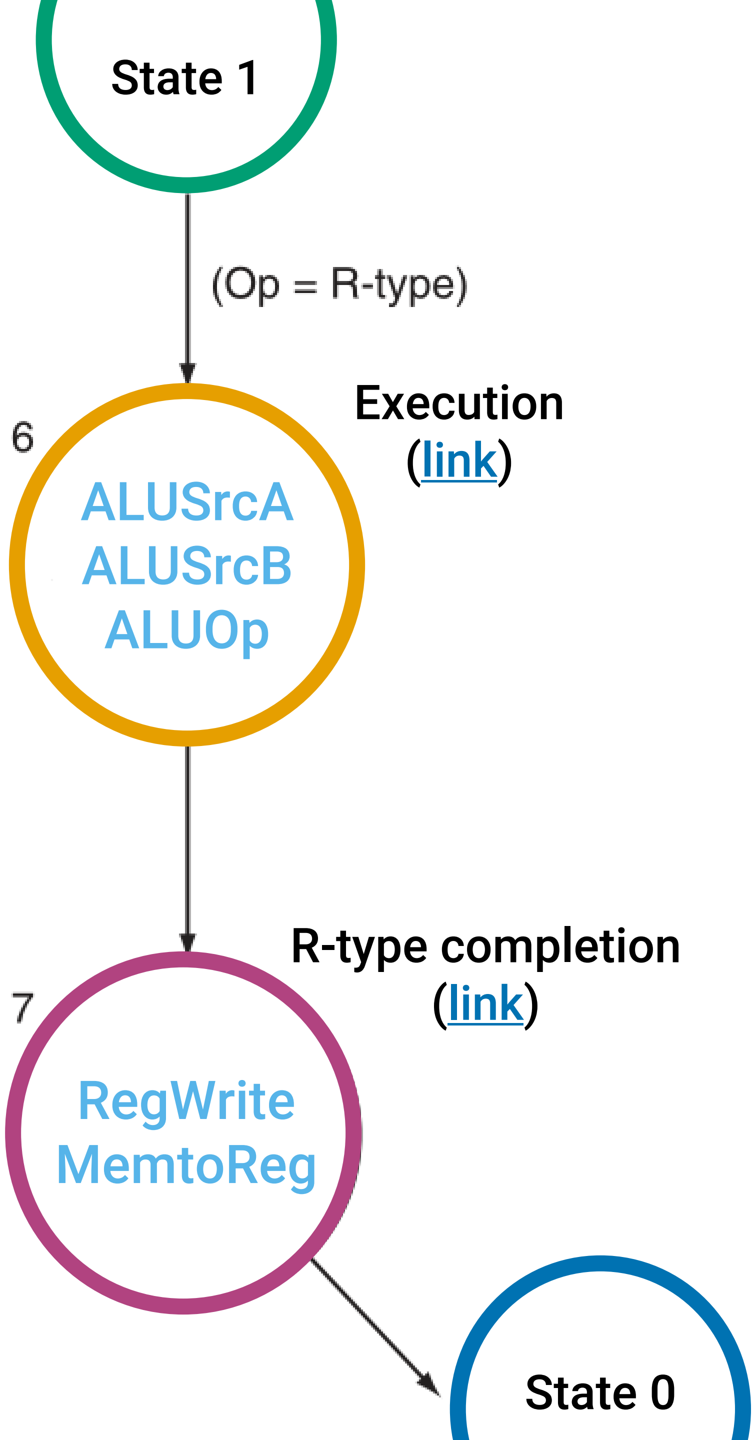


Execution (link)

R-type FSM

Final

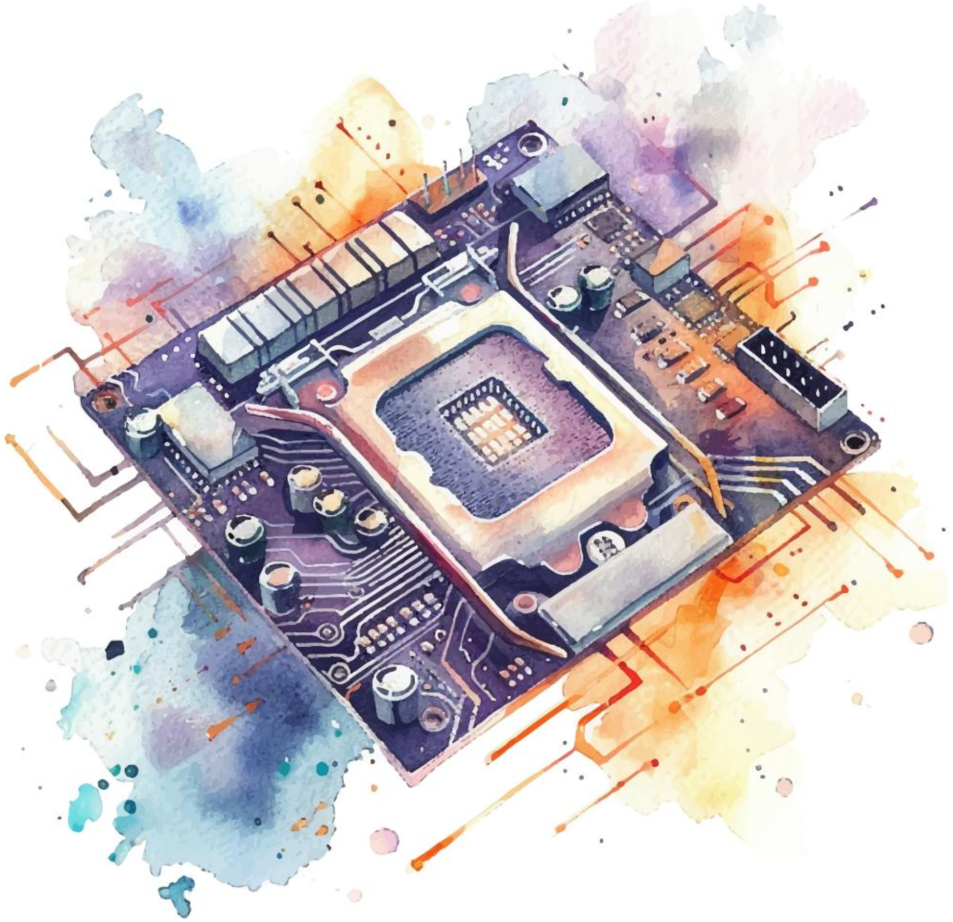
- *Recall:* An arithmetic-logical instruction writes its result to the register file
  - Assert **RegWrite** to write to the register file
  - Set **MemtoReg** to zero, so that the output of the ALUOut is written into the register file, as opposed to the MDR



Control FSM

Outline

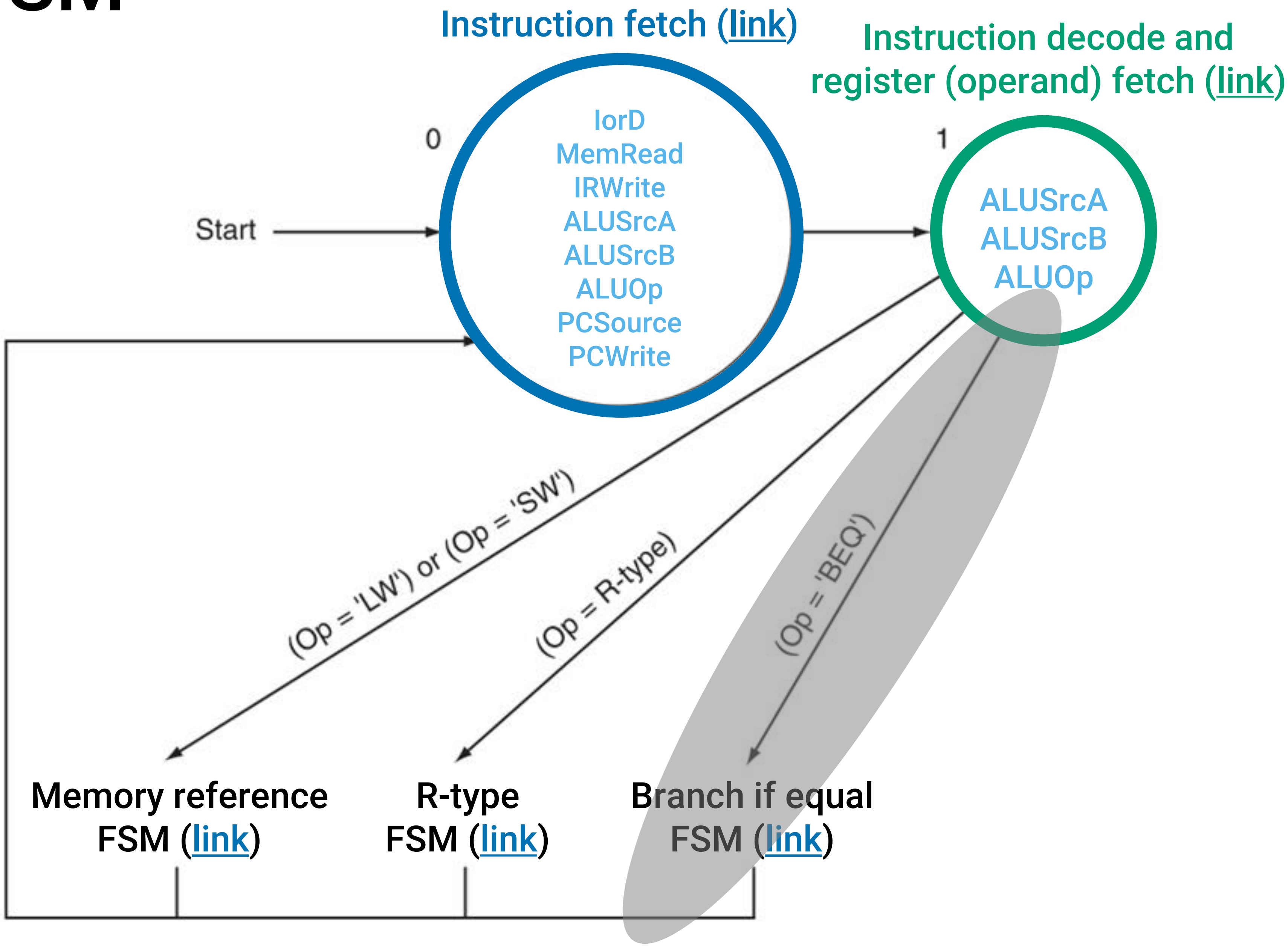
- Instruction fetch
- Instruction decode and register fetch
- Memory reference FSM
- R-type FSM
- **Branch if equal FSM**



© EnelEva / Adobe Stock

Control FSM

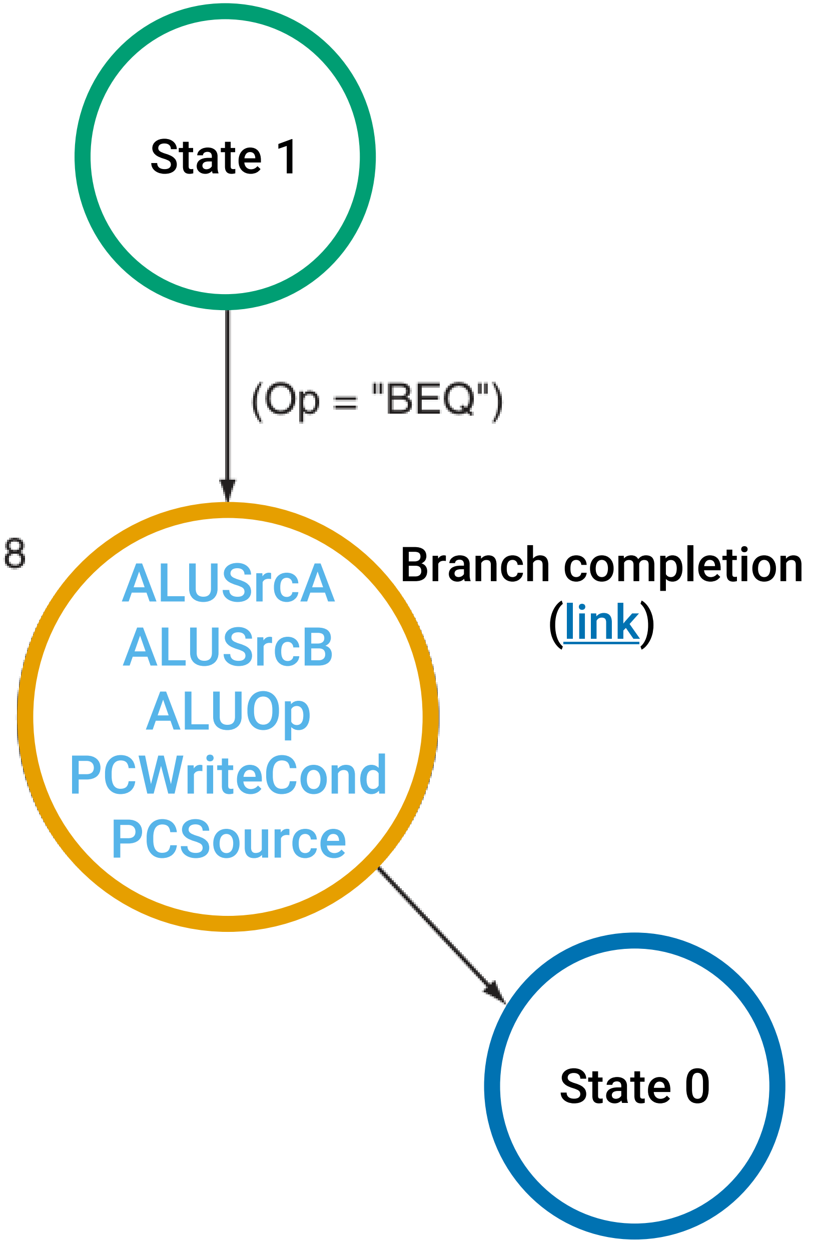
Branch if Equal



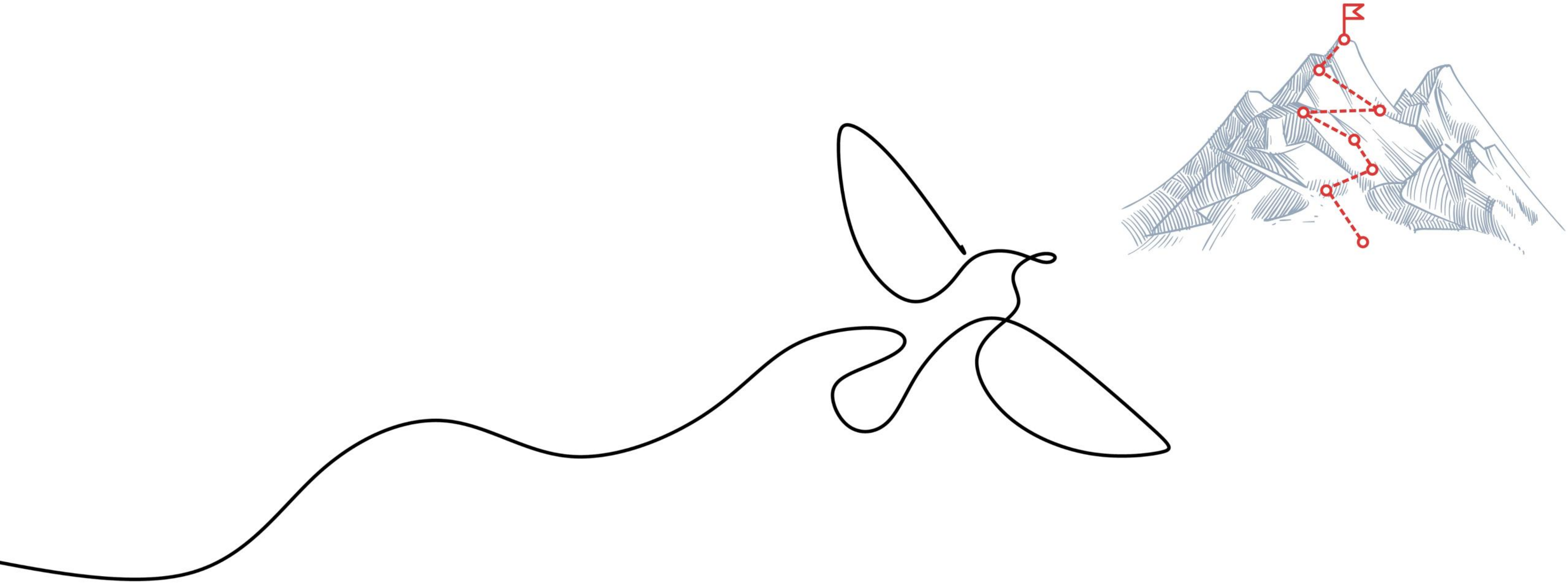
Branch FSM

Final

- *Recall:* ALU operates on the operands prepared in the previous cycle, performing a function depending on the instruction class
  - Set **ALUSrcA** to 1 and **ALUSrcB** so that both ALU operands are from the register file
  - Set **ALUOp** so that ALU subtracts
  - Assert **PCWriteCond** to conditionally update the PC if the Zero output of the ALU is asserted
  - Set **PCSource** to 1 so the value written into PC comes from ALUOut, which holds the branch target address computed in the previous cycle

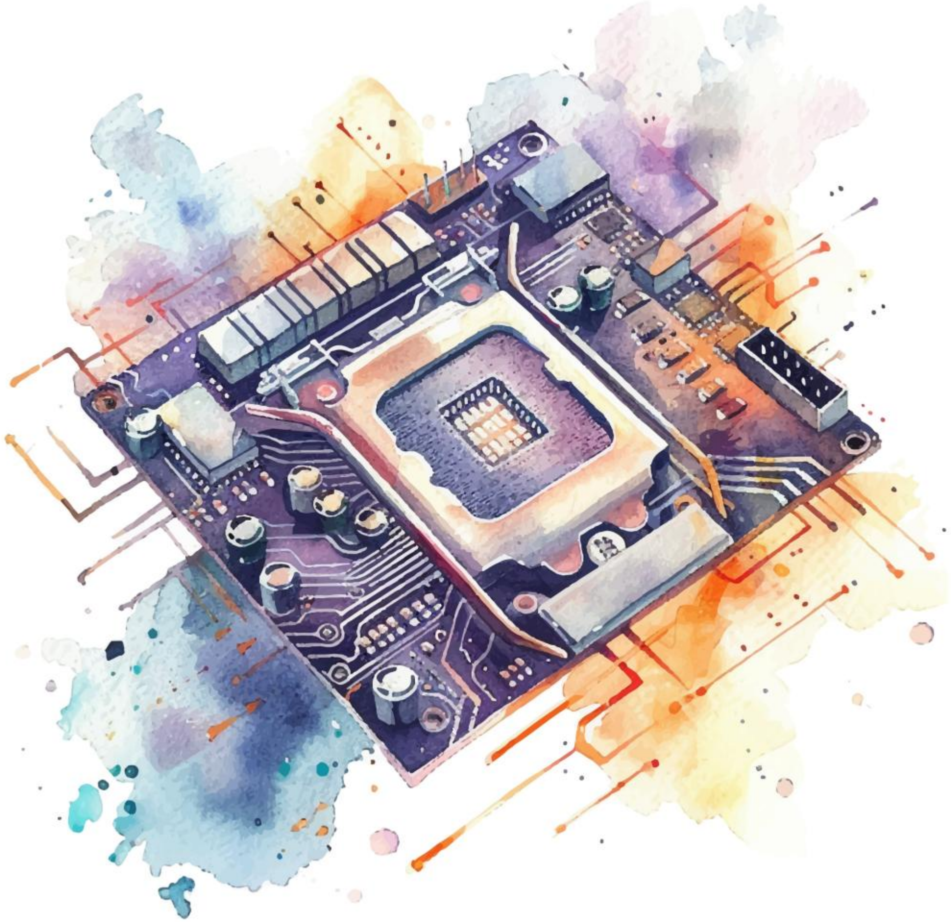


Branch completion
(link)



Complete Finite State Machine

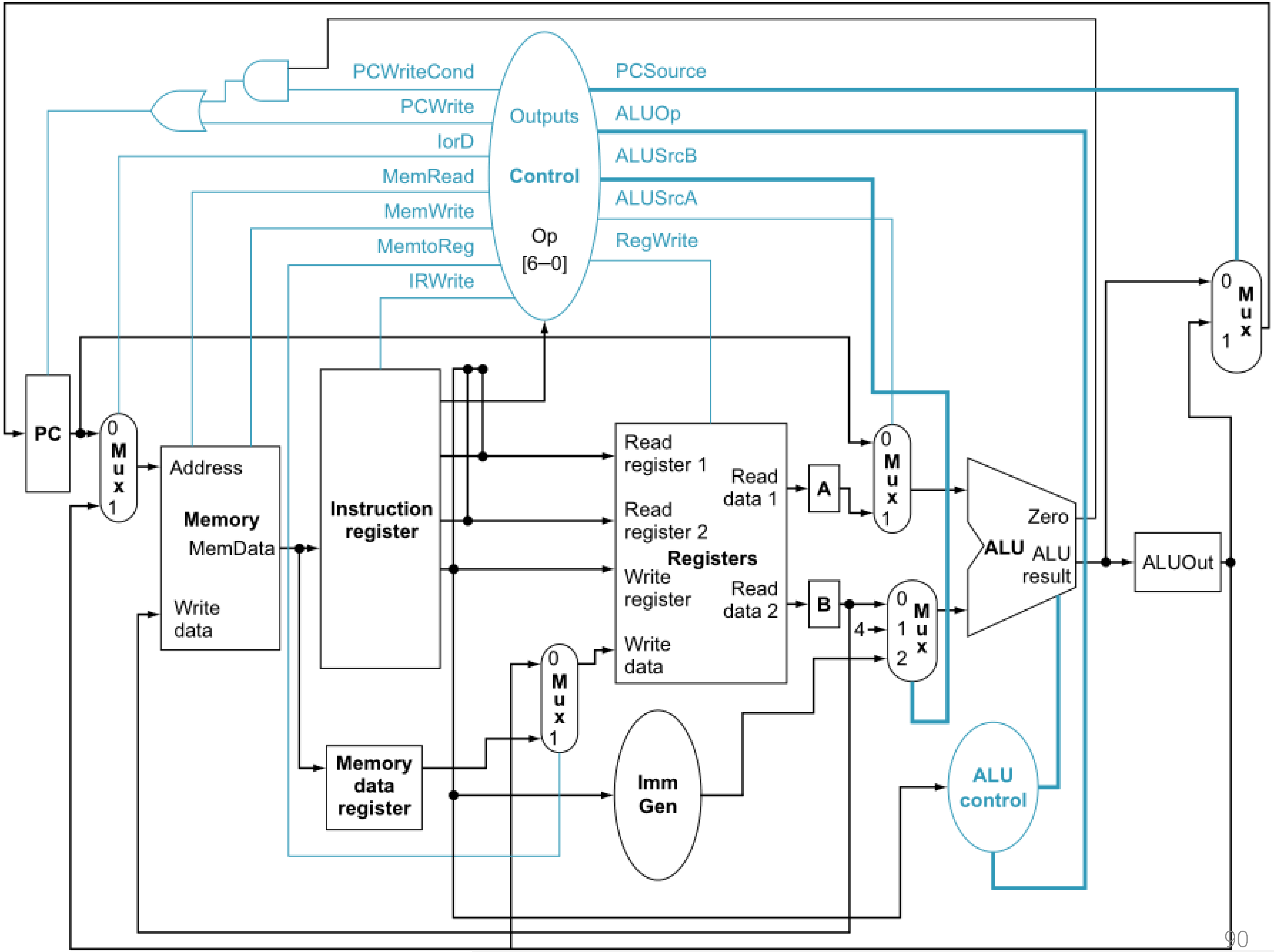
Multicycle CPU



© EnelEva / Adobe Stock

Recall:

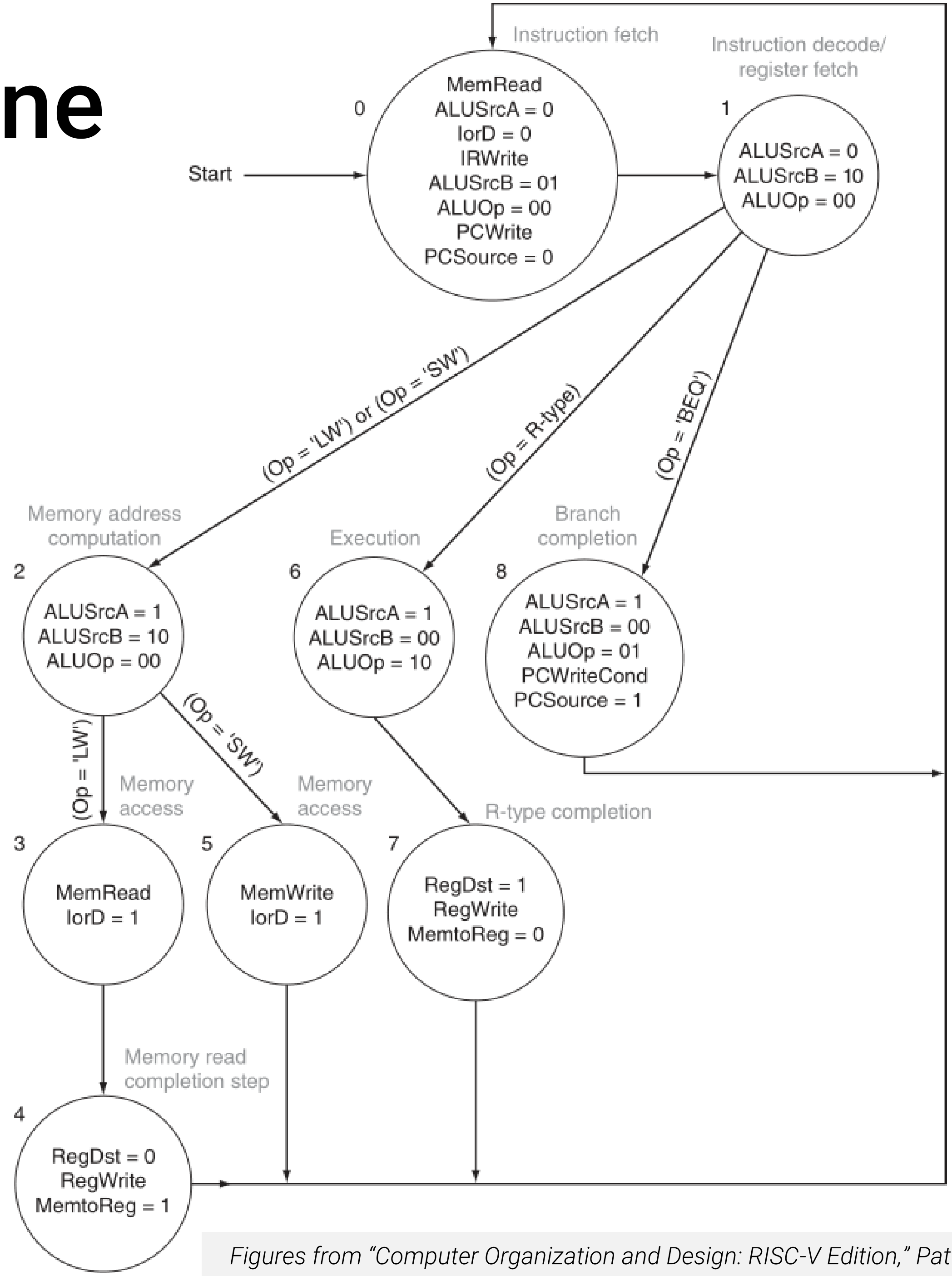
A Simple Multicycle CPU



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2nd Ed.

Finite State Machine

Complete



Figures from "Computer Organization and Design: RISC-V Edition," Patterson & Hennessy, 2^{nd} Ed.

How Many is Multi?

- **Q:** How many cycles do R-type, memory load, memory store, and branch if equal instructions take in our simple multicycle CPU?
- **A:**

R-type:	4 cycles
Memory load:	5 cycles
Memory store:	4 cycles
Branch:	3 cycles

CPI in a Multicycle CPU

- Given the multicycle CPU implementation and the following mix of instructions in a program, find the corresponding CPI:
  - 20% memory loads
  - 8% memory stores
  - 10% branches
  - 62% ALU (the rest of the mix)

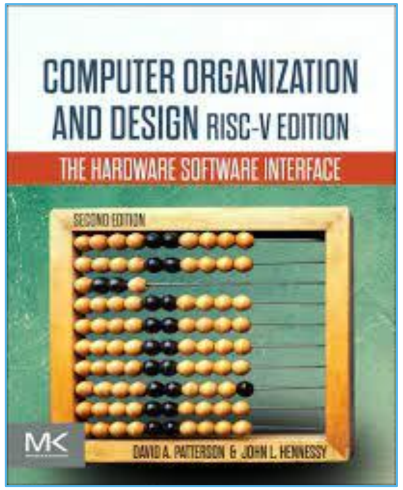
CPI in a Multicycle CPU

Solution

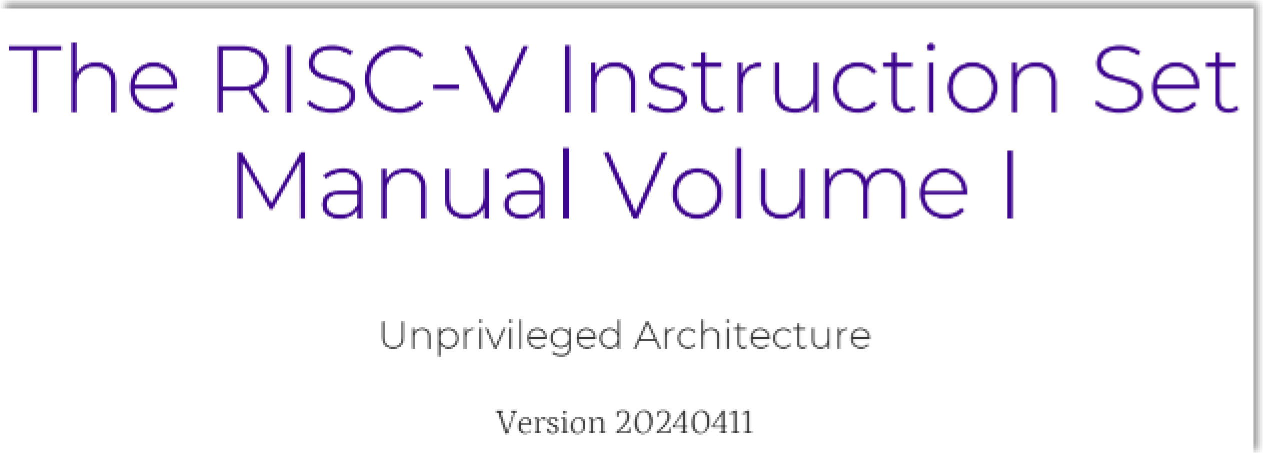
$$\textbf{CPU cycles} = \frac{\textbf{Instructions}}{\textbf{\textit{for a program}}} \times \frac{\textbf{Average clock cycles}}{\textbf{\textit{per instruction}}}$$

- CPU cycles = $\sum_{i=1}^{n}$ Instruction count$_i$ × CPI$_i$
- CPI = CPU cycles / Instruction count
- Therefore, CPI = 0.20×5 + 0.08×4 + 0.10×3 + 0.62×4 = 4.10
- This CPI is better (lower) than if all the instructions were to take the same number of clock cycles (here, five)

Literature


COMPUTER ORGANIZATION AND DESIGN RISC-V EDITION
THE HARDWARE SOFTWARE INTERFACE
SECOND EDITION
MK
DAVID A. PATTERSON & JOHN L. HENNESSY

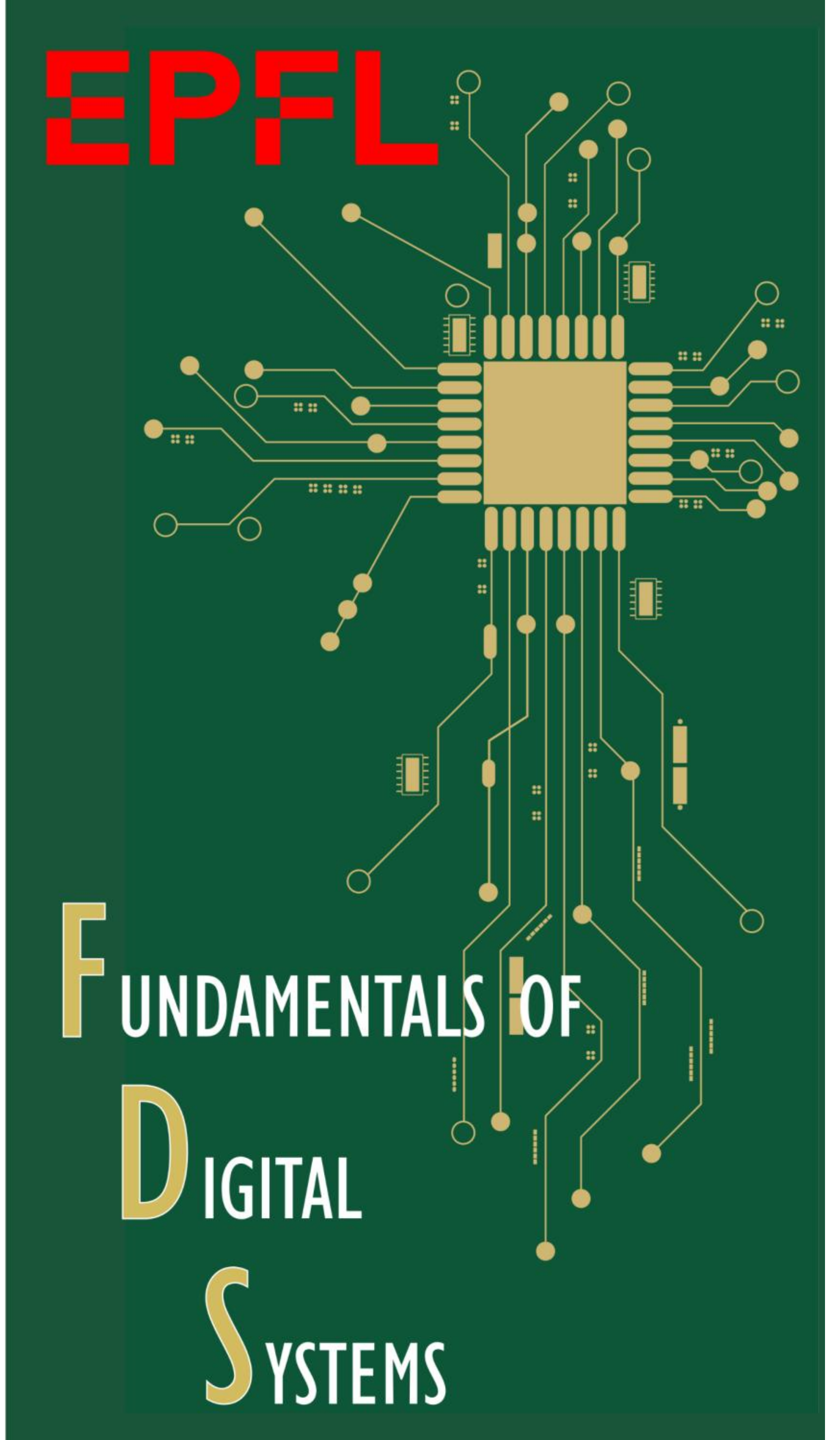
- Chapter 4: The Processor
  - 4.5


The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20240411

Visit online: Link



The End

CS-173 Fundamentals of Digital Systems

Mirjana Stojilović

Spring 2025