Introduction to Computer Architecture
Chapter 4 - 1

The Processor

Hyungmin Cho

Department of Computer Science and Engineering
Sungkyunkwan University

Introduction

- **Micro-architecture** means the **internal** structure of a CPU
  - Externally, the CPUs of the same ISA has the same interface.
  - Internally, the CPUs can be composed different to each other, depending on their performance/efficiency targets

- Microarchitecture examples we will study
  - **Single-cycle**
    - Each instruction is executed in one cycle
    - It suffers from the long critical path delay, which limits the clock frequency
  - **Pipelined**
    - Execution is broken up into a series of steps
    - Multiple instructions are executed in an overlapped fashion.

Introduction

- Our example supports only a subset of RISC-V ISA
  - Arithmetic/logic: **`add, sub, and, or`**
  - Memory access: **`lw, sw`**
  - Control transfer: **`beq`**

→ Covers most common operations of a typical RISC-style CPU

What Happens in CPU

- Load an instruction from (instruction) memory (***fetch***)
  - Memory address: `PC`
- Use the opcode field to identify the instruction type (***decode***)
  - `Instruction[6:0]`   (i.e., lower 7 bits)
- These first two steps are identical for all instruction types.

What Happens in CPU

- Arithmetic/Logic instructions: **`add, sub, and, or`**
  - Read two operand values from the register file (*`rs1`*, *`rs2`*)
  - Perform calculation
  - Write the result back into the register file (*`rd`*)
  - Add 4 to `PC`

What Happens in CPU

- Memory access: **`lw`**
  - Read 1 word from the register file (*`rs1`*): base address
  - Add the base address with the offset (*Immediate field*)
  - Load word from data memory (Address: base address + offset)
  - Write loaded word into the register file (*`rd`*)
  - Add 4 to `PC`

What Happens in CPU

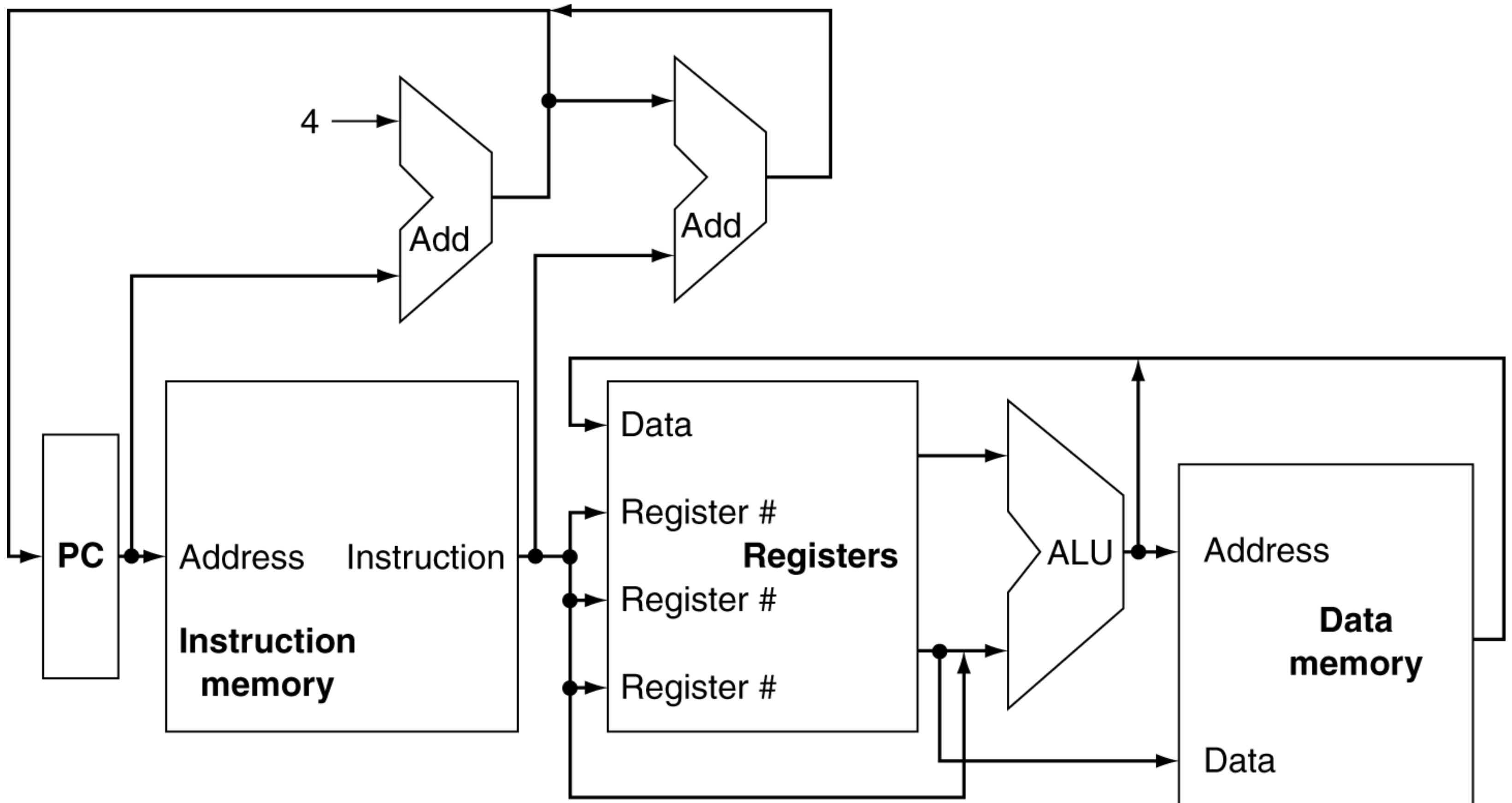
- Memory access: **`sw`**
  - Read 1 word from the register file (*`rs1`*): base address
  - Add the base address with the offset (*Immediate field*)
  - Read 1 word from the register file (*`rs2`*): value to store
  - Store the word to data memory (Address: base address + offset)
  - Add 4 to PC

What Happens in CPU

- Control transfer: **`beq`**
  - Read 2 words from the register file (*`rs1`*, *`rs2`*)
  - Perform comparison (using ALU): Subtraction
  - If *`*rs1 ==*rs2`*
    - PC ← PC + immediate
  - else
    - PC ← PC + 4

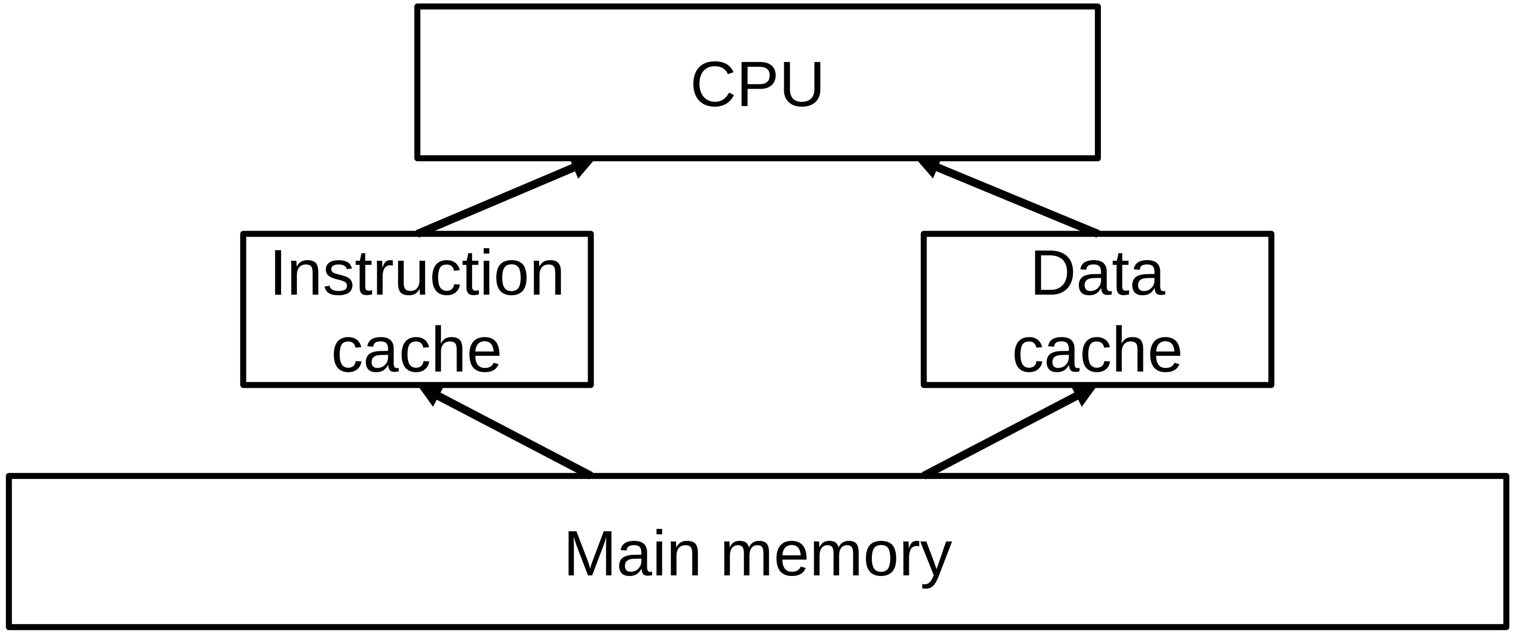
Single-cycle CPU Overview

- 1 cycle for all instructions (CPI = 1)



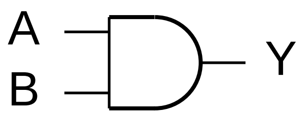
Instruction / Data Memory?

- When describing a processor archicture in this lecture, we use separate units for instruction memory / data memory
- In reality, these are the same main memory.
- From the processor side, they may look like separate memory because we will use separate cache memory

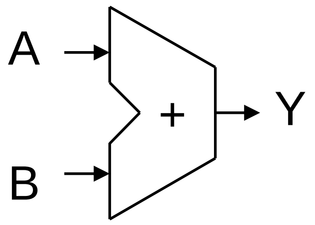


Logic Design Basics: Combinational Logic

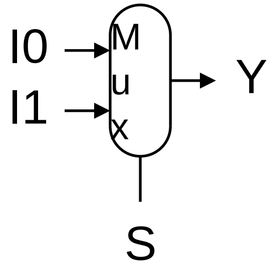
- Output is a function of the **current** input
  - AND-gate
    - Y = A & B



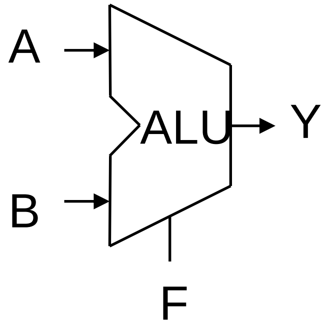
  - Adder
    - Y = A + B



  - Multiplexer
    - Y = S ? I1 : I0

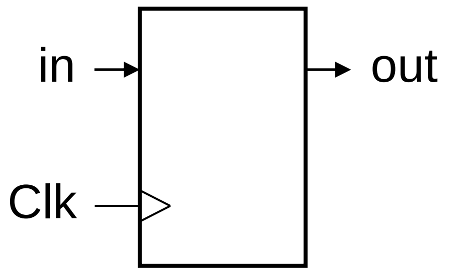


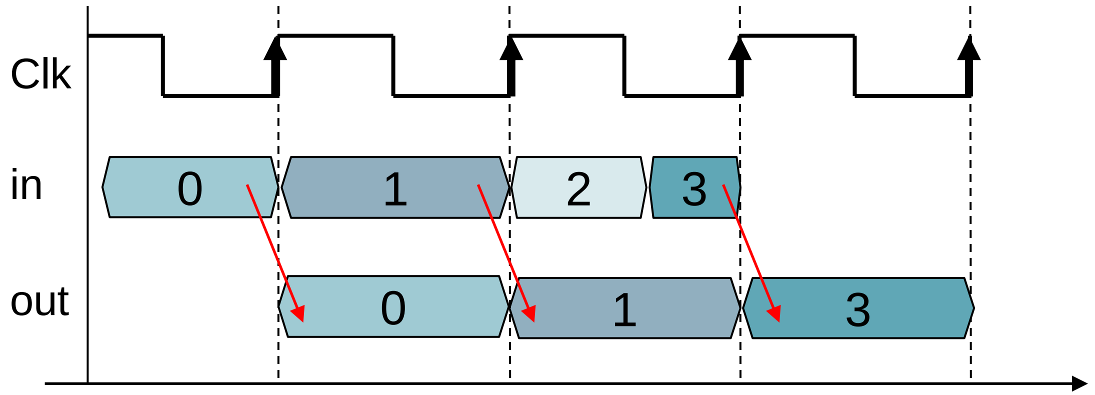
  - Arithmetic/Logic Unit
    - Y = F(A, B)



Logic Design Basics: Sequential Elements

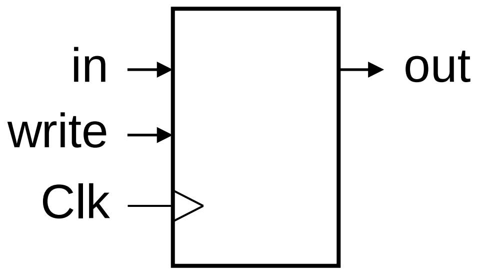
- Register: stores data
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

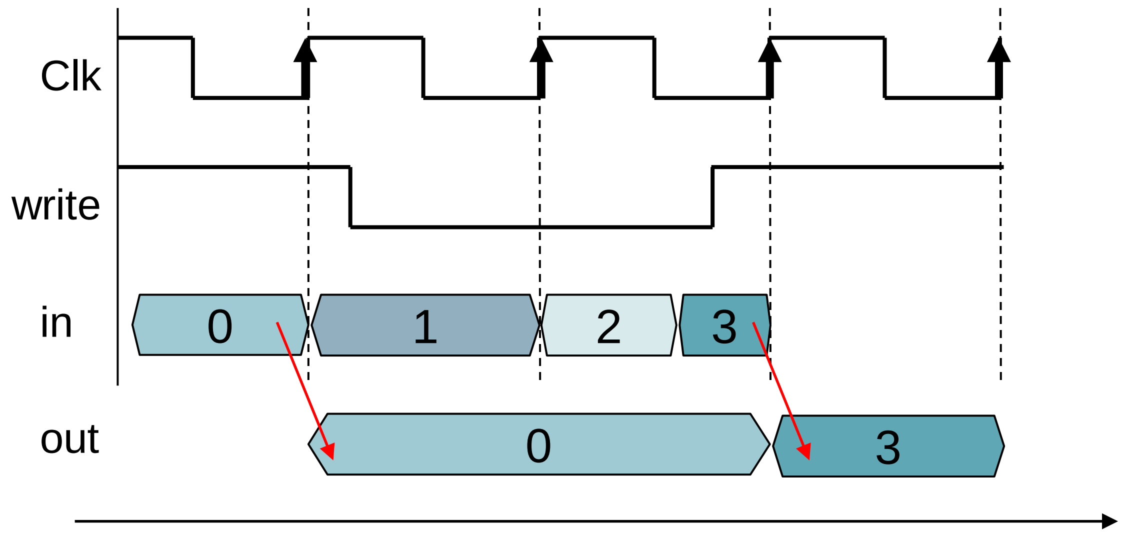




Logic Design Basics: Sequential Elements

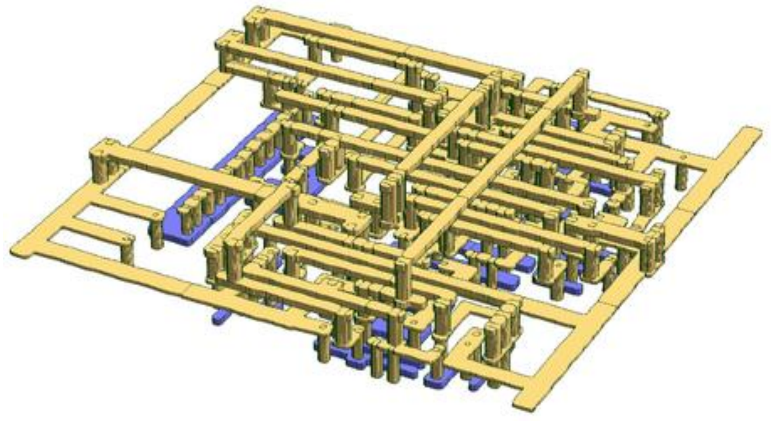
- Register with write control
  - Only updates on clock edge when write control input in 1



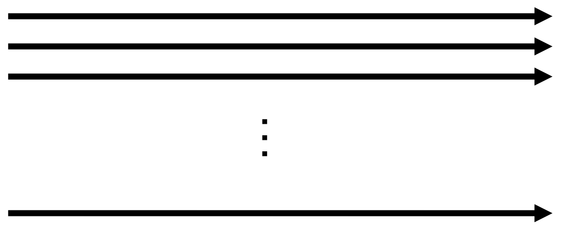


Logic Design Basics: Wires

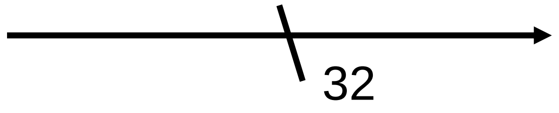
- Logic values (0/1) are delivered through wires



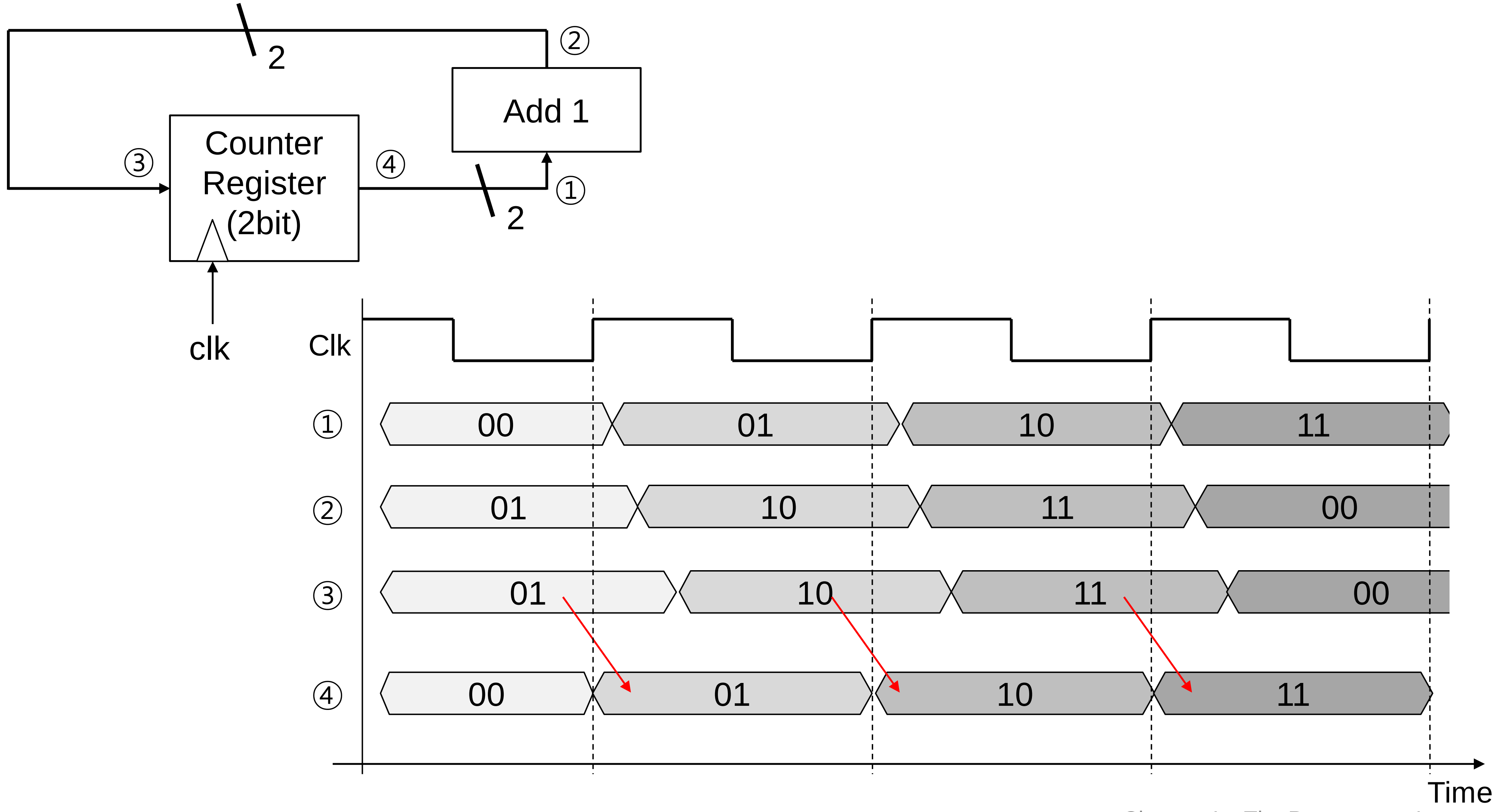
- If you need deliver multiple bits of data, need multiple wires



Simplified representation:

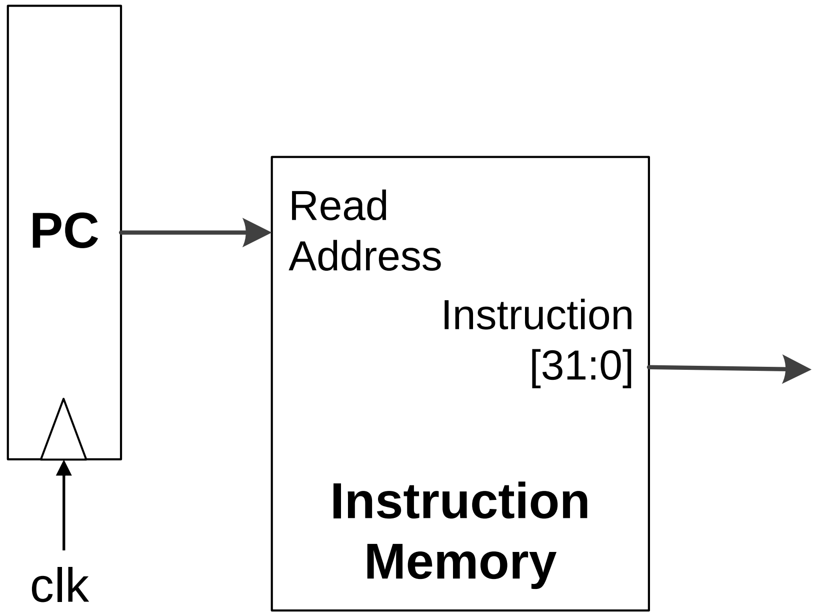


Logic Design Basics: Counters



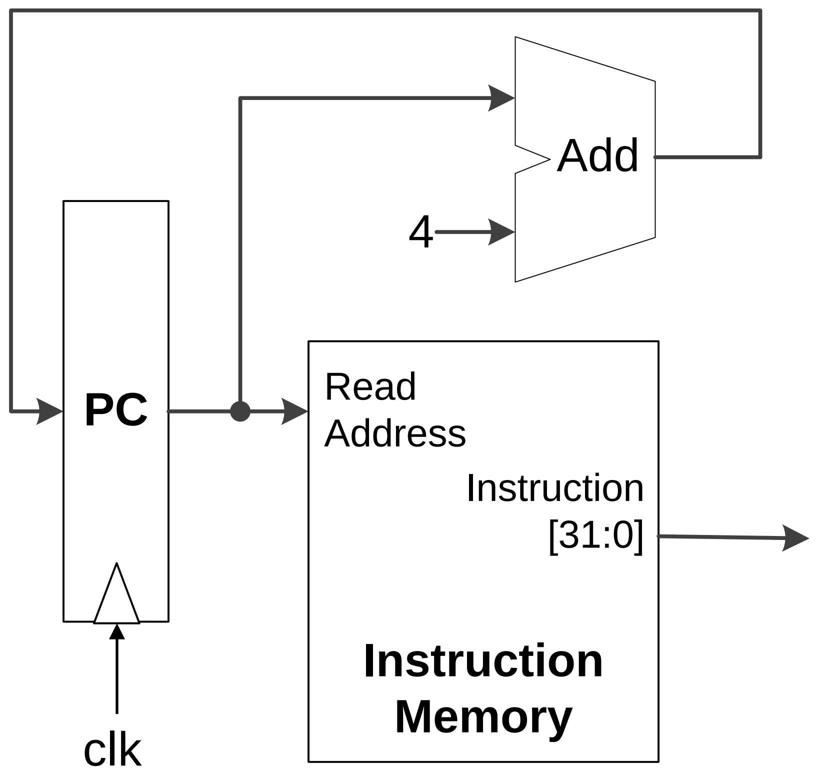
Instruction Fetch

- **PC**: 32-bit register
  - Remembers the current instruction address
- **Instruction Memory**:
  - For now, let’s assume it behaves like a combinational logic i.e., it directly gives us the instruction at the PC address



Instruction Fetch

- Advance to the next instruction
  - Next instruction is at `PC` + 4
  - In the next cycle (next clock period), `PC` should be changed to `PC` + 4



R-Format Instructions

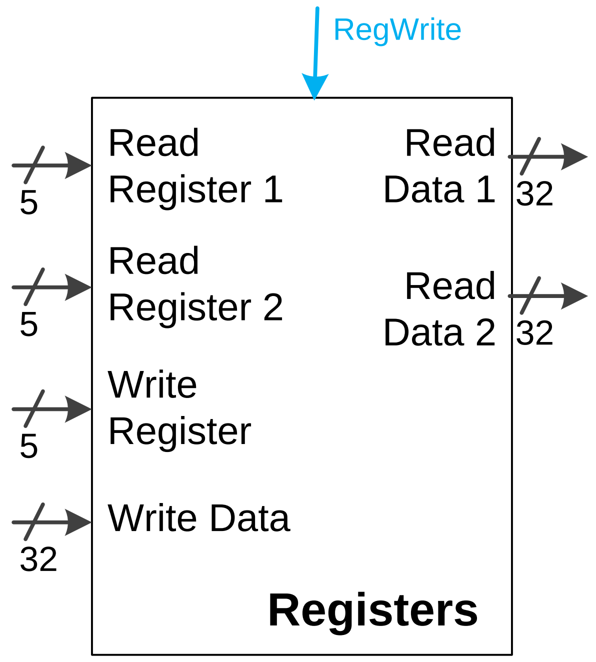
- Read two register operands
- Perform arithmetic/logic operation
- Write the result into the destination register

Bit 31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … Bit 0
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- `add → op: 0110011, funct3: 000 funct7: 0000000`
- `sub → op: 0110011, funct3: 000 funct7: 0100000`
- `and → op: 0110011, funct3: 111 funct7: 0000000`
- …

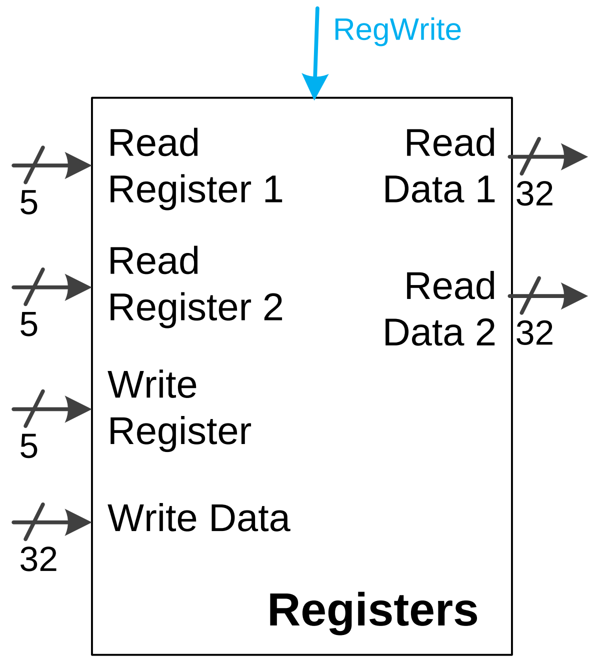
Register File

- Read: Behaves like a combinational logic
  - Read Register 1 / 2: Register IDs from the current instruction
  - Read Data 1 / 2: The current value of the corresponding registers

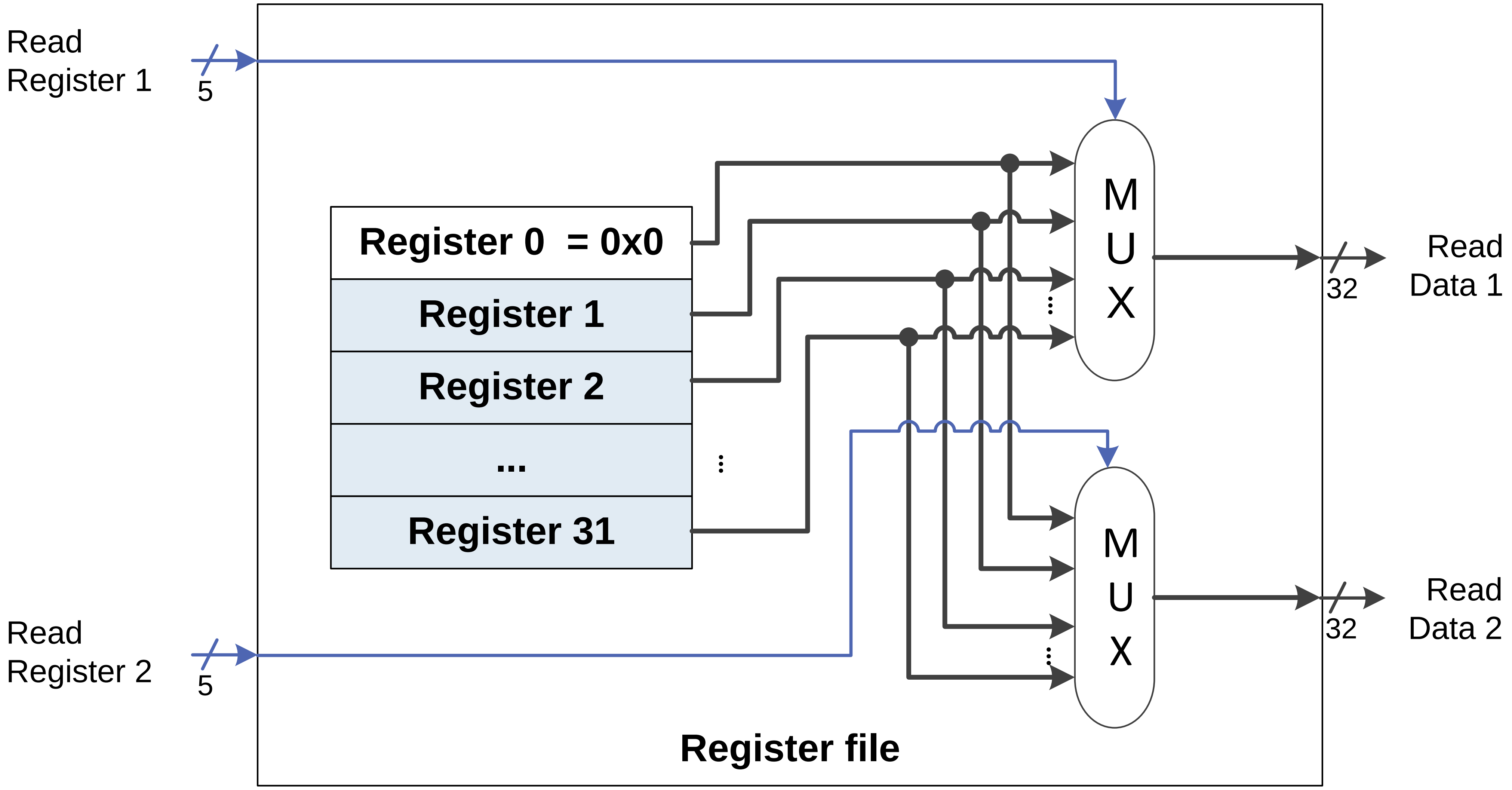


Register File

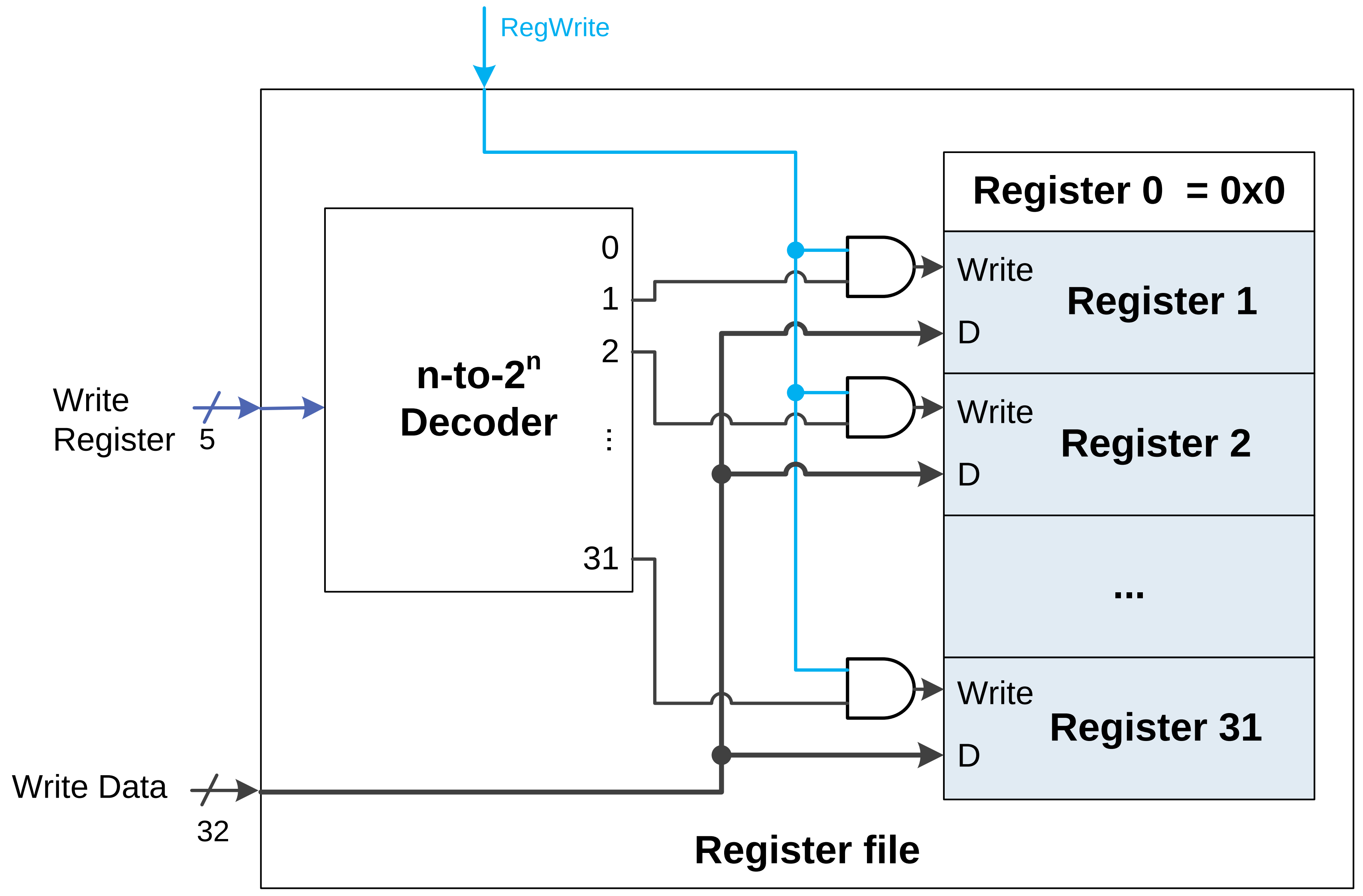
- Write: Behaves like a sequential logic
  - If `RegWrite` is “1”, writes the value to the target register
  - Actual write happens when moving to the next cycle



Two Read Ports

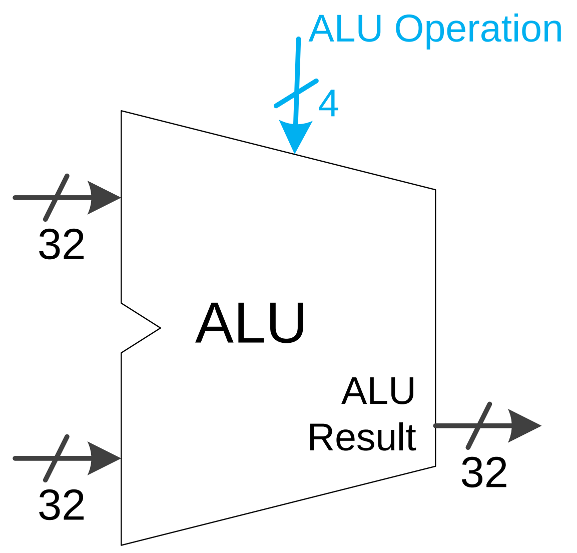


Write Port


RegWrite
Write Register
5
n-to-2^n Decoder
0
1
2
⋮
31
Write Data
32
Register 0 = 0x0
Write
D
Register 1
Write
D
Register 2
...
Write
D
Register 31
Register file

Arithmetic Logic Unit (ALU)

- Multiple functions
  - Add, subtract, shift, and, or, …
  - ALU operation (4-bit input) selects actual function



Arithmetic Logic Unit (ALU)

- ALU used for
  - Load/Store: Function = add
  - Branch: Function = subtract
  - R-type: Function is determined by the `funct7 & funct3` fields

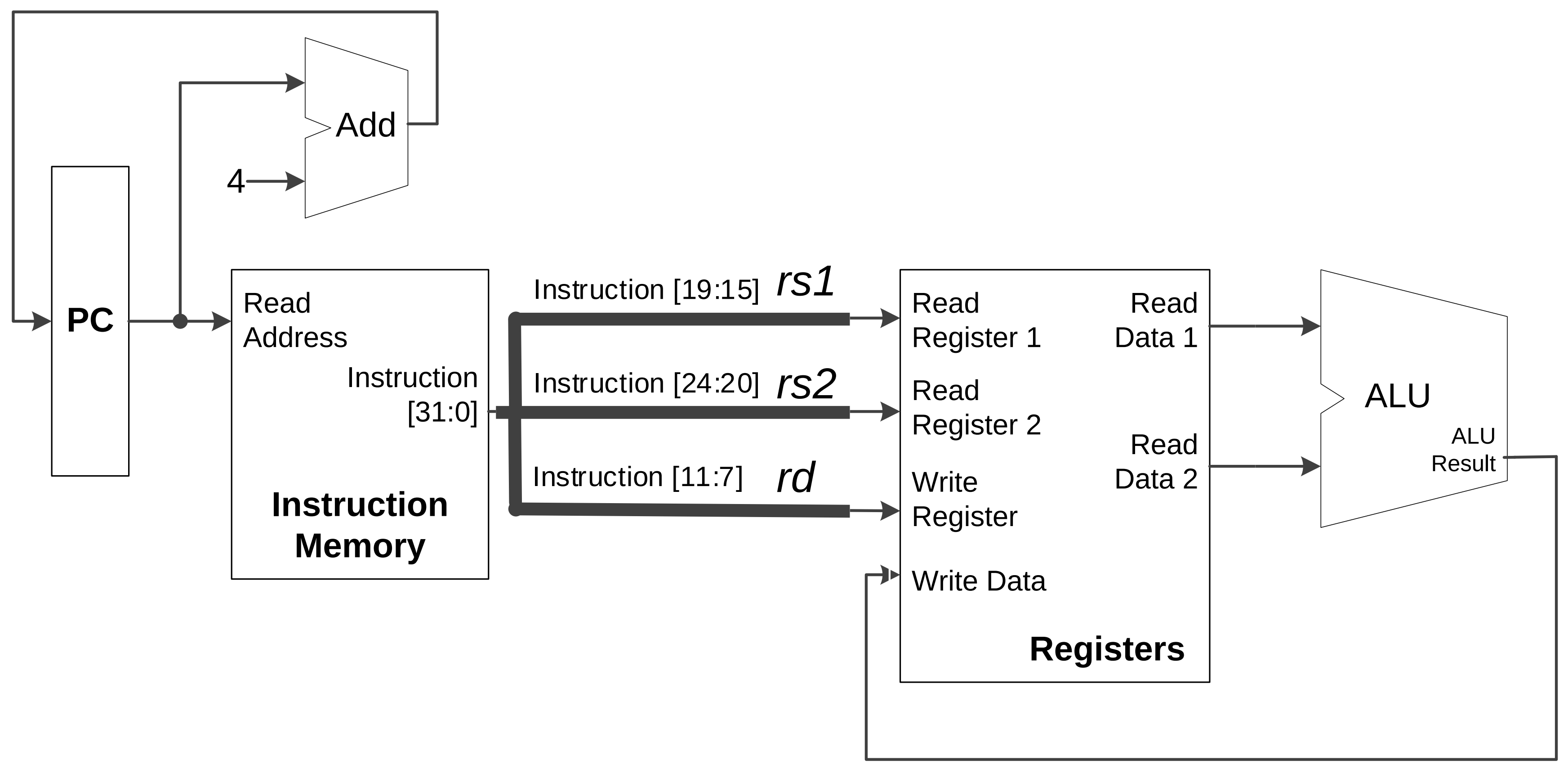
ALU control lines	**ALU Function**
0000	AND
0001	OR
0010	add
0110	subtract
…	…
…	…

ALU Control Bits

Opcode	ALUOp	Operation	Funct7	Funct3	ALU function	ALU control
`lw`	00	load word	XXXXXXX	XXX	add	0010
`sw`	00	store word	XXXXXXX	XXX	add	0010
`beq`	01	branch if equal	XXXXXXX	XXX	subtract	0110
`R-type`	10	add	0000000	000	add	0010
		sub	0100000	000	subtract	0110
		and	0000000	111	AND	0000
		or	0000000	110	OR	0001

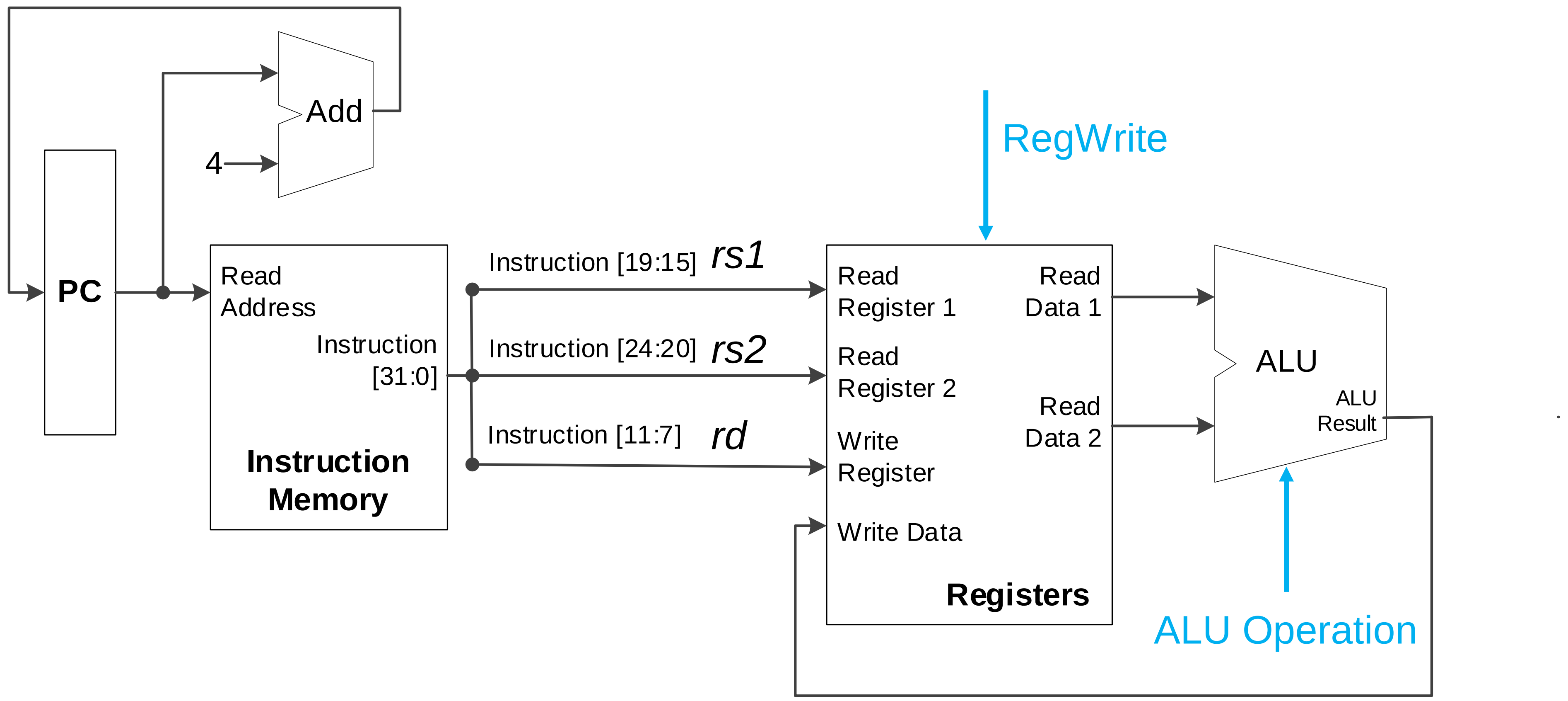
CPU for R-type Instructions

- Datapath



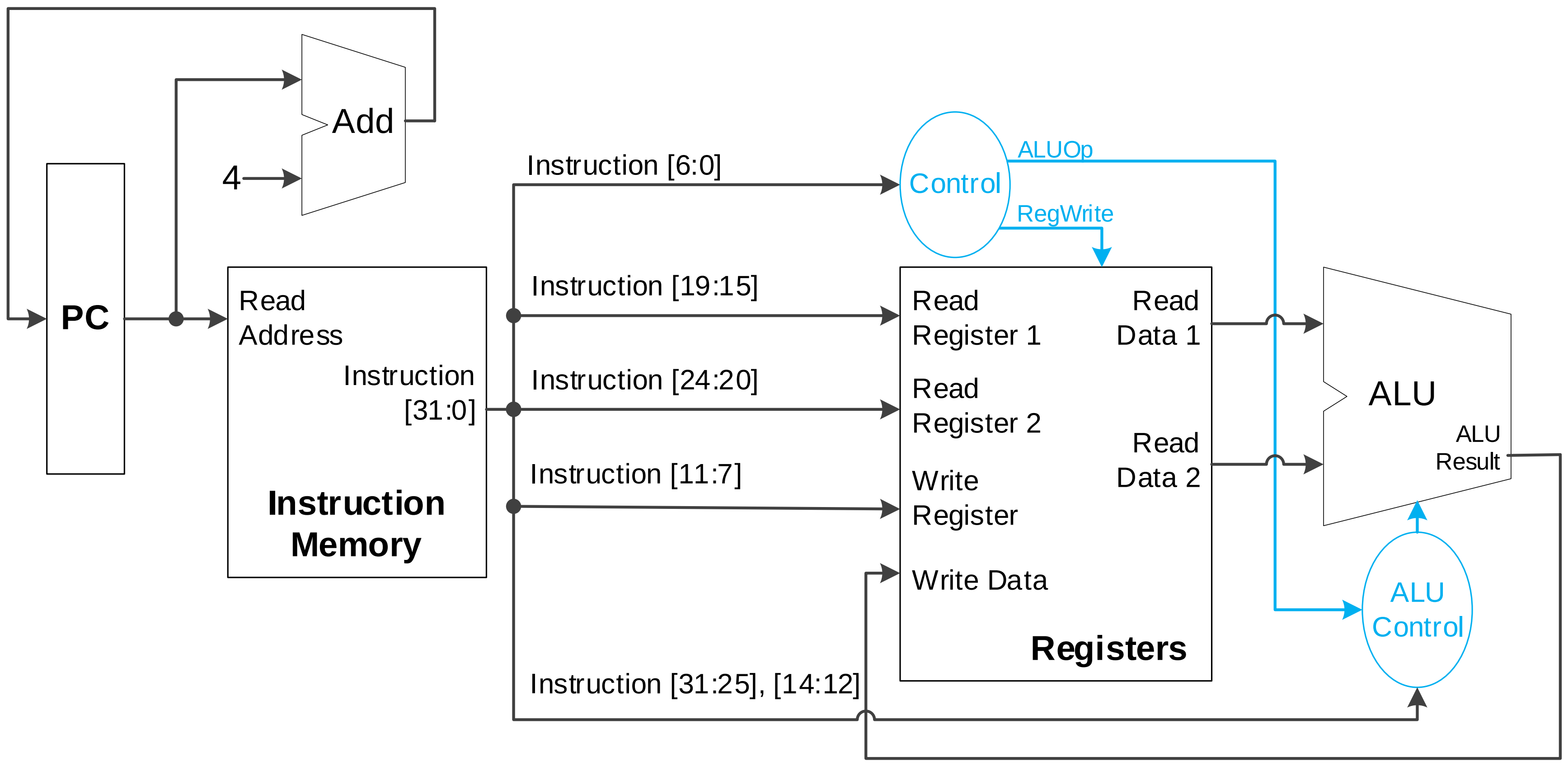
CPU for R-type Instructions

- Datapath



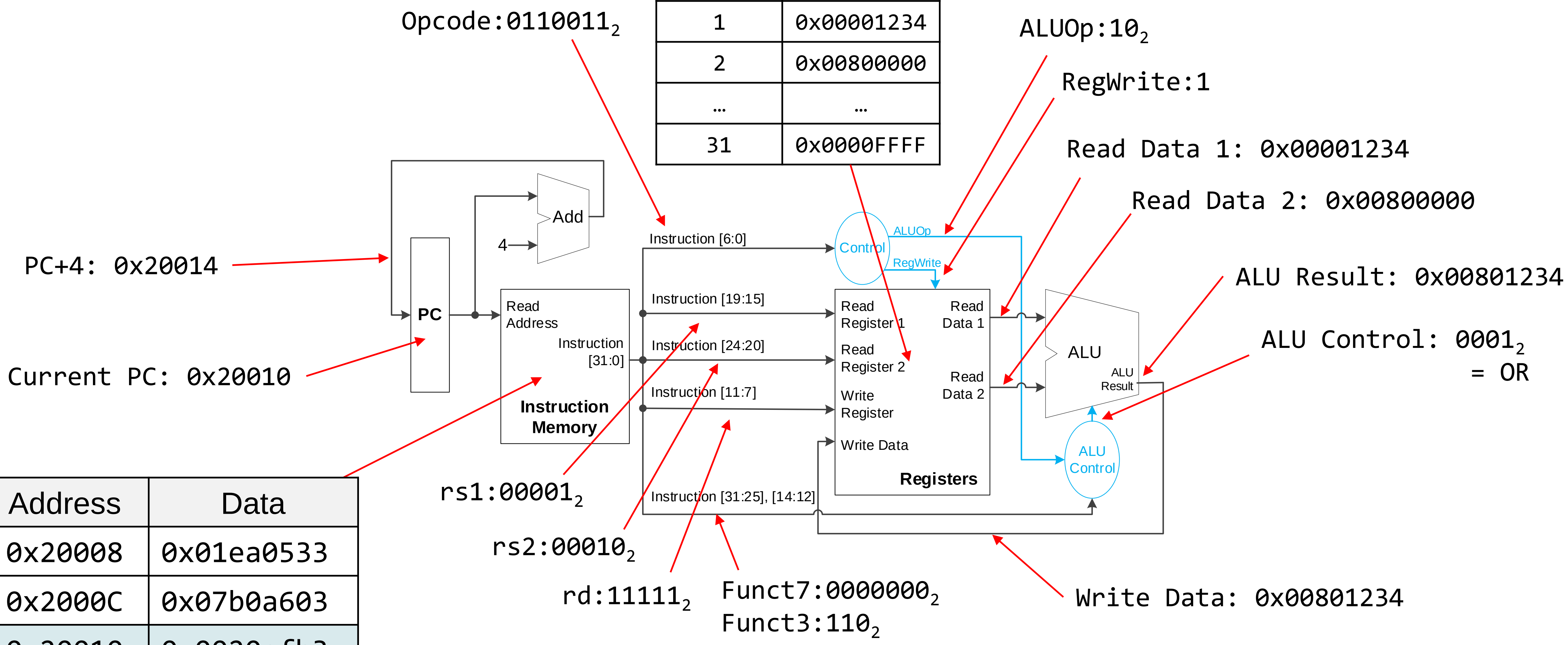
CPU for R-type Instructions

- Datapath + Control signals



R-Type Execution Example

Register	Data
1	0x00001234
2	0x00800000
…	…
31	0x0000FFFF



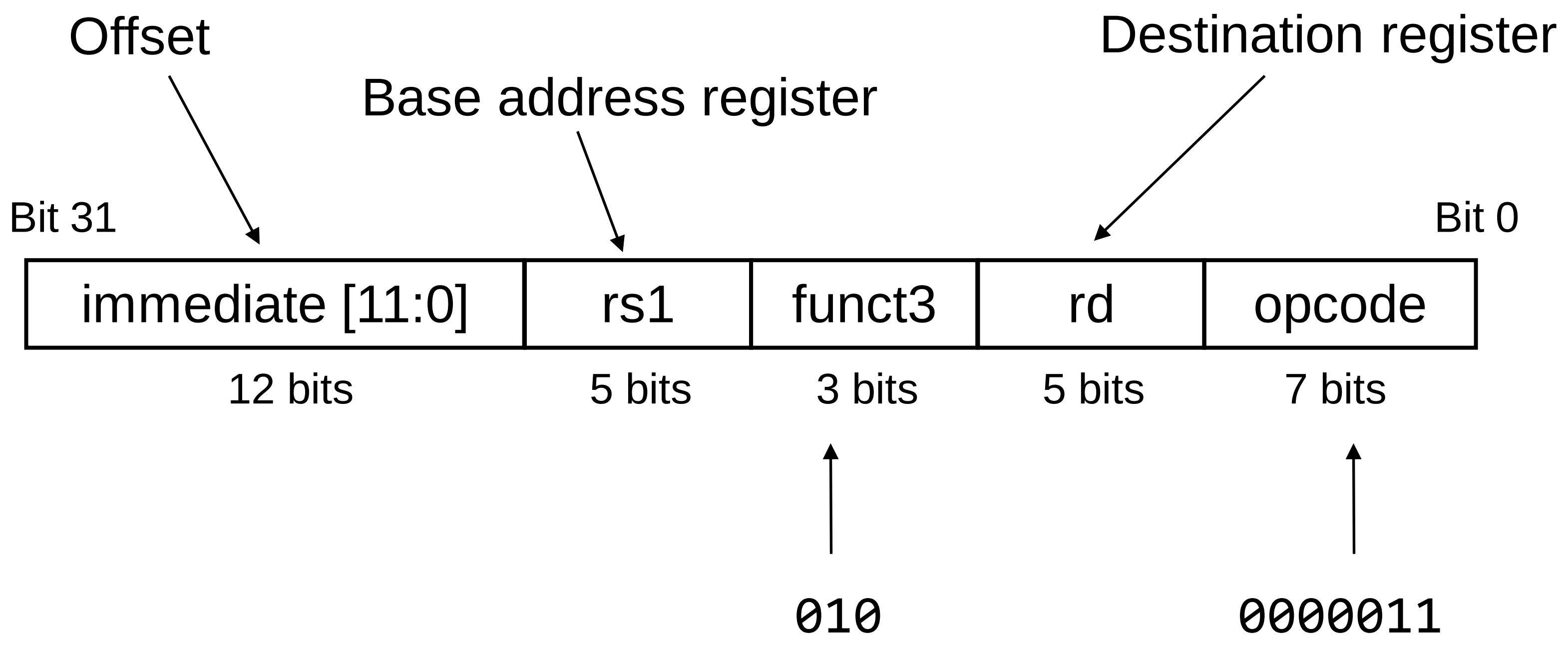
Address	Data
0x20008	0x01ea0533
0x2000C	0x07b0a603
0x20010	0x0020efb3
0x20014	0xfed488e3

0000 0000 0010 0000 1110 1111 1011 0011 → or x31, x1, x2

0000000	00010	00001	110	11111	0110011
funct7	rs2=x2	rs1=x1	funct3	rd=x31	opcode=R-type arith./logic

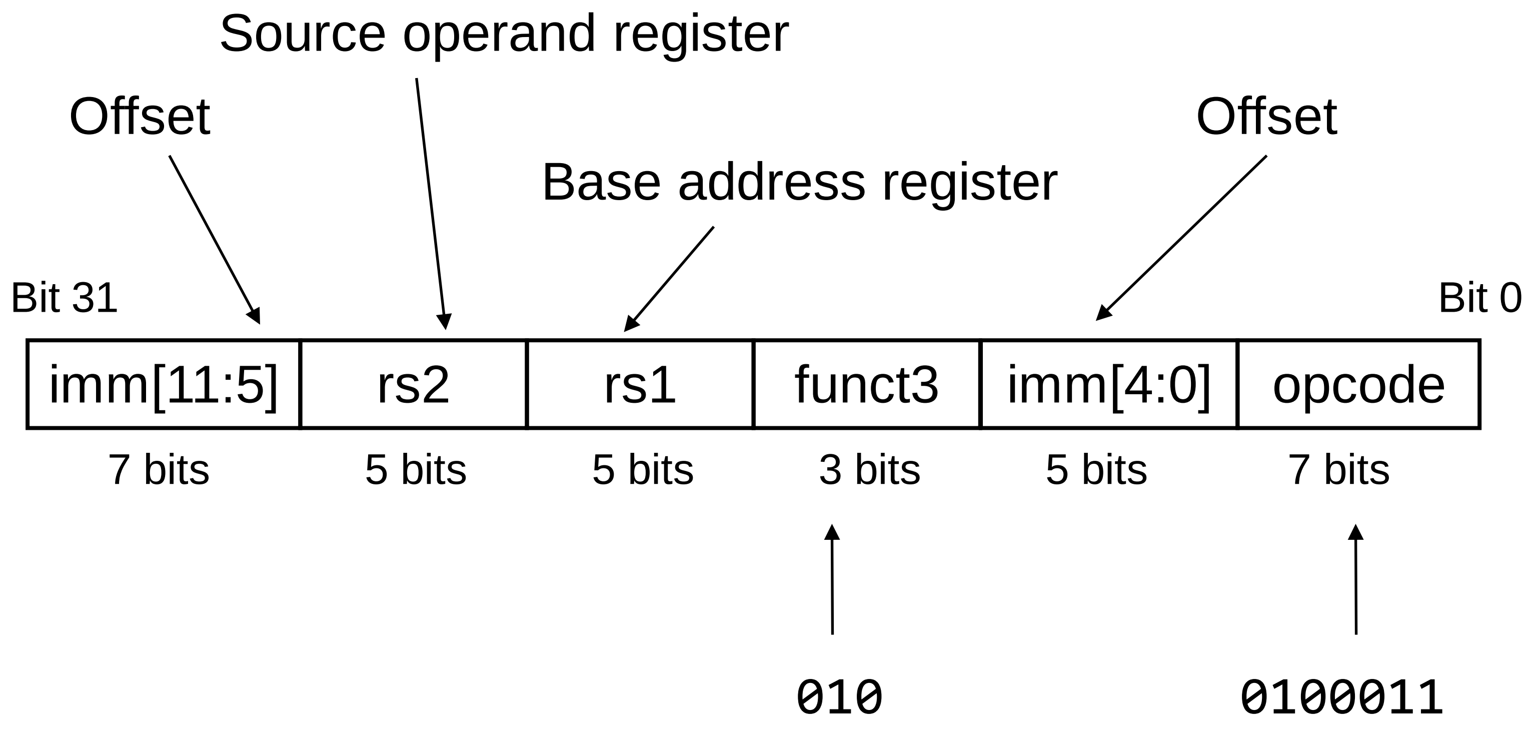
Load Instruction

- Address: Base address register + Offset



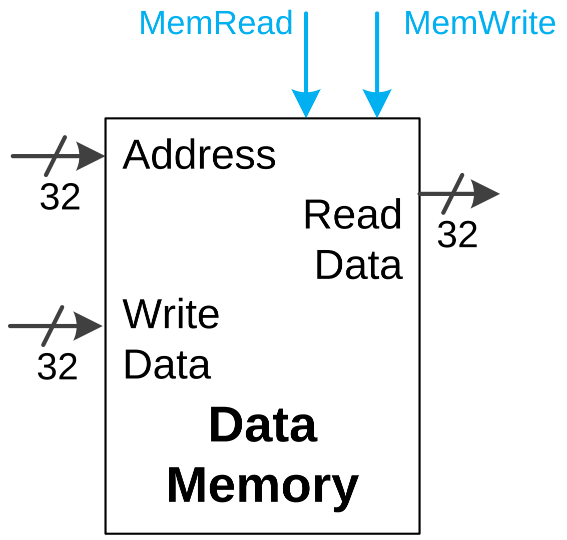
Store Instruction

- Address: Base address register + Offset



Data Memory

- Read: Assume it behaves like a combinational logic
  - If `MemRead` is “1”, gives the current memory value from the address
- Write: Behaves like a sequential logic
  - If `MemWrite` is “1”, writes the data to the address

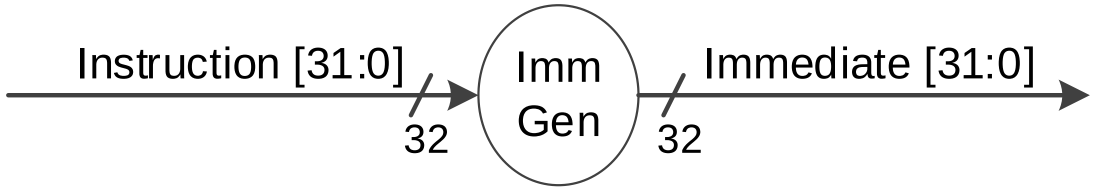


- Accessing main memory takes a long time
  - Typically, hundreds of cycles
  - Cache memory reduces the access time
- For now, we assume memory access does not add long latencies
  - i.e., assume we always have cache hit

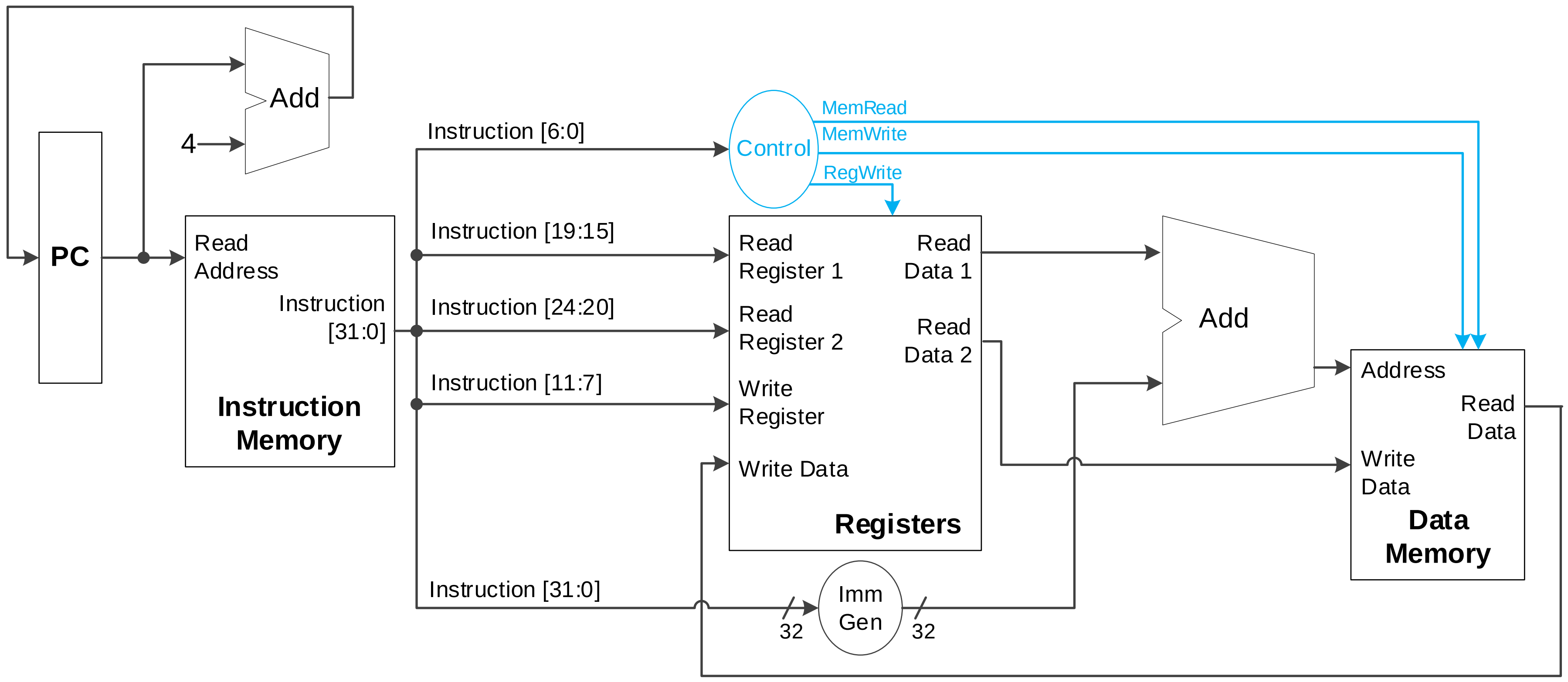
Immediate Generation Unit (ImmGen)

- RISC-V instructions have immediate fields in various bit places
  - Can identify the format from the opcode
  - Sign-extend

I format	immediate [11:0]		rs1	funct3	rd		opcode	
S format	imm[11:5]	rs2	rs1	funct3	imm[4:0]		opcode	
SB format	12	imm[10:5]	rs2	rs1	funct3	imm[4:1]	11	opcode
UJ format	20	imm[10:1]	11	imm[19:12]	rd	opcode		
U format	imm[31:12]	rd	opcode					



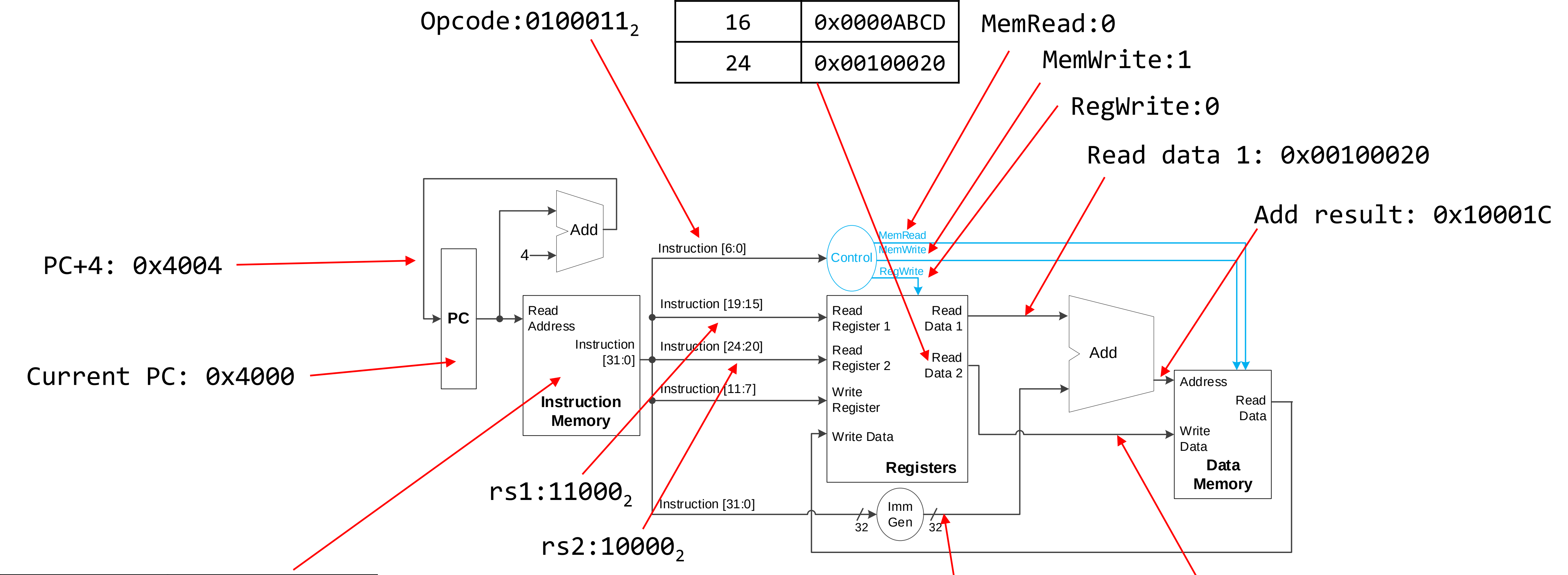
CPU for Load/Store



	MemRead	MemWrite	RegWrite
Load	1	0	1
Store	0	1	0

Store Execution Example

Register	Data
16	0x0000ABCD
24	0x00100020



Address	Data
0x4000	0xFF0C2E23

```
1111 1111 0000 1100 0010 1110 0010 0011   → sw x16, -4(x24)
1111111  10000  11000  010  11100  0100011
Imm[11:5]  rs2=x16  rs1=x24  funct3  Imm[4:0]  opcode = sw
```