National Yang Ming Chiao Tung University
Computer Architecture & System Lab

Lecture 5: Single-Cycle CPU
CS10014 Computer Organization

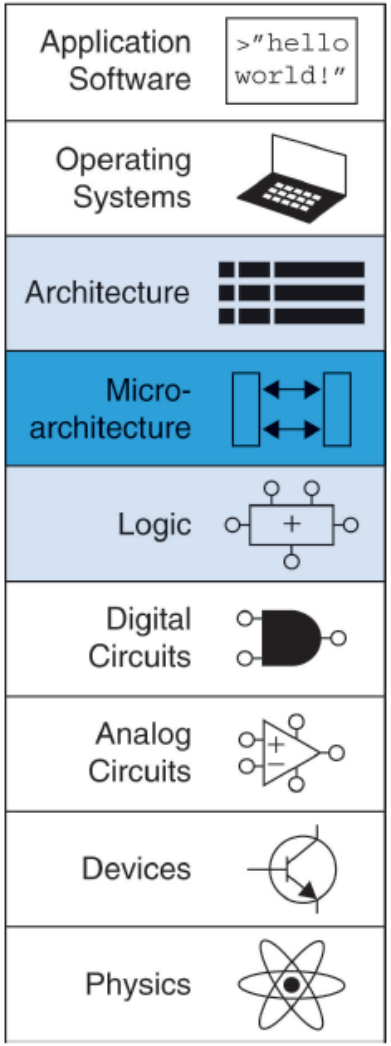
Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - E85 at HMC
    - https://pages.hmc.edu/harris/class/e85/old/fall21/

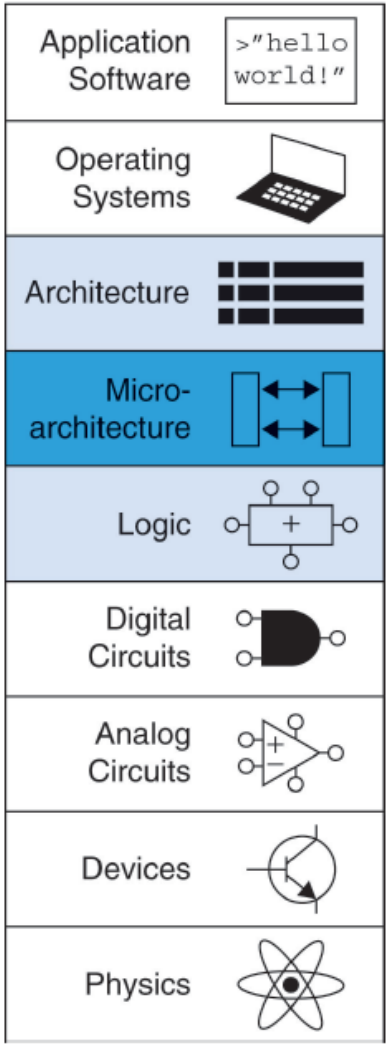
Outline

- ALU
- State Element
- Single-Cycle CPU
- Design the Datapath



Introduction

- **Microarchitecture**
  - How to implement an architecture in hardware
- **Processor**
  - Datapath: functional blocks
  - Control: control signals



RISC-V ISAs

- **R-type instructions:**
  - `add, sub, and, or, slt`
- **I-type instruction:**
  - `lw`
- **S-type instruction:**
  - `sw`
- **B-type instructions:**
  - `beq`

R-Type

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

I-Type

31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

S-Type

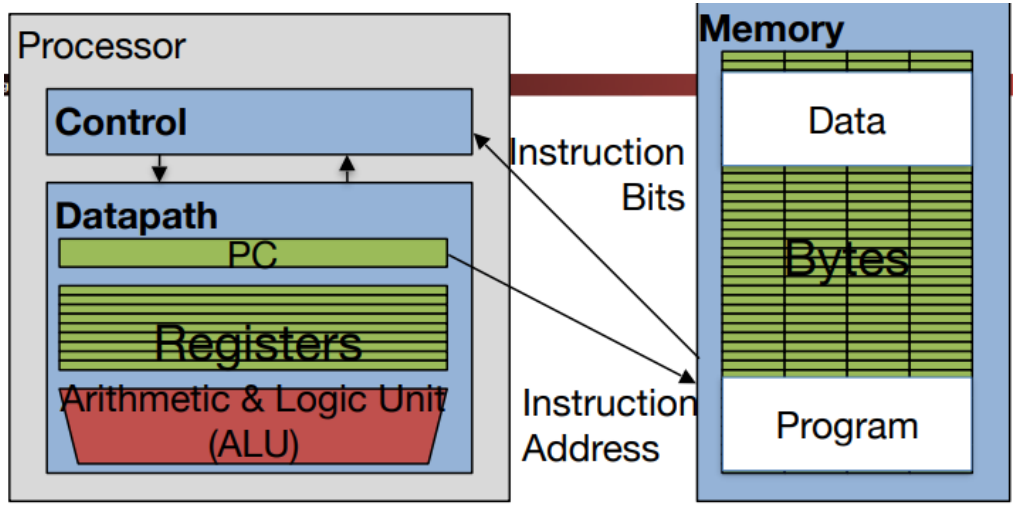
31:25	24:20	19:15	14:12	11:7	6:0
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Single Core Processor

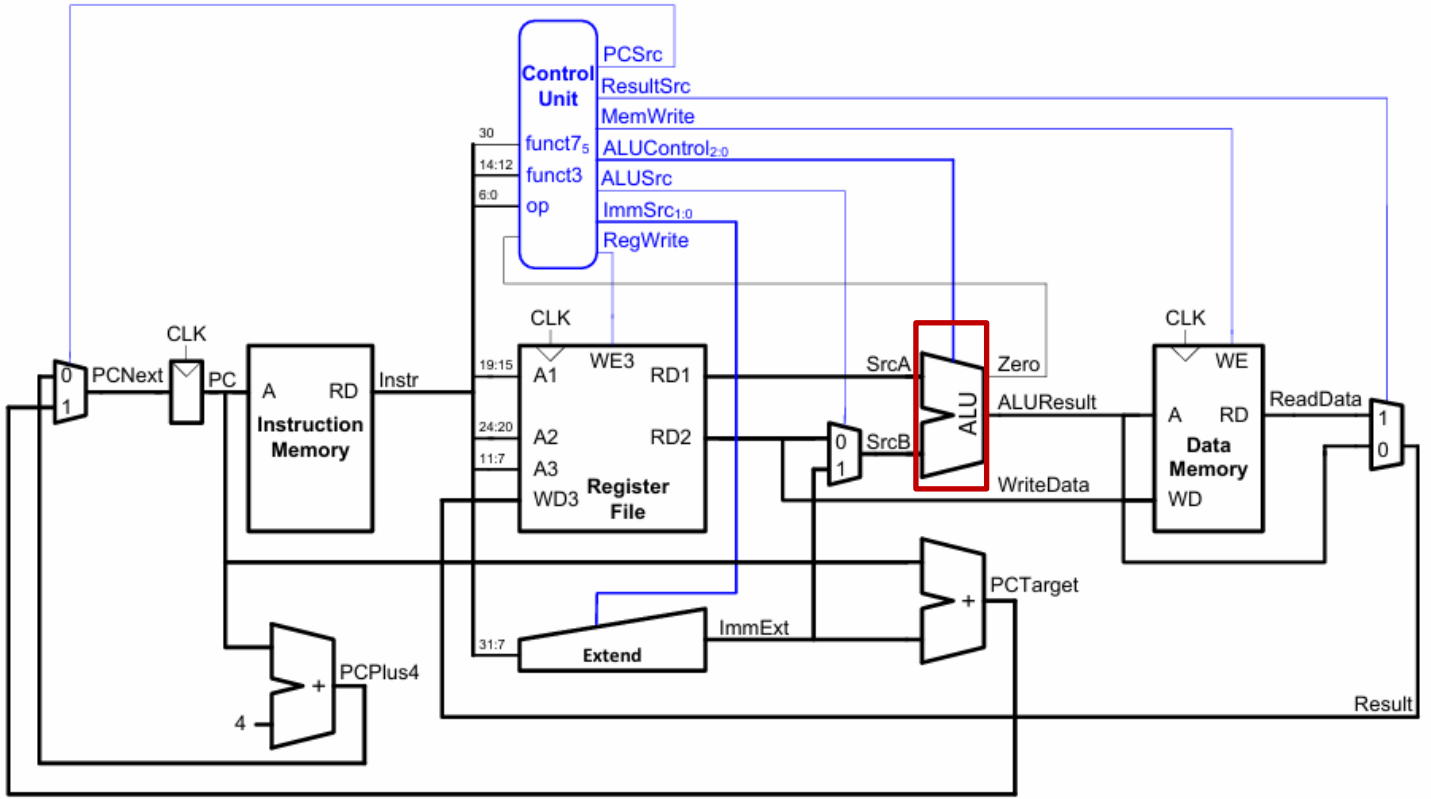
- **Processor (CPU)**
  - The active part of the computer that does all the work (data manipulation and decision-making)
- **Datapath**
  - Contains hardware necessary to perform operations required by the processor
- **Control**
  - Tells the datapath what needs to be done



Microarchitecture

- Multiple implements for a single architecture
  - Single-cycle:
    - Each instruction executes in a single cycle
  - Multi-Cycle:
    - Each instruction is broken up into series of shorter steps
  - Pipelined:
    - Each instruction broken up into series of steps & multiple instructions execute at once

Single-Cycle RISC-V Processor



ALU: Arithmetic Logic Unit

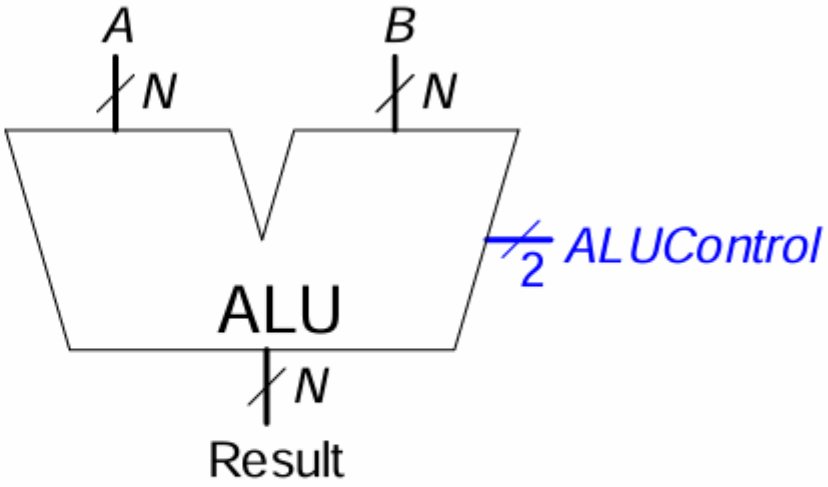
- ALU should perform
  - Addition
  - Subtraction
  - AND
  - OR

ALU: Arithmetic Logic Unit

ALUControl$_1$

ALUControl$_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

ALUControl$_0$



Example: Perform *A* OR *B*

ALUControl$_{1:0}$ = 11

Result* = *A* OR *B

ALU: Arithmetic Logic Unit

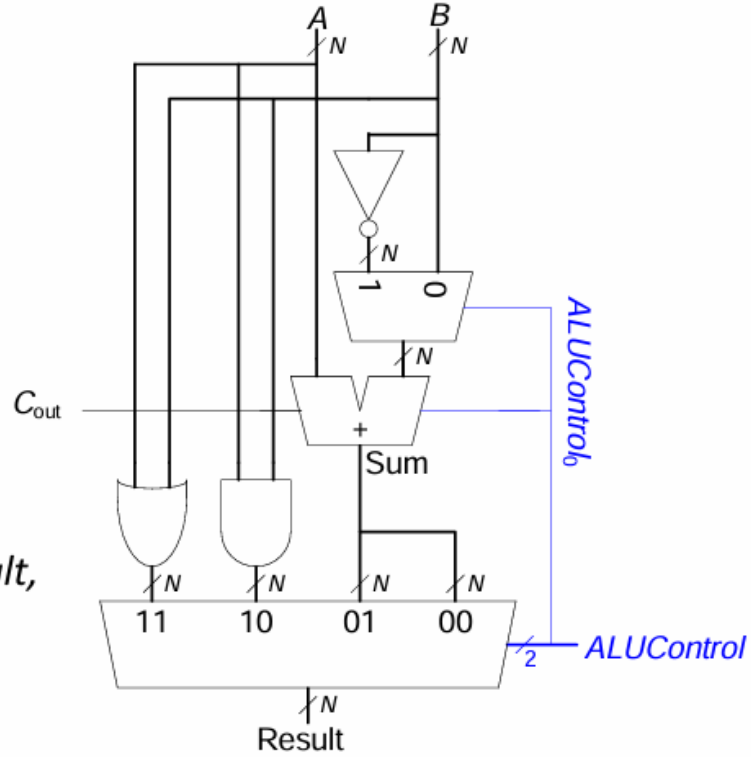
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* OR *B*

$ALUControl_{1:0}$ = 11

Mux selects output of OR gate as *Result*, so:

Result* = *A* OR *B



ALU: Arithmetic Logic Unit

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

Example: Perform *A* + *B*

$ALUControl_{1:0}$ = 00

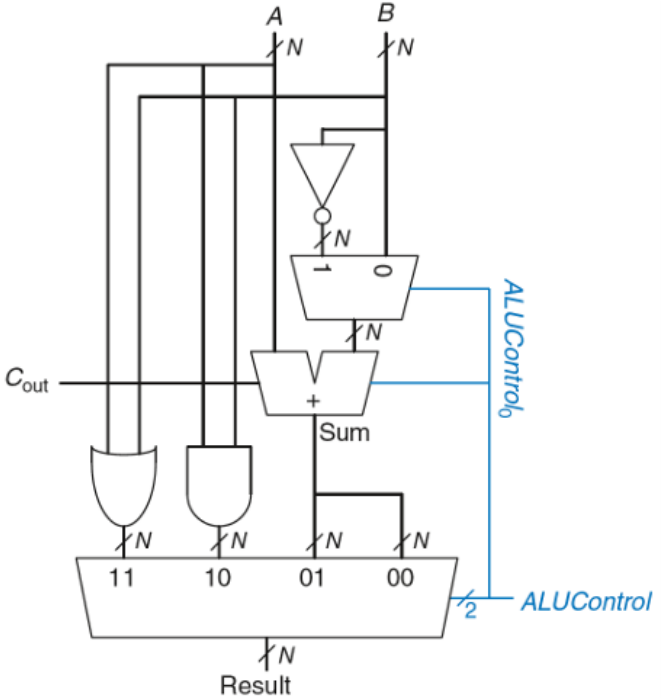
$ALUControl_0$ = 0, so:

C_{in} to adder = 0

2nd input to adder is *B*

Mux selects *Sum* as *Result*, so

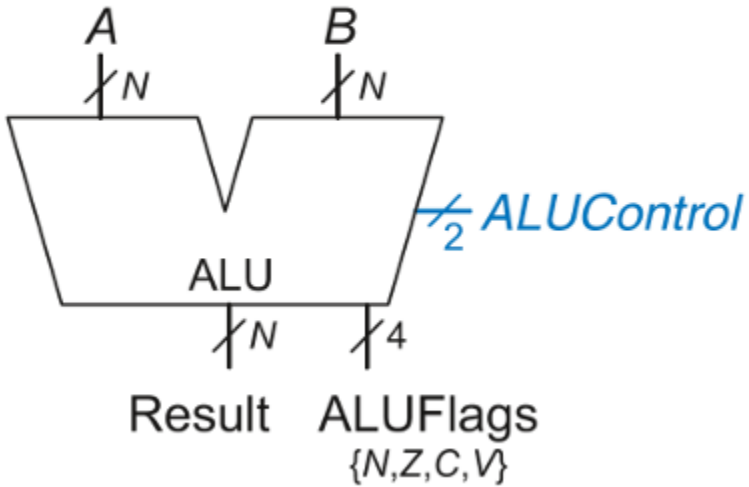
Result* = *A* + *B



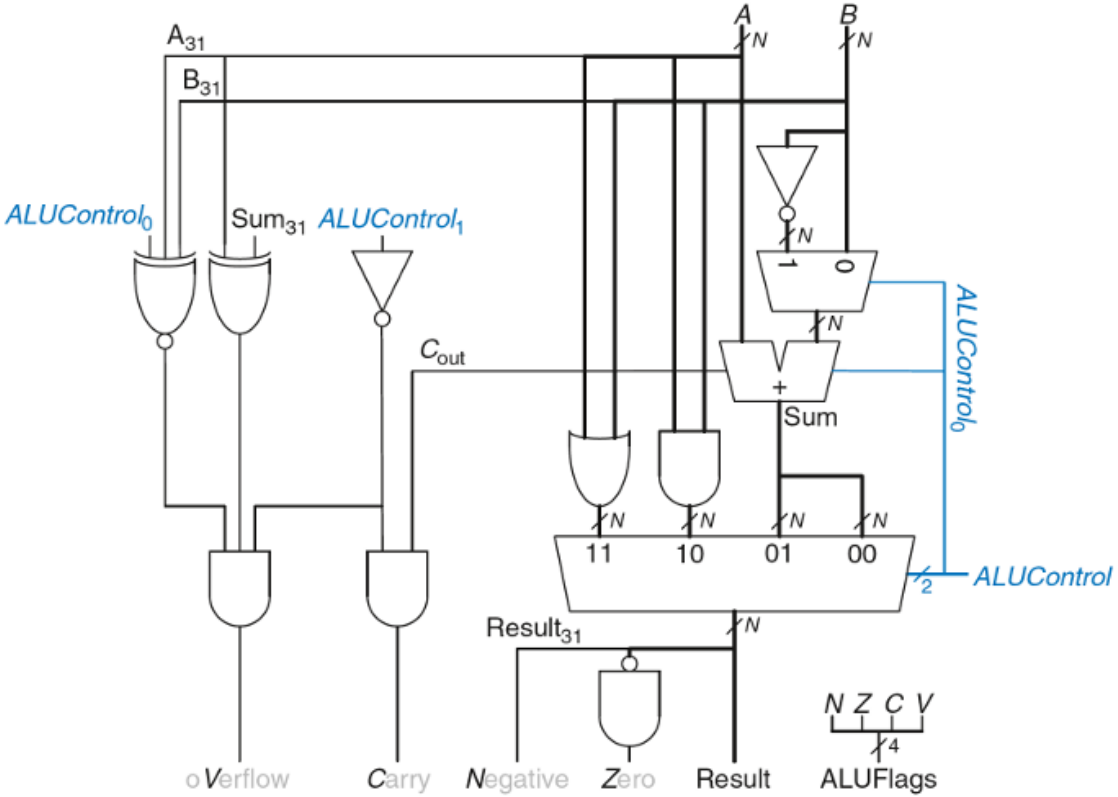
- How to subtract using an adder?
  - sub A B = add A -B
  - Negate B before adding (fast negation trick: -B = B' + 1)

ALU with Status Flags

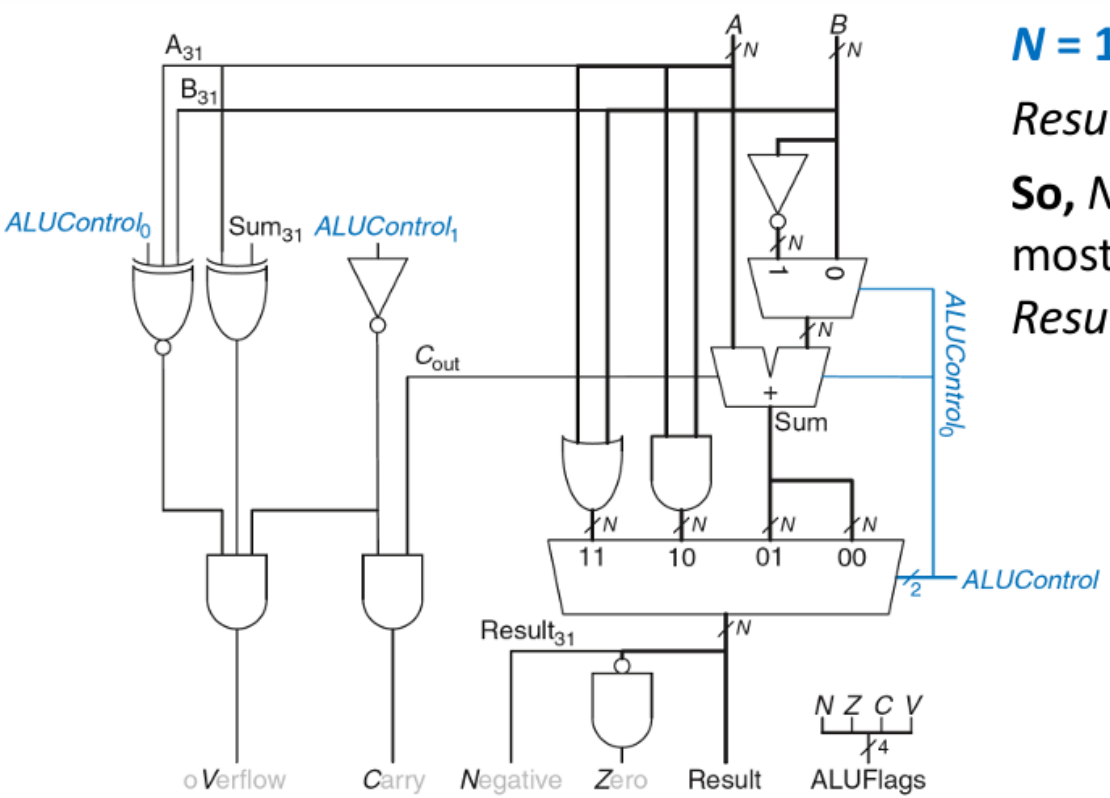
Flag	Description
N	*Result* is **N**egative
Z	*Result* is **Z**ero
C	Adder produces **C**arry out
V	Adder o**V**erflowed



ALU with Status Flags



ALU with Status Flags: Negative

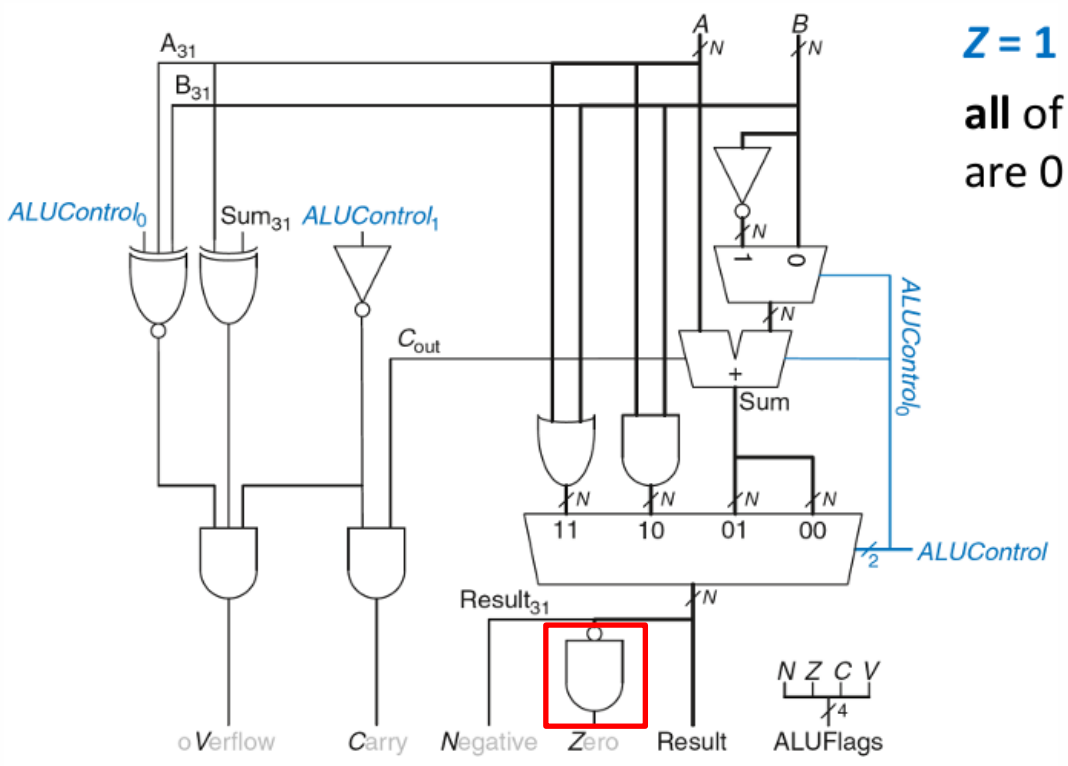


N = 1 if:

Result is **negative**

So, *N* is connected to most significant bit of *Result*.

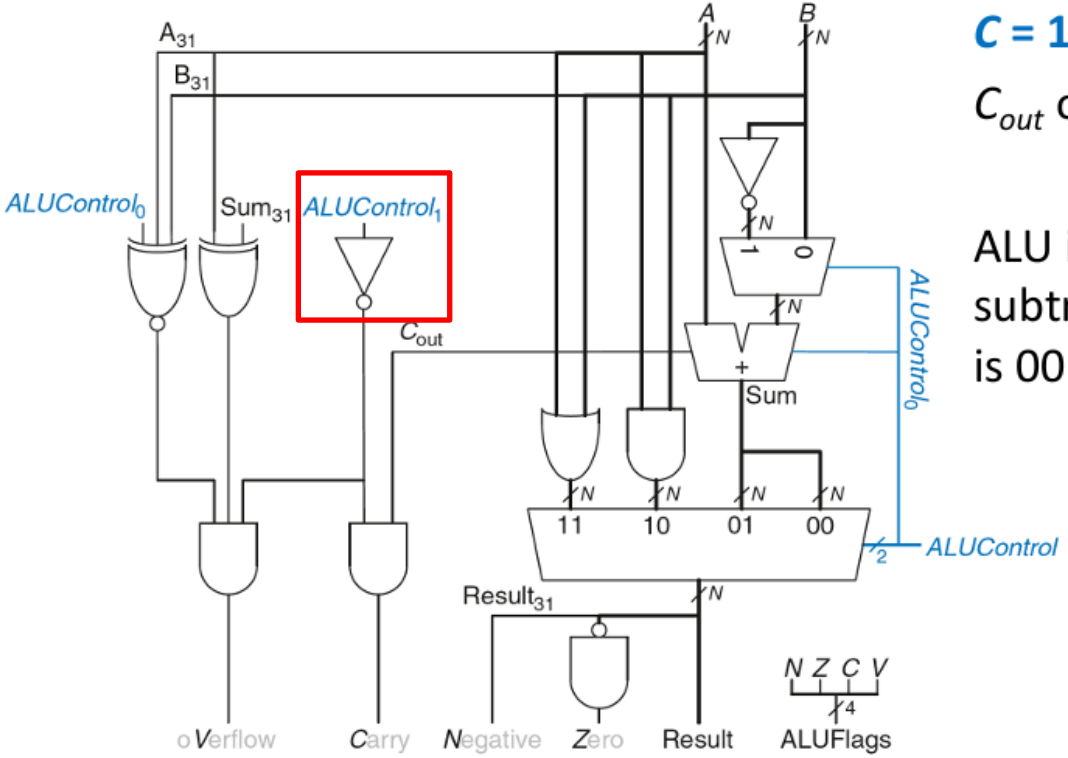
ALU with Status Flags: Zero



Z = 1 if:

all of the bits of *Result* are 0

ALU with Status Flags: Carry



$C = 1$ if:

C_{out} of Adder is 1

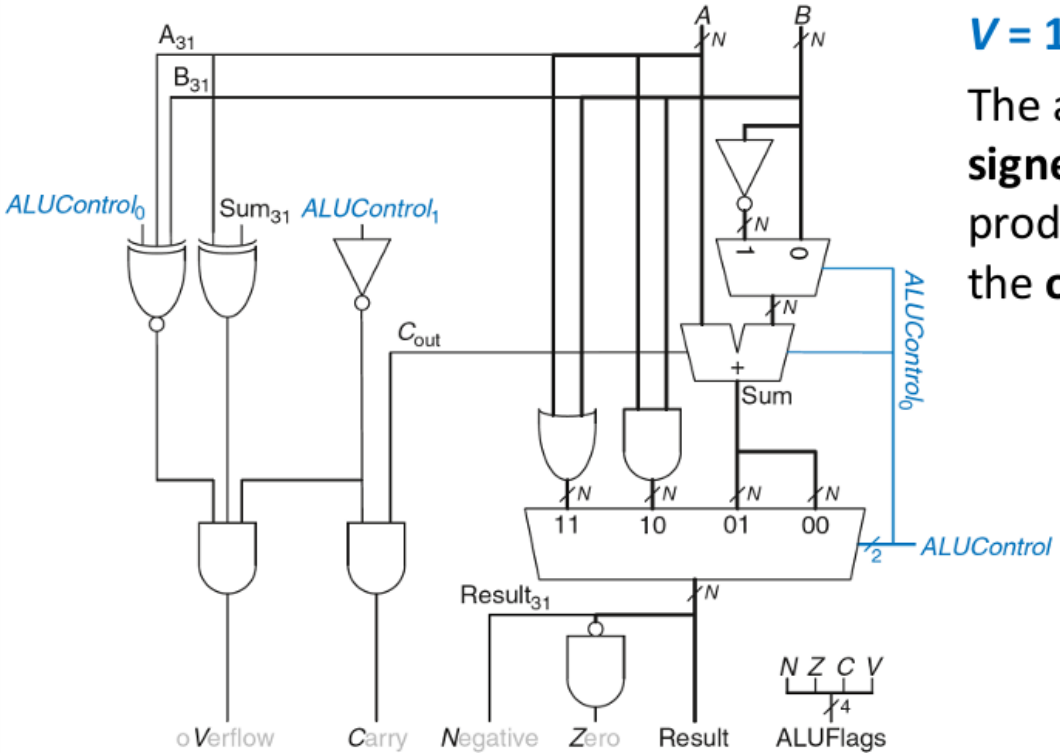
AND

ALU is adding or subtracting (ALUControl is 00 or 01)

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

ALU with Status Flags: oVerflow

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR

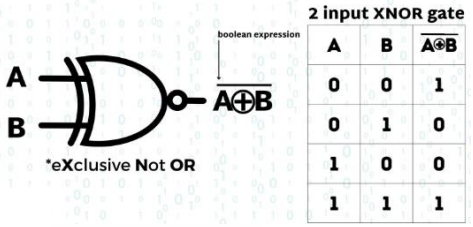


V = 1 if:

The addition of 2 **same-signed numbers** produces a result with the **opposite sign**

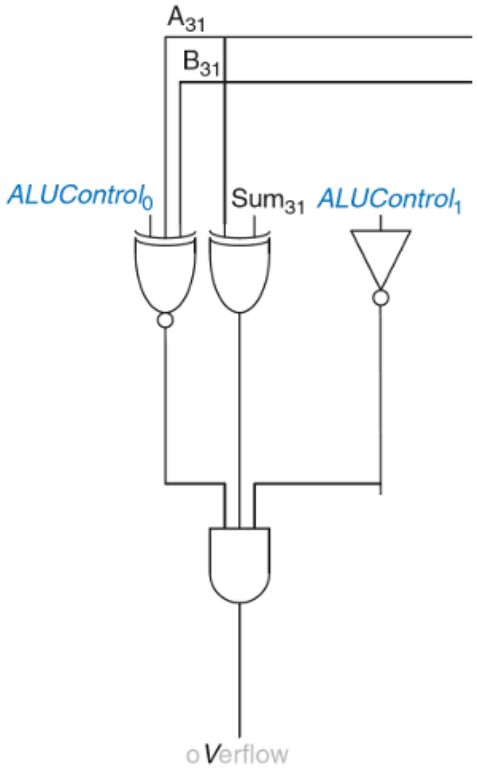
XOR

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0



ALU with Status Flags: oVerflow

$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR



V = 1 if:

ALU is performing addition or subtraction

($ALUControl_1 = 0$)

AND

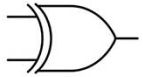
A and Sum have opposite signs

AND

A and B have same signs for addition ($ALUControl_0 = 0$)

A and B have different signs for subtraction ($ALUControl_0 = 1$)

XOR

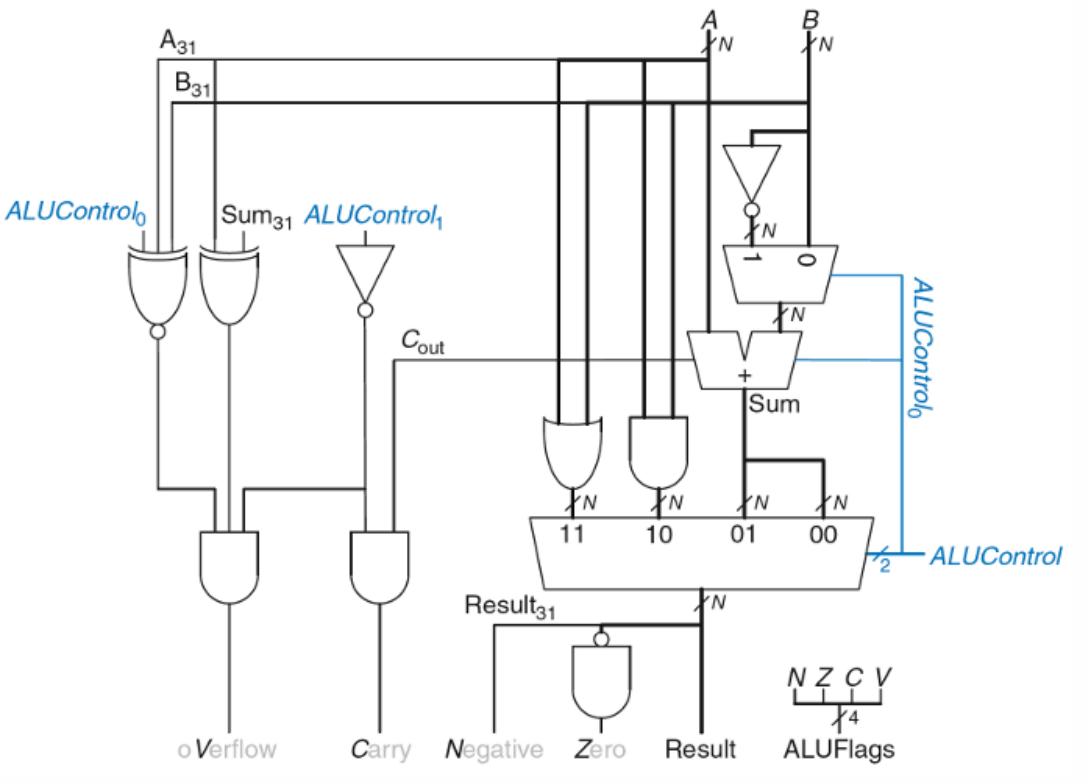


A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

2 input XNOR gate

A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

ALU with Status Flags



Comparison based on Flags

Compare by subtracting and checking flags

Different for signed and unsigned

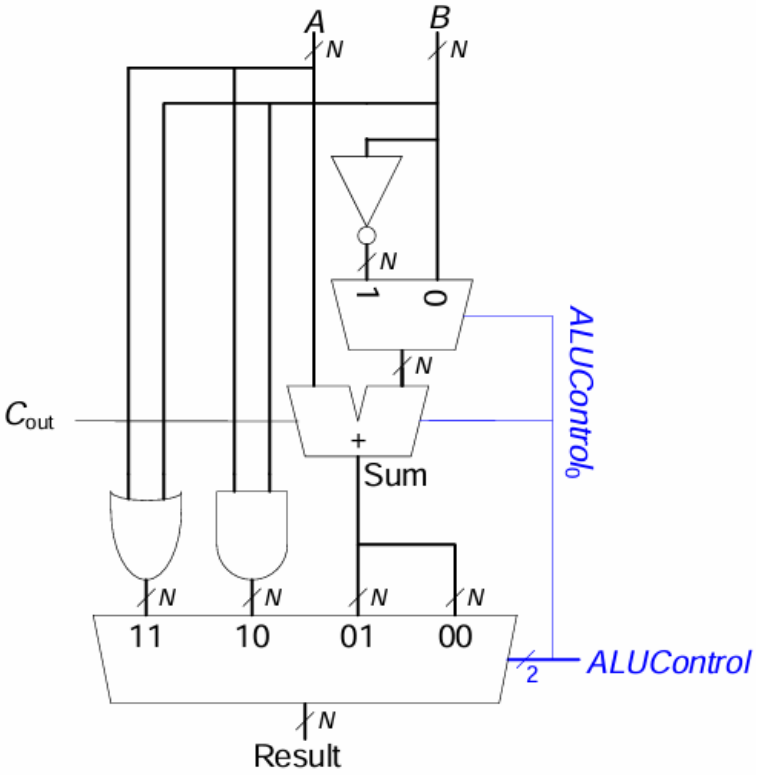
Comparison	Signed	Unsigned
==	Z	Z
!=	~Z	~Z
<	N ^ V	~C
<=	Z | (N ^ V)	Z | ~C
>	~Z & ~(N ^ V)	~Z & C
>=	~(N ^ V)	C

Other ALU Operations

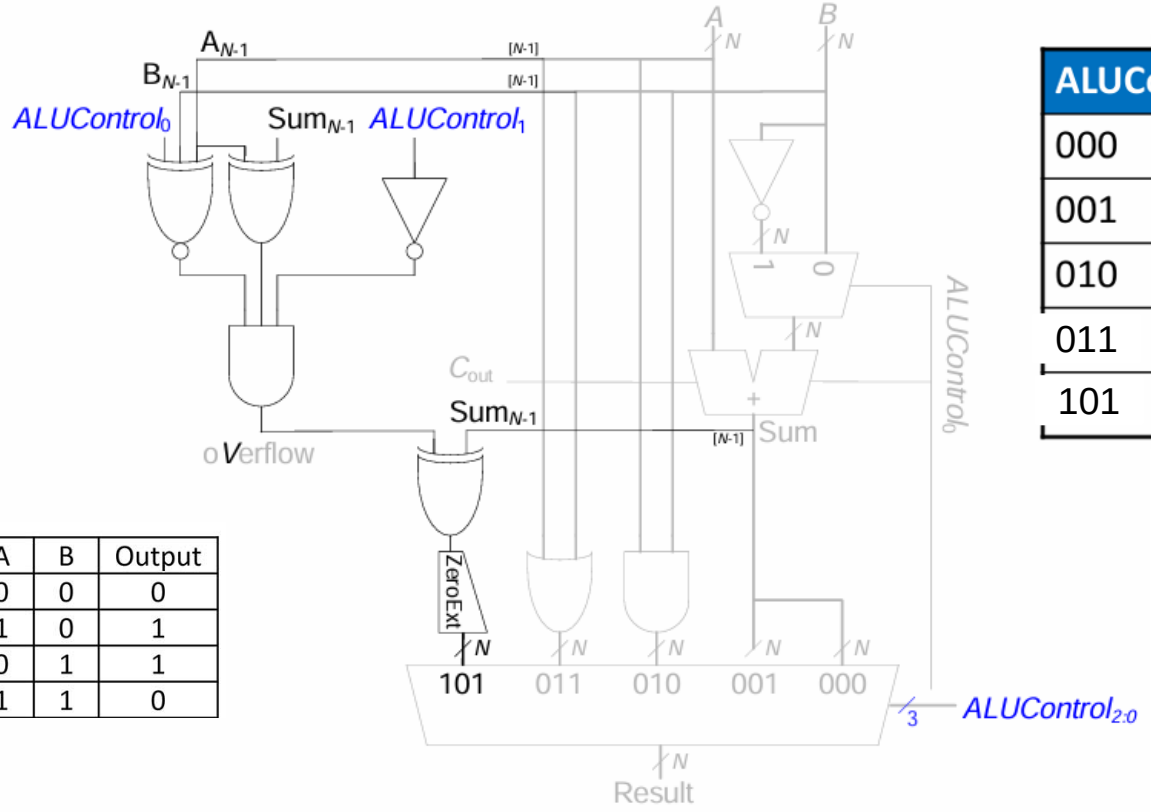
- **Set Less Than** (also called Set if Less Than)
  - **Sets lsb** of result **if A < B**
    - Result = 0000…001 if A < B
    - Result = 0000…000 otherwise
  - Comes in signed and unsigned flavors
- **XOR**
  - Result = A XOR B

Extending ALU: SLT

$ALUControl_{2:0}$	Function
00	add
01	subtract
10	and
11	or
	SLT



Fixing Overflow Error in SLT Logic

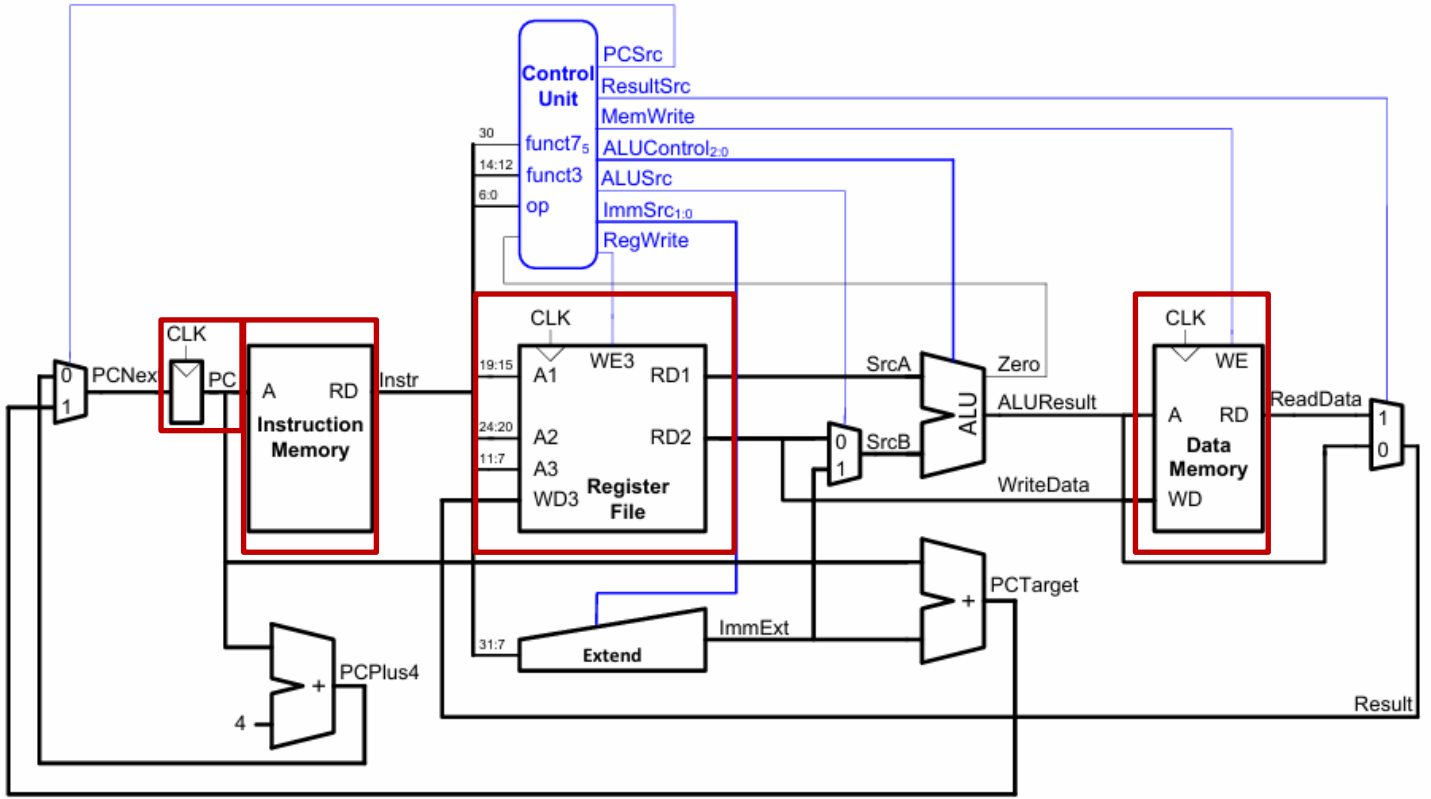


$ALUControl_{2:0}$	Function
000	A + B
001	A - B
010	A & B
011	A | B
101	SLT

XOR

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

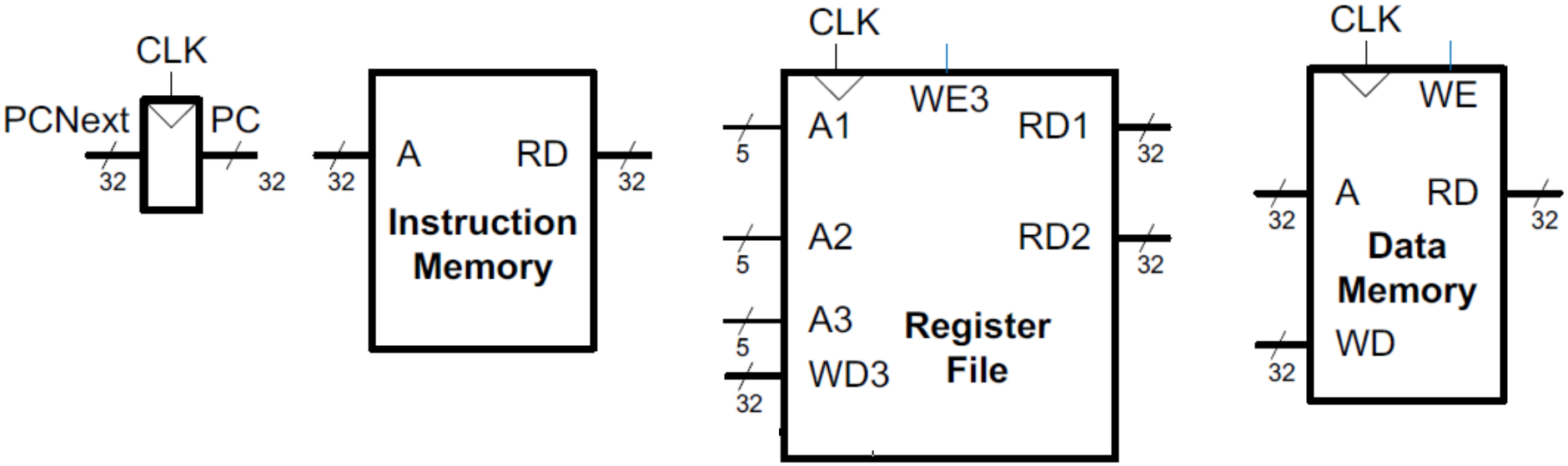
Single-Cycle RISC-V Processor



Architectural State Elements

- Determines everything about a processor
  - Architectural state:
    - 32 registers
    - PC
    - Memory

RISC-V Architectural State Elements



WE: Write Enable

State Elements

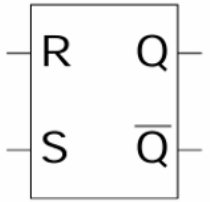
- Outputs of sequential logic depend on current and prior input values – it has **memory**
- **Sequential circuits**
  - Give sequence to events
  - Have memory (short-term)
  - Use feedback from output to input to store information

State Elements

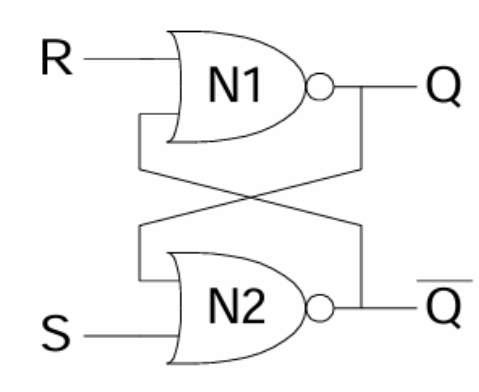
- **State**
  - Everything about the prior inputs to the circuit necessary to predict its future behavior
  - Usually just 1 bit, the last value captured
  - State elements store state
    - SR Latch
    - D Latch
    - D Flip-flop

SR (Set/Reset) Latch

SR Latch Symbol



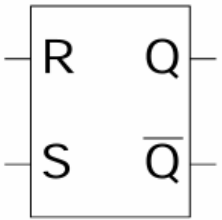
- **Latch**
  - An asynchronous circuit (it doesn't require a clock signal to work)
  - Has two stable states, HIGH ("1"), and LOW ("0")
  - Can be used for storing binary data
- **Set-Reset (S-R) latch**
  - Two NOR gates with a cross-feedback loop
  - The feedback path stores one bit of data as long as the circuit is powered
  - Two inputs (R and S)
  - Two outputs (Q and inverted Q)



SR (Set/Reset) Latch

- **SR** stands for **S**et/**R**eset Latch
  - Stores one bit of state (Q)
- Control what value is being stored with S, R inputs
  - **Set:** Make the output 1
    $S = 1, R = 0, Q = 1$
  - **Reset:** Make the output 0
    $S = 0, R = 1, Q = 0$
  - **Memory:** Retain value
    $S = 0, R = 0, Q = Q_{prev}$

SR Latch Symbol



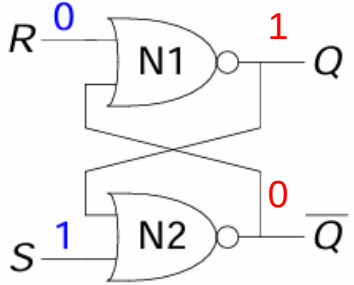
- **Must do something to avoid invalid state (when $S = R = 1$)**

SR (Set/Reset) Latch

Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

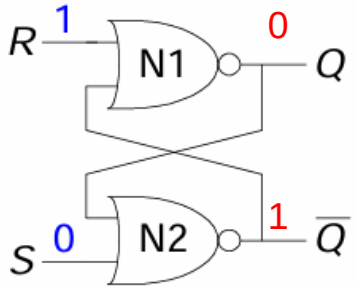
– $S = 1, R = 0$:

Set state



– $S = 0, R = 1$:

Reset state



SR (Set/Reset) Latch

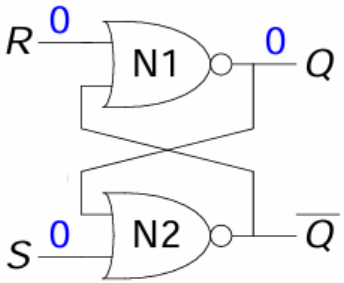
Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

– **$S = 0, R = 0$:**

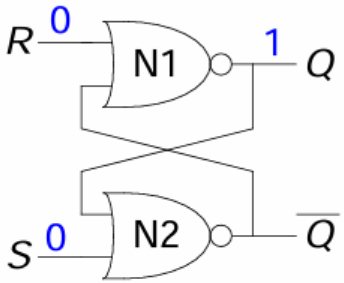
Previous state

then $Q = Q_{prev}$

$Q_{prev} = 0$



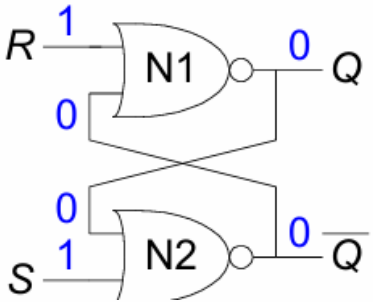
$Q_{prev} = 1$



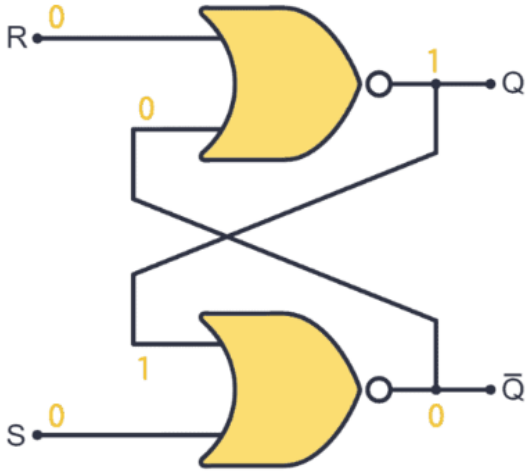
– **$S = 1, R = 1$:**

Invalid state

then $Q = 0, \bar{Q} = 0$

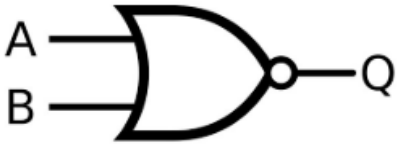


SR (Set/Reset) Latch



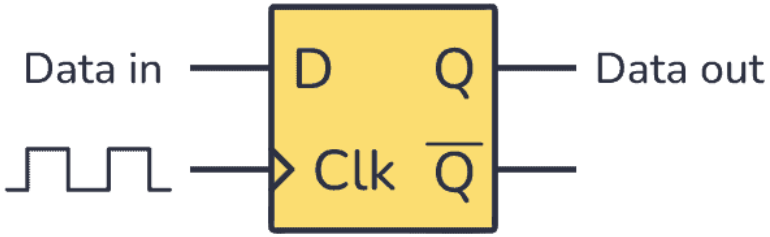
- Set-Reset (S-R) latch
  - NOR gate gives 1 when both of its inputs are “0”
  - When both inputs S and R are equal to “0”, the output Q remains the same as it was (saves the previous value)

Input S	Input R	Output Q
0	0	Previous State
0	1	0
1	0	1
1	1	0 (Invalid)



Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

D Flip Flop

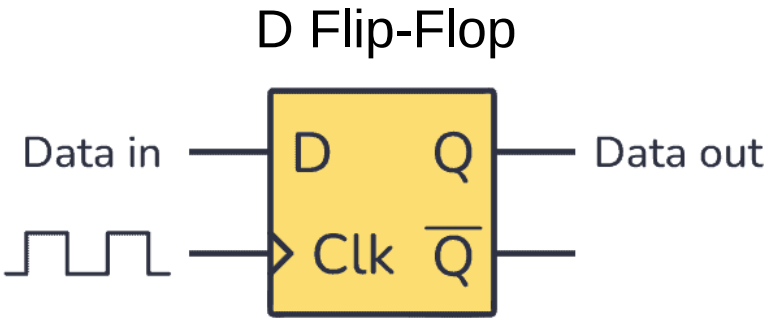


D Flip-Flop

- Flip Flops
  - A flip-flop is a state element
  - State element: A circuit component that can hold a value
  - Store one bit (1 or 0) on flip flop output
  - Synchronous circuit that need a clock signal (Clk)
- D Flip-Flop
  - The D Flip-Flop will only store a new value from the D input when the clock goes from 0 to 1 (rising edge) or 1 to 0 (falling edge)
  - Commonly used as a basic building block to create counters or memory blocks such as shift register

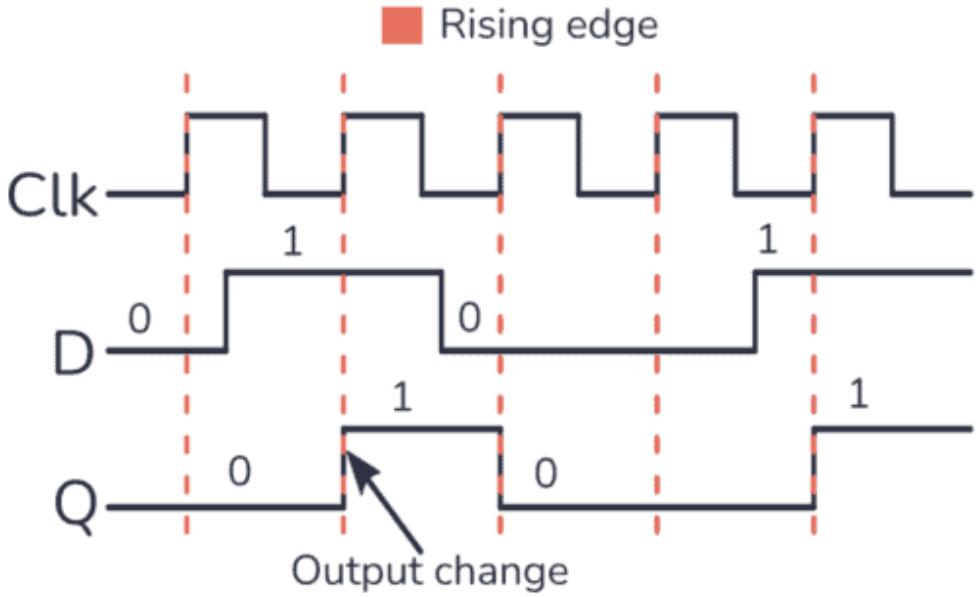
D Flip Flop

- Samples D on rising edge of CLK
  - When CLK rises from 0 to 1, D passes through to Q
  - Otherwise, Q holds its previous value
  - Q changes only on rising edge of CLK
  - Called edge-triggered
  - Activated on the clock edge



D Flip-Flop

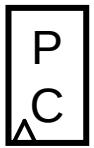
- The output Q only changes to the value of the D input at the moment the clock goes from 0 to 1



State Elements

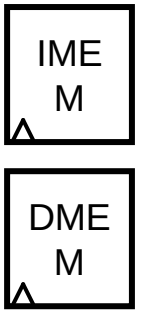
- State elements required by RV32I ISA
  - During CPU execution, each RV32I instruction reads and/or updates these state elements

Program Counter

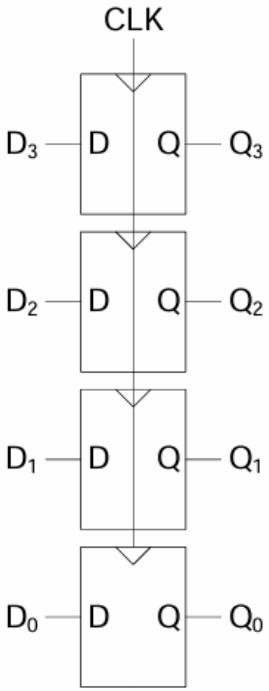


Register File `Reg`

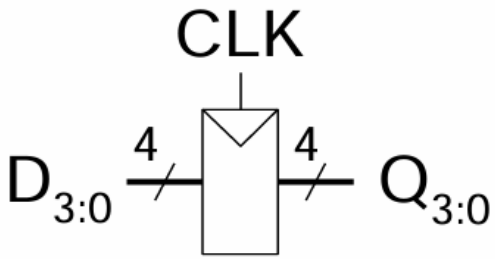
Memory `MEM`



Registers: One or More Flip-Flops

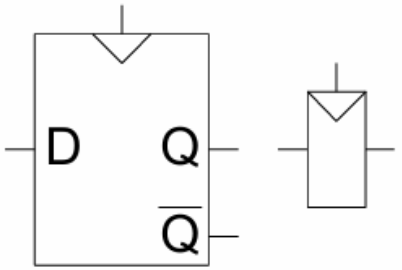


4-bit Register



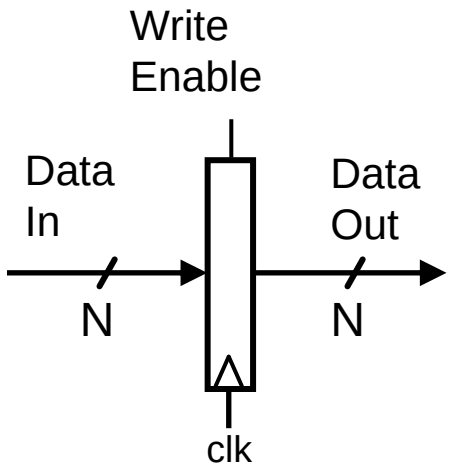
4-bit Register

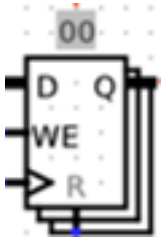
D Flip-Flop Symbols



State Elements

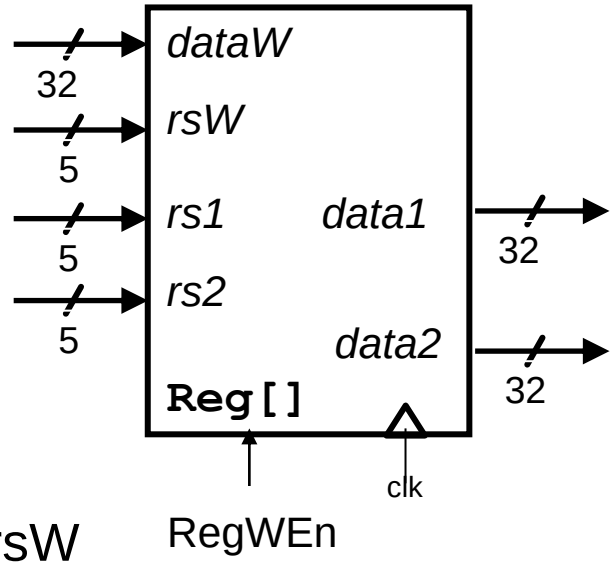
- **Program Counter**
  - The program counter is a 32-bit register
  - **Input**
    - N-bit data input bus
    - Write Enable: “Control” bit (1: asserted/high 0: de-asserted/0)
  - **Output**
    - N-bit data output bus
  - **Behavior**
    - If write enable is 1 on the rising clock edge, set Data Out = Data in
    - At all other times, Data out will not change, it will output it current value





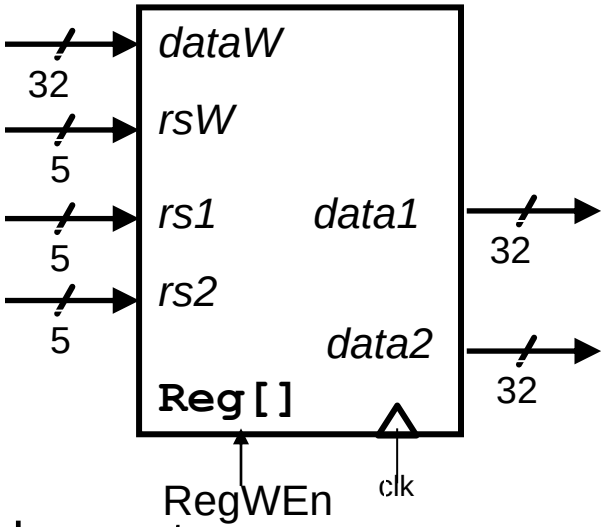
A register in Logisim

State Elements



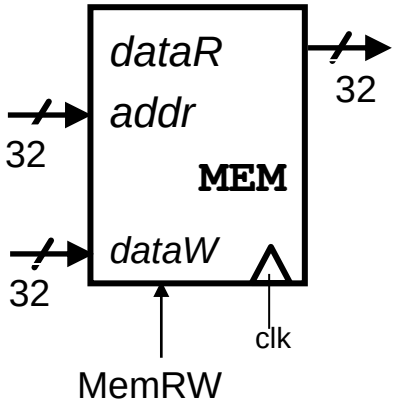
- **Register File**
  - The Register File (RegFile) has 32 registers
  - **Input**
    - One 32-bit input data bus, dataW
    - Three 5-bit select busses, rs1, rs2, and rsW
  - **Output**
    - Two 32-bit output data busses, data1 and data2

State Elements



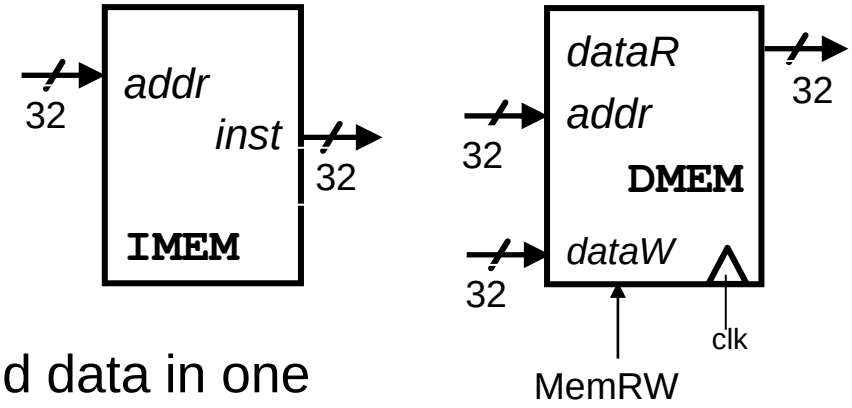
- **Register File**
  - Registers are accessed via their 5-bit register numbers
    - R[rs1]: rs1 selects register to put on data1 bus out
    - R[rs2]: rs2 selects register to put on data2 buts out
    - R[rd]: rsW selects register to be written via dataW when RegWEn = 1
  - Clock behavior: Write operation occurs on rising clock edge
    - Clock input only a factor on write!
    - All read operations behave like a combinational block
      - If rs1, rs2 valid, then data1, data2 valid after access time

State Elements



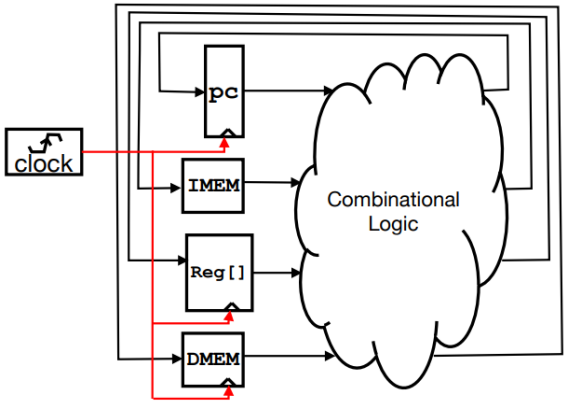
- **Memory**
  - 32-bit byte-addressed memory space
  - Memory access with 32-bit words
  - Memory words are accessed as follows
    - **Read**: Address addr selects word to put on dataR bus
    - **Write**: Set MemRW = 1
      Address addr selects word to be written with dataW bus
  - Like RegFile, clock input is only a factor on write
    - If MemRW = 1, write occurs on rising clock edge
    - If MemRW = 0 and addr valid, then dataR valid after access time

State Elements



- **Two Memories (IMEM, DMEM)**
  - Memory holds both instructions and data in one contiguous 32-bit memory space
  - The processor will use two “separate” memories
    - **IMEM**: A read-only memory for fetching instruction
    - **DMEM**: A memory for loading (read) and storing (write) data words
  - Because IMEM is read-only, it always behaves like a combinational block
    - If addr valid, then instr valid after access time

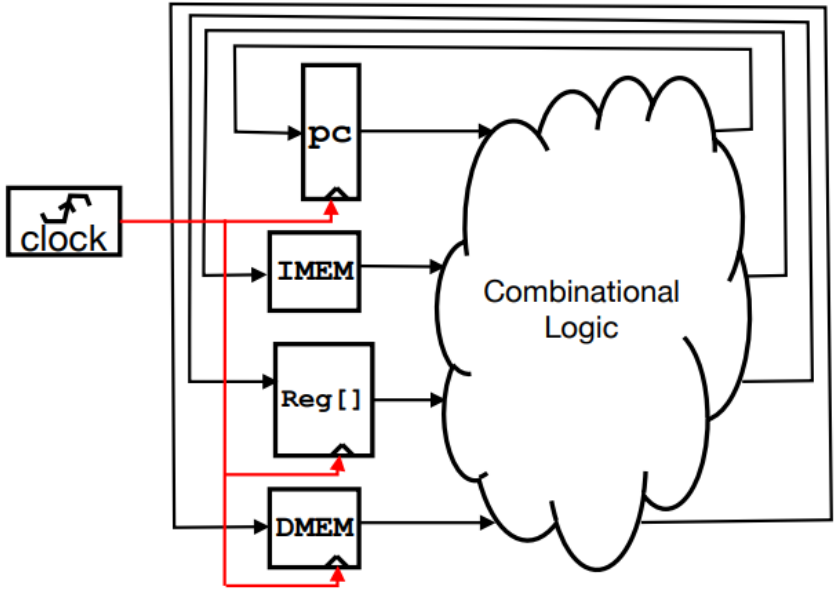
Single-Cycle RISC-V Processor



- **Task: “Execute an instruction”**
  - All necessary operations starting with fetching the instruction
  - Problem:
    - A single “monolithic’ block would be bulky and inefficient
  - Solution:
    - Break up the process into stages, then connect the stages to create the whole datapath
    - Smaller stages are easier to design!
    - Modularity: Easy to optimize one stage without touching the others

RISC-V Single Cycle Processor

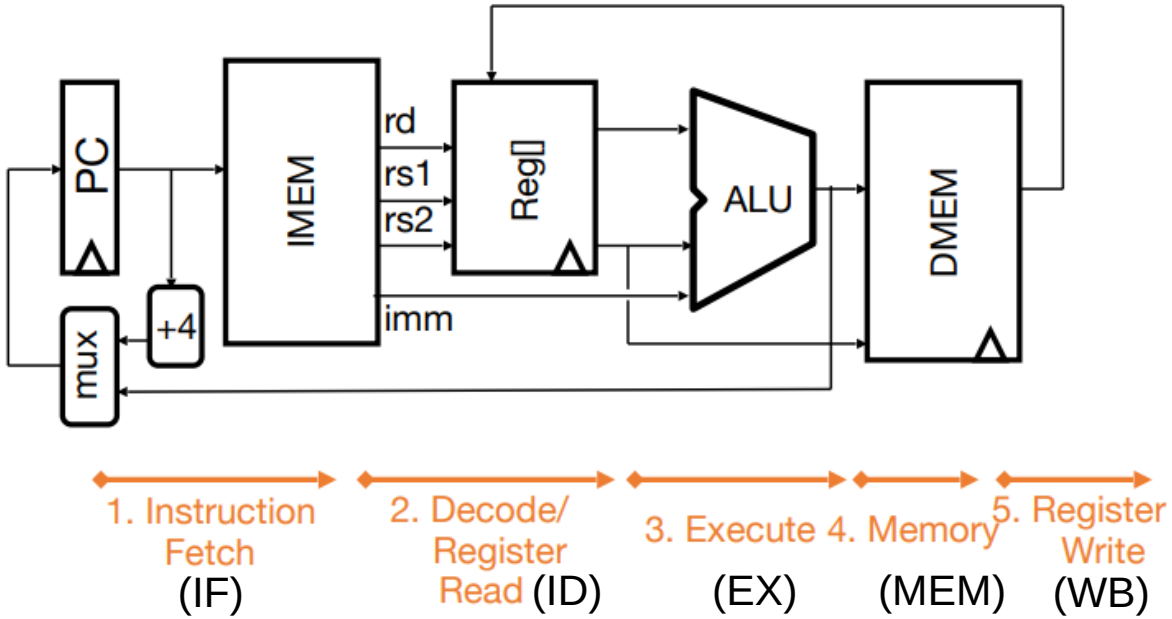
- The CPU comprises two types of circuit
  - One-instruction-per-cycle
  - One every tick of the clock, the computer executes one instruction
  - At the rising clock edge
    - All the state elements are updated with the combinational logic outputs
    - Execution moves to the next clock cycle



Single-Cycle RISC-V Processor

- **The single-cycle processor**
  - All stages of one RV32I instruction execute within the same clock cycle

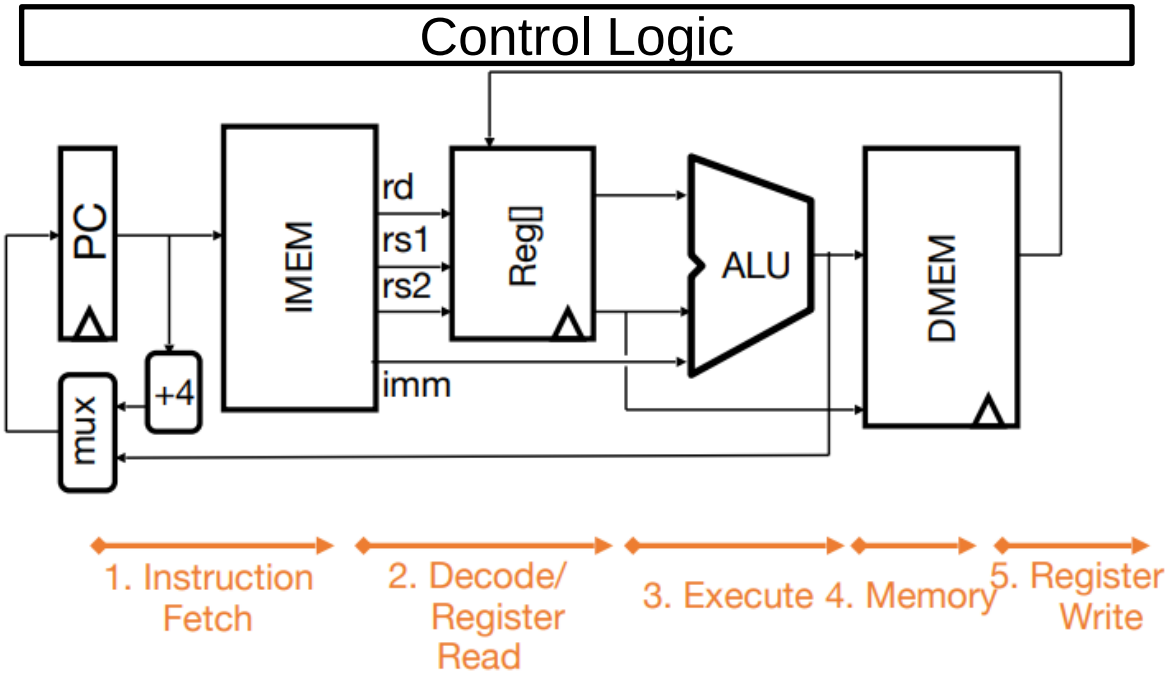
5 basic stages of instruction execution



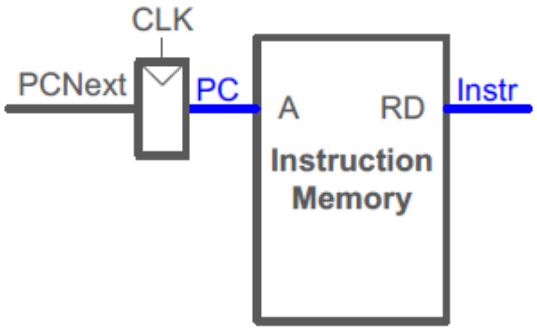
Single-Cycle RISC-V Processor

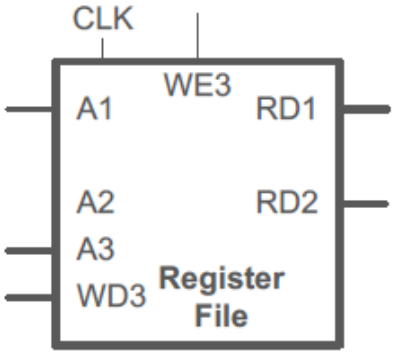
- The **control logic** selects "needed" datapath lines based on the instruction
  - MUX selector
  - ALU op selector
  - Write Enable …

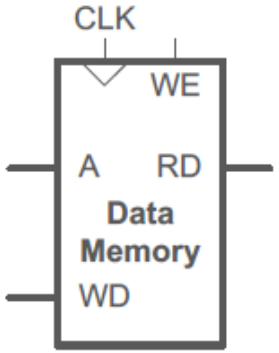
Not all instructions need all 5 stages



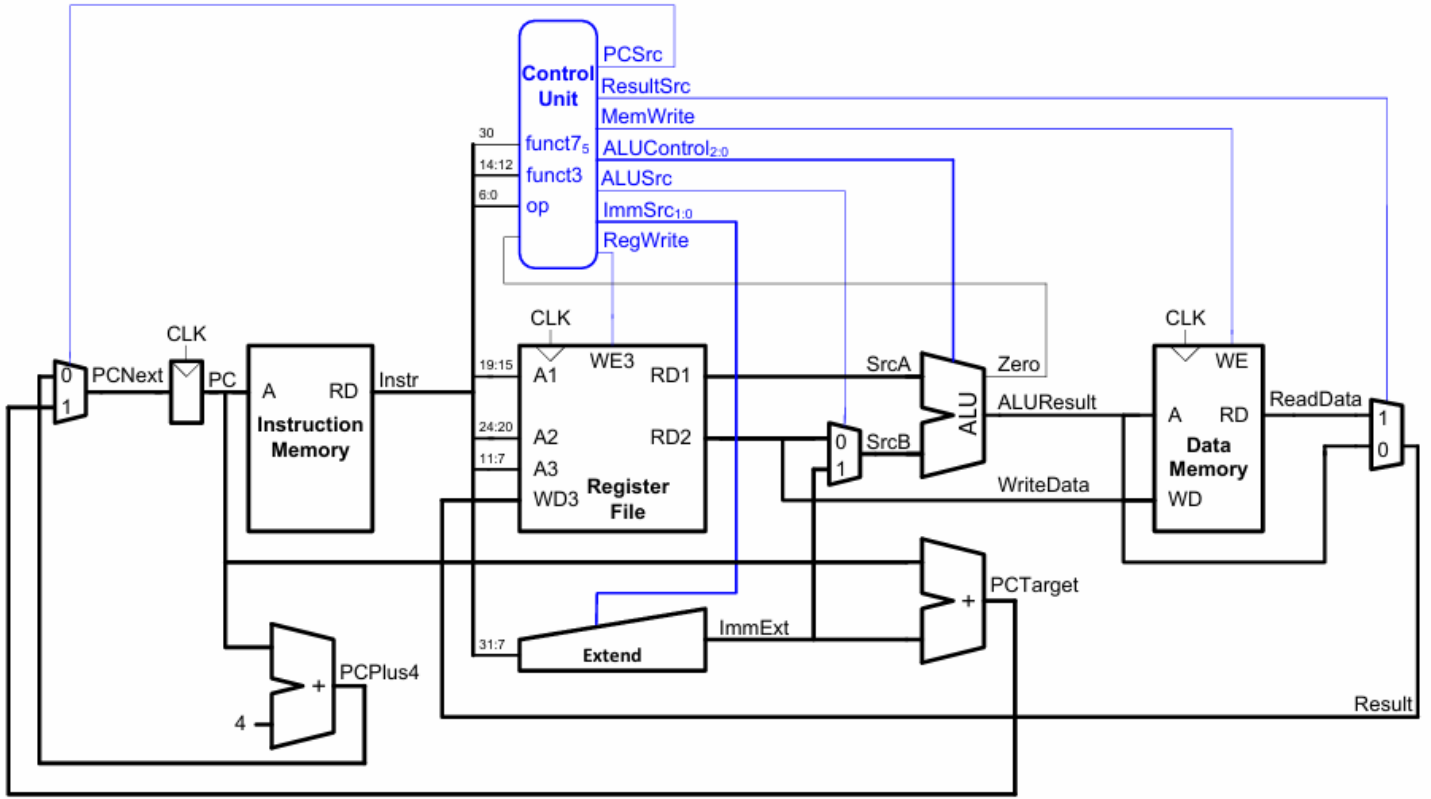
Single-Cycle RISC-V Datapath







Single-Cycle RISC-V Processor



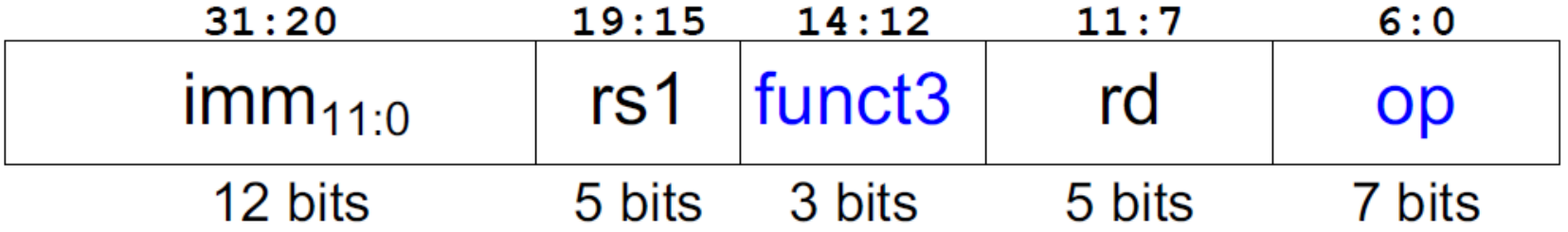
Single-Cycle RISC-V Processor

- Datapath: start with `lw` instruction
- Example:

```
lw t2, -8(s3)
```

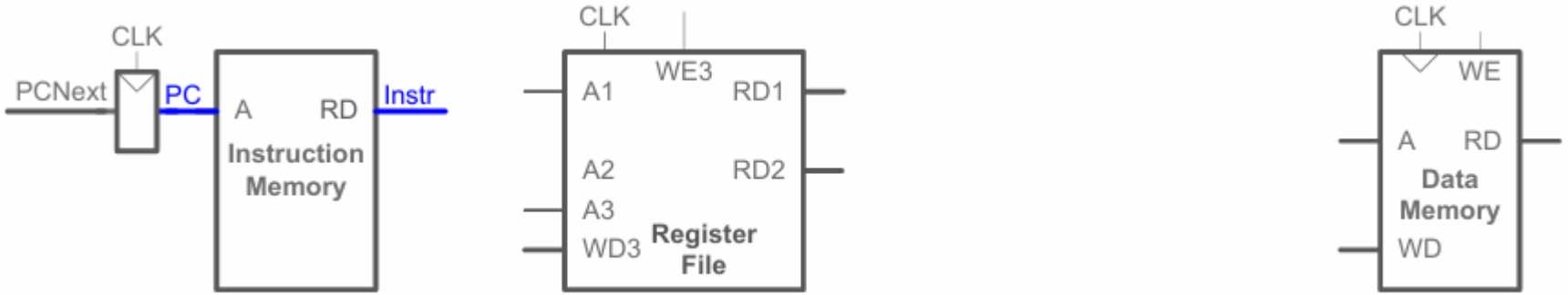
`lw rd, imm(rs1)`

I-Type



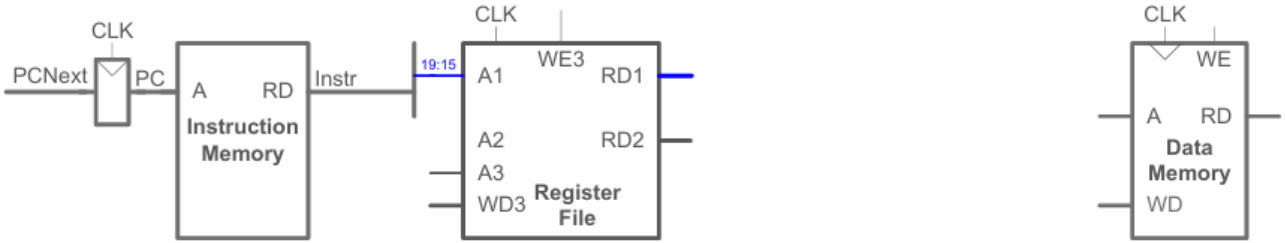
Single-Cycle Datapath: `lw` fetch

STEP 1: Fetch instruction



Single-Cycle Datapath: `lw` Reg Read

STEP 2: Read source operand (**rs1**) from RF



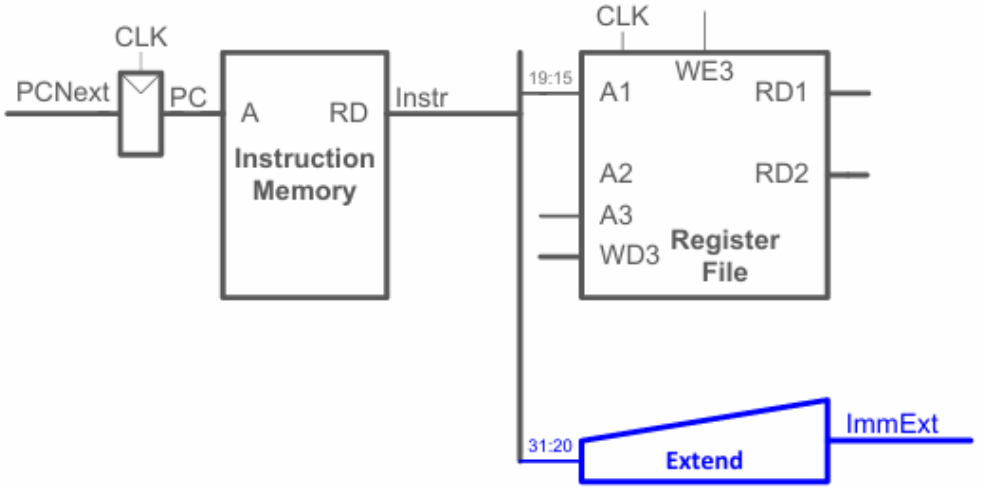
I-Type

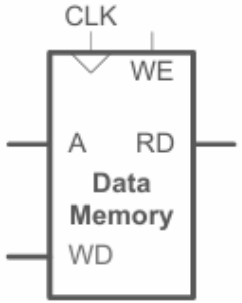
31:20	19:15	14:12	11:7	6:0
$imm_{11:0}$	rs1	funct3	rd	op
12 bits	5 bits	3 bits	5 bits	7 bits

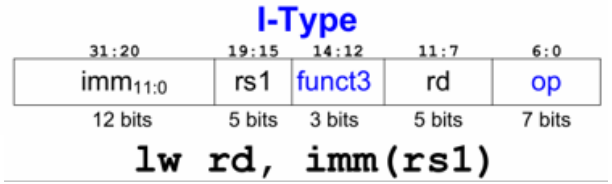
```
lw rd, imm(rs1)
```

Single-Cycle Datapath: `lw` Immediate

STEP 3: Extend the immediate



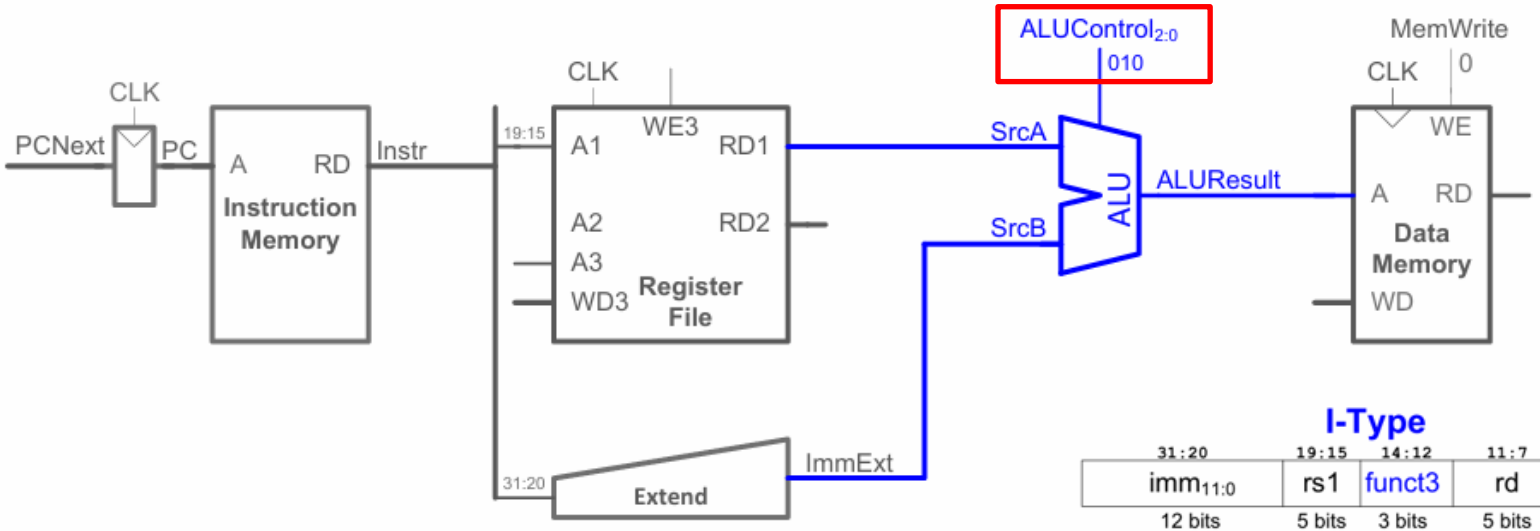




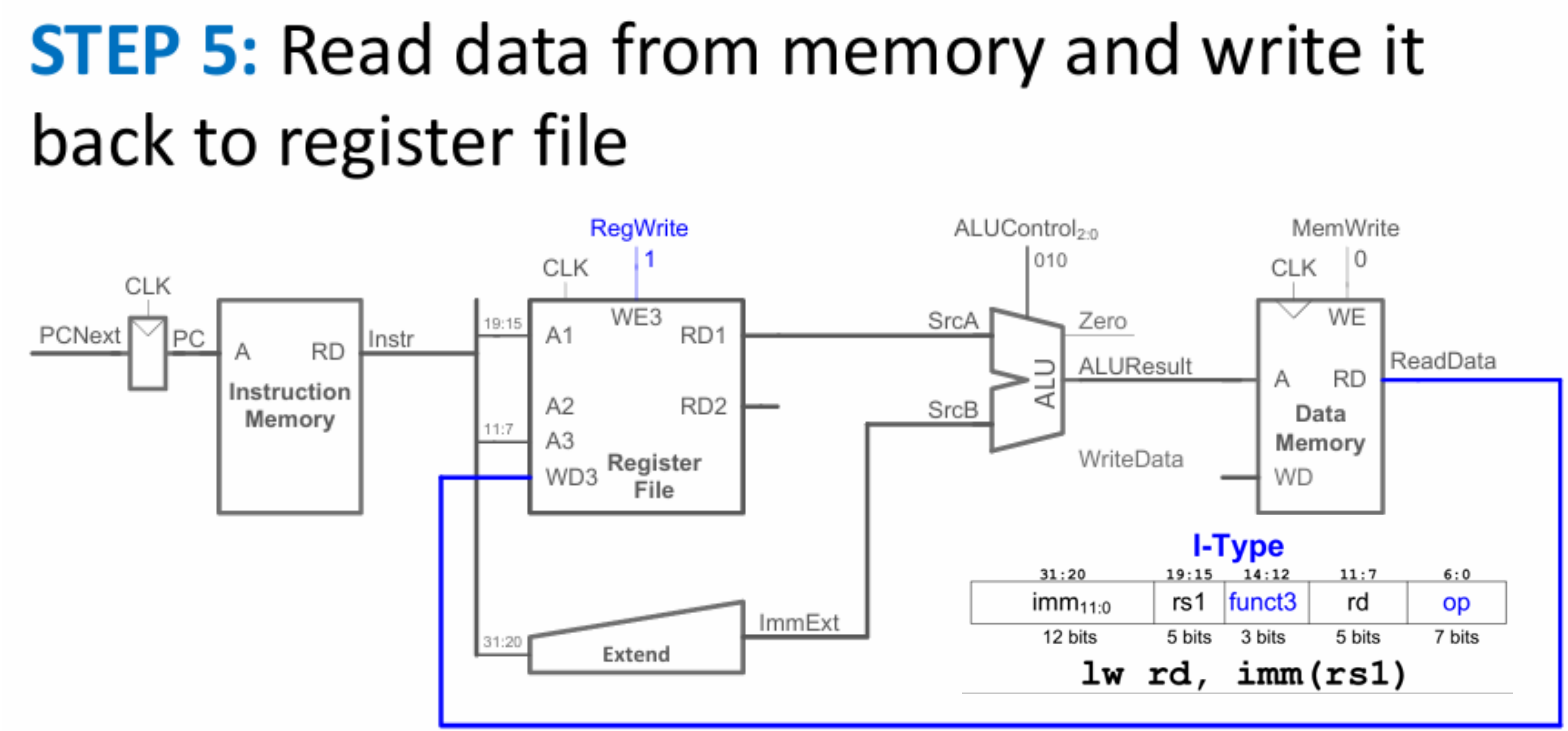
Single-Cycle Datapath: `lw` Address

$ALUControl_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

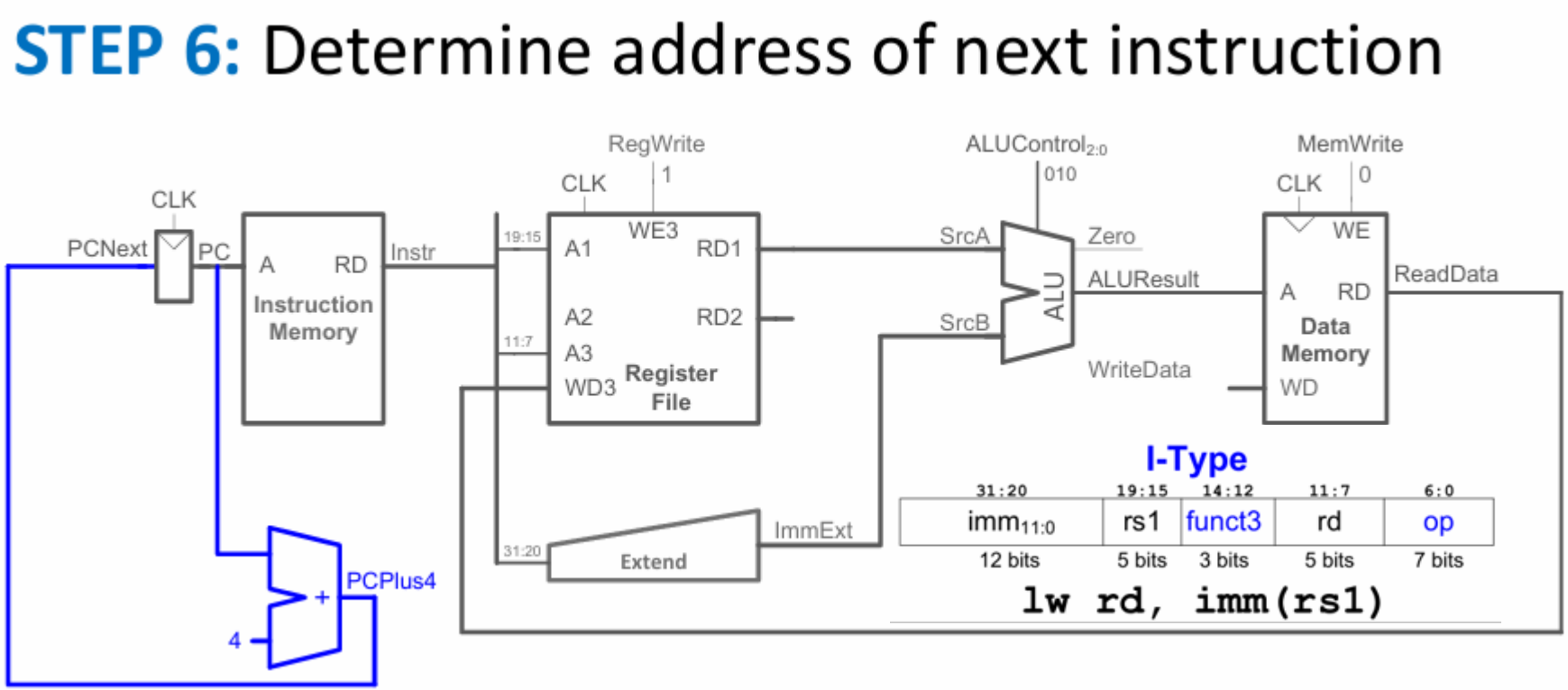
STEP 4: Compute the memory address



Single-Cycle Datapath: `LDR` Mem Read



Single-Cycle Datapath: PC Increment



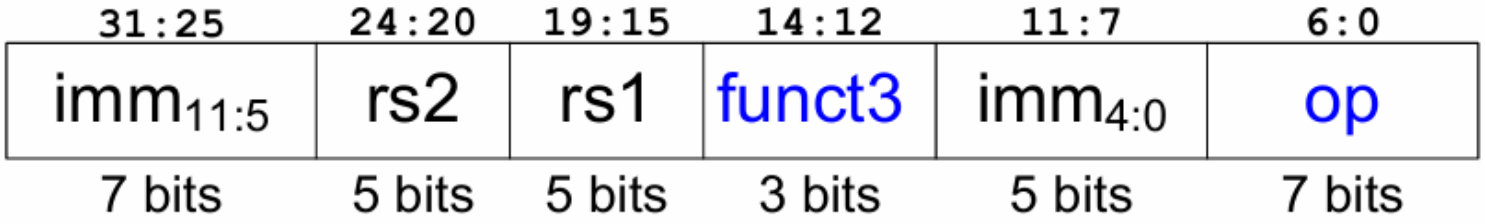
Single-Cycle Datapath: `SW`

Expand datapath to handle `sw`:

- Write data in `rs2` to memory
- **Example:** `sw t2, 0xc(s3)`

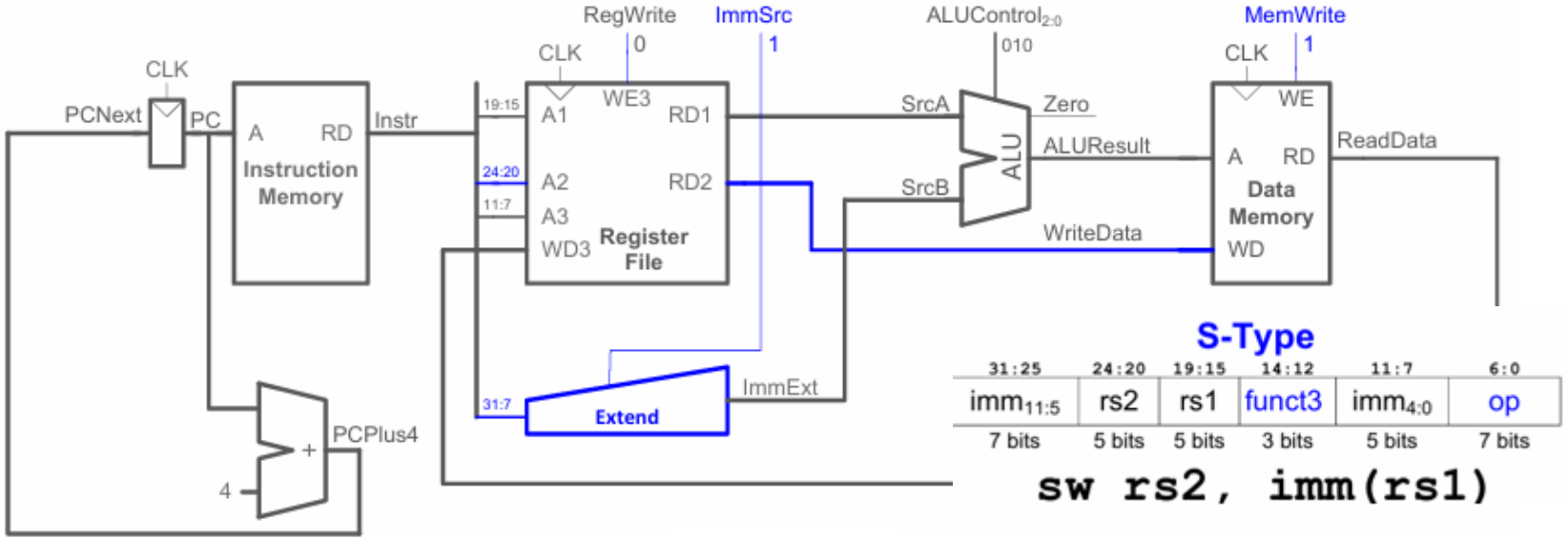
  `sw rs2, imm(rs1)`

S-Type



Single-Cycle Datapath: Data-Processing

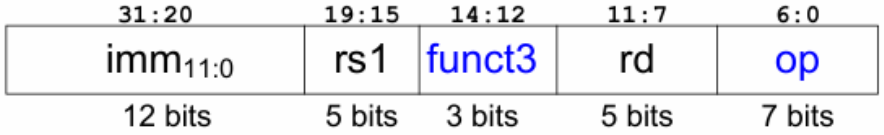
- **Immediate:** now in {instr[31:25], instr[11:7]}
- **Add control signals:** ImmSrc, MemWrite



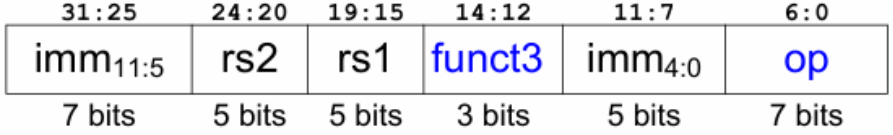
Single-Cycle Datapath: Immediate

ImmSrc	ImmExt	Instruction Type
0	{{20{instr[31]}}, instr[31:20]}	I-Type
1	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type

I-Type



S-Type

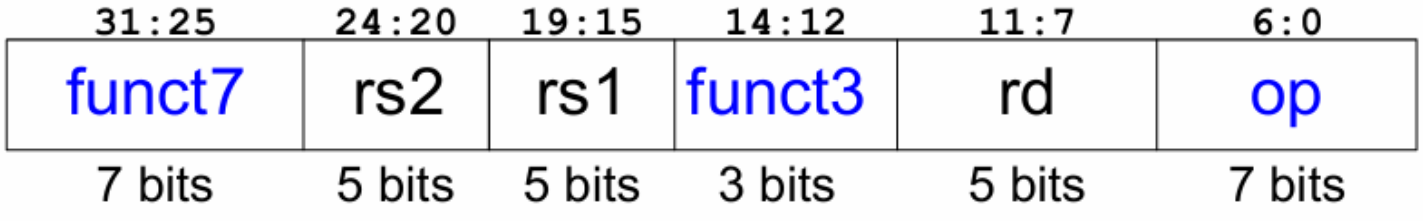


Single-Cycle Datapath: R-Type

- **Instructions:** `add, sub, and, or, slt,`
- **Example:** `add s1, s2, s3`

  `op rd, rs1, rs2`

R-Type



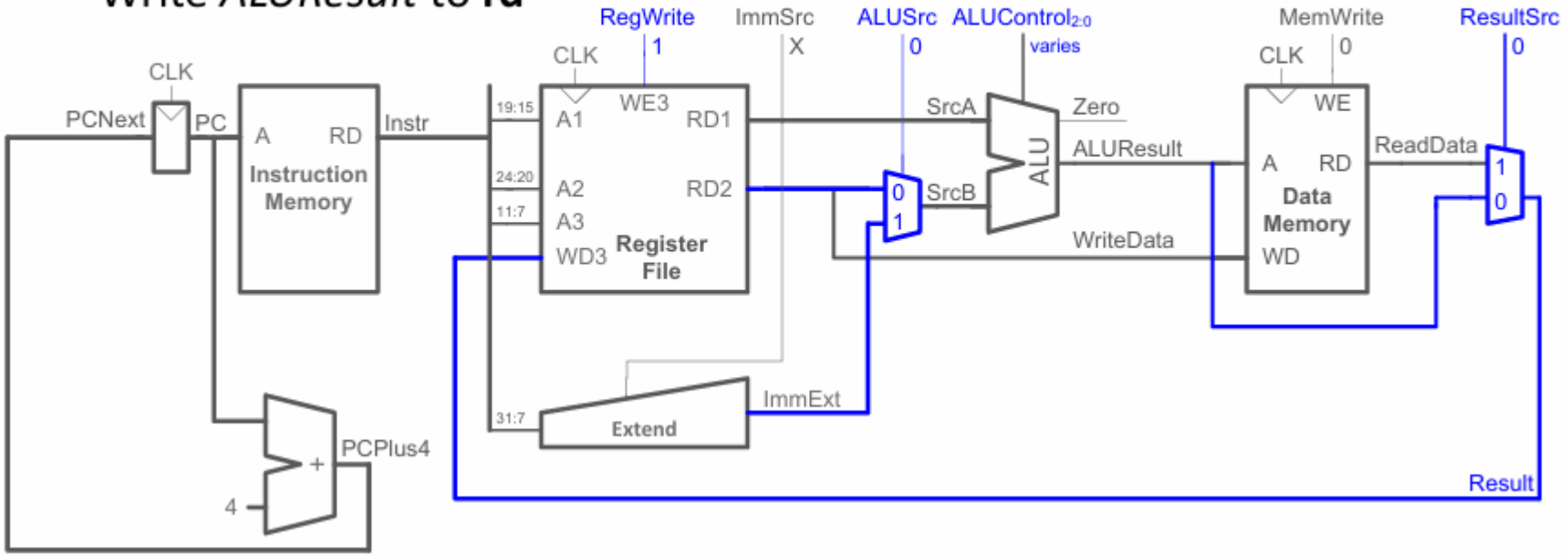
Single-Cycle Datapath: R-Type

R-Type

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

```
add rd, rs1, rs2
```

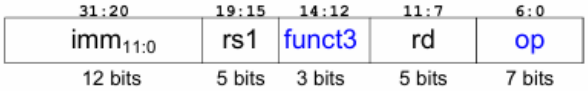
- Read from **rs1** and **rs2** (instead of **imm**)
- Write *ALUResult* to **rd**



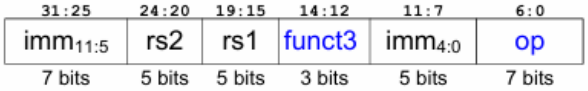
Single-Cycle Datapath: `immExt`

$ImmSrc_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type

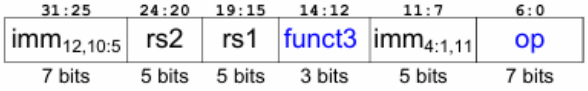
I-Type



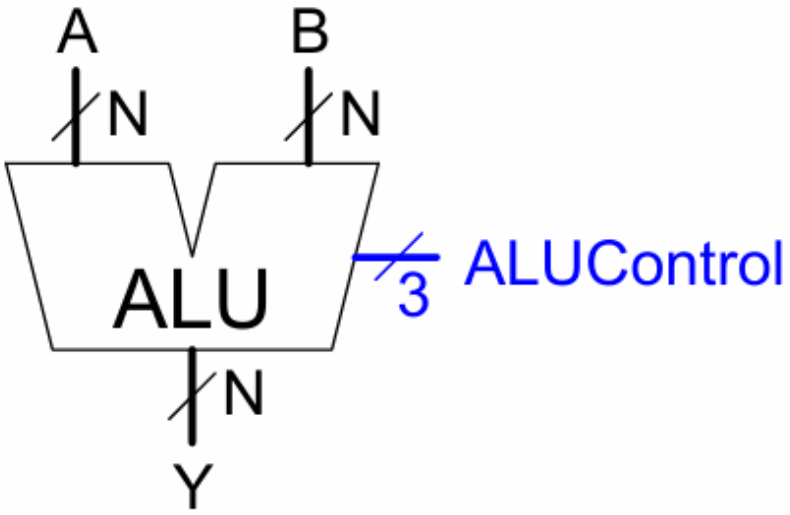
S-Type



B-Type

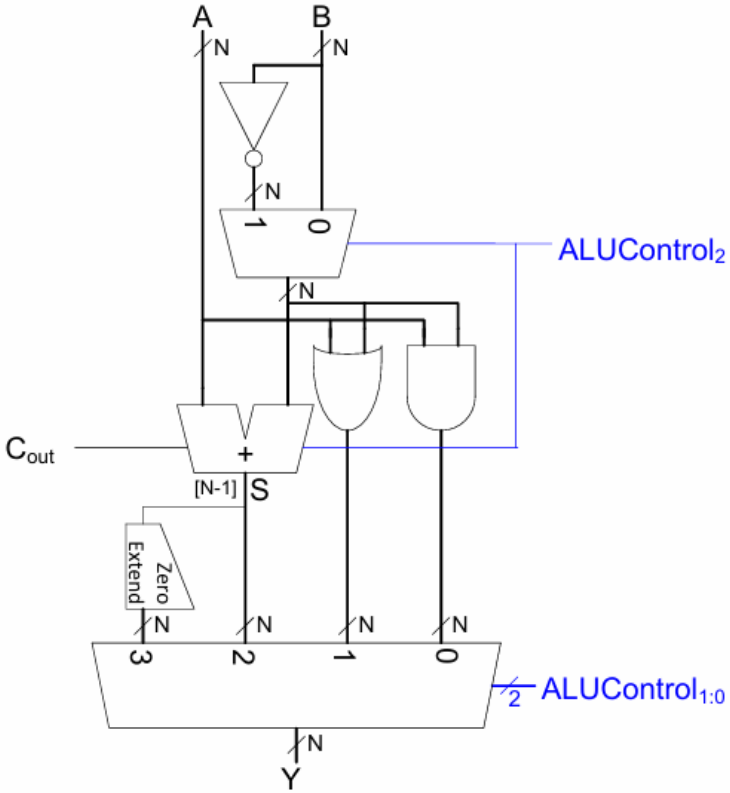


Review: ALU



$ALUControl_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

Review: ALU



$ALUControl_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

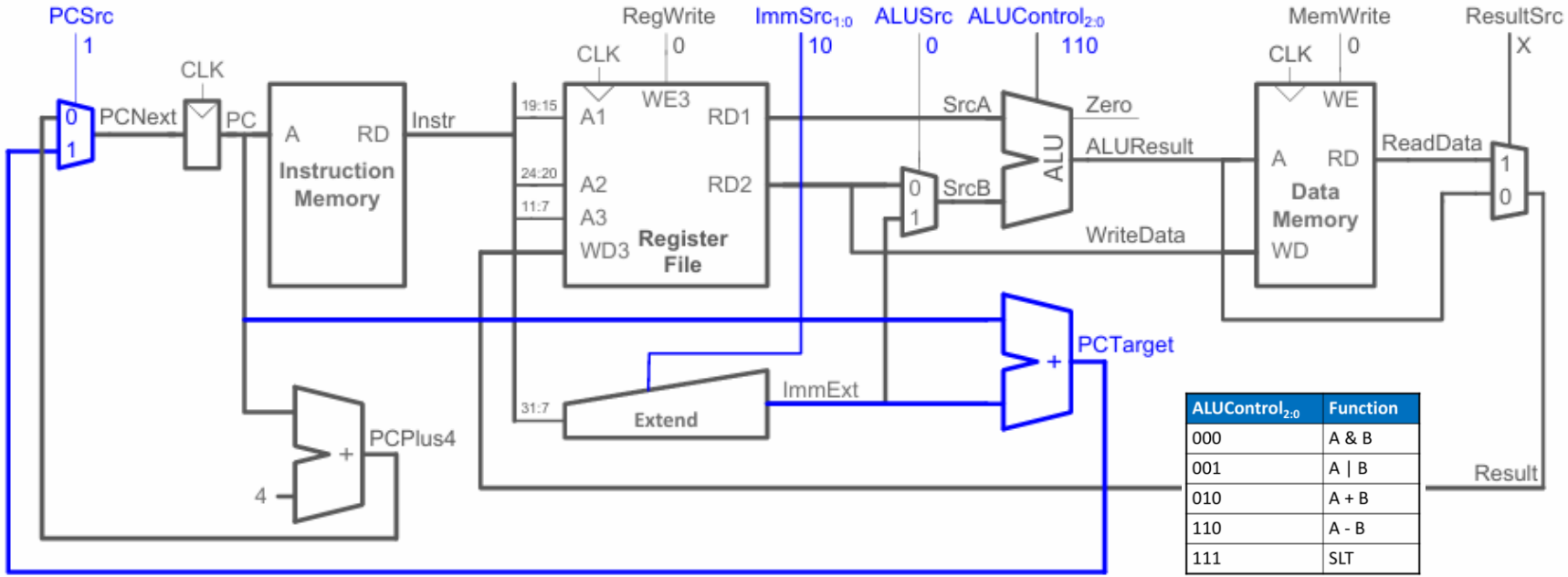
Single-Cycle Datapath: `beq`

B-Type

31:25	24:20	19:15	14:12	11:7	6:0
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

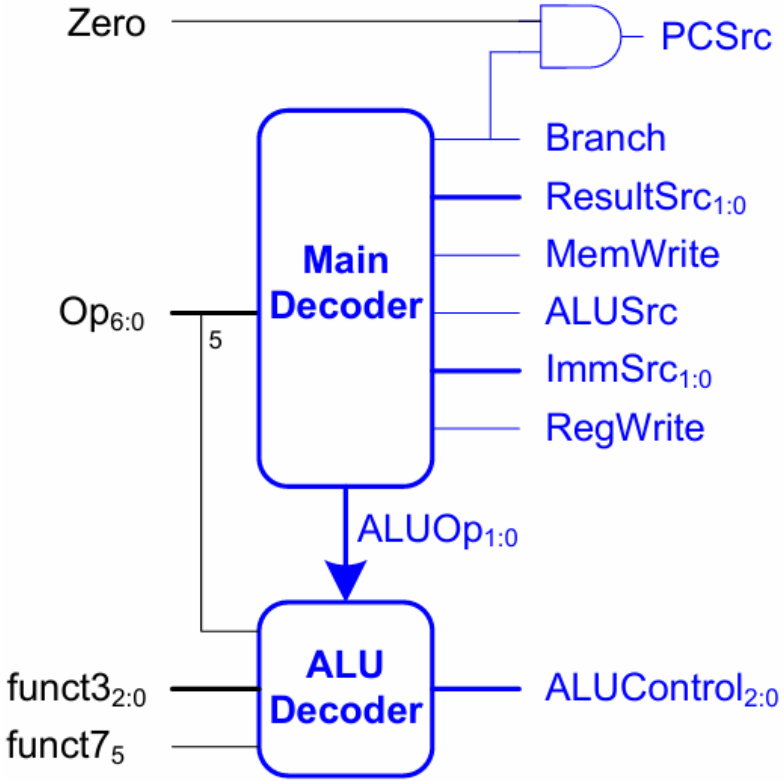
```
beq rs1, rs2, Label
```

Calculate branch target address: PCTarget = PC + imm



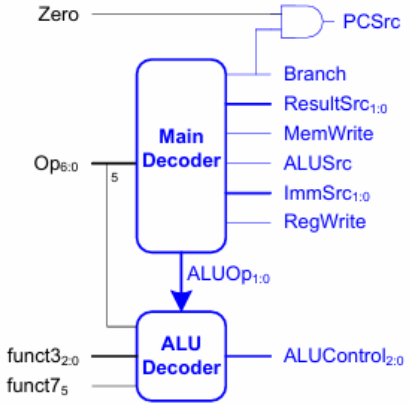
$ALUControl_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

Single-Cycle Control



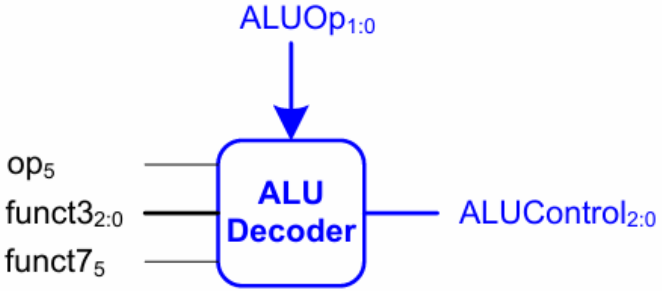
Control Unit: Main Decoder

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	**lw**	1	00	1	0	1	0	00
35	**sw**	0	01	1	1	X	0	00
51	**R-type**	1	XX	0	0	0	0	10
99	**beq**	0	10	0	0	X	1	01



Single-Cycle Control: ALU Decoder

ALUOp		funct3	$funct7_5$	Instruction	$ALUControl_{2:0}$
00		010	X	`lw, sw`	010 (add)
01		000	X	`beq`	110 (subtract)
10		000	0	`add`	010 (add)
		000	1	`sub`	110 (subtract)
		010	0	`slt`	111 (set less than)
		110	0	`or`	001 (or)
		111	0	and	000 (and)



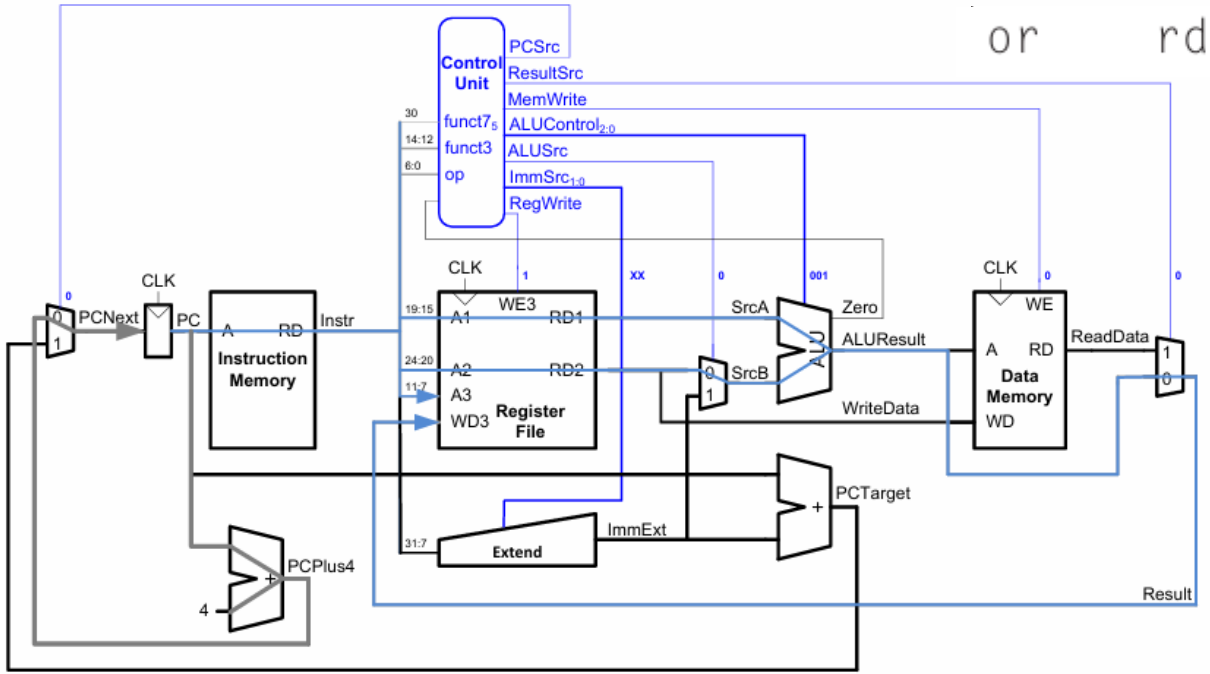
Example: `or`

R-Type

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

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
51	**R-type**	1	XX	0	0	0	0	10

```
or    rd, rs1, rs2
```



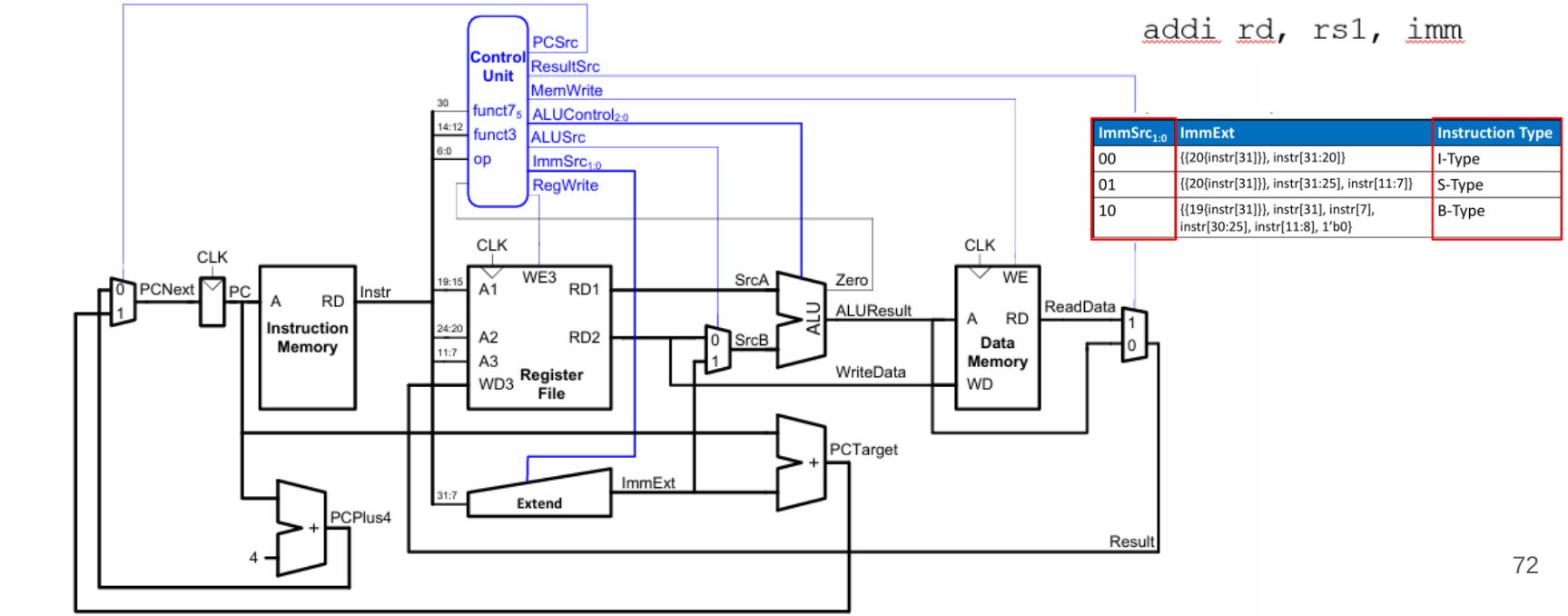
$ALUControl_{2:0}$	Function
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

Extended Functionality: `addi`

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
3	**lw**	1	00	1	0	1	0	00
35	**sw**	0	01	1	1	X	0	00
51	**R-type**	1	XX	0	0	0	0	10
99	**beq**	0	10	0	0	X	1	01
19	**addi**	**1**	**00**	**1**	**0**	**0**	**0**	**10**

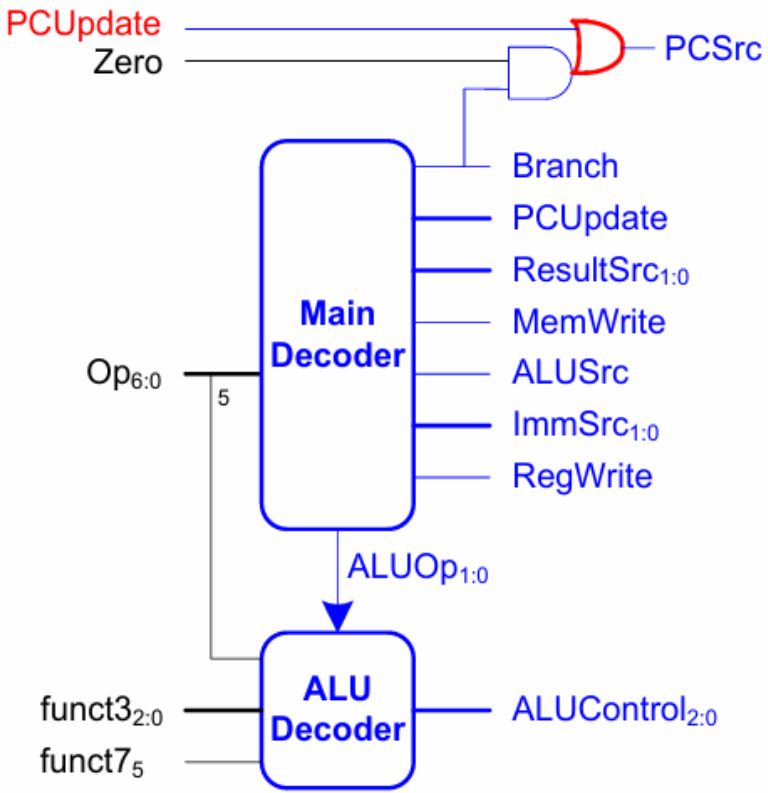
Extended Functionality: `addi`

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
19	addi	1	00	1	0	0	0	10



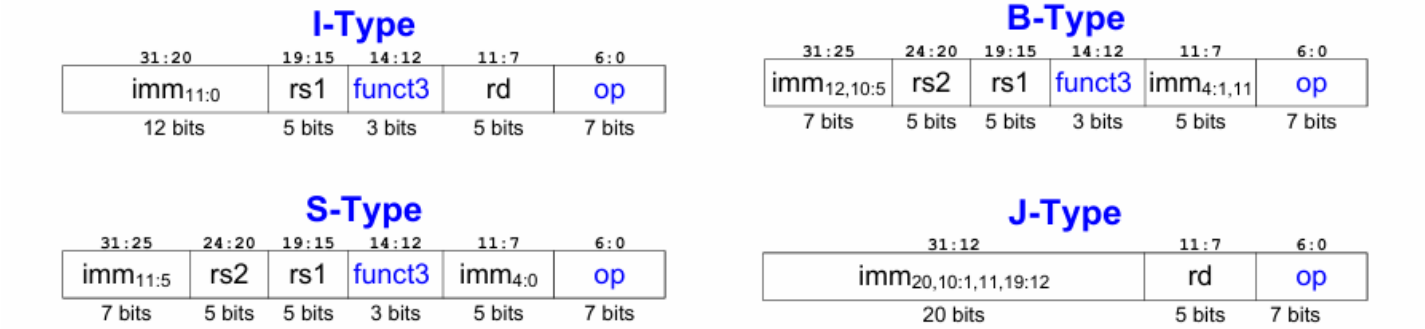
ImmSrc1:0	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type

Extended Functionality: `jal`



Single-Cycle Datapath: ImmExt

$ImmSrc_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, instr[31:20]}	I-Type
01	{{20{instr[31]}}, instr[31:25], instr[11:7]}	S-Type
10	{{19{instr[31]}}, instr[31], instr[7], instr[30:25], instr[11:8], 1'b0}	B-Type
11	**{{12{instr[31]}}, instr[19:12], instr[20], instr[30:21], 1'b0}**	**J-Type**

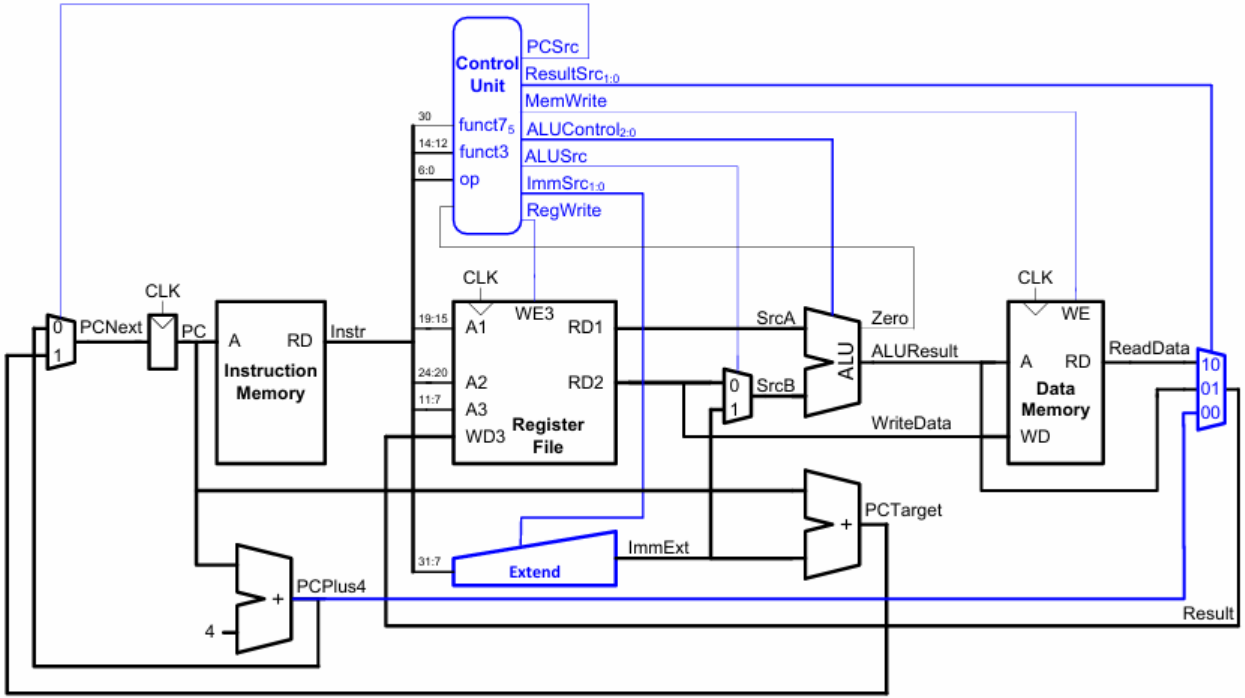


Extended Functionality: `jal`

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	PCUpdate
3	**lw**	1	00	1	0	10	0	00	**0**
35	**sw**	0	01	1	1	XX	0	00	**0**
51	**R-type**	1	XX	0	0	01	0	10	**0**
99	**beq**	0	10	0	0	XX	1	01	**0**
19	**addi**	1	00	1	0	01	0	10	**0**
111	**jal**	**0**	**11**	**X**	**0**	**00**	**0**	**XX**	**1**

Extended Functionality: `jal`

op	Instruct	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	PCUpdate
111	**jal**	0	11	X	0	00	0	XX	1



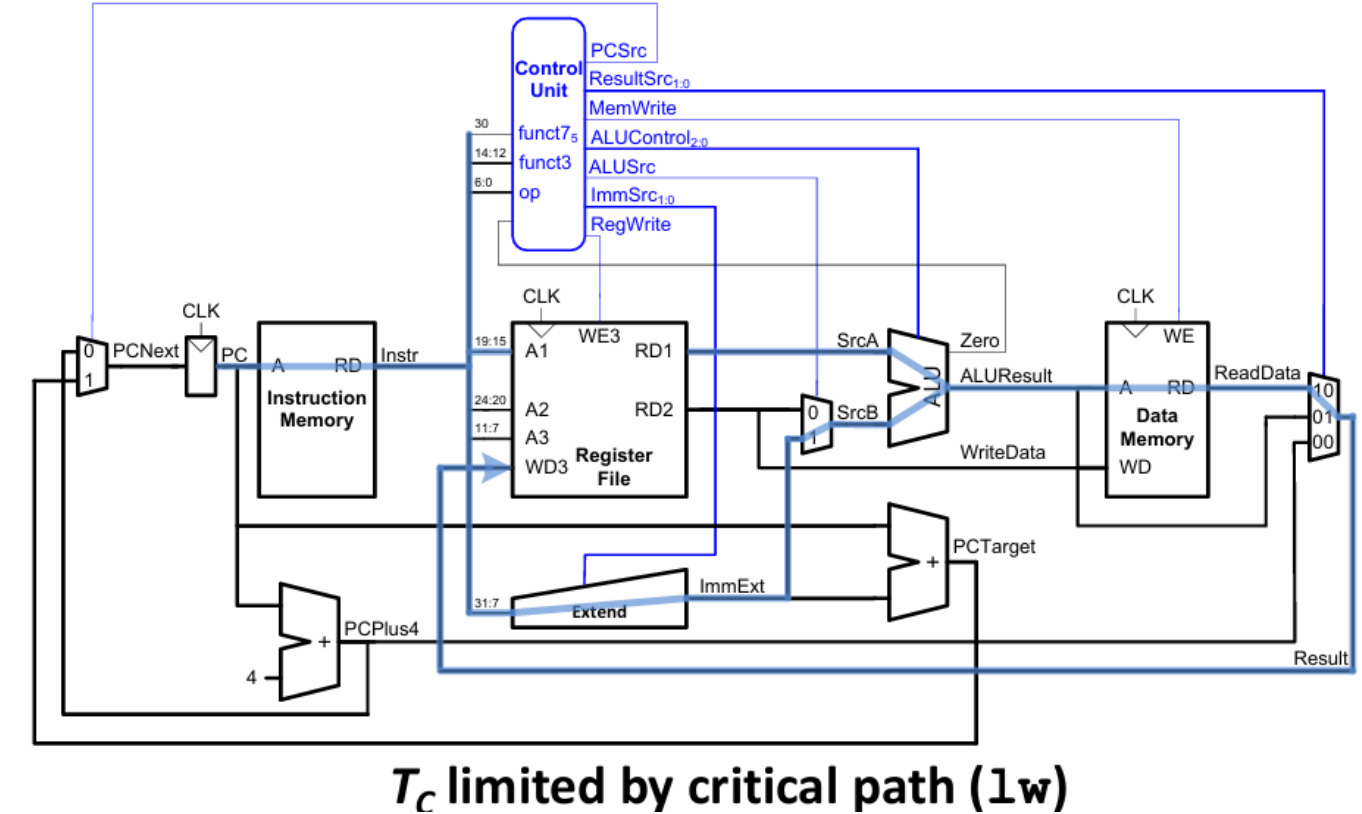
Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

Single-Cycle Performance



T_C limited by critical path (lw)

Single-Cycle Performance

- **Single-cycle critical path:**

$$T_{c1} = t_{pcq\_PC} + t_{\text{mem}} + \max[t_{mux} + t_{RF\text{read}}, t_{ext} + t_{\text{mux}}] + t_{\text{ALU}} + t_{\text{mem}} + t_{\text{mux}} + t_{RF\text{setup}}$$

IMEM DMEM

- **Typically, limiting paths are:**
  - memory, ALU, register file
  - $T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{RF\text{read}} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{RF\text{setup}}$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

$T_{c1} = ?$

Single-Cycle Performance Example

Element	Parameter	Delay (ps)
Register clock-to-Q	t_{pcq_PC}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder (Control Unit)	t_{dec}	70
Memory read	t_{mem}	200
Register file read	$t_{RF\text{read}}$	100
Register file setup	$t_{RF\text{setup}}$	60

$$T_{c1} = t_{pcq\_PC} + 2t_{\text{mem}} + t_{dec} + t_{RF\text{read}} + t_{\text{ALU}} + 2t_{\text{mux}} + t_{RF\text{setup}}$$
$$= [50 + 2(200) + 70 + 100 + 120 + 2(25) + 60] \text{ ps}$$
$$= \mathbf{840 \text{ ps}}$$

Single-Cycle Performance Example

Program with 100 billion instructions:

Execution Time = # instructions x CPI x T_C

= $(100 \times 10^9)(1)(840 \times 10^{-12}\ s)$

= **84 seconds**

Conclusion

- ALU
- State Element
- Single-Cycle CPU
- Design the Datapath