

UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture (a.k.a. Machine Structures)

RISC V Datapath I

Building a RISC-V Processor

- Building a RISC-V Processor
- CPU Elements and Stages
- R-Type: **add** datapath
- R-Type: **sub** datapath
- Datapath with immediates:
addi

Great Idea #1: Abstraction (Levels of Representation/Interpretation)



High Level Language
Program (e.g., C)

| *Compiler*

Assembly Language
Program (e.g., RISC-V)

| *Assembler*

Machine Language
Program (RISC-V)

Hardware Architecture Description
(e.g., block diagrams)

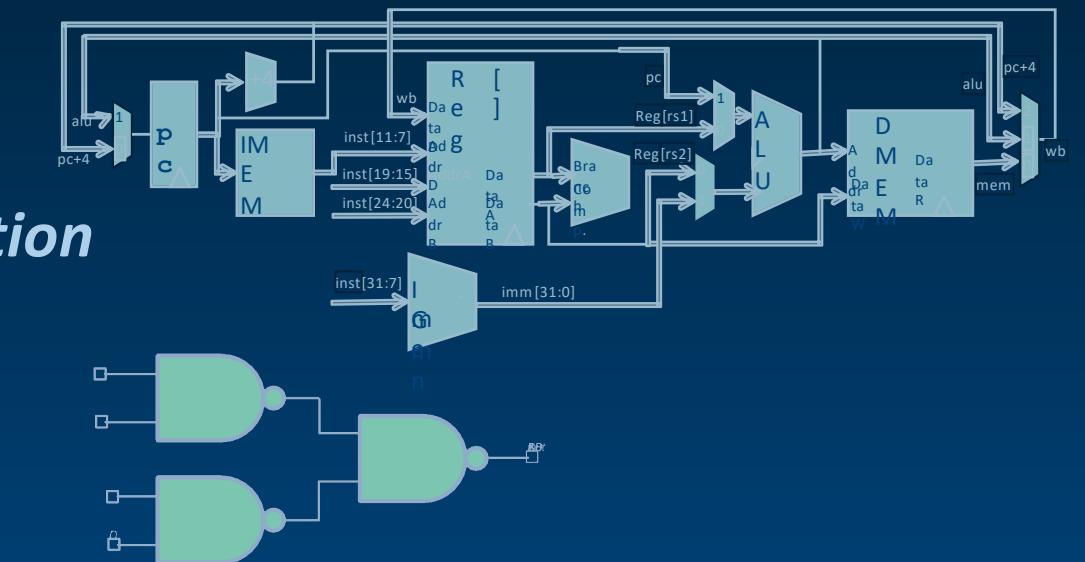
| *Architecture Implementation*

Logic Circuit Description
(Circuit Schematic Diagrams)

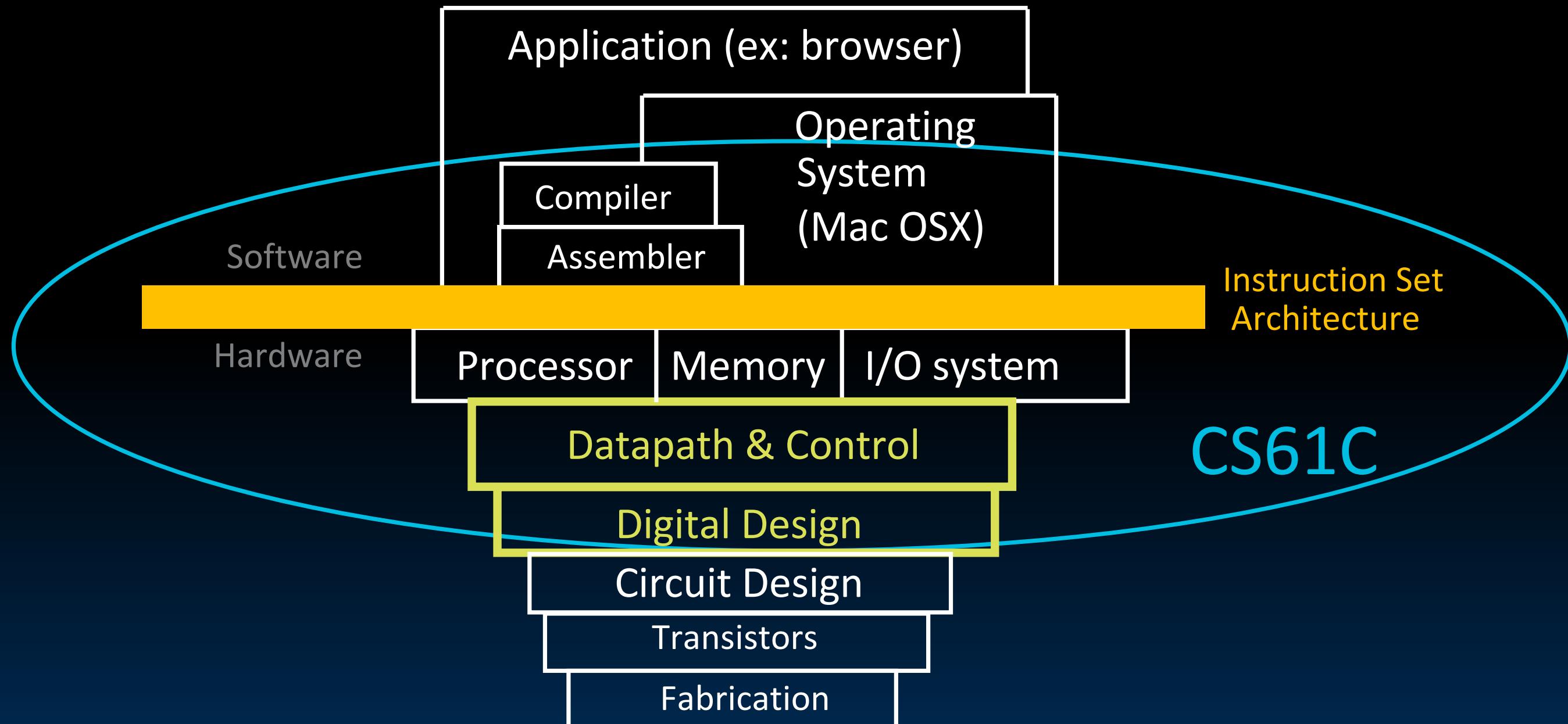
```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    x3, 0(x10)  
lw    x4, 4(x10)  
sw    x4, 0(x10)  
sw    x3, 4(x10)
```

```
1000 1101 1110 0010 0000 0000 0000 0000 0000  
1000 1110 0001 0000 0000 0000 0000 0000 0100  
1010 1110 0001 0010 0000 0000 0000 0000 0000  
1010 1101 1110 0010 0000 0000 0000 0000 0100
```



Old-school Machine Structures



New-school Machine Structures

Software

Parallel Requests

Assigned to computer

e.g., Search “Cats”

Parallel Threads

Assigned to core e.g., Lookup, Ads

Parallel Instructions

>1 instruction @ one time

e.g., 5 pipelined instructions

Parallel Data

>1 data item @ one time

e.g., Add of 4 pairs of words

Hardware descriptions

All gates work in parallel at same time

Harness Parallelism &
Achieve High Performance

Hardware

Warehouse Scale Computer



Smart Phone



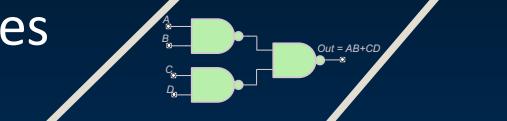
Computer

Core
Memory
Input/Output

Core
(Cache)



Logic Gates

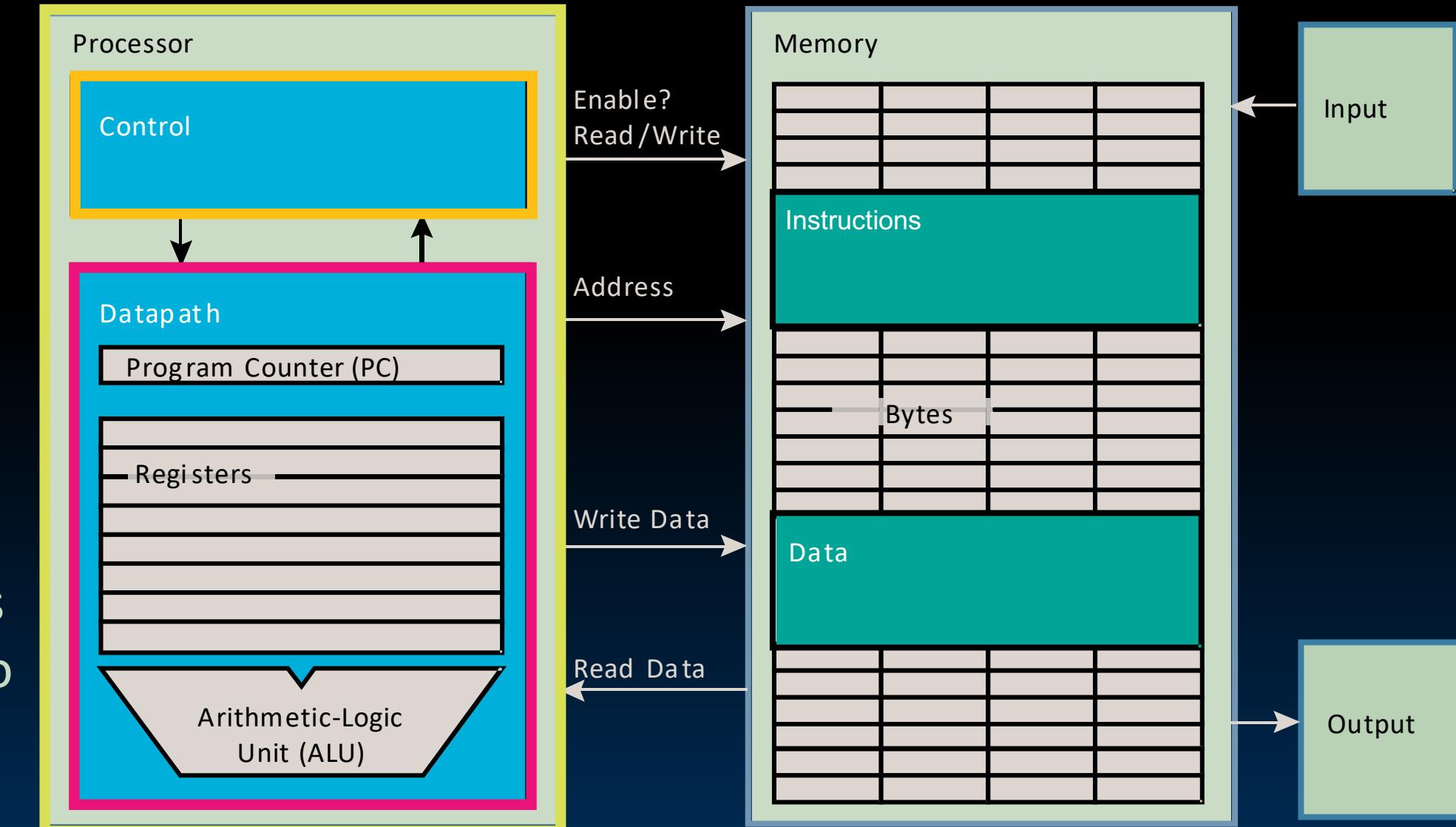


How do we build a Single-Core Processor?

Processor (CPU): the active part of the computer that does all the work (data manipulation and decision-making).

Datapath (“the brawn”): portion of the processor that contains hardware necessary to perform operations required by the processor.

Control (“the brain”): portion of the processor (also in hardware) that tells the datapath what needs to be done.



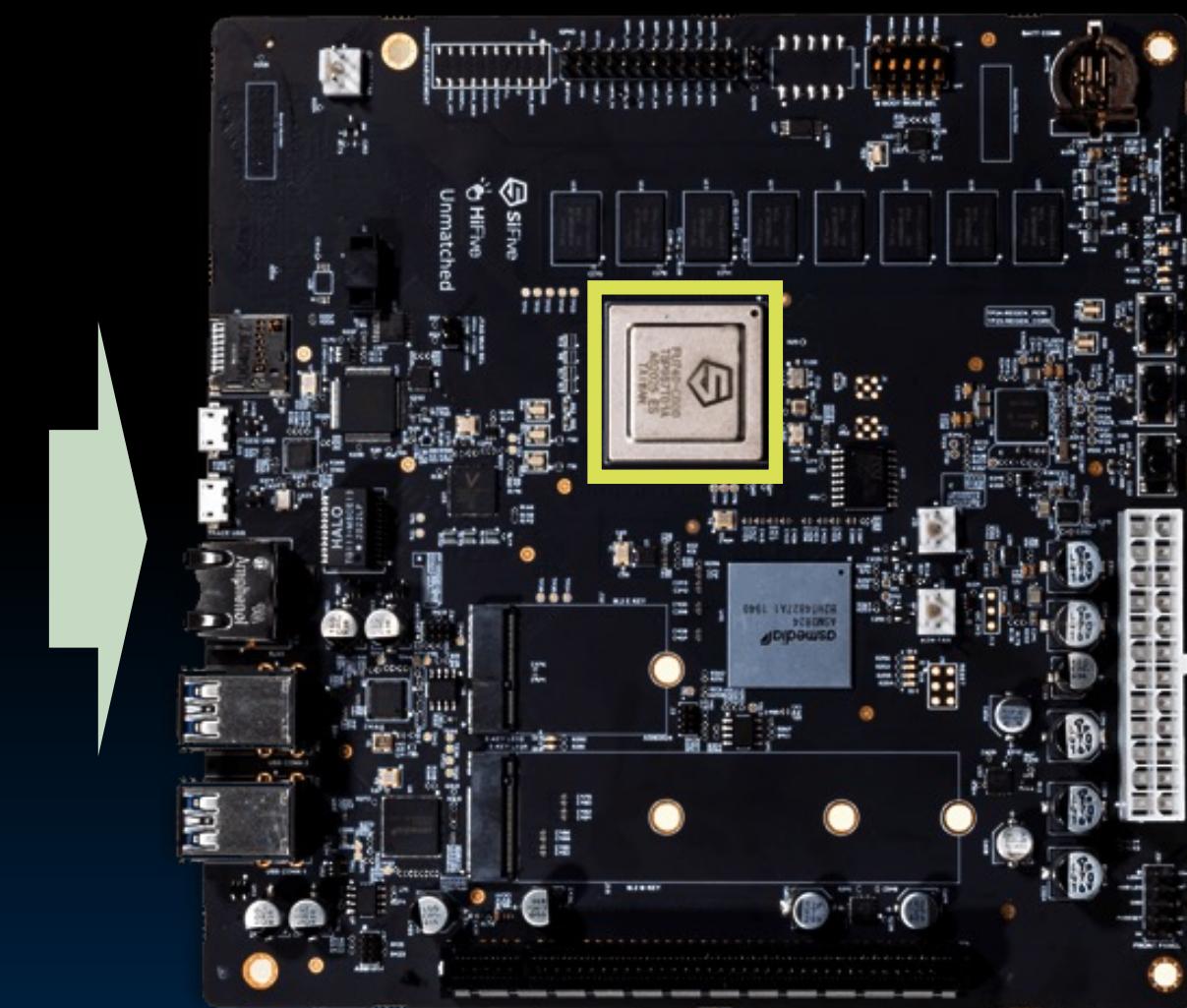
Goal: Design HW to Execute All RV32I Instructions

Open



Reference Card

Category	Name	Fmt	RV32I Base	Category	Name	Fmt	RV32I Base
Shifts	Shift Left			Loads	Load	I	LB rd,rs1,imm
Logical		R	SLL rd,rs1,rs2	Byte	Load Halfword	I	LH rd,rs1,imm
	Shift Left Log. Imm.	I	SLLI rd,rs1,shamt	Load Byte Unsigned	I	LBU rd,rs1,imm	
	Shift Right Logical	R	SRL rd,rs1,rs2	Load Half Unsigned	I	LHU rd,rs1,imm	
	Shift Right Log. Imm.	I	SRLI rd,rs1,shamt	Load Word	I	LW rd,rs1,imm	
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	Stores	Store	S	SB rs1,rs2,imm
	Shift Right Arith. Imm.	I	SRAI rd,rs1,shamt	Byte			
Arithmetic				Store Halfword	S	SH rs1,rs2,imm	
ADD		R	ADD rd,rs1,rs2	Store Word	S	SW rs1,rs2,imm	
	ADD Immediate	I	ADDI rd,rs1,imm	Branches	Branch	B	BEQ rs1,rs2,imm
	SUBtract	R	SUB rd,rs1,rs2	=			
	Load Upper Imm	U	LUI rd,imm		Branch ≠	B	BNE rs1,rs2,imm
Add Upper Imm to PC		U	AUIPC rd,imm		Branch <	B	BLT rs1,rs2,imm
Logical					Branch ≥	B	BGE rs1,rs2,imm
XOR		R	XOR rd,rs1,rs2		Branch < Unsigned	B	BLTU rs1,rs2,imm
	XOR Immediate	I	XORI rd,rs1,imm		Branch ≥ Unsigned	B	BGEU rs1,rs2,imm
	OR	R	OR rd,rs1,rs2	Jump & Link			
				J&L	J	JAL rd,imm	
	OR Immediate	I	ORI rd,rs1,imm	Jump & Link Register	I	JALR rd,rs1,imm	
	AND	R	AND rd,rs1,rs2				
	AND Immediate	I	ANDI rd,rs1,imm				
Compare				Synch		I	FENCE
Set <		R	SLT rd,rs1,rs2	thread			Not in 61C
	Set < Immediate	I	SLTI rd,rs1,imm				
	Set < Unsigned	R	SLTU rd,rs1,rs2	Environment			
	Set < Imm Unsigned	I	SLTIU rd,rs1,imm	CALL	I	ECALL	
				BREAK	I	EBREAK	



HiFive Unmatched (RISC-V Linux)
<https://www.sifive.com/boards/hifive-unmatched>

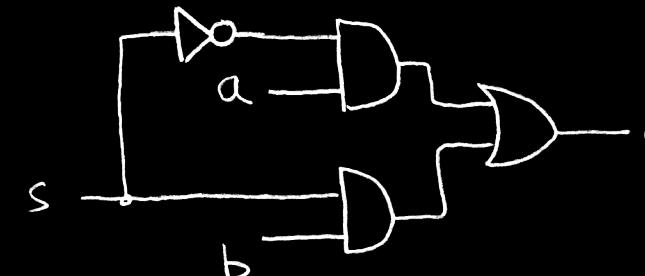
CPU Elements and Stages

- Building a RISC-V Processor
- CPU Elements and Stages
- R-Type: **add** datapath
- R-Type: **sub** datapath
- Datapath with immediates:
addi

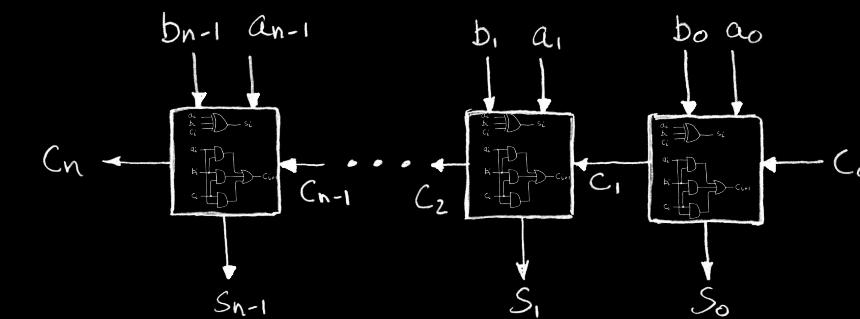
Combinational Logic Blocks

- Today, we will think of all combinational logic subcircuits as block diagrams:

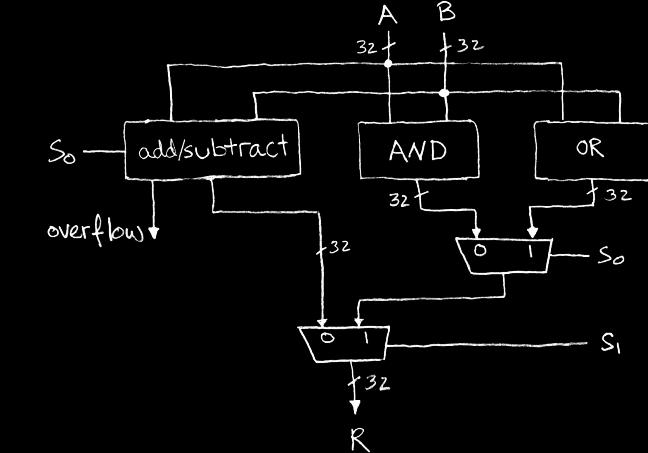
Last time



1-bit-wide 2-to-1MUX

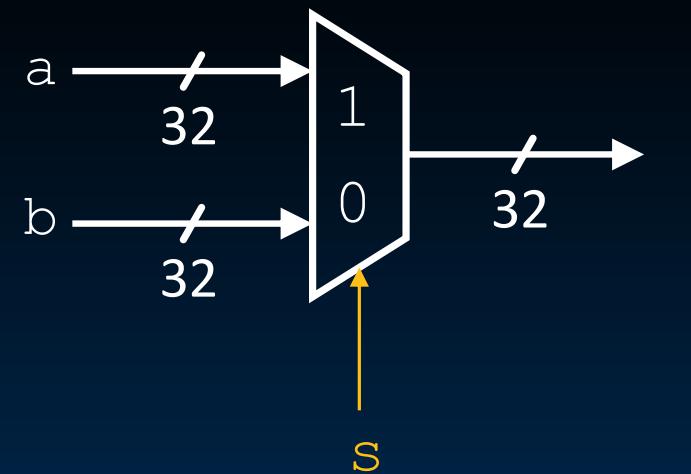


N-Bit Adder

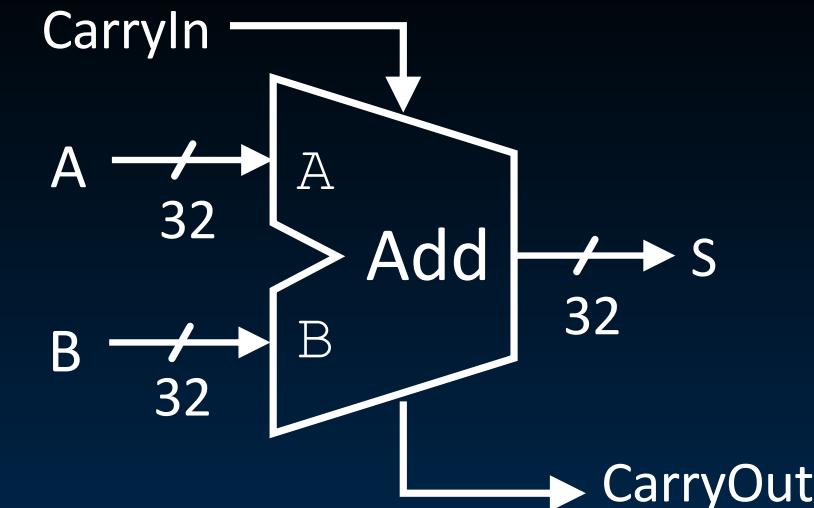


Simple ALU (add,sub, and,or)

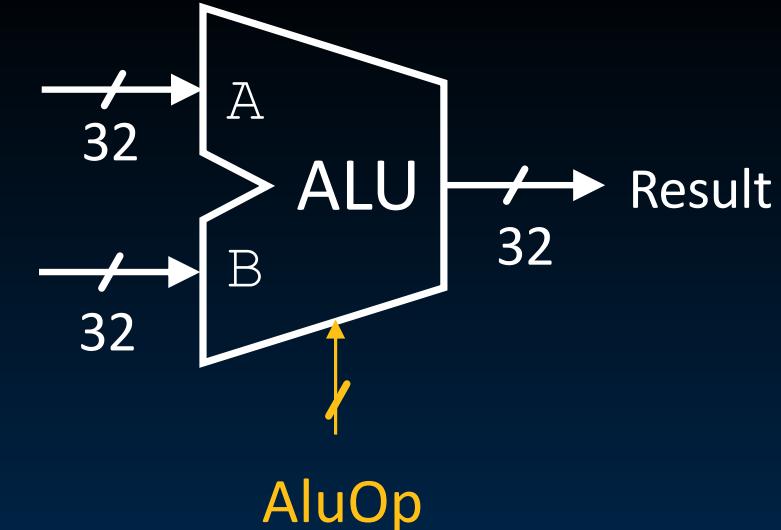
This time



32-bit-wide 2-to-1 MUX



32-Bit Adder

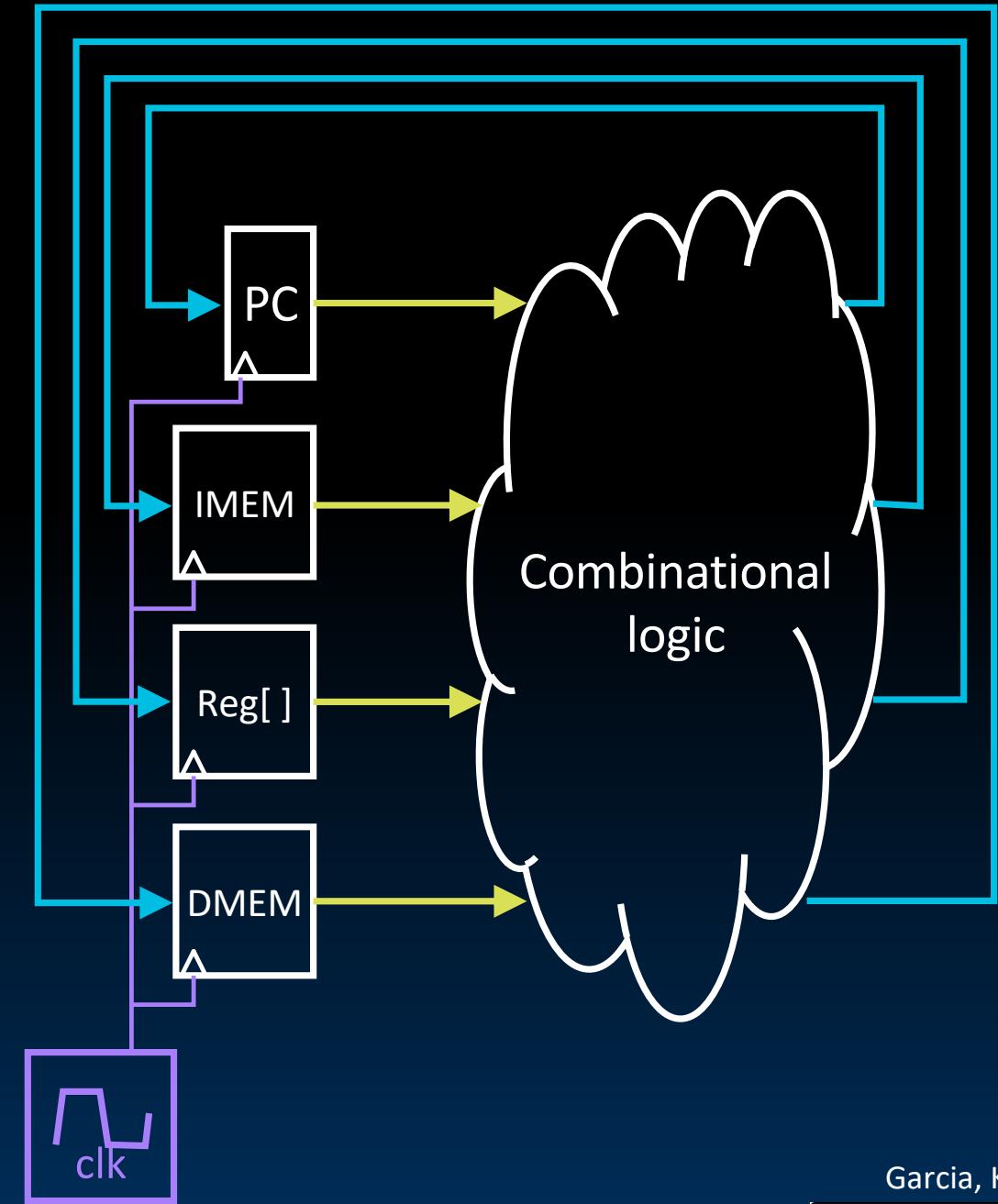


ALU (ALUOp selects from multiple operations)

One-Instruction-Per-Cycle RISC V Machine

The CPU is composed of two types of subcircuits:
combinational logic blocks and *state elements*.

- On every tick of the clock, the computer executes one instruction:
 - Current outputs of the *state elements* drive the inputs to *combinational logic*...
 - ...whose outputs settle at the *inputs to the state elements* before the next rising clock edge.
- At the rising clock edge:
 - All the *state elements* are updated with the combinational logic outputs...
 - and execution moves to the next clock cycle.



State Elements Required by RV32I ISA

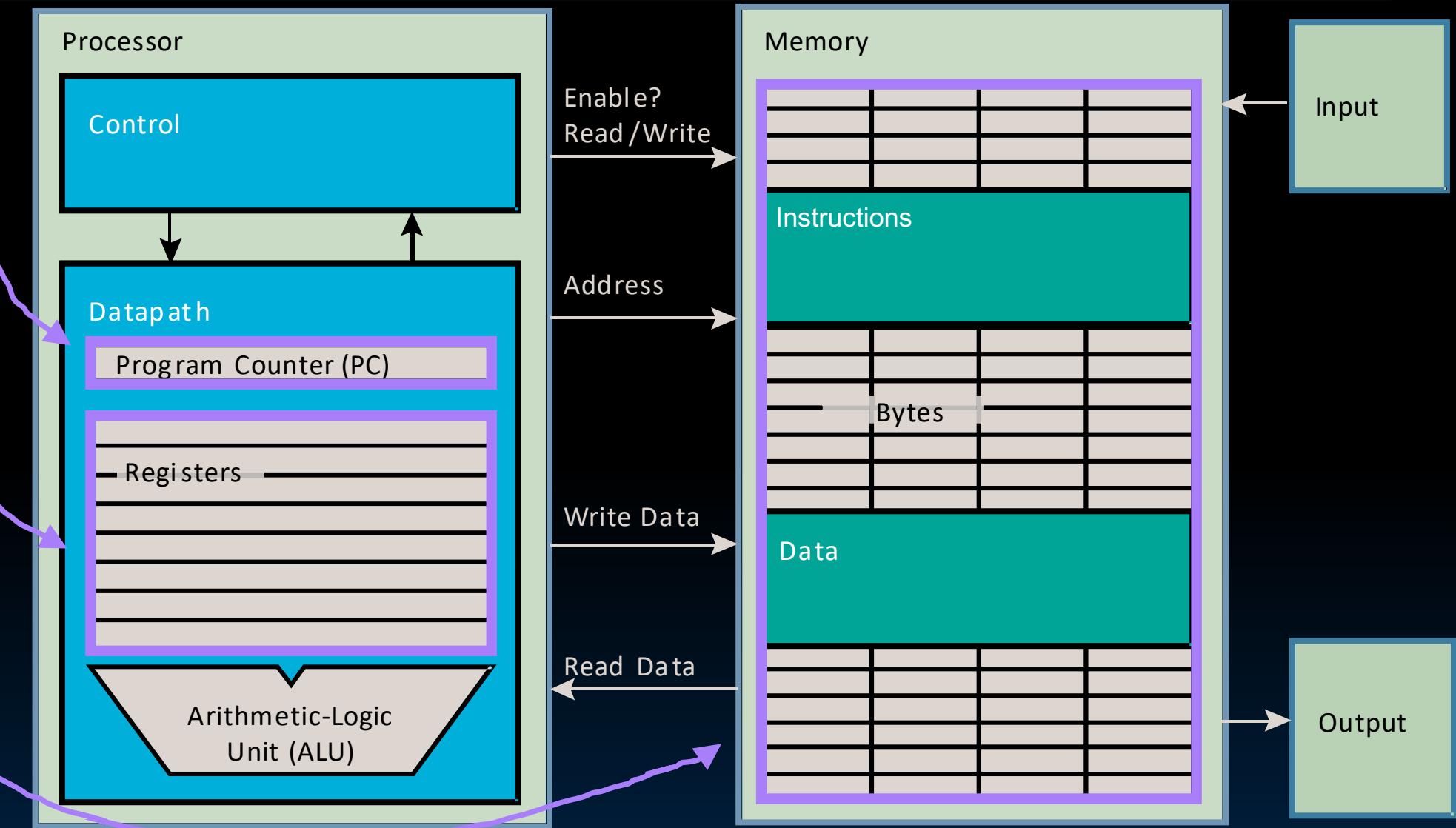
Program Counter



Register File Reg



Memory MEM



During CPU execution, each RV32I instruction reads and/or updates these state elements.

State Elements (1/3): Program Counter

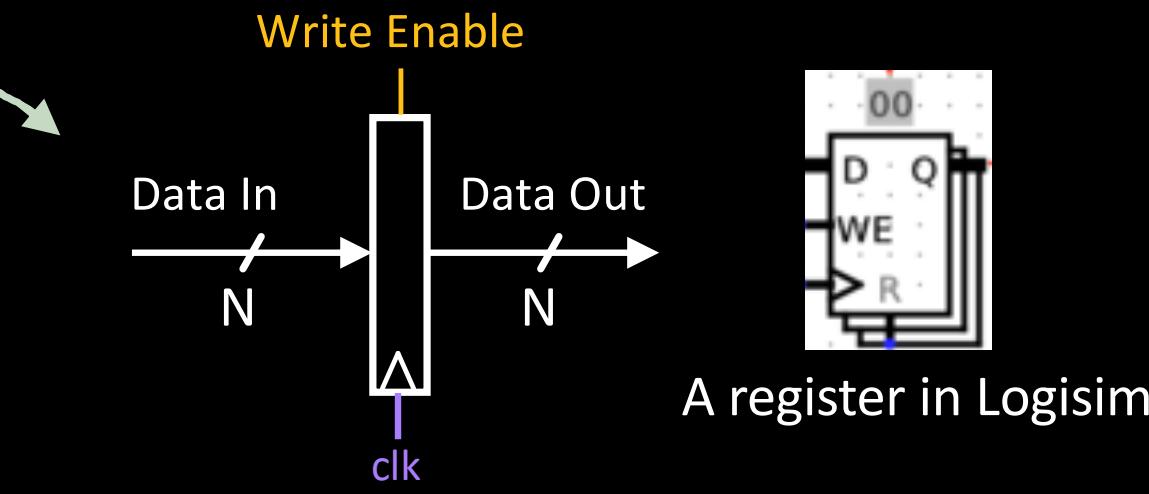
Program Counter



Register File Reg



Memory MEM



A register in Logisim

Input:

N-bit data input bus

Write Enable “Control” bit (1: asserted/high, 0: deasserted/0)

Output:

N-bit data output bus

Behavior:

If Write Enable is 1 on *rising clock edge*, set Data Out=Data In.

At all other times, Data Out will not change; it will output its current value.

The Program Counter is a 32-bit Register.

State Elements (2/3): Register File

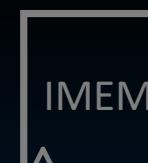
Program Counter



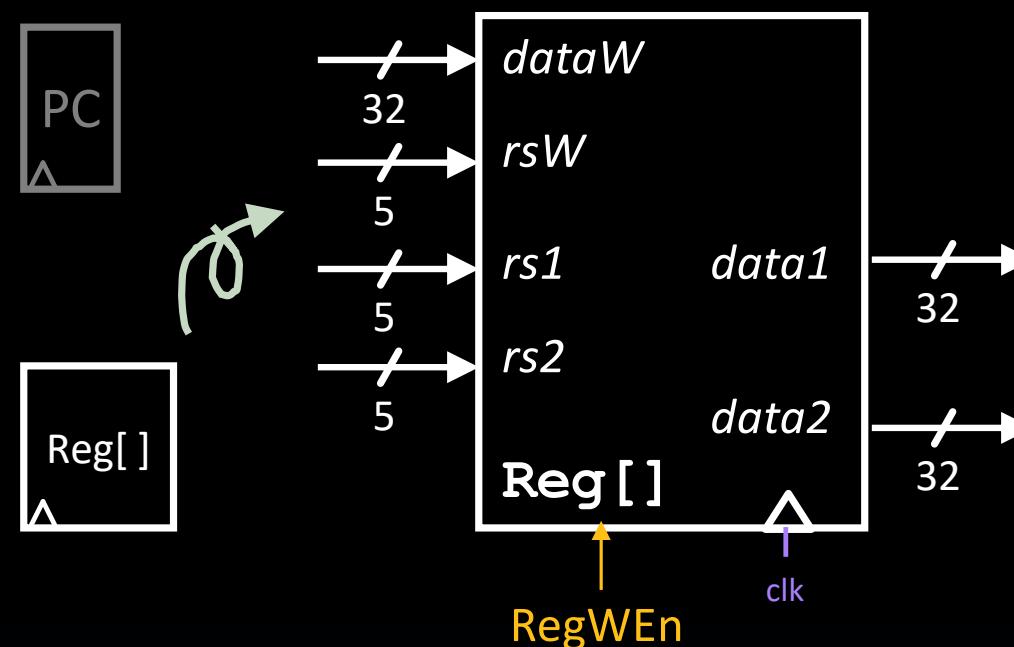
Register File Reg



Memory MEM



RegFile behaves like a combinational block for read operations!



Register File (RegFile) has 32 registers.

Input:

One 32-bit input data bus, *dataW*.

Three 5-bit select busses, *rs1*, *rs2*, and *rsW*.

RegWEn control bit.

Output:

Two 32-bit output data busses, *data1* and *data2*.

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* bus out.

R[rd]: *rsW* selects register to be written via *dataW* when RegWEn=1.

Clock behavior: Write operation occurs on *rising clock edge*.

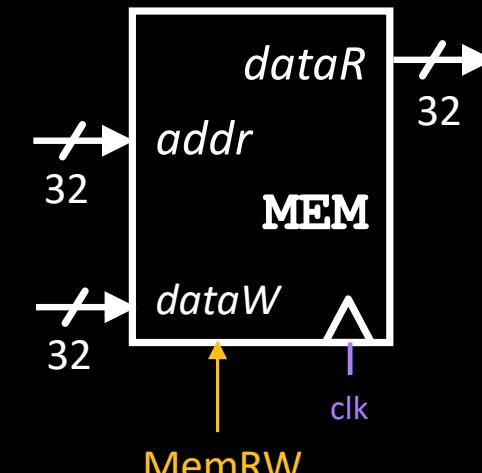
Clock input only a factor on write!

All read operations behave like a combinational block:

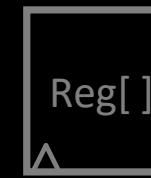
If *rs1*, *rs2* valid, then *data1*, *data2* valid after *access time*.

State Elements (3/3): Memory

Program Counter



Register File Reg



Memory MEM



Memory is “magic.” For this class:
32-bit byte-addressed memory space; and
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.

If MemRW=0, **MEM** behaves like a combinational block.

“Two” Memories: IMEM, DMEM

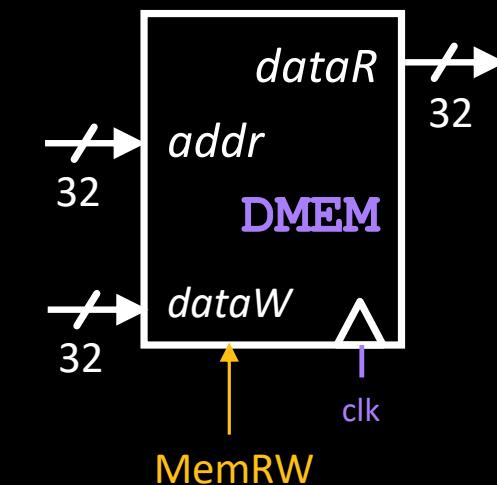
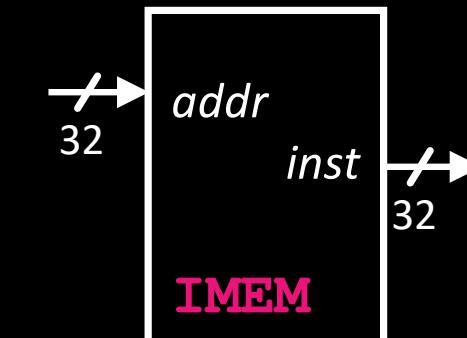
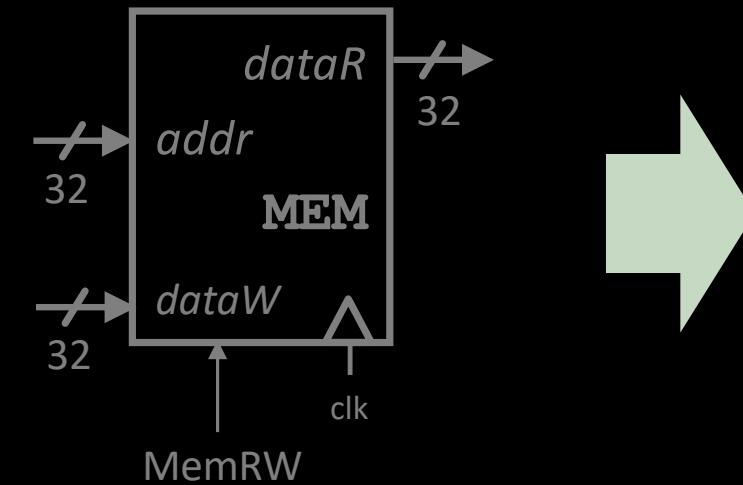
Program Counter



Register File Reg



Memory MEM



Current abstraction: Memory holds both instructions and data in one contiguous 32-bit memory space.

In our processor, we'll use two “separate” memories:

IMEM: A *read-only* memory for fetching instructions.

DMEM: A memory for loading (read) and storing (write) data words.

Under the hood, these are placeholders for caches. (more later)

Because IMEM is read-only, it always behaves like a combinational block:

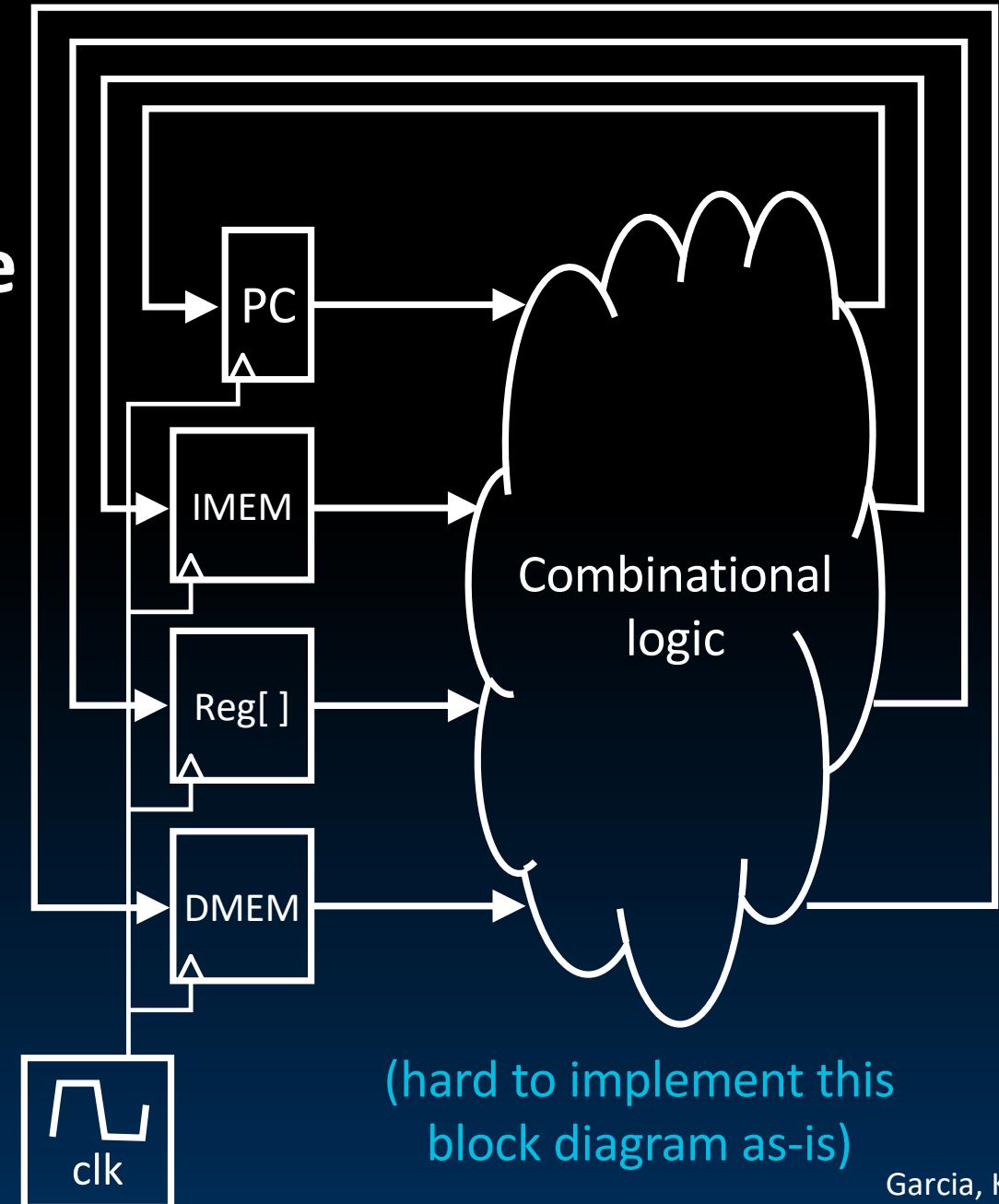
If *addr* valid, then *instr* valid after access time.

Design the Datapath in Phases

- **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.



5 Basic Stages (Phases) of Instruction Execution

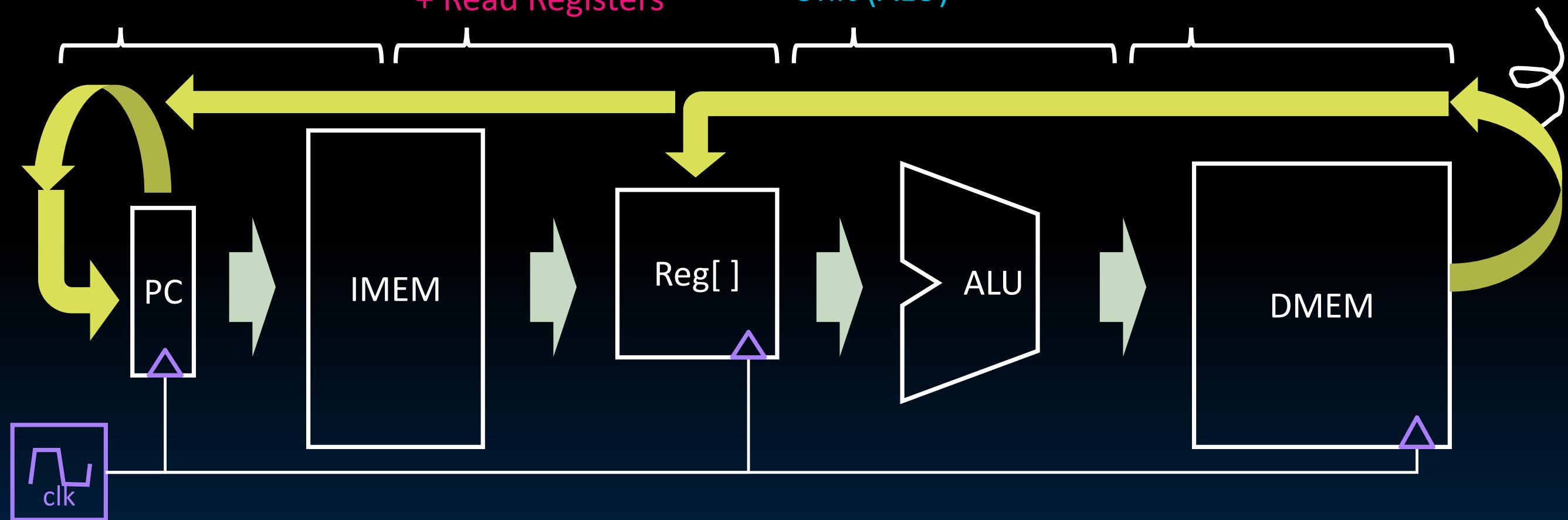
1. Instruction Fetch (**IF**)

2. Instruction Decode (**ID**)
+ Read Registers

3. Execute (**EX**)
Arithmetic Logic Unit (ALU)

4. Memory Access (**MEM**)

5. Write back to Register (**WB**)



- We will implement a single-cycle processor:

All stages of one RV32I instruction execute within the same clock cycle.

Not All Instructions Need All 5 Stages!

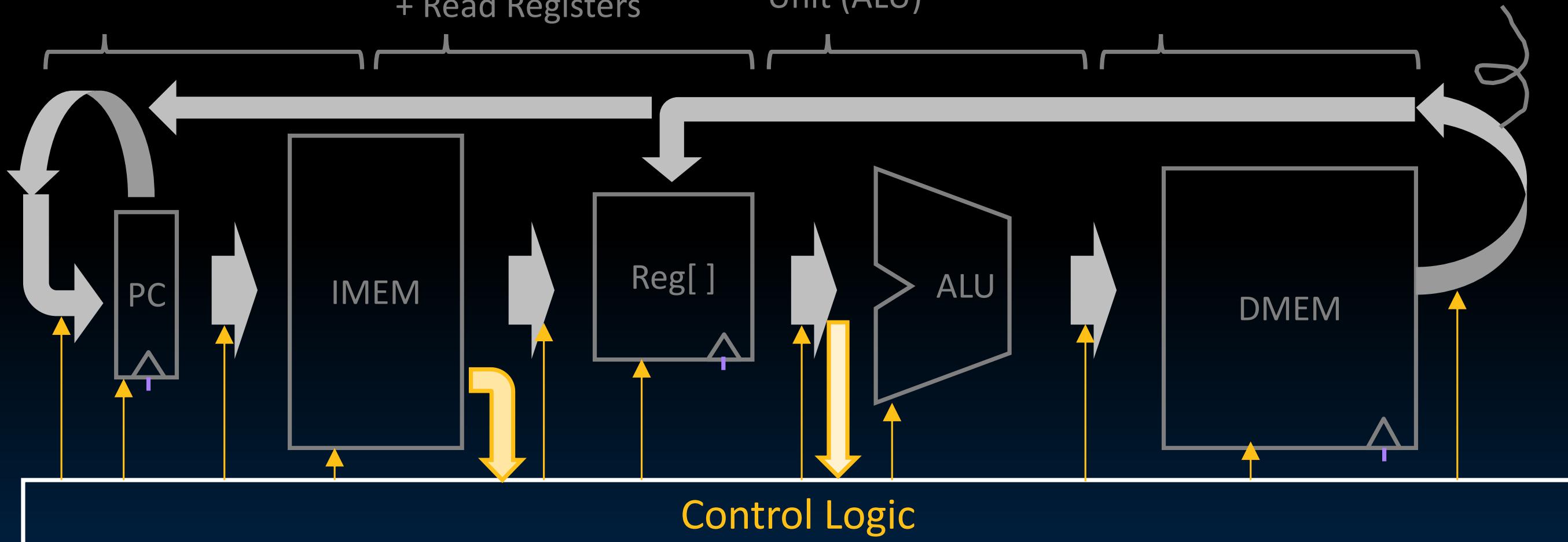
1. Instruction Fetch (**IF**)

2. Instruction Decode (**ID**)
+ Read Registers

3. Execute (**EX**)
Arithmetic Logic Unit (ALU)

4. Memory Access (**MEM**)

5. Write back to Register (**WB**)



The **control logic** selects “needed” datapath lines based on the instruction.

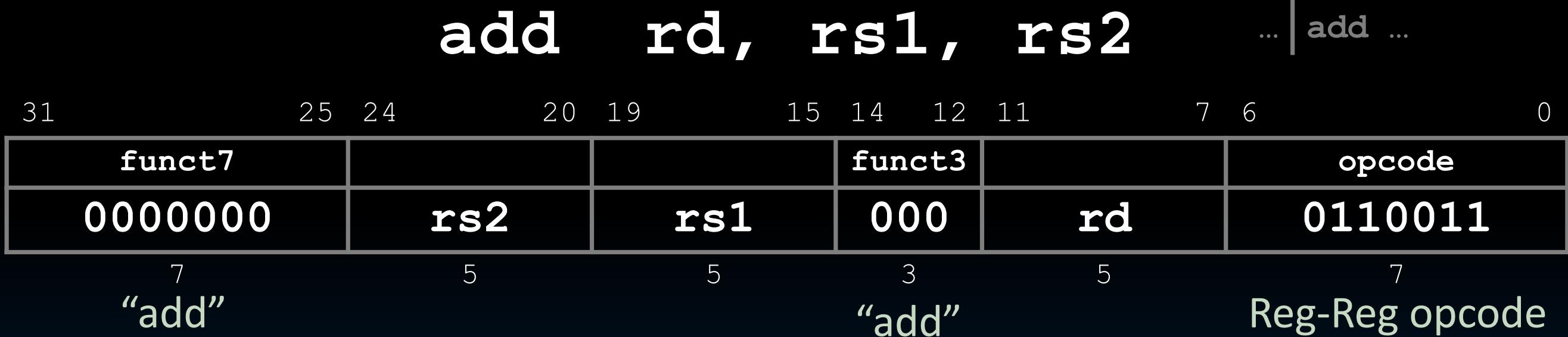
MUX selector, ALU op selector, write enable, etc.

R-Type: add Datapath

- Building a RISC-V Processor
- CPU Elements and Stages
- R-Type: **add** datapath
- R-Type: **sub** datapath
- Datapath with immediates:
addi

Implementing the add Instruction

- Suppose we had a single instruction in our RISC-V ISA: add.



- The add instruction makes two changes to processor state:

RegFile

$$R[rd] = R[rs1] + R[rs2]$$

PC

$$PC = PC + 4$$

Example add-only program

0x100	add x18,x18,x10
0x104	add x18,x18,x18
...	add ...

Datapath for add



$$PC = PC + 4$$

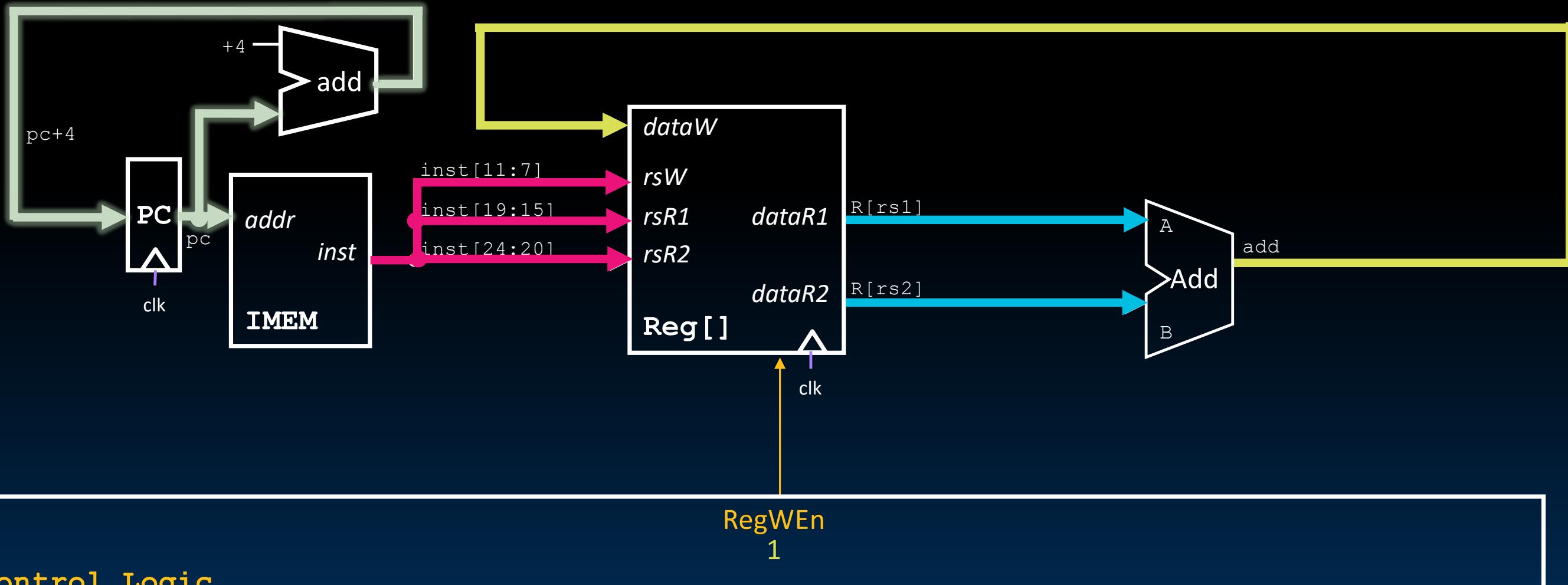
$$R[rd] = R[rs1] + R[rs2]$$

Increment PC to next instruction.

Split instruction to index into RegFile.

Feed read register values into Add.

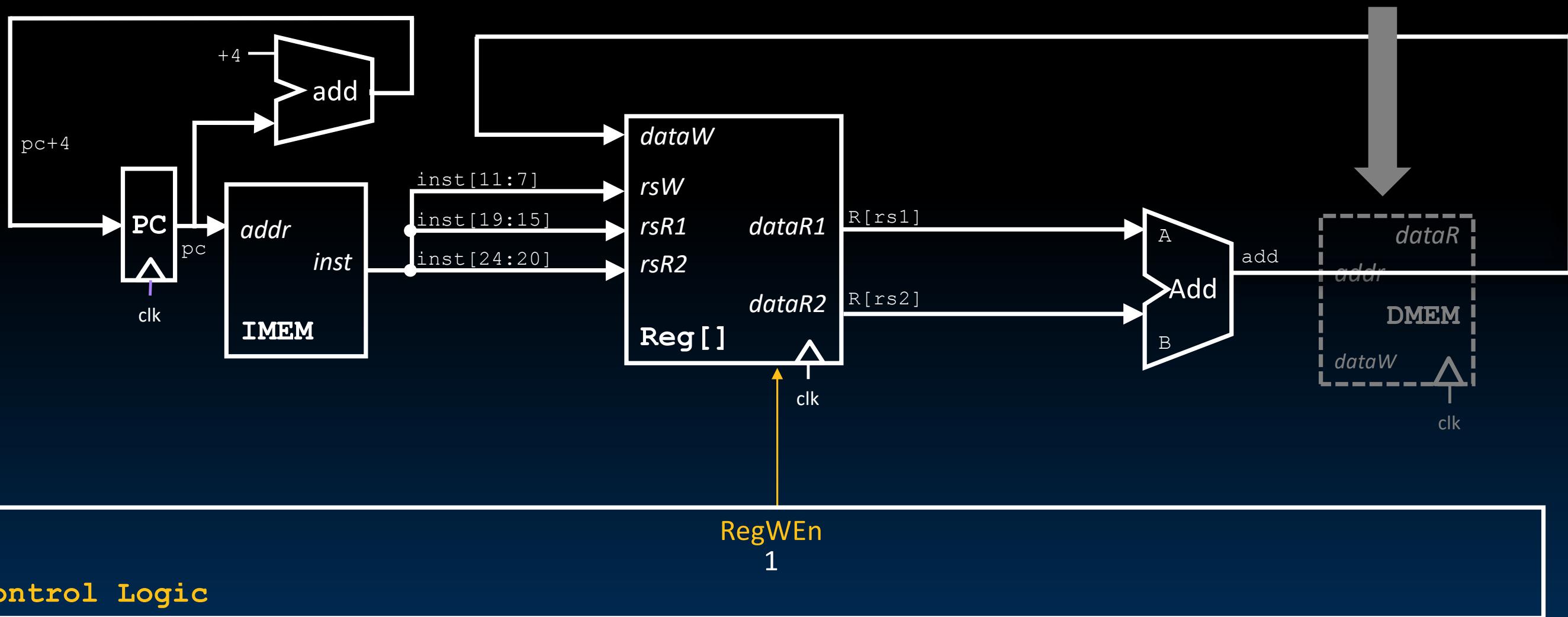
Write Add output to destination register.



Control Logic

Plan: Iteratively Build Full Datapath

For now, we will disconnect memory.
By the end of next lecture, we'll build the datapath that supports the full RV32I ISA.



Control Logic

R-Type: sub Datapath

- Building a RISC-V Processor
- CPU Elements and Stages
- R-Type: add datapath
- R-Type: sub datapath
- Datapath with immediates:
addi

Implementing the sub Instruction

- What if our ISA now had two instructions: add/sub?

funct7	rs2	rs1	funct3	rd	opcode	
0000000			000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

sub rd, rs1, rs2

- sub is almost the same as add, except now the ALU subtracts operands instead of adding them:

$$\text{RegFile} \quad R[rd] = R[rs1] - R[rs2]$$

$$\text{PC} \quad \text{PC} = \text{PC} + 4$$

- Instruction bit `inst[30]` selects between add/sub.

Details left to *control logic*.

Datapath for sub



$$PC = PC + 4$$

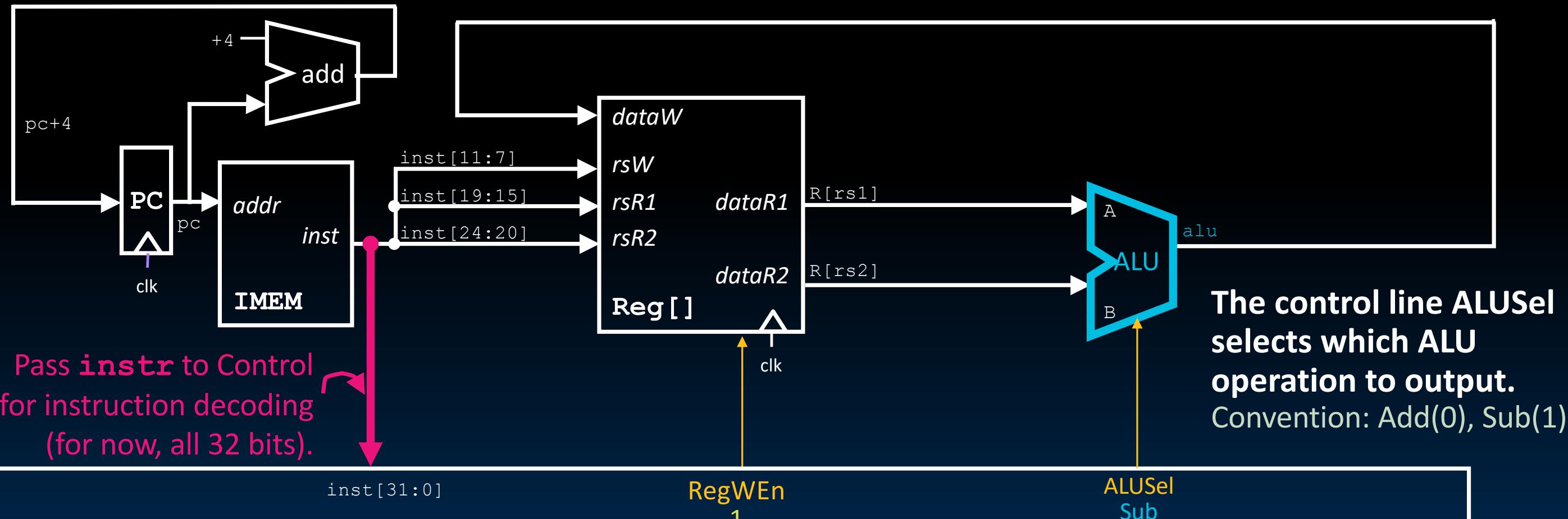
$$R[rd] = R[rs1] - R[rs2]$$

Increment PC to next instruction.

Split instruction to index into RegFile.

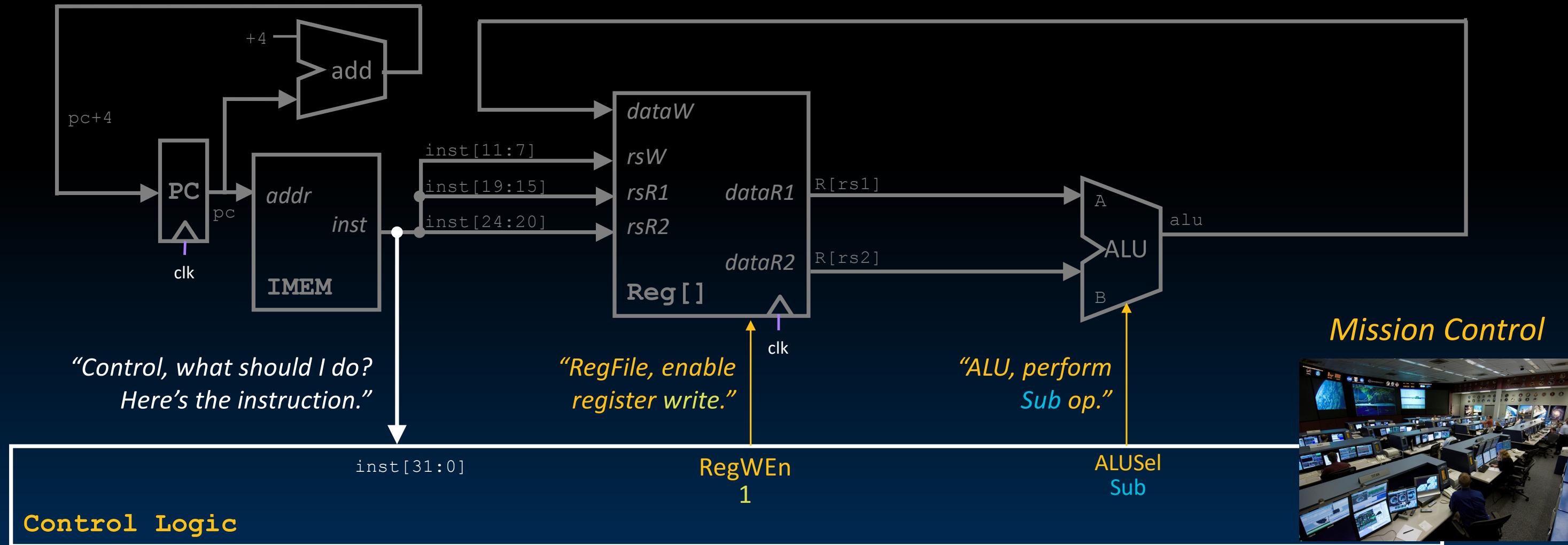
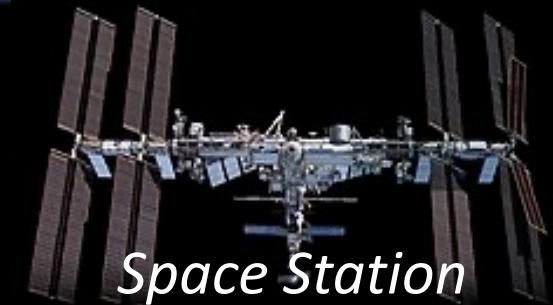
Feed read register values *into ALU*.

Write *ALU output* to destination register.



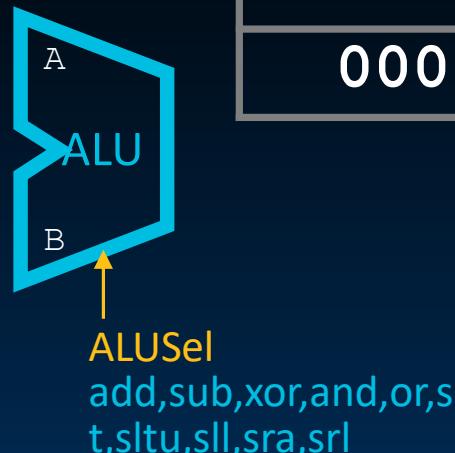
Control vs. Datapath Analogy: Space Mission

Control and Datapath are two *abstractions* that together comprise the CPU. Under the hood, both are designed with combinational logic and/or state elements.



Supporting all R-Type Instructions

funct7			funct3		opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	001	rd	0110011	sll
0000000	rs2	rs1	010	rd	0110011	slt
0000000	rs2	rs1	011	rd	0110011	sltu
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	101	rd	0110011	srl
0100000	rs2	rs1	101	rd	0110011	sra
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and



The Control Logic decodes **funct3**, **funct7** instruction fields and selects appropriate ALU function by setting the control line **ALUSel**.

Check out Project 3
for the full ALU!

Datapath with Immediates: **addi**

- Building a RISC-V Processor
- CPU Elements and Stages
- R-Type: **add** datapath
- R-Type: **sub** datapath
- Datapath with immediates:
addi

Implementing the addi Instruction

- Suppose we add a new instruction: addi, RV32I I-Format:

addi rd, rs1, imm



- The addi instruction updates the same two states as before.
But we now need to build an immediate imm!

RegFile

$\text{Reg}[\text{rd}] = \text{Reg}[\text{rs1}] + \text{imm}$

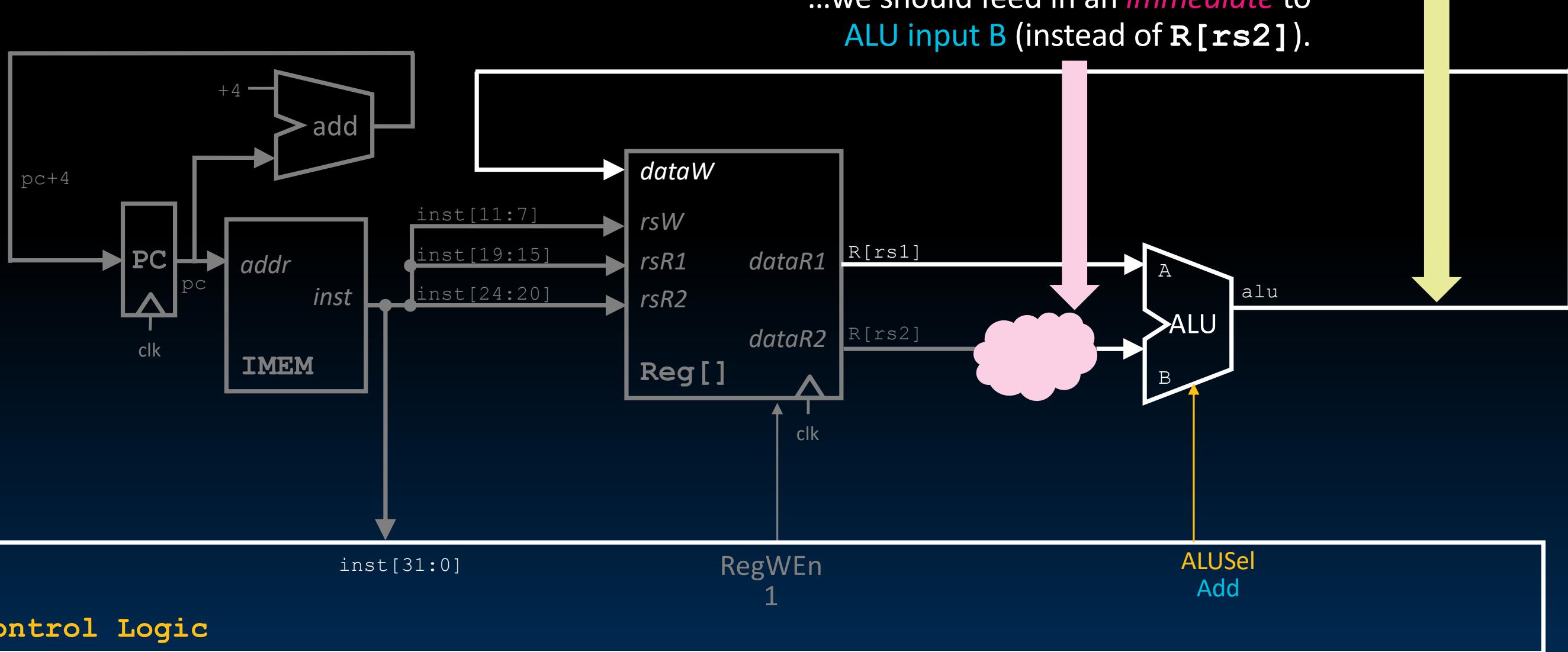
PC

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

What Does the Datapath Need for addi?

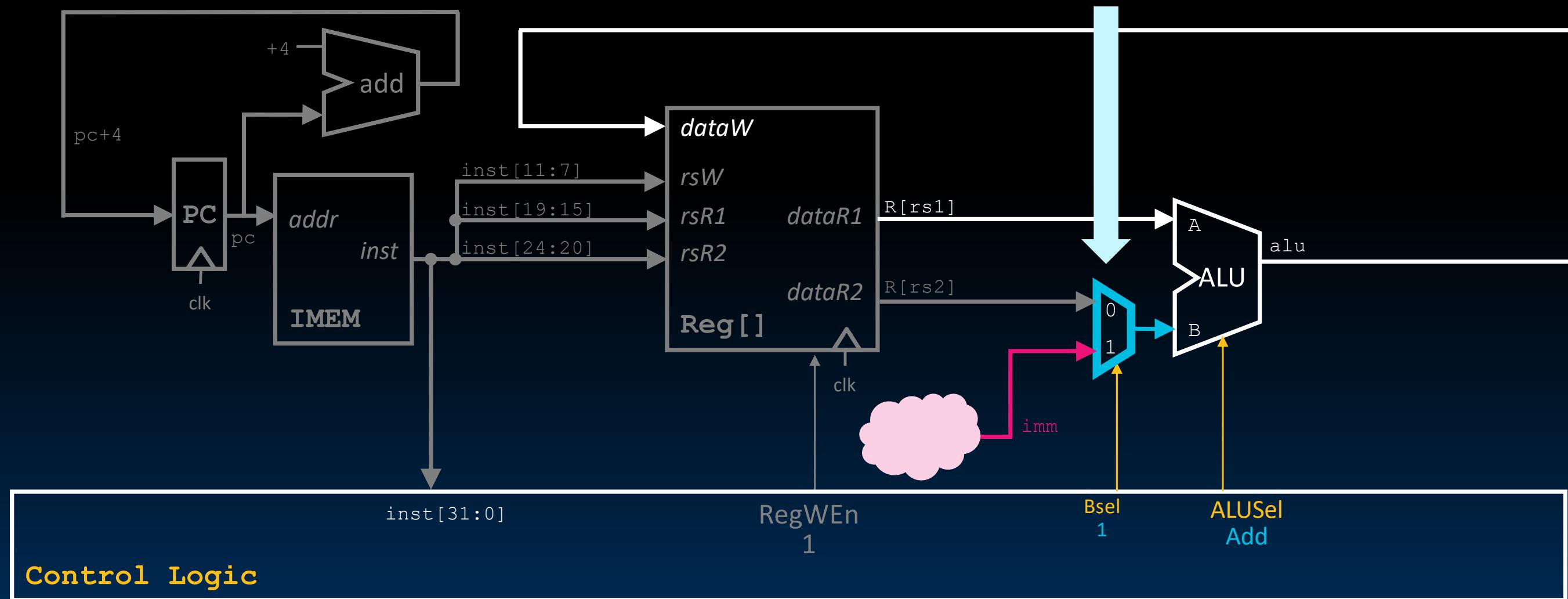
$$PC = PC + 4$$

$$R[rd] = R[rs1] + imm$$



1. New MUX to Select Immediate for ALU

1. Control line $Bsel=1$ selects the generated immediate **imm** for ALU input B.

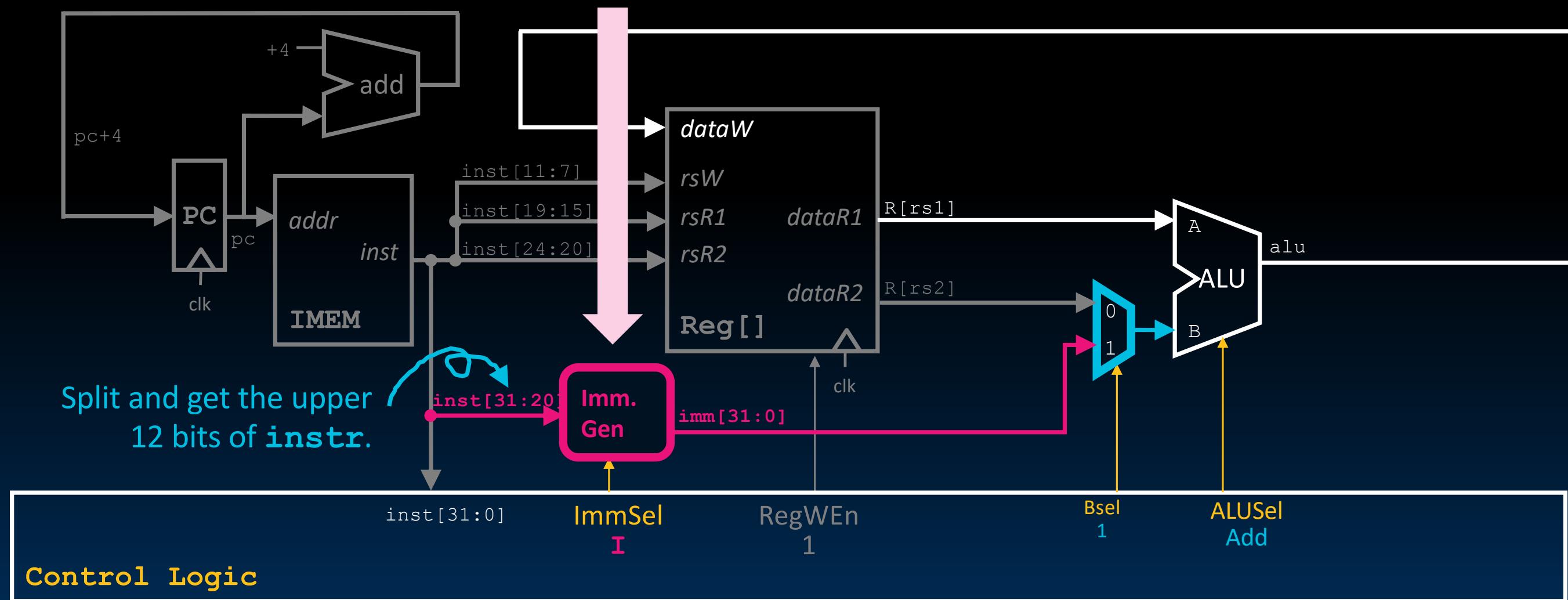


2. New Block to Generate 32-bit Immediate

2. Immediate Generation

Block builds a 32-bit immediate **imm** from instruction bits.

1. Control line **Bsel=1** selects the generated immediate **imm** for ALU input B.



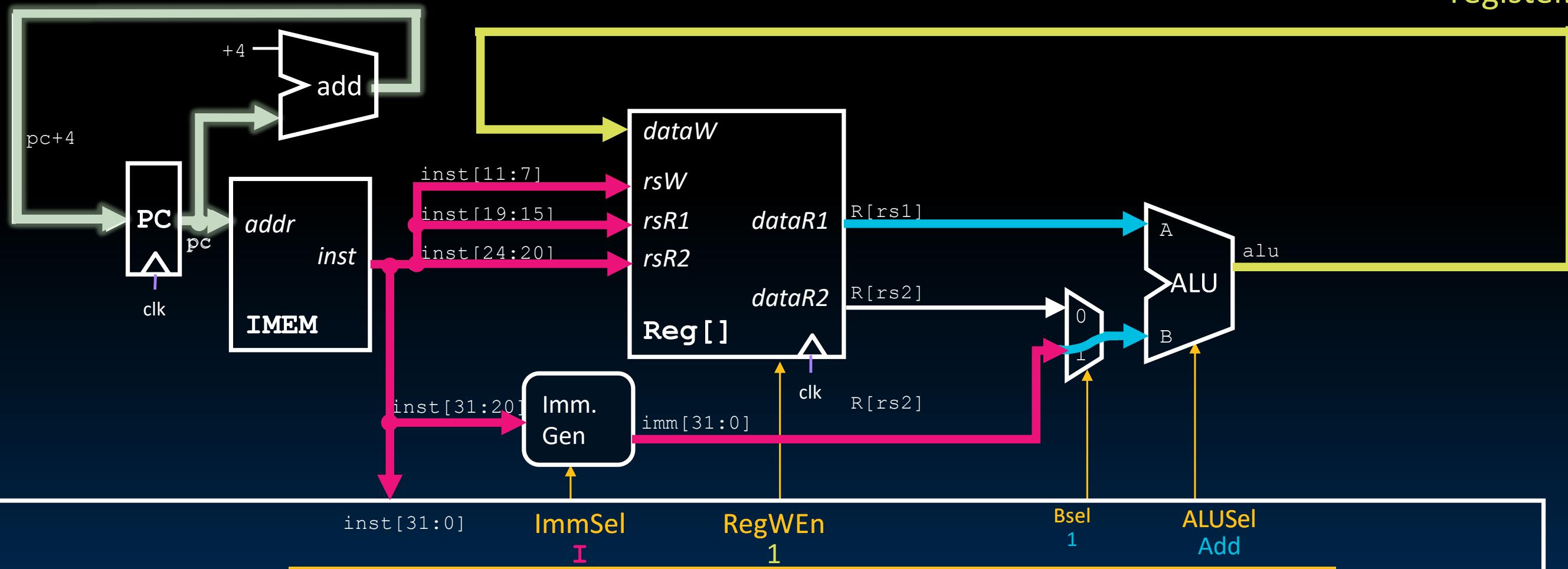
Tracing/Lighting the addi Datapath

Increment PC to next instruction.

Immediate Generation
Block builds a 32-bit immediate **imm** from instruction bits.

Control line **Bsel=1** selects the generated immediate **imm** for ALU input B.

Write ALU output to destination register.

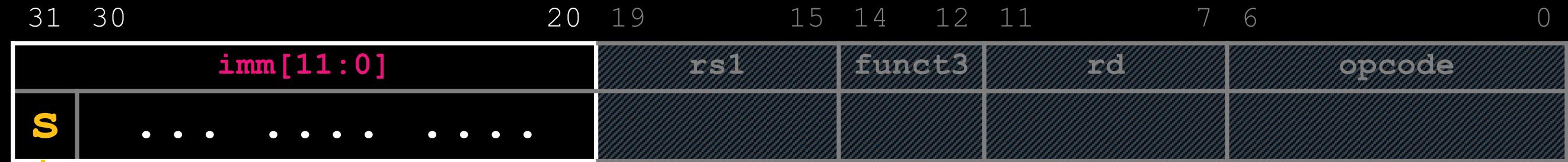


Data **inst[24:20]** still feeds into **Reg[]**, which still outputs **R[rs2]**. However, control **Bsel=1** means **R[rs2]** data line is ignored.

Immediate Generation Block Design

(more next time)

Instruction
 $\text{inst}[31:0]$



Copy *upper* 12 bits of instruction,
 $\text{inst}[31:20]$, to *lower* 12 bits of
immediate, $\text{imm}[11:0]$.

Immediate
 $\text{imm}[31:0]$





Your Own Design

1. We should use the main ALU to compute $PC=PC+4$ in order to save some gates.
2. The ALU is a synchronous state element.
3. The program counter (PC) is a register.



Your Own Design

1. We should use the main ALU to compute $PC=PC+4$ in order to save some gates.
2. The ALU is a synchronous state element.
3. The program counter (PC) is a register.

- $\frac{123}{FFF}$
- A. FFT
 - B. FTF
 - C. FTT
 - D. TFT
 - E. TFF
 - F. TFT
 - G. TTF
 - H. TTT

Summary: Arithmetic/Logical Datapath

- All data lines carry information.
- Control logic determines what is “useful/needed” vs. what is “ignored.”
e.g., ALUSel: chooses ALU operation; Bsel: chooses register/immediate for ALU input B.

