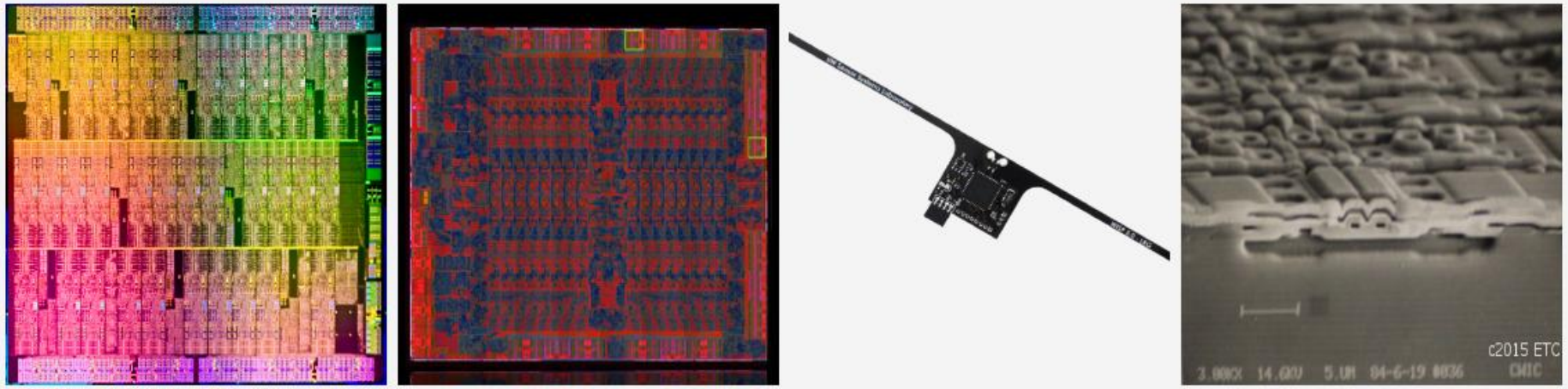
18-344: Computer Systems and the Hardware-Software Interface

Fall 2023



Course Description

Lecture 3: Computer Architecture Basics

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

Credit: Brandon Lucia

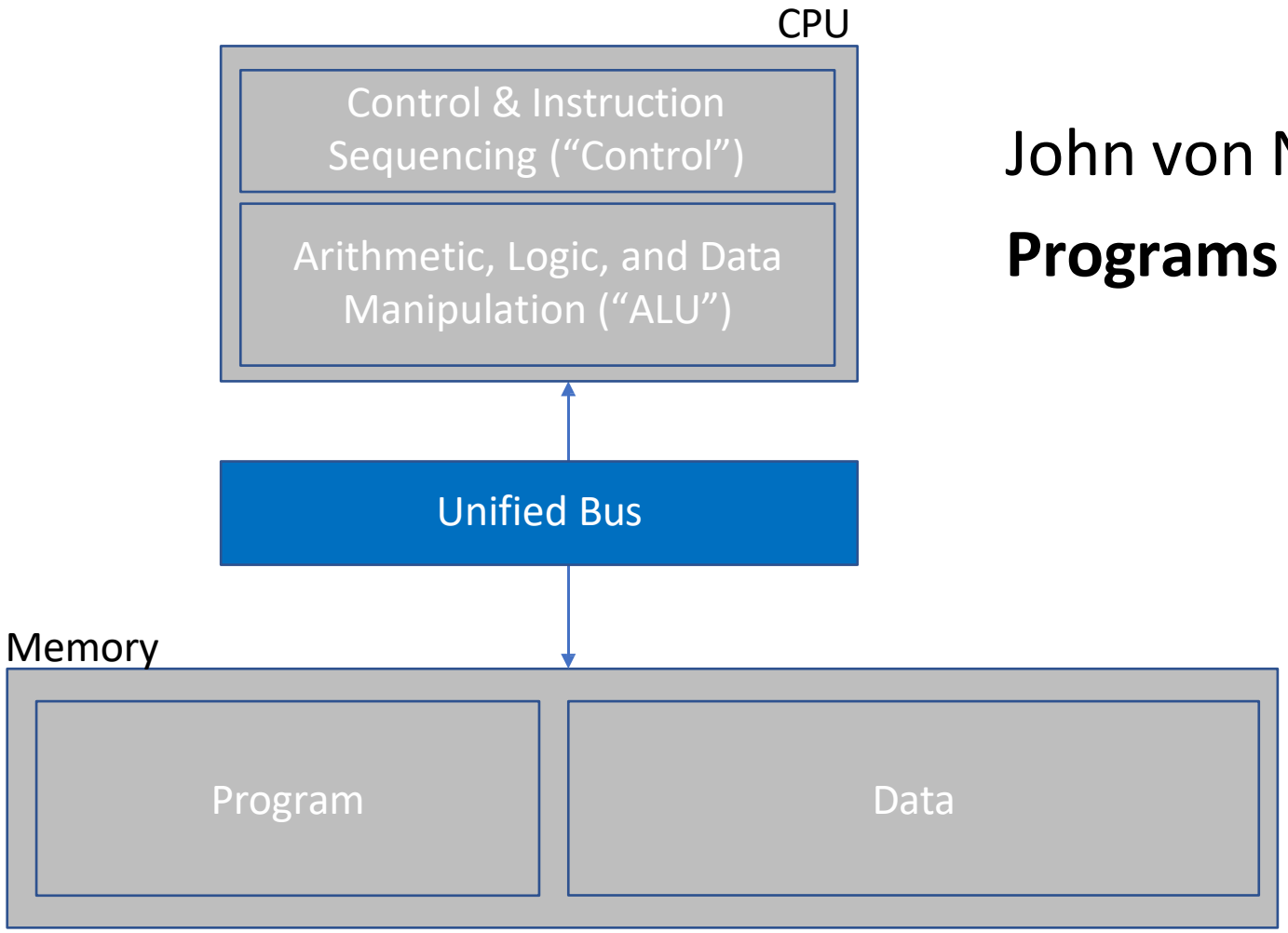
What did we talk about last time?

- Hardware vs. software tradeoffs
- von Neumann vs. Harvard architecture and the beginnings of a design space
- An optimization exercise by example
- Amdahl's Law (and Gustafson's Law, by contrast)

What is a Computer Architecture?

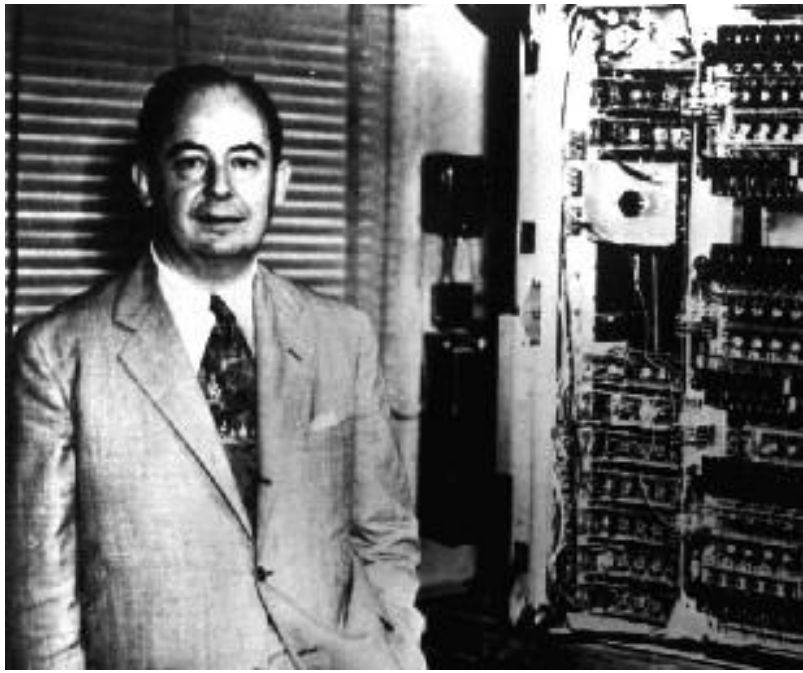
- Building up to our first architecture
- Defining the ISA: Architecture vs. Microarchitecture
- RISC vs. CISC ISAs
- RISCV ISA

Recall:
The Von Neumann Computing Model

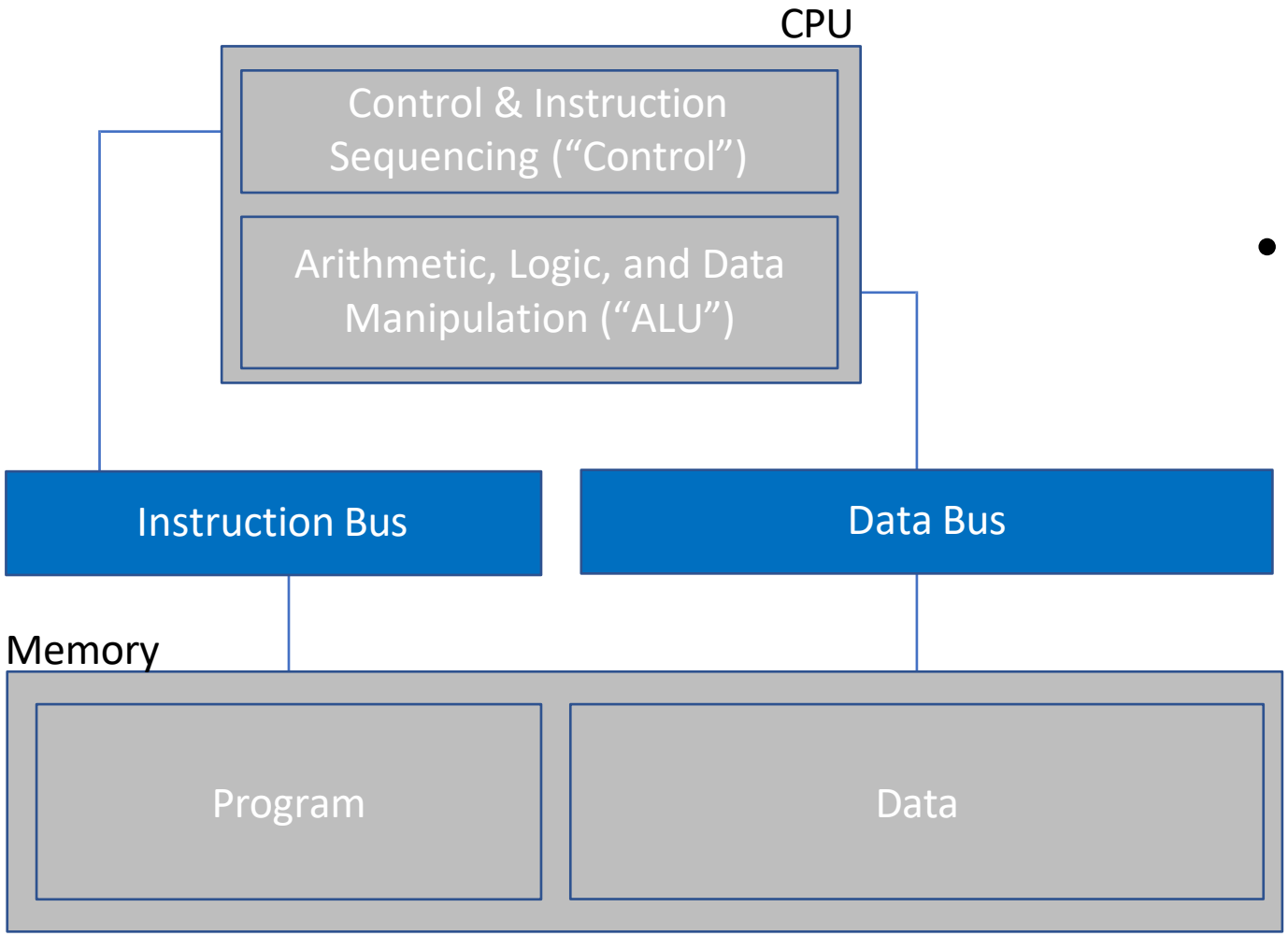


John von Neumann’s Big Idea:

Programs are data.

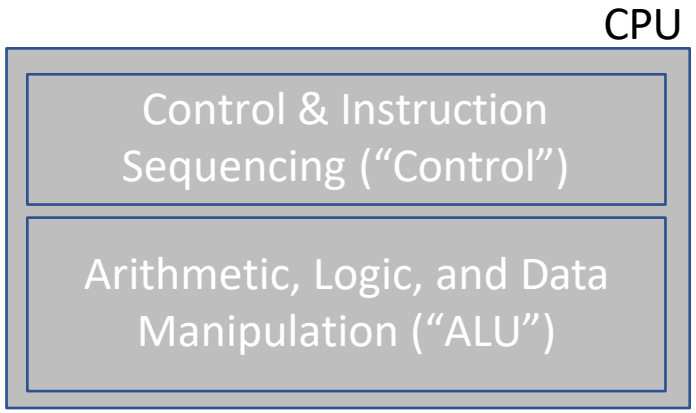


Recall: the Harvard Architecture



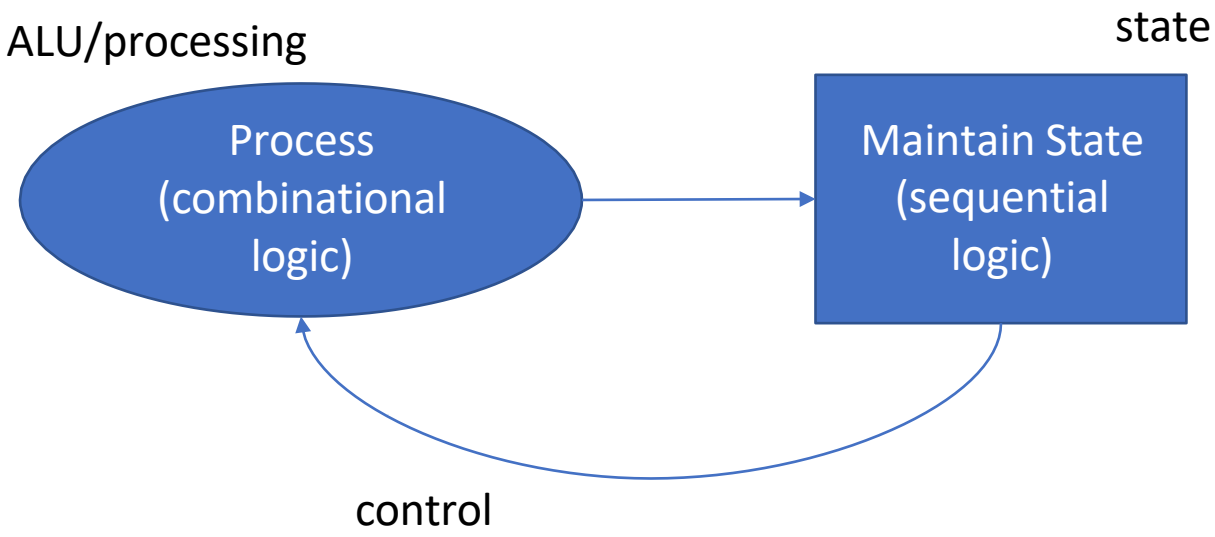
- Split bus architecture provides *simultaneous* access to program and to data

Either Way:
Our CPU from last time is incomplete

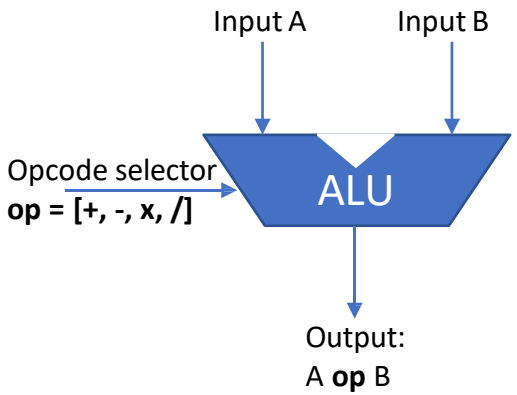


What’s missing?

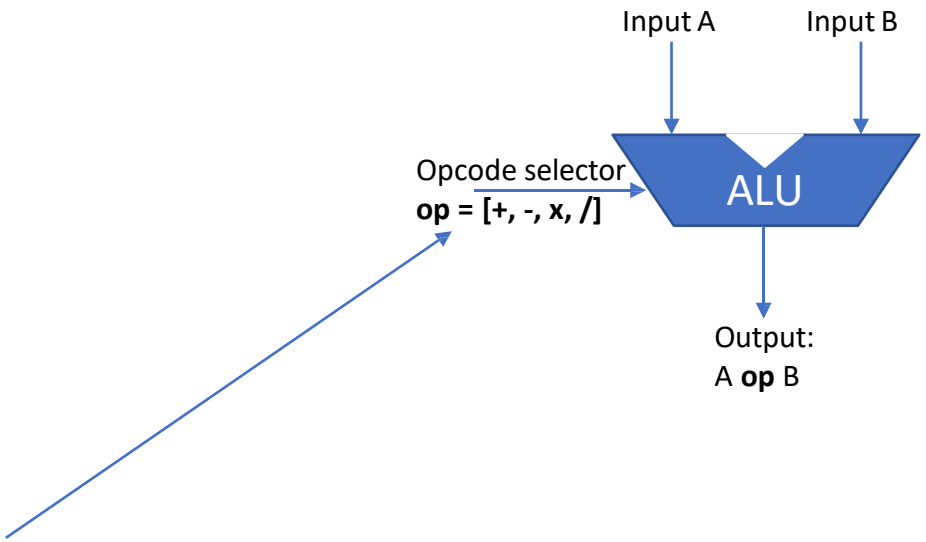
Basic Architecture: State + processing elements



Building up to our first architecture: ALU

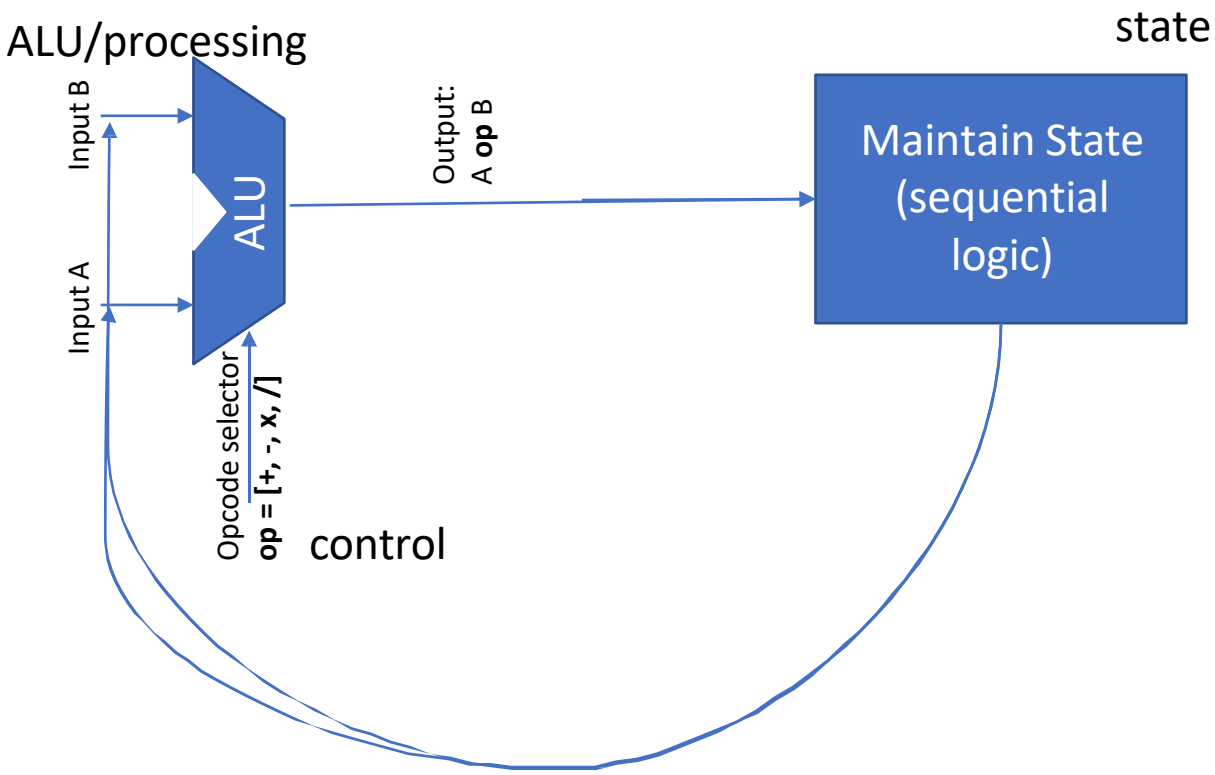


Building up to our first architecture: ALU

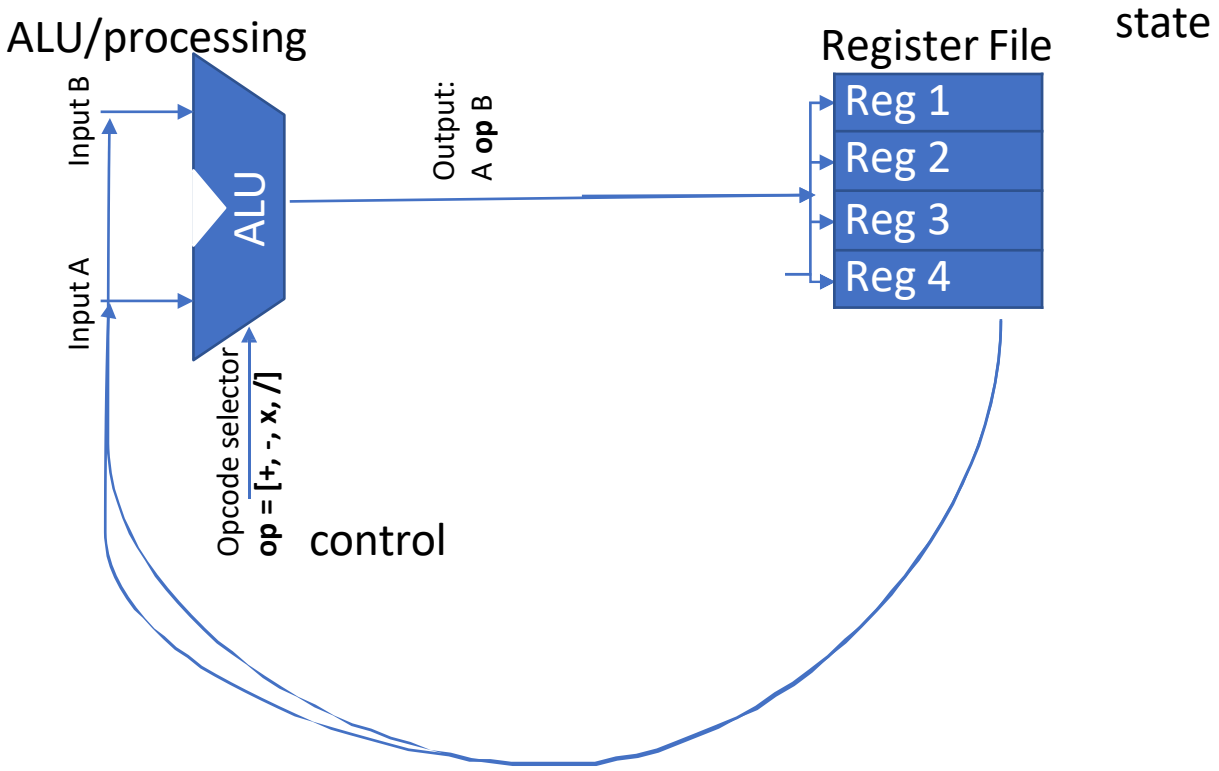


Design choice – what operations do we support here?

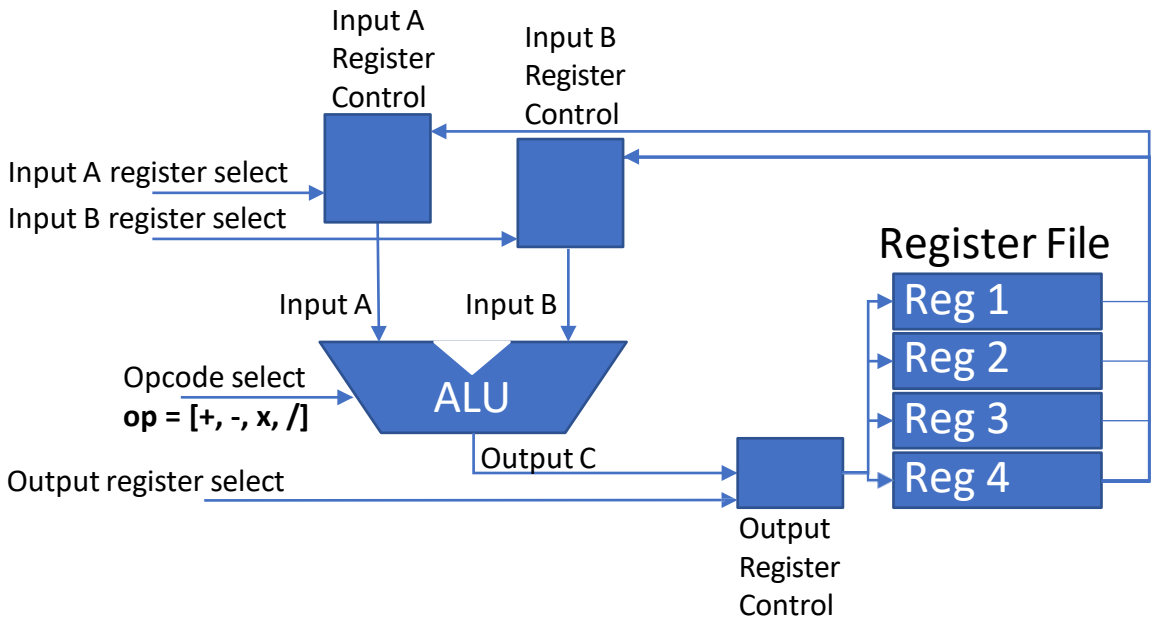
Basic Architecture: State + processing elements



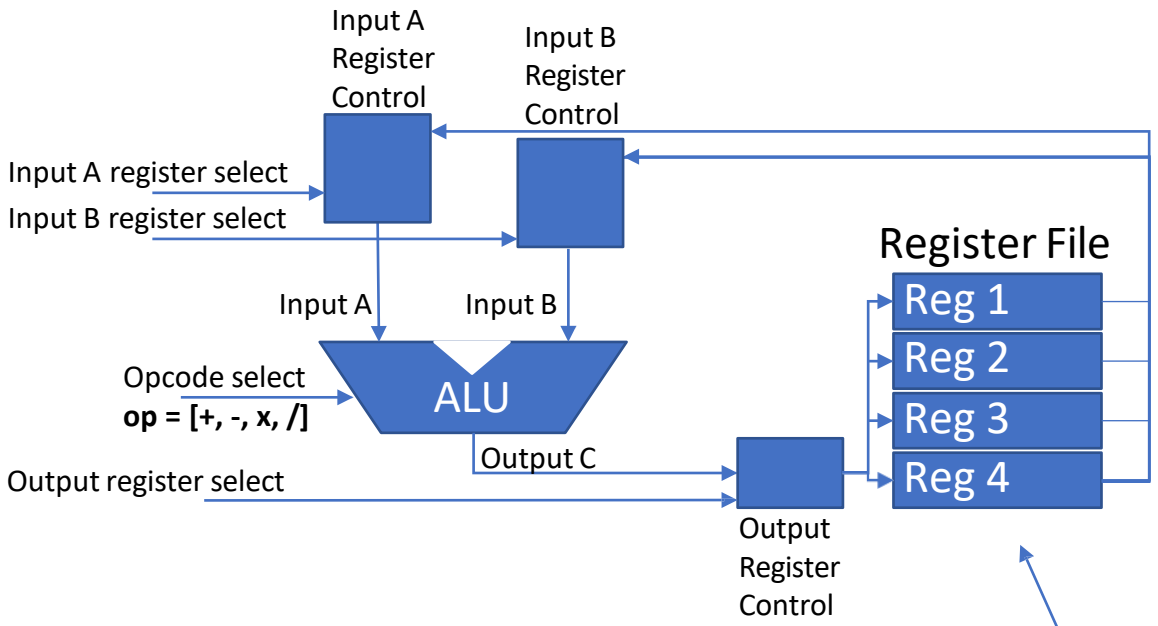
Basic Architecture: State + processing elements



Building up to our first architecture: ALU + Registers

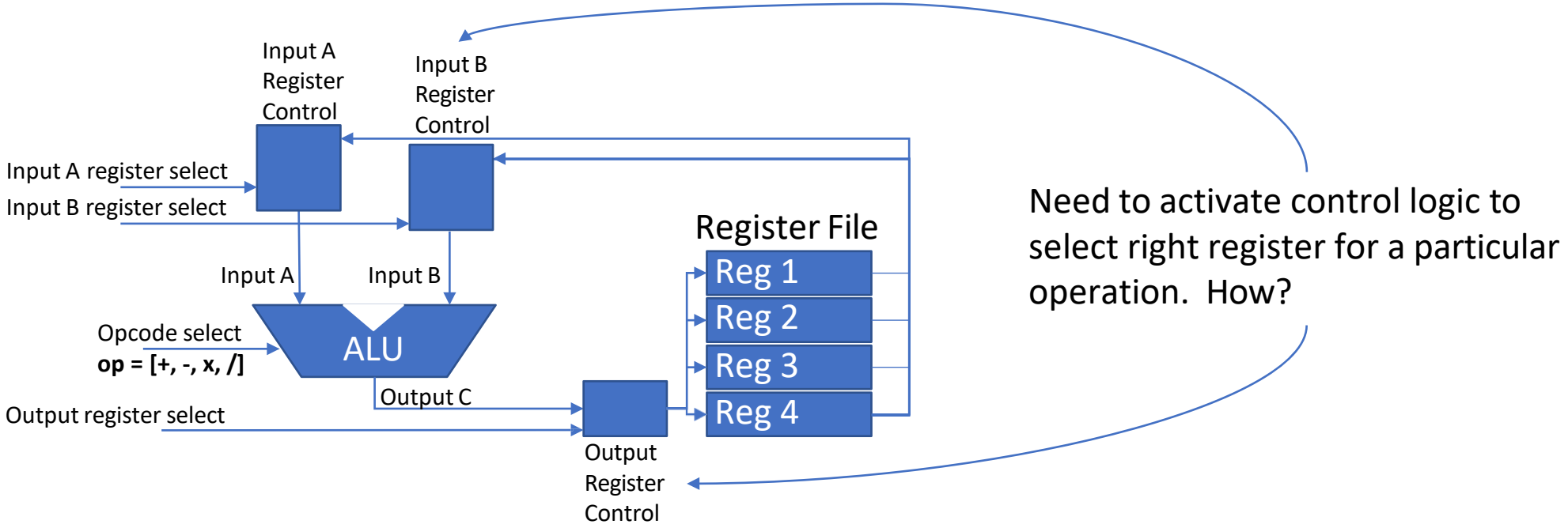


Building up to our first architecture: ALU + Registers



Stateful Elements plus control required to access them, providing inputs to operations and storing outputs of operations

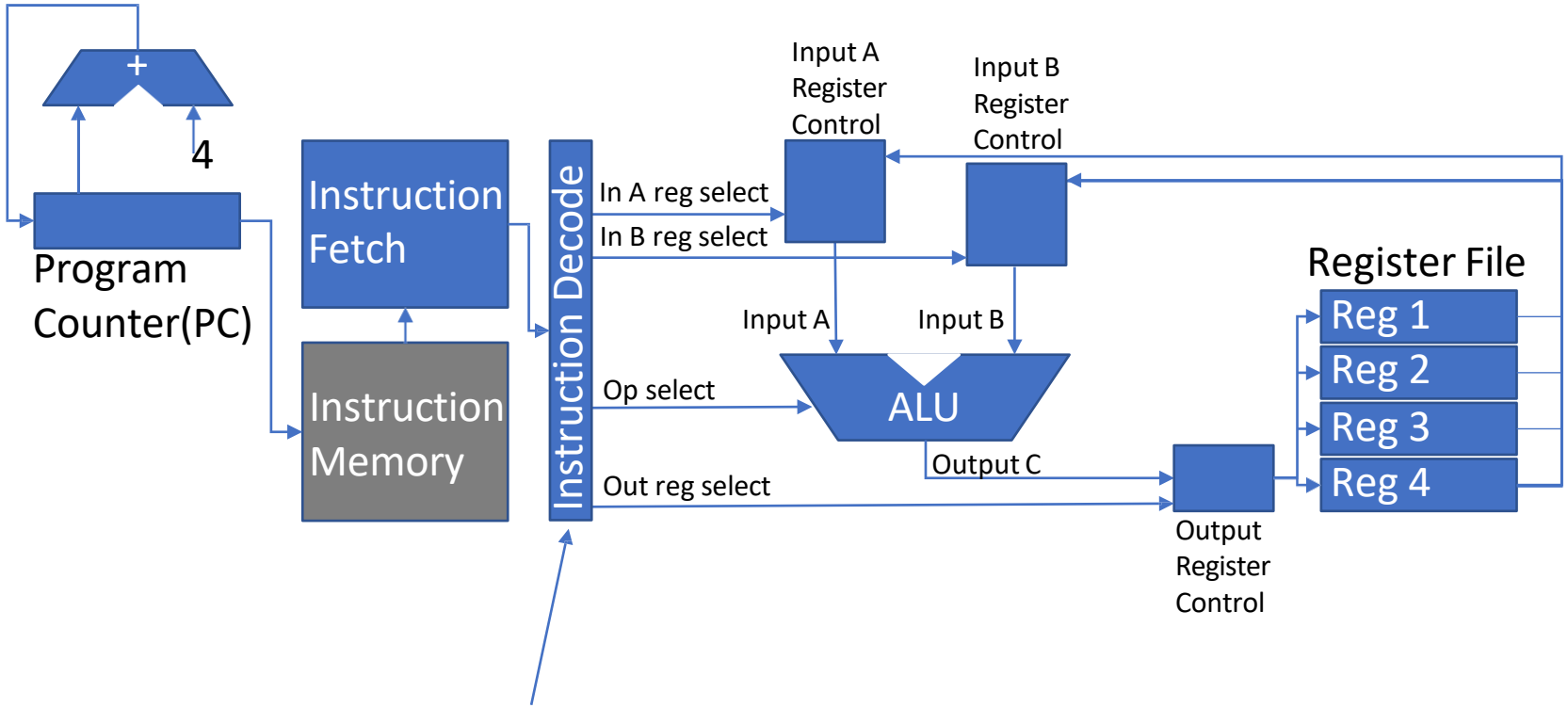
Building up to our first architecture: ALU + Registers



Registers are **named & explicit**.
Implication of explicit names?

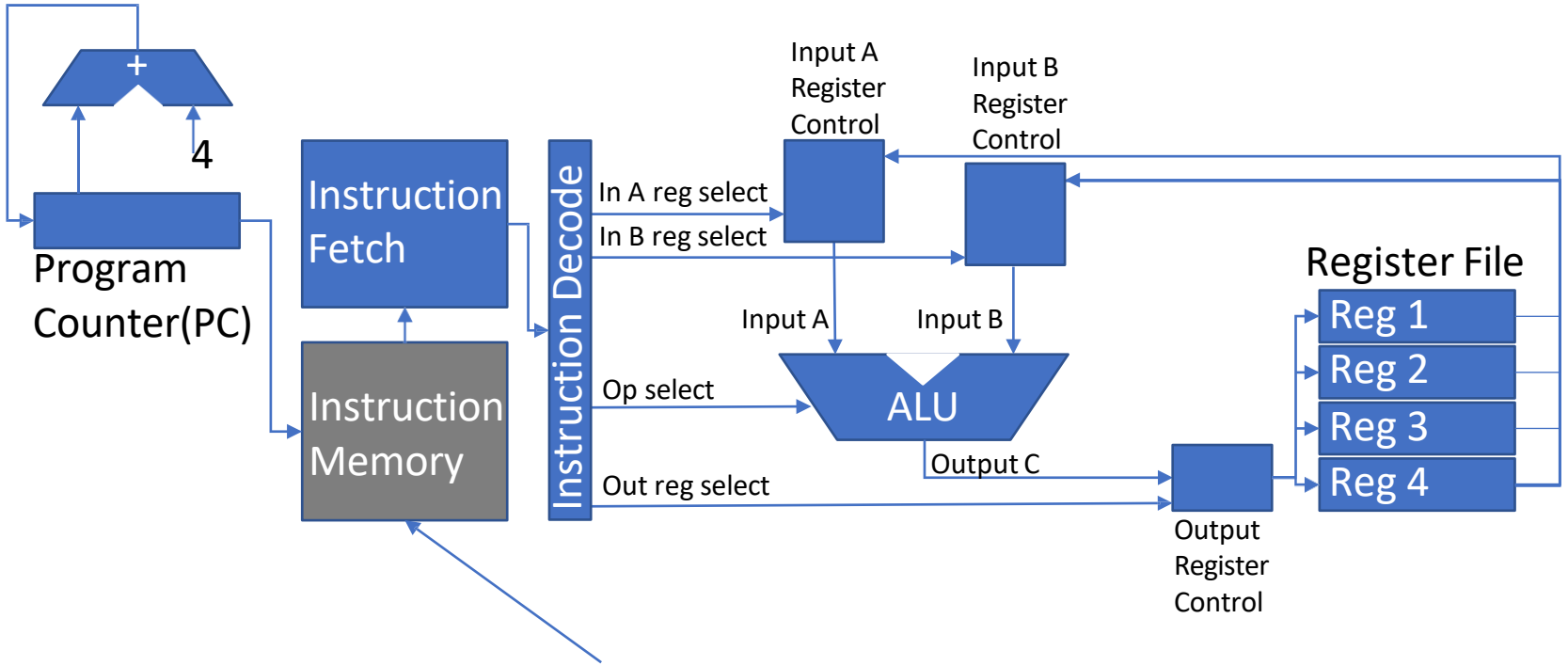
Design choice – how many stateful elements / registers do we support?

Building up to our first architecture: Control



Instruction gets decoded into signals that control the other parts of the system (more on encoding / decoding in a few slides)

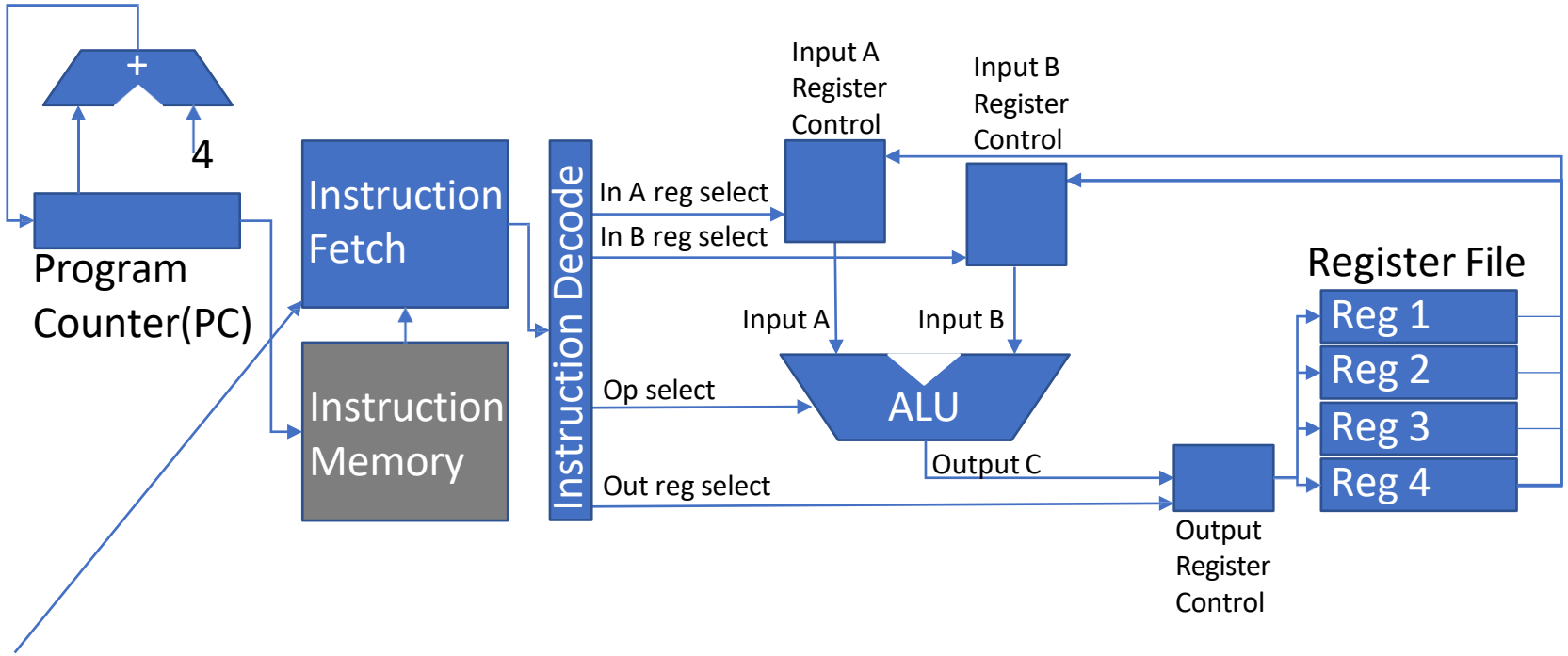
Building up to our first architecture: Control



Instruction memory holds all of the bits of all of the instructions that we might ever use to control other units.

Design choice: Need to think about where we put this memory (and its hierarchy of caches)

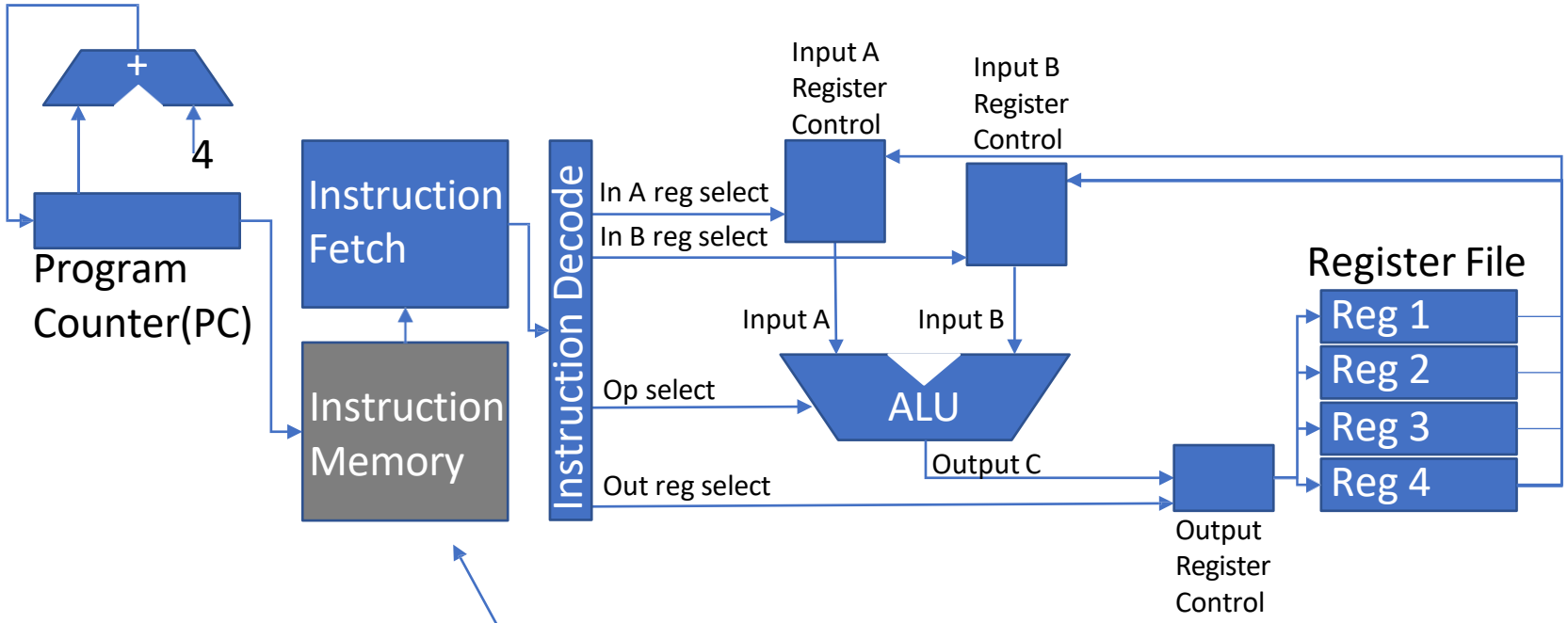
Building up to our first architecture: Control



Instruction fetch logic refers to PC, loads instruction from instruction memory and sends to decode.

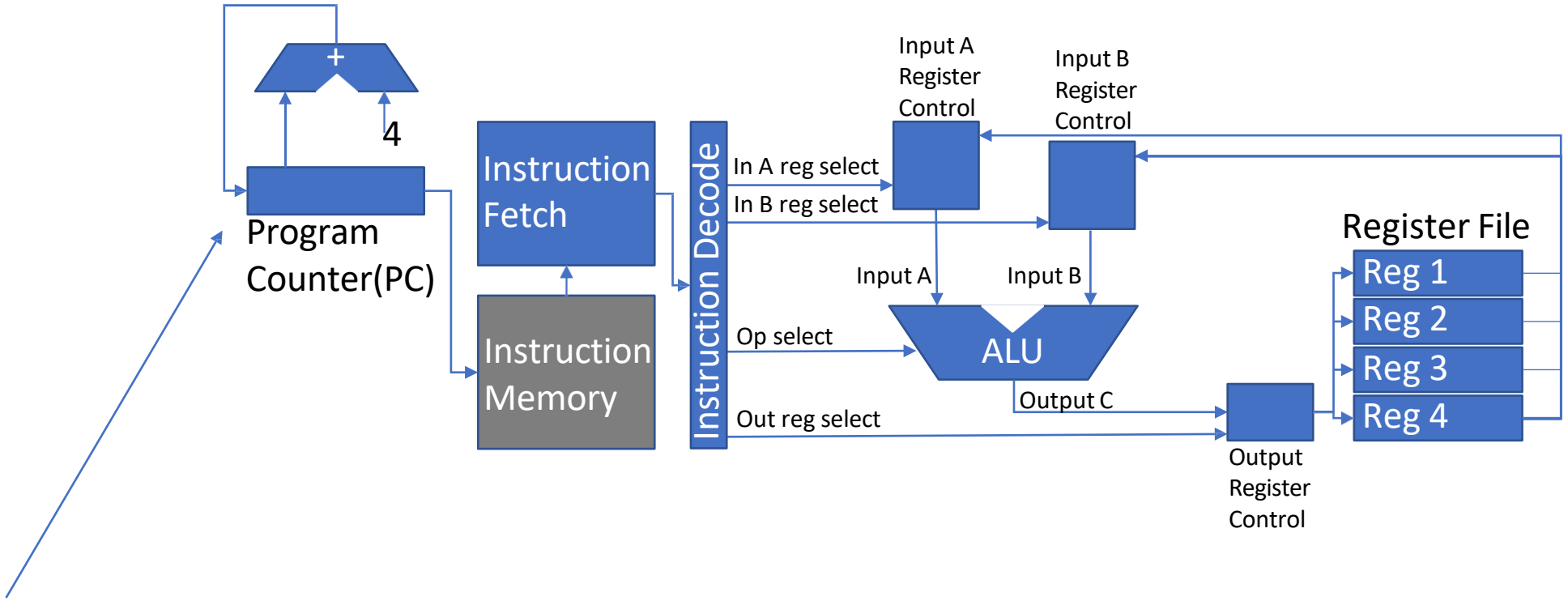
Design choices: how much to fetch at once? What to fetch next (not always obvious)?

Building up to our first architecture: Control



Remember our fetch optimization from last time? That would go here. Specialized instruction memory access logic. (Physical memory may be the same, though)

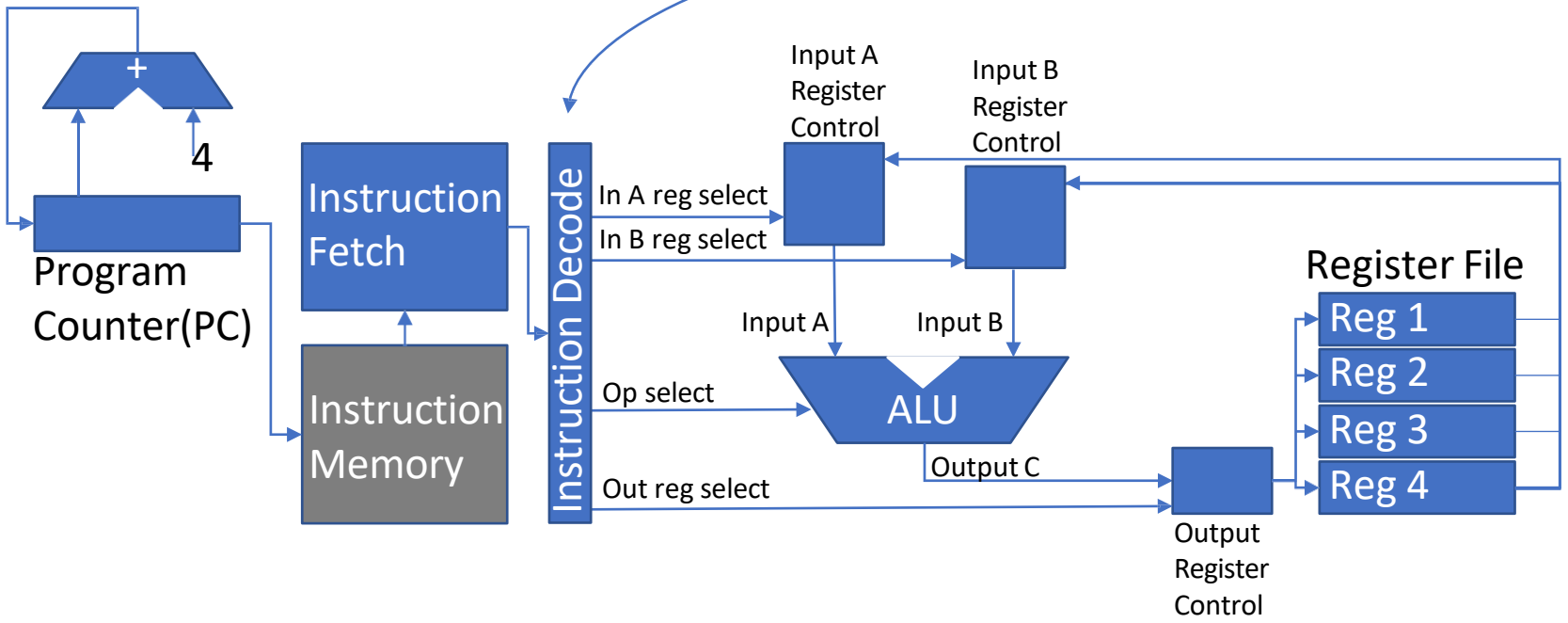
Building up to our first architecture: Control



Sequential Control:
Each cycle, update the PC by adding 4.
Implication for software of our current design?

Building up to our first architecture: Control

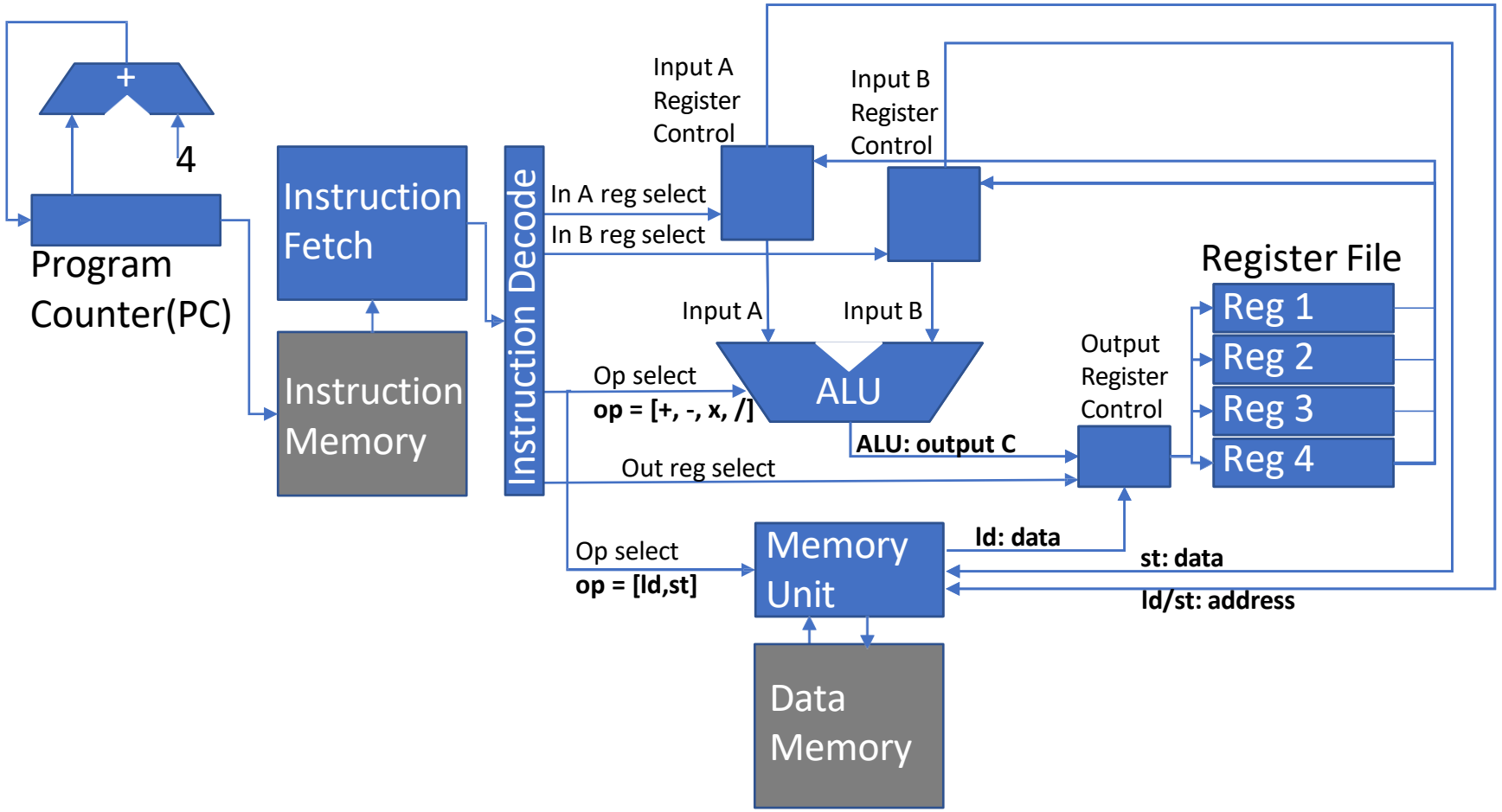
Decoding a fetched operation breaks it from a blob of bits into a set of signals that we can use to configure the rest of the units in this diagram



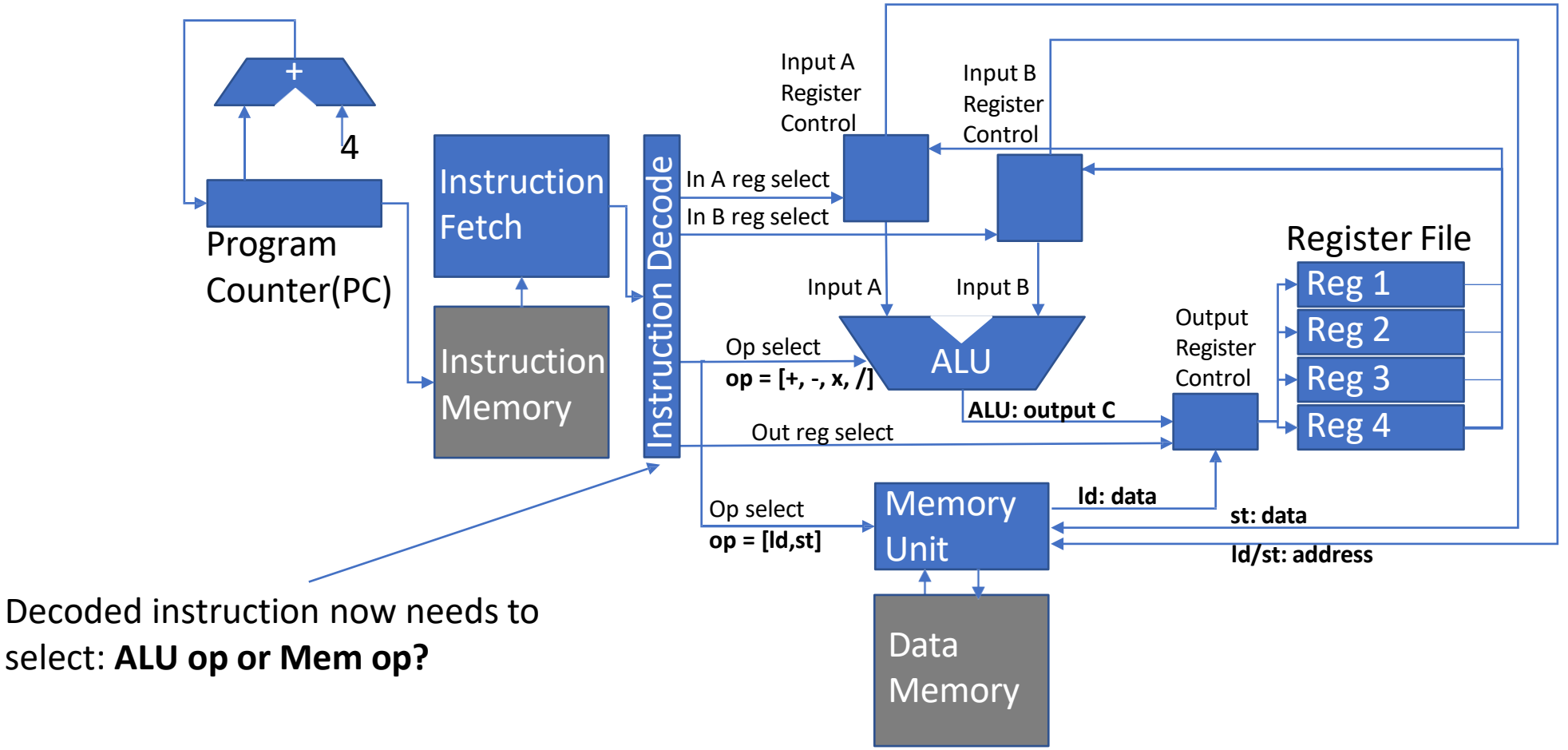
Key Idea: What we encode here has implications for other units and software layers above the instruction definition level.

Mechanism of decoding and **content** of encoded/decoded instructions are orthogonal concepts. **How?** vs **what?.**

Building up to our first architecture: Memory

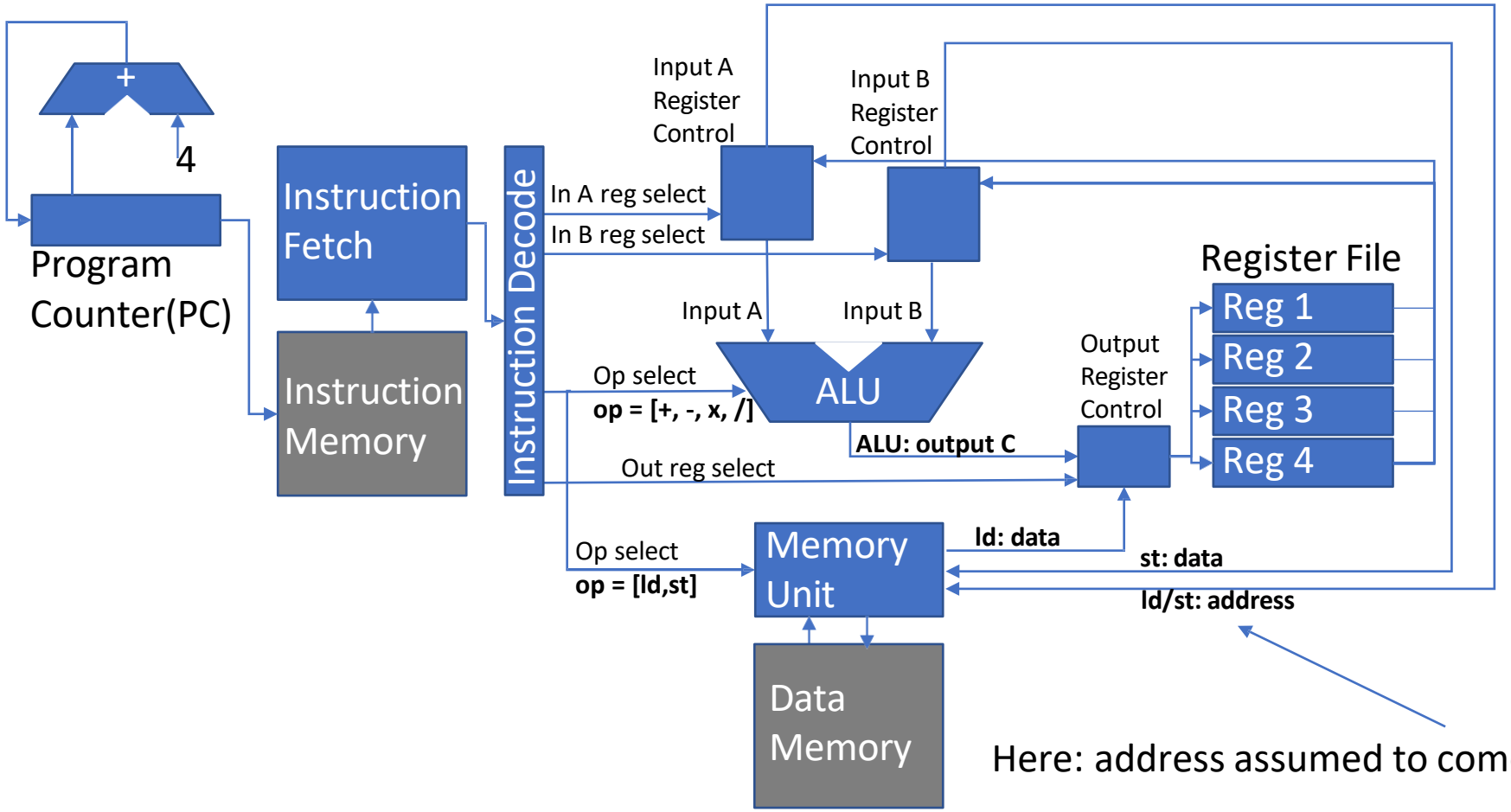


Building up to our first architecture: Memory



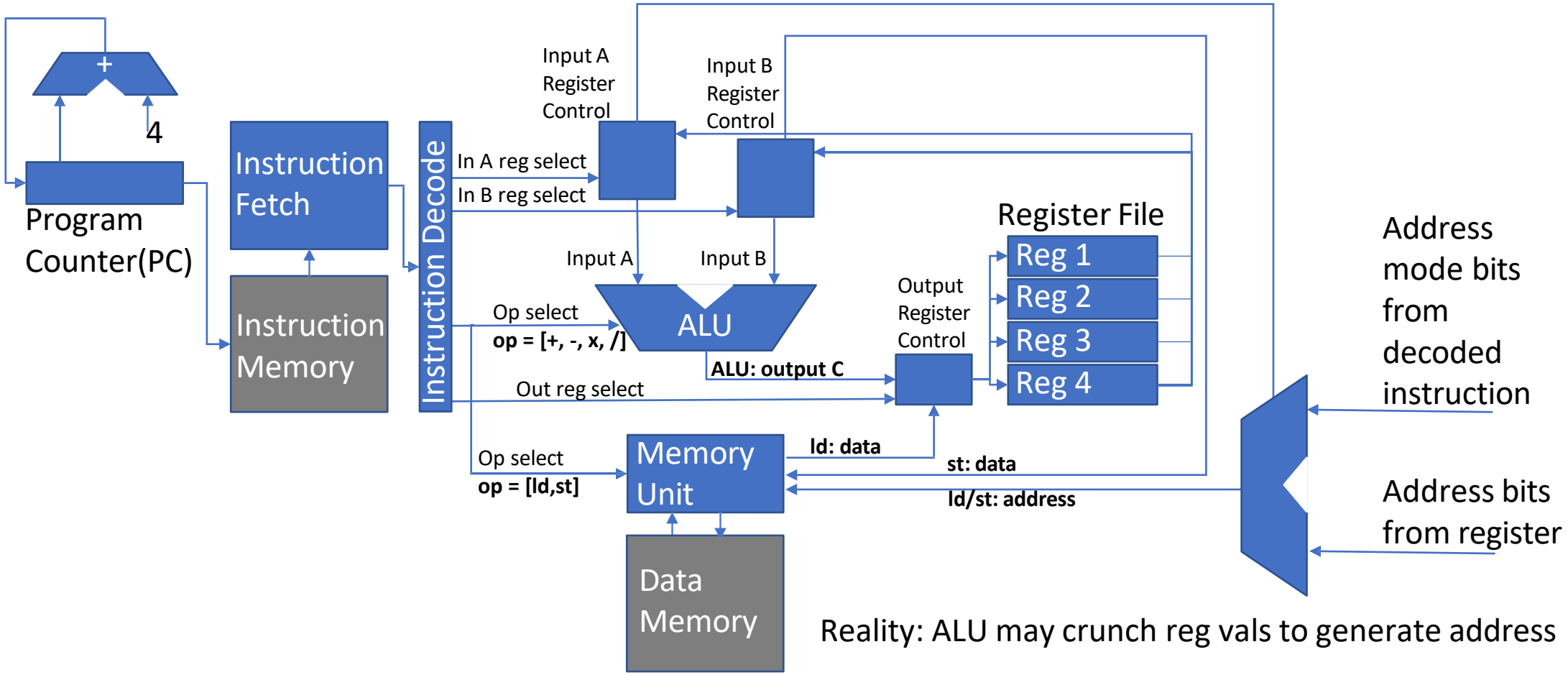
Decoded instruction now needs to select: **ALU op or Mem op?**

Building up to our first architecture: Memory

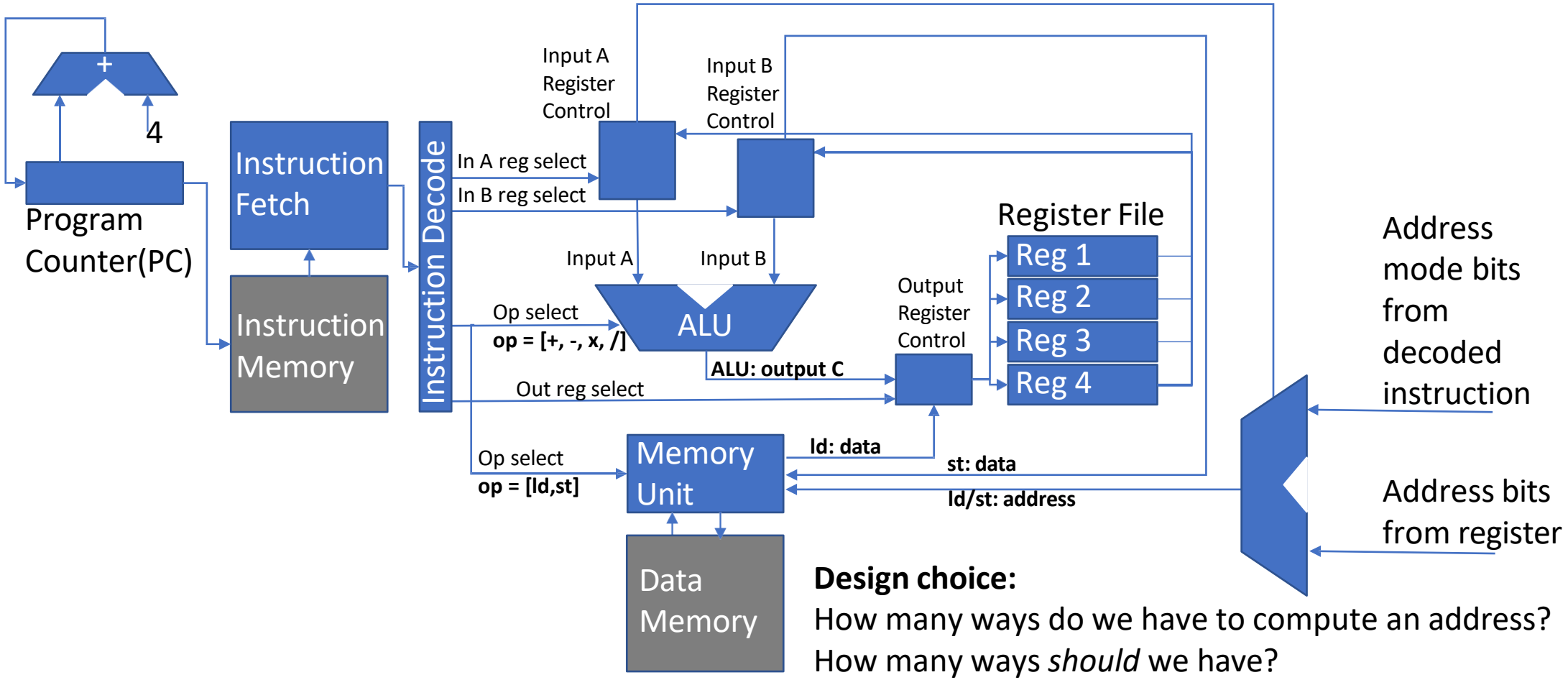


Here: address assumed to come from register directly

Building up to our first architecture: Memory



Building up to our first architecture: Memory



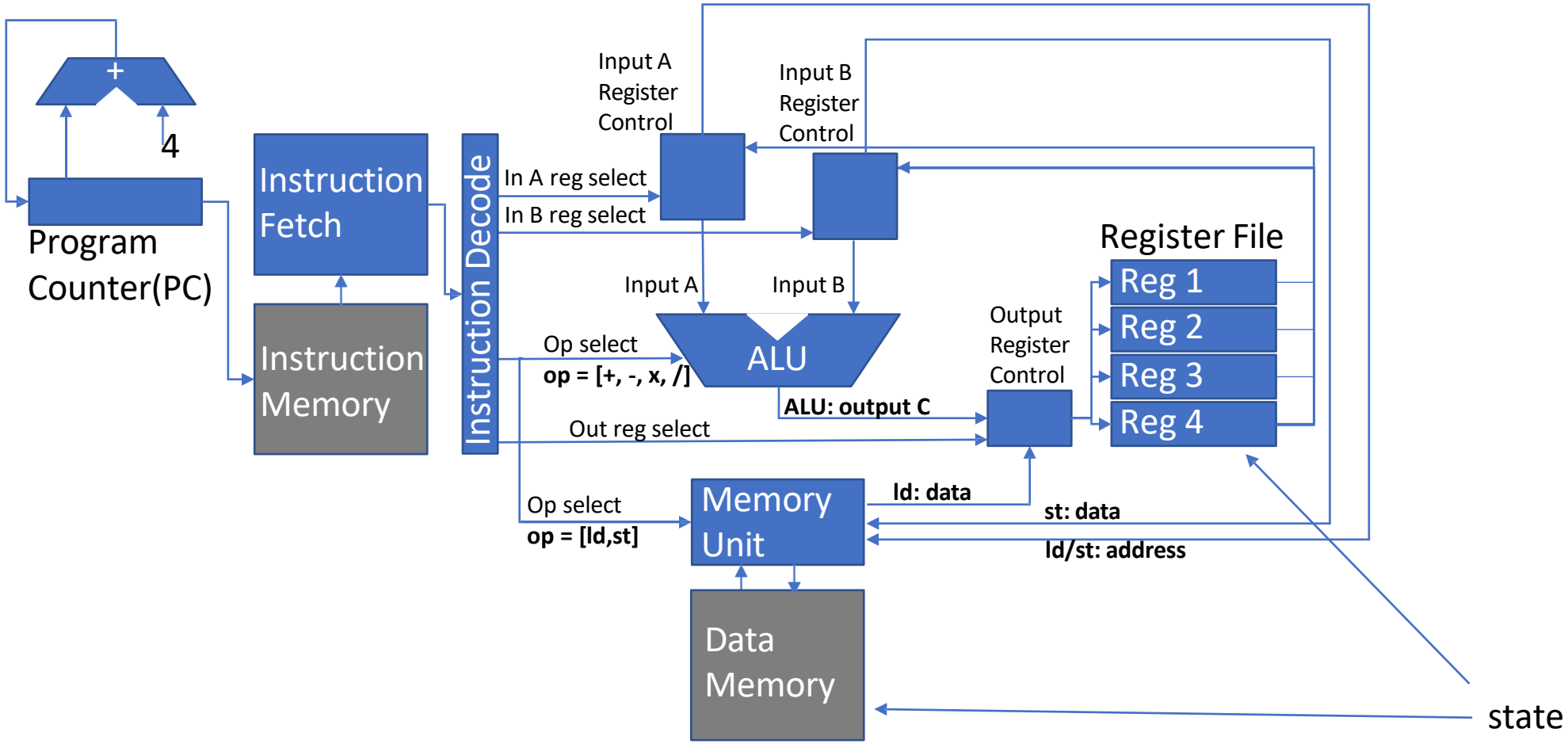
Design choice:

How many ways do we have to compute an address?

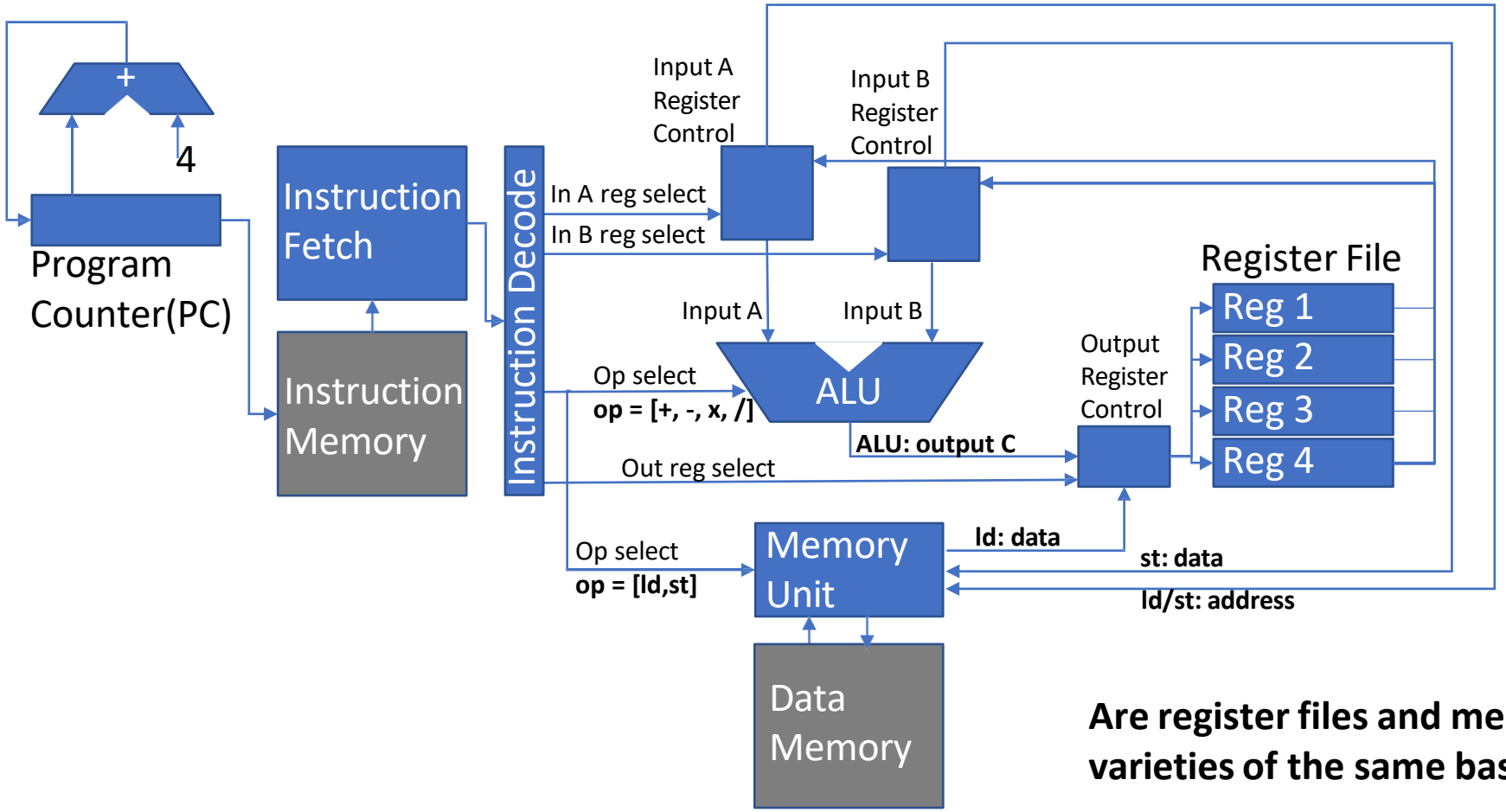
How many ways *should* we have?

Guess at implications of more ways?

Building up to our first architecture: Memory

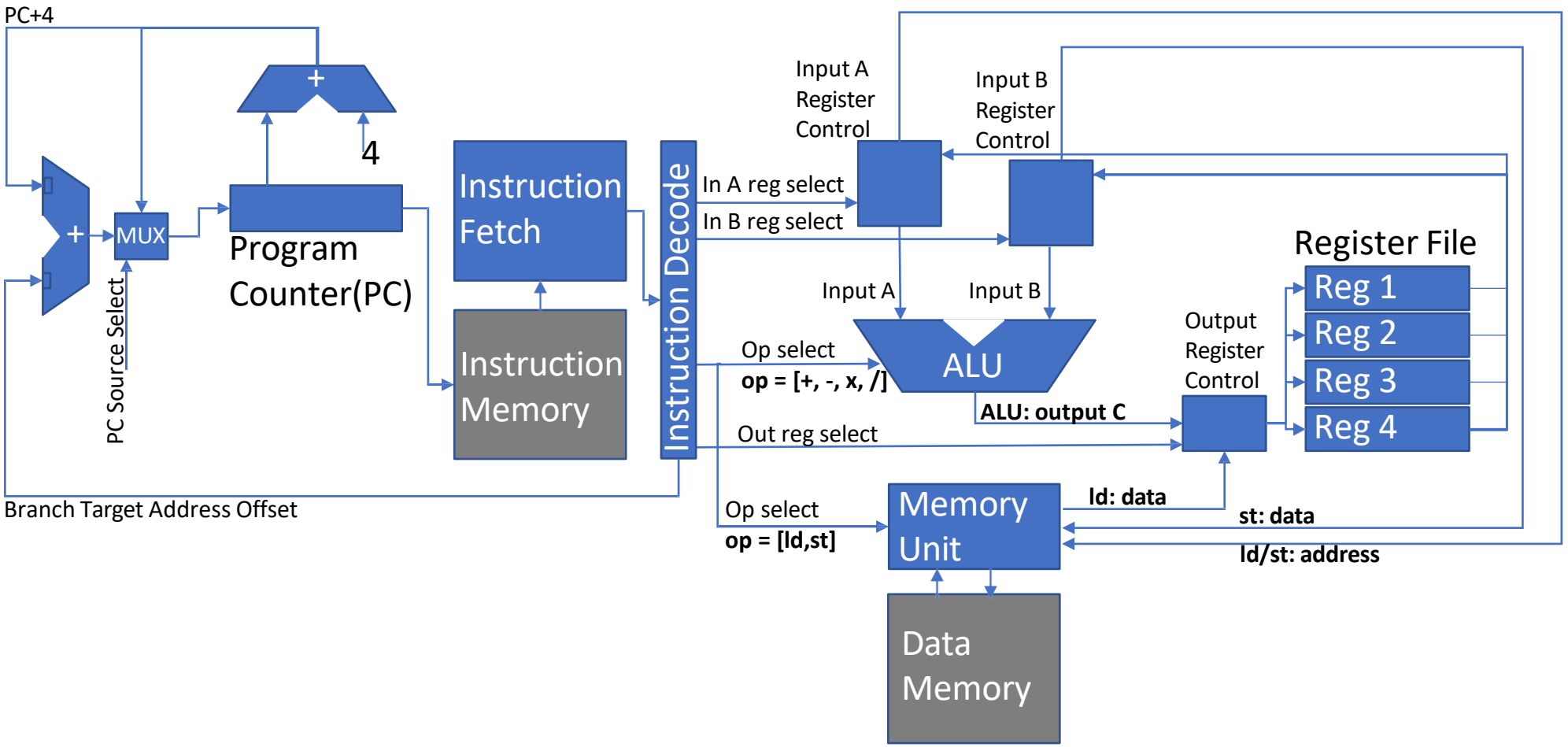


Building up to our first architecture: Memory

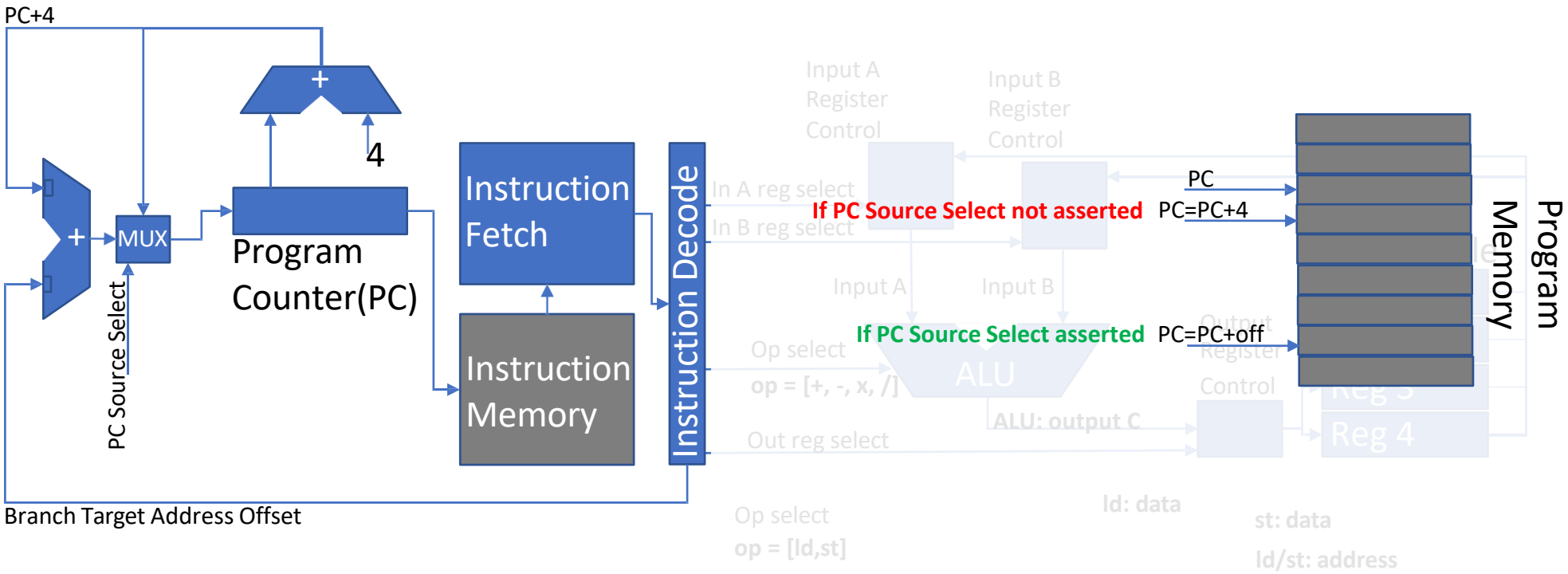


Are register files and memory two varieties of the same basic thing?

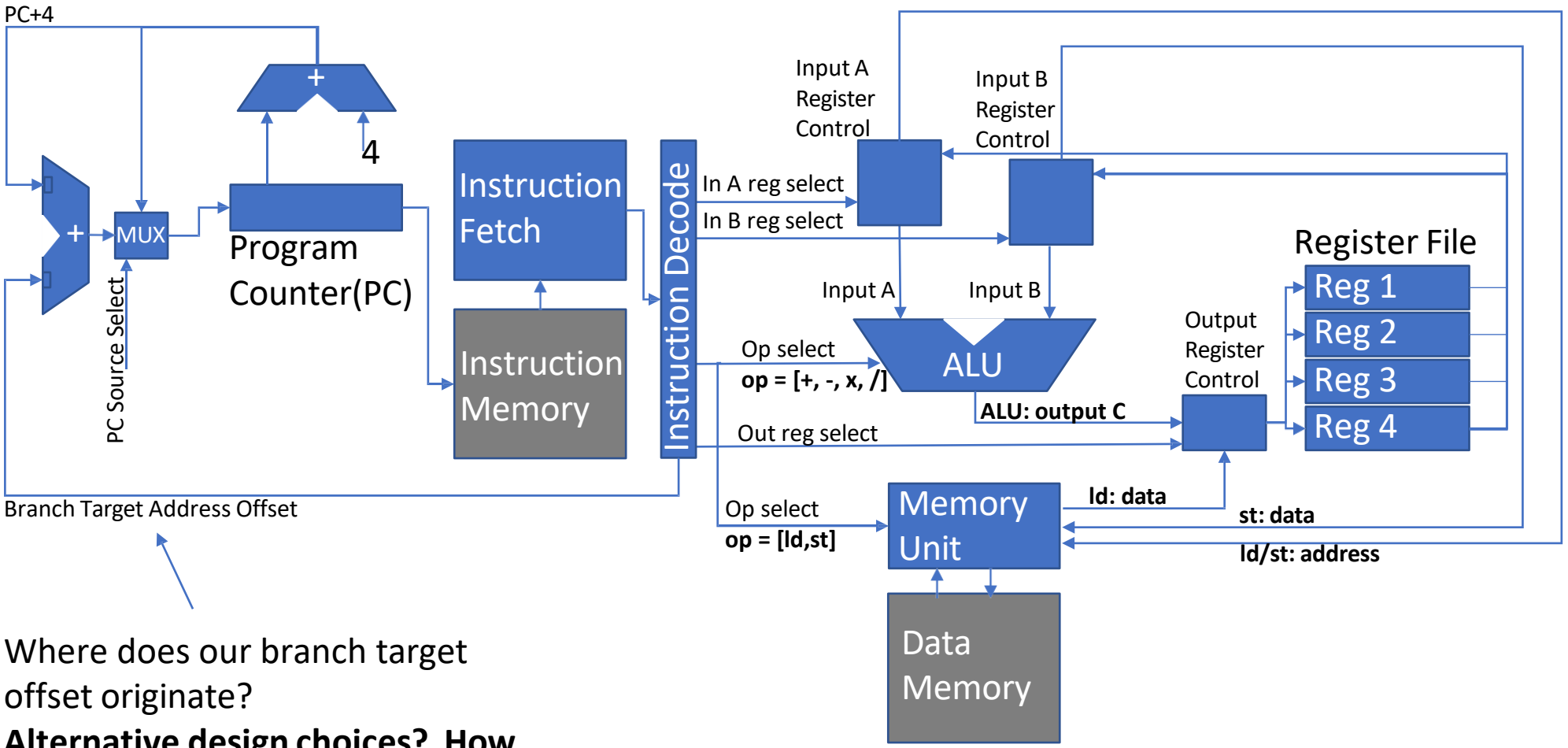
Building up to our first architecture: Branching



Building up to our first architecture: Branching



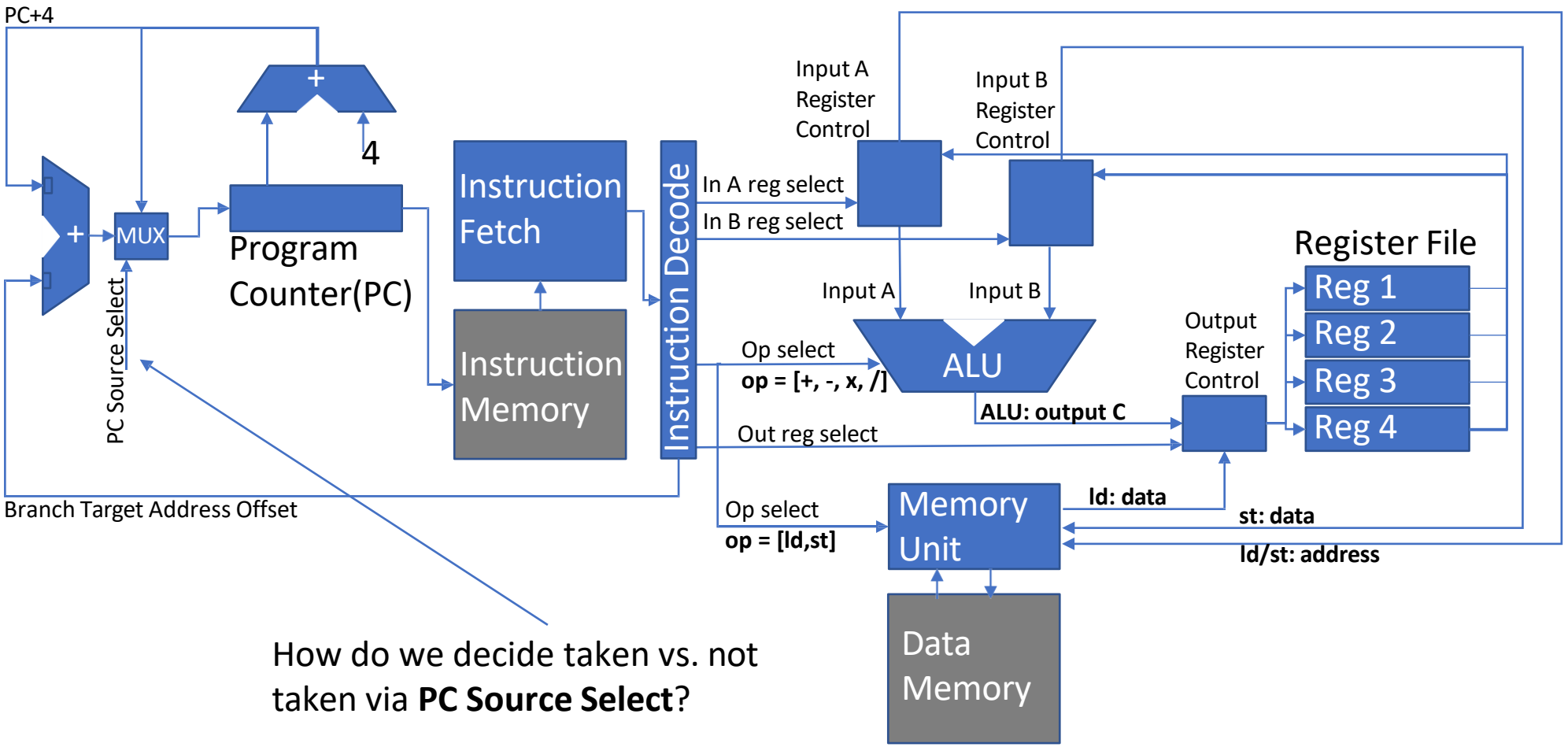
Building up to our first architecture: Branching



Where does our branch target offset originate?

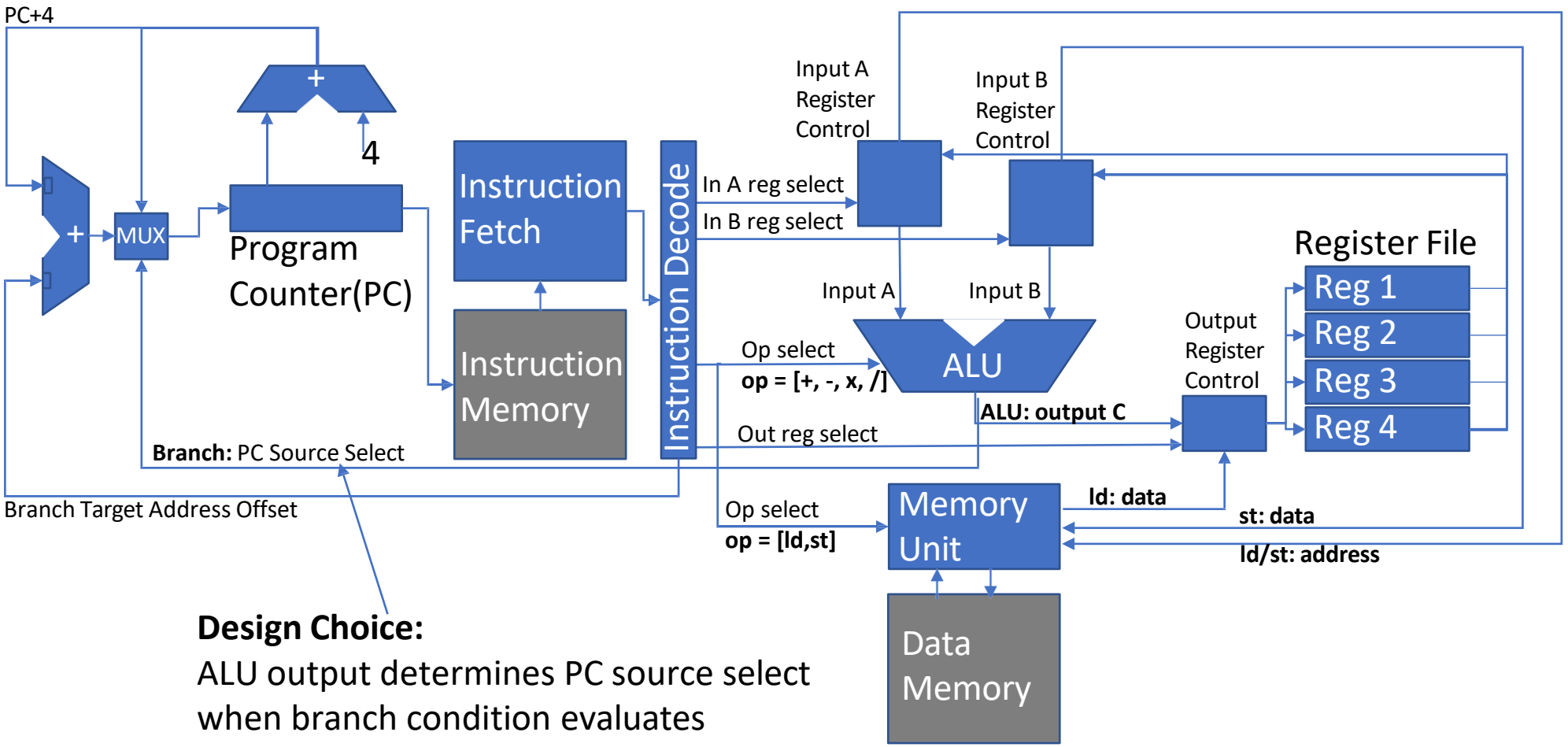
Alternative design choices? How are each of those used in code?

Building up to our first architecture: Branching



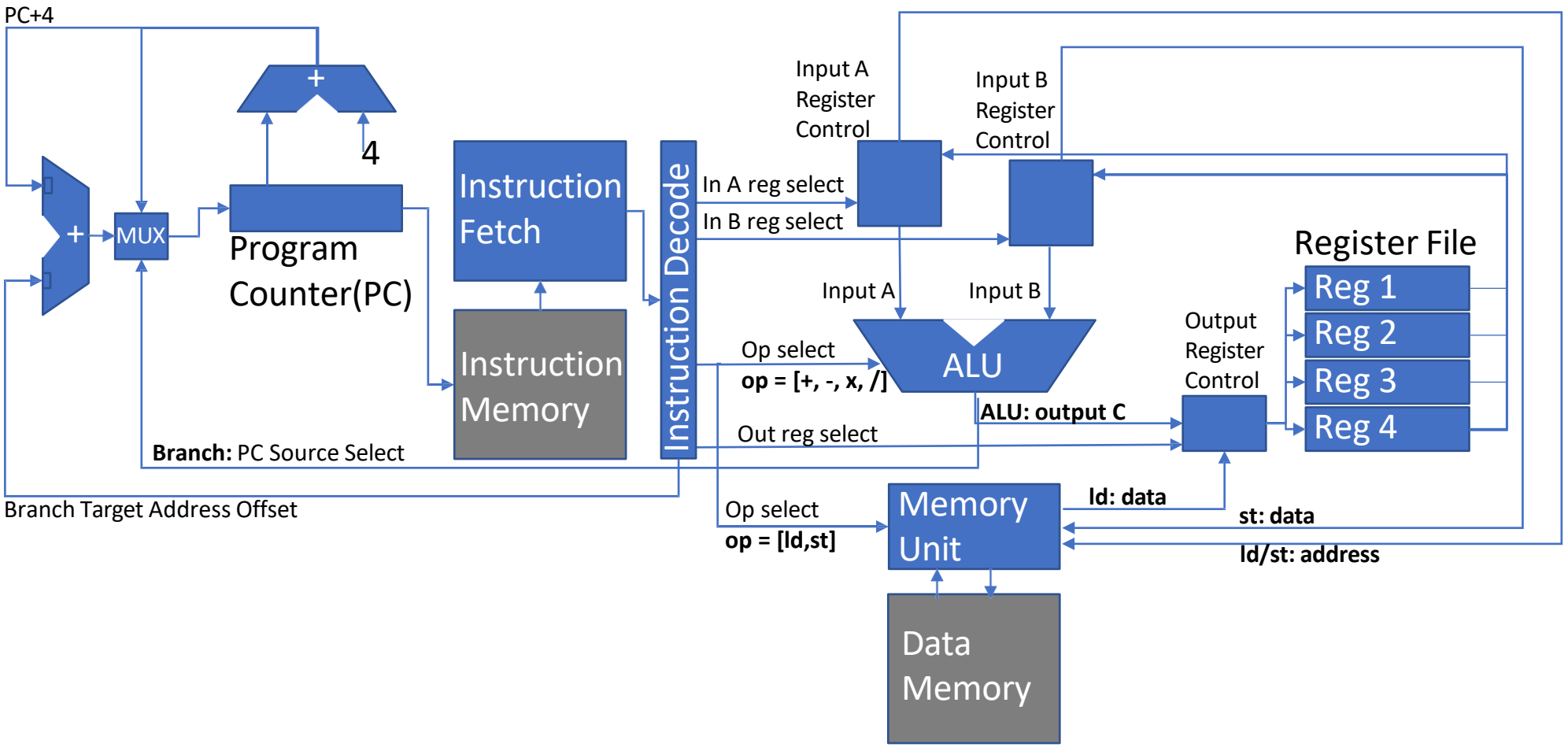
How do we decide taken vs. not taken via **PC Source Select**?

Building up to our first architecture: Branching

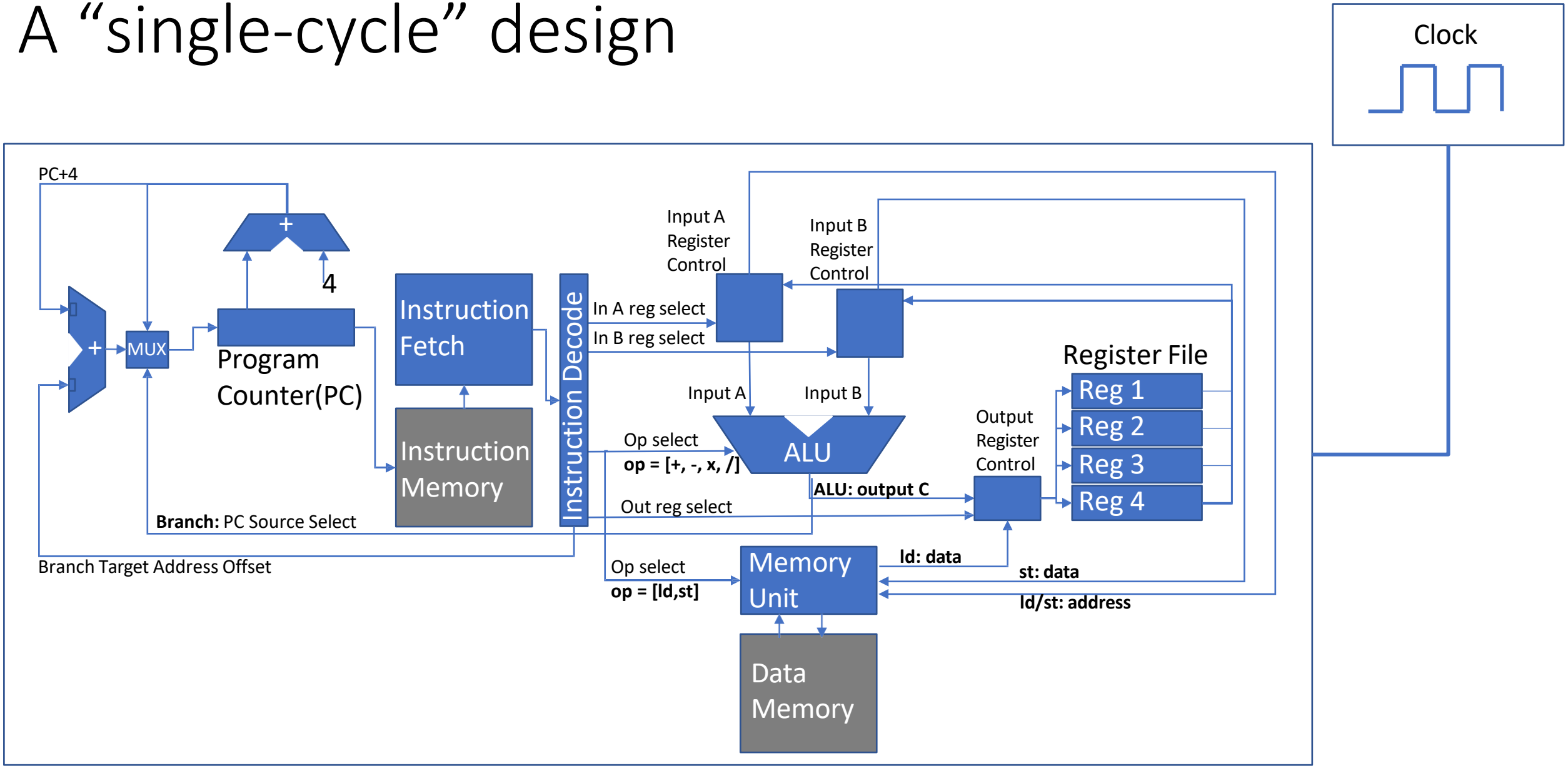


Design Choice:
ALU output determines PC source select when branch condition evaluates
Alternative Design?

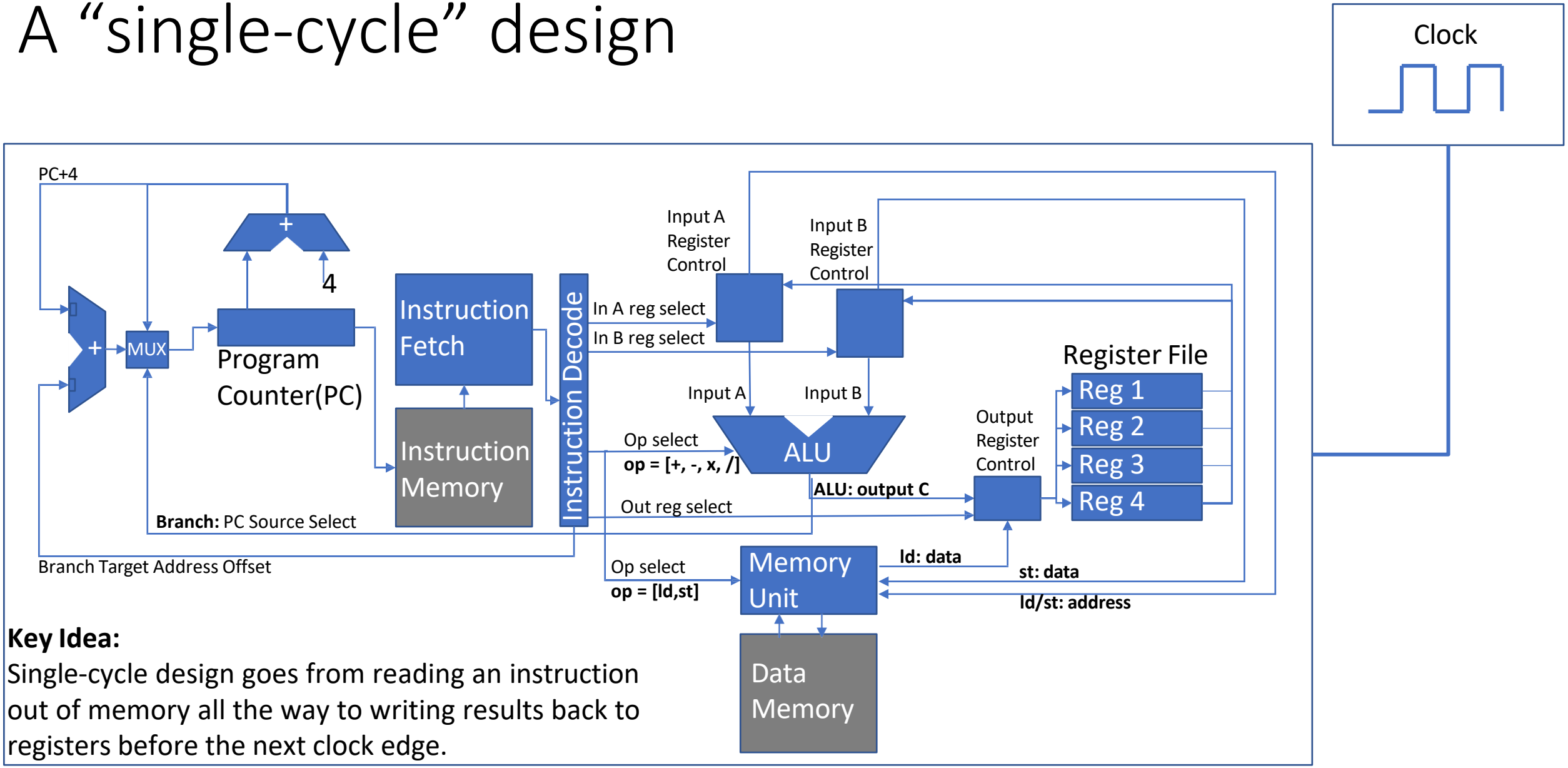
A Complete (but slightly messy) RISCV-ish Datapath



A “single-cycle” design

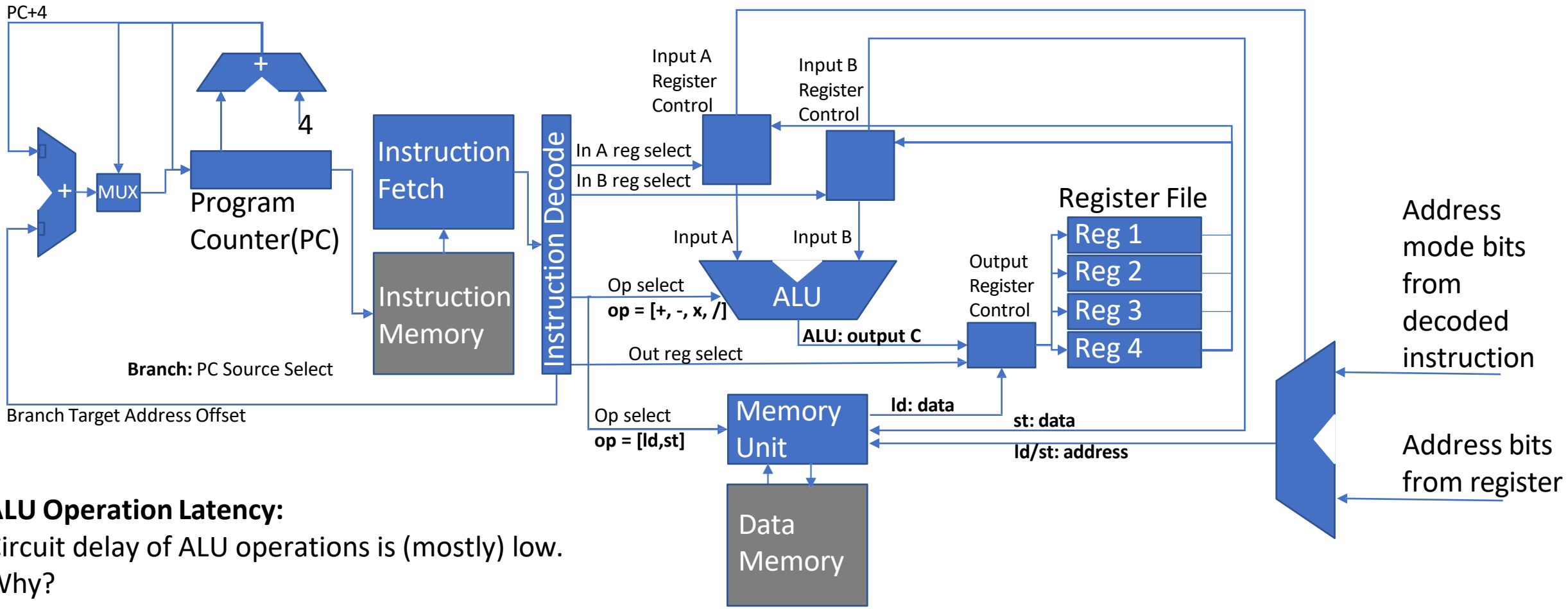


A “single-cycle” design



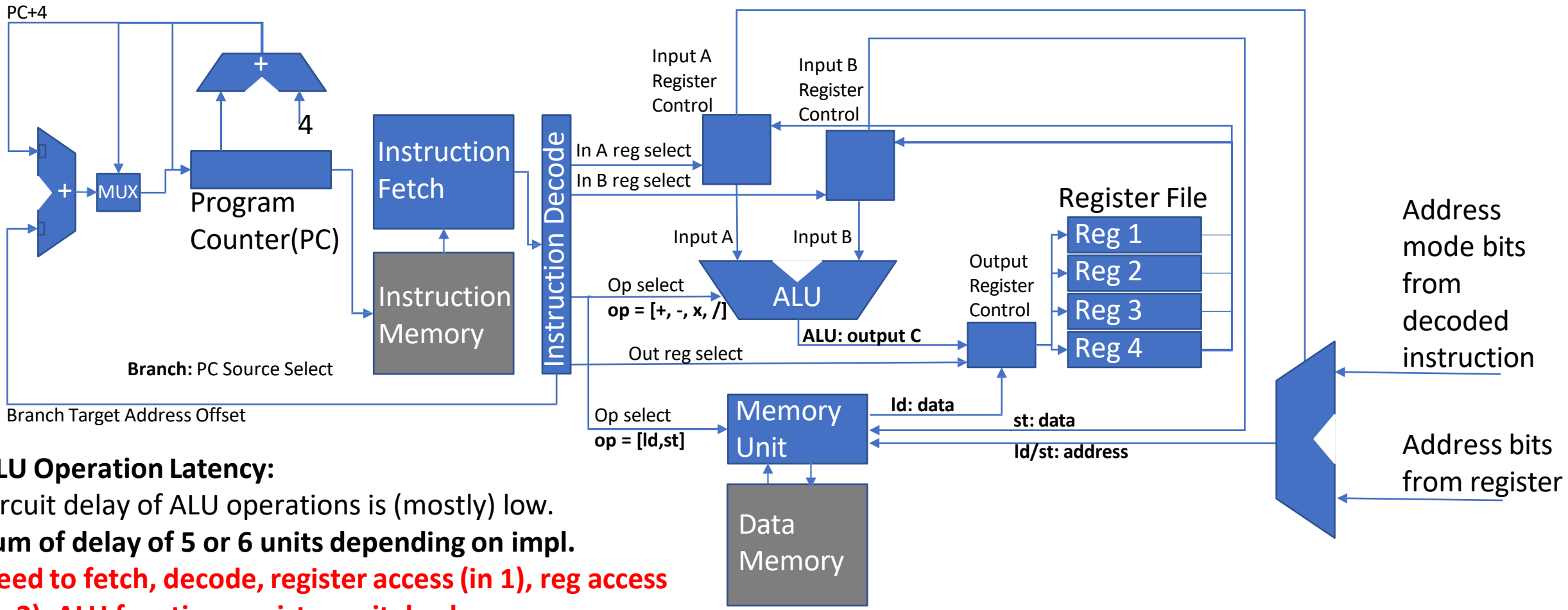
Key Idea:
Single-cycle design goes from reading an instruction out of memory all the way to writing results back to registers before the next clock edge.

Thinking about latency: ALU Operations



ALU Operation Latency:
Circuit delay of ALU operations is (mostly) low.
Why?

Thinking about latency: ALU Operations



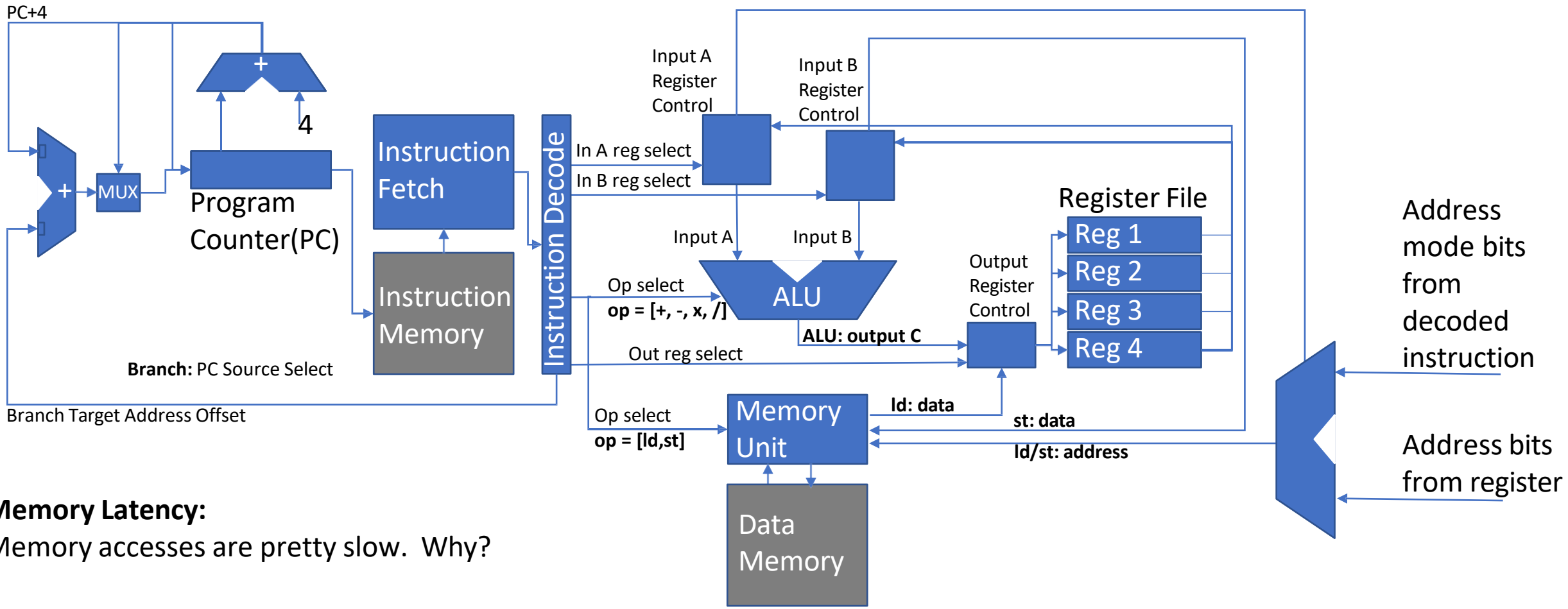
ALU Operation Latency:

Circuit delay of ALU operations is (mostly) low.

Sum of delay of 5 or 6 units depending on impl.

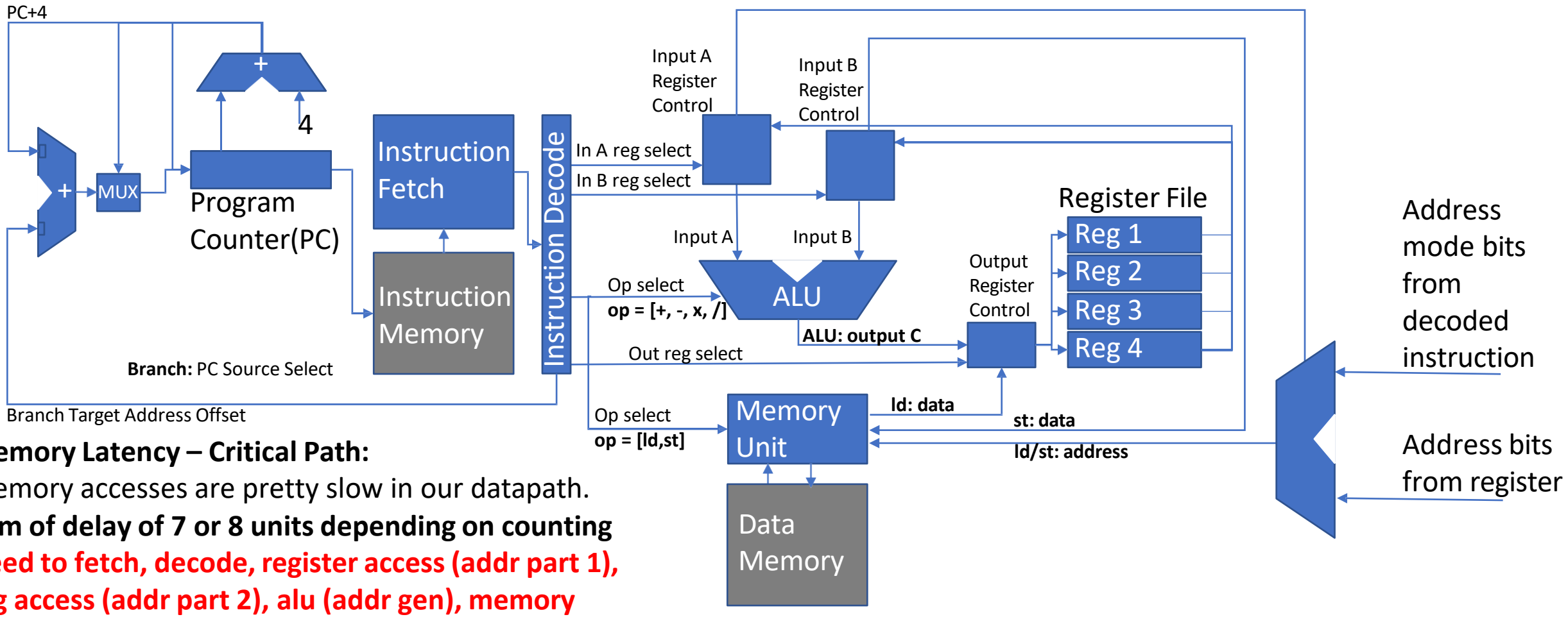
Need to fetch, decode, register access (in 1), reg access (in 2), ALU function, register writeback

Thinking about latency: Memory



Memory Latency:
Memory accesses are pretty slow. Why?

Thinking about latency: Memory



Memory Latency – Critical Path:

Memory accesses are pretty slow in our datapath.

Sum of delay of 7 or 8 units depending on counting

Need to fetch, decode, register access (addr part 1), reg access (addr part 2), alu (addr gen), memory unit, memory bank, register write

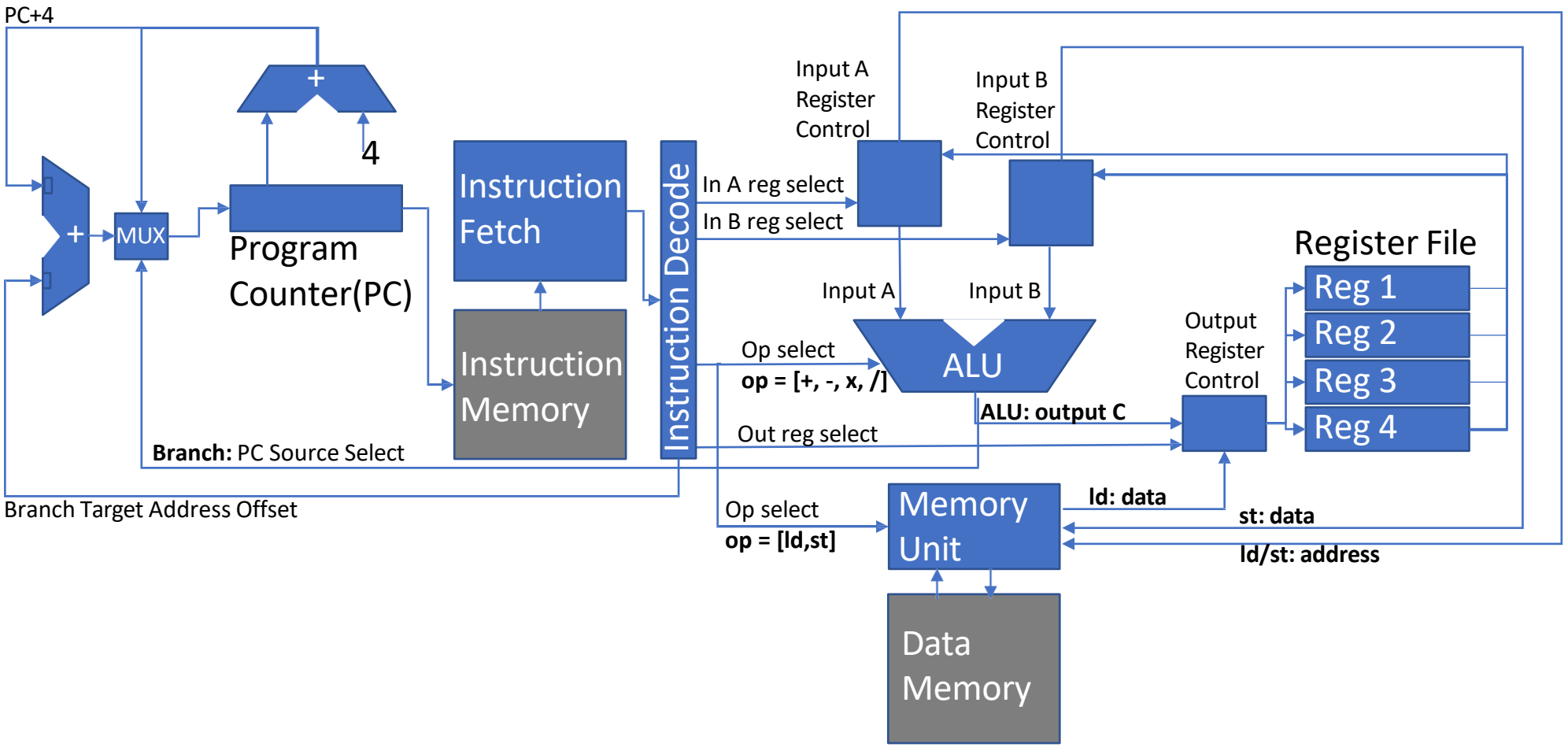
Implication of operation latencies?

- Single-cycle design means that the cycle time for the system is ***defined*** by the latency of the **longest-latency** operation

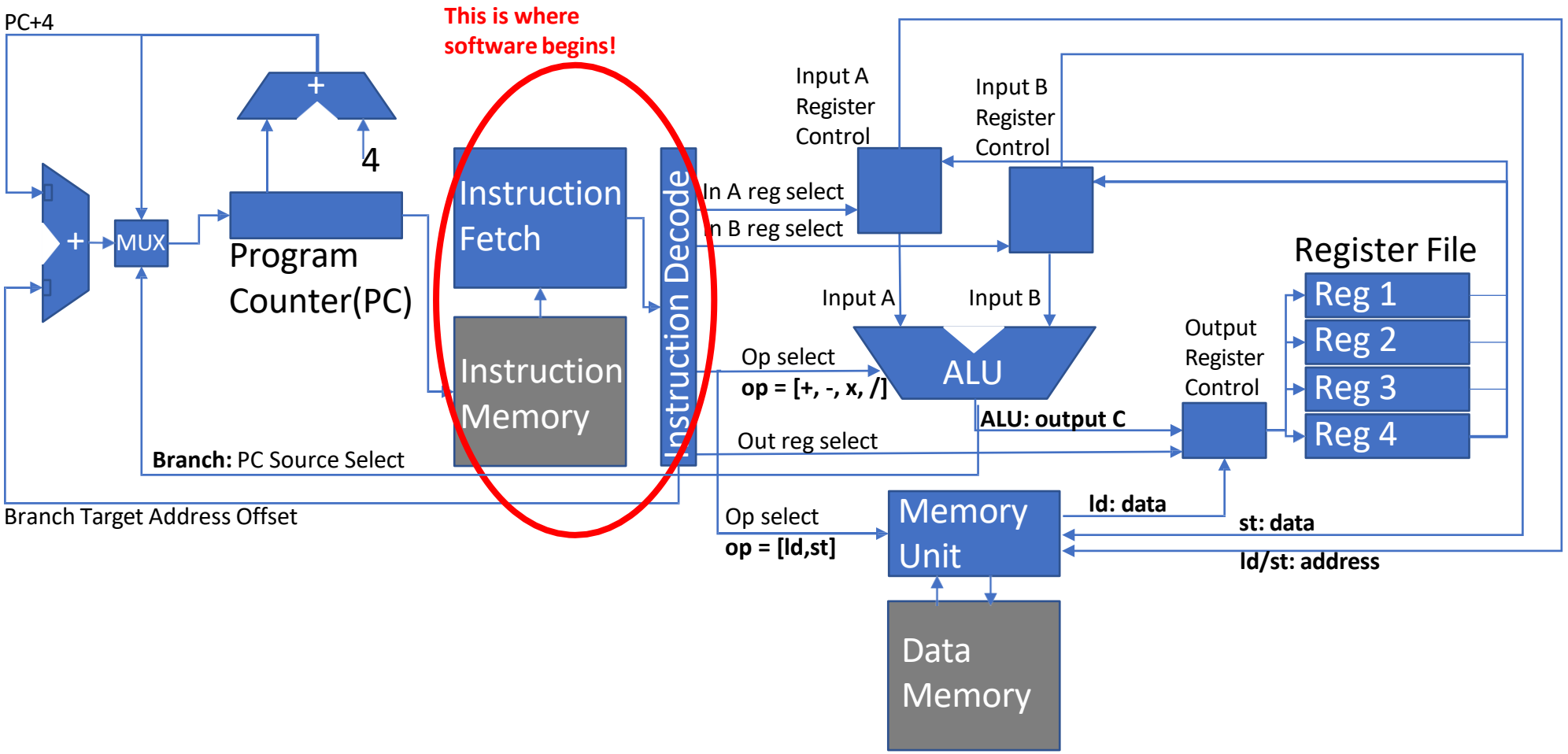
- In our case, that would be the memory latency (and ALU latency has some slack from the cycle time)

- ***If every operation is not a memory operation, then we have over-provisioned the cycle time of the system***

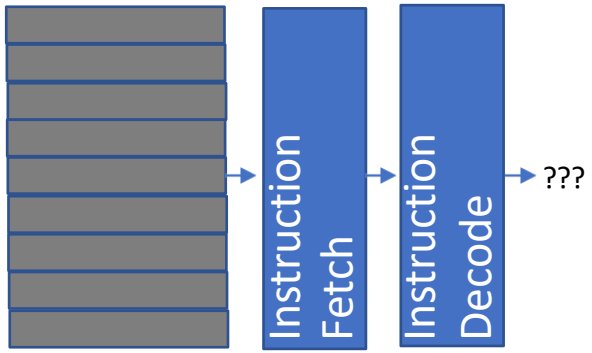
Where is the HW/SW Interface in the Datapath?



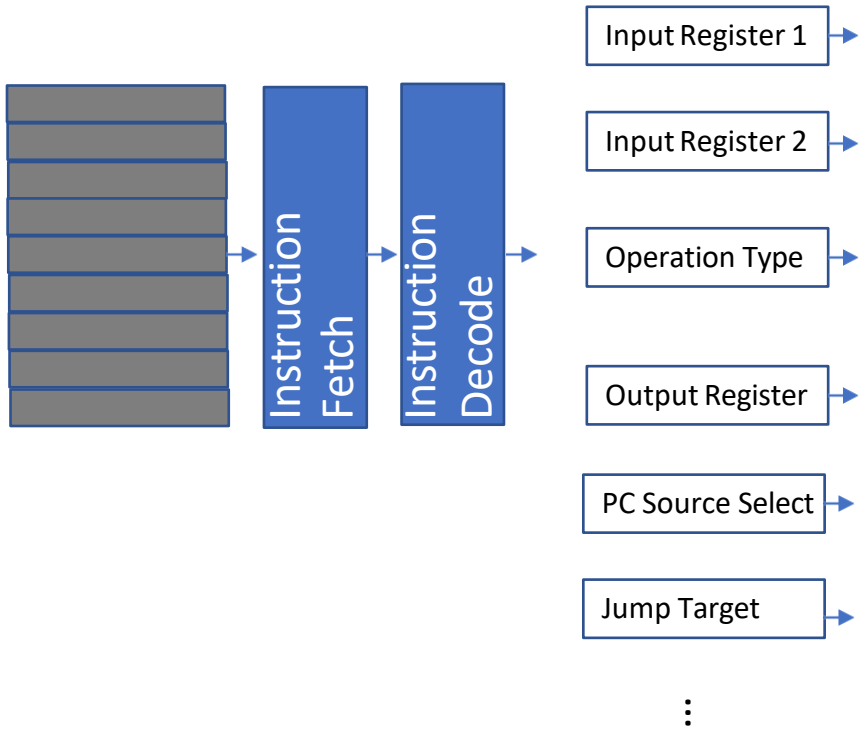
Where is the HW/SW Interface?



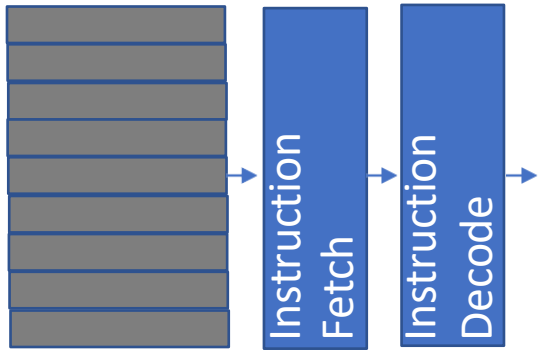
Instruction memory holds software



Big Idea: Instruction Bits are Control Signals

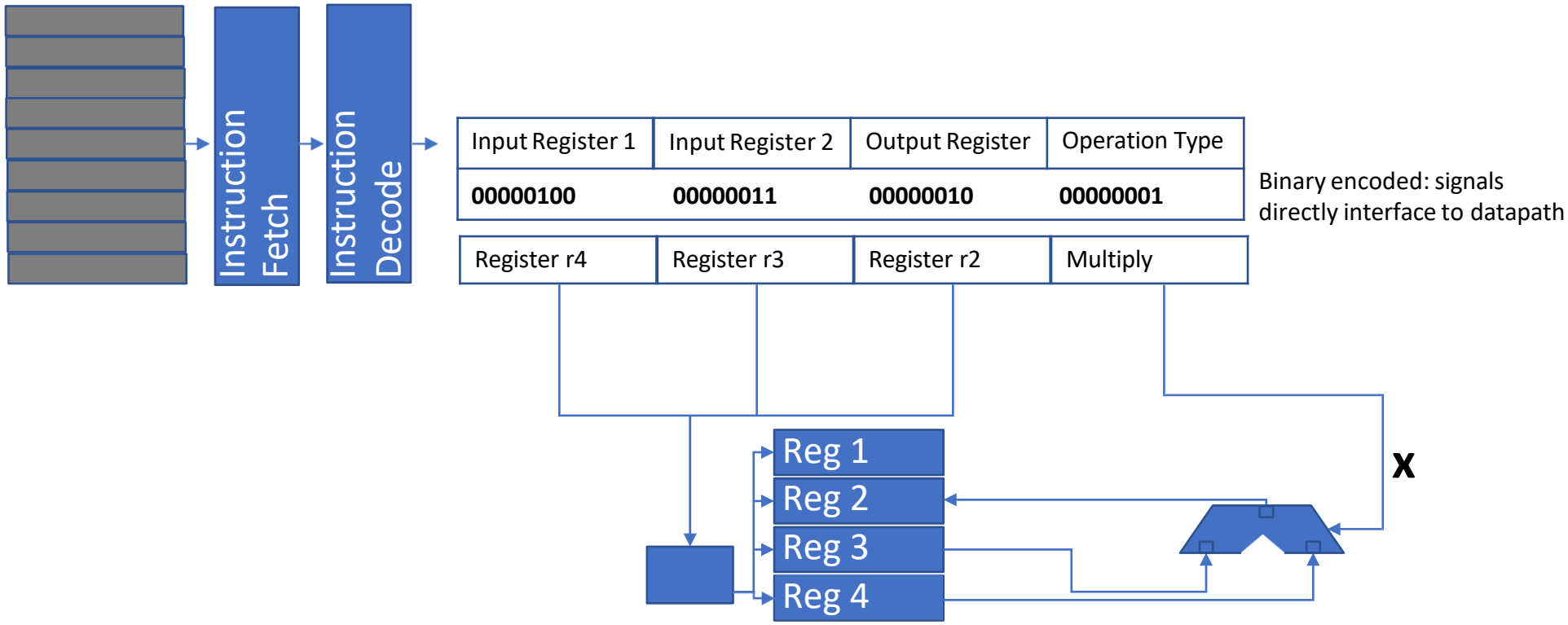


Big Idea: Instruction Bits are Control Signals



Input Register 1	Input Register 2	Output Register	Operation Type
00000100	**00001000**	**00001001**	**00000001**

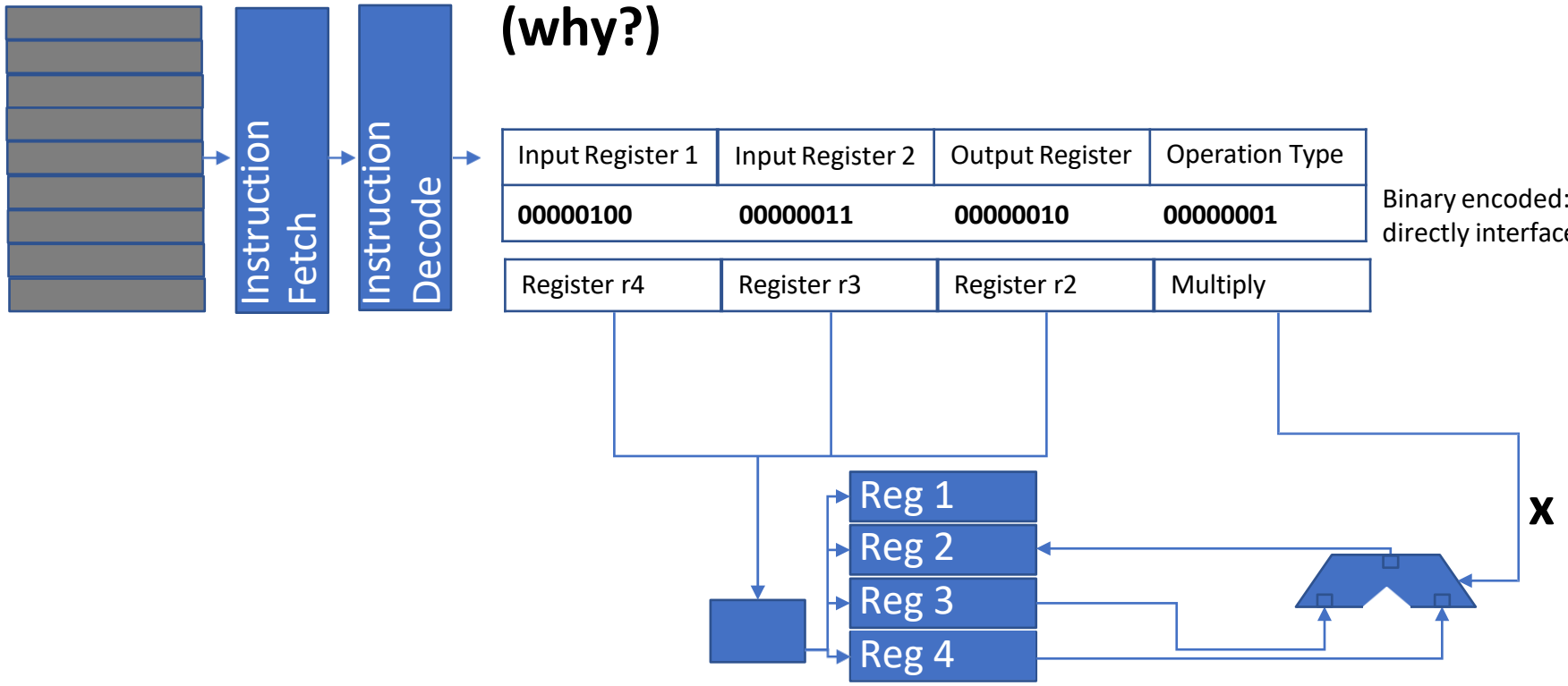
Big Idea: Instruction Bits are Control Signals



Instruction Set Architecture

The ISA defines the **architecture** of the machine

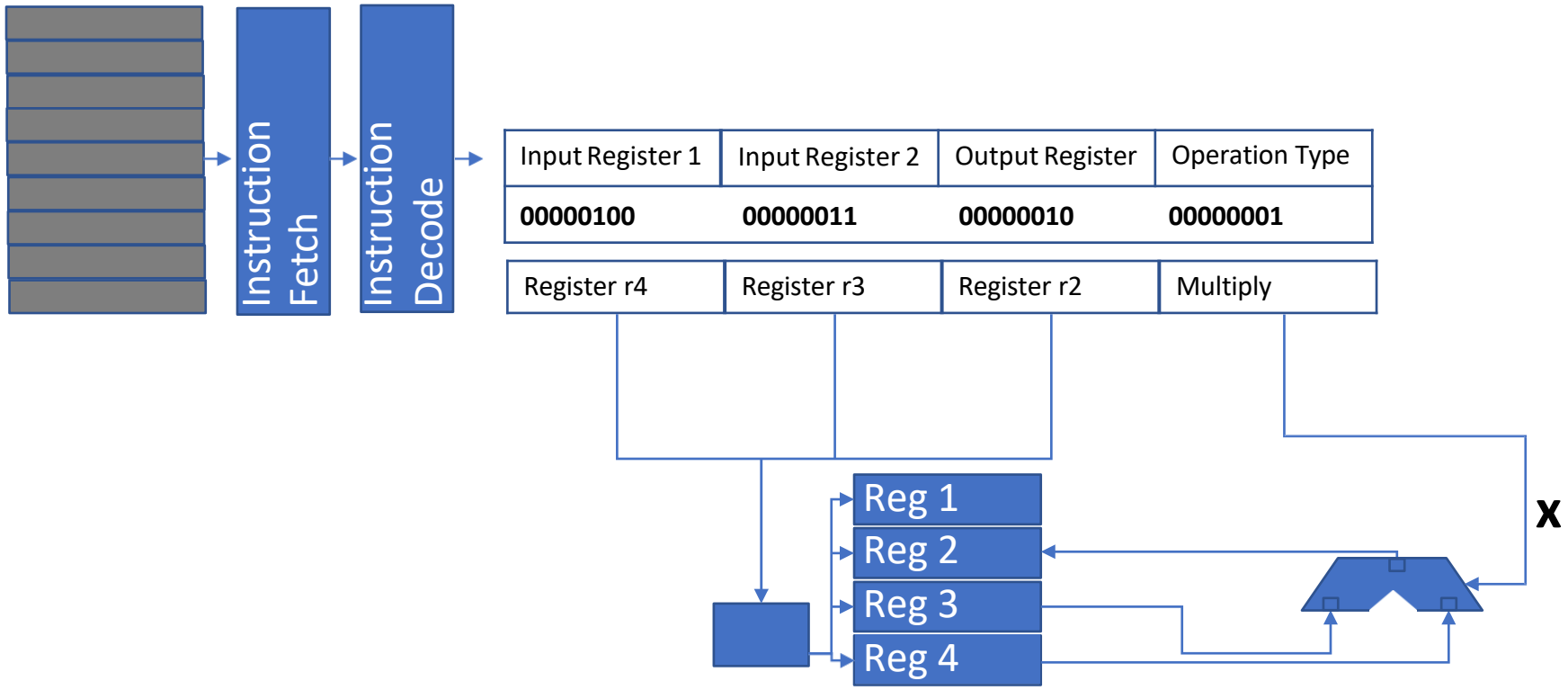
Any **implementation** of the architecture must support the features exposed through the ISA **(why?)**



Architecture vs. Microarchitecture

The ISA defines the **architecture** of the machine

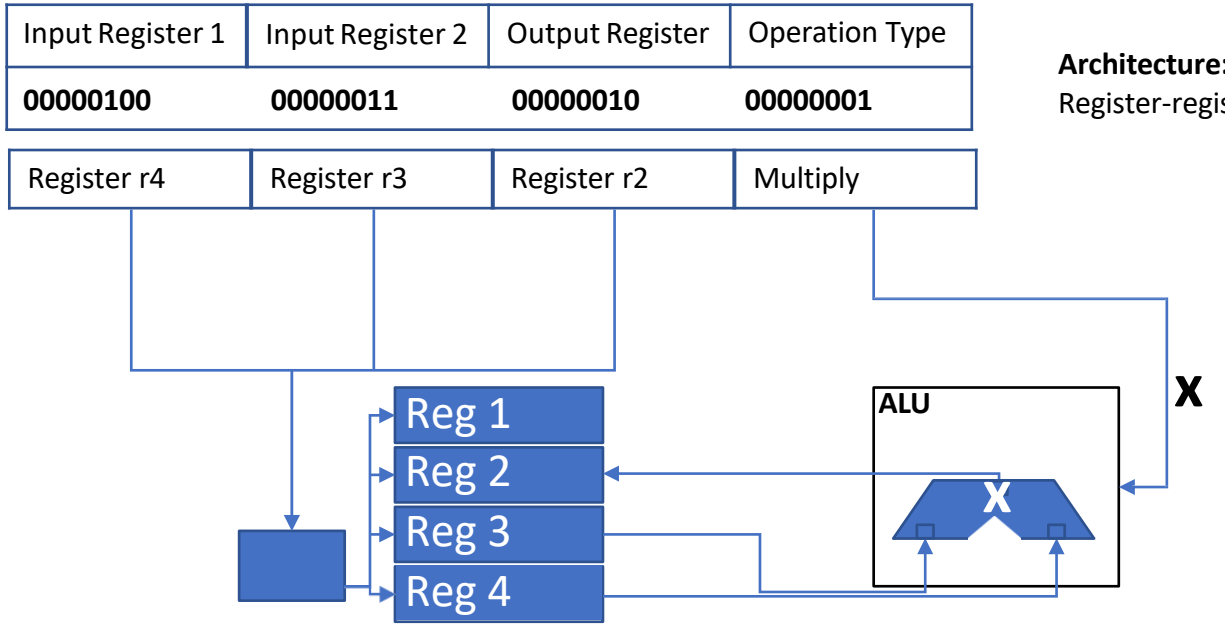
A **microarchitecture** implements the features of the architecture



Architecture vs. Microarchitecture

The ISA defines the **architecture** of the machine

A **microarchitecture** implements the features of the architecture

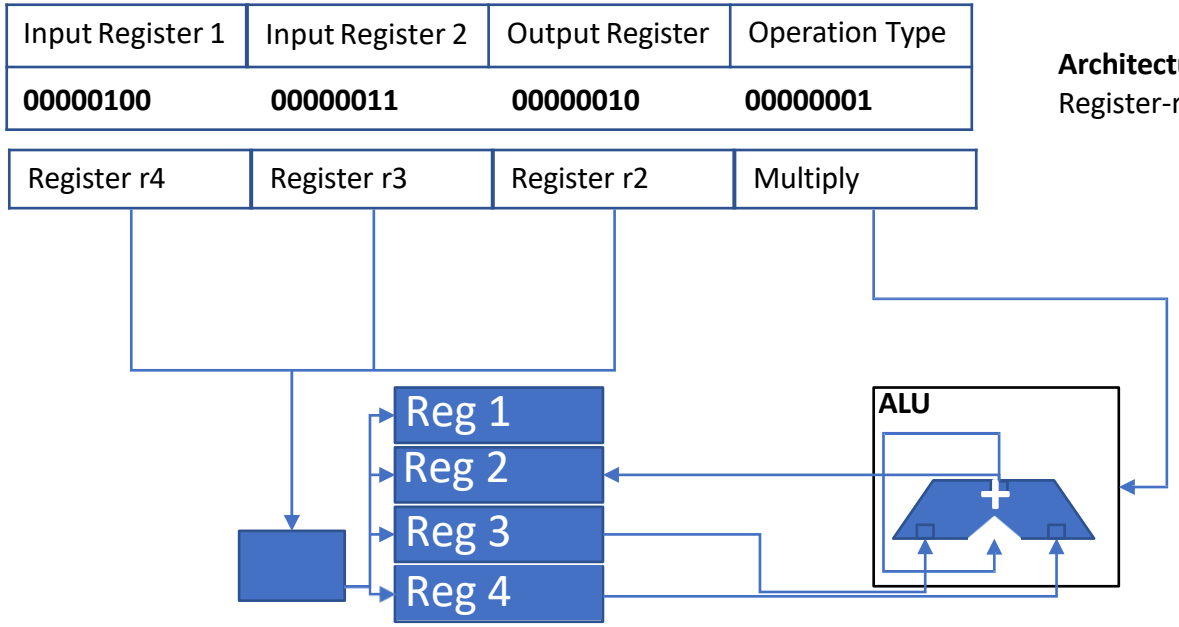


Architecture:
Register-register ALU ops, registers numbering 0-4

Microarchitecture:
One ALU containing a multiplier,
physical register file with registers numbering 0-3

Architecture vs. Microarchitecture

For a given architecture there are **many** perfectly good microarchitectural implementations



Architecture:
Register-register ALU ops, registers numbering 0-4

Microarchitecture:
One ALU containing an adder; multiply w/ iterated addition, physical register file with registers numbering 0-3

Architecture vs. Microarchitecture

For a given architecture there are **many** perfectly good microarchitectural implementations

Architecture:
Sequentially-numbered, general-purpose registers

Input Register 1	Input Register 2	Output Register	Operation Type
00000100	**00000011**	**00000010**	**00000001**

Register r4	Register r3	Register r2	Multiply

Register File

SRAM Bank #1
Reg 1
Reg 3

SRAM Bank #2
Reg 2
Reg 4

ALU

X

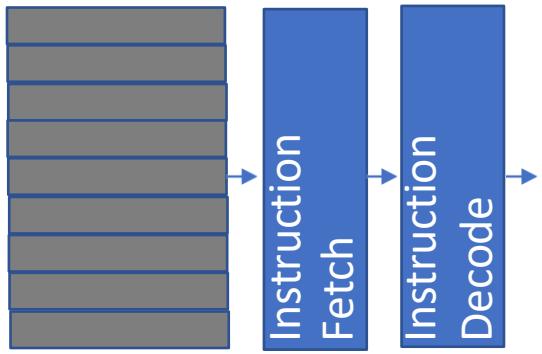
Microarchitecture:
Two SRAM banks storing regs based on parity

Architecture:
Register-register ALU ops, registers numbering 0-4

Microarchitecture:
One ALU containing an adder; multiply w/ iterated addition, physical register file with registers numbering 0-3

Instruction Set Architecture

The ISA is the **vocabulary** of the machine

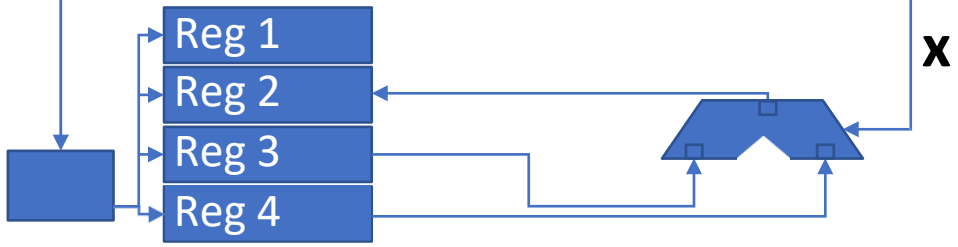


The ISA/vocabulary determines the types of programs/sentences that it is possible to write

Input Register 1	Input Register 2	Output Register	Operation Type
00000100	**00000011**	**00000010**	**00000001**

Binary encoded: signals directly interface to datapath

Register r4	Register r3	Register r2	Multiply



What should go in the ISA?

Reduced Instruction Set Computer

Simple primitives:
Let software compose complex operations

Register operands:
Decouple functionality from memory accesses

Few total operations:
Usually only one way to do something

Complex Instruction Set Computer

Simple & complex operations:
Hardware provides complex functionality

Many operations:
Often several ways to do the same thing

Register and memory operands:
Operations may directly manipulate memory

What should go in the ISA?

Reduced Instruction Set Computer

Simple primitives:
Let software compose complex operations

Register operands:
Decouple functionality from memory accesses

Few total operations:
Usually only one way to do something

Few cases to map to control signals in microarchitecture

```
rd = M[imm]
rd = M[reg]
rd = M[reg + imm]
rd = M[PC + imm]
```

Complex Instruction Set Computer

Simple & complex operations:
Hardware must support complex functionality

Many operations:
Often several ways to do the same thing

Register and memory operands:
Operations may directly manipulate memory

Many cases to map to control signals in microarchitecture

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Remember this from 18-213?

Plus all of these combinations

D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

What should go in the ISA?

Reduced Instruction Set Computer

Simple primitives:
Let software compose complex operations

Register operands:
Decouple functionality from memory accesses

Few total operations:
Usually only one way to do something

Complex Instruction Set Computer

Simple & complex operations:
Hardware must support complex functionality

Register and memory operands:
Operations may directly manipulate memory

Many operations:
Often several ways to do the same thing

What are the pros and cons of each?

How does RISC vs. CISC affect the microarchitecture, compiler, program, programmer?

Principles of ISA Design

General Principles

Regularity – "Law of least astonishment"
Orthogonality – keep separable concerns separate
Composability – regular, orthogonal ops combine easily

Specific Principles

One vs. All – precisely one way to do it, or all ways should be possible
Primitives, not solutions – solve by coding, compiling, & synthesizing

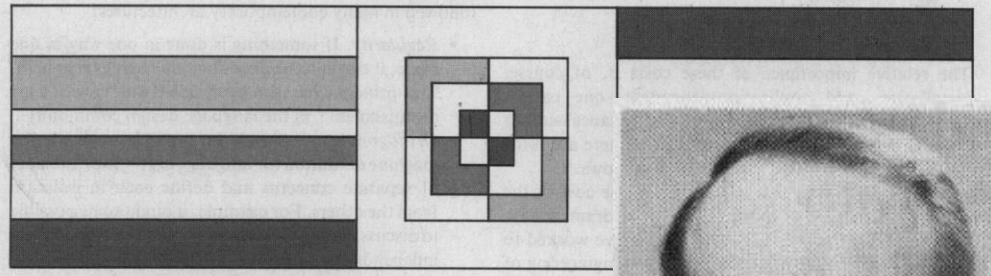
"Blatant opinions" (matters of taste)

Addressing – not limited to simple arrays, etc.
Environment Support – exceptions, processes, debugging, etc
Deviations – deviate from these rules only in implementation-specific ways

An examination of the relation between architecture and compiler design leads to several principles which can simplify compilers and improve the object code they produce.

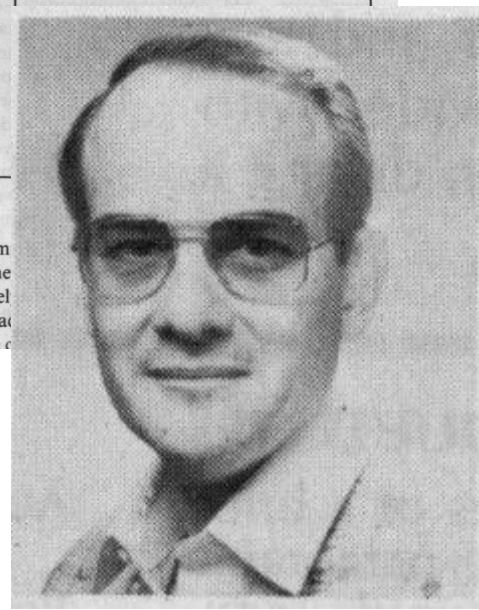
Compilers and Computer Architecture

William A. Wulf
Carnegie-Mellon University



The interactions between the design of a computer's instruction set and the design of compilers that generate code for that computer have serious implications for overall computational cost and efficiency. This article, which investigates those interactions, should ideally be

simplify com
programs the
are absolutel
ever, they lea
people have



Designing irregular structures at the chip level is *very* expensive.

Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.

What did we just learn?

- Computer architectures define the HW/SW interface through the ISA
- There is a difference between architecture and microarchitecture
- Many valid microarch. implementations of an architecture exist
- RISC vs. CISC architectures are extrema on a spectrum
- Principles of ISA design (Wulf)

What to think about next?

- The basics of the RISCV-RV32I ISA and some other hw/sw interfaces
- More microarchitectural concepts
  - Pipelining our microarchitecture & instruction-level parallelism
  - Control hazards & branch prediction