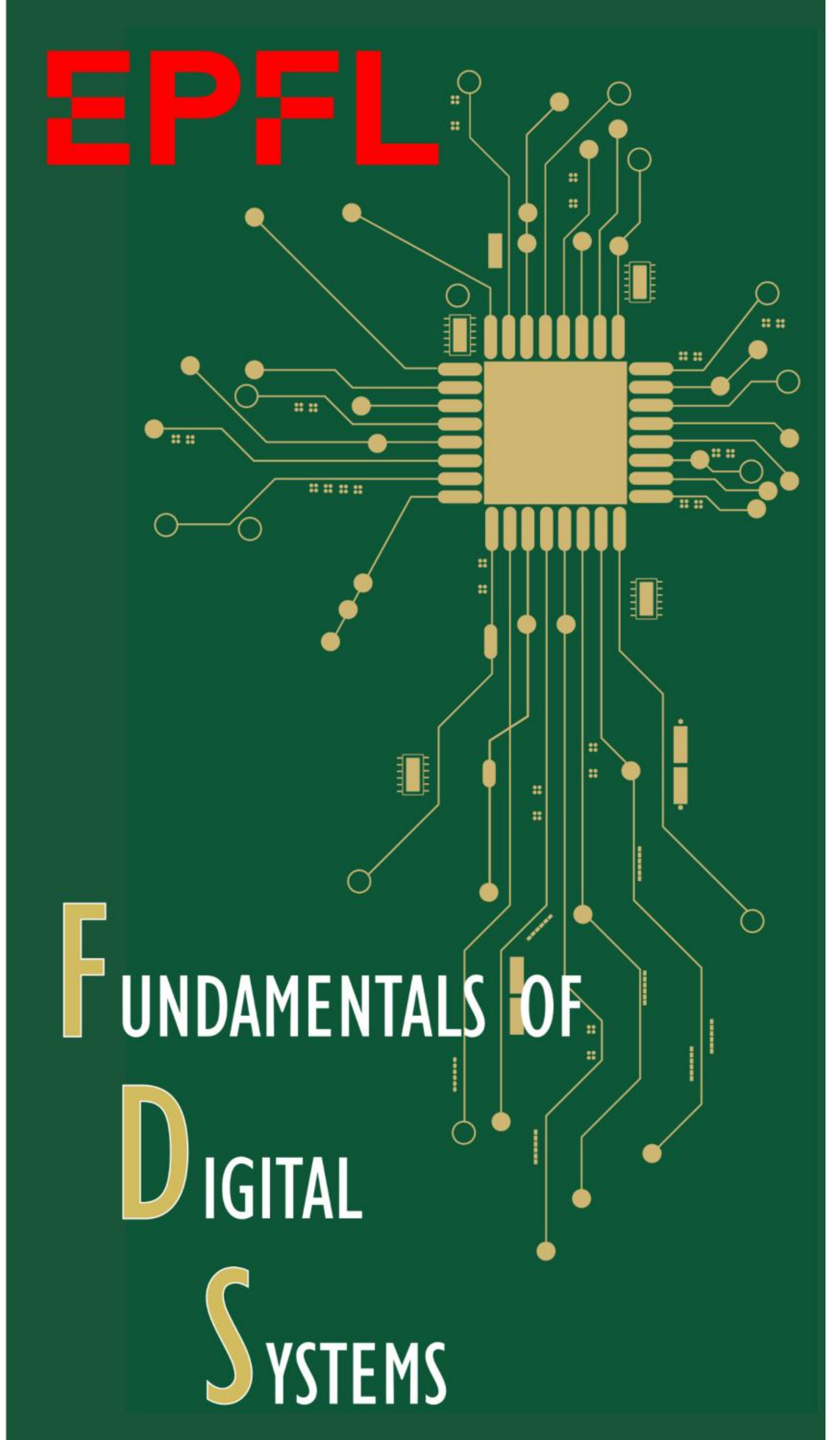


Computer Architecture

Introduction to Processors

CS-173 Fundamentals of Digital Systems

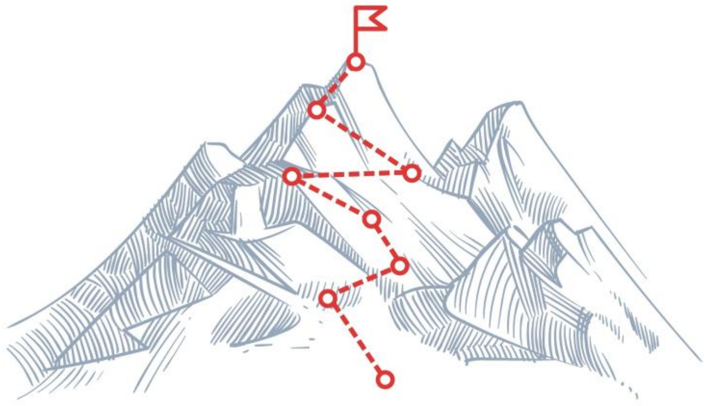
Mirjana Stojilović

Spring 2025

Previously on FDS

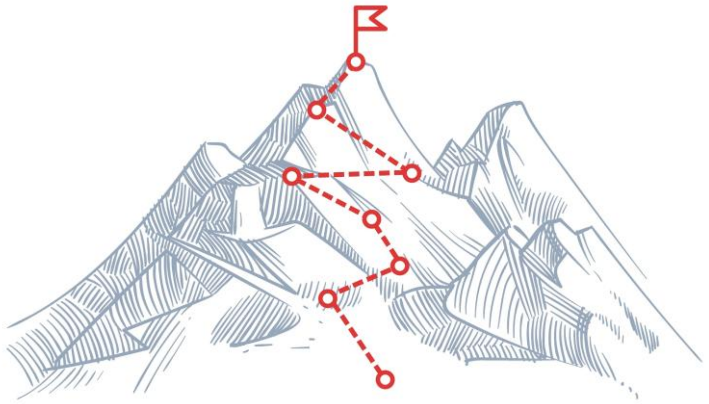
Designing Digital Systems

Previously



- Designing complete digital systems
- **Buses** in digital systems
  - With tri-state drivers or multiplexers
  - **Swapping registers** example
- Verilog **loops** and **generate** construct for **instantiating** modules
  - **Ripple-carry adder** example
- Verilog: **reduction** operators and **generate** constructs

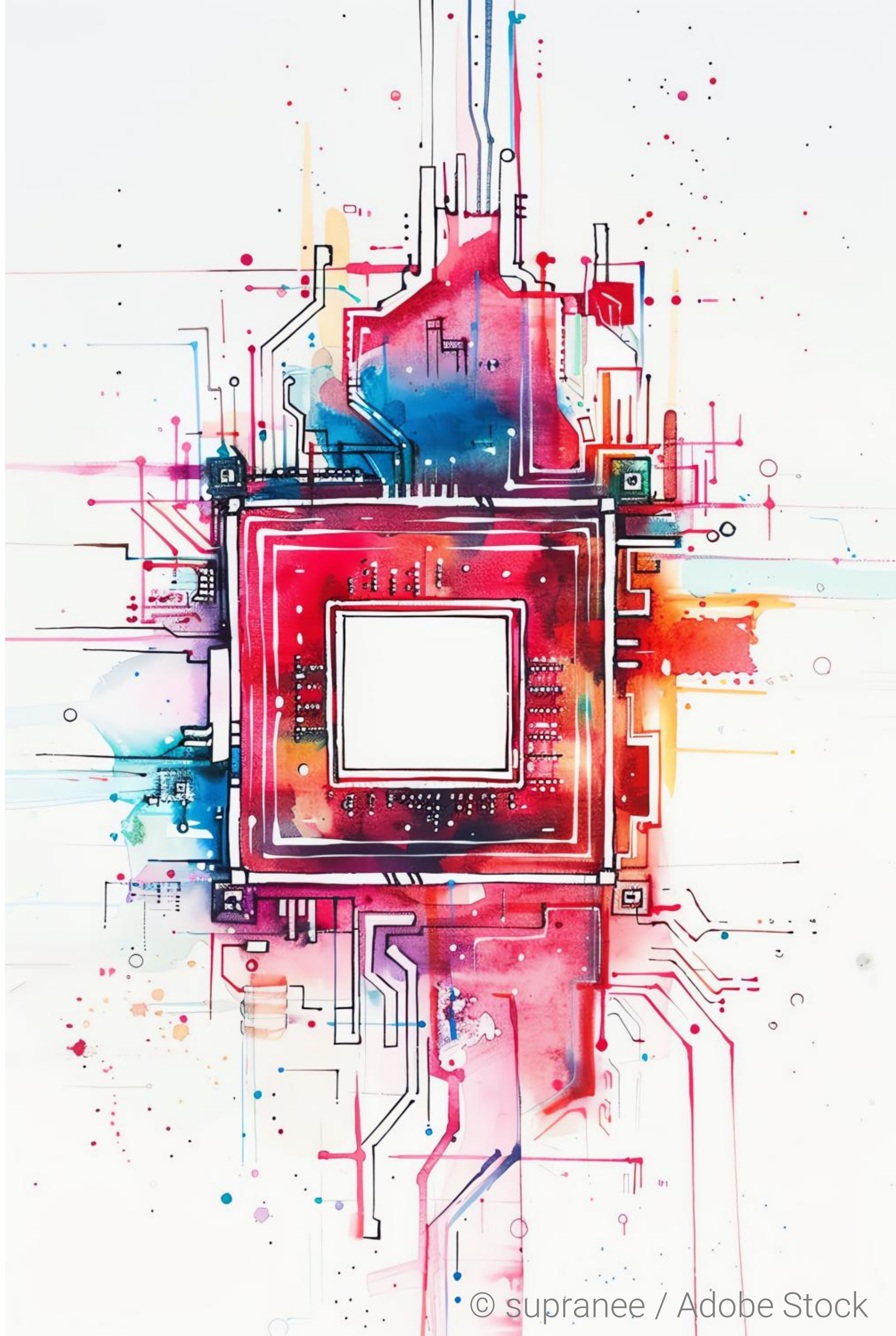
What Have We Learnt So Far?



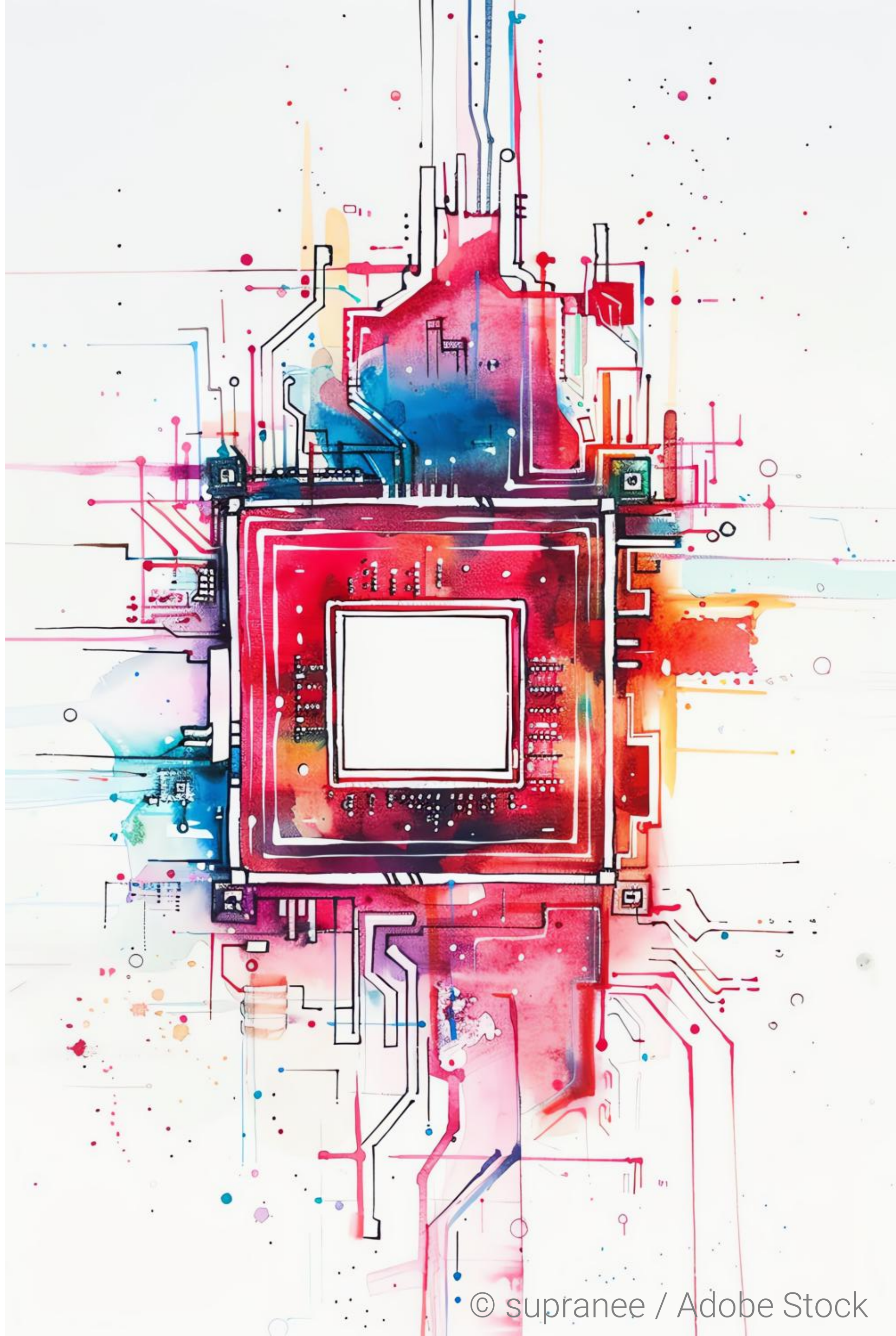
- Important classes of digital building blocks
  - **Sequential** building blocks: flip-flops, registers, counters, …
  - **Combinational** logic components: gates, multiplexers, …
  - **Memories**: devices that store information
  - **Controllers:** finite state machines
- Verilog: A language for describing and modelling digital circuits
- **With that knowledge, we can design an entire simple processor**

Let’s Talk About…

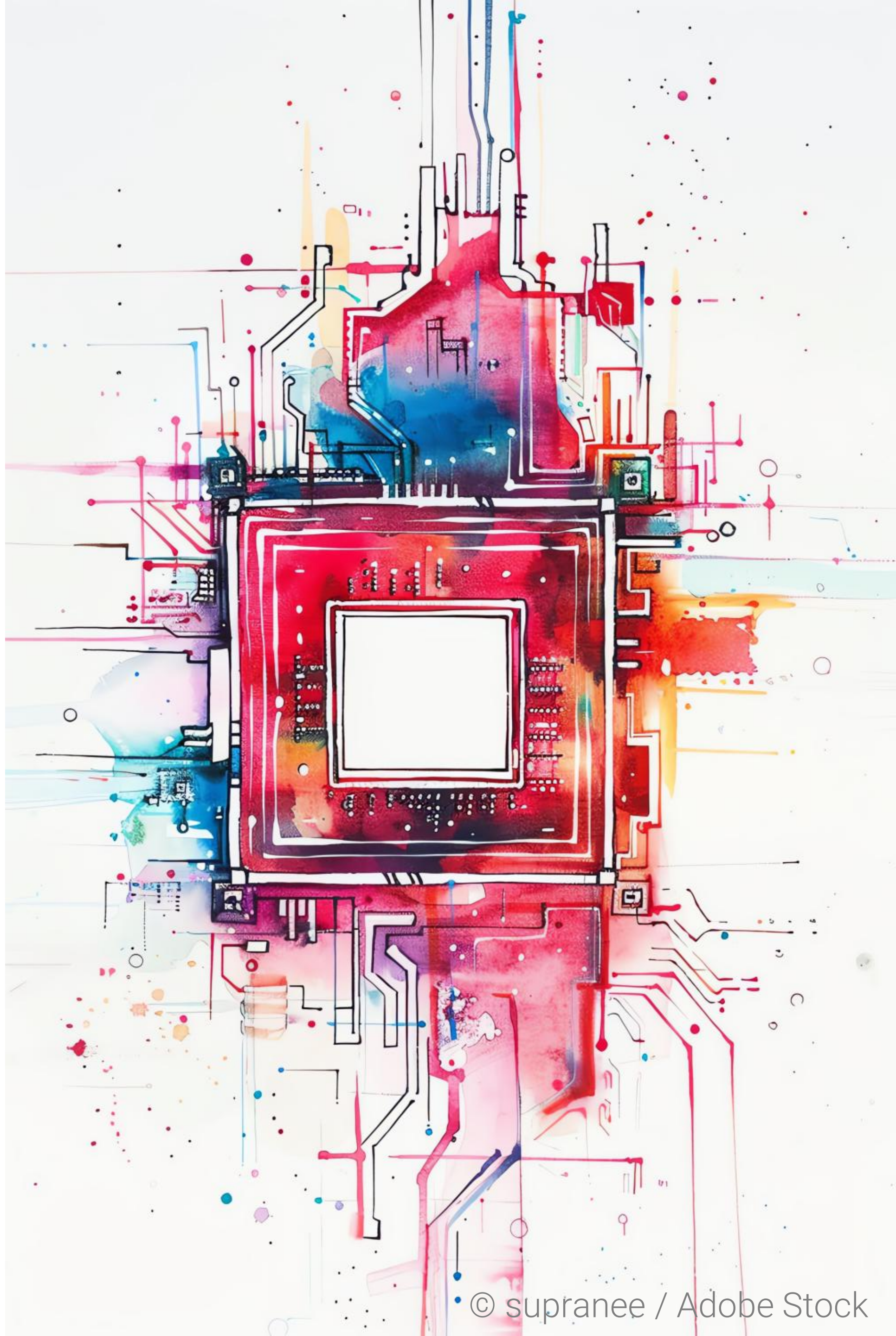
…Designing a Simple Processor



Quick Outline

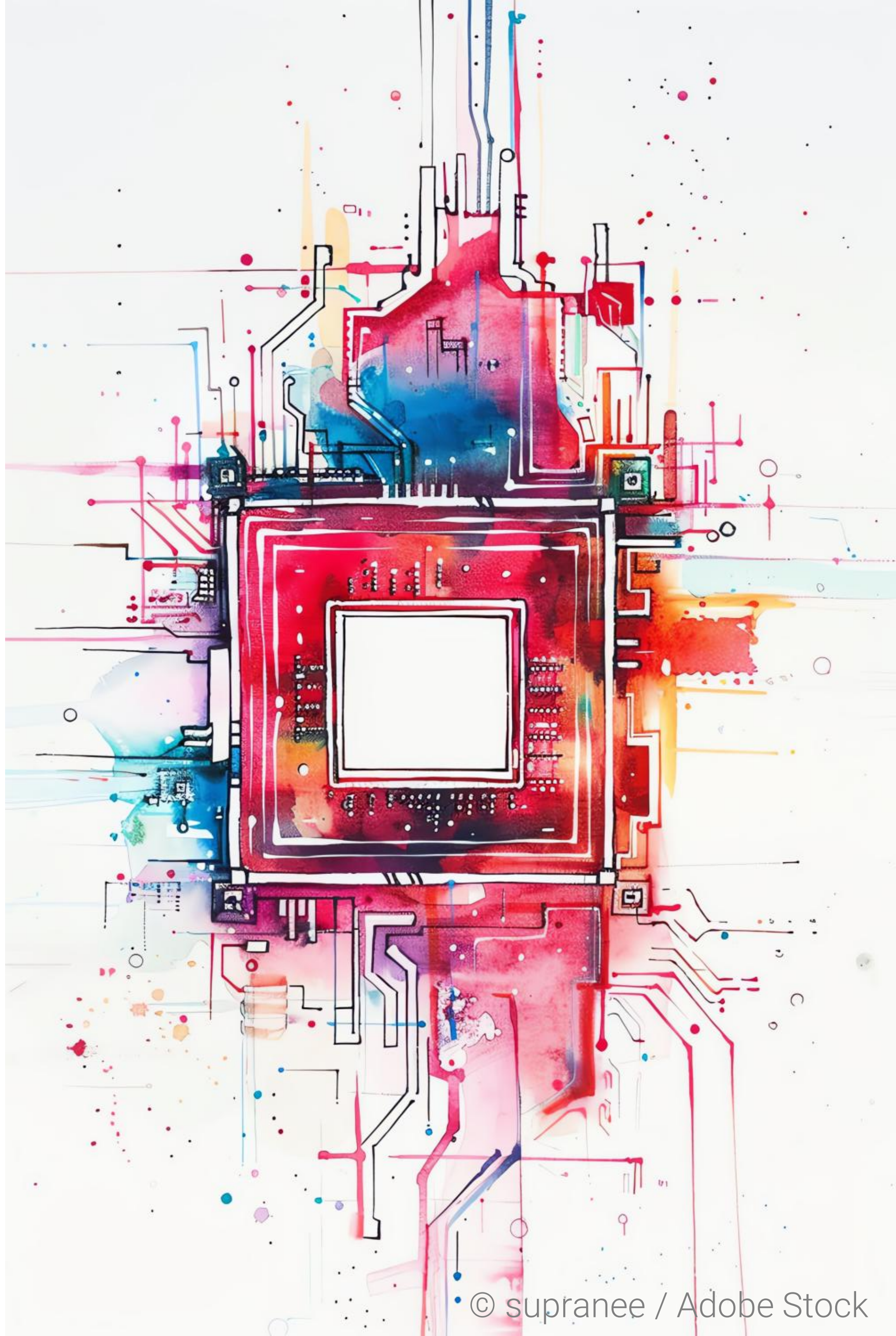


A Processor

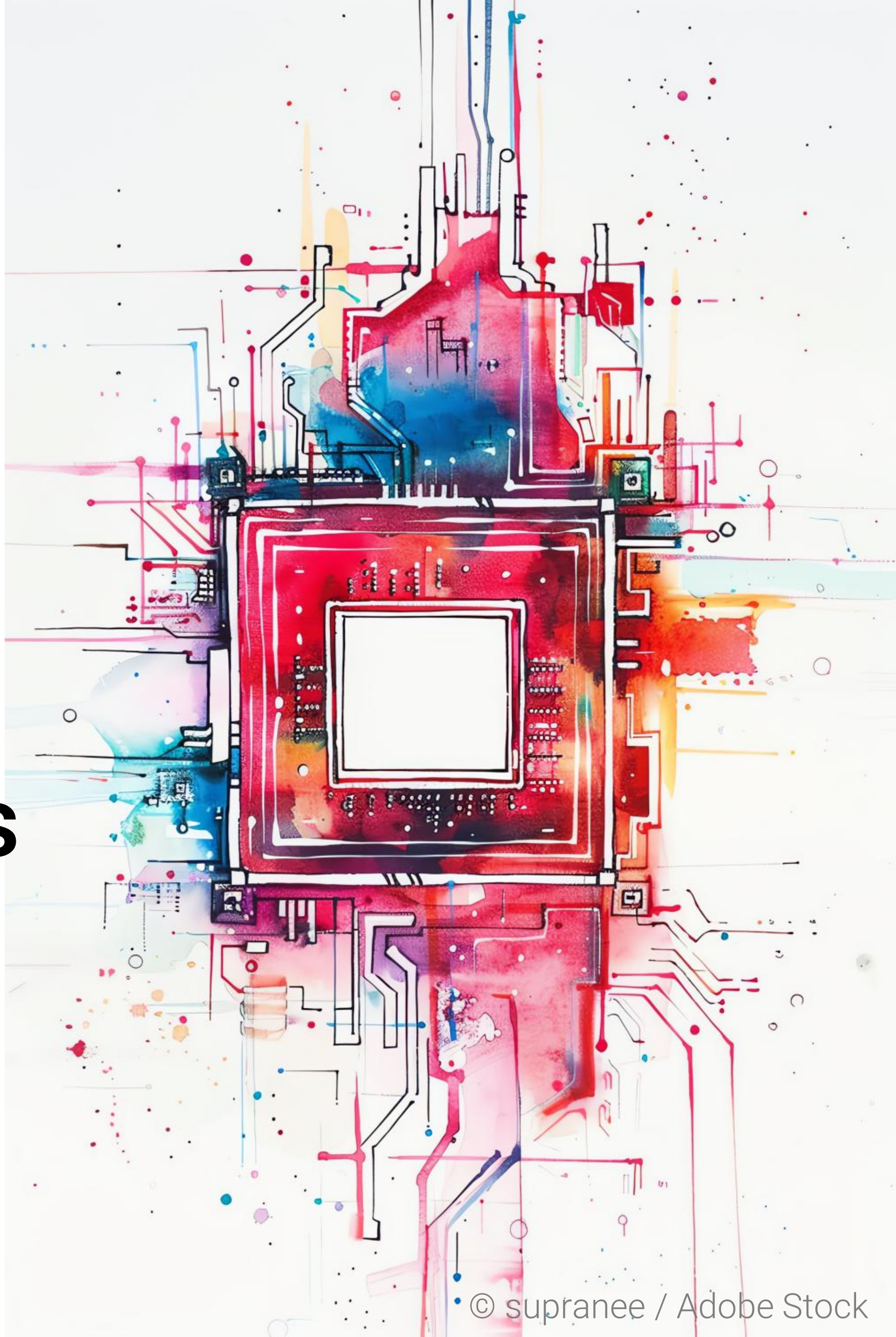


A Processor

- A processor, also referred to as a **Central Processing Unit (CPU),** is the central component in any general-purpose computing system
  - E.g., phones, laptops, tablets, servers, …
- CPU is responsible for executing software applications and facilitating data processing
- CPU orchestrates the data manipulations according to the instructions provided by software programs



From Programs to Computers



A Simple Computer Program

C Programming Language

```
// variable initialization
int data   = 0x00123456;  // hexadecimal
int result = 0;
int mask   = 1;
int count  = 0;
int temp   = 0;
int limit  = 32;
do {                      // loop
  temp   = data & mask;   // bitwise and
  result = result + temp; // addition
  data   = data >> 1;     // shift right
  count  = count + 1;     // addition
} while (count != limit); // condition
```

- Consider this piece of code
- **Q:** What does it do?
- **A:** It counts the number of ones in the 32-bit integer `data` and stores the result in the variable `result`
  - `result = 9`

A Simple Computer Program

C Programming Language

- Solution; variable updates through loop iterations

Step	data	temp	result
Initialization	0000 0000 0001 0010 0011 0100 0101 0110	0	0
1st Loop iteration	0000 0000 0001 0010 0011 0100 0101 0110	0	0
2nd Loop iteration	0000 0000 0000 1001 0001 1010 0010 1011	1	1
3rd Loop iteration	0000 0000 0000 0100 1000 1101 0001 0101	1	2
4th Loop iteration	0000 0000 0000 0010 0100 0110 1000 1010	0	2
5th Loop iteration	0000 0000 0000 0001 0010 0011 0100 0101	1	3
…	…	…	…
31st Loop iteration	0000 0000 0000 0000 0000 0000 0000 0000	0	9
32nd Loop iteration	0000 0000 0000 0000 0000 0000 0000 0000	0	9

The number of ones in the 32-bit variable **`data`**

A Simple Computer Program

Low level programming, C language

```
// variable initialization
int data   = 0x00123456;
int result = 0;
int mask   = 1;
int count  = 0;
int temp   = 0;
int limit  = 32;
do {
  temp   = data & mask;
  result = result + temp;
  data   = data >> 1;
  count  = count + 1;
} while (count != limit);
```

- **Q:** Identify lines that concern data
  - Read, write/update, compute
- **A:**
  - variable initializations
  - operations: bitwise, arithmetic, logic, relational, shift, etc.
  - assignments

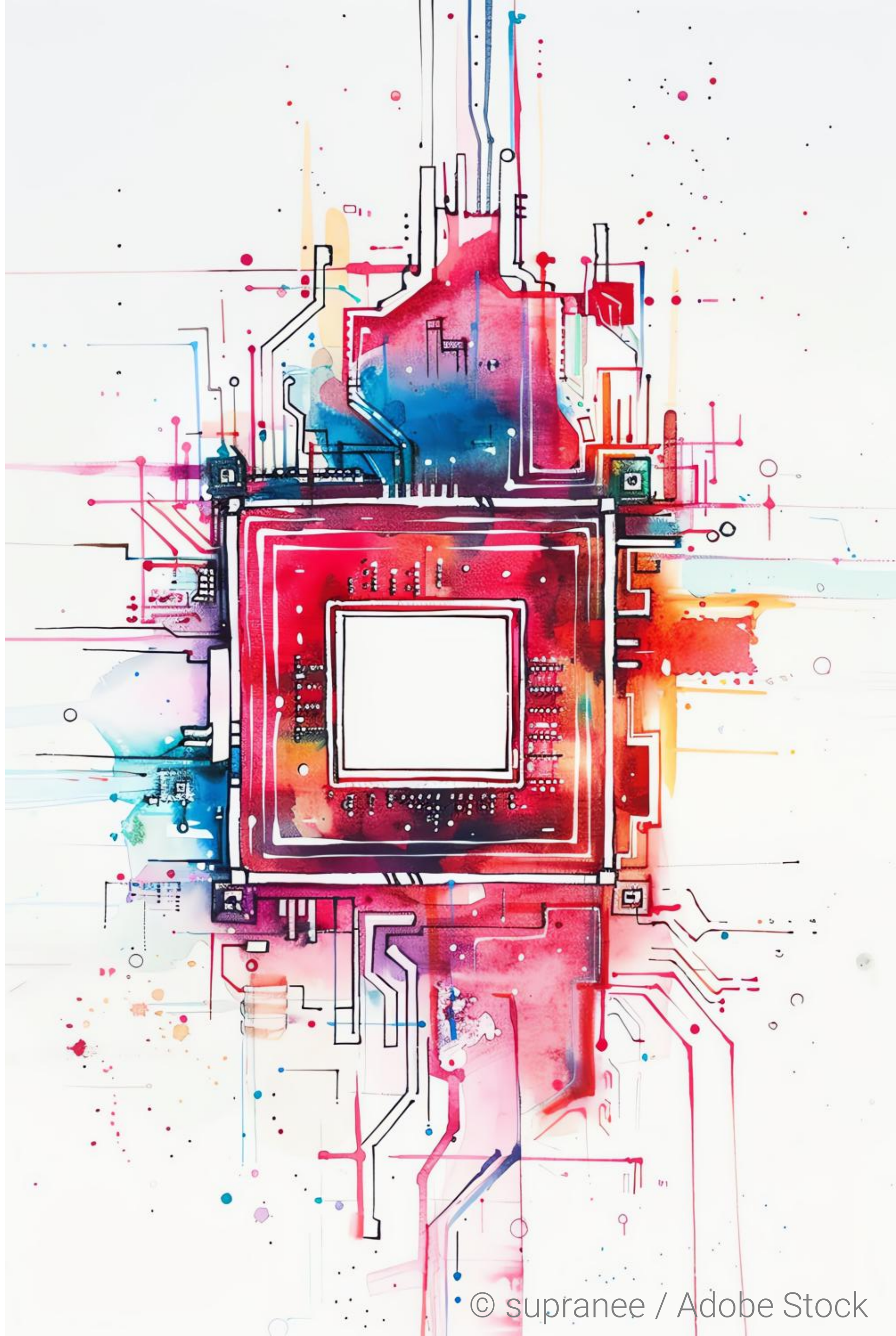
A Simple Computer Program

Low level programming, C language

```
// variable initialization
int data   = 0x00123456;
int result = 0;
int mask   = 1;
int count  = 0;
int temp   = 0;
int limit  = 32;
do {
  temp   = data & mask;
  result = result + temp;
  data   = data >> 1;
  count  = count + 1;
} while (count != limit);
```

- **Q:** Identify lines that control the execution of the program and the flow of data
  - if-else, switch, loop, etc.
- **A:** do-while loop
- There exists also some hidden control that makes the program execute sequentially (one line after another), loop, skip some lines, or return to an earlier line

Hardware-Friendly Programs



Machine Language
Binary

- Hardware does not *speak* C, Java, Python, …; it *speaks* ‘0’ and ‘1’
- Machine language is the lowest level of programming language
- CPU code consists of machine language **instructions**
  - Instructions **instruct** the processor to execute a specific task
  - Instructions are **patterns of bits** that correspond to CPU commands

Translating High-Level Code Into Binary

- **Source code**
  - The process begins with a high-level source code written in a programming language
- **Compilation or interpretation**
  - The source code is processed by a compiler or interpreter
  - Compiler translates the code in intermediate formats, specific to the target platform
  - Interpreter executes the source code line by line, translating them into machine code (binary) and executing them immediately
- **Optimization**
  - Compiler may optimize the code to improve performance or reduce its size

Translating High-Level Code Into Binary

Contd.

- **Linking** (for compiled languages)
  - Combining intermediate formats with external libraries and functions, if needed
- **[Optional] Assembly code generation**
  - Human-readable representation of machine code
- **Binary code generation**
  - Sequences of 0s and 1s representing machine instructions

Note: Assembly and binary code will be in our focus…

From High-Level Programs to Assembly
Algorithm

Let us now translate our simple program to something hardware can be made to understand…

- **Step 1:** Add line numbers and labels
- **Step 2:** Assign variables to registers
- **Step 3:** Replace each code line with a corresponding machine language instruction

Step 1: Line Numbering and Labeling

- Number lines of code
  - Do not number lines that have no effect on the program execution
- Label what may be important lines of code
  - E.g., loop body
- Use appropriate symbols for comments

```
                # variable initialization
 0              int data   = 0x00123456;
 1              int result = 0;
 2              int mask   = 1;
 3              int count  = 0;
 4              int temp   = 0;
 5              int limit  = 32;
                do {
 6   loop:        temp   = data & mask;
 7                result = result + temp;
 8                data   = data >> 1;
 9                count  = count + 1;
10              } while (count != limit);
```

Note: Program conversion in progress…

Step 2: Assign Variables to Registers

- Variables are data
  - Variables are read
  - Variables are updated (overwritten)
- Registers store data
  - Registers are read
  - Registers are updated (overwritten)

Program variables reside in registers

Step 2: Assign Variables to Registers
Contd.

- Assuming registers are named `x0`, `x1`, `x2`, etc., and each register stores 32 bits, let us assign variables as follows
  - `data:   x1`
  - `result: x2`
  - `mask:   x3`
  - `count:  x4`
  - `temp:   x5`
  - `limit:  x6`

```
                # variable initialization
 0              x1   = 0x00123456; # data
 1              x2   = 0;          # result
 2              x3   = 1;          # mask
 3              x4   = 0;          # count
 4              x5   = 0;          # temp
 5              x6   = 32;         # limit
                do {
 6      loop:     x5   = x1 & x3;
 7                x2   = x2 + x5;
 8                x1   = x1 >> 1;
 9                x4   = x4 + 1;
10              } while (x4 != x6);
```

Note: Program conversion in progress…

Step 3: Instructions
In Assembly

- Let us rewrite every line using commands/instructions in the format below

Operation name	Destination,	Left operand,	Right operand

- Destination and operands are variables (registers)
- Let us give the operations some simple **codenames**:
  - **li:** **l**oad a literal (an **i**mmediate) into a variable
  - **and:** bitwise and of two variables
  - **add:** addition of two variables
  - **addi:** addition of a variable and a literal (an **i**mmediate, a value)
  - **srli:** **s**hift **r**ight **l**ogical by a literal (an **i**mmediate)
  - **bne:** if two variables are **n**ot **e**qual, go to the line with the label (**b**ranch)

Step 3: Instructions

In Assembly, Contd.

Operation name | Destination, | Left operand, | Right operand

```
 0         li   x1, 0x00123456
 1         li   x2, 0
 2         li   x3, 1
 3         li   x4, 0
 4         li   x5, 0
 5         li   x6, 32
                               # do {
 6  loop:  and  x5, x1, x3      # x5    = x1 & x3
 7         add  x2, x2, x5      # x2    = x2 + x5
 8         srli x1, x1, 1       # x1    = x1 >> 1
 9         addi x4, x4, 1       # x4    = x4 + 1
10         bne  x4, x6, loop    # while (x4 != limit)
```

Note: Program conversion in progress…

Et Voila !

- Our first program in assembly
- Although it seems less readable (to humans), it is way more suitable for hardware implementation
  - Operations are encoded in a simple and regular manner
  - Translation to binary is trivial
  - Variables (data) live in registers

Let us now design a digital circuit able to interpret and execute this program: the CPU …

```
      li   x1, 0x00123456
      li   x2, 0
      li   x3, 1
      li   x4, 0
      li   x5, 0
      li   x6, 32

loop: and  x5, x1, x3
      add  x2, x2, x5
      srli x1, x1, 1
      addi x4, x4, 1
      bne  x4, x6, loop
```

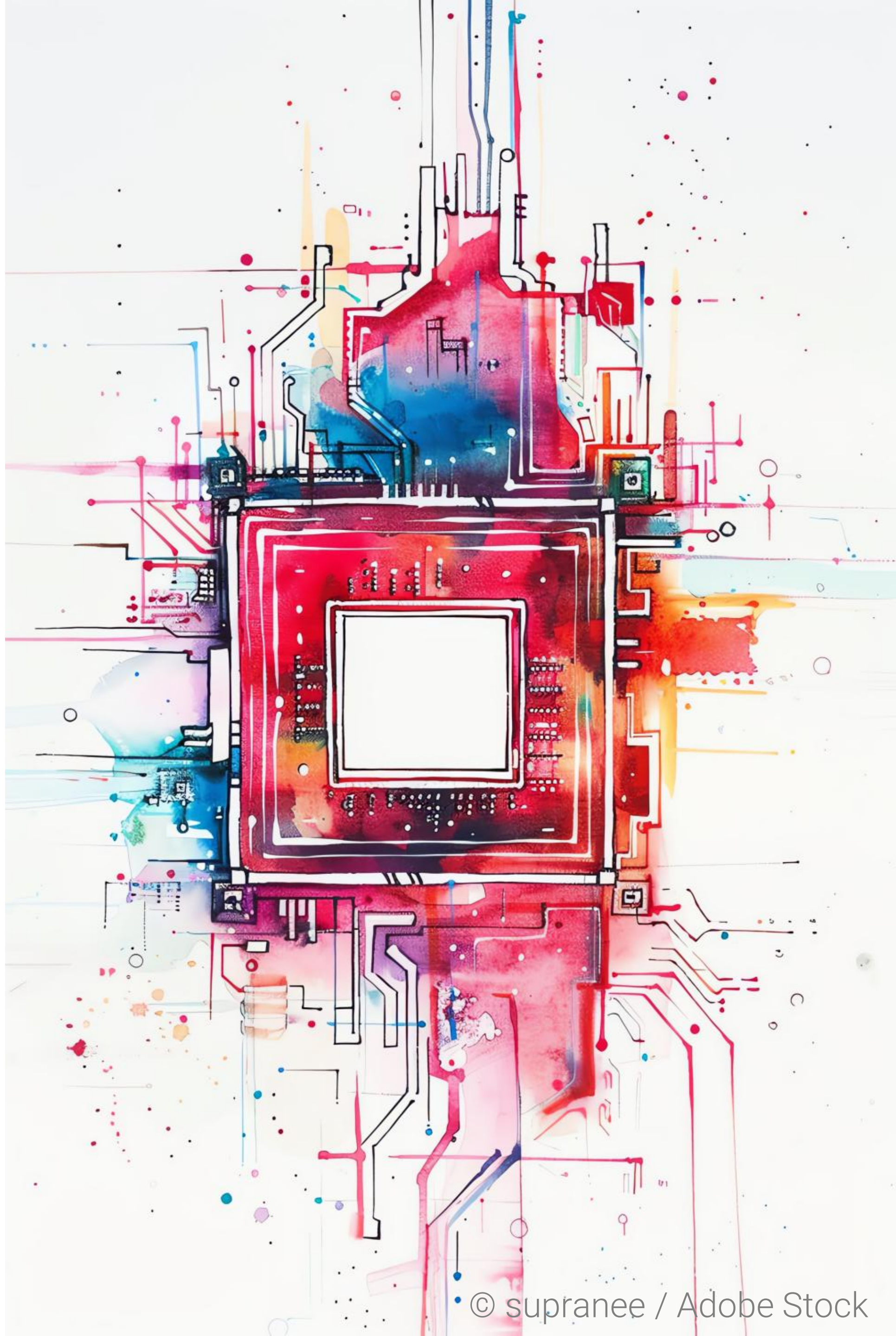
Note: Program conversion completed

Why Design a Processor?

- We know how to design a fast digital logic circuit that counts the number of ones in a 32-bit binary number
  - From truth tables to efficient implementation with logic gates or
  - With shift registers and counters, as a more algorithmic approach
- **Q: Why** design a processor now?
- **A:** Processors can do much more than counting the number of ones. When the sequence of instructions changes, the processor’s work also changes. Yet, this versatility comes with a runtime penalty, as specialized solutions always outperform the general-purpose ones

Under The Hood

Inside Processors



Two Parts of a Processor

- **Part I: Data path**
  - Concerns everything related to data
    - Variable storage/read/write
    - Operations on variables
    - Includes the digital logic circuits that perform operations on data or hold data
- **Part II: Control path**
  - Concerns everything related to the code execution
    - Instruction reading/decoding/sequencing
    - Manages the digital circuits of the datapath so they perform the operations as specified by the instructions

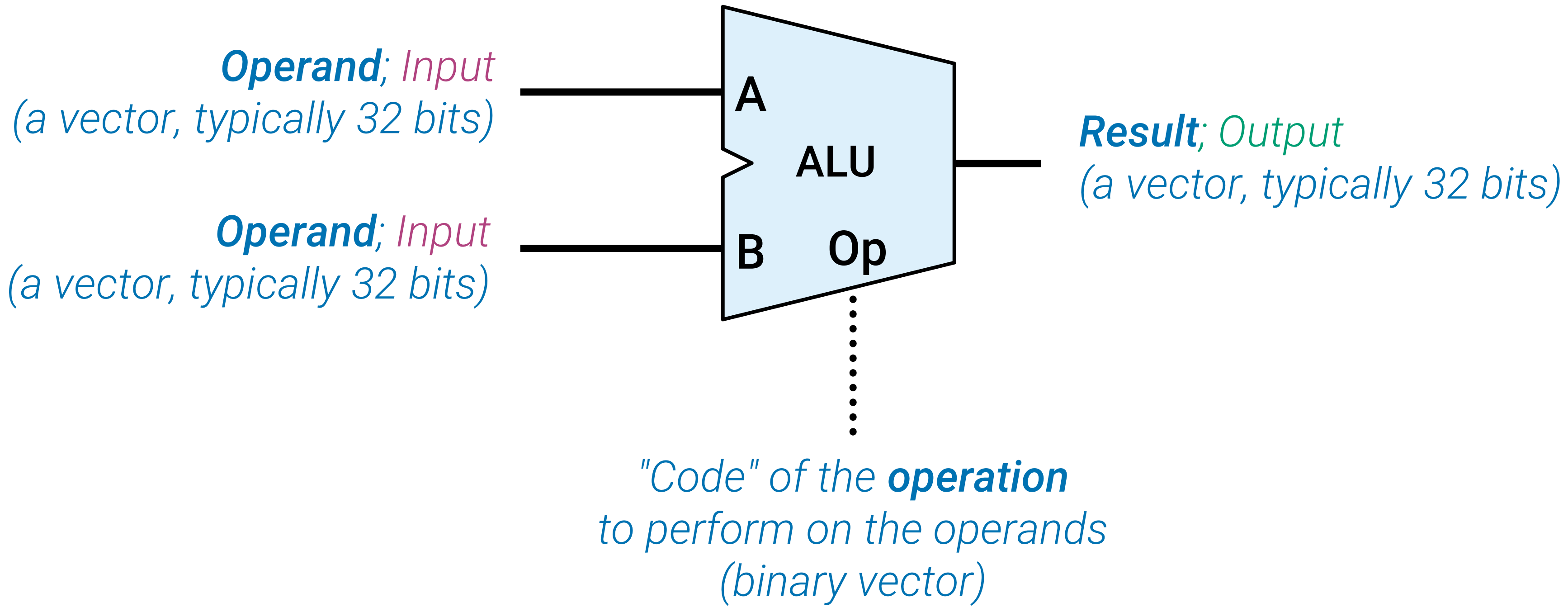
Part I: Data Path

- Data path, also called **datapath,** contains
  - An arithmetic-logic unit (ALU)
  - A register file

- **ALU** performs the operations on program variables
  - E.g., bitwise, logic, shift, comparisons, etc.
- **Register file** is an array of registers for keeping the program variables and some other special uses

Arithmetic-Logic Unit
A High-Level View

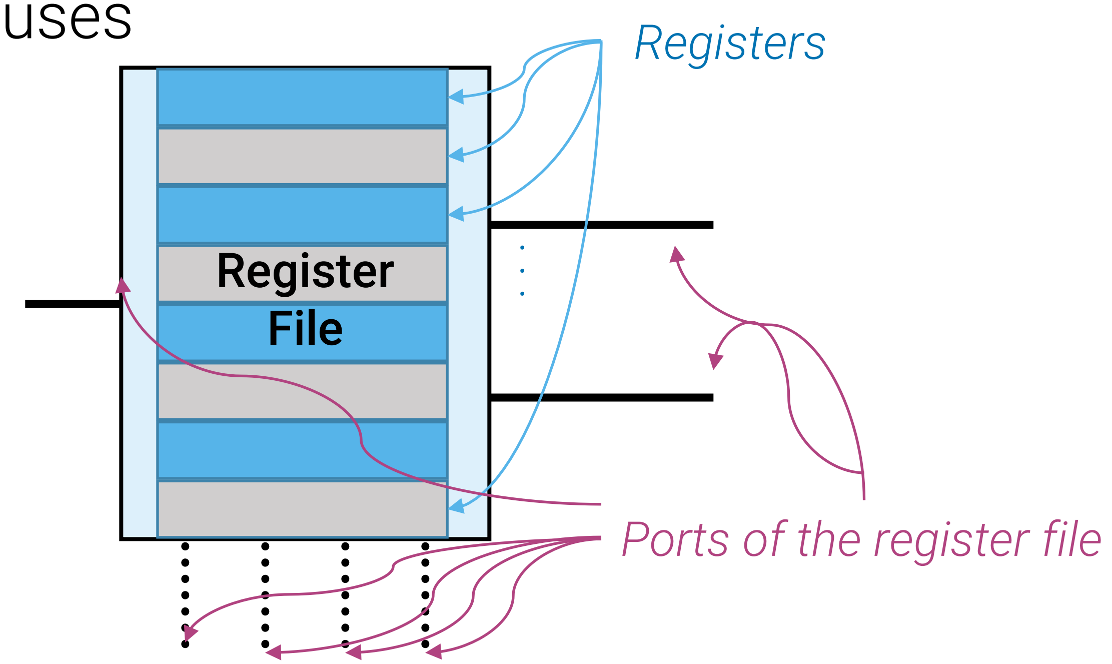
- *Recall:* ALU performs operations on program variables
    - E.g., bitwise, logic, shift, comparisons, …


Operand; *Input*
(a vector, typically 32 bits)
Operand; *Input*
(a vector, typically 32 bits)
A
ALU
B
Op
Result; *Output*
(a vector, typically 32 bits)
"Code" of the ***operation***
to perform on the operands
(binary vector)

Register File

A High-Level View

- *Recall:* Register file is an array of registers for keeping the program variables and some other special uses

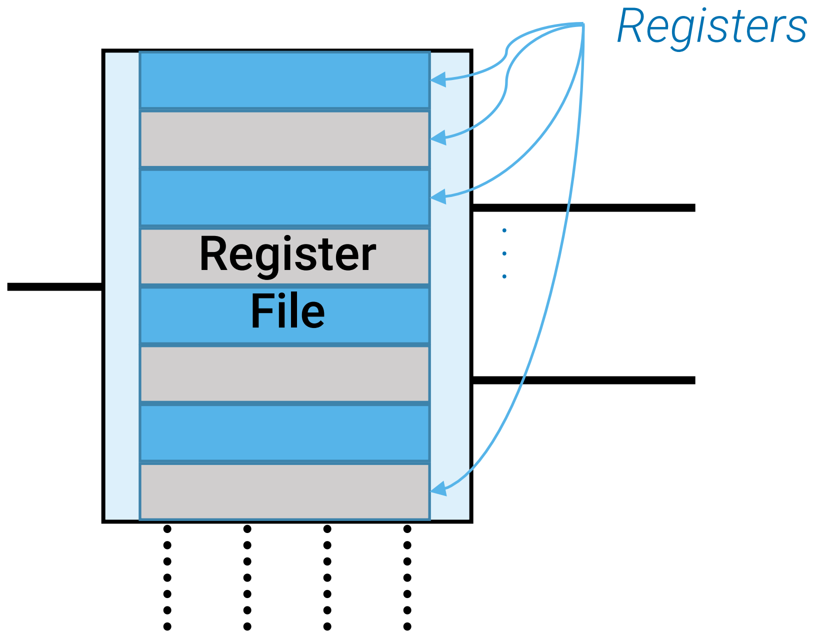


- All ports (in, out) of the register file connect to all registers in the array
- Two registers can be read in the same clock cycle

Register File

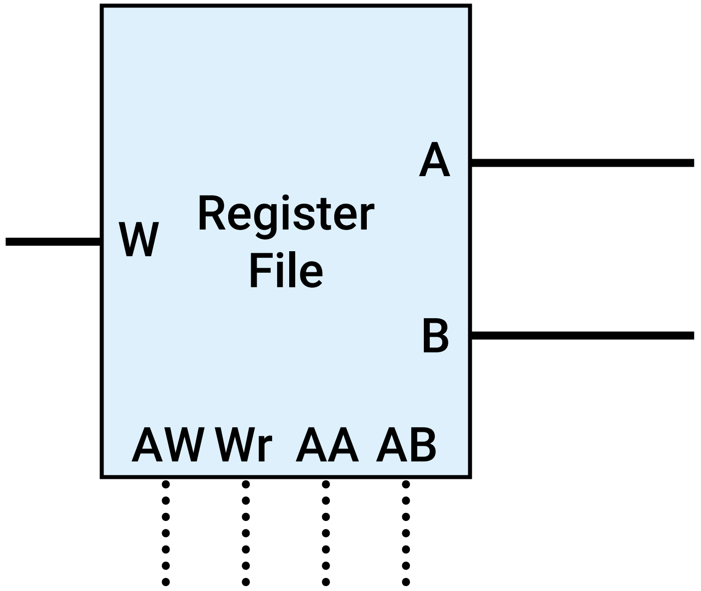
A High-Level View

- Two registers can be read in the same clock cycle
- Reading from and writing to registers is fast (one CPU clock cycle)
- FF is expensive per bit
  - DFF typically contains ~20 transistors
- In practice, register files are constructed using SRAM memory cells
  - SRAM cell typically contains ~6 transistors



Register File

A High-Level View



***Data to write** to register "AW"*
(a vector, typically 32 bits); Input

***Data from register** "A"; Output*
(a vector, typically 32 bits)

***Data from register** "B"; Output*
(a vector, typically 32 bits)

***Index of the register** "AW" to write to*
(index is often called "address"); Input

***Write enable** – if active,*
data on input W port is
written to register AW; Input

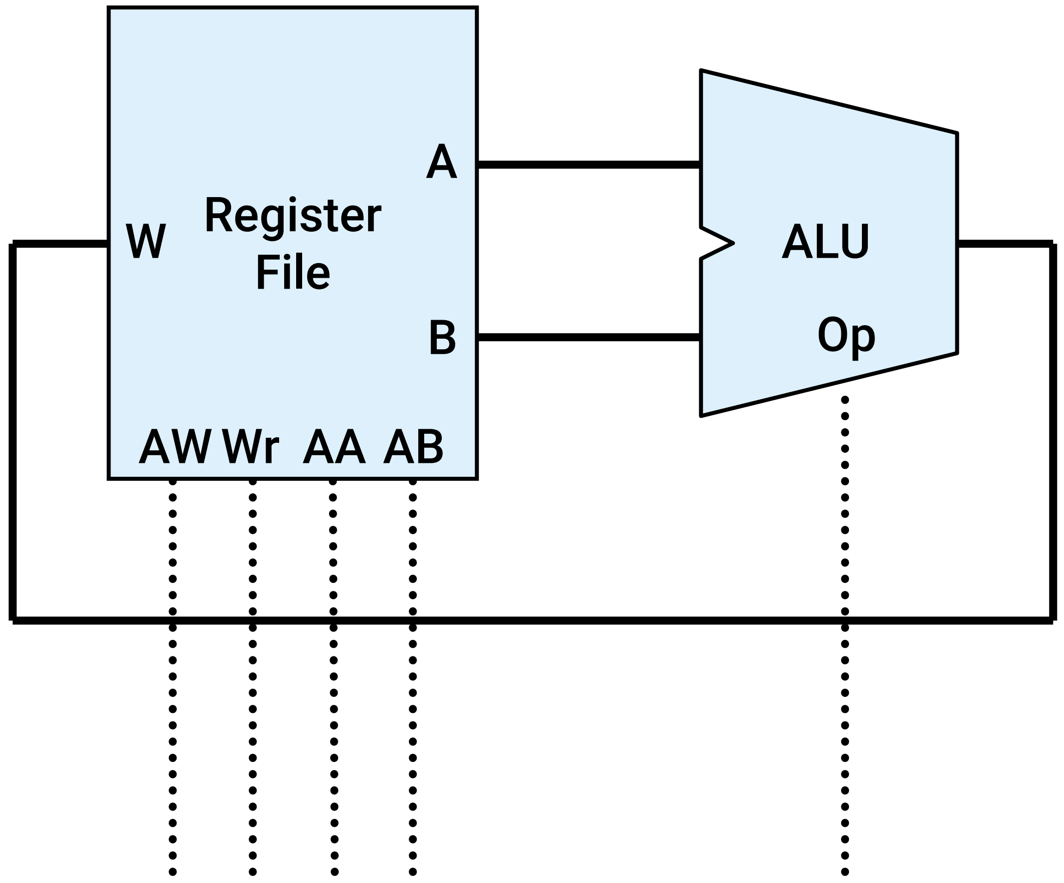
***Index of the register** "B"; Input*
(index is often called "address")

***Index of the register** "A"; Input*
(index is often called "address")

Datapath

Connected, Final

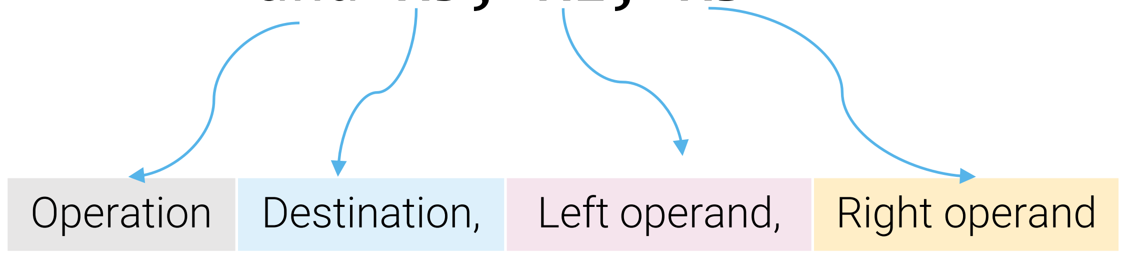
- ALU receives operands from the outputs of the register file
- Register file receives the result of the operation performed by the ALU and saves it in one of its registers
- Registers in the register file and the ALU are tightly coupled (i.e., close, wires connecting them are short), which makes data transfer between them **fast**



Datapath

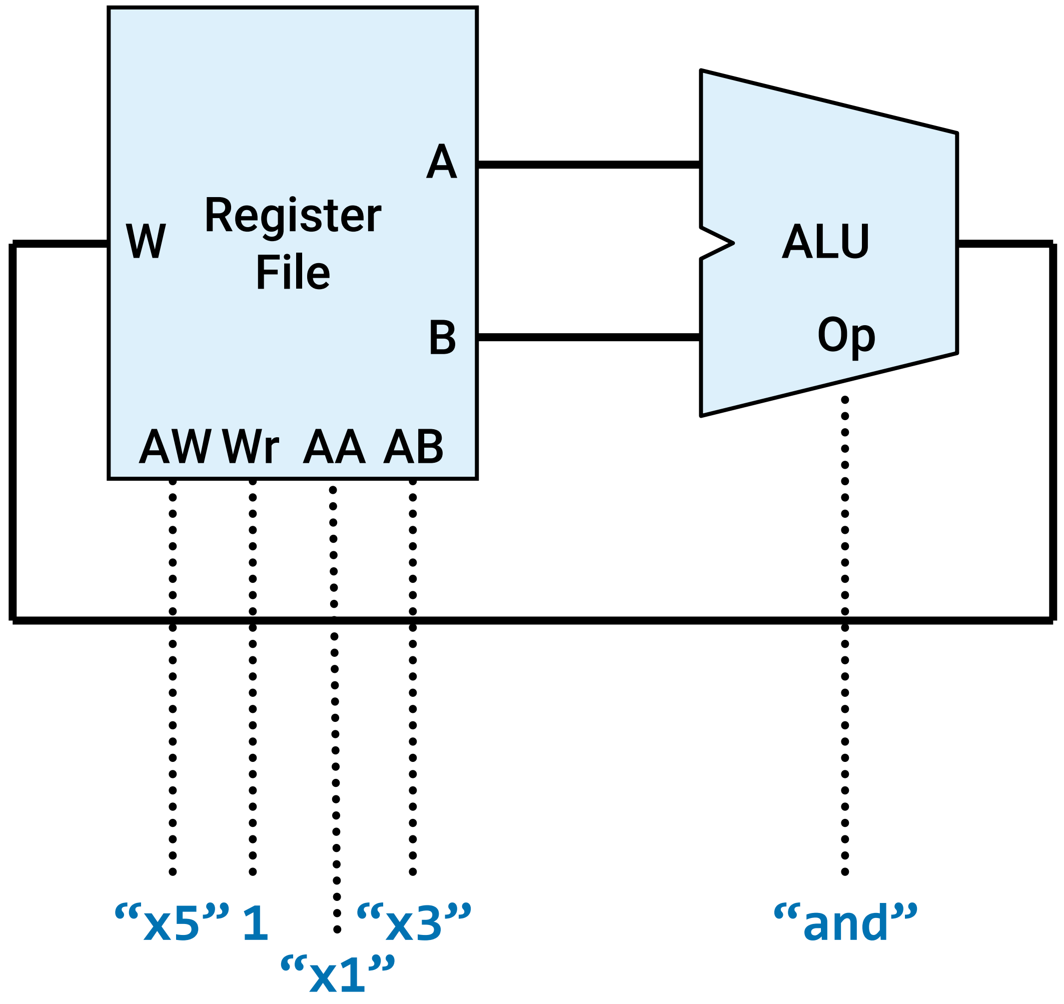
Connected, Final

- Example instruction
  - `and x5, x1, x3`

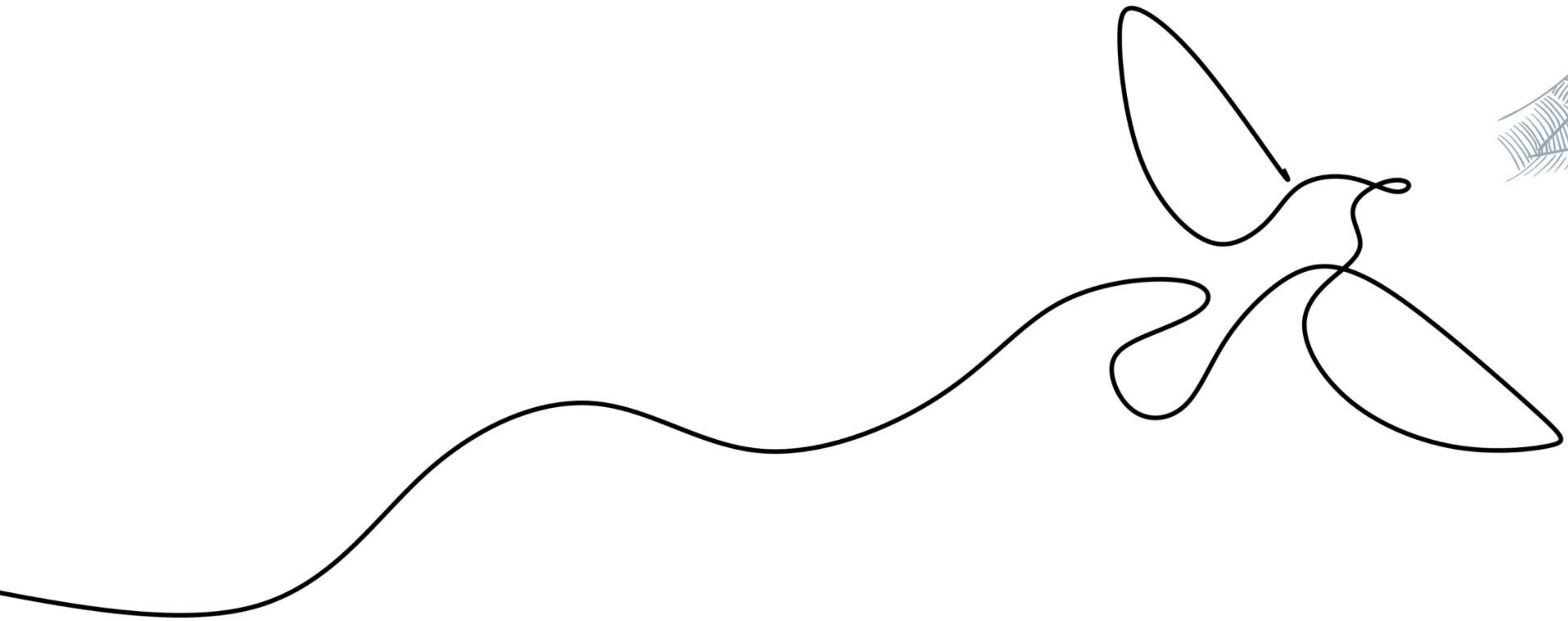
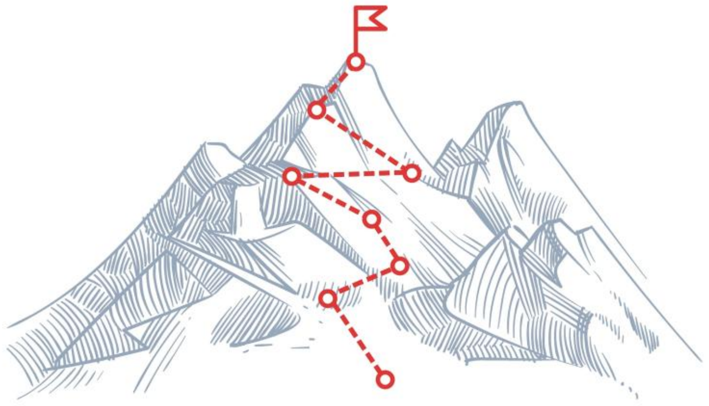


Note: Instruction is a binary vector, structured

- operation code for **and**
- $(101)_2$ for register x5
- $(001)_2$ for register x1
- $(011)_2$ for register x3



Note: “x5” and others should be replaced by the corresponding binary vectors

Two Parts of a Processor

Recall

- **Part I: Data path**
  - Concerns everything related to data
    - Variable storage/read/write
    - Operations on variables
    - Includes the digital circuits that perform operations on data or hold data
- **Part II: Control path**
  - Concerns everything related to the code execution
    - Instruction reading, decoding, and sequencing
    - Manages the digital circuits of the data path so they perform the operations as specified by the instructions

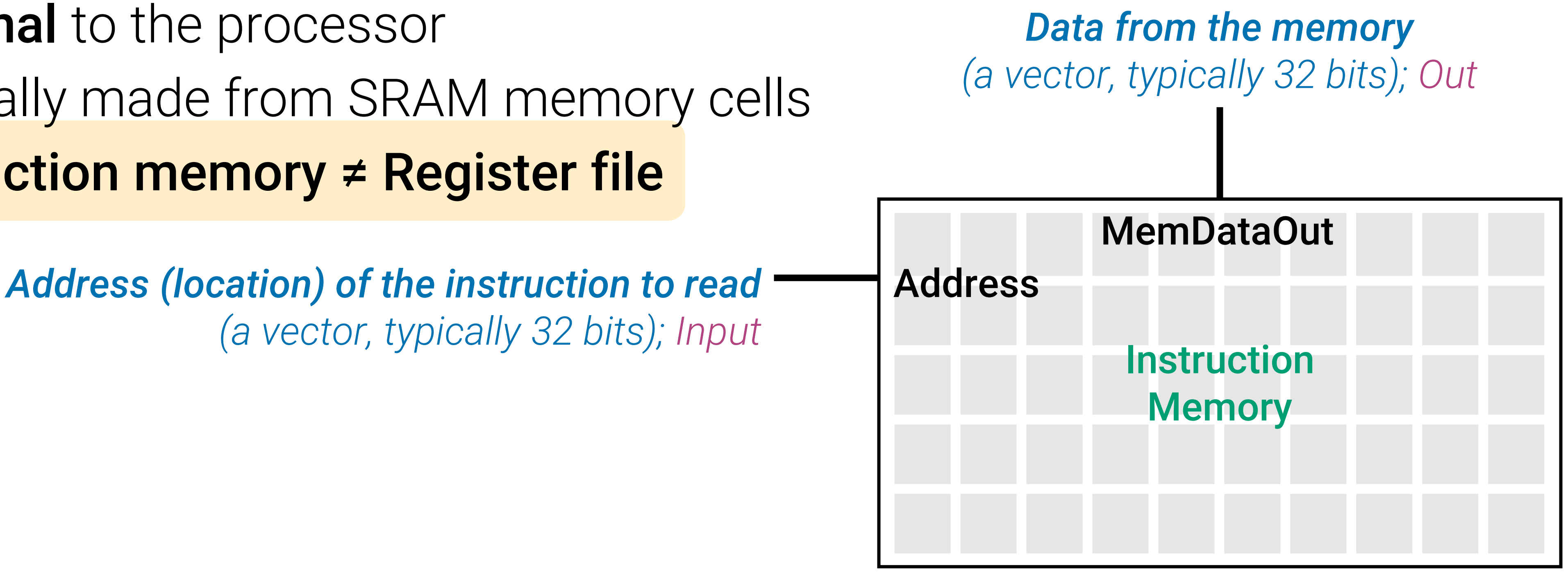
Part II: Control Path

- The control logic of the processor
- Roles
  - Instruction reading (loading instructions from the program in memory)
  - Sequencing instructions (ensuring the correct program order)
  - Decodes the instruction from its binary form
    - Depending on the instruction, sets control signals for the ALU and the Register File accordingly
- Implements the finite-state machine of the processor

Instruction Memory

Role I: Reading Instructions

- Where is the program?
  - The program (instructions) resides in a larger memory external to the CPU
- We call this memory **instruction memory**
  - **External** to the processor
  - Typically made from SRAM memory cells
  - **Instruction memory ≠ Register file**

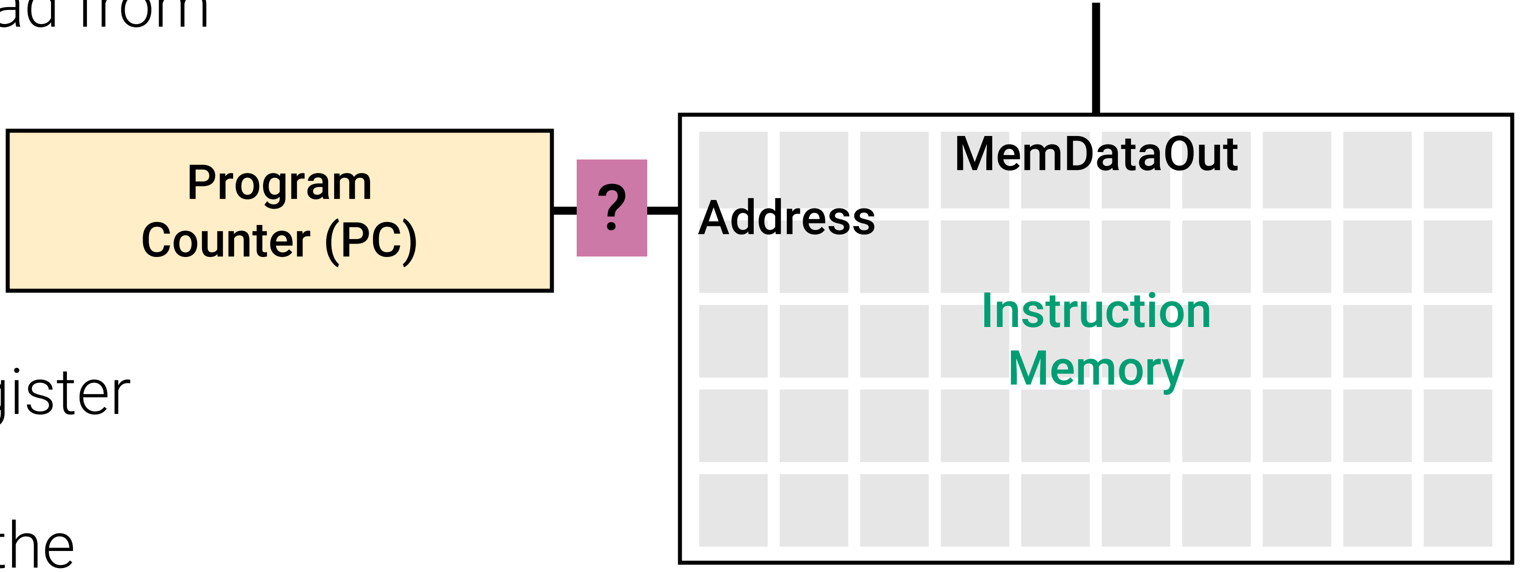

Data from the memory
(a vector, typically 32 bits); Out
MemDataOut
Address (location) of the instruction to read
(a vector, typically 32 bits); Input
Address
Instruction Memory

Note: Additional interfaces not shown

Control Path

Role II: Sequencing Instructions

- Control path logic sets the address of the instruction to be read from the memory

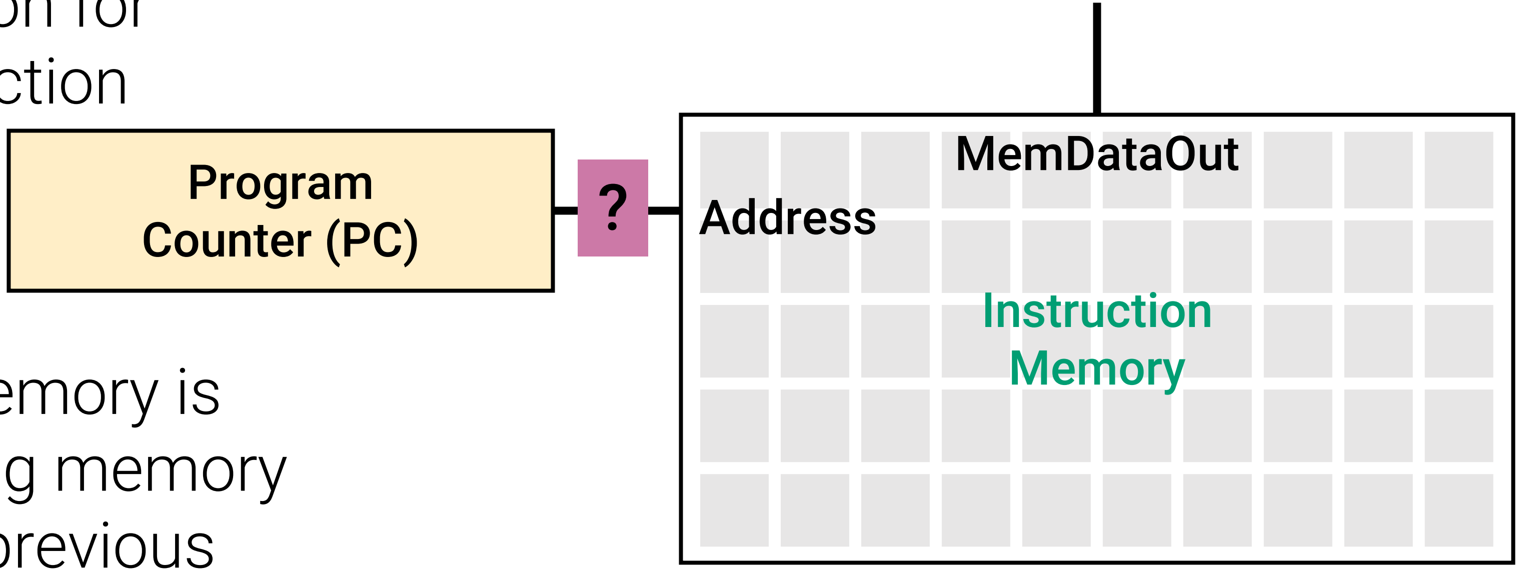


- CPU has an additional register dedicated to keeping the address (location) of the instruction in memory
  - **Program Counter (PC)**

Control Path

Role I: Sequencing Instructions, Contd.

- Control logic updates the program counter (PC), in preparation for reading of the **next** instruction



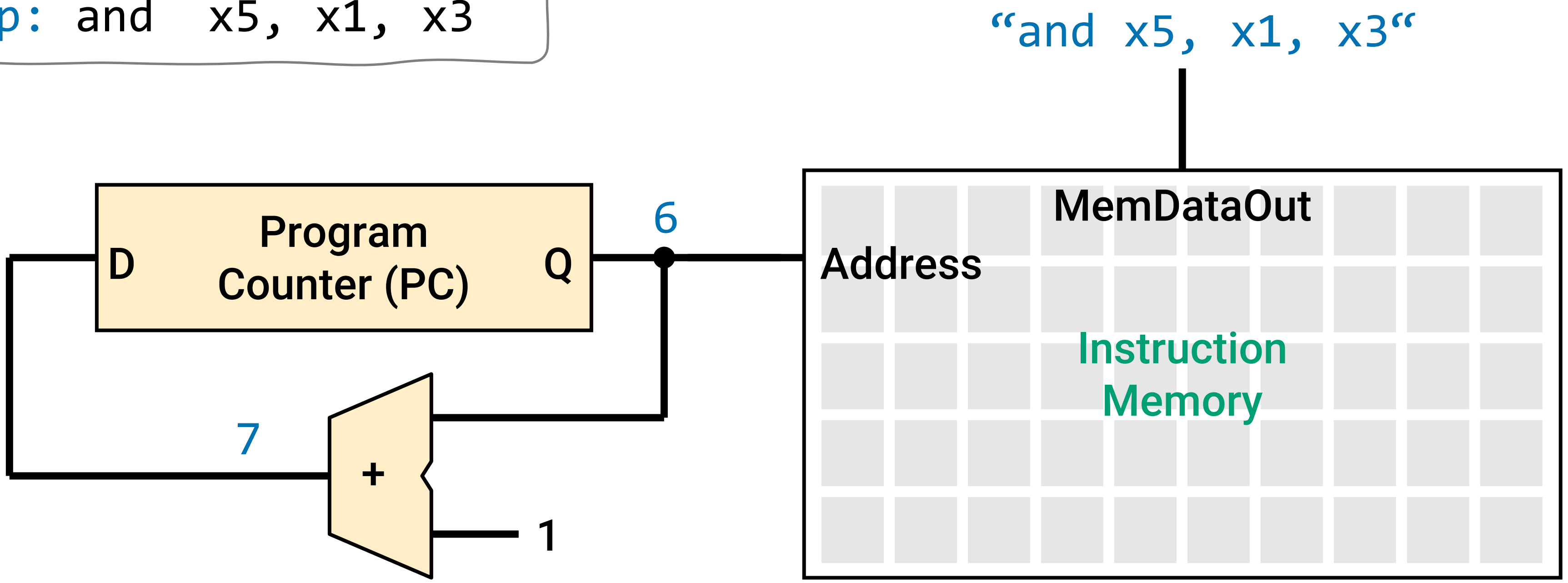
- The next instruction in memory is typically at the neighboring memory address, higher than the previous
  - PC should be incremented to "point" to the next instruction

Sequencing Instructions

Contd.

- *Recall:* PC register holds the instruction address
- Adder increments the contents of the PC, so that the subsequent instruction is read in the next clock cycle
- Example instruction at the line/index/address **6**

```
6 loop: and  x5, x1, x3
```

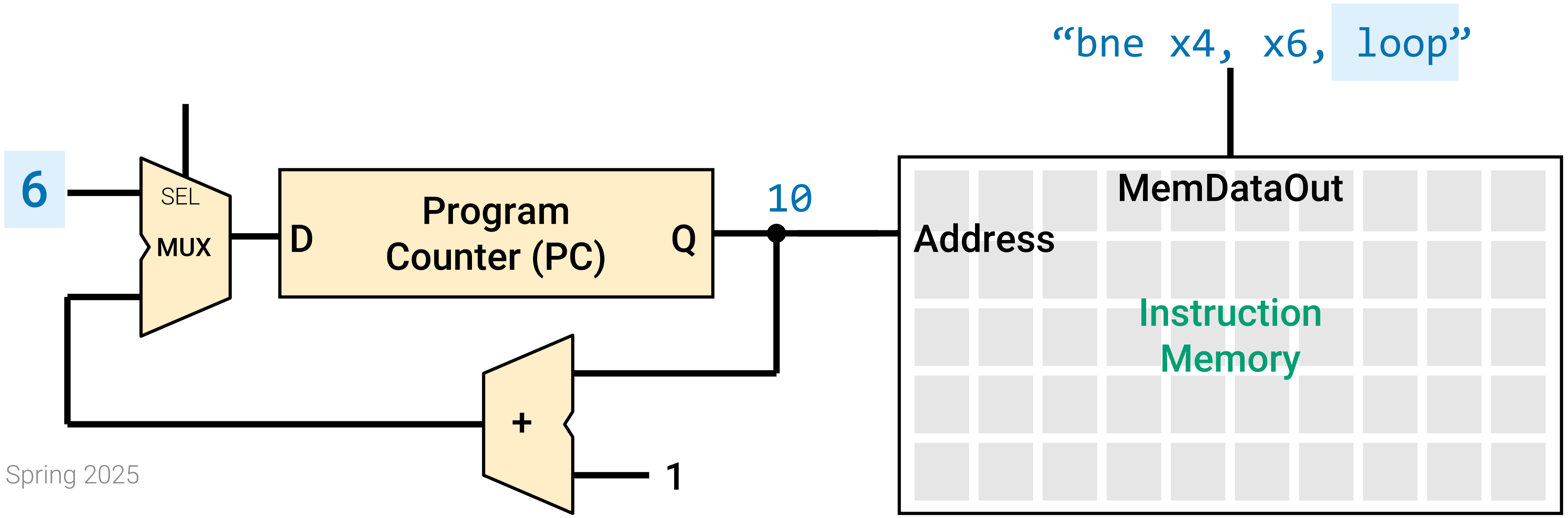


Sequencing Instructions

Contd.

- Example instruction at the line/index/address **10**
  - Next instruction (label **loop**) on line/index/address **6**
  - PC must be able to accept a value other than the one computed by the adder

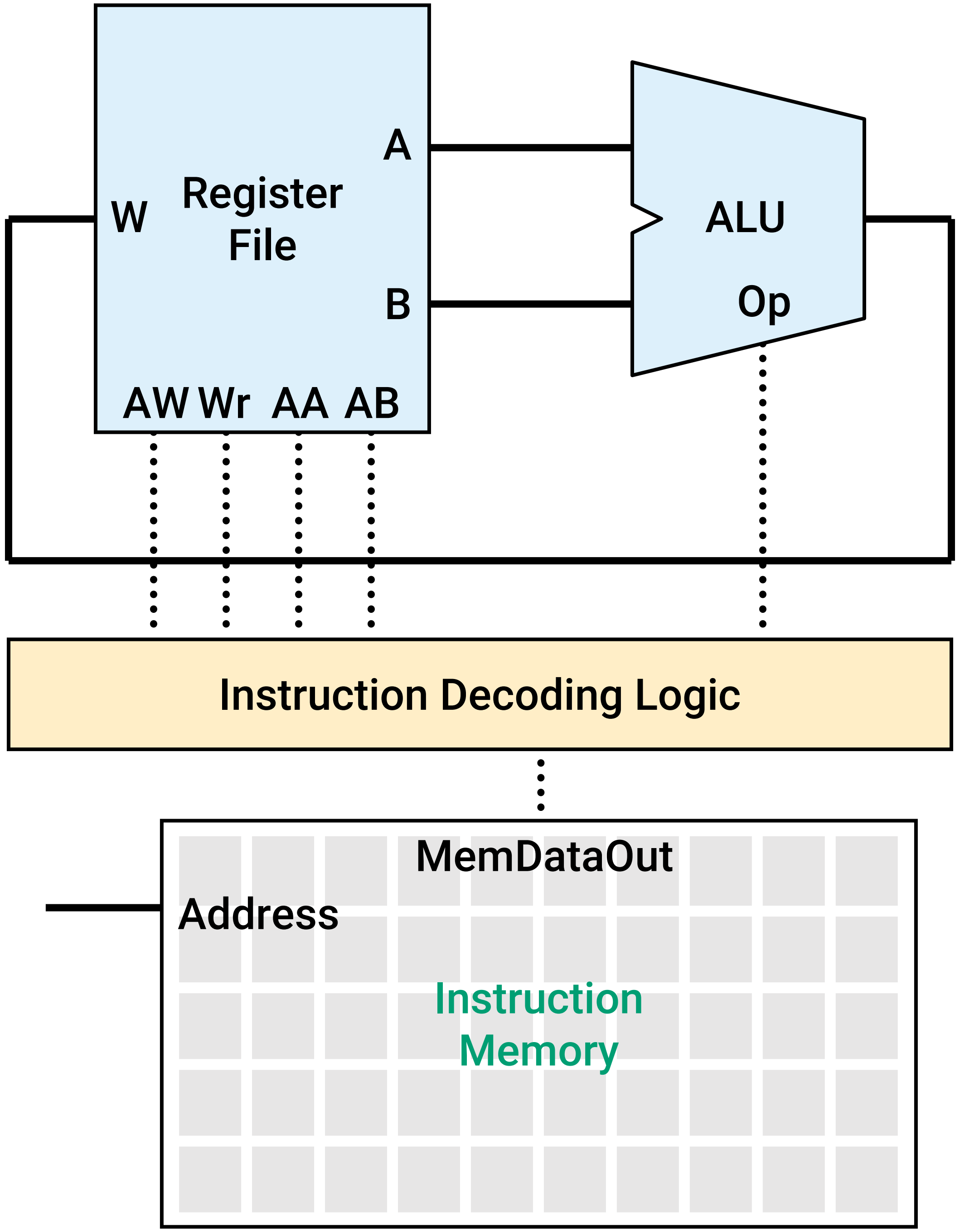
```
 6  loop: and  x5, x1, x3
 …  …
10        bne  x4, x6, loop
```

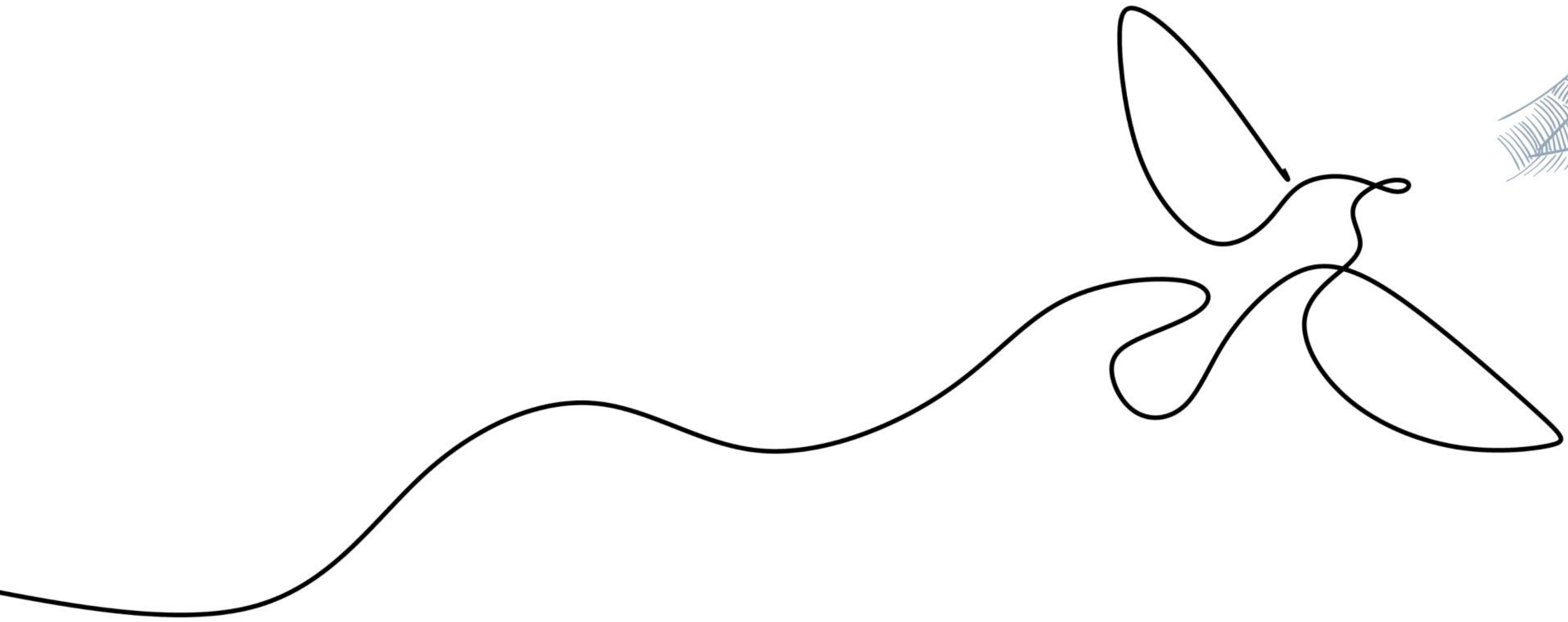
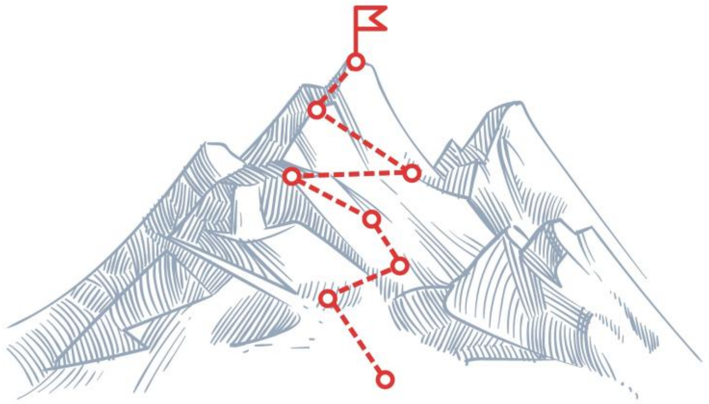


Control Path

Role III: Decoding Instructions

- Once the instruction is read from the instruction memory, it must be **decoded**
- Decoding is **parsing** the binary representation of the instruction, to **identify** all the relevant info
    - operation
    - destination
    - operands

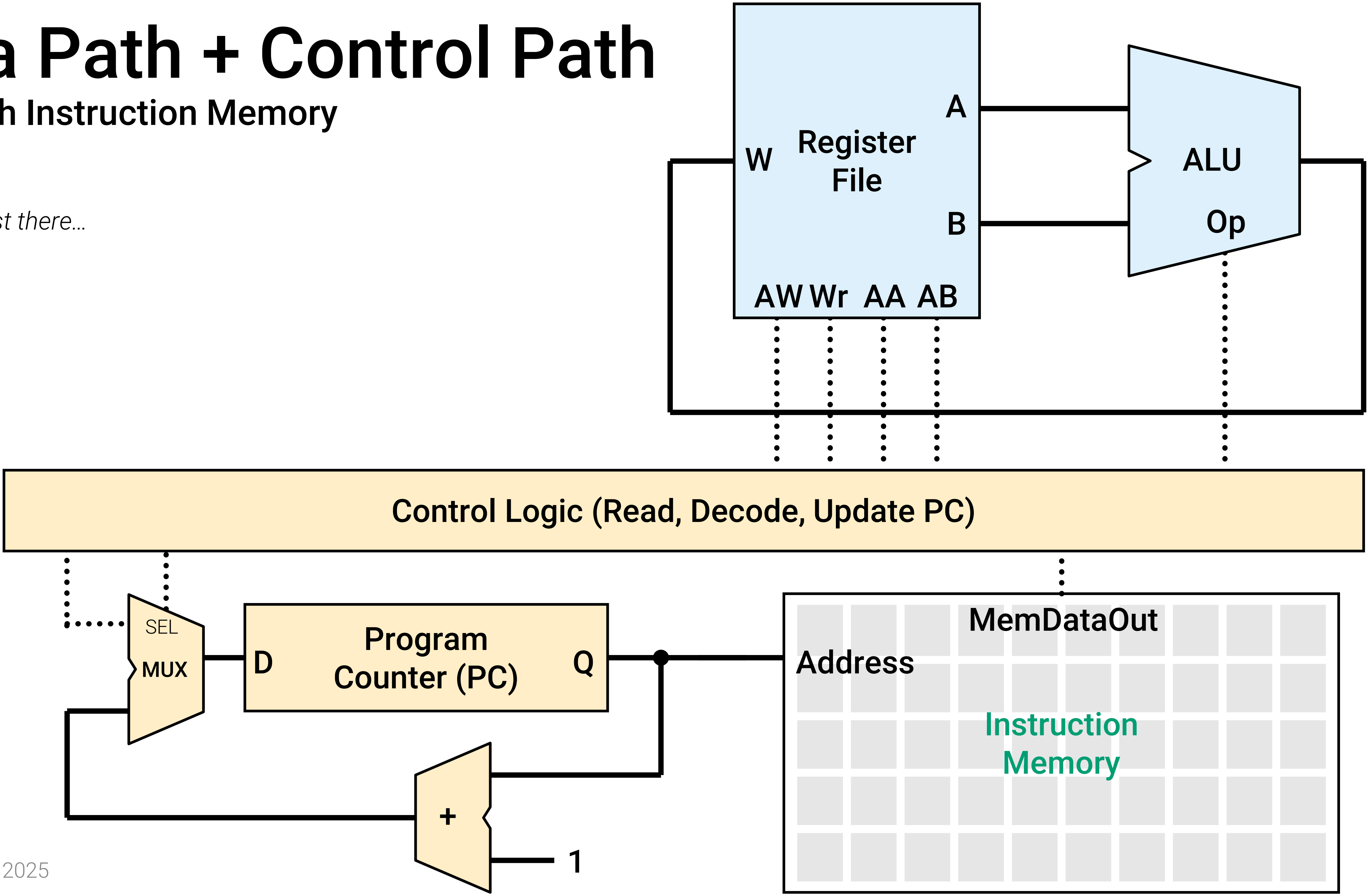


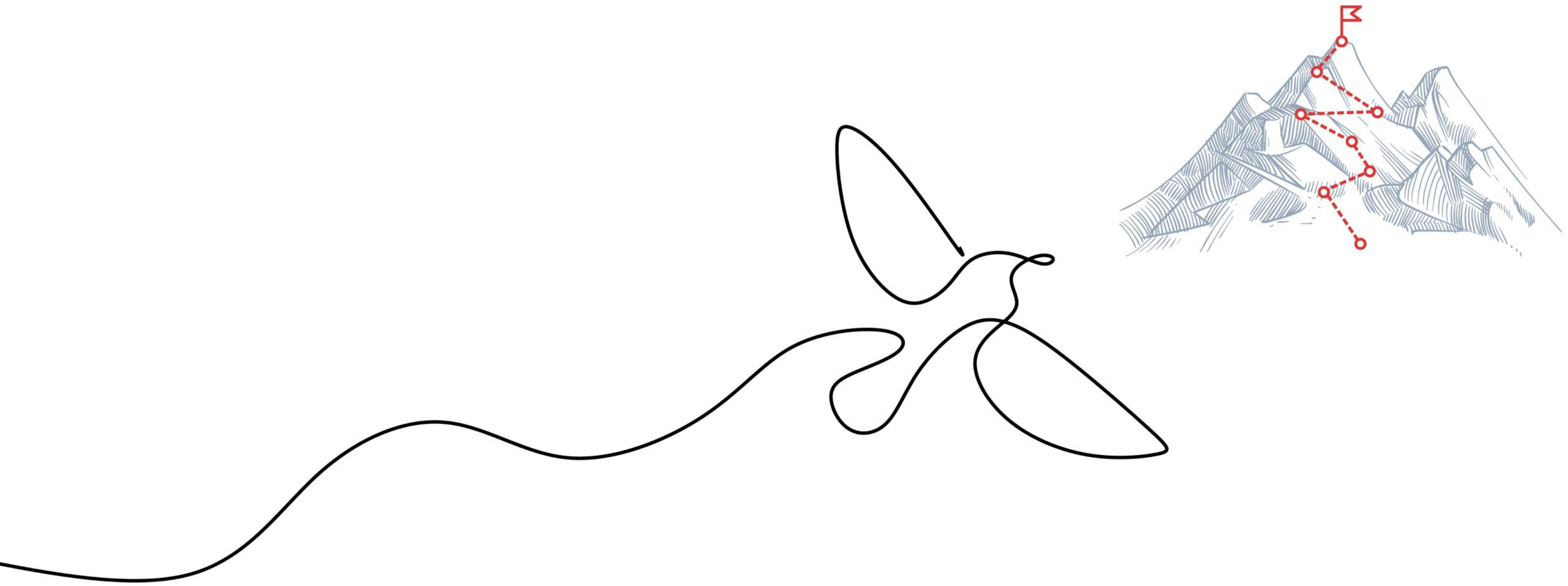



Data Path + Control Path

CPU with Instruction Memory

Note: Almost there…



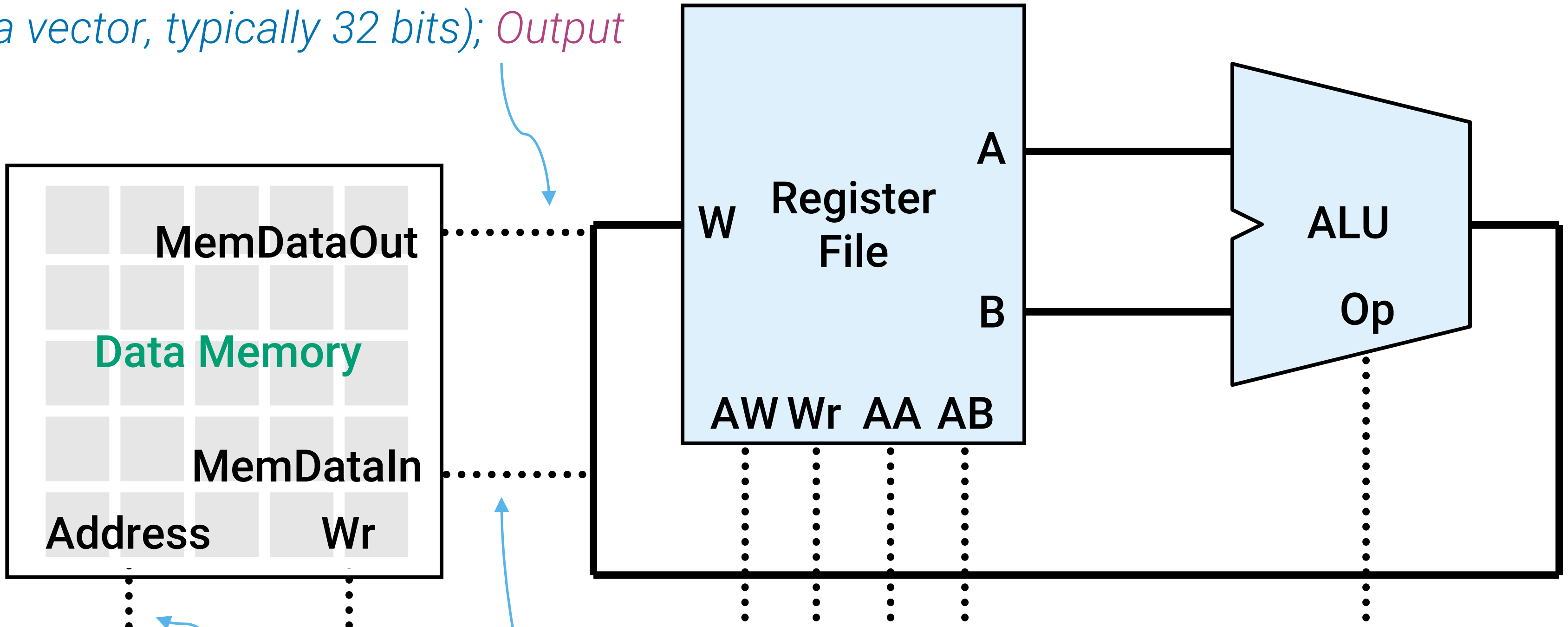


Register File is Not Sufficient

- One remaining challenge: In practice, the quantity of data on which programs operate is way too large to fit in the register file
- There is always an external memory for data, which
  - … can be dedicated to data only or
  - … can be shared by the data and the instructions
  - … is much larger (in capacity) and slower than the register file

Data Memory

Data from the memory
(a vector, typically 32 bits); Output



Address (location) of the data to read
(a vector, typically 32 bits); Input

***Control signal, write enable** (one bit); Input*

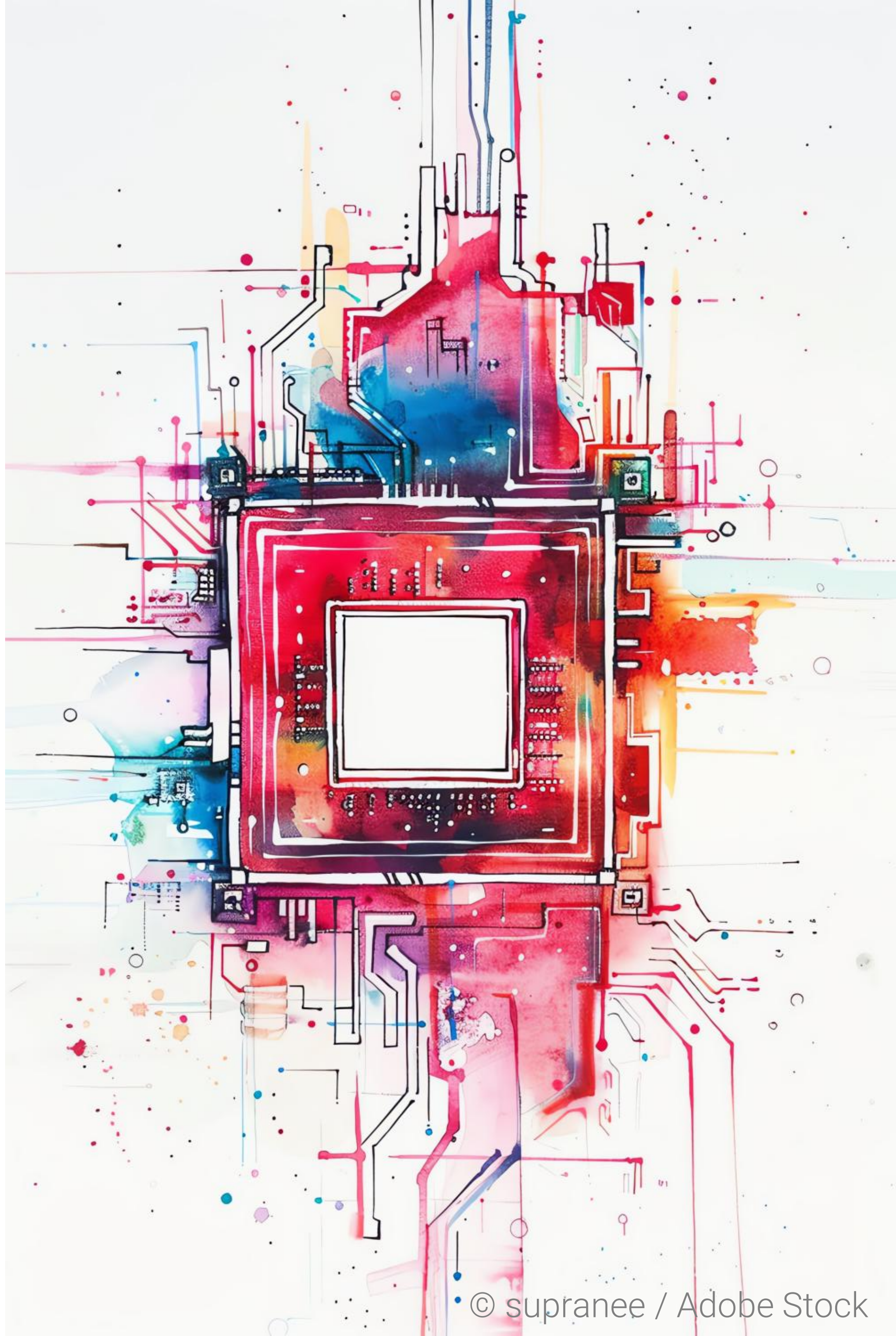
Data to the memory
(a vector, typically 32 bits); Input

Now We Have Everything…

- **…to build a simple CPU**
  - Arithmetic-logic unit (ALU) and Register File for variables and computation
  - Instruction memory
  - Program counter (PC) register for instructions fetched from memory
  - Data memory

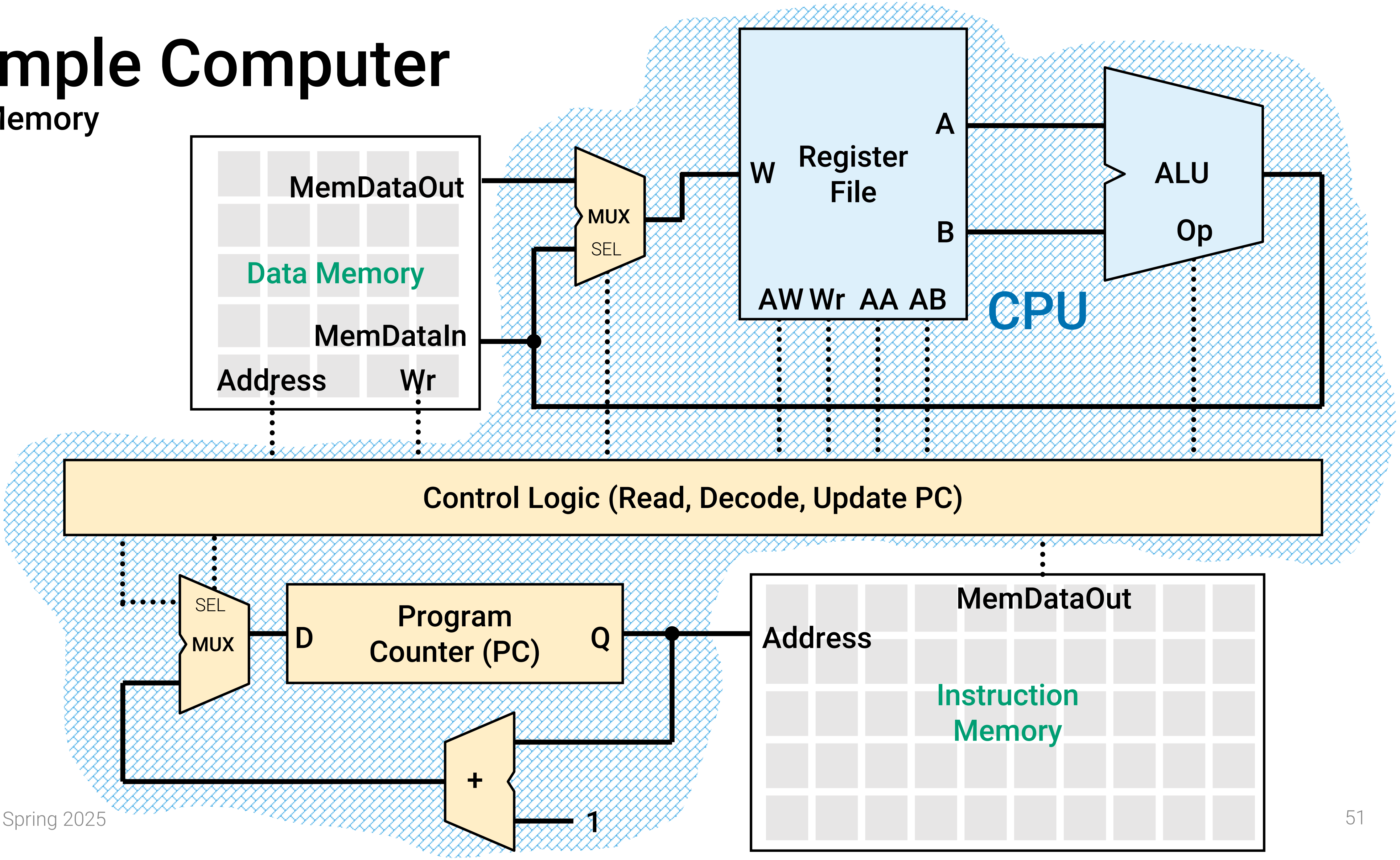
A Simple Computer

Processor (= Data Path + Control Path)
and the data/instruction memory



A Simple Computer

CPU + Memory



Disclaimer…

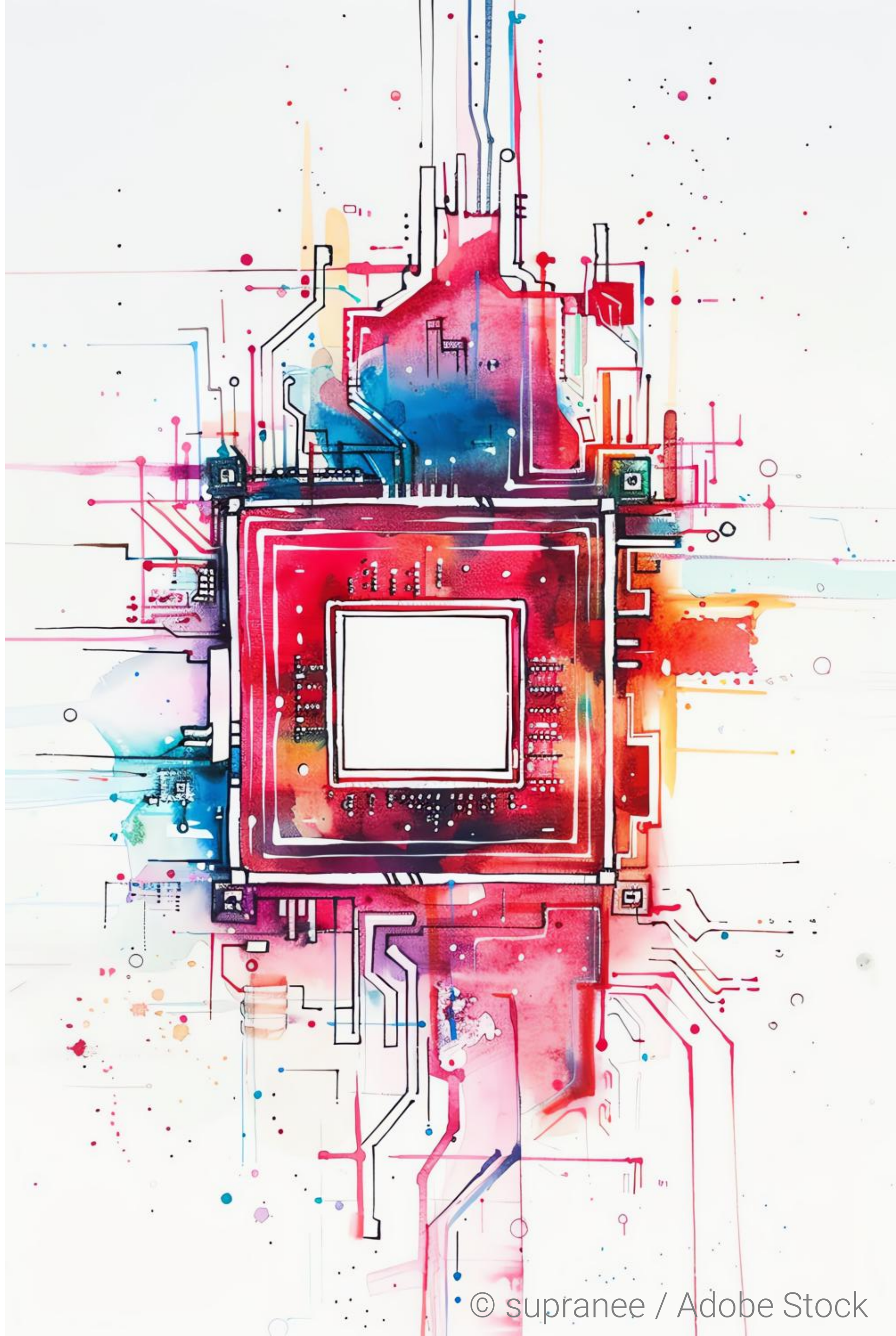
- *A number of simplifications in the previous figure, but the main idea is there*
- *In practice, many more signals and ports exist, along with multiplexers and other logic to enable the execution of any program*
- *Memories support both read and write and have more complex interfaces*

Types of Computer Architecture

- **Harvard** architecture
  - Instructions and data memory reside in **separate** memories
  - See the block diagram on the previous slide
- **Von Neuman** architecture
  - Instructions and data reside in the **same** memory
  - We say the memory is **unified**
  - In general-purpose computers (desktops, laptops, servers, etc.), Von Neumann architecture is **predominant**

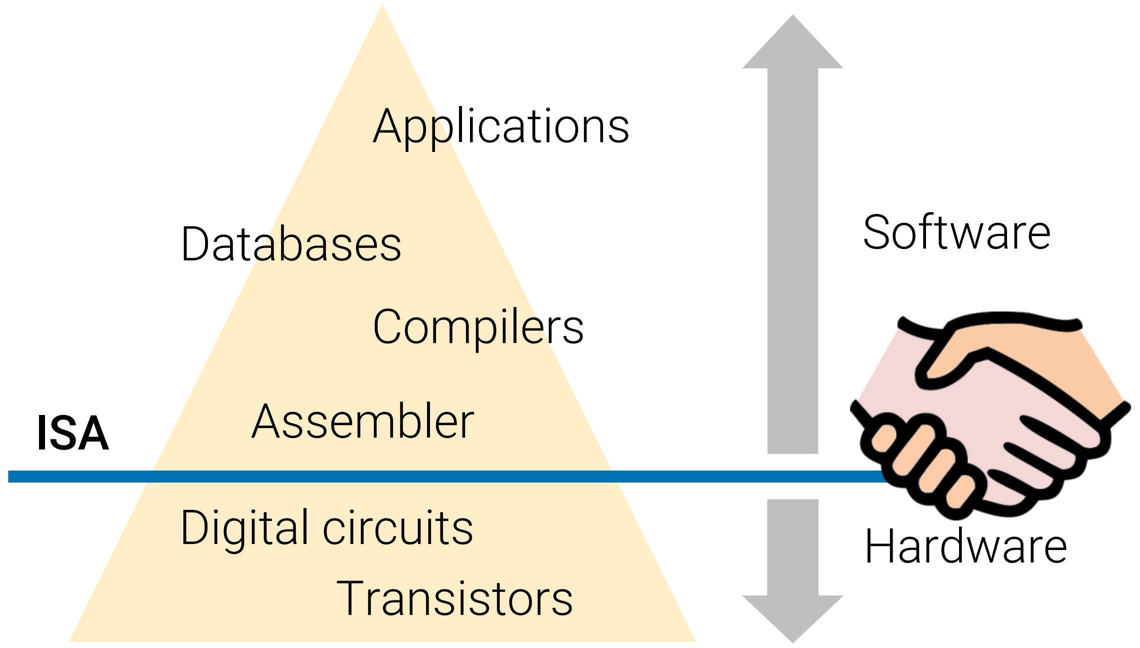
Instruction Set Architecture

One of many important computing abstractions



Instruction Set Architecture

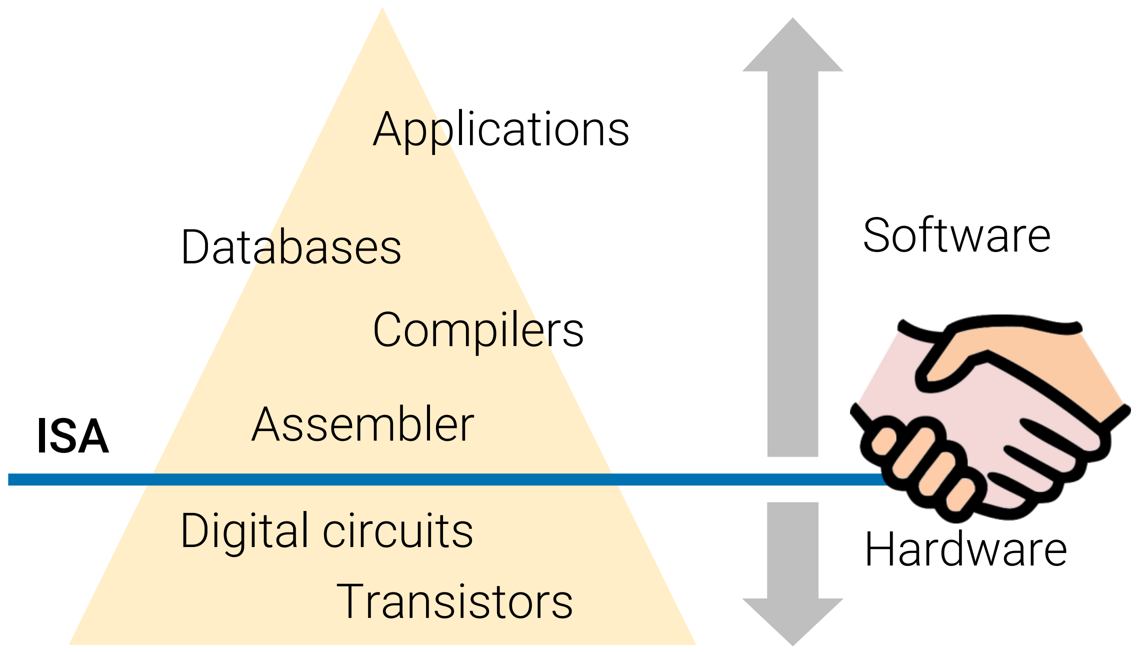
- Instruction Set Architecture (**ISA**) refers to the **set of instructions** that a CPU can execute and the **programming model** that these instructions define for software developers



Instruction Set Architecture

Contd.

- ISA **abstracts** away the low-level details of the CPU hardware implementation
- ISA defines an **interface** between the hardware and software components of a computer system



Why Is ISA So Important?

✓Formalization of the programmer view allows the CPUs to continue supporting the same ISA while evolving and improving over time

✓Users profit from the advancements in the CPU architecture (e.g., higher speed) for free, without having to update their programs

- Drawback: fundamental innovations that require changes to ISA are thus prevented; fixed ISA enforces binary compatibility

- Example:
  - IA-32/x86 is a very common ISA introduced by Intel. Intel's Core, Pentium, and Xeon series, as well as AMD's Ryzen, Athlon, and EPYC series conform to this ISA

Typical Details of an ISA

- **Instruction set**
  - What operations the processor can directly execute
- **Instruction encoding**
  - Representations of instructions in binary
- **Registers**
  - Where the processor can store intermediate results and operands
- **Data types and formats**
  - Example: integers, floating-point numbers, characters
- **Memory addressing modes**
  - How the processor can access the operands from the memory

RISC and CISC ISAs

- **C**omplex vs. **R**educed **I**nstruction-**S**et **C**omputers (CISC vs. RISC)
- RISC ISAs are simpler, contain fewer classes of instructions, and are more regular and easier to implement than CISC
  - Most Intel processors are CISC; MIPS, Alpha, Sparc, RISC-V are RISC
  - *Note: More about what makes RISC ISAs advantageous will be discovered and analyzed in CS-208 Computer Architecture course*

- In CS-173, we will study RISC-V open-source ISA

RISC-V®

Why RISC-V®

Simplicity is the ultimate sophistication.

—Leonardo da Vinci

- RISC-V ISA was born in the last decade with the goal to become a universal ISA, suiting all purposes (tiny devices, high-performance computers)
- Belongs to an open, non-profit foundation whose goal is to maintain the stability of RISC-V, carefully and slowly evolve it, and make it as popular for hardware as Linux is for operating systems

Why RISC-V®
Contd.

- RISC-V ISA is **modular**
  - At the core is a base ISA called RV32I (32-bit integer operations)
  - RV32I is frozen and will never change
  - Modularity is achieved with optional standard ISA extensions that hardware can include or not
  - For example, RVIMFD ISA has
    - RV32I,
    - multiply,
    - single-precision floating point, and
    - double-precision floating-point extensions

Why Modular ISA?

- Conventional approach in computer architecture is **incremental** ISA
  - New CPUs must implement both the new ISA extensions and the past ones
- The result is a substantial growth of ISAs, even when some instructions are no longer in use
- Modular ISA is free of these artifacts

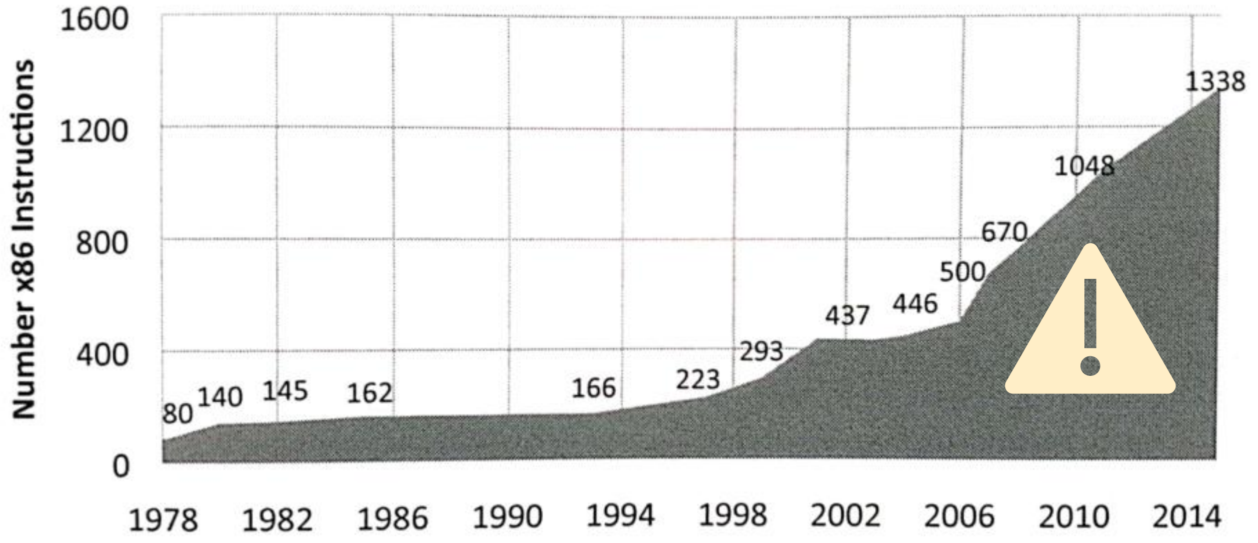
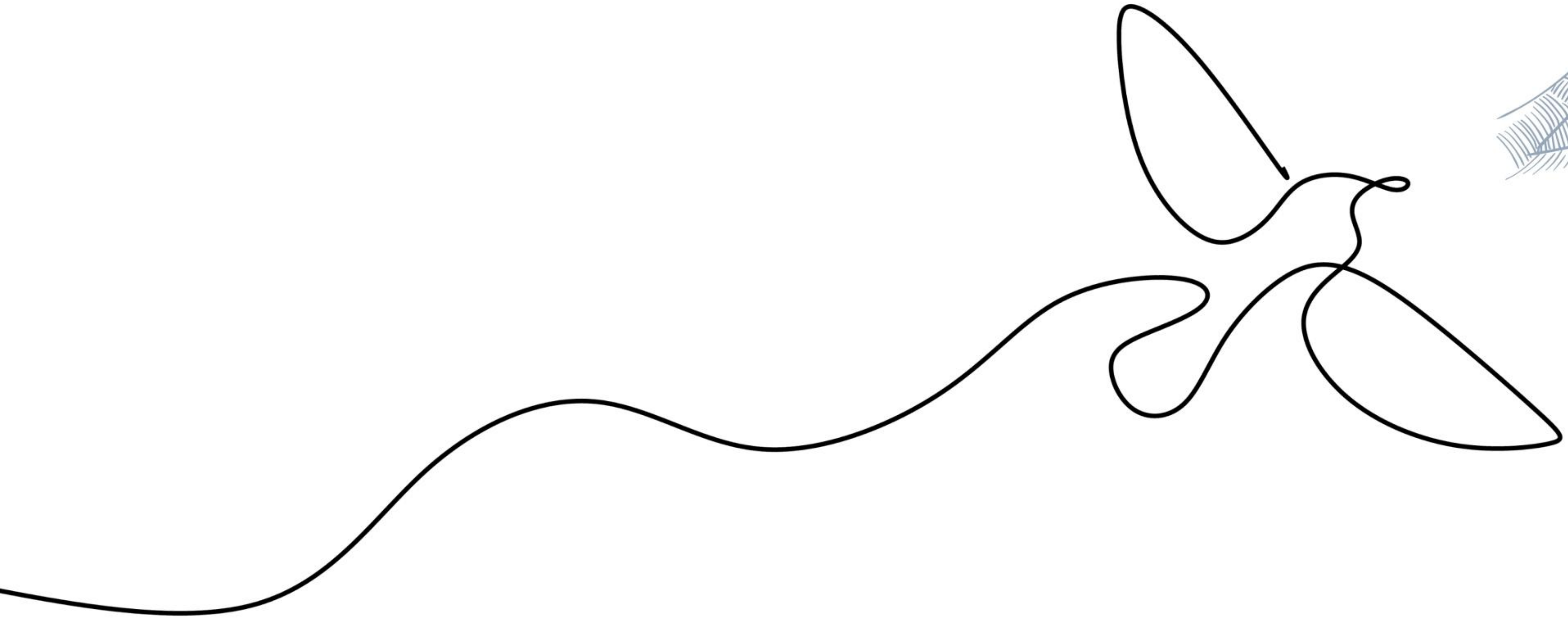
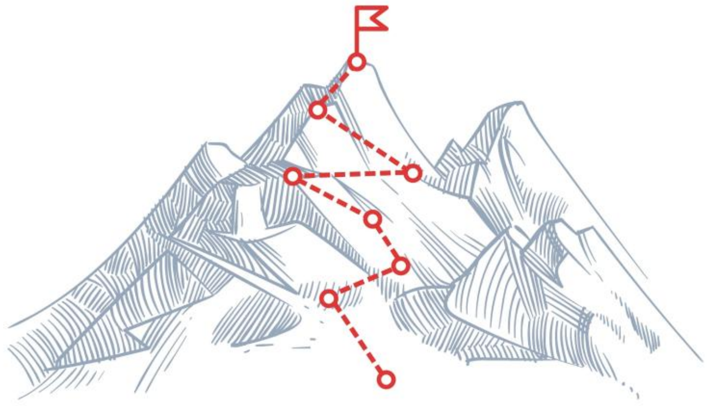


Figure 1.2: Growth of x86 instruction set over its lifetime. x86 started with 80 instructions in 1978. It grew 16X to 1338 instructions by 2015, and it's still growing. Amazingly, this graph is conservative. An Intel blog puts the count at 3600 instructions in 2015 [Rodgers and Uhlig 2017], which would raise the x86 rate to one new instruction *every four days* between 1978 and 2015.

Figure from The RISC-V Reader: An Open Architecture Atlas
Patterson and Waterman





Executive Summary

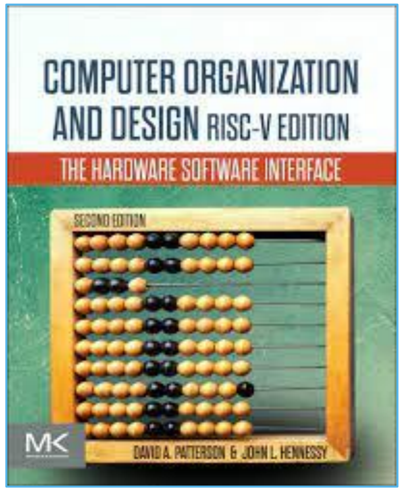
- Basic computer components are
  - Arithmetic-logic unit (ALU)
  - Register file
  - Program counter (PC) register
  - Control logic
  - Instruction and data memory
- **Assembly** language is the last human-readable level of code
- Below assembly is only the machine code (**binary**)
- CPU instructions are kept in the instruction memory, which is different from the register file

Executive Summary

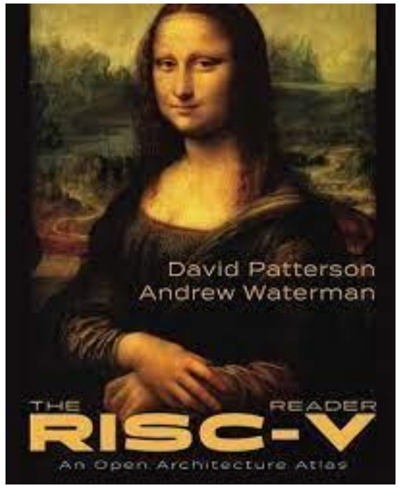
Contd.

- In a Von Neuman computer architecture, instruction and data memories are unified
- Instructions are **encoded** as binary data words, and often contain the type of operation, the sources (operands) and the destination (result)
- **Instruction Set Architecture (ISA)** is one of the important **abstractions** in computing
- ISA refers to the set of instructions that a CPU can execute and the programming model that these instructions define for software developers

Literature


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

- Chapter 4: The Processor
  - 4.1, 4.3



- Chapter 1: Why RISC-V?