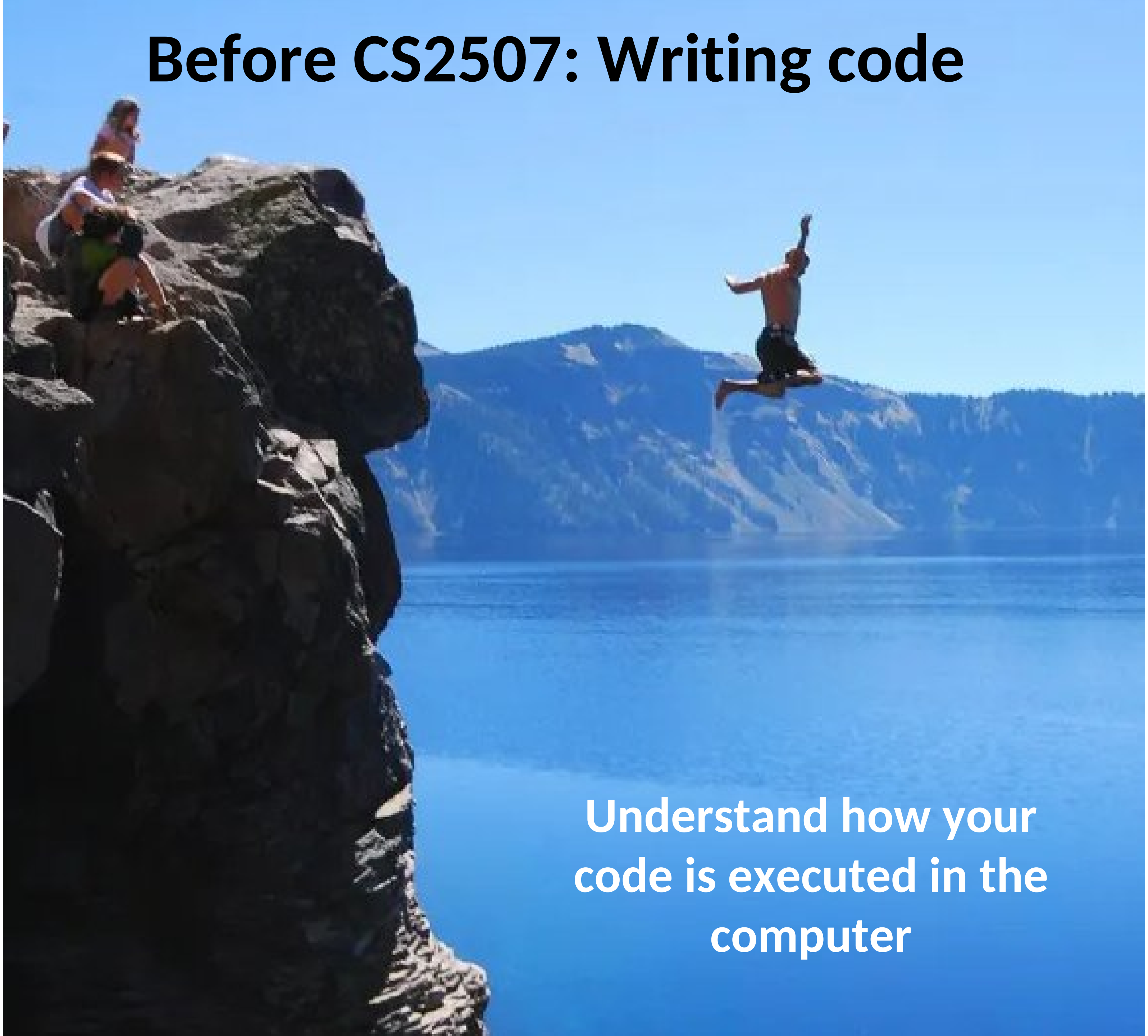
Before CS2507: Writing code



Before

After

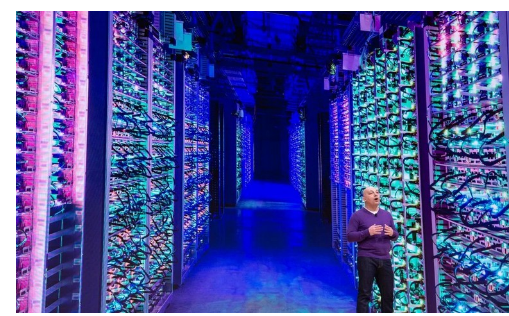
Computer Architecture

Sections 1.1 - 1.4
Section 1.5 (optional)

1

Computers are designed to serve different purposes.

The application domain defines the computer design goals that could be a combination of **performance**, **cost**, energy consumption, and many other aspects



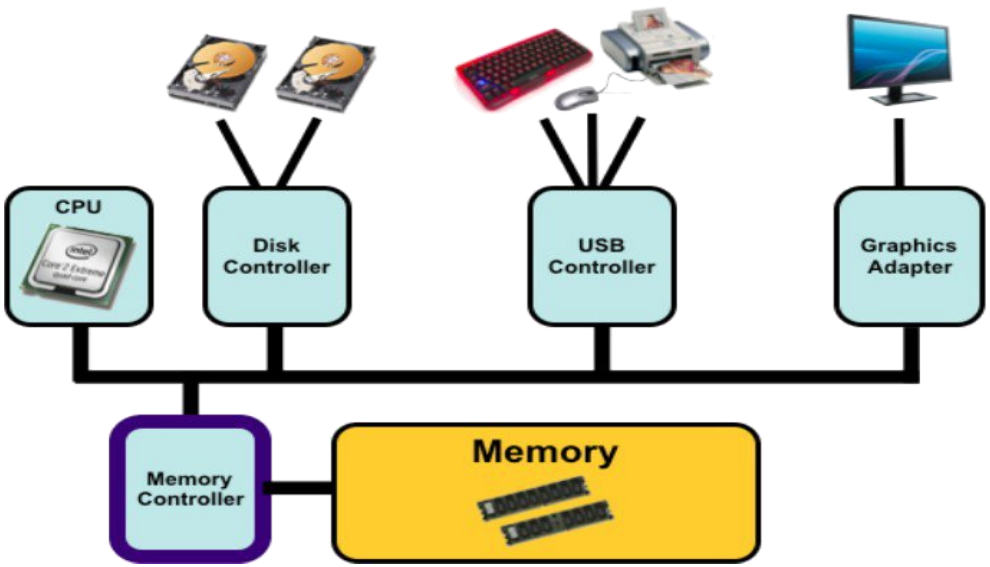
Computer Components

- Same components for all computer

  1Inputs/Outputs

  2)Memory/storage

  3)Processor



- These components would have distinct physical and logical **implementations**
  - the HW choice depends on many factors such as usage, cost, and energy efficiency

What is computer architecture?

Computer Architecture

- Computer architecture is the ***science*** and ***art*** of designing hardware components to create computers that meet ***functional***, ***performance*** and ***cost*** goals

Design Goals
Performance, cost, energy efficiency, reliability, time-to-market

Technology
Circuit, packaging, memory, …

Domains
PMD, server, game consoles, …

Key Decisions in Computer Architecture Design

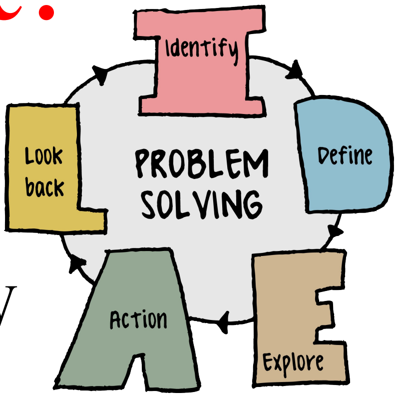
- **Instruction Set Architecture (ISA)** - supported data types, instructions, addressing modes, instruction format (CS2507 Module 2)

- **Processor design** - one or more core? supporting multiple instructions simultaneously? (CS2507 Module3)

- **Memory design** - cache memory and virtual memory (CS2507 Module 4)

- **System Design** - I/O CPU interaction, bus design, power management (CS2507 Module 5)

Why Study Computer Architecture?



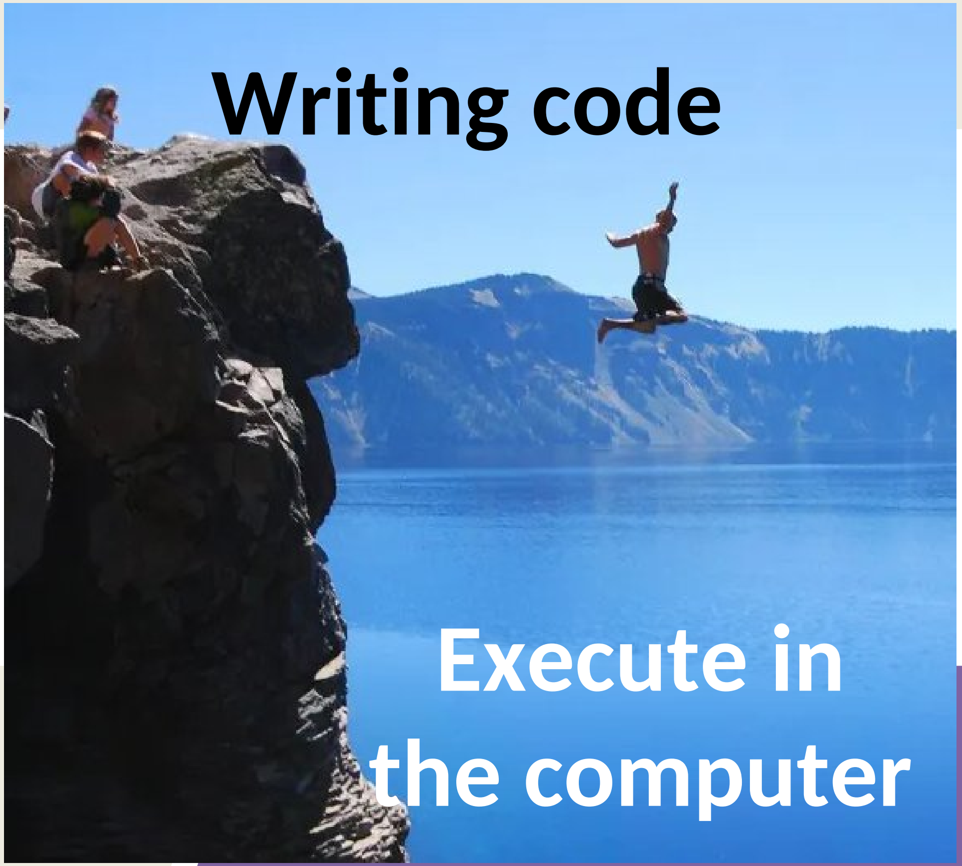
Enhanced Problem-Solving Abilities

- **Debugging:** Understanding the underlying HW helps in identifying and resolving SW issues more efficiently.
- **Performance Optimization:** By understanding how HW works, you can write code that takes advantage of HW features, leading to better performance.
- **Innovation:** Knowledge of computer architecture can inspire new ideas and approaches.

Career Opportunities

- **Hardware Design:** A strong foundation in computer architecture is relevant to companies focused on HW.
- **Systems Programming:** Operating system development, device drivers, and embedded systems require a deep understanding of hardware.

You can code, Right?



How the computer execute your code?

This is the program that is executed in your CPU

Guess what is this code doing?

Code
0x24020004
0x3c011001
0x34240000
0x0000000c
0x24020005
0x0000000c
0x00024021
0x24020004
0x3c011001
0x34240016
0x0000000c
0x24020005
0x0000000c
0x00024821
0x01095020
0x24020004
0x3c011001

For Humans: What is this code doing?

```
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))
sum = num1 + num2
print("The sum  is: ", sum)
```

High level language statements forming the needed logic to receive the input (from user or files), process it, and generate output (to screen or files)

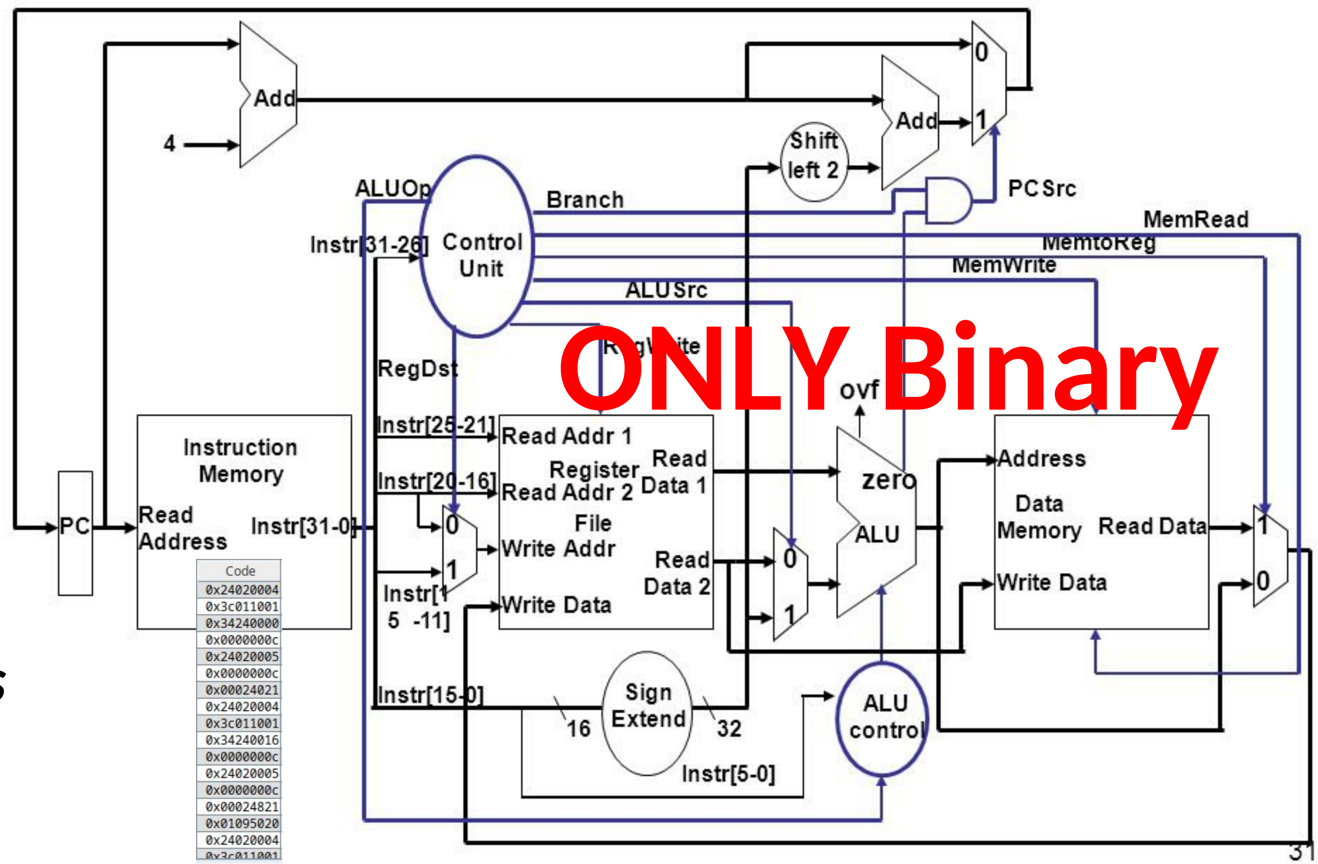
```
#include <iostream>
 int main() {
   int num1, num2;
   std::cout << "Enter the first integer: "; std::cin >>
   num1;
   std::cout << "Enter the second integer: "; std::cin >>
   num2;
   int sum = num1 + num2;
   std::cout << "The sum  is: " << sum << std::endl; return 0;
}
```

Code Execution

The code runs in a CPU that only deal with binary instructions and data

The HLL ***statements*** should be encoded to equivalent **<u>CPU-specific binary instructions</u>**

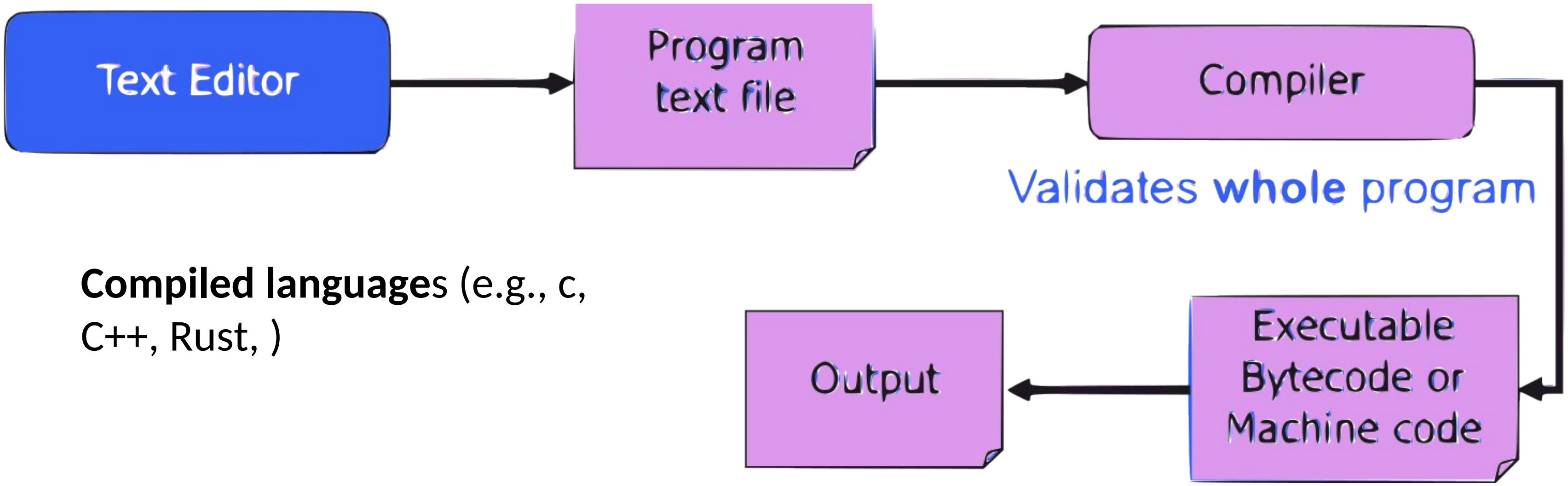
Single Cycle Datapath with Control Unit



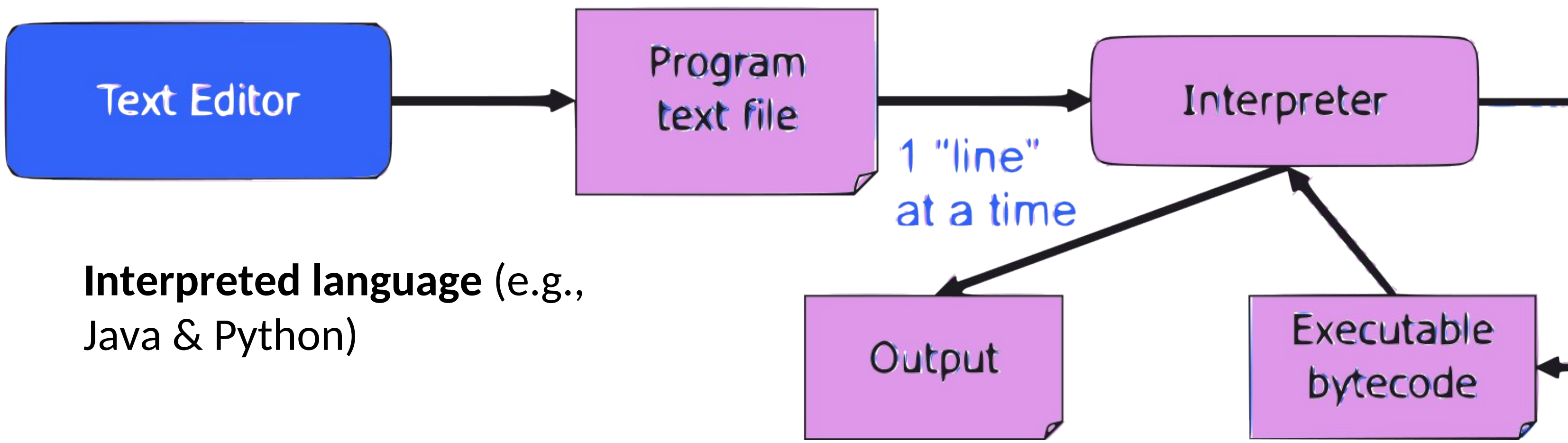
Datapath: is where data is processed (fetch, decode, execute)

Control unit: controls the cpu behavior

From HLL to machine code?

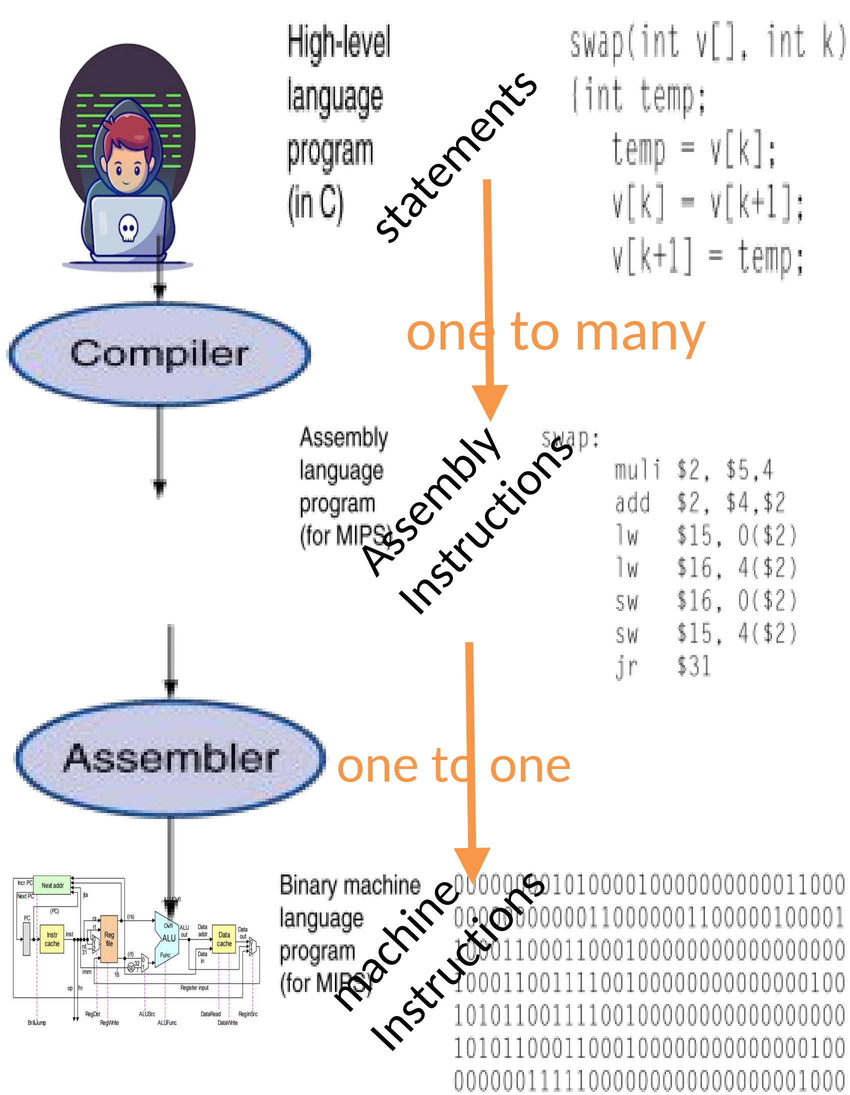


Compiled languages (e.g., c, C++, Rust,)



Interpreted language (e.g., Java & Python)

Code transformation



- Statements forming the needed logic to process the input (from user or files) and generate output (to screen or files)

Instruction Elements

- **Operation**: arithmetic, logic, memory, ..
- **Operands:** data in registers and memory
- Binary instructions encoding the operation and operands
- CPU decode the instructions and execute them to generate the desired output.

Machine vs Assembly instructions

Machine instructions are the "**BINARY**" commands that tell CPU (Central Processing Unit) what operations to perform, such as arithmetic operations, data movement, and control flow management.

Assembly Instructions a more human-readable version of machine instructions of a particular CPU.

- Each assembly language command corresponds directly to a machine instruction.
- Assembly language uses mnemonic codes (like `MOV`, `ADD`, `SUB`, etc.) to represent the binary machine instructions.

```
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))
sum = num1 + num2
print("The sum is:", sum)`
```

```
.data
prompt1: .asciiz "Enter the first integer: "
prompt2: .asciiz "Enter the second integer: "
result_msg: .asciiz "The sum is: "
.text
.globl main
main:
    li $v0, 4
    la $a0, prompt1
    syscall
    li $v0, 5
    syscall
    move $t0, $v0
    li $v0, 4
    la $a0, prompt2
    syscall
    li $v0, 5
    syscall
    move $t1, $v0
    add $t2, $t0, $t1
    li $v0, 4
    la $a0, result_msg
    syscall
    li $v0, 1
    move $a0, $t2
    syscall
    li $v0, 10
    syscall
```

Other processors will have different assembly code

Why so many lines?

- *Because the processor should be instructed to perform a single **atomic** operation at a time (CPU or memory read/write)*

```
num1 = int(input("Enter the first integer: "))
```

1. store the prompt in the memory
2. print the prompt to the user
3. read the user input from keyboard
4. we need to have it as integer
5. store this input in memory (num1)

It is important to note that I/O operations are abstracted through the OS

```python
num1 = int(input("Enter the first integer: "))
num2 = int(input("Enter the second integer: "))
sum = num1 + num2
print("The sum is:", sum)`
```

```
.data
prompt1: .asciiz "Enter the first integer: "
prompt2: .asciiz "Enter the second integer: "
result_msg: .asciiz "The sum is: "
.text
.globl main
main:
        li $v0, 4
        la $a0, prompt1
        syscall
        li $v0, 5
        syscall
        move $t0, $v0
        li $v0, 4
        la $a0, prompt2
        syscall
        li $v0, 5
        syscall
        move $t1, $v0
        add $t2, $t0, $t1
        li $v0, 4
        la $a0, result_msg
        syscall
        li $v0, 1
        move $a0, $t2
        syscall
        li $v0, 10
        syscall
```

Print fixed messeges (prompt and result)

Getting user input

Printing the sum

Why is it not easy to interpret?

- *Because assembly is **HW-specific***

- *the code contains implied information about the processor like register names ($t0, $t1, …, $v0 , $a0, ..)*

- *The instructions are defined by the **Instruction Set Architecture** of the target processor → different programs for Intel or ARM processors*

Commenting is an essential practice!

```
# Prompt user for the first integer
li $v0, 4         # Print string syscall code
la $a0, prompt1 # Load the address of the prompt string
syscall
# Read the first integer from the user
li $v0, 5         # Read integer syscall code
syscall
move $t0, $v0     # Store the first integer in $t0
# Prompt user for the second integer
li $v0, 4         # Print string syscall code
la $a0, prompt2 # Load the address of the prompt string
syscall
# Read the second integer from the user
li $v0, 5         # Read integer syscall code
syscall
move $t1, $v0     # Store the second integer in $t1
```

Commenting is an essential practice!

```
# Calculate the sum
add $t2, $t0, $t1
# Print the result message
li $v0, 4           # Print string syscall code
la $a0, result_msg # Load the address of the result message
syscall
# Print the sum
li $v0, 1           # Print integer syscall code
move $a0, $t2       # Load the sum into $a0
syscall
# Exit the program
li $v0, 10          # Exit syscall code
syscall
```

HLL vs Assembly

Feature	High-Level Language	Assembly Language
Abstraction Level	High	Low
Readability	Easy to understand	Difficult to read
Portability	High	Low
Efficiency	Generally slower	Generally faster
Complexity	Simpler to learn	complex to learn
Control	Less control over hardware	More control over hardware

What happens when you execute a binary (executable) file?

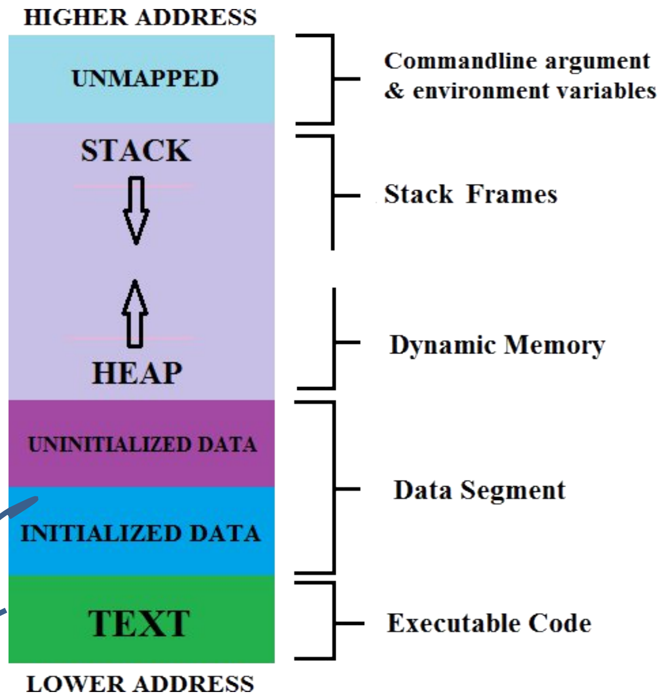
First: Binary File Format [Unix]

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information

- **Header** describes the size and position of the other pieces of the file
- **Text segment** contains the machine language code for routines in the source file (.text section in assembly)
- **Data segment** contains a binary representation of the data in the source file (.data section in assembly)
- **Relocation information** identifies instructions and data words that depend on absolute addresses
- **Symbol table** associates addresses with external labels in the source file and lists unresolved references
- **Debugging information** contains a concise description of the way in which the program was compiled

Binary file loading

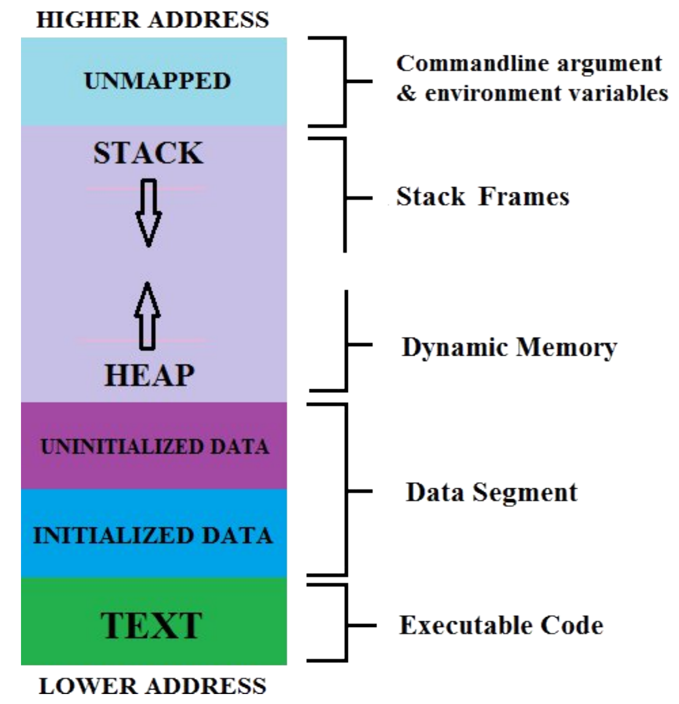
Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information



Binary file loading

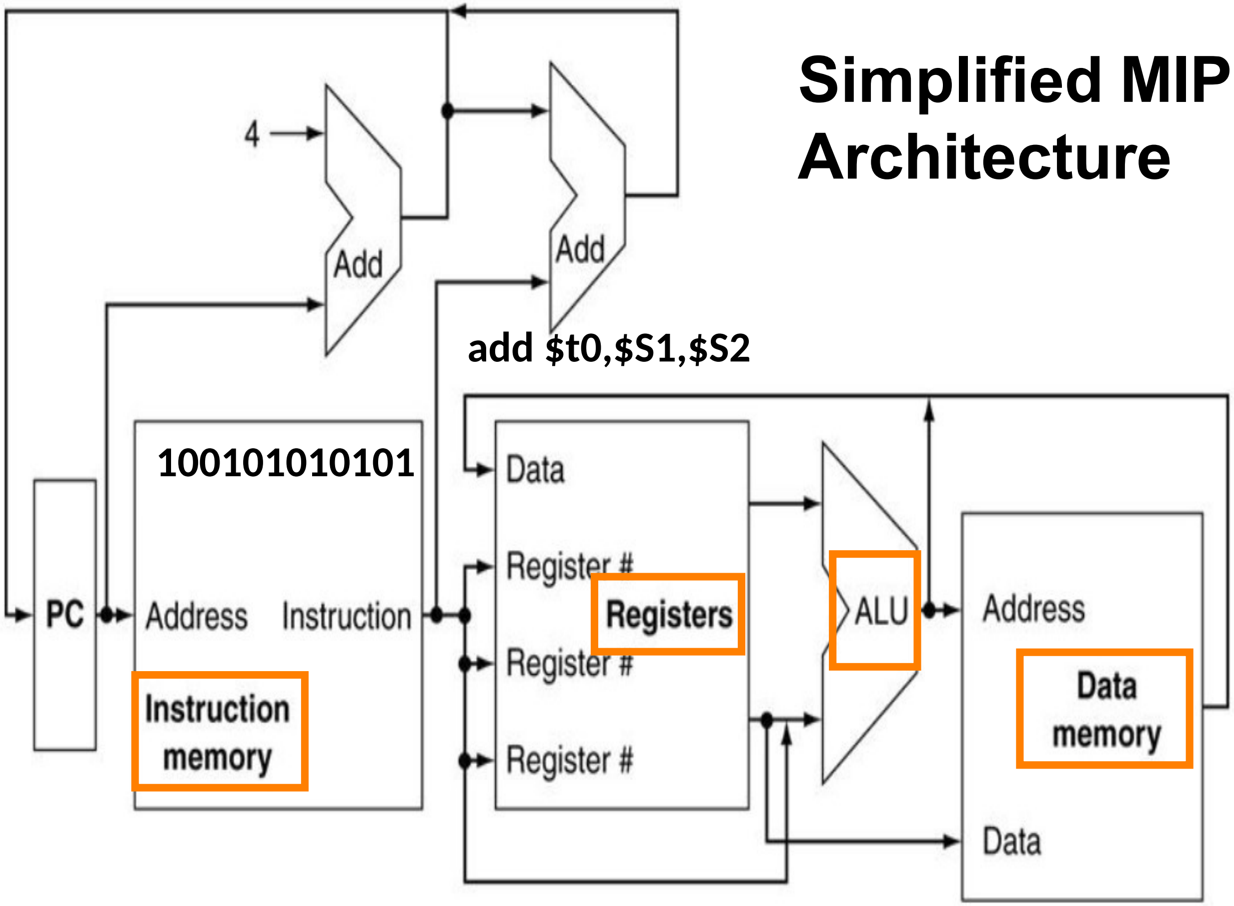
In Unix, the **operating system** performs the following steps

- **Reads** the file's header to determine the size of the text and data segments
- **Creates** a **<u>new address space</u>** for the program
- **Copies** **text** and **data** to respective memories
- **Copies** passed program arguments to the stack
- **Initializes** the **CPU registers** (IP, SP, ..)
- **Runs** the program's first instruction



Machine Language Execution

Simplified MIPS Architecture



- We will use Stanford MIPS commercialized by MIPS Technologies (www.mips.com)

MARS: (*MIPS Assembler and Runtime Simulator*)

MARS is a lightweight interactive development environment (IDE)