

Computer Architecture



Sections 1.1 - 1.4
Section 1.5 (optional)



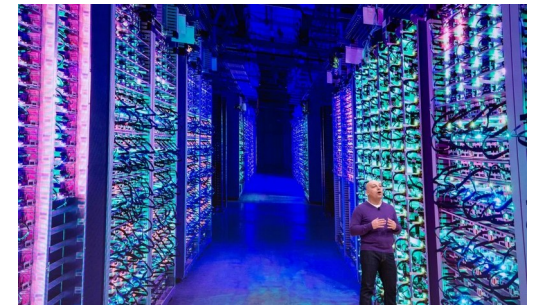
1



What is computer architecture?

Computers are designed to serve different purposes.

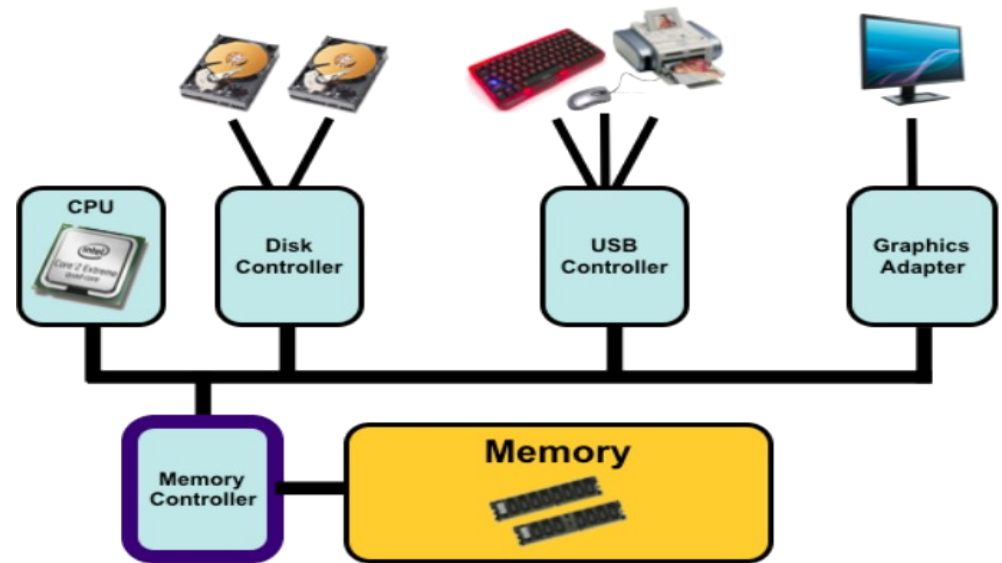
The application domain defines the computer design goals that could be a combination of **performance**, **cost**, **energy consumption**, and many other aspects



Computer Components

- Same components for all computer

- 1) Inputs/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

Computer Architecture

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

Technology
Circuit, packaging,
memory, ...

Domains
PMD, server, game
consoles, ...

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



Key Concepts: Implementation Abstraction

Implementation: A SW Analogy

- Write a function that gets an array and return it sorted



Different implementations

Insertion

Quic

Bubble

...

HW Implementation

- Component Function



- These **components** can be realized using different **HW implementations**
 - Faster vs slower memory (same capacity)
 - CPUs (speed, supported data types, ..)
 - Single clock cycle, pipelined, parallel
 - **Selection?** depends on many factors such as usage, cost, and energy efficiency

Abstraction

Abstraction: a key engineering design methodology when dealing with **complex** systems (problems)

- **Approach**: Divide the beast into multiple parts, each with **Interface**: knob(s) (API in SW)

Implementation: “black box”

- Only **specialists** deal with box internals (implementation), the rest of us with interface

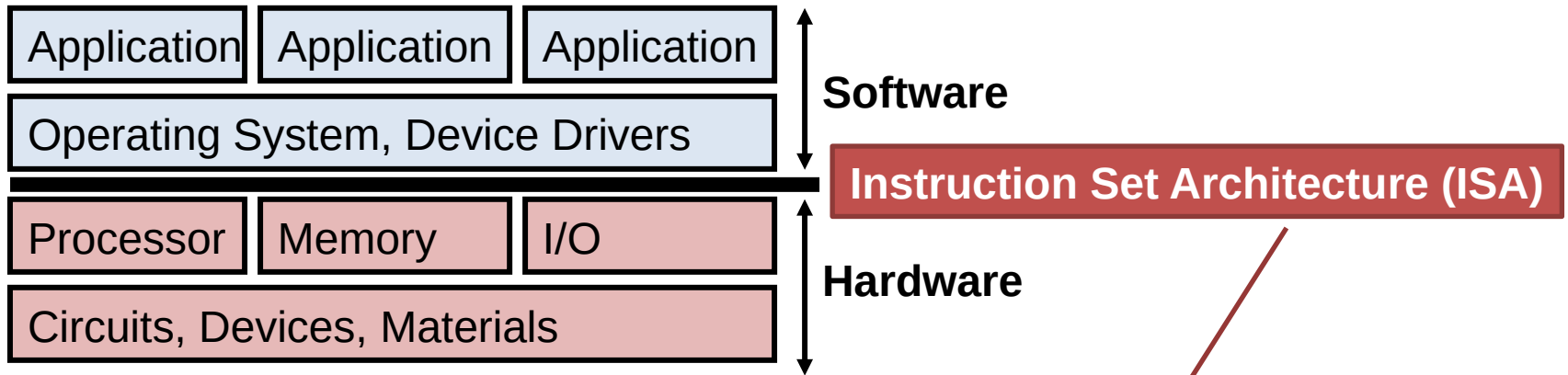
- **Examples**:

SW abstraction (Classes OOP)

Network Abstraction (TCP/IP Layered model)



Instruction Set Architecture



- **Definition:** *is the key interface between the HW and low-level software (i.e., Assembly instructions to be executed on a specific processor)*
- *Defined by a set of instructions that can be executed on the underlying HW*
- *enables many **implementations** of varying cost and performance for the underlying HW*

Key Points

- Any computer (functional) component could have different ***implementations*** with diverse costs and performance
- ***Abstraction*** is an intrinsic principle in hardware and software design
- The ***instruction set architecture*** is the key interface between the hardware and low-level software

What happens when you run your code?

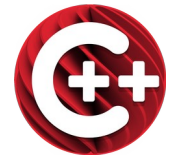


What is this code doing?

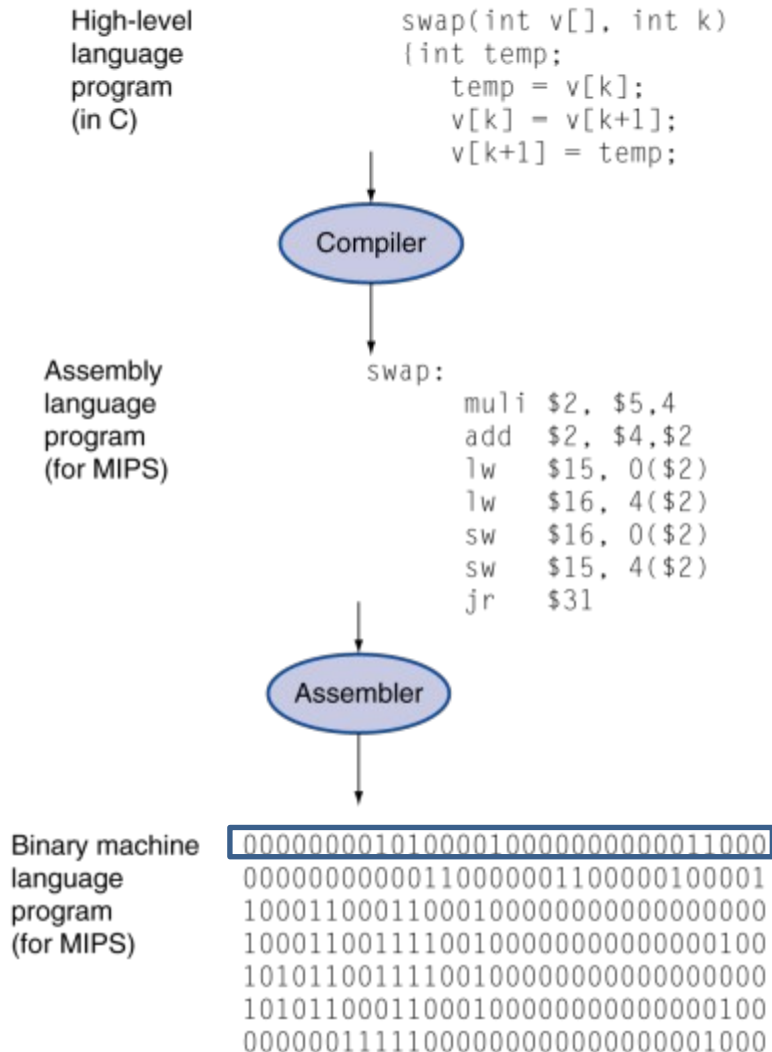


```
num1 = int(input("Enter the first integer: ")) python™
num2 = int(input("Enter the second integer: "))
sum = num1 + num2
print("The sum is: ", sum)
```

```
#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;
}
```



How is this code executed?



- *High-level language*
 - Level of abstraction closer to problem domain
 - Provides for productivity and *portability* (can be used on different machines)
- *Assembly language*
 - Textual representation of **ISA** instructions
 - Different for various architectures (x86, ARM, **MIPS**)
- *Machine language*
 - Binary digits (bits)
 - **Encoded** instructions and data

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



```
1  .data
2  prompt1: .ascii "Enter the first integer: "
3  prompt2: .ascii "Enter the second integer: "
4  result_msg: .ascii "The sum is: "
5  .text
6  .globl main
7  main:
8      li $v0, 4
9      la $a0, prompt1
10     syscall
11     li $v0, 5
12     syscall
13     move $t0, $v0
14     li $v0, 4
15     la $a0, prompt2
16     syscall
17     li $v0, 5
18     syscall
```

```
19     move $t1, $v0
20     add $t2, $t0, $t1
21     li $v0, 4
22     la $a0, result_msg
23     syscall
24     li $v0, 1
25     move $a0, $t2
26     syscall
27     li $v0, 10
28     syscall
```

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

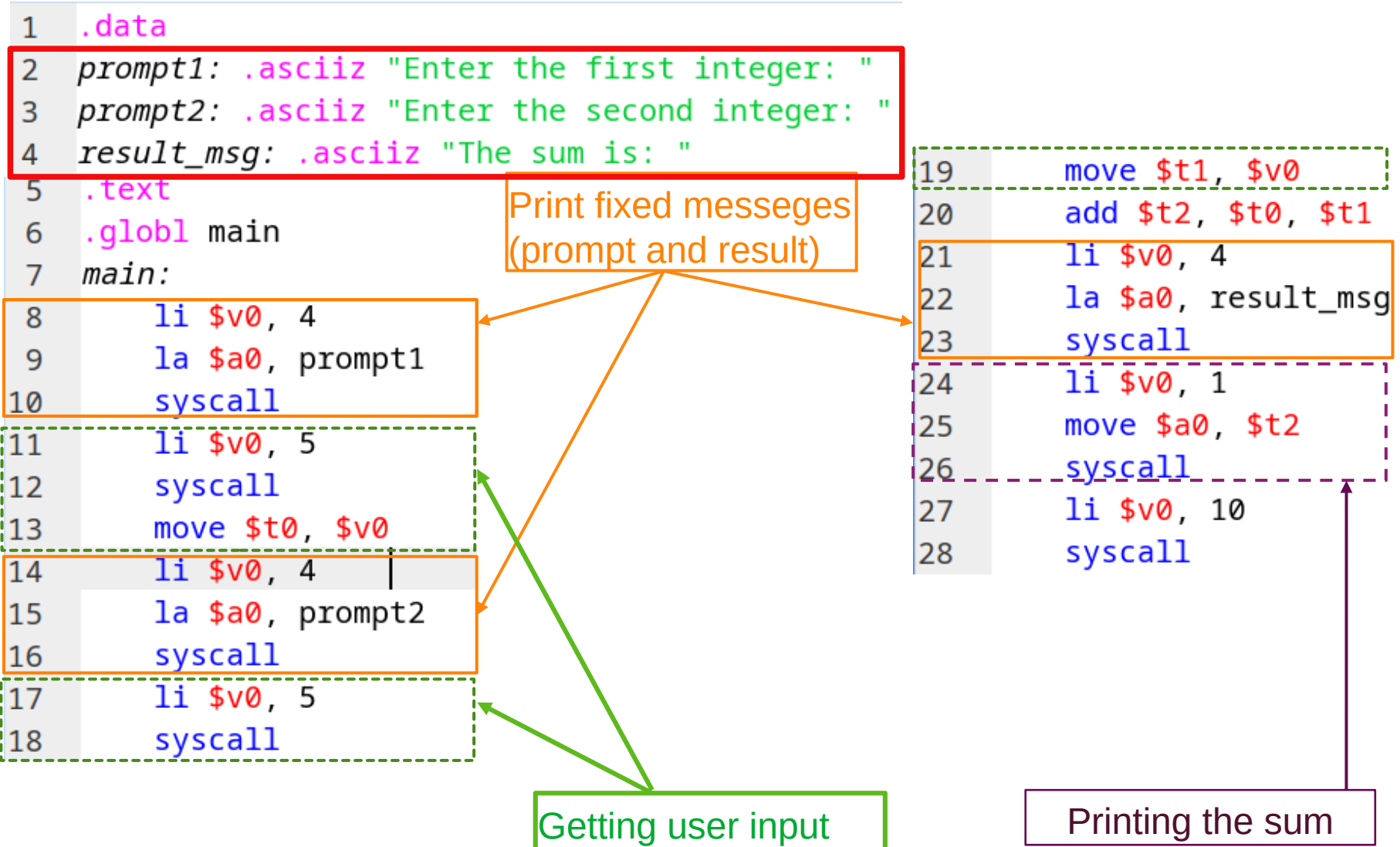
```
num1 = int(input("Enter the first integer: "))
```

- store the prompt in the memory
- print the prompt to the user
- read the user input from keyboard
- we need to have it as integer
- store this input in num1

It is important to note that I/O operations are abstracted through the OS (more later in the lab)

- **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 ISA of the target processor → different program for Intel or ARM processors*

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



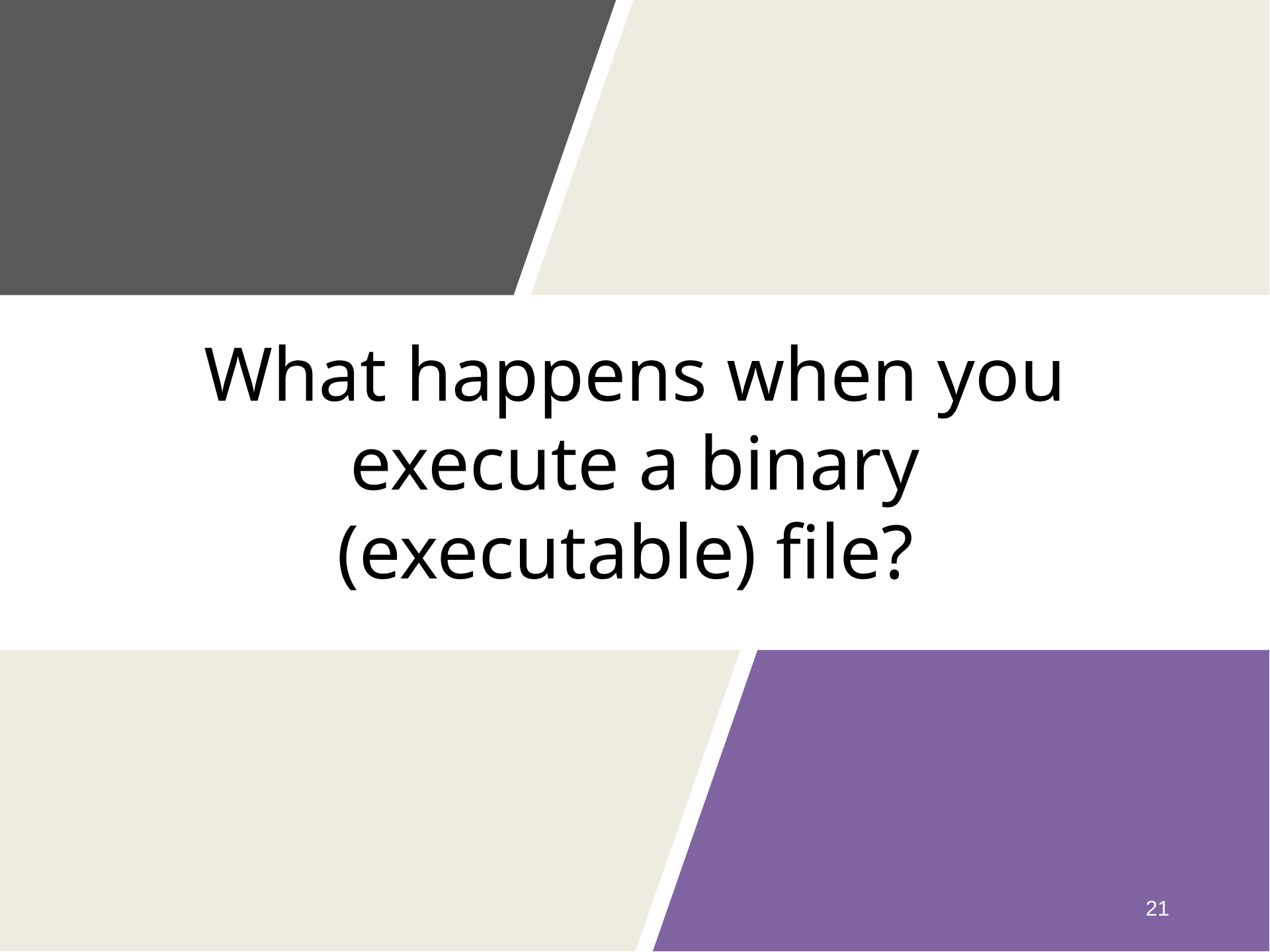
- **Commenting is an essential practice!**

```
# 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
```



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

Binary File Format [Unix]

Object file
header

Text
segment

Data
segment

Relocation
information

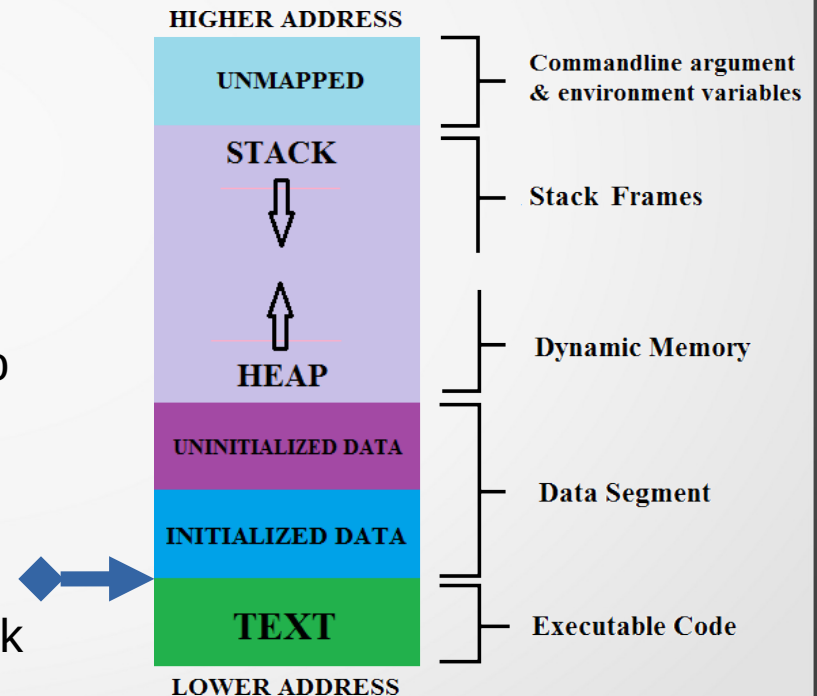
Symbol
table

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)
- **Data segment** contains a binary representation of the data in the source file (.data section)
- **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

- 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 new address space for the program
 - } Copies text and data to respective memories
 - } Copies passed program arguments to the stack
 - } initializes the machine registers
 - } Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's main routine



Execution in MARS

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001 lui \$1, 4097	20: lw \$t0, aSide	
	0x00400004	0x8c280000 lw \$8, 0(\$1)		
	0x00400008	0x3c011001 lui \$1, 4097	21: lw \$t1, bSide	
	0x0040000c	0x8c290004 lw \$9, 4(\$1)		
	0x00400010	0x3c011001 lui \$1, 4097	22: lw \$t2, cSide	
	0x00400014	0x8c2a0008 lw \$10, 8(\$1)		
	0x00400018	0x71095802 mul \$11, \$8, \$9	28: mul \$t3, \$t0, \$t1	
	0x0040001c	0x716a0002 mul \$12, \$11, \$10	29: mul \$t4, \$t3, \$t2	
	0x00400020	0x3c011001 lui \$1, 4097	30: sw \$t4, volume	
	0x00400024	0xac2c000c sw \$12, 12(\$1)		
	0x00400028	0x71095802 mul \$11, \$8, \$9	36: mul \$t3, \$t0, \$t1 # aSide * bSide	
	0x0040002c	0x710a6002 mul \$12, \$8, \$10	37: mul \$t4, \$t0, \$t2 # aSide * cSide	
	0x00400030	0x712a6802 mul \$13, \$9, \$10	38: mul \$t5, \$t1, \$t2 # bSide * cSide	
	0x00400034	0x016c7020 add \$14, \$11, \$12	39: add \$t6, \$t3, \$t4	
	0x00400038	0x01cd7820 add \$15, \$14, \$13	40: add \$t7, \$t6, \$t5	
	0x0040003c	0x20010002 addi \$1, \$0, 2	41: mul \$t7, \$t7, 2	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	73	14	16	0	0	0	0	0
0x10010020	0	0	0	0	0	0	0	0
0x10010040	0	0	0	0	0	0	0	0
0x10010060	0	0	0	0	0	0	0	0
0x10010080	0	0	0	0	0	0	0	0
0x100100a0	0	0	0	0	0	0	0	0
0x100100c0	0	0	0	0	0	0	0	0
0x100100e0	0	0	0	0	0	0	0	0
0x10010100	0	0	0	0	0	0	0	0
0x10010120	0	0	0	0	0	0	0	0
0x10010140	0	0	0	0	0	0	0	0
0x10010160	0	0	0	0	0	0	0	0
0x10010180	0	0	0	0	0	0	0	0
0x100101a0	0	0	0	0	0	0	0	0

0x10010000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

Registers

Name	Coproc 1	Coproc 0	Number	Value
\$zero			0	0
\$at			1	0
\$v0			2	0
\$v1			3	0
\$a0			4	0
\$a1			5	0
\$a2			6	0
\$a3			7	0
\$t0			8	0
\$t1			9	0
\$t2			10	0
\$t3			11	0
\$t4			12	0
\$t5			13	0
\$t6			14	0
\$t7			15	0
\$s0			16	0
\$s1			17	0
\$s2			18	0
\$s3			19	0
\$s4			20	0
\$s5			21	0
\$s6			22	0
\$s7			23	0
\$t8			24	0
\$t9			25	0
\$k0			26	0
\$k1			27	0
\$gp			28	268468224
\$sp			29	2147479548
\$fp			30	0
\$ra			31	0
\$pc				4194304
\$hi				0
\$lo				0

Mars Messages Run I/O

Clear