Computer Architecture

Sections 1.1 - 1.4
Section 1.5 (optional)

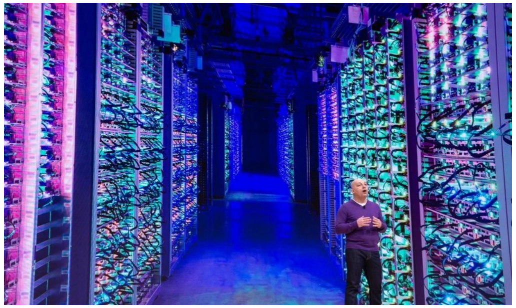
1

What is computer architecture?

THE BIG PICTURE

Computers are designed to serve different purposes.

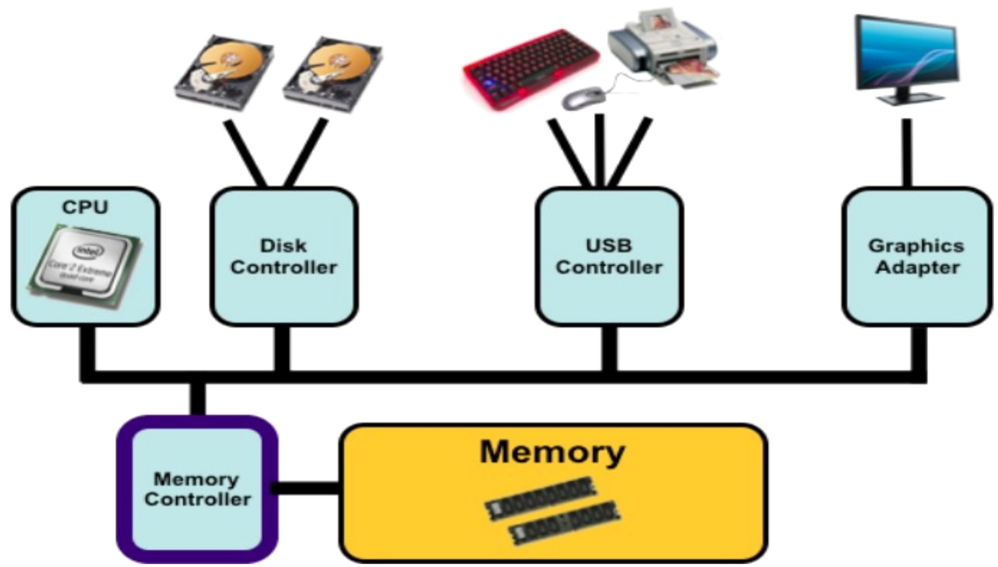
The application domain defines the computer design goals that could be a combination of **performance**, **cost**, energy consumption, and many other aspects



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

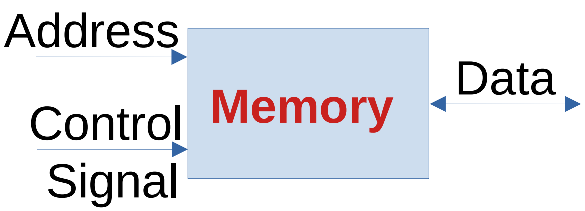
- Write a function that gets an array and return it sorted

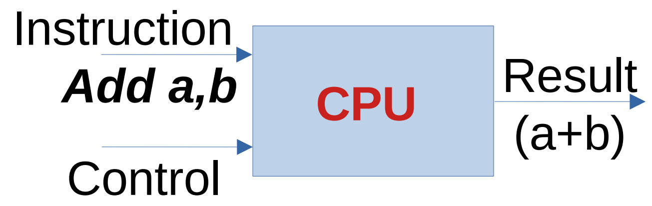
Different implementaitons

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

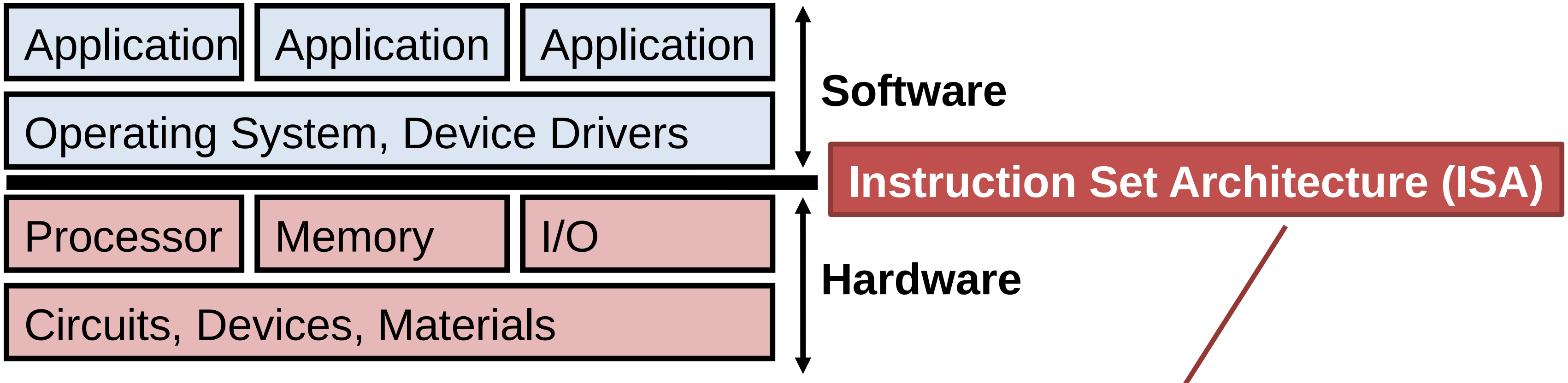
Abstraction: a key engineering design methodology when dealing with **complex** systems (problems)

- 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

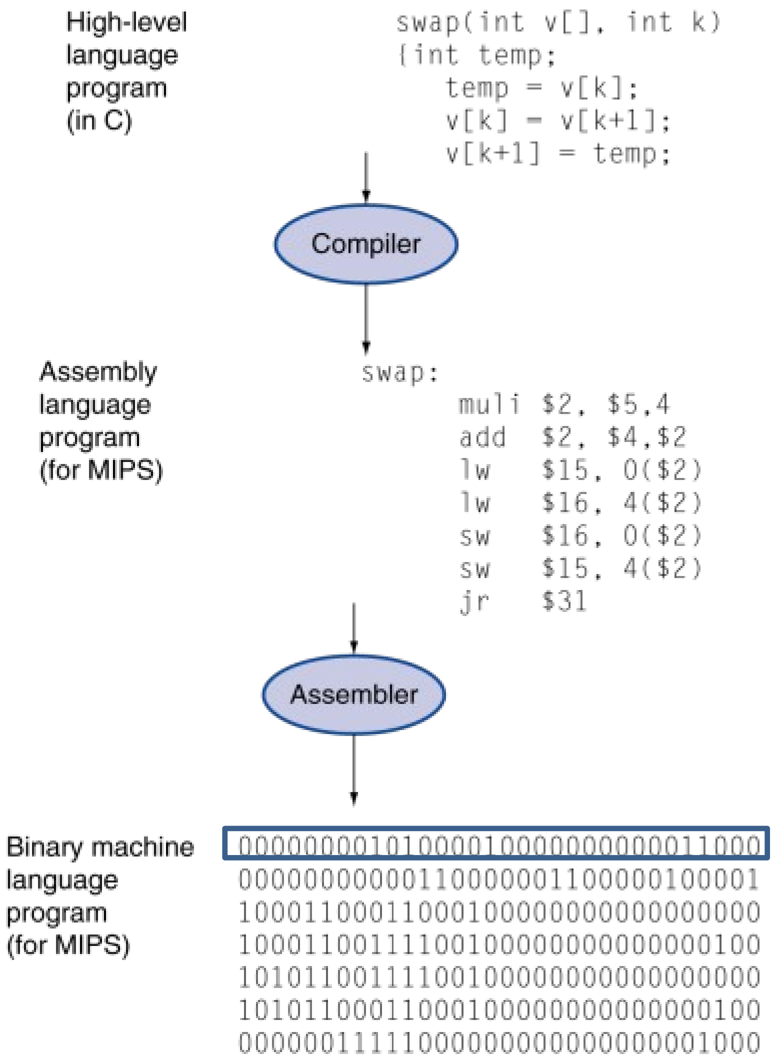
What happens when you run your code?

What is this code doing?

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

```cpp
#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

```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
```

- **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*

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

python

```
.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

- **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?

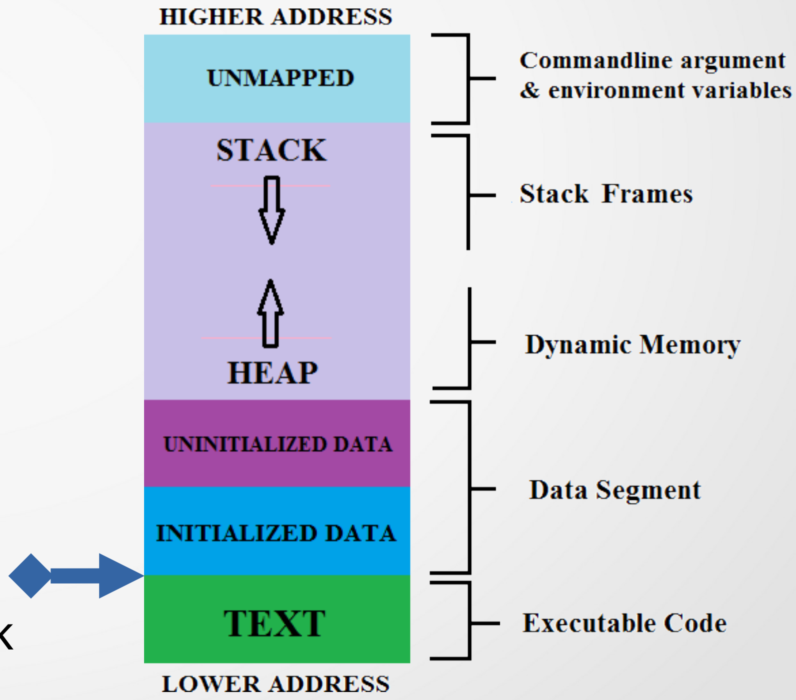
Binary File Format [Unix]

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