CSE251: System Programming

2. Basics of C and Computer Systems

Seongil Wi

UNIST
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY
2 0 0 7

Department of Computer Science and Engineering

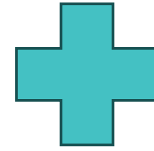
Notification: Your Playground

- Did you receive the server account information from the TA?
- Access the server using SSH.
- Become familiar with using the terminal (shell commands).
  - All assignments and activities should be done on your server.

Recap: Overview

You will learn **how hardware + software combine** to support the execution of **application programs**!

Processors, Memories, Disks, I/O devices…

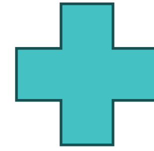


Operating systems, Compilers, Libraries, …

You will learn **how hardware + software combine** to <u>support the execution of</u> **<u>application programs</u>**!

Web Browser, PowerPoint, …

Processors, Memories, Disks, I/O devices…



Operating systems, Compilers, Libraries, …

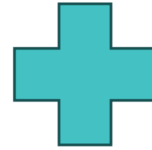
Hardware

System Software

In particular, you will learn **how to use these resources** effectively as a programmer

Processors, Memories, Disks, I/O devices…

Hardware



Operating systems, Compilers, Libraries, …

System Software

In particular, you will learn **how to use these resources** effectively as a programmer

How to directly access these resources?

By using C language!

C Programming Basics

Overview of C

- Developed in the early 1970s by Dennis Ritchie at Bell Labs
  - C is used for developing the UNIX operating system

- A low-level language (closer to the hardware)
- Most operating system kernels are written in C

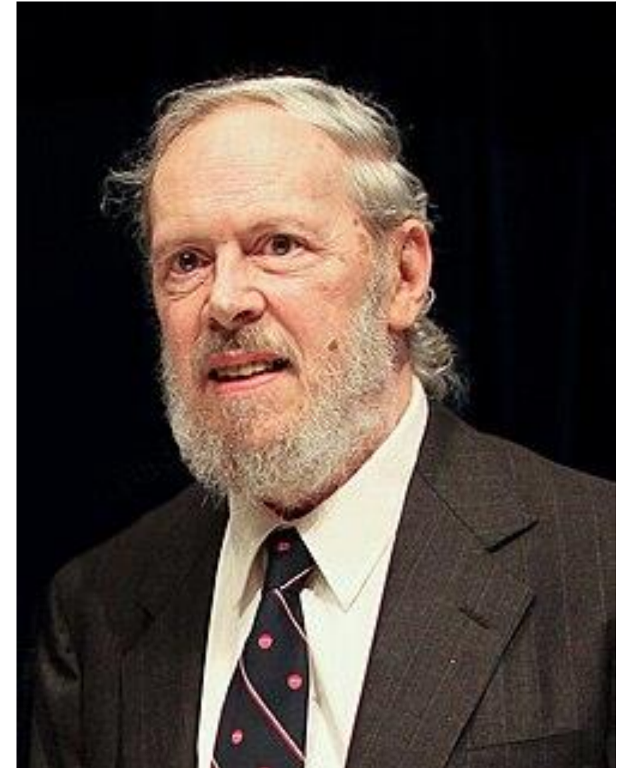


Image from https://en.wikipedia.org/wiki/Dennis_Ritchie

Why C?

There are dozens of programming languages.
Why C?

Why C?

1. Allow direct interactions with the hardware and system software
   - Provides direct memory/register access
   - Provides pointer arithmetic
   - Provides low-level input/output functions

2. Programs written in C are efficient and execute much faster

Warning!

- This is not a C programming course
- We will not cover C syntax, so study it on your own

Our Scope

- Today, we will study basic I/O functions and pointers

Hello World Demo

```
#include <stdio.h>

int main() {
    printf(“Hello world!\n”);
    return 0;
}
```

test.c

```
$ gcc test.c -o test
$ ./test
Hello world!
```

Compilation

Execution

Hello World Demo

```
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

test.c

Standard input and output header

Hello World Demo

```
#include <stdio.h>

int main() {
    printf(“Hello world!\n”);
    return 0;
}
```

test.c

Standard input and output header

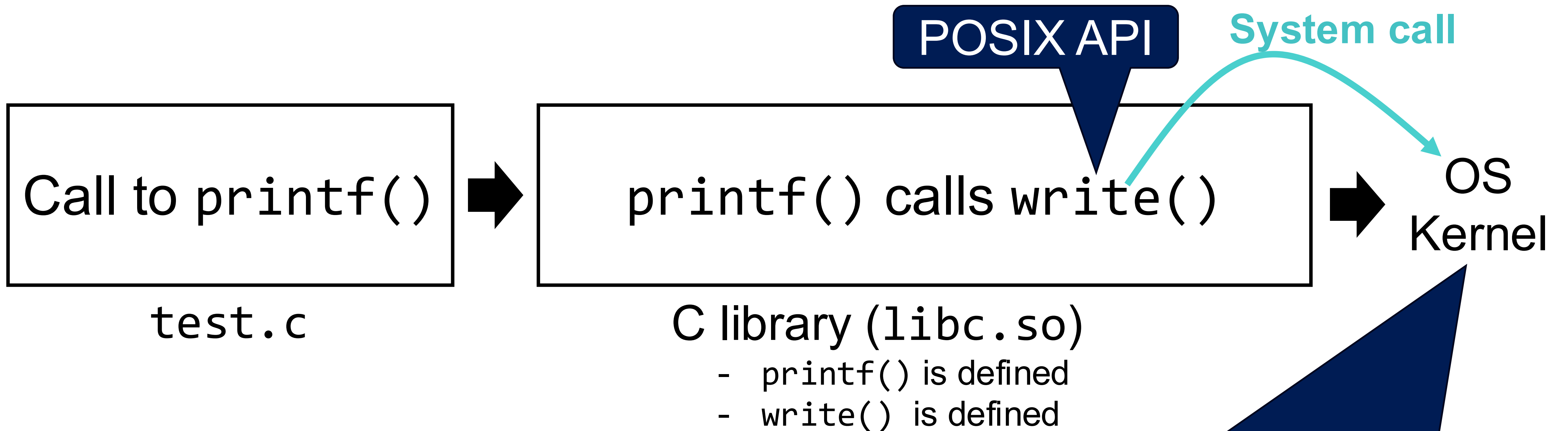
Basic I/O APIs

- C provide a wide variety of input and output (I/O) facilities

- These APIs are defined in the (located in `/usr/include/stdio.h`) header `stdio.h`
  - only contains the header information for the `printf` function

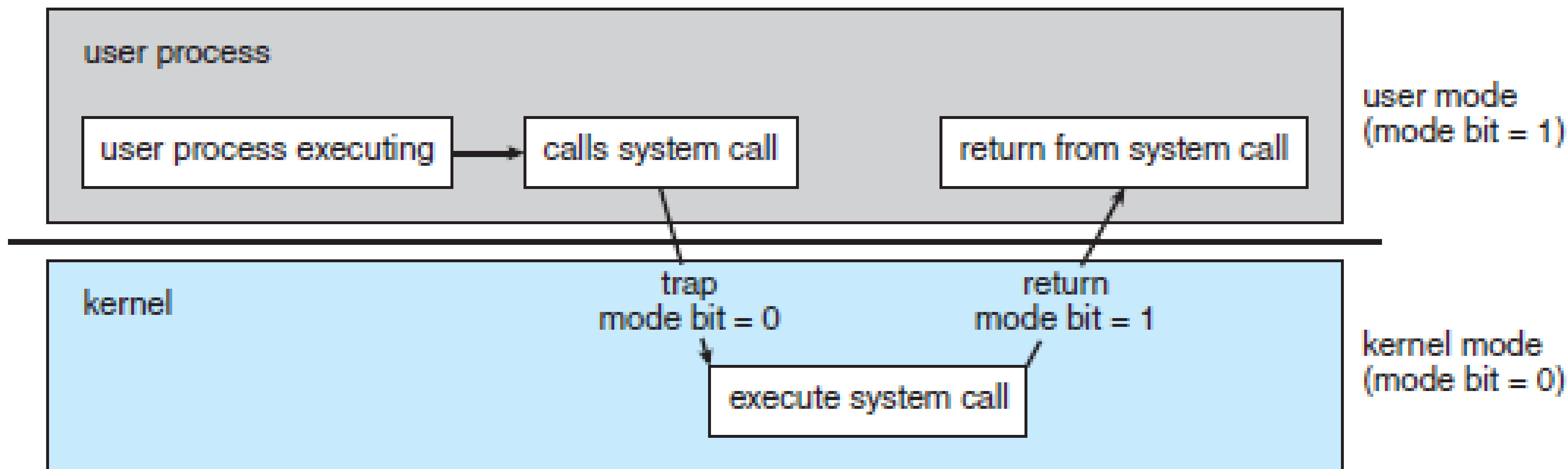
- The actual `printf` function is defined in the C library (`libc.so`)

(FYI) Library Calls Workflow



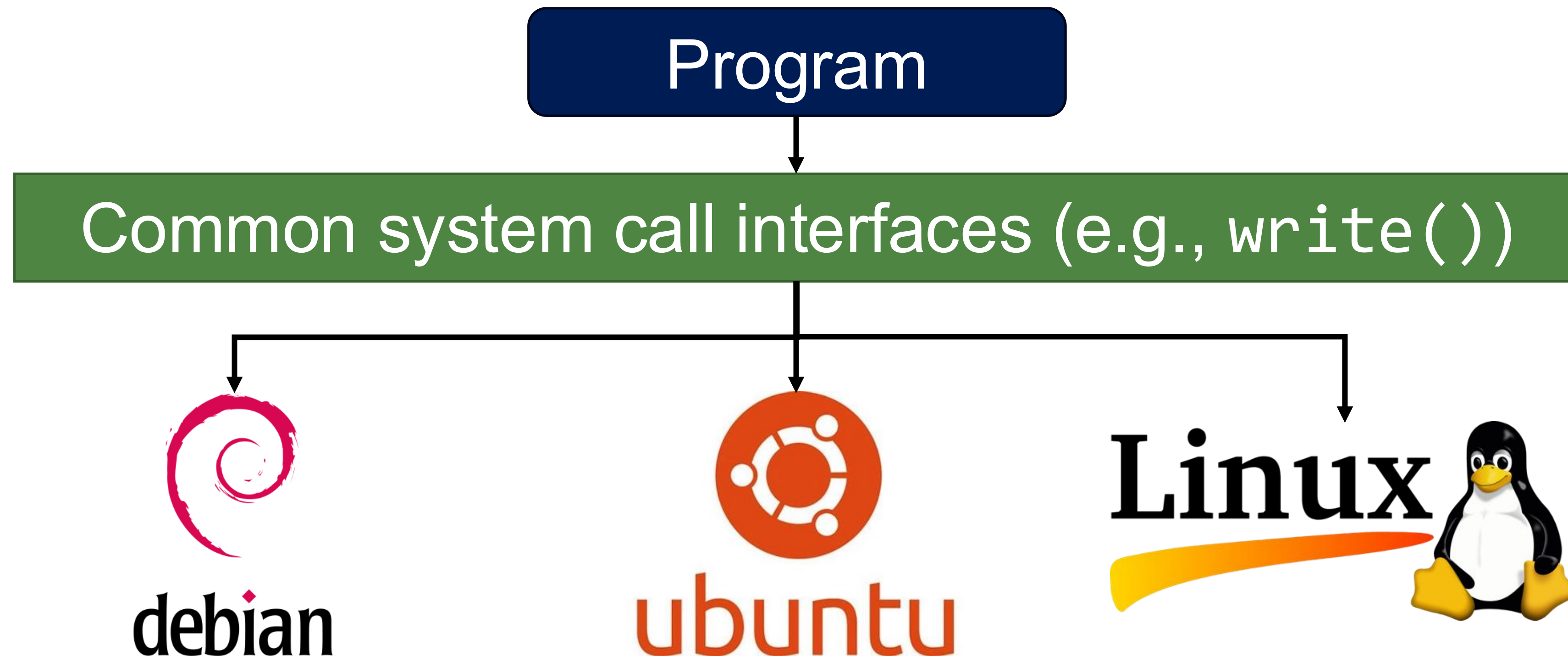
System Call

- The programmatic way in which a computer program requests a service from the operating system



Portable Operating System Interface (POSIX)

- A family of standards that define **interfaces (e.g., system call interfaces, shell interfaces, file formats, …)** for UNIX-like operating systems
  - Designed to ensure portability and compatibility across operating systems
  - Developed by the IEEE



An Operating System Family Tree

MS-DOS systems

Windows 9.x kernel

Microsoft Windows 95

Microsoft Windows 98

Windows NT kernel

Windows 2000

to

Windows 10

"Classic" Mac systems

Mac System Software 1.0 to Mac OS 9

UNIX

UNIX kernel

Linux kernel + GNU software

debian

ubuntu

Linux Mint

fedora

redhat.

Scientific Linux

Gentoo

CentOS

Exec kernel

AMIGA OS

QNX kernel

QNX

Mach kernel

BSD kernel

XNU kernel

android

openSUSE

archlinux

Chrome OS

SunOS

solaris

OpenBSD

FreeBSD

DragonFlyBSD

NetBSD

DarwinOS

OpenDarwin

PureDarwin

OSX/macOS

iOS

Designed by Ethan Gates
10/2018

POSIX Defines

- An API for interacting with the OS kernel
  - E.g., File operations, Processes, threads, shared memory, Memory management, Network
- General Concepts (e.g., added rules for writing programs, like safety for the initialization of pointer types and concurrent executions)
- File formats (e.g., rules for formatting strings)
- Environment Variables (e.g., reserved environment variables)
- Directory structure (e.g., /tmp directory to create temporary files and directories)
- …

POSIX API

- This API is a set of C language functions and variables
- On a UNIX system, the functions are mostly system calls
- System calls are requests to the OS kernel

Example POSIX APIs

- open(): Opens a file for reading or writing
- fork(): Creates a new process
- connect(): Creates a network connection
- exit(): Gracefully terminates the current process
- tcsetattr(): Configures a serial (or virtual) terminal
- time(): Get the current time
- Write(): write to a file descriptor
- …

Basic I/O APIs

- There are several functions provided for basic I/O:
  - **`puts`**`(str)`: prints a string to `stdout` with a trailing newline
  - **`fputs`**`(str, stream)`: prints a string to a specified stream
  - **`printf`**`(format, …)`: prints a string to `stdout`, providing sophisticated formatting capabilities
  - **`fprintf`**`(stream, format, …)`: prints a string to stream, providing sophisticated formatting capabilities
  - **`gets`**`()`: this function is dangerous, do not use it.
  - **`fgets`**`(str, size, stream)`: read a single line of text from the specified stream, but no more than size - 1 bytes.
  - **`fscanf`**`(stream, format, …)`: read complex formatted data from the specified stream

Basic I/O APIs – Example

```
char buffer[100];
printf("Enter your name: ");
gets(buffer);
printf("Hello, %s!\n", buffer);
```

Execution result:

```
> Enter your name: Seongil Wi
> Hello, Seongil Wi!
```

Basic I/O APIs – Example

```
#include <stdio.h>

int main() {

  double pi = 3.14159265359;
  int age = 25;
  char name[] = "Alice";

  printf("My name is %s and I am %d years old.\n", name, age);
  printf("The value of pi is approximately %.2f\n", pi);
  printf("My name is %s, I am %d years old, and pi is %.5f\n",
                                                   name, age, pi);

  return 0;
}
```

Pointer

- A variable that stores the **memory address** of another variable

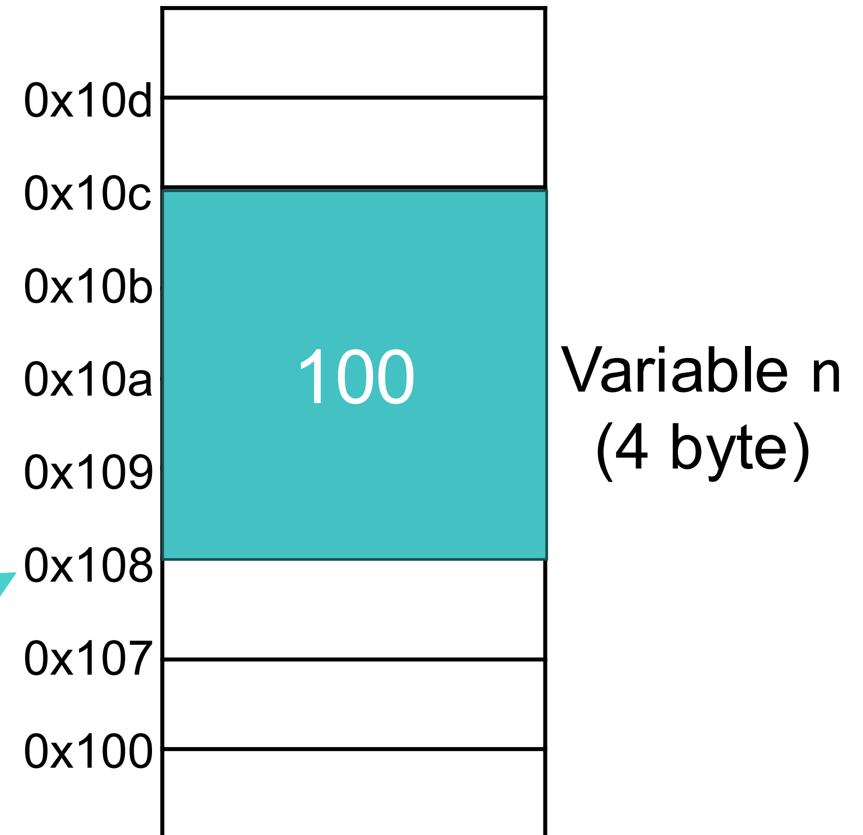
```
int n = 100;
int *ptr = &n;
```

A pointer variable that stores the address of n

Get the memory address of n

0x108

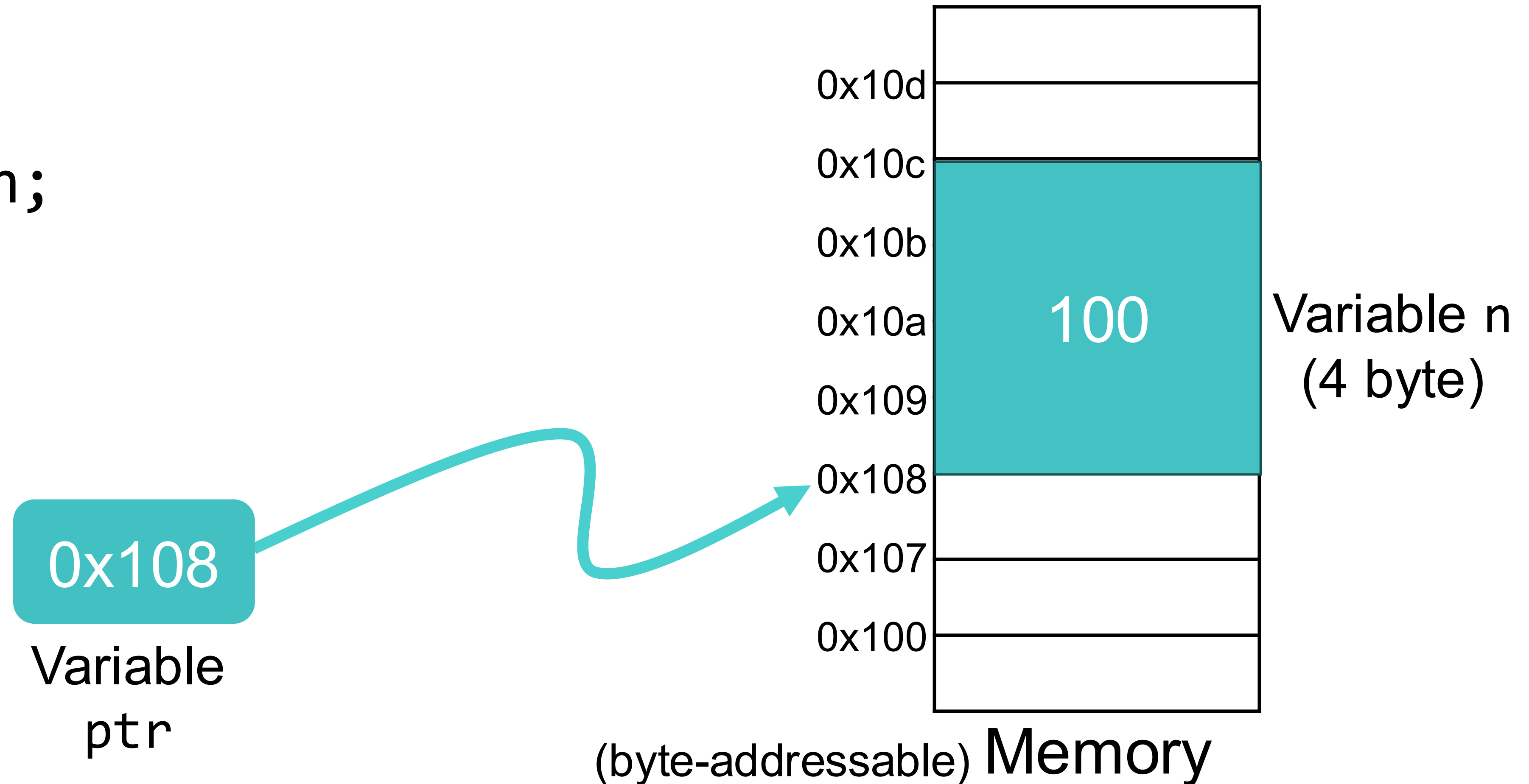
Variable ptr



Pointer Syntax – Declaration

- Associates a type with the manipulated memory

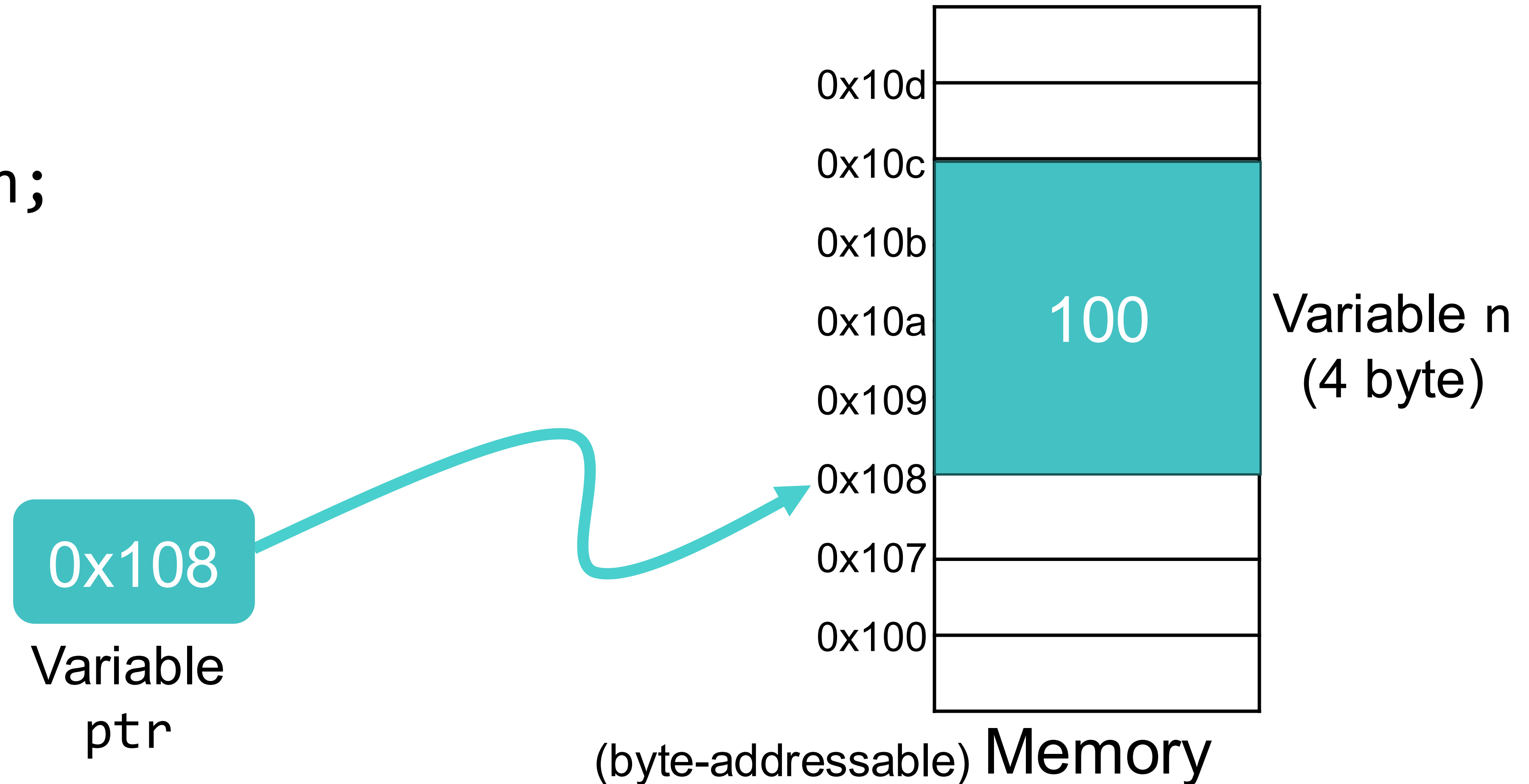
```
int n = 100;
int *ptr = &n;
```



Pointer Syntax – Declaration

- Associates a type with the manipulated memory
- In declaration, a pointer variable is marked with *

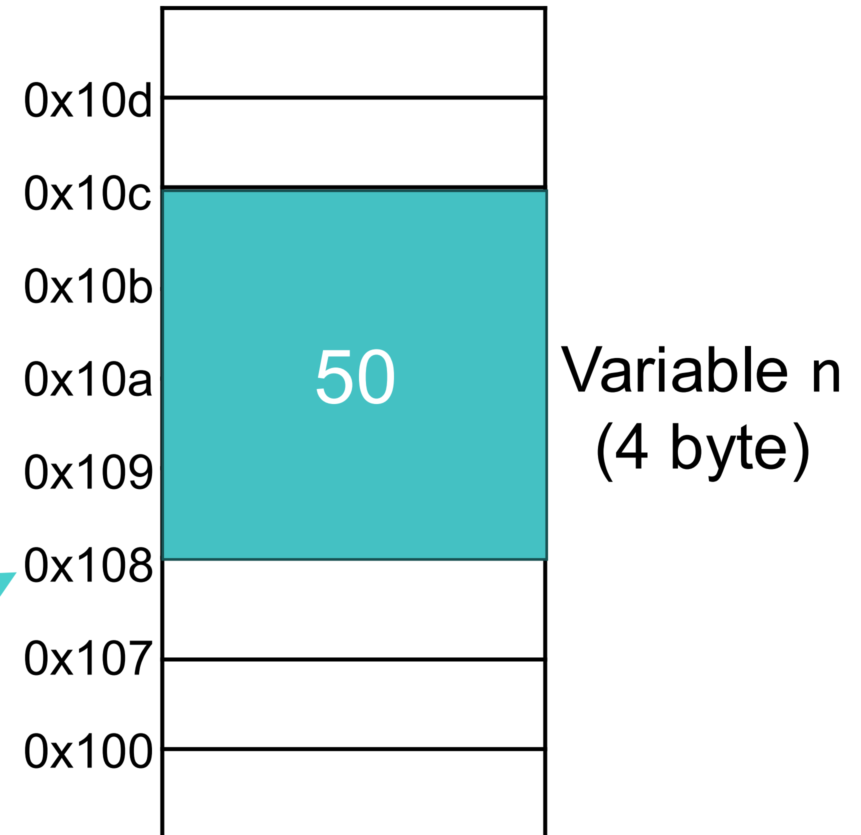
```
int n = 100;
int *ptr = &n;
```



Pointer Syntax – Dereferencing

- Access the value stored at the memory address that a pointer points to
  - A pointer is dereferenced with * , ->, or [ ]

```
int n = 100;
int *ptr = &n;
*ptr = 50
printf("%d", n); // 50
```



0x108

Variable
ptr

Arrays in C

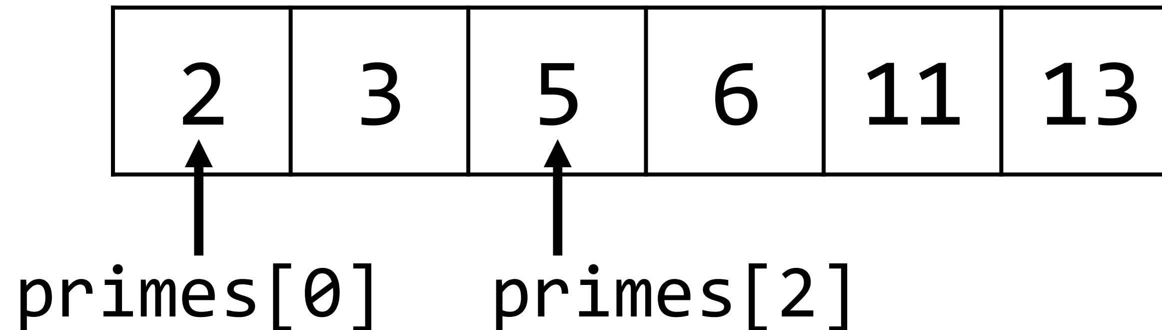
- Definition: `type name[size]`
  - Allocates `size*sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size` (e.g., `int score[175]`)
  - Initially, array values are “garbage”

Using C Arrays

- <u>Initialization</u>: `type name[size] = {val0, ..., valN};`
  - `{val0, ..., valN}` initialization can only be used at time of definition
  - If no `size` supplied, infers from length of array initializer

Using C Arrays – Examples

- `int primes[6] = {2, 3, 5, 6, 11, 13}`



A string in C is represented as an array of characters

Same with `{'H','e','l','l’,...,'\0'};`

- `char arr[] = “Hello World”;`

Using C Arrays

- Initialization: `type name[size] = {val0, ..., valN};`
  - `{val0, ..., valN}` initialization can only be used at time of definition
  - If no `size` supplied, infers from length of array initializer
- Array name used as identifier for "collection of data"
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7  // {2, 3, 5, 7, 11, 13}
premes[100] = 0; // Error: memory smash!
```

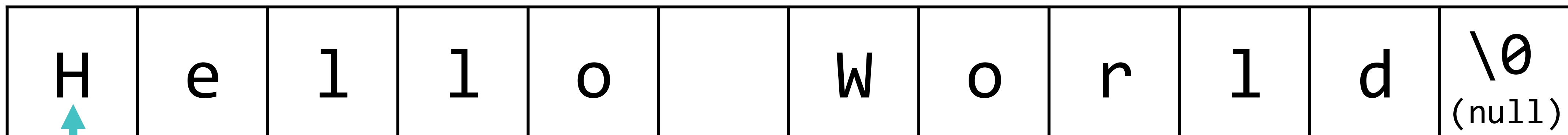
C Pointers and Arrays

- Arrays and pointers are closely related in C
  - Array name (by itself) produces the **address of the start of the array!**

C Pointers and Arrays

- Arrays and pointers are closely related in C
  - Array name (by itself) produces the **address of the start of the array!**

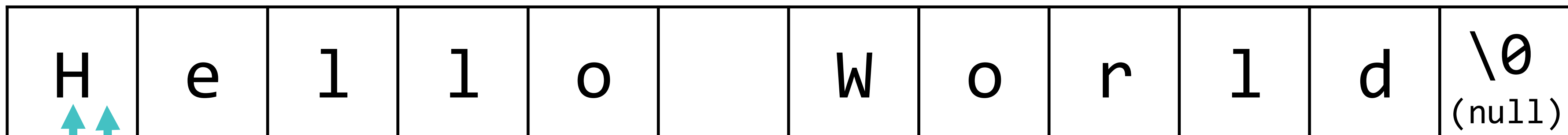
```
char arr[] = “Hello World”;
```



C Pointers and Arrays

- Arrays and pointers are closely related in C
  - Array name (by itself) produces the **address of the start of the array!**

```
char arr[] = “Hello World”;
char *ptr = arr    // &arr[0]
```



Address of arr[0]

Variable
`arr`

Address of arr[0]

Variable
`ptr`

C Pointers and Arrays

- Arrays and pointers are closely related in C
  - Array name (by itself) produces the **address of the start of the array!**

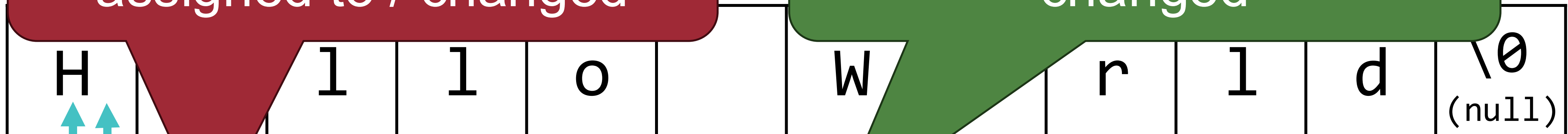
```
char arr[] = “Hello World”;
char *ptr = arr    // &arr[0]
```

H	e	l	l	o		W	o	r	l	d	\0 (null)

(FYI) Arrays are Not Pointers

- Arrays and pointers are closely related in C
  - Array name (by itself) produces the **address of the start of the array!**

```
char arr[] = "Hello World";
```



(FYI) Arrays are Not Pointers

Array variable
→ address is fixed
→ cannot be assigned

```
char arr1[] = "Hello";
char arr2[] = "World";

arr1 = arr2;        // Error
arr1 = “World”      // Error
```

C Pointer Arithmetic

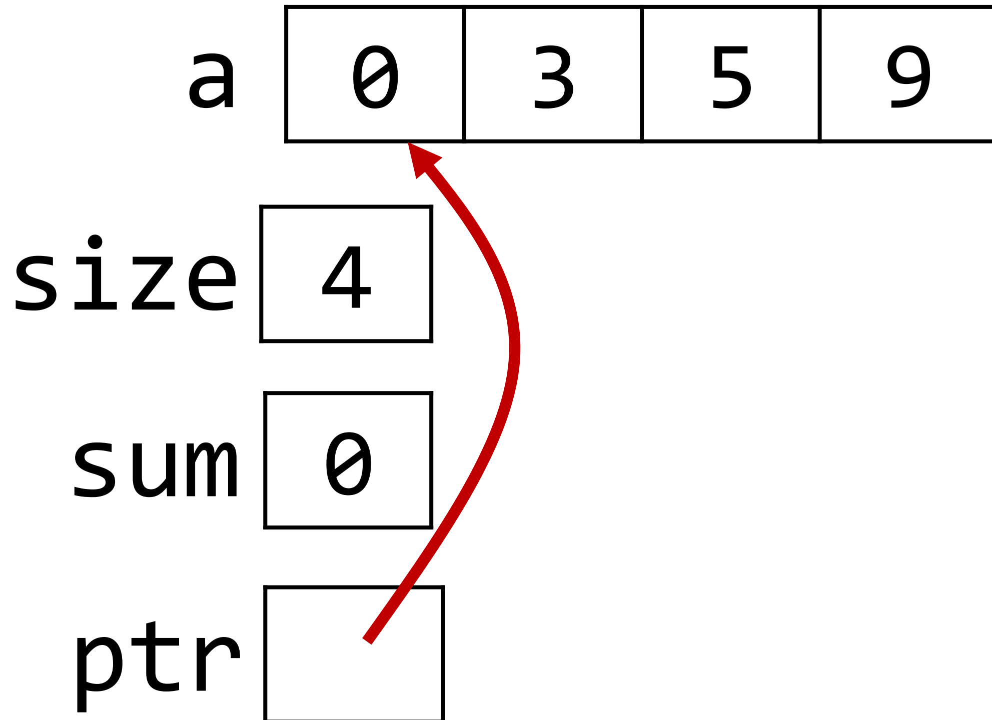
- We can do arithmetic on addresses to iterate through arrays

```c
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
for (int i = 0; i < size; i++) {
 sum += ptr[i];
}
```

```c
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
   sum += *ptr;
}
```

C Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays




```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
    sum += *ptr;
}
```

C Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays

a | 0 | 3 | 5 | 9 | (empty)

size: 4

sum: 0

ptr

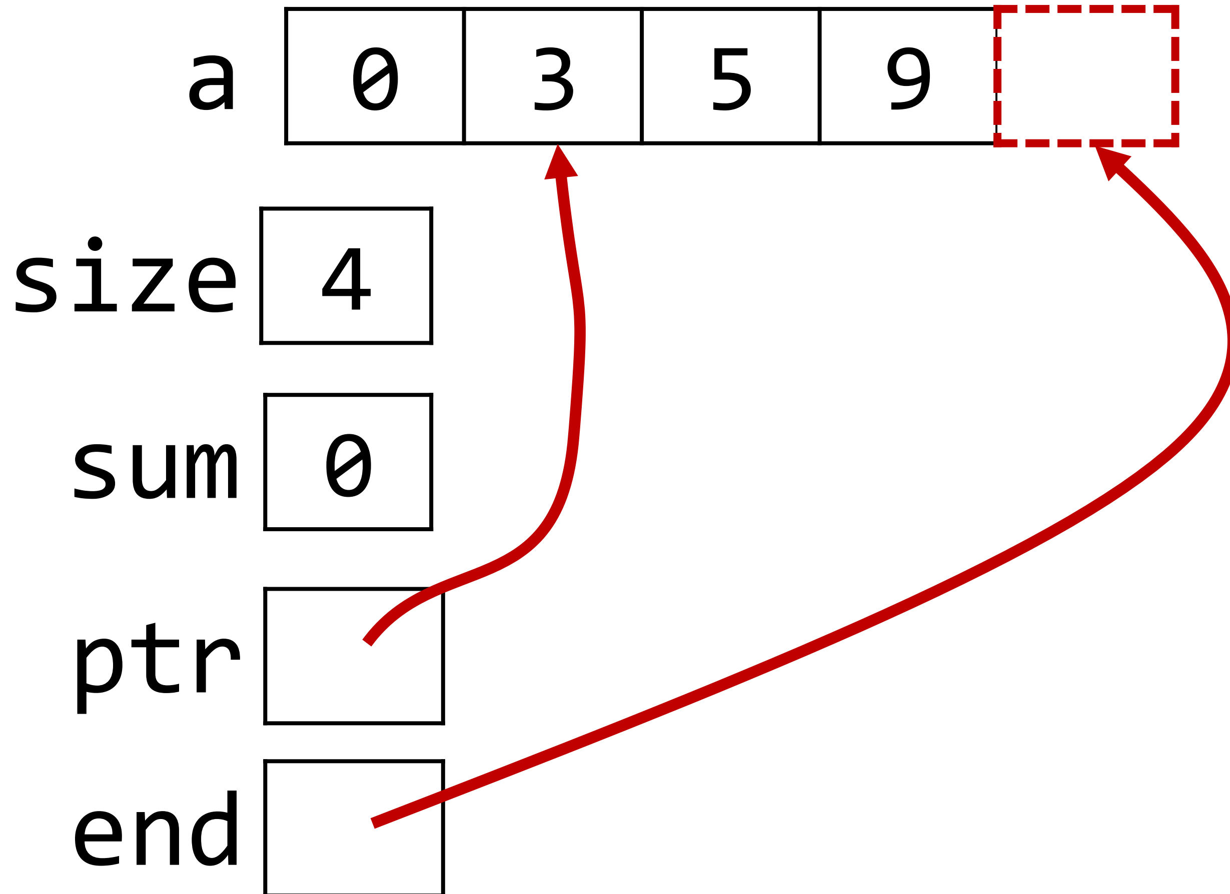
end

1 past the end of the array!

```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
    sum += *ptr;
}
```

C Pointer Arithmetic

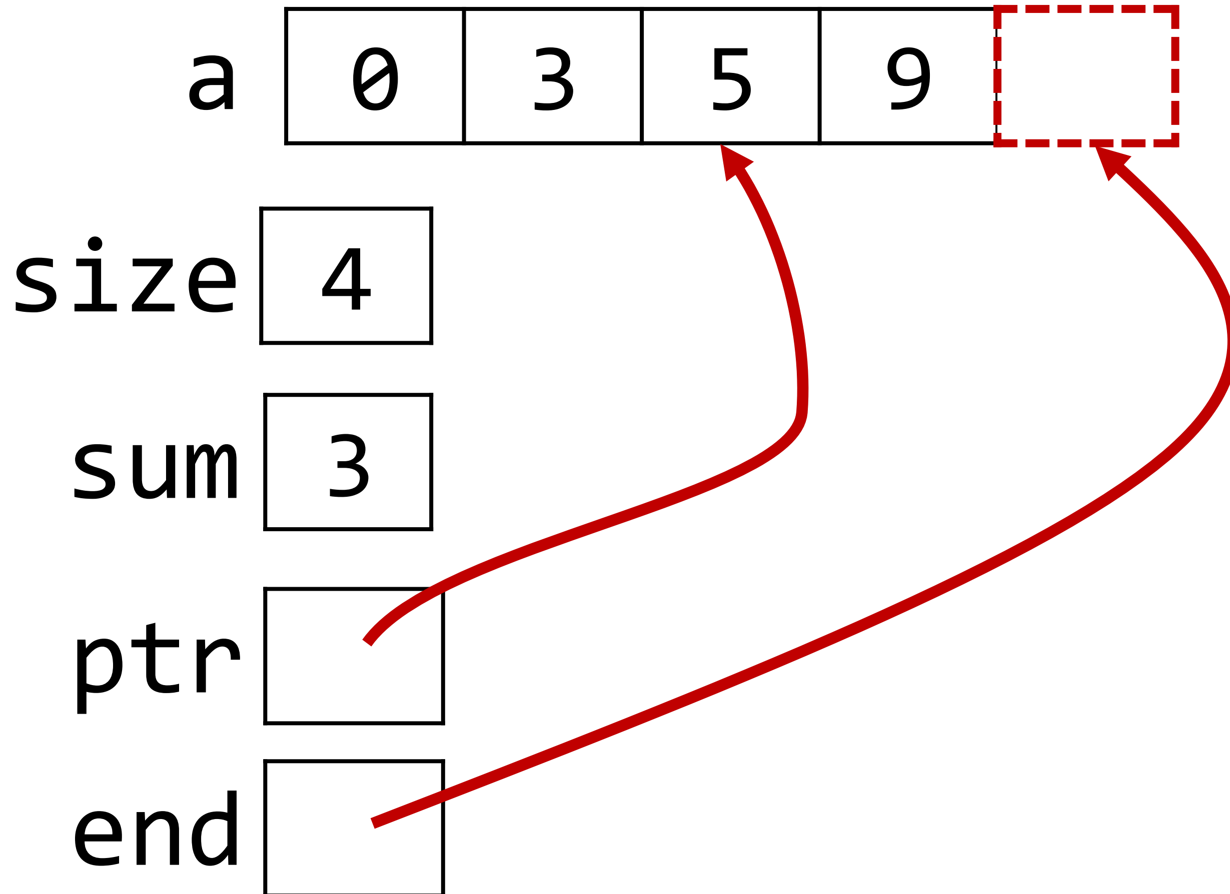
- We can do arithmetic on addresses to iterate through arrays




```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
   sum += *ptr;
}
```

C Pointer Arithmetic

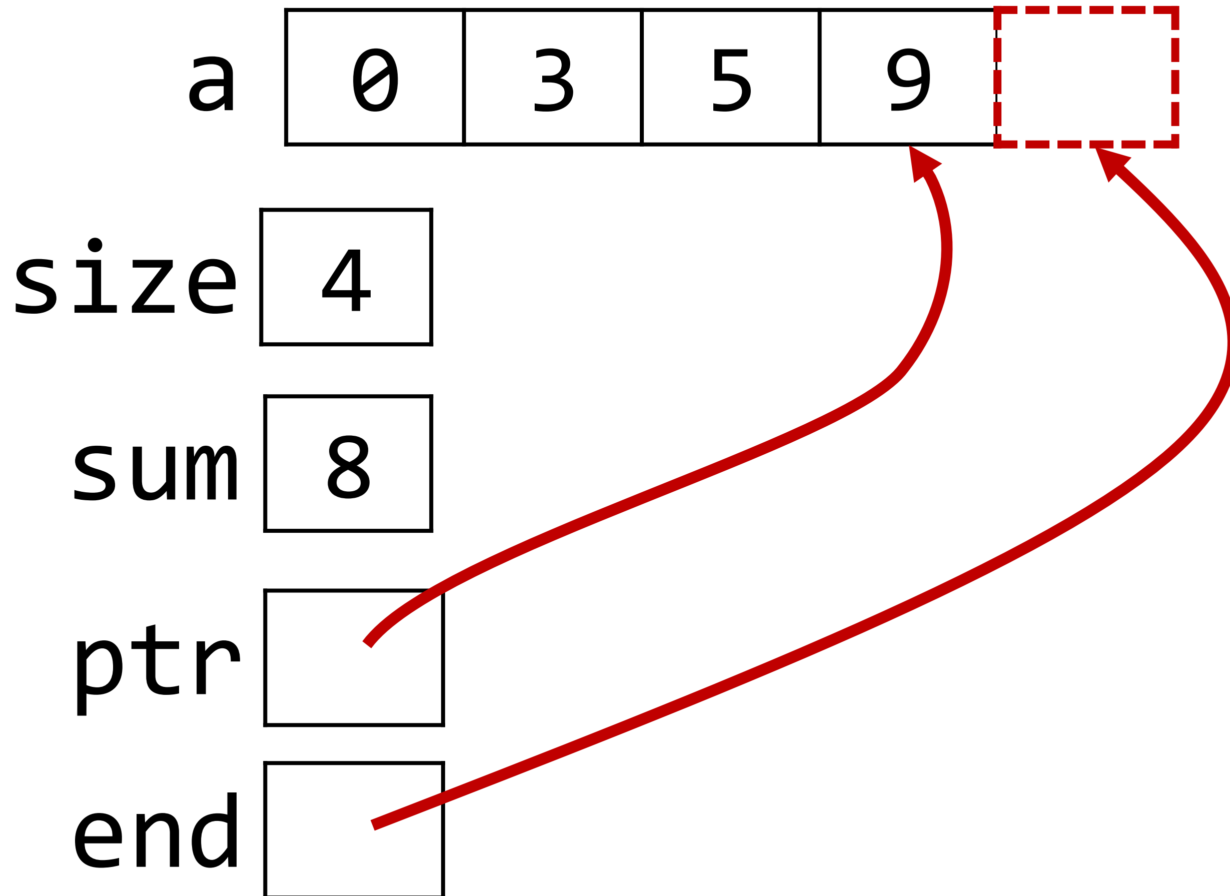
- We can do arithmetic on addresses to iterate through arrays




```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
    sum += *ptr;
}
```

C Pointer Arithmetic

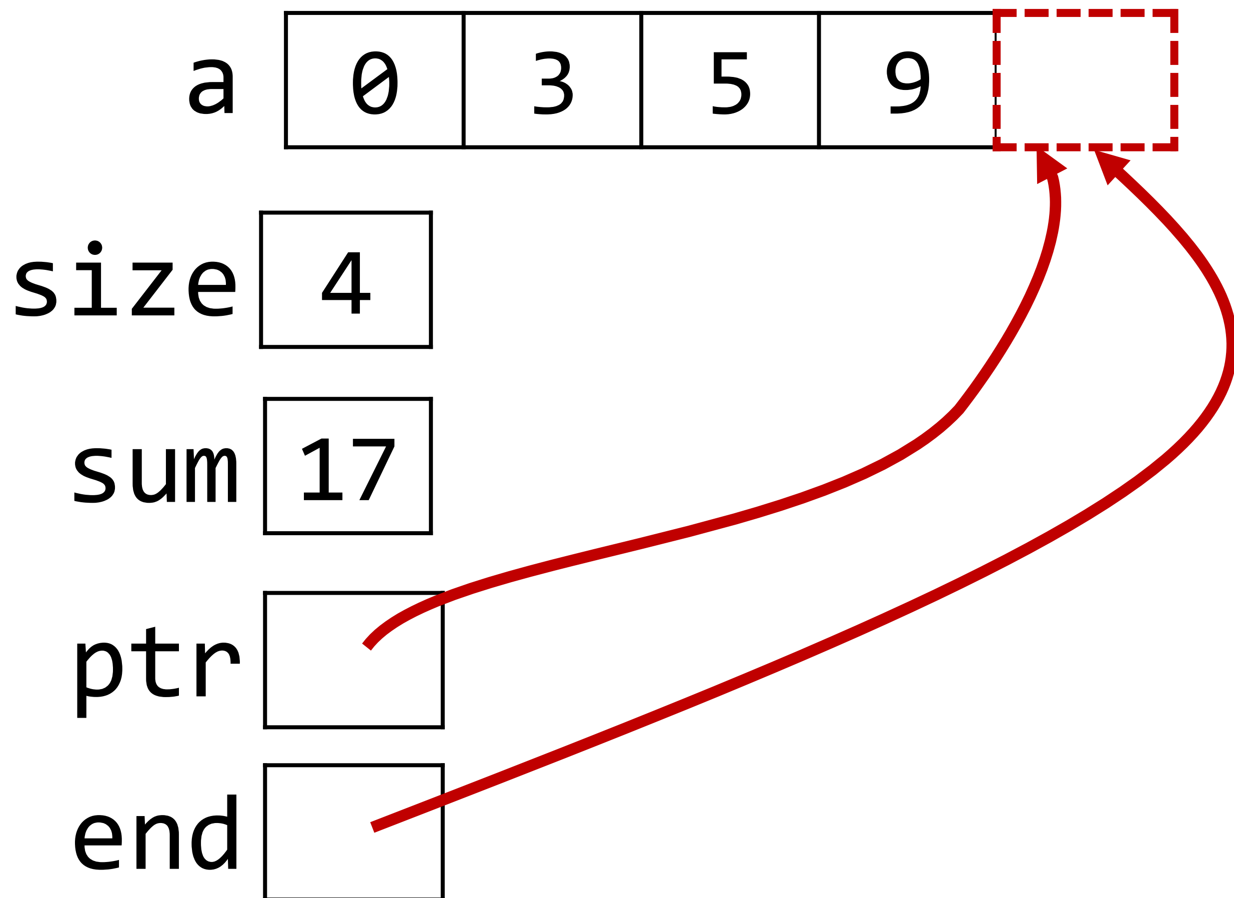
- We can do arithmetic on addresses to iterate through arrays




```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
    sum += *ptr;
}
```

C Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays




```
int a[] = {0, 3, 5, 9};
int size = 4;
int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
    sum += *ptr;
}
```

The C Toolchain

The C Toolchain

- A set of tools that are used in a chain with the purpose of compiling C code

The C Toolchain

- A set of tools that are used in a chain with the purpose of **compiling C code**

```c
int main() {
  return 0;
}
```

test.c

```
$gcc test.c -o test
```

```
0110101001010010
1010011101011000
0101010111010110
```

test

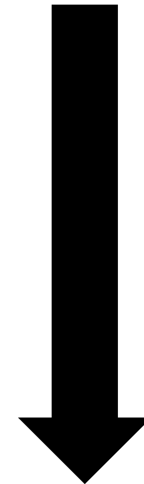
The C Toolchain

- A set of tools that are used in a chain with the purpose of **compiling C code**

```
int main() {
  return 0;
}
```

test.c

High-level language

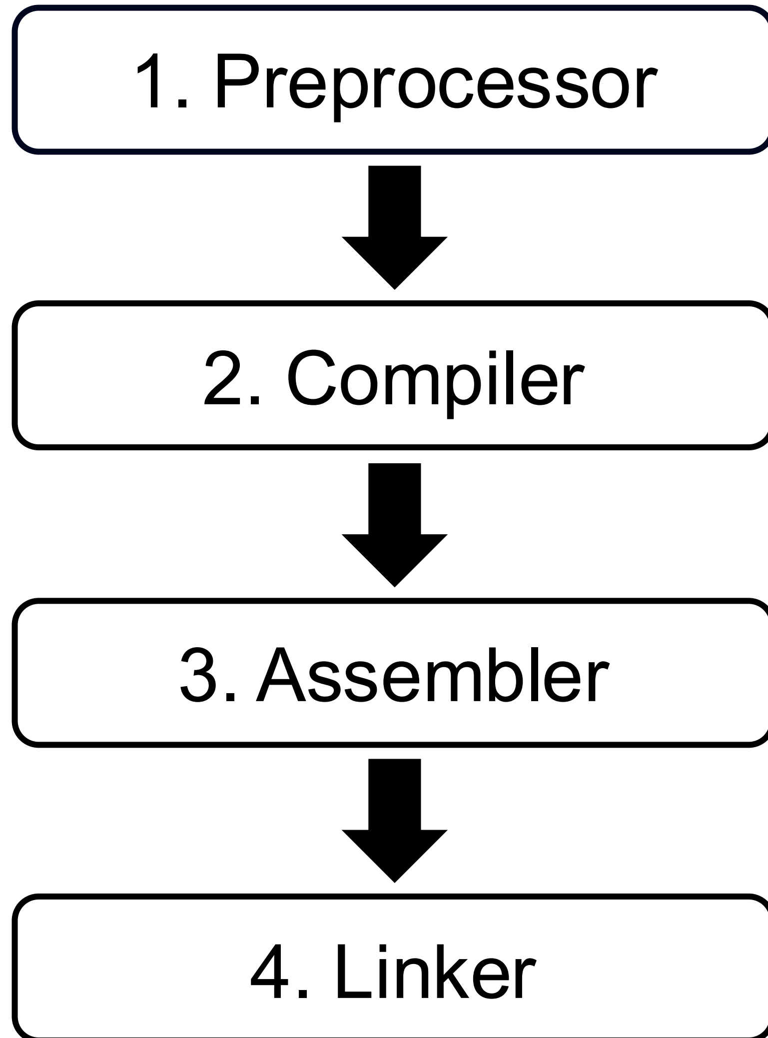


Machine language (Executable)

```
0110101001010010
1010011101011000
0101010111010110
```

test

The C Toolchain



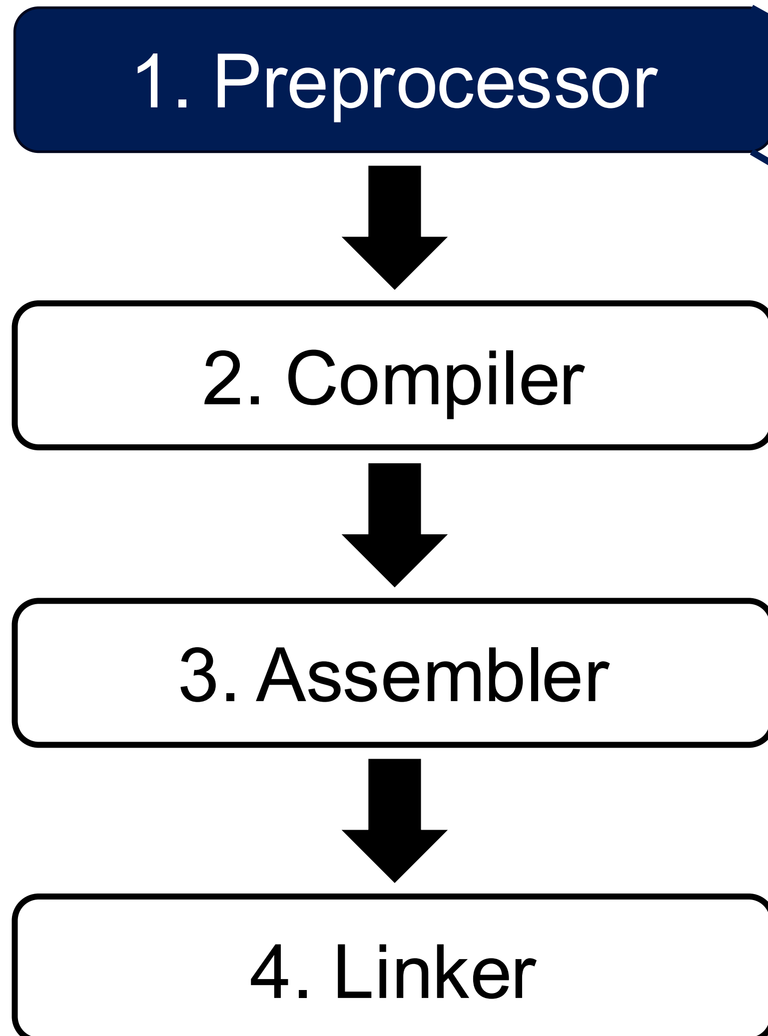
```c
int main() {
  return 0;
}
```

test.c

```
0110101001010010
1010011101011000
0101010111010110
```

test

1. Preprocessor



test.c

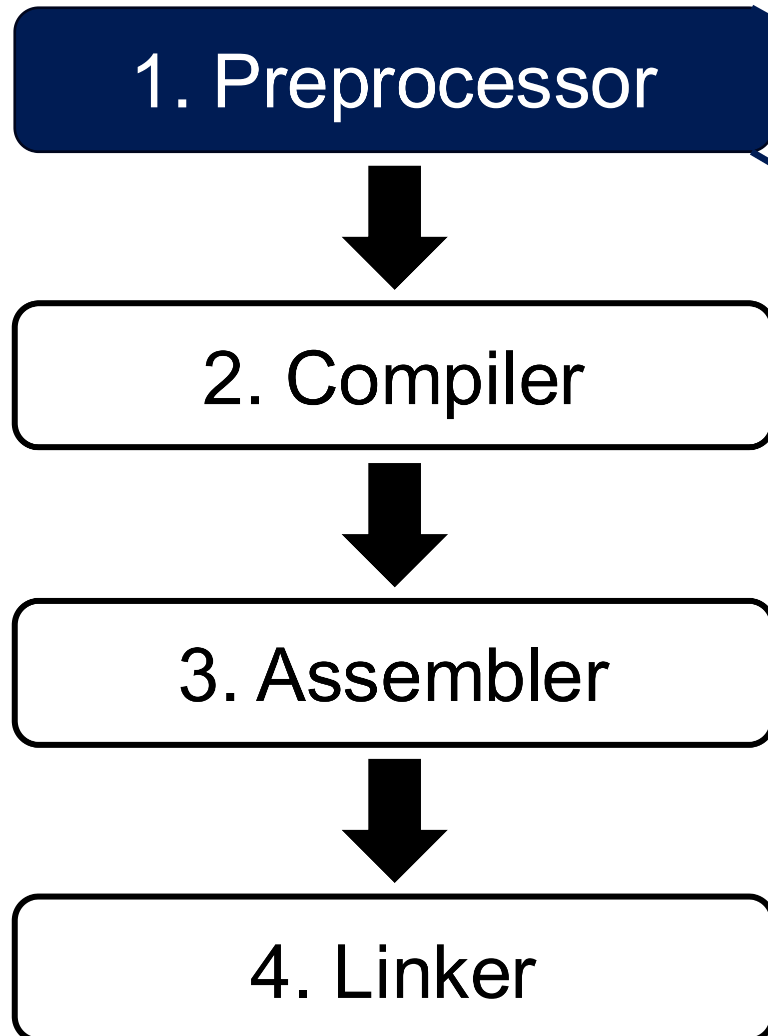
```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

`$gcc -E test.c -o test.i`

test.i

```c
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

1. Preprocessor



test.c

```c
#include <stdio.h>

int main() {
    printf("Hello world!\n");
    return 0;
}
```

Include a file

```
$gcc -E test.c -o test.i
```

test.i

```c
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

1. Preprocessor

- Perform preliminary operations, such as file inclusions and text substitutions
  - Especially, it modifies the original C program according to directives that begin with the '#' character

- Directives begin with the '#'
  - `#include`: insert another file

Including Headers (#include)

- Primarily used to incorporate headers

- There are two syntaxes for inclusion:
  - `#include <file>`

    Include a file from the system include path (e.g., `/usr/include/*.h`)

  - `#include “file”`

    Include a file from the current directory

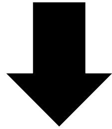
1. Preprocessor

- Perform preliminary operations, such as file inclusions and text substitutions
  - Especially, it modifies the original C program according to directives that begin with the ‘#’ character

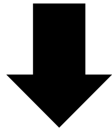
- Directives begin with the ‘#’
  - `#include`: insert another file
  - `#define`: define a symbol or macro
  - `#ifdef` / `#endif`: include the enclosed block only if a symbol is defined
  - `#if` / `#endif`: include only if a condition is true
  - …

2. Compiler

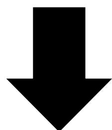
1. Preprocessor



2. Compiler



3. Assembler



4. Linker

test.i

```
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

```
$gcc -S -masm=intel test.i -o
test.s
```

test.s

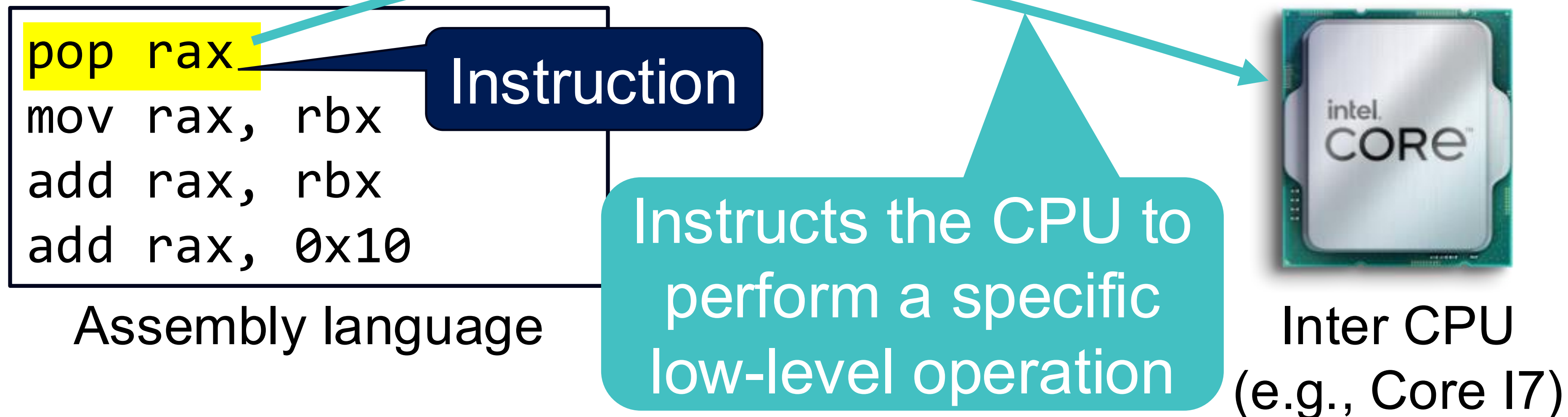
```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

The last human-readable format

2. Compiler

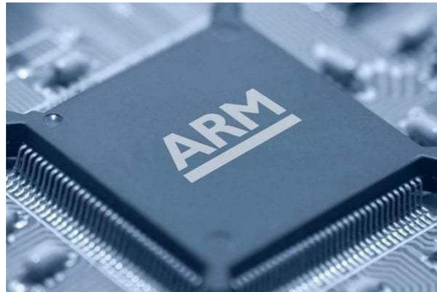
- Translates the source into the *assembly-language program*

- Assembly language (a.k.a, assembly, asm)
  - Low-level programming language that helps to communicate directly with computer hardware



Instruction Set

- The commands understood by a given architecture

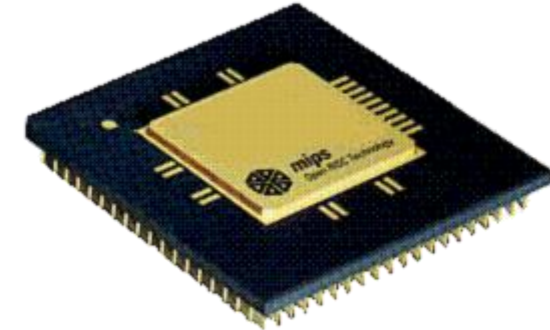


ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Instruction Set

- The commands understood by a given architecture

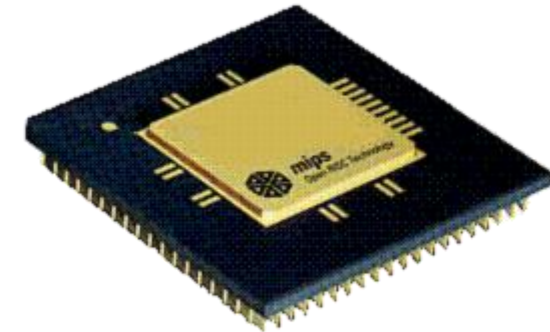
Different chips have different instruction sets

ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```



MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

Instruction Set

- The commands understood by a given architecture

Different chips have different instruction sets

ARM's instructions

```
pop {r0}
mov r0, r1
add r0, r0, r1
add r0, #16
```

Intel's Instructions

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

Our focus: Intel X86-64 (a.k.a., X64)

MIPS's Instructions

```
slt $t0, $s0, $s1
add $s2, $s0, $s1
sub $t2, $s1, $zero
lw  $t0, 8($s3)
```

(FYI) Instruction Format: AT&T vs Intel

- There are two ways to represent x86-64 assembly code

AT&T

```
mov %rax, %rbx
```

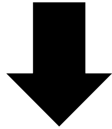
Intel

```
mov rbx, rax
```

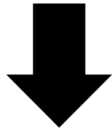
We will use the Intel syntax

2. Compiler

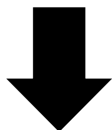
1. Preprocessor



2. Compiler



3. Assembler



4. Linker

test.i

```
...
extern int printf (const char
*__restrict __format, ...);
...
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Intel syntax

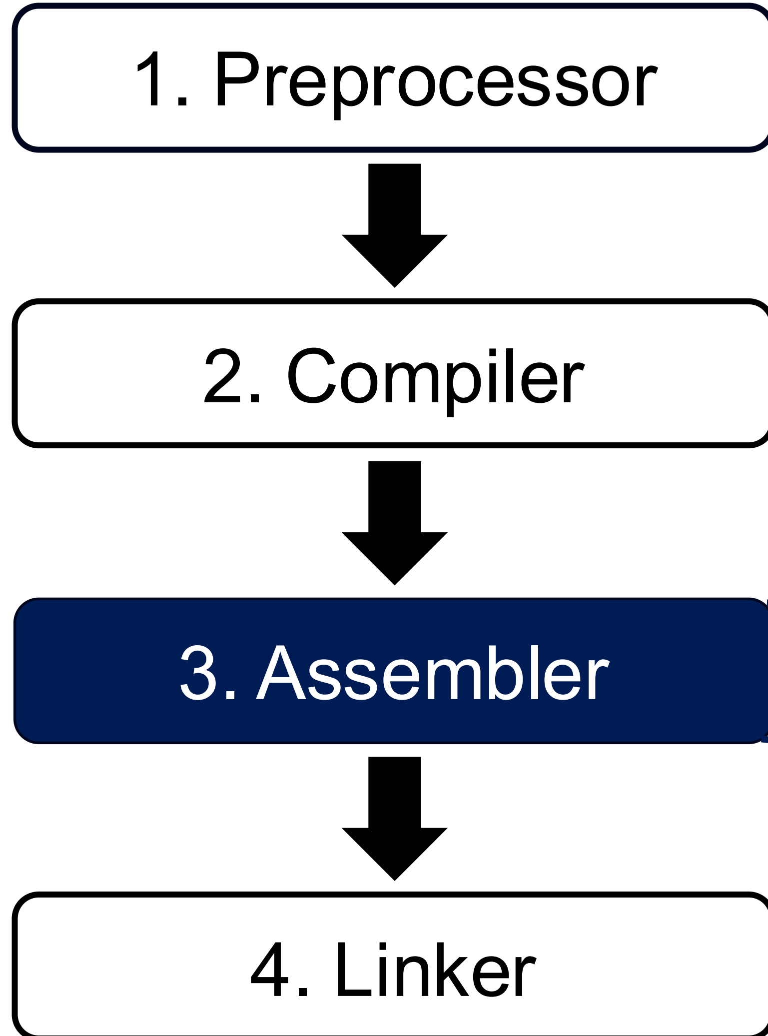
```
$gcc -S -masm=intel test.i -o
test.s
```

test.s

```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

The last human-readable format

3. Assembler



test.s

```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

```
$as test.s -o test.o
```

test.o

```
01000101010011
10101000101010
01000101010110
00010101111001 1
```

3. Assembler

- Translates assembly code into machine language instructions (binary)

Mapped to a group of bits

Decodes the bits to understand and perform operations

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

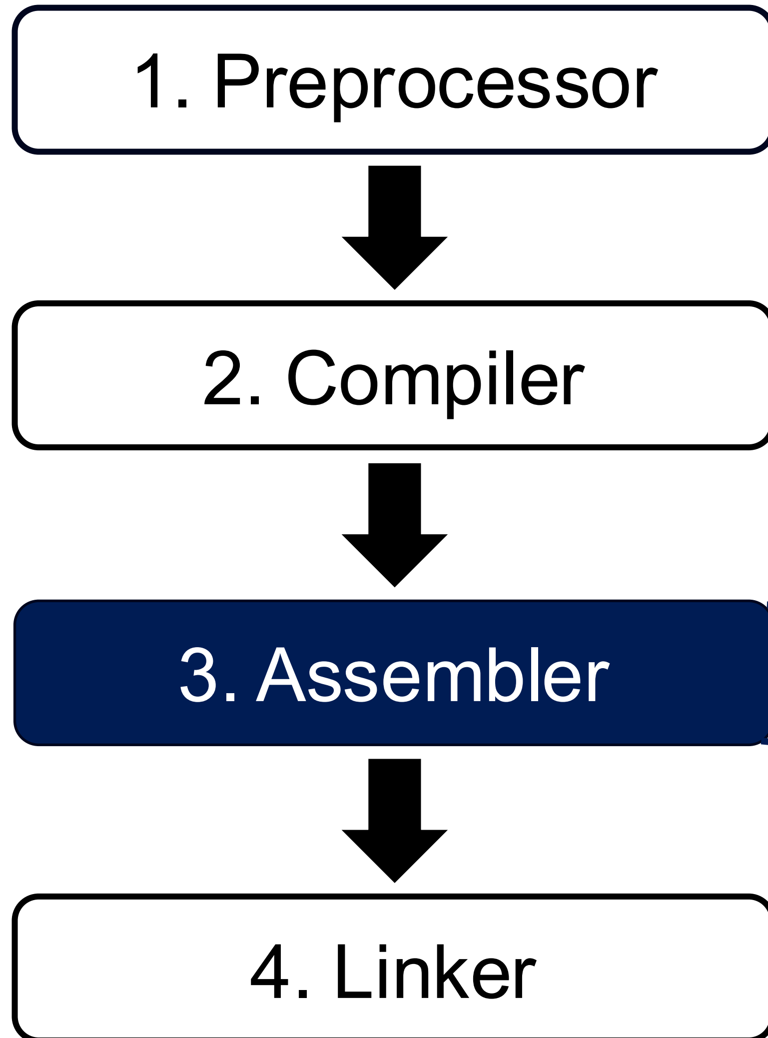
Assembly language

→ Assembler →

```
0100010101000011
1010100010101000
0100010101011000
0001010111100011
```

Machine language

(FYI) Disassembling Binary Code



test.s

```
main:
  push rbp
  mov rbp, rsp
  lea rax, .LC0[rip]
  mov rdi, rax
  call puts@PLT
  mov eax, 0
```

test.o

```
010001010100011
101010001010100
010001010101100
000101011110011
```

Disassembly! (a.k.a., reverse engineering)

(FYI) Disassembly: objdump


```
$ objdump -Mintel -d test.o

test.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:  f3 0f 1e fa                endbr64
   4:  55                         push   rbp
   5:  48 89 e5                   mov    rbp,rsp
   8:  48 8d 05 00 00 00 00       lea    rax,[rip+0x0]
   f:  48 89 c7                   mov    rdi,rax
  12:  e8 00 00 00 00             call   17 <main+0x17>
  17:  b8 00 00 00 00             mov    eax,0x0
  1c:  5d                         pop    rbp
  1d:  c3                         ret
```

(FYI) Disassembly: objdump

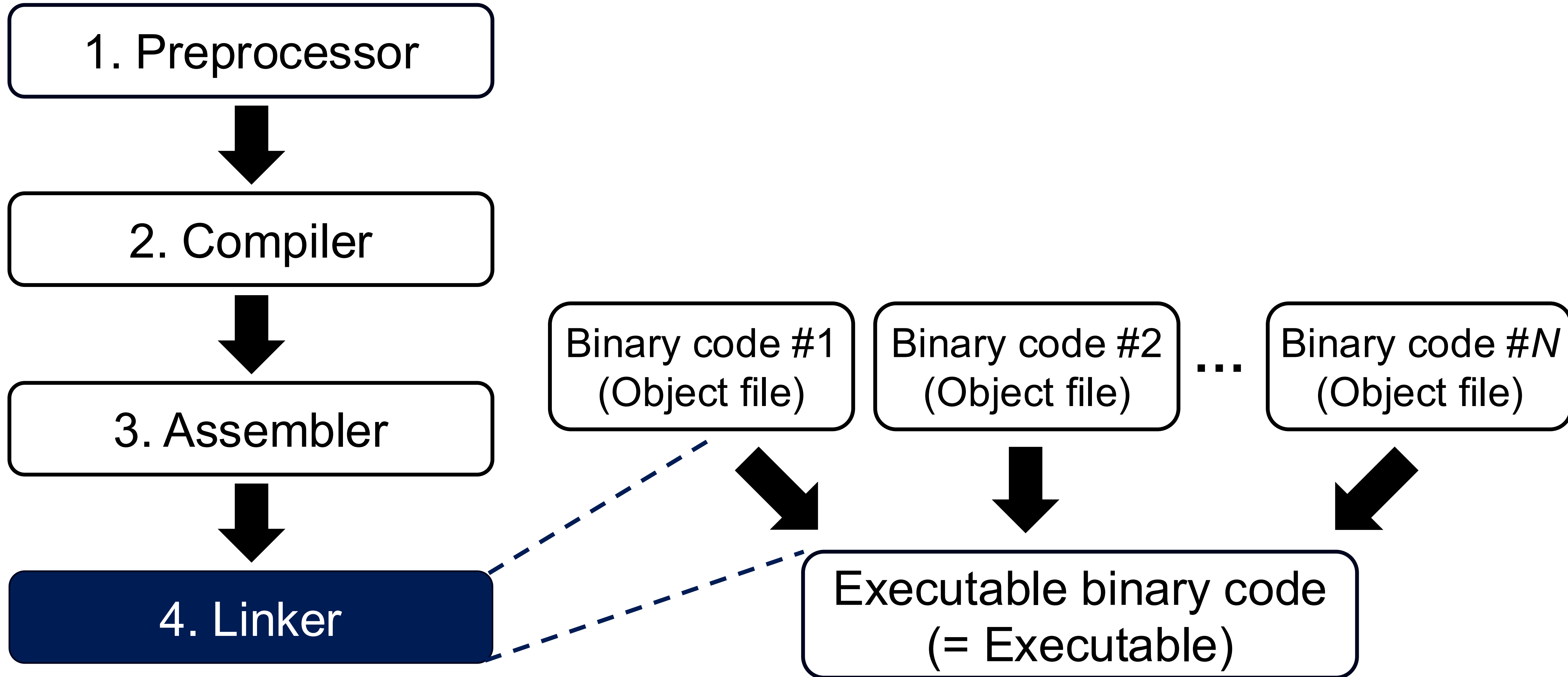
```
$ objdump -Mintel -d test.o

test.o:      file format elf64-x86-6

             of                  xt:

0000000000000000 <main>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   rbp
   5:   48 89 e5                mov    rbp,rsp
   8:   48 8d 05 00 00 00 00    lea    rax,[rip+0x0]
   f:   48 89 c7                mov    rdi,rax
  12:   e8 00 00 00 00          call   17 <main+0x17>
  17:   b8 00 00 00 00          mov    eax,0x0
  1c:   5d                      pop    rbp
  1d:   c3                      ret
```

Address

Binary code

Disassembled assembly code

4. Linker



4. Linker

- Joins multiple object files, including libraries, into **an executable**
  - It maintains a symbol table for each object file
  - Unresolved symbols in one object file may be found (and thus resolved) in other object files/libraries

test.o

```
01000101010011
10101000101010
01000101010110
00010101111001
1
```

The `printf` function is not defined

Linking is Essential for C Programs

```
$ ld -o test test.o
```

Only a single file is linked

Linking command

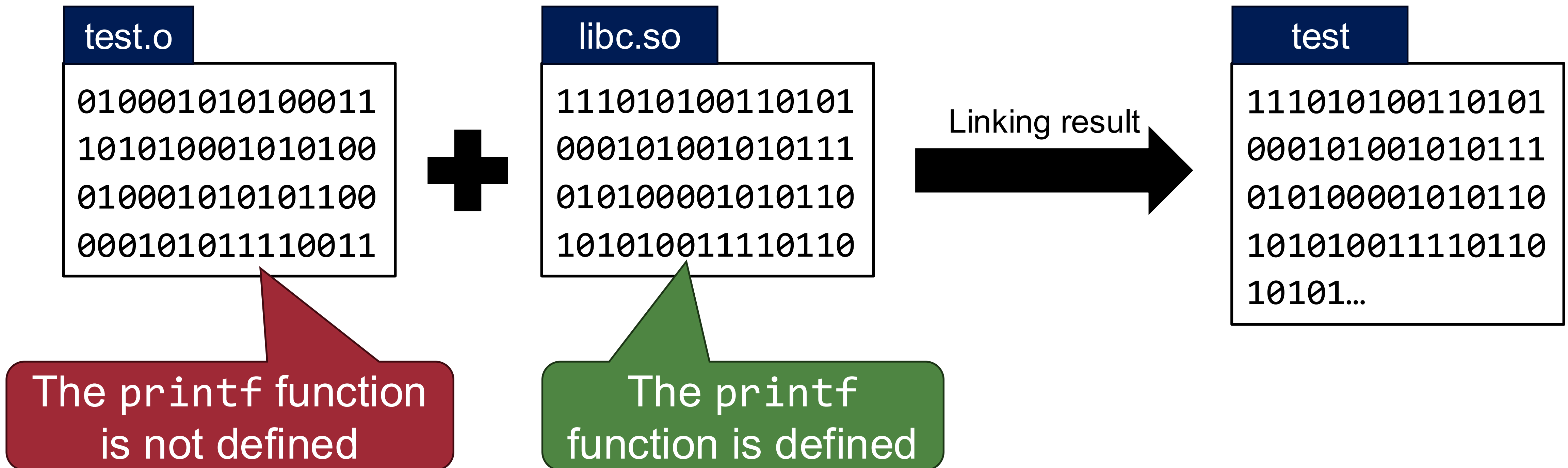
Object files to link

Results: Linking failure

```
ld: warning: cannot find entry symbol _start;
defaulting to 0000000000401000
ld: test.o: in function `main':
test.c:(.text+0x13): undefined reference to `puts'
```

4. Linker

- Joins multiple object files, including libraries, into **an executable**
  - It maintains a symbol table for each object file
  - Unresolved symbols in one object file may be found (and thus resolved) in other object files/libraries



Linking Internals

```
$ gcc test.o -o test
```

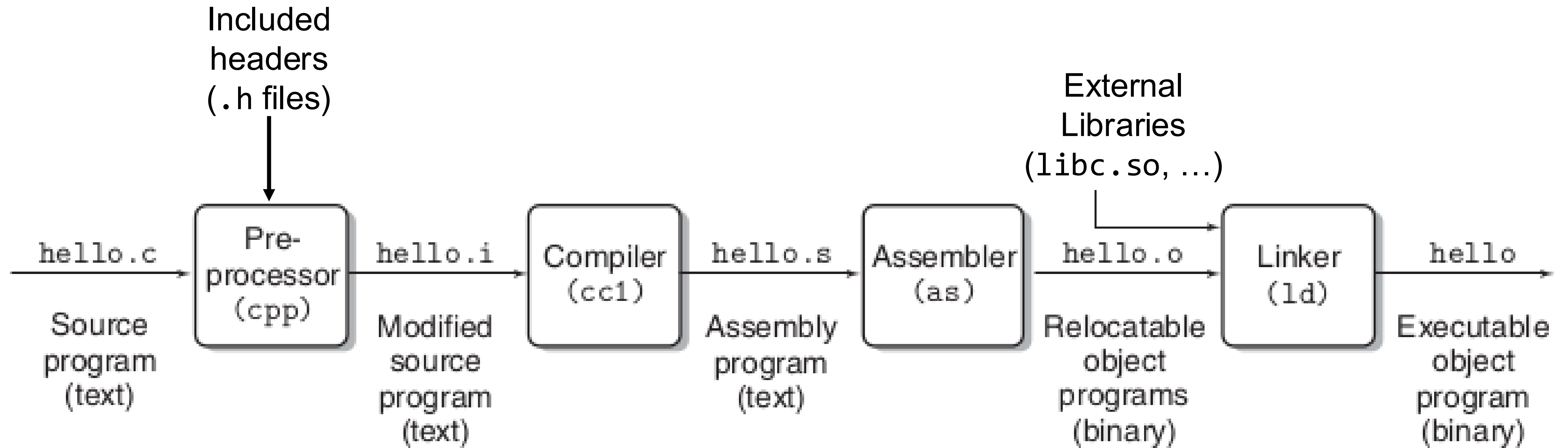
Link `test.o` with the system libraries to produce `test`

```
$ ./test
Hello world!
```

View the linkage

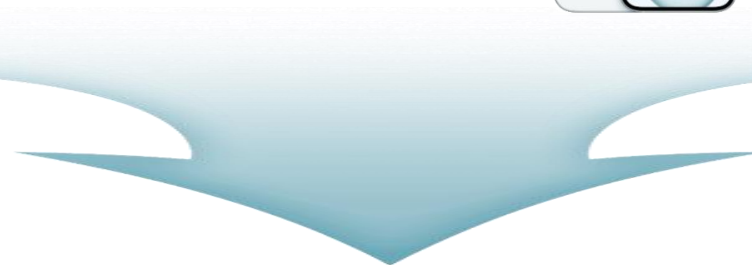
```
$ ldd test
linux-vdso.so.1 (0x00007ffc4a73d000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
(0x00007f2af3c44000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2af3e54000)
```

Summary of the C Toolchain



Hardware Organization of a system

Hardware Organization of a System



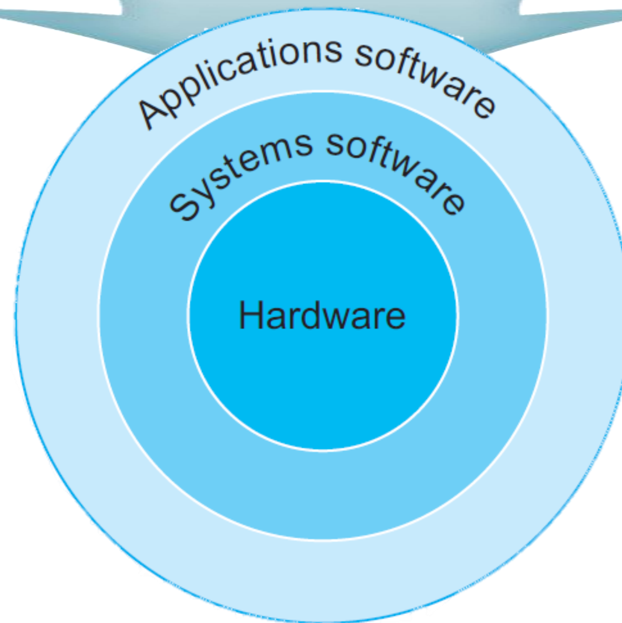
Your program

Computer

Computation results

Hardware Organization of a System

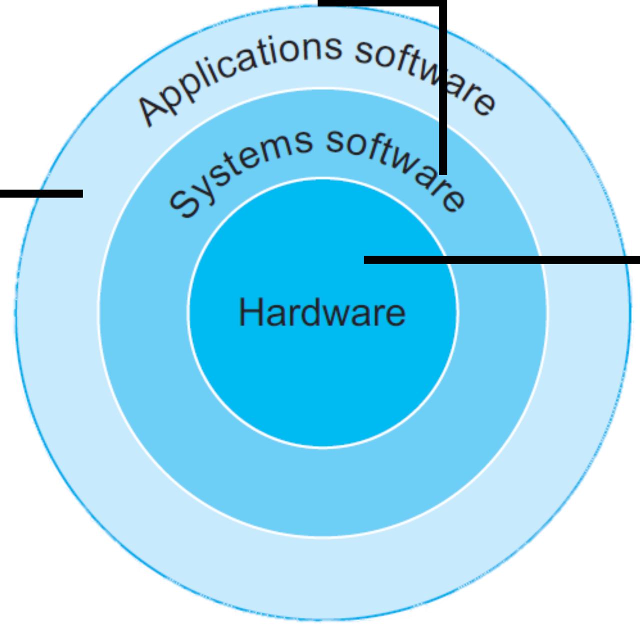
Your program



Computation results

Hardware Organization of a System

Your program



Computation results

Hardware Organization of a System

Abstraction layer #1: Application software

Abstraction layer #2: System software

Abstraction layer #3: Hardware

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

High-level language

Compiler

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

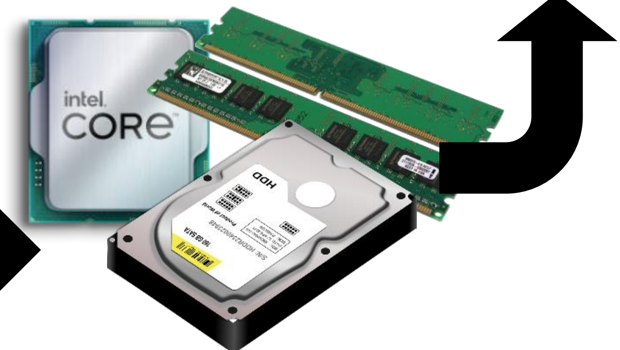
Assembly language

Assembler

```
010001010010
001101001001
```

Machine language

OS



Computation results

Hardware Organization of a System

Your program

```
#include <stdio.h>

swap(int v[], int k)
{
   int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

High-level language

Compiler

Compilation

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

Assembly language

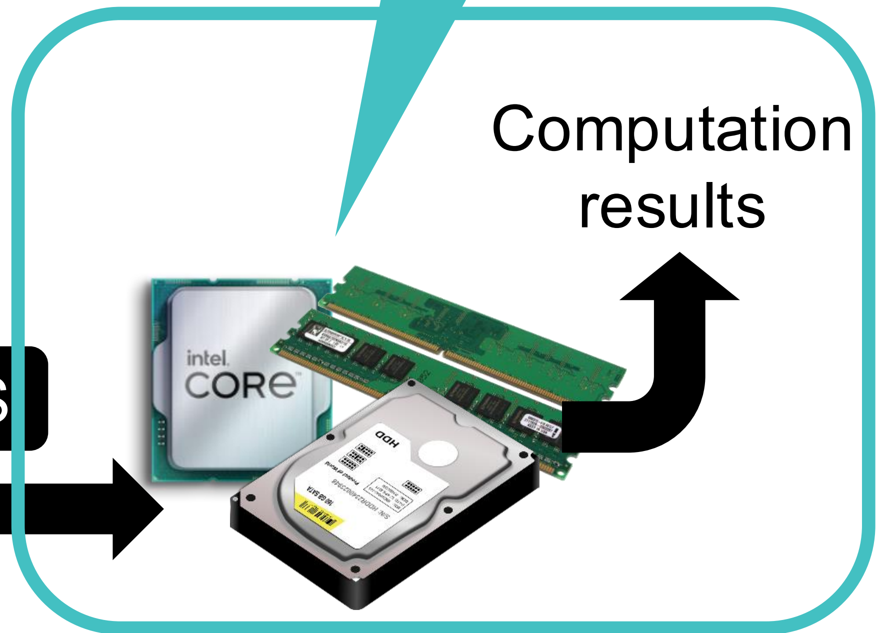
Assembler

```
010001010010
001101001001
```

Machine language

OS

Execute the program



Computation results

Hardware Organization of a System

```
#include <stdio.h>

swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

Your program

High-level language

Compiler

```
pop rax
mov rax, rbx
add rax, rbx
add rax, 0x10
```

Assembly language

Compilation

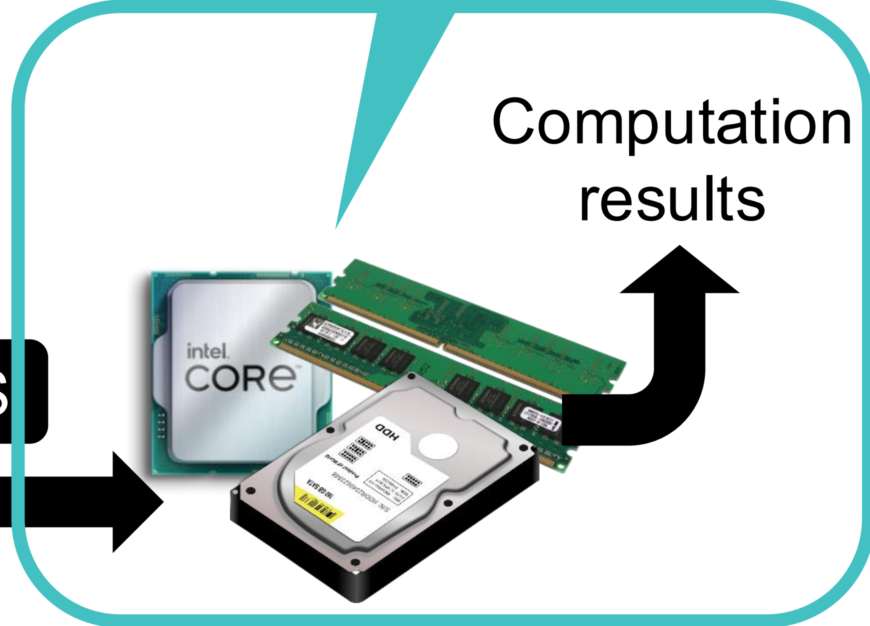
Assembler

```
010001010010
001101001001
```

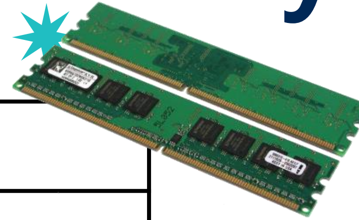
Machine language

OS

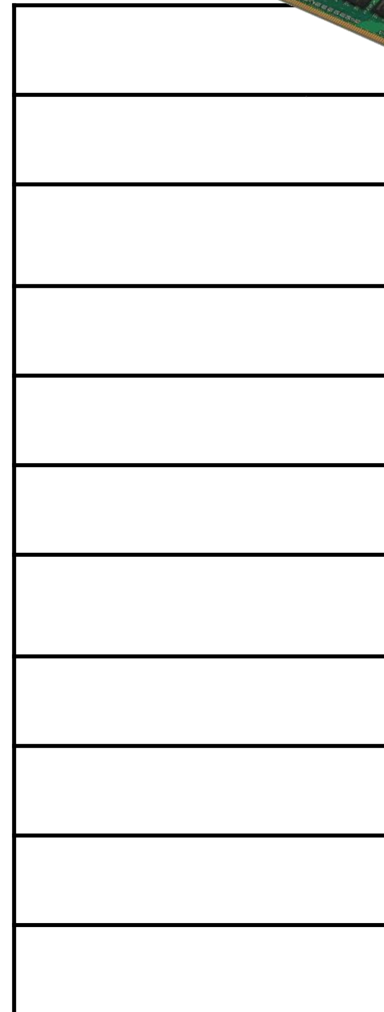
Execute the program



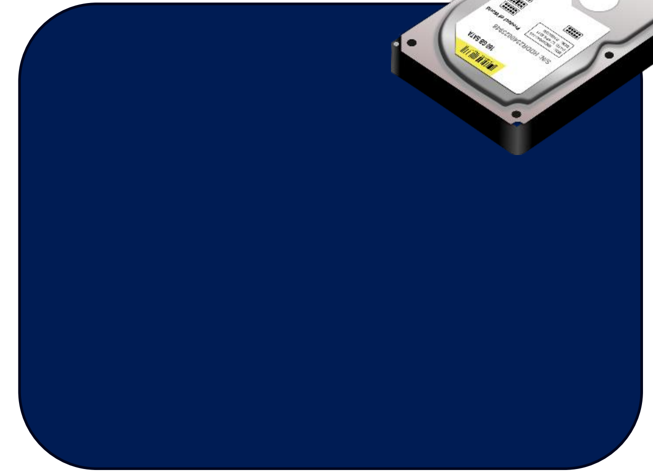
Hardware Organization of a System



Processor
(CPU)
(e.g., Intel I7)



Main memory

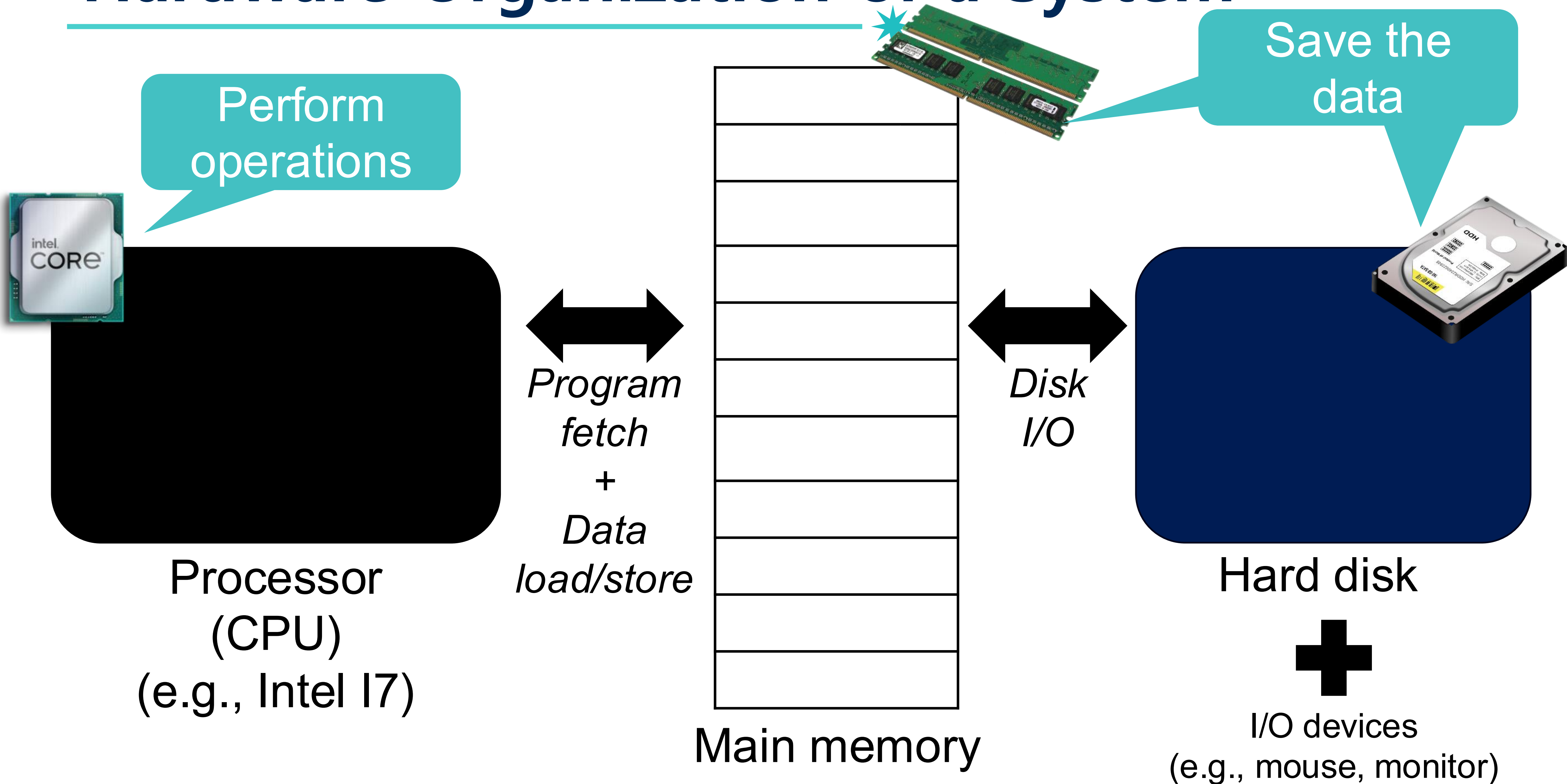


Hard disk

➕

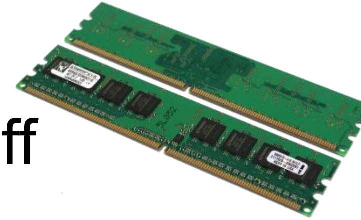
I/O devices
(e.g., mouse, monitor)

Hardware Organization of a System



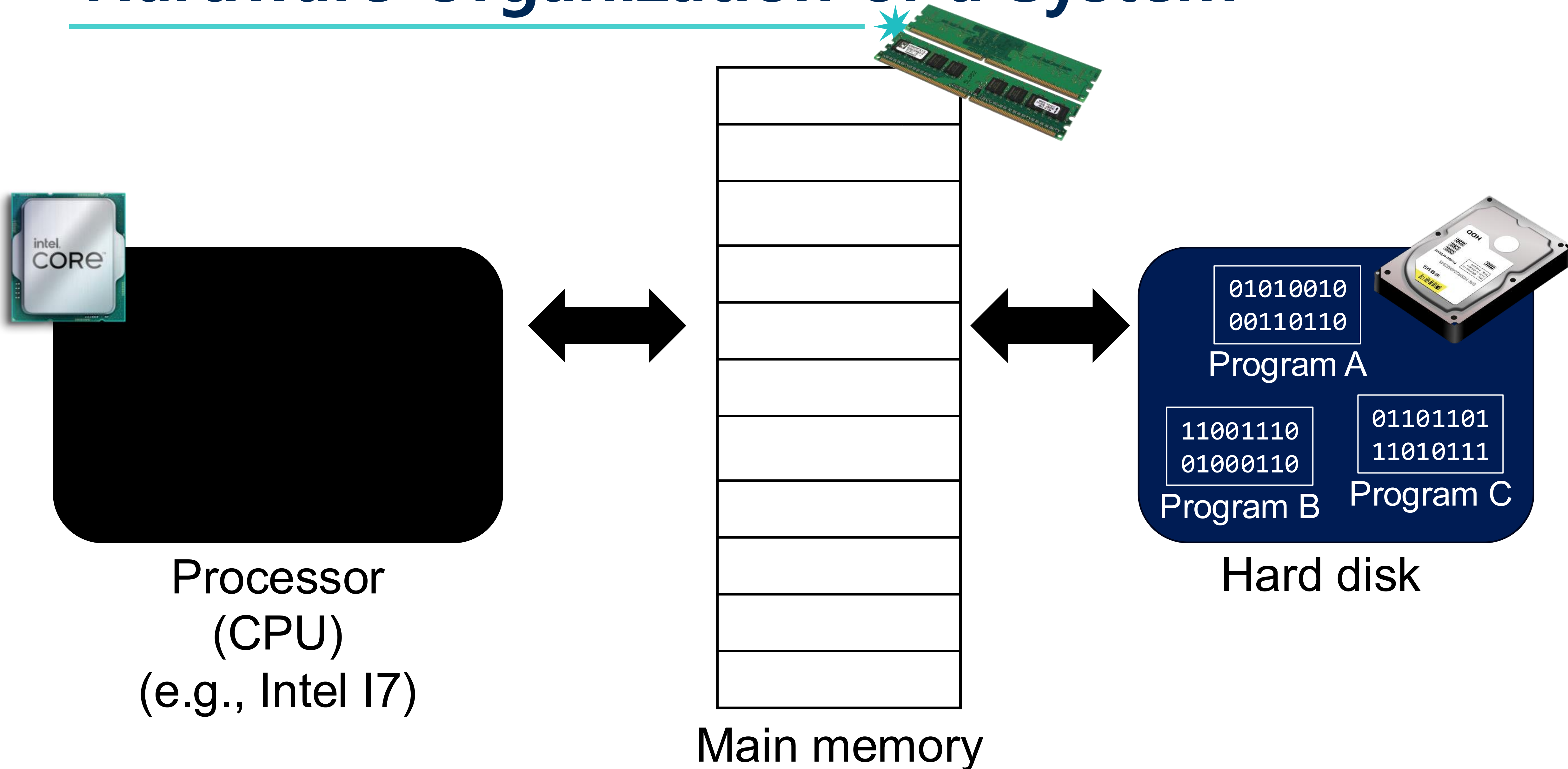
A Safe Place for Data

- **Volatile main memory**
  - Loses instructions and data when power off
  - E.g., DRAM

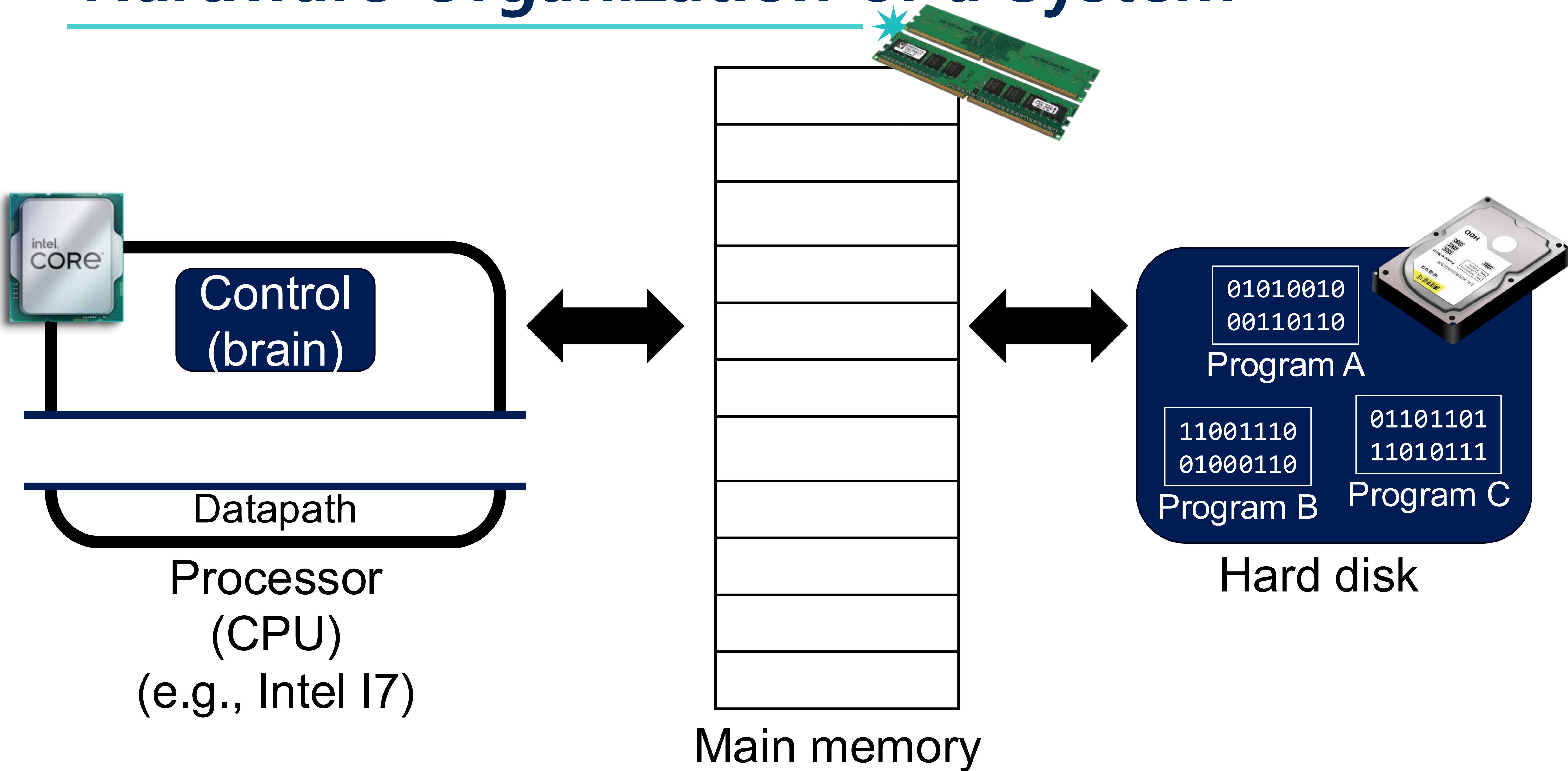


- **Non-volatile secondary memory**
  - HDD, SSD
  - Magnetic disk
  - Flash memory
  - Optical disk (CDROM, DVD)

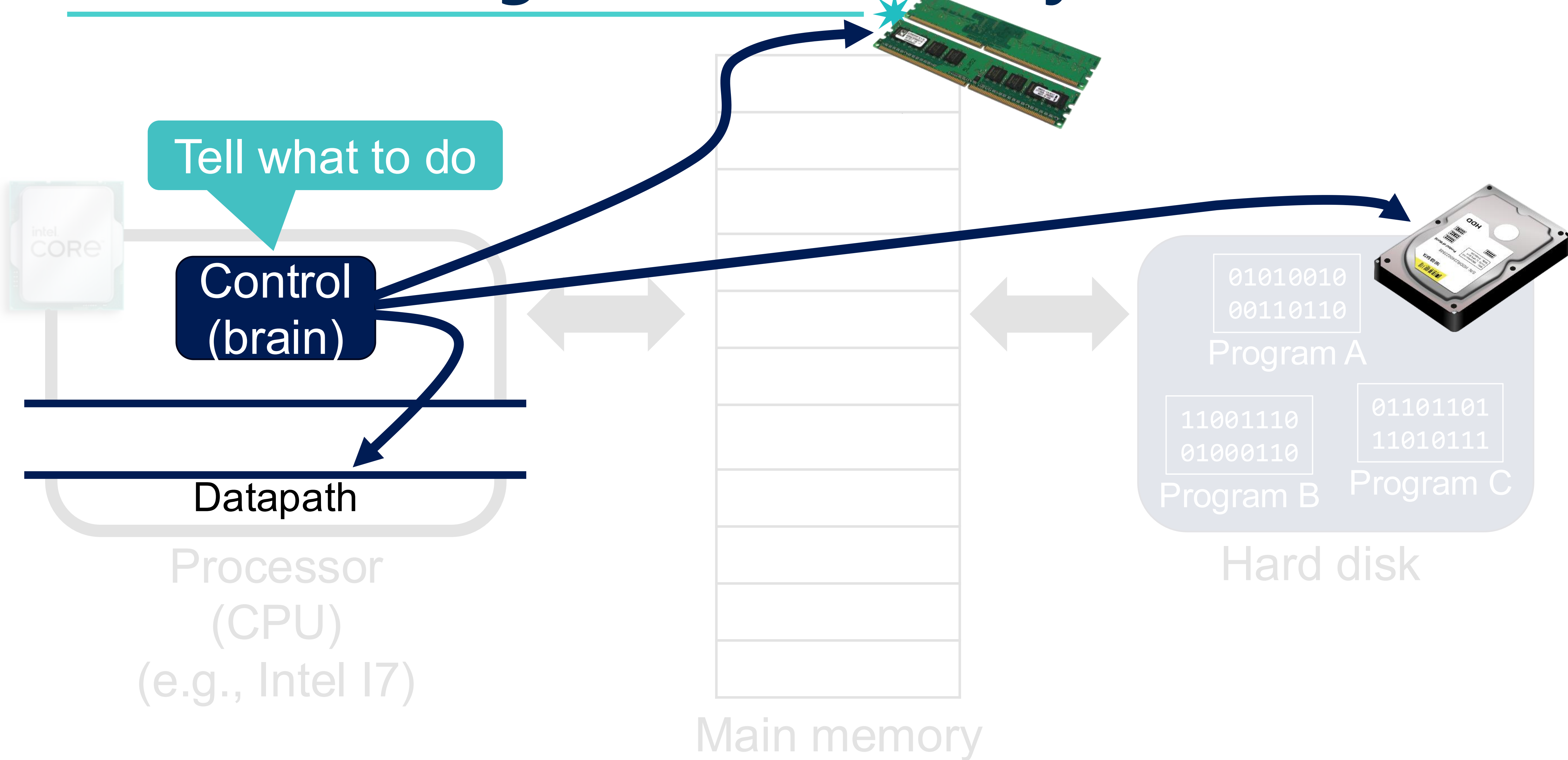
Hardware Organization of a System



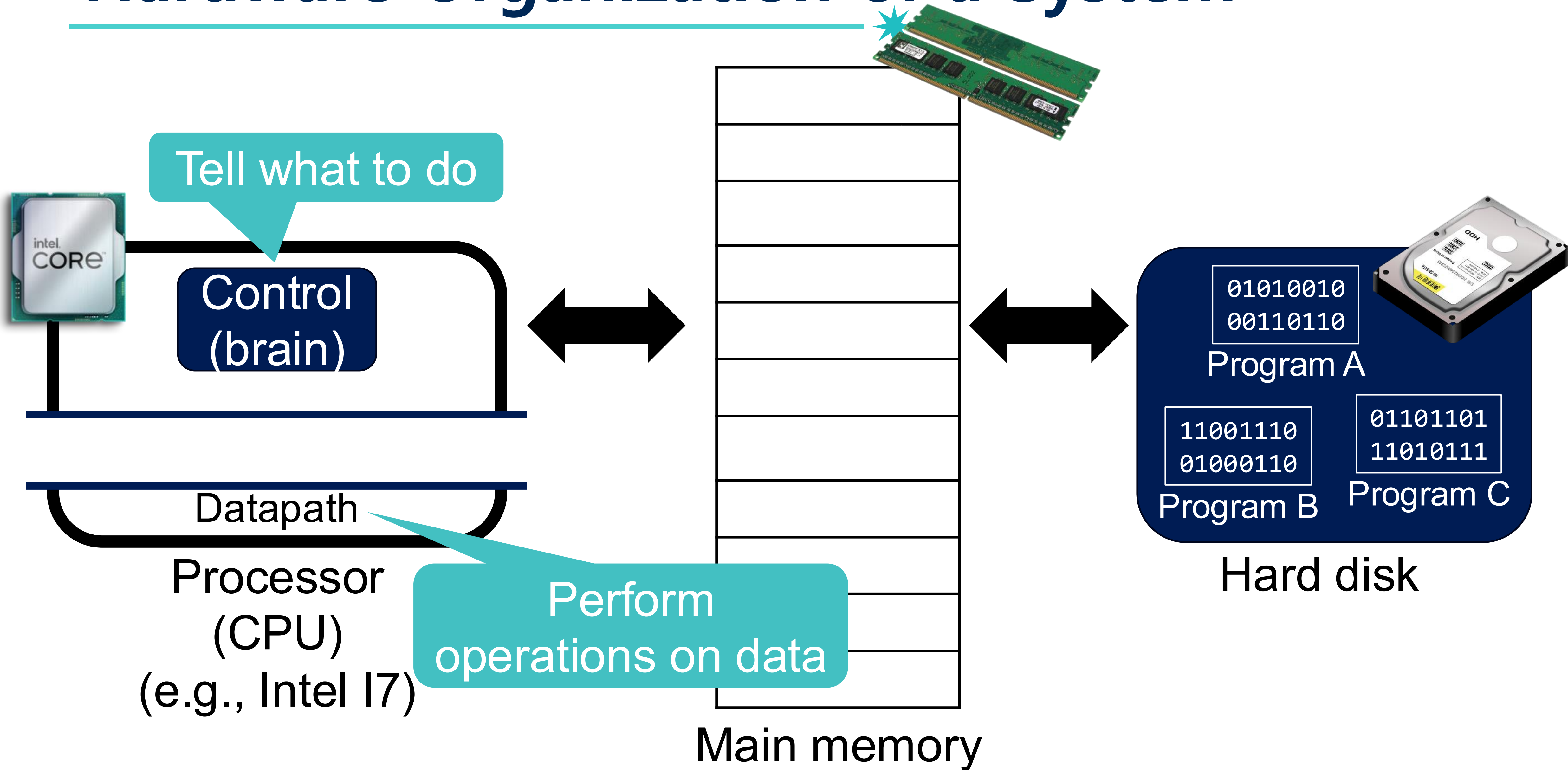
Hardware Organization of a System



Hardware Organization of a System



Hardware Organization of a System

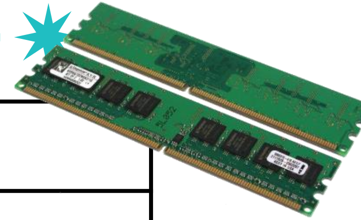


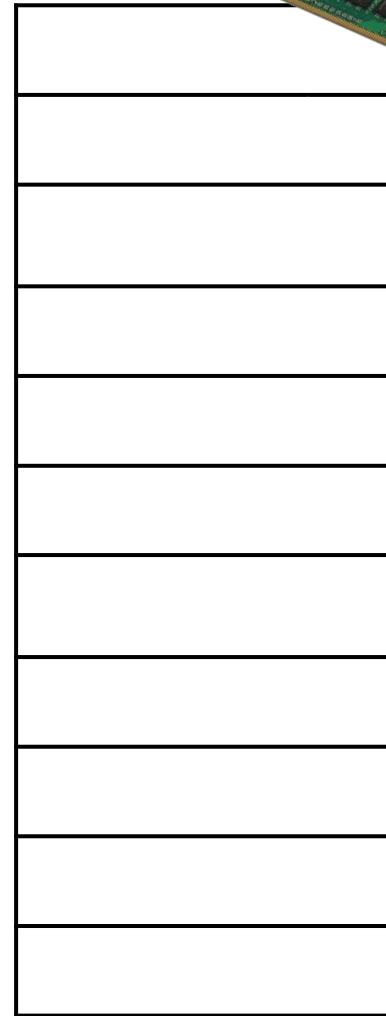
Let's Execute the Program!

Control (brain)

Datapath

Processor (CPU) (e.g., Intel I7)





Main memory

01010010
00110110
Program A

11001110
01000110
Program B

01101101
11010111
Program C

Hard disk

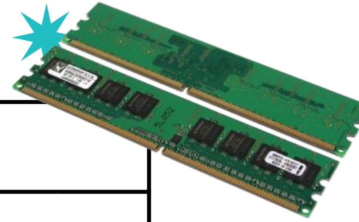
Let’s Execute the Program!

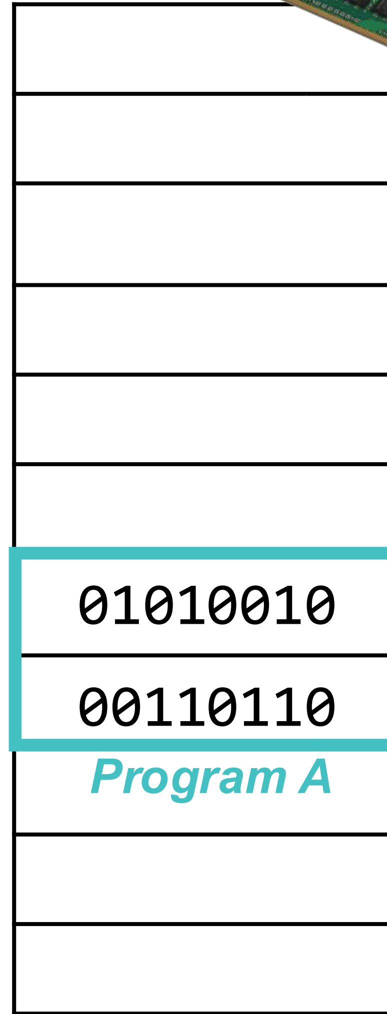
Execute program A
($ ./programA)

Control
(brain)

Datapath

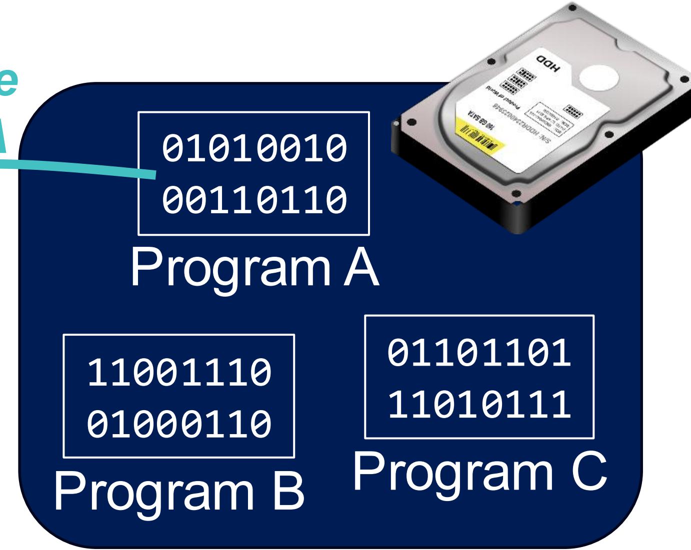
Processor
(CPU)
(e.g., Intel I7)





Main memory

1. Load the Program A



Hard disk

Let's Execute the Program!

Execute program A
($ ./programA)

Control
(brain)

01010010

Datapath

Processor
(CPU)
(e.g., Intel I7)

1. Load the Program A

01010010
00110110
Program A

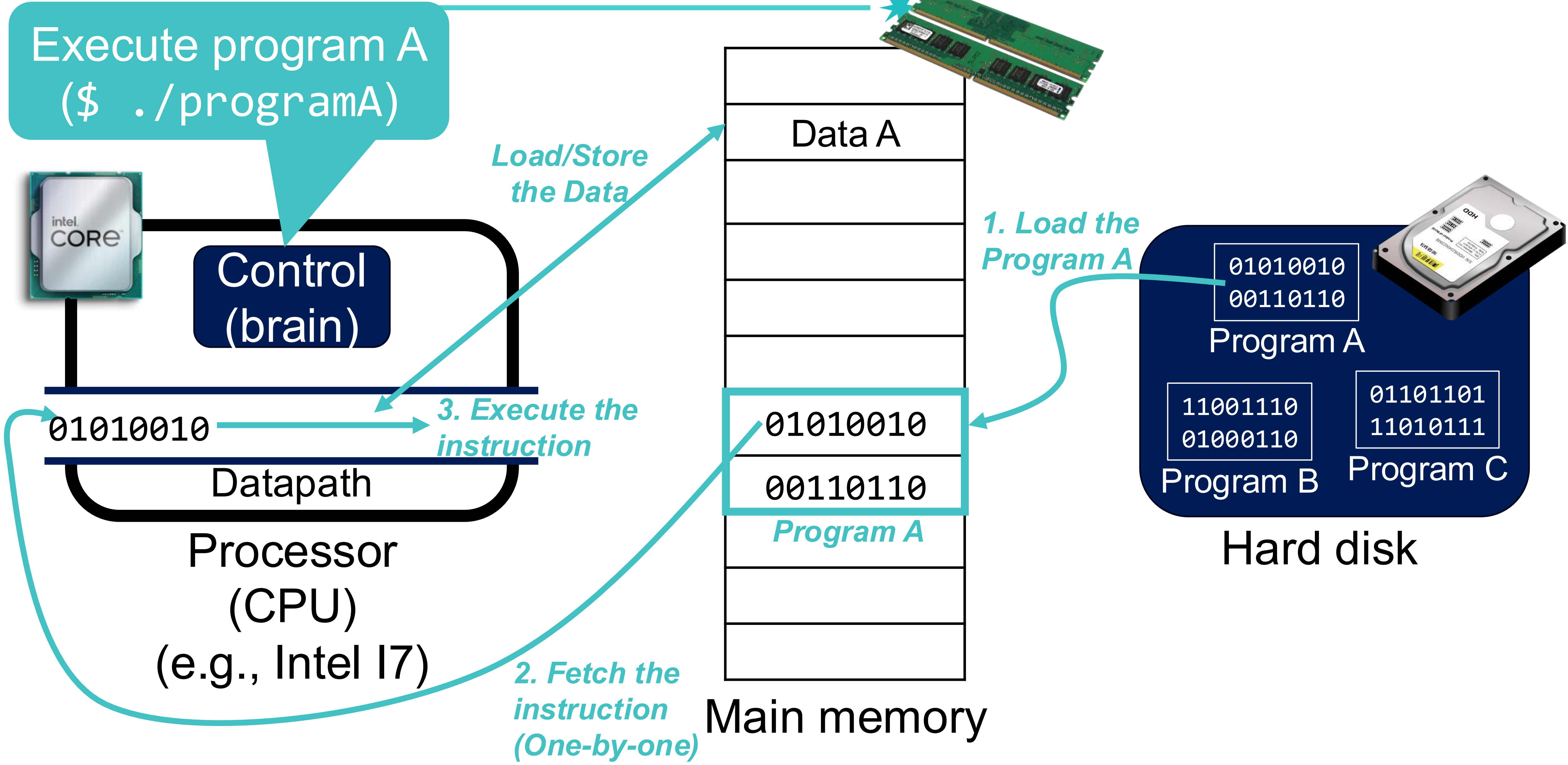
Main memory

01010010
00110110
Program A
11001110
01000110
Program B
01101101
11010111
Program C

Hard disk

2. Fetch the instruction (One-by-one)

Let's Execute the Program!



Inside the Processor (CPU)

- **Datapath**
  - Perform operations on data
- **Control**
  - Tell the datapath, memory, and I/O devices what to do according to program
- **Registers/cache memory**
  - Small fast memory for immediate access to data

Hardware Organization of a System

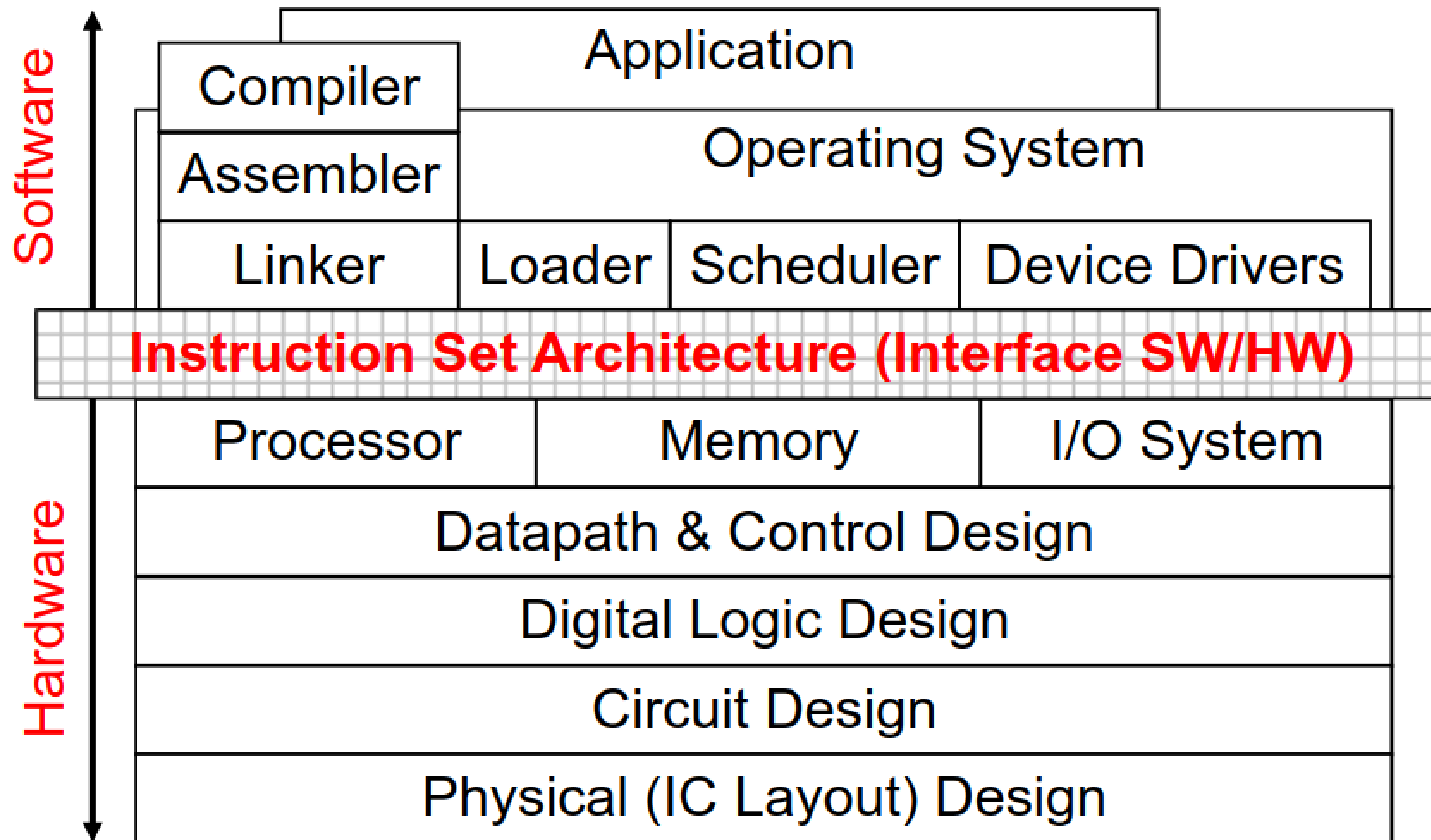


Image from: https://medium.com/@shb8086/tutorial-series-isa-fa9648c5e9e5

Final Remark

- Get familiar
  - C language
  - shell commands
  - gcc
  - git

- They are not covered directly in our class but are required for assignments and course activities

Question?

Today, everyone will be acknowledged for attendance!