ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2025/26

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

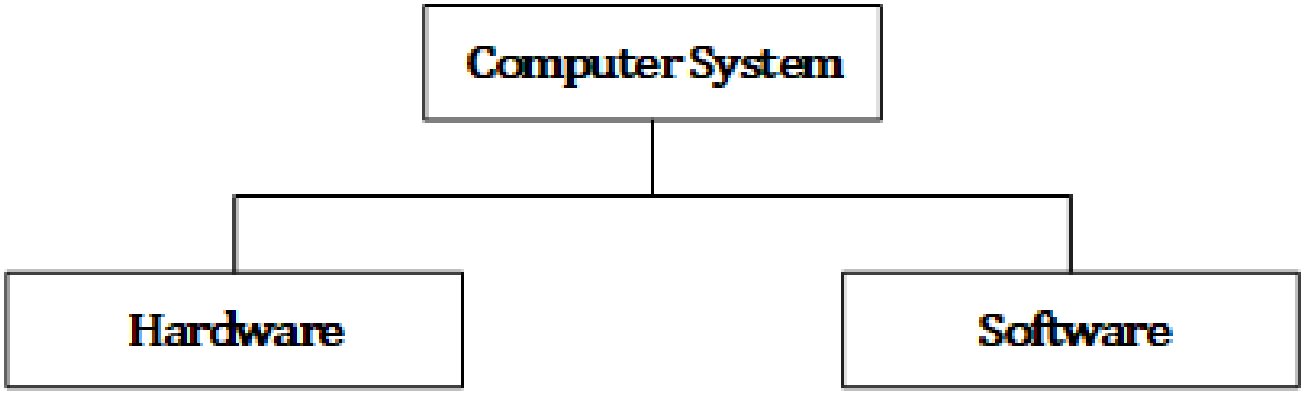
Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

What is?

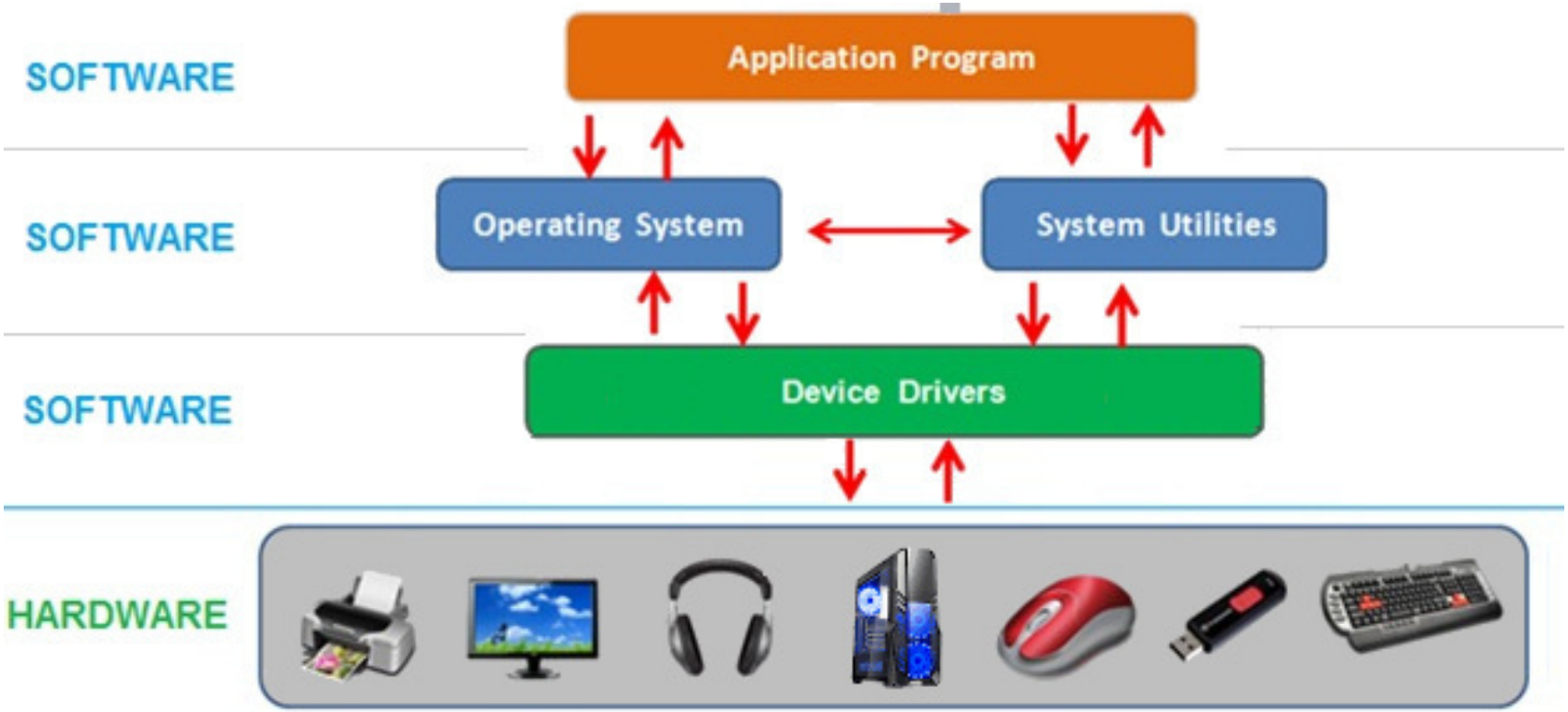
- A **computer system** consists of hardware and system software that work together to run application programs.



  - All physical components that forms computer system are known as computer **hardware**.
  - **Software** is basically collection of different programs that tells computer's hardware what to do.
- The way these components work and interact with each other affects the **correctness and performance of application programs**.

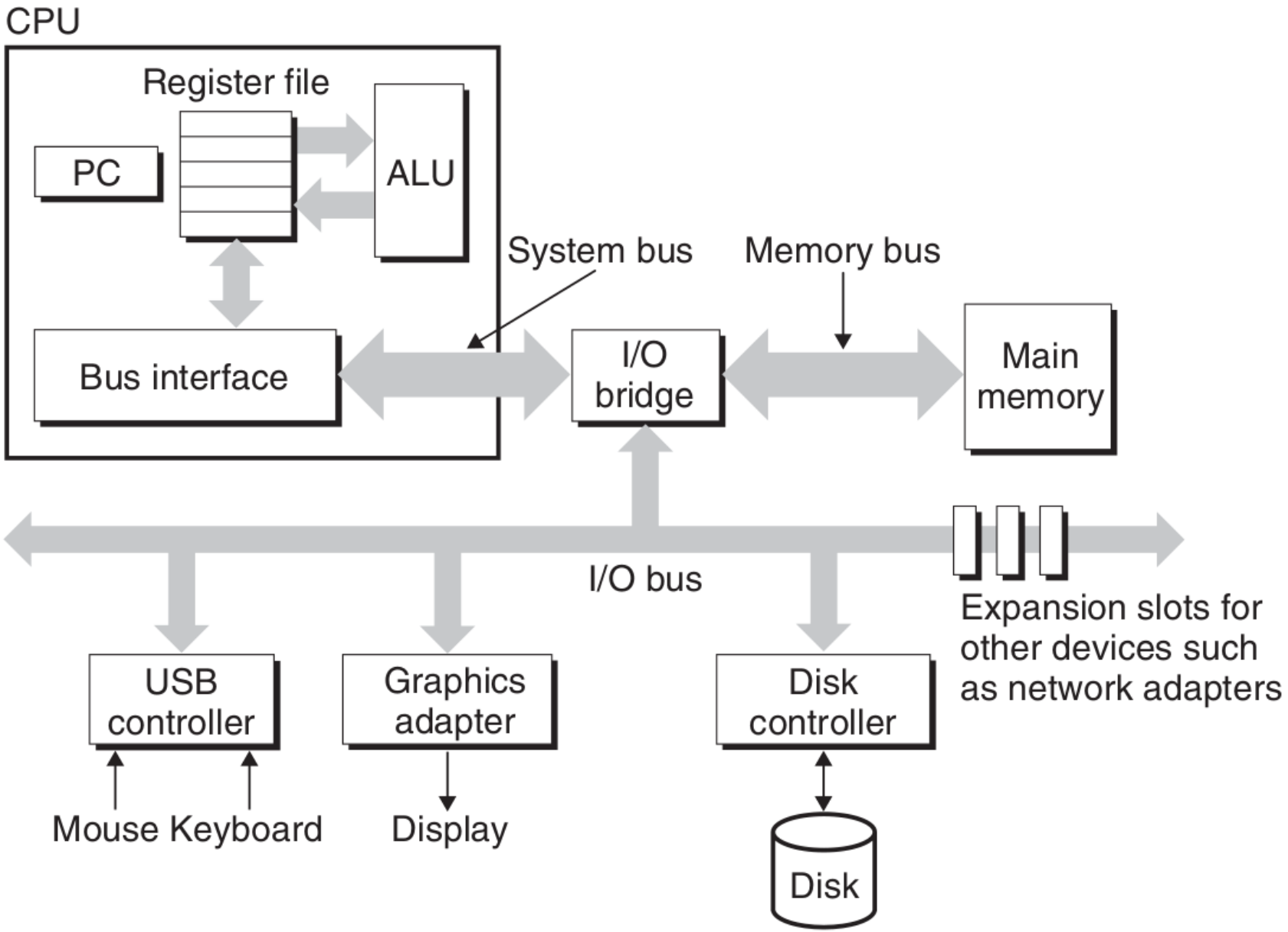
How is it structured?

- Typically, it follows a **layer architecture**.



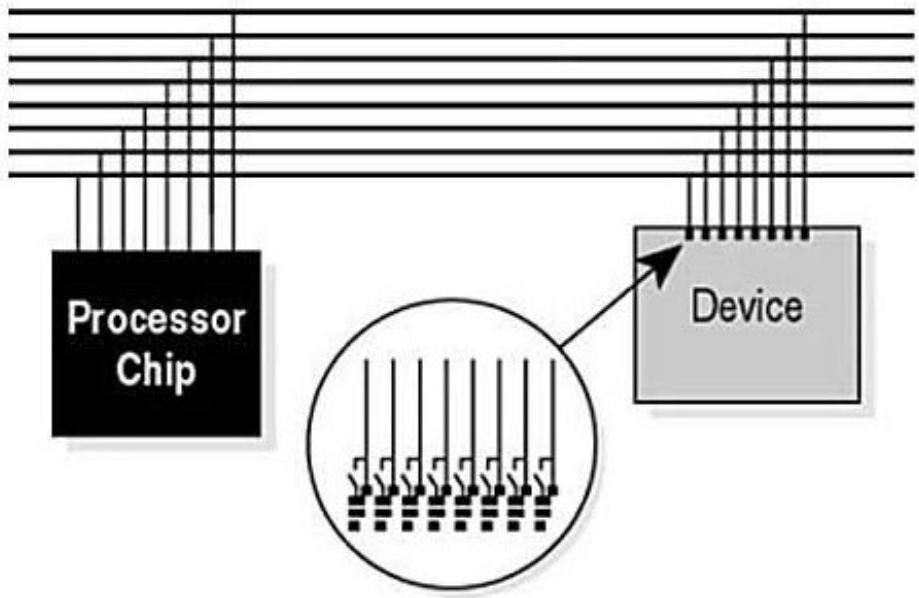
- Operating System, System Utilities, and Device Drivers are **out of scope of this course**.

Hardware Organization



Buses

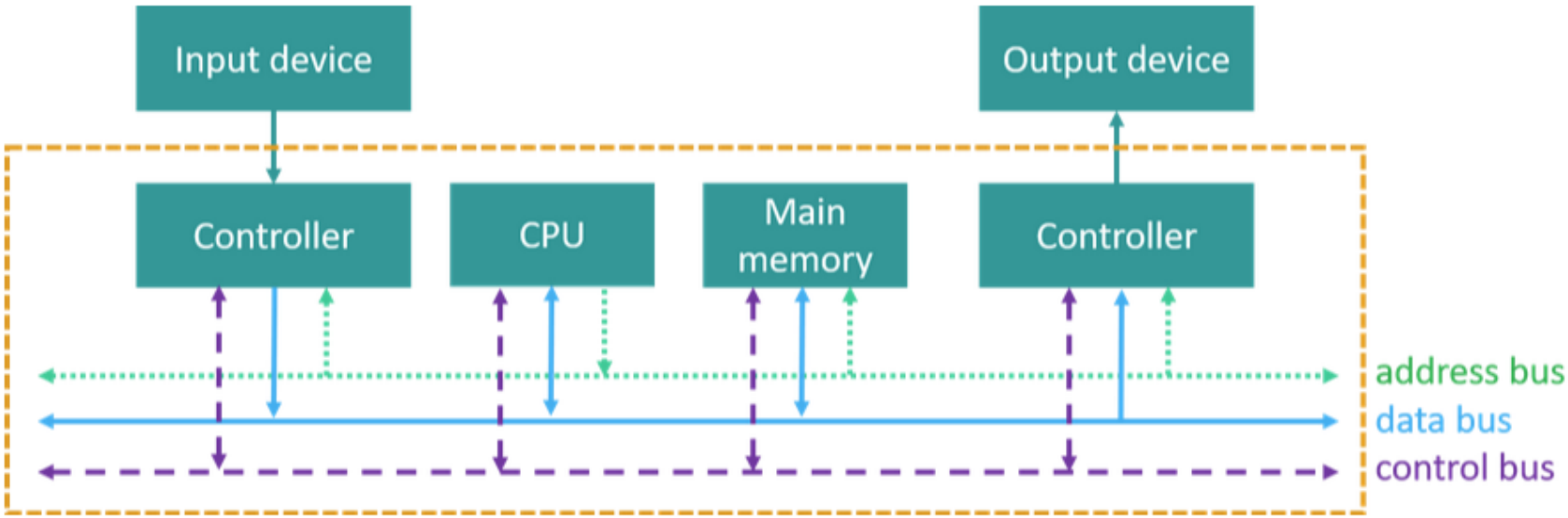
- Running throughout the system is a collection of electrical circuits that **carry bytes of information back and forth between the components**.



- Buses are typically designed to transfer fixed-sized chunks of bytes known as **words**.
  - The number of bytes in a word (the **word size**) is a fundamental system parameter that varies across systems.
  - Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits).

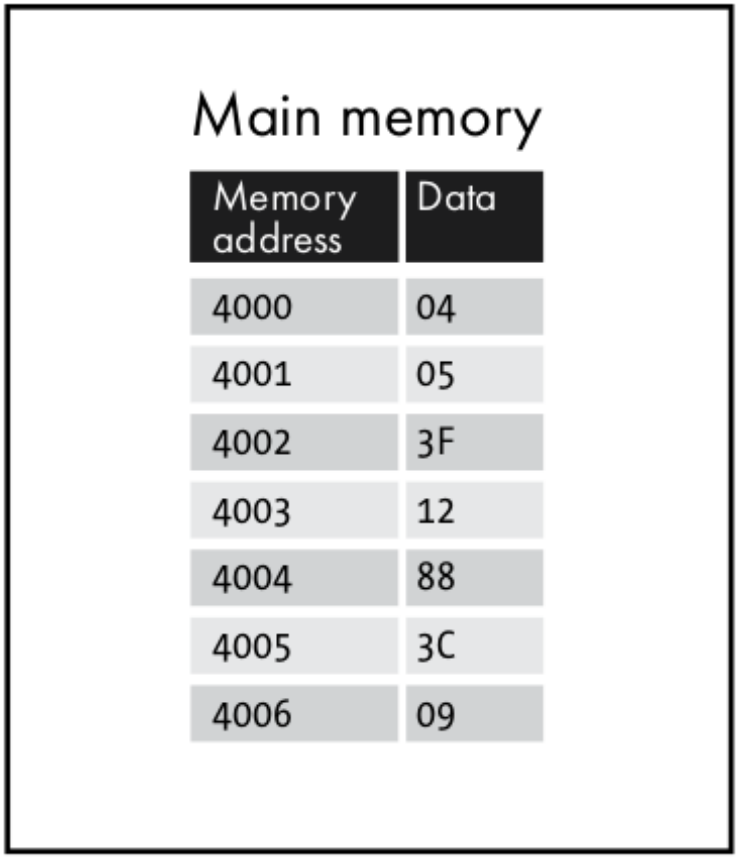
I/O Devices

- Input/output (I/O) devices are the system's connection to the external world.
- Each I/O device is connected to the I/O bus by either a **controller** or an **adapter**
  - The purpose of each is to transfer information back and forth between the I/O bus and an I/O device.



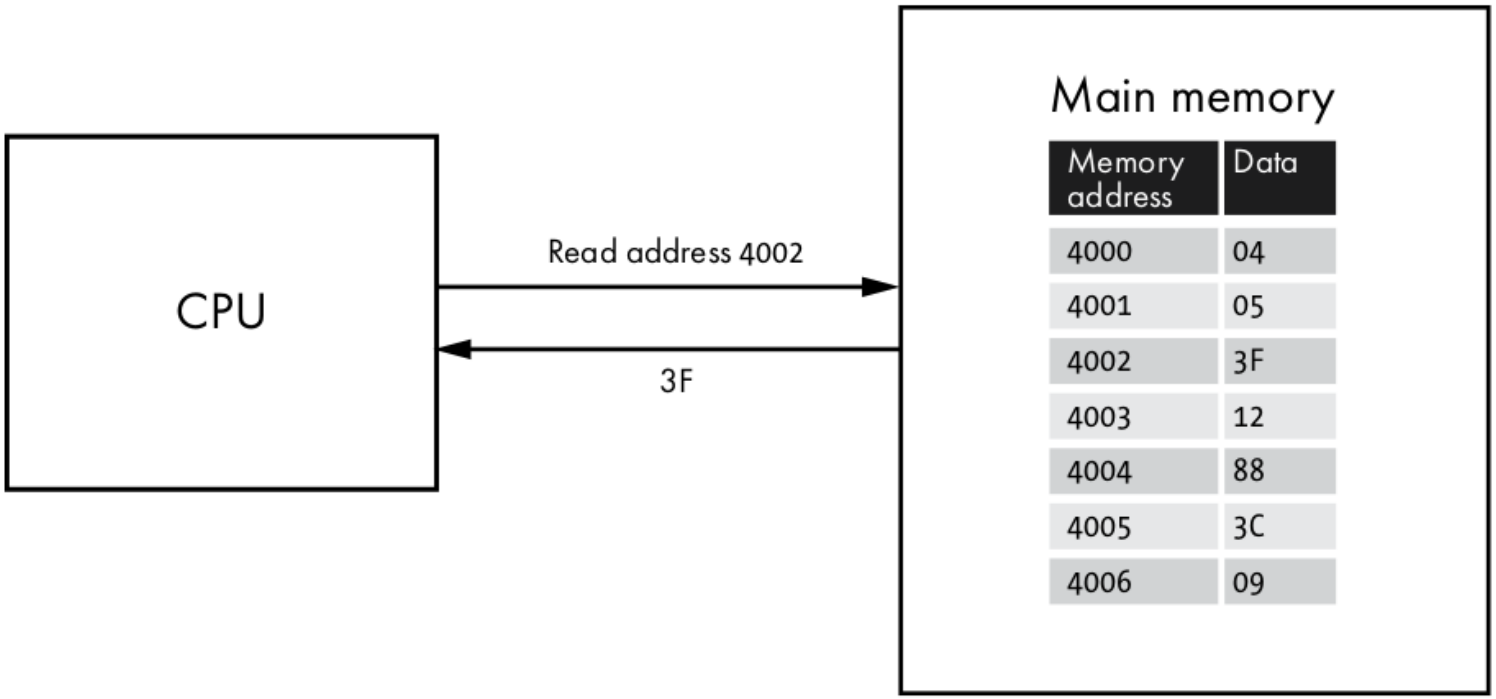
Main Memory

- The main memory is a **temporary storage device that holds both a program (instructions) and the data**.
- Logically, **memory is organized as a linear array of bytes**, each with **its own unique address** (array index) starting at zero.



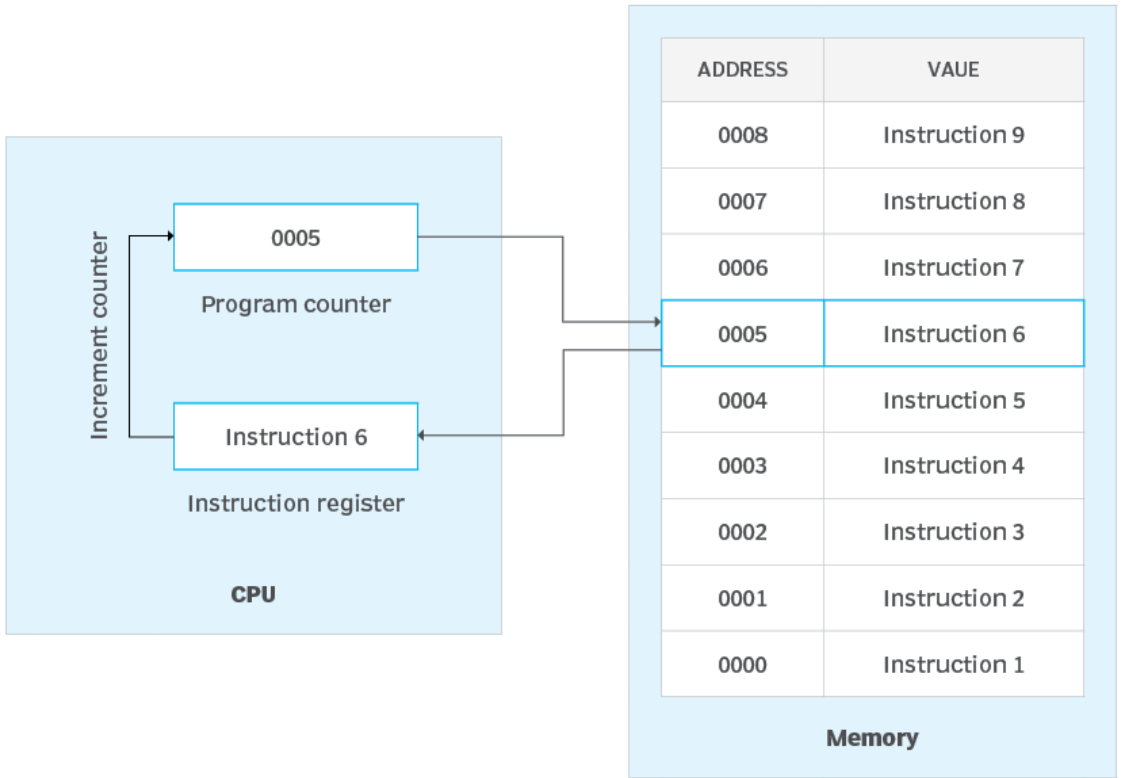
CPU (I)

- The **central processing unit** (CPU), or simply processor, is the **engine that interprets (or executes) instructions stored in main memory**.



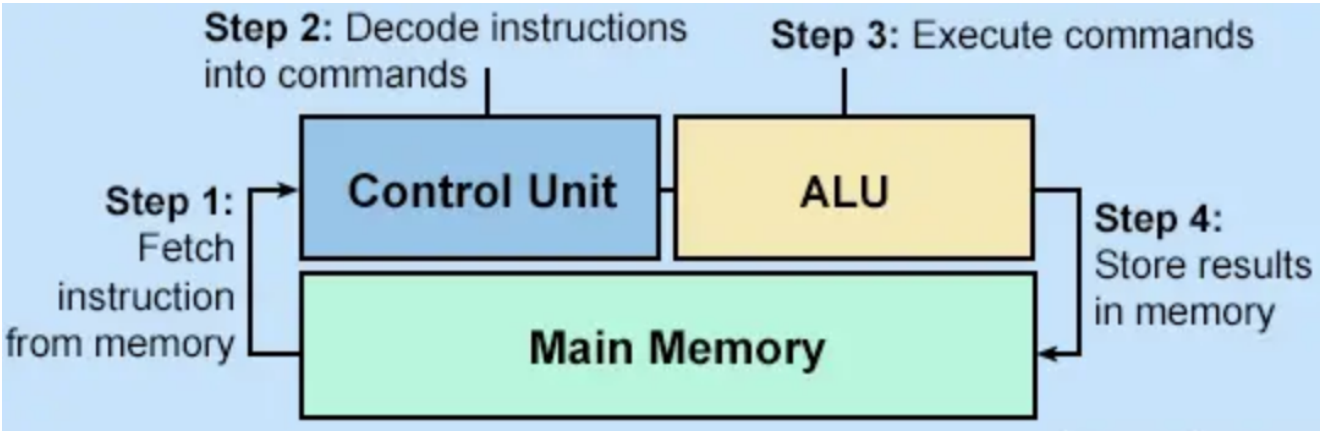
CPU (II)

- At its core is a word-sized storage device (or register) called the **program counter (PC)**.
  - PC contains the memory address (location) of the next program instruction to be executed.
  - After the CPU fetches the instruction, it increases the PC by 1 so it points to the next instruction in the program's sequence.



CPU (III)

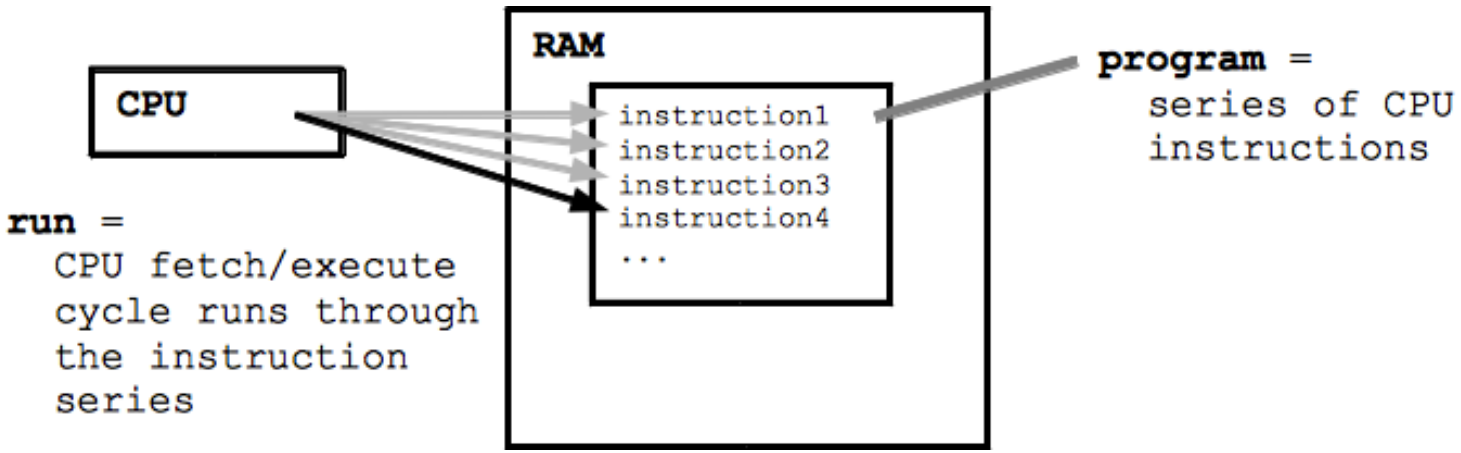
- Instruction cycle
  - The **instruction cycle is the time required by the CPU to execute one single instruction**.
  - The **instruction cycle is the basic operation** of the CPU which consists on four steps:
    - **Fetch** the next instruction from memory
    - **Decode** the instruction just fetched
    - **Execute** this instruction as decoded
    - **Store** the result



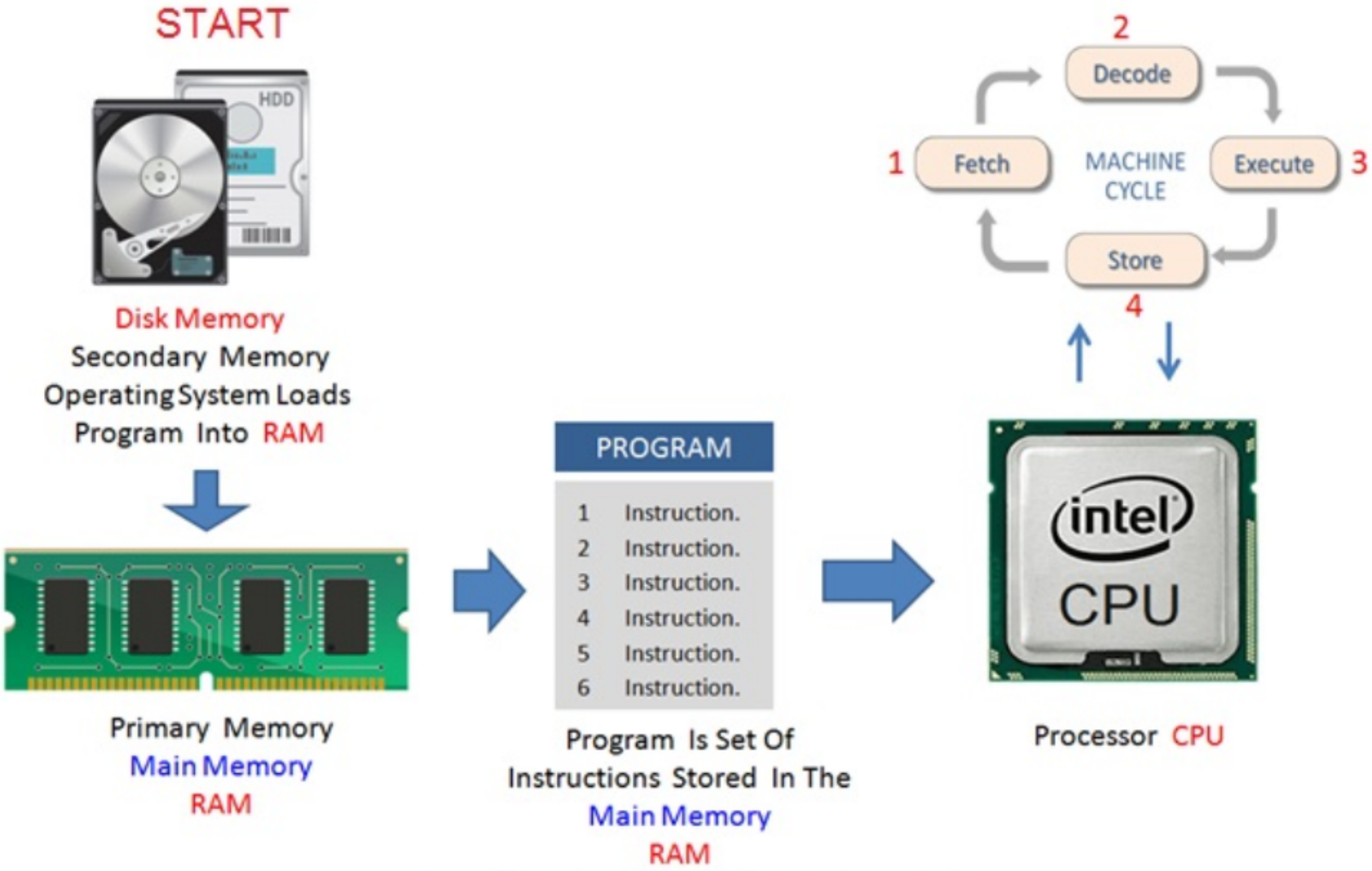
  - At the end of each instruction cycle **CPU advances PC register**.

How Computer Executes a Program (I)

- A computer **program is a file**, in which its content is a **set of CPU instructions**.
- The program **instructions are loaded into the main memory** (RAM).
- The CPU **initiates the program execution by fetching the instructions one by one from memory to registers**.
- The CPU executes these instructions **by repetitively performing an instruction cycle**.
- **At the end of each instruction cycle it increments the PC register**



How Computer Executes a Program (II)



Instruction Set Architecture (ISA)

- An ISA is part of the abstract model of a computer that defines how the CPU is controlled by the software.
- The ISA acts as an interface between the hardware and the software, specifying both **what the CPU is capable of doing** as well as **how it gets done**.
  - The ISA defines the **set of commands that the CPU can perform to execute the program instructions**.
    - Instructions are **bit-patterns**.
- Different “families” of processors, such as RISC-V, Intel x86-64, IBM/Freescale PowerPC, and the ARM processor family have **different ISAs**.
  - A program **compiled for one type of machine will not run on another**.

Source code file (I)

- When you create a program, you tell the **computer what to do**.
- Using a text editor, you create what is called a **source code** file.
  - The only special thing about this file is that it has to contain **statements according to the programming language rules**.
    - In this case, the statements are written using the C programming language, so the source code file must be saved with ".c" extension (in this case `hello.c`)

```c
#include <stdio.h>

int main()
{
   printf("hello, world\n");
   return 0;
}
```

- The source code file is a **text file**

Source code file (II)

- The `hello.c` contains **a sequence of bytes and each byte has a bit representation that corresponds to some character**.
- `xxd -b hello.c`

```
00000000: 00100011 01101001 01101110 01100011 01101100 01110101 #inclu
00000006: 01100100 01100101 00100000 00111100 01110011 01110100 de <st
0000000c: 01100100 01101001 01101111 00101110 01101000 00111110 dio.h>
00000012: 00001010 00001010 01101001 01101110 01110100 00100000 ..int
00000018: 01101101 01100001 01101001 01101110 00101000 00101001 main()
0000001e: 00001010 01111011 00001010 00100000 00100000 00100000 .{.
...
```

- `hexdump -C hello.c`

```
00000000 23 69 6e 63 6c 75 64 65 20 3c 73 74 64 69 6f 2e |#include <stdio.|
00000010 68 3e 0a 0a 69 6e 74 20 6d 61 69 6e 28 29 0a 7b |h>..int main().{|
00000020 0a 20 20 20 20 70 72 69 6e 74 66 28 22 68 65 6c |. printf("hel|
00000030 6c 6f 2c 20 77 6f 72 6c 64 5c 6e 22 29 3b 0a 20 |lo, world\n");. |
00000040 20 20 72 65 74 75 72 6e 20 30 3b 0a 7d 0a | return 0;.}.|
0000004f
```

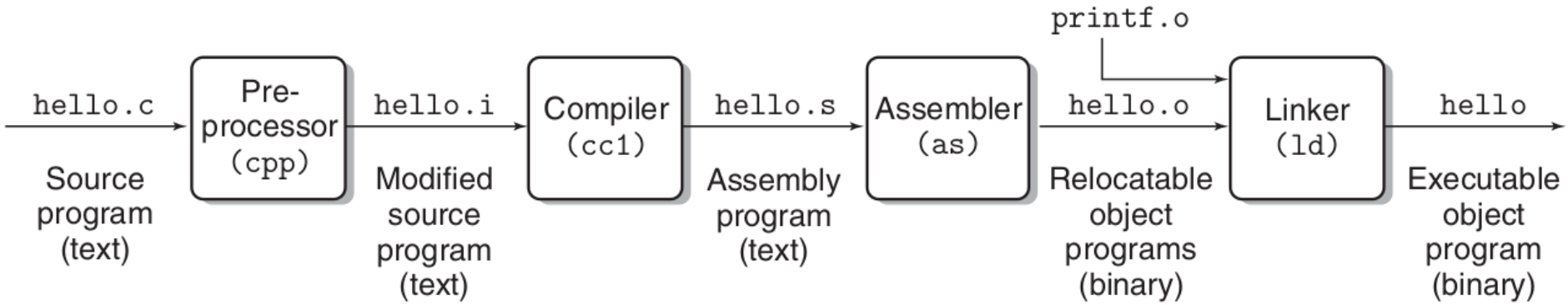
Source code file (IV)

- The `hello.c` program is a high-level C program because it can be read and understood by human beings, but not by processor.
- **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute.
- **Each type of processor** has its own set of **machine language instructions** (defined by ISA).
- To run `hello` program on the computer, the **C statements must be translated into a sequence of low-level machine-language instructions**.
- These instructions are then packaged in a form called **an executable object program and stored as a binary disk file**.
- This translation process is designated by **compilation**.
  - To perform compilation, it is required a **compiler**
    - A compiler is **a special computer program that translates a programming language's source code into machine code** (according to correspondent ISA).

Compilation

- GNU Compiler Collection (GCC) is a **free and open source set of compilers and development tools** for C, C++ and other programming languages.
  - It is available for Linux, Windows and other operating systems.
- For compiling `hello` program

  > `gcc -o hello hello.c`
  - The gcc compiler **reads the source file `hello.c`** and **translates it into an executable object file `hello`**.
  - The translation is carried out **in a sequence of four stages**



Compilation Phases (I)

1. **Preprocessing phase**.
   - The preprocessor (`cpp`) modifies the original C program according to directives that begin with the # character.
     - For example, the `#include <stdio.h>` command in line 1 of `hello.c` tells the preprocessor to read the contents of the system header file `stdio.h` and insert it directly into the program text.
   - The result is another **text file**, source file C program, typically with the `.i` suffix.
2. **Compilation phase**.
   - The compiler (`cc1`) **translates the text file `hello.i` into the text file `hello.s`**, which contains an **assembly-language program**.
     - Each statement in an assembly-language program exactly describes one low-level machine-language instruction (according to the ISA) in a standard text form.

Compilation Phases (II)

3. **Assembly phase.**
   - The assembler (`as`) **translates `hello.s` into machine- language instructions**, packages them in a form known as a relocatable object program, and stores the result in the object file `hello.o`.
   - The `hello.o` file is **a binary file whose bytes encode machine language instructions (according to the ISA)** rather than characters.
4. **Linking phase.**
   - The linker (`ld`) links/**merges the object code with the library code to produce an executable file**.
     - Notice that our hello program calls the `printf` function, which is part of the standard C library provided by every C compiler.
     - The `printf` function resides in a separate precompiled object file called `printf.o`, which must somehow be merged with our `hello.o` program.
     - The result is the `hello` file, which is an executable object file (or simply executable) that is ready to be loaded into memory and executed by the system.

Executable Object File (I)

- The `hello` program **is stored in a file as a sequence of bytes and each byte (or a set of bytes) has a bit representation that corresponds to some machine instruction.**
- `xxd -b hello`

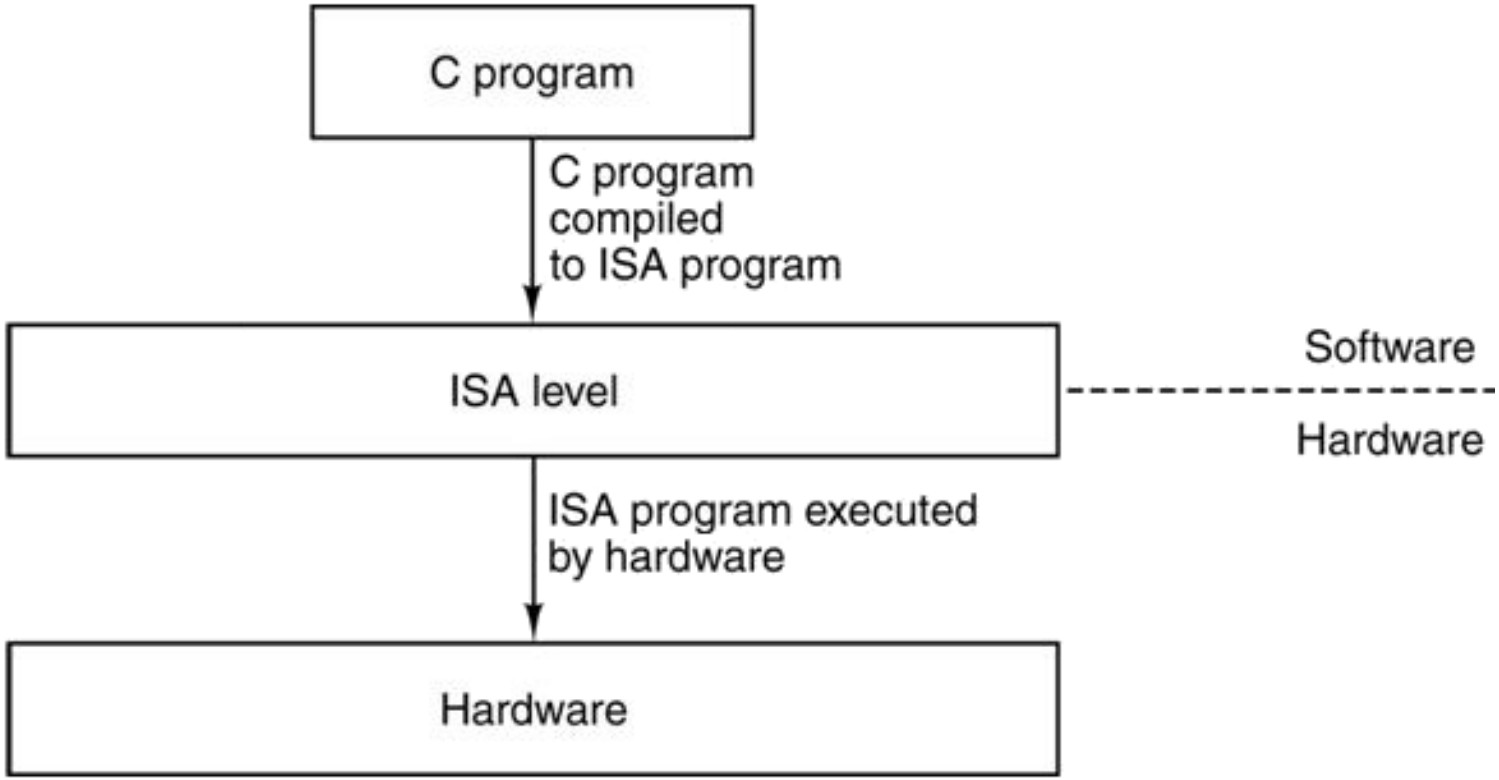
```
00000000: 01111111 01000101 01001100 01000110 00000010 00000001 .ELF..
00000006: 00000001 00000000 00000000 00000000 00000000 00000000 ......
0000000c: 00000000 00000000 00000000 00000000 00000011 00000000 ......
00000012: 00111110 00000000 00000001 00000000 00000000 00000000 >.....
00000018: 01100000 00010000 00000000 00000000 00000000 00000000 `.....
0000001e: 00000000 00000000 01000000 00000000 00000000 00000000 ..@...
00000024: 00000000 00000000 00000000 00000000 01111000 00111001 ....x9
0000002a: 00000000 00000000 00000000 00000000 00000000 00000000 ......
...
```

- `hexdump -C hello`

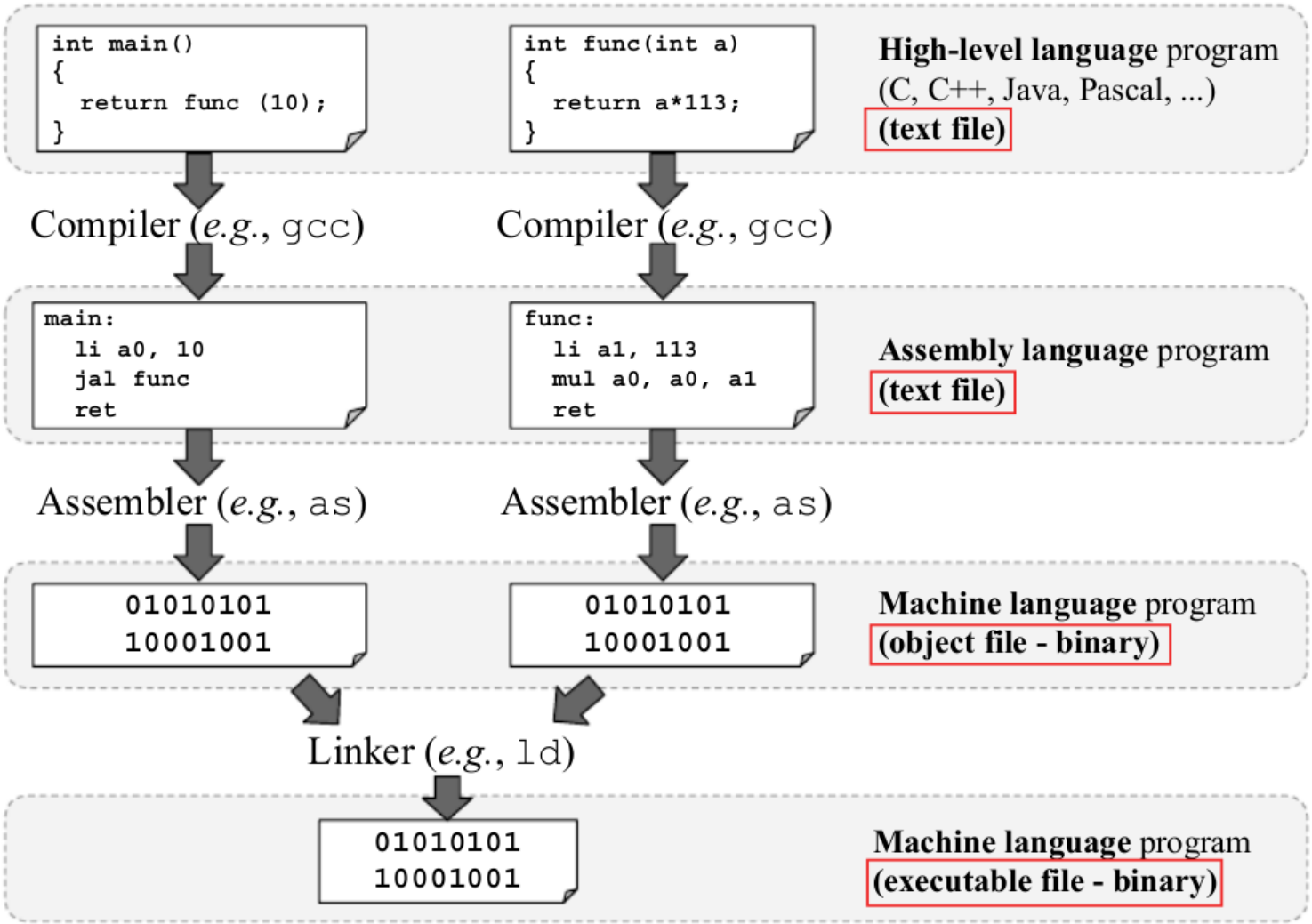
```
00000000 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00 |.ELF............|
00000010 03 00 3e 00 01 00 00 00 60 10 00 00 00 00 00 00 |..>.....`.......|
00000020 40 00 00 00 00 00 00 00 78 39 00 00 00 00 00 00 |@.......x9......|
00000030 00 00 00 00 40 00 38 00 0d 00 40 00 1f 00 1e 00 |....@.8...@.....|
00000040 06 00 00 00 04 00 00 00 40 00 00 00 00 00 00 00 |........@.......|
00000050 40 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 |@.......@.......|
...
```

Executable Object File (II)

- The content of an executable object is a **set of CPU instructions** (according to ISA).



Compilation process



Types

Feature	Text File	Binary File
Readability	Human-readable	Requires specific software
Data Type	ASCII or Unicode characters	Any data type (machine code, images, audio, etc.)
File Size	Generally larger due to character encoding	More compact, efficient storage
Use Cases	Source code files, configuration files, script files, and etc.	Object files, executable files, media files and etc.

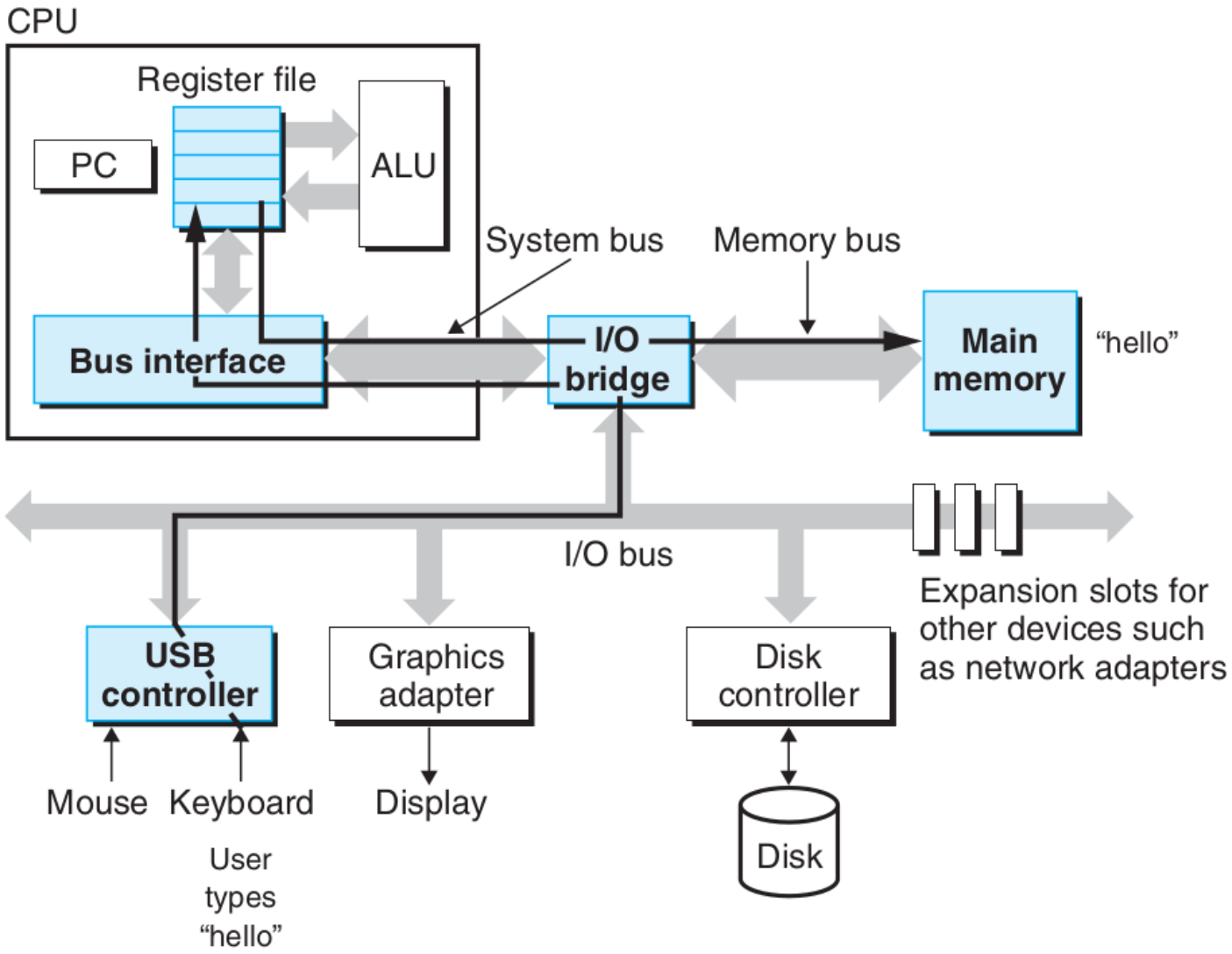
Object *vs* Executable Binary Files

- Object files are produced as the output of the assembler.
  - Typically, they consist of **function definitions in binary form**, but they are not executable by themselves.
    - Object files end in `.o` by convention, although on some operating systems (e.g. Windows, MS-DOS), they often end in `.obj`.
- Executable files are produced as the output of the linker.
  - The linker links together a number of object files to produce a binary file which can be directly executed.
  - To be an executable file, requires **an entry point**, i.e. , the point from which the CPU must start executing the program.
    - Once the operating system loads the program into the main memory, **it sets the PC with the entry point address** so the program starts executing.
    - Binary executables **have no special suffix** on Unix operating systems, although they generally end in `.exe` on Windows.

Executing (I)

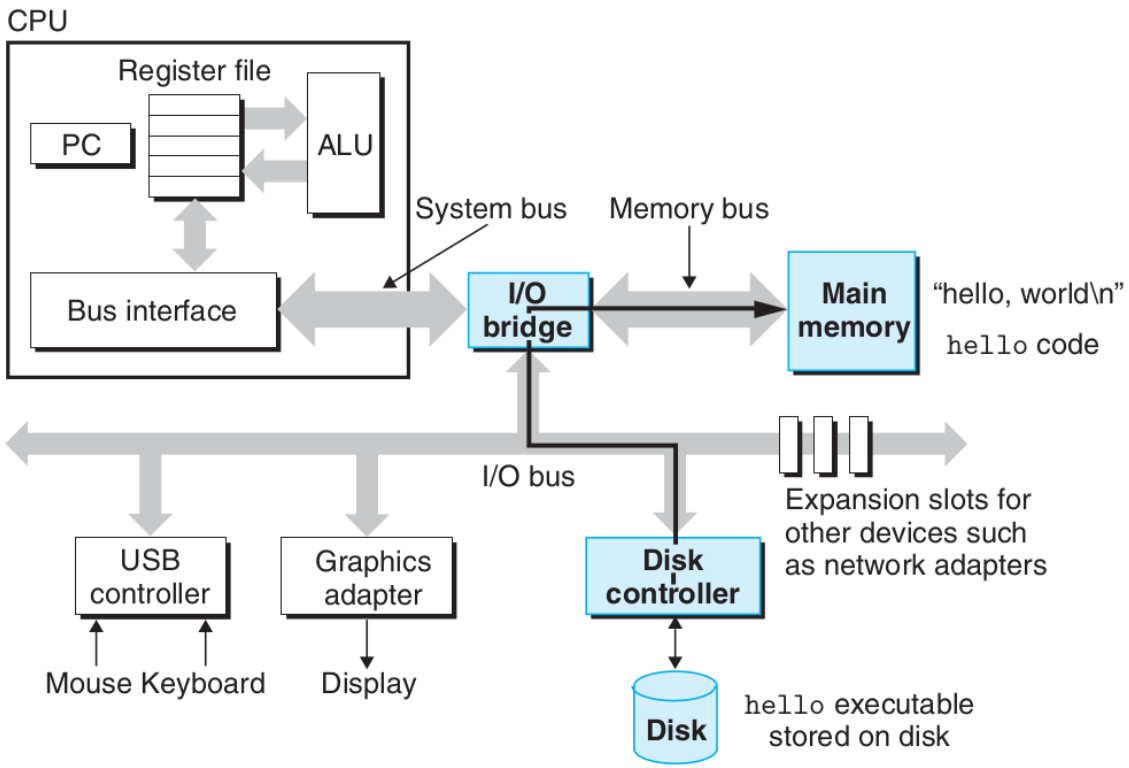
- In order to run the executable file on a Unix derived system, we type its name to an application program known as a **shell**:
  `> ./hello`
  - **A shell is a program that takes commands from the keyboard and gives them to the operating system to perform**.
- As we type the characters `./hello` at the keyboard, the shell program reads each one into a register, and then stores it in memory.
- When we hit the `enter` key on the keyboard, the shell knows that we have finished typing the command.

Executing (II)



Executing (III)

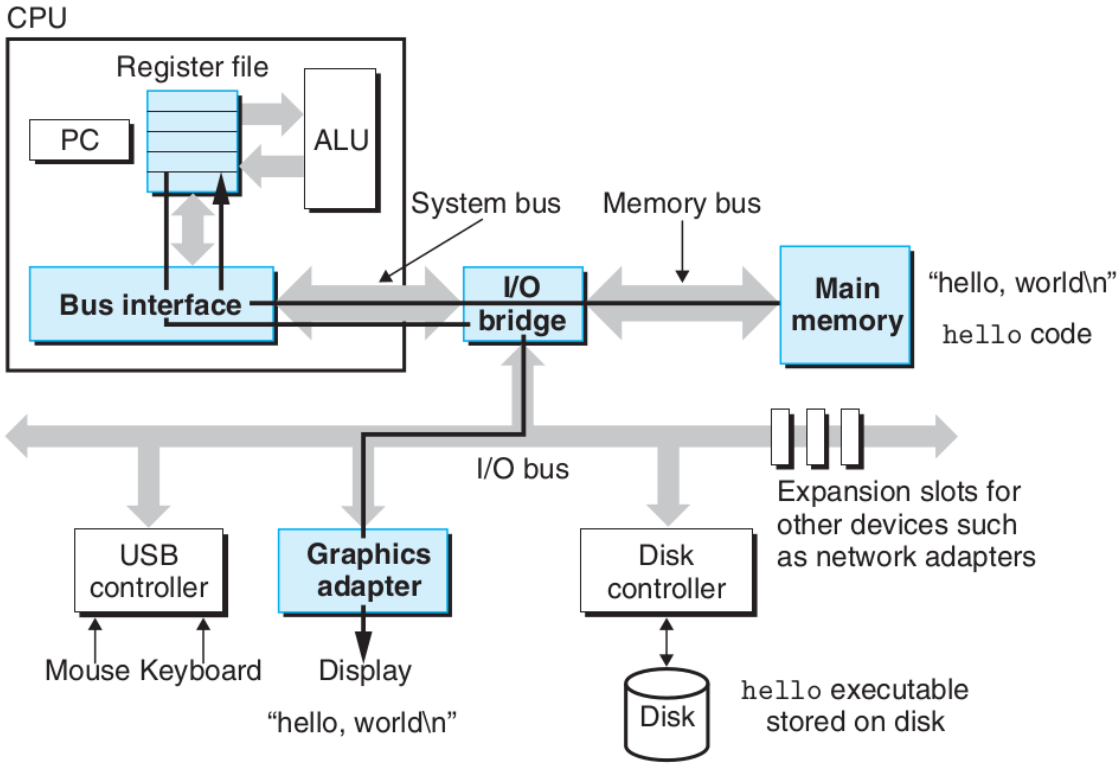
- A set of operations are executed by operating systems that **copies the contents of the `hello` object file from disk to main memory**.



  - Using a technique known as direct memory access (DMA), the data travels directly from disk to main memory, without passing through the CPU.

Executing (IV)

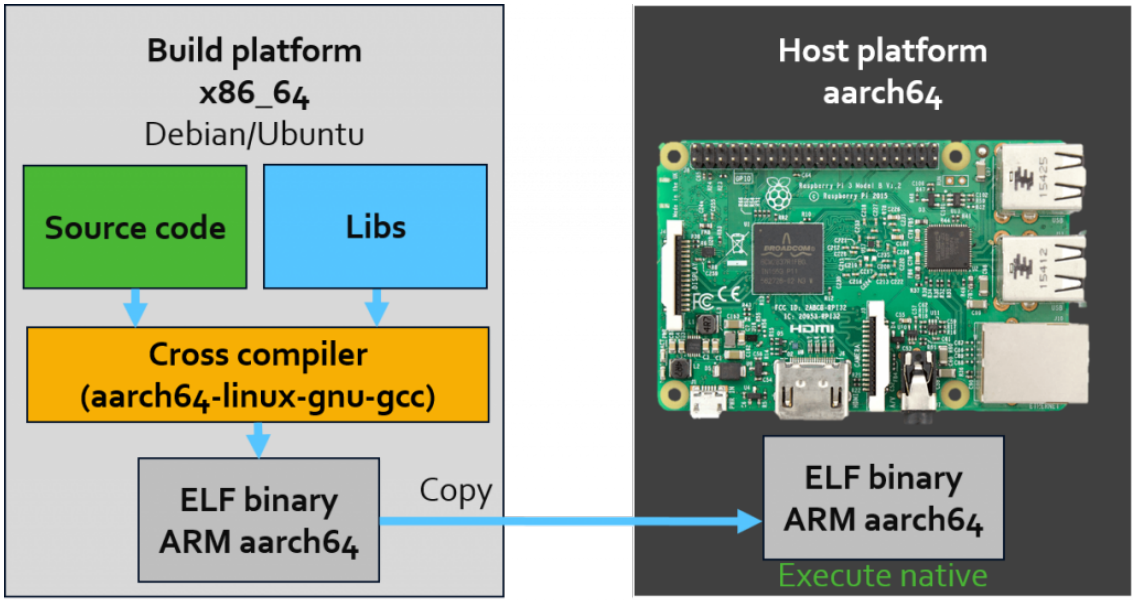
- Once the contents of the `hello` **object file are loaded into memory**, the **CPU begins executing the machine-language instructions** in the `hello` program's `main` routine.



  - These instructions copy the bytes in the `hello, world\n` string from memory to the register file, and from there to the display device, where they are displayed on the screen.

What is?

- The act of compiling applications to run on a different computer system (with different ISA) is referred to as **cross compiling**.
  - A cross-compiler is a compiler that runs on platform A (the host), but generates executables for platform B (the target).
  - These two platforms may (but do not need to) differ in CPU, operating system, and/or executable format.



- This is oppose to **native compiling** where what you compiled is supposed to run on the same system that you compiled it on.

Target triplet (I)

- Cross-compiling means that **up to three differently-targeted compilers** might be in play:
  - If you are building a GCC on platform A, which will run on platform B, which produces executables for platform C.
- Target triplets let **build systems understand exactly which system the code will run on and allows enabling platform-specific features automatically**.
  - **Build Platform**: This is the platform on which the compilation tools are executed.
  - **Host Platform**: This is the platform on which the code will eventually run.
  - **Target Platform**: If this is a compiler, this is the platform that the compiler will generate code for.

Target triplet (II)

- A target triplet is usually specified at the command line as a - delimited string in the format:
  - `<arch>-<vendor>-<operatingsystem>`
    - The `<operatingsystem>` may be further divided into an optional operating system application binary interface (ABI) format.
    - The `<vendor>` field can be omitted or specified as `none` or `unknown`.
- **GCC prefixes the compilation tools with the target triplet.**
  - This prevents the wrong compiler from being used (and prevents things from the build machine leaking onto the target machine)
  - Examples:
    - `riscv32-linux-gcc`
    - `arm-none-eabi-gcc`
    - `riscv64-unknown-elf-gcc`
- The `gcc -dumpmachine` command prints the compiler's target triplet of the machine

```
→  ~ gcc -dumpmachine
x86_64-linux-gnu
→  ~
```

QEMU

- Emulators are defined **as hardware or software platforms** that allow **a computer system to behave like another** so that the former – the host – can run applications and services designed for the latter – the guest.
- QEMU, short for “Quick Emulator”, is a generic and open source machine emulator and virtualizer.
  - When used as a machine emulator, QEMU **can run OSes and programs made for one machine** (e.g. an ARM board) **on a different machine** (e.g. your x86 PC).