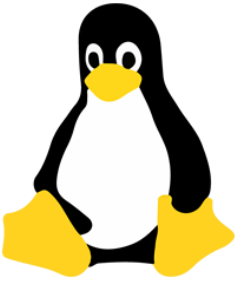


CS416: Systems programming

Abdullah Alfarrarjeh

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

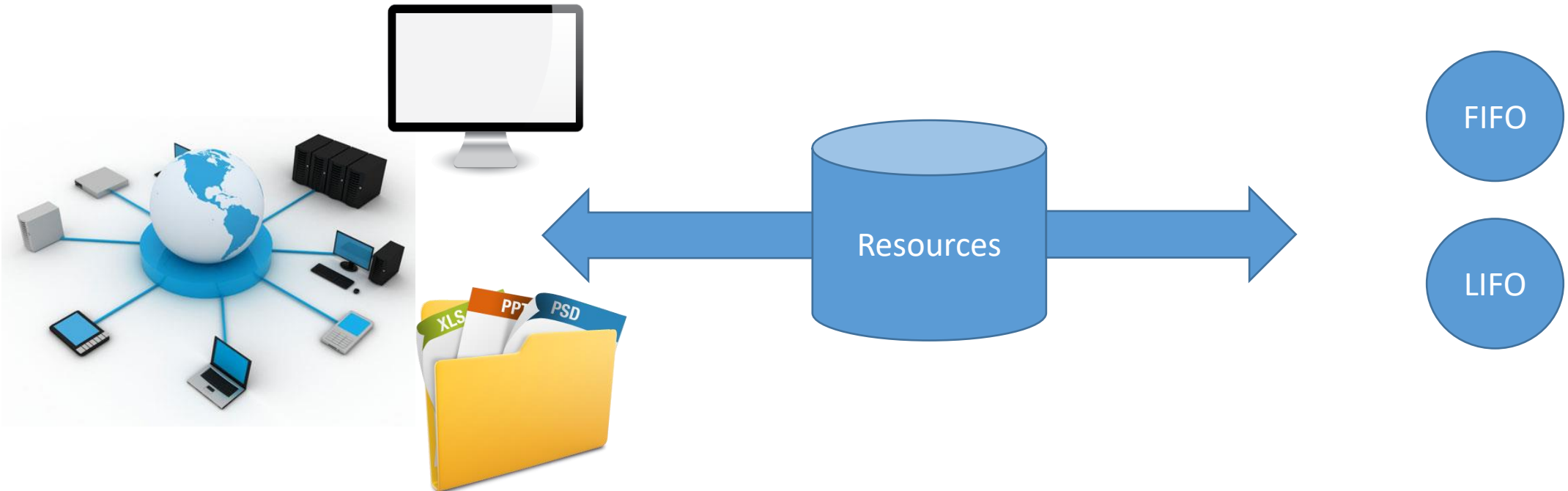
A computer system

- Is a collection of hardware and software components that work together to run computer programs.
- Specific implementations of systems change over time, but the underlying concepts do not.
- All systems have similar hardware and software components/ resources that have similar roles.

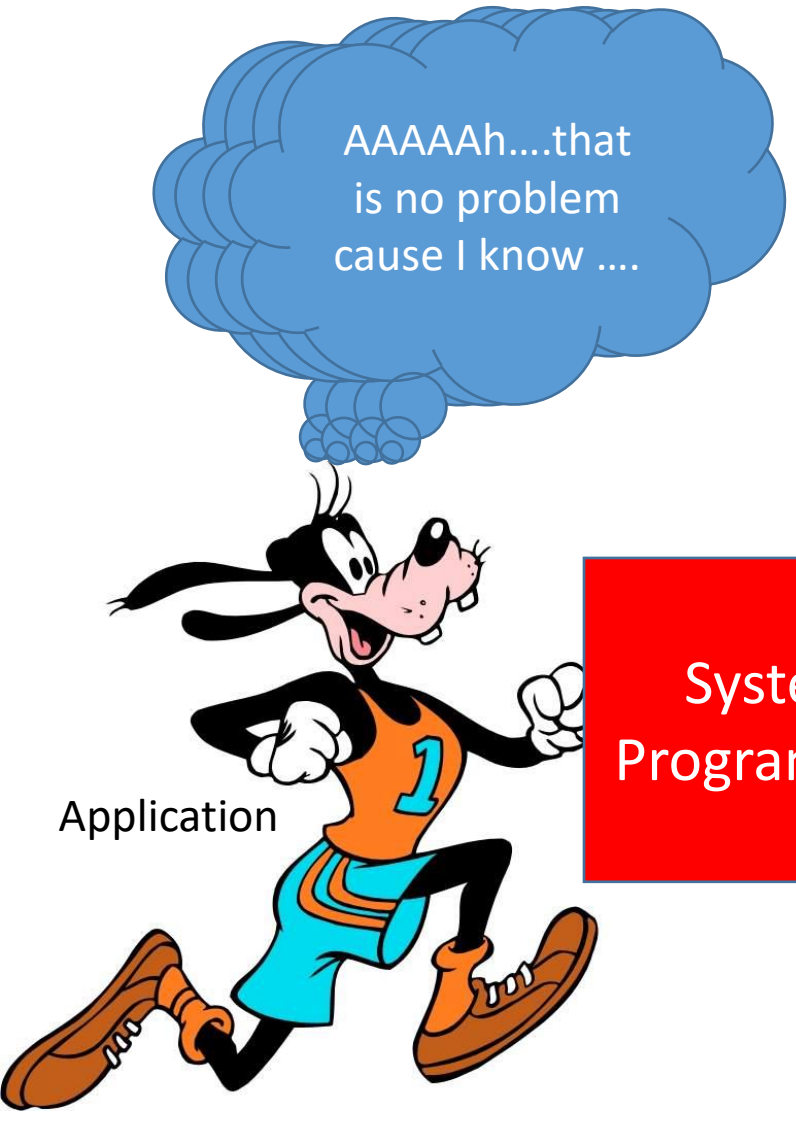
Types of resources

Two main resources should be managed by modern software applications

- **Private resources:** like data structures (Hash tables, Stacks, queues,…)
- **System resources:** files, databases, network, displays,…



Application programming interfaces (APIs)



Systems Programming

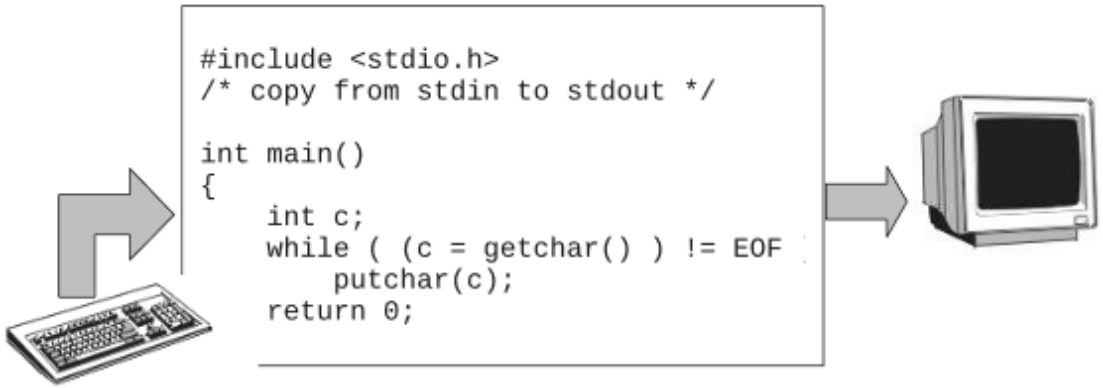
API

Sure…but
through my API
if you can…
TRAAAAAA

System resources

System Programming

- An API typically consists of a collection of functions.
- Software applications are ……NOT…… allowed to use system resources directly.
- The OS interface that applications use to access resources is called ……API…….
- The program that uses the API system services is called a……Systems program…… and the type of programming is called ……Systems programming……



- Although it may seem that functions from the C standard library access resources directly, they do not; they make calls to system routines that do the work on their behalf

Let’s start from the basics.
Information are bits

- Data is a sequence of bits, each with a value of 0 or 1, organized in 8-bit chunks called bytes
- Modern systems represent text characters using the ASCII (American Standard Code for Information Interchange) standard that represents each character with a unique byte-sized integer value

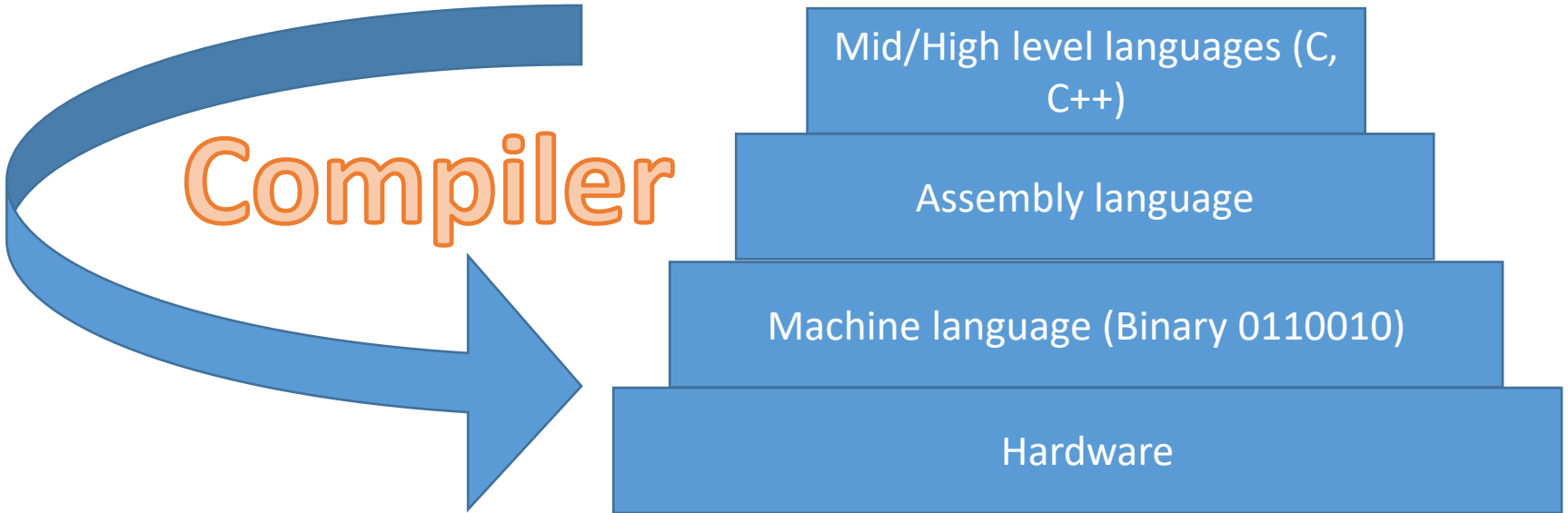
How you see it	How it is actually in ASCII
#include	# i n c l u d e 35 105 110 99 108 117 100 101

- All information in a system — including disk files, programs stored in memory, user data stored in memory, and data transferred across a network — is represented as a bunch of bits

What about programming languages??

Programs are translated into different forms
C language

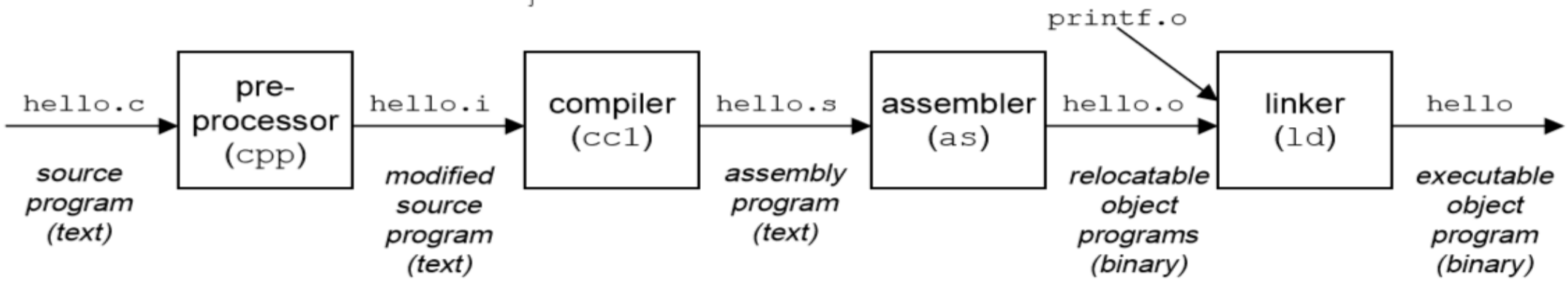
- People can read and understand the C programs in the standard form
- The program must be translated into a set of low-level machine instructions
- These instructions are then packaged in a form called an executable object program and stored as a binary disk file.



GCC compiler for example

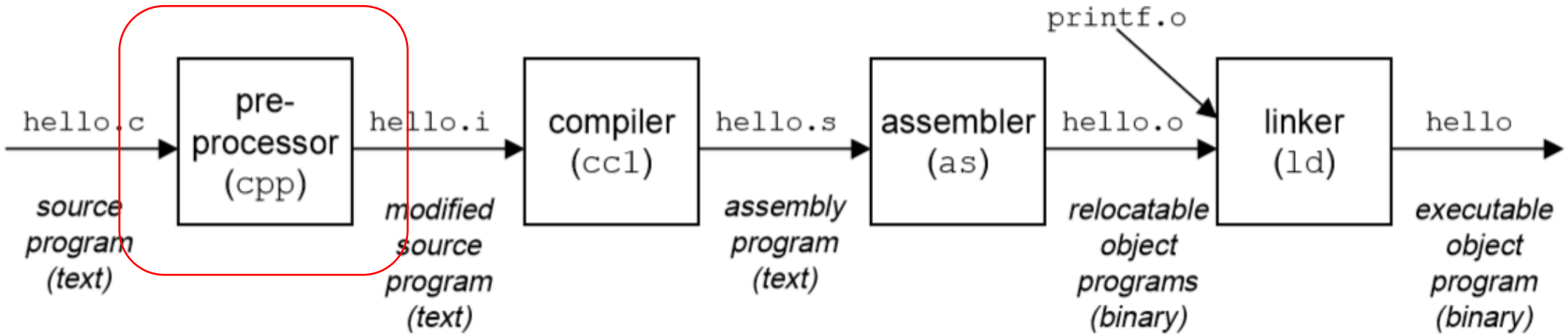
- The GCC compiler driver reads the source file hello.c and translates it into an executable object file hello.
- The translation is performed in a sequence of four phases
- The programs that perform the four phases (preprocessor, compiler, assembler, linker) are known collectively as the compilation system.

```
#include <stdio.h>

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



The compilation system.

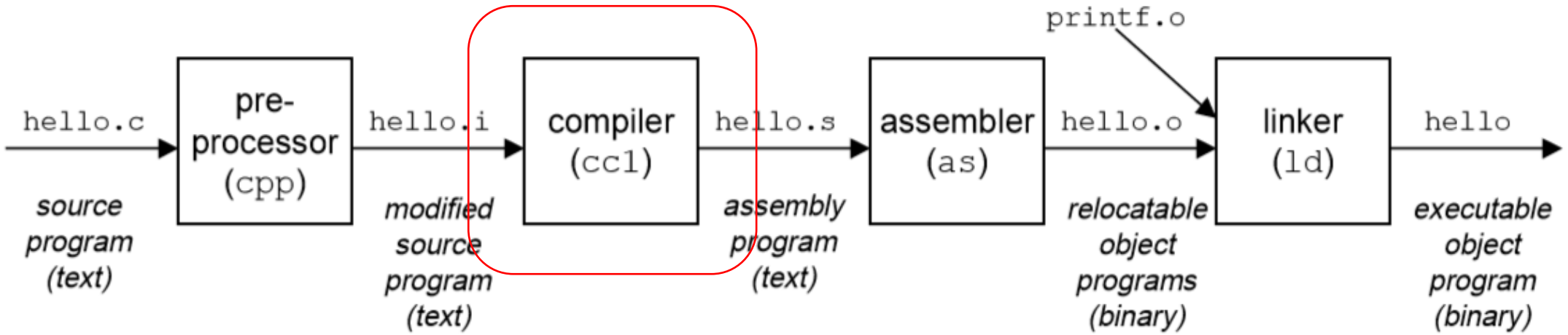
How GCC compiler works (contd.)



The compilation system.

- **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 C program, typically with the .i suffix.

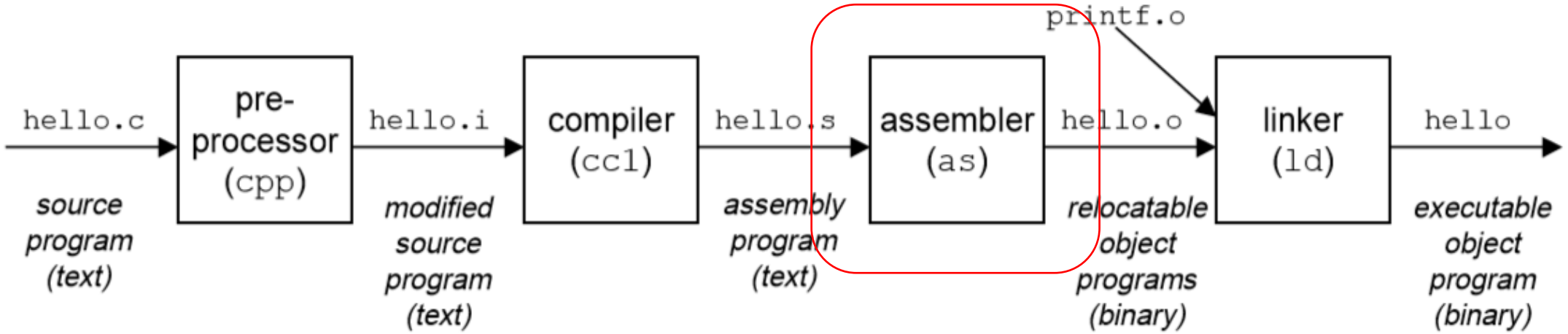
How GCC compiler works (contd.)



The compilation system.

- **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 in a standard text form. Assembly language is useful because it provides a common output language for different compilers for different high-level languages. For example, C compilers and Fortran compilers both generate output files in the same assembly language.

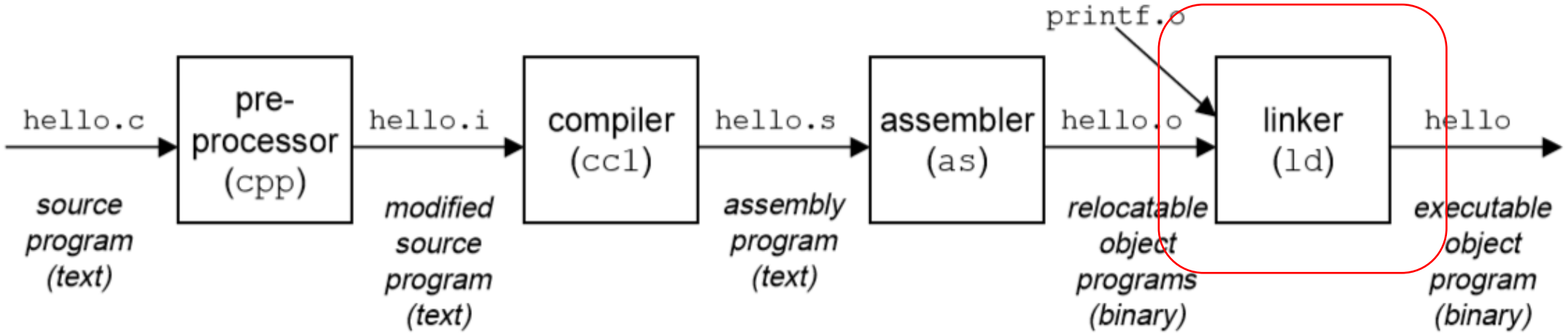
How GCC compiler works (contd.)



The compilation system.

- **Assembly phase.** Next, 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 rather than characters. If we were to view hello.o with a text editor, it would appear to be gibberish (meaningless).

How GCC compiler works (contd.)



The compilation system.

- Linking phase: The 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 linker (ld) handles this merging. The result is an object that's is ready to be loaded into memory and executed by the system.

Running the hello.c program

- To run the executable on a Unix system, we type its name to an application program known as a *shell command* line interpreter

unix>./hello

hello, world

unix>

The hello program prints its message to the screen and then terminates. The shell then prints a prompt and waits for the next input command line.

Using the gcc compiler

hello.c

```c
#include <stdio.h>

int main(void) {
   int x = 3;

   printf("Hello World\n");
   printf("x = %d\n", x);
   return 0;
}
```

Terminal

```
(base) dell@idl:~/CS415_Fall2022$ gcc hello.c -o hello.o
(base) dell@idl:~/CS415_Fall2022$ ./hello.o
Hello World
x = 3
(base) dell@idl:~/CS415_Fall2022$
```

- To compile a C program (e.g., hello.c), you can use the “-o” to declare the name of the compilation output file (e.g., hello.o). The compilation output file is an executable file.
  - **e.g., >> gcc hello.c -o hello.o**
- The following command shows how to run the executable file of the program.
  - **e.g., >>./hello.o**

Using the gcc compiler: Stopping the compilation after the preprocessing stage

- To stop the compilation of a C program (e.g., hello.c), you can use the “-E” option.
  - e.g., >> gcc hello.c -o hello_stage1.o -E

Using the gcc compiler: Stopping the compilation after the preprocessing stage

The input is a C program of 9 lines. The output is a C program of 802 lines.

★ The content of stdio.h is appended

1 Function prototype

2 Original Content of the C program

```
(base) dell@idl:~/CS415_Fall2022$ gcc hello.c -o hello_stage1.o -E
(base) dell@idl:~/CS415_Fall2022$ cat hello_stage1.o
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 424 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 427 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 428 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 429 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 425 "/usr/include/features.h" 2 3 4
# 448 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 449 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4



# 1 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 1 3 4
# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4

# 216 "/usr/lib/gcc/x86_64-linux-gnu/7/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
# 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4


typedef unsigned char __u_char;
typedef unsigned short int __u_short;
```

```
extern int fprintf (FILE *__restrict __stream,
      const char *__restrict __format, ...);

1

extern int printf (const char *__restrict __format, ...);

extern int sprintf (char *__restrict __s,
      const char *__restrict __format, ...) __attribute__ ((__nothrow__));



extern int vfprintf (FILE *__restrict __s, const char *__restrict __format,
      __gnuc_va_list __arg);



extern int vprintf (const char *__restrict __format, __gnuc_va_list __arg);

extern int vsprintf (char *__restrict __s, const char *__restrict __format,
      __gnuc_va_list __arg) __attribute__ ((__nothrow__));



extern int snprintf (char *__restrict __s, size_t __maxlen,
      const char *__restrict __format, ...)
     __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3
, 4)));

extern int vsnprintf (char *__restrict __s, size_t __maxlen,
       const char *__restrict __format, __gnuc_va_list __arg)
     __attribute__ ((__nothrow__)) __attribute__ ((__format__ (__printf__, 3
, 0)));
# 365 "/usr/include/stdio.h" 3 4
extern int vdprintf (int __fd, const char *__restrict __fmt,
      __gnuc_va_list __arg)
     __attribute__ ((__format__ (__printf__, 2, 0)));
extern int dprintf (int __fd, const char *__restrict __fmt, ...)
     __attribute__ ((__format__ (__printf__, 2, 3)));
```

```
extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__))
;



extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __
leaf__)) ;
# 800 "/usr/include/stdio.h" 3 4
extern FILE *popen (const char *__command, const char *__modes) ;



extern int pclose (FILE *__stream);



extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf_
_));


extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __lea
f__)) ;


extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __lea
f__));
# 868 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

2

# 3 "hello.c"
int main(void) {
  int x = 3;

  printf("Hello World\n");
  printf("x = %d\n", x);
  return 0;
}
```

Using the gcc compiler: Stopping the compilation after the second stage

- To stop the compilation of a C program (e.g., hello.c) after the second stage, you can use the “-S” option.
  - e.g., >> gcc hello.c -o hello_stage2.o -S

Using the gcc compiler: Stopping the compilation after the second stage

The output is an assembly program.

1. Original Content of the C program
2. Assembly program of hello.c

1

```
(base) dell@idl:~/CS415_Fall2022$ cat hello.c
#include <stdio.h>

int main(void) {
  int x = 3;

  printf("Hello World\n");
  printf("x = %d\n", x);
  return 0;
}
```

2

```
(base) dell@idl:~/CS415_Fall2022$ gcc hello.c -o hello_stage2.o -S
(base) dell@idl:~/CS415_Fall2022$ cat hello_stage2.o
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello World"
.LC1:
        .string "x = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $3, -4(%rbp)
        leaq    .LC0(%rip), %rdi
        call    puts@PLT
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
        .section        .note.GNU-stack,"",@progbits
```

Using the gcc compiler: Stopping the compilation after the assembly stage

- To stop the compilation of a C program (e.g., hello.c) after the assembly stage, you can use the “-C” option.
  - e.g., >> gcc hello.c -o hello_stage3.o **-C**

- The output of the assembly stage is an object file which contains binary code. To decode this output file, you can use the “objdump” command as follows.
  - E.g., >>objdump -D hello_stage3.o > hello_stage3_decoded.o

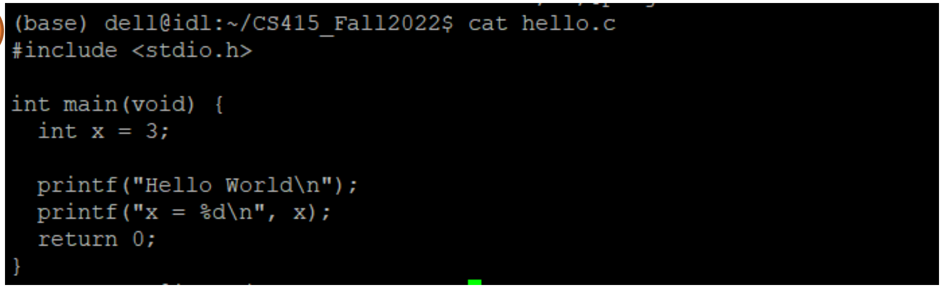
Using the gcc compiler: Stopping the compilation after the assembly stage

The output is an object file (binary code).

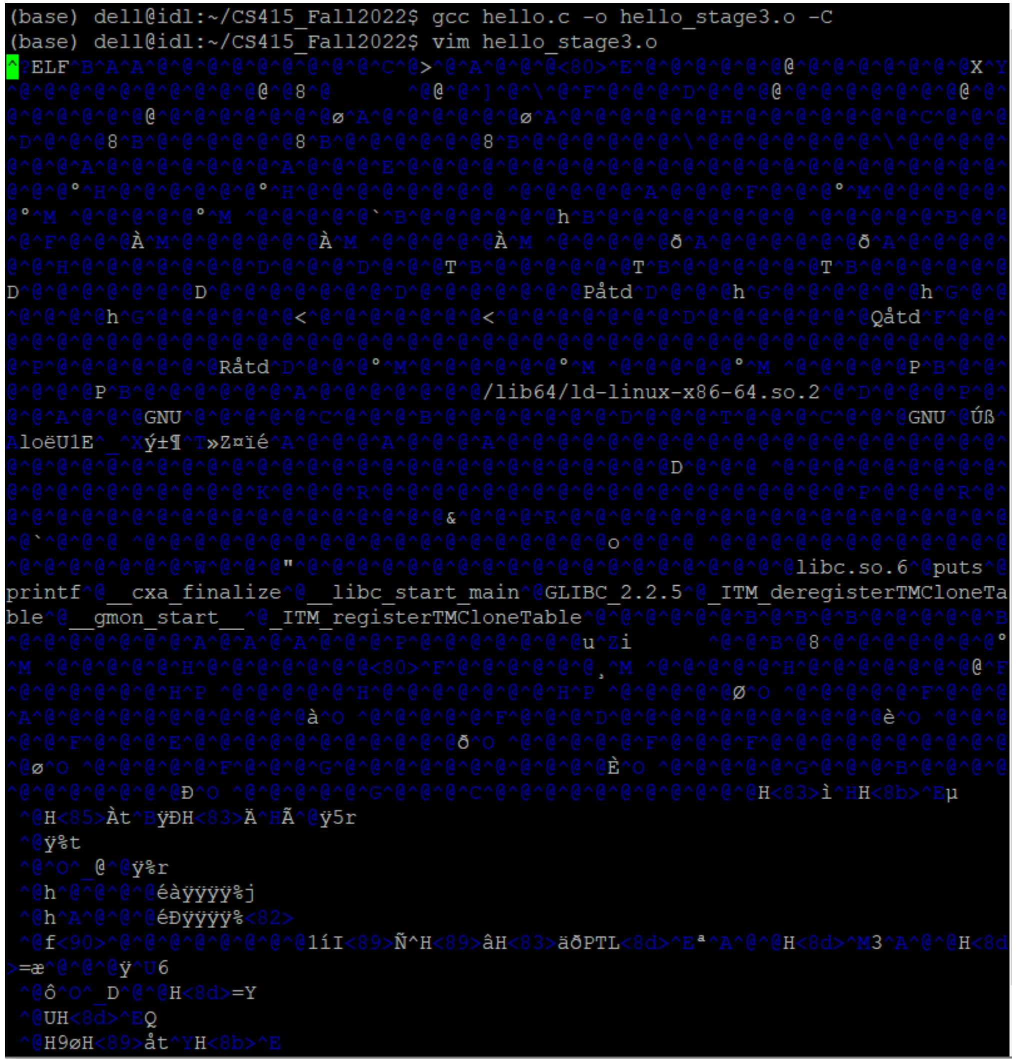
1 Original Content of the C program

2 Object program of hello.c

1



2



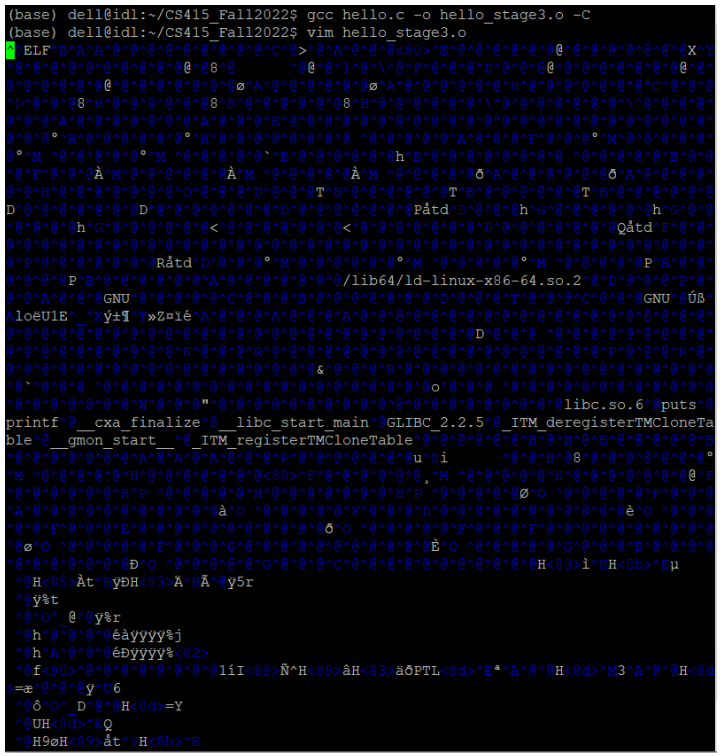
Using the gcc compiler: Stopping the compilation after the assembly stage

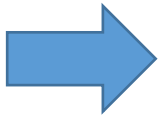
The output is an object file (binary code).

1. Object program of hello.c
2. Binary Instruction (Machine Instruction) of each assembly instruction
3. Assembly instructions of the object file

1

```
(base) dell@idl:~/CS415_Fall2022$ gcc hello.c -o hello_stage3.o -C
(base) dell@idl:~/CS415_Fall2022$ vim hello_stage3.o
```

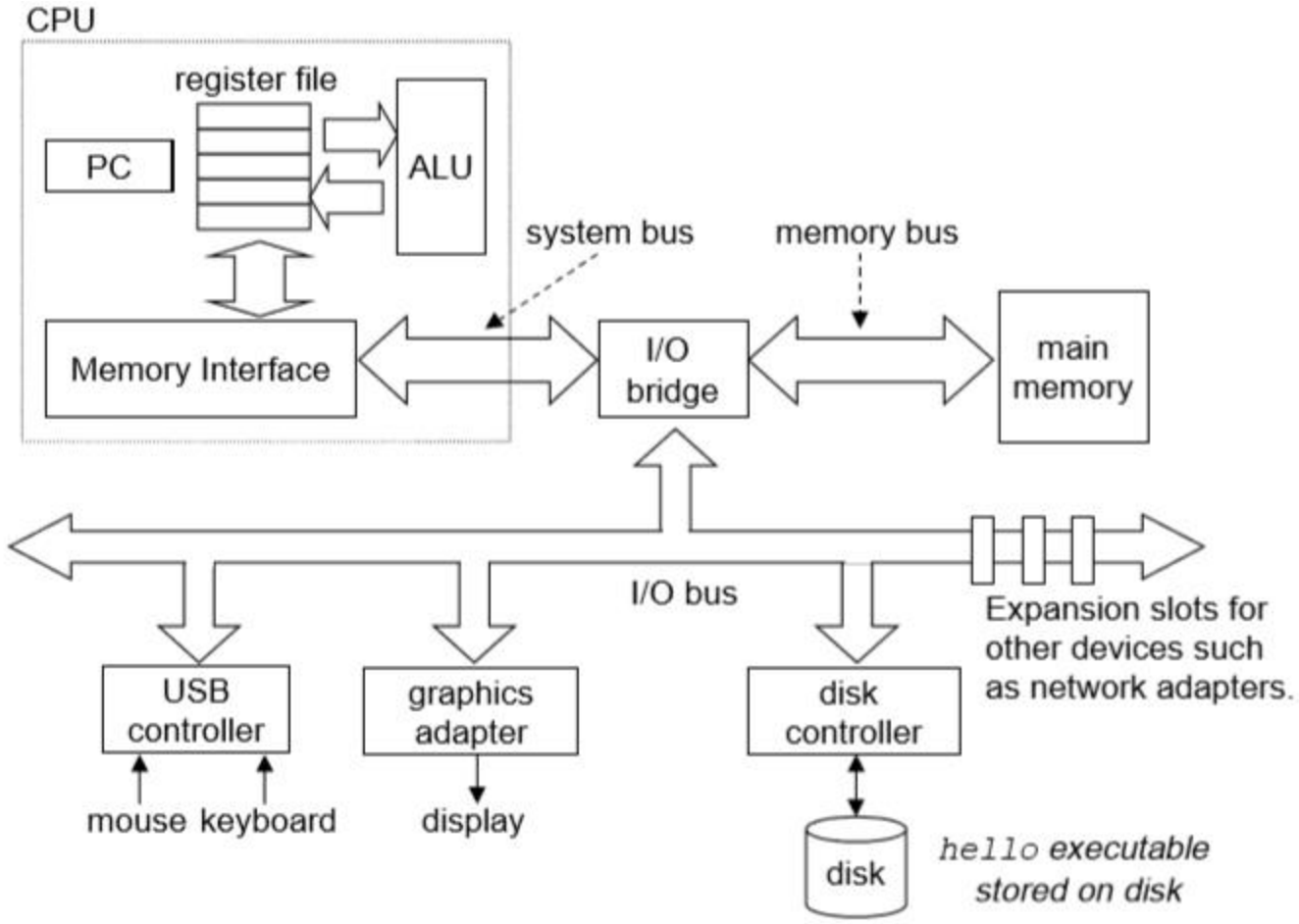




```
hello_stage3.o:     file format elf64-x86-64

Disassembly of section .interp:

0000000000000238 <.interp>:
 238:   2f                      (bad)
 239:   6c                      insb   (%dx),%es:(%rdi)
 23a:   69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
 241:   2d 6c 69 6e 75          sub    $0x756e696c,%eax
 246:   78 2d                   js     275 <_init-0x2b3>
 248:   78 38                   js     282 <_init-0x2a6>
 24a:   36 2d 36 34 2e 73       ss sub $0x732e3436,%eax
 250:   6f                      outsl  %ds:(%rsi),(%dx)
 251:   2e 32 00                xor    %cs:(%rax),%al

Disassembly of section .note.ABI-tag:

0000000000000254 <.note.ABI-tag>:
 254:   04 00                   add    $0x0,%al
 256:   00 00                   add    %al,(%rax)
 258:   10 00                   adc    %al,(%rax)
 25a:   00 00                   add    %al,(%rax)
 25c:   01 00                   add    %eax,(%rax)
 25e:   00 00                   add    %al,(%rax)
 260:   47                      rex.RXB
 261:   4e 55                   rex.WRX push %rbp
 263:   00 00                   add    %al,(%rax)
 265:   00 00                   add    %al,(%rax)
 267:   00 03                   add    %al,(%rbx)
 269:   00 00                   add    %al,(%rax)
 26b:   00 02                   add    %al,(%rdx)
 26d:   00 00                   add    %al,(%rax)
 26f:   00 00                   add    %al,(%rax)
 271:   00 00                   add    %al,(%rax)
        ...

Disassembly of section .note.gnu.build-id:

0000000000000274 <.note.gnu.build-id>:
 274:   04 00                   add    $0x0,%al
 276:   00 00                   add    %al,(%rax)
 278:   14 00                   adc    $0x0,%al
 27a:   00 00                   add    %al,(%rax)
 27c:   03 00                   add    (%rax),%eax
 27e:   00 00                   add    %al,(%rax)
 280:   47                      rex.RXB
 281:   4e 55                   rex.WRX push %rbp
"hello_stage3_decoded.o" 900L, 38387C
```

Running the hello.c program (Hardware side)

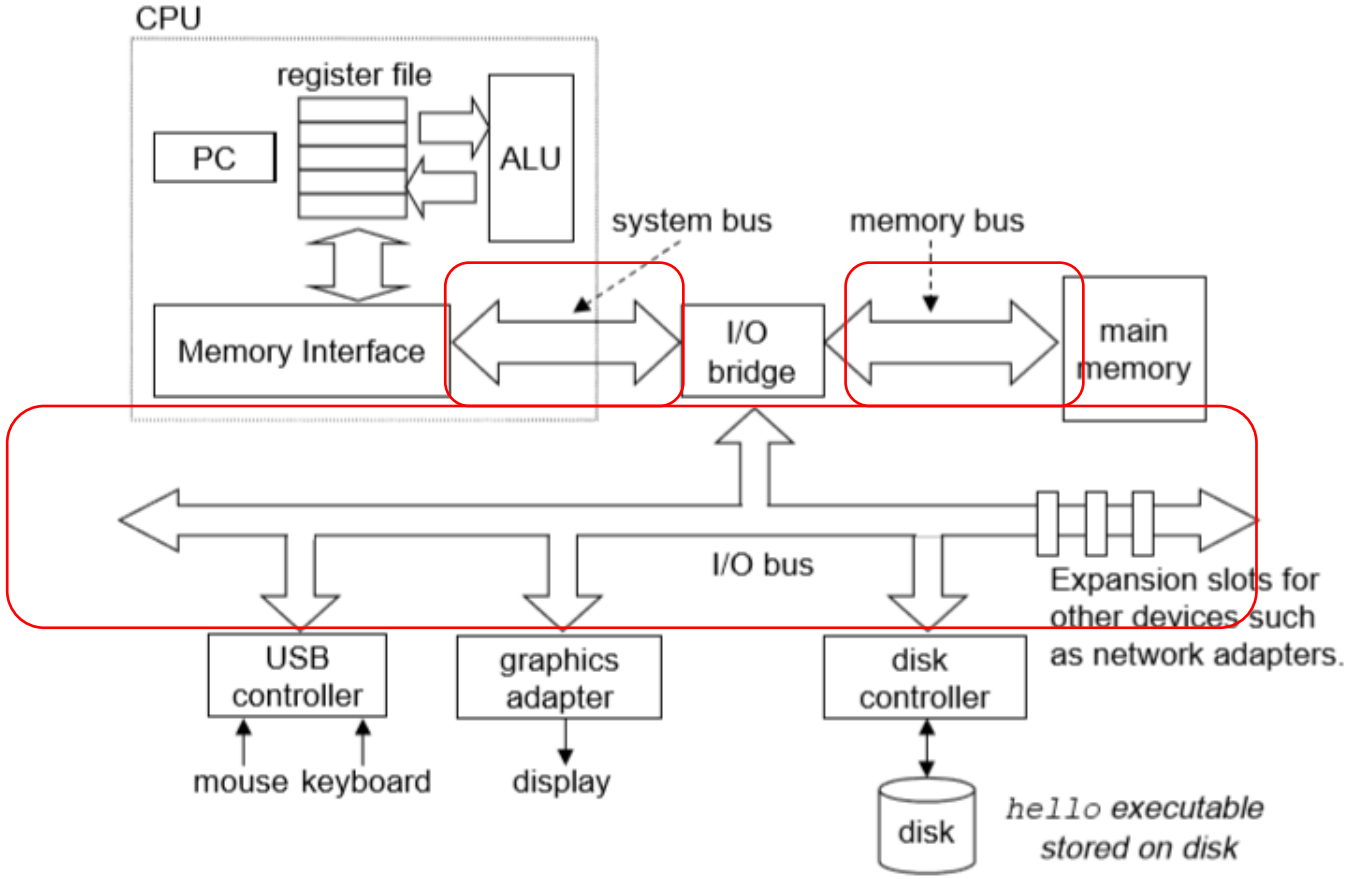
- Let's learn a little about the hardware



Example system

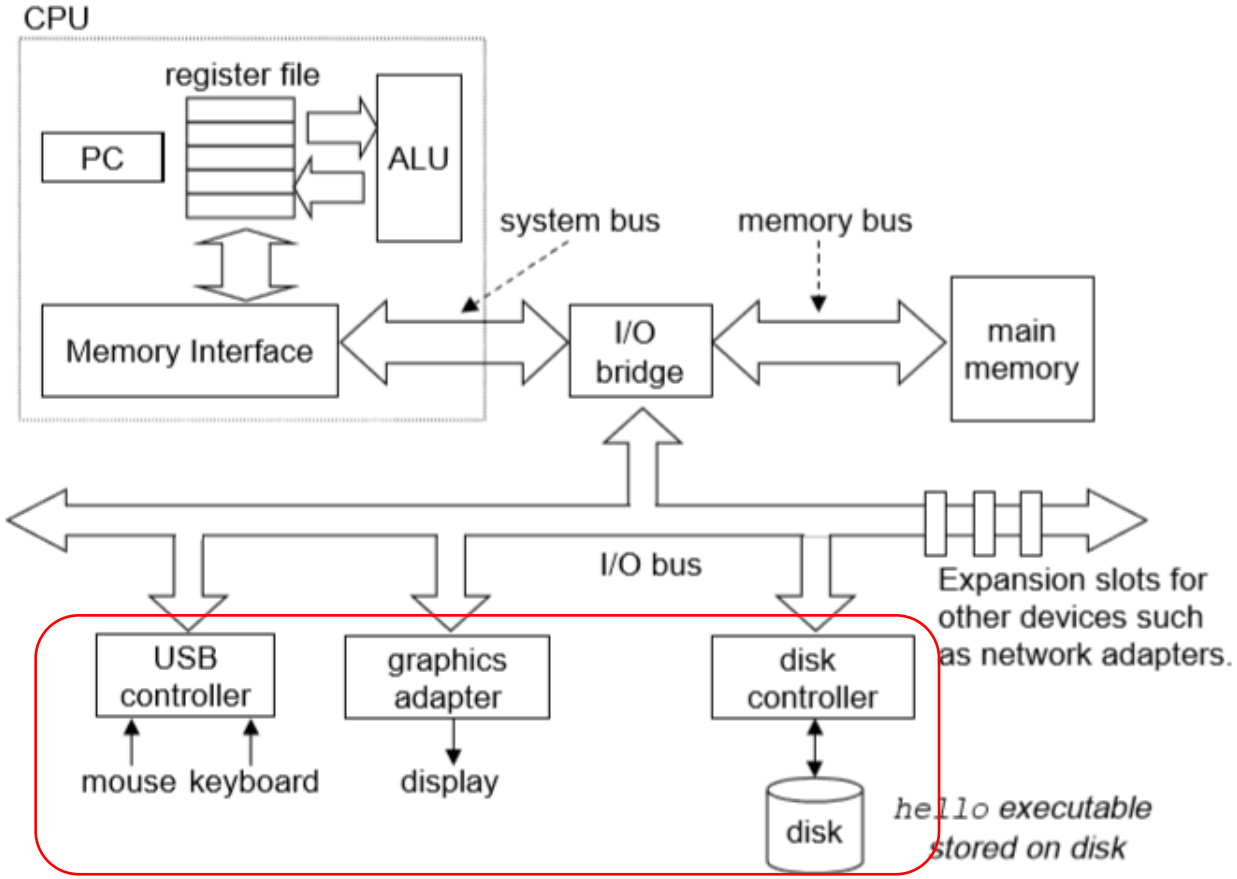
Figure 1.4: **Hardware organization of a typical system.** CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.

Hardware side (contd.)



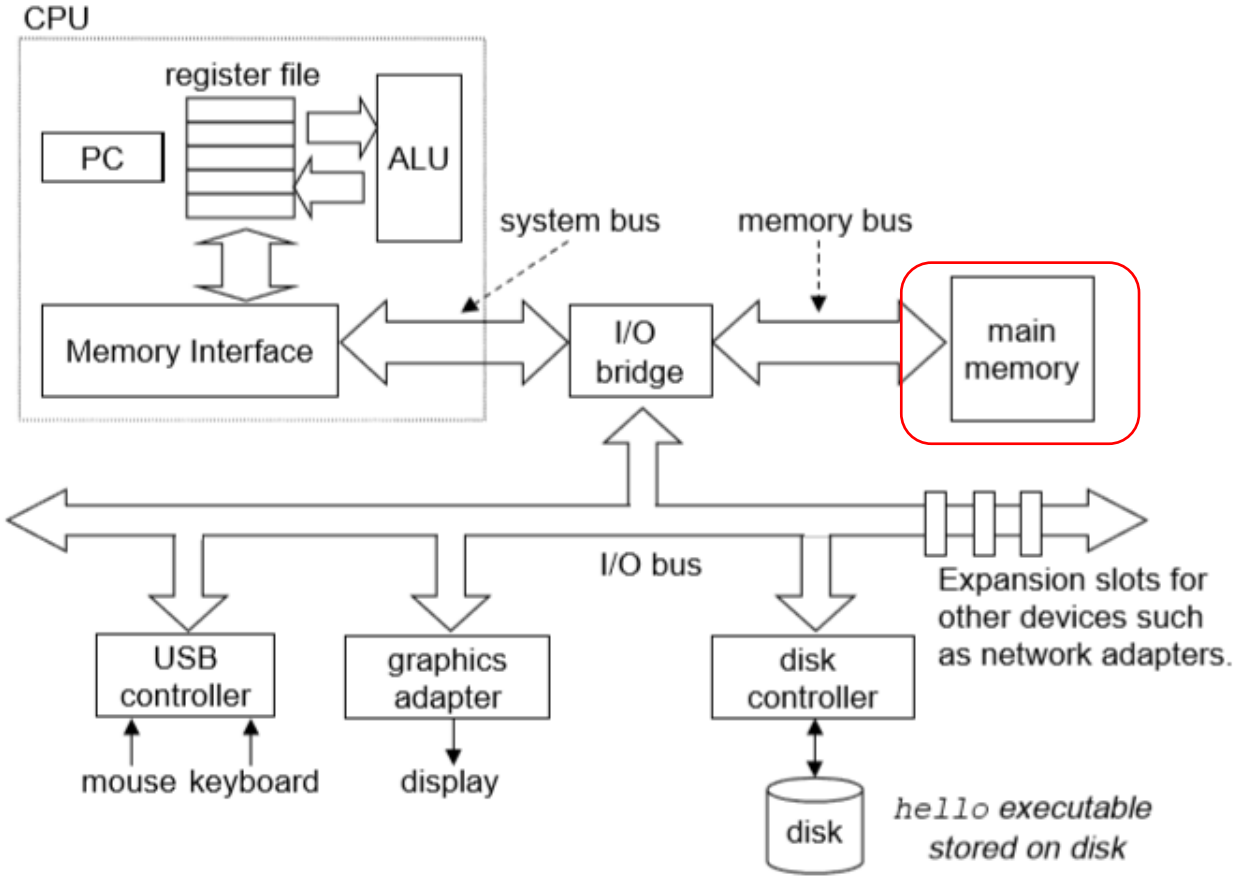
- **Buses** are a collection of electrical conduits called buses that carry bytes of information back and forth between the components.
- Buses are typically designed to transfer **fixed-sized chunks** of bytes known as **words**.

Hardware side (contd.)



- I/O devices: Our example system has four I/O devices: a keyboard and mouse for user input, a display for user output, and a disk drive.
- Each I/O device is connected to the I/O bus by either a controller or an adapter
  - A **controller** is a chipset in the device itself or on the system's main printed circuit board (often called the motherboard).
  - An **adapter** is a card that plugs into a slot on the motherboard.

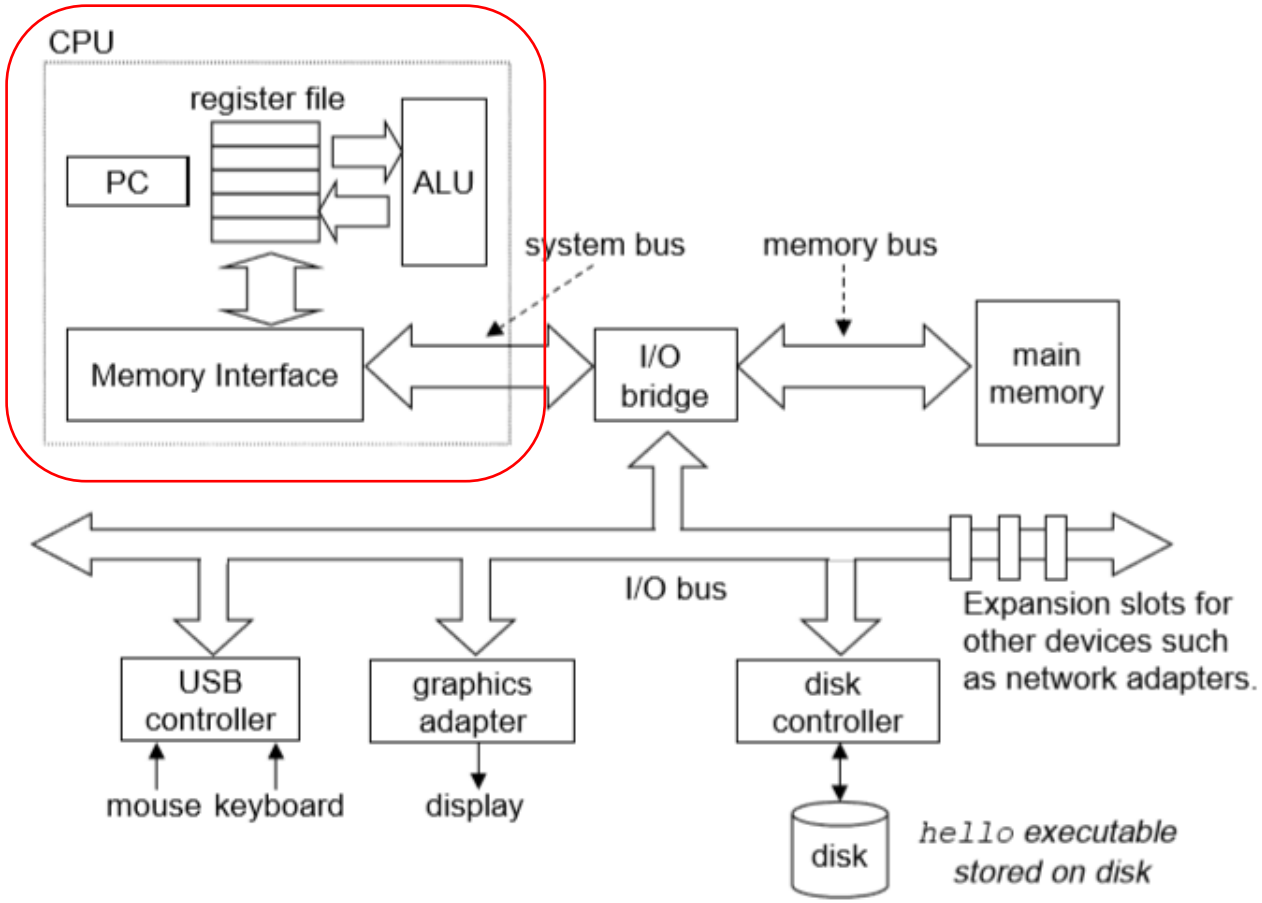
Hardware side (contd.)



- Main memory: A temporary storage device that holds both a program and the data it manipulates while the processor is executing the program
- Physically, main memory consists of a collection of *Dynamic Random Access Memory (DRAM)* chips

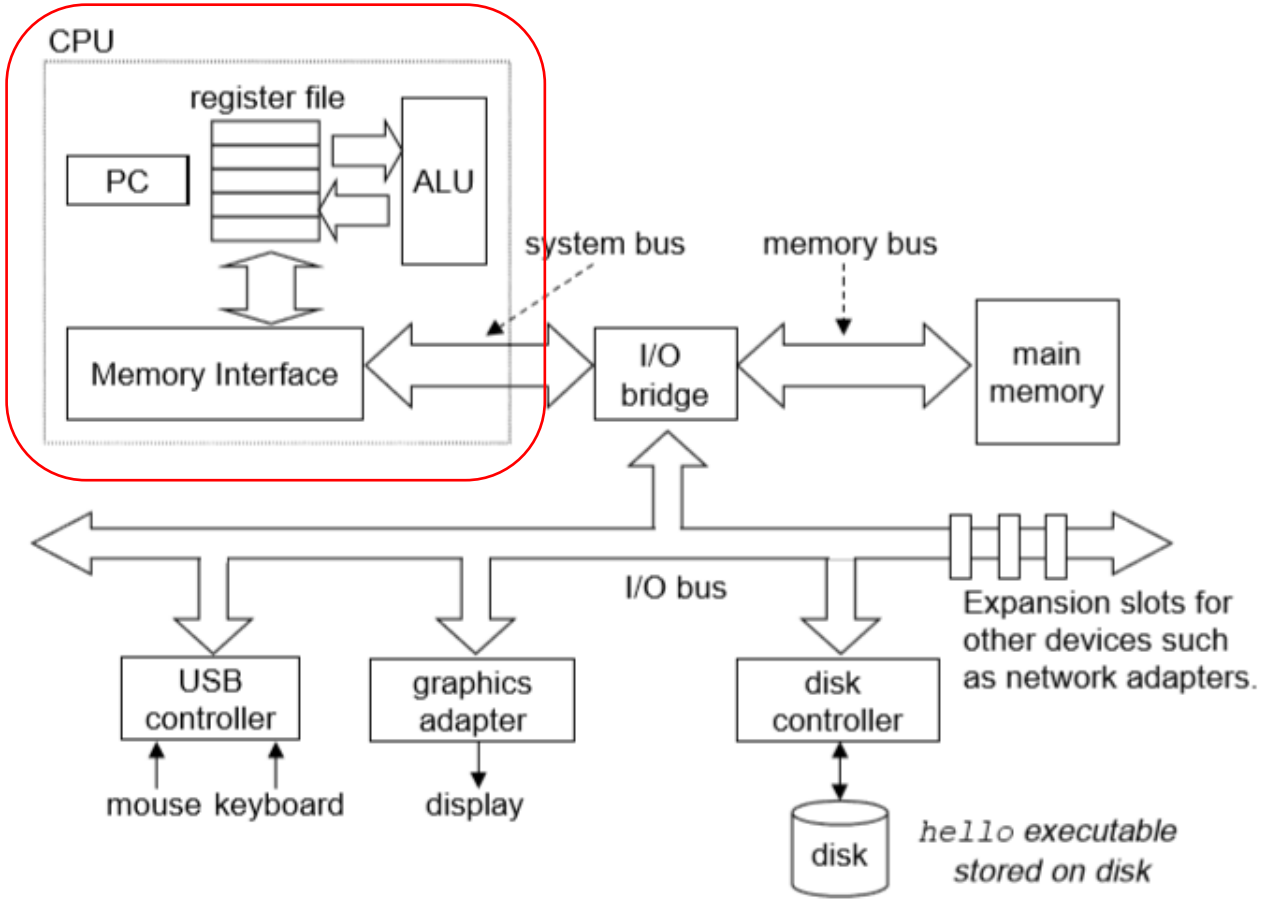
Hardware side (contd.)

IBM701 PC 1952



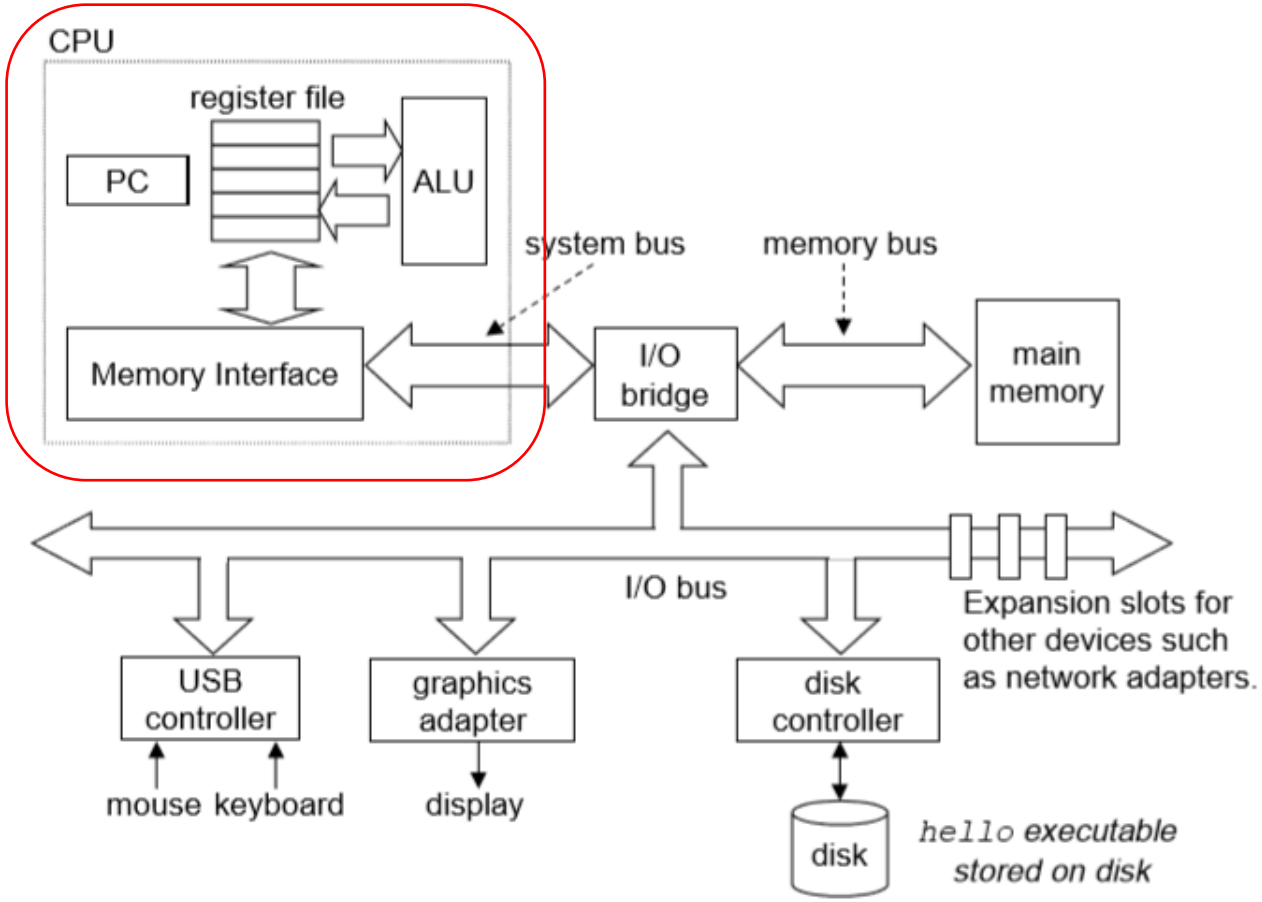
- The central processing unit (CPU), or simply processor, is the engine that interprets (or executes) instructions stored in main memory
- At its core is a word-sized storage device (or register) called the program counter (PC). At any point in time, the PC points at (contains the address of) some machine-language instruction in main memory

Hardware side (contd.)



- From the time that power is applied to the system, until the time that the power is shut off, the processor blindly and repeatedly performs the same basic task. It reads the instruction from memory pointed at by the PC, interprets the bits in the instruction, performs some simple operation dictated by the instruction, and then updates the PC to point to the next instruction.

Hardware side (contd.)



- CPU might carry out at the request of an instruction:

Load: Copy a byte or a word from main memory into a register.

Store: Copy a byte or a word from a register to a location in main memory.

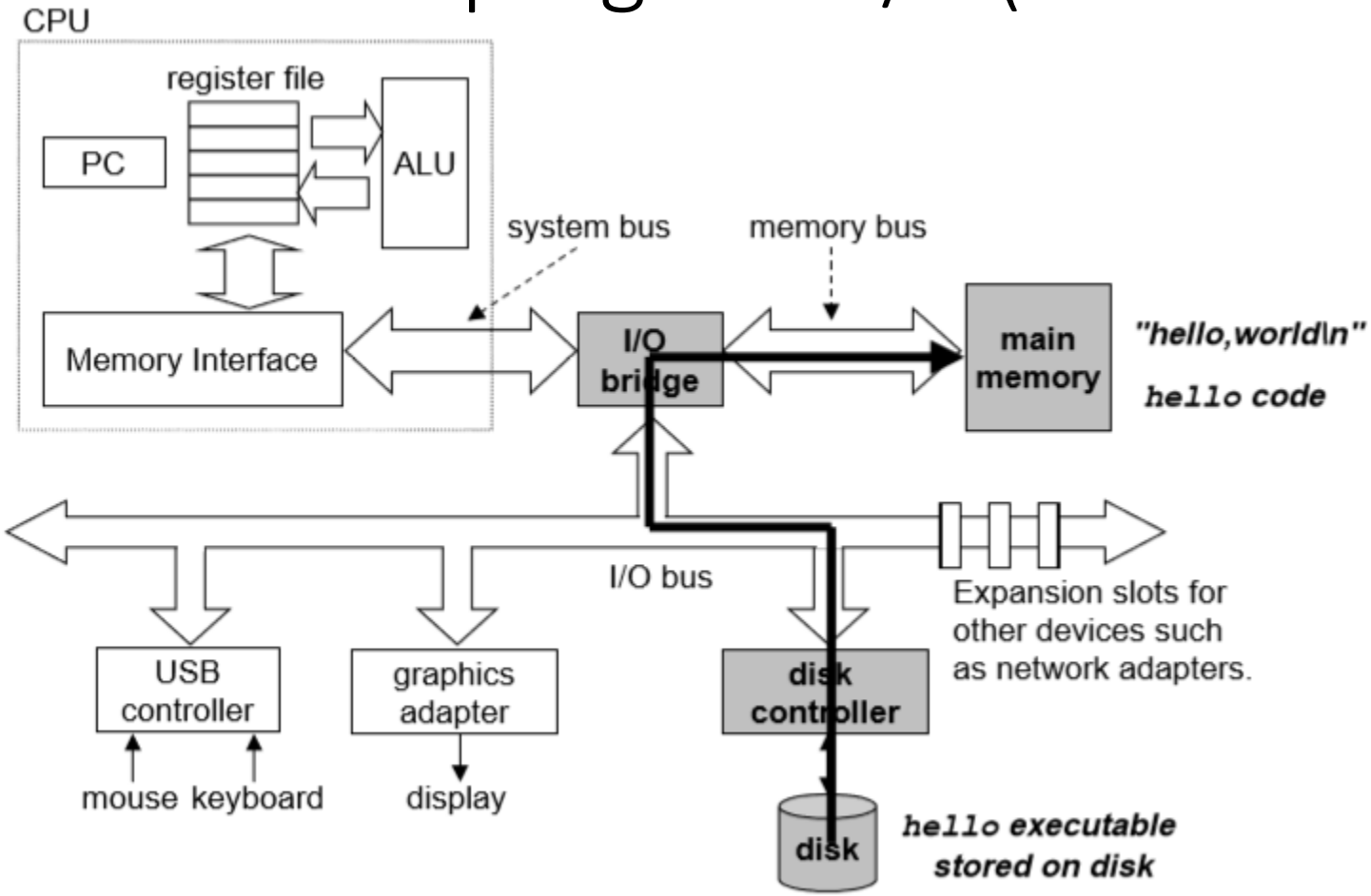
Add: Copy the contents of two registers to the ALU, which adds the two words together and stores the result in a register.

I/O Read: Copy a byte or a word from an I/O device into a register**.**

I/O Write: Copy a byte or a word from a register to an I/O device.

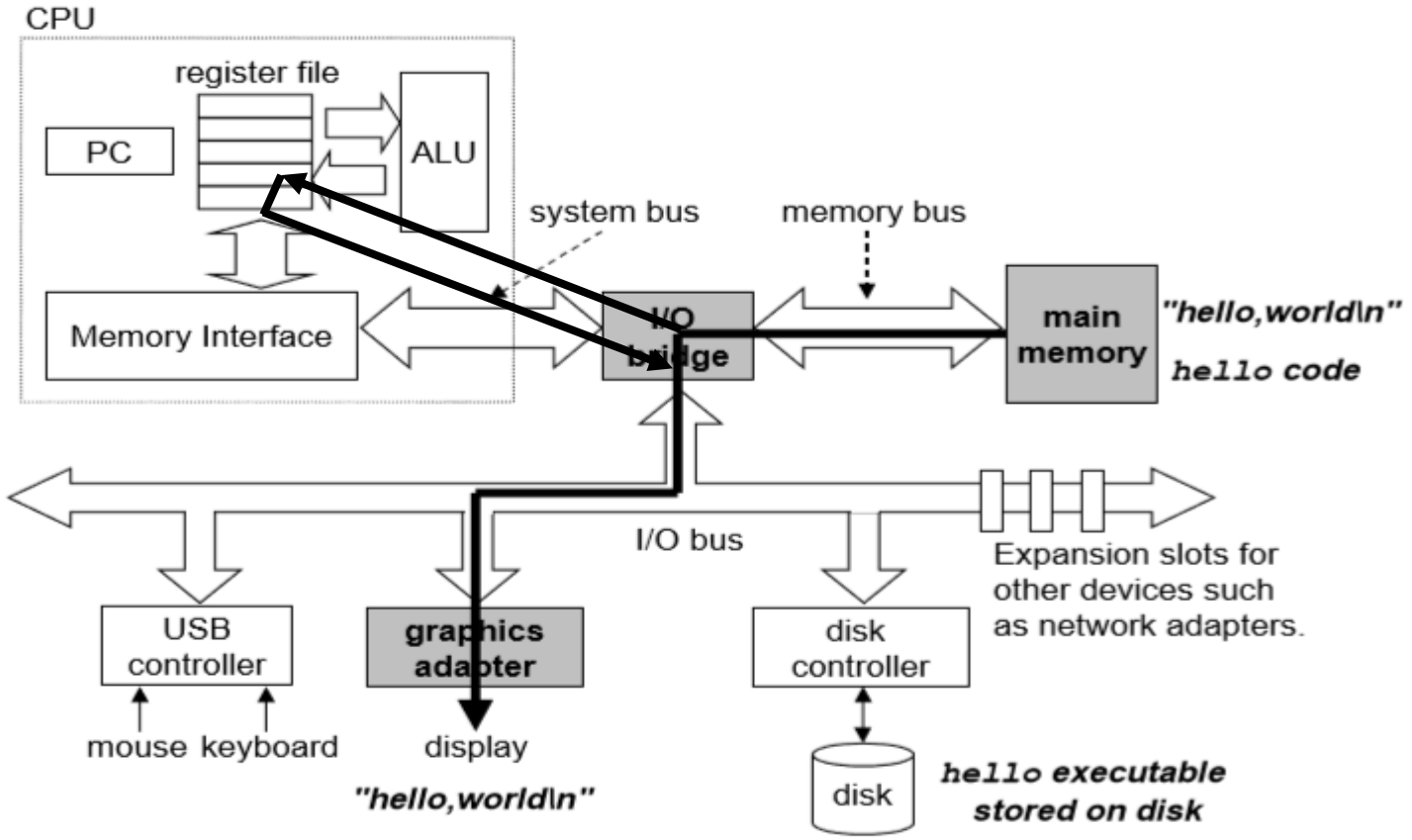
Jump: Extract a word from the instruction itself and copy that word into the program counter (PC)

Running the hello.c program 1/3 (Hardware side)



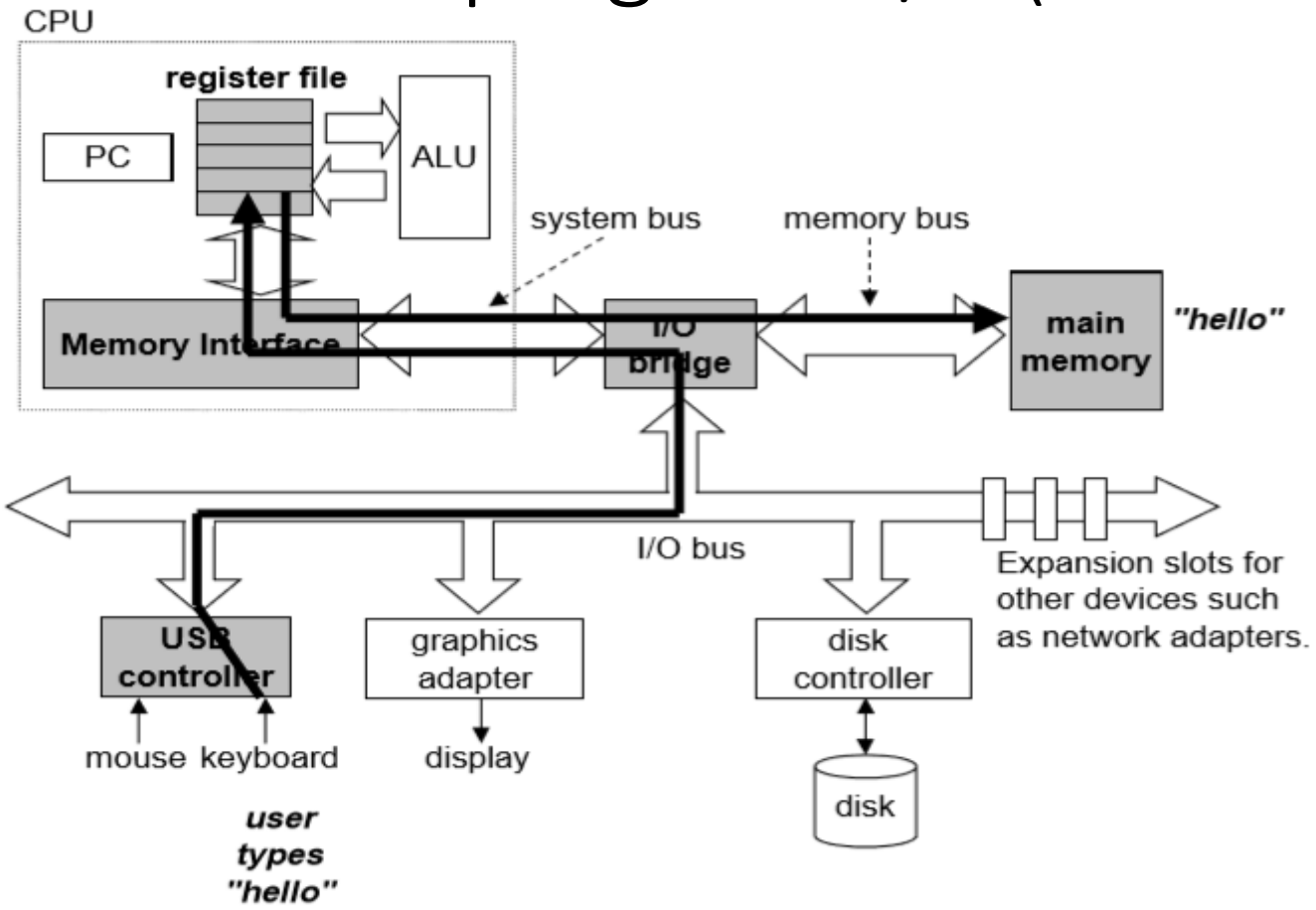
- Using a technique known as direct memory access (DMA), the data travels directly from disk to main memory, without passing through the processor

Running the hello.c program 2/3 (Hardware side)



- Once the code and data in the hello object file are loaded into memory, the processor begins executing the machine-language instructions in the hello program's main routine. These instruction 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

Running the hello.c program 3/3 (Hardware side)



- As we type the characters hello at the keyboard, the shell program reads each one into a register, and then stores it in memory