1.3 Below your program

> "In Paris they simply stared when I spoke to them in French; I never did succeed in making those idiots understand their own language.
> *Mark Twain, The Innocents Abroad, 1869*

A typical application, such as a word processor or a large database system, may consist of millions of lines of code and rely on sophisticated software libraries that implement complex functions in support of the application. As we will see, the hardware in a computer can only execute extremely simple low-level instructions. To go from a complex application to the primitive instructions involves several layers of software that interpret or translate high-level operations into simple computer instructions, an example of the great idea of **abstraction**.

The animation below shows that these layers of software are organized primarily in a hierarchical fashion, with applications being the outermost ring and a variety of *systems software* sitting between the hardware and application software.

Systems software: Software that provides services that are commonly useful, including operating systems, compilers, loaders, and assemblers.

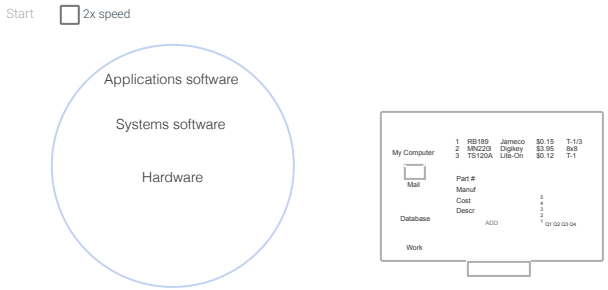
There are many types of systems software, but two types of systems software are central to every computer system today: an operating system and a compiler. An *operating system* interfaces between a user's program and the hardware and provides a variety of services and supervisory functions. Among the most important functions are:

- Handling basic input and output operations
- Allocating storage and memory
- Providing for protected sharing of the computer among multiple applications using it simultaneously

Operating system: Supervising program that manages the resources of a computer for the benefit of the programs that run on that computer.

Examples of operating systems in use today are Linux, iOS, and Windows.

PARTICIPATION ACTIVITY 1.3.1: A simplified view of hardware and software as hierarchical layers, shown as concentric circles with hardware in the center and applications software outermost (COD Figure 1.3).



In complex applications, multiple application layers may exist as well. Ex: A database system may run on the systems software, and another application runs on top of the database.

Compilers perform another vital function: the translation of a program written in a high-level language, such as C, C++, Java, or Visual Basic into instructions that the hardware can execute. Given the sophistication of modern programming languages and the simplicity of the instructions executed by the hardware, the translation from a high-level language program to hardware instructions is complex. We give a brief overview of the process here and then go into more depth in COD Chapter 2 (Instructions: Language of the Computer).

Compiler: A program that translates high-level language statements into assembly language statements.

PARTICIPATION ACTIVITY 1.3.2: Below your program.

1) Behind a car's simple items like steering wheel, gas pedal, and brake pedal are complex mechanical/computerized details. Those simple items represent ____.
   - abstraction
   - an operating system
   - Linux

2) The physical machinery on which software runs.
   - Instructions
   - Hardware
   - Compiler

3) Software that manages hardware

resources on behalf of other programs.

- ○ Hardware
- ○ Compiler
- ○ Operating system

4) A program to play tic-tac-toe.
   - ○ Systems software
   - ○ Application software

5) The collection of software on a computer that provides services to application software.
   - ○ Systems software
   - ○ Compiler

From a high-level language to the language of hardware

To speak directly to electronic hardware, you need to send electrical signals. The easiest signals for computers to understand are *on* and *off*, and so the computer alphabet is just two letters. Just as the 26 letters of the English alphabet do not limit how much can be written, the two letters of the computer alphabet do not limit what computers can do. The two symbols for these two letters are the numbers 0 and 1, and we commonly think of the computer language as numbers in base 2, or ***binary numbers***. We refer to each "letter" as a *binary digit* or *bit*. Computers are slaves to our commands, which are called *instructions*. Instructions, which are just collections of bits that the computer understands and obeys, can be thought of as numbers. For example, the bits

```
1000110010100000
```

tell one computer to add two numbers. COD Chapter 2 (Instructions: Language of the Computer) explains why we use numbers for instructions *and* data; we don't want to steal that chapter's thunder, but using numbers for both instructions and data is a foundation of computing.

Binary digit: Also called a ***bit***. One of the two numbers in base 2 (0 or 1) that are the components of information.

Instruction: A command that computer hardware understands and obeys.

The first programmers communicated to computers in binary numbers, but this was so tedious that they quickly invented new notations that were closer to the way humans think. At first, these notations were translated to binary by hand, but this process was still tiresome. Using the computer to help program the computer, the pioneers invented software to translate from symbolic notation to binary. The first of these programs was named an *assembler*. This program translates a symbolic version of an instruction into the binary version. For example, the programmer would write

```
ADD A,B
```

and the assembler would translate this notation into

```
1000110010100000
```

This instruction tells the computer to add the two numbers `A` and `B`. The name coined for this symbolic language, still used today, is *assembly language*. In contrast, the binary language that the machine understands is the *machine language*.

Assembler: A program that translates a symbolic version of instructions into the binary version.

Assembly language: A symbolic representation of machine instructions.

Machine language: A binary representation of machine instructions.

PARTICIPATION ACTIVITY 1.3.3: Bits, assembly language, and machine language.

1) "Bit" is short for "binary digit."
   - ○ True
   - ○ False

2) Binary refers to 10^{-1}.
   - ○ True
   - ○ False

3) Computers use binary because binary is more powerful than decimal numbers.
   - ○ True
   - ○ False

4) Although binary's alphabet contains only two "letters", 0 and 1, the binary alphabet can represent as much information as the English alphabet's 26 letters.
   - ○ True
   - ○ False

5) The number 12 can be represented in

binary as 1100. If a computer's memory location contains 00001100, then that location contains the number 12.

- ○ True
- ○ False

6) The following could be a machine-language instruction: 1000110010100000.

- ○ True
- ○ False

7) The following could be an assembly language instruction: 1000110010100000.

- ○ True
- ○ False

8) An assembler translates assembly language instructions like `ADD A,B` to machine-language instructions like 1000110010100000.

- ○ True
- ○ False

Although a tremendous improvement, assembly language is still far from the notations a scientist might like to use to simulate fluid flow or that an accountant might use to balance the books. Assembly language requires the programmer to write one line for every instruction that the computer will follow, forcing the programmer to think like the computer.

The recognition that a program could be written to translate a more powerful language into computer instructions was one of the great breakthroughs in the early days of computing. Programmers today owe their productivity—and their sanity—to the creation of *high-level programming languages* and compilers that translate programs in such languages into instructions. The animation below shows the relationships among these programs and languages, which are more examples of the power of **abstraction**.

High-level programming language: A portable language such as C, C++, Java, or Visual Basic that is composed of words and algebraic notation that can be translated by a compiler into assembly language.

PARTICIPATION ACTIVITY 1.3.4: C program compiled into assembly language and then assembled into binary machine language (COD Figure 1.4).

Start ☐ 2x speed

High-level language program (in C)

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

Compiler

Assembly language program (for LEGv8)

```
swap:
  LSL  X10, X1,3
  ADD  X10, X0,X10
  LDUR X9, [X10,0]
  LDUR X11,[X10,8]
  STUR X11,[X10,0]
  STUR X9, [X10,8]
  BR   LR
```

Assembler

Binary machine language program (for LEGv8)

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111100010000000000000000
10001111000010010000000000000100
10101111000010010000000000000000
10101101111100010000000000000100
00000011111000000000000000001000
```

Note: Above, although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in COD Chapter 2 (Instructions: Language of the Computer).

A compiler enables a programmer to write this high-level language expression:

```
A + B
```

The compiler would compile it into this assembly language statement:

```
ADD A,B
```

As shown above, the assembler would translate this statement into the binary instructions that tell the computer to add the two numbers `A` and `B`.

High-level programming languages offer several important benefits. First, they allow the programmer to think in a more natural language, using English words and algebraic notation, resulting in programs that look much more like text than like tables of cryptic symbols (see the animation above). Moreover, they allow languages to be designed according to their intended use. Hence, Fortran was designed for scientific computation, Cobol for business data processing, Lisp for symbol manipulation, and so on. There are also domain-specific languages for even narrower groups of users, such as those interested in simulation of fluids, for example.

The second advantage of programming languages is improved programmer productivity. One of the few areas of widespread agreement in software development is that it takes less time to develop programs when they are written in languages that require fewer lines to express an idea. Conciseness is a clear advantage of high-level languages over assembly language.

The final advantage is that programming languages allow programs to be independent of the computer on which they were developed, since compilers and assemblers can translate high-level language programs to the binary instructions of any computer. These three advantages are so strong that today little programming is done in assembly language.

PARTICIPATION ACTIVITY 1.3.5: High-level programming language.

1) Which is a high-level language instruction?
   - `00000000101000100000000100011000`
   - `ADD X2, X4, X2`
   - temp = v[k];

2) What kind of language is C?
   - Machine
   - Assembly
   - High-level

3) An advantage of a high-level language is allowing a programmer to _____.
   - think more naturally
   - think like a machine

4) An advantage of a high-level language is enabling a programmer to _____.
   - change a program
   - implement a program in less time

5) An advantage of a high-level language is that a program _____.
   - is specific to a particular machine
   - is independent of a particular machine

Provide feedback on this section