Chapter 1

Introduction to Computation

Algorithms, Processors, and Programs

Outline

- Concepts of algorithm, processor, and memory
- Translation and interpretation of programs
- Java virtual machine (JVM)
- Java source code compiler and byte code
1.1 Introduction

In this chapter we present a brief overview of the nature of computation and the process whereby a computer program written in a high-level language such as Java can be executed by a computer system. First we introduce the concept of an algorithm which describes how to solve a problem or accomplish a task in a sequence of well defined steps. We are interested in algorithms that can be executed by a computer system. To do this it is necessary to translate the algorithm into a program in a language that the computer understands. When this is done the computer can execute the steps of the program to complete the task for which the algorithm was designed.

The design of efficient algorithms, their expression in a programming language, and their execution by a computer system is the essence of computer science. We are primarily concerned with the problem solving process whereby algorithms are constructed, converted to programs in the Java language, compiled, and executed by the Java Virtual Machine.

1.2 Algorithms

We are all somewhat familiar with algorithms. A recipe for baking a loaf of bread is a standard example. The recipe is expressed as a number of steps that, if followed, will produce a loaf of bread. It is important that each step be specified without ambiguity. For example, the step “add some flour” is not precise enough and should be replaced by something like “add 3 cups of flour”. Here we are assuming that the baker knows how much is in a cup. There are many other examples from everyday life that can be expressed as a number of English statements that are sufficiently detailed for an average person to understand and follow.

A recipe is not unique. It is possible to express a recipe for a particular kind of bread as a large number of simple steps for a novice baker or as a smaller number of steps for an experienced baker, who would understand a vague statement such as “season to taste”. A novice would require further instructions on the kind and quantity of spices or herbs to add. Generalizing, we can say that an algorithm is a sequence of unambiguous steps, for accomplishing some task or process. Each step is a simple instruction or statement. The word “sequence” is important since it implies that the steps must be performed in a specific order.

We are interested in algorithms that can eventually be executed by a computer system so it will be very important that the individual instructions be unambiguous. We also require that the total number of steps in an algorithm is finite although algorithms can have an infinite number of steps and still be well-defined. For example, in mathematics any algorithm to calculate \( \sqrt{2} \) necessarily has an infinite number of steps since the answer is an irrational number that has an infinite non-repeating decimal expansion.

We can rephrase the algorithm in a finite form by asking for an algorithm to compute \( \sqrt{2} \) accurate to a specified number of digits. Such an algorithm is given in Figure 1.1. Here the left arrow means that the value on the right side of the arrow is computed and assigned as the value of the variable on the left side of the arrow. This algorithm is still not completely precise because it doesn’t specify how many digits to use in the intermediate calculations. For example, if you use at least 16 significant digits in each arithmetic operation then the algorithm will produce the approximation 1.41421356237 accurate to the number of digits shown. The algorithm also assumes that
the user knows how to add and divide real numbers (floating point numbers). Otherwise it would have to be rewritten as an enormous number of smaller steps that explain the + and / operations.

Thus, algorithms have a context: they assume that the user (or computer) understands how to execute a basic set of unambiguous instructions. Unfortunately, as we will see, the instructions understood by a computer system are very elementary and far removed from the combination of natural language and mathematics that are normally used to formulate algorithms at the human problem solving level.

1.3 Processors

A processor is any device, computer, calculator, or human being, for example, that can process or execute a sequence of instructions. A computer system contains a processor, called a central processing unit or CPU, which knows how to perform (execute) instructions from a specific instruction set called the machine language of the CPU. Each kind of CPU has a different machine language. For example, the machine language of the Pentium CPU used in a PC is very different from that of the PowerPC CPU used in the Macintosh computer. The CPU in one of these personal computers is built on a single microchip and is called a microprocessor.

Each machine language instruction is defined by a specific binary code: a binary string of 0’s and 1’s. This code determines what operation is to be performed and where to locate any required data. This data is also represented in binary form. Compared to human problem solving instructions and algorithms, written in English and mathematics, machine language instructions are very primitive. In fact a single English instruction such as “compute the value of this polynomial at $x = 3.14$” may have to be translated into several or even hundreds of machine language instructions before it is understood by the CPU.

Fortunately, this translation process can be automated today, although this wasn’t always the case for the first electronic computers. Our goal in this chapter is to briefly explain how the human problem solving domain is mapped down to this primitive machine language level.
1.3.1 Functional units of a CPU

A CPU is composed of three basic functional units.

- **Arithmetic-Logic Unit (ALU):** performs logical and arithmetic operations on data, such as adding, subtracting, multiplying or dividing two binary numbers, determining if a number is zero or greater than zero, or comparing two numbers to see if one is equal to the other, less than the other, or greater than the other.

- **Registers:** storage locations within the CPU that hold data and numbers needed by the ALU in performing operations, storing the results of these operations, or storing the address of the next instruction to be executed.

- **Control Unit (CU):** controls the operation of the processor such as fetching from memory the next instruction to be executed and any data needed, or decoding binary instructions to determine what operation is to be performed.

1.4 Memory and I/O devices

To execute a machine language program the CPU needs to access the instructions and data. In the early computer systems instructions and data were separate concepts. Data was stored in a memory but the program was not stored anywhere. Instead it was painstakingly constructed by connecting various cables and hardware together. In essence the control unit was being “re-programmed” for each program.

Von Neumann introduced what is known as the “stored-program concept” that is the basis of modern computing systems. The idea is that since instructions and data are both strings of binary digits (bits) they can both be stored in a memory external to the CPU: instructions are just a form of data understood by the CPU. Thus, a machine language program and its data is simply a large string of 0’s and 1’s stored in a memory. The CPU keeps track of the location in memory of the next instruction to be executed and each instruction specifies the location in memory of any required data. To execute another program it is not necessary to “rewire” the hardware: simply store a new program and its data in the memory and tell the CPU where to find the first instruction.

A memory location is a basic unit of memory that has an address associated with it to distinguish it from other memory locations. Today most memories are organized as a sequence of bytes. Each byte is a memory location containing 8-bits of information. We say that the memory is “byte-addressable” since every byte has a unique address. Each instruction is now stored in one or more bytes of memory and similarly for data items such as integers and real numbers. Many modern microprocessors use 32-bit instructions each stored in four consecutive bytes and use 32-bit addresses to locate each byte. Similarly, they can process data in arithmetic and logical operations using up to 32-bits at a time. A pictorial representation of six consecutive bytes of memory, their addresses, and their content is shown in Figure 1.2. The content of each of the six bytes is shown inside the boxes but the addresses are shown as labels beside each box. They are obtained using address decoding logic built into the memory chips and CPU.

There are two categories of memory chips. ROM is read only memory that once written can not be changed and RAM is memory that can be read or written. The contents of RAM memory
1.4 Memory and I/O devices

Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001100...000</td>
<td>11100111</td>
</tr>
<tr>
<td>0001100...001</td>
<td>00000110</td>
</tr>
<tr>
<td>0001100...010</td>
<td>01011101</td>
</tr>
<tr>
<td>0001100...011</td>
<td>10010010</td>
</tr>
<tr>
<td>0001100...100</td>
<td>01110111</td>
</tr>
<tr>
<td>0001100...101</td>
<td>10100000</td>
</tr>
</tbody>
</table>

Figure 1.2: Computer memory locations (bytes) and addresses

Main Memory

Secondary Memory

Registers

Control Unit

ALU

Input Devices

Output Devices

CPU

Figure 1.3: Block diagram of a computer system

(volatile memory) are lost when power to the memory chips is interrupted (by turning the computer off, for example) but the contents of ROM memory is permanent. There are also intermediate types of memory that can be read and written but do not lose their contents when power is interrupted.

The ROM and RAM memory devices are often called main memory as opposed to secondary memory which consists of external storage devices such as floppy disks and hard disks. They are needed to store files and results of computations in a more permanent form that is not lost when the computer is turned off.

Floppy disks and hard disks are both input and output devices, whereas the keyboard and the mouse are classified as input devices. Here we use the term “input data” to describe incoming data that is needed by the processor as it is executing instructions. The term “output data” refers to data produced by the processor as a result of executing instructions. The block diagram in Figure 1.3 shows how the CPU communicates with the “outside world” using main memory (RAM and ROM), secondary memory (floppy disks, hard disks), input devices (such as a mouse or key-
board), and output devices (such as a display screen or a printer).

1.5 Programs

A program is a representation of an algorithm as a sequence of instructions that can be understood and executed by a processor to complete the task, problem, or process described by the algorithm. For a CPU each program is a sequence of machine language instructions. Since there are only a few registers (32, for example) within a CPU for holding instructions to be executed, the data required, and intermediate results, machine language programs and their data are stored in main memory (see Figure 1.3).

The CPU’s control unit (CU) is designed to fetch instructions from memory and have them executed using the ALU for operations and the registers for temporary storage. One of the registers is a special one called the instruction address register (IAR). It’s purpose is to hold the memory address of the next machine language instruction to be executed. Program execution at the machine level can be briefly described by the following steps:

1. Load the IAR with the starting address in memory of the first machine language instruction.
2. Use IAR to fetch from memory the instruction to be executed and store it in an instruction register (IR).
3. Use the CU to decode this instruction to determine the specific operation and the location of any data required.
4. Fetch any data required and store it in registers.
5. Execute the instruction using this data.
6. Increment the address in the IAR to the next machine language instruction.
7. Return to step 2.

This fetch-decode-execute cycle is fundamental to the operation of all CPU’s and is called interpretation. Thus, a CPU is an interpreter for a machine language program.

1.6 Computer languages

At the lowest level we can write programs directly in machine language as long strings of 0’s and 1’s. The only advantage is that the CPU directly understands this language so maximum efficiency and execution speed can be achieved. However, no one does this anymore because it is too difficult for all but the simplest programs.

The next step up is to write programs in assembly language. An assembly language is essentially a mnemonic form of machine language. Instructions have names and addresses can be represented symbolically with names rather than explicit 32-bit binary numbers. The CPU doesn’t understand assembly language so it is necessary to convert an assembly language program to a
1.6 Computer languages

Table 1.1: Intel 8088 machine language program and data in both hexadecimal and binary notation

<table>
<thead>
<tr>
<th>Instructions (hex)</th>
<th>Instructions (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 0000</td>
<td>1010 0001</td>
</tr>
<tr>
<td>03 06 0002</td>
<td>0000 0011 0000 0110</td>
</tr>
<tr>
<td>03 06 0004</td>
<td>0000 0011 0000 0110</td>
</tr>
<tr>
<td>A3 0006</td>
<td>1010 0011</td>
</tr>
<tr>
<td>Data (hex)</td>
<td>Data (binary)</td>
</tr>
<tr>
<td>00 06</td>
<td>0000 0000 0000 0110</td>
</tr>
<tr>
<td>00 07</td>
<td>0000 0000 0000 0111</td>
</tr>
<tr>
<td>01 FF</td>
<td>0000 0001 1111 1111</td>
</tr>
<tr>
<td>02 0C</td>
<td>0000 0110 0000 1100</td>
</tr>
</tbody>
</table>

Machine language program. This process of converting a program in one language to a program in another language is called translation or compilation. Of course, it is necessary to have a machine language program to do the translation. The translator that converts an assembly language program to a machine language program is called an assembler. Once this translation process is completed the resulting machine language program can be stored in memory and executed by the CPU.

To briefly illustrate these ideas we will use a machine language program for an Intel 8088 processor (processor used in the original IBM PC) that simply adds three integers.

1.6.1 Machine and assembly languages

To understand what machine and assembly language programs looks like let us consider a small Intel 8088 program to add three numbers and store the results. For this processor the standard size for an integer is 16 bits. This means that each integer occupies two consecutive bytes of memory. Therefore we assume that the three numbers to be added are stored in 6 bytes of memory and 2 further bytes of memory are reserved for storing the sum. Therefore the data part of the program occupies 8 bytes. The processor has an instruction to move a 16-bit integer from memory into a processor register, an instruction to add an integer in memory to a number in the register, and an instruction to move the number in a register back to memory again. Therefore four machine language instructions are required to add the three numbers and store the result. The four machine language instructions and the data are shown in Table 1.1. The right column shows the binary format and the left column shows the more compact hexadecimal notation (in base 10 the 10 digits ‘0’ to ‘9’ are used, in base 2 the 2 digits ’0’ and ’1’ are used, and in base 16 ’0’ to ’9’ and the six new “digits” ’A’ to ’F’ are used).

The first instruction, A1 0000, occupies 3 bytes. The first byte, A1, is called the opcode and means that this is an instruction that moves a 16-bit number into a specific processor register called ax. The remaining two bytes contain a special address used to locate the 16-bit number in a special area of memory called the data segment. The second and third instructions are addition instructions and the fourth instruction is another move instruction that stores the result in the data segment.

The final four rows of the table show the 6 bytes in the data segment containing the three
numbers to be added and the 2 bytes reserved for the answer. In the particular case shown the sum, using decimal notation is $6 + 7 + 511$, and the result ($524$ decimal, $020C$ hexadecimal) is show in the last two bytes. Thus, the four-instruction program is just the string of 112 bits:

```
1010 0001 0000 0000 0000 0000 0000 0011 0000 0110 0000 0000 000 0 0010
0000 0011 0000 0110 0000 0000 0000 0100 1010 0011 0000 0000 0000 0010
```

and is not very intelligible. Nevertheless, this is the only program understood by the processor.

In order to make programming at this level manageable assembly languages were invented. In an assembly language symbolic names are given to instruction opcodes, to processor registers, and to memory addresses. For example, our four instruction machine language program in Table 1.1 is shown in assembly language in Table 1.2. For example, the instruction

```
mov ax, i
```

corresponds to the first three bytes ($A1 0000$) of the machine language program. Instructions that move data from one place to another have the name `mov` and one of the 16-bit processor registers has the name `ax`. Also we have used the symbolic names `i`, `j`, `k` for the addresses of the three integers to be added, and the name `sum` for the address of the memory location that will hold the result. These symbolic names are used as labels on `dw` directives (`dw` means “define word”) which allocate memory for data in 16-bit units. After the statements are executed by the processor the result ($524$ decimal, $020C$ hex) will be stored in the two bytes reserved for `sum` in the last row of Table 1.2. We don’t care about the actual addresses: the assembler will take care of it. It is clear that this program is much more understandable than the 112-bit string of 0’s and 1’s in machine language.

### 1.6.2 High-level languages

Even though an assembly language program is much easier to write and read than a machine language program, it is still very far from the familiar mathematical notation used at the human-level of problem solving which we might express as shown in Figure 1.4.
Therefore, most programs today are written in a high-level language such as C, C++, or Java. For example, the statements in Figure 1.5 show that Java statements that correspond to the machine language program in Table 1.1 or the assembly language program in Table 1.2. Even with no programming experience these statements can probably be understood. The \texttt{int} modifier indicates that the name of the following variable refers to an integer number (a 32-bit integer in Java) and the equal sign indicates that the variable on the left of the equal sign receives the value of the expression on the right side of the equal sign. The equal sign corresponds to the left arrow that we have used to denote assignment of a value to a variable.

### 1.7 Translation and interpretation of programs

Since the CPU only understands machine language it is necessary to convert programs written in assembly language or a high-level language to machine language. There are three ways to do this

1. Write a program that converts the source program to a machine language program for a specific CPU and then have the CPU execute this program directly. This is called translation. The translation of an assembly language program to a machine language program is done by an assembler. In general a program that converts a program in one language to a program in another language is called a \textbf{compiler}. This process is illustrated in Figure 1.6.

2. Write a program called an \textbf{interpreter} that reads the source program one statement at a time.
and has each statement executed by the CPU. Early versions of the BASIC programming language were executed in this way. This interpretation process is shown in Figure 1.7.

3. Write a compiler program that translates the high-level language source program to a program in some intermediate machine language representation for a hypothetical machine called a virtual machine. Now write an interpreter for this machine that has the program executed on a real CPU. The interpreter can also compile groups of statements so that, if they are encountered again, they do not need to be re-interpreted again.

Interpreters are more flexible than compilers but they can result in much slower execution since each high-level language statement must be converted “on the fly” each time the statement is encountered. In program loops, where the same sequence of statements can be executed many times, execution will be slower than for the corresponding machine language program for the real CPU. The benefits of both interpretation and compilation can be achieved by the hybrid approach (item 3 above) which is often called “just-in-time” compilation. Most Java program language interpreters can use this hybrid approach to achieve speeds approaching that of the machine language for the CPU.

### 1.8 Java virtual machine

Java is a peculiar high level language since it is first compiled, not to the machine language of a specific real computer, but rather to a machine neutral object code called bytecode. This is illustrated in part (a) of Figure 1.8. The bytecode is the machine language for a hypothetical (virtual) machine called the Java Virtual Machine (or JVM). The Java bytecode and the Java compiler are both computer independent. Once a bytecode program is obtained it can be run on any JVM. The JVM is the only machine dependent part of the process, shown in Figure 1.8(b). Here the JVM interprets the bytecode instructions, converts them to the machine language of the real machine, and has them executed on the real machine. Essentially, it is the Java interpreter.

Why does the process of running Java programs require an extra step? While it is possible to design compilers or interpreters to convert Java to a specific machine language, converting it first to Java bytecode has the advantage of portability. A bytecode program does not depend on the hardware of any real computer system. It can be run on any computer system without the need for recompilation as long as someone has written a JVM for the target computer system. Today a JVM is available for almost every computer system.
1.9 Java source code to bytecode translation example

The entire process of translating a program from source code to assembly language to object code can be illustrated by the Java statements in Figure 1.5 that define three integer variables, \( i \), \( j \), and \( k \) having the values 6, 7, and 511, respectively, then add them together and store the result in a variable called \( \text{sum} \).

The Java compiler translates these statements into the language of the Java Virtual Machine. Table 1.3 shows three forms of this translation. The first column shows the 12 bytecode instructions in assembly language for the four Java statements. They are not as recognizable as the Java statements, and are quite different from the Intel 8088 instructions in Table 1.2 but you can still see familiar words such as “load”, “store”, and “add”. The \textit{bipush} instructions store the two byte size integers, 6 and 7, in an area of memory called the stack, the \textit{sipush} instruction does the same for the 16-bit size integer 511, the \textit{istore} instructions store data from the stack into memory, and the \textit{iload} instructions load the stack with the numbers to be added. Finally, the \textit{iadd} instructions add the numbers and the result is stored in memory.

The second column shows the bytecode in hexadecimal format, and the final column shows the bytecode in binary form. Thus, the 3 Java statements ultimately result in the following sequence of 0’s and 1’s:

```
0001 0000 0000 0110 0011 1100 0001 0000 0000 0111 0011 1101 0001 0001 0000 0001 1111 1111 0011 1110 0001 1011 0001 1100 0110 0000 0001 1101 0110 0000 0011 0110 0000 0100
```

This program is actually longer than required since it was obtained from the output of the Java compiler which is assuming that local variables are used for the variables \( i \), \( j \), \( k \), and \( \text{sum} \). If we were to write directly in assembly language the program could be expressed more simply as

```
bipush 6
bipush 7
iadd
```
Table 1.3: Example of Java source code to bytecode translation

```
sipush 511
iadd
```
which uses the stack area of memory to add three numbers and store the result.

1.10 Review exercises

▲ Exercise 1.1 Think of something that you know how to do and try to write an algorithm in English that could be followed by someone who has never done it before.

▲ Exercise 1.2 Use your calculator to execute “by hand” the algorithm in Figure 1.1. Compare your answer with the one produced using the square root key.

▲ Exercise 1.3 How would you modify the algorithm in Figure 1.1 to compute an approximation to $\sqrt{a}$ for any $a > 0$? Test your algorithm by computing approximations to $\sqrt{3}$, $\sqrt{4}$, $\sqrt{100}$, and $\sqrt{10000}$. Note: a good way to test algorithms is to try them using data for which you know the correct answer (e.g., $\sqrt{10000}$ is 100).