CHAPTER 1

Introduction

Recent advances in microelectronic technology have made computers an integral part of our society. Each step in our everyday lives is influenced by computer technology: we awake to a digital alarm clock's beaming of preselected music at the right time, drive to work in a digital processor–controlled automobile, work in an extensively automated office, shop for computer-coded grocery items, and return to rest in the computer-regulated heating and cooling environment of our homes. It may not be necessary to understand the detailed operating principles of a jet plane or an automobile to use and enjoy the benefits of these technical marvels. Computer systems technology has also reached the level of sophistication wherein an average user need not be familiar with all the intricate technical details of an operation to use them efficiently. Computer scientists, engineers, and application developers, however, require a fair understanding of the operating principles, capabilities, and limitations of digital computers to enable the development of complex yet efficient and user-friendly systems. This book is designed to give such an understanding of the operating principles of digital computers. This chapter begins by describing the organization of a general-purpose digital computer system and briefly traces the evolution of computers.

1.1 COMPUTER SYSTEM ORGANIZATION

The primary function of a digital computer is to process data input to it to produce results that can be better used in a specific application environment. For example, consider a digital computer used to control the traffic light at an intersection. The *input* data are the number of cars passing through the intersection during a specified time period, the processing consists of the computation of red–yellow–green time periods as a function of the number of cars, and the *output* is the variation of the red–yellow–green time intervals based on the results of processing. In this system, the data input device is a sensor that can detect the passing of a car at the intersection. Traffic lights are the output devices. The electronic device that keeps track of the number of cars and computes the red–yellow–green time periods is the processor. These physical devices constitute the hardware components of the system. The processing hardware is programmed to compute the red–yellow–green time periods according to some rule. This rule is the *algorithm* used to solve the particular problem. The algorithm (a logical sequence of steps to solve a problem) is translated into a program (a set of instructions) for the processor to follow in solving the problem. Programs are written in a language "understandable" by the processing hardware. The collection of such programs constitutes the software component of the computer system.

1.1.1 Hardware

The traffic light controller is a very simple special-purpose computer system requiring only a few of the physical hardware components that constitute a general-purpose computer system (see Figure 1.1). The four major hardware blocks of a general-purpose computer system are its memory unit (MU), arithmetic and logic unit (ALU), input/output unit (IOU), and control unit (CU). Input/output (I/O) devices input and output data into and out of the MU. In some systems, I/O devices send and receive data into and from the ALU rather than the MU. Programs reside in the MU. The ALU processes the data taken from the MU (or the ALU) and stores the processed data back in the MU (or the ALU). The CU coordinates the activities of the other three units. It retrieves instructions from programs resident in the MU, decodes these instructions, and directs the ALU to perform corresponding processing steps. It also oversees I/O operations.

Some representative I/O devices are shown in Figure 1.1. A keyboard and a mouse are the most common input devices nowadays. Touch screens are taking over as common interaction devices. A video display and a printer are the most common output devices. Scanners are used to input data from hard copy sources. Magnetic tapes and disks are used as I/O devices. These devices are also used as memory devices to increase the capacity of the MU. The console is a special-purpose I/O device that permits the system operator to interact with the computer system. In modern-day computer systems, the console is typically a dedicated terminal.

1.1.2 Software

The hardware components of a computer system are electronic devices in which the basic unit of information is either a 0 or a 1, corresponding to two states of an electronic signal. For instance, in one of the popular hardware technologies, a 0 is represented by 0 V, while a 1 is represented by 5 V. Programs and data must therefore be expressed using this binary alphabet consisting of 0 and 1. Programs written using only these binary digits are *machine language* programs. At this level of programming, operations such as ADD and SUBTRACT are each represented by

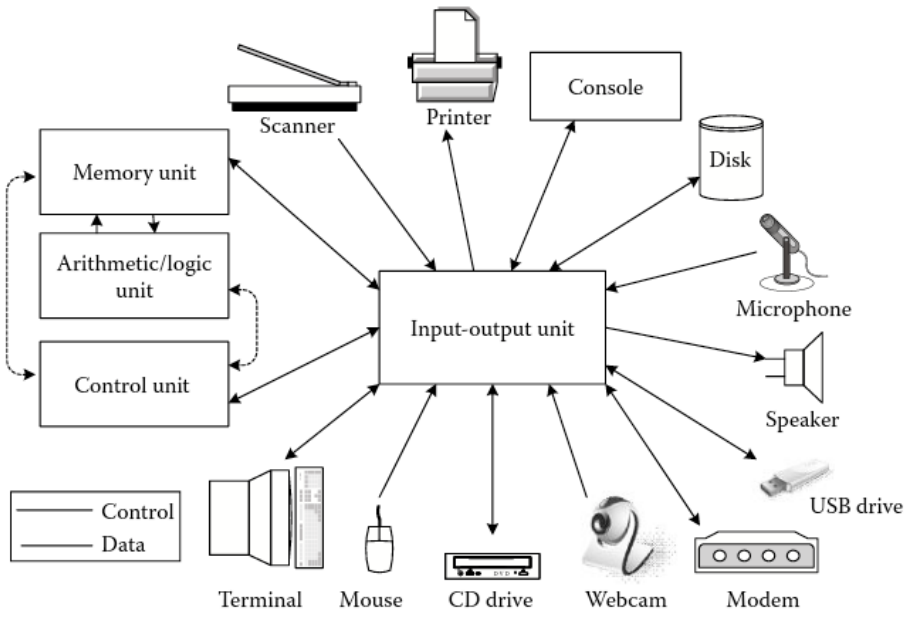

Figure 1.1 Typical computer system.

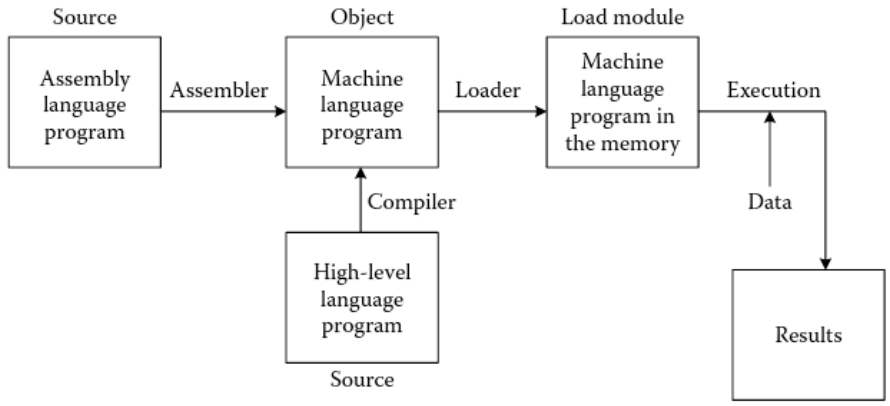

Figure 1.2 Program translation and execution.

a unique pattern of 0s and 1s, and the computer hardware is designed to interpret these sequences. Programming at this level is tedious since the programmer has to work with sequences of 0s and 1s and needs to have very detailed knowledge of the computer structure.

The tedium of machine language programming is partially alleviated by using symbols such as ADD and SUB rather than patterns of 0s and 1s for these operations. Programming at the symbolic level is called *assembly language* programming. An assembly language programmer is also required to have a detailed knowledge of the machine structure, because the operations permitted in the assembly language are primitive and the instruction format and capabilities depend on the hardware organization of the machine. An assembler program is used to translate assembly language programs into machine language.

Use of high-level programming languages such as FORTRAN, COBOL, C, and JAVA further reduces the requirement of an intimate knowledge of the machine organization. A compiler program is needed to translate a high-level language program into the machine language. A separate compiler is needed for each high-level language used in programming the computer system. Note that the assembler and the compiler are also programs written in one of those languages and can translate an assembly or high-level language program, respectively, into the machine language.

Figure 1.2 shows the sequence of operations that occurs once a program is developed. A program written in either the assembly language or a high-level language is called a source program. An assembly language source program is translated by the assembler into the machine language program. This machine language program is the object code. A compiler converts a high-level language source into an object code. The object code ordinarily resides on an intermediate device such as a magnetic disk or tape. A loader program loads the object code from the intermediate device into the MU. The data required by the program will be either available in the memory or supplied by an input device during the execution of the program. The effect of program execution is the production of processed data or results.

1.1.3 System

Operations such as selecting the appropriate compiler for translating the source into object code; loading the object code into the MU; and starting, stopping, and accounting for the computer system usage are automatically done by the system. A set of supervisory programs that permit such automatic operation is usually provided by the computer system manufacturer. This set, called the *operating system*, receives the information it needs through a set of command language statements from the user and manages the overall operation of the computer system.

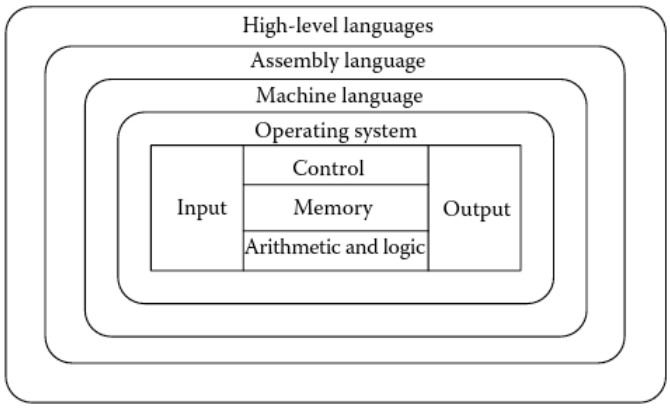

Figure 1.3 Hardware and software components.

Operating system and other utility programs used in the system may reside in a memory block that is typically read only. Special devices are needed to write these programs into read-only memory (ROM). Such programs and commonly used data are termed firmware. Figure 1.3 is a simple rendering of the complete hardware–software environment of a general-purpose computer system.

1.2 COMPUTER EVOLUTION

Man has always been in search of mechanical aids for computation. The development of the abacus around 3000 BC introduced the positional notation of number systems. In seventeenth-century France, Pascal and Leibnitz developed mechanical calculators that were later developed into desk calculators. In 1801, Jacquard used punched cards to instruct his looms in weaving various patterns on cloth.

In 1822, Charles Babbage, an Englishman, developed the difference engine, a mechanical device that carried out a sequence of computations specified by the settings of levers, gears, and cams. Data were entered manually as the computations progressed. Around 1820, Babbage proposed the analytical engine, which would use a set of punched cards for program input, another set of cards for data input, and a third set of cards for output of results. The mechanical technology was not sufficiently advanced and the analytical engine was never built; nevertheless, the analytical engine as designed probably was the first computer in the modern sense of the word.

Several unit-record machines to process data on punched cards were developed in the United States in 1880 by Herman Hollerith for census applications. In 1944, Mark I, the first automated computer, was announced. It was an electromechanical device that used punched cards for input and output of data and paper tape for program storage. The desire for faster computations than those Mark I could provide resulted in the development of the electronic numerical integrator and computer (ENIAC), the first electronic computer built out of vacuum tubes and relays by a team led by Americans Eckert and Mauchly. ENIAC employed the *stored-program concept* in which a sequence of instructions (i.e., the program) is stored in the memory for use by the machine in processing data. ENIAC had a control board on which the programs were wired. A rewiring of the control board was necessary for each computation sequence.

John von Neumann, a member of the Eckert–Mauchly team, developed the electronic discrete variable automatic computer (EDVAC), the first stored-program computer. At the same time, Wilkes developed the electronic delay storage automatic calculator (EDSAC), the first operational stored-program machine, which also introduced the concept of primary and secondary memory hierarchy.

von Neumann is credited for developing the stored-program concept, beginning with his 1945 first draft of EDVAC. The structure of EDVAC established the organization of the stored-program computer (von Neumann machine), which contains

1. An input device through which data and instructions can be entered
2. A storage unit into which results can be entered and from which instructions and data can be fetched
3. An arithmetic unit to process data
4. A CU to fetch, interpret, and execute the instructions from the storage
5. An output device to deliver the results to the user

All contemporary computers are von Neumann machines, although various alternative architectures have evolved.

1.2.1 von Neumann Model

Figure 1.4 shows the von Neumann model, a typical uniprocessor computer system consisting of the MU, the ALU, the CU, and the IOU. The MU is a single-port device consisting of a memory address register (MAR) and a memory buffer register (MBR)—also called a memory data register (MDR). The memory cells are arranged in the form of several memory words, where each word is the unit of data that can be read or written. All the read and write operations on the memory utilize the memory port. The ALU performs the arithmetic and logic operations on the data items in the accumulator (ACC) and/or MBR and typically the ACC retains the results of such operations. The CU consists of a program counter (PC) that contains the address of the instruction to be fetched and an instruction register (IR) into which the instructions are fetched from the memory for execution. Two registers are included in the structure. These can be used to hold the data and address values during computation. For simplicity, the I/O subsystem is shown to input to and output from the ALU subsystem. In practice, the I/O may also occur directly between the memory and I/O devices without utilizing any processor registers. The components of the system are interconnected by a multiple-bus structure on which the data and addresses flow. The CU manages this flow through the use of appropriate control signals.

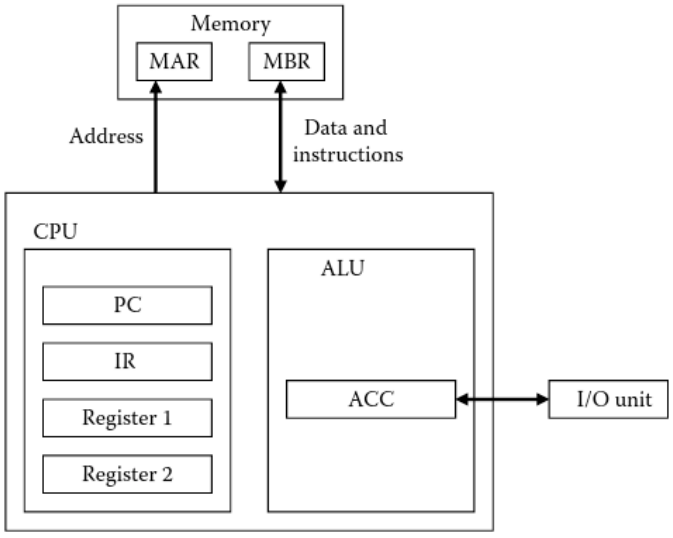

Figure 1.4 von Neumann architecture.

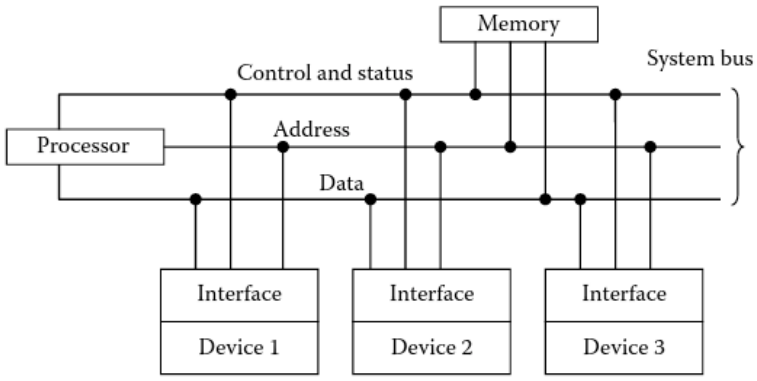

Figure 1.5 General computer system.

Figure 1.5 shows a more generalized computer system structure representative of modern-day architectures. The processor subsystem (i.e., the central processing unit—CPU) now consists of the ALU and CU and various processor registers. The processor, memory, and I/O subsystems are interconnected by the system bus, which consists of data, address, control, and status lines.

Practical systems may differ from the single-bus architecture of Figure 1.5, in the sense that they may be configured around multiple buses. For instance, there may be a memory bus that connects the processor to the memory subsystem and an I/O bus to interface I/O devices to the processor, forming a two-bus structure. Further, it is possible to configure a system with several I/O buses wherein each bus may interface one type of I/O device to the processor. Since multiple-bus structures allow simultaneous operations on the buses, a higher throughput is possible, compared to single-bus architectures. However, because of the multiple buses, the system complexity increases. Thus, a speed/cost trade-off is required to decide on the system structure. The processor subsystem may also consist of several processors. Each processor may be dedicated to a specific computational task (arithmetic/logic, graphics and display, I/O handling, etc.), or processors as a group may share the overall computational load as the load changes. The memory subsystem may also contain several modules and types of memory.

It is important to note the following characteristics of the von Neumann model that make it inefficient:

1. Programs and data are stored in a single sequential memory, which can create a memory access "bottleneck."
2. There is no explicit distinction between data and instruction representations in the memory. This distinction has to be brought about by the CPU during the execution of programs.
3. High-level language programming environments utilize several data structures (such as single and multidimensional arrays and linked lists). The memory, being 1D, requires that such data structures be linearized for representation.
4. The data representation does not retain any information on the type of data. For instance, there is nothing to distinguish a set of bits representing floating-point data from that representing a character string. Such distinction has to be brought about by the program logic.

Because of these characteristics, the von Neumann model is overly general and requires excessive mapping by compilers to generate the code executable by the hardware from the programs written in high-level languages. This problem is termed as the *semantic* gap. In spite of these deficiencies, the von Neumann model has been the most practical structure for digital computers. The central concept in the model devised by von Neumann was inspired by the theoretical work of Allan Turing and Kurt Godel. Several efficient compilers have been developed over the years, which have narrowed the semantic gap to the extent that it is almost invisible for a high-level language programming environment.

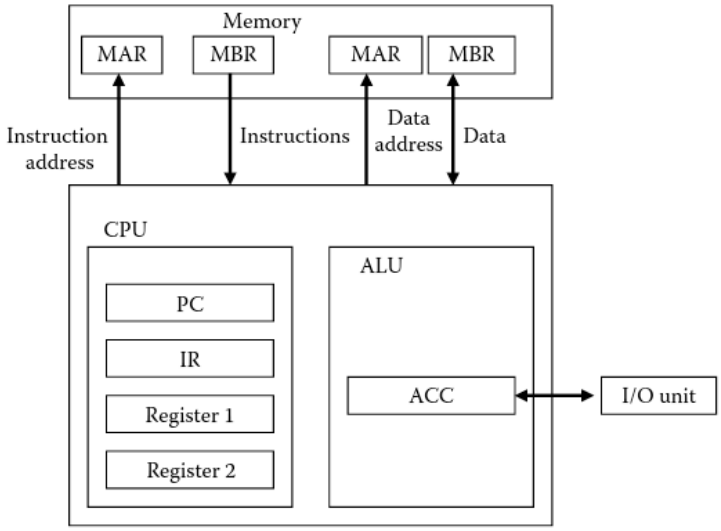

Figure 1.6 Harvard architecture.

Note that the von Neumann model of Figure 1.4 provides one path for addresses and a second path for data and instructions, between the CPU and the memory. An early variation of this model is the Harvard architecture shown in Figure 1.6. This architecture provides independent paths for data addresses, data, instruction addresses, and instructions. This allows the CPU to access instruction and data simultaneously. The name *Harvard architecture* is due to Howard Aiken's work on Mark I through Mark IV computers at Harvard University. These machines had separate storage for data and instructions. Current Harvard architectures do not use separate storage for data and instructions but have separate paths and buffers to access data and instructions simultaneously.

Early enhancements to the von Neumann model were mainly concentrated on increasing the speed of the basic hardware structure. As the hardware technology progressed, efforts were made to incorporate as many high-level language features as possible into hardware and firmware in an effort to reduce the semantic gap.

Note that the hardware enhancements alone may not be sufficient to attain the desired performance. The architecture of the overall computing environment starting from the algorithm development to execution of programs needs to be analyzed to arrive at the appropriate hardware, software, and firmware structures. If possible, these structures should exploit the parallelism in the algorithms themselves. Thus, performance enhancement is one reason for *parallel processing*. There are also other reasons such as reliability, fault tolerance, expandability, and modular development that dictate parallel processing structures. We will introduce parallel processing concepts further later in the book.

1.2.2 Generations of Computer Technology

Commercial computer system development has followed development of hardware technology and is usually divided into five generations:

1. First generation (1945–1955)—vacuum tube technology
2. Second generation (1955–1965)—transistor technology
3. Third generation (1965–1980)—integrated circuit (IC) technology
4. Fourth generation (1980s–present)—very large-scale integrated (VLSI) circuit technology
5. Fifth generation (present and beyond)—artificial intelligence-based computing devices, extensive parallel processing structures, using superconductors, nanotechnology, and quantum computing

We will not elaborate on the architectural details of the various machines developed during these generations, except for the following brief evolution account.

First-generation machines such as UNIVAC 1 and IBM 701, built out of vacuum tubes, were slow and bulky and accommodated a limited number of I/O devices. Magnetic tape was the predominant I/O medium. Data access time was measured in milliseconds.

Second-generation machines (IBM 1401, 7090; RCA 501; CDC 6600; Burroughs 5500; DEC PDP-1) used random-access core memories, transistor technology, multifunctional units, and multiple processing units. Data access time was measured in microseconds. Assembler and high-level languages were developed.

The IC technology used in third-generation machines such as the IBM 360, UNIVAC 1108, ILLIAC-IV, and CDC STAR-100 contributed to nanosecond data access and processing times. Multiprogramming, array, and pipeline processing concepts came into being.

Computer systems were viewed as general-purpose data processors until the introduction in 1965 of DEC PDP-8, a *minicomputer*. Minicomputers were regarded as dedicated application machines with limited processing capability compared to that of large-scale machines. Since then, several new minicomputers have been introduced and this distinction between the mini- and large-scale machines is becoming blurred due to advances in hardware and software technology.

The development of *microprocessors* in the early 1970s allowed a significant contribution to the third class of computer systems: microcomputers. Microprocessors are essentially computers on an IC chip that can be used as components to build a dedicated controller or processing system. Advances in IC technology leading to the current VLSI era have made microprocessors as powerful as minicomputers of the 1970s. VLSI-based systems are called fourth-generation systems since their performance is so much higher than that of third-generation systems.

Modern computer system architecture exploits the advances in hardware and software technologies to the fullest extent. Due to advances in IC technology that make the hardware much less expensive, the architectural trend is to interconnect several processors to form a high-throughput system.

We are now witnessing the development of fifth-generation systems. There is no accepted definition of what a fifth-generation computer is. Fifth-generation development efforts in the United States involve building supercomputers with very high computational capability, large memory capacity, and flexible multiple-processor architectures, employing extensive parallelism. Japanese fifth-generation activities aimed toward building artificial intelligence-based machines with very high numeric and symbolic processing capabilities, large memories, and user-friendly natural interfaces. Some attribute fifth generation to biology-inspired (neural networks, DNA) computers and optical computer systems.

The current generation of computer systems exploits parallelism in algorithms and computations to provide high performance. The simplest example of parallel architecture is the Harvard architecture, which utilizes two buses operating simultaneously. Parallel processing architectures utilize a multiplicity of processors and memories operating concurrently. We will describe various parallel and pipelined architecture structures later in this book.

1.2.3 Moore's Law

The progress in hardware technology resulting in the current VLSI era has given us the capability to fabricate millions of transistors on an IC (chip). This has allowed us to build chips with powerful processing structures with large memory systems. The early supercomputers such as CDC 6600 had about 128 kB of memory and could perform 10 million instructions per second. Today's supercomputers contain thousands of processors and terabytes of memory and perform trillions of operations per second. All this is possible because of the miniaturization of transistors. How far can this miniaturization continue? In 1965, Gordon Moore (Founder of Intel Corporation) stated that "the density

of transistors in an IC will double every year." This so-called Moore's law is now modified to state that "the density of chips doubles every 18 months." This law has held true for more than 40 years and many believe that it will continue to hold for a few more decades. Arthur Rock proposed a corollary to Moore's law that states that "the cost of capital equipment to manufacture ICs will double every four years." Indeed, the cost of building new IC fabrication facilities has escalated from about $10,000 in the 1960s to $10 million in the 1990s to about $3 billion today. Thus, the cost might get prohibitive even though the technology allows building denser chips. Newer technologies and processing paradigms are continually being invented to achieve this cost/technology compromise.

1.3 ORGANIZATION VERSUS DESIGN VERSUS ARCHITECTURE

Computer organization addresses issues such as types and capacity of memory, control signals, register structure, and instruction set. It answers the question "How does a computer work?" basically from a programmer's point of view. Architecture is the art or science of building, a method or style of building. Thus, a computer architect develops the performance specifications for various components of the computer system and defines the interconnections between them. A computer designer, on the other hand, refines these component specifications and implements them using hardware, software, and firmware elements. An architect's capabilities are greatly enhanced if he or she is also exposed to the design aspects of the computer system.

A computer system can be described at the following levels of detail:

1. Processor-memory-switch (PMS) level, at which an architect views the system. It is simply a description of system components and their interconnections. The components are specified to the block diagram level.
2. Instruction set level, at which level the function of each instruction is described. The emphasis of this description level is on the behavior of the system rather than the hardware structure of the system.
3. Register transfer level, at which level the hardware structure is more visible than previous levels. The hardware elements at this level are registers that retain the data that are processed until the current phase of processing is complete.
4. Logic gate level, at which level the hardware elements are logic gates and flip-flops. The behavior is now less visible, while the hardware structure predominates.
5. Circuit level, at which level the hardware elements are resistors, transistors, capacitors, and diodes.
6. Mask level, at which level the silicon structures and their layout that implements the system as an IC are shown.

As one moves from the first level of description toward the last, it is evident that the behavior of the machine is transformed into a hardware–software structure. A computer architect concentrates on the first two levels described earlier, whereas the computer designer takes the system design to the remaining levels.

1.4 PERFORMANCE EVALUATION

Several measures of performance have been used in the evaluation of computer systems. The most common ones are million instructions per second (MIPS), million operations per second (MOPS), million floating-point operations per second (MFLOPS or megaflops), billion floating-point operations per second (GFLOPS or gigaflops), and million logical inferences per second (MLIPS). Machines capable of trillion floating-point operations per second (teraflops) are now available. Table 1.1 lists the common prefixes used for these measures. Power-of-10 prefixes shown

Table 1.1 Common Prefixes Used in Computer Systems Measurements

Power of 10	Power of 2	Prefix	Symbol
Thousand (10^3)	2^{10}	Kilo	k
Million (10^6)	2^{20}	Mega	M
Billion (10^9)	2^{30}	Giga	G
Trillion (10^{12})	2^{40}	Tera	T
Quadrillion (10^{15})	2^{50}	Peta	P
Quintillion (10^{18})	2^{60}	Exa	E
Sextillion (10^{21})	2^{70}	Zetta	Z
Septillion (10^{24})	2^{80}	Yotta	Y
Thousandth (10^{-3})	2^{-10}	Milli	m
Millionth (10^{-6})	2^{-20}	Micro	μ
Billionth (10^{-9})	2^{-30}	Nano	n
Trillionth (10^{-12})	2^{-40}	Pico	p
Quadrillionth (10^{-15})	2^{-50}	Femto	f
Quintillionth (10^{-18})	2^{-60}	Atto	a
Sextillionth (10^{-21})	2^{-70}	Zepto	z
Septillionth (10^{-24})	2^{-80}	Yocto	y

in the first column are typically used for power, frequency, voltage, and computer performance measurements. Power-of-2 prefixes shown in the second column are typically used for memory, file, and register sizes. Columns three and four of the table show the prefixes and symbols used in denoting the power-of-2 prefixes.

The measure used depends on the type of operations one is interested in, for the particular application for which the machine is being evaluated. As such, these measures have to be based on the mix of operations representative of their occurrence in the application.

The performance rating could be either the peak rate (i.e., the MIPS rating the CPU cannot exceed) or the more realistic average or sustained rate. In addition, a comparative rating that compares the average rate of the machine to that of other well-known machines is also used. In addition to the performance, other factors considered in evaluating architectures are generality (how wide is the range of applications suited for this architecture), ease of use, and expandability or scalability. One feature that is receiving considerable attention now is the openness of the architecture. The architecture is said to be open if the designers publish the architecture details such that others can easily integrate standard hardware and software systems to it. The other guiding factor in the selection of architecture is the cost.

Several analytical techniques are used in estimating the performance. All these techniques are approximations, and as the complexity of the system increases, most of these techniques become unwieldy. A practical method for estimating the performance in such cases is using benchmarks.

1.4.1 Benchmarks

Benchmarks are standardized batteries of programs run on a machine to estimate its performance. The results of running a benchmark on a given machine can then be compared with those on a known or standard machine, using criteria such as CPU and memory utilization and throughput and device utilization.

Benchmarks are useful in evaluating hardware as well as software and single processor as well as multiprocessor systems. They are also useful in comparing the performance of a system before and after certain changes are made.

As a high-level language host, a computer architecture should execute efficiently those features of a programming language that are most frequently used in actual programs. This ability is often measured by benchmarks. Benchmarks are considered to be representative of classes of applications envisioned for the architecture.

There are several benchmark suites in use today and more are being developed. It is important to note that the benchmarks provide only a broad performance guideline. It is the responsibility of the user to select the benchmark that comes close to his application and further evaluate the machine based on scenarios expected in the application for which the machine is being evaluated. One simple example for benchmark of general-purpose computers is drawing polygons, typically triangles, with single/multiple colors at a specified frame rate. This benchmark exercises the computer's processing capacity related to generating diagrams and also colors. Another very popular example is calculating the fast Fourier transform (FFT) of standard functions. This exercises all the common arithmetic operations like addition and subtraction. Chapter 16 provides further details on benchmarks and performance evaluation.

1.5 SUMMARY

This chapter has introduced the basic terminology associated with modern computer systems. The five common levels of architecture abstractions were introduced along with a trace of the evolution of computer generations. Performance issues and measures were briefly introduced. Subsequent chapters of the book expand on the concepts and issues highlighted in this chapter.