COMPUTER ARCHITECTURE

by

Prof. Dr. Mostafa Y. Makkey

Contents

Chapter 1
Introduction

1.1 Architecture & Organization:

In describing computers, a distinction is often made between **computer architecture** and **computer organization**. Although it is difficult to give precise definitions for these terms, a consensus exists about the general areas covered by each.

Computer architecture refers to those attributes of a system visible to a programmer or, put another way, those attributes that have a direct impact on the logical execution of a program.

Computer organization refers to the operational units and their interconnections that realize the architectural specifications.

Examples of architectural attributes include the instruction set, the number of bits used to represent various data types (e.g., numbers, characters), I/O mechanisms, and techniques for addressing memory. Organizational attributes include those hardware details transparent to the programmer, such as control signals; interfaces between the computer and peripherals; and the memory technology used.

For example, it is an architectural design issue whether a computer will have a multiply instruction. It is an organizational issue whether that instruction will be implemented by a special multiply unit or by a mechanism that makes repeated use of the add unit of the system. The organizational decision may be based on the anticipated frequency of use of the multiply instruction, the relative speed of the two approaches, and the cost and physical size of a special multiply unit.

Historically, and still today, the distinction between architecture and organization has been an important one. Many computer manufacturers offer a family of computer models, all with the same architecture but with differences in organization. Consequently, the different models in the family have different price and performance characteristics. Furthermore, a particular architecture may span many years and encompass a number of different computer models, its organization changing with changing technology. A prominent example of both these phenomena is the IBM System/370 architecture. This architecture was first introduced in 1970 and included a number of models. The customer with modest requirements could buy a cheaper, slower model and, if demand increased, later upgrade to a more expensive, faster model without having to abandon software that had already been developed.

Over the years, IBM has introduced many new models with improved technology to replace older models, offering the customer greater speed, lower cost, or both. These newer models retained the same architecture so that the customer's software investment was protected. Remarkably, the System/370 architecture, with a few enhancements, has survived to this day as the architecture of IBM's mainframe product line.

In a class of computers called microcomputers, the relationship between architecture and organization is very close. Changes in technology not only influence organization but also result in the introduction of more powerful and more complex architectures. Generally, there is less of a requirement for generation-to-generation compatibility for these smaller machines. Thus, there is more interplay between organizational and architectural design decisions.

1.2 STRUCTURE AND FUNCTION:

A computer is a complex system; contemporary computers contain millions of elementary electronic components. A hierarchical system is a set of interrelated subsystems until we reach some lowest level of elementary subsystem. The designer need only deal with a particular level of the system at a time. At each level, the system consists of a set of components and their interrelationships. The behavior at each level depends only on a simplified, abstracted characterization of the system at the next lower level. At each level, the designer is concerned with structure and function:

- Structure: The way in which the components are interrelated
- Function: The operation of each individual component as part of the structure

The computer system is described from the top down. We begin with the major components of a computer, describing their structure and function, and proceed to successively lower layers of the hierarchy.

1.2.1 Function

Both the structure and functioning of a computer are, in essence, simple.

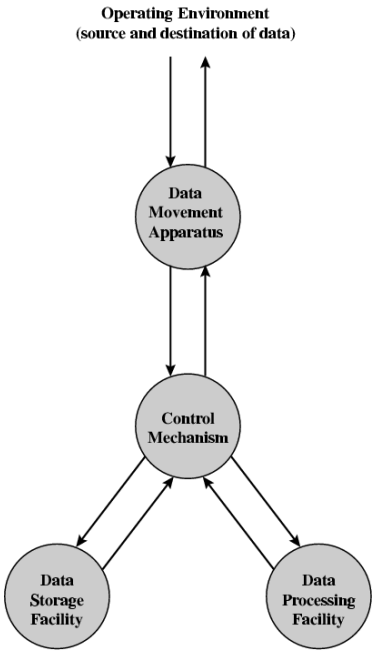


Figure 1.1 A Functional View of the Computer

Figure 1.1 depicts the basic functions that a computer can perform. In general terms, there are only four:

- **Data processing:** The data may take a wide variety of forms, and the range of processing requirements is broad. However, we shall see that there are only a few fundamental methods or types of data processing.
- **Data storage**: It is also essential that a computer store data. Even if the computer is processing data on the fly, the computer must temporarily store at least those pieces of data that are being worked on at any given moment. Thus, there is at least a **short-term** data storage function. Equally important, the computer performs a **long-term** data storage function. Files of data are stored on the computer for subsequent retrieval and update.
- **Data movement**: The computer must be able to move data between itself and the outside world. The computer's operating environment consists of devices that serve as either sources or destinations of data. When data are received from or delivered to a device that is directly connected to the computer, the process is known as **input–output (I/O)**, and the device is referred to as a **peripheral**. When data are moved over longer distances, to or from a remote device, the process is known as **data communications**.
- **Control**: Within the computer, a control unit manages the computer's resources and orchestrates the performance of its functional parts in response to those instructions.

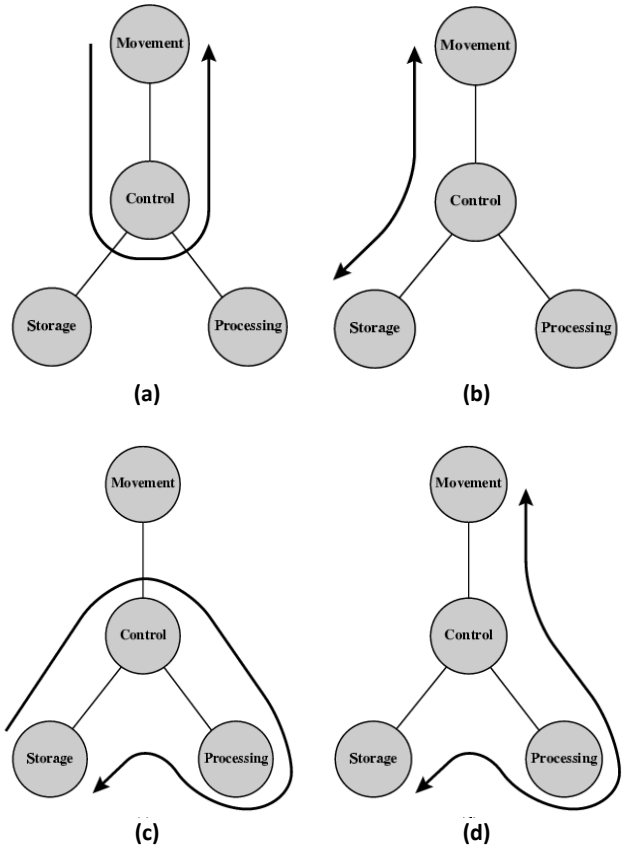


Figure 1.2 Possible Computer Operations

At this general level of discussion, the number of possible operations that can be performed is few. Figure 1.2 depicts the four possible types of operations. The computer can function as a data movement device (Figure 1.2a), simply transferring data from one peripheral or communications line to another. It can also function as a data storage device (Figure 1.2b), with data transferred from the external environment to computer storage (read) and vice versa (write). The final two diagrams show operations involving data processing, on data either in storage (Figure 1.2c) or route between storage and the external environment (Figure 1.2d).

1.2.2 Structure

There are four main structural components as shown in Figure 1.3:

- **Central processing unit (CPU)**: Controls the operation of the computer and performs its data processing functions; often simply referred to as processor.
- **Main memory:** Stores data.
- **I/O:** Moves data between the computer and its external environment.
- **System interconnection:** Some mechanism that provides for communication among CPU, main memory, and I/O. A common example of system interconnection is by means of a **system bus**, consisting of a number of conducting wires to which all the other components attach.

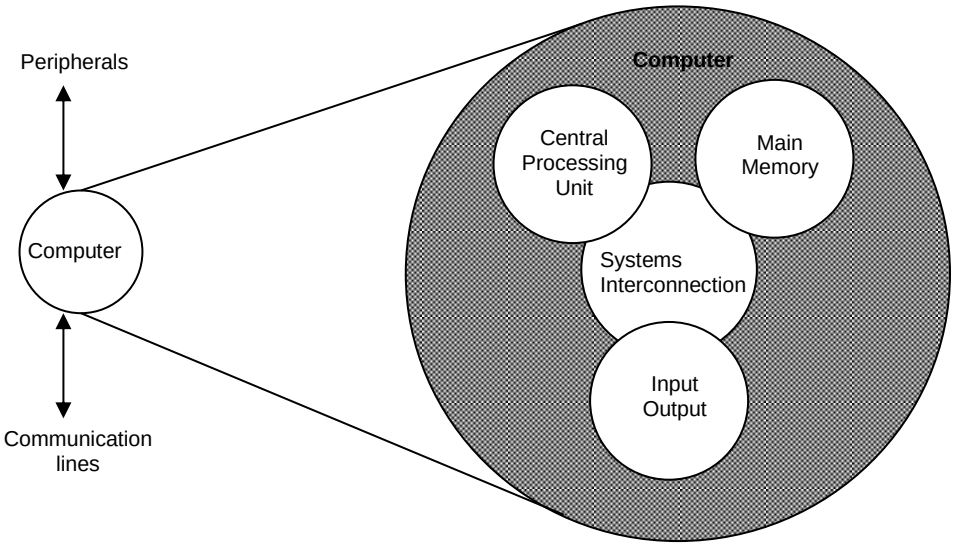


Figure 1.3 The Computer: Top-Level Structure

There may be one or more of each of the aforementioned components. Traditionally, there has been just a single processor. In recent years, there has been increasing use of multiple processors in a single computer.

The most interesting and in some ways the most complex component is the CPU. Its major structural components as shown in Figure 1.4 are:

- **Control unit**: Controls the operation of the CPU and hence the computer.
- **Arithmetic and logic unit (ALU)**: Performs the computer's data processing functions.

- **Registers**: Provides storage internal to the CPU.
- **CPU interconnection**: Some mechanism that provides for communication among the control unit, ALU, and registers

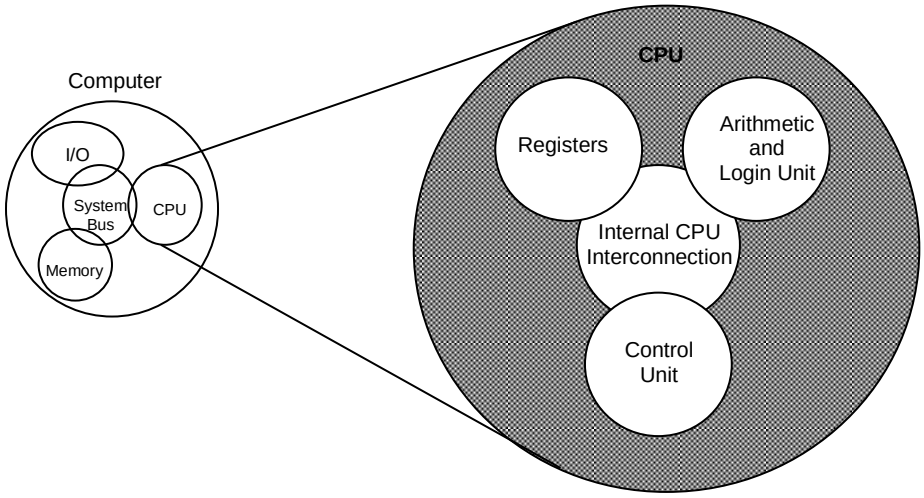


Figure 1.4 CPU: Top-Level Structure

Finally, there are several approaches to the implementation of the control unit; one common approach is a **microprogrammed** implementation. In essence, a microprogrammed control unit operates by executing microinstructions that define the functionality of the control unit. With this approach, the structure of the control unit can be depicted, as in Figure 1.5.

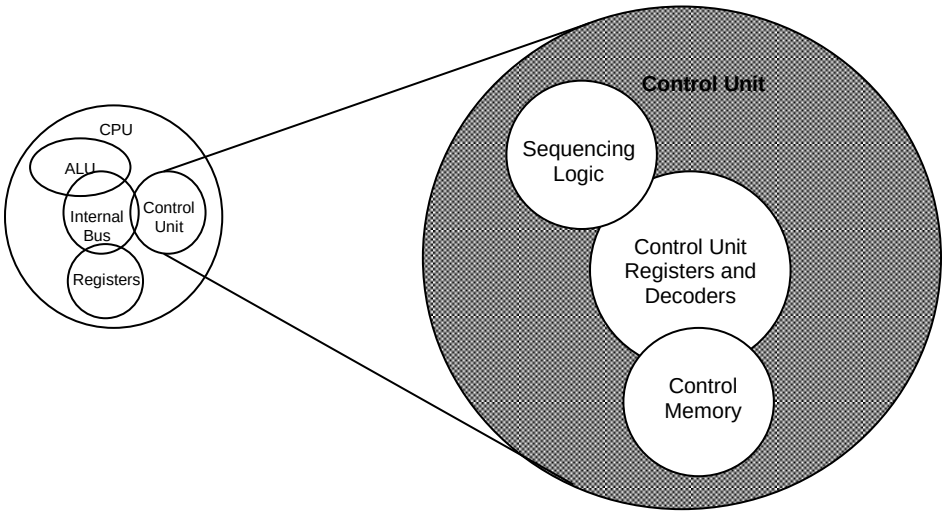


Figure 1.5 Control Unit: Top-Level Structure

Chapter 2
Computer Evolution and Performance

2.1 A Brief History of Computers

2.1.1 The First Generation: Vacuum Tubes

John Mauchly, a professor of electrical engineering at the University of Pennsylvania, and John Eckert, one of his graduate students, proposed to build a general-purpose computer using vacuum tubes for the BRL's application. In 1943, the Army accepted this proposal, and work began on the **ENIAC** (Electronic Numerical Integrator And Computer). The resulting machine was enormous, weighing 30 tons, occupying 1500 square feet of floor space, and containing more than 18,000 vacuum tubes. When operating, it consumed 140 kilowatts of power. It was also substantially faster than any electromechanical computer, capable of 5000 additions per second. The ENIAC was a decimal rather than a binary machine.

Figure 2.1 Vacuum Tubes

2.1.2 The Second Generation: Transistors

The first major change in the electronic computer came with the replacement of the vacuum tube by the transistor shown in Figure 2.2. The transistor is smaller, cheaper, and dissipates less heat than a vacuum tube but can be used in the same way as a vacuum tube to construct computers. Unlike the vacuum tube, which requires wires, metal plates, a glass capsule, and a vacuum, the transistor is a solid-state device, made from silicon. The transistor was invented at Bell Labs in 1947 and by the 1950s had launched an electronic revolution. It was not until the late 1950s, however, that fully transistorized computers were commercially available.

Figure 2.2 Transistors

But there are other changes as well. The second generation saw the introduction of more complex arithmetic and logic units and control units, the use of high level programming languages, and the provision of system software with the computer.

2.1.3 The Third Generation: Integrated Circuits

Early second-generation computers contained about 10,000 transistors. This makes the entire manufacturing process, from transistor to circuit board, was expensive and cumbersome. These facts of life were beginning to create problems in the computer industry.

In 1958 came the achievement that revolutionized electronics and started the era of **microelectronics**: the invention of the integrated circuit . It is the integrated circuit that defines the third generation of computers. In integrated circuit, many transistors can be produced at the same time on a single wafer of silicon shown in Figure 2.3. Initially, only a few gates or memory cells could be reliably manufactured and packaged together. These early integrated circuits are referred to as **small-scale integration(SSI)**.

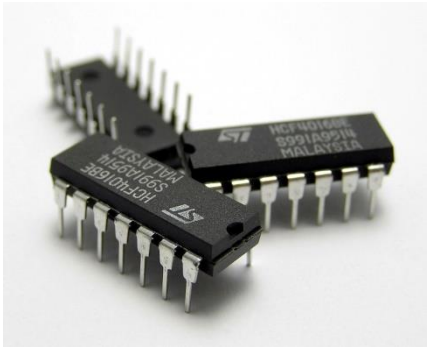


Figure 2.3 Integrated Circuits

Only two fundamental types of components are required and implemented as Integrated circuits:

1. gates and memory cells. A gate is a device that implements a simple Boolean or logical function, such as AND, OR, or XOR. Such devices are called gates because they control data flow in much the same way that canal gates do.

2. The memory cell (Flip-Flops) is a device that can store one bit of data; that is, the device can be in one of two stable states at any time.

By interconnecting large numbers of these fundamental devices, we can construct a computer. We can relate this to our four basic functions as follows:

- **Data storage:** Provided by memory cells.
- **Data processing:** Provided by gates.
- **Data movement**: The paths among components are used to move data from memory to memory and from memory through gates to memory.
- **Control**: The paths among components can carry control signals.

2.1.4 Later Generations

With the introduction of **largescale integration (LSI),** more than 1000 components can be placed on a single integrated circuit chip. **Very-large-scale integration (VLSI)** achieved more than 10,000 components per chip as shown in Figure 2.4, while current **ultra-large-scale integration (ULSI)** chips can contain more than one million components.

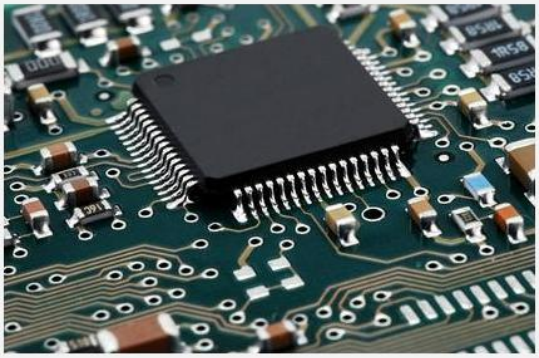


Figure 2.4 VLSI

With the rapid pace of technology, the high rate of introduction of new products, and the importance of software and communications as well as hardware, the classification by generation becomes less clear and less meaningful. It could be said that the commercial application of new developments resulted in a major change in the early 1970s and that the results of these changes are still being worked out.

2.2 Microprocessors

The microprocessor contains all of the components of a CPU on a single chip as shown in Figure 2.5. It was achieved in 1971, when Intel developed its 4004.

This evolution can be seen most easily in the number of bits that the processor deals with at a time. There is no clear-cut measure of this, but perhaps the best measure is the data bus width: the number of bits of data that can be brought into or sent out of the processor at a time. Another measure is the number of bits in the accumulator or in the set of general-purpose registers.

The next major step in the evolution of the microprocessor was the introduction in 1972 of the Intel 8008. This was the first 8-bit microprocessor and was almost twice as complex as the 4004.

About the same time, 16-bit microprocessors began to be developed. However, it was not until the end of the 1970s that powerful, general-purpose 16-bit microprocessors appeared. One of these was the 8086. The next step in this trend occurred in 1981, when both Bell Labs and Hewlett-Packard developed 32-bit, single-chip microprocessors. Intel introduced its own 32-bit microprocessor, the 80386, in 1985 (Table 2.6).

Figure 2.5 Microprocessor

2.3 Moore's Law

Initially, only a few gates or memory cells could be reliably manufactured and packaged together. As time went on, it became possible to pack more and more components on the same chip. This growth in density is illustrated in Figure 2.6; it is one of the most remarkable technological trends ever recorded. This figure reflects the famous Moore's law, which was propounded by Gordon Moore, cofounder of Intel, in 1965.

Moore observed that the number of transistors that could be put on a single chip was doubling every year and correctly predicted that this pace would continue into the near future.

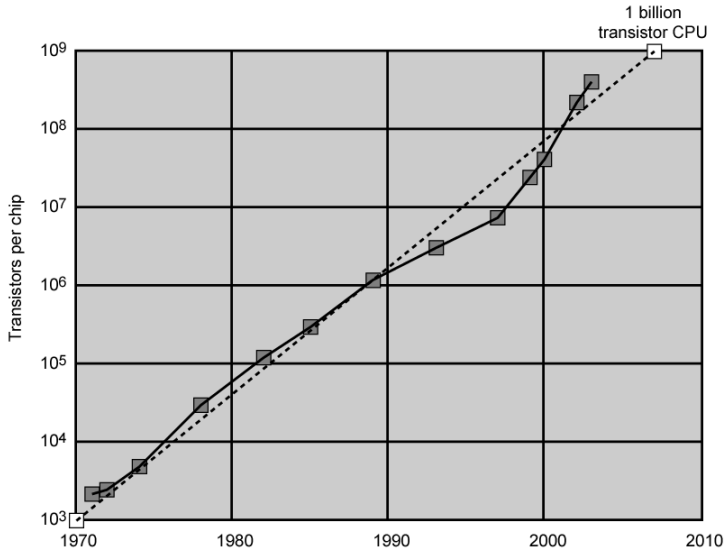


Figure 2.6 Growth in CPU Transistor Count

To the surprise of many, including Moore, the pace continued year after year and decade after decade. The pace slowed to a doubling every 18 months in the 1970s but has sustained that rate ever since.

The consequences of Moore's law are profound:

1. The cost of a chip has remained virtually unchanged during this period of rapid growth in density. This means that the cost of computer logic and memory circuitry has fallen at a dramatic rate
2. Because logic and memory elements are placed closer together on more densely packed chips, the electrical path length is shortened, increasing operating speed.
3. The computer becomes smaller, making it more convenient to place in a variety of environments.
4. There is a reduction in power and cooling requirements.
5. The interconnections on the integrated circuit are much more reliable than solder connections. With more circuitry on each chip, there are fewer inter-chip connections.

2.4 The Von Neumann Architecture

In 1946, von Neumann and his colleagues began the design of a new stored program computer, referred to as the IAS computer, at the Princeton Institute for Advanced Studies. The IAS computer, although not completed until 1952, is the prototype of all subsequent general-purpose computers. Figure 2.7 shows the general structure of the IAS computer which consists of:

- **main memory**, which stores both data and instructions.
- **An arithmetic and logic unit (ALU)** capable of operating on binary data.
- **A control unit**, which interprets the instructions in memory and causes them to be executed.
- **Input and output (I/O)** equipment operated by the control unit.

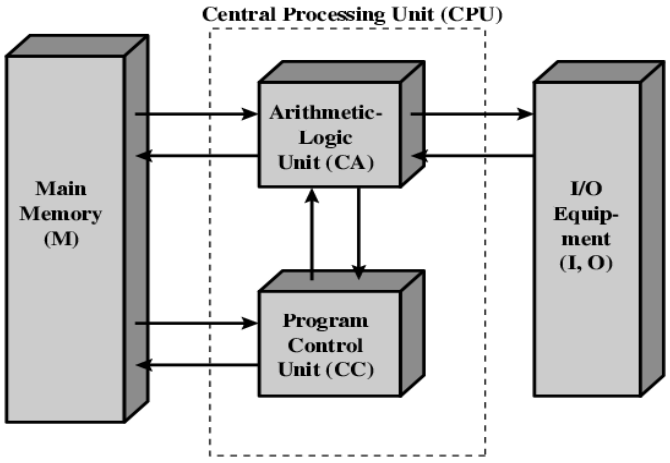


Figure 2.7 Structure of the IAS Computer

With rare exceptions, all of today's computers have this same general structure and function and are thus referred to as von Neumann machines. Thus, it is worthwhile at this point to describe briefly the operation of the IAS computer.

The control unit operates the IAS by fetching instructions from memory and executing them one at a time. To explain this, a more detailed structure diagram is needed, as indicated in Figure 2.8. This figure reveals that both the control unit and the ALU contain storage locations, called registers, defined as follows:

- **Memory buffer register (MBR)**: Contains a word to be stored in memory or sent to the I/O unit, or is used to receive a word from memory or from the I/O unit.
- **Memory address register (MAR)**: Specifies the address in memory of the word to be written from or read into the MBR.
- **Instruction register (IR)**: Contains the 8-bit opcode instruction being executed.
- **Instruction buffer register (IBR)**: Employed to hold temporarily the right hand instruction from a word in memory.
- **Program counter (PC)**: Contains the address of the next instruction-pair to be fetched from memory.
- **Accumulator (AC)** and **multiplier quotient (MQ)**: Employed to hold temporarily operands and results of ALU operations. For example, the result of multiplying two 40-bit numbers is an 80-bit number; the most significant 40 bits are stored in the AC and the least significant in the MQ.

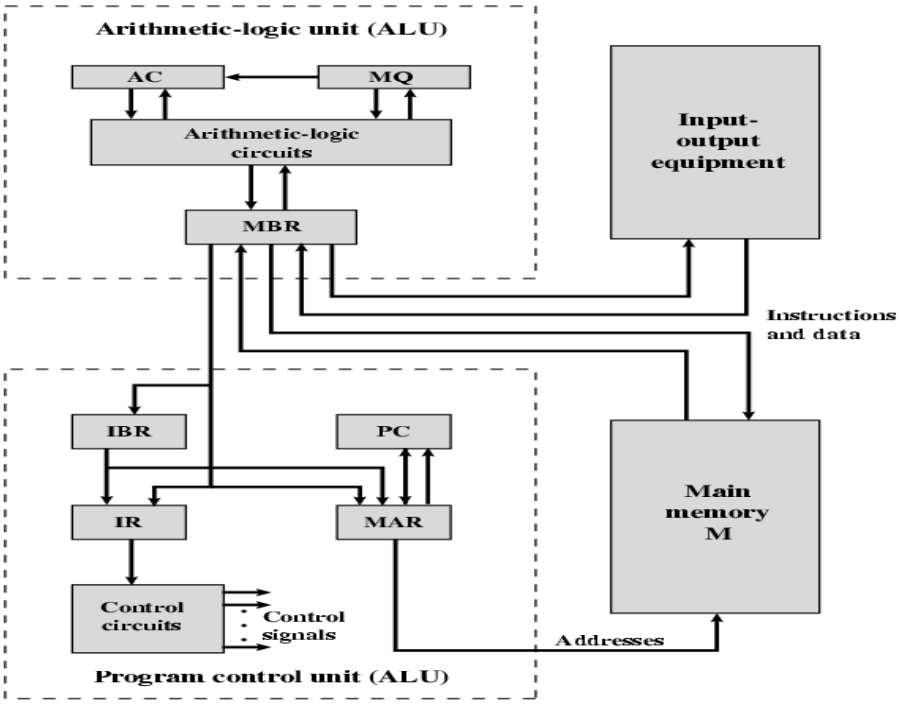


Figure 2.8 Expanded Structure of IAS Computer

2.5 Performance Assessment

2.5.1 Clock Speed and Instructions per Second

The system clock operations performed by a processor, such as fetching an instruction, decoding the instruction, performing an arithmetic operation, and so on, are governed by a system clock. Typically, all operations begin with the pulse of the clock. Thus, at the most fundamental level, the speed of a processor is dictated by the pulse frequency produced by the clock, measured in cycles per second, or Hertz (Hz).

Typically, clock signals are generated by a quartz crystal, which generates a constant signal wave while power is applied. This wave is converted into a digital voltage pulse stream that is provided in a constant flow to the processor circuitry (Figure 2.9). For example, a 1-GHz processor receives 1 billion pulses per second. The rate of pulses is known as the clock rate, or clock speed. One increment, or pulse, of the clock is referred to as a clock cycle, or a clock tick. The time between pulses is the cycle time.

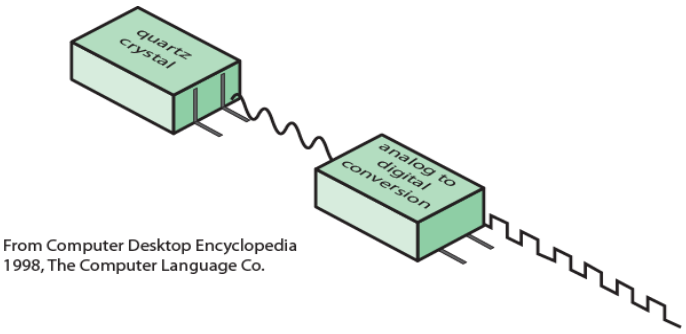


Figure 2.9 System Clock

The clock rate is not arbitrary, but must be appropriate for the physical layout of the processor. Actions in the processor require signals to be sent from one processor element to another. When a signal is placed on a line inside the processor, it takes some finite amount of time for the voltage levels to settle down so that an accurate value (1 or 0) is available. Furthermore, depending on the physical layout of the processor circuits, some signals may change more rapidly than others. Thus, operations must be synchronized and paced so that the proper electrical signal (voltage) values are available for each operation.

The execution of an instruction involves a number of discrete steps, such as fetching the instruction from memory, decoding the various portions of the instruction, loading and storing data, and performing arithmetic and logical operations. Thus, most instructions on most processors require multiple clock cycles to complete.

Some instructions may take only a few cycles, while others require dozens. In addition, when pipelining is used, multiple instructions are being executed simultaneously. Thus, a straight comparison of clock speeds on different processors does not tell the whole story about performance.

2.5.2 INSTRUCTION EXECUTION RATE

A processor is driven by a clock with a constant frequency f or, equivalently, a constant cycle time τ, where $\tau = 1/f$. Define the instruction count, I_c, for a program as the number of machine instructions executed for that program until it runs to completion or for some defined time interval. Note that this is the number of instruction executions, not the number of instructions in the object code of the program. An important parameter is the average cycles per instruction CPI for a program. If all instructions required the same number of clock cycles, then CPI would be a constant value for a processor. However, on any give processor, the number of clock cycles required varies for different types of instructions, such as load, store, branch, and so on. Let CPI_i be the number of cycles required for instruction type i. and I_i be the number of executed instructions of type i for a given program. Then, we can calculate an overall CPI as follows

$$CPI = \frac{\sum_{i=1}^{n} CPI_i \times I_i}{I_c}$$

The processor time T needed to execute a given program can be expressed as

$$T = I_c \times CPI \times \tau$$

We can refine this formulation by recognizing that during the execution of an instruction, part of the work is done by the processor, and part of the time a word is being transferred to or from memory. In this latter case, the time to transfer depends on the memory cycle time, which may be greater than the processor cycle time. We can rewrite the preceding equation as

$$T = I_c \times [\, p + (m \times k)] \times \tau$$

where p is the number of processor cycles needed to decode and execute the instruction, m is the number of memory references needed, and k is the ratio between memory cycle time and processor cycle time. The five performance factors in the preceding equation (I_c, p, m, k, τ) are influenced by four system attributes:

1- The design of the instruction set (known as **instruction set architecture**)
2- Compiler technology: how effective the compiler is in producing an efficient machine language program from a high-level language program
3- Processor implementation.
4- Cache and memory hierarchy.

Table 2.1 Performance Factors and System Attributes

	I_c	p	m	k	τ
Instruction set architecture	×	×			
Compiler technology	×	×	×		
Processor implementation		×			×
Cache and memory hierarchy				×	×

A common measure of performance for a processor is the rate at which instructions are executed, expressed as millions of instructions per second (MIPS), referred to as the **MIPS rate**. We can express the MIPS rate in terms of the clock rate and CPI as follows:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6}$$

Example 2.1: consider the execution of a program which results in the execution of 2 million instructions on a 400-MHz processor. The program consists of four major types of instructions. The instruction mix and the CPI for each instruction type are given below based on the result of a program trace experiment:

Instruction Type	CPI	Instruction Mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

Answer: The average CPI when the program is executed on a uniprocessor with the above trace results is $CPI = 0.6 + (2 \times 0.18) + (4 \times 0.12) + (8 \times 0.1) = 2.24$. The corresponding MIPS rate is $(400 \times 10^6)/(2.24 \times 10^6) \approx 178$.

Another common performance measure deals only with floating-point instructions. These are common in many scientific and game applications. Floating point performance is expressed as millions of floating-point operations per second (MFLOPS), defined as follows:

$$\text{MFLOPS rate} = \frac{\text{Number of executed floating point operations in a program}}{\text{Execution time} * 10^6}$$

2.5.3 Benchmarks

Measures such as MIPS and MFLOPS have proven inadequate to evaluating the performance of processors. Because of differences in instruction sets, the instruction execution rate is not a valid means of comparing the performance of different architectures. For example, consider this high-level language statement

```
A = B + C /* assume all quantities in main memory */
```

With a traditional instruction set architecture, referred to as a complex instruction set computer (CISC), this instruction can be compiled into one processor instruction:

```
add    mem(B), mem(C), mem (A)
```

On a typical RISC machine, the compilation would look something like this:

```
load   mem(B), reg(1);
load   mem(C), reg(2);
add    reg(1), reg(2), reg(3);
store  reg(3), mem(A)
```

Because of the nature of the RISC (reduced instruction set computer) architecture, both machines may execute the original high-level language instruction in about the same time. If this example is representative of the two machines, then if the CISC machine is rated at 1 MIPS, the RISC machine would be rated at 4 MIPS. But both do the same amount of high-level language work in the same amount of time.

Further, the performance of a given processor on a given program may not be useful in determining how that processor will perform on a very different type of application. Accordingly, beginning in the late 1980s and early 1990s, industry and academic interest shifted to measuring the performance of systems using a set of benchmark programs. The same set of programs can be run on different machines and the execution times compared.

The following as desirable characteristics of a benchmark program:

1. It is written in a high-level language, making it portable across different machines.
2. It is representative of a particular kind of programming style, such as systems programming, numerical programming, or commercial programming.
3. It can be measured easily.
4. It has wide distribution.

2.5.4 Averaging Results

To obtain a reliable comparison of the performance of various computers, it is preferable to run a number of different benchmark programs on each machine and then average the results. For example, if m different benchmark program, then a simple **arithmetic mean** can be calculated as follows:

$$R_A = \frac{1}{m}\sum_{i=1}^{m} R_i$$

where R_i is the high-level language instruction execution rate for the ith benchmark program.

An alternative is to take the **harmonic mean**:

$$R_H = \frac{m}{\sum_{i=1}^{m} \frac{1}{R_i}}$$

Ultimately, the user is concerned with the execution time of a system, not its execution rate. If we take arithmetic mean of the instruction rates of various benchmark programs, we get a result that is proportional to the sum of the inverses of execution times. But this is not inversely proportional to the sum of execution times. In other words, the arithmetic mean of the instruction rate does not cleanly relate to execution time. On the other hand, the harmonic mean instruction rate is the inverse of the average execution time.

SPEC benchmarks (standard benchmarks) do not concern themselves with instruction execution rates. Rather, two fundamental metrics are of interest: a **speed metric** and a rate metric. The speed metric measures the ability of a computer to complete a single task. SPEC defines a base runtime for each benchmark program using a reference machine. Results for a system under test are reported as the ratio of the reference run time to the system run time. The ratio is calculated as follows:

$$r_i = \frac{T_{ref_i}}{T_{sut_i}}$$

where T_{ref_i} is the execution time of benchmark program i on the reference system and T_{sut_i} is the execution time of benchmark program i on the system under test.

Because the time for the system under test is in the denominator, the larger the ratio, the higher the speed. An overall performance measure for the system under test is calculated by averaging the values for the ratios for all benchmarks. SPEC specifies the use of a **geometric mean**, defined as follows:

$$r_G = \left(\prod_{i=1}^{n} r_i \right)^{1/n}$$

where r_i is the ratio for the ith benchmark program.

Example 2.2: the Sun Blade 6250, which consists of two chips with four cores, or processors, per chip. One of the SPEC CPU2006 integer benchmark is 464.h264ref. This is a reference implementation of H.264/AVC (Advanced Video Coding), the latest state-of-the-art video compression standard. The Sun system executes this program in 934 seconds. The reference implementation requires 22,135 seconds. Calculate the benchmark ratio.

Answer: The ratio is calculated as: $22136/934 = 23.7$

Example 2.3: For the Sun Blade 6250, the SPEC integer speed ratios were reported as follows:

Benchmark Ratio	Ratio
400.perlbench	17.5
401.bzip2	14.0
403.gcc	13.7
429.mcf	17.6
445.gobmk	14.7
456.hmmer	18.6

Benchmark Ratio	Ratio
458.sjeng	17.0
462.libquantum	31.3
464.h264ref	23.7
471.omnetpp	9.23
473.astar	10.9
483.xalancbmk	14.7

Answer: The speed metric is calculated by taking the twelfth root of the product of the ratios:

$$(17.5 * 14 * 13.7 * 17.6 * 14.7 * 18.6 * 17 * 31.3 * 23.7 * 9.23 * 10.9 * 14.7)^{1/12} = 18.5$$

The **rate metric** measures the throughput or rate of a machine carrying out a number of tasks. For the rate metrics, multiple copies of the benchmarks are run simultaneously. Typically, the number of copies is the same as the number of processors on the machine. Again, a ratio is used to report results, although the calculation is more complex. The ratio is calculated as follows:

$$r_i = \frac{N \times T_{ref_i}}{T_{sut_i}}$$

where T_{ref_i} is the reference execution time for benchmark i, N is the number of copies of the program that are run simultaneously, and T_{sut_i} is the elapsed time from the start of the execution of the program on all N processors of the system under test until the completion of all the copies of the program. Again, a geometric mean is calculated to determine the overall performance measure.

2.5.5 Amdahl's Law

When considering system performance, computer system designers look for ways to improve performance by improvement in technology or change in design. Examples include the use of parallel processors, the use of a memory cache hierarchy, and speedup in memory access time and I/O transfer rate due to technology improvements. In all of these cases, it is important to note that a speedup in one aspect of the technology or design does not result in a corresponding improvement in performance. This limitation is succinctly expressed by Amdahl's law.

Amdahl's law was first proposed by Gene Amdahl in and deals with the potential speedup of a program using multiple processors compared to a single processor. Consider a program running on a single processor such that a fraction $(1 - f)$ of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead. Let T be the total execution time of the program using a single processor. Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on N parallel processors}}$$

$$= \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by 1/(1 f), so that there are diminishing returns for using more processors.

   Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system. Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}}$$

Suppose that a feature of the system is used during execution a fraction of the time f, before enhancement, and that the speedup of that feature after enhancement is SU_f. Then the overall speedup of the system is

$$\text{Speedup} = \frac{1}{(1-f)+\frac{f}{SU_f}}$$

Example 2.4: suppose that a task makes extensive use of floating-point operations, with 40% of the time is consumed by floating-point operations. With a new hardware design, the floating-point module is speeded up by a factor of K. Then the overall speedup is:

$$\text{Speedup} = \frac{1}{.6+\frac{.4}{k}}$$

Chapter 3
A Top-Level View Of Computer Function And Interconnection

3.1 Program Concept

As discussed in Chapter 2, virtually all contemporary computer designs are based on concepts developed by John von Neumann at the Institute for Advanced Studies, Princeton. Such a design is referred to as the von Neumann architecture and is based on three key concepts:

1. Data and instructions are stored in a single read–write memory.
2. The contents of this memory are addressable by location, without regard to the type of data contained there.
3. Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

Before von Neumann architecture, The computer was a small set of basic logic components that can be combined in various ways to store binary data and to perform arithmetic and logical operations on that data. If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed. We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting "program" is in the form of hardware and is termed a **hardwired program**.

Now consider this alternative. Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware. In the original case of customized hardware, the system accepts data and produces results (Figure 3.1a). With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.

The entire program is actually a sequence of steps. At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed. Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals (Figure 3.1b).

Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes. Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a sequence of codes or instructions is called software.

Figure 3.1b indicates two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU. Several other components are needed to yield a functioning computer. Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system. A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as **I/O components**.

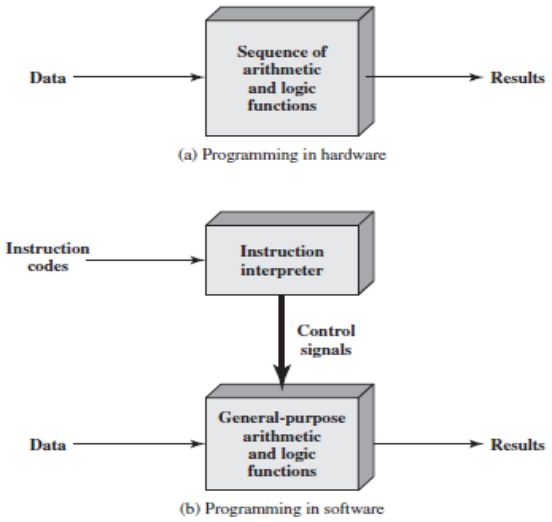


Figure 3.1 Hardware and Software Approaches

Figure 3.2 illustrates these top-level components and suggests the interactions among them. The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a memory address register (**MAR**), which specifies the address in memory for the next read or write, and a memory buffer register (**MBR**), which contains the data to be written into memory or receives the data read from memory. Similarly, an I/O address register (**I/OAR**) specifies a particular I/O device. An I/O buffer (**I/OBR**) register is used for the exchange of data between an I/O module and the CPU.

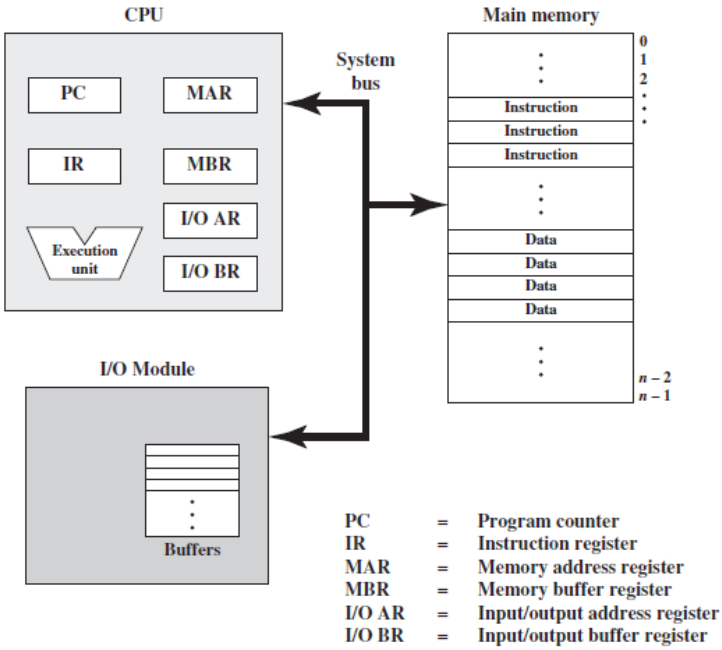


Figure 3.2 Computer Components:Top-Level View

A memory module consists of a set of locations, defined by sequentially numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data. An

I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.

3.2 Computer Function:

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program. This section provides an overview of the key elements of program execution. In its simplest form, instruction processing consists of two steps:

- The processor reads (fetches) instructions from memory one at a time and executes each instruction.
- Program execution consists of repeating the process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an **instruction cycle**. Using the simplified two-step description given previously, the instruction cycle is depicted in Figure 3.3. The two steps are referred to as the fetch cycle and the execute cycle. Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

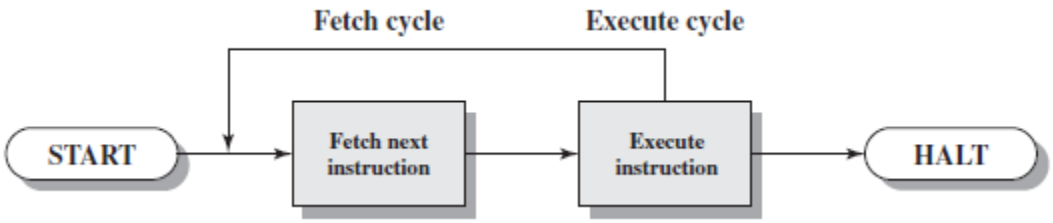


Figure 3.3 Basic Instruction Cycle

3.2.1 Instruction Fetch and Execute

At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter **(PC)** holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence.

So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to location 300.The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.

The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories:

- Processor-memory: Data may be transferred from processor to memory or from memory to processor.
- Processor-I/O: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- Data processing: The processor may perform some arithmetic or logic operation on data.
- Control: An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182.Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

Consider a simple example using a hypothetical machine that includes the characteristics listed in Figure 3.4. The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long. Thus, it is convenient to organize memory using 16-bit words. The instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.

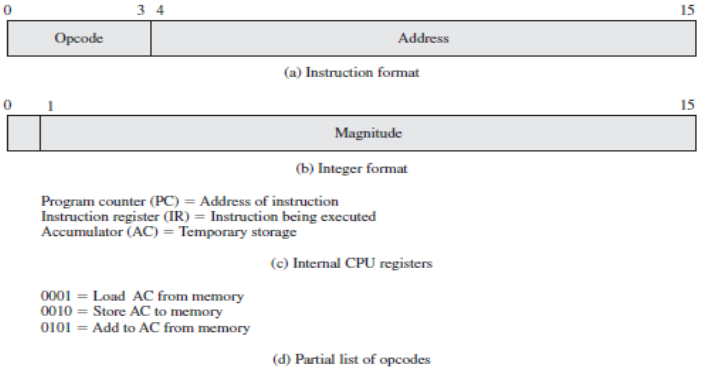


Figure 3.4 Characteristics of a Hypothetical Machine

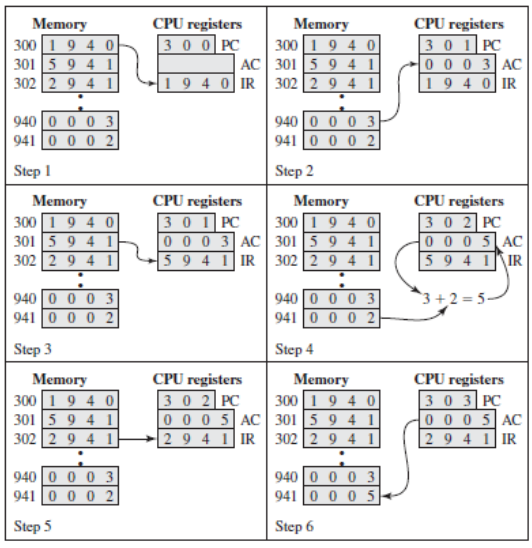


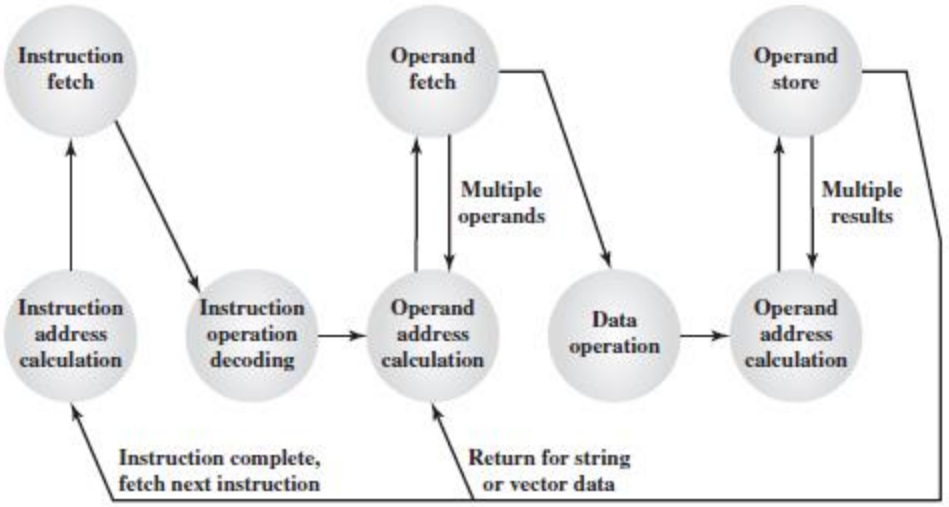
Figure 3.5 Example of Program Execution (contents of memory and registers in hexadecimal)

Figure 3.5 illustrates a partial program execution, showing the relevant portions of memory and processor registers. The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location. Three instructions, which can be described as three fetch and three execute cycles, are required:

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR and the PC is incremented. Note that this process involves the use of a memory address register (MAR) and a memory buffer register (MBR). For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301 and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.
6. The contents of the AC are stored in location 941.

Figure 3.6 provides a more detailed look at the basic instruction cycle of Figure 3.3 .The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once.The states can be described as follows:

- **Instruction address calculation (iac)**: Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
- **Instruction fetch (if)**: Read instruction from its memory location into the processor.
- **Instruction operation decoding (iod)**: Analyze instruction to determine type of operation to be performed and operand(s) to be used.
- **Operand address calculation (oac)**: If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
- **Operand fetch (of)**: Fetch the operand from memory or read it in from I/O.
- **Data operation (do)**: Perform the operation indicated in the instruction.
- **Operand store (os)**: Write the result into memory or out to I/O.



- **Figure 3.6 Instruction Cycle State Diagram**

3.2.2 Interrupts

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal processing of the processor. Table 3.1 lists the most common classes of interrupts.

Table 3.1 Classes of Interrupts

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure such as power failure or memory parity error.

Interrupts are provided primarily as a way to improve processing efficiency. For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme of Figure 3.3. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory. Clearly, this is a very wasteful use of the processor.

Figure 3.7a illustrates this state of affairs. The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.

- The actual I/O command.Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device).The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
- A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

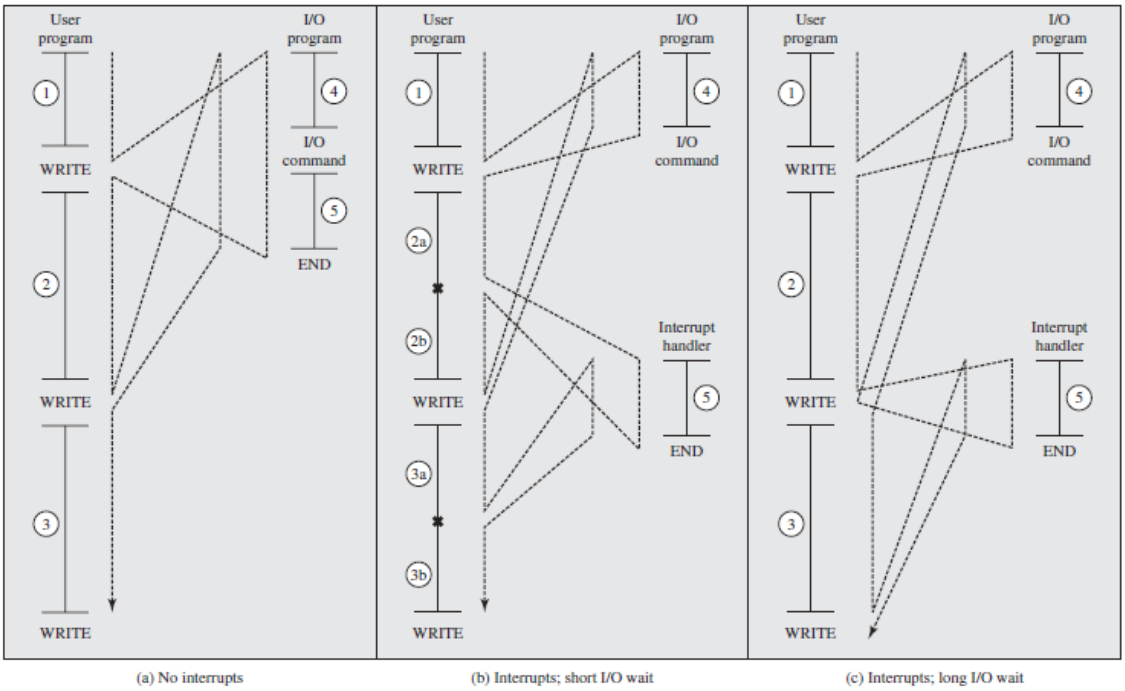


Figure 3.7 Program Flow of Control without and with Interrupts

3.2.3 Interrupts And The Instruction Cycle

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in Figure 3.7b. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program. Meanwhile, the external device is busy accepting data from computer memory and printing it.This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor,—the I/O module for that external device sends an interrupt request signal to the processor. The processor responds by suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler**, and resuming the original execution after the device is serviced. The points at which such interrupts occur are indicated by an asterisk in Figure 3.7b.

From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes (Figure 3.8). Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

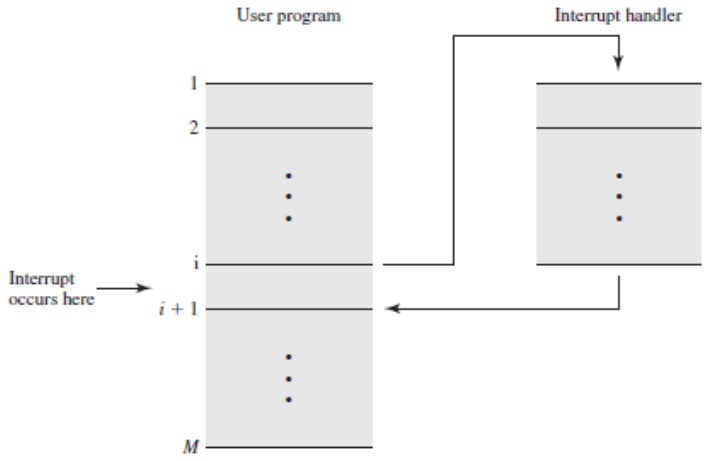


Figure 3.8 Transfer of Control via Interrupts

To accommodate interrupts, an interrupt cycle is added to the instruction cycle, as shown in Figure 3.9. In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed (current contents of the program counter) and any other data relevant to the processor's current activity.
- It sets the program counter to the starting address of an interrupt handler routine.

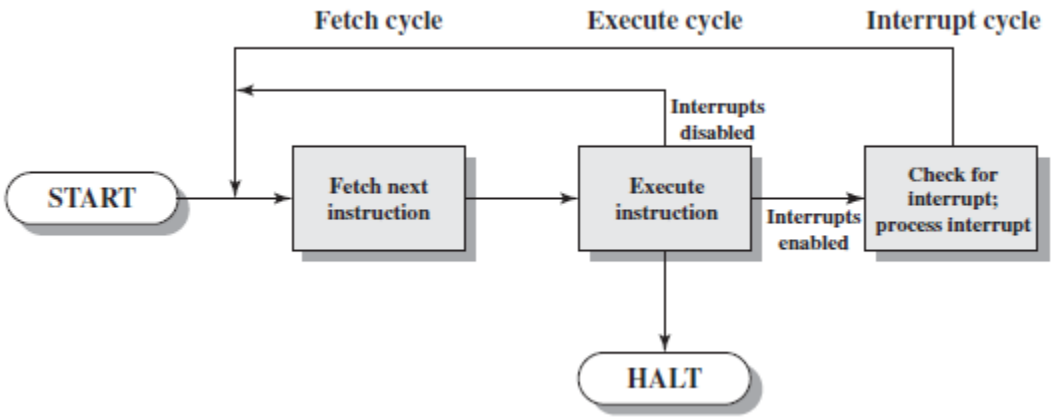


Figure 3.9 Instruction Cycle with Interrupts

The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this program determines the nature of the interrupt and performs whatever actions are needed. In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

Figure 3.10 shows a revised instruction cycle state diagram that includes interrupt cycle processing.

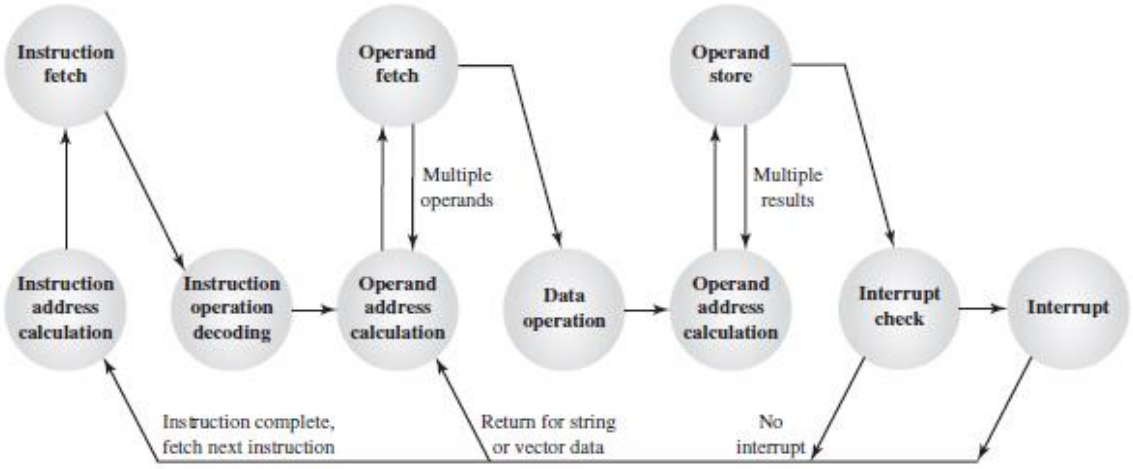


Figure 3.10 Instruction Cycle State Diagram, with Interrupts

3.3 Interconnection Structures

A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.

The collection of paths connecting the various modules is called the interconnection structure. The design of this structure will depend on the exchanges that must be made among modules.

Figure 3.11 suggests the types of exchanges that are needed by indicating the major forms of input and output for each module type:

- **Memory**: Typically, a memory module will consist of N words of equal length. Each word is assigned a unique numerical address (0, 1, . . . ,N – 1). A word of data can be read from or written into the memory. The nature of the operation is indicated by read and write control signals. The location for the operation is specified by an address.
- **I/O module**: From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations, read and write. Further, an I/O module may control more than one external device. We can refer to each of the interfaces to an external device as a port and give each a unique address (e.g., 0, 1, . . . ,M– 1). In addition, there are

external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.

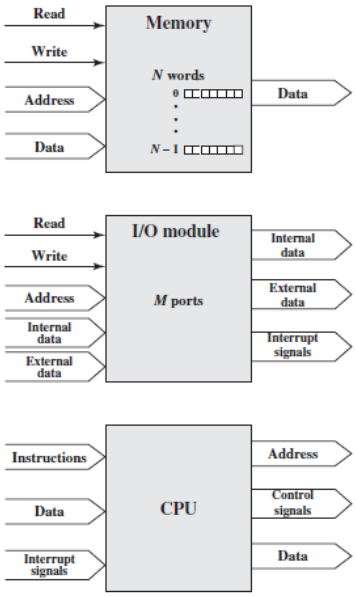


Figure 3.11 Computer Modules

- **Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.

The preceding list defines the data to be exchanged. The interconnection structure must support the following types of transfers:

- **Memory to processor:** The processor reads an instruction or a unit of data from memory.
- **Processor to memory**: The processor writes a unit of data to memory.
- **I/O to processor**: The processor reads data from an I/O device via an I/O module.
- **Processor to I/O**: The processor sends data to the I/O device.
- **I/O to or from memory**: For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access (DMA).

3.4 Bus Interconnection

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium. Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus. If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Over time, a sequence of binary digits can be transmitted across a single line. Taken together, several lines of a bus can be used to transmit

binary digits simultaneously (in parallel). For example, an 8-bit unit of data can be transmitted over eight bus lines.

Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a system bus. The most common computer interconnection structures are based on the use of one or more system buses.

3.5 Bus Structure

A system bus consists, typically, of from about 50 to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups (Figure 3.12): data, address, and control lines. In addition, there may be power distribution lines that supply power to the attached modules.

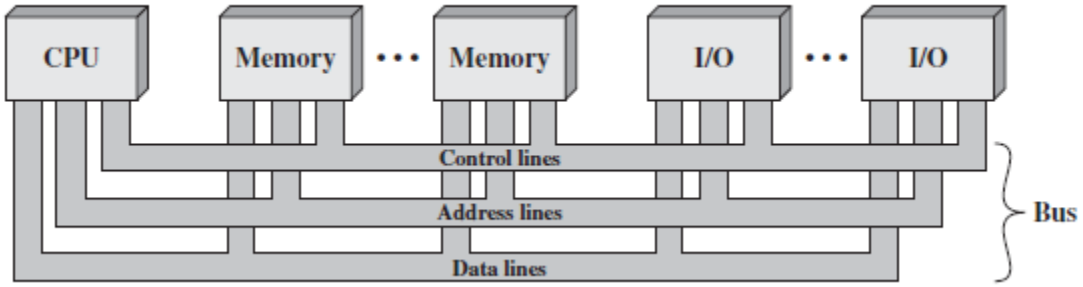


Figure 3.12 Bus Interconnection Scheme

The **data lines** provide a path for moving data among system modules. These lines, collectively, are called the data bus. The data bus may consist of 32, 64, 128, or even more separate lines, the number of lines being referred to as the width of the data bus. Because each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a time. The width of the data bus is a key factor in determining overall system performance. For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.

The **address lines** are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the address bus determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports. Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module. For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

The **control lines** are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of

controlling their use. Control signals transmit both command and timing information among system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include

- **Memory write**: Causes data on the bus to be written into the addressed location.
- **Memory read**: Causes data from the addressed location to be placed on the bus.
- **I/O write**: Causes data on the bus to be output to the addressed I/O port.
- **I/O read:** Causes data from the addressed I/O port to be placed on the bus.
- **Transfer ACK**: Indicates that data have been accepted from or placed on the bus.
- **Bus request**: Indicates that a module needs to gain control of the bus.
- **Bus grant**: Indicates that a requesting module has been granted control of the bus.
- **Interrupt request**: Indicates that an interrupt is pending.
- **Interrupt ACK**: Acknowledges that the pending interrupt has been recognized.
- **Clock**: Is used to synchronize operations.
- **Reset**: Initializes all modules

The operation of the bus is as follows. If one module wishes to send data to another, it must do two things: (1) obtain the use of the bus, and (2) transfer data via the bus. If one module wishes to request data from another module, it must (1) obtain the use of the bus, and (2) transfer a request to the other module over the appropriate control and address lines. It must then wait for that second module to send the data.

3.5.1 Multiple-Bus Hierarchies

If a great number of devices are connected to the bus, performance will suffer. There are two main causes:

1. In general, the more devices attached to the bus, the greater the bus length and hence the greater the propagation delay. This delay determines the time it takes for devices to coordinate the use of the bus. When control of the bus passes from one device to another frequently, these propagation delays can noticeably affect performance.
2. The bus may become a bottleneck as the aggregate data transfer demand approaches the capacity of the bus. This problem can be countered to some extent by increasing the data rate that the bus can carry and by using wider buses (e.g., increasing the data bus from 32 to 64 bits). However, because the data rates generated by attached devices (e.g., graphics and video controllers, network interfaces) are growing rapidly, this is a race that a single bus is ultimately destined to lose.

Accordingly, most computer systems use multiple buses, generally laid out in a hierarchy. A typical traditional structure is shown in Figure 3.13a.There is a local bus that connects the processor to a cache memory and that may support one or more local devices. The cache memory controller connects the cache not only to this local bus, but to a system bus to which are attached all of the main memory modules. The use of a cache structure insulates the processor from a requirement to access main memory frequently. Hence, main memory can be moved off of the local bus onto a

system bus. In this way, I/O transfers to and from the main memory across the system bus do not interfere with the processor's activity.

It is possible to connect I/O controllers directly onto the system bus. A more efficient solution is to make use of one or more expansion buses for this purpose. An expansion bus interface buffers data transfers between the system bus and the I/O controllers on the expansion bus. This arrangement allows the system to support a wide variety of I/O devices and at the same time insulate memory-to-processor traffic from I/O traffic.

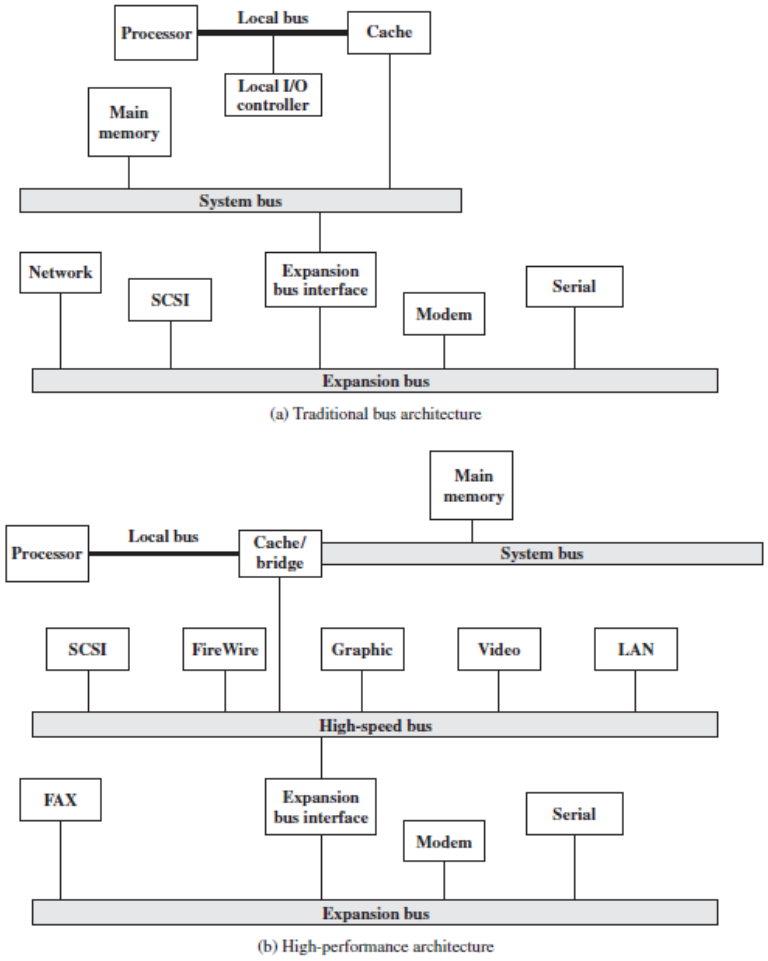


Figure 3.13 Example Bus Configurations

Figure 3.13a shows some typical examples of I/O devices that might be attached to the expansion bus. Network connections include local area networks (LANs) such as a 10-Mbps Ethernet and connections to wide area networks (WANs) such as a packet-switching network. SCSI (small computer system interface) is itself a type of bus used to support local disk drives and other peripherals. A serial port could be used to support a printer or scanner.

This traditional bus architecture is reasonably efficient but begins to break down as higher and higher performance is seen in the I/O devices. In response to these growing demands, a common approach taken by industry is to build a high-speed bus that is closely integrated with the rest of

the system, requiring only a bridge between the processor's bus and the high-speed bus. This arrangement is sometimes known as a mezzanine architecture.

Figure 3.18b shows a typical realization of this approach. Again, there is a local bus that connects the processor to a cache controller, which is in turn connected to a system bus that supports main memory. The cache controller is integrated into a bridge, or buffering device, that connects to the high-speed bus. This bus supports connections to high-speed LANs, such as Fast Ethernet at 100 Mbps, video and graphics workstation controllers, as well as interface controllers to local peripheral buses, including SCSI and FireWire. The latter is a high-speed bus arrangement specifically designed to support high-capacity I/O devices. Lower-speed devices are still supported off an expansion bus, with an interface buffering traffic between the expansion bus and the high-speed bus.

The advantage of this arrangement is that the high-speed bus brings high demand devices into closer integration with the processor and at the same time is independent of the processor. Thus, differences in processor and high-speed bus speeds and signal line definitions are tolerated. Changes in processor architecture do not affect the high-speed bus, and vice versa.

3.6 Elements of Bus Design

Although a variety of different bus implementations exist, there are a few basic parameters or design elements that serve to classify and differentiate buses. Table 3.2 lists key elements.

Table 3.2 Elements of Bus Design

Type	**Bus Width**
Dedicated	Address
Multiplexed	Data
Method of Arbitration	**Data Transfer Type**
Centralized	Read
Distributed	Write
Timing	Read-modify-write
Synchronous	Read-after-write
Asynchronous Block	

3.6.1 Bus Types

Bus lines can be separated into two generic types: **dedicated** and **multiplexed**. A dedicated bus line is permanently assigned either to one function or to a physical subset of computer components.

An example of functional dedication is the use of separate dedicated address and data lines, which is common on many buses. However, it is not essential. For example, address and data information may be transmitted over the same set of lines using an Address Valid control line. At the beginning of a data transfer, the address is placed on the bus and the Address Valid line is activated. At this point, each module has a specified period of time to copy the address and determine if it is the addressed module. The address is then removed from the bus, and the same

bus connections are used for the subsequent read or write data transfer. This method of using the same lines for multiple purposes is known as **time multiplexing**.

The advantage of time multiplexing is the use of fewer lines, which saves space and, usually, cost. The disadvantage is that more complex circuitry is needed within each module. Also, there is a potential reduction in performance because certain events that share the same lines cannot take place in parallel.

Physical dedication refers to the use of multiple buses, each of which connects only a subset of modules. A typical example is the use of an I/O bus to interconnect all I/O modules; this bus is then connected to the main bus through some type of I/O adapter module. The potential advantage of physical dedication is high throughput, because there is less bus contention. A disadvantage is the increased size and cost of the system.

3.6.2 Method Of Arbitration

In all but the simplest systems, more than one module may need control of the bus. For example, an I/O module may need to read or write directly to memory, without sending the data to the processor. Because only one unit at a time can successfully transmit over the bus, some method of arbitration is needed. The various methods can be roughly classified as being either centralized or distributed. In a centralized scheme, a single hardware device, referred to as a **bus controller** or **arbiter**, is responsible for allocating time on the bus. The device may be a separate module or part of the processor. In a distributed scheme, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus. With both methods of arbitration, the purpose is to designate one device, either the processor or an I/O module, as master. The master may then initiate a data transfer (e.g., read or write) with some other device, which acts as slave for this particular exchange.

3.6.3 Timing

Timing refers to the way in which events are coordinated on the bus. Buses use either synchronous timing or asynchronous timing.

With synchronous timing, the occurrence of events on the bus is determined by a clock. The bus includes a clock line upon which a clock transmits a regular sequence of alternating 1s and 0s of equal duration. A single 1–0 transmission is referred to as a clock cycle or bus cycle and defines a time slot. All other devices on the bus can read the clock line, and all events start at the beginning of a clock cycle. Figure 3.14 shows a typical, but simplified, timing diagram for synchronous read and write operations. Other bus signals may change at the leading edge of the clock signal (with a slight reaction delay). Most events occupy a single clock cycle. In this simple example, the processor places a memory address on the address lines during the first clock cycle and may assert various status lines. Once the address lines have stabilized, the processor issues an address enable signal. For a read operation, the processor issues a read command at the start of the second cycle. A memory module recognizes the address and, after a delay of one cycle, places the data on the data lines. The processor reads the data from the data lines and drops the read signal. For a write operation, the processor puts the data on the data lines at the start of the second cycle, and issues

a write command after the data lines have stabilized. The memory module copies the information from the data lines during the third clock cycle.

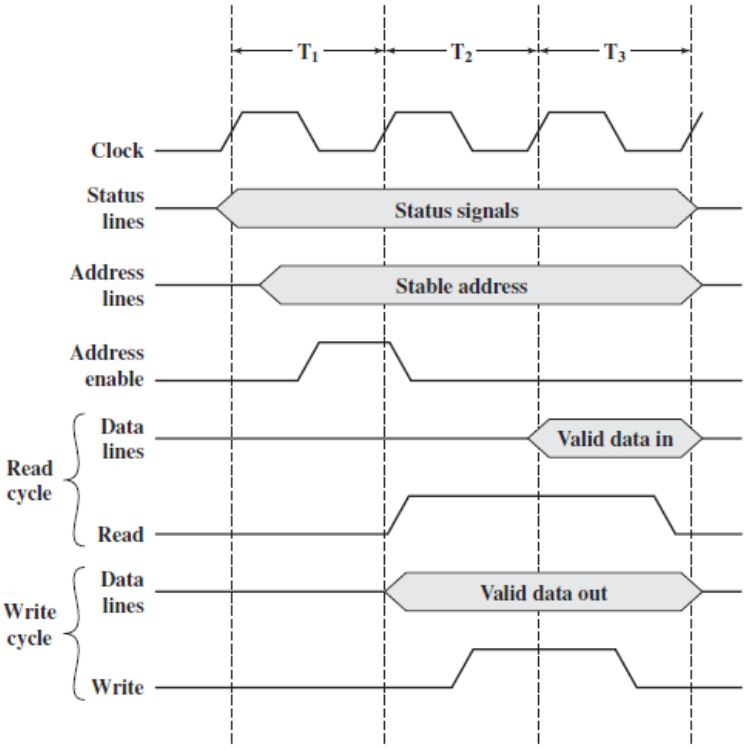


Figure 3.14 Timing of Synchronous Bus Operations

With asynchronous timing, the occurrence of one event on a bus follows and depends on the occurrence of a previous event. In the simple read example of Figure 3.15a, the processor places address and status signals on the bus. After pausing for these signals to stabilize, it issues a read command, indicating the presence of valid address and control signals. The appropriate memory decodes the address and responds by placing the data on the data line. Once the data lines have stabilized, the memory module asserts the acknowledged line to signal the processor that the data are available. Once the master has read the data from the data lines, it de-asserts the read signal. This causes the memory module to drop the data and acknowledge lines. Finally, once the acknowledge line is dropped, the master removes the address information.

Figure 3.15b shows a simple asynchronous write operation. In this case, the master places the data on the data line at the same time that is puts signals on the status and address lines. The memory module responds to the write command by copying the data from the data lines and then asserting the acknowledge line. The master then drops the write signal and the memory module drops the acknowledge signal.

Synchronous timing is simpler to implement and test. However, it is less flexible than asynchronous timing. Because all devices on a synchronous bus are tied to a fixed clock rate, the system cannot take advantage of advances in device performance. With asynchronous timing, a mixture of slow and fast devices, using older and newer technology, can share a bus.

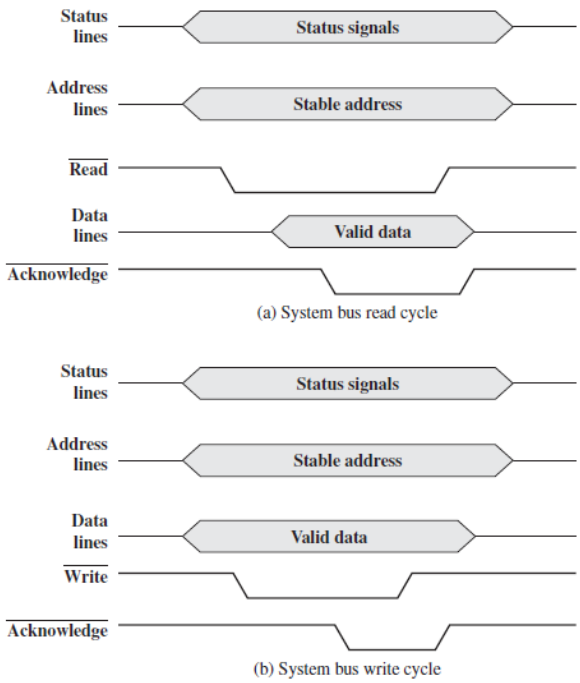


Figure 3.15 Timing of Asynchronous Bus Operations

3.6.4 Bus Width

We have already addressed the concept of bus width. The width of the data bus has an impact on system performance: The wider the data bus, the greater the number of bits transferred at one time. The width of the address bus has an impact on system capacity: the wider the address bus, the greater the range of locations that can be referenced.

3.6.5 Data Transfer Type

Finally, a bus supports various data transfer types, as illustrated in Figure 3.16. All buses support both write (master to slave) and read (slave to master) transfers. In the case of a multiplexed address/data bus, the bus is first used for specifying the address and then for transferring the data. For a read operation, there is typically a wait while the data are being fetched from the slave to be put on the bus. For either a read or a write, there may also be a delay if it is necessary to go through arbitration to gain control of the bus for the remainder of the operation (i.e., seize the bus to request a read or write, then seize the bus again to perform a read or write).

In the case of dedicated address and data buses, the address is put on the address bus and remains there while the data are put on the data bus. For a write operation, the master puts the data onto the data bus as soon as the address has stabilized and the slave has had the opportunity to recognize its address. For a read operation, the slave puts the data onto the data bus as soon as it has recognized its address and has fetched the data.

There are also several combination operations that some buses allow. A read–modify–write operation is simply a read followed immediately by a write to the same address. The address is only broadcast once at the beginning of the operation. The whole operation is typically indivisible to

prevent any access to the data element by other potential bus masters. The principal purpose of this capability is to protect shared memory resources in a multiprogramming system.

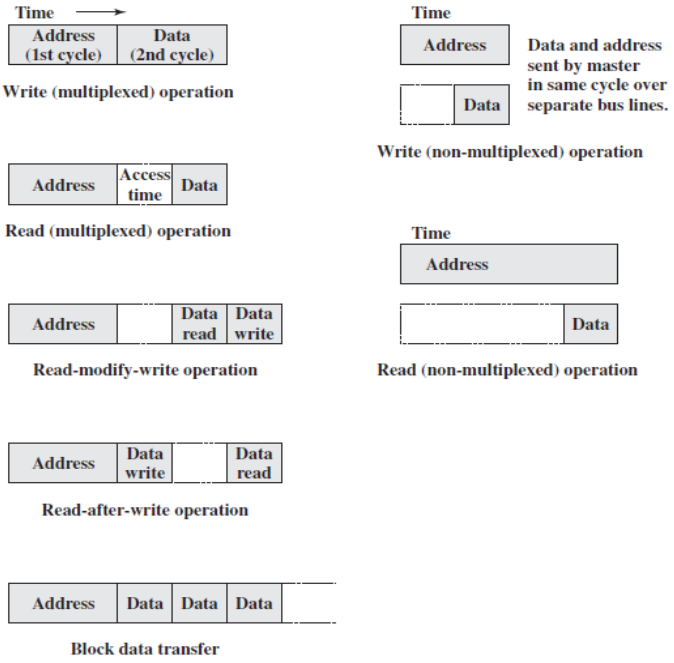


Figure 3.16 Bus Data Transfer Types

Read-after-write is an indivisible operation consisting of a write followed immediately by a read from the same address. The read operation may be performed for checking purposes.

Some bus systems also support a block data transfer. In this case, one address cycle is followed by n data cycles. The first data item is transferred to or from the specified address; the remaining data items are transferred to or from subsequent addresses.

Chapter 4
Cache Memory

4.1 Computer Memory System Overview

4.1.1 Characteristics of Memory Systems

The complex subject of computer memory is made more manageable if we classify memory systems according to their key characteristics. The most important of these are listed in Table 4.1.

Table 4.1 Key Characteristics of Computer Memory Systems

Location	**Performance**
Internal (e.g. processor registers, main memory, cache)	Access time
External (e.g. optical disks, magnetic disks, tapes)	Cycle time
	Transfer rate
Capacity	**Physical Type**
Number of words	Semiconductor
Number of bytes	Magnetic
Unit of Transfer	Optical
Word	Magneto-optical
Block	
Access Method	**Physical Characteristics**
Sequential	Volatile/nonvolatile
Direct	Erasable/nonerasable
Random	**Organization**
Associative	Memory modules

The term **location** in Table 4.1 refers to whether memory is internal and external to the computer. Internal memory is often equated with main memory. But there are other forms of internal memory. The processor requires its own local memory, in the form of registers. Further, as we shall see, the control unit portion of the processor may also require its own internal memory. Cache is another form of internal memory. External memory consists of peripheral storage devices, such as disk and tape, that are accessible to the processor via I/O controllers.

An obvious characteristic of memory is its **capacity**. For internal memory, this is typically expressed in terms of bytes (1 byte = 8 bits) or words. Common word lengths are 8, 16, and 32 bits. External memory capacity is typically expressed in terms of bytes.

A related concept is the **unit of transfer**. For internal memory, the unit of transfer is equal to the number of electrical lines into and out of the memory module. This may be equal to the word length, but is often larger, such as 64, 128, or 256 bits. To clarify this point, consider three related concepts for internal memory:

- **Word**: The “natural” unit of organization of memory. The size of the word is typically equal to the number of bits used to represent an integer and to the instruction length. Unfortunately, there are many exceptions. For example, the CRAY C90 (an older model CRAY supercomputer) has a 64-bit word length but uses a 46-bit integer representation.

The Intel x86 architecture has a wide variety of instruction lengths, expressed as multiples of bytes, and a word size of 32 bits.

- **Addressable units**: In some systems, the addressable unit is the word. However, many systems allow addressing at the byte level. In any case, the relationship between the length in bits A of an address and the number N of addressable units is $2^A = N$.
- **Unit of transfer**: For main memory, this is the number of bits read out of or written into memory at a time. The unit of transfer need not equal a word or an larger units than a word, and these are referred to as blocks.

Another distinction among memory types is the **method of accessing** units of data. These include the following:

- **Sequential access**: Memory is organized into units of data, called records. Access must be made in a specific linear sequence. Stored addressing information is used to separate records and assist in the retrieval process. A shared read–write mechanism is used, and this must be moved from its current location to the desired location, passing and rejecting each intermediate record. Thus, the time to access an arbitrary record is highly variable. Tape units are sequential access.
- **Direct access**: As with sequential access, direct access involves a shared read–write mechanism. However, individual blocks or records have a unique address based on physical location. Access is accomplished by direct access to reach a general vicinity plus sequential searching, counting, or waiting to reach the final location. Again, access time is variable. Disk units are direct access.
- **Random access**: Each addressable location in memory has a unique, physically wired-in addressing mechanism. The time to access a given location is independent of the sequence of prior accesses and is constant. Thus, any location can be selected at random and directly addressed and accessed. Main memory and some cache systems are random access.
- **Associative**: This is a random access type of memory that enables one to make a comparison of desired bit locations within a word for a specified match, and to do this for all words simultaneously. Thus, a word is retrieved based on a portion of its contents rather than its address. As with ordinary random-access memory, each location has its own addressing mechanism, and retrieval time is constant independent of location or prior access patterns. Cache memories may employ associative access.

From a user's point of view, the two most important characteristics of memory are capacity and **performance**. Three performance parameters are used:

- **Access time (latency)**: For random-access memory, this is the time it takes to perform a read or write operation, that is, the time from the instant that an address is presented to the memory to the instant that data have been stored or made available for use. For non-random-access memory, access time is the time it takes to position the read–write mechanism at the desired location.
- **Memory cycle time**: This concept is primarily applied to random-access memory and consists of the access time plus any additional time required before a second access can

commence. This additional time may be required for transients to die out on signal lines or to regenerate data if they are read destructively. Note that memory cycle time is concerned with the system bus, not the processor.

- **Transfer rate**: This is the rate at which data can be transferred into or out of a memory unit. For random-access memory, it is equal to 1/(cycle time).

For non-random-access memory, the following relationship holds:

$$T_N = T_A + \frac{n}{R}$$

where

T_N Average time to read or write N bits
T_A Average access time
n Number of bits
R Transfer rate, in bits per second (bps)

A variety of physical types of memory have been employed. The most common today are semiconductor memory, magnetic surface memory, used for disk and tape, and optical and magneto-optical.

Several physical characteristics of data storage are important. In a volatile memory, information decays naturally or is lost when electrical power is switched off. In a nonvolatile memory, information once recorded remains without deterioration until deliberately changed; no electrical power is needed to retain information. Magnetic-surface memories are nonvolatile. Semiconductor memory may be either volatile or nonvolatile. Nonerasable memory cannot be altered, except by destroying the storage unit. Semiconductor memory of this type is known as read-only memory (ROM). Of necessity, a practical nonerasable memory must also be nonvolatile.

4.1.2 The Memory Hierarchy

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open ended. If the capacity is there, applications will likely be developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access time

The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with short access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A typical hierarchy is illustrated in Figure 4.1. As one goes down the hierarchy, the following occur:

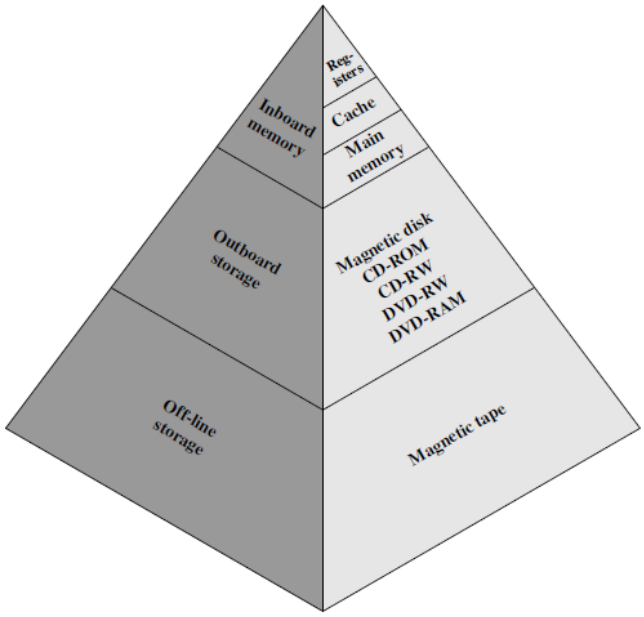


Figure 4.1 The Memory Hierarchy

a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access of the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. The key to the success of this organization is item (d): decreasing frequency of access.

Example 4.1 Suppose that the processor has access to two levels of memory. Level 1 contains 1000 words and has an access time of $0.01\ \mu s$; level 2 contains 100,000 words and has an access time of $0.1\ \mu s$. Assume that if a word to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the word is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the word is in level 1 or level 2. Figure 4.2 shows the general shape of the curve that covers this situation. The figure shows the average access time to a two-level memory as a function of the hit ratio H, where H is defined as the fraction of all memory accesses that are found in the faster memory (e.g., the cache), T_1 is the access time to level 1, and T_2 is the access time to level 2. As can

be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2.

In our example, suppose 95% of the memory accesses are found in the cache. Then the average time to access a word can be expressed as

$$(0.95)(0.01\,\mu s) + (0.05)(0.01\,\mu s + 0.1\,\mu s) = 0.0095 + 0.0055 = 0.015\,\mu s$$

The average access time is much closer to $0.01\,\mu s$ than to $0.1\,\mu s$, as desired.

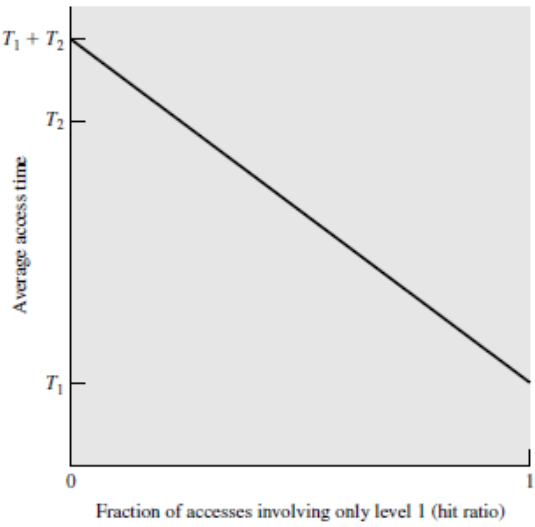


Figure 4.2 Performance of accesses involving only

The use of two levels of memory to reduce average access time works in principle, but only if conditions (a) through (d) apply. By employing a variety of technologies, a spectrum of memory systems exists that satisfies conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as **locality of reference**. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider the two-level example already presented. Let level 2 memory contain all program instructions and data. The current clusters can be temporarily placed in level 1. From time to time, one of the clusters in level 1 will have to be swapped back to level 2 to make room for a new cluster coming in to level 1. On average, however, most references will be to instructions and data contained in level 1.

This principle can be applied across more than two levels of memory, as suggested by the hierarchy shown in Figure 4.1. The fastest, smallest, and most expensive

type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Skipping down two levels, main memory is the principal internal memory system of the computer. Each location in main memory has a unique address. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable magnetic disk, tape, and optical storage. External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words. Disk is also used to provide an extension to main memory known as virtual memory.

Other forms of memory may be included in the hierarchy. For example, large IBM mainframes include a form of internal memory known as expanded storage. This uses a semiconductor technology that is slower and less expensive than that of main memory. Strictly speaking, this memory does not fit into the hierarchy but is a side branch: Data can be moved between main memory and expanded storage but not between expanded storage and external memory. Other forms of secondary memory include optical and magneto-optical disks. Finally, additional levels can be effectively added to the hierarchy in software. A portion of main memory can be used as a buffer to hold data temporarily that is to be read out to disk. Such a technique, sometimes referred to as a disk cache, improves performance in two ways:

- Disk writes are clustered. Instead of many small transfers of data, we have a few large transfers of data. This improves disk performance and minimizes processor involvement.
- Some data destined for write-out may be referenced by a program before the next dump to disk. In that case, the data are retrieved rapidly from the software cache rather than slowly from the disk.

4.2 CACHE MEMORY PRINCIPLES

Cache memory is intended to give memory speed approaching that of the fastest memories available, and at the same time provide a large memory size at the price of less expensive types of semiconductor memories. The concept is illustrated in Figure 4.3a. There is a relatively large and slow main memory together with a smaller, faster cache memory. The cache contains a copy of portions of main memory. When the processor attempts to read a word of memory, a check is made to determine if the word is in the cache. If so, the word is delivered to the processor. If not, a block of main memory, consisting of some fixed number of words, is read into the cache and then

the word is delivered to the processor. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that there will be future references to that same memory location or to other words in the block.

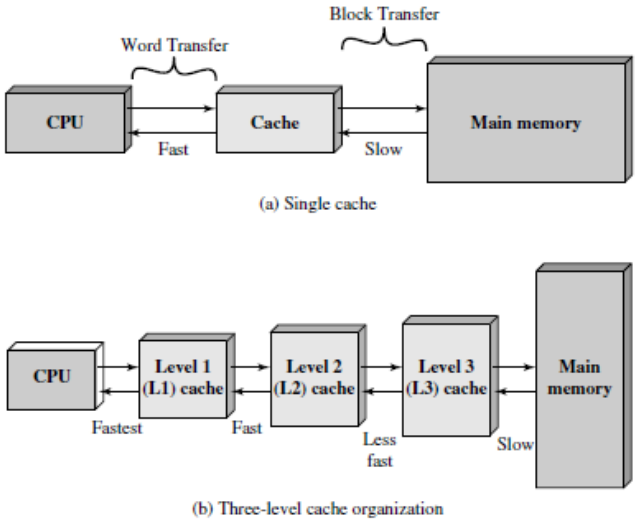


Figure 4.3 Cache and Main Memory

Figure 4.3b depicts the use of multiple levels of cache. The L2 cache is slower and typically larger than the L1 cache, and the L3 cache is slower and typically larger than the L2 cache.

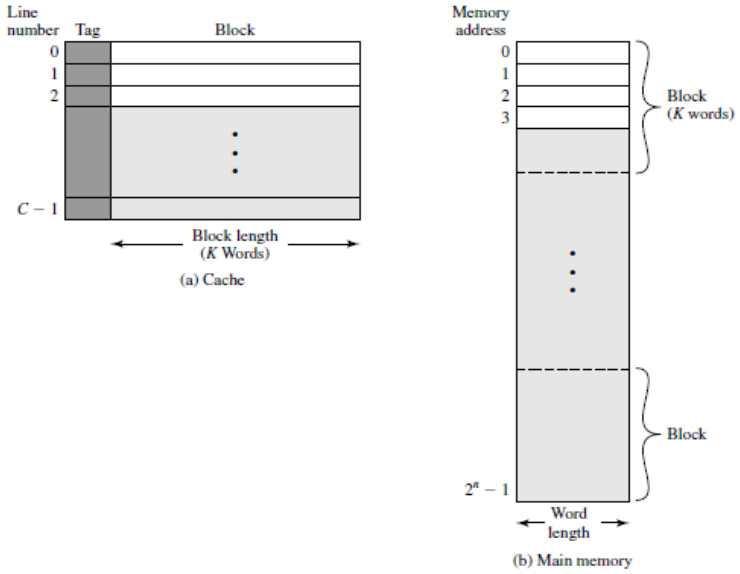


Figure 4.4 Cache/Main Memory Structure

Figure 4.4 depicts the structure of a cache/main-memory system. Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address. For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each. That is, there are $M = \frac{2^n}{K}$ blocks in main memory. The cache consists of m blocks, called **lines**. Each line contains

K words, plus a tag of a few bits. Each line also includes control bits (not shown), such as a bit to indicate whether the line has been modified since being loaded into the cache. The length of a line, not including tag and control bits, is the **line size**. The line size may be as small as 32 bits, with each "word" being a single byte; in this case the line size is 4 bytes.

The number of lines is considerably less than the number of main memory blocks ($m << M$). At any time, some subset of the blocks of memory resides in lines in the cache. If a word in a block of memory is read, that block is transferred to one of the lines of the cache. Because there are more blocks than lines, an individual line cannot be uniquely and permanently dedicated to a particular block. Thus, each line includes a **tag** that identifies which particular block is currently being stored. The tag is usually a portion of the main memory address, as described later in this section.

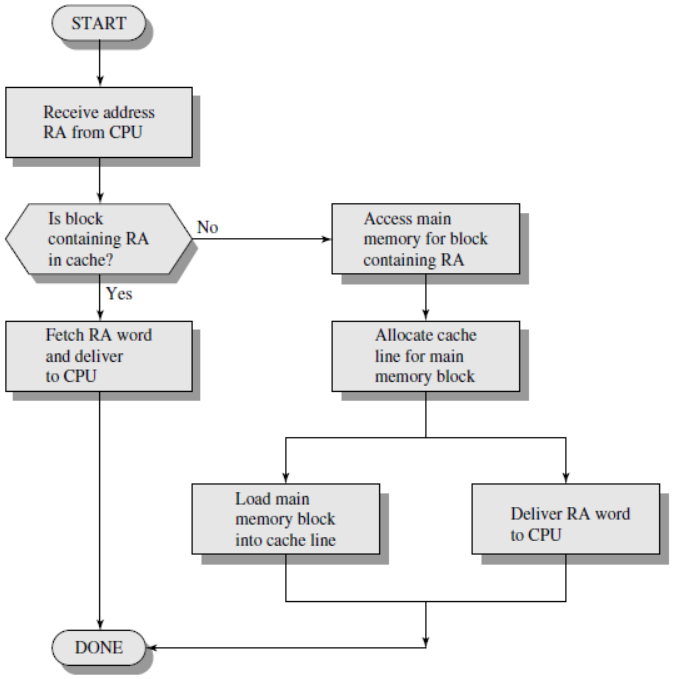


Figure 4.5 Cache Read Operation

Figure 4.5 illustrates the read operation. The processor generates the read address (RA) of a word to be read. If the word is contained in the cache, it is delivered the processor. Otherwise, the block containing that word is loaded into the cache, and the word is delivered to the processor. Figure 4.5 shows these last two operations occurring in parallel and reflects the organization shown in Figure 4.6, which is typical of contemporary cache organizations. In this organization, the cache connects to the processor via data, control, and address lines. The data and address lines also attach to data and address buffers, which attach to a system bus from which main memory is reached. When a cache hit occurs, the data and address buffers are disabled and communication is only between processor and cache, with no system bus traffic. When a cache miss occurs, the desired address is loaded onto the system bus and the data are returned through the data buffer to both the cache and the processor. In other organizations, the cache is physically interposed

between the processor and the main memory for all data, address, and control lines. In this latter case, for a cache miss, the desired word is first read into the cache and then transferred from cache to processor.

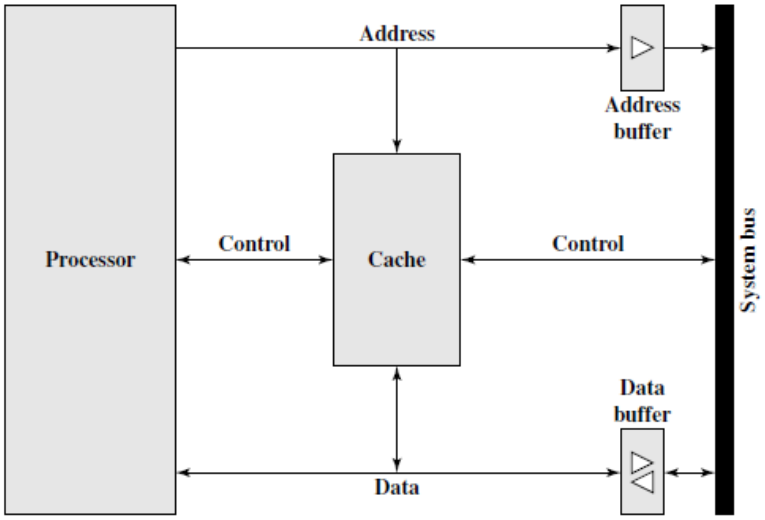


Figure 4.6 Typical Cache Organization

4.3 ELEMENTS OF CACHE DESIGN

This section provides an overview of cache design parameters and reports some typical results. We occasionally refer to the use of caches in high-performance computing (HPC). HPC deals with supercomputers and supercomputer software, especially for scientific applications that involve large amounts of data, vector and matrix computation, and the use of parallel algorithms. Cache design for HPC is quite different than for other hardware platforms and applications. Indeed, many researchers have found that HPC applications perform poorly on computer architectures that employ caches. Other researchers have since shown that a cache hierarchy can be useful in improving performance if the application software is tuned to exploit the cache.

Table 4.2 Elements of Cache Design

Cache Addresses Logical Physical	**Write Policy** Write through Write back Write once
Cache Size	**Line Size**
Mapping Function Direct Associative Set Associative	**Number of caches** Single or two level Unified or split
Replacement Algorithm Least recently used (LRU) First in first out (FIFO) Least frequently used (LFU) Random	

Although there are a large number of cache implementations, there are a few basic design elements that serve to classify and differentiate cache architectures. Table 4.2 lists key elements.

4.3.1 Cache Addresses

Almost all non- embedded processors, and many embedded processors, support virtual memory. In essence, virtual memory is a facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available. When virtual memory is used, the address fields of machine instructions contain virtual addresses. For reads to and writes from main memory, a hardware memory management unit (MMU) translates each virtual address into a physical address in main memory.

When virtual addresses are used, the system designer may choose to place the cache between the processor and the MMU or between the MMU and main memory (Figure 4.7). A logical cache, also known as a virtual cache, stores data using virtual addresses. The processor accesses the cache directly, without going through the MMU. A physical cache stores data using main memory physical addresses.

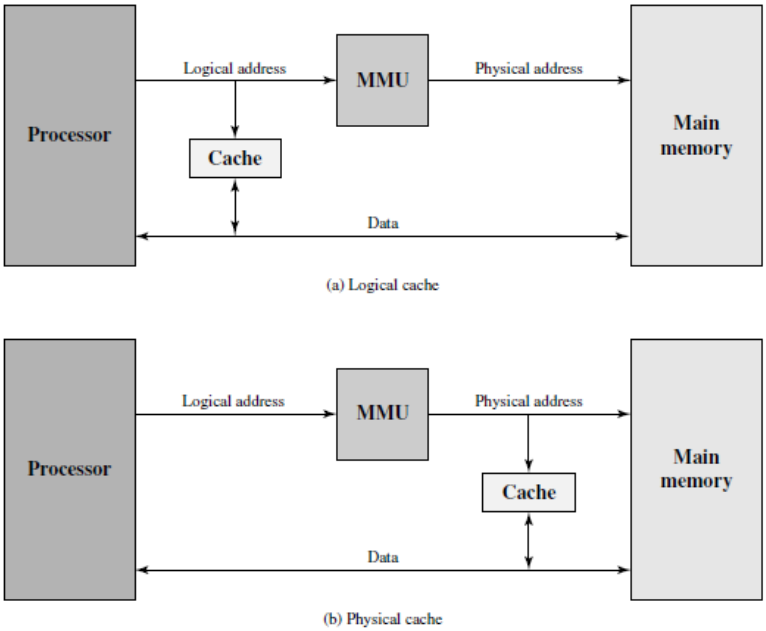


Figure 4.7 Logical and Physical Caches

One obvious advantage of the logical cache is that cache access speed is faster than for a physical cache, because the cache can respond before the MMU performs an address translation. The disadvantage has to do with the fact that most virtual memory systems supply each application with the same virtual memory address space. That is, each application sees a virtual memory that starts at address 0. Thus, the same virtual address in two different applications refers to two different physical addresses. The cache memory must therefore be completely flushed with each application context switch, or extra bits must be added to each line of the cache to identify which virtual address space this address refers to.

4.3.2 Cache Size

The first item in Table 4.2, cache size, has already been discussed. We would like the size of the cache to be small enough so that the overall average cost per bit is close to that of main memory alone and large enough so that the overall average access time is close to that of the cache alone. There are several other motivations for minimizing cache size. The larger the cache, the larger the number of gates involved in addressing the cache. The result is that large caches tend to be slightly slower than small ones—even when built with the same integrated circuit technology and put in the same place on chip and circuit board. The available chip and board area also limits cache size. Because the performance of the cache is very sensitive to the nature of the workload, it is impossible to arrive at a single "optimum" cache size.

4.3.3 Mapping Function

Because there are fewer cache lines than main memory blocks, an algorithm is needed for mapping main memory blocks into cache lines. Further, a means is needed for determining which main memory block currently occupies a cache line. The choice of the mapping function dictates how the cache is organized. Three techniques can be used: direct, associative, and set associative. We examine each of these in turn. In each case, we look at the general structure and then a specific example.

Example 4.2 For all three cases, the example includes the following elements:

- The cache can hold 64 KBytes.
- Data are transferred between main memory and the cache in blocks of 4 bytes each. This means that the cache is organized as 16K 2_{14} lines of 4 bytes each.
- The main memory consists of 16 Mbytes, with each byte directly addressable by a 24-bit address (2_{24} 16M).Thus, for mapping purposes, we can consider main memory to consist of 4M blocks of 4 bytes each.

4.3.3.1 DIRECT MAPPING

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. The mapping is expressed as

$$i = j \text{ modulo } m$$

Where

$i =$ cache line number

$j =$ main memory block number

$m =$ number of lines in the cache

Figure 4.8a shows the mapping for the first m blocks of main memory. Each block of main memory maps into one unique line of the cache. The next m blocks of main memory map into the

cache in the same fashion; that is, block B_m of main memory maps into line L_0 of cache, block B_{m+1} maps into line L_1, and so on.

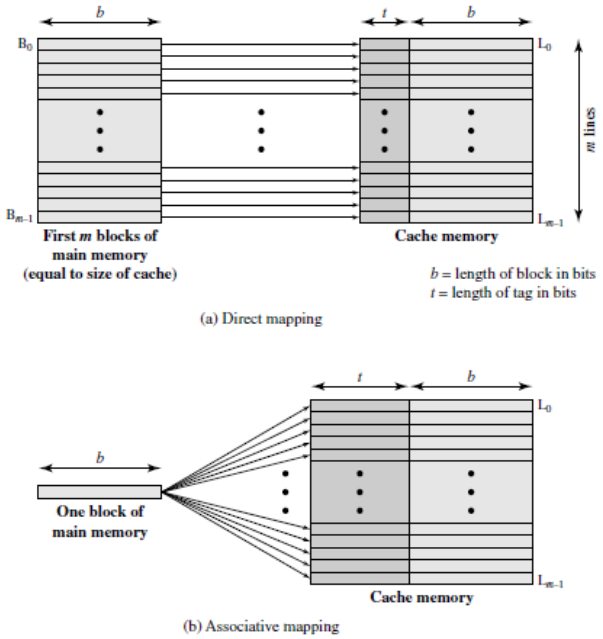


Figure 4.8 Mapping from Main Memory to Cache: Direct and Associative

The mapping function is easily implemented using the main memory address. Figure 4.9 illustrates the general mechanism. For purposes of cache access, each main memory address can be viewed as consisting of three fields. The least significant w bits identify a unique word or byte within a block of main memory; in most contemporary machines, the address is at the byte level. The remaining s bits specify one of the 2^s blocks of main memory. The cache logic interprets these s bits as a tag of $s - r$ bits (most significant portion) and a line field of r bits. This latter field identifies one of the $m = 2^r$ lines of the cache. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^w$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in cache $= m = 2^r$
- Size of cache $= 2^{r+w}$ words or bytes
- Size of tag $= (s - r)$ bits

The effect of this mapping is that blocks of main memory are assigned to lines of the cache as follows:

Cache line	Main memory blocks assigned
0	$0, m, 2m, \dots, 2^s - m$
1	$1, m+1, 2m+1, \dots, 2^s - m + 1$
⋮	⋮
$m-1$	$m-1, 2m-1, 3m-1, \dots, 2^s - 1$

Thus, the use of a portion of the address as a line number provides a unique mapping of each block of main memory into the cache. When a block is actually read into its assigned line, it is necessary to tag the data to distinguish it from other blocks that can fit into that line. The most significant $s - r$ bits serve this purpose.

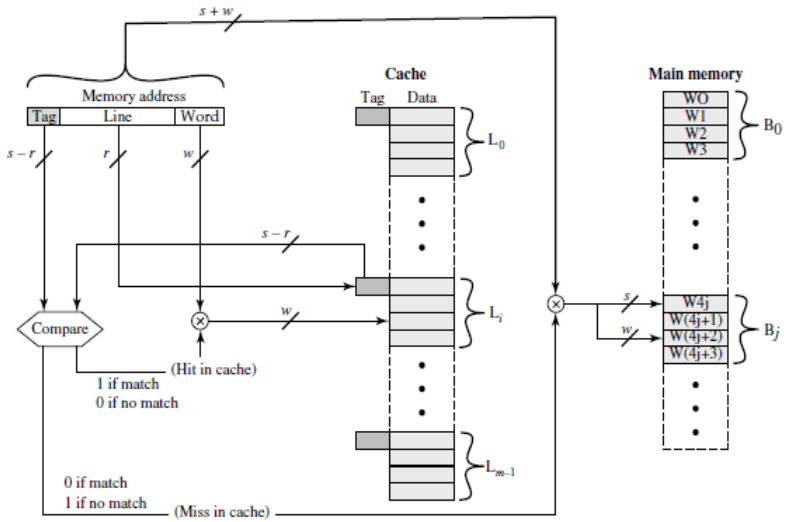


Figure 4.9 Direct-Mapping Cache Organization

Example 4.2a: Figure 4.10 shows our example system using direct mapping. In the example, $m = 16K = 2^{14}$ and $i = j \; modulo \; 2^{14}$. The mapping becomes

Cache Line	Starting Memory Address of Block
0	$000000, 010000, \dots FF0000$
1	$000004, 010004, \dots FF0004$
⋮	⋮
$2^{14} - 1$	$00FFFC, 01FFFC, \dots, FFFFFC$

Note that no two blocks that map into the same line number have the same tag number. Thus, blocks with starting addresses 000000, 010000, FF0000 have tag numbers 00, 01, FF, respectively.

Referring back to Figure 4.5, a read operation works as follows. The cache system is presented with a 24-bit address. The 14-bit line number is used as an index into the cache to access a particular line. If the 8-bit tag number matches the tag number currently stored in that line, then the 2-bit word number is used to select one of the 4 bytes in that line. Otherwise, the 22-bit tag-plus-line field is used to fetch a block from main memory. The actual address that is used for the fetch is the 22-bit tag-plus-line concatenated with two 0 bits, so that 4 bytes are fetched starting on a block boundary.

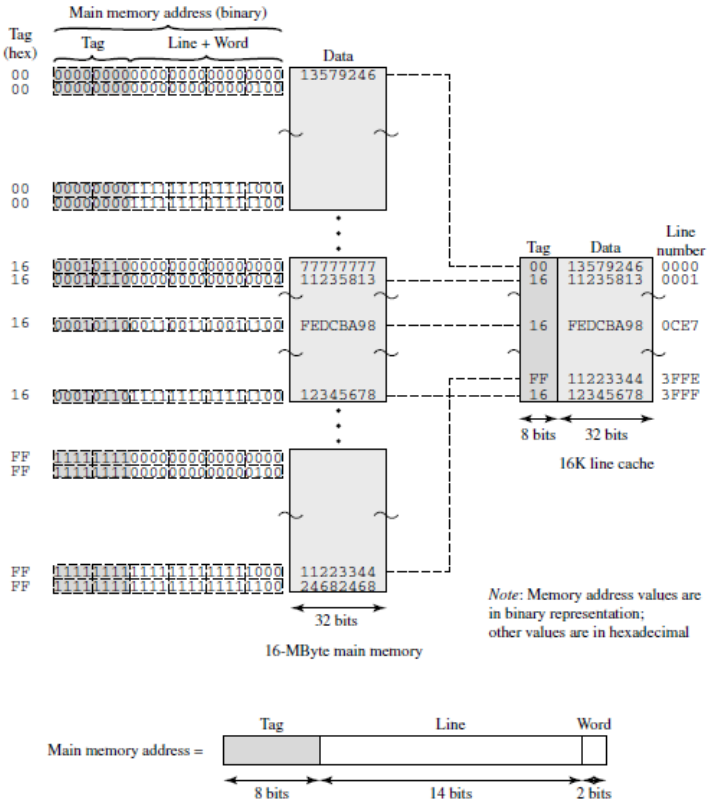


Figure 4.10 Direct Mapping Example

The direct mapping technique is simple and inexpensive to implement. Its main disadvantage is that there is a fixed cache location for any given block. Thus, if a program happens to reference words repeatedly from two different blocks that map into the same line, then the blocks will be continually swapped in the cache, and the hit ratio will be low (a phenomenon known as **thrashing**).

One approach to lower the miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at a small cost. Such recycling is possible using a victim cache. Victim cache was originally proposed as an approach to reduce the conflict misses of direct mapped caches without affecting its fast access time. Victim cache is a fully associative cache, whose size is typically 4 to 16 cache lines, residing between a direct mapped L1 cache and the next level of memory.

4.3.3.2 ASSOCIATIVE MAPPING

Associative mapping overcomes the disadvantage of direct mapping by permitting each main memory block to be loaded into any line of the cache (Figure 4.8b). In this case, the cache control logic interprets a memory address simply as a Tag and a Word field. The Tag field uniquely identifies a block of main memory. To determine whether a block is in the cache, the cache control logic must simultaneously examine every line's tag for a match. Figure 4.11 illustrates the logic. Note that no field in the address corresponds to the line number, so that the number of lines in the cache is not determined by the address format. To summarize,

- Address length $= (s + w)$ bits
- Number of addressable units $= 2^{s+w}$ words or bytes
- Block size $=$ line size $= 2^{w}$ words or bytes
- Number of blocks in main memory $= \frac{2^{s+w}}{2^{w}} = 2^{s}$
- Number of lines in cache $=$ undetermined
- Size of tag $= s$ bits

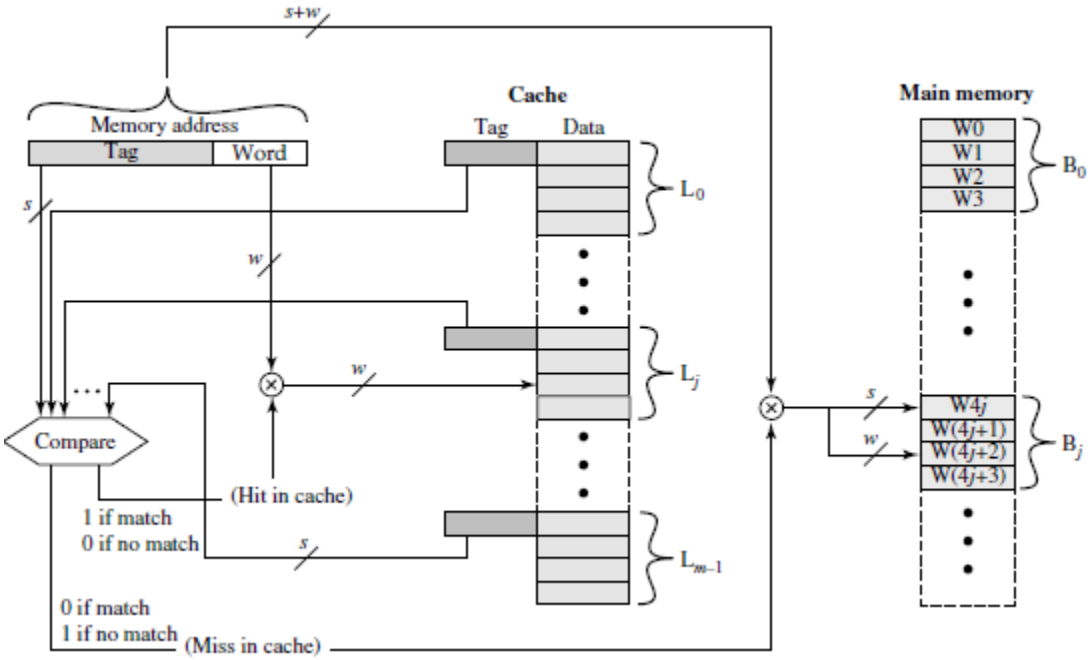


Figure 4.11 Fully Associative Cache Organization

Example 4.2b: Figure 4.12 shows our example using associative mapping. A main memory address consists of a 22-bit tag and a 2-bit byte number. The 22-bit tag must be stored with the 32-bit block of data for each line in the cache. Note that it is the leftmost (most significant) 22 bits of the address that form the tag. Thus, the 24-bit hexadecimal address 16339C has the 22-bit tag 058CE7.This is easily seen in binary notation:

memory address	0001	0110	0011	0011	1001	1100	(binary)
	1	6	3	3	9	C	(hex)
tag (leftmost 22 bits)	00	0101	1000	1100	1110	0111	(binary)
	0	5	8	C	E	7	(hex)

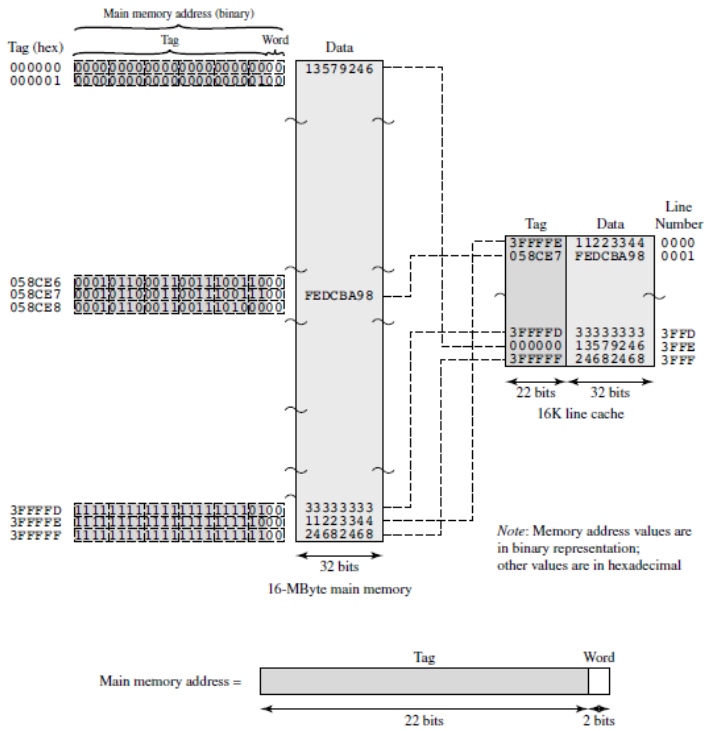


Figure 4.12 Associative Mapping Example

With associative mapping, there is flexibility as to which block to replace when a new block is read into the cache. Replacement algorithms, discussed later in this section, are designed to maximize the hit ratio. The principal disadvantage of associative mapping is the complex circuitry required to examine the tags of all cache lines in parallel

4.3.3.3 SET-ASSOCIATIVE MAPPING

Set-associative mapping is a compromise that exhibits the strengths of both the direct and associative approaches while reducing their disadvantages.

In this case, the cache consists of a number sets, each of which consists of a number of lines. The relationships are

$$m = v \times k$$

$$i = j \text{ modulo } v$$

where

$i =$ cache set number

$j =$ main memory block number

$m =$ number of lines in the cache

v = number of sets

k = number of lines in each set

This is referred to as k-way set-associative mapping. With set-associative mapping, block B_j can be mapped into any of the lines of set j. Figure 4.13a illustrates this mapping for the first blocks of main memory. As with associative mapping, each word maps into multiple cache lines. For set-associative mapping, each word maps into all the cache lines in a specific set, so that main memory block B_0 maps into set 0, and so on. Thus, the set-associative cache can be physically implemented as v associative caches. It is also possible to implement the set-associative cache as k direct mapping caches, as shown in Figure 4.13b. Each direct-mapped cache is referred to as a way, consisting of v lines. The first v lines of main memory are direct mapped into the lines of each way; the next group of v lines of main memory are similarly mapped, and so on. The direct-mapped implementation is typically used for small degrees of associativity (small values of k) while the associative- mapped implementation is typically used for higher degrees of associativity.

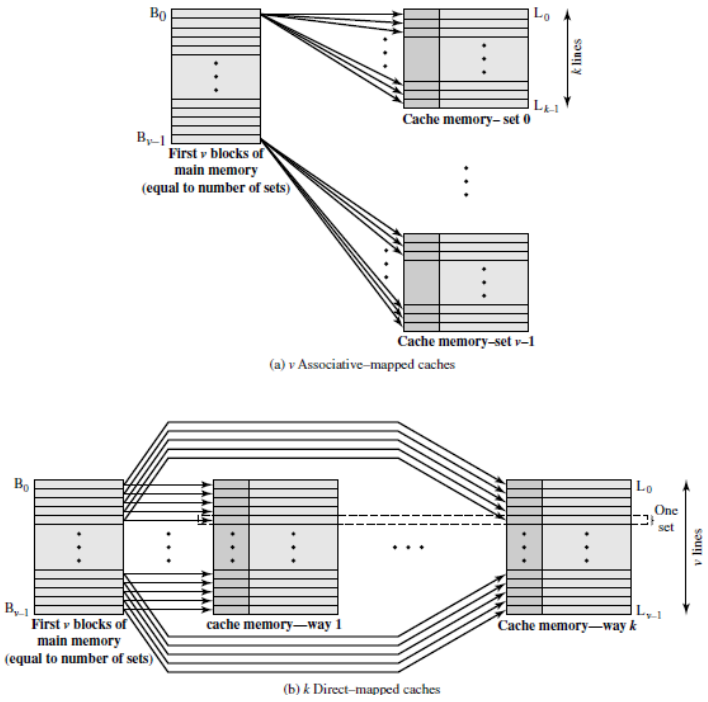


Figure 4.13 Mapping from Main Memory to Cache: k-way Set Associative

For set-associative mapping, the cache control logic interprets a memory address as three fields: Tag, Set, and Word. The d set bits specify one of $v = 2^d$ sets. The s bits of the Tag and Set fields specify one of the 2^s blocks of main memory. Figure 4.14 illustrates the cache control logic. With fully associative mapping, the tag in a memory address is quite large and must be compared to the tag of every line in the cache. With k-way set-associative mapping, the tag in a memory address is much smaller and is only compared to the k tags within a single set. To summarize,

- Address length = $(s + w)$ bits
- Number of addressable units = 2^{s+w} words or bytes
- Block size = line size = 2^w words or bytes
- Number of blocks in main memory = $\frac{2^{s+w}}{2^w} = 2^s$
- Number of lines in set = k
- Number of sets = $v = 2^d$
- Number of lines in cache = $m = kv = k \times 2^d$
- Size of cache = $k \times 2^{d+w}$ words or bytes
- Size of tag = $(s - d)$ bits

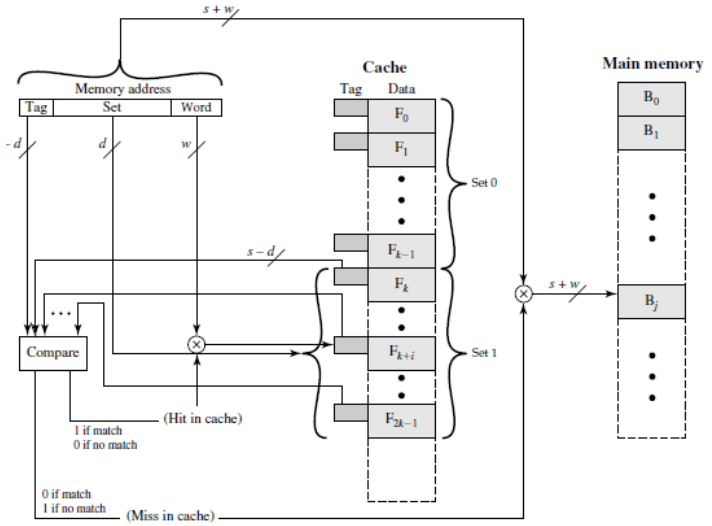


Figure 4.14 K-Way Set Associative Cache Organization

Example 4.2c Figure 4.15 shows our example using set-associative mapping with two lines in each set, referred to as two-way set-associative. The 13-bit set number identifies a unique set of two lines within the cache. It also gives the number of the block in main memory, modulo 2^{13}. This determines the mapping of blocks into lines. Thus, blocks 000000, 008000, FF8000 of main memory map into cache set 0. Any of those blocks can be loaded into either of the two lines in the set. Note that no two blocks that map into the same cache set have the same tag number. For a read operation, the 13-bit set number is used to determine which set of two lines is to be examined. Both lines in the set are examined for a match with the tag number of the address to be accessed.

In the extreme case of $m = v$, $k = 1$, the set-associative technique reduces to direct mapping, and for $v = 1$, $k = m$, it reduces to associative mapping. The use of two lines per set ($v = m/2$, $k = 2$) is the most common set-associative organization. It significantly improves the hit ratio over direct mapping. Four-way set associative ($v = m/4$, $k = 4$) makes a modest additional improvement for a relatively small additional cost. Further increases in the number of lines per set have little effect.

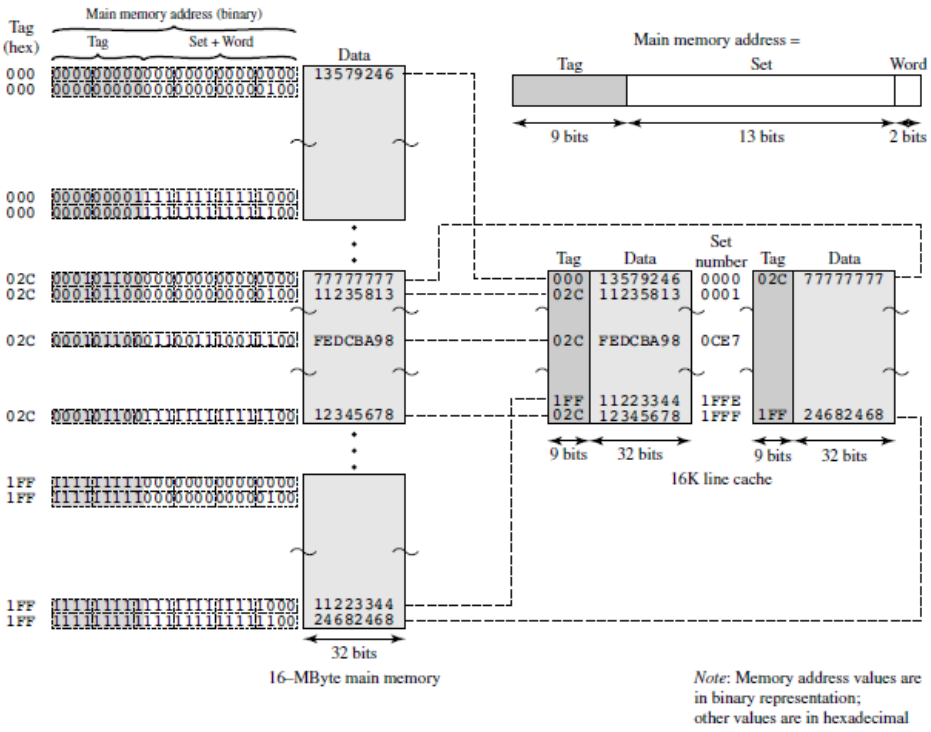


Figure 4.15 Two-Way Set Associative Mapping Example

4.3.4 Replacement Algorithms

Once the cache has been filled, when a new block is brought into the cache, one of the existing blocks must be replaced. For direct mapping, there is only one possible line for any particular block, and no choice is possible. For the associative and set-associative techniques, a replacement algorithm is needed. To achieve high speed, such an algorithm must be implemented in hardware. A number of algorithms have been tried. We mention four of the most common. Probably the most effective is least recently used (LRU): Replace that block in the set that has been in the cache longest with no reference to it. For two-way set associative, this is easily implemented. Each line includes a USE bit. When a line is referenced, its USE bit is set to 1 and the USE bit of the other line in that set is set to 0.When a block is to be read into the set, the line whose USE bit is 0 is used. Because we are assuming that more recently used memory locations are more likely to be referenced, LRU should give the best hit ratio. LRU is also relatively easy to implement for a fully associative cache. The cache mechanism maintains a separate list of indexes to all the lines in the cache. When a line is referenced, it moves to the front of the list. For replacement, the line at the back of the list is used. Because of its simplicity of implementation, LRU is the most popular replacement algorithm.

Another possibility is first-in-first-out (FIFO): Replace that block in the set that has been in the cache longest. FIFO is easily implemented as a round-robin or circular buffer technique. Still another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each line. A technique not based on usage (i.e., not LRU, LFU, FIFO, or some variant) is to pick a line at random

from among the candidate lines. Simulation studies have shown that random replacement provides only slightly inferior performance to an algorithm based on usage.

4.3.5 Write Policy

When a block that is resident in the cache is to be replaced, there are two cases to consider. If the old block in the cache has not been altered, then it may be overwritten with a new block without first writing out the old block. If at least one write operation has been performed on a word in that line of the cache, then main memory must be updated by writing the line of cache out to the block of memory before bringing in the new block. A variety of write policies, with performance and economic trade-offs, is possible. There are two problems to contend with. First, more than one device may have access to main memory. For example, an I/O module may be able to read-write directly to memory. If a word has been altered only in the cache, then the corresponding memory word is invalid. Further, if the I/O device has altered main memory, then the cache word is invalid. A more complex problem occurs when multiple processors are attached to the same bus and each processor has its own local cache. Then, if a word is altered in one cache, it could conceivably invalidate a word in other caches.

The simplest technique is called **write through**. Using this technique, all write operations are made to main memory as well as to the cache, ensuring that main memory is always valid. Any other processor–cache module can monitor traffic to main memory to maintain consistency within its own cache. The main disadvantage of this technique is that it generates substantial memory traffic and may create a bottleneck. An alternative technique, known as **write back**, minimizes memory writes. With write back, updates are made only in the cache. When an update occurs, a **dirty bit**, or **use bit**, associated with the line is set. Then, when a block is replaced, it is written back to main memory if and only if the dirty bit is set. The problem with write back is that portions of main memory are invalid, and hence accesses by I/O modules can be allowed only through the cache. This makes for complex circuitry and a potential bottleneck. Experience has shown that the percentage of memory references that are writes is on the order of 15%. However, for HPC applications, this number may approach 33% (vector-vector multiplication) and can go as high as 50% (matrix transposition).

Example 4.3: Consider a cache with a line size of 32 bytes and a main memory that requires 30 ns to transfer a 4-byte word. For any line that is written at least once before being swapped out of the cache, what is the average number of times that the line must be written before being swapped out for a write-back cache to be more efficient that a write-through cache?

For the write-back case, each dirty line is written back once, at swap-out time, taking $8 \times 30 = 240$ ns. For the write-through case, each update of the line requires that one word be written out to main memory, taking 30 ns. Therefore, if the average line that gets written at least once gets written more than 8 times before swap out, then write back is more efficient.

In a bus organization in which more than one device (typically a processor) has a cache and main memory is shared, a new problem is introduced. If data in one cache are altered, this invalidates not only the corresponding word in main memory, but also that same word in other

caches (if any other cache happens to have that same word). Even if a write-through policy is used, the other caches may contain invalid data. A system that prevents this problem is said to maintain cache coherency. Possible approaches to cache coherency include the following:

- **Bus watching with write through:** Each cache controller monitors the address lines to detect write operations to memory by other bus masters. If another master writes to a location in shared memory that also resides in the cache memory, the cache controller invalidates that cache entry. This strategy depends on the use of a write-through policy by all cache controllers.
- **Hardware transparency:** Additional hardware is used to ensure that all updates to main memory via cache are reflected in all caches. Thus, if one processor modifies a word in its cache, this update is written to main memory. In addition, any matching words in other caches are similarly updated.
- **Non-cacheable memory:** Only a portion of main memory is shared by more than one processor, and this is designated as non-cacheable. In such a system, all accesses to shared memory are cache misses, because the shared memory is never copied into the cache. The non-cacheable memory can be identified using chip-select logic or high-address bits.

4.3.6 Line Size

Another design element is the line size. When a block of data is retrieved and placed in the cache, not only the desired word but also some number of adjacent words are retrieved. As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality, which states that data in the vicinity of a referenced word are likely to be referenced in the near future. As the block size increases, more useful data are brought into the cache. The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced. Two specific effects come into play:

- Larger blocks reduce the number of blocks that fit into a cache. Because each block fetch overwrites older cache contents, a small number of blocks results in data being overwritten shortly after they are fetched.
- As a block becomes larger, each additional word is farther from the requested word and therefore less likely to be needed in the near future.

The relationship between block size and hit ratio is complex, depending on the locality characteristics of a particular program, and no definitive optimum value has been found. A size of from 8 to 64 bytes seems reasonably close to optimum. For HPC systems, 64- and 128-byte cache line sizes are most frequently used.

4.3.7 Number of Caches

When caches were originally introduced, the typical system had a single cache. More recently, the use of multiple caches has become the norm. Two aspects of this design issue concern the number of levels of caches and the use of unified versus split caches.

4.3.7.1 MULTILEVEL CACHES

As logic density has increased, it has become possible to have a cache on the same chip as the processor: the on-chip cache. Compared with a cache reachable via an external bus, the on-chip cache reduces the processor's external bus activity and therefore speeds up execution times and increases overall system performance. When the requested instruction or data is found in the on-chip cache, the bus access is eliminated. Because of the short data paths internal to the processor, compared with bus lengths, on-chip cache accesses will complete appreciably faster than would even zero-wait state bus cycles. Furthermore, during this period the bus is free to support other transfers.

The inclusion of an on-chip cache leaves open the question of whether an off-chip, or external, cache is still desirable. Typically, the answer is yes, and most contemporary designs include both on-chip and external caches. The simplest such organization is known as a two-level cache, with the internal cache designated as level 1 (L1) and the external cache designated as level 2 (L2).The reason for including an L2 cache is the following: If there is no L2 cache and the processor makes an access request for a memory location not in the L1 cache, then the processor must access DRAM or ROM memory across the bus. Due to the typically slow bus speed and slow memory access time, this results in poor performance. On the other hand, if an L2 SRAM (static RAM) cache is used, then frequently the missing information can be quickly retrieved. If the SRAM is fast enough to match the bus speed, then the data can be accessed using a zero-wait state transaction, the fastest type of bus transfer. Two features of contemporary cache design for multilevel caches are noteworthy. First, for an off-chip L2 cache, many designs do not use the system bus as the path for transfer between the L2 cache and the processor, but use a separate data path, so as to reduce the burden on the system bus. Second, with the continued shrinkage of processor components, a number of processors now incorporate the L2 cache on the processor chip, improving performance.

The potential savings due to the use of an L2 cache depends on the hit rates in both the L1 and L2 caches. Several studies have shown that, in general, the use of a second-level cache does improve performance. However, the use of multilevel caches does complicate all of the design issues related to caches, including size, replacement algorithm, and write policy.

Figure 4.16 shows the results of one simulation study of two-level cache performance as a function of cache size. The figure assumes that both caches have the same line size and shows the total hit ratio. That is, a hit is counted if the desired data appears in either the L1 or the L2 cache. The figure shows the impact of L2 on total hits with respect to L1 size. L2 has little effect on the total number of cache hits until it is at least double the L1 cache size. Note that the steepest part of the slope for an L1 cache of 8 Kbytes is for an L2 cache of 16 Kbytes. Again for an L1 cache of 16 Kbytes, the steepest part of the curve is for an L2 cache size of 32 Kbytes. Prior to that point, the L2 cache has little, if any, impact on total cache performance. The need for the L2 cache to be larger than the L1 cache to affect performance makes sense. If the L2 cache has the same line size and capacity as the L1 cache, its contents will more or less mirror those of the L1 cache.

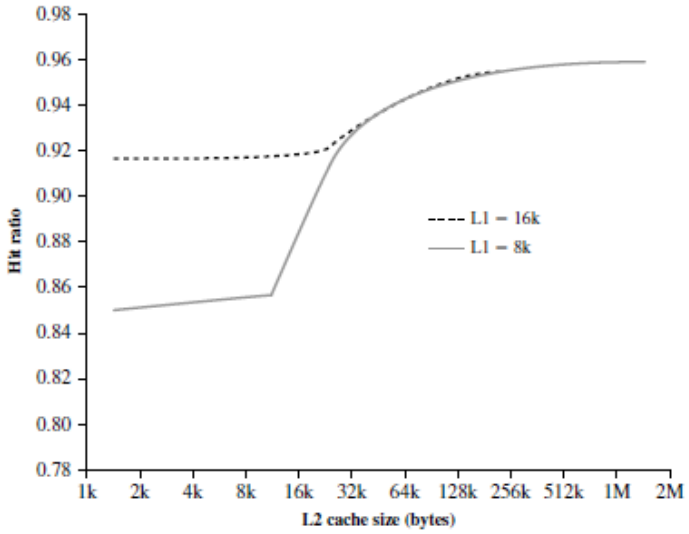


Figure 4.16 Total Hit Ratio (L1 and L2) for 8-Kbyte and 16-Kbyte L1

With the increasing availability of on-chip area available for cache, most contemporary microprocessors have moved the L2 cache onto the processor chip and added an L3 cache. Originally, the L3 cache was accessible over the external bus.

More recently, most microprocessors have incorporated an on-chip L3 cache. In either case, there appears to be a performance advantage to adding the third level.

4.3.7.2 UNIFIED VERSUS SPLIT CACHES

When the on-chip cache first made an appearance, many of the designs consisted of a single cache used to store references to both data and instructions. More recently, it has become common to split the cache into two: one dedicated to instructions and one dedicated to data. These two caches both exist at the same level, typically as two L1 caches. When the processor attempts to fetch an instruction from main memory, it first consults the instruction L1 cache, and when the processor attempts to fetch data from main memory, it first consults the data L1 cache.

There are two potential advantages of a unified cache:

- For a given cache size, a unified cache has a higher hit rate than split caches because it balances the load between instruction and data fetches automatically. That is, if an execution pattern involves many more instruction fetches than data fetches, then the cache will tend to fill up with instructions, and if an execution pattern involves relatively more data fetches, the opposite will occur.
- Only one cache needs to be designed and implemented.

Despite these advantages, the trend is toward split caches, particularly for superscalar machines such as the Pentium and PowerPC, which emphasize parallel instruction execution and the prefetching of predicted future instructions. The key advantage of the split cache design is that it eliminates contention for the cache between the instruction fetch/decode unit and the execution unit. This is important in any design that relies on the pipelining of instructions. Typically, the processor will fetch instructions ahead of time and fill a buffer, or

pipeline, with instructions to be executed. Suppose now that we have a unified instruction/data cache. When the execution unit performs a memory access to load and store data, the request is submitted to the unified cache. If, at the same time, the instruction prefetcher issues a read request to the cache for an instruction, that request will be temporarily blocked so that the cache can service the execution unit first, enabling it to complete the currently executing instruction. This cache contention can degrade performance by interfering with efficient use of the instruction pipeline. The split cache structure overcomes this difficulty.

Chapter 5
Internal Memory

5.1 Semiconductor Main Memory

In earlier computers, the most common form of random-access storage for computer main memory employed an array of doughnut-shaped ferromagnetic loops referred to as cores. Hence, main memory was often referred to as core, a term that persists to this day. The advent of, and advantages of, microelectronics has long since vanquished the magnetic core memory. Today, the use of semiconductor chips for main memory is almost universal. Key aspects of this technology are explored in this section.

5.1.1 Organization

The basic element of a semiconductor memory is the memory cell. Although a variety of electronic technologies are used, all semiconductor memory cells share certain properties:

- They exhibit two stable states, which can be used to represent binary 1 and 0.
- They are capable of being written into (at least once), to set the state.
- They are capable of being read to sense the state.

Figure 5.1 depicts the operation of a memory cell. Most commonly, the cell has three functional terminals capable of carrying an electrical signal. The select terminal, as the name suggests, selects a memory cell for a read or write operation. The control terminal indicates read or write. For writing, the other terminal provides an electrical signal that sets the state of the cell to 1 or 0. For reading, that terminal is used for output of the cell's state. The details of the internal organization, functioning, and timing of the memory cell depend on the specific integrated circuit technology used and are beyond the scope of this book, except for a brief summary. For our purposes, we will take it as given that individual cells can be selected for reading and writing operations.

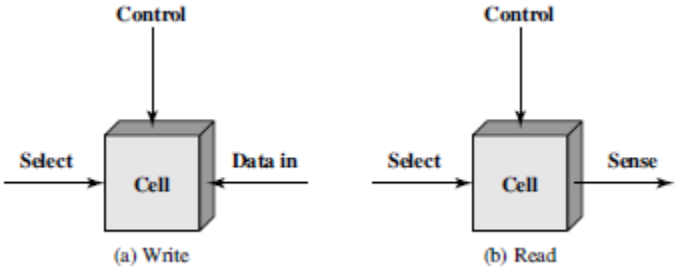


Figure 5.1 Memory Cell Operation

5.1.2 DRAM and SRAM

All of the memory types that we will explore in this chapter are random access. That is, individual words of memory are directly accessed through wired-in addressing logic.

Table 5.1 lists the major types of semiconductor memory. The most common is referred to as **random-access memory (RAM)**. This is, of course, a misuse of the term, because all of the types

listed in the table are random access. One distinguishing characteristic of RAM is that it is possible both to read data from the memory and to write new data into the memory easily and rapidly. Both the reading and writing are accomplished through the use of electrical signals.

Table 5.1 Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

The other distinguishing characteristic of RAM is that it is volatile. A RAM must be provided with a constant power supply. If the power is interrupted, then the data are lost. Thus, RAM can be used only as temporary storage. The two traditional forms of RAM used in computers are DRAM and SRAM.

5.1.3 DYNAMIC RAM

RAM technology is divided into two technologies: dynamic and static. A dynamic RAM (DRAM) is made with cells that store data as charge on capacitors. The presence or absence of charge in a capacitor is interpreted as a binary 1 or 0. Because capacitors have a natural tendency to discharge, dynamic RAMs require periodic charge refreshing to maintain data storage. The term **dynamic** refers to this tendency of the stored charge to leak away, even with power continuously applied.

Figure 5.2a is a typical DRAM structure for an individual cell that stores 1 bit. The address line is activated when the bit value from this cell is to be read or written. The transistor acts as a switch that is closed (allowing current to flow) if a voltage is applied to the address line and open (no current flows) if no voltage is present on the address line.

For the write operation, a voltage signal is applied to the bit line; a high voltage represents 1, and a low voltage represents 0. A signal is then applied to the address line, allowing a charge to be transferred to the capacitor.

For the read operation, when the address line is selected, the transistor turns on and the charge stored on the capacitor is fed out onto a bit line and to a sense amplifier. The sense amplifier compares the capacitor voltage to a reference value and determines if the cell contains a

logic 1 or a logic 0. The readout from the cell discharges the capacitor, which must be restored to complete the operation.

Although the DRAM cell is used to store a single bit (0 or 1), it is essentially an analog device. The capacitor can store any charge value within a range; a threshold value determines whether the charge is interpreted as 1 or 0.

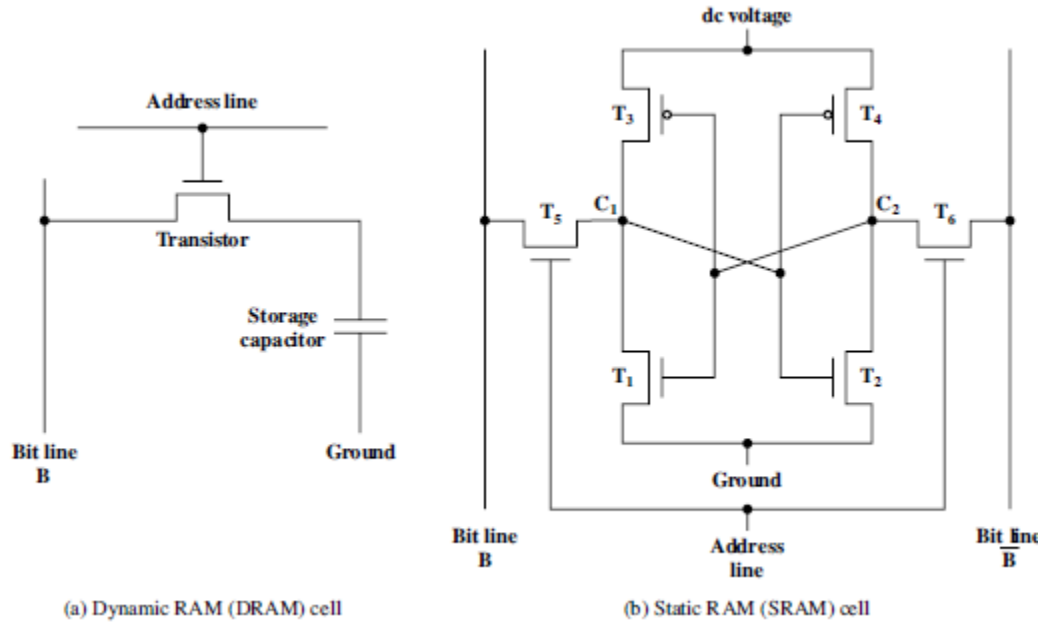


Figure 5.2 Typical Memory Cell Structures

5.1.4 STATIC RAM

In contrast, a static RAM (SRAM) is a digital device that uses the same logic elements used in the processor. In a SRAM, binary values are stored using traditional flip-flop logic-gate configurations. A static RAM will hold its data as long as power is supplied to it.

Figure 5.2b is a typical SRAM structure for an individual cell. Four transistors (T_1,T_2,T_3,T_4) are cross connected in an arrangement that produces a stable logic state. In logic state 1, point C_1 is high and point C_2 is low; in this state, T_1 and T_4 are off and T_2 and T_3 are on.1 In logic state 0, point C_1 is low and point C_2 is high; in this state, T_1 and T_4 are on and T_2 and T_3 are off. Both states are stable as long as the direct current (dc) voltage is applied. Unlike the DRAM, no refresh is needed to retain data.

As in the DRAM, the SRAM address line is used to open or close a switch. The address line controls two transistors (T_5 and T_6). When a signal is applied to this line, the two transistors are switched on, allowing a read or write operation. For a write operation, the desired bit value is applied to line B, while its complement is applied to line. This forces the four transistors (T_1, T_2, T_3, T_4) into the proper state. For a read operation, the bit value is read from line B.

5.1.5 SRAM VERSUS DRAM

Both static and dynamic RAMs are volatile; that is, power must be continuously supplied to the memory to preserve the bit values. A dynamic memory cell is simpler and smaller than a static memory cell. Thus, a DRAM is more dense (smaller cells more cells per unit area) and less expensive than a corresponding SRAM. On the other hand, a DRAM requires the supporting refresh circuitry. For larger memories, the fixed cost of the refresh circuitry is more than compensated for by the smaller variable cost of DRAM cells. Thus, DRAMs tend to be favored for large memory requirements. A final point is that SRAMs are generally somewhat faster than DRAMs. Because of these relative characteristics, SRAM is used for cache memory (both on and off chip), and DRAM is used for main memory.

5.1.6 Types of ROM

As the name suggests, a **read-only memory** (ROM) contains a permanent pattern of data that cannot be changed. A ROM is nonvolatile; that is, no power source is required to maintain the bit values in memory.While it is possible to read a ROM, it is not possible to write new data into it. An important application of ROMs is microprogramming. Other potential applications include

- Library subroutines for frequently wanted functions
- System programs
- Function tables

For a modest-sized requirement, the advantage of ROM is that the data or program is permanently in main memory and need never be loaded from a secondary storage device. A ROM is created like any other integrated circuit chip, with the data actually wired into the chip as part of the fabrication process. This presents two problems:

- The data insertion step includes a relatively large fixed cost, whether one or thousands of copies of a particular ROM are fabricated.
- There is no room for error. If one bit is wrong, the whole batch of ROMs must be thrown out.

When only a small number of ROMs with a particular memory content is needed, a less expensive alternative is the **programmable ROM** (PROM). Like the ROM, the PROM is nonvolatile and may be written into only once. For the PROM, the writing process is performed electrically and may be performed by a supplier or customer at a time later than the original chip fabrication. Special equipment is required for the writing or “programming” process. PROMs provide flexibility and convenience. The ROM remains attractive for high-volume production runs.

Another variation on read-only memory is the read-mostly memory, which is useful for applications in which read operations are far more frequent than write operations but for which nonvolatile storage is required. There are three common forms of read-mostly memory: EPROM, EEPROM, and flash memory.

The optically **erasable programmable read-only memory** (EPROM) is read and written electrically, as with PROM. However, before a write operation, all the storage cells must be erased

to the same initial state by exposure of the packaged chip to ultraviolet radiation. Erasure is performed by shining an intense ultraviolet light through a window that is designed into the memory chip. This erasure process can be performed repeatedly; each erasure can take as much as 20 minutes to perform. Thus, the EPROM can be altered multiple times and, like the ROM and PROM, holds its data virtually indefinitely. For comparable amounts of storage, the EPROM is more expensive than PROM, but it has the advantage of the multiple update capability.

A more attractive form of read-mostly memory is **electrically erasable programmable read-only memory** (EEPROM). This is a read-mostly memory that can be written into at any time without erasing prior contents; only the byte or bytes addressed are updated. The write operation takes considerably longer than the read operation, on the order of several hundred microseconds per byte. The EEPROM combines the advantage of nonvolatility with the flexibility of being updatable in place, using ordinary bus control, address, and data lines. EEPROM is more expensive than EPROM and also is less dense, supporting fewer bits per chip.

Another form of semiconductor memory is **flash memory** (so named because of the speed with which it can be reprogrammed). First introduced in the mid-1980s, flash memory is intermediate between EPROM and EEPROM in both cost and functionality. Like EEPROM, flash memory uses an electrical erasing technology. An entire flash memory can be erased in one or a few seconds, which is much faster than EPROM. In addition, it is possible to erase just blocks of memory rather than an entire chip. Flash memory gets its name because the microchip is organized so that a section of memory cells are erased in a single action or “flash.” However, flash memory does not provide byte-level erasure. Like EPROM, flash memory uses only one transistor per bit, and so achieves the high density (compared with EEPROM) of EPROM.

5.1.7 Chip Logic

As with other integrated circuit products, semiconductor memory comes in packaged chips. Each chip contains an array of memory cells.

In the memory hierarchy as a whole, we saw that there are trade-offs among speed, capacity, and cost. These trade-offs also exist when we consider the organization of memory cells and functional logic on a chip. For semiconductor memories, one of the key design issues is the number of bits of data that may be read/written at a time. At one extreme is an organization in which the physical arrangement of cells in the array is the same as the logical arrangement (as perceived by the processor) of words in memory. The array is organized into W words of B bits each. For example, a 16-Mbit chip could be organized as 1M 16-bit words. At the other extreme is the so-called 1-bit-per-chip organization, in which data are read/written 1 bit at a time.

Figure 5.3 shows a typical organization of a 16-Mbit DRAM. In this case, 4 bits are read or written at a time. Logically, the memory array is organized as four-square arrays of 2048 by 2048 elements. Various physical arrangements are possible. In any case, the elements of the array are connected by both horizontal (row) and vertical (column) lines. Each horizontal line connects to the Select terminal of each cell in its row; each vertical line connects to the Data-In/Sense terminal of each cell in its column.

Address lines supply the address of the word to be selected. A total of $\log_2 W$ lines are needed. In our example, 11 address lines are needed to select one of 2048 rows. These 11 lines are fed into a row decoder, which has 11 lines of input and 2048 lines for output. The logic of the decoder activates a single one of the 2048 outputs depending on the bit pattern on the 11 input lines ($2^{11} = 2048$).

An additional 11 address lines select one of 2048 columns of 4 bits per column. Four data lines are used for the input and output of 4 bits to and from a data buffer. On input (write), the bit driver of each bit line is activated for a 1 or 0 according to the value of the corresponding data line. On output (read), the value of each bit line is passed through a sense amplifier and presented to the data lines. The row line selects which row of cells is used for reading or writing.

Because only 4 bits are read/written to this DRAM, there must be multiple DRAMs connected to the memory controller to read/write a word of data to the bus.

Note that there are only 11 address lines (A0–A10), half the number you would expect for a 2048 2048 array. This is done to save on the number of pins. The 22 required address lines are passed through select logic external to the chip and multiplexed onto the 11 address lines. First, 11 address signals are passed to the chip to define the row address of the array, and then the other 11 address signals are presented for the column address. These signals are accompanied by row address select ($\overline{\text{RAS}}$) and column address select ($\overline{\text{CAS}}$) signals to provide timing to the chip.

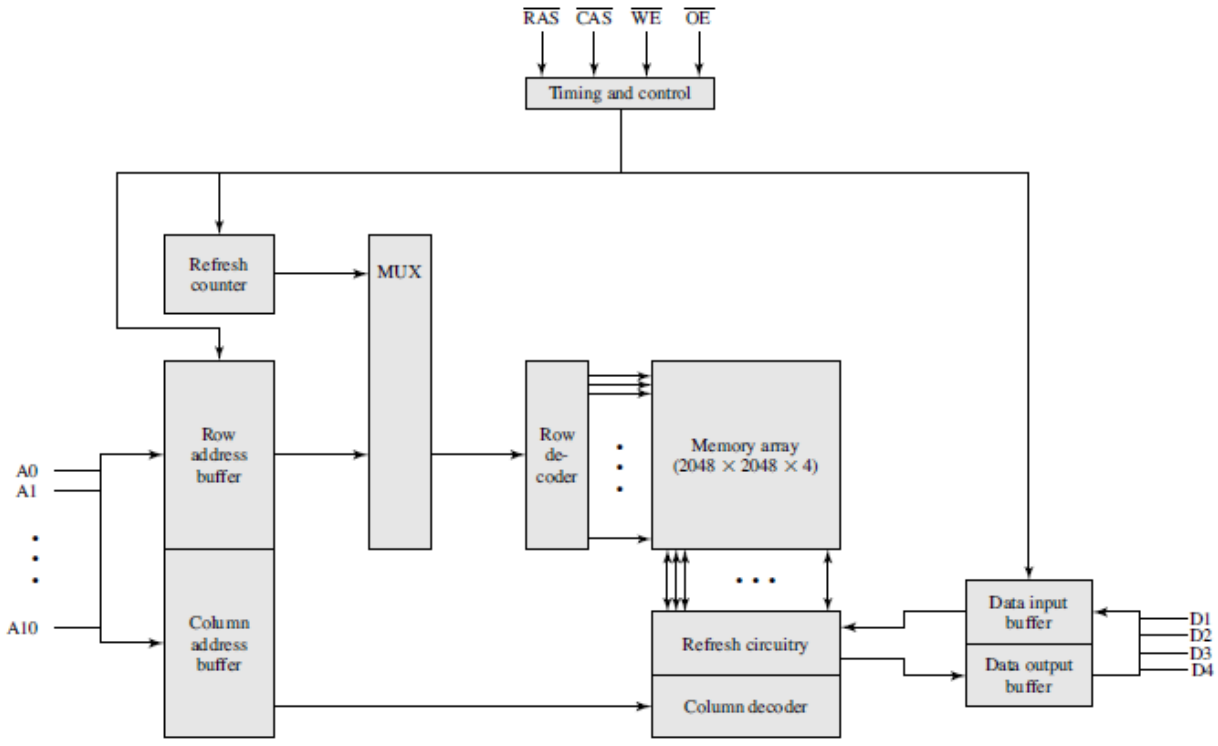


Figure 5.3 Typical 16 Megabit DRAM (4M * 4)

The write enable ($\overline{\text{WE}}$) and output enable ($\overline{\text{OE}}$) pins determine whether a write or read operation is performed. Two other pins, not shown in Figure 5.3, are ground (Vss) and a voltage source (Vcc).

As an aside, multiplexed addressing plus the use of square arrays result in a quadrupling of memory size with each new generation of memory chips. One more pin devoted to addressing doubles the number of rows and columns, and so the size of the chip memory grows by a factor of 4.

Figure 5.3 also indicates the inclusion of refresh circuitry. All DRAMs require a refresh operation. A simple technique for refreshing is, in effect, to disable the DRAM chip while all data cells are refreshed. The refresh counter steps through all of the row values. For each row, the output lines from the refresh counter are supplied to the row decoder and the RAS line is activated. The data are read out and written back into the same location. This causes each cell in the row to be refreshed.

5.1.8 Module Organization

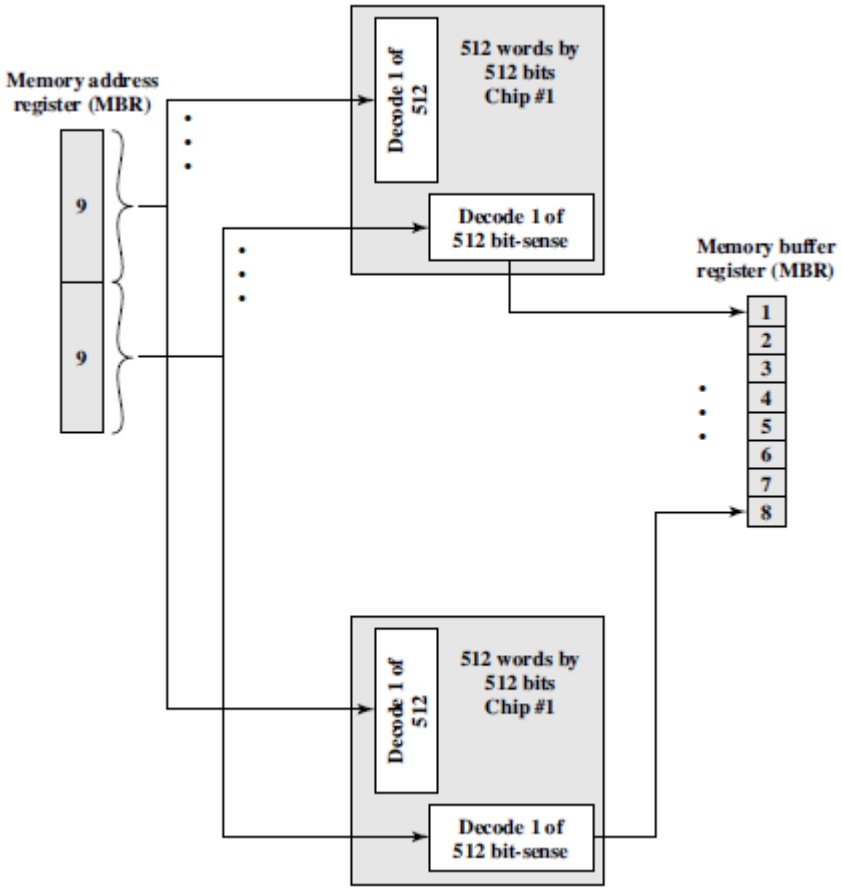


Figure 5.4 256-KByte Memory Organization

If a RAM chip contains only 1 bit per word, then clearly we will need at least a number of chips equal to the number of bits per word. As an example, Figure 5.4 shows how a memory module

consisting of 256K 8-bit words could be organized. For 256K words, an 18-bit address is needed and is supplied to the module from some external source (e.g., the address lines of a bus to which the module is attached).The address is presented to 8 256K 1-bit chips, each of which provides the input/output of 1 bit. This organization works as long as the size of memory equals the number of bits per chip. In the case in which larger memory is required, an array of chips is needed. Figure 5.5 shows the possible organization of a memory consisting of 1M word by 8 bits per word. In this case, we have four columns of chips, each column containing 256K words arranged as in Figure 5.4. For 1M word, 20 address lines are needed. The 18 least significant bits are routed to all 32 modules. The high-order 2 bits are input to a group select logic module that sends a chip enable signal to one of the four columns of modules.

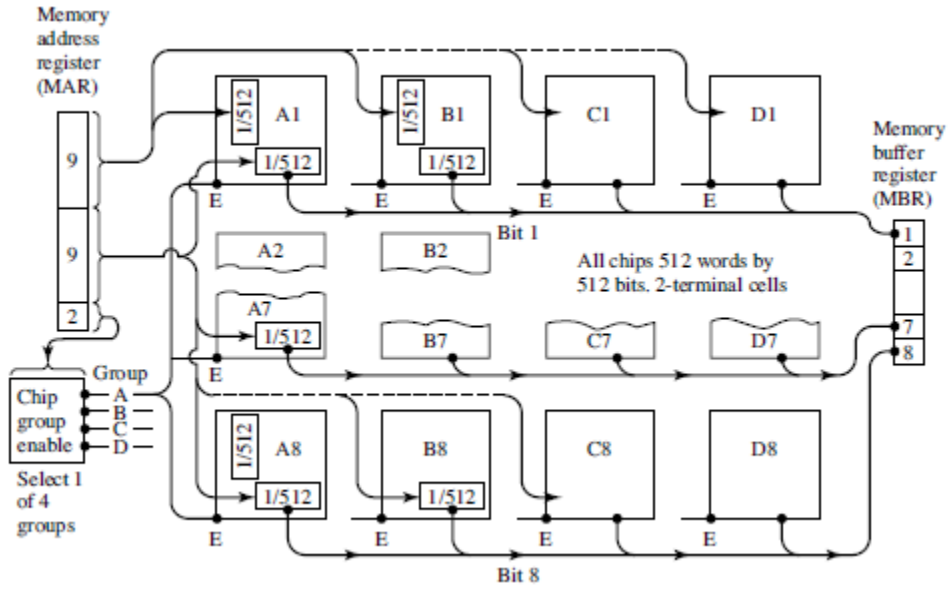


Figure 5.5 1-Mbyte Memory Organization

5.1.9 Interleaved Memory

Main memory is composed of a collection of DRAM memory chips. A number of chips can be grouped together to form a **memory bank**. It is possible to organize the memory banks in a way known as interleaved memory. Each bank is independently able to service a memory read or write request, so that a system with K banks can service K requests simultaneously, increasing memory read or write rates by a factor of K. If consecutive words of memory are stored in different banks, then the transfer of a block of memory is speeded up.

5.2 ERROR CORRECTION

A semiconductor memory system is subject to errors. These can be categorized as hard failures and soft errors. A **hard failure** is a permanent physical defect so that the memory cell or cells affected cannot reliably store data but become stuck at 0 or 1 or switch erratically between 0 and 1. Hard errors can be caused by harsh environmental abuse, manufacturing defects, and wear. A **soft error** is a random, nondestructive event that alters the contents of one or more memory cells

without damaging the memory. Soft errors can be caused by power supply problems or alpha particles. These particles result from radioactive decay and are distressingly common because radioactive nuclei are found in small quantities in nearly all materials. Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors.

Figure 5.6 illustrates in general terms how the process is carried out. When data are to be read into memory, a calculation, depicted as a function f, is performed on the data to produce a code. Both the code and the data are stored. Thus, if an M-bit word of data is to be stored and the code is of length K bits, then the actual size of the stored word is $M + K$ bits.

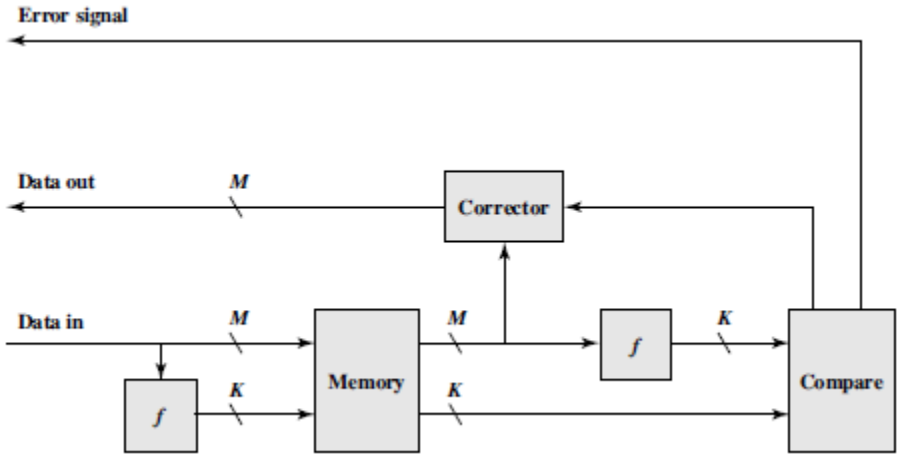


Figure 5.6 Error-Correcting Code Function

When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of K code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The fetched data bits are sent out.
- An error is detected, and it is possible to correct the error. The data bits plus error correction bits are fed into a corrector, which produces a corrected set of M bits to be sent out.
- An error is detected, but it is not possible to correct it. This condition is reported.

Codes that operate in this fashion are referred to as **error-correcting codes**. A code is characterized by the number of bit errors in a word that it can correct and detect. The simplest of the error-correcting codes is the **Hamming code** devised by Richard Hamming at Bell Laboratories. Figure 5.7 uses Venn diagrams to illustrate the use of this code on 4-bit words ($M = 4$). With three intersecting circles, there are seven compartments. We assign the 4 data bits to the inner compartments (Figure 5.7a). The remaining compartments are filled with what are called **parity bits**.

Each parity bit is chosen so that the total number of 1s in its circle is even (Figure 5.7b). Thus, because circle A includes three data 1s, the parity bit in that circle is set to 1. Now, if an error

changes one of the data bits (Figure 5.7c), it is easily found. By checking the parity bits, discrepancies are found in circle A and circle C but not in circle B. Only one of the seven compartments is in A and C but not B. The error can therefore be corrected by changing that bit.

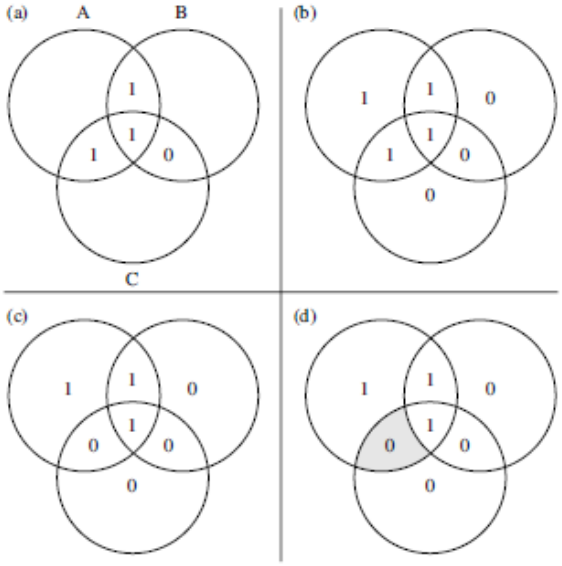


Figure 5.7 Hamming Error-Correcting Code

To clarify the concepts involved, we will develop a code that can detect and correct single-bit errors in 8-bit words.

To start, let us determine how long the code must be. Referring to Figure 5.6, the comparison logic receives as input two K-bit values. A bit-by-bit comparison is done by taking the exclusive-OR of the two inputs. The result is called the **syndrome word**. Thus, each bit of the syndrome is 0 or 1 according to if there is or is not a match in that bit position for the two inputs.

The syndrome word is therefore K bits wide and has a range between 0 and $2^K - 1$. The value 0 indicates that no error was detected, leaving $2^K - 1$ values to indicate, if there is an error, which bit was in error. Now, because an error could occur on any of the M-data bits or K check bits, we must have

$$2^K - 1 \geq M + K$$

This inequality gives the number of bits needed to correct a single bit error in a word containing M data bits. For example, for a word of 8 data bits ($M = 8$), we have

- $K = 3: 2^3 - 1 < 8 + 3$
- $K = 4: 2^4 - 1 > 8 + 4$

Thus, eight data bits require four check bits. The first three columns of Table 5.2 lists the number of check bits required for various data word lengths.

For convenience, we would like to generate a 4-bit syndrome for an 8-bit data word with the following characteristics:

- If the syndrome contains all 0s, no error has been detected.
- If the syndrome contains one and only one bit set to 1, then an error has occurred in one of the 4 check bits. No correction is needed.

- If the syndrome contains more than one bit set to 1, then the numerical value of the syndrome indicates the position of the data bit in error. This data bit is inverted for correction.

Table 5.2 Increase in Word Length with Error Correction

	Single-Error Correction		Single-Error Correction/ Double-Error Detection	
Data Bits	Check Bits	%increase	Check Bits	%increase
8	4	50	5	62.5
16	5	31.25	6	37.5
32	6	18.75	7	21.875
64	7	10.94	8	12.5
128	8	6.25	9	7.03
256	9	3.52	10	3.91

To achieve these characteristics, the data and check bits are arranged into a 12-bit word as depicted in Figure 5.8. The bit positions are numbered from 1 to 12. Those bit positions whose position numbers are powers of 2 are designated as check bits. The check bits are calculated as follows, where the symbol designates the exclusive-OR operation:

$$C1 = D1 \oplus D2 \oplus D4 \oplus D5 \oplus D7$$
$$C2 = D1 \oplus D3 \oplus D4 \oplus D6 \oplus D7$$
$$C4 = D2 \oplus D3 \oplus D4 \oplus D8$$
$$C8 = D5 \oplus D6 \oplus D7 \oplus D8$$

Bit position	12	11	10	9	8	7	6	5	4	3	2	1
Position number	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Data bit	D8	D7	D6	D5		D4	D3	D2		D1		
Check bit					C8				C4		C2	C1

Figure 5.8 Layout of Data Bits and Check Bits

Each check bit operates on every data bit whose position number contains a 1 in the same bit position as the position number of that check bit. Thus, data bit positions 3, 5, 7, 9, and 11 (D1, D2, D4, D5, D7) all contain a 1 in the least significant bit of their position number as does C1; bit positions 3, 6, 7, 10, and 11 all contain a 1 in the second bit position, as does C2; and so on. Looked at another way, bit position n is checked by those bits Ci such that $\sum_i = n$. For example, position 7 is checked by bits in position 4, 2, and 1; and $7 = 4 + 2 + 1$.

Let us verify that this scheme works with an example. Assume that the 8-bit input word is 00111001, with data bit D1 in the rightmost position. The calculations are as follows:

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C4 = 0 \oplus 0 \oplus 1 \oplus 0 = 1$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

Suppose now that data bit 3 sustains an error and is changed from 0 to 1. When the check bits are recalculated, we have

$$C1 = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$$
$$C2 = 1 \oplus 1 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C4 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$$
$$C8 = 1 \oplus 1 \oplus 0 \oplus 0 = 0$$

When the new check bits are compared with the old check bits, the syndrome word is formed:

$$\begin{array}{cccc} C8 & C4 & C2 & C1 \\ 0 & 1 & 1 & 1 \\ \oplus\ 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \end{array}$$

The result is 0110, indicating that bit position 6, which contains data bit 3, is in error.

The code just described is known as a **single-error-correcting** (SEC) code. More commonly, semiconductor memory is equipped with a **single-error-correcting, double-error-detecting** (SEC-DED) code. As Table 5.2 shows, such codes require one additional bit compared with SEC codes.

Figure 5.9 illustrates how such a code works, again with a 4-bit data word. The sequence shows that if two errors occur (Figure 5.9c), the checking procedure goes astray (d) and worsens the problem by creating a third error (e). To overcome the problem, an eighth bit is added that is set so that the total number of 1s in the diagram is even. The extra parity bit catches the error (f).

An error-correcting code enhances the reliability of the memory at the cost of added complexity. With a 1-bit-per-chip organization, an SEC-DED code is generally

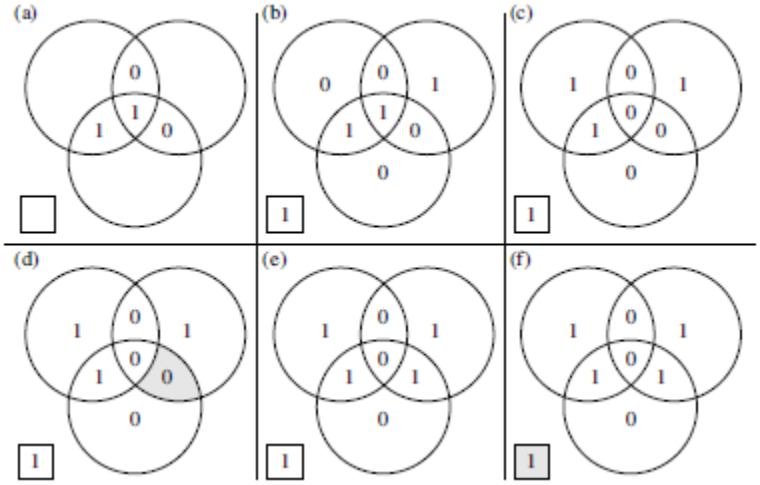


Figure 5.9 Hamming SEC-DEC Code

Chapter 6
Computer Arithmetic

6.1 The Arithmetic And Logic Unit

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU. An ALU and, indeed, all electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. Figure 6.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Data are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.8). The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored. The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

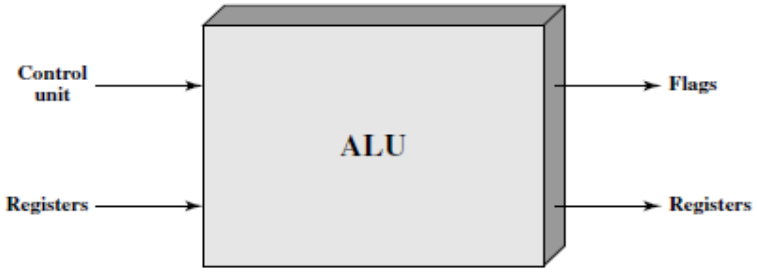


Figure 6.1 ALU Inputs and Outputs

6.2 Integer Representation

In the binary number system, arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or **radix point**.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

6.2.1 Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost bits hold the magnitude of the integer.

$$+18 = 00010010$$
$$-18 = 10010010 \quad \text{(sign magnitude)}$$

The general case can be expressed as follows:

$$A = \begin{cases} \sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases}$$

There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. Another drawback is that there are two representations of 0:

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \quad \text{(sign magnitude)}$$

This is inconvenient because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation. Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.

6.2.2 Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It differs from the use of the sign-magnitude representation in the way that the other bits are interpreted. Table 6.1 highlights key characteristics of twos complement representation and arithmetic, which are elaborated in this section and the next.

The twos complement representation is best understood by defining it in terms of a weighted sum of bits, as we did previously for unsigned and sign-magnitude representations. The advantage of this treatment is that it does not leave any lingering doubt that the rules for arithmetic operations in twos complement notation may not work for some special cases.

Table 6.1 Characteristics of Twos Complement Representation and Arithmetic

Range	-2^{n-1} through $2^{n-1}-1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract B from A, take the twos complement of B and add it to A.

Consider an n-bit integer, A, in twos complement representation. If A is positive, then the sign bit, a_{n-1} is zero. The remaining bits represent the magnitude of the number in the same fashion as for sign magnitude:

$$A = \sum_{i=0}^{n-2} 2^i a_i \quad \text{for } A \geq 0$$

The number zero is identified as positive and therefore has a 0 sign bit and a magnitude of all 0s.We can see that the range of positive integers that may be represented is from 0 (all of the magnitude bits are 0) through $2^{n-1}-1$ (all of the magnitude bits are 1). Any larger number would require more bits.

Now, for a negative number $A(A < 0)$the sign bit a_{n-1}, is one. The remaining $n-1$ bits can take on any one of 2^{n-1} values. Therefore, the range of negative integers that can be represented is from -1 to -2^{n-1} We would like to assign the bit values to negative integers in such a way that arithmetic can be handled in a straightforward fashion, similar to unsigned integer arithmetic. In unsigned integer representation, to compute the value of an integer from the bit representation, the weight of the most significant bit is $+2^{n-1}$. For a representation with a sign bit, it turns out that the desired arithmetic properties are achieved, if the weight of the most significant bit is This is -2^{n-1}. the convention used in twos complement representation, yielding the following expression for negative numbers:

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \quad \text{(Twos Complement)}$$

The previous equation defines the twos complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1} = 0$ and the equation defines a nonnegative integer. When $a_{n-1} = 1$ the term 2^{n-1} is subtracted from the summation term, yielding a negative integer.

Table 6.2 compares the sign-magnitude and twos complement representations for 4-bit integers. Although twos complement is an awkward representation from the human point of view, we will see that it facilitates the most important arithmetic operations, addition and subtraction. For this reason, it is almost universally used as the processor representation for integers.

Table 6.2 Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	—	—
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	—	1000	—

A useful illustration of the nature of twos complement representation is a value box, in which the value on the far right in the box is $1\ (2^0)$ and each succeeding position to the left is double in value, until the leftmost position, which is negated. As you can see in Figure 6.2a, the most negative twos complement number that can be represented is -2^{n-1}; if any of the bits other than the sign bit is one, it adds a positive amount to the number. Also, it is clear that a negative number must have a 1 at its leftmost position and a positive number must have a 0 in that position. Thus, the largest positive number is a 0 followed by all 1s, which equals $2^{n-1} - 1$.

The rest of Figure 6.2 illustrates the use of the value box to convert from twos complement to decimal and from decimal to twos complement.

−128	64	32	16	8	4	2	1

(a) An eight-position twos complement value box

−128	64	32	16	8	4	2	1
1	0	0	0	0	0	1	1

$-128 \qquad +2 \quad +1 \quad = -125$

(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$-120 = -128 \qquad +8$

Figure 6.2 Use of a Value Box for Conversion between Twos Complement Binary and Decimal

6.2.3 Converting between Different Bit Lengths

It is sometimes desirable to take an n-bit integer and store it in m bits, where $m > n$. In sign-magnitude notation, this is easily accomplished: simply move the sign bit to the new leftmost position and fill in with zeros.

$+18 = 00010010$ (sign magnitude, 8 bits)
$+18 = 0000000000010010$ (sign magnitude, 16 bits)
$-18 = 10010010$ (sign magnitude, 8 bits)
$-18 = 1000000000010010$ (sign magnitude, 16 bits)

This procedure will not work for twos complement negative integers. Using the same example,

$+18 = 00010010$ (twos complement, 8 bits)
$+18 = 0000000000010010$ (twos complement, 16 bits)
$-18 = 11101110$ (twos complement, 8 bits)
$-32{,}658 = 1000000001101110$ (twos complement, 16 bits)

Instead, the rule for twos complement integers is to move the sign bit to the new leftmost position and fill in with copies of the sign bit. For positive numbers, fill in with zeros, and for negative numbers, fill in with ones. This is called sign extension.

$-18 = 11101110$ (twos complement, 8 bits)
$-18 = 1111111111101110$ (twos complement, 16 bits)

6.2.4 Fixed-Point Representation

Finally, we mention that the representations discussed in this section are sometimes referred to as fixed point. This is because the radix point (binary point) is fixed and assumed to be to the right of the rightmost digit. The programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location.

6.3 INTEGER ARITHMETIC

This section examines common arithmetic functions on numbers in twos complement representation.

6.3.1 Negation

In sign-magnitude representation, the rule for forming the negation of an integer is simple: invert the sign bit. In twos complement notation, the negation of an integer can be formed with the following rules:

1. Take the Boolean complement of each bit of the integer (including the sign bit).That is, set each 1 to 0 and each 0 to 1.
2. Treating the result as an unsigned binary integer, add 1.

This two-step process is referred to as the twos complement operation, or the taking of the twos complement of an integer.

$$\begin{aligned} +18 &= 00010010 \quad \text{(twos complement)} \\ \text{bitwise complement} &= 11101101 \\ + & \quad 1 \\ \hline 11101110 &= -18 \end{aligned}$$

As expected, the negative of the negative of that number is itself:

$$\begin{aligned} +18 &= 11101110 \quad \text{(twos complement)} \\ \text{bitwise complement} &= 00010001 \\ + & \quad 1 \\ \hline 00010010 &= +18 \end{aligned}$$

The preceding derivation assumes that we can first treat the bitwise complement of A as an unsigned integer for the purpose of adding 1, and then treat the result as a twos complement integer. There are two special cases to consider. First, consider $A = 0$. In that case, for an 8-bit representation:

$$\begin{aligned} +0 &= 00000000 \quad \text{(twos complement)} \\ \text{bitwise complement} &= 11111111 \\ + & \quad 1 \\ \hline 100000000 &= 0 \end{aligned}$$

There is **carry** out of the most significant bit position, which is ignored. The result is that the negation of 0 is 0, as it should be.

The second special case is more of a problem. If we take the negation of the bit pattern of 1 followed by $n - 1$ zeros, we get back the same number. For example, for 8-bit words,

$$\begin{aligned} -128 &= 10000000 \quad \text{(twos complement)} \\ \text{bitwise complement} &= 01111111 \\ + & \quad 1 \\ \hline 100000000 &= -128 \end{aligned}$$

Some such anomaly is unavoidable. The number of different bit patterns in an n-bit word is 2^n which is an even number. We wish to represent positive and negative integers and 0. If an equal number of positive and negative integers are represented (sign magnitude), then there are two representations for 0. If there is only one representation of 0 (twos complement), then there must be an unequal number of negative and positive numbers represented. In the case of twos complement, for an n-bit length, there is a representation for -2^{n-1} but not for $+2^{n-1}$.

6.3.2 Addition and Subtraction

Addition in twos complement is illustrated in Figure 6.3. Addition proceeds as if the two numbers were unsigned integers. The first four examples illustrate successful operations. If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form. If the result of the operation is negative, we get a negative number in twos complement form. Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.

On any addition, the result may be larger than can be held in the word size being used. This condition is called **overflow**. When overflow occurs, the ALU must signal this fact so that no attempt is made to use the result. To detect overflow, the following rule is observed:

OVERFLOW RULE: If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

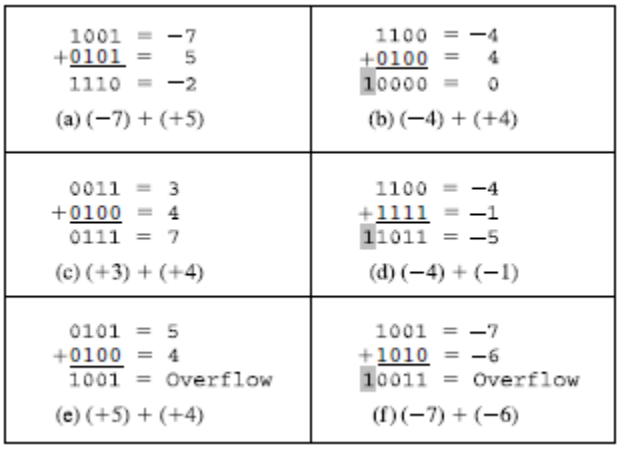


Figure 6.3 Addition of Numbers in Twos Complement Representation

Figures 6.3e and f show examples of overflow. Note that overflow can occur whether or not there is a carry.

Subtraction is easily handled with the following rule:

SUBTRACTION RULE: To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Thus, subtraction is achieved using addition, as illustrated in Figure 6.4. The last two examples demonstrate that the overflow rule still applies.

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 10011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7 (c) M = −5 = 1011 S = 2 = 0010 −S = 1110	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 +1100 = −4 10110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 6.4 Subtraction of Numbers in Twos Complement Representation ($M - S$)

Figure 6.5 suggests the data paths and hardware elements needed to accomplish addition and subtraction. The central element is a binary adder, which is presented two numbers for addition and produces a sum and an overflow indication. The binary adder treats the two numbers as unsigned integers. For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers. The result may be stored in one of these registers or in a third. The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow). For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder. Note that Figure 6.5 only shows the data paths. Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.

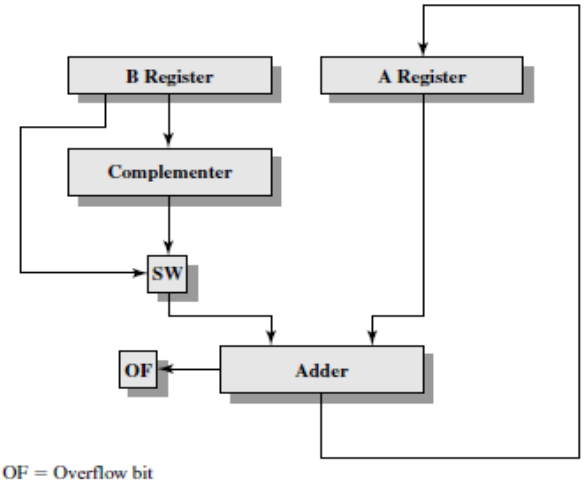


Figure 6.5 Block Diagram of Hardware for Addition and Subtraction

6.3.3 Multiplication

Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software. A wide variety of algorithms have been used in various computers. The purpose of this subsection is to give the reader some feel for the type of approach typically taken. We begin with the simpler problem of multiplying two unsigned (nonnegative) integers, and then we look at one of the most common techniques for multiplication of numbers in twos complement representation.

6.3.3.1 Unsigned Integers

Figure 6.6 illustrates the multiplication of unsigned binary integers, as might be carried out using paper and pencil. Several important observations can be made:

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two n-bit binary integers results in a product of up to bits in length (e.g., $11 \times 11 = 1001$).

Compared with the pencil-and-paper approach, there are several things we can do to make computerized multiplication more efficient. First, we can perform a running addition on the partial products rather than waiting until the end. This eliminates the need for storage of all the partial products; fewer registers are needed. Second, we can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.

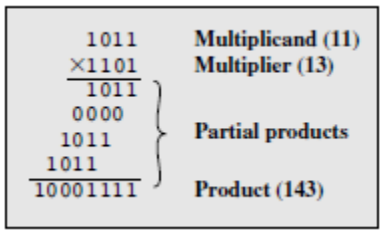
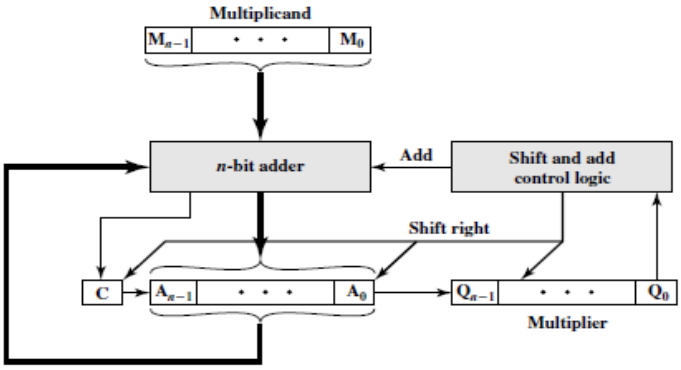


Figure 6.6 Multiplication of Unsigned Binary Integers

Figure 6.7a shows a possible implementation employing these measures. The multiplier and multiplicand are loaded into two registers (Q and M). A third register, the A register, is also needed and is initially set to 0. There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition. The operation of the multiplier is as follows. Control logic reads the bits of the multiplier one at a time. If Q_0 is 1, then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow. Then all of the bits of the C, A, and

Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1} and Q_0 is lost. If Q_0 is 0, then no addition is performed, just the shift. This process is repeated for each bit of the original multiplier. The resulting $2n$-bit product is contained in the A and Q registers.



(a) Block diagram

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

b) Example from Figure 6.6 (product in A, Q)

Figure 6.7 Hardware Implementation of Unsigned Binary Multiplication

A flowchart of the operation is shown in Figure 6.8, and an example is given in Figure 6.7b. Note that on the second cycle, when the multiplier bit is 0, there is no add operation.

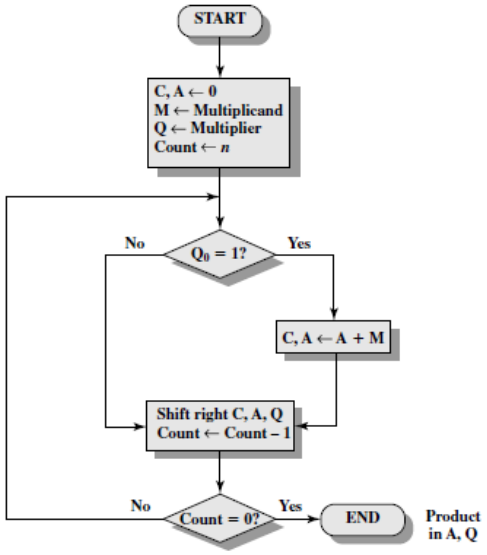


Figure 6.8 Flowchart for Unsigned Binary Multiplication

6.3.4 Twos Complement Multiplication

We have seen that addition and subtraction can be performed on numbers in twos complement notation by treating them as unsigned integers. Consider

$$\begin{array}{r} 1001 \\ +\ 0011 \\ \hline 1100 \end{array}$$

If these numbers are considered to be unsigned integers, then we are adding 9 (1001) plus 3 (0011) to get 12 (1100).As twos complement integers, we are adding -7 (1001) to 3 (0011) to get -4 (1100).

Unfortunately, this simple scheme will not work for multiplication. To see this, consider again Figure 6.6. We multiplied 11 (1011) by 13 (1101) to get 143 (10001111). If we interpret these as twos complement numbers, we have -5 (1011) times -3 (1101) equals -113 (10001111). This example demonstrates that straightforward multiplication will not work if both the multiplicand and multiplier are negative. In fact, it will not work if either the multiplicand or the multiplier is negative. To justify this statement, we need to go back to Figure 6.6 and explain what is being done in terms of operations with powers of 2. Recall that any unsigned binary number can be expressed as a sum of powers of 2.Thus,

$$1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 2^3 + 2^2 + 2^0$$

Further, the multiplication of a binary number by 2^n is accomplished by shifting that number to the left n bits. With this in mind, Figure 6.9 recasts Figure 6.6 to make the generation of partial products by multiplication explicit. The only difference in Figure 6.9 is that it recognizes that the partial products should be viewed as 2n-bit numbers generated from the n-bit multiplicand.

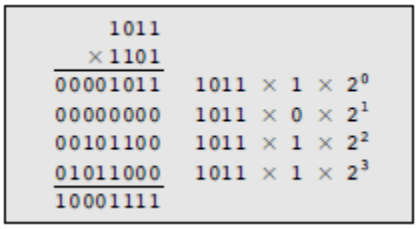


Figure 6.9 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Thus, as an unsigned integer, the 4-bit multiplicand 1011 is stored in an 8-bit word as 00001011. Each partial product (other than that for) consists of this number shifted to the left, with the unoccupied positions on the right filled with zeros (e.g., a shift to the left of two places yields 00101100).

Now we can demonstrate that straightforward multiplication will not work if the multiplicand is negative. The problem is that each contribution of the negative multiplicand as a partial product must be a negative number on a $2n$-bit field; the sign bits of the partial products must line up. This

is demonstrated in Figure 6.10, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of 9 * 3 = 27 proceeds simply. However, if 1001 is interpreted as

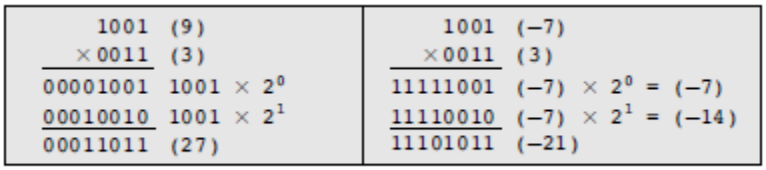


Figure 6.10 Comparison of Multiplication of Unsigned and Twos Complement Integers

the twos complement value -7, then each partial product must be a negative twos complement number of (8) bits, as shown in Figure 6.10b. Note that this is accomplished by padding out each partial product to the left with binary 1s.

If the multiplier is negative, straightforward multiplication also will not work. The reason is that the bits of the multiplier no longer correspond to the shifts or multiplications that must take place. For example, the 4-bit decimal number -3 is written 1101 in twos complement. If we simply took partial products based on each bit position, we would have the following correspondence:

$$1101 \leftrightarrow -(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0) = -(2^3 + 2^2 + 2^0)$$

In fact, what is desired is $-(2^1 + 2^0)$. So this multiplier cannot be used directly in the manner we have been describing. There are a number of ways out of this dilemma. One would be to convert both multiplier and multiplicand to positive numbers, perform the multiplication, and then take the twos complement of the result if and only if the sign of the two original numbers differed. Implementers have preferred to use techniques that do not require this final transformation step. One of the most common of these is Booth's algorithm. This algorithm also has the benefit of speeding up the multiplication process, relative to a more straightforward approach.

Booth's algorithm is depicted in Figure 6.11 and can be described as follows. As before, the multiplier and multiplicand are placed in the Q and M registers, respectively. There is also a 1-bit register placed logically to the right of the least significant bit (Q_0) of the Q register and designated Q_{-1}; its use is explained shortly. The results of the multiplication will appear in the A and Q registers. A and Q_{-1} are initialized to 0. As before, control logic scans the bits of the multiplier one at a time. Now, as each bit is examined, the bit to its right is also examined. If the two bits are the same (1–1 or 0–0), then all of the bits of the A,Q, and Q_{-1} registers are shifted to the right 1 bit. If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits are 0–1 or 1–0. Following the addition or subtraction, the right shift occurs. In either case, the right shift is such that the leftmost bit of A, namely A_{n-1}, not only is shifted into A_{n-2}, but also remains in A_{n-1}. This is required to preserve the sign of the number in A and Q. It is known as an arithmetic shift, because it preserves the sign bit.

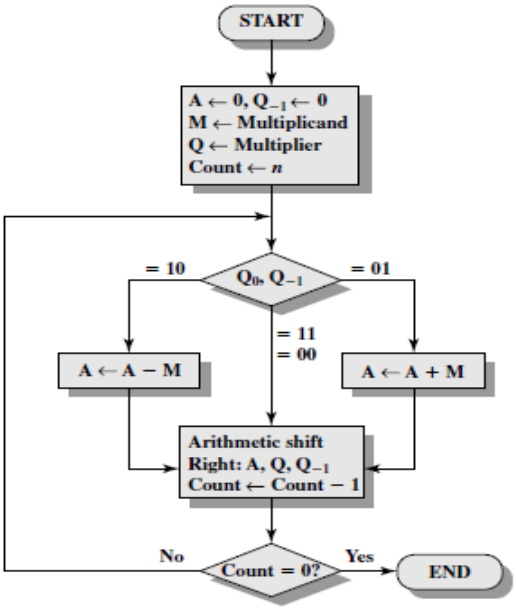


Figure 6.11 Booth's Algorithm for Twos Complement Multiplication

Figure 6.12 shows the sequence of events in Booth's algorithm for the multiplication of 7 by 3.

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial values	
1001	0011	0	0111	A ← A − M	First cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second cycle
0101	0100	1	0111	A ← A + M	Third cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth cycle

Figure 6.12 Example of Booth's Algorithm (7 × 3)

More compactly, the same operation is depicted in Figure 6.13a. The rest of Figure 6.13 gives other examples of the algorithm. As can be seen, it works with any combination of positive and negative numbers. Note also the efficiency of the algorithm. Blocks of 1s or 0s are skipped over, with an average of only one addition or subtraction per block.

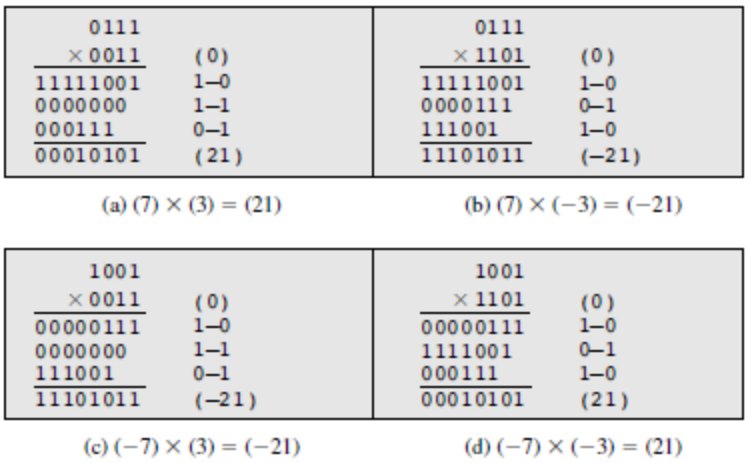


Figure 6.13 Examples Using Booth's Algorithm

6.3.5 Division

Division is somewhat more complex than multiplication but is based on the same general principles. As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.

Figure 6.14 shows an example of the long division of unsigned binary integers. It is instructive to describe the process in detail. First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor; this is referred to as the divisor being able to divide the number. Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a **partial remainder**. From this point on, the division follows a cyclic pattern. At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor. As before, the divisor is subtracted from this number to produce a new partial remainder. The process continues until all the bits of the dividend are exhausted.

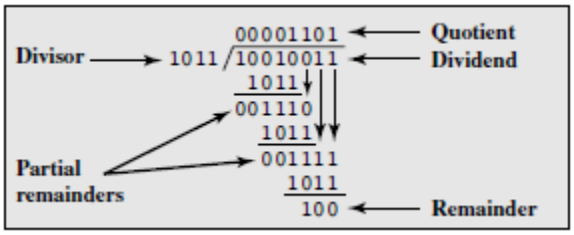


Figure 6.14 Example of Division of Unsigned Binary Integers

Figure 6.15 shows a machine algorithm that corresponds to the long division process. The divisor is placed in the M register, the dividend in the Q register. At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder. If it does, then Q_0 gets a 1 bit. Otherwise, Q_0 gets a 0 bit and M must be added back to A to restore the previous value. The count is then decremented, and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

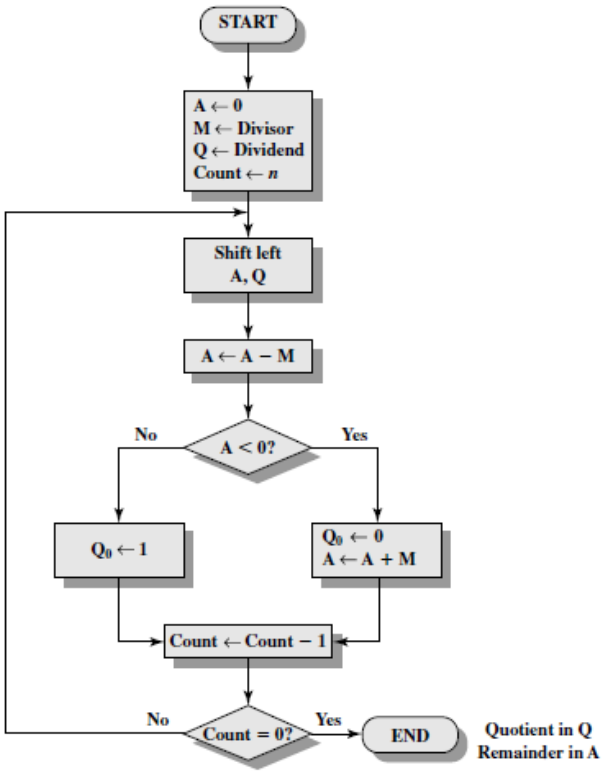


Figure 6.15 Flowchart for Unsigned Binary Division

This process can, with some difficulty, be extended to negative numbers. We give here one approach for twos complement numbers. An example of this approach is shown in Figure 6.16.

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 6.16 Example of Restoring Twos Complement Division (7/3)

The algorithm assumes that the divisor V and the dividend D are positive and that |V|<|D|. If |v|=|D|, then the quotient Q=1 and the remainder R=0. If |v|>|D|, then Q=0 and R=D. The algorithm can be summarized as follows:

1. Load the twos complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a 2n-bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform A ← A – M. This operation subtracts the divisor from the contents of A.
4. a. If the result is nonnegative (most significant bit of A=0), then set Q_0←1.
   b. If the result is negative (most significant bit of A=1), then set Q_0←0 and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

To deal with negative numbers, we recognize that the remainder is defined by This is because the remainder is defined by

$$\mathrm{D} = \mathrm{Q} \times \mathrm{V} + \mathrm{R}$$

Consider the following examples of integer division with all possible combinations of signs of D and V:

$$\begin{array}{llcll} \mathrm{D} = \text{-}7 & \mathrm{V} = \text{-}3 & \rightarrow & \mathrm{Q} = 2 & \mathrm{R} = \text{-}1 \\ \mathrm{D} = \text{-}7 & \mathrm{V} = 3 & \rightarrow & \mathrm{Q} = \text{-}2 & \mathrm{R} = \text{-}1 \\ \mathrm{D} = 7 & \mathrm{V} = \text{-}3 & \rightarrow & \mathrm{Q} = \text{-}2 & \mathrm{R} = 1 \\ \mathrm{D} = 7 & \mathrm{V} = 3 & \rightarrow & \mathrm{Q} = 2 & \mathrm{R} = 1 \end{array}$$

The reader will note from Figure 6.16 that (-7)/(3) and (7)/(-3) produce different remainders. We see that the magnitudes of Q and R are unaffected by the input signs and that the signs of Q and R are easily derivable form the signs of D and V. Specifically, sign(R)=sign(D) and sign(Q)=sign(D)×sign(V). Hence, one way to do twos complement division is to convert the operands into unsigned values and, at the end, to account for the signs by complementation where needed. This is the method of choice for the restoring division algorithm.

6.4 Floating-Point Representation

6.4.1 Principles

With a fixed-point notation (e.g., twos complement) it is possible to represent a range of positive and negative integers centered on 0. By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well.

This approach has limitations. Very large numbers cannot be represented, nor can very small fractions. Furthermore, the fractional part of the quotient in a division of two large numbers could be lost.

For decimal numbers, we get around this limitation by using scientific notation. Thus, 976,000,000,000,000 can be represented as 9.76×10^{14}, and 0.0000000000000976 can be represented as 9.76×10^{-14}. What we have done, in effect, is dynamically to slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point. This allows a range of very large and very small numbers to be represented with only a few digits.

This same approach can be taken with binary numbers. We can represent a number in the form

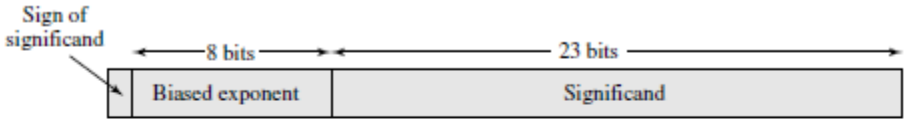
$$\pm S \times B^{\pm E}$$

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E

The **base** B is implicit and need not be stored because it is the same for all numbers. Typically, it is assumed that the radix point is to the right of the leftmost, or most significant, bit of the significand. That is, there is one bit to the left of the radix point.

The principles used in representing binary floating-point numbers are best explained with an example. Figure 6.17a shows a typical 32-bit floating-point format. The leftmost bit stores the sign of the number (0 = positive, 1 = negative). The **exponent** value is stored in the next 8 bits. The representation used is known as a **biased representation**. A fixed value, called the bias, is subtracted from the field to get the true exponent value. Typically, the bias equals $(2^{k-1} - 1)$, where k is the number of bits in the binary exponent. In this case, the 8-bit field yields the numbers 0 through 255. With a bias of 127 $(2^7 - 1)$, the true exponent values are in the range -127 to +128. In this example, the base is assumed to be 2.



(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Figure 6.17 Typical 32-Bit Floating-Point Format

Table 6.2 shows the biased representation for 4-bit integers. Note that when the bits of a biased representation are treated as unsigned integers, the relative magnitudes of the numbers do not change. For example, in both biased and unsigned representations, the largest number is 1111 and the smallest number is 0000.This is not true of sign-magnitude or twos complement representation. An advantage of biased representation is that nonnegative floating-point numbers can be treated as integers for comparison purposes. The final portion of the word (23 bits in this case) is the **significand**.

Any floating-point number can be expressed in many ways.

$$0.0110 * 2^6$$
$$110 * 2^2$$
$$0.110 * 2^5$$

To simplify operations on floating-point numbers, it is typically required that they be normalized. A **normalized number** is one in which the most significant digit of the significand is nonzero. For base 2 representation, a normalized number is therefore one in which the most significant bit of the significand is one. As was mentioned, the typical convention is that there is one bit to the left of the radix point. Thus, a normalized nonzero number is one in the form

$$\pm 1.\text{bbb} \ldots \text{b} \times 2^{\pm \text{E}}$$

where b is either binary digit (0 or 1). Because the most significant bit is always one, it is unnecessary to store this bit; rather, it is implicit. Thus, the 23-bit field is used to store a 24-bit significand with a value in the half open interval [1, 2). Given a number that is not normalized, the number may be normalized by shifting the radix point to the right of the leftmost 1 bit and adjusting the exponent accordingly.

Figure 6.17 gives some examples of numbers stored in this format. For each example, on the left is the binary number; in the center is the corresponding bit pattern; on the right is the decimal value. Note the following features:

- The sign is stored in the first bit of the word.
- The first bit of the true significand is always 1 and need not be stored in the significand field.
- The value 127 is added to the true exponent to be stored in the exponent field.
- The base is 2.

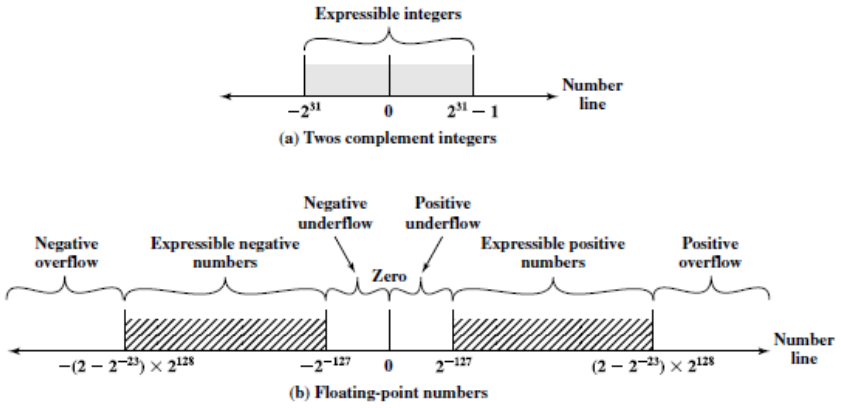


Figure 6.18 Expressible Numbers in Typical 32-Bit Formats

For comparison, Figure 6.18 indicates the range of numbers that can be represented in a 32-bit word. Using twos complement integer representation, all of the integers from -2^{31} to $2^{31}-1$ can be represented, for a total of 2^{32} different numbers. With the example floating-point format of Figure 6.17, the following ranges of numbers are possible:

- Negative numbers between $-(2 - 2^{-23}) \times 2^{128}$ and -2^{-127}.
- Positive numbers between 2^{-127} and $(2 - 2^{-23}) \times 2^{128}$

Five regions on the number line are not included in these ranges:

- Negative numbers less than $-(2 - 2^{-23}) \times 2^{128}$ called **negative overflow**
- Negative numbers greater than -2^{-127} called **negative underflow**
- Zero
- Positive numbers less than 2^{-127} called **positive underflow**
- Positive numbers greater than $(2 - 2^{-23}) \times 2^{128}$ called **positive overflow**

6.4.2 IEEE Standard for Binary Floating-Point Representation

The most important floating-point representation is defined in IEEE Standard 754, adopted in 1985. This standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs. The standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors.

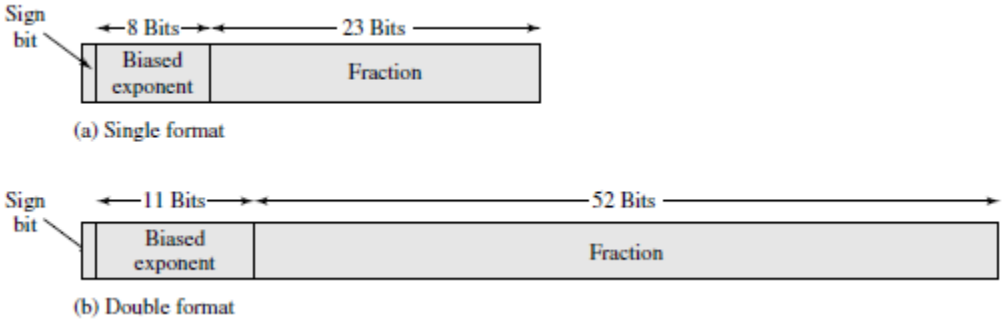


Figure 6.19 IEEE 754 Formats

The IEEE standard defines both a 32-bit single and a 64-bit double format (Figure 6.19), with 8-bit and 11-bit exponents, respectively. The implied base is 2. In addition, the standard defines two extended formats, single and double, whose exact format is implementation dependent. The extended formats include additional bits in the exponent (extended range) and in the significand (extended precision). The extended formats are to be used for intermediate calculations. With their greater precision, the extended formats lessen the chance of a final result that has been contaminated by excessive round-off error; with their greater range, they also lessen the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format. An additional motivation for the single extended format is that it affords some of the benefits of a double format without incurring the time penalty usually associated with higher precision. Table 6.3 summarizes the characteristics of the four formats.

Table 6.3 IEEE 754 Format Parameters

Parameter	Format			
	Single	Single Extended	Double	Double Extended
Word width (bits)	32	≥43	64	≥79
Exponent width (bits)	8	≥11	11	≥15
Exponent bias	127	unspecified	1023	unspecified
Maximum exponent	127	≥1023	1023	≥16383
Minimum exponent	−126	≤−1022	−1022	≤−16382
Number range (base 10)	$10^{-38}, 10^{+38}$	unspecified	$10^{-308}, 10^{+308}$	unspecified
Significand width (bits)*	23	≥31	52	≥63
Number of exponents	254	unspecified	2046	unspecified
Number of fractions	2^{23}	unspecified	2^{52}	unspecified
Number of values	1.98×2^{31}	unspecified	1.99×2^{63}	unspecified

*not including implied bit

6.5 Floating-Point Arithmetic

Table 6.4 summarizes the basic operations for floating-point arithmetic. For addition and subtraction, it is necessary to ensure that both operands have the same exponent value. This may require shifting the radix point on one of the operands to achieve alignment. Multiplication and division are more straightforward.

Table 6.4 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$ $X - Y = (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E}$ } $X_E \le Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_S}{Y_S}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

A floating-point operation may produce one of these conditions:

- **Exponent overflow**: A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as $+\infty$ or $-\infty$.
- **Exponent underflow**: A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.

- **Significand underflow**: In the process of aligning significands, digits may flow off the right end of the significand. As we shall discuss, some form of rounding is required.
- **Significand overflow**: The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we shall explain.

6.5.1 Addition and Subtraction

In floating-point arithmetic, addition and subtraction are more complex than multiplication and division. This is because of the need for alignment. There are four basic phases of the algorithm for addition and subtraction:

1. Check for zeros.
2. Align the significands.
3. Add or subtract the significands.
4. Normalize the result.

typical flowchart is shown in Figure 6.20. A step-by-step narrative highlights the main functions required for floating-point addition and subtraction. We assume a format similar to those of Figure 6.19. For the addition or subtraction operation, the two operands must be transferred to registers that will be used by the ALU. If the floating-point format includes an implicit significand bit, that bit must be made explicit for the operation.

Phase 1: **Zero check**. Because addition and subtraction are identical except for a sign change, the process begins by changing the sign of the subtrahend if it is a subtract operation. Next, if either operand is 0, the other is reported as the result.

Phase 2: **Significand alignment**. The next phase is to manipulate the numbers so that the two exponents are equal.

$$(123 * 10^0) + (456 * 10^{-2}) = (123 * 10^0) + (4.56 * 10^0) = 127.56 * 10^0$$

Alignment may be achieved by shifting either the smaller number to the right (increasing its exponent) or shifting the larger number to the left. Because either operation may result in the loss of digits, it is the smaller number that is shifted; any digits that are lost are therefore of relatively small significance. The alignment is achieved by repeatedly shifting the magnitude portion of the significand right 1 digit and incrementing the exponent until the two exponents are equal. (Note that if the implied base is 16, a shift of 1 digit is a shift of 4 bits.) If this process results in a 0 value for the significand, then the other number is reported as the result. Thus, if two numbers have exponents that differ significantly, the lesser number is lost.

Phase 3: **Addition**. Next, the two significands are added together, taking into account their signs. Because the signs may differ, the result may be 0.There is also the possibility of significand overflow by 1 digit. If so, the significand of the result is shifted right and the exponent is incremented. An exponent overflow could occur as a result; this would be reported and the operation halted.

Phase 4: **Normalization**. The final phase normalizes the result. Normalization consists of shifting significand digits left until the most significant digit (bit, or 4 bits for base-16 exponent) is nonzero. Each shift causes a decrement of the exponent and thus could cause an exponent underflow. Finally, the result must be rounded off and then reported. We defer a discussion of rounding until after a discussion of multiplication and division.

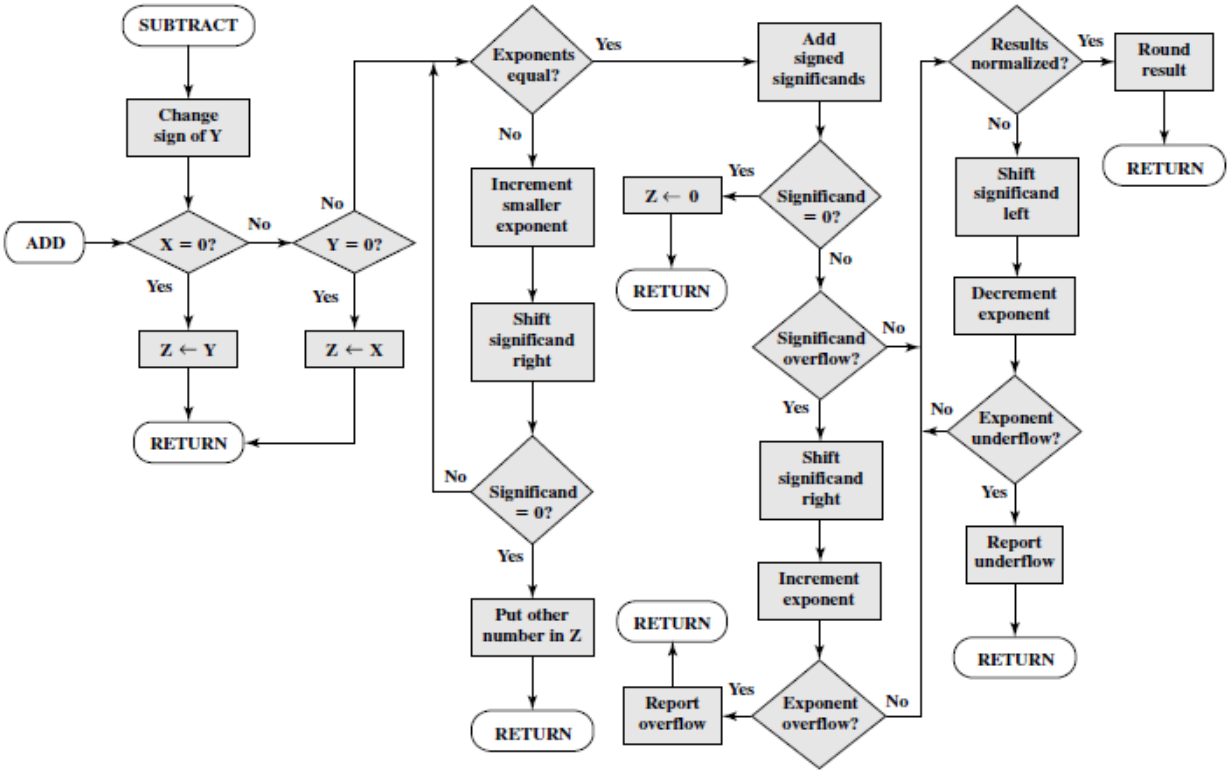


Figure 6.20 Floating-Point Addition and Subtraction (Z ← Z ± Y)

6.5.2 Multiplication and Division

Floating-point multiplication and division are much simpler processes than addition and subtraction, as the following discussion indicates.

We first consider multiplication, illustrated in Figure 9.21. First, if either operand is 0, 0 is reported as the result. The next step is to add the exponents. If the exponents are stored in biased form, the exponent sum would have doubled the bias. Thus, the bias value must be subtracted from the sum. The result could be either an exponent overflow or underflow, which would be reported, ending the algorithm.

If the exponent of the product is within the proper range, the next step is to multiply the significands, taking into account their signs. The multiplication is performed in the same way as for integers. In this case, we are dealing with a sign–magnitude representation, but the details are similar to those for twos complement representation. The product will be double the length of the multiplier and multiplicand. The extra bits will be lost during rounding.

After the product is calculated, the result is then normalized and rounded, as was done for addition and subtraction. Note that normalization could result in exponent underflow.

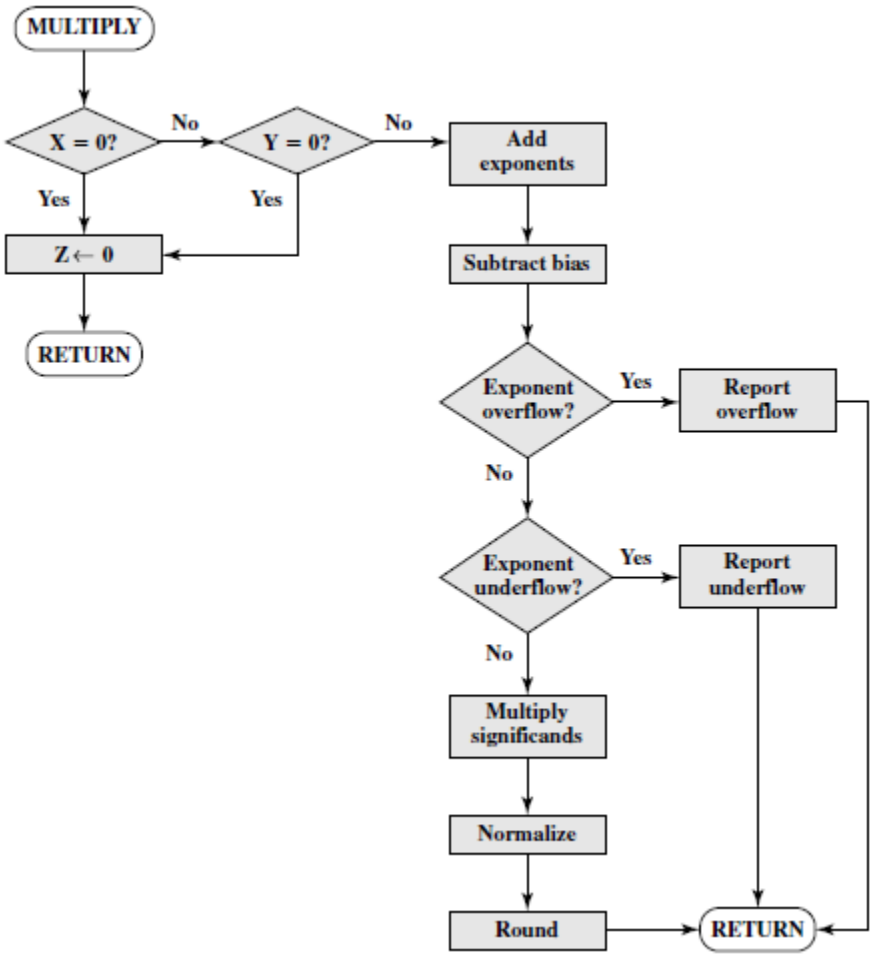


Figure 6.21 Floating-Point Multiplication (Z ← Z × Y)

Finally, let us consider the flowchart for division depicted in Figure 6.22. Again, the first step is testing for 0. If the divisor is 0, an error report is issued, or the result is set to infinity, depending on the implementation. A dividend of 0 results in 0. Next, the divisor exponent is subtracted from the dividend exponent. This removes the bias, which must be added back in. Tests are then made for exponent underflow or overflow.

The next step is to divide the significands. This is followed with the usual normalization and rounding.

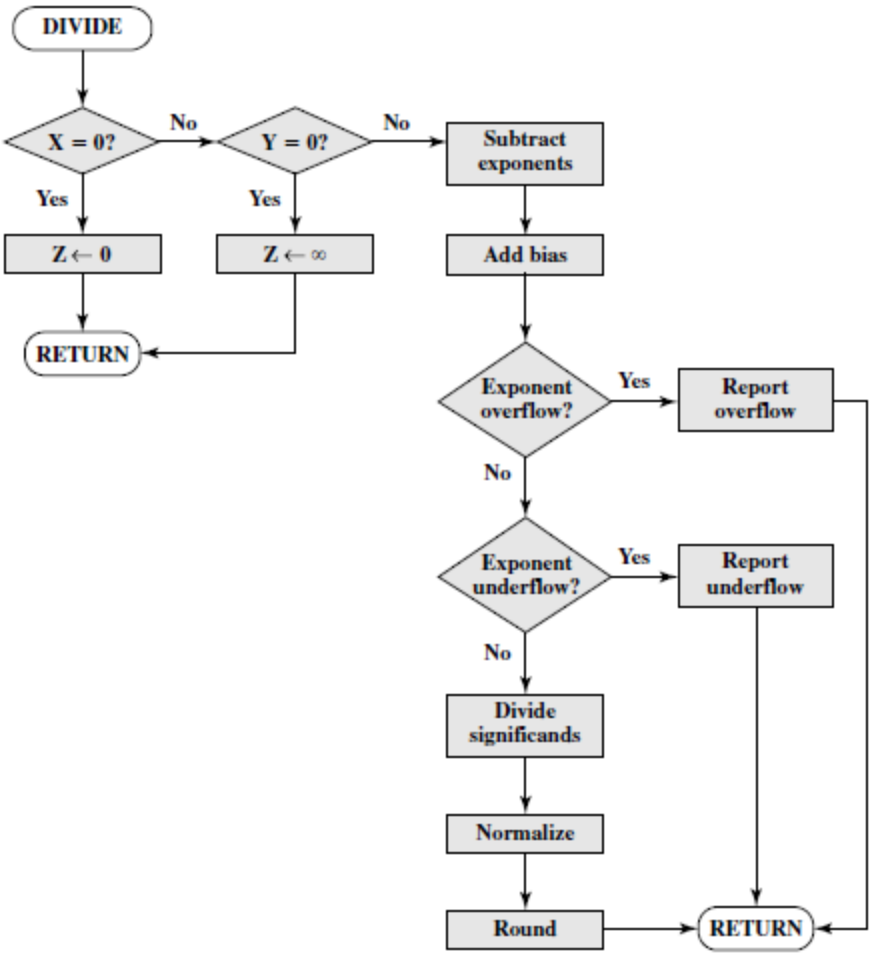


Figure 6.22 Floating-Point Division (Z ← Z / Y)

Chapter 7
Instruction Sets:
Characteristics And Functions

7.1 Machine Instruction Characteristics

The operation of the processor is determined by the instructions it executes, referred to as **machine instructions** or **computer instructions**. The collection of different instructions that the processor can execute is referred to as the processor's *instruction set.*

7.1.1 Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution. Figure 7.1, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction.

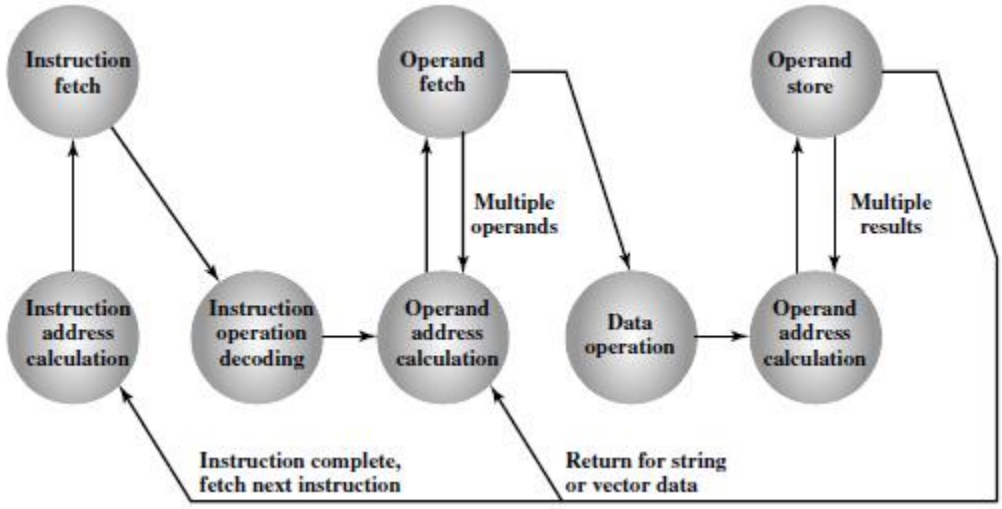


Figure 7.1 Instruction Cycle State Diagram

These elements are as follows:

- **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.
- **Result operand reference:** The operation may produce a result.
- **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture. Generally, the distinction is transparent to the instruction set architecture. In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be supplied.

Source and result operands can be in one of four areas:

- **Main or virtual memory**: As with next instruction references, the main or virtual memory address must be supplied.
- **Processor register**: With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.
- **Immediate**: The value of the operand is contained in a field in the instruction being executed.
- **I/O device**: The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

7.1.2 Instruction Representation

Within the computer, each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. A simple example of an instruction format is shown in Figure 7.2. With most instruction sets, more than one format is used. During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

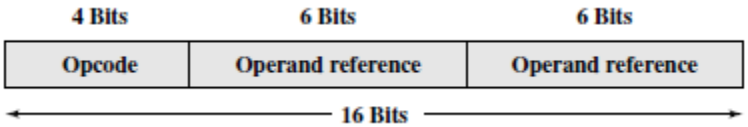


Figure 7.2 A Simple Instruction Format

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a symbolic representation of machine instructions. Common examples include

```
ADD    Add
SUB    Subtract
MUL    Multiply
DIV    Divide
LOAD   Load data from memory
STOR   Store data to memory
```

Operands are also represented symbolically. For example, the instruction

```
ADD R, Y
```

may mean add the value contained in data location Y to the contents of register R. In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

```
X = 513
Y = 514
```

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language. However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

7.1.3 Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

```
X = X + Y
```

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions. This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.

With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language. With this in mind we can categorize instruction types as follows:

- **Data processing:** Arithmetic and logic instructions
- **Data storage:** Movement of data into or out of register and or memory locations
- **Data movement:** I/O instructions
- **Control:** Test and branch instructions

Arithmetic instructions provide computational capabilities for processing numeric data.

Logic (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers.

Therefore, there must be memory instructions for moving data between memory and the registers.

I/O instructions are needed to transfer programs and data into memory and the results of computations back out to the user.

Test instructions are used to test the value of a data word or the status of a computation.

Branch instructions are then used to branch to a different set of instructions depending on the decision made.

7.1.4 Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.

What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could plausibly be required to contain four address references: two source operands, one destination operand, and the address of the next instruction. In most architectures, most instructions have one, two, or three operand addresses, with the address of the next instruction being implicit (obtained from the program counter). Most architectures also have a few special-purpose instructions with more operands.

Figure 7.3 compares typical one-, two-, and three-address instructions that could be used to compute **Y = (A - B)/[C + (D * E)]** . With three addresses, each instruction specifies two source operand locations and a destination operand location. Because we choose not to alter the value of any of the operand locations, a temporary location, T, is used to store some intermediate results. Note that there are four instructions and that the original expression had five operands.

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references. With two-address instructions, and for binary operations, one address must do double duty as both an operand and a result .Thus, the instruction **SUB Y, B** carries out the calculation **Y – B** and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness. To avoid altering

the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Instruction		Comment
SUB	Y, A, B	Y ← A − B
MPY	T, D, E	T ← D × E
ADD	T, T, C	T ← T + C
DIV	Y, Y, T	Y ← Y ÷ T

(a) Three-address instructions

Instruction		Comment
MOVE	Y, A	Y ← A
SUB	Y, B	Y ← Y − B
MOVE	T, D	T ← D
MPY	T, E	T ← T × E
ADD	T, C	T ← T + C
DIV	Y, T	Y ← Y ÷ T

(b) Two-address instructions

Instruction		Comment
LOAD	D	AC ← D
MPY	E	AC ← AC × E
ADD	C	AC ← AC + C
STOR	Y	Y ← AC
LOAD	A	AC ← A
SUB	B	AC ← AC − B
DIV	Y	AC ← AC ÷ Y
STOR	Y	Y ← AC

(c) One-address instructions

Figure 7.3 Programs to Execute Y = (A - B)/[C + (D * E)]

Simpler yet is the one-address instruction. For this to work, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization, called a **stack**. A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers .Thus, zero-address instructions would reference the top two stack elements.

Table 7.1 summarizes the interpretations to be placed on instructions with zero, one, two, or three addresses. In each case in the table, it is assumed that the address of the next instruction is implicit, and that one operation with two source operands and one result operand is to be performed.

Table 7.1 Utilization of Instruction Addresses (Nonbranching Instructions)

Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	A ← B OP C
2	OP A, B	A ← A OP B
1	OP A	AC ← AC OP A
0	OP	T ← (T − 1) OP T

AC = accumulator
T = top of stack
(T − 1) = second element of stack
A, B, C = memory or register locations

The number of addresses per instruction is a basic design decision. Fewer addresses per instruction result in instructions that are more primitive, requiring a less complex processor. It also results in instructions of shorter length. On the other hand, programs contain more total instructions, which in general results in longer execution times and longer, more complex programs. Also, there is an important threshold between one-address and multiple-address instructions. With one-address instructions, the programmer generally has available only one general-purpose register, the accumulator. With multiple-address instructions, it is common to have multiple general-purpose registers. This allows some operations to be performed solely on registers. Because register references are faster than memory references, this speeds up execution. For reasons of flexibility and ability to use multiple registers, most contemporary machines employ a mixture of two- and three-address instructions.

The design trade-offs involved in choosing the number of addresses per instruction are complicated by other factors. There is the issue of whether an address references a memory location or a register. Because there are fewer registers, fewer bits are needed for a register reference. Also, as we shall see in the next chapter, a machine may offer a variety of addressing modes, and the specification of mode takes one or more bits. The result is that most processor designs involve a variety of instruction formats.

7.1.5 Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown. The most important of these fundamental design issues include the following:

- **Operation repertoire**: How many and which operations to provide, and how complex operations should be.
- **Data types**: The various types of data upon which operations are performed
- **Instruction format**: Instruction length (in bits), number of addresses, size of various fields, and so on
- **Registers**: Number of processor registers that can be referenced by instructions, and their use
- **Addressing**: The mode or modes by which the address of an operand is specified

7.2 TYPES OF OPERANDS

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

We shall see, in discussing addressing modes in next chapter, that addresses are, in fact, a form of data. In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address.

In this context, addresses can be considered to be unsigned integers. Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section. Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.

7.2.1 Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses. First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are common in computers:

- Binary integer or binary fixed point
- Binary floating point
- Decimal

Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers. Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is **packed decimal**.

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, 0 = 0000, 1 = 0001, … , 8 = 1000, and 9 = 1001. Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is 0000 0010 0100 0110. This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either

the left or right end of a string of packed decimal digits. Standard sign values are 1100 for positive and 1101 for negative .

Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers.

7.2.2 Characters

A common form of data is text or character strings. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data .Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used character code in the International Reference Alphabet (IRA), referred to in the United States as the **American Standard Code for Information Interchange (ASCII)**. Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent **control characters**. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (odd parity) or always even (even parity).

Another code used to encode characters is the **Extended Binary Coded Decimal Interchange Code (EBCDIC).** EBCDIC is used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal.

7.2.3 Logical Data

Normally, each word or other addressable unit (byte, half-word, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1.When data are viewed this way, they are considered to be logical data.

There are two advantages to the bit-oriented view. First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false).With logical data, memory can be used most efficiently for this storage. Second, there are occasions when we wish to manipulate the bits of a data item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.

Note that, in the preceding examples, the same data are treated sometimes as logical and other times as numerical or text. The “type” of a unit of data is determined by the operation being performed on it. While this is not normally the case in high-level languages, it is almost always the case with machine language.

7.3 TYPES OF OPERATIONS

The number of different opcodes varies widely from machine to machine. However, the same general types of operations are found on all machines. A useful and typical categorization is the following:

- Data transfer
- Arithmetic
- Logical
- Conversion
- I/O
- System control
- Transfer of control

This section provides a brief survey of these various types of operations, together with a brief discussion of the actions taken by the processor to execute a particular type of operation (summarized in Table 7.2).

Table 7.2 Processor Actions for Various Types of Operations

Data Transfer	Transfer data from one location to another
	If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of Control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

7.3.1 Data Transfer

The most fundamental type of machine instruction is the data transfer instruction. The data transfer instruction must specify several things. First, the location of the source and destination operands must be specified. Each location could be memory, a register, or the top of the stack. Second, the length of data to be transferred must be indicated. Third, as with all instructions with operands, the mode of addressing for each operand must be specified. Table 7.3 shows an example of Common Instruction Set Operations for Data Transfer.

In terms of processor action, data transfer operations are perhaps the simplest type. If both source and destination are registers, then the processor simply causes data to be transferred from one register to another; this is an operation internal to the processor. If one or both operands are in memory, then the processor must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode.
2. If the address refers to virtual memory, translate from virtual to real memory address.
3. Determine whether the addressed item is in cache.
4. If not, issue a command to the memory module.

Table 7.3 Common Instruction Set Operations for Data Transfer

Operation Name	Description
Move (transfer)	Transfer word or block from source to Destination
Store	Transfer word from processor to memory
Load (fetch)	Transfer word from memory to processor
Exchange	Swap contents of source and destination
Clear (reset)	Transfer word of 0s to destination
Set	Transfer word of 1s to destination
Push	Transfer word from source to top of stack
Pop	Transfer word from top of stack to Destination

7.3.2 Arithmetic

Most machines provide the basic arithmetic operations of add, subtract, multiply, and divide. These are invariably provided for signed integer (fixed-point) numbers. Often they are also provided for floating-point and packed decimal numbers.

Other possible operations include a variety of single-operand instructions; for example,

- **Absolute:** Take the absolute value of the operand.
- **Negate:** Negate the operand.
- **Increment:** Add 1 to the operand.
- **Decrement:** Subtract 1 from the operand.

Table 7.4 Common Instruction Set for Arithmetic

Operation Name	Description
Add	Compute sum of two operands
Subtract	Compute difference of two operands
Multiply	Compute product of two operands
Divide	Compute quotient of two operands
Absolute	Replace operand by its absolute value
Negate	Change sign of operand
Increment	Add 1 to operand
Decrement	Subtract 1 from operand

The execution of an arithmetic instruction may involve data transfer operations to position operands for input to the ALU, and to deliver the output of the ALU. In addition, of course, the ALU

portion of the processor performs the desired operation. Table 7.4 shows examples of arithmetic instructions.

7.3.3 Logical

Most machines also provide a variety of operations for manipulating individual bits of a word or other addressable units, often referred to as "bit twiddling." They are based upon Boolean operations. Table 7.5 shows some common logic instructions.

Table 7.6 Common Instruction Set for Logic operations

Operation Name	Description
AND	Perform logical AND
OR	Perform logical OR
NOT (complement)	Perform logical NOT
Exclusive-OR	Perform logical XOR
Test	Test specified condition; set flag(s) based on outcome
Compare	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome
Set Control Variables	Class of instructions to set controls for protection purposes, interrupt handling, timer control, etc.
Shift Left (right)	shift operand, introducing constants at end
Rotate Left (right)	shift operand, with wrap around end

Some of the basic logical operations that can be performed on Boolean or binary data. The NOT operation inverts a bit. AND, OR, and Exclusive-OR (XOR) are the most common logical functions with two operands. EQUAL is a useful binary test.

These logical operations can be applied bitwise to n-bit logical data units. Thus, if two registers contain the data

```
(R1) = 10100101
(R2) = 00001111
```

then

```
(R1) AND (R2) = 00000101
```

where the notation (X) means the contents of location X. Thus, the AND operation can be used as a **mask** that selects certain bits in a word and zeros out the remaining bits. As another example, if two registers contain

```
(R1) = 10100101
(R2) = 11111111
```

Then

```
(R1) XOR (R2) = 01011010
```

With one word set to all 1s, the XOR operation inverts all of the bits in the other word (ones complement). In addition to bitwise logical operations, most machines provide a variety of shifting and rotating functions. The most basic operations are illustrated in Figure 7.4. With a **logical shift**, the bits of a word are shifted left or right. On one end, the bit shifted out is lost. On the other end, a 0 is shifted in. Logical shifts are useful primarily for isolating fields within a word. The 0s that are shifted into a word displace unwanted information that is shifted off the other end.

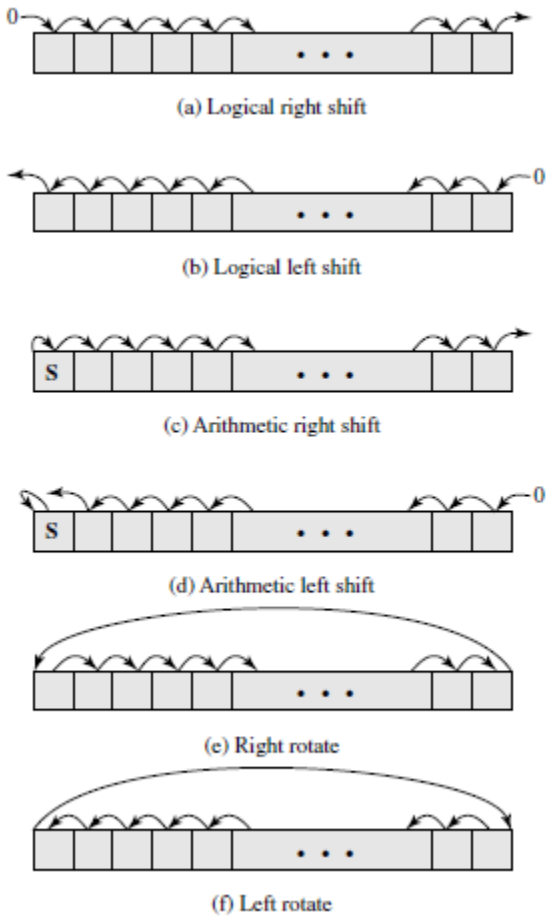


Figure 7.4 Shift and Rotate Operations

As an example, suppose we wish to transmit characters of data to an I/O device 1 character at a time. If each memory word is 16 bits in length and contains two characters, we must **unpack** the characters before they can be sent. To send the two characters in a word,

1. Load the word into a register.
2. Shift to the right eight times. This shifts the remaining character to the right half of the register.
3. Perform I/O. The I/O module reads the lower-order 8 bits from the data bus.

The preceding steps result in sending the left-hand character. To send the right-hand character,

1. Load the word again into the register.

2. AND with 0000000011111111.This masks out the character on the left.
3. Perform I/O.

The **arithmetic shift** operation treats the data as a signed integer and does not shift the sign bit. On a right arithmetic shift, the sign bit is replicated into the bit position to its right. On a left arithmetic shift, a logical left shift is performed on all bits but the sign bit, which is retained. These operations can speed up certain arithmetic operations. With numbers in twos complement notation, a right arithmetic shift corresponds to a division by 2, with truncation for odd numbers. Both an arithmetic left shift and a logical left shift correspond to a multiplication by 2 when there is no overflow. If overflow occurs, arithmetic and logical left shift operations produce different results, but the arithmetic left shift retains the sign of the number. Because of the potential for overflow, many processors do not include this instruction. Curiously, others include an arithmetic left shift but defines it to be identical to a logical left shift.

Rotate, or cyclic shift, operations preserve all of the bits being operated on. One use of a rotate is to bring each bit successively into the leftmost bit, where it can be identified by testing the sign of the data (treated as a number). As with arithmetic operations, logical operations involve ALU activity and may involve data transfer operations. Table 7.7 gives examples of all of the shift and rotate operations discussed in this subsection.

Table 7.7 Examples of Shift and Rotate Operations

Input	Operation	Result
10100110	Logical right shift (3 bits)	00010100
10100110	Logical left shift (3 bits)	00110000
10100110	Arithmetic right shift (3 bits)	11110100
10100110	Arithmetic left shift (3 bits)	10110000
10100110	Right rotate (3 bits)	11010100
10100110	Left rotate (3 bits)	00110101

7.3.4 Conversion

Conversion instructions are those that change the format or operate on the format of data as shown in Table 7.8. An example is converting from decimal to binary. An example of a more complex editing instruction is the EAS/390 Translate (TR) instruction. This instruction can be used to convert from one 8-bit code to another, and it takes three operands:

```
TR R1 (L), R2
```

The operand R2 contains the address of the start of a table of 8-bit codes. The L bytes starting at the address specified in R1 are translated, each byte being replaced by the contents of a table entry indexed by that byte.

For example, to translate from EBCDIC to IRA, we first create a 256-byte table in storage locations, say, 100010FF hexadecimal. The table contains the characters of the IRA code in the sequence of the binary representation of the EBCDIC code; that is, the IRA code is placed in the table at the relative location equal to the binary value of the EBCDIC code of the same character. Thus, locations 10F0 through 10F9 will contain the values 30 through 39, because F0 is the EBCDIC

code for the digit 0, and 30 is the IRA code for the digit 0, and so on through digit 9. Now suppose we have the EBCDIC for the digits 1984 starting at location 2100 and we wish to translate to IRA. Assume

the following:

- Locations 2100–2103 contain F1 F9 F8 F4.
- R1 contains 2100.
- R2 contains 1000.

Then, if we execute

```
TR R1 (4), R2
```

locations 2100–2103 will contain 31 39 38 34.

Table 7.8 Common Instruction Set for Conversion Operations

Operation Name	Description
Translate	Translate values in a section of memory based on a table of correspondences
Convert	Convert the contents of a word from one form to another (e.g., packed decimal to binary)

7.3.5 Input/Output

Many architecture implementations provide only a few I/O instructions, with the specific actions specified by parameters, codes, or command words. Examples of these instructions are shown in Table 7.9

Table 7.9 Common Instruction Set for I/O Operations

Operation Name	Description
Input (read)	Transfer data from specified I/O port or device to destination (e.g., main memory or processor register)
Output (write)	Transfer data from specified source to I/O port or device
Start I/O	Transfer instructions to I/O processor to initiate I/O operation
Test I/O	Transfer status information from I/O system to specified destination

7.3.6 System Control

System control instructions are those that can be executed only while the processor is in a certain privileged state or is executing a program in a special privileged area of memory. Typically, these instructions are reserved for the use of the operating system.

Some examples of system control operations are as follows. A system control instruction may read or alter a control register. Another example is an instruction to read or modify a storage protection key. Another example is access to process control blocks in a multiprogramming system.

7.3.7 Transfer of Control

For all of the operation types discussed so far, the next instruction to be performed is the one that immediately follows, in memory, the current instruction. However, a significant fraction of the instructions in any program have as their function changing the sequence of instruction execution.

For these instructions, the operation performed by the processor is to update the program counter to contain the address of some instruction in memory.

Table 7.10 Common Instruction Set Operations for Transfer of Control

Type	Operation Name	Description
Transfer of Control	Jump (branch)	Unconditional transfer; load PC with specified Address
	Jump Conditional	Test specified condition; either load PC with specified address or do nothing, based on condition
	Jump to Subroutine	Place current program control information in known location; jump to specified address
	Return	Replace contents of PC and other register from known location
	Execute	Fetch operand from specified location and execute as instruction; do not modify PC
	Skip	Increment PC to skip next instruction
	Skip Conditional	Test specified condition; either skip or do nothing based on condition
	Halt	Stop program execution
	Wait (hold)	Stop program execution; test specified condition repeatedly; resume execution when condition is satisfied
	No operation	No operation is performed, but program execution is continued

There are a number of reasons why transfer-of-control operations are required as summery of them is sown in Table 7.10. Among the most important are the following:

1. In the practical use of computers, it is essential to be able to execute each instruction more than once and perhaps many thousands of times. It may require thousands or perhaps millions of instructions to implement an application. This would be unthinkable if each instruction had to be written out separately. If a table or a list of items is to be processed, a program loop is needed. One sequence of instructions is executed repeatedly to process all the data.
2. Virtually all programs involve some decision making. We would like the computer to do one thing if one condition holds, and another thing if another condition holds. For example, a sequence of instructions computes the square root of a number. At the start of the sequence, the sign of the number is tested. If the number is negative, the computation is not performed, but an error condition is reported.
3. To compose correctly a large or even medium-size computer program is an exceedingly difficult task. It helps if there are mechanisms for breaking the task up into smaller pieces that can be worked on one at a time.

We now turn to a discussion of the most common transfer-of-control operations found in instruction sets: branch, skip, and procedure call.

Branch Instructions

A branch instruction, also called a jump instruction, has as one of its operands the address of the next instruction to be executed. Most often, the instruction is a **conditional branch** instruction. That is, the branch is made (update program counter to equal address specified in operand) only if a certain condition is met. Otherwise, the next instruction in sequence is executed (increment program counter as usual). A branch instruction in which the branch is always taken is an **unconditional branch**.

There are two common ways of generating the condition to be tested in a conditional branch instruction. First, most machines provide a 1-bit or multiple-bit condition code that is set as the result of some operations. This code can be thought of as a short user-visible register. As an example, an arithmetic operation (ADD, SUBTRACT, and so on) could set a 2-bit condition code with one of the following four values: 0, positive, negative, overflow. On such a machine, there could be four different conditional branch instructions:

```
BRP X Branch to location X if result is positive.
BRN X Branch to location X if result is negative.
BRZ X Branch to location X if result is zero.
BRO X Branch to location X if overflow occurs.
```

In all of these cases, the result referred to is the result of the most recent operation that set the condition code.

Another approach that can be used with a three-address instruction format is to perform a comparison and specify a branch in the same instruction. For example,

```
BRE R1, R2, X Branch to X if contents of R1 contents of R2.
```

Note that a branch can be either **forward** (an instruction with a higher address) or **backward** (lower address).

Skip Instructions

Another form of transfer-of-control instruction is the skip instruction. The skip instruction includes an implied address. Typically, the skip implies that one instruction be skipped; thus, the implied address equals the address of the next instruction plus one instruction length.

Because the skip instruction does not require a destination address field, it is free to do other things. A typical example is the **increment-and-skip-if-zero (ISZ)** instruction. Consider the following program fragment:

```
301
¦
309 ISZ R1
310 BR 301
311
```

In this fragment, the two transfer-of-control instructions are used to implement an iterative loop. R1 is set with the negative of the number of iterations to be performed. At the end of the loop, R1 is incremented. If it is not 0, the program branches back to the beginning of the loop.

Otherwise, the branch is skipped, and the program continues with the next instruction after the end of the loop.

Procedure Call Instructions

Perhaps the most important innovation in the development of programming languages is the **procedure**. A procedure is a self-contained computer program that is incorporated into a larger program. At any point in the program the procedure may be invoked, or *called.* The processor is instructed to go and execute the entire procedure and then return to the point from which the call took place.

The two principal reasons for the use of procedures are economy and modularity. A procedure allows the same piece of code to be used many times. This is important for economy in programming effort and for making the most efficient use of storage space in the system (the program must be stored). Procedures also allow large programming tasks to be subdivided into smaller units. This use of **modularity** greatly eases the programming task.

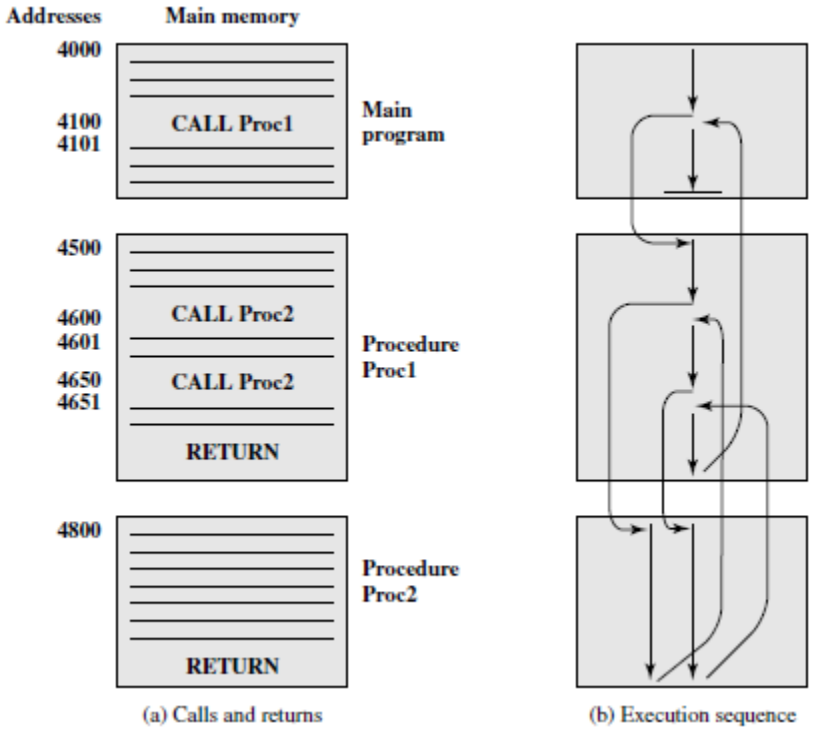


Figure 7.5 Nested Procedures

The procedure mechanism involves two basic instructions: a call instruction that branches from the present location to the procedure, and a return instruction that returns from the procedure to the place from which it was called. Both of these are forms of branching instructions.

Figure 7.5a illustrates the use of procedures to construct a program. In this example, there is a main program starting at location 4000.This program includes a call to procedure PROC1, starting at location 4500. When this call instruction is encountered, the processor suspends execution of the main program and begins execution of PROC1 by fetching the next instruction from location

4500.Within PROC1, there are two calls to PROC2 at location 4800. In each case, the execution of PROC1 is suspended and PROC2 is executed. The RETURN statement causes the processor to go back to the calling program and continue execution at the instruction after the corresponding CALL instruction. This behavior is illustrated in Figure 7.5b.

Three points are worth noting:

1. A procedure can be called from more than one location.
2. A procedure call can appear in a procedure. This allows the **nesting** of procedures to an arbitrary depth.
3. Each procedure call is matched by a return in the called program. Because we would like to be able to call a procedure from a variety of points,

the processor must somehow save the return address so that the return can take place appropriately. There are three common places for storing the return address:

- Register
- Start of called procedure
- Top of stack

Consider a machine-language instruction **CALL X**, which stands for **call procedure at location X**. If the register approach is used, CALL X causes the following actions:

```
RN ← PC + Δ
PC ← X
```

where RN is a register that is always used for this purpose, PC is the program counter, and Δ is the instruction length. The called procedure can now save the contents of RN to be used for the later return.

A second possibility is to store the return address at the start of the procedure. In this case, CALL X causes

```
X ← PC + Δ
PC ← X + 1
```

This is quite handy. The return address has been stored safely away.

Both of the preceding approaches work and have been used. The only limitation of these approaches is that they complicate the use of **reentrant** procedures. A reentrant procedure is one in which it is possible to have several calls open to it at the same time. A recursive procedure (one that calls itself) is an example of the use of this feature. If parameters are passed via registers or memory for a reentrant procedure, some code must be responsible for saving the parameters so that the registers or memory space are available for other procedure calls.

A more general and powerful approach is to use a stack . When the processor executes a call, it places the return address on the stack. When it executes a return, it uses the address on the stack. Figure 7.6 illustrates the use of the stack.

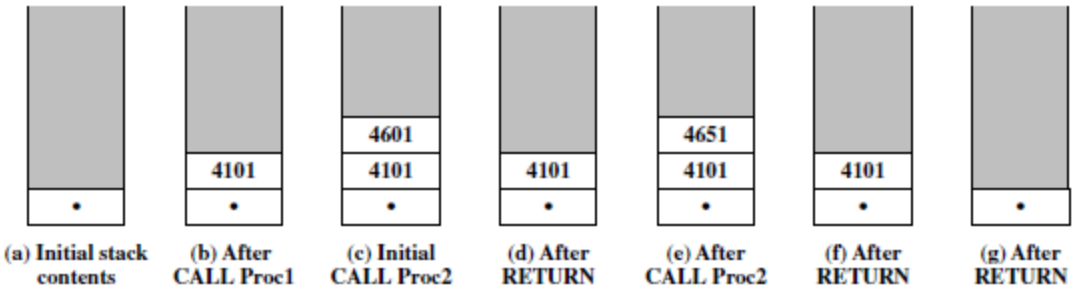


Figure 7.6 Use of Stack to Implement Nested Subroutines of Figure 7.5

In addition to providing a return address, it is also often necessary to pass parameters with a procedure call. These can be passed in registers. Another possibility is to store the parameters in memory just after the CALL instruction. In this case, the return must be to the location following the parameters. Again, both of these approaches have drawbacks. If registers are used, the called program and the calling program must be written to assure that the registers are used properly. The storing of parameters in memory makes it difficult to exchange a variable number of parameters. Both approaches prevent the use of reentrant procedures.

A more flexible approach to parameter passing is the stack. When the processor executes a call, it not only stacks the return address, it stacks parameters to be passed to the called procedure. The called procedure can access the parameters from the stack. Upon return, return parameters can also be placed on the stack. The entire set of parameters, including return address, that is stored for a procedure invocation is referred to as a **stack frame**.

An example is provided in Figure 7.7. The example refers to procedure P in which the local variables x1 and x2 are declared, and procedure Q, which P can call and in which the local variables y1 and y2 are declared. In this figure, the return point for each procedure is the first item stored in the corresponding stack frame. Next is stored a pointer to the beginning of the previous frame. This is needed if the number or length of parameters to be stacked is variable.

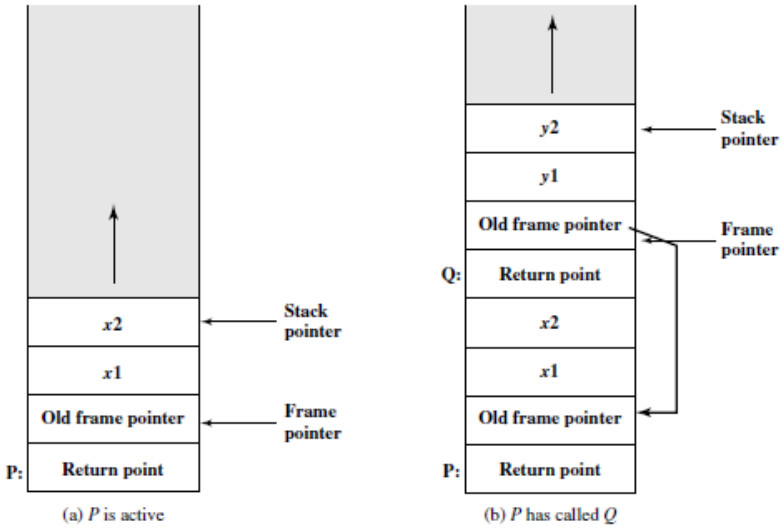


Figure 7.7 Stack Frame Growth Using Sample Procedures P and Q

7.4 Stacks

A **stack** is an ordered set of elements, only one of which can be accessed at a time. The point of access is called the **top** of the stack. The number of elements in the stack, or **length** of the stack, is variable. The last element in the stack is the **base** of the stack. Items may only be added to or deleted from the top of the stack. For this reason, a stack is also known as a **pushdown list** or a **last-in-first-out (LIFO)** list.

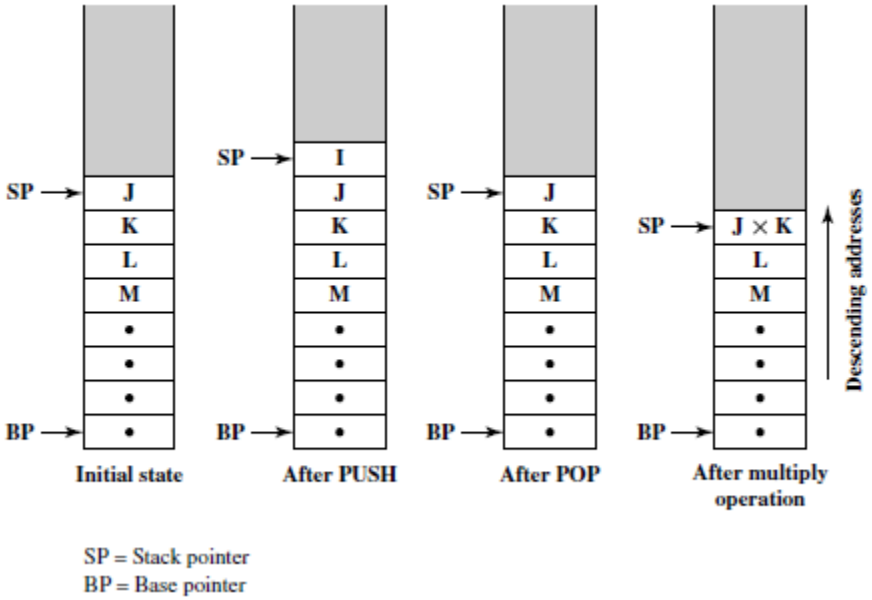


Figure 7.8 Basic Stack Operation (full/descending)

Figure 7.8 shows the basic stack operations .We begin at some point in time when the stack contains some number of elements. A **PUSH** operation appends one new item to the top of the stack. A **POP** operation removes the top item from the stack. In both cases, the top of the stack moves accordingly. Binary operators, which require two operands (e.g., multiply, divide, add, subtract), use the top two stack items as operands, pop both items, and push the result back onto the stack. Unary operations, which require only one operand (e.g., logical NOT), use the item on the top of the stack. All of these operations are summarized in Table 7.11.

Table 7.11 Stack-Oriented Operations

PUSH	Append a new element on the top of the stack.
POP	Delete the top element of the stack.
Unary operation	Perform operation on top element of stack. Replace top element with result.
Binary operation	Perform operation on top two elements of stack. Delete top two elements of stack. Place result of operation on top of stack.

7.4.1 Stack Implementation

The stack is a useful structure to provide as part of a processor implementation. One use, discussed in Section 7.3.7, is to manage procedure calls and returns. Stacks may also be useful to the programmer. An example of this is expression evaluation, discussed later in this section.

The implementation of a stack depends in part on its potential uses. If it is desired to make stack operations available to the programmer, then the instruction set will include stack-oriented operations, including PUSH, POP, and operations that use the top one or two stack elements as operands. Because all of these operations refer to a unique location, namely the top of the stack, the address of the operand or operands is implicit and need not be included in the instruction. These are the zero-address instructions. referred to in Section 7.1.4.

If the stack mechanism is to be used only by the processor, for such purposes as procedure handling, then there will not be explicit stack-oriented instructions in the instruction set. In either case, the implementation of a stack requires that there be some set of locations used to store the stack elements. A typical approach is illustrated in Figure 7.9. A contiguous block of locations is reserved in main memory (or virtual memory) for the stack. Most of the time, the block is partially filled with stack elements and the remainder is available for stack growth.

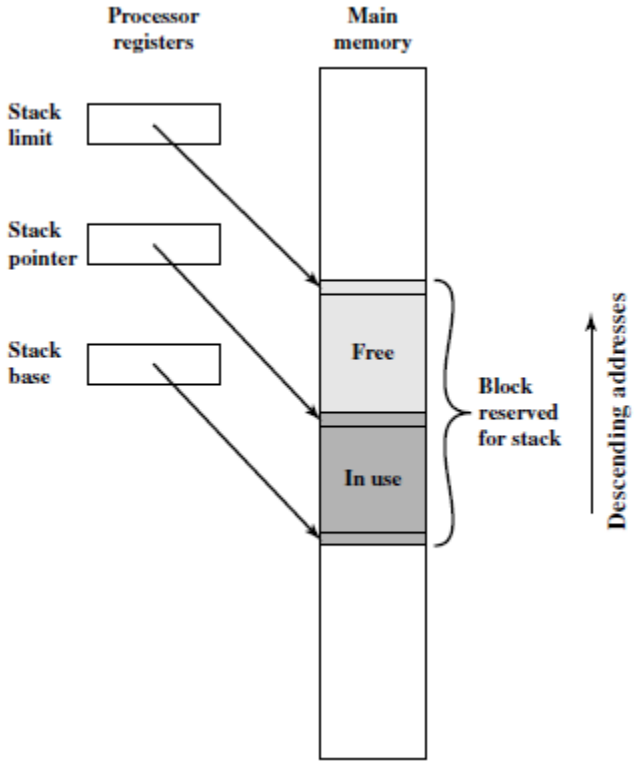


Figure 7.9 Typical Stack Organization (full/descending)

Three addresses are needed for proper operation, and these are often stored in processor registers:

- **Stack pointer (SP):** Contains the address of the top of the stack. If an item is appended to or deleted from the stack, the pointer is incremented or decremented to contain the address of the new top of the stack.
- **Stack base:** Contains the address of the bottom location in the reserved block. If an attempt is made to POP when the stack is empty, an error is reported.

- **Stack limit:** Contains the address of the other end of the reserved block. If an attempt is made to PUSH when the block is fully utilized for the stack, an error is reported. Stack implementations have two key attributes:
- **Ascending/descending:** An ascending stack grows in the direction of ascending addresses, starting from a low address and progressing to a higher address. That is, an ascending stack is one in which the SP is incremented when items are pushed and decremented when items are pulled. A descending stack grows in the direction of descending addresses, starting from a high address and progressing to a lower one. Most machines implement descending stacks as a default.
- **Full/empty:** This is a misleading terminology, because this does not refer to whether the stack is completely full or completely empty. Rather, the SP can either point to the top item in the stack (full method), or the next free space on the stack (an empty method). For the full method, when the stack is completely full, the SP points to the upper limit of the stack. For the empty method, when the stack is completely empty, the SP points to the base of the stack.

Figure 7.8 is an example of a descending/full implementation (assuming that numerically lower addresses are depicted higher on the page).

7.4.2 Expression Evaluation

Mathematical formulas are usually expressed in what is known as **infix** notation. In this form, a binary operator appears between the operands (e.g., $a + b$). For complex expressions, parentheses are used to determine the order of evaluation of expressions. For example, $a + (b \times C)$ will yield a different result than $(a + b) \times C$ To minimize the use of parentheses, operations have an implied precedence. Generally, multiplication takes precedence over addition, so that $a + b \times C$ is equivalent to $a + (b \times C)$.

An alternative technique is known as **reverse Polish**, or **postfix**, notation. In this notation, the operator follows its two operands. For example,

```
a + b              becomes a b +
a + (b × c)        becomes a b c × +
(a + b) × c        becomes a b + c ×
```

Note that, regardless of the complexity of an expression, no parentheses are required when using reverse Polish.

The advantage of postfix notation is that an expression in this form is easily evaluated using a stack. An expression in postfix notation is scanned from left to right. For each element of the expression, the following rules are applied:

1. If the element is a variable or constant, push it onto the stack.
2. If the element is an operator, pop the top two items of the stack, perform the operation, and push the result.

After the entire expression has been scanned, the result is on the top of the stack.

The simplicity of this algorithm makes it a convenient one for evaluating expressions. Accordingly, many compilers will take an expression in a high-level language, convert it to postfix notation, and then generate the machine instructions from that notation. Figure 7.10 shows the sequence of machine instructions for evaluating f = (a - b)/(c + d × e) using stack-oriented instructions.

	Stack	General Registers	Single Register
	Push a Push b Subtract Push c Push d Push e Multiply Add Divide Pop f	Load R1, a Subtract R1, b Load R2, d Multiply R2, e Add R2, c Divide R1, R2 Store R1, f	Load d Multiply e Add c Store f Load a Subtract b Divide f Store f
Number of instructions	10	7	8
Memory access	10 op + 6 d	7 op + 6 d	8 op + 8 d

Figure 7.10 Comparison of Three Programs to Calculate f = (a - b)/(c + d × e)

The figure also shows the use of one-address and two-address instructions. Note that, even though the stack-oriented rules were not used in the last two cases, the postfix notation served as a guide for generating the machine instructions. The sequence of events for the stack program is shown in Figure 7.11.

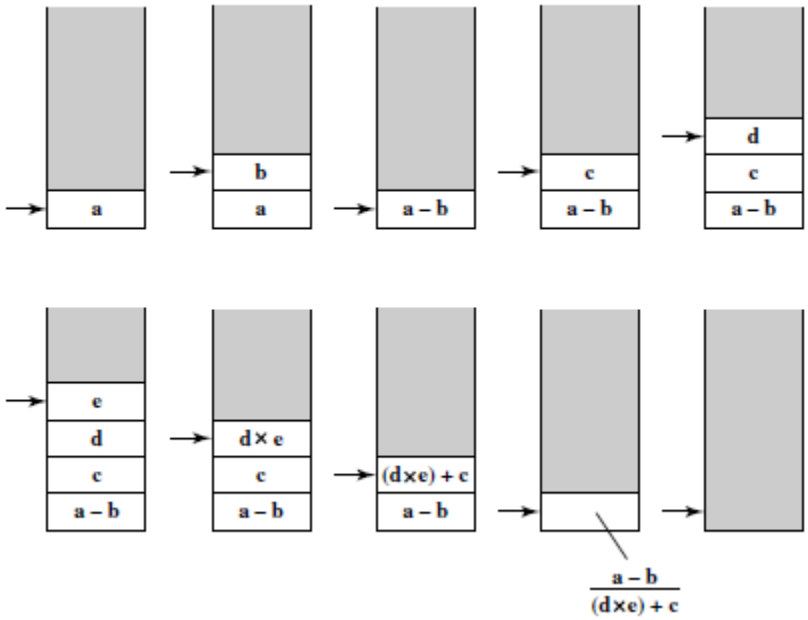


Figure 7.11 Use of Stack to Compute f = (a - b)/(c + d × e)

7.5 LITTLE- , AND BIG-ENDIAN

An annoying and curious phenomenon relates to how the bytes within a word and the bits within a byte are both referenced and represented. We look first at the problem of byte ordering and then consider that of bits.

7.5.1 Byte Ordering

With respect to bytes, endianness has to do with the byte ordering of multi-byte scalar values. The issue is best introduced with an example. Suppose we have the 32-bit hexadecimal value 12345678 and that it is stored in a 32-bit word in byte-addressable memory at byte location 184. The value consists of 4 bytes, with the least significant byte containing the value 78 and the most significant byte containing the value 12. There are two obvious ways to store this value shown in Figure 7.12

Address	Value
184	12
185	34
186	56
187	78

Address	Value
184	78
185	56
186	34
187	12

Figure 7.12 big endian and little endian

The mapping on the left stores the most significant byte in the lowest numerical byte address; this is known as **big endian** and is equivalent to the left-to-right order of writing in Western culture languages. The mapping on the right stores the least significant byte in the lowest numerical byte address; this is known as **little endian** and is reminiscent of the right-to-left order of arithmetic operations in arithmetic units. For a given multi-byte scalar value, big endian and little endian are byte-reversed mappings of each other.

Chapter 8
Instruction Sets:
Addressing Modes And Formats

8.1 ADDRESSING

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

These modes are illustrated in Figure 8.1. In this section, we use the following notation:

```
A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a
    register
EA = actual (effective) address of the location containing the
    referenced operand
(X) = contents of memory location X or register X
```

Table 8.1 indicates the address calculation performed for each addressing mode.

Table 8.1 Basic Addressing Modes

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Before beginning this discussion, two comments need to be made. First, virtually all computer architectures provide more than one of these addressing modes. The question arises as to how the processor can determine which address mode is being used in a particular instruction. Several approaches are taken. Often, different opcodes will use different addressing modes. Also, one or

more bits in the instruction format can be used as a **mode field**. The value of the mode field determines which addressing mode is to be used.

The second comment concerns the interpretation of the effective address (EA). In a system without virtual memory, the **effective address** will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the memory management unit (MMU) and is invisible to the programmer.

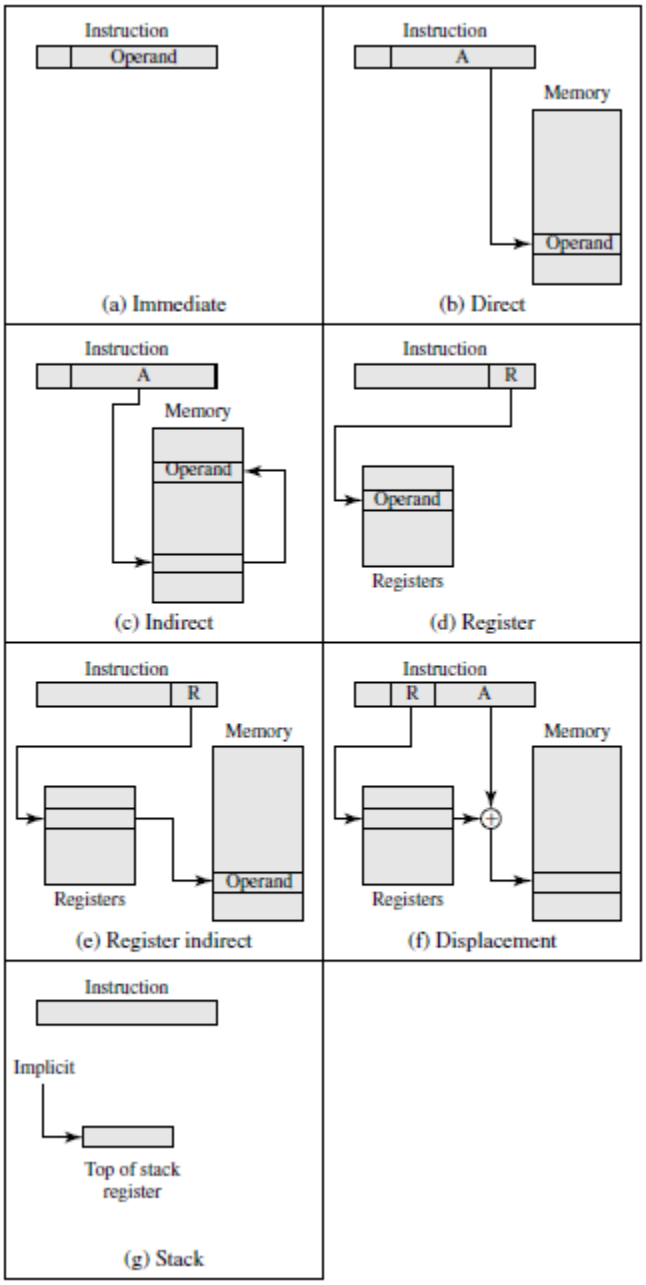


Figure 8.1 Addressing Modes

8.1.1 Immediate Addressing

The simplest form of addressing is immediate addressing, in which the operand value is present in the instruction

```
Operand = A
```

This mode can be used to define and use constants or set initial values of variables. Typically, the number will be stored in twos complement form; the leftmost bit of the operand field is used as a sign bit. When the operand is loaded into a data register, the sign bit is extended to the left to the full data word size. In some cases, the immediate binary value is interpreted as an unsigned nonnegative integer.

The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand, thus saving one memory or cache cycle in the instruction cycle. The disadvantage is that the size of the number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

8.1.2 Direct Addressing

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

```
EA = A
```

The technique was common in earlier generations of computers but is not common on contemporary architectures. It requires only one memory reference and no special calculation. The obvious limitation is that it provides only a limited address space.

8.1.3 Indirect Addressing

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand. This is known as **indirect addressing***:*

```
EA = (A)
```

As defined earlier, the parentheses are to be interpreted as meaning **contents of***.* The obvious advantage of this approach is that for a word length of N, an address space of 2^N is now available. The disadvantage is that instruction execution requires two memory references to fetch the operand: one to get its address and a second to get its value.

8.1.4 Register Addressing

Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address:

EA = R

To clarify, if the contents of a register address field in an instruction is 5, then register R5 is the intended address, and the operand value is contained in R5. Typically, an address field that

references registers will have from 3 to 5 bits, so that a total of from 8 to 32 general-purpose registers can be referenced.

The advantages of register addressing are that (1) only a small address field is needed in the instruction, and (2) no time-consuming memory references are required. As was discussed in Chapter 4, the memory access time for a register internal to the processor is much less than that for a main memory address. The disadvantage of register addressing is that the address space is very limited.

If register addressing is heavily used in an instruction set, this implies that the processor registers will be heavily used. Because of the severely limited number of registers (compared with main memory locations), their use in this fashion makes sense only if they are employed efficiently. If every operand is brought into a register from main memory, operated on once, and then returned to main memory, then a wasteful intermediate step has been added. If, instead, the operand in a register remains in use for multiple operations, then a real savings is achieved.

8.1.5 Register Indirect Addressing

Just as register addressing is analogous to direct addressing, register indirect addressing is analogous to indirect addressing. In both cases, the only difference is whether the address field refers to a memory location or a register. Thus, for register indirect address,

```
EA = (R)
```

The advantages and limitations of register indirect addressing are basically the same as for indirect addressing. In both cases, the address space limitation (limited range of addresses) of the address field is overcome by having that field refer to a word-length location containing an address. In addition, register indirect addressing uses one less memory reference than indirect addressing.

8.1.6 Displacement Addressing

A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing. It is known by a variety of names depending on the context of its use, but the basic mechanism is the same. We will refer to this as **displacement addressing**:

```
EA = A + (R)
```

Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are added to A to produce the effective address.

We will describe three of the most common uses of displacement addressing:

- Relative addressing
- Base-register addressing
- Indexing

8.1.6.1 RELATIVE ADDRESSING

For relative addressing, also called PC-relative addressing, the implicitly referenced register is the program counter (PC). That is, the next instruction address is added to the address field to produce the EA. Typically, the address field is treated as a twos complement number for this operation. Thus, the effective address is a displacement relative to the address of the instruction.

Relative addressing exploits the concept of locality. If most memory references are relatively near to the instruction being executed, then the use of relative addressing saves address bits in the instruction.

8.1.6.2 BASE-REGISTER ADDRESSING

For base-register addressing, the interpretation is the following: The referenced register contains a main memory address, and the address field contains a displacement (usually an unsigned integer representation) from that address. The register reference may be explicit or implicit.

Base-register addressing also exploits the locality of memory references. It is a convenient means of implementing segmentation In some implementations, a single segment-base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. In this latter case, if the length of the address field is K and the number of possible registers is N, then one instruction can reference any one of N areas of 2^K words.

8.1.6.3 INDEXING

For indexing, the interpretation is typically the following: The address field references a main memory address, and the referenced register contains a positive displacement from that address. Note that this usage is just the opposite of the interpretation for base-register addressing. Of course, it is more than just a matter of user interpretation. Because the address field is considered to be a memory address in indexing, it generally contains more bits than an address field in a comparable base-register instruction. Also, we shall see that there are some refinements to indexing that would not be as useful in the base-register context. Nevertheless, the method of calculating the EA is the same for both base-register addressing and indexing, and in both cases the register reference is sometimes explicit and sometimes implicit (for different processor types).

An important use of indexing is to provide an efficient mechanism for performing iterative operations. Consider, for example, a list of numbers stored starting at location A. Suppose that we would like to add 1 to each element on the list. We need to fetch each value, add 1 to it, and store it back. The sequence of effective addresses that we need is A, A+1, A+2 . . ., up to the last location on the list. With indexing, this is easily done. The value A is stored in the instruction's address field, and the chosen register, called an **index register**, is initialized to 0. After each operation, the index register is incremented by 1.

Because index registers are commonly used for such iterative tasks, it is typical That there is a need to increment or decrement the index register after each reference to it. Because this is such a

common operation, some systems will automatically do this as part of the same instruction cycle. This is known as **autoindexing**. If certain registers are devoted exclusively to indexing, then autoindexing can be invoked implicitly and automatically. If general-purpose registers are used, the autoindex operation may need to be signaled by a bit in the instruction. Autoindexing using increment can be depicted as follows.

```
EA = A + (R)
(R) ← (R) + 1
```

In some machines, both indirect addressing and indexing are provided, and it is possible to employ both in the same instruction. There are two possibilities: the indexing is performed either before or after the indirection.

If indexing is performed after the indirection, it is termed **postindexing***:*

```
EA = (A) + (R)
```

First, the contents of the address field are used to access a memory location containing a direct address. This address is then indexed by the register value. This technique is useful for accessing one of a number of blocks of data of a fixed format.

With **preindexing**, the indexing is performed before the indirection:

```
EA = (A + (R))
```

An address is calculated as with simple indexing. In this case, however, the calculated address contains not the operand, but the address of the operand. An example of the use of this technique is to construct a multiway branch table. At a particular point in a program, there may be a branch to one of a number of locations depending on conditions. A table of addresses can be set up starting at location A. By indexing into this table, the required location can be found.

Typically, an instruction set will not include both preindexing and postindexing.

8.1.7 Stack Addressing

The final addressing mode that we consider is stack addressing. A stack is a linear array of locations. It is sometimes referred to as a **pushdown list** or **last-in-first-out queue**. The stack is a reserved block of locations. Items are appended to the top of the stack so that, at any given time, the block is partially filled. Associated with the stack is a pointer whose value is the address of the top of the stack. Alternatively, the top two elements of the stack may be in processor registers, in which case the stack pointer references the third element of the stack. The stack pointer is maintained in a register. Thus, references to stack locations in memory are in fact register indirect addresses.

The stack mode of addressing is a form of implied addressing. The machine instructions need not include a memory reference but implicitly operate on the top of the stack.

8.2 INSTRUCTION FORMATS

An instruction format defines the layout of the bits of an instruction, in terms of its constituent fields. An instruction format must include an opcode and, implicitly or explicitly, zero or more operands. Each explicit operand is referenced using one of the addressing modes described in

Section 8.1. The format must, implicitly or explicitly, indicate the addressing mode for each operand. For most instruction sets, more than one instruction format is used.

8.2.1 Instruction Length

The most basic design issue to be faced is the instruction format length. This decision affects, and is affected by, memory size, memory organization, bus structure, processor complexity, and processor speed. This decision determines the richness and flexibility of the machine as seen by the assembly-language programmer.

The most obvious trade-off here is between the desire for a powerful instruction repertoire and a need to save space. Programmers want more opcodes, more operands, more addressing modes, and greater address range. More opcodes and more operands make life easier for the programmer, because shorter programs can be written to accomplish given tasks. Similarly, more addressing modes give the programmer greater flexibility in implementing certain functions, such as table manipulations and multiple-way branching. And, of course, with the increase in main memory size and the increasing use of virtual memory, programmers want to be able to address larger memory ranges. All of these things (opcodes, operands, addressing modes, address range) require bits and push in the direction of longer instruction lengths. But longer instruction length may be wasteful. A 64-bit instruction occupies twice the space of a 32-bit instruction but is probably less than twice as useful.

Beyond this basic trade-off, there are other considerations. Either the instruction length should be equal to the memory-transfer length (in a bus system, data-bus length) or one should be a multiple of the other. Otherwise, we will not get an integral number of instructions during a fetch cycle. A related consideration is the memory transfer rate. This rate has not kept up with increases in processor speed. Accordingly, memory can become a bottleneck if the processor can execute instructions faster than it can fetch them. One solution to this problem is to use cache memory (see Section 4.3); another is to use shorter instructions. Thus, 16-bit instructions can be fetched at twice the rate of 32-bit instructions but probably can be executed less than twice as rapidly.

A seemingly mundane but nevertheless important feature is that the instruction length should be a multiple of the character length, which is usually 8 bits, and of the length of fixed-point numbers. To see this, we need to make use of that unfortunately ill-defined word, **word**. The word length of memory is, in some sense, the “natural” unit of organization. The size of a word usually determines the size of fixed-point numbers (usually the two are equal). Word size is also typically equal to, or at least integrally related to, the memory transfer size. Because a common form of data is character data, we would like a word to store an integral number of characters. Otherwise, there are wasted bits in each word when storing multiple characters, or a character will have to straddle a word boundary.

8.2.2 Allocation of Bits

We’ve looked at some of the factors that go into deciding the length of the instruction format. An equally difficult issue is how to allocate the bits in that format. The trade-offs here are complex.

For a given instruction length, there is clearly a trade-off between the number of opcodes and the power of the addressing capability. More opcodes obviously mean more bits in the opcode field. For an instruction format of a given length, this reduces the number of bits available for addressing. There is one interesting refinement to this trade-off, and that is the use of variable-length opcodes. In this approach, there is a minimum opcode length but, for some opcodes, additional operations may be specified by using additional bits in the instruction. For a fixed-length instruction, this leaves fewer bits for addressing. Thus, this feature is used for those instructions that require fewer operands and/or less powerful addressing.

The following interrelated factors go into determining the use of the addressing bits.

- **Number of addressing modes:** Sometimes an addressing mode can be indicated implicitly. For example, certain opcodes might always call for indexing. In other cases, the addressing modes must be explicit, and one or more mode bits will be needed.
- **Number of operands:** We have seen that fewer addresses can make for longer, more awkward programs. Typical instructions on today's machines provide for two operands. Each operand address in the instruction might require its own mode indicator, or the use of a mode indicator could be limited to just one of the address fields.
- **Register versus memory:** A machine must have registers so that data can be brought into the processor for processing. With a single user-visible register (usually called the accumulator), one operand address is implicit and consumes no instruction bits. However, single-register programming is awkward and requires many instructions. Even with multiple registers, only a few bits are needed to specify the register. The more that registers can be used for operand references, the fewer bits are needed. A number of studies indicate that a total of 8 to 32 user-visible registers is desirable. Most contemporary architectures have at least 32 registers.
- **Number of register sets:** Most contemporary machines have one set of general-purpose registers, with typically 32 or more registers in the set. These registers can be used to store data and can be used to store addresses for displacement addressing. Some architectures, including that of the x86, have a collection of two or more specialized sets (such as data and displacement). One advantage of this latter approach is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. For example, with two sets of eight registers, only 3 bits are required to identify a register; the opcode or mode register will determine which set of registers is being referenced.
- **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register. Even so, it is still convenient to allow rather large displacements from the register address, which requires a relatively large number of address bits in the instruction.
- **Address granularity:** For addresses that reference memory rather than registers, another factor is the granularity of addressing. In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is

convenient for character manipulation but requires, for a fixed-size memory, more address bits.

Thus, the designer is faced with a host of factors to consider and balance. How critical the various choices are is not clear.

8.2.3 Variable-Length Instructions

The examples we have looked at so far have used a single fixed instruction length, and we have implicitly discussed trade-offs in that context. But the designer may choose instead to provide a variety of instruction formats of different lengths. This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. With variable-length instructions, these many variations can be provided efficiently and compactly.

The principal price to pay for variable-length instructions is an increase in the complexity of the processor. Falling hardware prices, the use of microprogramming, and a general increase in understanding the principles of processor design have all contributed to making this a small price to pay. However, we will see that RISC and superscalar machines can exploit the use of fixed-length instructions to provide improved performance.

The use of variable-length instructions does not remove the desirability of making all of the instruction lengths integrally related to the word length. Because the processor does not know the length of the next instruction to be fetched, a typical strategy is to fetch a number of bytes or words equal to at least the longest possible instruction. This means that sometimes multiple instructions are fetched.

8.3 ASSEMBLY LANGUAGE

A processor can understand and execute machine instructions. Such instructions are simply binary numbers stored in the computer. If a programmer wished to program directly in machine language, then it would be necessary to enter the program as binary data.

Consider the simple BASIC statement

```
N = I + J + K
```

Suppose we wished to program this statement in machine language and to initialize I, J, and K to 2, 3, and 4, respectively. This is shown in Figure 8.2a. The program starts in location 101 (hexadecimal). Memory is reserved for the four variables starting at location 201.The program consists of four instructions:

1. Load the contents of location 201 into the AC.
2. Add the contents of location 202 to the AC.
3. Add the contents of location 203 to the AC.
4. Store the contents of the AC in location 204.

This is clearly a tedious and very error-prone process.

Address	Contents			
101	0010	0010	101	2201
102	0001	0010	102	1202
103	0001	0010	103	1203
104	0011	0010	104	3204
201	0000	0000	201	0002
202	0000	0000	202	0003
203	0000	0000	203	0004
204	0000	0000	204	0000

(a) Binary program

Address	Contents
101	2201
102	1202
103	1203
104	3204
201	0002
202	0003
203	0004
204	0000

(b) Hexadecimal program

Address	Instruction	
101	LDA	201
102	ADD	202
103	ADD	203
104	STA	204
201	DAT	2
202	DAT	3
203	DAT	4
204	DAT	0

(c) Symbolic program

Label	Operation	Operand
FORMUL	LDA	I
	ADD	J
	ADD	K
	STA	N
I	DATA	2
J	DATA	3
K	DATA	4
N	DATA	0

(d) Assembly program

Figure 8.2 Computation of the Formula N = I + J + K

A slight improvement is to write the program in hexadecimal rather than binary notation. We could write the program as a series of lines. Each line contains the address of a memory location and the hexadecimal code of the binary value to be stored in that location. Then we need a program that will accept this input, translate each line into a binary number, and store it in the specified location.

For more improvement, we can make use of the symbolic name or mnemonic of each instruction. This results in the **symbolic program** shown in Figure 8.2c. Each line of input still represents one memory location. Each line consists of three fields, separated by spaces. The first field contains the address of a location. For an instruction, the second field contains the three-letter symbol for the opcode. If it is a memory-referencing instruction, then a third field contains the address. To store arbitrary data in a location, we invent a **pseudoinstruction** with the symbol DAT. This is merely an indication that the third field on the line contains a hexadecimal number to be stored in the location specified in the first field.

For this type of input we need a slightly more complex program. The program accepts each line of input, generates a binary number based on the second and third (if present) fields, and stores it in the location specified by the first field.

The use of a symbolic program makes life much easier but is still awkward. In particular, we must give an absolute address for each word. This means that the program and data can be loaded into only one place in memory, and we must know that place ahead of time. Worse, suppose we wish to change the program someday by adding or deleting a line. This will change the addresses of all subsequent words.

A much better system, and one commonly used, is to use symbolic addresses. This is illustrated in Figure 8.2d. Each line still consists of three fields. The first field is still for the address, but a symbol is used instead of an absolute numerical address. Some lines have no address, implying that the address of that line is one more than the address of the previous line. For memory-reference instructions, the third field also contains a symbolic address.

With this last refinement, we have an **assembly language***.* Programs written in assembly language (assembly programs) are translated into machine language by an *assembler.* This program must not only do the symbolic translation discussed earlier but also assign some form of memory addresses to symbolic addresses.

The development of assembly language was a major milestone in the evolution of computer technology. It was the first step to the high-level languages in use today. Although few programmers use assembly language, virtually all machines provide one. They are used, if at all, for systems programs such as compilers and I/O routines.