

Course Outline

Course Title: Digital Electronics-II
Course Code: (Phy-21306) 2 + 1 Credit Hours
Class: M. Sc. Semester-IV & BS Semester VIII
Convener: Prof. Dr. Sheikh Aftab Ahmad

Major Topics	
8-1 Shift Register Operations 8-2 Types of Shift Register Data I/O 8-3 Bidirectional Shift Registers 8-4 Shift Register Counters 8-5 Shift Register Applications 9-2 Asynchronous Counters 9-3 Synchronous Counters 9-4 Up/Down Synchronous Counters 9-5 Design of Synchronous Counters 9-6 Cascaded Counters 9-7 Counter Decoding 9-8 Counter Applications 11-1 Semiconductor Memory Basics 11-2 The Random-Access Memory (RAM) 11-3 The Read-Only Memory (ROM) 11-4 Programmable ROMs 11-5 The Flash Memory 11-6 Memory Expansion 11-7 Special Types of Memories 11-8 Magnetic and Optical Storage	12-1 Analog-to-Digital Conversion 12-2 Methods of Analog-to-Digital Conversion 12-3 Methods of Digital-to-Analog Conversion 12-4 Digital Signal Processing 14-1 The Computer System 14-2 Practical Computer System Considerations 14-3 The Processor: Basic Operation 14-4 The Processor: Addressing Modes 14-5 The Processor: Special Operations 14-6 Operating Systems and Hardware 14-7 Programming Introducing embedded systems and the microcontroller Embedded systems and their characteristics The microprocessor reviewed Some microprocessor design options The microcontroller: its applications and environment

Text Book

Digital Fundamentals (11th Edition) by Thomas L. Floyd

Reference Books

Logic and Computer Design Fundamentals 3rd Edition by Mano and Kime, Pearson Prentice Hall, (ISBN: 0-13-140539-X)

Digital Computer Fundamentals by T. C. Bartee, McGraw-Hill

Digital Electronics, 5th Edition by R. L. Tokheim, McGraw-Hill (ISBN 0-07-116796-)

Assignments: You will allocated and asked to submit five assignments on the following contact.

Contact: Professor (R) Sheikh Aftab Ahmad via email draftab2@yahoo.com

Lab Couse

It includes Experiments 10 to 17 of the Lab. Manual.

12 - Signal Conversion and Processing

INTRODUCTION

This chapter provides an introduction to interfacing digital and analog systems using methods of analog-to-digital and digital-to-analog conversions. Digital signal processing is a technology that is widely used in many applications, such as automotive, consumer, graphics/imaging, industrial, instrumentation, medical, military, telecommunications, and voice/speech applications. Digital signal processing incorporates mathematics, software programming, and processing hardware to manipulate analog signals.

12-1 Analog-to-Digital Conversion:

In order to process signals using digital techniques, the incoming analog signal must be converted into digital form.

Sampling and Filtering

The sample-and-hold function does two operations, the first of which is sampling. **Sampling** is the process of taking a sufficient number of discrete values at points on a waveform that will define the shape of the waveform. The more samples you take, the more accurately you can define a waveform.

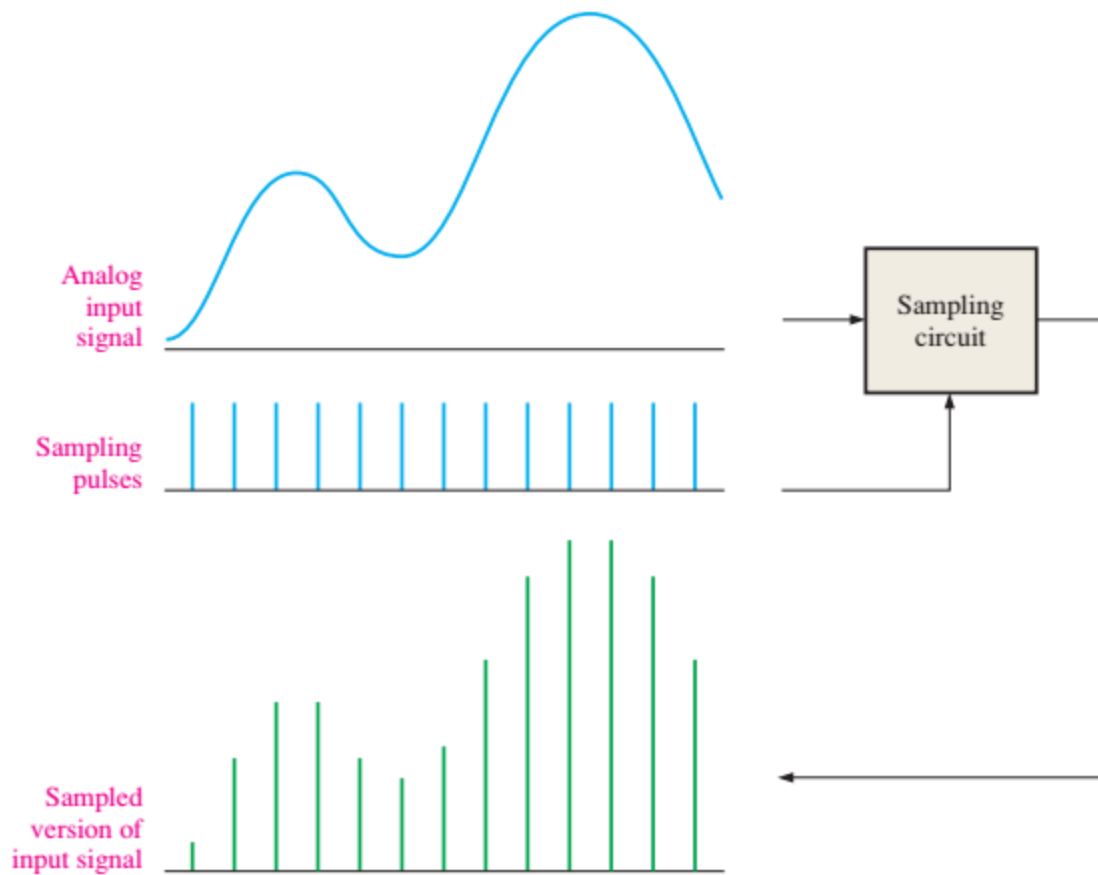


FIGURE 12-1 Illustration of the sampling process.

All analog signals (except a pure sine wave) contain a spectrum of component frequencies. For a pure sine wave, these frequencies appear in multiples called *harmonics*. The harmonics of an analog signal are sine waves of different frequencies and amplitudes. When the harmonics of a given periodic waveform are added, the result is the original signal. Before a signal can be sampled, it must be passed through a low-pass filter (anti-aliasing filter) to eliminate harmonic frequencies above a certain value as determined by the **Nyquist frequency**.

The Sampling Theorem

In order to represent an analog signal, the sampling frequency, f_{sample} , must be at least twice the highest frequency component $f_a(\text{max})$ of the analog signal. The frequency $f_a(\text{max})$ is known as the **Nyquist frequency**. In practice, the sampling frequency should be more than twice the highest analog frequency.

$$f_{\text{sample}} > 2f_{a(\text{max})}$$

Equation 12-1

To intuitively understand the sampling theorem, a simple “bouncing-ball” analogy may be helpful. If you take four photos, as shown in part (c), then the path that the

ball follows during a bounce begins to emerge. The more photos (samples) that you take, the more accurately you can determine the path of the ball as it bounces.

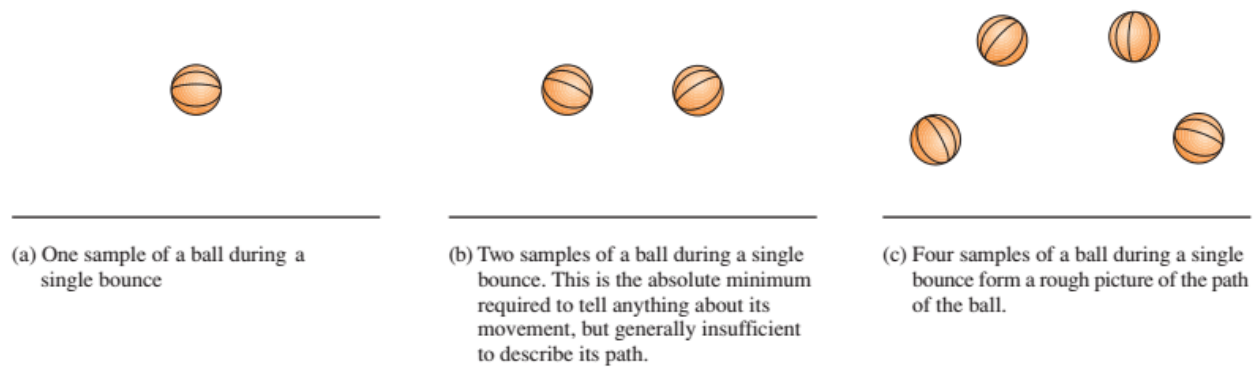


FIGURE 12-2 Bouncing ball analogy of sampling theory.

The Need for Filtering

Low-pass filtering is necessary to remove all frequency components (harmonics) of the analog signal that exceed the Nyquist frequency.

Another way to view aliasing is by considering that the sampling pulses produce a spectrum of harmonic frequencies above and below the sample frequency, as shown in Figure 12-3. If the analog signal contains frequencies above the Nyquist frequency, these frequencies overlap into the spectrum of the sample waveform as shown and interference occurs.

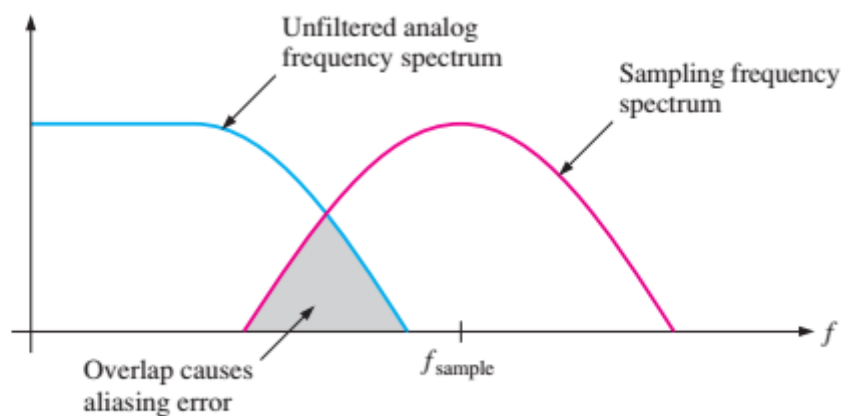


FIGURE 12-3 A basic illustration of the condition $f_{\text{sample}} < 2f_{a(\text{max})}$.

A low-pass anti-aliasing filter must be used to limit the frequency spectrum of the analog signal for a given sample frequency. To avoid an aliasing error, the filter must at least eliminate all analog frequencies above the minimum frequency in the sampling spectrum, as illustrated in Figure 12-4.

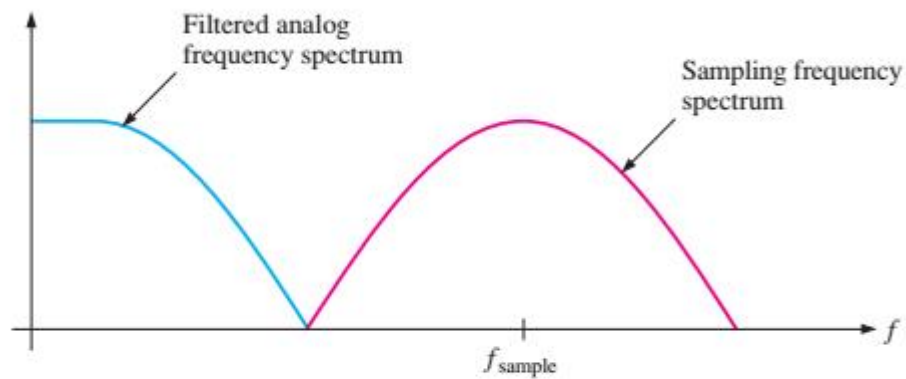


FIGURE 12-4 After low-pass filtering, the frequency spectra of the analog and the sampling signals do not overlap, thus eliminating aliasing error.

An Application

An example of the application of sampling is in digital audio equipment. The sampling rates used are 32 kHz, 44.1 kHz, or 48 kHz (the number of samples per second). The 48 kHz rate is the most common, but the 44.1 kHz rate is used for audio CDs and prerecorded tapes. According to the Nyquist rate, the sampling frequency must be at least twice the audio signal. Therefore, the CD sampling rate of 44.1 kHz captures frequencies up to about 22 kHz, which exceeds the 20 kHz specification that is common for most audio equipment.

Holding the Sampled Value

The holding operation is the second part of the sample-and-hold function. After filtering and sampling, the sampled level must be held constant until the next sample occurs. This is necessary for the ADC to have time to process the sampled value. This sample-and-hold operation results in a “stairstep” waveform that approximates the analog input waveform, as shown in Figure 12-5.

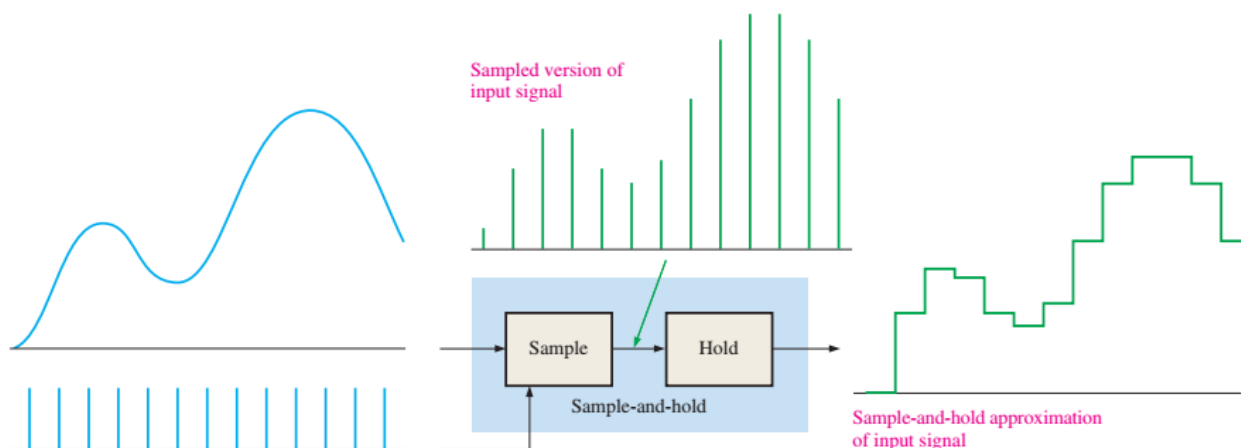


FIGURE 12-5 Illustration of a sample-and-hold operation.

Analog-to-Digital Conversion

Analog-to-digital conversion is the process of converting the output of the sample and-hold circuit to a series of binary codes that represent the amplitude of the analog input at each of the sample times. The sample-and-hold process keeps the amplitude of the analog input signal constant between sample pulses

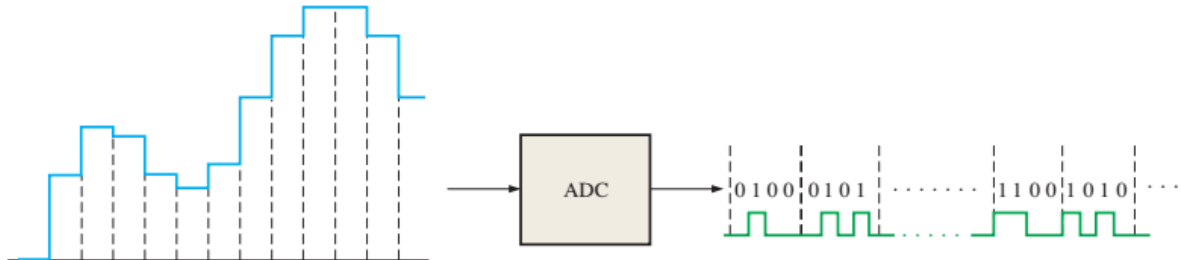


FIGURE 12-6 Basic function of an analog-to-digital converter (ADC) (The binary codes and number of bits are arbitrarily chosen for illustration only). The ADC output waveform that represents the binary codes is also shown.

Quantization

The process of converting an analog value to a code is called **quantization**. During the quantization process, the ADC converts each sampled value of the analog signal to a binary code. The more bits that are used to represent a sampled value, the more accurate is the representation.

As shown in Figure 12-7, each quantization level is represented by a 2-bit code on the vertical axis, and each sample interval is numbered along the horizontal axis.

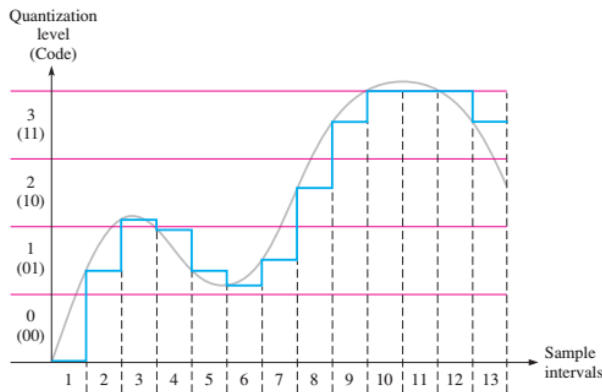


FIGURE 12-7 Sample-and-hold output waveform with four quantization levels. The original analog waveform is shown in light gray for reference.

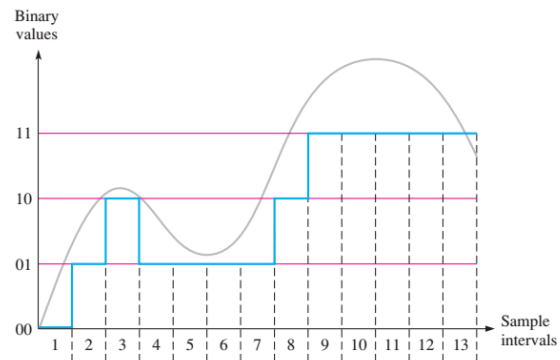


FIGURE 12-8 The reconstructed waveform in Figure 12-7 using four quantization levels (2 bits). The original analog waveform is shown in light gray for reference.

TABLE 12-1

Two-bit quantization for the waveform in Figure 12-7.

Sample Interval	Quantization Level	Code
1	0	00
2	1	01
3	2	10
4	1	01
5	1	01
6	1	01
7	1	01
8	2	10
9	3	11
10	3	11
11	3	11
12	3	11
13	3	11

As you can see, quite a bit of accuracy is lost using only two bits to represent the sampled values. Figure 12-9 shows the same waveform with sixteen quantization levels (4 bits). The 4-bit quantization process is summarized in Table 12-2. If the resulting 4-bit digital codes are used to reconstruct the original waveform, you would get the waveform shown in Figure 12-10. As you can see, the result is much more like the original waveform than for the case of four quantization levels in Figure 12-8. This shows that greater accuracy is achieved with more quantization bits. Typical integrated circuit ADCs use from 12 to 24 bits, and the sample-and-hold function is sometimes contained on the ADC chip.

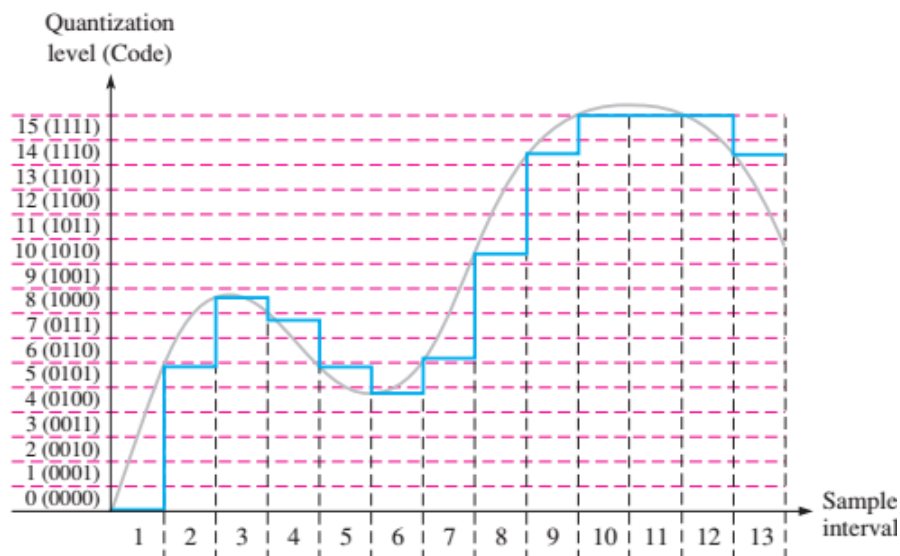


FIGURE 12-9 Sample-and-hold output waveform with sixteen quantization levels. The original analog waveform is shown in light gray for reference.

TABLE 12-2

Four-bit quantization for the waveform in Figure 12-9.

Sample Interval	Quantization Level	Code
1	0	0000
2	5	0101
3	8	1000
4	7	0111
5	5	0101
6	4	0100
7	6	0110
8	10	1010
9	14	1110
10	15	1111
11	15	1111
12	15	1111
13	14	1110

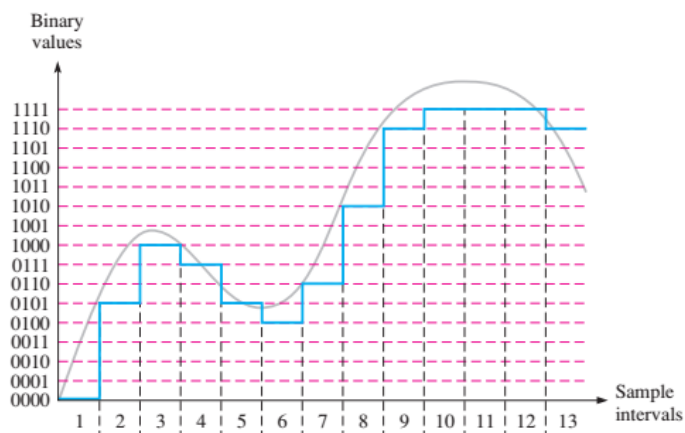


FIGURE 12-10 The reconstructed waveform in Figure 12-9 using sixteen quantization levels (4 bits). The original analog waveform is shown in light gray for reference.

A Quick Look at an Operational Amplifier

The operational amplifier is common to most types of ADCs and digital-to-analog converters (DACs). An **op-amp** is a linear amplifier that has two inputs (inverting and non-inverting) and one output. It has a very high voltage gain and very high input impedance, as well as very low output impedance. The op-amp symbol is shown in Figure 12-11(a).

$$\frac{V_{out}}{V_{in}} = -\frac{R_f}{R_i} \quad \text{Equation 12-2}$$

In the inverting amplifier configuration, the inverting input of the op-amp is approximately at ground potential (0 V) because feedback and the extremely high open-loop gain make the differential voltage between the two inputs extremely small. Since the non-inverting input is grounded, the inverting input is at approximately 0 V, which is called *virtual ground*. When the op-amp is used as a **comparator**, as shown in Figure 12-11(c), two voltages are applied to the inputs. When these input voltages differ by a very small amount, the op-amp is driven into one of its two saturated output states, either HIGH or LOW, depending on which input voltage is greater.

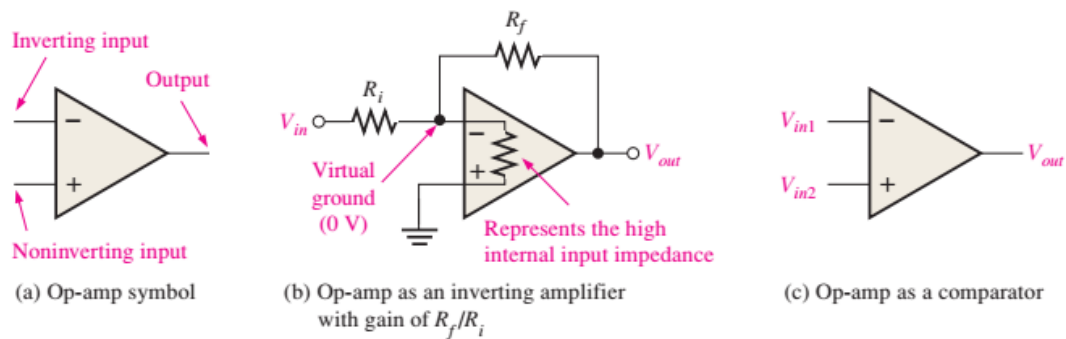


FIGURE 12-11 The operational amplifier (op-amp).

12-2 Methods of Analog-to-Digital Conversion:

As you have seen, analog-to-digital conversion is the process by which an analog quantity is converted to digital form. It is necessary when measured quantities must be in digital form for processing or for display or storage.

Flash (Simultaneous) Analog-to-Digital Converter

The flash method utilizes special high-speed comparators that compare reference voltages with the analog input voltage. When the input voltage exceeds the reference voltage for a given comparator, a HIGH is generated. Figure 12-12 shows a 3-bit converter that uses seven comparator circuits; a comparator is not needed for the all-0s condition. A 4-bit converter of this type requires fifteen comparators. In general, $(2n - 1)$ comparators are required for conversion to an n -bit binary code. The number of bits used in an ADC is its **resolution**. The large number of comparators necessary for a reasonable-sized binary number is one of the disadvantages of the **flash ADC**. Its chief advantage is that it provides a fast conversion time because of a high *throughput*, measured in samples per second.

The reference voltage for each comparator is set by the resistive voltage-divider circuit. The output of each comparator is connected to an input of the priority encoder. The encoder is enabled by a pulse on the EN input, and a 3-bit code representing the value of the input appears on the encoder's outputs. The binary code is determined by the highest-order input having a HIGH level.

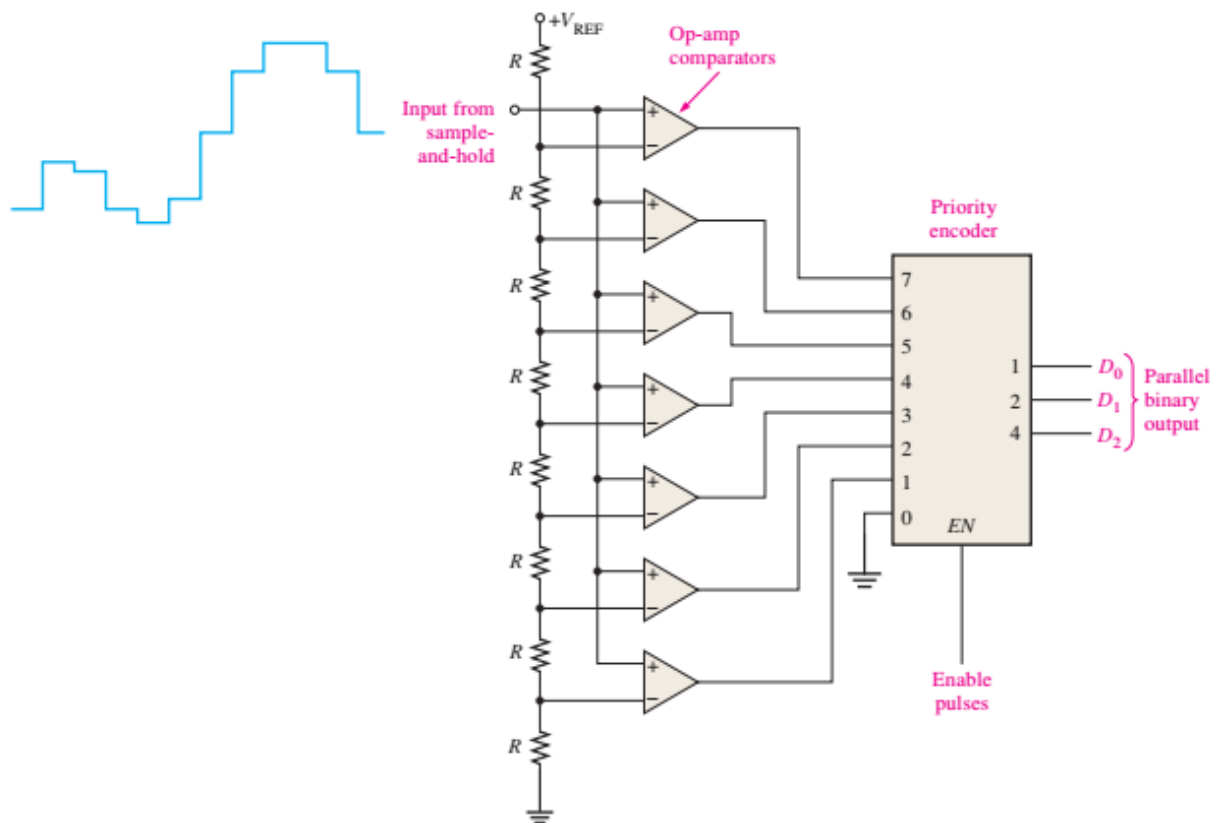


FIGURE 12-12 A 3-bit flash ADC.

The frequency of the enable pulses and the number of bits in the binary code determine the accuracy with which the sequence of binary codes represents the input of the ADC. The signal is sampled each time the enable pulse is active.

EXAMPLE 12-1

Determine the binary code output of the 3-bit flash ADC in Figure 12-12 for the input signal in Figure 12-13 and the encoder enable pulses shown. For this example, $V_{REF} = +8$ V.

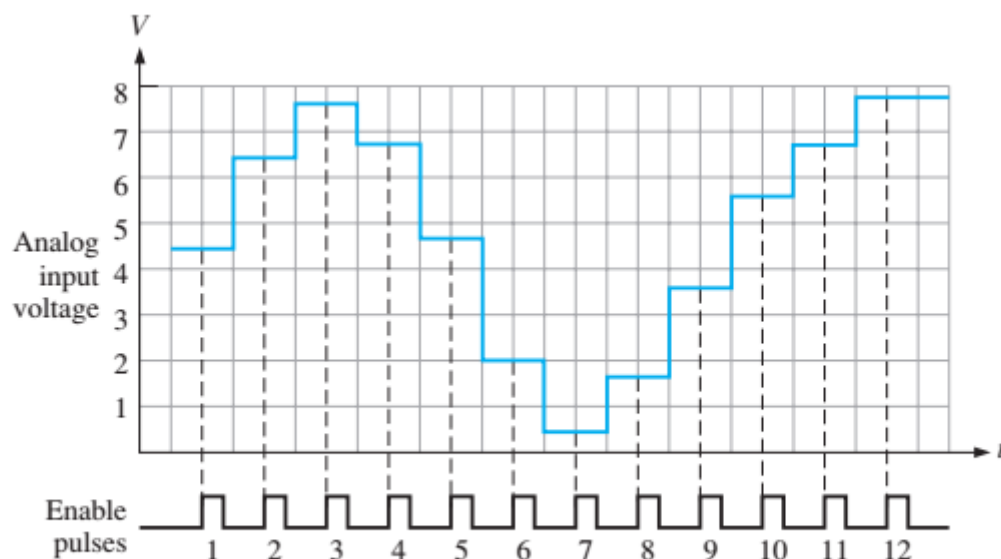


FIGURE 12-13 Sampling of values on a waveform for conversion to binary code.

Solution

The resulting digital output sequence is listed as follows and shown in the waveform diagram of Figure 12–14 in relation to the enable pulses:

100, 110, 111, 110, 100, 010, 000, 001, 011, 101, 110, 111

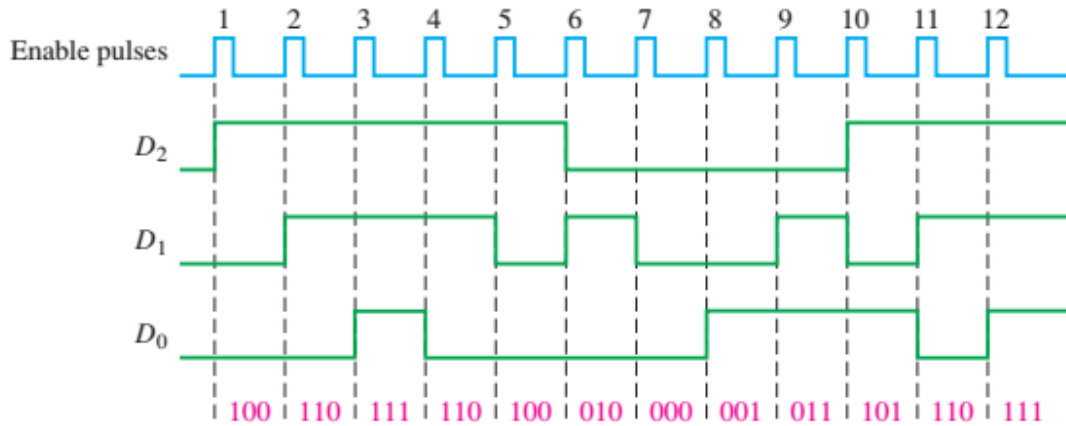


FIGURE 12–14 Resulting digital outputs for sample-and-hold values. Output D_0 is the LSB of the 3-bit binary code.

Dual-Slope Analog-to-Digital Converter

A dual-slope ADC is common in digital voltmeters and other types of measurement instruments. A ramp generator (integrator) is used to produce the dual-slope characteristic. A block diagram of a dual-slope ADC is shown in Figure 12–15.

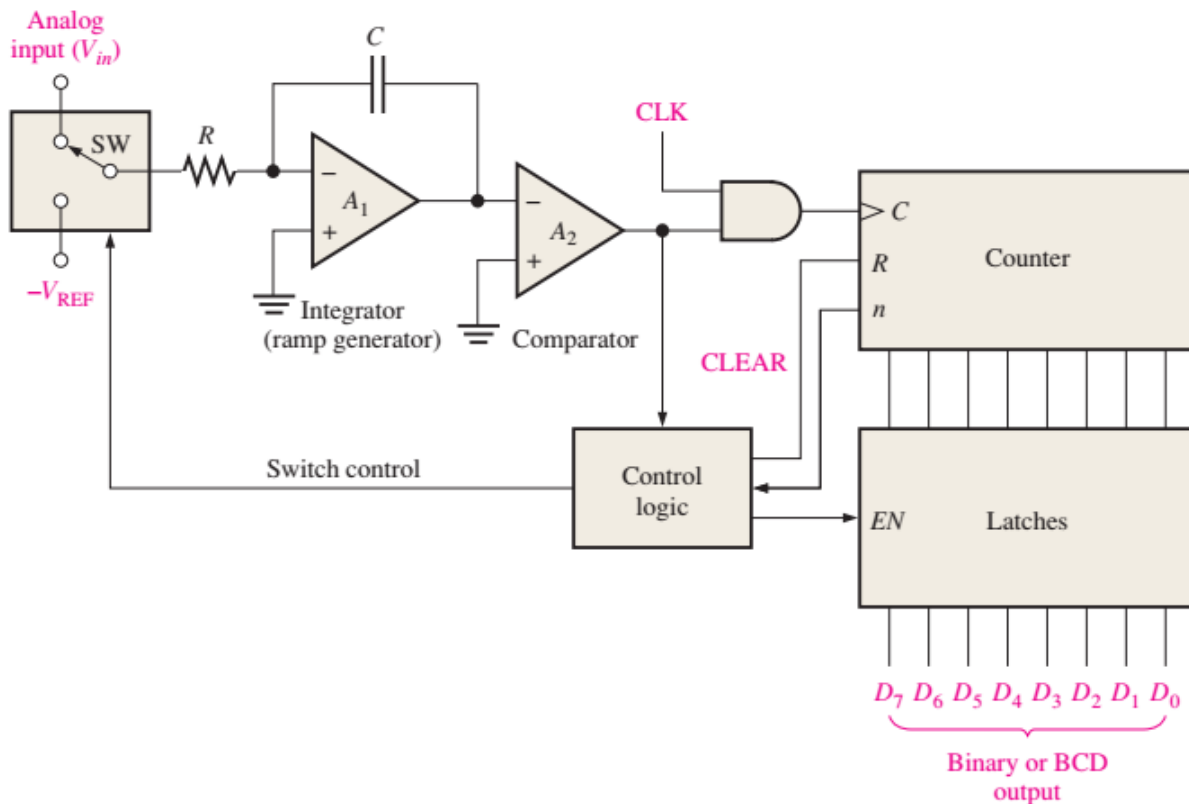
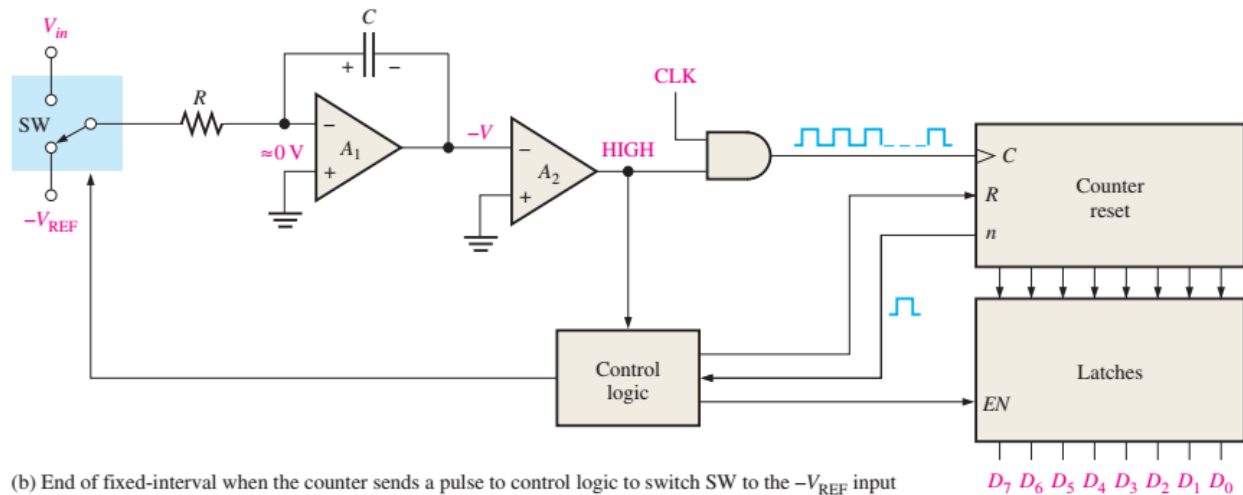
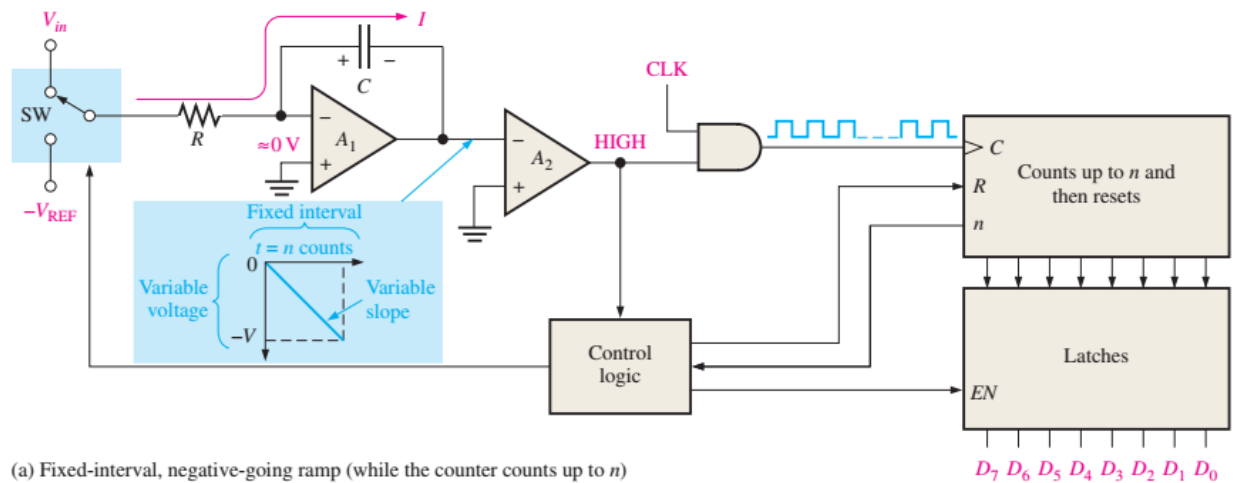


FIGURE 12–15 Basic dual-slope ADC.

Figure 12–16 illustrates dual-slope conversion. Start by assuming that the counter is reset and the output of the integrator is zero. Now assume that a positive input voltage is applied to the input through the switch (SW) as selected by the control logic. Since the inverting input of A_1 is at virtual ground, and assuming that V_{in} is constant for a period of time, there will be constant current through the input resistor R and therefore through the capacitor C . Capacitor C will charge linearly because the current is constant, and as a result, there will be a negative-going linear voltage ramp on the output of A_1 , as illustrated in Figure 12–16(a).



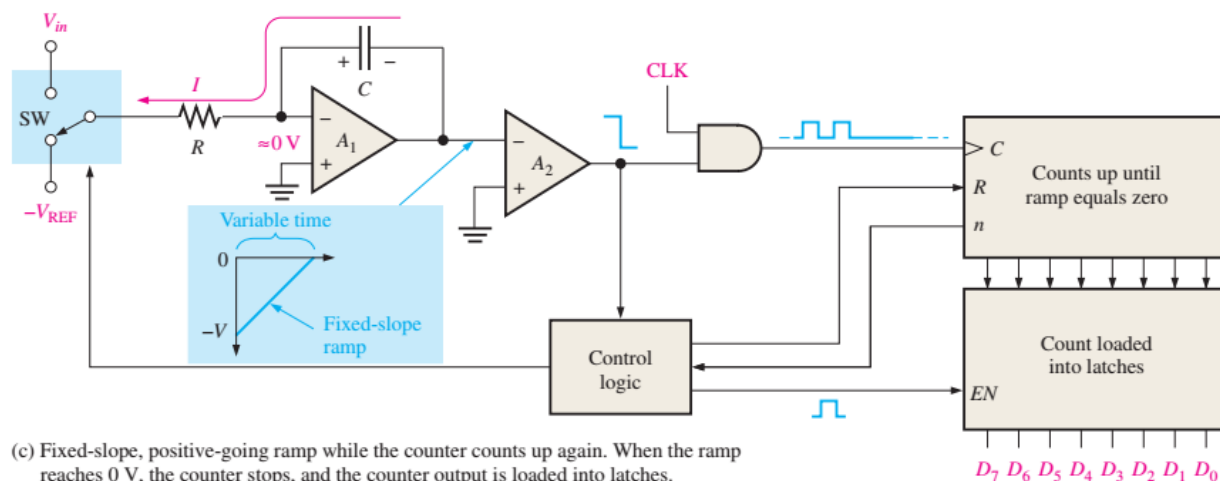


FIGURE 12–16 Illustration of dual-slope conversion.

When the counter reaches a specified count (n), it will be reset (R), and the control logic will switch the negative reference voltage ($-V_{REF}$) to the input of A_1 , as shown in Figure 12–16(b). At this point the capacitor is charged to a negative voltage ($-V$) proportional to the input analog voltage.

Now the capacitor discharges linearly because of the constant current from the $-V_{REF}$, as shown in Figure 12–16(c). This linear discharge produces a positive-going ramp on the A_1 output, starting at $-V$ and having a constant slope that is independent of the charge voltage. As the capacitor discharges, the counter advances from its RESET state. The time it takes the capacitor to discharge to zero depends on the initial voltage $-V$ (proportional to V_{in}) because the discharge rate (slope) is constant. When the integrator (A_1) output voltage reaches zero, the comparator (A_2) switches to the LOW state and disables the clock to the counter. The binary count is latched, thus completing one conversion cycle. The binary count is proportional to V_{in} because the time it takes the capacitor to discharge depends only on $-V$, and the counter records this interval of time.

Successive-Approximation Analog-to-Digital Converter

One of the most widely used methods of analog-to-digital conversion is successive approximation. It has a much faster conversion time than the dual-slope conversion, but it is slower than the flash method. It also has a fixed conversion time that is the same for any value of the analog input. Figure 12-17 shows a basic block diagram of a 4-bit successive approximation ADC.

Operation: The input bits of the DAC are enabled (made equal to a 1) one at a time, starting with the most significant bit (MSB). As each bit is enabled, the comparator

produces an output that indicates whether the input signal voltage is greater or less than the output of the DAC. If the DAC output is greater than the input signal, the comparator's output is LOW, causing the bit in the register to reset. If the output is less than the input signal, the 1 bit is retained in the register. The system does this with the MSB first, then the next most significant bit, then the next, and so on. After all the bits of the DAC have been tried, the conversion cycle is complete.

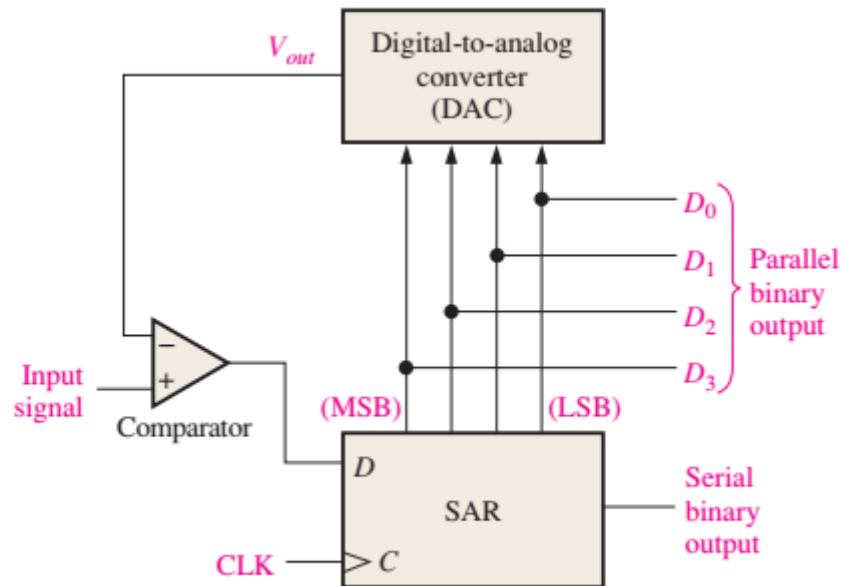
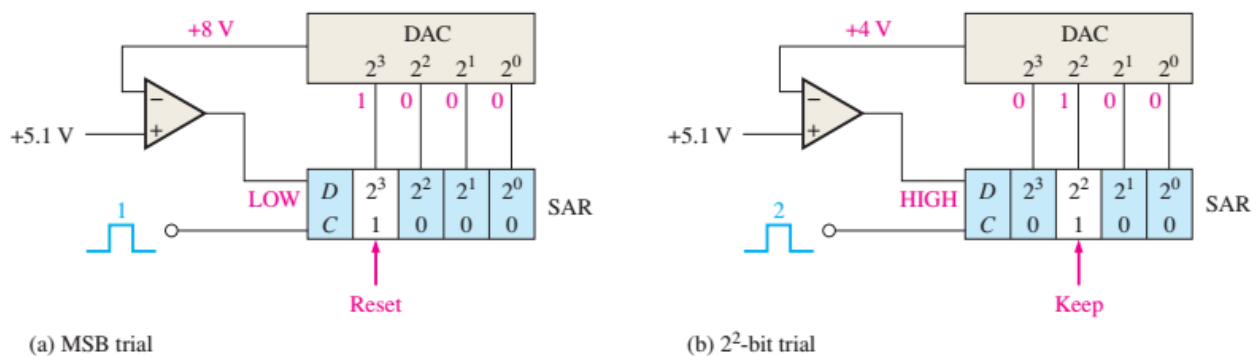


FIGURE 12-17 Successive-approximation ADC.

Figure 12–18 illustrates the step-by-step conversion of a constant input voltage (5.1 V in this case). Let's assume that the DAC has the following output characteristics: $V_{out} = 8\text{ V}$ for the 2^3 bit (MSB), $V_{out} = 4\text{ V}$ for the 2^2 bit, $V_{out} = 2\text{ V}$ for the 2^1 bit, and $V_{out} = 1\text{ V}$ for the 2^0 bit (LSB).



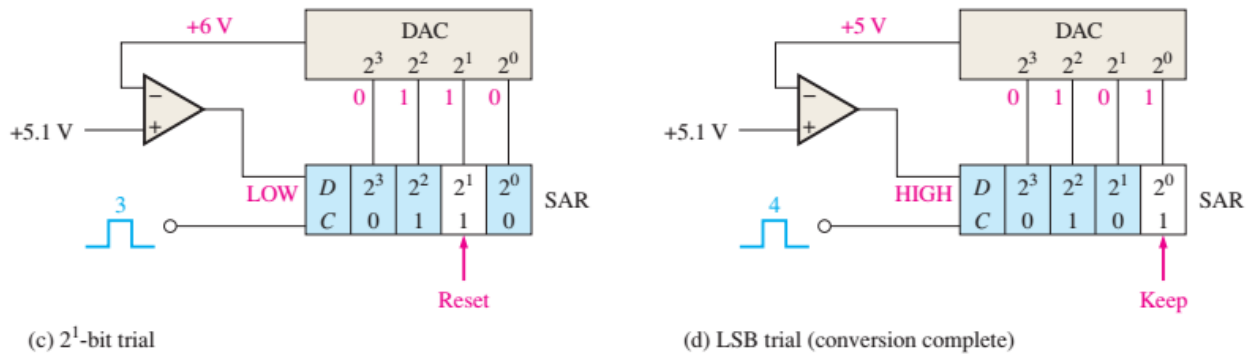


FIGURE 12-18 Illustration of the successive-approximation conversion process.

Figure 12–18(a) shows the first step in the conversion cycle with the MSB = 1. The output of the DAC is 8 V. Since this is greater than the input of 5.1 V, the output of the comparator is LOW, causing the MSB in the SAR to be reset to a 0.

Figure 12–18(b) shows the second step in the conversion cycle with the 2² bit equal to a 1. The output of the DAC is 4 V. Since this is less than the input of 5.1 V, the output of the comparator switches to a HIGH, causing this bit to be retained in the SAR.

Figure 12–18(c) shows the third step in the conversion cycle with the 2¹ bit equal to a 1. The output of the DAC is 6 V because there is a 1 on the 2² bit input and on the 2¹ bit input; 4 V + 2 V = 6 V. Since this is greater than the input of 5.1 V, the output of the comparator switches to a LOW, causing this bit to be reset to a 0.

Figure 12–18(d) shows the fourth and final step in the conversion cycle with the 2⁰ bit equal to a 1. The output of the DAC is 5 V because there is a 1 on the 2² bit input and on the 2⁰ bit input; 4 V + 1 V = 5 V.

The four bits have all been tried, thus completing the conversion cycle. At this point the binary code in the register is 0101, which is approximately the binary value of the input of 5.1 V. Additional bits will produce an even more accurate result. Another conversion cycle now begins, and the basic process is repeated. The SAR is cleared at the beginning of each cycle.

Sigma-Delta Analog-to-Digital Converter

Sigma-delta is a widely used method of analog-to-digital conversion, particularly in telecommunications using audio signals. The method is based on **delta modulation** where the difference between two successive samples (increase or decrease) is quantized; other ADC methods were based on the absolute value of a sample. Delta modulation is a 1-bit quantization method. The output of a delta modulator is a

single-bit data stream where the relative number of 1s and 0s indicates the level or amplitude of the input signal. The number of 1s over a given number of clock cycles establishes the signal amplitude during that interval. A maximum number of 1s corresponds to the maximum positive input voltage. A number of 1s equal to one-half the maximum corresponds to an input voltage of zero. No 1s (all 0s) corresponds to the maximum negative input voltage. This is illustrated in a simplified way in Figure 12–20. For example, assume that 4096 1s occur during the interval when the input signal is a positive maximum. Since zero is the midpoint of the dynamic range of the input signal, 2048 1s occur during the interval when the input signal is zero. There are no 1s during the interval when the input signal is a negative maximum. For signal levels in between, the number of 1s is proportional to the level.

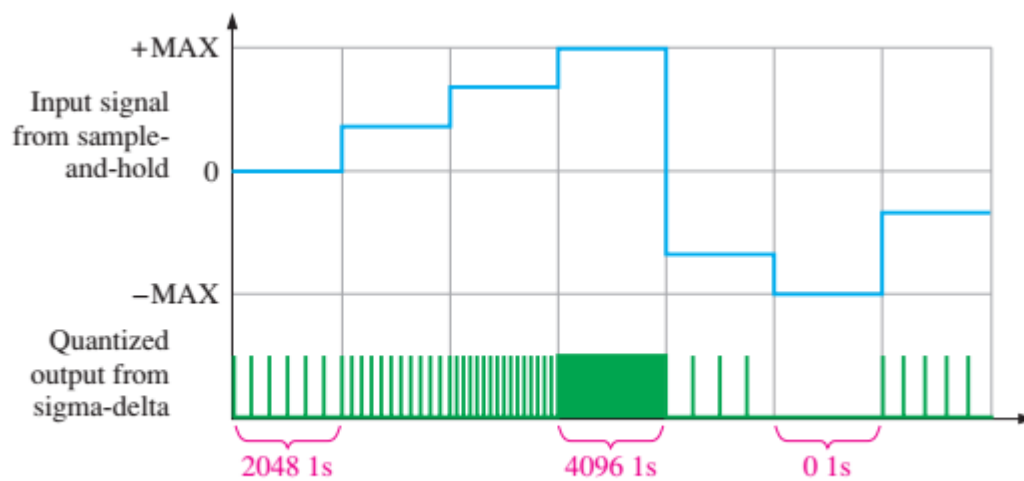


FIGURE 12–20 A simplified illustration of sigma-delta analog-to-digital conversion.

12–3 Methods of Digital-to-Analog Conversion

Digital-to-analog conversion is an important part of a digital processing system. Once the digital data has been processed, it is converted back to analog form.

Binary-Weighted-Input Digital-to-Analog Converter

One method of digital-to-analog conversion uses a resistor network with resistance values that represent the binary weights of the input bits of the digital code. Figure 12–26 shows a 4-bit DAC of this type. Each of the input resistors will either have current or have no current, depending on the input voltage level. If the input voltage is zero (binary 0), the current is also zero. If the input voltage is HIGH (binary 1), the amount of current depends on the input resistor value and is different for each input resistor, as indicated in the figure.

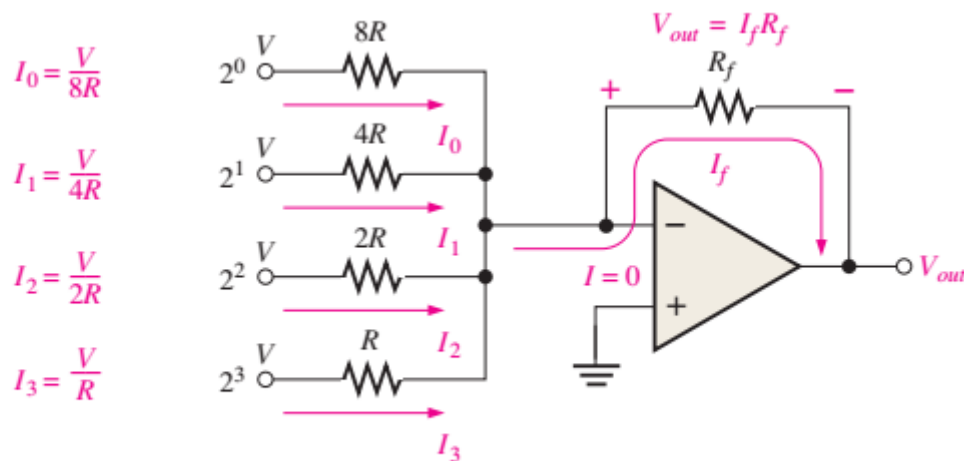


FIGURE 12–26 A 4-bit DAC with binary-weighted inputs.

Since there is practically no current into the op-amp inverting (-) input, all of the input currents sum together and go through R_f . Since the inverting input is at 0 V (virtual ground), the drop across R_f is equal to the output voltage, so $V_{out} = I_f R_f$.

The values of the input resistors are chosen to be inversely proportional to the binary weights of the corresponding input bits. The lowest-value resistor (R) corresponds to the highest binary-weighted input (2^3). The other resistors are multiples of R (that is, $2R$, $4R$, and $8R$) and correspond to the binary weights 2^2 , 2^1 , and 2^0 , respectively. The input currents are also proportional to the binary weights. Thus, the output voltage is proportional to the sum of the binary weights because the sum of the input currents is through R_f .

Disadvantages of this type of DAC are the number of different resistor values and the fact that the voltage levels must be exactly the same for all inputs. For example,

an 8-bit converter requires eight resistors, ranging from some value of R to $128R$ in binary-weighted steps. This range of resistors requires tolerances of one part in 255 (less than 0.5%) to accurately convert the input, making this type of DAC very difficult to mass-produce.

EXAMPLE 12-3

Determine the output of the DAC in Figure 12-27(a) if the waveforms representing a sequence of 4-bit numbers in Figure 12-27(b) are applied to the inputs. Input D_0 is the least significant bit (LSB).

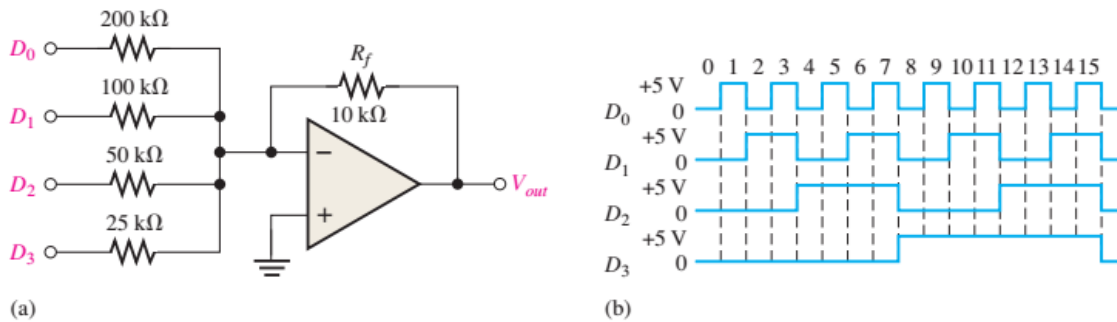


FIGURE 12-27

Solution

First, determine the current for each of the weighted inputs. Since the inverting (-) input of the op-amp is at 0 V (virtual ground) and a binary 1 corresponds to +5 V, the current through any of the input resistors is 5 V divided by the resistance value.

$$I_0 = \frac{5 \text{ V}}{200 \text{ k}\Omega} = 0.025 \text{ mA}$$

$$I_1 = \frac{5 \text{ V}}{100 \text{ k}\Omega} = 0.05 \text{ mA}$$

$$I_2 = \frac{5 \text{ V}}{50 \text{ k}\Omega} = 0.1 \text{ mA}$$

$$I_3 = \frac{5 \text{ V}}{25 \text{ k}\Omega} = 0.2 \text{ mA}$$

Almost no current goes into the inverting op-amp input because of its extremely high impedance. Therefore, assume that all of the current goes through the feedback resistor R_f . Since one end of R_f is at 0 V (virtual ground), the drop across R_f equals the output voltage, which is negative with respect to virtual ground.

$$V_{out(D0)} = (10\text{ k}\Omega)(-0.025\text{ mA}) = -0.25\text{ V}$$

$$V_{out(D1)} = (10\text{ k}\Omega)(-0.05\text{ mA}) = -0.5\text{ V}$$

$$V_{out(D2)} = (10\text{ k}\Omega)(-0.1\text{ mA}) = -1\text{ V}$$

$$V_{out(D3)} = (10\text{ k}\Omega)(-0.2\text{ mA}) = -2\text{ V}$$

From Figure 12–27(b), the first binary input code is 0000, which produces an output voltage of 0 V. The next input code is 0001, which produces an output voltage of -0.25 V. The next code is 0010, which produces an output voltage of -0.5 V. The next code is 0011, which produces an output voltage of -0.25 V + -0.5 V = -0.75 V. Each successive binary code increases the output voltage by -0.25 V, so for this particular straight binary sequence on the inputs, the output is a stair-step waveform going from 0 V to -3.75 V in -0.25 V steps. This is shown in Figure 12–28.

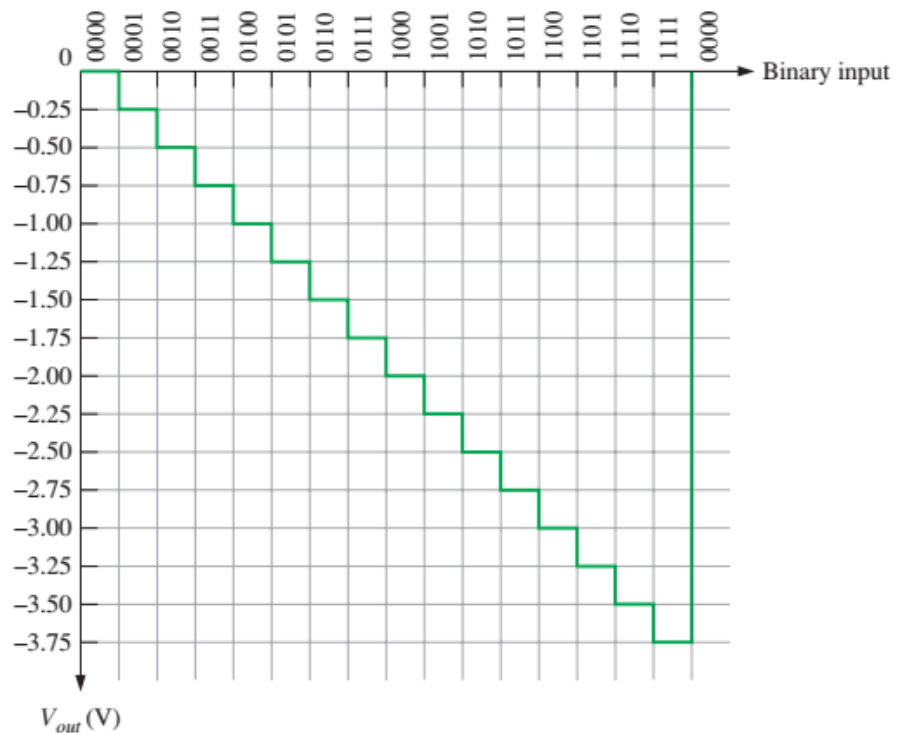


FIGURE 12–28 Output of the DAC in Figure 12–27.

The $R/2R$ Ladder Digital-to-Analog Converter

Another method of digital-to-analog conversion is the $R/2R$ ladder, as shown in Figure 12–29 for four bits. It overcomes one of the problems in the binary-weighted-input DAC in that it requires only two resistor values.

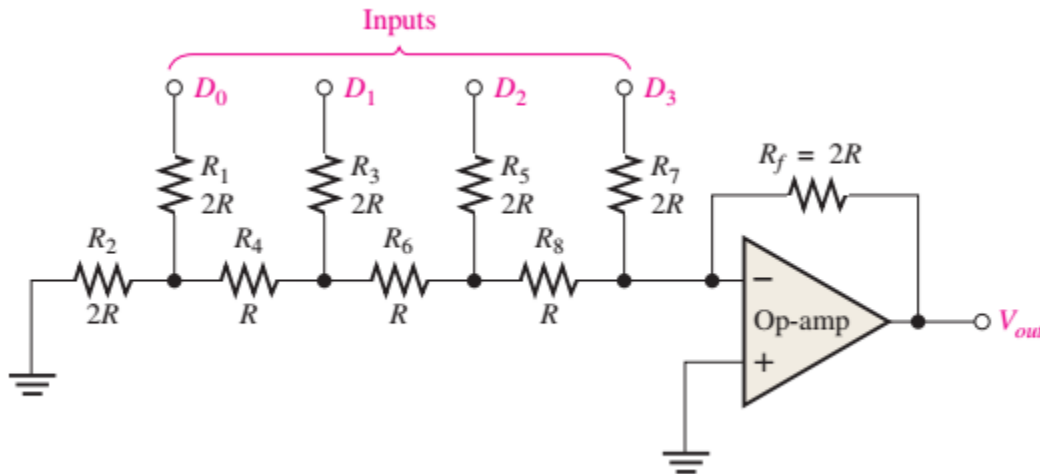
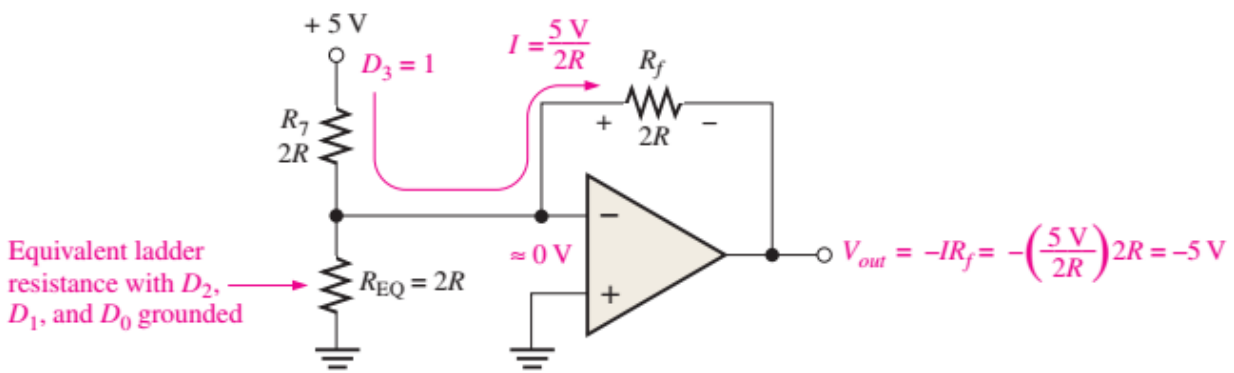
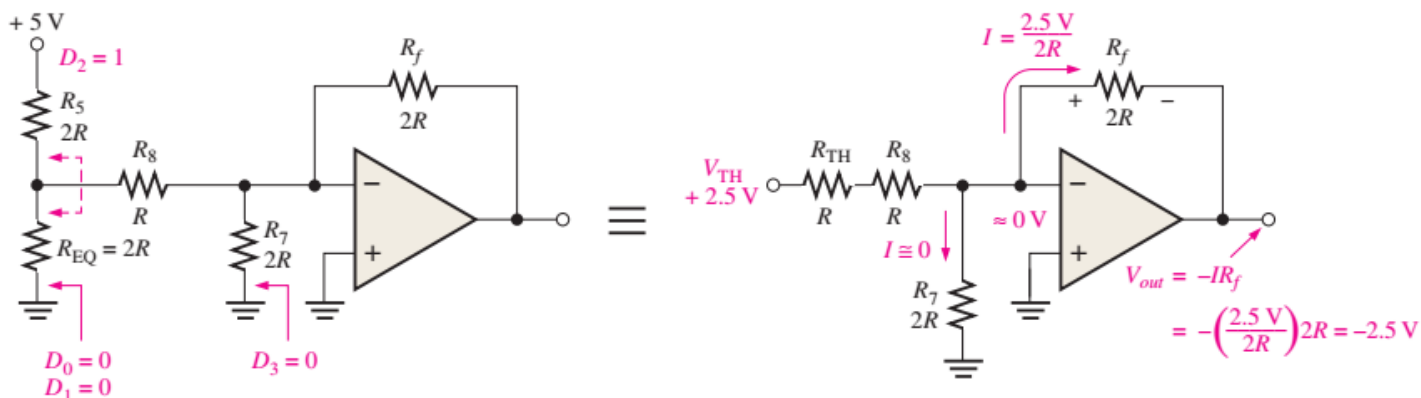


FIGURE 12–29 An $R/2R$ ladder DAC.

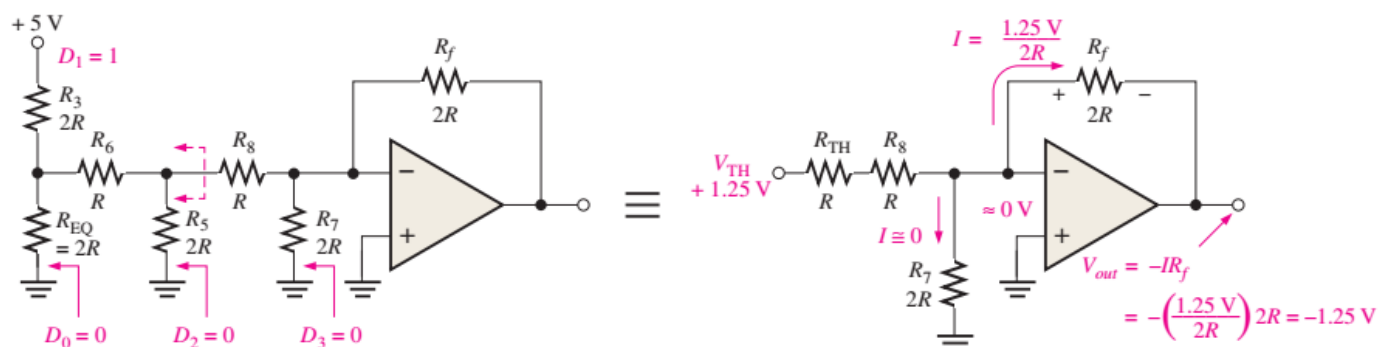
Start by assuming that the D_3 input is HIGH (+5 V) and the others are LOW (ground, 0 V). This condition represents the binary number 1000. A circuit analysis will show that this reduces to the equivalent form shown in Figure 12–30(a). Essentially no current goes through the $2R$ equivalent resistance because the inverting input is at virtual ground. Thus, all of the current ($I = 5 \text{ V}/2R$) through R_7 also goes through R_f , and the output voltage is -5 V. The operational amplifier keeps the inverting (-) input near zero volts ($\approx 0 \text{ V}$) because of negative feedback. Therefore, all current goes through R_f rather than into the inverting input.



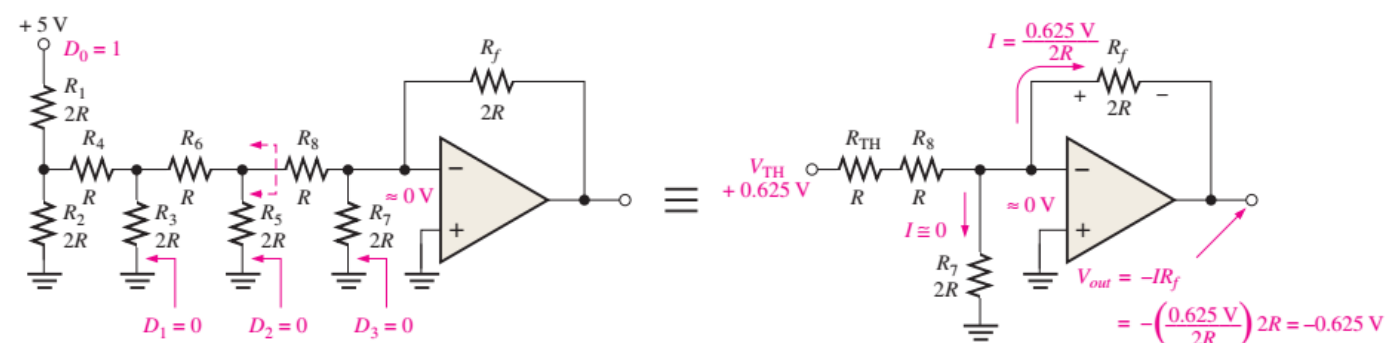
(a) Equivalent circuit for $D_3 = 1$, $D_2 = 0$, $D_1 = 0$, $D_0 = 0$



(b) Equivalent circuit for $D_3 = 0, D_2 = 1, D_1 = 0, D_0 = 0$



(c) Equivalent circuit for $D_3 = 0, D_2 = 0, D_1 = 1, D_0 = 0$



(d) Equivalent circuit for $D_3 = 0, D_2 = 0, D_1 = 0, D_0 = 1$

FIGURE 12-30 Analysis of the $R/2R$ ladder DAC.

Figure 12–30(b) shows the equivalent circuit when the D_2 input is at $+5\text{ V}$ and the others are at ground. This condition represents 0100. If we thevenize* looking from R_8 , we get 2.5 V in series with R , as shown. This results in a current through R_f of $I = 2.5\text{ V}/2R$, which gives an output voltage of -2.5 V .

Figure 12–30(c) shows the equivalent circuit when the D_1 input is at $+5\text{ V}$ and the others are at ground. This condition represents 0010. Again thevenizing looking from R_8 , you get 1.25 V in series with R as shown. This results in a current through R_f of $I = 1.25\text{ V}/2R$, which gives an output voltage of -1.25 V .

In part (d) of Figure 12–30, the equivalent circuit representing the case where D_0 is at +5 V and the other inputs are at ground is shown. This condition represents 0001. Thevenizing from R_8 gives an equivalent of 0.625 V in series with R as shown. The resulting current through R_f is $I = 0.625 \text{ V}/2R$, which gives an output voltage of -0.625 V. Notice that each successively lower-weighted input produces an output voltage that is halved, so that the output voltage is proportional to the binary weight of the input bits.

Performance Characteristics of Digital-to-Analog Converters

The performance characteristics of a DAC include resolution, accuracy, linearity, monotonicity, and settling time, each of which is discussed in the following list:

- **Resolution.** The resolution of a DAC is the reciprocal of the number of discrete steps in the output. This, of course, is dependent on the number of input bits. For example, a 4-bit DAC has a resolution of one part in $2^4 - 1$ (one part in fifteen). Expressed as a percentage, this is $(1/15)100 = 6.67\%$. The total number of discrete steps equals $2^n - 1$, where n is the number of bits. Resolution can also be expressed as the number of bits that are converted.
- **Accuracy.** Accuracy is derived from a comparison of the actual output of a DAC with the expected output. It is expressed as a percentage of a full-scale, or maximum, output voltage. For example, if a converter has a full-scale output of 10 V and the accuracy is; 0.1%, then the maximum error for any output voltage is $(10 \text{ V})(0.001) = 10 \text{ mV}$. Ideally, the accuracy should be no worse than; 1/2 of a least significant bit. For an 8-bit converter, the least significant bit is 0.39% of full scale. The accuracy should be approximately; 0.2%.
- **Linearity.** A linear error is a deviation from the ideal straight-line output of a DAC. A special case is an offset error, which is the amount of output voltage when the input bits are all zeros.
- **Monotonicity.** A DAC is **monotonic** if it does not take any reverse steps when it is sequenced over its entire range of input bits.
- **Settling time.** Settling time is normally defined as the time it takes a DAC to settle within; 1/2 LSB of its final value when a change occurs in the input code.

EXAMPLE 12–4

Determine the resolution, expressed as a percentage, of the following:

(a) an 8-bit DAC

(b) a 12-bit DAC

Solution

(a) For the 8-bit converter,

$$\frac{1}{2^8 - 1} \times 100 = \frac{1}{255} \times 100 = \mathbf{0.392\%}$$

(b) For the 12-bit converter,

$$\frac{1}{2^{12} - 1} \times 100 = \frac{1}{4095} \times 100 = \mathbf{0.0244\%}$$

The Reconstruction Filter

The output of the DAC is a “stair-step” approximation of the original analog signal after it has been processed by the **digital signal processor (DSP)**, which is a special type of micro-processor that processes data in real time. The purpose of the low-pass reconstruction filter (sometimes called a postfilter) is to smooth out the DAC output by eliminating the higher frequency content that results from the fast transitions of the “stair-steps,” as roughly illustrated in Figure 12–34.

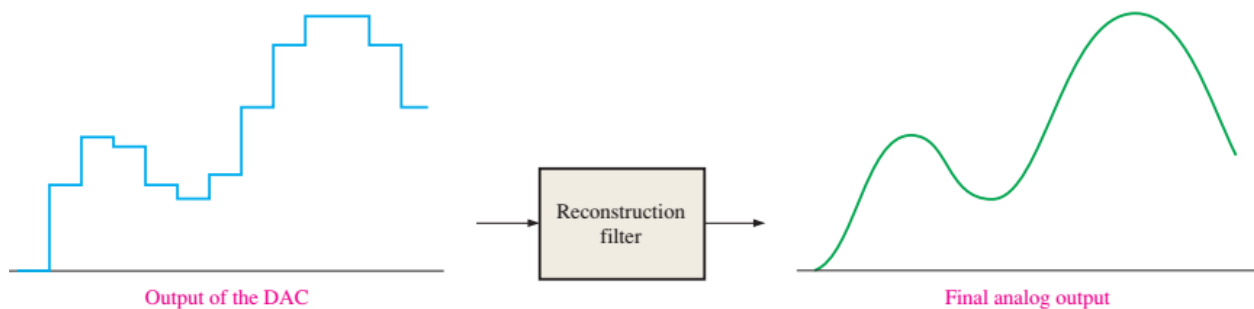


FIGURE 12–34 The reconstruction filter smooths the output of the DAC.

14 - Data Processing and Control

INTRODUCTION

This chapter provides a basic introduction to computers, microprocessors, and microcontrollers. It gives you a fundamental coverage of basic concepts related to data processing and control. For the most part, a generic approach is used to present basic concepts of the topics. The total computer system with practical considerations is covered. Various aspects of a microprocessor and its role as the CPU in computer systems are presented and programming is briefly discussed.

14–1 The Computer System

General-purpose computers, with which most are familiar, and special-purpose computers are used to control various functions or perform specific tasks in areas such as automotive, consumer appliances, manufacturing processes, and navigation. The general-purpose computer system, which can be programmed to do many different things, is the focus in this section.

All computer systems work with information, or data, to produce a desired result. To accomplish this, computer systems must perform the following tasks:

- Acquire information from data sources, including human operators, sensors, memory and storage devices, communication networks, and other computer systems
- Process information by interpreting, evaluating, manipulating, converting, formatting, or otherwise working with acquired data in some intended fashion as directed by a step-by-step set of instructions called a *program*
- Provide information in a meaningful form to data recipients, including human operators, actuators, memory and storage devices, communication networks, and other computer systems.

Information processing is performed by the central processing unit, or CPU, which is the brain of the computer system. The CPU acquires information through the input section of the computer system, provides information through the output section, and uses the system memory and storage to store and retrieve information as needed. The CPU transfers information to and from other sections of the computer system over special groups of signal lines called *buses*. Figure 14–1 shows a block diagram of a general-purpose computer system. Each block will be discussed in terms of its purpose and function.

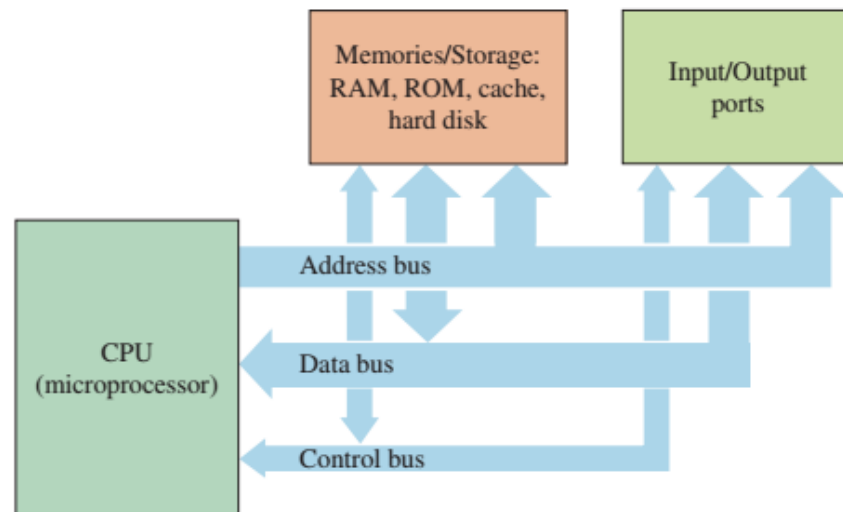


FIGURE 14-1 Basic computer block diagram.

The Central Processing Unit

The central processing unit (**CPU**) performs much of what is associated with the term *computer*. It executes the instruction sequences (called *programs*) in the computer system, directly processes much of the data that pass through the computer system, and controls and coordinates the various sections that make up the computer system. The CPU is basically a **microprocessor** (or simply processor). A single IC package can contain two or more processors, forming a **multicore processor**.

Memory and Storage

Computer systems must have some means of storing and retrieving the information with which they work and use two types of devices—**memory devices** and **storage devices**—to do so. Memory devices typically are semiconductor devices that store information electronically, interface with the computer system through the system buses, and contain dynamic information, such as programs and program variables that is frequently accessed or modified. Storage devices typically store information on some physical medium, interface with the system through a peripheral interface, and contain primarily static information, such as program and data files, that is accessed or modified relatively infrequently. Memory devices are faster than storage devices; however, memory devices have lower storage capacities and higher cost per bit than storage devices. Memory in computer systems can be classified both by the type of memory and the function it performs. The different types and characteristics of memory were discussed in Chapter 11. Here we examine the functional requirements of memory in computer systems.

Main Memory

The **main memory** is the computer system memory that contains programs and data associated with them, such as program variables, the program stack, and information the operating system requires to execute the program. The earliest 8-bit processors (for example, the Intel 8080, Motorola 6800, and Intel MOS 6502) had 16-bit address buses that could access $2^{16} = 65,536$ bytes (64 kilobytes or 64 KB) of memory. However, the main memory in 8-bit PCs was actually less than this because other devices in the system used part of the address space. The 16-bit computers that followed had 20-bit address buses that could access $2^{20} = 1,048,756$ bytes (1 megabyte or 1 MB) of memory. Modern computers require gigabytes of main memory to support the requirements of their graphical user interface (GUI) operating systems and application programs. Main memory must meet the requirements of a large storage capacity at an economical price and also allow the computer system to modify data within it. Because of these requirements, computer systems typically use some form of dynamic RAM (DRAM) for main memory that features large capacity, low cost per bit, and read/write capability.

Cache Memory

Cache memory is memory that computer systems use to overcome the relatively slow speed of main memory DRAM. **Caching** is a process that copies frequently accessed instructions or data from slow main memory into faster cache memory to reduce access time and improve system performance. Because of these requirements, computer systems use some form of static RAM (SRAM) for cache memory.

Basic Input / Output System (BIOS) Memory

The design of every computer system differs to some extent from other systems. The basic input/output system (**BIOS**) memory contains system-specific low-level code that runs the power on self-test (POST), installs specialized software called drivers to configure and provide access to the computer system hardware, and loads the operating system. The BIOS memory must retain its contents when power is removed so that the BIOS code is ready to run when the computer first powers up. This requires computer systems to use some form of nonvolatile memory for BIOS. Computers use a low-power CMOS device with a back-up battery to preserve the contents when the system power was shut off. This allowed users to change and save BIOS settings when they made changes to system hardware configuration.

Most recently, computers have used EEPROM and flash devices so that users can easily upgrade the BIOS firmware to the latest revision.

System Bus

As you have learned, computers acquire, process, and provide information. Computers must be able

- (a) to specify where to acquire and return information,
- (b) to transfer the information from its source to its destination, and
- (c) to coordinate the movement of data within the computer system. The mechanism by which the computer accomplishes this is the **system bus**, which consists of three component buses: the address bus, the data bus, and the control bus.

The Address Bus

The **address bus** is the means by which a processor specifies the system location from which data are to be read or to which data are to be written. For example, the processor sends an address code to the memory specifying where certain data are stored. If the address bus is 32 bits wide, 2^{32} or 4,294,967,296 memory locations can be accessed.

The Data Bus

The **data bus** consists of signal lines over which the computer system transfers information from one device to another. Because the processor can both read data from and write data to system devices, each data line is bidirectional. The number of data lines determines the width of the data bus, which is a factor in how quickly the processor can process data. The earliest microprocessors had 4-bit and 8-bit data buses, but modern processors have 64-bit data buses.

The Control Bus

The **control bus** is the collection of signals that controls the transfer of data within the system and coordinates the operation of system hardware. Unlike the address and data buses, which consist of functionally identical signals that function as a group, the individual signals lines that make up the control bus vary in characteristics, nature, and function. Control signals can be unidirectional or bidirectional, can function individually or with other control signals, can be active-HIGH or active-LOW, can operate synchronously or asynchronously, and can be

edge-oriented or level-oriented. Despite this individual diversity, computer systems and processor operations are similar enough that the signals that make up the control bus—read, write, interrupt, and others—are also similar.

A Typical Computer System

The block diagram in Figure 14–2 shows the main elements in a typical computer system and how they are interconnected. Notice that the computer itself is connected with several peripheral units. For the computer to accomplish a given task, it must communicate with the “outside world” by interfacing with people, sensing devices, or devices to be controlled through input and output ports.

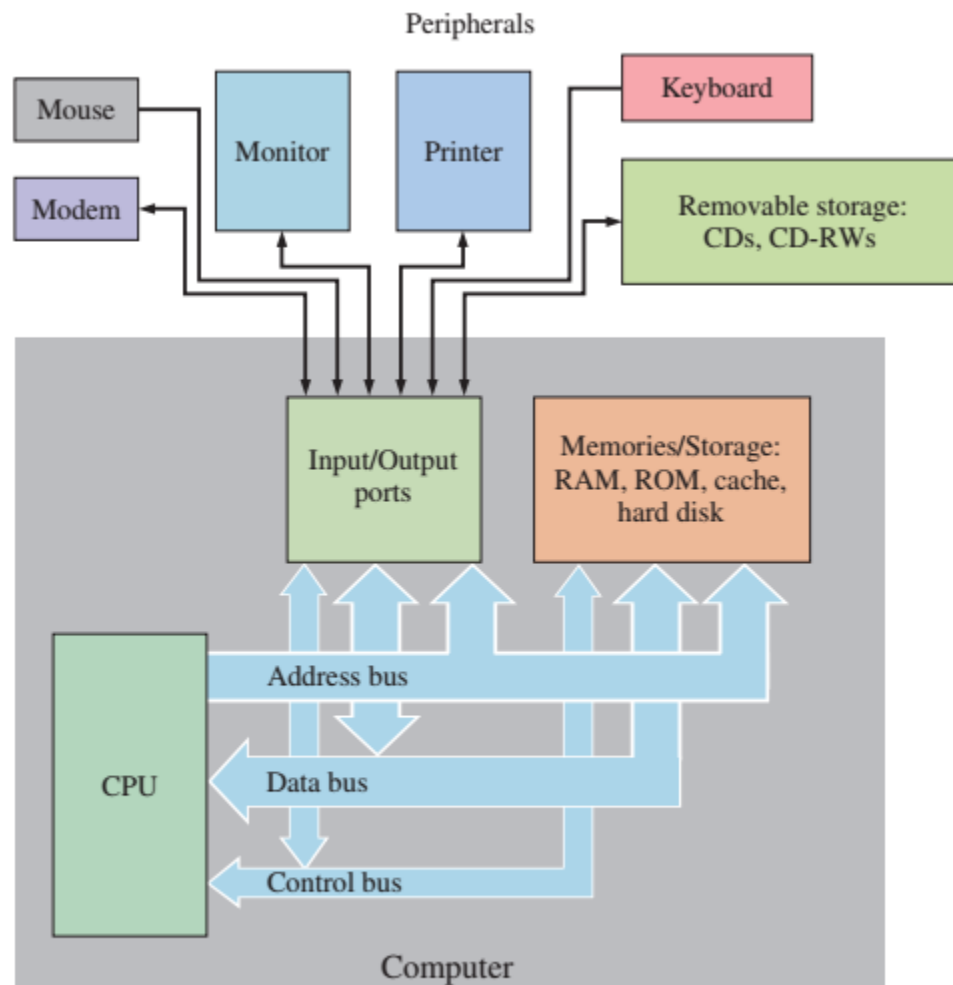


FIGURE 14–2 Basic block diagram of a typical computer system including common peripherals. The computer itself is shown in the gray block.

Computer Software

In addition to the hardware, a major part of a computer system is the software. The software makes the hardware perform. The two major categories of software used in computers are the [system software](#) and the [application software](#). The system

software is called the operating system (OS) and allows the user to interface with the computer. The most common operating systems are Windows and Mac OS. Many other operating systems are used in special-purpose and mainframe computers.

System software performs two basic functions. It manages all the hardware and software in a computer. For example, the operating system manages and allots space on the hard disk. System software also provides a consistent interface between applications software and hardware. This allows an applications program to work on various computers that may differ in hardware details. The operating system on your computer allows you to have several programs running at the same time. This is called *multitasking*.

Application software is used to accomplish a specific job or task, such as word processing, accounting, tax preparation, circuit simulation, graphic design, to name only a very few.

14–3 The Processor: Basic Operation

As you have learned, a microprocessor forms the CPU of a computer system. A microprocessor is a single integrated circuit that consists of several units, each designed for a specific job. The four basic elements that are common to all microprocessors are the arithmetic logic unit (ALU), the instruction decoder, the register set, and the timing and control unit, as shown in Figure 14–9.

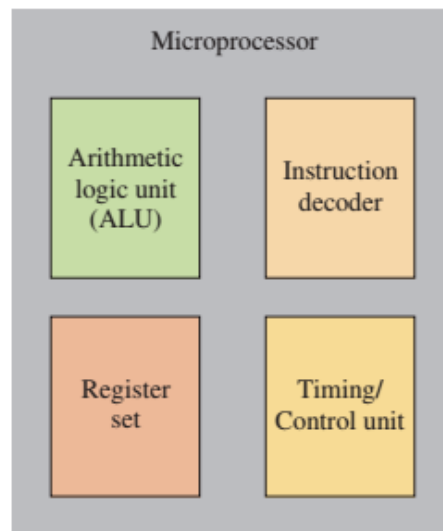


FIGURE 14–9 Elements of a microprocessor (CPU).

Figure 14–10 shows a simple block diagram of a microprocessor. The elements shown are common to most processors, although the internal arrangement or architecture and complexity vary. This generic block diagram of an 8-bit processor with a small register set is used to illustrate fundamental operation. Today, processors have data buses that are 64 bits.

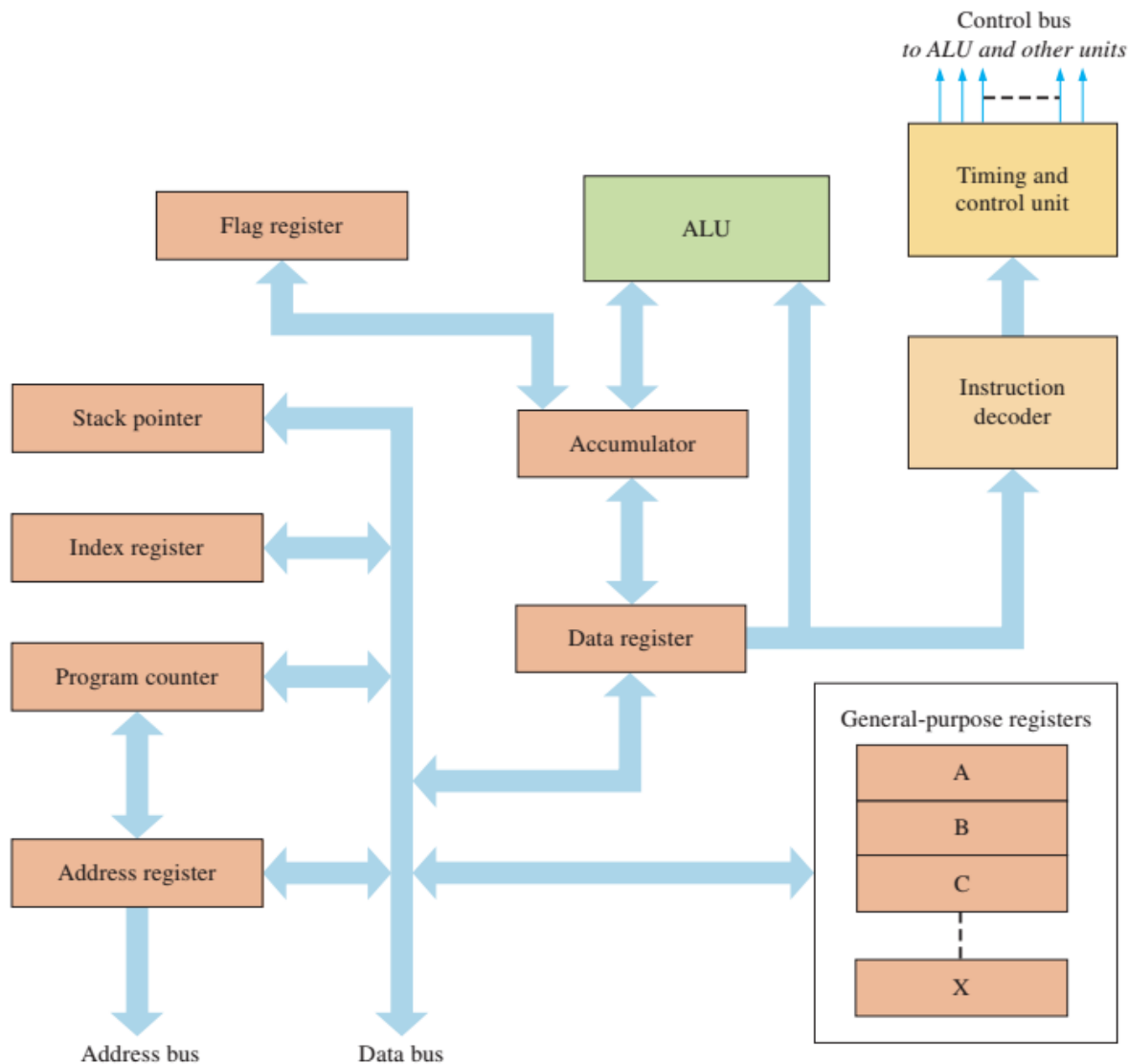


FIGURE 14-10 Basic model of a simplified processor.

The Fetch/Execute Cycle

When a program is being run, the processor goes through a repetitive cycle consisting of two fundamental phases, as shown in Figure 14-11. One phase is called *fetch* and the other is called *execute*. During the **fetch** phase, an instruction is read from the memory and decoded by the instruction decoder. During the **execute** phase, the processor carries out the sequence of operations called for by the instruction. As soon as one instruction has been executed, the processor returns to the fetch phase to get the next instruction from the memory.

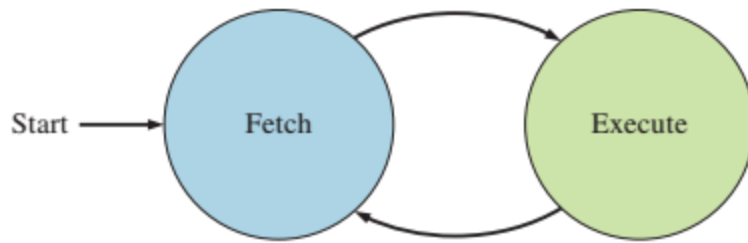


FIGURE 14-11 The fetch/execute cycle of a processor.

Pipelining

A technique where the microprocessor begins executing the next instruction in a program before the previous instruction has been completed is called **pipelining**. That is, several instructions are in the pipeline simultaneously, each at a different processing stage. Typically, a pipeline is divided into stages or segments, and each stage can execute its operation concurrently with the other stages. When a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operation from the preceding segment. The final results of each instruction emerge at the end of the pipeline in rapid succession. Figure 14-12 is a simplified illustration of non-pipelined processing compared to pipelined processing using three stages of execution.

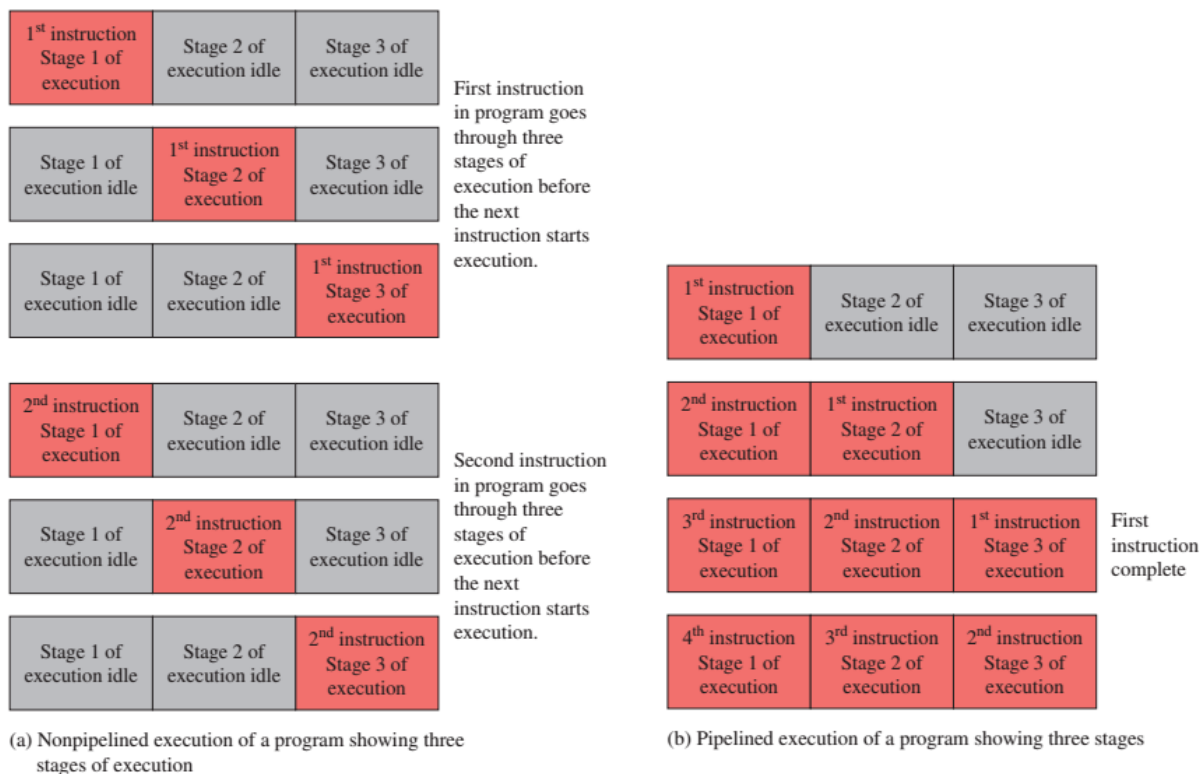


FIGURE 14-12 Illustration of pipelining.

As shown in the figure, in non-pipelined processing of a program, one instruction at a time is executed through all of its stages before the next instruction begins

execution. As you can see in part (a), all the stages of execution are idle (gray) except the one that is active (red). In pipelined processing, as soon as one instruction has finished an execution stage, the next instruction begins that stage. Pipelining results in much shorter overall execution times. Once the pipeline is “full,” there are no idle processing stages.

Processor Elements

ALU

This part of the processor contains the logic to perform arithmetic and logic operations. Data are transferred into the **ALU** from the accumulator and from the data register. For the model in Figure 14–10, the accumulator and data register are 8-bit registers that hold one byte of data. Each byte transferred into the ALU is called an *operand* because it is operated on by the ALU. As an example, Figure 14–13 shows an 8-bit number from the accumulator being added to an 8-bit number from the data register. The result of this addition operation (sum) is put back into the accumulator and replaces the original operand that was stored there. When the ALU performs an operation on two operands, the result always goes into the accumulator to replace the previous operand.

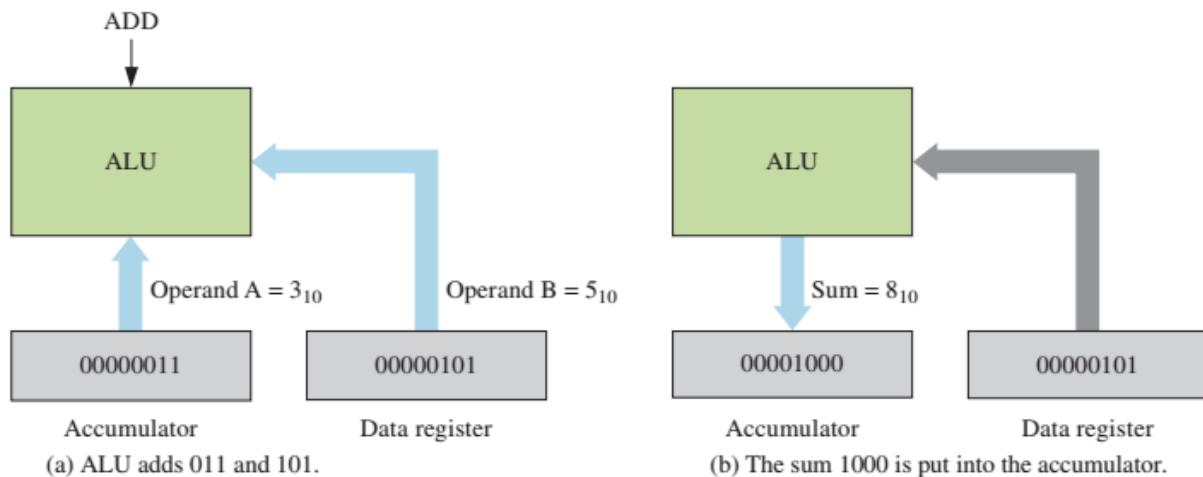


FIGURE 14–13 Example of the ALU adding two operands.

As demonstrated in Figure 14–13, one function of the accumulator is to store an operand prior to an operation by the ALU. Another function is to store the result of the operation after it has been performed. The data register temporarily stores data that is to be put onto the data bus or that has been taken off of the data bus.

Instruction Decoder and Timing/Control Unit

An instruction is a binary code that tells the processor what it is to do. An orderly arrangement of many different instructions makes up a program. A **program** is a step-by-step procedure used by the processor to carry out a specified task. The instruction decoder within the processor decodes an instruction code that has been transferred on the data bus from the memory. The instruction code is commonly known as an **op-code**. When the op-code is decoded, the instruction decoder provides the timing and control unit with this information. The timing and control unit can then produce the proper signals and timing sequence to execute the instruction.

Register Set

Processors typically have two categories of registers for temporary storage of data: general purpose registers and special-purpose registers. General-purpose registers are used to store any type data that may be required by a program. Special-purpose registers are dedicated to a specific function. Some typical special-purpose registers are described as follows.

Flag register: This register is sometimes called a condition code register or status register. It indicates the status of the contents of the accumulator or certain other conditions within the processor. For example, it can indicate a zero result, a negative result, the occurrence of a carry, or the occurrence of an overflow from the accumulator.

Program counter: This counter produces the sequence of memory addresses from which the program instructions are taken. The content of the program counter is always the memory address from which the next byte is to be taken. In some processors, the program counter is known as the *instruction pointer*.

Address register: This register temporarily stores an address from the program counter in order to place it on the address bus. As soon as the program counter loads an address into the address register, it is incremented (increased by 1) to the address of the next instruction.

Stack pointer: The stack pointer is a register that is mainly used during program subroutines and interrupts. It is used in conjunction with the memory stack.

Index register: The index register is used as one means of addressing the memory in mode of addressing called *indexed addressing*.

The Processor and the Memory

The processor is connected to a memory with the address bus and data bus. Also, there are certain control signals that must be sent between the processor and the memory, such as the read and write controls. The address bus is unidirectional so the address bits go only one way, from the processor to the memory. The data bus is bidirectional, so data bits are transferred between the processor and memory in either direction. This is illustrated in Figure 14–14.

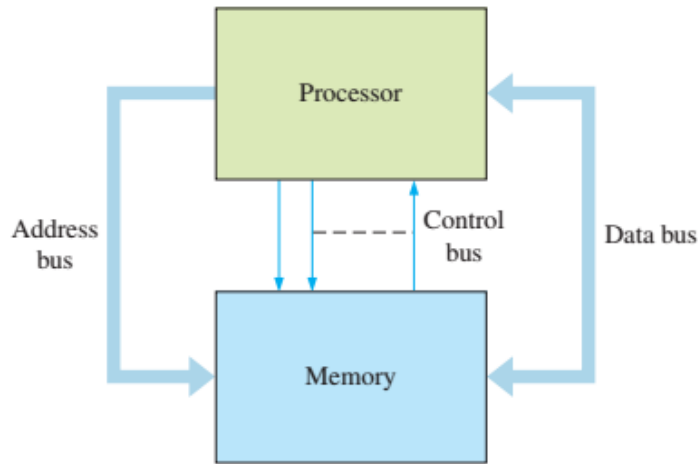
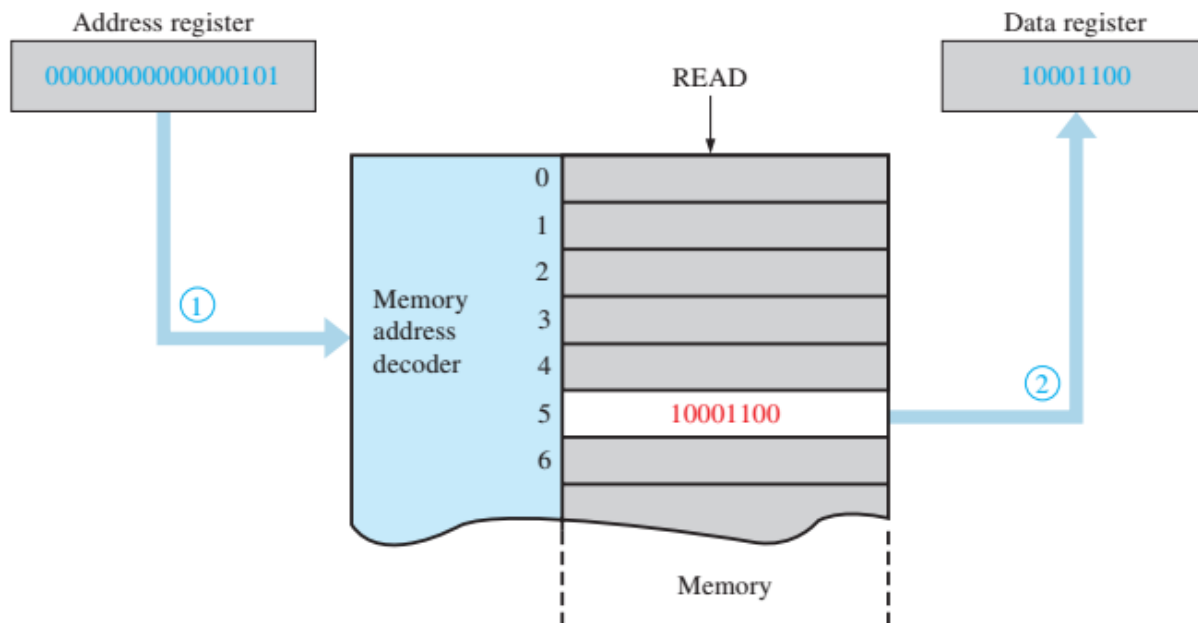


FIGURE 14–14 A processor and memory.

The Read Operation

To transfer data from the memory to the processor, a **read** operation must be performed, as shown in Figure 14–15, using an 8-bit data bus and a 16-bit address bus for illustration. To start, the program counter contains the address of the data to be read from the memory. This address is loaded into the address register and placed onto the address bus. The program counter is then incremented (advanced by one) to the next address and waits. Once the address code is on the bus, the processor timing and control unit sends a read signal to the memory. At the memory, the address bits are decoded and the desired memory location is selected. The read signal causes the contents of the selected address to be placed on the data bus.



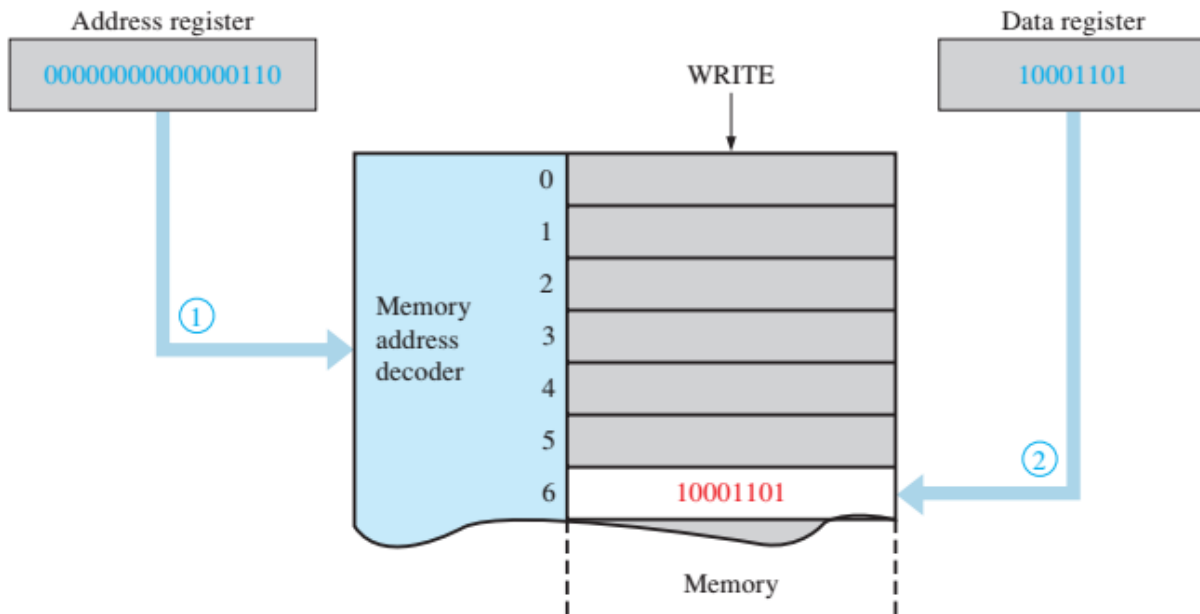
- ① Address S_{10} is placed on address bus and followed by the read signal.
- ② Contents of address S_{10} in memory is placed on data bus and stored in data register.

FIGURE 14–15 Illustration of the read operation.

The data are then loaded into the data register to be used by the processor, completing the read operation. In this illustration, each memory location contains one byte of data. When a byte is read from memory, it is not destroyed but remains in the memory. This process of “copying” the contents of a memory location without destroying the contents is called *nondestructive read*.

The Write Operation

To transfer data from the processor to the memory, a **write** operation is required, as illustrated in Figure 14–16. A data byte held in the data register is placed on the data bus, and the processor sends the memory a write signal. This causes the byte on the data bus to be stored at the memory location selected by the address code. The existing contents of that particular memory location are replaced by the new data. This completes the write operation.



- ① Address code for address 6_{10} is placed on address bus.
- ② Data are placed on data bus and followed by the write signal. Data are stored at address 6_{10} in memory.

FIGURE 14-16 Illustration of the write operation.

Roles of the CPU

The CPU has three major roles in a computer system. The first role of the CPU is to control the system hardware. Specifically, the CPU determines how data move through the computer system, which devices are active, and when specific operations and data transactions occur. In computers, some of this control is decentralized by assigning some tasks (such as peripheral access and communications and graphics processing) to devices that can perform those tasks more quickly and efficiently than the CPU itself. Even so, the CPU still coordinates the operation of the computer system as a whole.

The second role of the CPU is to provide hardware support for the operating system software. The first computers were large mainframes that were too expensive to devote to a single user or program. The operating systems allowed these computers to support multiple users and programs, but they required special hardware to ensure that users and programs would not accidentally or deliberately interfere with each other. As the operating systems in personal computers evolved from single-user single-application platforms to multitasking and multiprocessing systems, the microprocessors have incorporated the features required to support them.

The third role of the CPU is to execute application programs. The CPU accesses the system hardware and controls the flow of data through the system largely because

some application program requires that it do so. This role greatly influenced the development of many early complex instruction set computing (CISC) microprocessors. Reduced instruction set computing (RISC) processors emphasize smaller and more efficient instruction sets than those in CISC processors and place the burden of high-level programming support on the **compilers**, which are programs that convert the source code written by programmers to executable code that is executed by the processor.

14–6 Operating Systems and Hardware

Each computer system consists of two main components. The microprocessor, memories, interface circuits, peripherals, power supplies, and other electronic components make up what is collectively referred to as computer **hardware**. The programs that the microprocessor executes and that control the computer system are collectively referred to as computer **software**. One general rule is anything in a computer system that you can physically touch is hardware, and anything that you can't physically touch is software.

Operating System Basics

The **operating system** (OS) of a computer is a special program that establishes the environment in which application programs operate. The operating system provides the functional interface between application programs in the system, called **processes**, and the computer hardware. Because the operating system must work closely with the computer hardware, it is often written in assembly language or programming language with low-level hardware support, such as C++. An operating system increases the overall complexity of a computer system, but using an operating system offers a number of advantages over running stand-alone application programs. The operating system tests and initializes hardware in the computer system, eliminating the need for each application to duplicate these functions. Operating systems also provide a standard computing environment so that applications can execute consistently. Finally, operating systems provide system services that allow applications access to commonly used system resources (such as the real-time clock, I/O ports, and data files), which simplify the code for applications programs. A drawback of operating systems is that processes may execute more slowly; accessing system resources through an operating system can take longer than a program accessing them directly. An operating system has three basic duties.

1. To schedule and allocate system resources (CPU time, memory, access to system peripherals)
2. To protect system processes and resources (preventing accidental or deliberate corruption of process code and data, unauthorized access to hardware and memory)
3. To provide system services (messaging between processes, low-level hardware drivers)

Multiple Processes

Computers can run multiple processes in two basic ways. The first way, called **multitasking**, shares a single-core processor among multiple processes. The processor runs more than one process but switches between them so that each process uses only part of the processor's available time. Multitasking systems use different techniques to decide when to switch between processes. One technique allows a process to run until it must wait for some event, such as a keypress, before it can continue and switches to another process that is ready to run. Another technique, called *preemptive multitasking*, allows each process to run for a specific amount of time before the operating system switches to another process. A third technique, called *non-preemptive multitasking*, allows a process to run until it voluntarily relinquishes the processor to another process. Figure 14–26 illustrates how a single-core processor multitasks.

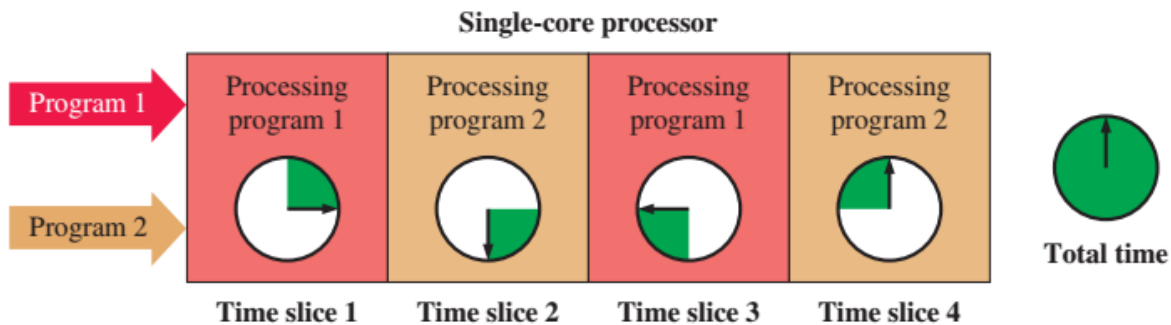


FIGURE 14–26 Simplified model of processor multitasking.

The second way for a computer system to run multiple processes, called **multiprocessing**, uses multiple processors, each of which can either multitask or run a single process. Figure 14–27 illustrates the concept of multitasked multiprocessing.

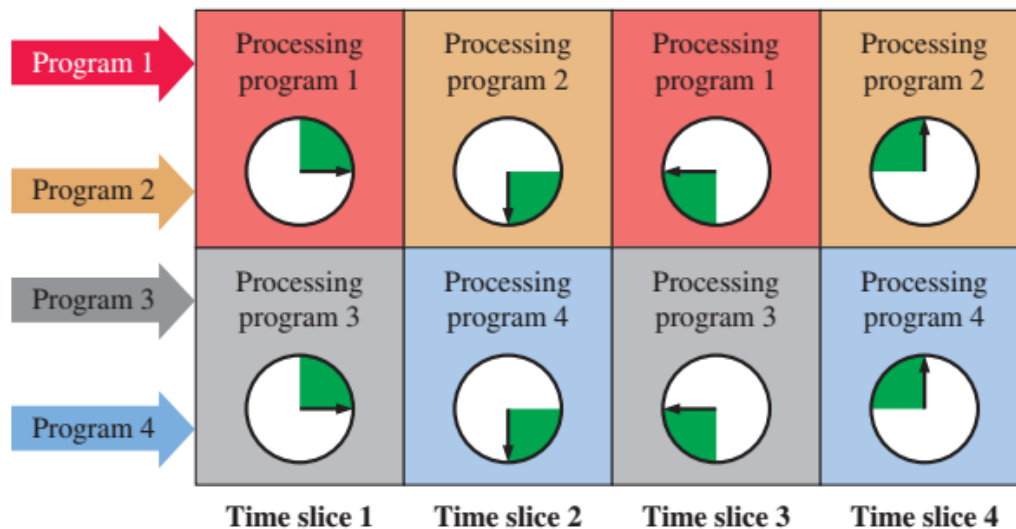


FIGURE 14-27 Multitasked multiprocessing in a multicore processor.

Supervisor and User States

It is difficult for multiple users or processes to coexist in a computer system if processes have unrestricted access to system resources. Once a process takes control of the processor and is running, it can modify or disable any software or hardware in the system that exists to control it. The solution to this is to restrict what the process can access. Some processors use the user/supervisor state bit so only trusted code, like the operating system, can run under certain circumstances. For multi-process or multiuser systems, the processor executes in supervisor state when it first powers up, while the operating system is running, and when the processor responds to an interrupt. When the operating system loads and transfers control to an application program, it first clears the user/supervisor state bit. This places the process in user state and prevents it from accessing restricted parts of the computer system's hardware or software.

Memory Management Unit

One device in the computer system that has not yet been discussed is the memory management unit, or **MMU**. Memory management units are very sophisticated logic devices that handle many details associated with accessing memory in computer systems, including memory protection, wait-state generation, address translation for handling virtual memory, and cache control. As an example, consider a simplified MMU that simply provides memory protection. The processor can program the MMU with the start and end addresses of a memory range. The MMU then acts as a comparator. If the MMU detects a value on the address bus that is less

than the programmed start address or greater than the programmed end address, it will generate a hardware interrupt to the processor.

System Services

Operating systems provide system services that allow applications access to commonly used system resources. This is essential for allowing processes to interact and communicate with each other to share information, coordinate operations, and otherwise function in unison. Interprocess communication uses software interrupts (also called *traps*). When one process wishes to utilize a system service, it loads specific registers with values and then invokes a specific trap to pass control to the operating system's exception handler for that trap. When the process executes the trap, the processor enters supervisor mode; and the exception handler uses the register contents to fulfill the requested service. If, for example, the requested service was to send several bytes from one process to another, the exception handler would use the starting address of the data and the number of data bytes contained in the processor registers to copy the data from the user memory of the source process to the user memory of the destination process. It would then load a condition code indicating that the service had been completed successfully (or failed) in one of the processor registers and would return processor control to the requesting process. When processes are meant to interact with other processes, they each must be carefully designed to ensure that messages are passed at the right time and in the right order and that the processes can recover from communication errors. Otherwise, one process may believe that it has sent out a valid message and await a response, while the intended destination process is waiting for the first process to send a message to which it can respond. The result is that neither process can proceed.

14-7 Programming

Assembly language is a way to express machine language in English-like terms, so there is a one-to-one correspondence. Assembly language has limited applications and is not portable from one processor to another, so most computer programs are written in high-level languages such as C++, JAVA and BASIC. High-level languages are portable and therefore can be used in different computers. High-level languages must be converted to the machine language for a specific microprocessor by a process called *compiling*.

Levels of Programming Languages

A hierarchy diagram of computer programming languages relative to the computer hardware is shown in Figure 14–28. At the lowest level is the computer hardware (CPU, memory, disk drive, input/output). Next is the **machine language** that the hardware understands because it is written with 1s and 0s (remember, a logic gate can recognize only a LOW (0) or a HIGH (1)). The level above machine language is **assembly language** where the 1s and 0s are represented by English-like words. Assembly languages are considered low-level because they are closely related to machine language and are machine dependent, which means a given assembly language can only be used on a specific microprocessor. The level above assembly language is **high-level language**, which is closer to human language and further from machine language. An advantage of high-level language over assembly language is that it is portable, which means that a program can run on a variety of computers. Also, high-level language is easier to read, write, and maintain than assembly language. Most system software (e.g., Windows), and applications software (e.g., word processors and spreadsheets) are written with high-level languages.

Assembly Language

To avoid having to write out long strings of 1s and 0s to represent microprocessor instructions, English-like terms called mnemonics or op-codes are used. Each type of microprocessor has its own set of mnemonic instructions that represent binary codes for the instructions. All of the mnemonic instructions for a given microprocessor are called the instruction set. Assembly language uses the instruction set to create programs for the microprocessor; and because an assembly language is directly related to the machine language (binary code instructions), it is classified as a low-level language. Assembly language is one step removed from machine language.

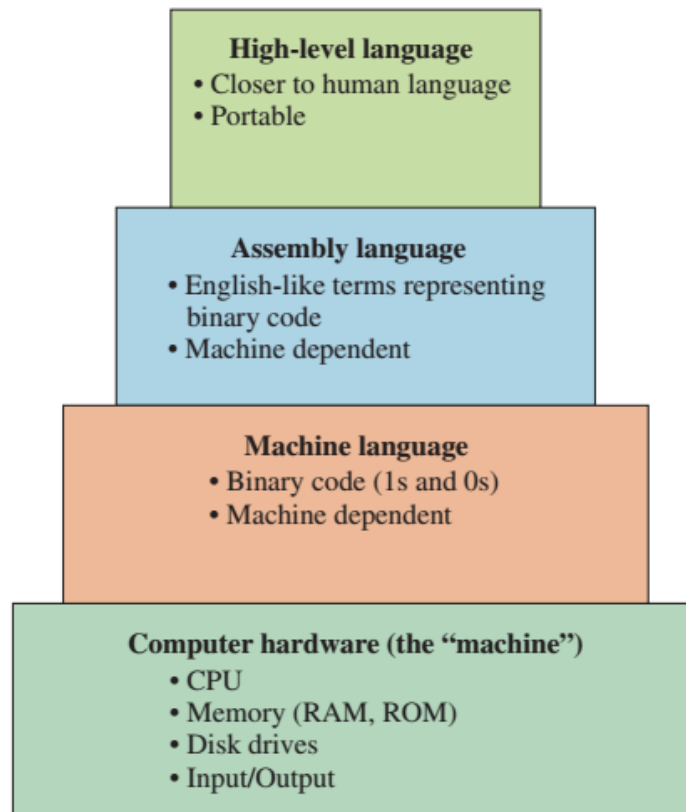


FIGURE 14–28 Hierarchy of programming languages relative to computer hardware.

Assembly language and the corresponding machine language that it represents is specific to the type of microprocessor or microprocessor family. Assembly language is not portable; that is, you cannot directly run an assembly language program written for one type of microprocessor on another type of microprocessor. For example, an assembly program for the Motorola processors will not work on the Intel processors. Even within a given family different microprocessors may have different instruction sets.

An **assembler** is a program that converts an assembly language program to machine language that is recognized by the microprocessor. Also, programs called **cross-assemblers** translate an assembly language program for one type of microprocessor to an assembly language for another type of microprocessor. Assembly language is rarely used to create large application programs. However, assembly language is often used in a subroutine (a small program within a larger program) that can be called from a high-level language program. Assembly language is useful in subroutine applications because it usually runs faster and has none of the restrictions of a high-level language. Assembly language is also used in machine control, such as for industrial processes. Another area for assembly language is in video game programming.

Conversion of a Program to Machine Language

All programs written in either an assembly language or a high-level language must be converted into machine language in order for a particular computer to recognize the program instructions.

Assemblers

An assembler translates and converts a program written in assembly language into machine code, as indicated in Figure 14–29. The term **source program** is often used to refer to a program written in either assembly or high-level language. The term **object program** refers to a machine language translation of a source program.

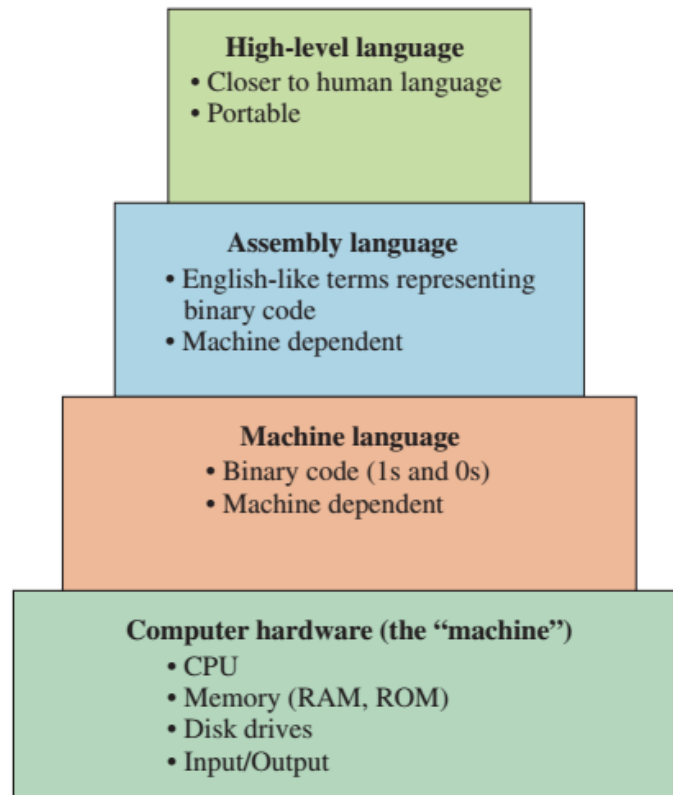


FIGURE 14–28 Hierarchy of programming languages relative to computer hardware.

instructions. All of the mnemonic instructions for a given microprocessor are called the instruction set. Assembly language uses the instruction set to create programs for the microprocessor; and because an assembly language is directly related to the machine language (binary code instructions), it is classified as a low-level language. Assembly language is one step removed from machine language. Assembly language and the corresponding machine language that it represents is specific to the type of microprocessor or microprocessor family. Assembly language is not portable; that is, you cannot directly run an assembly language program written for one type of microprocessor on another type of microprocessor. For example, an assembly program for the Motorola processors will not work on the Intel

processors. Even within a given family different microprocessors may have different instruction sets.

An **assembler** is a program that converts an assembly language program to machine language that is recognized by the microprocessor. Also, programs called **cross-assemblers** translate an assembly language program for one type of microprocessor to an assembly language for another type of microprocessor. Assembly language is rarely used to create large application programs. However, assembly language is often used in a subroutine (a small program within a larger program) that can be called from a high-level language program. Assembly language is useful in subroutine applications because it usually runs faster and has none of the restrictions of a high-level language. Assembly language is also used in machine control, such as for industrial processes. Another area for assembly language is in video game programming.

Conversion of a Program to Machine Language

All programs written in either an assembly language or a high-level language must be converted into machine language in order for a particular computer to recognize the program instructions.

Assemblers

An assembler translates and converts a program written in assembly language into machine code, as indicated in Figure 14–29. The term **source program** is often used to refer to a program written in either assembly or high-level language. The term **object program** refers to a machine language translation of a source program.

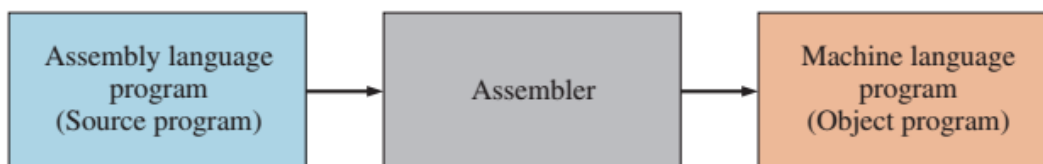


FIGURE 14–29 Assembly to machine conversion using an assembler.

Compilers

A *compiler* is a program that compiles or translates a program written in a high-level language and converts it into machine code, as shown in Figure 14–30. The compiler examines the entire source program and collects and reorganizes the instructions. Every high-level language comes with a specific compiler for a specific computer, making the high-level language independent of the computer on

which it is used. Some high-level languages are translated using what is called an *interpreter* that translates each line of program code to machine language.



FIGURE 14-30 High-level to machine conversion with a compiler.

All high-level languages, such as C++, will run on any computer. A given high-level language is valid for any computer, but the compiler that goes with it is specific to a particular type of CPU. This is illustrated in Figure 14-31, where the same high-level language program (written in C++ in this case) is converted by different machine-specific compilers.

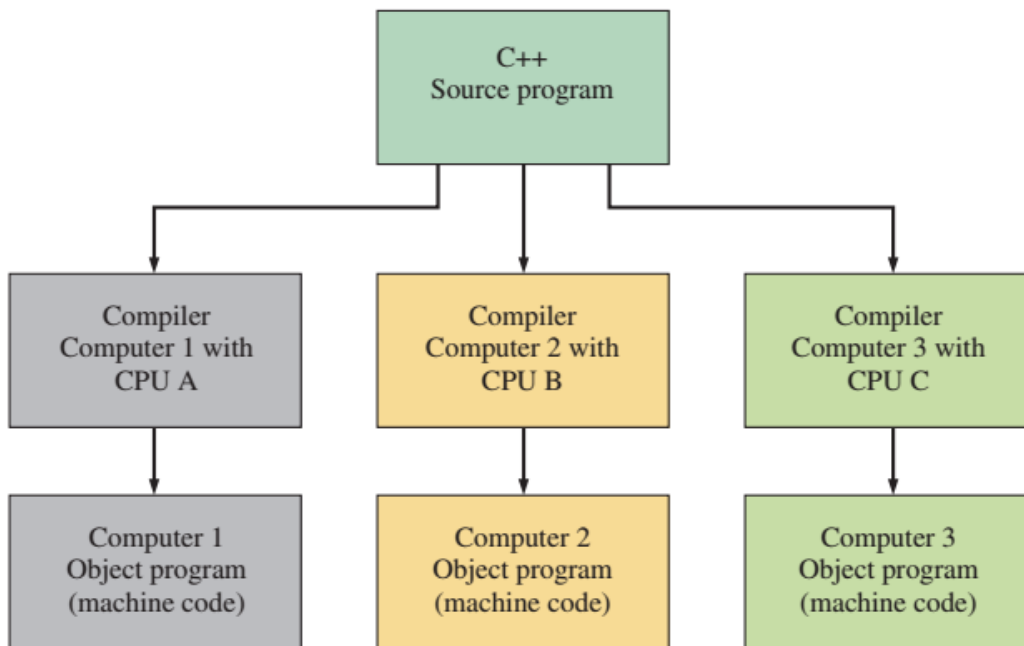


FIGURE 14-31 Machine independence of a program written in a high-level language.

Example of an Assembly Language Program

For a simple assembly language program, let's say that we want the computer to add a list of numbers from the memory and place the sum of the numbers back into the memory. A zero is used as the last number in the list to indicate the end of the list of numbers. The steps required to accomplish this task are as follows

1. Clear a register (in the microprocessor) for the total or sum of the numbers.
2. Point to the first number in the memory (RAM).
3. Check to see if the number is zero. If it is zero, all the numbers have been added.
4. If the number is not zero, add the number in the memory to the total in the

register.

5. Point to the next number in the memory.

6. Repeat steps 3, 4, and 5.

A flowchart is often used to diagram the sequence of steps in a computer program.

Figure 14–32 shows the flowchart for the program represented by the six steps.

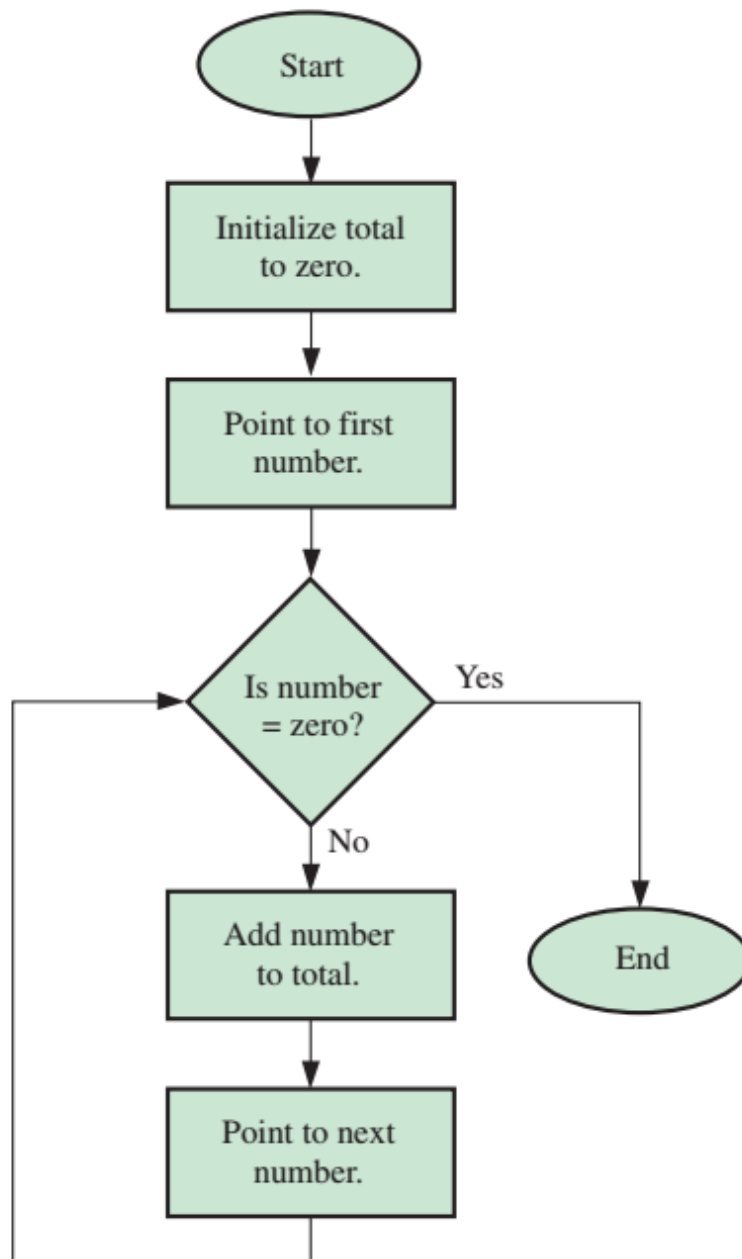


FIGURE 14–32 Flowchart for adding a list of numbers.

The working portion of the assembly language program implements the addition problem shown in the flowchart in Figure 14–32. Two of the registers in the

microprocessor are named `eax` and `ebx`. The comments preceded by a semicolon are not recognized by the computer; they are for explanation only.

```
mov eax,0                ;Replaces the contents of the eax
                          ;register with zero.
                          ;Register eax will store the total of
                          ;the addition.
mov ebx, OFFSET NumArray ;Places memory address of NumArray
                          ;into the ebx register.
next: cmp dword ptr [ebx],0 ;Compares the number stored in the ebx
                          ;register to zero.
      jz done             ;If the number in the ebx register is
                          ;zero, jump to "done".
      add eax,[ebx]        ;Add the number in the ebx register to
                          ;the eax register.
      add ebx, 4
      jmp next
done: mov [ebx],eax
      call WriteInteger    ;WriteInteger utility by Floyd to view
                          ;integer values
      exitProg            ;exitProg utility provided by Floyd
                          ;utility to end the executable
```

Depending on the assembler, most programs in assembly language will have a number of assembler directives that are used by the assembler to do a variety of tasks. These tasks include setting up segments, using the appropriate instruction set, describing data sizes, and performing many other “housekeeping” functions. To simplify the explanation, only two directives were shown in the preceding program. The `word ptr` directive was used to indicate the size of the data pointed to by the `ebx` register, and `OFFSET`.

EXAMPLE 14–2

Write the instructions for an assembly language program that will find the largest unsigned number in the data and place it in the last position. Assume the last data point is signaled with a zero.

Solution

The flowchart is shown in Figure 14–33.

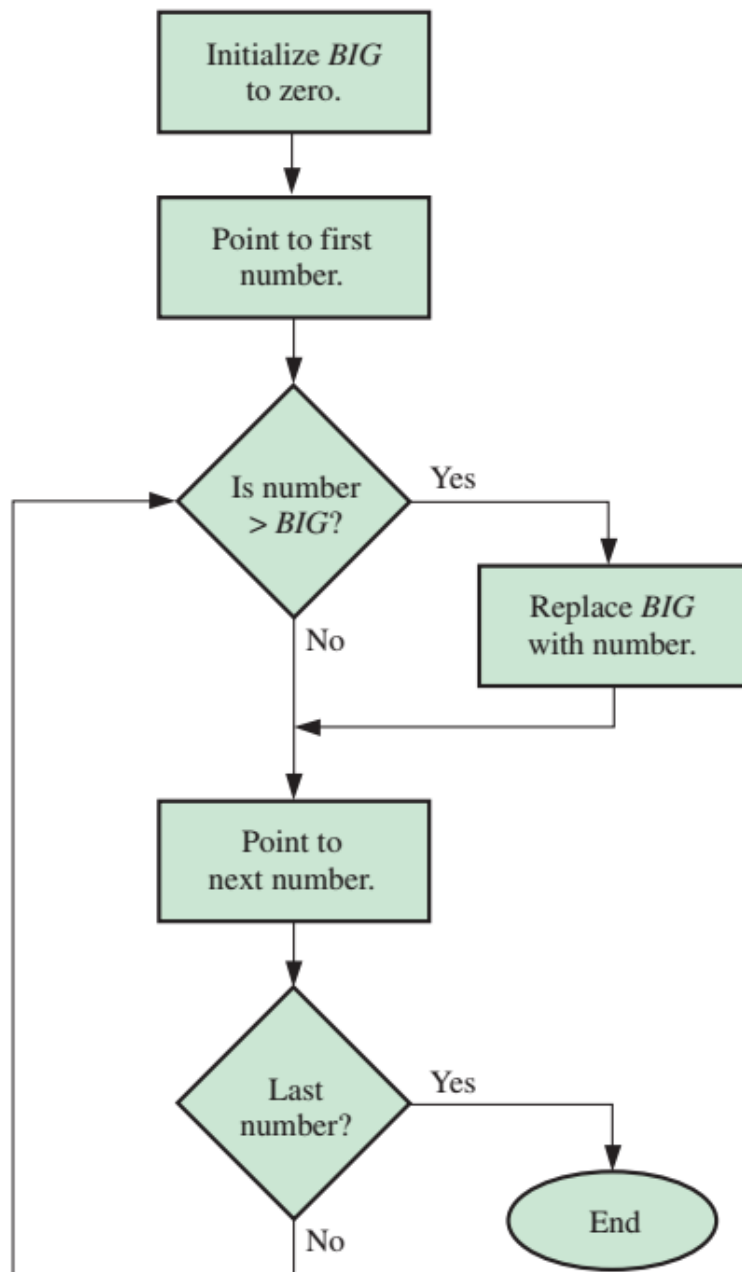


FIGURE 14–33 Flowchart. The variable *BIG* represents the largest value.

The data are assumed to be the same as before. The program listing (with comments) is as follows:

```

mov eax,0                ;initial value of BIG is in the eax
                           register
mov ebx,OFFSET NumArray  ;point to the location in memory where
                           the data are stored
next:  cmp dword ptr [ebx],eax ;is the data point larger than BIG?
        jbe check           ;if the data point is smaller, go
                           to "check"
        mov eax, [ebx]      ;otherwise, put the new largest data
                           point in eax
check:  add ebx,4           ;point to the next number in memory
                           (four bytes per word)
        cmp dword ptr [ebx], 0 ;test for the last data point
        jnz next           ;continue if the data point is not
                           a zero
        mov [ebx], eax     ;save BIG in memory
        call WriteInteger  ;WriteInteger utility by Floyd to
                           view integer values
        exitProg          ;exitProg utility provided by Floyd
                           utility to end the executable

```

Types of Instructions

The programs in this section only show a few of the hundreds of variations of instructions available to programmers. Generally, an instruction set can be divided into categories, which are described here.

Data Transfer

The most basic data transfer instruction MOV was introduced in the example programs. The MOV instruction, for example, can be used in several ways to copy a byte, a word (16 bits), or a double word (32 bits) between various sources and destinations such as registers, memory, and I/O ports. (A better mnemonic for MOV might have been “COPY” because this is what the instruction actually does.) Other data transfer instructions include IN (get data from a port), OUT (send data to a port), PUSH (copy data onto the stack, a separate area of memory), POP (copy data from the stack), and XCHG (exchange).

Arithmetic

There are a number of instructions and variations of these instructions for addition, subtraction, multiplication, and division. The ADD instruction was used in both example programs. Other arithmetic instructions include INC (increment), DEC (decrement), CMP (compare), SUB (subtract), MUL (multiply), and DIV (divide).

Variations of these instructions allow for carry operations and for signed or unsigned arithmetic. These instructions allow for specification of operands located in memory, registers, and I/O ports.

Bit Manipulation

This group of instructions includes those used for three classes of operations: logical (Boolean) operations, shifts, and rotations. The logical instructions are NOT, AND, OR, XOR, and TEST. An example of a shift instruction is SAR (shift arithmetic right). An example of a rotate instruction is ROL (rotate left). When bits are shifted out of an operand, they are lost; but when bits are rotated out of an operand, they are looped back into the other end. These logical, shift, and rotate instructions can operate on bytes or words in registers or memory.

Loops and Jumps

These instructions are designed to alter the normal (one after the other) sequence of instructions. Most of these instructions test the processor's flags to determine which instruction should be processed next. In Example 14–2, the instructions JBE and JNZ were used to alter the path. Other instructions in this group include JMP (unconditional jump), JA (jump above), JO (jump overflow), LOOP (decrement the CX register and repeat if not zero) and many others.

Strings

A **string** is a **contiguous** (one after the other) sequence of bytes or words. Strings are common in computer programs. A simple example is a sentence that the programmer wishes to display on the screen. There are five basic string instructions that are designed to copy, load, store, compare, or scan a string—either as a byte at a time or a word at a time. Examples of string instructions are MOVSB (copy a string, one byte at a time) and MOVSW (copy a string, one word at a time).

Subroutine and Interrupts

A **subroutine** is a mini-program that can be used repeatedly but programmed only once. For example, if a programmer needs to convert ASCII numbers from a keyboard to a BCD format, a simple programming structure is to make the required instructions a separate process and “call” the process whenever necessary. Instructions in this group include CALL (begin the subroutine) and RET (return to the main program).

Processor Control

This is a small group of instructions that allow direct control of some of the processor's flags and other miscellaneous tasks. An example is the STC (set carry flag) instruction.

High-Level Programming

The basic steps to take when you write a high-level computer program, regardless of the particular programming language that you use, are as follows:

1. Determine and specify the problem that is to be solved or task that is to be done.
2. Create an algorithm; that is, develop a series of steps to accomplish the task.
3. Express the steps using a particular programming language and enter them on the software text editor.
4. Compile (or assemble) and run the program.

A simple program will show an example of high-level programming. The following C++ program implements the same addition problem defined by the flowchart in Figure 14–32 and implemented using assembly language.

```
int total = 0;           //Initialize the total to zero.
int *number = NumArray; //Initialize a pointer to the array of integers.
while (*number != 0x00)  //Loop while the value is not found. The
                        //asterisk preceding the pointer identifier
                        //number says the contents of the
                        //memory location pointed to by the
                        //Identifier number are being evaluated.
{
    total = total + *number; //Accumulate summation of total
    number++;               //Increment pointer to next number in memory
}
cout << total;            //C++ cout statement used to view integer value
```

This C++ program is equivalent to the assembly program that adds a series of numbers and produces a total value.

```
int total = 0;
int *number = NumArray;
while (*number != 0x00)

{
    total = total + *number;
    number++;
}

cout << total;
```

[C++]

 Equivalent

```
mov eax, 0
mov ebx, OFFSET NumArray
next: cmp DWORD PTR [ebx], 0
      jz done

      add eax, [ebx]
      add ebx, 4

      jmp next
      mov [ebx], eax
done: mov [ebx], eax
      call WriteInteger
```

Assembly

Introducing embedded systems and the microcontroller (Last Chapter)

We are living in a second age of industrial revolution. Nowadays the **availability and processing of information are causing untold changes in all our lives.**

Mankind has dreamed for many years of the possibility of building computing machines, the dream started to become a reality in the late 1940s.

In 1948 the transistor was invented, and the first integrated circuit was built in 1959. This set the stage for a spectacular process of electronic miniaturization.

In 1971 Intel produced the first microprocessor, the 4004, which handled data as 4-bit numbers, and contained 2250 transistors. With this the age of the microprocessor had arrived!

By the end of the 1970s, two trends were emerging for these microprocessors.

One was to scale down, in size if not computing power, the general-purpose computer; this led quickly to the first desktop machines.

The other, much more revolutionary, was to place the microprocessor in products which apparently had nothing to do with computing. They began to find their way into photocopiers, grocery scales, washing machines, and a host of other products, wherever there was a requirement to exercise some control function. While the first trend led to an inexorable demand for faster and bigger processors with increasingly sophisticated mathematical capability, the second placed lower demands on computational power and speed. It wanted physically small and cheap devices, with as much functionality of the system as possible squeezed onto one integrated circuit.

Such microprocessors became known as microcontrollers, and the systems they controlled, embedded systems. Microcontrollers sell in far greater volume, and their impact has been enormous. To the electronic and system designer they offer huge opportunities.

Aims of this chapter:

1. to introduce the embedded system and describe its characteristics
2. to review prerequisite microprocessor knowledge
3. to consider certain fundamental choices in microprocessor design
4. to introduce the features of a general-purpose microcontroller
5. to introduce three microcontroller families.

1.1 Embedded systems and their characteristics

The essence of the embedded system

Cambridge engineer Paul Ford has fitted a home-designed jet engine to his bicycle and created a potentially record-breaking machine capable of travelling at 100 mph.



Figure 1.1 A high-speed embedded system: the jet-propelled bicycle (Paul Ford on his jet powered bike).

The above figure shows a jet-propelled bicycle. Attention of common people was drawn entirely by the novel combination of a jet engine and a bike. The microcontroller used for the success of the engine was not visible; there was no indication of the presence of a computer. **However, the engine could not have functioned for more than a few seconds without the continuous action of the control system, which not only enabled successful operation, but also provided the condition monitoring to eliminate situations of danger.**

We call this type of control *embedded control* – and the overall system an embedded system. A definition of an embedded system is as follows:

An embedded system is a system whose principal function is not computational, but which is controlled by a computer embedded within it.

The computer is likely to be a microprocessor or microcontroller. The word embedded implies that it lies inside the overall system, hidden from view, forming an integral part of a greater whole. One consequence of this is that the user may be unaware of the computer's existence. Another is that the computer is usually purpose designed, or at least customized, for the single function of controlling its system. If removed from the system it would be an odd assortment of printed circuit boards and/or integrated circuits, recognizable only to the specialist as something which might be called a computer.

Applying this definition tells us that a personal computer, even though it contains a microprocessor, is not an embedded system. Its end function is to compute. Even if the same computer was connected to a set of instruments, which it then controlled, that would not be an embedded system. If, however, the same computer was built permanently into an identifiable system, and customized so that its sole purpose was to control the one system (which may mean losing such apparently essential features as its case, keyboard, screen, or disk drives), then it would form part of an embedded system. Embedded systems come in many forms. They are extremely common in the home, the motor vehicle and the workplace. Most modern domestic appliances – washing machines, dishwashers, ovens, central heating and burglar alarms – are embedded systems. The motor car is full of them, in engine management, security (for example locking and anti-theft devices), air-conditioning, brakes, radio, and so on. They are found across industry and commerce, in machine control, factory automation,

robotics, electronic commerce and office equipment. The list has almost no end, and it continues to grow.

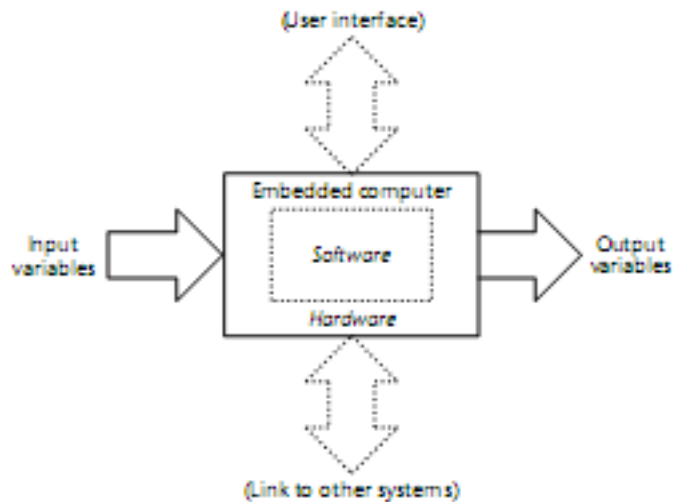


Figure 1.2 The essence of the embedded system.

Figure 1.2 re-expresses the embedded system definition as a simple block diagram. There is a set of inputs from the controlled system. Based on information supplied from these inputs, the controller computes certain outputs, which are connected to actuators within the system. There may be interaction with a user, e.g. via keypad and display, and there may be interaction with other sub-systems elsewhere, though neither of these is essential to the general concept.

In the jet-propelled bicycle the control system measures three variables from the engine (temperature, pressure and rotational speed), and also receives a control input from the driver. Its only output controls the fuel flow to the engine. From the inputs it first of all determines whether the engine is operating safely. If a danger condition is detected (for example the motor is running too hot or too fast), the controller takes emergency action. In the absence of a danger condition, it computes the appropriate drive signal for the fuel flow.

1.1.2 Further features of the embedded system.

1.1.2.1 Constituents of the embedded computer: hardware and software

As with all computer systems, the embedded computer is made up of hardware and software, as symbolized in Fig. 1.2. Design of the computing core of the embedded system, in many cases viewed as a comparatively straightforward affair. The design attention now has shifted to some extent towards software development, with advanced languages and tools available to develop sophisticated programs.

1.1.2.2 Timeliness

The example jet engine is able to change its speed extremely fast, and can easily self-destruct. The controller must be able to respond fast enough to keep its operation within a safe region. This is a characteristic of operating in ‘real time’; the controller must be able to respond to inputs as they happen and make responses within the time-frame set by the controlled system. This style of operation is different from the mode of operation, for example, of a personal computer. While it may be annoying, you can tolerate waiting for your computer to refresh the graphics display or complete a computation. You cannot tolerate waiting while your car’s antiskid braking system decides whether or not to apply the brakes! Some embedded systems operate within absolutely rigid time demands; for others

the demands are less stringent. They all, however, exhibit the characteristics of timeliness: a need for the designer to understand fully the time demands of the controlled system and be responsive to them.

1.1.2.3 System interconnection

Figure 1.2 raises the possibility of interaction with other systems. While some embedded systems clearly need only one controller, others are likely to use several or many, each to control one sub-system. Necessary shared information is then passed between them by a simple network, devised to suit the needs of the overall system. A good example of this is the modern motor car. Though each of the ‘embedded sub-systems’ in it may be controlled by one microcontroller, they can all be linked together to form one overall interconnected system. This approach is made more attractive due to the extremely low cost of most commercial microcontrollers. A network of low-cost microcontrollers is often cheaper, and simpler to develop, than a single complex computer undertaking many tasks. With the advent of the Internet, a generation of Internet-compatible embedded systems is emerging. The cooker, television and washing machine may soon be communicating together! It is anticipated that within a few years even the most simple of devices may be Internet-linked. The truly standalone device will then exist in a dwindling minority.

1.1.2.4 Reliability

Suppliers of software packages designed to run on Personal Computers release them on the market knowing that they are likely to contain software errors (bugs). It is vitally important to get them to market early, and fixes can always be distributed after the faults have been discovered. Suppliers of most embedded systems cannot afford this luxury. One significant software error in a car model could destroy the reputation of the manufacturer for ever. Therefore the embedded system designer must develop a good grasp of reliability issues, and how a reliable system can be achieved. This implies good design procedures in both hardware and software, coupled with systematic testing and commissioning.

1.1.2.5 The market-place

The market that the embedded system sells into is very competitive. As with other ‘hi-tech’ markets, the challenge is increased greatly by the very rapid advances of technology. New products may quickly be rendered obsolete by technological change, and thus potentially have very short life cycles. This lays the stress on excellent design and development strategy. Adding all these features together, a second definition of the embedded system now follows, more descriptive and verbose.

An embedded system is a microcontroller-based, software-driven, reliable, real-time control system, autonomous, or human or network interactive, operating on diverse physical variables and in diverse environments, and sold into a competitive and cost-conscious market.

1.1.3 The skills of the embedded system designer

It is becoming clear that embedded systems have enormous variety, and call upon many technical disciplines. This is indeed one of the attractions of working with them. This multi-disciplinary nature is illustrated in Fig. 1.3. The need for control, which inevitably implies measurement and actuation, leads us into further branches of electrical and electronic engineering. Associated with the measurement, we find a need for analogue as well as digital electronics. One could go on adding further disciplines, for example Digital Signal Processing or Electromagnetic Compatibility, to the diagram.

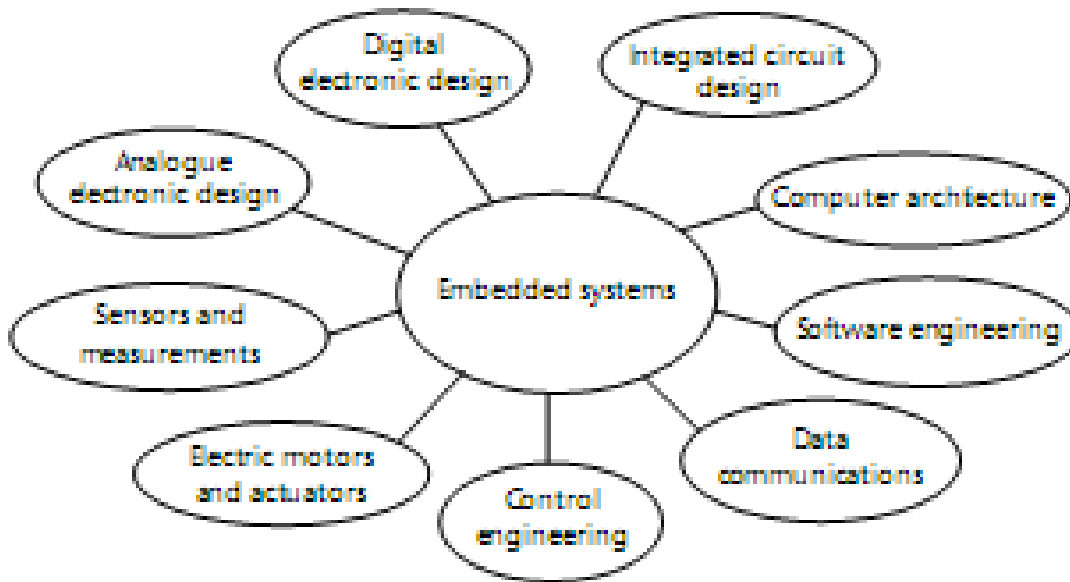


Figure 1.3 Embedded system design calls on many disciplines.

1.2.1 The microprocessor reviewed

Our approaching study of the microcontroller will rely on the reader having a reasonable knowledge of microprocessors. We will pause briefly to review this knowledge, to ensure a defined starting point. Figures 1.4–1.6

summarize what we need to know. The contents of each diagram will be briefly reviewed. Both of these have excellent introductory chapters on microprocessors. A microprocessor is a simple computer, contained more or less in one integrated circuit (IC, also colloquially called a ‘chip’). Like any computer it follows a sequence of instructions, known as a program. Each instruction causes a very simple action to take place, generally either a computation, a transfer of data or a decision. The microprocessor can perform each instruction extremely fast, so that by building on these very simple actions much more complex tasks can be undertaken.

A diagram of the hardware of a simple microprocessor-based system is shown in Fig. 1.4. The essential features are:

- the microprocessor
- a section of memory to store the program
- another section to store temporary data
- some contact with the outside world (through the input/output port)
- a means of interconnecting these elements (i.e. data and address bus, together with some control lines)

Program memory is usually stored in a form of memory called ROM –Read-Only Memory. Data memory is usually stored in a type of memory called RAM – Random Access Memory. ROM retains its contents when the system is powered down; RAM does not. Memories are defined according to size, generally in terms of numbers of bytes. For this the prefixes K- and M-(or Mega) have gained ubiquitous customary usage. These

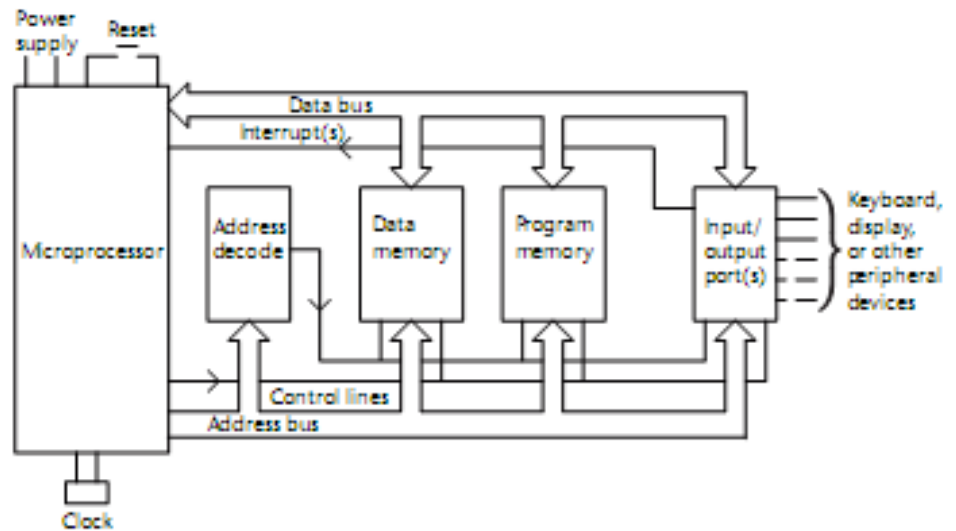


Figure 1.4 A simple microprocessor system.

differ from the

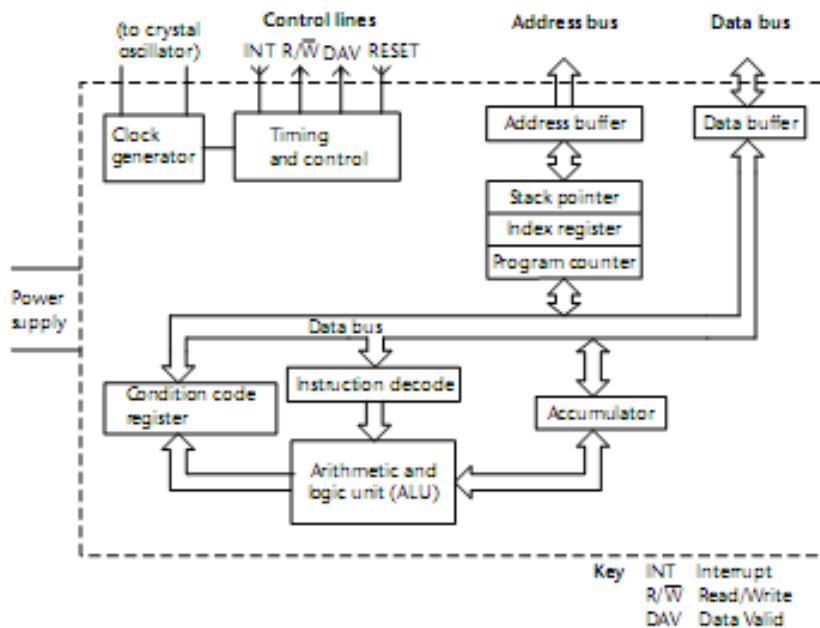


Figure 1.5 A typical microprocessor.

A block diagram of a ‘typical’ imaginary microprocessor appears in Fig. 1.5. The computing function takes place in the Arithmetic Logic Unit (ALU), where arithmetic and logical operations take place. Part of the ALU is the accumulator. This is the register where the operand, the number on which the operation is being performed, is held. The size of the Accumulator, in number of bits, determines the size of number that the processor can operate on. It is reflected across the whole microcomputer system, for example in the size of the data bus and memory locations. The ALU, together with the control section around it, is known as the Central Processing Unit (CPU).

The action of the microprocessor is synchronized to the clock generator, often based on a quartz crystal oscillator. Any microprocessor can only operate within a certain range of clock frequencies, whose limits are set by the fabrication technology of the device and specified by the manufacturer. Each has a maximum (for microcontrollers usually in the range 4 MHz to around 30 MHz). Those based on dynamic logic have a minimum as well. Those which can operate down to DC are known as ‘fully static’.

The clock oscillator frequency is divided down within the microprocessor (generally by a factor between 4 and 12, depending on the microprocessor), giving a lower internal operating frequency. One period of this internal frequency is sometimes called a machine cycle, or an instruction cycle. All instruction execution is made up of integer numbers of machine or instruction cycles.

In normal system operation the processor works down the list of instructions which make up the program. It fetches each one from program memory, decodes it with its Instruction Decode circuit, and then executes it. The instruction is in many cases accompanied by further pieces of code, also stored in program memory, which are treated as operand data, or addresses where the operand data may be found. The microprocessor 'keeps its place' in the program by means of the Program Counter, which always holds the address of the next instruction to be executed. In order to fetch the next instruction, the processor places the value held in the Program Counter on the address bus, and signals through the control lines that it wishes to read data. Memory corresponding to that address will, upon receiving the address and control signals, place the instruction word on the data bus, which the processor can then read. As each word is read from program memory, the Program Counter is incremented.

Figure 1.6 illustrates this sequence of activities, for the processor of Fig. 1.5 and for a certain instruction, as a timing diagram. It can be seen that there are four clock cycles in each machine cycle. The first cycle shown is an 'instruction fetch' cycle. The address of the instruction to be fetched is placed on the address bus, and the R/W line indicates that the data transfer is to be a 'read'. In response the addressed memory places data onto the bus. This is received by the microprocessor and decoded by the Instruction

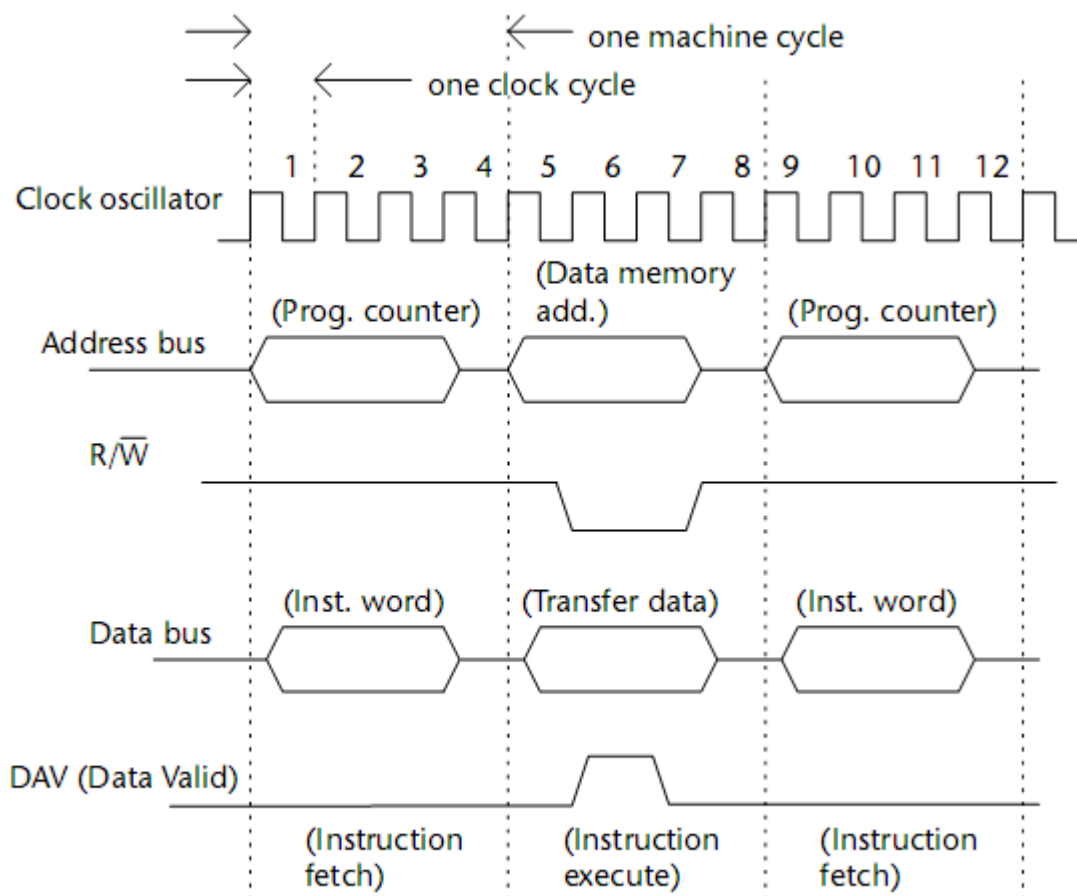


Figure 1.6 The microprocessor fetch/execute cycle.

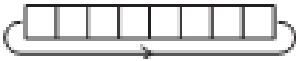

Decode circuit. In the second machine cycle the instruction is executed; the example illustrates a data move from processor to memory. The processor sets values on the address and data buses, and signals a write by

setting the R/W line low. The DAV line goes high to indicate that the bus data is valid. The falling edge of this signal is used to latch the data into memory. This particular instruction has taken two machine cycles to complete. It is then followed by the Instruction Fetch cycle of the next instruction. It follows that simple microprocessor operation can be seen as a relentless cycle of instruction fetch, decode and execute.

1.2.2 More on instructions, and the ALU

A typical 8-bit ALU is able to perform the operations shown in Table 1.1. Using combinations of these very simple operations, almost any other mathematical function can be implemented, albeit sometimes laboriously. Each processor (or processor family) has its own instruction set, from which the program is written. Each instruction is a binary word, known individually as the op code (operation code), or collectively as machine code. The processor CPU can recognize and respond to these codes. The instruction set is the collection of all these op codes. It uses the basic ALU operations listed earlier, and adds to these certain data transfer and branch instructions. This gives an instruction set the following typical instruction categories:

Table 1.1 What an ALU can do.

Increment A	$A = A + 1$
Decrement A	$A = A - 1$
Add A to M	$A = A + M$
Subtract M from A	$A = A - M$
*AND A with M	$A = A \cdot M$
*OR A with M	$A = A + M$
*Exclusive OR A with M	$A = A \oplus M$
Shift A left	$A = 2A$
Shift A right	$A = A/2$
Rotate A left	
Rotate A right	
Complement A	NOT A
Clear A	$A = 0$

A represents the contents of the accumulator; M is a number held in memory.
The statement 'A = ' implies 'A becomes' (original value of the accumulator overwritten).
*The logical function is performed between corresponding bits of the two operands.

A represents the contents of the accumulator; M is a number held in memory.

The statement 'A = ' implies 'A becomes' (original value of the accumulator overwritten).

*The logical function is performed between corresponding bits of the two operands.

- *Data transfer*: instructions which move data from one register or memory location to another.
- *Arithmetic*: instructions which perform arithmetic operations between specified data words.
- *Logical*: instructions which perform logical functions between specified data bits or words, for example INVERT, AND, OR, Rotate.
- *Program branch*: instructions which cause a program to deviate from simple sequential execution of instructions held in program memory, for example as a subroutine call or return, or conditional branch.

The result of an operation undertaken in an accumulator frequently exceeds the range of the number which can be held in the Accumulator. Therefore associated with the ALU is a 'Flag Register'; this contains a number of bits which give further information about the result of the previous instruction. It is known as the Status Register (Microchip Inc.), Condition Code Register (Motorola), or Program Status Word (Intel and Philips). These bits may include:

- **a zero bit**, indicating whether the result was zero
- **a carry bit**, indicating whether there was a carry from the most significant bit (msb) of the accumulator, also used as a ‘borrow’ in subtraction
- **a sign, or negative bit**, indicating whether the result was negative (interpreting the result in two’s complement arithmetic) – hence this bit is simply set to the msb of the result
- **a half-carry bit**, indicating whether there was a carry between the lower and higher nibbles of the result – this is useful for Binary Coded Decimal (BCD) arithmetic
- **an overflow bit**, indicating whether the two’s complement range has been exceeded. It is set if there has been a carry out of bit 7 but not bit 6, or a carry out of bit 6 but not bit 7
- **a parity flag**, indicating whether an odd or even number of 1 bits are in the accumulator

As there are not usually enough ‘condition code’ flags to fill an 8-bit register, many processors use the remaining few bits for other purposes, for example interrupt mask bits or register bank address bits.

1.3 Some microprocessor design options

We go on to consider some aspects of microprocessor design which go beyond the basic structure assumed so far. These aspects are discussed at a level appropriate to the small-scale microprocessor or controller; their application in larger computers is far more sophisticated. Readers who wish to gain further background in these areas are referred to Refs. 1.4 and 1.5.

1.3.3 Instruction pipelining

As Fig. 1.6 showed, conventional microprocessor program execution is a relentless sequential cycle of instruction fetch, decode and execute. For a given processor the only way of speeding up this operation is by speeding up the clock.

Consider an alternative: that as one instruction is being executed, the next is already being fetched. If this is done, the instruction throughput can be dramatically increased without reducing the actual instruction execute time. This is the basis of pipelining – it’s a simple idea which can make processors run much faster, but it does place certain strict requirements on the nature of the instructions.

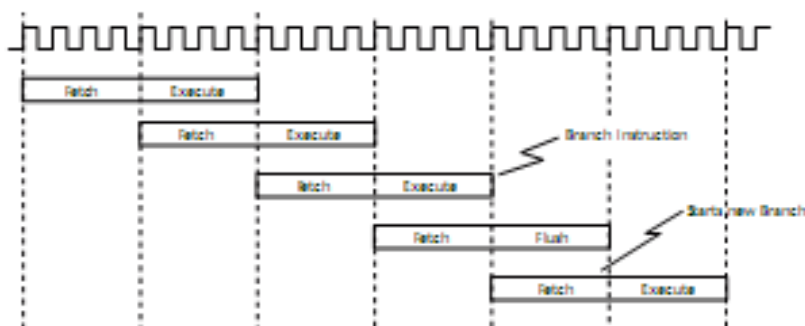


Figure 1.9 Pipelined instruction execution.

In order to work, all the instructions of the processor must have the same duration of execution, and it must be possible to split the fetch–decode–execute cycle for all individual instructions into a number of stages of equal duration. Then, as any one instruction enters its second stage, the following instruction enters its first. This is illustrated in Fig. 1.9, for instructions divided into just two stages (i.e. fetch and execute). As one instruction is executing, the next is already being fetched. It can be seen that the instruction throughput for the first three instructions is twice as fast as in Fig. 1.6. If the instructions had been broken into three stages, it would have been three times as fast, and so on.

Simple pipelining fails at conditional program branches. When the processor is executing a branch instruction, it is already fetching the next instruction in the program, but if the branch does take place that next instruction is no longer needed. So it must 'flush out' that instruction, and fetch the one where the branch starts. This is why branch instructions often take longer in a pipelined architecture. The example of Fig. 1.9 shows two instructions being successfully fetched and executed. The third is a branch, and the fourth instruction, though fetched, is never executed, and one machine cycle is lost. The fifth instruction shown is from the start of the program section to where the branch has taken place.

1.4 The microcontroller: its applications and environment

A microcontroller is a particular type of microprocessor, optimized to perform control functions for the lowest cost and at the smallest size possible. Generally microcontrollers are used in a recognizably 'embedded system' environment. There is a huge range of microcontroller applications. Some are drawn from volume markets – the motor car, domestic appliances, mobile phones and toys. These applications are sold in such high volume that dedicated controllers are frequently developed for them. Others, like medical or scientific instruments, are sold in smaller numbers, and are more likely to make use of the wide variety of general-purpose controllers that are available. At one extreme of complexity, simple (and very cheap) controllers are used to replace 'glue logic' in a digital system. At the other extreme, advanced 32-bit controllers perform sophisticated signal processing activities.

1.4.1 Microcontroller characteristics

Arising from their 'embedded control' environment, microcontrollers usually have the following features:

- input/output intensive, i.e. they are capable of direct interface to a significant number of sensors and actuators
- a high level of integration, with many peripheral devices included 'on-chip'
- physically small
- comparatively simple program and data storage requirements
- ability to operate in the real-time environment
- an instruction set optimized for the embedded environment, e.g. yielding compact code, limited arithmetic and addressing capability, strong in bit manipulation
- low cost

In many microcontroller applications either or both of the following features are also essential:

- an ability to operate in hostile environments, for example of high or low temperature, or high electromagnetic radiation;
- a low power capability, and features which ease the use of battery power.

In today's fast-moving world, both the manufacturers of microcontrollers and the people who design with them work under a complex set of sometimes conflicting forces. On the one hand, semiconductor technology is advancing inexorably. Every year it becomes possible to integrate more onto a single IC, and to do this more cheaply, with the chip operating at faster speed and lower power. Interacting with this technological change are powerful market forces, spurring on the development by demanding new capabilities from the microcontrollers. Against this, however, is set a certain conservative tendency. Companies using the microcontrollers have invested time and money in supporting work with a particular device, and don't want all this wasted when they move on to its more powerful successor.

The outcome of all this is that the manufacturer usually develops a family of microcontrollers all based around one core, where the core contains the CPU and its surrounding control features (i.e. essentially the features of the early microprocessor, as in Fig. 1.5). The core defines the instruction set, and hence keeping the core design constant ensures software compatibility between different members of a processor family. To the core, and on the same IC, can be added the peripheral devices which seem best to meet a particular need. Even though the microprocessor world is one of such great change, many microcontrollers can trace their history very directly back for over 20 years! Once a company has committed itself to designing with a particular microcontroller family, it is reluctant to change, but looks to the manufacturer to supply it with the necessary technological advances, based around a familiar core. Infrequently, the manufacturer makes a step change by introducing a new core.

1.4.2 Features of a general-purpose microcontroller

Every microcontroller is different, and each has its own unique combination of core and peripherals. Figure 1.10 shows, in block diagram form and with no interconnections, the features which might be found in a simple general-purpose controller. The core is the element that remains constant for the whole family built around it. Ideally all memory is on-chip, and several different memory technologies may be applied to meet the differing needs of program and data storage. Interconnection to the outside world is through a number of parallel and serial ports. A counter/timer is available for event counting, or to measure or generate timing intervals.

1.4.3 Some example controllers

There are a huge number of different microcontrollers now on the market, and it is not easy to select a small number of representative devices to illustrate the hardware principles described in this text. Three example microcontroller families have been chosen, and in keeping with the introductory nature of the book they are all 8-bit devices. The families selected are well established in industry, and are chosen to illustrate the variety of approaches taken to solve common problems. Each one carries at least one

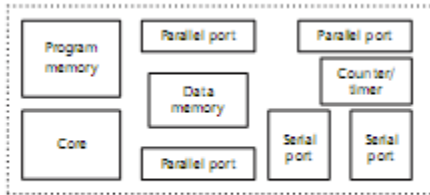


Figure 1.10 An example microcontroller block diagram.

feature not shared with the others. The example controllers are not, however, selected or treated as equals. First of all, the PIC 16F84 device is very small and low-cost, and used for some of the simplest possible embedded systems. The 80C552, on the other hand, is a comparatively complex (and more costly) device, rich in peripherals and with extensive memory addressing capability. The 68HC05 and 68HC08 lie in complexity between these two. Second, to encourage in-depth knowledge to be built up of one microcontroller, most early examples are of the PIC 16F84. Later examples are drawn more generally from the microcontrollers named, as well as one or two other close relatives.

It should be understood that the selection of these devices as examples is not intended to compare them in a competitive way (in the sense of seeking out ‘the best’), nor does it necessarily represent an endorsement of any one of them.

It is recommended that the reader obtains at least the full data sheet of the 16F84 (Ref. 1.6). It will also be useful, but not essential, to have access to data on one or more of the others (Refs. 1.7 and 1.8). There are also many very useful Application Notes, published by the manufacturers, as well as books targeted towards individual devices, for example Refs. 1.9 and 1.10. The remainder of this chapter introduces the three example families, looking particularly at their background, architecture and CPU. In the coming chapters we build on this introduction by looking at certain features of these microcontrollers in greater detail.

1.5 Microchip Inc. and the PIC™ microcontroller

1.5.1 Background, and meet the family

It was the General Instruments Corporation, back in the late 1970’s, that first produced the PIC microcontroller (Ref. 1.11). In its early years it did not make a wide impact. The design was later taken over by Microchip Inc., and PICs are now one of the fastest moving families in the 8-bit arena, in more senses than one. First, they run very fast; second, the family is growing at a tremendous rate; and third, at the time of writing Microchip only operates with 8-bit controllers, and therefore has a special interest in making this controller size as attractive as possible. PICs cover a very wide range of 8-bit operation. At the lower end, they are simpler, cheaper and smaller than most devices that the competition can offer, and are thus used in situations where controllers would not be thought of as the right solution, even down to simple glue logic applications. At the high end, however, they are quite ready to take on the best of the 8-bit competition, with sophisticated devices equipped with excellent peripherals. PICs have made themselves particularly attractive to the student and low-budget developer. Development tools (both hardware and software) are cheap and readily available, and Microchip is very supportive of the novice designer.

Table 1.2 PIC microcontroller families.

Family	Program word size	Number of instructions	Minimum instruction execution time
12CXX	12/14-bit	33/35	400 ns
16C5X	12-bit	33	200 ns
16C/FXX	14-bit	35	200 ns
17CXX	16-bit	58	120 ns
18CXX	16-bit (enhanced)	77	100 ns