CHAPTER 14

Data Processing and Control

CHAPTER OUTLINE

CHAPTER OBJECTIVES

- Name the basic units of a computer
- Name the computer buses and how they are used
- Discuss the considerations for a practical computer system
- Describe the purpose of buffers, decoders, and wait-state generators in a computer system
- Define and explain the advantage of DMA
- Name the basic elements of a microprocessor
- Describe the basic architecture of a microprocessor
- Explain basic microprocessor (CPU) operation
- List and describe some microprocessor addressing modes
- Define and describe microprocessor polling, interrupts, exceptions, and bus requests
- Discuss the operating system of a computer
- Explain pipelining, multitasking, and multiprocessing
- Describe a simple assembly language program
- List some typical microprocessor instructions
- Distinguish between assembly language and machine language
- Describe the architecture of a microcontroller and explain how it differs from a microprocessor
- Discuss embedded systems
- Discuss some microcontroller applications
- Describe a system on chip (SoC)

KEY TERMS

Key terms are in order of appearance in the chapter.

- CPU
- Microprocessor
- Main memory
- Caching
- BIOS
- System bus
- Signal loading
- Buffer
- Wait state
- Pipelining
- ALU
- Program
- Op-code
- Operand
- Interrupt
- Exception
- Interrupt vector table
- Bus master
- DMA
- Hardware
- Software
- Operating system
- Multitasking
- Multiprocessing
- Machine language
- Assembly language
- High-level language
- Microcontroller
- System on chip

VISIT THE WEBSITE

Study aids for this chapter are available at
http://www.pearsonglobaleditions.com/floyd

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. Microcontrollers and system on chip (SoC) are also introduced, and some applications are described.

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.

After completing this section, you should be able to

- Describe the basic elements of a general-purpose computer
- Discuss each part of a computer
- Explain a peripheral device

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

Specific sections and components in computer systems accomplish each of these tasks. 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.

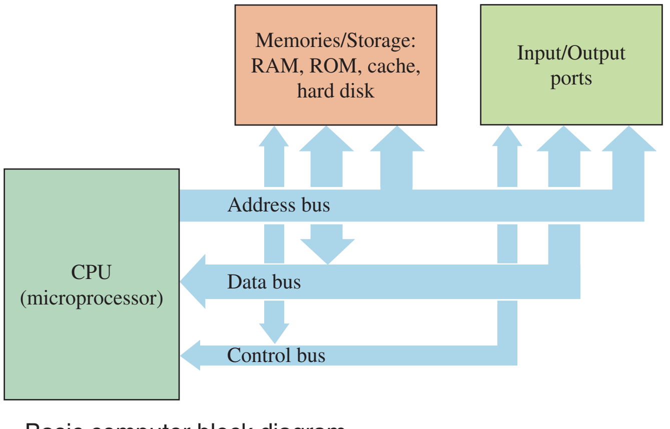


FIGURE 14–1 Basic computer block diagram.

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. To play such a large role in the computer system, the CPU consists of four separate units: the arithmetic logic unit (ALU), the instruction decoder, the timing and control unit, and the register set. 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. Although the usage and meanings of the terms can overlap somewhat, they primarily differ in the construction of the devices and the information they contain. 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 InMOS 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.

The earliest personal computers used read-only memory (ROM) for BIOS, so any change to the BIOS required the user to replace the ROM chip (which was often socketed) itself. Later computers used 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. **Firmware** is software programs or data that have been written into ROM.

Content-Addressable Memory

Computers often use specialized types of memory in addition to those types mentioned previously. One specialized type of memory is the content-addressable (or associative) memory, whose operation differs from that of conventional memory. Conventional memory returns the data stored at a specified address. Content-addressable memory returns the address that contains a specified data value. Computers use content-addressable memory for special data tables that support caching and paging operations.

FIFO

Another specialized type of memory is the FIFO (first-in, first-out) memory. Conventional memory, such as SRAM and DRAM, allow computers to store data and to retrieve data from any memory location in any order. FIFO memory returns data only in the order in which the data were stored. As the acronym *FIFO* indicates, the first data stored in memory must be the first data taken out of memory. Computers use FIFO memory for special data structures called **queues**. Queues temporarily store data for which the sequence of data must be preserved, such as program instructions.

Input/Output Ports

Input/output (or I/O) ports are interfaces that allow computers to transfer data to and from external entities such as users, peripherals (such as mice, keyboards, video monitors, scanners, printers, modems, and network adapters), and other computers. I/O ports vary greatly in complexity and capability. An I/O port can be serial or parallel, operate as an input, output, or both, and transfer several thousand to several billion bits per second. Many I/O ports, such as RS-232, USB 3.0, SCSI-5, Firewire, and Ethernet ports, conform to official or de facto standards to simplify computer system connections. These standards are usually developed by international organizations and typically specify not only the type of connectors but also the pin assignments, electrical signal levels, signal timing, data transmission rates, and communication protocols (i.e., the format, organization, and meaning of data patterns). EIA 802, for example, is the international standard for Ethernet communications and IEEE 1394 is the standard for Firewire. These standards ensure that all devices that comply with the standard will be able to communicate with each other.

Processors support I/O ports and operations in one of two ways. One way is memory-mapped I/O, in which the processor treats I/O ports as memory locations and external circuitry converts standard read and write operations into I/O port accesses. The second way is direct I/O, in which specific processor pins and instructions are exclusively dedicated to data input and output operations. In either case, general-purpose processors require additional circuitry and program code to implement specific communications standards and protocols. Specialized microcontrollers like the Motorola MC68360 and NXP LPC2292 improve on this by incorporating additional circuitry and embedded firmware to support

UART, I2C, Ethernet, CAN, SPI and other popular communication standards on their I/O ports with a minimum of driver coding and external interface circuitry.

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.

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

InfoNote

Grace Hopper, a mathematician and pioneer programmer, developed considerable troubleshooting skills as a naval officer working with the Harvard Mark I computer in the 1940s. She found and documented in the Mark I's log the first real computer bug. It was a moth that had been trapped in one of the electromechanical relays inside the machine, causing the computer to malfunction. From then on, when asked if anything was being accomplished, those working on the computer would reply that they were "debugging" the system. The term stuck, and finding problems in a computer (or other electronic system), particularly the software, would always be known as debugging.

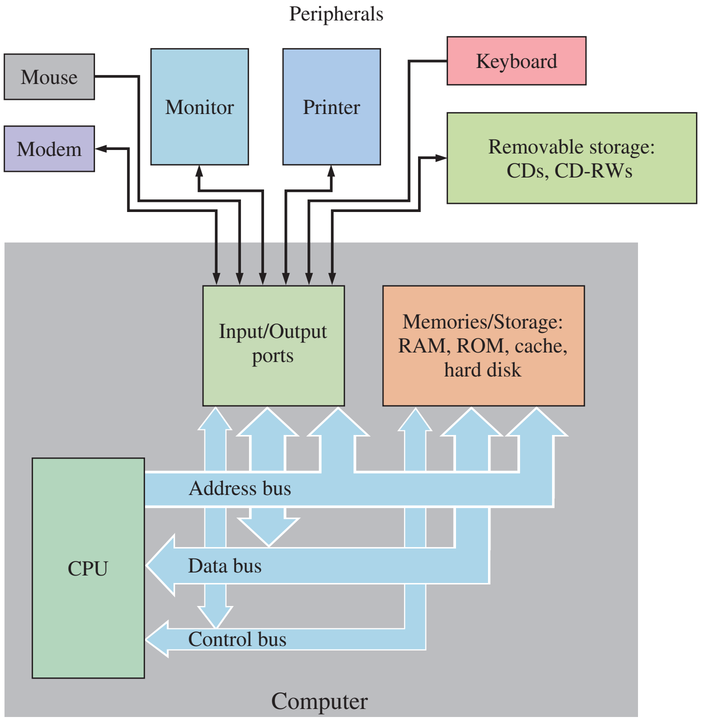


FIGURE 14–2 Basic block diagram of a typical computer system including common peripherals. The computer itself is shown in the gray block.

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.

SECTION 14–1 CHECKUP

Answers are at the end of the chapter.

1. What are the major functional blocks in a computer?
2. What are peripherals?
3. What is the difference between computer hardware and computer software?
4. How does content-addressable memory differ from conventional memory?
5. Compare and contrast the characteristics of the address, data, and control buses in a computer system.

14–2 Practical Computer System Considerations

Practical computer designs incorporate special circuitry that resolves four issues that exist in real-world systems: shared signal lines, signal loading, device selection, and system timing.

After completing this section, you should be able to

- Identify design considerations for practical computer systems
- Explain the role and operation of buffers, decoders, and wait-state generators in practical computer systems

Figure 14–3 shows a block diagram of a practical computer system, based on the consideration for shared signal lines, signal loading, device selection, and system timing.

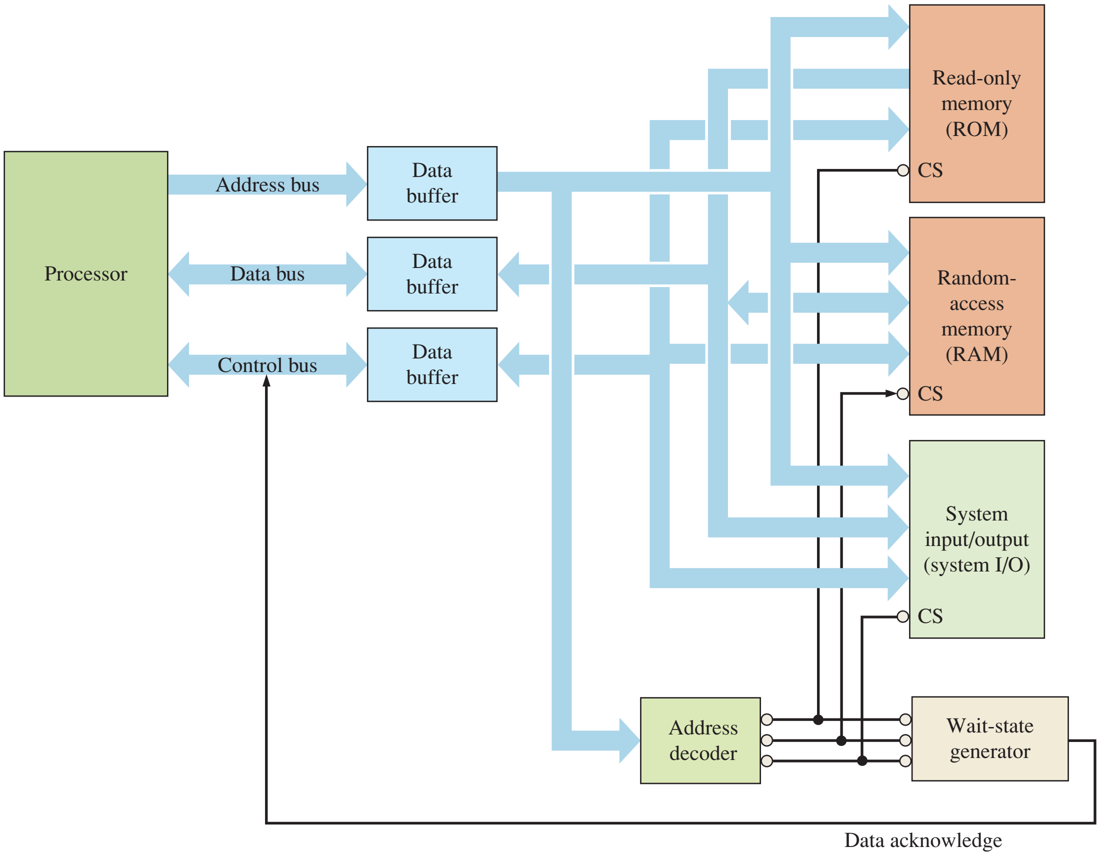


FIGURE 14–3 Block diagram of a practical computer system.

Shared Signal Lines

When the outputs of two or more devices connect to the same signal line, the potential for bus contention exists. Bus contention occurs when device outputs attempt to drive a signal line to different voltage levels. This causes high current to flow from one output into the other, which can damage the devices. Typically, bus contention occurs when device outputs are at different logic levels. However, even when devices are at the same logic level, the variation for different devices will cause some device output voltages to be higher than others so that bus contention will occur. Two special types of output, the tri-state output and open collector output, allow devices to share signal lines, while avoiding bus contention.

The term *tri-state* is a registered trademark of National Semiconductor but is often used interchangeably with the generic terms *three-state* or *3-state*. As the name suggests, the tri-state output adds a third output state, called the high-impedance or high-Z state, to the usual logic LOW and HIGH states. The tri-state switch is effectively a switch that disconnects the output of the tri-state device from the signal line so that it does not interfere with other devices from driving the line. When a tri-state device is enabled, it outputs a logic LOW or HIGH as other digital devices. When a tri-state device is disabled, the output assumes the high-Z state and the output is said to be tri-stated. When tri-state outputs share a signal line, only one output at a time must be enabled to ensure that bus contention will not occur. Figure 14–4 shows the operation of tri-state outputs.

Devices that are designed to connect to processor buses, such as memory and interface devices, typically have tri-state outputs built into them. Devices that do not have tri-state outputs or open-collector outputs must use tri-state buffers to connect to buses.

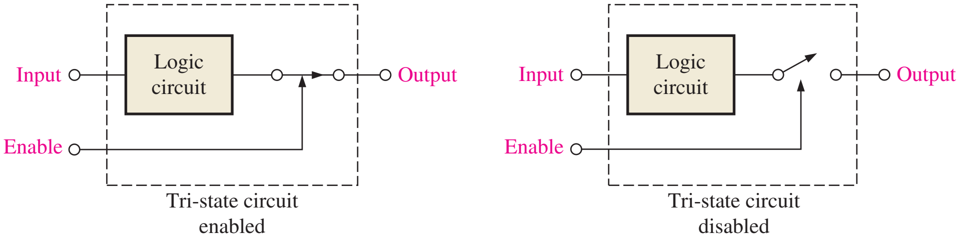
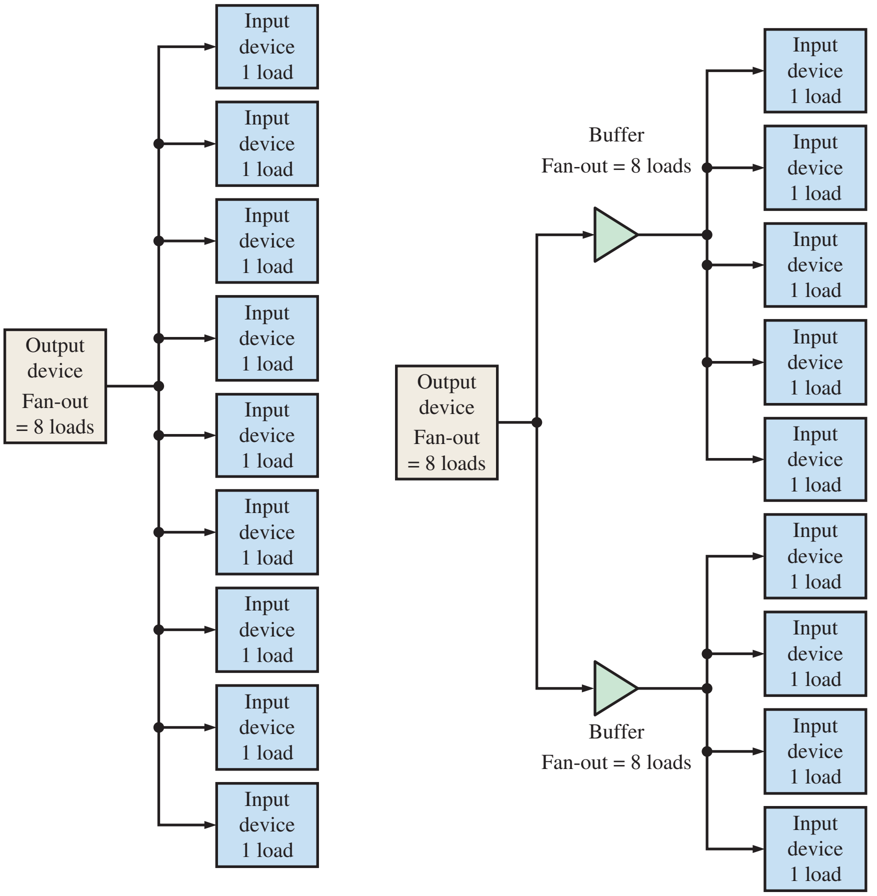


FIGURE 14–4 Logic devices with tri-state outputs.

Signal Loading and Buffering

Digital outputs are affected by the inputs of the devices to which they connect. There is a limit to the number of digital inputs that the outputs can reliably drive; this limit is called the device *fan-out*. When the number of inputs exceeds the fan-out of an output device, the operation of that output device may not meet the specified voltages or timing for that device. The issue of inputs affecting the performance of an output to which they are connected is called **signal loading**. To avoid problems with signal loading, special digital devices called buffers are used to ensure that device fan-outs are not exceeded. A **buffer** is a special circuit that isolates the output of a device from the loading effects of other devices.

Figure 14–5 illustrates the use of buffers to prevent the nine input devices from exceeding the eight-load fan-out of the output device. Note that up to seven input devices could



(a) Nine input device loads exceed 8-load fan-out of output device

(b) Buffering of output device prevents signal loading

FIGURE 14–5 Buffers are used to prevent overloading of driving device.

have been connected directly to the output device and a single buffer used to connect the remaining input devices. This would have reduced the parts count, but one characteristic of buffers is that each buffer will increase the propagation delay. If a single buffer were used, the response of the input devices connected to the buffer would be slower than that of the input devices connected directly to the output device. Using two buffers as shown helps match the propagation delay to all the input devices.

The buffers shown in Figure 14–5 are simple noninverting buffers, which means that the buffer output signal is identical to the buffer input signal. There are other types of buffers to ensure that devices will not degrade the performance of a device to which they are connected. These buffers include tri-state buffers like those mentioned previously, inverting buffers that invert the input signal, bidirectional buffers that can pass information through the buffer in both directions as on the data bus, and Schmitt triggers. A Schmitt trigger is a special device that helps prevent logic devices from acting erratically due to system noise affecting slowly changing inputs.

Device Selection

The processor uses the address bus to access ROM, RAM, hardware I/O ports, and other system devices. A question that naturally arises is how a device knows when the processor is attempting to access it rather than some other system device. The answer is that these devices have a special input, usually called a chip select (CS) or chip enable (CE), that enables the device. When the processor must access a specific device, it must assert the select line of the intended device.

While in theory processors could provide separate control lines to select system devices, this is not practical for general-purpose computers because there is no way for the system designers to know what devices a system will contain. Instead, system designers use PLDs or dedicated hardware decoders, similar to that in Figure 14–6, to decode processor addresses and generate the device select lines. For this example, the processor uses a 16-bit address bus where the upper (most significant) four bits are used to generate device select outputs.

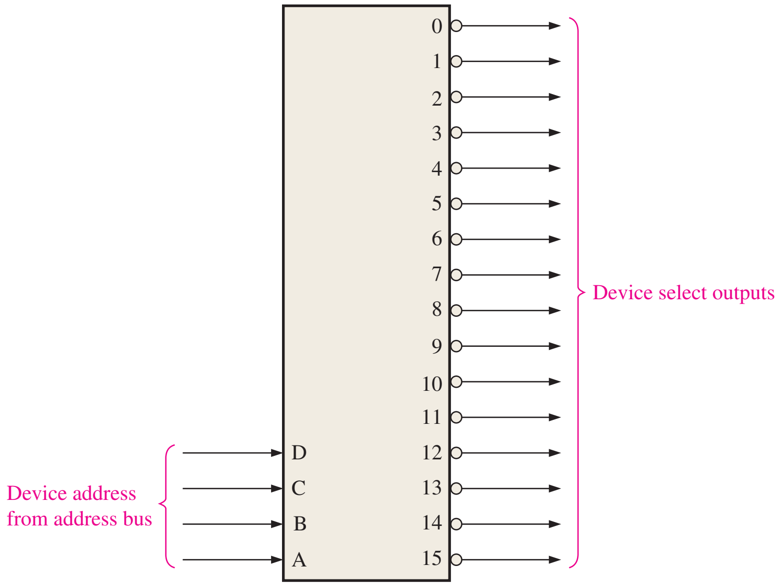


FIGURE 14–6 Address decoding for the purpose of device selection.

System Timing

A final issue with practical computer systems is system timing. In a computer system, the processor signals must meet the setup and hold times for each peripheral so that data are properly stored and accessed. As you have seen, decoding logic or buffers in the system can slow the processor signals. In some cases, the processor runs much faster than the peripherals that are available; in other cases, fast peripherals are available but their cost prohibits designers from using them. In addition, some peripherals, such as SRAM, are inherently faster than others, such as DRAM, so the signal timing that meets the setup and hold times for some devices will not meet the setup and hold times for others. To resolve this issue, three different types of system buses can be used: synchronous, asynchronous, and semisynchronous.

Synchronous buses include a synchronizing clock to ensure that signals from the processor meet the setup and hold times of the peripheral. Synchronous buses are faster than asynchronous or semisynchronous buses.

Asynchronous buses will automatically insert wait states in a bus cycle until a signal indicates that the bus cycle can finish. A **wait state** holds the state of the bus signals for one processor clock cycle so that the read or write operation is "frozen" for one clock period when the processor is accessing memory or other devices that are slow to respond. Several wait states may be necessary. Computer CPUs run at very high speeds, while memory technology does not seem to be able to catch up. Typical processors like the Intel Core 2 and the AMD Athlon 64 X2 run with a clock of several GHz, while the main memory clock generally is in the several hundred to over 1000 MHz range. Even some second-level CPU caches run slower than the processor core. In order to minimize the use of wait states, which slow the computer down, techniques such as CPU caches, instruction pipelines, instruction prefetch, and simultaneous multithreading are used.

Semisynchronous buses are similar to asynchronous buses except that a semisynchronous bus will complete the bus cycle unless a signal indicates that the processor should not complete bus cycle. Until the processor can complete the cycle, it will insert wait states.

Memory and other peripheral devices do not, as a rule, have signals indicating when data are ready. The signals that instruct the processor to insert wait states must be generated by an additional logic circuit, called a *wait-state generator*, which can be basically a programmable timer or shift register. The wait-state generator is clocked by the same clock as the processor and enabled by the device select line for a specific memory or other device. After the wait-state generator is enabled by the device select line, it will generate a ready signal after a specific number of clock cycles. Figure 14–7 shows an 8-bit parallel-in/serial-out shift register circuit that can insert up to six wait states for an asynchronous processor by delaying the ready signal to the processor by up to six clock cycles.

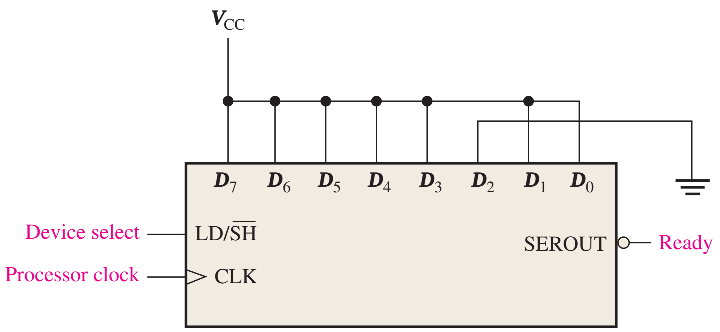


FIGURE 14–7 A wait-state generator programmed for one wait state.

The circuit of Figure 14–7, which inserts wait states for a single device, can be expanded to support more than one device. If two or more devices have the same number of wait states, their device select lines can be ANDed together (assuming the select lines are active-LOW).

EXAMPLE 14–1

For the wait-state generator in Figure 14–8, how many waits states will be generated when the device is selected?

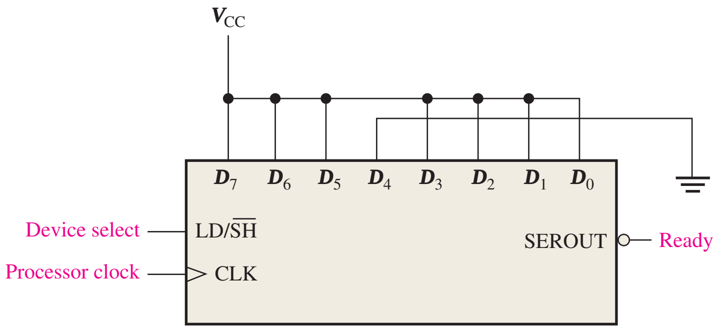


FIGURE 14–8

Solution

The initial pattern loaded into the shift register is 11101111_2. This shifted pattern for each clock and the corresponding number of wait states are

Clock 1 (0 wait states): 11110111_2

Clock 2 (1 wait state): 11111011_2

Clock 3 (2 wait states): 11111101_2

Clock 4 (3 wait states): 11111110_2

On the fourth clock after Device select goes LOW, the most significant bit of the SEROUT line for the shift register goes LOW. This causes the Ready output to go LOW, terminating the bus cycle. Therefore, the wait-state generator inserts **three** wait states.

Related Problem*

Which data input line of the shift register must be tied LOW for the wait-state generator in Figure 14–8 to insert five wait states?

*Answers are at the end of the chapter.

SECTION 14–2 CHECKUP

1. Define *bus contention* and discuss types of devices used to prevent it.
2. How does a processor enable various devices?
3. Define *wait state* and give its purpose.
4. What is the purpose of a buffer?

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 specific units, their design and organization, are called the *architecture* (do not confuse the term with the VHDL element). The architecture determines the instruction set and the process for executing those instructions.

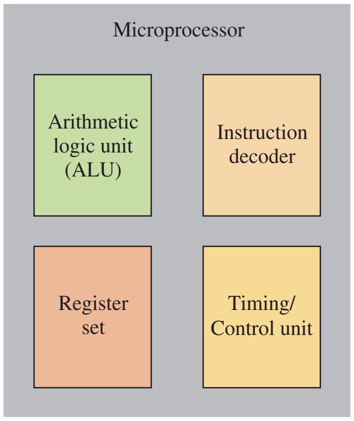


FIGURE 14–9 Elements of a microprocessor (CPU).

After completing this section, you should be able to

- Name the four basic elements of a microprocessor
- Describe the fetch/execute cycle
- Explain the read and write operations

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 14–10 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.

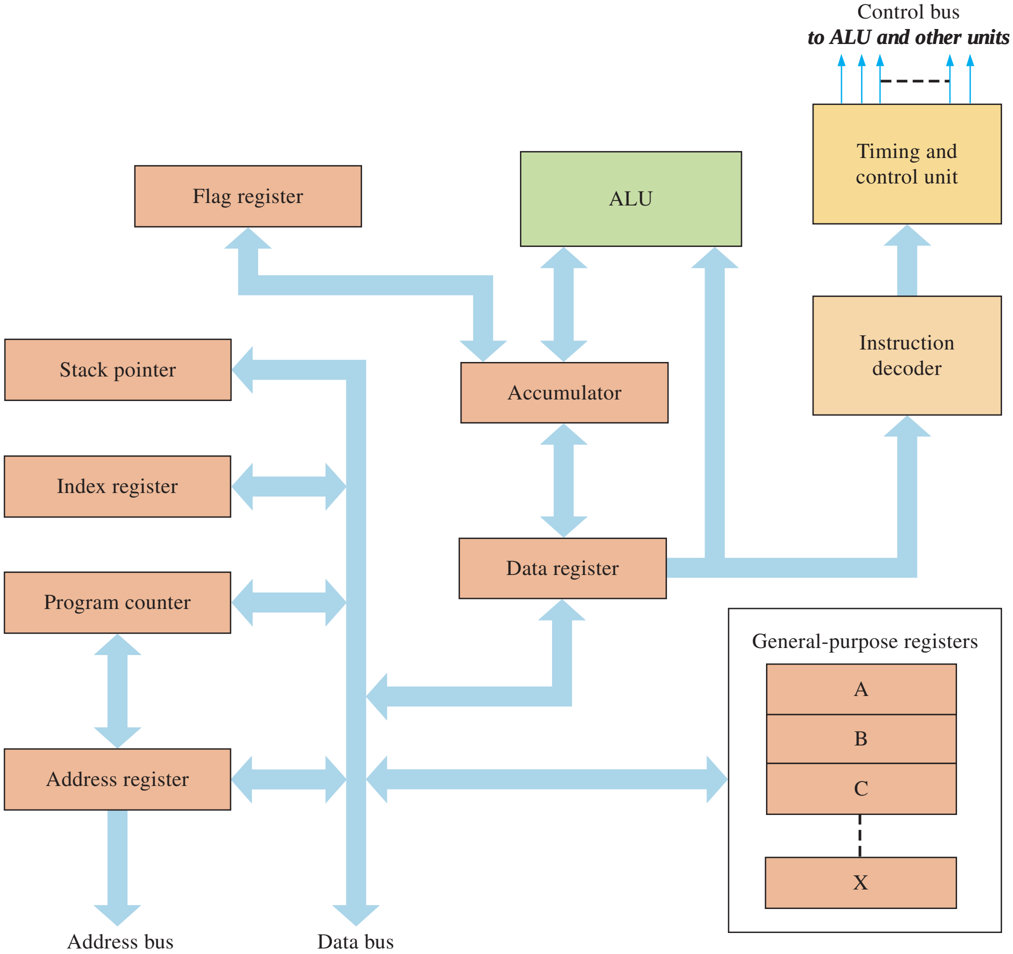


FIGURE 14–10 Basic model of a simplified processor.

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.

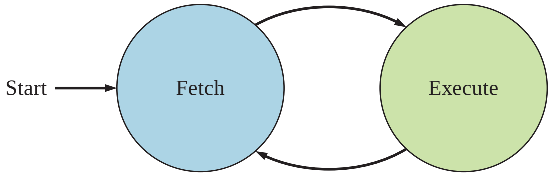
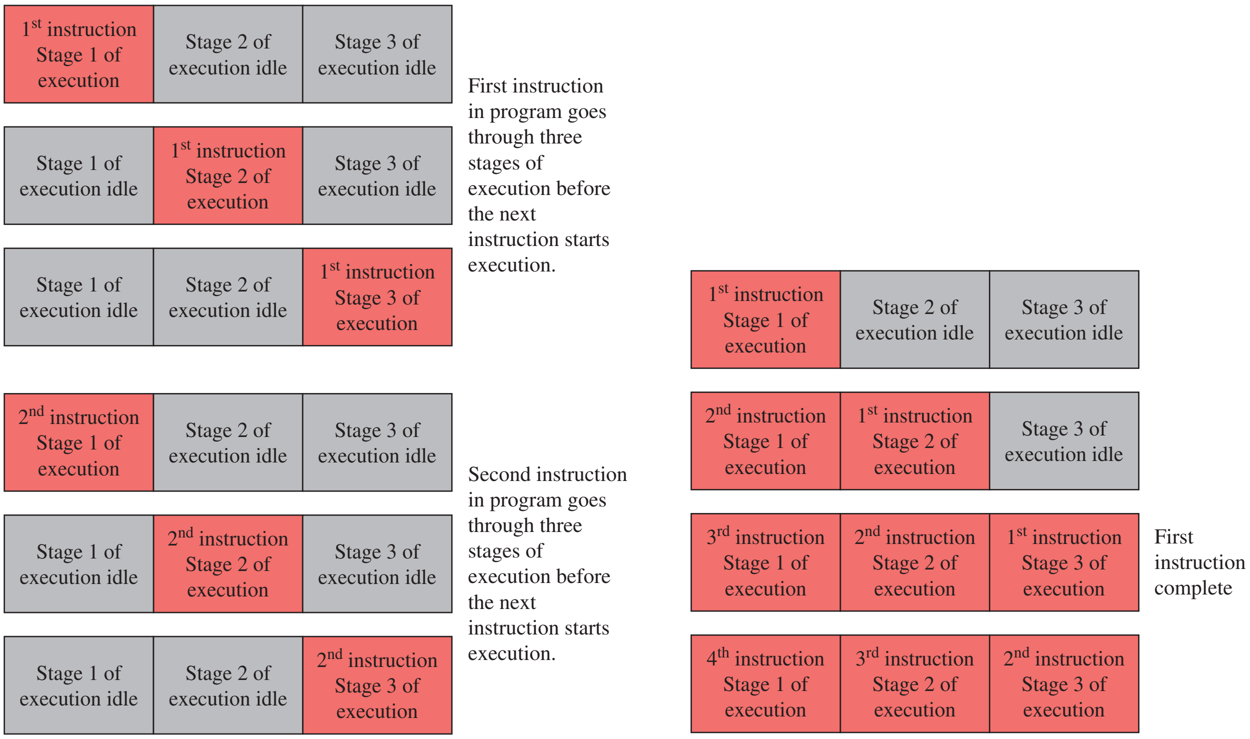


FIGURE 14–11 The fetch/execute cycle of a processor.

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 nonpipelined processing compared to pipelined processing using three stages of execution.



(a) Nonpipelined execution of a program showing three stages of execution

(b) Pipelined execution of a program showing three stages

FIGURE 14–12 Illustration of pipelining.

As shown in the figure, in nonpipelined 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.

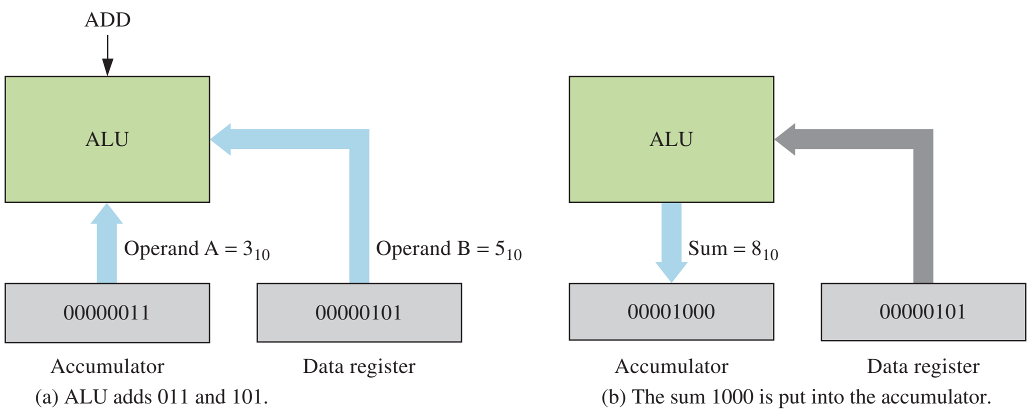


FIGURE 14–13 Example of the ALU adding two operands.

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 a 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.

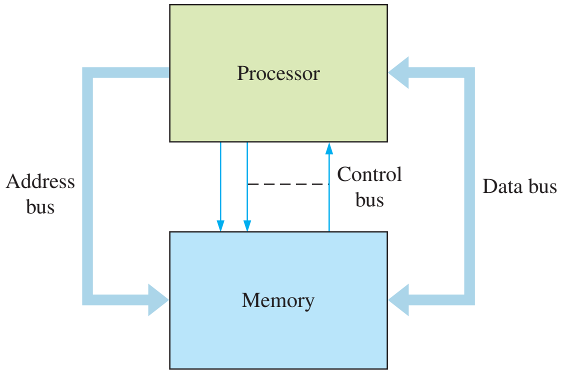
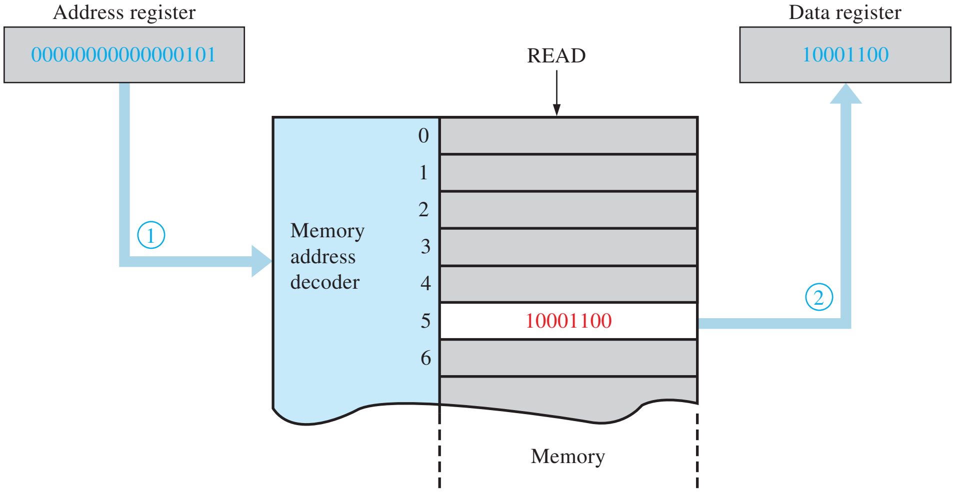


FIGURE 14–14 A processor and memory.

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



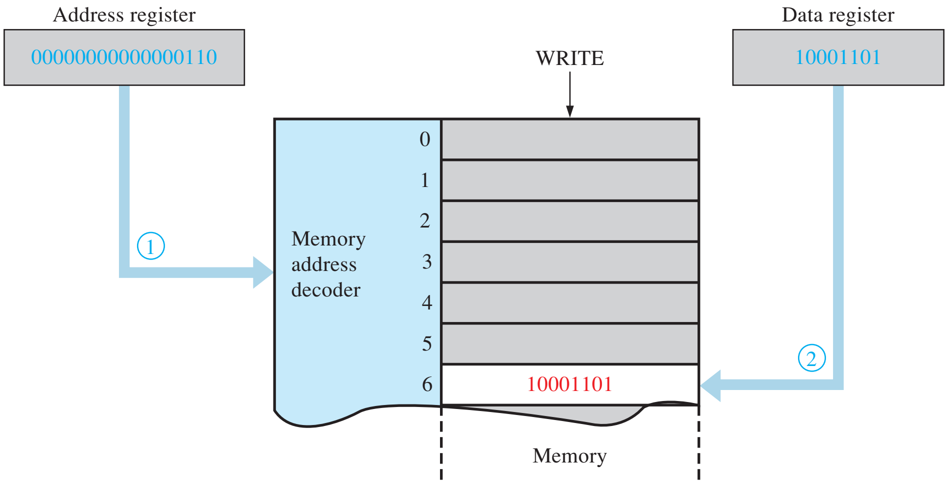
1 Address 5_{10} is placed on address bus and followed by the read signal.
2 Contents of address 5_{10} in memory is placed on data bus and stored in data register.

FIGURE 14–15 Illustration of the read operation.

data bus. 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.

SECTION 14–3 CHECKUP

1. Describe the fetch/execute cycle.
2. Name the four elements in a microprocessor.
3. What is the ALU and its purpose?
4. What happens during a read operation?
5. What happens during a write operation?

14–4 The Processor: Addressing Modes

A processor must address the memory to obtain data or store data. There are several ways in which the processor can generate an address when it is executing an instruction. These ways are called addressing modes and they provide for wide programming flexibility. Each instruction in a processor's instruction set generally has a certain addressing mode associated with it. The type and number of addressing modes vary from one processor to another. In this section, five common addressing modes are discussed, and generic instructions are used for illustration.

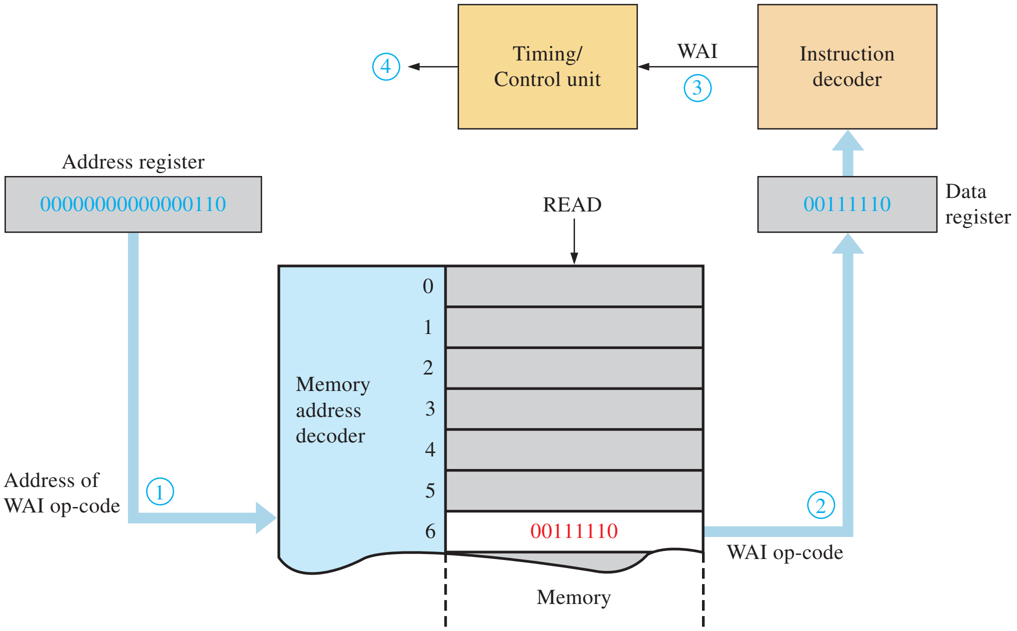
After completing this section, you should be able to

- Explain inherent addressing
- Explain immediate addressing
- Explain direct addressing

- Explain indexed addressing
- Explain relative addressing

Inherent Addressing

Inherent addressing is sometimes known as *implied addressing*. The one-byte instructions using this mode generally require no operand, or the operand is implied by the op-code, which is a mnemonic form of an instruction. An **operand** is the object to be manipulated by the instruction. For example, an instruction used to clear the accumulator (CLRA) has an implied operand of all zeros. The implied all-zeros operand ends up in the accumulator after the instruction is executed. Another example is a halt or wait instruction (WAI), which requires no operand because it simply tells the processor to stop all operations. The sequence that the processor goes through in handling an instruction with inherent addressing is illustrated in Figure 14–17. The op-codes used for illustration are similar to the op-codes of a typical processor.



(1) Address code (6_{10}) is placed on address bus.
(2) Data are placed on data bus and stored in data register by the read signal.
(3) Instruction is decoded.
(4) Timing/Control unit stops processor operation.

FIGURE 14–17 Fetch/execute cycle for the wait (WAI) instruction. This illustrates inherent addressing.

Immediate Addressing

Immediate addressing is used in conjunction with two-byte instructions where the first byte is the op-code and the second byte is the operand. The load accumulator (LDA) and the add to accumulator (ADDA) instructions are two examples that use immediate addressing.

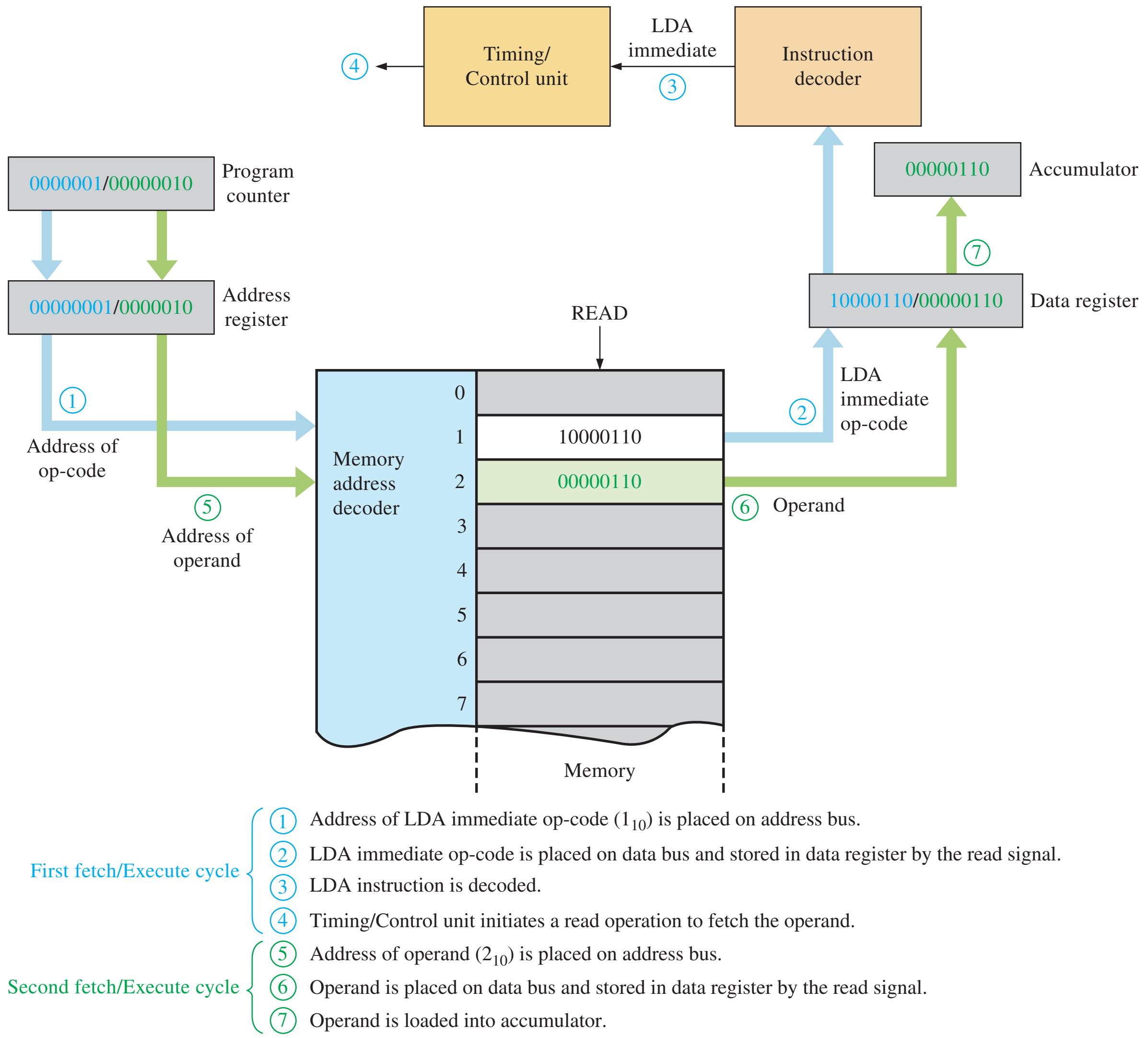


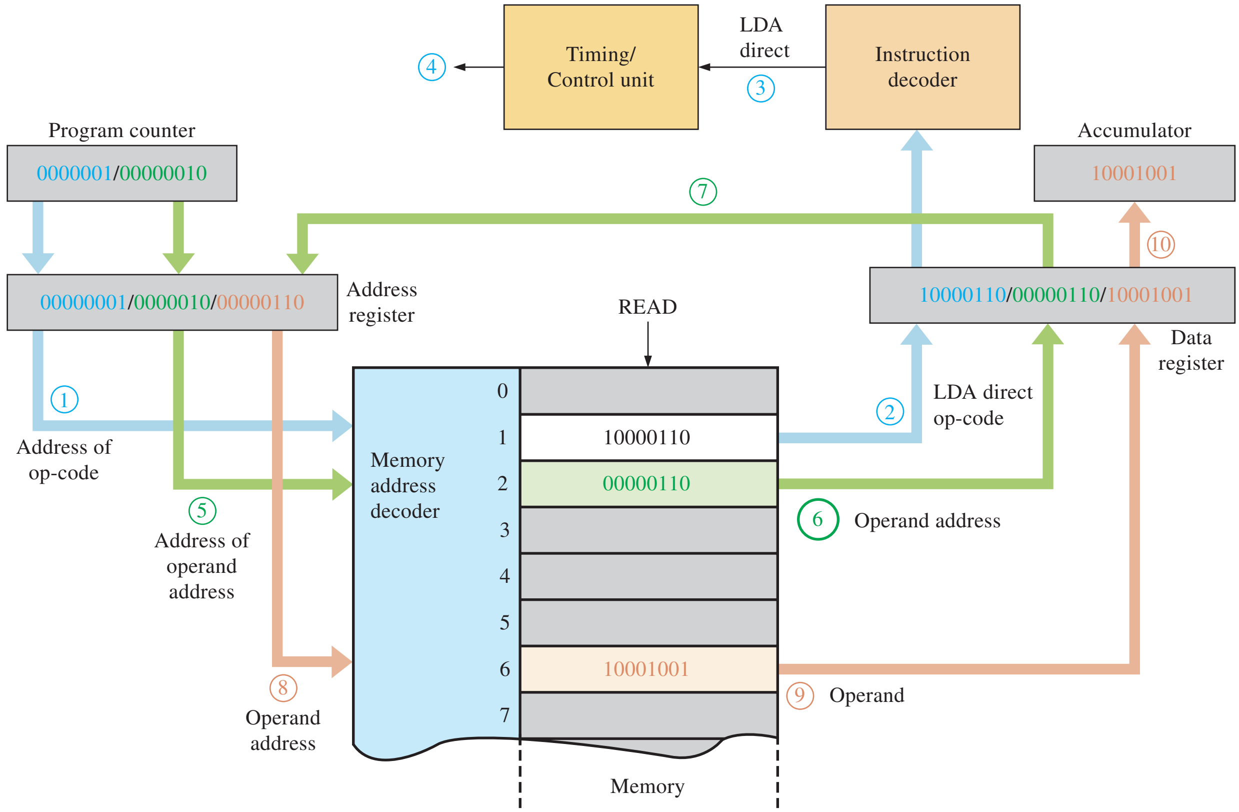
FIGURE 14–18 Illustration of immediate addressing. The process steps are numbered in sequence, and the cycle operations are color-coded.

The LDA immediate op-code is stored in one memory address, and the operand is stored in the address immediately following the op-code. That is, the op-code and operand are at consecutive memory addresses. When the LDA immediate instruction is fetched and executed, it tells the processor to get the contents of the next memory location (operand) and load it into the accumulator, as illustrated in Figure 14–18.

Direct Addressing

For an instruction using direct addressing, the first part is the op-code and the second part is the address of the operand, not the operand itself. For example, the LDA instruction uses direct addressing as well as immediate addressing. LDA direct has a different op-code than LDA immediate. Let's assume each part is one byte for simplicity.

The LDA direct instruction is used to illustrate direct addressing. Figure 14–19 shows the LDA direct instruction in memory addresses 1 and 2. The first byte is the op-code, and the second byte is the operand address. When the LDA direct instruction is fetched and executed, it tells the processor to load the accumulator with the operand located at the memory address specified by the second byte of the instruction. The process is illustrated in Figure 14–19.



First fetch/Execute cycle
1. Address of LDA direct op-code (1_{10}) is placed on address bus.
2. LDA direct op-code is placed on data bus and stored in data register by the read signal.
3. LDA instruction is decoded.
4. Timing/Control unit initiates a read operation to fetch the address of the operand.

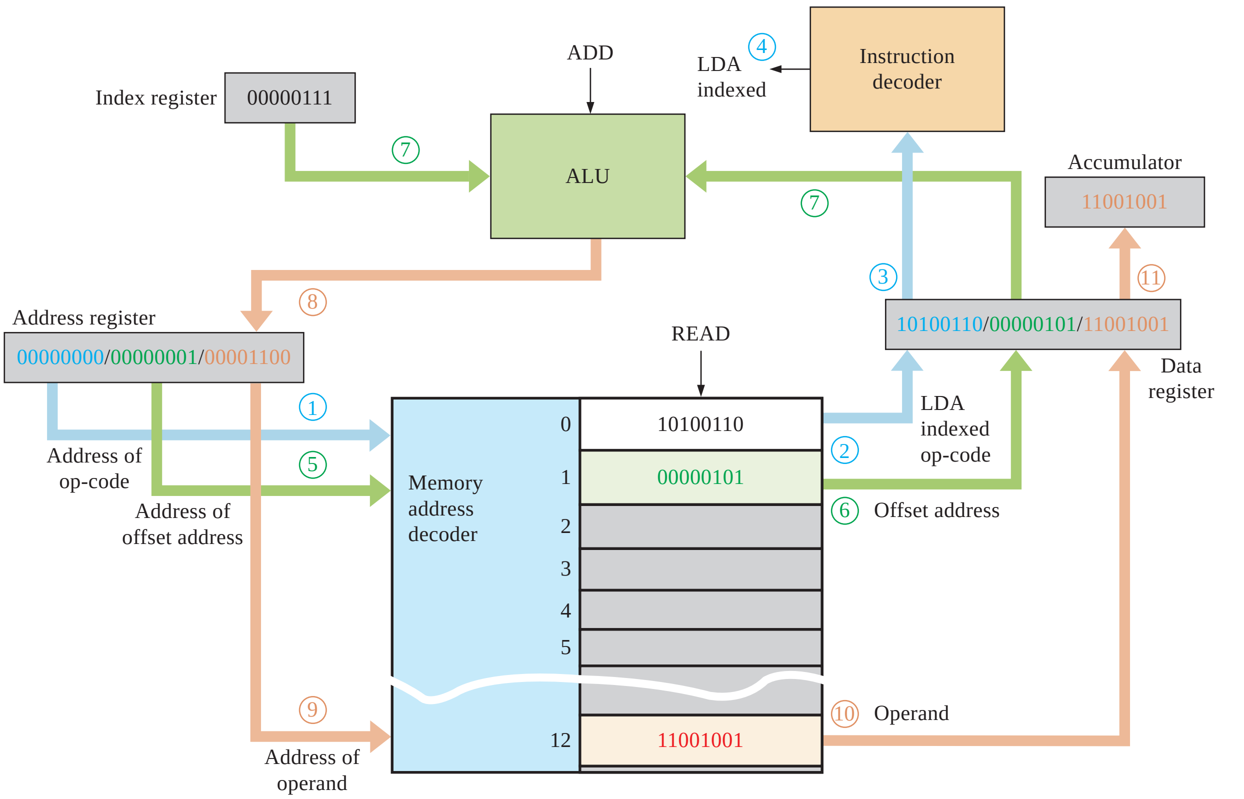
Second fetch/Execute cycle
5. Address of operand address (2_{10}) is placed on address bus.
6. Operand address is placed on data bus and stored in data register by the read signal.
7. Operand address (6_{10}) is loaded into address register.

Third fetch/Execute cycle
8. Operand address (6_{10}) is placed on address register.
9. Operand is placed on data bus and loaded into data register.
10. Operand is loaded into accumulator.

FIGURE 14–19 Illustration of direct addressing.

Indexed Addressing

Indexed addressing is used in conjunction with the index register. An instruction using indexed addressing consists of the op-code and the offset address. When an indexed instruction is executed, the offset address is added to the contents of the index register to produce



First fetch/Execute cycle
1 Address of LDA indexed op-code (0_{10}) is placed on address bus.
2 LDA indexed op-code is placed on data bus and stored in data register by the read signal.
3 LDA indexed instruction is decoded.
4 Timing/Control unit initiates a read operation to fetch the address of the operand.

Second fetch/Execute cycle
5 Offset address (5_{10}) is selected at address 1_{10}.
6 Offset address is placed on data bus and stored in data register by the read signal.
7 Offset address is added to contents of index register (7_{10}) by the ALU to produce address of operand (12_{10}).

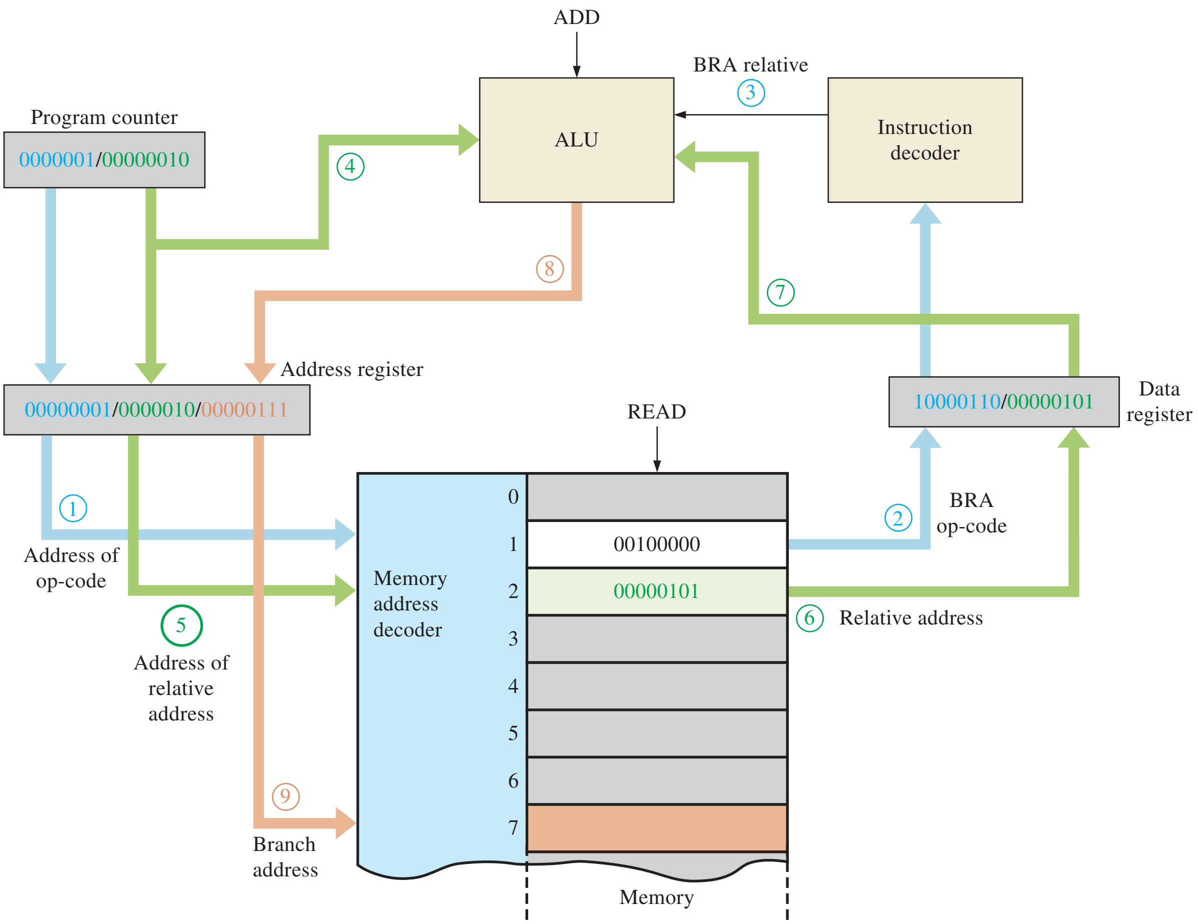
Third fetch/Execute cycle
8 Address of operand (12_{10}) is loaded into address register.
9 Address of operand is placed on address bus.
10 Operand is placed on data bus and stored in data register by the read signal.
11 Operand is transferred into accumulator.

FIGURE 14–20 Illustration of indexed addressing.

an operand address. In Figure 14–20 the LDA (load accumulator) instruction is again used to illustrate indexed addressing.

Relative Addressing

Relative addressing is used by a class of instructions known as *branch instructions*. Basically, a branch instruction allows the CPU to go back or skip ahead for a specified number of addresses in a program instead of going to the next address in sequence. Branching instructions are used to form program loops. For a relative addressing instruction (branch instruction), the first byte is the op-code and the second byte is the relative address. When a branch instruction is executed, the relative address is added to the contents of the program counter to form the address to which the program is



First fetch/Execute cycle
1. Address of BRA relative op-code (1_{10}) is placed on address bus.
2. BRA op-code is placed on data bus and stored in data register by the read signal.
3. BRA instruction is decoded.

Second fetch/Execute cycle
4. Program count is transferred to ALU and address register.
5. Address of relative address (2_{10}) is placed on address bus.
6. Relative address (5_{10}) is placed on data bus and stored in data register by the read signal.
7. Relative address is transferred to ALU.

8. Program count and relative address are added by ALU and resulting branch address (7_{10}) is placed in address register.
9. Program branches to specified address (7_{10}).

FIGURE 14–21 Illustration of relative addressing (branching).

to branch. Figure 14–21 illustrates relative addressing using a branch relative always (BRA) instruction that can branch both forward or backward. Forward branching is shown.

SECTION 14–4 CHECKUP

1. List five types of addressing.
2. What is an op-code?
3. What is an operand?
4. Explain branching.

14–5 The Processor: Special Operations

During normal operation the CPU fetches instructions from system memory, and these instructions are decoded by the instruction decoder. Each decoded instruction affects the operation of the timing and control unit, which in turn synchronizes the operation of the CPU, system buses, and system components to execute the instruction. In this section, specific CPU operations (polling, interrupts, exceptions, and bus requests) that occur when special circumstances or events arise that preempt normal processor operation are discussed.

After completing this section, you should be able to

- Define *polling*
- Define the terms *interrupt* and *exception*
- Describe the process by which a processor responds to and services an interrupt
- Explain how an interrupt service routine differs from a subroutine
- Explain why computer systems use bus requests

A computer runs programs that limit what the computer is permitted to do and how it will respond to situations that arise. Some situations are predictable and others are not. Even when a situation is predictable, just when it will occur may not be. As an example, every word processor program must respond to input from a keyboard, but the program cannot predict just when someone will press a key.

Polling

One technique to deal with unpredictable events is to have the CPU poll, or repeatedly check, the keyboard. The same occurs for other peripheral devices that may require attention from the CPU. Each time the CPU polls a device, it must stop the program that it is currently processing, go through the polling sequence, provide service if needed, and then return to the point where it left off in its current program. This process is inefficient and is suitable only for devices that can be serviced at regular and predictable intervals. Figure 14–22 illustrates **polling**, where the CPU sequentially selects each peripheral device via the multiplexer to see if it needs service.

Interrupts and Exceptions

A more efficient approach than polling is to have the processor perform its normal operations and deviate from them only when some special event requires the processor to take special action to handle it. Some sources use the term *exception* for any event that requires special handling by the processor. Other sources use *exception*, *software interrupt* (*SWI*), or *trap* for an event due to software and *interrupt* or *hardware interrupt* (*HWI*) for an event due to hardware. We will use **interrupt** to refer to a hardware event and **exception** for a software event that require the CPU to deviate from its normal operation.

When the processor receives an interrupt or an exception, it finishes executing the current instruction and then runs a special sequence of instructions called an *interrupt service routine* (*ISR*) or *exception handler*. An ISR similar to calling a standard program subroutine but with three important differences. Because the processor cannot know when an interrupt will occur, it automatically saves on the register stack status information about the program that is executing at the time the interrupt or exception occurs. The information includes the contents of the condition code register as well as the address of the next instruction to be executed when the ISR is finished. Sometimes the accumulator and condition code register, which make up the program status word, are both saved. The ISR must save on the stack any other

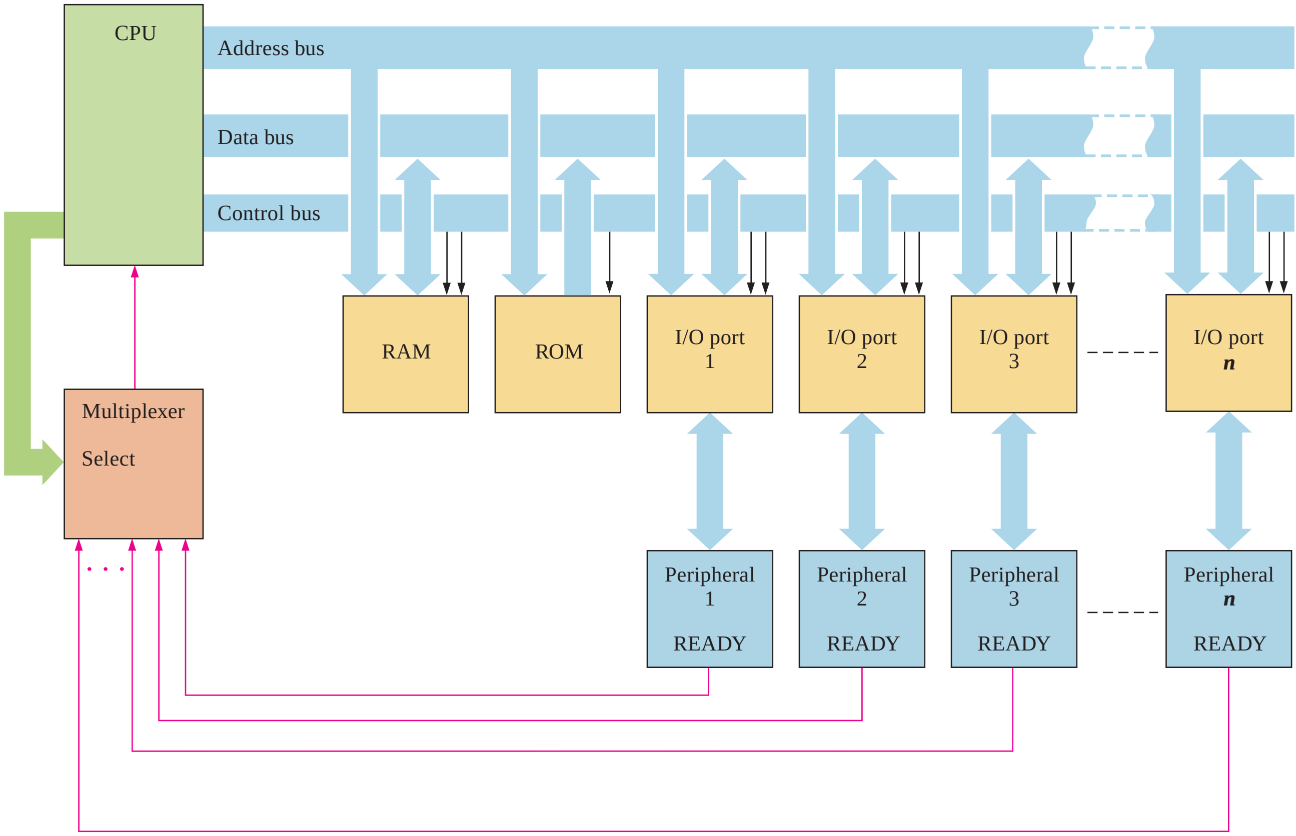


FIGURE 14–22 Basic concept of CPU polling peripheral devices.

registers it may use to ensure that the interrupted process will not be affected when it resumes executing.

Secondly, the processor obtains the address of the ISR based on the specific interrupt or exception that occurs. In some systems, a programmable interrupt controller (*PIC*) provides the address of the ISR over the data bus when the processor acknowledges an interrupt request. Other systems use autovectored interrupts that obtain the address for each interrupt from entries in an **interrupt vector table** stored in memory. Each vector, or ISR address, in the table specifies the starting address of an ISR. The programmer must write the ISRs and place the starting address for each in the correct location of the interrupt vector table. If no ISR exists for an entry in the vector interrupt table, or if the interrupt vector table is not properly initialized, interrupts and exceptions can cause the processor to behave erratically, "hang" (stop responding), or "crash" (abort and restart).

A third difference is that the ISR uses a special *return from interrupt (RTI)* instruction, which restores the additional status information as well as the address of the next instruction. RTI is used rather than a standard return from subroutine (RET) instruction, which restores only the address of the next instruction, to exit and return processor control to the interrupted process. Before executing the RTI instruction, the ISR must restore any registers it saved on the stack.

Specific interrupts and exceptions vary with each processor, but the following list describes some typical ones.

Reset This is sometimes called a cold boot. A cold boot completely restarts the system so that the processor runs the power on self-test (POST), initializes the hardware, loads the hardware drivers and operating system, and performs all other tasks necessary to prepare the system for operation.

Software reset This is a software exception and is sometimes called a warm boot. This also restarts the system but bypasses many of the hardware initialization tasks performed by a cold boot.

Divide by zero This is a software exception and occurs when the processor attempts to divide a number by zero.

System timer This is a hardware interrupt and occurs when a special timer asserts a signal indicating that a specified time interval or "time tick" (such as 1/60th of a second) has elapsed since the last occurrence.

Unrecognized instruction This is a hardware interrupt that occurs when the instruction decoder determines that the value it contains is not a valid instruction.

As the above list shows, ISRs must perform a variety of tasks. Just what the ISR does can be as complex or simple as the programmer desires. Figure 14–23 shows the basic concept of interrupts where a device called a programmable interrupt controller (**PIC**) is used to monitor peripheral devices for interrupt requests and send the appropriate address to the CPU so it can take the required action.

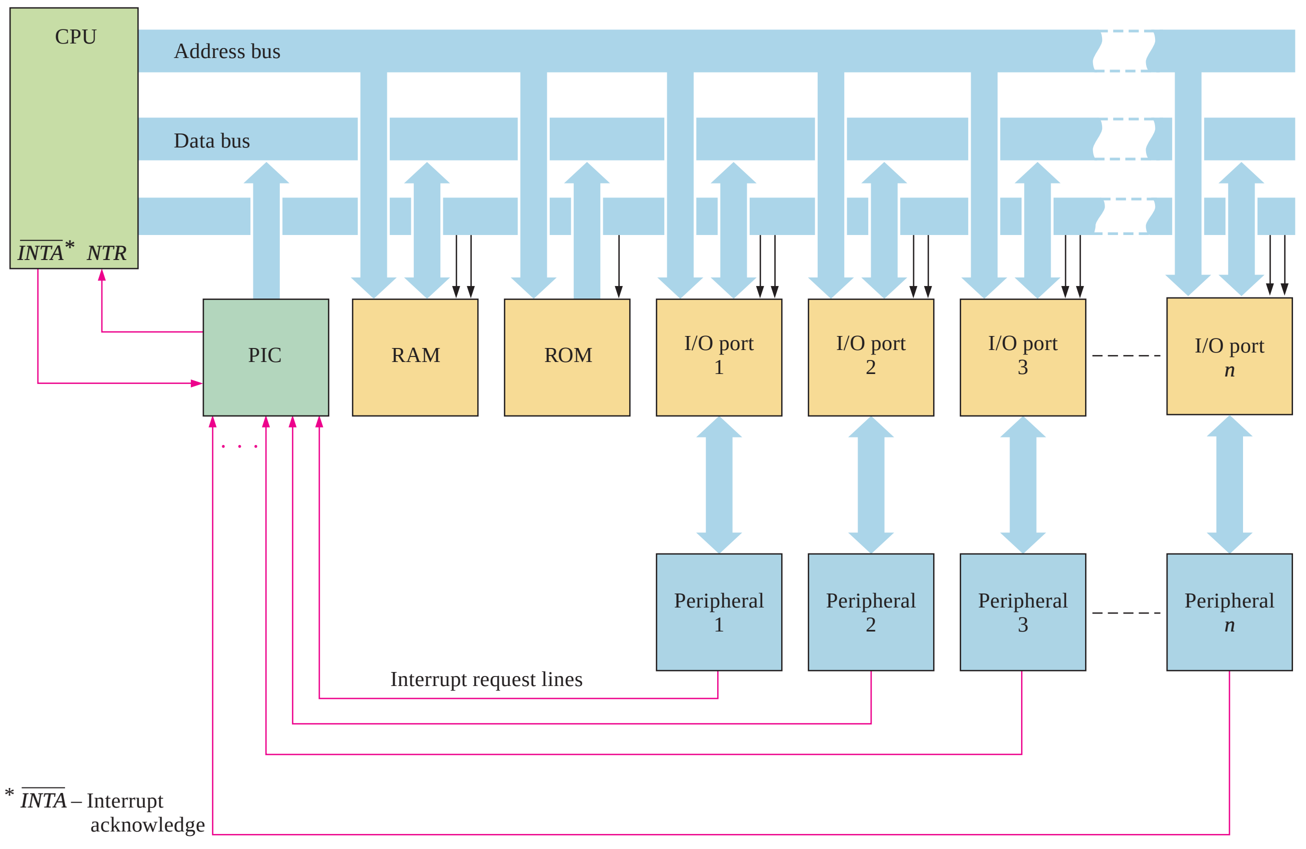


FIGURE 14–23 Basic concept of interrupt control.

Bus Request Operations

The device in a computer that drives the address bus and the bus control signals is called the **bus master**. In a simple computer architecture, only the CPU can be bus master, which means that all communications between I/O devices must involve the CPU. More complex architectures allow other devices (or multiple CPUs) to take turns at controlling the bus. For example, a network controller card can be used to access a disk controller directly while the CPU performs other tasks that do not require the bus. Any device can place data on the data bus when the CPU reads from that device, but only the bus master drives the address bus and control signals.

Although processors operate at high speeds, they are not always efficient at transferring data. When a processor transfers data from one device to another, it must use a bus cycle to read in the data from the source device and use another bus cycle to write the data back out to the destination device. The overhead in reading data into the processor and writing it out again greatly slows data transfers. The bus request operation allows other bus masters to take control of the system buses and rapidly transfer data between system devices.

Bus request operations are similar to interrupts and exceptions but differ in three important ways. Bus request operations do not complete the current instruction cycle before proceeding. Instructions can take hundreds or even thousands of clock cycles, and the circumstances that generated the bus request may be too urgent to be delayed. For example, a CD drive may be on the verge of a buffer under run and require data immediately to refill the buffer, or a memory controller may need to immediately refresh the system DRAM to prevent data from being lost. Interrupts and exceptions allow the processor to complete the current instruction cycle before processing the interrupt or exception.

Secondly, in a bus request operation, the processor passes control of the system buses to the requesting device, which then handles all bus operations. The processor continues to execute instructions in the ISR or exception handler during interrupts.

A third difference is that once the processor grants the bus request and relinquishes the system buses, the processor cannot regain control of the system until the requesting device relinquishes control or the processor is reset. The sequence of events during a bus request operation is as follows:

1. The bus master requesting control of the system buses submits a request by asserting the processor's bus request (BR) line.
2. The processor tri-states the system buses and signals that it has released control of the buses by asserting the bus grant (BG) line.
3. The requesting bus master uses the system address, data, and control lines to transfer data between system devices.
4. After completing the data transfers, the requesting bus master tri-states the system buses and signals the end of the bus request operation by asserting the bus grant acknowledge (BGACK) line.

Direct Memory Access (DMA)

One important class of bus master is the **DMA** (direct memory access) controller. These devices are designed specifically to transfer large amounts of data between system devices in a fraction of the time that the system processor would require. To utilize a DMA controller, the processor first writes the starting source address, starting destination address, and number of bytes to transfer to registers within the DMA controller. The processor next enables the transfer by writing to a control register within the controller, which then initiates the bus request operation. Computer systems typically use DMA controllers to transfer data between memory and hardware peripherals, such as when loading a program or data file from a hard drive to memory or when transferring a message from system memory to the transmit buffer of an Ethernet controller. DMA controllers can also move data between memory devices, for example, when moving data from main memory to cache memory.

DMA speeds up data transfers between RAM and certain peripheral devices. Basically, DMA bypasses the CPU for certain types of data transfers, thus eliminating the time consumed by normal fetch and execute cycles required for each CPU read or write operation. Transfers between the disk drive and RAM are particularly suited for DMA because of the large amount of data and the serial nature of the transfers. Generally, the DMA controller can handle data transfers several times faster than the CPU. Figure 14–24 shows a comparison of a data transfer handled by the CPU (part a) and one handled by the DMA (part b).

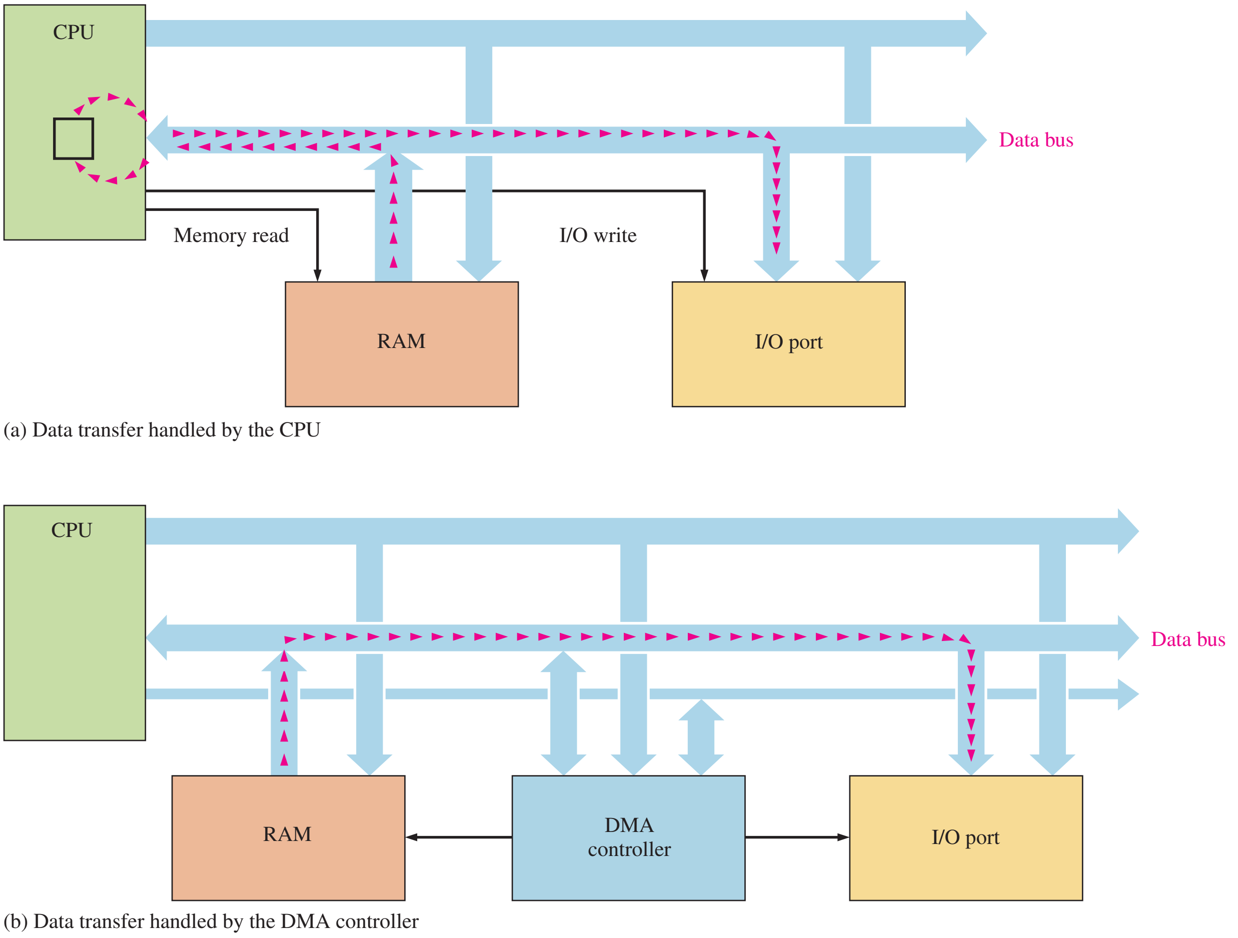


FIGURE 14–24 Illustration of DMA vs CPU data transfer.

Bus masters other than DMA controllers also use bus request operations. Processors in multiprocessor systems use bus request operations to access shared memory and other system resources. Memory controllers use bus request operations to perform background memory operations, such as refreshing DRAM and ensuring that the data in main memory and cache memory are consistent.

Figure 14–25 shows a computer system block diagram with a DMA controller and a PIC.

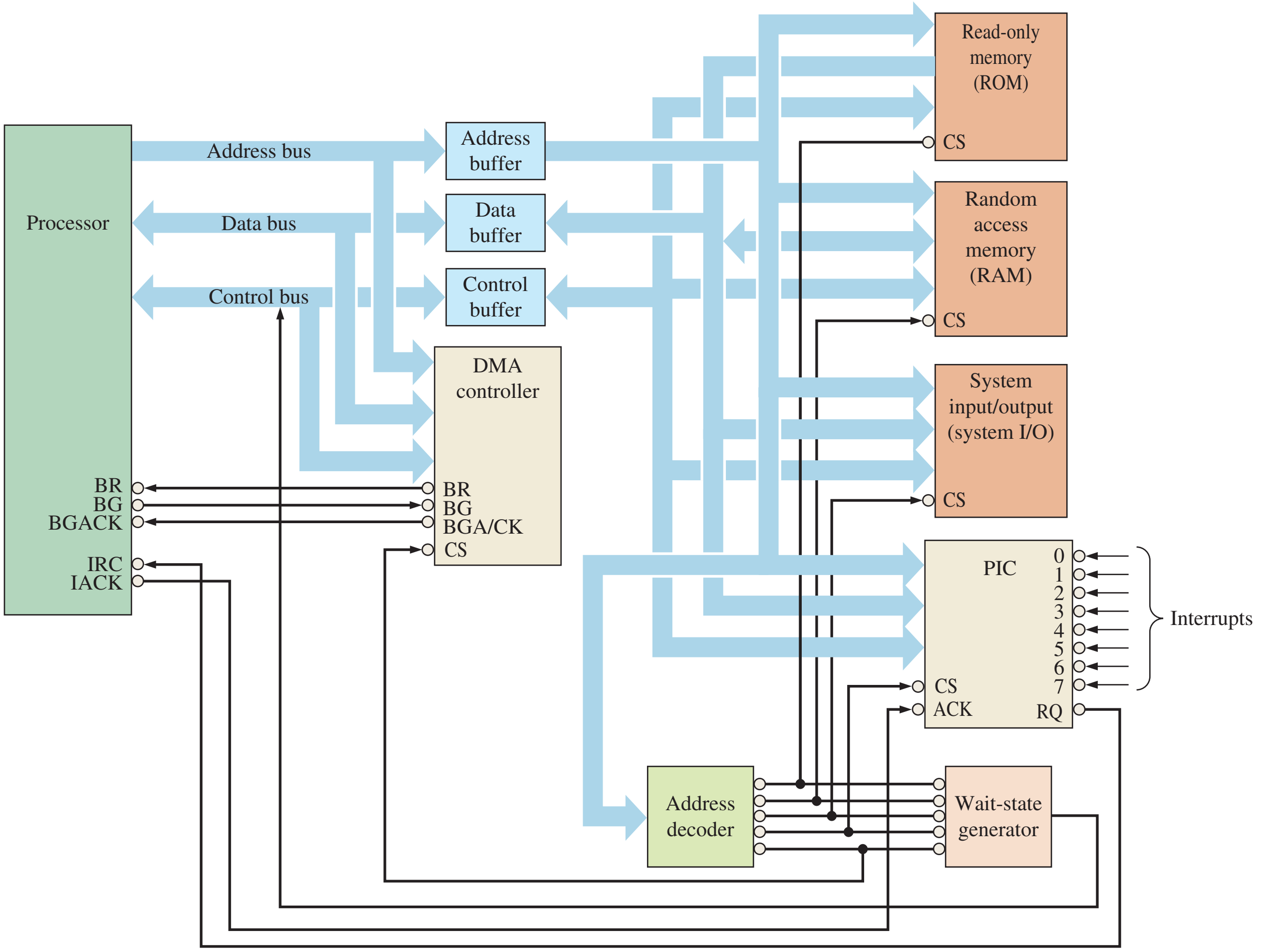


FIGURE 14–25 Block diagram of a typical computer.

SECTION 14–5 CHECKUP

1. Compare and contrast exceptions and interrupts.
2. Compare and contrast bus requests and interrupts.
3. Define and explain the purpose of direct memory accesses.

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.

After completing this section, you should be able to

- Explain the three basic duties of an operating system
- Discuss how an operating system functions in a computer system

- Compare and contrast the difference between multitasking and nonmultitasking operating systems
- Differentiate between multitasking and multiprocessing
- Identify and discuss the issues presented by multitasking

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 initalizes 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.

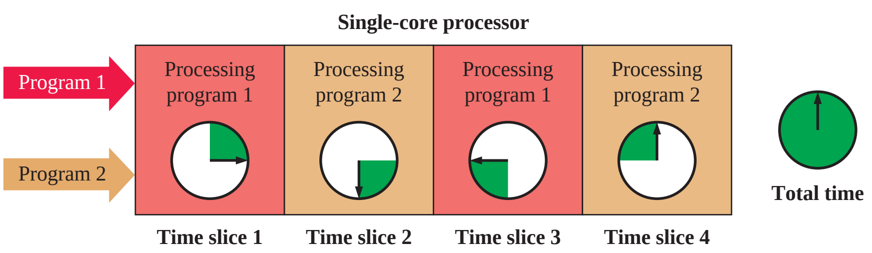


FIGURE 14–26 Simplified model of processor multitasking.

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.

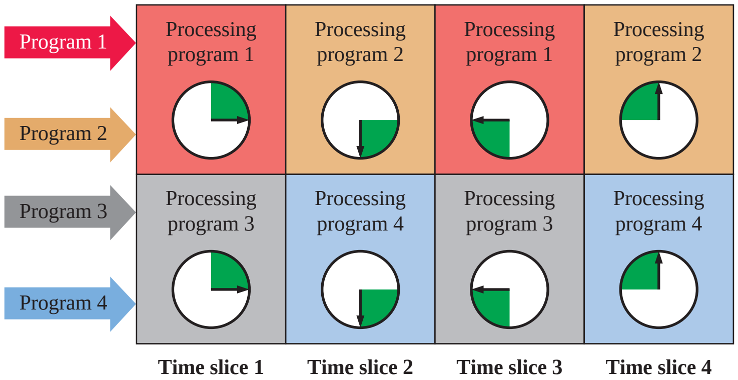


FIGURE 14–27 Multitasked multiprocessing in a multicore processor.

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 multiprocess 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.

SECTION 14–6 CHECKUP

1. What are the three basic duties of an operating system?
2. Compare and contrast multitasking and multiprocessing.
3. Describe how a memory management unit prevents one process from accessing the memory space of another process.
4. Explain how an operating system permits two processes to exchange information.

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*.

After completing this section, you should be able to

- Describe some programming concepts
- Discuss the levels of programming languages

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

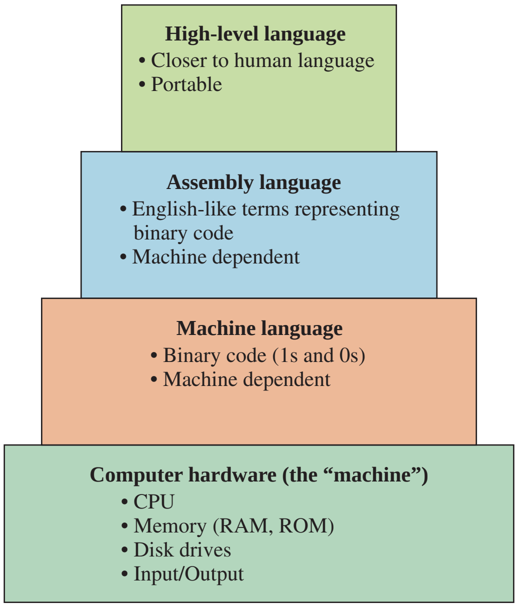


FIGURE 14–28 Hierarchy of programming languages relative to computer hardware.

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.

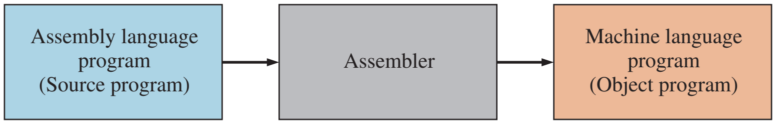


FIGURE 14–29 Assembly to machine conversion using an assembler.

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.

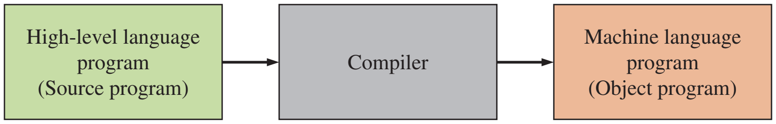


FIGURE 14–30 High-level to machine conversion with a compiler.

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.

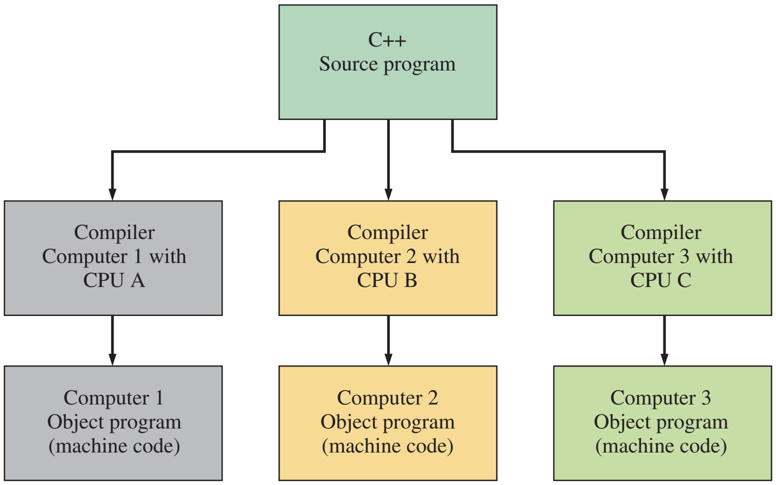


FIGURE 14–31 Machine independence of a program written in a high-level language.

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.

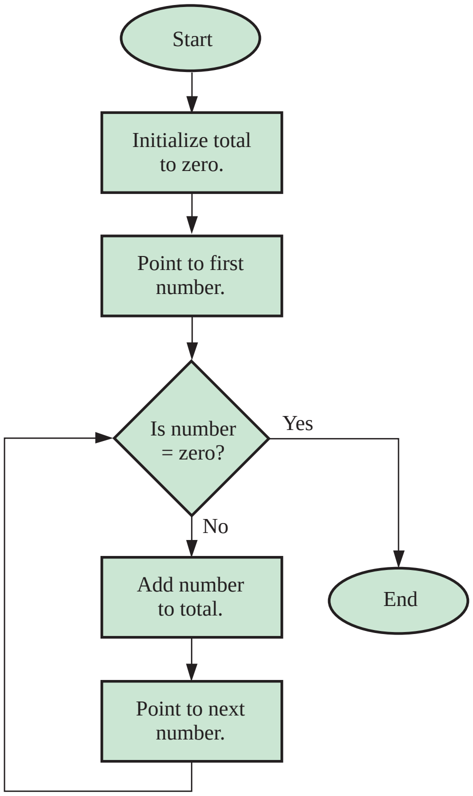


FIGURE 14–32 Flowchart for adding a list of numbers.

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 directives were **word ptr**, which is 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.

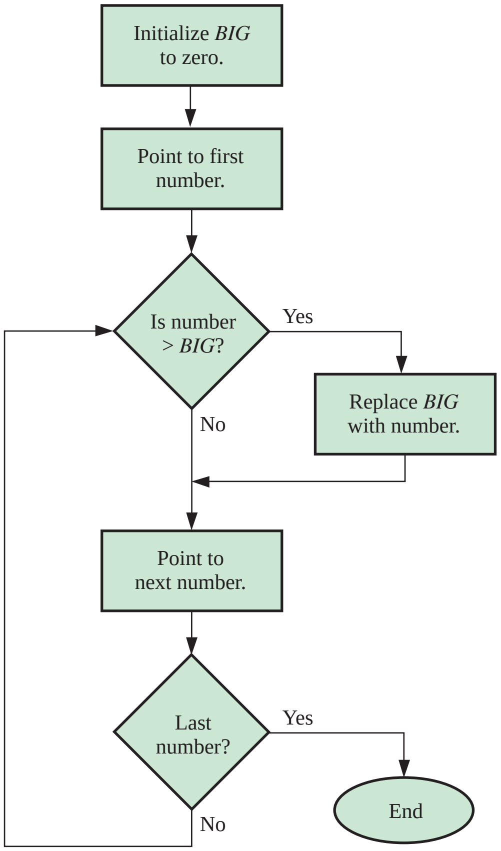


FIGURE 14–33 Flowchart. The variable *BIG* represents the largest value.

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
```

Related Problem

Explain how you could change the flowchart to find the smallest number in the list instead of the largest.

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 miniprogram 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

SECTION 14–7 CHECKUP

1. Define *program.*
2. What is an op-code?
3. What is a string?

14–8 Microcontrollers and Embedded Systems

Although a general-purpose microprocessor can interface with a variety of devices over its system buses, its ability to interface with the real world is limited. Most general-purpose microprocessors must use analog-to-digital converters (ADCs), digital-to-analog converters (DACs), universal asynchronous receivers and transmitters (UARTs) and other communication controllers, peripheral interface adapters (PIAs), external timers, and other specialized peripherals to process real-world information. Microcontrollers are used in microprocessor-controlled applications called **embedded systems** that perform a specific set of tasks and incorporate both the hardware and firmware required to perform them. Embedded systems include personal electronic devices such as cell phones, MP3 players, and calculators as well as consumer and industrial products as microwave ovens, automated assembly systems, and robots.

After completing this section, you should be able to

- Describe the general architecture of microcontrollers
- Discuss the types of peripherals found in common microcontrollers
- Describe how microcontroller peripherals are configured
- Describe how microcontrollers are used in various embedded systems

Microcontroller Basics

A special type of processor, called a **microcontroller**, sometimes abbreviated as μC or MCU, combines a microprocessor core, memory, and common peripherals in a single package. Microcontrollers can range in complexity from simple devices with a few dozen pins to very complex devices with hundreds of pins. A common aspect of all these processors is that the design of each seeks to incorporate all the elements of a microprocessor system into a single package. A microcontroller will typically include the following functional units:

- A microprocessor (called the processor core)
- Nonvolatile memory for program code, device configuration data, and similar data that must be preserved when power is removed
- RAM for program data, internal registers, peripheral device buffers, and other data storage
- Peripheral devices such as timers, ADCs, DACs, communication controllers, and I/O ports
- Internal buses to connect the processor core to internal memory
- Internal buses to connect the processor core and internal memory to peripheral devices
- Interface circuitry to connect the microcontroller with external devices

In addition to the above list of microcontroller features, more sophisticated microcontrollers can also include the following:

- A phase-locked loop (PLL) to multiply a low-frequency external clock to a higher internal frequency, increasing the speed of microcontroller operation
- DMA controllers to improve data transfer between internal memory and peripheral devices
- Programmable logic resources, or “fabric”, to implement custom functions
- A JTAG interface to support device testing and programming
- Special power modes for low-power and standby operation

Figure 14–34 shows a simplified block diagram of a typical microcontroller.

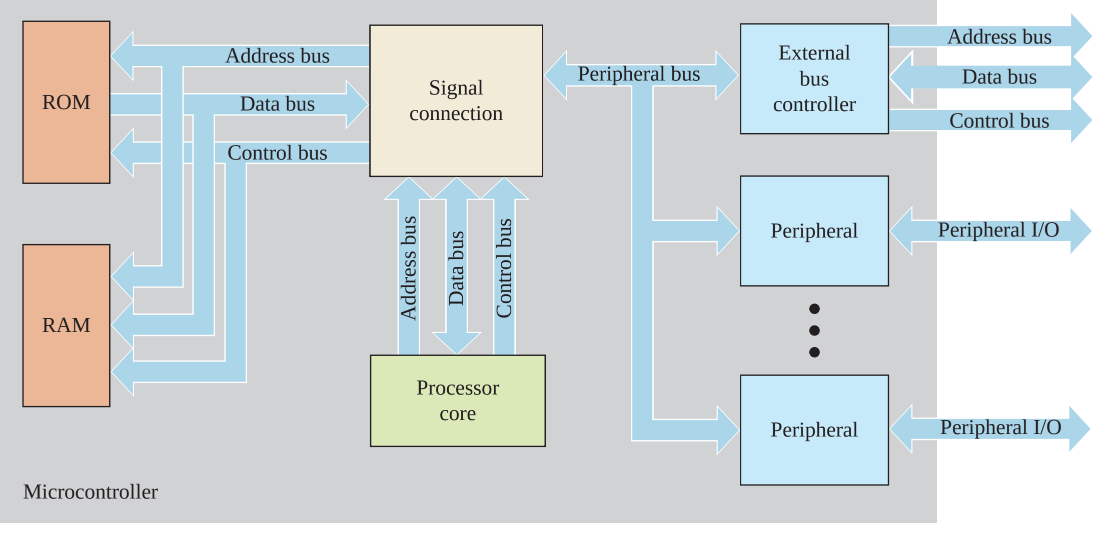


FIGURE 14–34 Simplified microcontroller block diagram.

Microcontroller Peripherals

Microcontrollers feature a wide variety of peripherals. Manufacturers select the type and number of peripherals, as well as the types and amounts of internal memory, to meet the requirements of specific applications, such as communication, automotive, and motion-control products. For example, microcontrollers that target communication applications will include a wide variety and number of communication controllers (such as Ethernet, I^2C, USB, and UART) to support the transmission and reception of data using multiple protocols. In contrast, microcontrollers meant for motion-control applications, such as robotic assemblies, will include ADCs, DACs, encoders, and pulse width modulators (PWMs) for position sensing and motor feedback and control.

Many pins on microcontrollers are multifunctional. This not only helps to reduce the total pin count and cost of the device but also limits the functions that can be used at the same time. The data sheet may state that a communications microcontroller has four USB controllers, two UART controllers, an Ethernet controller, an external memory interface, and 80 general-purpose I/O (GPIO) lines, but it is unlikely that a design can use all of these. A pin on a communications microcontroller, for example, might serve as a transmit line for USB communications, a clear-to-signal line for UART communications, a transmit line for Ethernet communications, an address line for the external bus controller, or a general-purpose I/O (GPIO) pin; but it can be configured for only one function at a time. Since applications rarely can change pin functions "on the fly," the circuit design permanently assigns the function of each microcontroller pin. If the circuit designer must use a set of pins for an external memory interface but also needs other functions that those pins provide, she either must find those functions on other pins (which is why microcontrollers offer more than one instance of a type of peripheral) or use external circuits to provide those functions.

The following describes some of the more common types of peripherals on microcontrollers.

General-Purpose I/O (GPIO) General-purpose I/O pins are typically the default function for many microcontroller pins. As the name suggests, these pins can be configured as input or outputs to read or write data, either as individual bits (for serial transfer of data) or groups of bits (for paraller transfer of data). Typical applications for GPIO lines are to read individual switches, to drive LED indicators, or to select or enable latches or buffers.

Communication Controllers Communication controllers allow microcontrollers to communicate with other devices using specific communication protocols. Some standard communication protocols are universal asynchronous receive and transmit (UART), Ethernet, universal serial bus (USB), inter-integrated circuit (IIC or I2C), serial peripheral interface (SPI), controller area network (CAN), and high-level data link control (HDLC). Because the timing, flow control, and data format of these protocols vary so widely, configuring communication controller functions is typically much more involved than for other peripheral functions.

Timers Microcontroller timers can have multiple uses. These include setting the frequency for a communication controller, indicating when a preset time interval has elapsed, determining the elapsed time between two events, and providing a periodic time tick for a system real-time clock.

ADCs and DACs ADCs and DACs are the means by which digital circuits interact with an analog world. As you know, digital circuits must use a limited set of values to represent a continuous range of analog data. Microcontrollers use ADCs to convert analog voltage and current measurements from sensors into digital values for processing and use DACs to convert digital values into analog voltages and currents to control electric and electronic circuits.

Quadrature Encoders Quadrature encoders are used to determine the speed, direction, and position of a moving object, such as a computer mouse or a stepper motor. A quadrature encoder represents the present position of a tracked object with a Gray code sequence. When the object moves, the Gray code value changes. Each change will increment or decrement a counter to represent a positive or negative change in position. For example, a system could use the sequence $00 \rightarrow 01 \rightarrow 11 \rightarrow 10 \rightarrow 00$ to represent a positive change so that the sequence $00 \rightarrow 10 \rightarrow 11 \rightarrow 01 \rightarrow 00$ would represent a negative change. The counter value represents the position of the tracked object in the physical system relative to the starting point or origin of the tracked object; how fast the counter value changes reflects the speed of the tracked object. Quadrature encoders typically use 32-bit or larger counters to prevent an underflow or overflow condition that would make it seem that the tracked device suddenly changed from a maximum or minimum position or vice versa. Tracked objects require one encoder for each dimension of movement. An object that moves in one dimension, such as a sliding door, needs only one encoder to track movement along the line of travel. An object that moves in two dimensions, such as a computer mouse, requires two encoders to track movement in the plane of travel. Objects that move in three dimensions, such as some robotic assemblies, require three encoders to track movement in the space of travel.

Pulse Width Modulators As you know, a pulse width modulator modulates, or varies, the pulse width of a repetitive digital signal to change the signal's duty cycle (i.e., the ratio of the time that a signal is HIGH to the period of the signal). Pulse width modulators are often used in motor control. Although motor controller circuits can use the amplitude of winding current to set the speed of some motors, a more typical approach is to keep the amplitude of the applied winding current constant and vary the duty cycle to control the speed. Microcontrollers can precisely control the duty cycle to accurately set the motor's running speed. Also, microcontrollers can change the duty cycle very quickly in response to the effects of motor speed due to line or load variations to maintain a constant running speed.

External Memory Controllers Although most microcontrollers contain internal ROM, RAM, EEPROM, flash, and other memory for code, data, and other program information, some applications require more memory than a microcontroller contains. Consequently, many microcontrollers feature external memory controllers that permit interfacing the microcontroller to external memory devices. Some microcontrollers do not contain any internal memory, so external memory must be used. External memory controllers often feature decoded chip select lines that allow programmers to configure the size of the memory range, the port size (8-, 16-, or 32-bits), and the number of wait states for each select line; they can contain memory management units that provide memory protection for multitasking applications. External memory devices are typically limited to SRAM, SDRAM, flash, and other memory types that do not require special bus operations, as do DRAM and EEPROM.

Configuring Peripherals

Microcontroller peripherals must be configured so that they operate the way an application requires them to do. Configuring means loading specific registers associated with the peripheral with values that control the function and operation of the peripheral. The register and values vary with each peripheral, but the registers fall into the general categories described next. Depending upon the number of bits required to configure some aspect of a peripheral, some categories may share one register, while others may require multiple registers to contain the necessary information.

Control Registers Control registers determine how the peripheral will function. For some peripherals a control register may select the specific peripheral as well as the characteristics for that peripheral. For example, the control registers for a communication controller could specify the specific communication protocol and the data rate, data packet size, error detection method, and operating mode (interrupt-driven or polled).

Status Registers Status registers contain information about how the peripheral is operating and conditions associated with peripheral operation. Applications use status registers to detect errors, determine when the peripheral has completed some task, and decide when conditions require some special handling. The microcontroller may automatically clear some status bits when firmware corrects a detected condition, while in other cases firmware may need to manually clear some status bits. For example, if an ADC sets the end-of-conversion status bit to indicate it has completed converting an analog value, reading the converted value from the ADC data register may automatically clear the bit; firmware may need to specifically clear the status bit to allow the ADC to perform another conversion.

Data Registers Data registers contain information that the peripheral processes in some way. The value in a data register can be data for the peripheral to process, data processed by the peripheral, or data currently being processed. The contents of data registers might not change unless firmware changes them, or operation of the peripheral may automatically update them. For example, the initialization register for a timer contains the initial count value that is loaded into the timer and may not change unless firmware writes a new value into the register. In contrast, the timer's count register holds the actual value of the timer and may update each time the counter is clocked. Some peripherals have only a few configuration registers.

The GPIO pins typically have only two registers: a control register that determines whether a pin is an input or output and a data register that contains the signal level of the pin. Other peripherals can have many more registers. A communication controller, for example, can have a control register to specify the communication parameters, a status register to monitor the operation of the controller, a transmit buffer descriptor register to specify the memory locations of data to be transmitted, a transmit length register to specify the number of bytes to transmit, a receive buffer descriptor register to specify the memory locations at which received data are to be stored, a receive length register to indicate the number of bytes received, an interrupt status register to signal communication events during reception and transmission, and an interrupt mask register to prevent or allow recognition of communication events. When configuring microcontroller peripherals, the programmer must carefully read the user manual and understand not only the operation of each peripheral he intends to use but also which configuration registers must be programmed and the configuration values to use.

As the number of products using microcontrollers has grown, manufacturers and third-party vendors have visual development and evaluation tools to simplify the process of programming microcontrollers. Many tools now allow programmers to use drop-down lists, check boxes, and other visual controls to generate C or C++ initialization code by specifying the peripherals they wish to use and how the peripherals should operate. While this is convenient and shortens development time, errors are still possible. Programmers should always review the code to verify it matches what they expected.

Microcontrollers in Embedded Systems

Personal Handheld Systems

Smart phones, digital media devices, calculators, and portable GPS units are only a few examples of portable handheld electronic devices that are microcontroller-based embedded systems. Microcontrollers are widely used in these products because they can easily interface with the input and output hardware, rapidly process data, and consume relatively little power. Some of the most popular microcontrollers for portable handheld devices are those based on the ARM (Advanced RISC Machine) processor.

A block diagram for a microcontroller-based programmable calculator is shown in Figure 14–35. The calculator incorporates a USB communication port. The ROM contains the embedded code that implements the calculator functions and processes while the RAM provides storage for the system stack, system data, user data, and programs. The USB controller transmits and receives data per the USB communications protocol and interfaces to the hardware that makes up the physical USB port. The calculator keypad connects to a parallel port formed by multiple GPIO lines, and the calculator LCD display interfaces with an LCD driver peripheral in the microcontroller to create the human machine interface, or HMI. A timer inside the microcontroller powers down the calculator after it has been active for a preset amount of time to save energy. Other timers in the microcontroller, which are not shown, set the communications rate for the USB controller, provide a real-time clock, and allow the user to set time-of-day alarms.

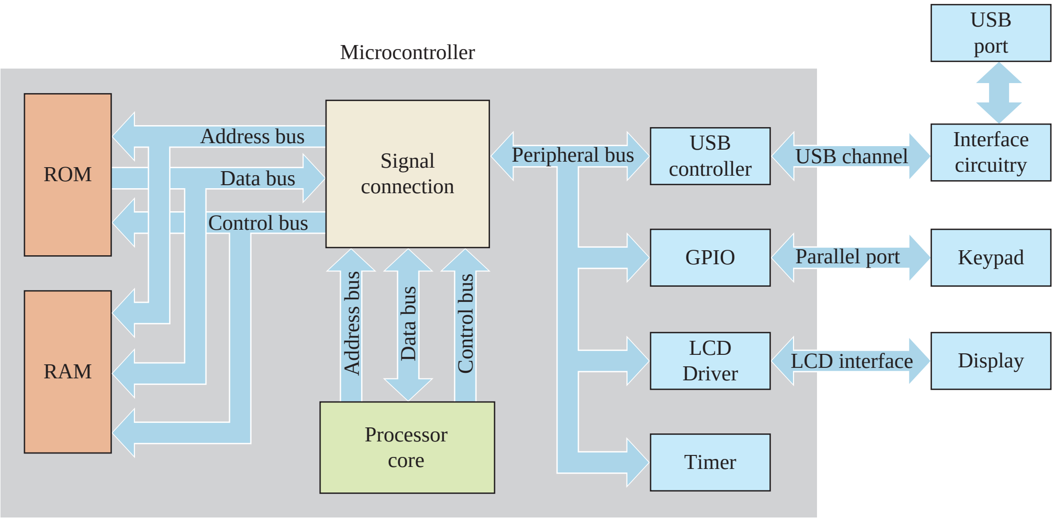


FIGURE 14–35 Microcontroller block diagram for programmable calculator.

Consumer Appliances

Virtually every electronic product today is a "smart" product that can make decisions, perform a preprogrammed sequence of events, or be manually programmed to do so. A short list of these products includes microwave ovens, coffee makers, washers and dryers, refrigerators, ovens, home entertainment components and systems, video game systems, and robotic vacuum cleaners.

Automobile Systems

Automobiles use microcontrollers in a number of embedded systems. Embedded systems in modern automobiles monitor vehicle operation and control the engine, fuel system, brakes, air bags, door locks, environmental system, instrument display, vehicle navigation, and virtually every aspect of vehicle operation. One specific factor that can affect microcontrollers in automotive applications is the operating environment. Microcontrollers must be able to operate properly when exposed to the vehicle's temperature, humidity, vibration, and electrical noise that they will encounter when the vehicle is operating.

Automated Systems

Two large areas of embedded applications are robotics and automation. Robotic and automated assemblies by nature must operate independently, perform repetitive tasks, process real-world data, and respond to circumstances that arise during operation. Embedded microcontroller systems can perform these tasks very well. One particular aspect of automated systems with which the microcontroller must deal is motion control. Microcontrollers must use feedback from the mechanical system to properly control the speed and acceleration of the system to ensure that it operates properly.

Figure 14–36 shows the block diagram for a basic robotics system. Although the block diagram is for a system that operates along a single axis, it can be extended to three axes for three-dimensional movement by using three microcontroller systems.

The ROM contains the embedded code that implements the robotic functions and processes; the RAM provides storage for the system stack and system data. The quadrature encoder receives encoded information from a motor position indicator and increments or decrements a counter depending upon how the encoded pattern changes. The pulse width modulator supplies a pulse train to a motor driver that in turn applies the voltages to the motor windings to turn the motor. GPIO lines detect when optical, magnetic, or other sensors indicate that the mechanical assembly has reached its maximum or minimum position.

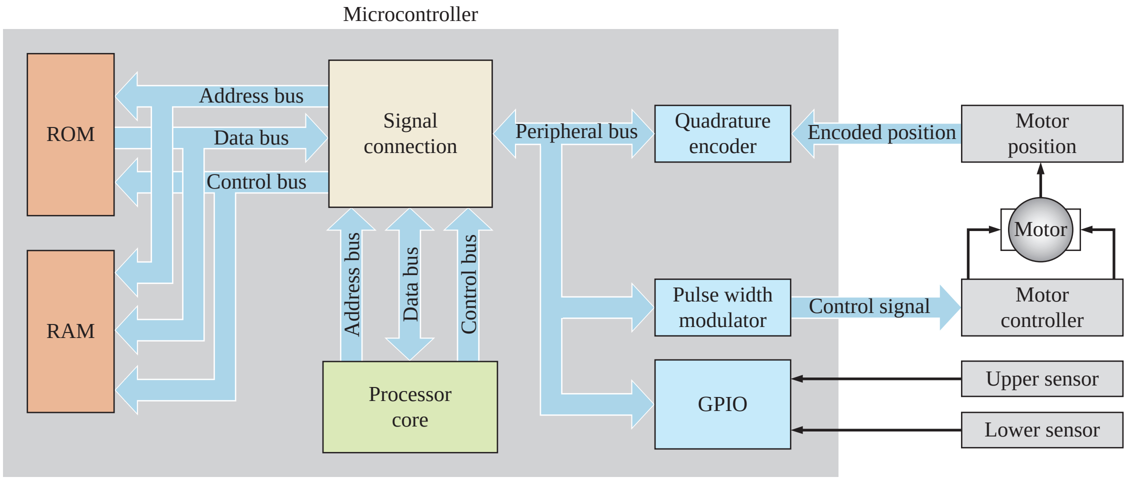


FIGURE 14–36 Basic block diagram for a robotics system.

When the system first powers up, the microcontroller uses the quadrature encoder and pulse width modulator to move the mechanical assembly to its minimum, or home, position and clears the counter so that zero corresponds to this home position and initializes the system. Once the system is initialized, the microcontroller then moves the mechanical assembly as programmed by driving the pulse width modulator to move the motor forward or backward and monitor the counter to determine how far and fast the mechanical assembly has moved. In most robotic systems, the microcontroller performs a complex series of calculations while monitoring the motor position and driving the motor to ensure that the mechanical assembly starts, stops, and operates smoothly.

SECTION 14–8 CHECKUP

1. How does a microcontroller differ from a microprocessor?
2. What are some common functional units found in a typical microcontroller?
3. Discuss an advantage and disadvantage of multifunctional pins on a microcontroller.
4. Which peripherals allow a microcontroller to interact with the real world?
5. How does an embedded system differ from a personal computer system?
6. Identify some types of embedded systems in which microcontrollers are found.

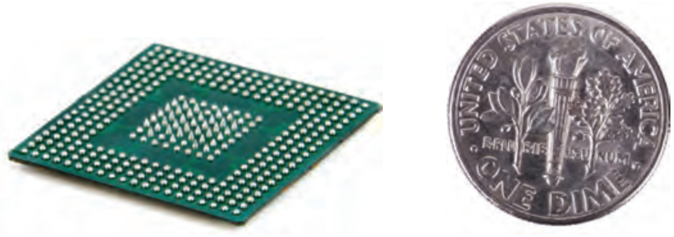
14–9 System on Chip (SoC)

The system on chip (SoC) is a major step up in complexity from the microcontroller and is what makes smaller and more powerful mobile devices possible. A SoC contains a variety of functional blocks integrated on a single semiconductor chip to meet specific application requirements. A SoC generally includes data processing, both digital and analog signal processing, data conversion, memory, interfacing, and other functions. The SoC is found in many devices such as smart phones, tablet computers, and digital cameras. Two important advantages of the SoP are small size and reduced power consumption, which make it ideal for small mobile devices.

After completing this section, you should be able to

- Describe a typical SoC
- List the functional elements of a SoC

A **system on chip (SoC)** is an integrated circuit that combines all components of a computer or other electronic system on a single chip. The SoC offers reduced manufacturing costs and smaller system configurations; Package sizes can be smaller than a dime, as shown in Figure 14–37.



(a) (b)

FIGURE 14–37 A typical SoC ball-grid package. The bottom of the package with the BG contacts is shown. (a) Boris Sosnovyy/Shutterstock (b) Eldad Carin/Shutterstock.

A typical SoC consists of the following functional elements, depending on the application:

- A single or multiple-processor (CPU) core
- A digital signal processor (DSP)
- A graphics processor (GPU)
- Memory (ROM, RAM, EEPROM, flash)
- Analog functions such as ADC and DAC
- I/O interfaces such as USB, Firewire, I^2C, USART
- Timing sources such as oscillators and phase-locked loops (PLL)
- Voltage regulators and other power management functions
- Bus bridges
- Various peripherals
- Programmable logic and application specific logic

In a system using a microprocessor as the CPU, a variety of other chips must be included to achieve full system capability. The same is true for systems using a microcontroller, although a smaller chip set may be required because the microcontroller typically has memory and some peripherals on a single chip. Actually, the microcontroller often is considered a SoC with limited functionality. The SoC provides all functions necessary for a given system application, such as a computer on a single chip. Figure 14–38 illustrates a simplified generic SoC block diagram. Actual SoCs feature a number of functions that vary from one manufacturer to another.

The CPU (central processing unit) in a typical SoC may feature one or more microprocessors (MPUs) as well as a graphics processor (GPU). Generally, SoCs use processors based on ARM architecture. The ARM processors, developed by Advanced RISC Machines, Ltd. in the 1980s, were very simple in terms of transistor count and instruction set. They used reduced instruction set computer (RISC) architecture which allowed them to have high performance and low energy consumption. The GPU (graphics processing unit) handles complex games and other video requirements that are found on smart phones, tablets, and other devices.

SoCs include various types of memory such as ROM, SRAM, DRAM, and cache as well as the accompanying control functions. A DSP (digital signal processor) is also a feature on many SoCs along with analog functions such as ADC (analog-to-digital conversion) and DAC (digital-to-analog conversion) elements. Of course, interfacing is a crucial part of any system and all SoCs provide a varying number of standard bus and other I/Os. These interfacing elements may include USB, SPI, CAN, I^2C, AGP, UART, Bluetooth,

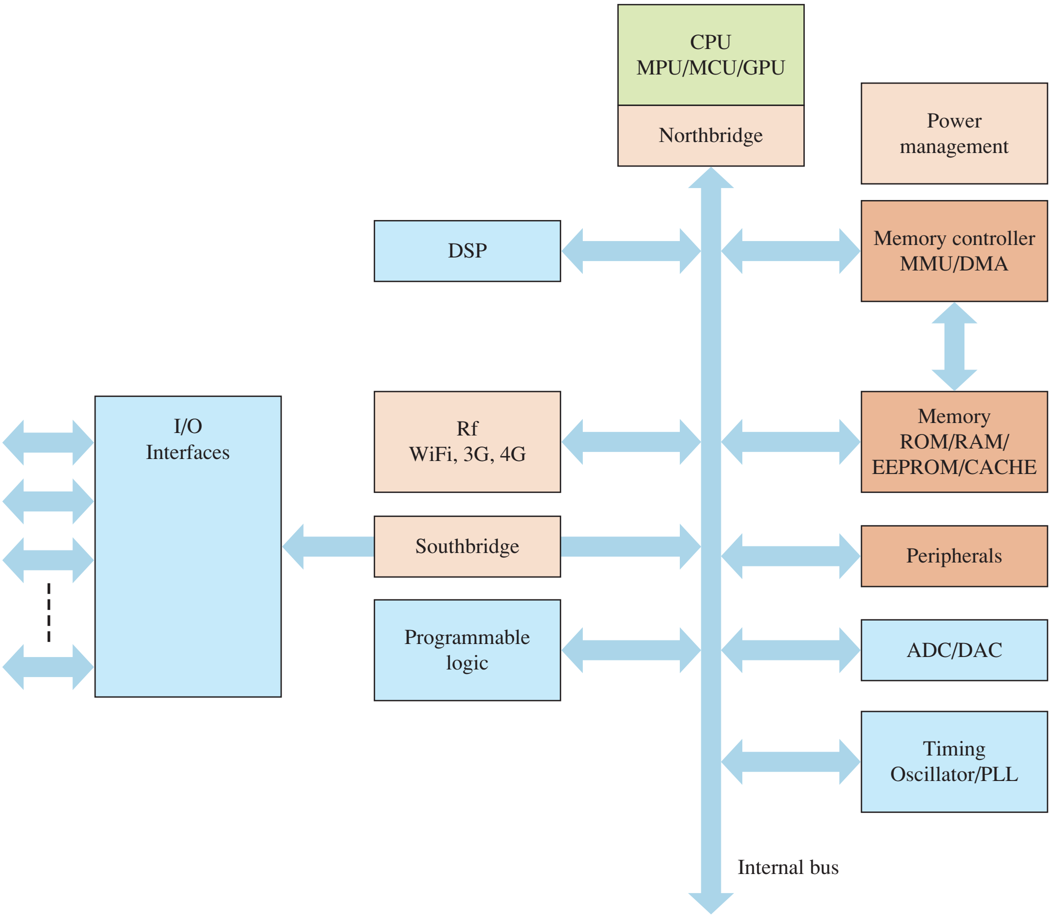


FIGURE 14–38 Generic block diagram of a typical SoC.

Wi-Fi, Ethernet, audio, rf, as well as others. The northbridge is a circuit that connect the CPU to the memory, and to the PCI internal bus. The southbridge is a circuit that controls connections to the I/Os.

SECTION 14–9 CHECKUP

1. What is a SoC?
2. List two advantages of a SoC.
3. Name at least five functional elements of a SoC.