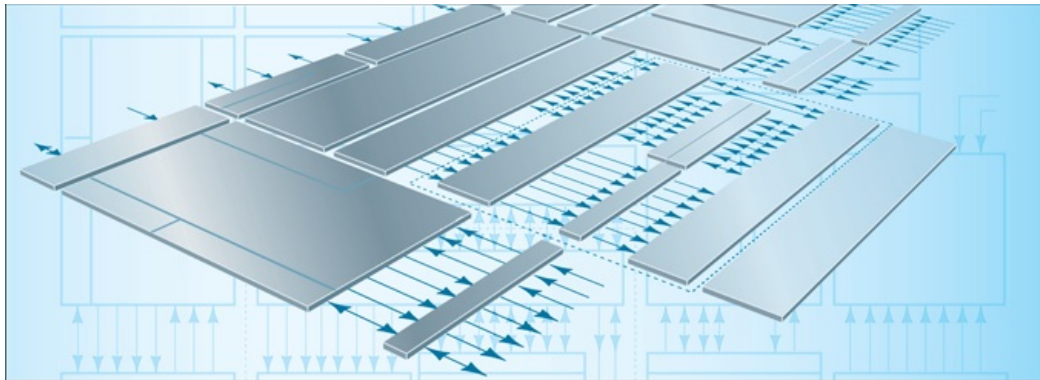

Chapter 8 Computers, Microcontrollers and Computer-Based Instrumentation Systems



Study of this chapter will enable you to:

- Identify and describe the functional blocks of a computer.
- Define the terms *microprocessor*, *microcomputer*, and *microcontroller*.
- Select the type of memory needed for a given application.
- Understand how microcontrollers can be applied in your field of specialization.
- Identify the registers in the programmer's model and their functions for the HCS12/9S12 microcontroller family from Freescale Semiconductor, Inc.
- List some of the instructions and addressing modes of HCS12/9S12 microcontrollers.
- Write simple assembly language programs, using the CPU12 instruction set.
- Describe the operation of the elements of a computer-based instrumentation system.
- Identify the types of errors that may be encountered in instrumentation systems.
- Avoid common pitfalls such as ground loops, noise coupling, and loading when using sensors.
- Determine specifications for the elements of computer-based instrumentation systems such as data-acquisition boards.

Introduction to this chapter:

*Certainly you are familiar with general-purpose electronic computers that are used for business, engineering design, word processing, and other applications. Although it is sometimes not readily apparent, special-purpose computers can be found in many products such as automobiles, appliances, cameras, fax machines, garage-door openers, and instrumentation. An **embedded computer** is part of a product that is not called a computer. Virtually any recently manufactured device that is partly electrical in nature is almost certain to contain one or more embedded computers. Typical automobiles contain over 100 embedded computers. The emphasis of this chapter is on embedded computers.*

*Relatively simple computers for embedded control applications can be completely implemented on a single silicon chip costing less than a dollar. This type of computer is often called a **microcontroller** (MCU) and is useful for problems such as control of a washing machine, a printer, or a toaster.*


An embedded computer is part of a product, such as an automobile, printer, or bread machine, that is not called a computer.

In this chapter, we give an overview of MCU organization and instruction sets using the Freescale Semiconductor HCS12/9S12 family as an example. Hundreds of types of MCUs and their variations are in use, but most of the underlying concepts are similar from one to another. The primary intent is to give you an understanding of these basic concepts. Space is not available in this book for the intensive coverage of a particular MCU needed to prepare you to design complex mechatronic systems.

Instrumentation concepts are important in systems with embedded microcontrollers. We discuss the concepts related to computer-based instrumentation in the last several sections of this chapter.

Computer capability has advanced rapidly and costs have fallen dramatically, a trend that will continue for the foreseeable future. For the past several decades, the price for a given computer capability has been cut in half about every 18 months. You should view embedded MCUs as powerful, but inexpensive, resources that are appropriate for solving virtually any control or instrumentation problem in your field of engineering, no matter how complex or mundane the problem may be.

8.1 Computer Organization

Figure 8.1  shows the system-level diagram of a computer. The **central processing unit** (CPU) is composed of the **arithmetic/logic unit** (ALU) and the **control unit**.

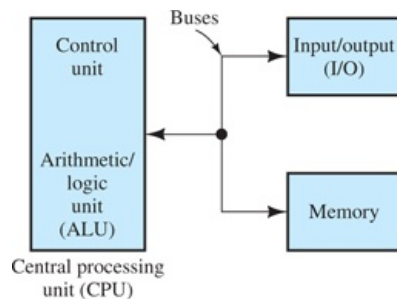




Figure 8.1

A computer consists of a central processing unit, memory, buses, and input output devices.

The ALU carries out arithmetic and logic operations on data such as addition, subtraction, comparison, or multiplication. Basically, the ALU is a logic circuit similar to those discussed in [Chapter 7](#)  (but much more complex).

The control unit supervises the operation of the computer, such as determining the location of the next instruction to be retrieved from memory and setting up the ALU to carry out operations on data. The ALU and control unit contain various **registers** that hold operands, results, and control signals. (Recall from [Section 7.6](#)  that a register is simply an array of flip-flops that can store a word composed of binary digits.) Later in this chapter, we will discuss the functions of various CPU registers for the CPU12 which is used in the Freescale HCS12/9S12 family.

Memory

Memory can be thought of as a sequence of locations that store data and instructions. Each memory location has a unique address and typically stores one byte of data, which can conveniently be represented by two hexadecimal digits. (Of course, within the computer circuits, data appear in binary form.)

Several notations are used for hexadecimal numbers, including the subscript 16, the subscript H, and the prefix \$. Thus, $F2_{16}$, $F2_H$, and $\$F2$ are alternative ways to indicate that F2 is a hexadecimal number.

Usually, memory capacity is expressed in Kbytes, where $1\text{ K} = 2^{10} = 1024$. Similarly, 1 Mbyte is $2^{20} = 1,048,576$ bytes. A 64-Kbyte memory is illustrated in [Figure 8.2](#). Under the direction of the control unit, information can either be written to or read from each memory location. (We are assuming that we have the read/write type of memory. Later, we will see that another type, known as read-only memory, can also be very useful.)

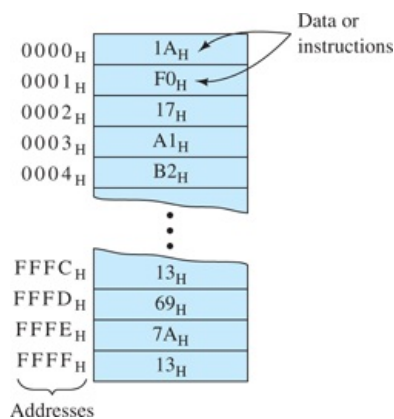


Figure 8.2

A 64-Kbyte memory that has $2^{16} = 65,536$ memory locations, each of which contains one byte (eight bits) of data. Each location has a 16-bit address. It is convenient to represent the addresses and data in hexadecimal form as shown. The addresses range from 0000_H to FFFF_H. For example, the memory location 0004_H contains the byte B2_H = 10110010₂.

Programs

Programs are sequences of instructions stored in memory. Typically, the controller fetches (i.e., retrieves) an instruction, determines what operation is called for by the instruction, fetches data from memory as required, causes the ALU to perform the operation, and writes results back to memory. Then, the next instruction is fetched, and the process is repeated. We will see that the CPU12 can execute a rich variety of instruction types.

Buses

The various elements of a computer are connected by **buses**, which are sets of conductors that transfer multiple bits at a time. For example, the **data bus** transfers data (and instructions) between the CPU and memory (or I/O devices). In small computers, the width of the data bus (i.e., the number of bits that can be transferred at a time) is typically eight bits. Then, one byte can be transferred between the CPU and memory (or I/O) at a time. (The bus is wider in more powerful general-purpose CPUs such as those found in personal computers, which typically have data-bus widths of 64 bits.)

Several **control buses** are used to direct the operations of the computer. For example, one control bus sends the addresses for memory locations (or I/O devices) as well as signals that direct whether data are to be read or written. With an address bus width of 16 bits, $2^{16} = 64K$ of memory locations (and I/O devices) can be addressed. Another control bus internal to the CPU transfers signals from the control unit to the ALU. These control signals direct the ALU to perform a particular operation, such as addition.

Buses can be bidirectional. In other words, they can transfer data in either direction. Let us consider the data bus connecting the CPU and memory. Of course, the memory and the CPU cannot apply conflicting data signals to the bus at the same time. Conflict is avoided by transferring data to the bus through **tristate buffers**, as illustrated in [Figure 8.3](#). Depending on the control signal, the tristate buffers function either as open or as closed switches. When a byte of data is to be transferred from the CPU to memory, the tristate buffers are enabled (switches closed) on the CPU and disabled (switches open) in the memory. The data inputs of both the CPU and the memory are connected to the bus at all times, so data can be accepted from the bus as desired. Thus, the data from the CPU appear on the bus and can be stored by the memory. When data are to be transferred from memory to the CPU, the conditions of the tristate buffers are reversed.

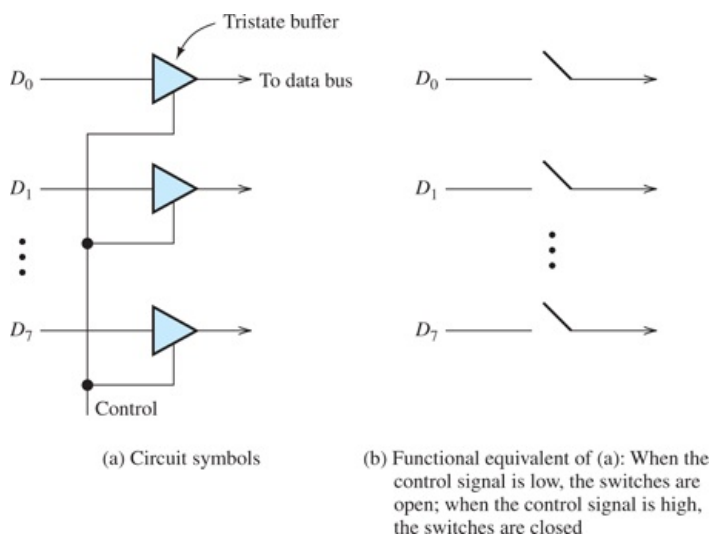


Figure 8.3

Data are applied to the data bus through tristate buffers, which can function either as closed or as open switches.

Input Output

Some examples of I/O devices are keyboards, display devices, and printers. An important category of input devices in control applications are **sensors**, which convert temperatures, pressures, displacements, flow rates, and other physical values to digital form that can be read by the computer. **Actuators** are output devices such as valves, motors, and switches that allow the computer to affect the system being controlled.

Some computers are said to have **memory-mapped I/O**, in which I/O devices are addressed by the same bus as memory locations. Then, the same instructions used for storing and reading data from memory can be used for I/O. Other computers have a separate address bus and instructions for I/O. We discuss primarily systems that use memory-mapped I/O.

A **microprocessor** is a CPU contained on a single integrated-circuit chip. The first microprocessor was the Intel 4004, which appeared in 1971 and cost several thousand dollars each. Subsequently, microprocessors have dramatically fallen in price and increased in performance. A **microcomputer**, such as a PC or a laptop, combines a microprocessor with memory and I/O chips. A MCU combines CPU, memory, buses, and I/O on a single chip and is optimized for embedded control applications. We give a more detailed overview of the HCS12/9S12 MCU family later in this chapter.

A microcontroller is a complete computer containing the CPU, memory, and I/O on a single silicon chip.

There are several variations of computer organization. In computers with **Harvard architecture**, there are separate memories for data and instructions. If the same memory contains both data and instructions, we have **von Neumann architecture**. The HCS12/9S12 MCU family uses von Neumann architecture.

Exercise 8.1

Suppose that a microprocessor has an address bus width of 20 bits. How many memory locations can it access?

Answer $2^{20} = 1,048,576 = 1024 \text{ K} = 1 \text{ M}.$

Exercise 8.2

How many bits can be stored in a 64-Kbyte memory?

Answer 524,288.

8.2 Memory Types

Several types of memory are used in computers: (1) Read-and-write memory (RAM), (2) Read-only memory (ROM), and (3) Mass storage. We discuss each type in turn. Then, we consider how to select the best type of memory for various applications.

RAM


Read-and-write memory (RAM) is used for storing data, instructions, and results during execution of a program. Semiconductor RAM consists of one or more silicon integrated circuits (each of which has many storage cells) and control logic so that information can be transferred into or out of the cell specified by the address.

Usually, the information that is stored in RAM is lost when power is removed. Thus, we say that RAM is **volatile**. Originally, the acronym RAM meant random-access memory, but the term has changed its meaning over time. As the term is used now, RAM means volatile semiconductor memory. (Actually, RAM is also available with small batteries that maintain information in the absence of other power.)

RAM and ROM do not incur any loss of speed when the memory locations are accessed in random order. In fact, RAM originally meant random-access memory.

The time required to access data in RAM is the same for all memory locations. The fastest RAM is capable of access times of a few nanoseconds. No time penalty is incurred by accessing locations in random order.

There are two types of RAM in common use. In **static RAM**, the storage cells are SR flip-flops that can store data indefinitely, provided that power is applied continuously. In **dynamic RAM**, information is stored in each cell as charge (or lack of charge) on a capacitor. Because the charge leaks off the capacitors, it is necessary to refresh the information periodically. This makes the use of dynamic RAM more complex than the use of static RAM. The advantage of dynamic RAM is that the basic storage cell is smaller, so that chips with larger capacities are available. A relatively small amount of RAM is needed in most control applications, and it is simpler to use static RAM.

An 8K-word by 8-bit static RAM chip is illustrated in [Figure 8.4](#) . The chip has 13 address lines, eight data lines, and three control lines. The “bubbles” on the control input lines indicate that they are active when low. Unless the chip select line is low, the chip neither stores data nor places data on the data bus. If both the output-enable and the chip-select inputs are low, the data stored in the location specified by the address appear on the data lines. If both the write-enable and the chip-select lines are low, the data appearing on the data bus are stored in the location specified by the address signals. In normal operation, both the output enable and write enable lines are never low at the same time.

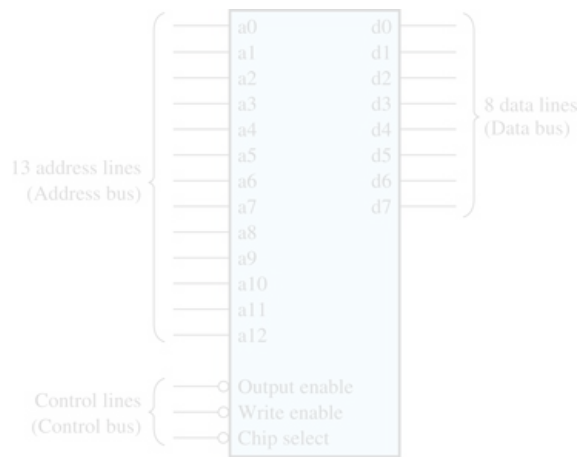


Figure 8.4

A generic 8K-word by 8-bit word RAM.

ROM

In normal operation, **read-only memory** (ROM) can be read, but not written to. The chief advantages of ROM are that data can be read quickly in random order and that information is not lost when power is turned off. Thus, we say that ROM is **non-volatile** (i.e., permanent). ROM is useful for storing programs such as the *boot program*, which is executed automatically when power is applied to a computer. In simple dedicated applications such as the controller for a clothes washer, all of the programs are stored in ROM.

The primary advantage of ROM is that it is non-volatile. Information stored in RAM is lost when power is interrupted.

Several types of ROM exist. For example, in **mask-programmable ROM**, the data are written when the chip is manufactured. A substantial cost is incurred in preparing the mask that is used to write the data while manufacturing this type of ROM. However, mask-programmable ROM is the least expensive form of ROM when the mask cost is spread over a sufficiently large number of units. Mask-programmable ROM is not a good choice if frequent changes in the information stored are necessary, as in initial system development.

In **programmable read-only memory** (PROM), data can be written by special circuits that blow tiny fuses or leave them unblown, depending on whether the data bits are zeros or ones. Thus, with PROM, we write data once and can read it as many times as desired. PROM is an economical choice if a small number of units are needed.

Erasable PROM (EPROM) is another type that can be erased by exposure to ultraviolet light (through a window in the chip package) and rewritten by using special circuits. **Electrically erasable PROMs** (EEPROMs) can be erased by applying proper voltages to the chip. Although we can write data to an EEPROM, the process is much slower than for RAM.

Flash memory is a non-volatile technology in which data can be erased and rewritten relatively quickly in blocks of locations, ranging in size from 512 bytes up to 512 Kbytes. Flash memory has a limited lifetime, typically on the order of 10 thousand to 100 thousand read/write cycles. Flash is a rapidly advancing technology and may eventually replace hard drives for mass storage in general purpose computers.

Mass Storage

Mass-storage units include hard disks and flash memory, both of which are read/write memory. Another type is CD-ROM and DVD-ROM disks, which are used for storing large amounts of data. Mass storage is the least expensive type of memory per unit of capacity. With all forms of mass storage except flash, a relatively long time is required to access a particular location. Initial access times for mass storage range upward from several milliseconds, compared with fractions of a microsecond for RAM or ROM. However, if mass-storage locations are accessed sequentially, the transfer rate is considerably higher (but still lower than for RAM or ROM). Usually, data and instructions need to be accessed quickly in random order during execution of a program. Thus, programs are stored in RAM or ROM during execution.

Hard disks, CD-ROMs, and DVDs are examples of sequential memories in which access is faster if memory locations are accessed in order.

Selection of Memory

The main considerations in choosing the type of memory to be used are:

1. The trade-off between speed and cost.
2. Whether the information is to be stored permanently or must be changed frequently.
3. Whether data are to be accessed in random order or in sequence.

In general-purpose computers, programs and data are read into RAM before execution from mass-storage devices such as hard disks. Because many different programs are used, it is not practical to store programs in semiconductor ROM, which would be too expensive for the large memory space required. Furthermore, information stored in ROM is more difficult to modify compared to data stored on a hard disk. We often find a small amount of ROM used for the startup or boot program in general-purpose computers, but most of the memory is RAM and mass storage.

On the other hand, in embedded MCUs, programs are usually stored in semiconductor ROM, and only a small amount of RAM is needed to store temporary results. For example, in a controller for a television receiver, the programs for operating the TV are stored in ROM, but time and channel information entered by the user is stored in RAM. In this application, power is applied to the RAM even when the TV is “turned off.” However, during a power failure, the data stored in RAM are lost (unless the TV has a battery backup for its RAM). Usually, we do not find mass-storage devices used in embedded computers.

8.3 Digital Process Control

Figure 8.5 shows the general block diagram of a control scheme for a physical process such as an internal combustion engine. Various physical inputs such as power and material flow are regulated by actuators that are in turn controlled by the MCU.

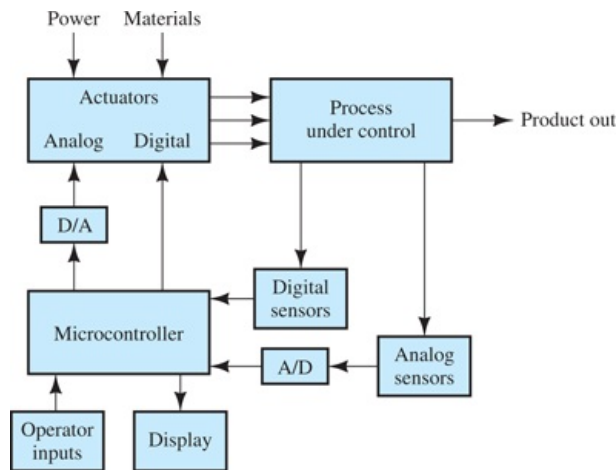


Figure 8.5

Microcontroller-based control of a physical process.

D/A and DAC are both acronyms for digital-to-analog converter.

Some actuators are analog and some are digital. Examples of digital actuators are switches or valves that are either on or off, depending on the logic value of their control signals. Digital actuators can be controlled directly by digital control lines. Analog actuators require an analog input. For example, the rudder of an airplane may deflect in proportion to an analog input signal. Then, **digital-to-analog (D/A) converters** are needed to convert the digital signals to analog form before they are applied to analog actuators.

Various sensors produce electrical signals related to process parameters (such as temperature, pressure, pH, velocity, or displacement) of the process under control. Some sensors are digital and some are analog. For example, a pressure sensor may consist of a switch that closes producing a high output signal when pressure exceeds a particular value. On the other hand, an analog pressure sensor produces an output voltage that is proportional to pressure. **Analog-to-digital (A/D) converters** are used to convert the analog sensor signals to digital form.

Often, a display is provided so that information about the process can be accessed by the operator. A keyboard or other input device enables the operator to direct operation of the control process.

Many variations of the system shown in **Figure 8.5** are possible. For example, sometimes we simply want to instrument a process and present information to the operator. This is the situation for automotive instrumentation in which sensors provide signals for speed, fuel reserve, oil pressure, engine temperature, battery voltage, and so on. These data are presented to the driver by one or more displays.

Actuators, sensors, and I/O tend to be unique to each application and do not lend themselves to integration with the MCU. A/D and D/A converters often are included within the MCU. Thus, a typical system consists of an MCU, sensors, actuators, and I/O devices. Systems may not contain all of these elements. Within a given MCU family, variations are usually available with respect to the amount and type of memory, the number of A/D channels, and so forth, that are included on the chip.

Virtually any system can be controlled or monitored by an MCU. Here is a short list: traffic signals, engines, chemical plants, antiskid brakes, manufacturing processes, stress measurement in structures, machine tools, aircraft instrumentation, monitoring of patients in a cardiac-care unit, nuclear reactors, and laboratory experiments.

Interrupts versus Polling

In many control applications, the MCU must be able to respond to certain input signals very quickly. For example, an overpressure indication in a nuclear power plant may require immediate attention. When such an event occurs, the MCU must **interrupt** what it is doing and start a program known as an **interrupt handler** that determines the source of the interrupt and takes appropriate action. Many MCUs have hardware capability and instructions for handling these interrupts.

Instead of using interrupts, an MCU can use **polling** to determine if any parts of the system need attention. The processor checks each sensor in turn and takes appropriate actions as needed. However, continuous polling is wasteful of processor time. In complex applications, the processor may be required to carry out extensive, but lower-priority activities, much of the time. In this case, interrupts provide faster response to critical events than polling does. For a breadmaker, polling would be acceptable because no activity ties up the MCU for more than a few milliseconds. Furthermore, any of the actions required could be delayed by a few tens of milliseconds without undue consequences.


PRACTICAL APPLICATION

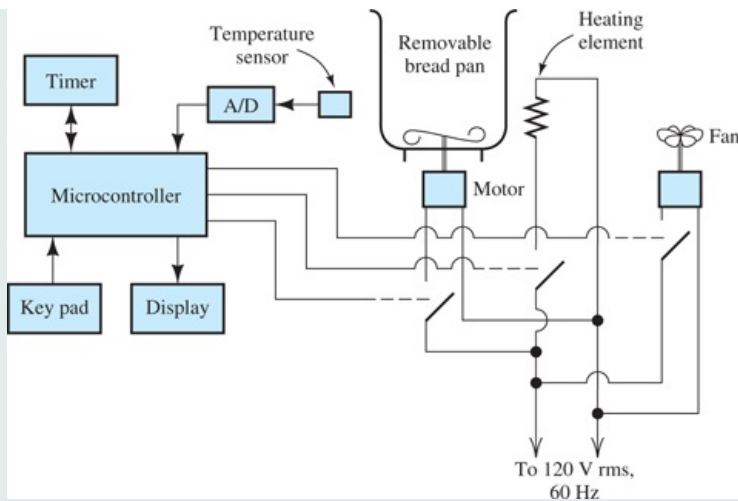
8.1



Fresh Bread Anyone?

Let us consider a relatively simple MCU application: a breadmaker. Possibly, you have had experience with this popular appliance. The chef measures and adds the ingredients (flour, water, dried milk, sugar, salt, yeast, and butter) to the bread pan, makes selections from the menu by using a keypad, and takes out a finished loaf of bread after about four hours.

The diagram of a bread machine is shown in [Figure PA8.1](#) . There are three digital actuators in the bread machine: a switch to control the heating element, a switch for the mixing and kneading motor, and a switch for the fan used to cool the loaf after baking is finished. Analog actuators are not needed in this application.



FIGURE

PA8.1

A relatively simple application for an MCU—a breadmaking machine.

An analog sensor is used to measure temperature. The sensor output is converted to digital form by an A/D converter.

A timer circuit is part of the MCU that is initially loaded with the time needed to complete the loaf. The timer is a digital circuit that counts down, similar to the counter circuits discussed in [Section 7.6](#).

The timer indicates the number of hours and minutes remaining in the process. The MCU can read the time remaining and use it to make decisions. Time remaining to completion of the loaf is also displayed for the convenience of the chef.

The control programs are stored in ROM. The parameters entered by the chef are written into RAM (for example, whether the bread crust is to be light, medium, or dark). The MCU continually checks the time remaining and the temperature. By executing the program stored in ROM, the computer determines when the machine should mix the ingredients, turn on the heating element to warm the dough and cause it to rise, knead the dough, bake, or cool down. The duration and temperature of the various parts of the cycle depend on the initial selections made by the chef.

First, the machine mixes the ingredients for several minutes, and the heating element is turned on to warm the yeast that makes the dough rise. While the dough is rising, a warm temperature is required, say 90°F. Thus, the heating element is turned on and the MCU reads the temperature frequently. When the temperature reaches the desired value, the heating element is turned off. If the temperature falls too low, the element is again turned on.

The MCU continues to check the time remaining and the temperature. According to the programs stored in ROM and the parameters entered by the chef (which are saved in RAM), the motors and heating element are turned on and off.

In this application, about 100 bytes of RAM would be needed to store information entered by the operator and temporary data. Also, about 16 Kbytes of ROM would be needed to store the programs. Compared to the total price of the appliance, the cost of this amount of ROM is very small. Therefore, many variations of the program can be stored in ROM, and bread machines can be very versatile. In addition to finished loaves of bread, they can also bake cakes, cook rice, make jam, or prepare dough for other purposes, such as cinnamon rolls.

8.4 Programming Model for the HCS12/9S12 Family

Earlier, we discussed a generic computer shown in [Figure 8.1](#) on page 409. In this section, we give a more detailed internal description of the HCS12/9S12 MCU family from Freescale Semiconductor. Space does not allow us to discuss all of the features, instructions, and programming techniques for these MCUs. However, we will describe the programming model, selected instructions, and a few simple programs to give you a better understanding of how MCUs can be used for embedded applications that you will encounter in your field.

The HCS12/9S12 Programming Model

The ALU and the control unit contain various registers that are used to hold operands, the address of the next instruction to be executed, addresses of data, and results. For example, the programmer's model for the CPU12 is illustrated in [Figure 8.6](#). (Actually, the MCUs contain many other registers—only the registers of concern to the programmer are shown in the figure; thus, [Figure 8.6](#) is often called the programming model.)

The **accumulators** are general-purpose registers that hold one of the arguments and the result of all arithmetic and logical operations. Registers A and B each contain 8 bits with the least significant bit on the right (bit 0 in [Figure 8.6](#)) and the most significant bit on the left. Sometimes A and B are used as separate registers, and other times they are used in combination as a single 16-bit register, denoted as register D. It is important to remember that D is not separate from A and B.

It is important to remember that register D is not separate from registers A and B.

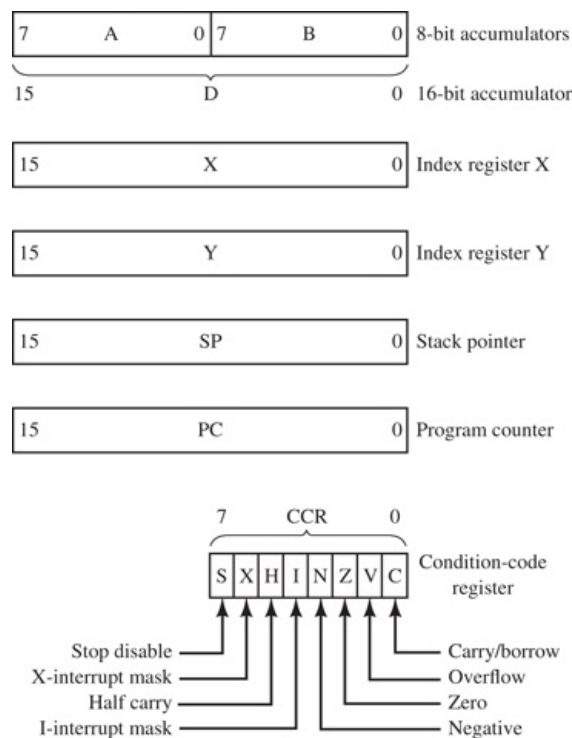


Figure 8.6
The CPU12 programmer's model.

The **program counter** (PC) is a 16-bit register that contains the address of the first byte of the next instruction to be fetched (read) from memory by the control unit. The size of the PC is the same as the size of memory addresses; thus, the memory potentially contains up to $2^{16} = 64\text{ K}$ locations, each of which contains one byte of data or instructions as illustrated in [Figure 8.2](#) on page 410.

The **index registers** X and Y are mainly used for a type of addressing (of data) known as indexed addressing, which we will discuss later.

The **condition-code register** (CCR) is an 8-bit register in which each bit depends either on a condition of the processor or on the result of the preceding logic or arithmetic operation. The details of the CCR are shown in [Figure 8.6](#). For example, the carry bit C (bit 0 of the CCR) is set (to logic 1) if a carry (or borrow) occurred in the preceding arithmetic operation. Bit 1 (overflow or V) is set if the result of the preceding operation resulted in overflow or underflow. Bit 2 (zero or Z) is set to 1 if the result of the preceding operation was zero. Bit 3 (negative or N) is set if the result was negative. The meaning and use of the remaining bits will be discussed as the need arises.

Stacks and the Stack Pointer Register

A **stack** is a sequence of locations in memory used to store information such as the contents of the program counter and other registers when a subroutine is executed or when an interrupt occurs. (We discuss subroutines shortly.) As the name implies, information is added to (pushed onto) the top of the stack and later read out (pulled off) in the reverse order that it was written. This is similar to adding plates to the top of a stack when clearing a dinner table and then taking the plates off the top of the stack when loading a dishwasher. After data are pulled off the stack, they are considered to no longer exist in memory and are written over by later push commands. The first word pushed onto the stack is the last to be pulled off, and stacks are called **last-in first-out memories** (LIFOs).

Stacks are last-in first-out memories. Information is added to (pushed onto) the top of the stack and eventually read out (pulled off) in the reverse order that it was written.

The **stack pointer** is a CPU register that keeps track of the address of the top of the stack. Each time the content of a register is pushed onto the stack, the content of the stack pointer is decremented by one if the register contained one byte. If the register contained two bytes, the stack-pointer content is decreased by two. (Addresses are smaller in value as we progress upward in the stack.) Conversely, when data is pulled from the stack and transferred to a register, the stack-pointer content is increased by one or two (depending on the length of the register). When the content of one of the 8-bit registers (A, B, or CCR) is pushed onto the stack (by the commands PSHA, PSHB, or PSHC, respectively), these operations take place:

1. The content of the stack pointer is reduced by one.
2. The content of the 8-bit register is stored at the address corresponding to the content of the stack pointer.

When the content of one of the 16-bit registers D, X, or Y is pushed onto the stack (by the commands PSHD, PSHX, or PSHY), the following operations take place:

1. The content of the stack pointer is decremented by one and the least significant byte (bits 8 through 15) of the content of the 16-bit register is stored at the address corresponding to the content of the stack pointer.
2. The content of the stack pointer is again decremented by one, and the most significant byte of the content of the 16-bit register is stored at the address corresponding to the content of the stack pointer.

In pulling data off of the stack, the operations are reversed. For an 8-bit register (commands PULA, PULB, or PULC):

1. The data in the memory location pointed to by the stack pointer is stored in the register.
2. The content of the stack pointer is incremented by one.

For a 16-bit register (commands PULD, PULX, or PULY):

1. The data in the memory location to which the stack pointer points is stored in the high byte of the register, and the content of the stack pointer is incremented by one.
2. The data in the memory location to which the stack pointer points is stored in the low byte of the register, and the content of the stack pointer is again incremented by one.

Figure 8.7 illustrates the effects of the command sequence PSHA, PSHB, PULX. **Figure 8.7(a)** shows the original contents of pertinent registers and memory locations. (Memory locations always contain something; they are never blank. However, when the content of a memory location is unknown or does not matter, we have left the location blank.) **Figure 8.7(b)** shows the new contents after the command PSHA has been executed. Notice that the initial content of register A has been stored in location 090A and that the content of SP has been decremented by one. (Furthermore, the initial contents of A and B are unchanged.) **Figure 8.7(c)** shows the contents after the command PSHB has been executed. Notice that the content of register B has been stored in location 0909 and that the content of SP has been decremented by one. Finally, **Figure 8.7(d)** shows the new contents after the command PULX has been executed. Notice that the content of memory location 0909 has been stored in the first byte of register X and the content of memory location 090A has been stored in the second byte of register X. Also, the content of SP has been increased by two.

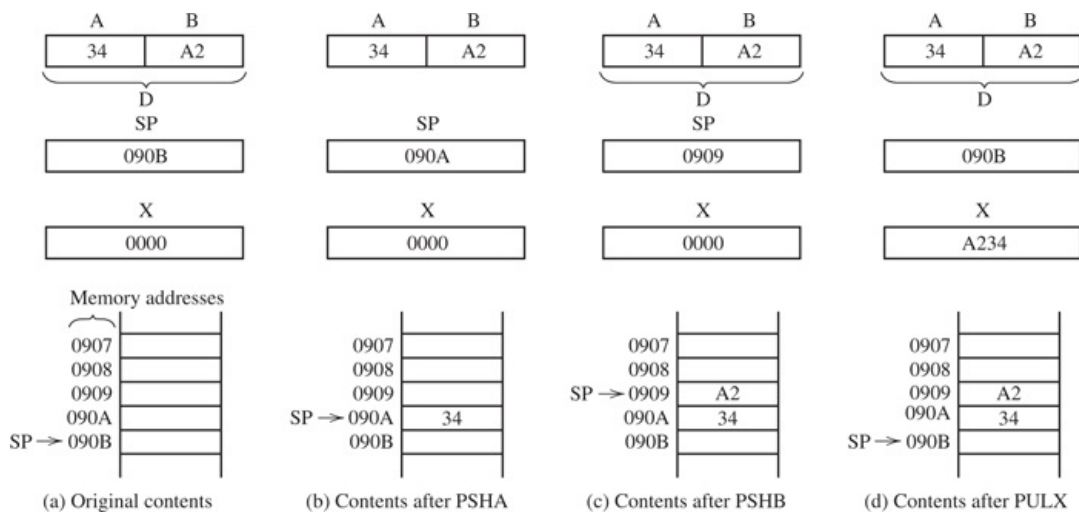


Figure 8.7

Register and memory contents for the command sequence: PSHA, PSHB, PULX.

Exercise 8.3

Starting from the initial contents shown in **Figure 8.7(a)**, determine the content of register X after execution of the command sequence PSHB, PSHA, PULX.

Answer The content of the X register is 34A2.

Exercise 8.4

Starting from the initial contents shown in **Figure 8.7(a)**, determine the content of register X after the command sequence PSHX, PSHA, PULX.

Answer The content of the X register is 3400ss.

Exercise 8.5

Suppose that initially the contents of all memory locations in the stack are 00, and the stack pointer register contains 0806. Then, the following operations occur in sequence:

1. The data byte $A7_H$ is pushed onto the stack.
2. 78_H is pushed onto the stack.
3. One byte is pulled from the stack.
4. FF is pushed onto the stack.

List the contents of memory locations 0800 through 0805 after each step. Also, give the content of the stack pointer (SP).

Answer After step 1, we have:

```
0800: 00    SP: 0805
0801: 00
0802: 00
0803: 00
0804: 00
0805: A7
```

After step 2, we have:

```
0800: 00    SP: 0804
0801: 00
0802: 00
0803: 00
0804: 78
0805: A7
```

After step 3, we have:

```
0800: 00    SP: 0805
0801: 00
0802: 00
0803: 00
0804: 78
0805: A7
```

After step 4, we have:

```
0800: 00    SP: 0804
0801: 00
0802: 00
0803: 00
0804: FF (New data replaces old.)
0805: A7
```


8.5 The Instruction Set and Addressing Modes for the CPU12

Computers excel at executing simple instructions, such as quickly and accurately adding a number stored in a given memory location to the content of a specified register. Computers are capable of highly sophisticated and seemingly intelligent behavior by following instruction sequences called **programs** or **software**. These are prepared by a human programmer.

Unfortunately, even the smallest oversight on the part of the programmer can render a program useless until the error is corrected. A substantial part of the effort in designing an MCU-based controller is in writing software. To be effective, the programmer must be fully knowledgeable about the fine details of the instruction set for the MCU in use. Our objective in this and the next section is simply to give you a brief overview, not to make you an expert programmer.

A substantial part of the effort in designing an MCU-based controller is in writing software.

In general, instruction sets are similar between different MCU types, but details differ. Once one has mastered programming of a given machine, it is much easier to learn and make good use of the instruction set of another processor. Here again, we take the CPU12 as an example. More details about it can be readily found on the Web.

Instructions for the CPU12

A selected set of instructions for the CPU12 is listed in [Table 8.1](#). The first column in the table gives the mnemonic for each instruction, the second column is a brief description and an equivalent Boolean expression for the instruction. For example, the ABA instruction adds the content of register B to the content of A with the result residing in A. We can indicate this operation as

$$(A) + (B) \rightarrow A$$

as shown in the second column of the table.

(A) represents the *content* of register A.

Table 8.1. Selected Instructions for the CPU12

Source Form	Operation	Addr. Mode	Machine Code	Condition Codes							
				S	X	H	I	N	Z	V	C
ABA	Add Accumulators $(A) + (B) \rightarrow A$	INH	18 06	-	-	↑	-	↑	↑	↑	↑
ADDA (opr)	Add Memory to A $(A) + (M) \rightarrow A$	IMM	8B ii	-	-	↑	-	↑	↑	↑	↑
		DIR	9B dd								
		EXT	BB hh								
		IDX	ii								
		AB	*								
ADDB (opr)	Add Memory to B $(B) + (M) \rightarrow B$	IMM	CB ii	-	-	↑	-	↑	↑	↑	↑
		DIR	DB dd								
		EXT	FB hh								
		IDX	ii								
		--	.								

			EB *									
ADDD (opr)	Add Memory to D (D) + (M: M + 1) → D	IMM DIR EXT IDX	C3 <i>jj</i> <i>kk</i> D3 <i>dd</i> F3 <i>hh</i> <i>ll</i> E3 *	-	-	-	-	↑	↑	↑	↑	
BCS (rel)	Branch if Carry Set (if C = 1)	REL	25 <i>rr</i>	-	-	-	-	-	-	-	-	
BEQ (rel)	Branch if Equal (if Z = 1)	REL	27 <i>rr</i>	-	-	-	-	-	-	-	-	
BLO (rel)	Branch if Lower ^U (if C = 1)	REL	25 <i>rr</i>	-	-	-	-	-	-	-	-	
BMI (rel)	Branch if Minus ^S (if N = 1)	REL	2B <i>rr</i>	-	-	-	-	-	-	-	-	
BNE (rel)	Branch if Not Equal (if Z = 0)	REL	26 <i>rr</i>	-	-	-	-	-	-	-	-	
BPL (rel)	Branch if Plus ^S (if N = 0)	REL	2A <i>rr</i>	-	-	-	-	-	-	-	-	
BRA (rel)	Branch Always	REL	20 <i>rr</i>	-	-	-	-	-	-	-	-	
CLRA	Clear Accumulator A ≥ 00 → A	INH	87	-	-	-	-	0	1	0	0	
CLRB	Clear Accumulator B ≥ 00 → B	INH	C7	-	-	-	-	0	1	0	0	
COMA	Complement Accumulator A (\bar{A}) → A	INH	41	-	-	-	-	↑	↑	0	1	
INCA	Increment Accumulator A (A) + ≥ 01 → A	INH	42	-	-	-	-	↑	↑	↑	-	
INCB	Increment Accumulator B (B) + ≥ 01 → B	INH	52	-	-	-	-	↑	↑	↑	-	
INX	Increment Index Register X (X) + ≥ 0001 → X	INH	08	-	-	-	-	-	↑	-	-	
JMP (opr)	Jump Routine Address → PC	EXT IDX	06 <i>hh</i> <i>ll</i> 05 *	-	-	-	-	-	-	-	-	
JSR (opr)	Jump to Subroutine (See Text)	DIR EXT IDX	17 <i>dd</i> 16 <i>hh</i> <i>ll</i> 15 *									
LDAA (opr)	Load Accumulator A (M) → A	IMM DIR EXT IDX	86 <i>ii</i> 96 <i>dd</i> B6 <i>hh</i> <i>ll</i> A6 *	-	-	-	-	↑	↑	0	-	
LDAB (opr)	Load Accumulator B (M) → B	IMM DIR EXT IDX	C6 <i>ii</i> D6 <i>dd</i> F6 <i>hh</i> <i>ll</i> E6 *	-	-	-	-	↑	↑	0	-	
LDD (opr)	Load Accumulator D (M) : (M + 1) → D	IMM DIR EXT IDX	CC <i>jj</i> <i>kk</i> DC <i>dd</i> FC <i>hh</i> <i>ll</i> EC *	-	-	-	-	↑	↑	0	-	
LDX (opr)	Load Index Register X (M) : (M + 1) → X	IMM DIR EXT	CE <i>jj</i> <i>kk</i> DE <i>dd</i>	-	-	-	-	↑	↑	0	-	

		IDX	FE ^{hh} _{ll} EE *								
LDY (opr)	Load Index Register Y (M) : (M + 1) → Y	IMM DIR DD ^{dd} EXT FD ^{hh} _{ll} IDX ED *	CD ^{jj} _{kk}	-	-	-	-	↑	↑	0	-
MUL	Multiply A by B ^U (A) × (B) → D	INH	12	-	-	-	-	-	-	-	↑
NOP	No Operation	INH	A7	-	-	-	-	-	-	-	-
PSHA	Push A onto Stack (SP) − 1 ⇒ SP; (A) ⇒ M _(SP)	INH	36	-	-	-	-	-	-	-	-
PSHB	Push B onto Stack (SP) − 1 ⇒ SP; (B) ⇒ M _(SP)	INH	37	-	-	-	-	-	-	-	-
PSHX	Push X onto Stack (SP) − 2 ⇒ SP; (X _H : X _L) ⇒ M _(SP) : M _(SP + 1)	INH	34	-	-	-	-	-	-	-	-
PSHY	Push Y onto Stack (SP) − 2 ⇒ SP; (Y _H : Y _L) ⇒ M _(SP) : M _(SP + 1)	INH	35	-	-	-	-	-	-	-	-
PULA	Pull A from Stack (M _(SP)) ⇒ A; (SP) + 1 ⇒ SP	INH	32	-	-	-	-	-	-	-	-
PULB	Pull B from Stack (M _(SP)) ⇒ B; (SP) + 1 ⇒ SP	INH	33	-	-	-	-	-	-	-	-
PULX	Pull X from Stack (M _(SP) : M _(SP + 1)) ⇒ X _H : X _L ; (SP) + 2 ⇒ SP	INH	30	-	-	-	-	-	-	-	-
PULY	Pull Y from Stack (M _(SP) : M _(SP + 1)) ⇒ Y _H : Y _L ; (SP) + 2 ⇒ SP	INH	31	-	-	-	-	-	-	-	-
RTS	Return from Subroutine (M _(SP) : M _(SP + 1)) ⇒ PC; (SP) + 2 ⇒ SP	INH	3D	-	-	-	-	-	-	-	-
STAA (opr)	Store Accumulator A (A) → M	DIR EXT IDX	5A ^{dd} 7A ^{hh} _{ll} 6A *	-	-	-	-	↑	↑	0	-
STAB (opr)	Store Accumulator B (B) → M	DIR EXT IDX	5B ^{dd} 7B ^{hh} _{ll} 6B *	-	-	-	-	↑	↑	0	-
STD (opr)	Store Accumulator D (A) → M; (B) → M + 1	DIR EXT IDX	5C ^{dd} 7C ^{hh} _{ll} 6C *	-	-	-	-	↑	↑	0	-
STOP	Stop Internal Clocks. If S control bit = 1, the STOP instruction is disabled and acts like a NOP	INH	18 3E	-	-	-	-	-	-	-	-
TSTA	Test Accumulator A; (A) − 00	INH	97	-	-	-	-	↑	↑	0	0
TSTB	Test Accumulator B; (B) − 00	INH	D7	-	-	-	-	↑	↑	0	0
<p>S indicates instructions that are intended for two's complement signed numbers U indicates instructions that are intended for unsigned numbers * For indexed addressing (IDX). Only the first byte of the machine coded is given. An additional one to three bytes are needed.</p> <p>The details are beyond the scope of our discussion.</p> <p>ll 8-bit immediate data dd 16-bit absolute address</p>											


Mnemonics are easy for humans to remember. However, in the microcomputer memory, the instructions are stored as machine codes (or op codes) consisting of one or more 8-bit numbers, each of which is represented in the table as a two-digit hexadecimal number. For example, in the row for the ABA

instruction, we see that the op code is 1806. Thus, the ABA instruction appears in memory as the binary numbers 00011000 and 00000110.

Look at the row for the ADDA(opr) instruction in which (opr) stands for the address of a memory location. The effect of the instruction is to add the content of a memory location to the content of accumulator A with the result residing in A. This is represented by the expression

$$(A) + (M) \rightarrow A$$

in which (M) represents the content of a memory location. Several **addressing modes** can be used to select the memory location to be accessed by some instructions. For example, the ADDA instruction can use any of several addressing modes. We will discuss the CPU12 addressing modes shortly.

Table 8.1  also shows the effect of each instruction on the contents of the CCR. The meanings of the symbols shown for each bit of the condition code are

-	the bit is unchanged by this instruction
0	the bit is always cleared by this instruction
1	the bit is always set by this instruction
↕	the bit is set or cleared depending on the result

The CPU12 has many more instructions than those listed in the table; we have just given a sample of various kinds. Next, we briefly describe each of the addressing modes used by the CPU12.

Extended (EXT) Addressing

Recall that the CPU12 uses 16 bits (usually written as four hexadecimal digits) for memory addresses. In extended addressing, the complete address of the operand is included in the instruction. Thus, the instruction

```
ADDA $CA01
```

In CPU12 assembly language, a prefix of \$ indicates that the number is hexadecimal.

adds the content of memory location CA01 to the content of register A. (Later, we will see that a program called an assembler is used to convert the mnemonics to op codes. The \$ sign indicates to the assembler that the address is given in hexadecimal form.) The op codes appear in three successive memory locations as

BB	(op code for ADDA with extended addressing)
CA	(high byte of address)
01	(low byte of address)

Notice that the high byte of the address is given first followed by the low byte.

Direct (DIR) Addressing

In **direct addressing**, only the least significant two (hexadecimal) digits of the address are given, and the most significant two digits are assumed to be zero. Therefore, the effective address falls between 0000 and 00FF. For example, the instruction

```
ADDA $A9
```

adds the content of memory location 00A9 to the content of register A. The instruction appears in two successive memory locations as

9B	(the op code for ADDA with direct addressing)
A9	(the low byte of the address)

Notice that the same result could be obtained by using extended addressing, in which case the instruction would appear as

```
ADDA $00A9
```

However, the extended addressing form of the instruction would occupy three bytes of memory, rather than two with direct addressing. Furthermore, the direct addressing form is completed more quickly.

Inherent (INH) Addressing

Some instructions, such as ABA, access only the MCU registers. We say that this instruction uses **inherent addressing**. An instruction sequence that adds the numbers in locations 23A9 and 00AA, then stores the result in location 23AB is

LDAA \$23A9	(extended addressing, load A from location 23A9)
LDAB \$AA	(direct addressing, load B from location 00AA)
ABA	(inherent addressing, add B to A)
STAA \$23AB	(extended addressing, store result in 23AB)

Immediate (IMM) Addressing

In CPU12 assembly language, the symbol # indicates immediate addressing.

In **immediate addressing**, which is denoted by the symbol#, the address of the operand is the address immediately following the instruction. For example, the instruction ADDA #\$83 adds the hexadecimal number 83 to the contents of A. It is stored in two successive memory locations as

8B	(op code for ADDA with immediate addressing)
83	(operand)

Because A is a single-byte register, only one byte of memory is needed to store the operand.

On the other hand, D is a double-byte (16-bit) register, and its operand is assumed to occupy two memory bytes. For example, the instruction ADDD #\$A276 adds the two-byte hexadecimal number A276 to the contents of D. It is stored in three successive memory locations as

C3	(op code for ADDD with immediate addressing)
A2	(high byte of operand)
76	(low byte of operand)

Indexed (IDX) Addressing

Indexed addressing is useful when we want to access a list of items either one after another in order, or perhaps skipping forward or backward through the list by twos, threes, etc. The CPU12 has a variety of

indexed addressing options. (In [Table 8.1](#), we have grouped all of these options under the IDX label. Only the first bite of the machine code, of two to four in all, is given in the table. This has been done because of space limitations in this chapter.)

Constant-Offset Indexed Addressing. In constant-offset indexed addressing, the effective address is formed by adding a signed offset to the content of a selected CPU register (X, Y, SP, or PC). The contents of X, Y, SP, or PC are not changed in this type of addressing. Suppose that X contains \$1005 and Y contains \$200A. Then, some examples of source code using this type of addressing and their effects are

STAA 5,X	store the content of A in location \$100A
STD - 3, Y	store the content of D in locations \$2007 and \$2008
ADDB \$A,X	add the content of location \$100F to register B

Accumulator-Offset Indexed Addressing. In this form of addressing, the content of one of the accumulators (A, B, or D) is added as an unsigned number to the content of a designated register (X, Y, SP, or PC) to obtain the effective address. For example, if X contains \$2000 and A contains \$FF, the command

LDAB A,X

loads the content of memory location \$20FF into B. The contents of A and X remain the same after the command is completed as they were before the command.

Next, we discuss four types of indexed addressing that increment or decrement the contents of the selected CPU register (X, Y, or SP) either before or after the instruction is carried out. The amount of the increment or decrement can range from 1 to 8 and the selected CPU register contains the incremented or decremented value after the instruction is completed.

Auto Pre-Incremented Indexed Addressing. Suppose that X contains \$1005. Then, the instruction

STAA 5,+X

pre-increments X so the content of X becomes \$100A and the content of A is stored in location \$100A. X contains \$100A after the completion of the instruction.

Auto Pre-Decrement Indexed Addressing. Again, suppose that X contains \$1005. Then, the instruction

STD 5,-X

pre-decrements X so the content of X becomes \$1000 and the content of D is stored in locations \$1000 and \$1001. X contains \$1000 after the completion of the instruction.

Notice that the sign preceding X determines whether we have a pre-increment or a pre-decrement. On the other hand, if the algebraic sign comes after the register name, we have either a post-increment or post-decrement. Here again, the increment or decrement can range from 1 to 8.

Auto Post-Incremented Indexed Addressing. Suppose that X contains \$1005. Then, the instruction

STAA 5,X+

stores the content of A in location 1005 and then increments X so the content of X becomes \$100A. As before, the increment can vary from 1 to 8.

Auto Post-Decremented Indexed Addressing. Again, suppose that X contains \$1005. Then, the instruction

STAA 3,X-

stores the content of A in location \$1005 and then decrements X so the content of X becomes \$1002.

Indirect Indexed Addressing. In this type of addressing, a 16-bit (or equivalently, four-digit hexadecimal) constant given in the command is added to the content of a selected CPU register (X, Y, SP, or PC). This results in a pointer to a location containing the address of the operand. To illustrate an example, first assume that the contents of the CPU registers and some of the memory locations are as shown in [Figure 8.8](#). Then, if the command

LDY [\$1002, X]

is executed, \$1002 is added to the content of X resulting in \$2003. The content of locations \$2003 and \$2004 contain the high byte and low byte for the starting address of the operand. Thus, the starting address of the operand is \$3003. Finally, the content of locations \$3003 and \$3004 is written to register Y. Thus, after the execution of this command Y contains \$A3F6.

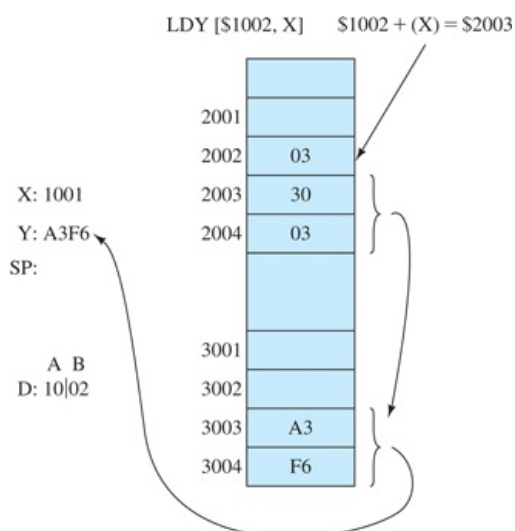


Figure 8.8

Illustration of the command `LDY [$1002, X]`.

Instead of specifying an offset value in the command, the content of register D can be used. Thus, in [Figure 8.8](#), the command

`LDY[D,X]`

adds the content of D to the content of X again resulting in \$2003. The content of locations \$2003 and \$2004 is \$3003. This is the (initial) address of the operand. So \$A3F6 is loaded into register Y as it was for the command `LDY[$1002,X]`.

To recap, the steps followed by indexed indirect addressing are:

1. Add either the offset or the content of D to the content of the CPU register (X, Y, SP, or PC) specified in the command.
2. Go to the address resulting from step 1. The content of this location and the next is the address of the operand.

Relative Addressing

Branch instructions are used to alter the sequence of program flow. Recall that if we add two numbers, the Z bit of the condition code register is clear if the result was not zero and is set if the result was zero. The BEQ (branch if result equals zero) command can be used to change the program flow depending on the value of the Z bit.

In the CPU12, branch instructions use only relative addressing. Conversely, branches are the only instructions that use relative addressing.

For example, suppose that initially the A register contains FF and the B register contains 01. Then, if the instruction ABA is executed, the binary addition shown in [Figure 8.9\(a\)](#) is performed. Because the result is zero, the Z bit of the condition code register is set. If the branch instruction BEQ \$05 is executed, the next instruction is the content of the program counter plus the offset (which is 05 in this case). If the Z bit had been clear, the instruction immediately following the branch instruction would have been executed. This is illustrated in [Figure 8.9\(b\)](#).

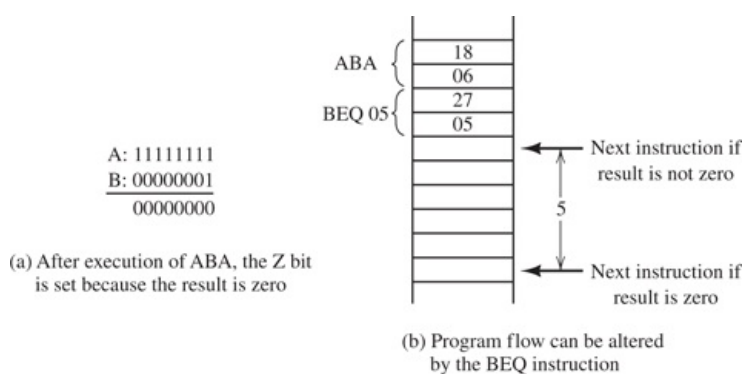


Figure 8.9

Using the BEQ instruction.

Machine Code and Assemblers

We have seen that ADDA is a mnemonic for an instruction executed by the processor. It turns out that in the CPU12, the instruction ADDA with extended addressing is stored in memory as $BB = 10111011_2$. We say that BB is the **machine code** for the instruction ADDA with extended addressing. Machine codes are also known as **operation codes** or simply **op codes**. In extended addressing, the address of the operand is stored in the two memory locations immediately following the instruction code. The instruction ADDA \$070A appears in three successive memory locations as

BB
07
0A

It would be a daunting task for a human to make conversions from instruction mnemonics to machine codes, whereas computers excel at this type of task. Furthermore, mnemonics are much easier for us to remember than machine codes. Thus, we generally start writing a program by using mnemonics. A computer program called an **assembler** is then employed to convert the mnemonics into machine code. Assemblers also help in other chores associated with programming, such as converting decimal numbers to hexadecimal and keeping track of branching addresses and operand addresses. We will have more to say about assembly language in the next section.

Exercise 8.6

Suppose that the contents of certain memory locations are as shown in [Figure 8.10](#). Furthermore, the content of register A is zero and the X register contains 0200 prior to execution of each of these instructions:

0200	10	0000	EF
0201	04	0001	01
0202	1A	0002	AE
0203	10	0003	F1
0204	07	0004	78
0205	FF	0005	96
0206	A3	0006	13
0207	16	0007	A4

Figure 8.10

Contents of memory for [Exercise 8.6](#).

- LDAA \$0202
- LDAA #\$43
- LDAA \$05,X
- LDAA \$06
- LDAA \$07,X-
- LDAA \$05,+X

Find the contents of registers A and X after each instruction.

Answer

- Register A contains 1A and X contains 0200.
- Register A contains 43 and X contains 0200.
- Register A contains FF and X contains 0200.
- Register A contains 13 and X contains 0200
- Register A contains 10 and X contains 01F9.
- f. Register A contains FF and X contains 0205.n

Exercise 8.7

Suppose that starting in location 0200 successive memory locations contain op codes for the instructions

```
CLRA  
BEQ $15
```

- Show the memory addresses and contents (in hexadecimal form) for these instructions.
- What is the address of the instruction executed immediately after the branch instruction?

Answer

- The memory addresses and contents are:

0200:87	(op code for CLRA)
0201:27	(op code for BEQ)
0202:15	(offset for branch instruction)

- The address of the next instruction is 0218.n

8.6 Assembly-Language Programming

A program consists of a sequence of instructions used to accomplish some task. No doubt, you have been introduced to programming that uses high-level languages such as BASIC, C, Java, MATLAB, or Pascal. When using high-level languages, a **compiler** or **interpreter** converts program statements into machine code before they are executed. It would be much too tedious to write machine-language programs for sophisticated engineering analysis. In application-oriented software, such as computer-aided design packages, even greater emphasis is placed on making the programs easy to use. However, in writing programs for embedded computers in control applications, we often need to keep the number of instructions relatively small and to minimize the time required to execute the various operations; quick response to events in the system being controlled can be highly important.

Though we may program MCUs for control applications in machine language, it is possible to relieve much of the drudgery by using an **assembler**. This provides many conveniences, such as allowing us to write instructions with mnemonics, using labels for memory addresses, and including user in the source program file.

In practice, we write the program as **source code** using a text editor on a general-purpose computer, called the **host computer**. The source code is then converted to **object code** (machine code) by the assembler program. Finally, the machine code is loaded into the memory of the MCU, which is called the **target system**. Sometimes, we say that the source code is written in assembly language. Assembly language code, nevertheless, is very close to the actual machine code executed by the computer.

In general, CPU12 assembly language statements take the following form:

LABEL	INSTRUCTION/DIRECTIVE	OPERAND	COMMENT
-------	-----------------------	---------	---------

Typically each line of source code is converted into one machine instruction. Some of the source code statements, called **directives**, however, are used to give commands to the assembler. One of these is the origin directive ORG. For example,

ORG	\$0100		
-----	--------	--	--

instructs the assembler to place the first instruction following the directive in memory location 0100 of the target system.

In CPU12 assembly language, labels must begin in the first column. The various fields are separated by spaces. Thus, when we want ORG to be treated as a directive, rather than as a label, we need to place one or more spaces ahead of it. If the first character of a line is a semicolon, the line is ignored by the assembler. Such lines are useful for comments and line spaces that make the source code more understandable to humans.

Usually, there are many ways to write a program to accomplish a given task.

In writing a program, we start by describing the algorithm for accomplishing the task. We then create a sequence of instructions to carry out the algorithm. Usually, there are many ways to write a program to accomplish a given task.

Example 8.1 An Assembly-Language Program

Suppose that we want a program starting in memory location 0400 that retrieves the number stored in location 0500, adds 5 to the number, writes the result to location 0500, and then stops. (We will use only the instructions listed in [Table 8.1](#), even though the CPU12 has many additional instructions, which often could make our programs shorter.)

Solution

The source code is:

```
; SOURCE CODE FOR EXAMPLE 8.1
; THIS LINE IS A COMMENT THAT IS IGNORED BY THE ASSEMBLER
;
      ORG      $0400      ;ORIGIN DIRECTIVE
BEGIN  LDAA     $0500      ;LOAD NUMBER INTO A
      ADDA     #$05        ;ADD 5, IMMEDIATE ADDRESSING
      STAA     $0500      ;STORE RESULT
      STOP
      END              ;END DIRECTIVE
```

Comments have been included to explain the purpose of each line. BEGIN is a label that identifies the address of the LDAA instruction. (In this case, BEGIN has a value of 0400.) If we wanted to reference this location somewhere in a more complex program, the label would be useful. STOP is the mnemonic for the instruction that halts further action by the MCU. END is a directive that informs the assembler that there are no further instructions. ■

Example 8.2 Absolute Value Assembly Program

Write the source code for a program starting in location \$0300 that loads register A with the signed two's-complement number in location \$0200, computes its absolute value, returns the result to location \$0200, clears the A register, and then stops. Use the instructions listed in [Table 8.1](#). (Assume that the initial content of location \$0200 is never $1000000 = -128_{10}$ which does not have a positive equivalent in 8-bit two's complement form.)

Solution

Recall that branch instructions (also known as conditional instructions) allow different sets of instructions to be executed depending on the values of certain bits in the condition code register. For example, in [Table 8.1](#), we see that the branch on plus instruction (BPL) causes a branch if the N bit of the condition code register is clear (i.e., logic 0).

Testing occurs automatically in many instructions. For example, in the load A instruction LDAA, the N and Z bits of the condition code register are set if the value loaded is negative or zero, respectively.

Our plan is to load the number, compute its two's complement if it is negative, store the result, clear the A register, and then stop. Recall that one way to find the two's complement is to first find the one's complement and add one. If the number is positive, no calculations are needed. The source code is:

```
; SOURCE CODE FOR EXAMPLE 8.2
;
      ORG      $0300      ;ORIGIN DIRECTIVE
      LDAA     $0200      ;LOAD NUMBER INTO REGISTER A
      BPL      PLUS      ;BRANCH IF A IS POSITIVE
      COMA     ;ONES' S COMPLEMENT
      INCA     ;ADD ONE TO FORM TWO'S COMPLEMENT
      STAA     $0200      ;RETURN THE RESULT TO MEMORY
PLUS   CLRA     ;CLEAR A
      STOP
      END          ;DIRECTIVE
```

In this program, the number is first loaded into register A from memory location \$0200. If the number is negative (i.e., if the most significant bit is 1), the N bit of the CCR is set (logic 1); otherwise, it is not set. If the N bit is zero, the BPL PLUS instruction causes the next instruction executed to be the one starting in the location labeled PLUS. On the other hand, if the N bit is one, the next instruction is the one immediately following the branch instruction. Thus, if the content of the memory location is negative, the two's complement is computed to change its sign. Then, the result is written to the original location and A is cleared.■

Next, to illustrate some of the chores performed by the assembler, we manually convert the source code of the previous example to machine code.

Example 8.3 Manual Conversion of Source Code to Machine Code

Manually determine the machine code for each memory location produced by the source code of [Example 8.2](#). What is the value of the label PLUS? (Hint: Use [Table 8.1](#) to determine the op codes for each instruction.)

Solution

The assembler ignores the title and other comments. Because of the ORG directive, the machine code is placed in memory starting at location 0300. The memory addresses and their contents are:

```
0300: B6 Op code for LDAA with extended addressing.
0301: 02 High byte of address.
0302: 00 Low byte of address.
0303: 2A Op code for BPL which uses relative addressing.
0304: 05 Offset (On the first pass this value is unknown.)
0305: 41 Op code for COMA which computes one's complement.
0306: 42 Op code for INCA.
0307: 7A Op code for STAA with extended addressing.
0308: 02 High byte of address.
0309: 00 Low byte of address.
030A: 87 Clear A.
030B: 18 Halt processor.
030C: 3E
```

A comment has been added to explain each line; however, the assembler does not produce these comments.

Recall that branch instructions use relative addressing. The required offset for the BPL command is not known on the first pass through the source code. However, after the first pass, we see that the location corresponding to the label PLUS is 030A and that an offset of 05 is needed for the BPL command. The END directive does not produce object code.■

Subroutines

Sometimes, certain sequences of instructions are used over and over in many different places. Memory is saved if these sequences are stored once and used wherever needed. A sequence of instructions such as this is called a **subroutine**. At any point in the main program that the subroutine needs to be executed, we place the *Jump to SubRoutine* instruction:

```
JSR address
```

in which address is a direct, extended, or indexed address of the first instruction of the subroutine. At the end of the subroutine, we place the *ReTurn from Subroutine* command

```
RTS
```

which causes the next instruction to be taken from the location following the JSR instruction in the main program. This is illustrated in [Figure 8.11](#).

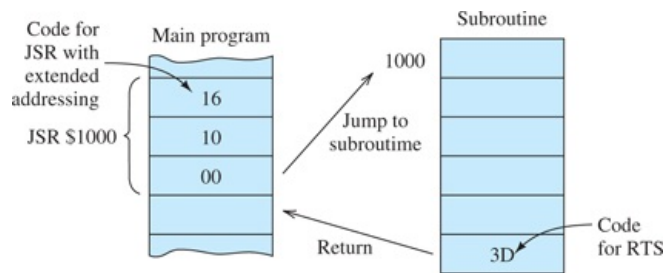


Figure 8.11

Illustration of the jump-to-subroutine command with extended addressing. (Other addressing modes are allowed with the JSR instruction.)

The stack is used to keep track of where to return after the subroutine is finished. The address of the instruction following the JSR is pushed onto the system stack when the jump is executed. This address is pulled off the stack and loaded into the program counter when the return instruction is executed. After the subroutine is completed, the next instruction executed is the one following the JSR.

One of the chores that the assembler can perform is to keep track of the starting addresses of the subroutines. We simply label the first instruction of the subroutine in the source code. This is convenient because we usually don't know where subroutines will eventually be located when writing programs. After all of the source code is written, the assembler can calculate the amount of memory needed for each portion of the program and determine the subroutine starting addresses, which are then substituted for the labels.

Example 8.4 Subroutine Source Code

Assume that the content of register A is a signed two's-complement number n . Using the instructions of [Table 8.1](#), write a subroutine called SGN that replaces the content of A with +1 (in signed two's-complement form) if n is positive, replaces it with -1 if n is negative, and does not change the content of A if n is zero.

Solution

The source code for the subroutine is:

```
; SOURCE CODE FOR SUBROUTINE OF EXAMPLE 8.4
;
SGN      TSTA          ;TEST CONTENT OF A
        BEQ      END      ;BRANCH IF A IS ZERO
        BPL      PLUS     ;BRANCH IF A IS POSITIVE
        LDAA     #$FF     ;LOAD -1, IMMEDIATE ADDRESSING
        JMP      END      ;JUMP TO END OF SUBROUTINE
PLUS     LDAA     #$01     ;LOAD +1, IMMEDIATE ADDRESSING
END      RTS           ;RETURN FROM SUBROUTINE
```

First, the number is tested. If it is zero, the Z flag is set. If it is negative, the N flag is set. Next, if the number is zero, the BEQ instruction compels a branch to END, causing a return from the subroutine. If the number is positive, the subroutine branches to PLUS, loads the hexadecimal code for the signed two's-complement representation of +1, and returns. If the number is negative, the LDA #FF instruction is executed, followed by the return. [Notice that in this subroutine, END is a label (rather than a directive) because it begins in column 1.]■

Exercise 8.8

Write a program starting in location \$0100 that adds

52_{10}

to the content of location \$0500, stores the result in location \$0501, and then stops. Assume that all values are represented in signed two's-complement form.

Answer The source code is:

```
; SOLUTION FOR Exercise 8.8
;
    ORG      $0100
    LDAA     #$34      ;LOAD HEX EQUIVALENT OF 52 BASE TEN
    ADDA     $0500      ;ADD CONTENT OF 0500
    STAA     $0501      ;STORE RESULT IN 0501
    STOP
    END              ;DIRECTIVE
```

Exercise 8.9

Write a subroutine named MOVE that tests the content of register A. If A is not zero, the subroutine should move the content of location 0100 to 0200. Otherwise, no move is made. The contents of A, B, X, and Y must be the same on return as before the subroutine is called.

Answer One version of the desired subroutine is:

```
; SUBROUTINE FOR Exercise 8.9.
;
MOVE   TSTA           ;TEST CONTENT OF A
BEQ    END            ;BRANCH IF CONTENT OF A IS ZERO
        PSHA          ;SAVE CONTENT OF A ON STACK
        LDAA $0100     ;LOAD CONTENT OF MEMORY LOCATION 0100
        STAA $0200     ;STORE CONTENT OF A IN LOCATION 0200
        PULA          ;RETRIEVE ORIGINAL CONTENT OF A
END     RTS           ;RETURN FROM SUBROUTINE
```

Resources for Additional Study

In this chapter, we have given a very brief overview of MCUs focusing on the HCS12/9S12 family of devices from Freescale Semiconductor. Much more detail about the HCS12/9S12 can be found at www.freescale.com.

We have emphasized assembly language programming because it gives a clear picture of the inner workings of MCUs. As MCUs have gained higher speed and complexity, the trend has been away from assembly language and toward higher-level languages such as C.

If you are interested in applying MCUs to a project of your own, you should take a course devoted to MCUs exclusively or a course in mechatronics. Hands-on work is very important in learning about MCUs. Typically, one starts with a training board containing the MCU of interest, provisions for prototyping, LEDs, a small display, switches, and other components. Such boards are often equipped with an interface so programs can be downloaded to the MCU from a host computer through a USB link. An example for the HCS12 MCU is the Dragon12-Plus-USB board from EVBplus.com.

Design and construction of robots, model railroads, remotely controlled model airplanes equipped with video cameras, and so forth can become an engrossing hobby for those with an interest in combining MCUs, mechanical systems, and electronic elements. For numerous examples, look at *Nuts and Volts Magazine* at www.nutsvolts.com or *Make Magazine* at www.makezine.com. These may give you some good ideas for a senior project.

The Arduino MCU boards are very popular with artists, do-it-yourself enthusiasts, and students. You will find many articles related to these boards in *Nuts and Volts* and *Make Magazine*. Also, look at <http://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino/0>.

8.7 Measurement Concepts and Sensors

Overview of Computer-Based Instrumentation

Computer-based instrumentation systems consist of four main elements: sensors, a DAQ board, software, and a general-purpose computer.

Figure 8.12 shows a computer-based system for instrumentation of a physical system such as an automobile or chemical process. Physical phenomena such as temperatures, angular speeds, displacements, and pressures produce changes in the voltages, currents, resistances, capacitances, or inductances of the **sensors**. If the sensor output is not already a voltage, **signal conditioners** provide an **excitation source** that transforms the changes in electrical parameters to voltages. Furthermore, the signal conditioner amplifies and filters these voltages. The conditioned signals are input to a **data-acquisition** (DAQ) board. On the DAQ board, each of the conditioned signals is sent to a **sample-and-hold circuit** (S/H) that periodically samples the signal and holds the value steady while the **multiplexer** (MUX) connects it to the (A/D or ADC) that converts the values to digital words. The words are read by the computer, which then processes the data further before storing and displaying the results. For example, the signals derived from a force sensor and a velocity sensor could be multiplied to obtain a plot of power versus time. Furthermore, the power could be integrated to show energy expended versus time. Long-term statistical analysis of a process can be carried out to facilitate quality control.

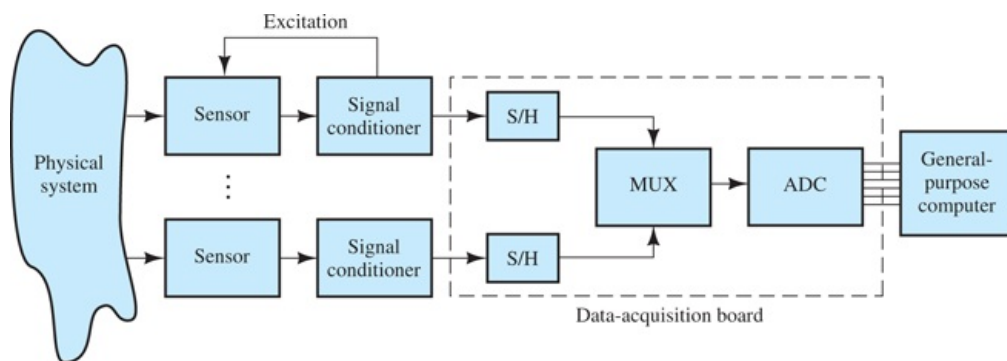


Figure 8.12
Computer-based DAQ system.

In this section, we consider sensors. Then, in the next several sections, we discuss other aspects of computer-based DAQ systems.

Sensors

We emphasize sensors or signal conditioners that produce electrical signals (usually voltages) which are analogous to the physical quantity, or **measurand**, to be measured. Often, the voltage is proportional to the measurand. Then, the sensor voltage is given by

in which V_{sensor} is the voltage produced by the sensor, K is the **sensitivity constant**, and m is the measurand. For example, a **load cell** is a sensor consisting of four strain-gauge elements (see page 30) connected in a Wheatstone bridge (see page 107) and bonded to a load-bearing element. As force is applied to the load cell, a proportional voltage appears across two terminals of the bridge. **Excitation** in

(8.1)

the form of a constant voltage is applied to the other two terminals of the bridge. For a given excitation voltage, the sensitivity constant has units of V/N or V/lbf.

A good source of detailed information about computer-based instrumentation and control, including sensors, is the National Instruments, website: www.ni.com.

Some examples of measurands and sensor types are shown in [Table 8.2](#). These are only a few of the many types of sensors that are available.

Table 8.2 Measurands and Sensor Types

Measurands	Sensor types
Acceleration	Seismic mass accelerometers
	Piezoelectric accelerometers
Angular displacement	Rotary potentiometers
	Optical shaft encoders
	Tachometric generators
Light	Photoconductive sensors
	Photovoltaic cells
	Photodiodes
Liquid level	Capacitance probes
	Electrical conductance probes
	Ultrasonic level sensors
	Pressure sensors
Linear displacement	Linear variable differential transformers (LVDTs) (see page 152)(LVDTs)
	Strain gauges (see page 30)
	Potentiometers
	Piezoelectric devices
	Variable-area capacitance sensors
Force/torque	Load cells
	Strain gauges
Fluid flow	Magnetic flowmeters (see page 758)
	Paddle wheel sensors

	Constriction-effect pressure sensors
	Ultrasonic flow sensors
Gas flow	Hot-wire anemometers
Pressure	Bourdon tube/linear variable differential transformer combinations
	Capacitive pressure sensors
Proximity	Microswitches
	Variable-reluctance proximity sensors
	Hall-effect proximity sensors
	Optical proximity sensors
	Reed-switch sensors
Temperature	Diode thermometers
	Thermistors
	Thermocouples

Equivalent Circuits and Loading

An equivalent circuit that applies to many sensors is shown in [Figure 8.13](#); the source voltage V_{sensor} is analogous to the measurand, and R_{sensor} is the Thévenin resistance. Frequently, as part of the signal conditioning, the sensor voltage must be amplified. [Figure 8.13](#) shows the sensor connected to the input terminals of an amplifier. (Amplifiers are discussed in [Chapter 10](#).) Looking into the input terminals of any amplifier, we see a finite impedance, which is represented as R_{in} in [Figure 8.13](#). Using the voltage-division principle, we have

Because of the current flowing through the circuit, the amplifier input voltage is less than the internal (8.2) voltage of the sensor. This effect is known as **loading**. Loading is unpredictable and, therefore, undesirable. Provided that R_{in} is very large compared with R_{sensor} , the amplifier input voltage is nearly equal to the internal sensor voltage. Thus, when we need to measure the internal voltage of the sensor, we should specify a signal-conditioning amplifier having an input impedance that is much larger in magnitude than the Thévenin impedance of the sensor.

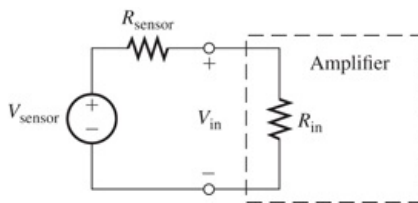


Figure 8.13

Model for a sensor connected to the input of an amplifier.

When we need to measure the internal voltage of the sensor, we should specify a signal-conditioning amplifier having an input impedance that is much larger in magnitude than the Thévenin impedance of the sensor.

Example 8.5 Sensor Loading

Suppose that we have a temperature sensor for which the open-circuit voltage is proportional to temperature. What is the minimum input resistance required for the amplifier so that the system sensitivity constant changes by less than 0.1 percent when the Thévenin resistance of the sensor changes from 15k Ω to 5k Ω ?

Solution

The sensitivity constant is proportional to the voltage division ratio between the input resistance and the Thévenin resistance of the sensor. We require that this ratio changes by 0.1% (or less) when the Thévenin resistance changes. Thus, with resistances in k Ω , we have

$$V_{\text{sensor}} \frac{R_{\text{in}}}{15 + R_{\text{in}}} \geq 0.999 V_{\text{sensor}} \frac{R_{\text{in}}}{5 + R_{\text{in}}}$$

Solving, we determine that R_{in} is required to be greater than 9985k Ω . ■

Sensors with Electrical Current Output

Some types of sensors produce electrical current that is proportional to the measurand. For example, with suitable applied voltages, photodiodes produce currents that are proportional to the light intensities falling on the diodes. A photodiode is shown in [Figure 8.14\(a\)](#). Like the load cell, the photodiode requires a constant-voltage excitation source. [Figure 8.14\(b\)](#) shows diode current versus diode voltage for various light intensities. If we want the current to depend only on light intensity, the diode voltage must be held nearly constant.

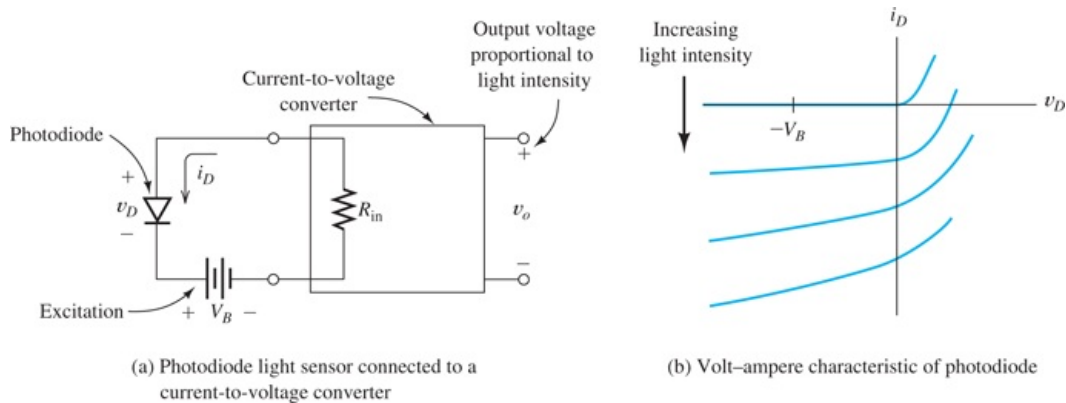


Figure 8.14

Photodiode light-sensing system. Because the diode voltage should be constant, R_{in} should ideally equal zero.

Usually a **current-to-voltage converter** (also known as a transresistance amplifier) is used to produce an output voltage that is proportional to the photodiode current. As in the case of amplifiers, we see an impedance looking into the input terminals of the current-to-voltage converter. This is shown as R_{in} in [Figure 8.14\(a\)](#). In order for the diode voltage to remain constant as the current varies, R_{in} must be very small (so the voltage across it is negligible). Thus, when we want to sense the current produced by a sensor, we use a current-to-voltage converter having a very small (ideally zero) input impedance magnitude.

When we want to sense the current produced by a sensor, we need a current-to-voltage converter having a very small (ideally zero) input impedance magnitude.

Variable-Resistance Sensors

Other sensors produce a changing resistance in response to changes in the measurand. For example, the resistance of a thermistor changes with temperature. Changes in resistance can be converted to changes in voltage by driving the sensor with a constant current source. To avoid loading effects, the voltage is applied to a high-input impedance amplifier as illustrated in [Figure 8.15](#). Similar circuits that use ac excitation can convert changes in capacitance or inductance into voltage changes.

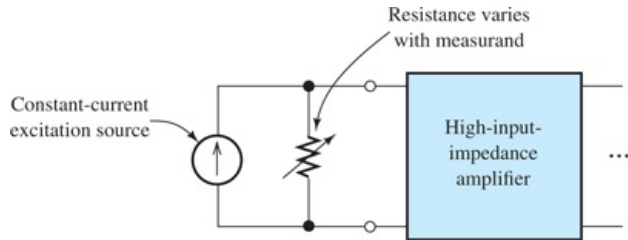


Figure 8.15
Variable-resistance sensor.

Errors in Measurement Systems

Many types of errors can occur in making measurements. We define the error of a measurement as

$$\text{Error} = x_m - x_{\text{true}} \quad (8.3)$$

in which x_m is the measured value and x_{true} is the actual or true value of the measurand. Often, error is expressed as a percentage of the full-scale value x_{full} (i.e., the maximum value that the system is designed to measure).

$$\text{Percentage error} = \frac{x_m - x_{\text{true}}}{x_{\text{full}}} \times 100\% \quad (8.4)$$

There are many possible sources of error, some of which are specific to particular measurands and measurement systems. However, it is useful to classify the types of errors that can occur. Some are **bias errors**, also called **systematic errors**, that are the same each time a measurement is repeated under the same conditions. Sometimes, bias errors can be quantified by comparing the measurements with more accurate standards. For example, we could calibrate a weight scale by using it to measure the weights of highly accurate standards of mass. Then, the calibration data could be used to correct subsequent weight measurements.

Bias errors include **offset**, **scale error**, **nonlinearity**, and **hysteresis**, which are illustrated in [Figure 8.16](#). Offset consists of a constant that is added to, or subtracted from, the true value. Scale error produces measurement errors that are proportional to the true value of the measurand. Nonlinearity can result from improper design or overdriving an electronic amplifier. When hysteresis error is present, the error depends on the direction and distance from which the measurand arrived at its current value. For example, hysteresis can result from static friction in measuring displacement or from materials effects in sensors that involve magnetic fields. All types of bias error are potentially subject to slow **drift** due to aging and changes in environmental factors such as temperature or humidity.

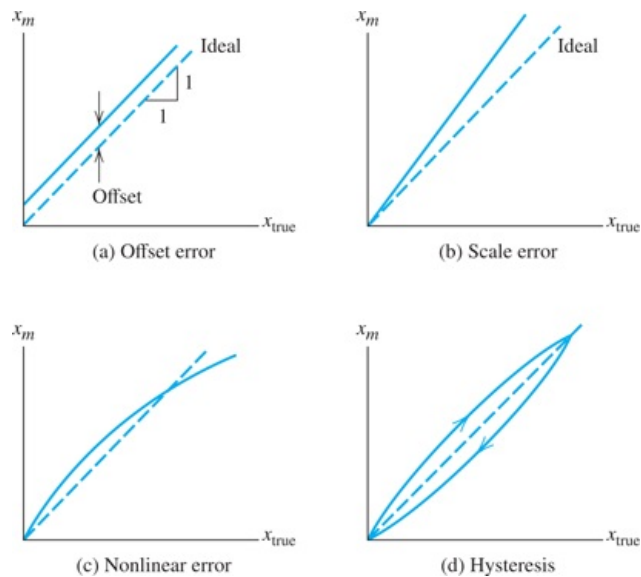


Figure 8.16

Illustration of some types of instrumentation error. x_m represents the value of the measurand reported by the measurement system, and x_{true} represents the true value.

Although bias errors are the same for each measurement made under apparently identical conditions (except for drift), **random errors** are different for each instance and have zero average value. For example, in measuring a given distance, friction combined with vibration may cause repeated measurements to vary. We can sometimes reduce the effect of random errors by making repeated measurements and averaging the results.

Some additional terms used in rating instrumentation performance are as follows:

1. **Accuracy:** The maximum expected difference in magnitude between measured and true values (often expressed as a percentage of the full-scale value).
2. **Precision:** The ability of the instrument to repeat the measurement of a constant measurand. More precise measurements have less random error.
3. **Resolution:** The smallest possible increment discernible between measured values. As the term is used, higher resolution means smaller increments. Thus, an instrument with a five-digit display (e.g., 0.0000 to 9.9999) is said to have higher resolution than an otherwise identical instrument with a three-digit display (e.g., 0.00 to 9.99).

Exercise 8.10

Suppose that a given magnetic flow sensor has an internal resistance (say, with variations in the electrical conductivity of the fluid) that varies from 5 k Ω to 10 k Ω . The internal (open-circuit) voltage of the sensor is proportional to the flow rate. Suppose that we want the changes in the sensitivity constant of the measurement system (including loading effects) to vary by less than 0.5 percent with changes in sensor resistance. What specification is required for the input resistance of the amplifier in this system?

Answer The input resistance of the amplifier must be greater than 990 k Ω .

Exercise 8.11

- a. Can a very precise instrument be very inaccurate?
- b. Can a very accurate instrument be very imprecise?

Answer

- a. Yes. Precision implies that the measurements are repeatable; however they could have large bias errors.
- b. No. If repeated measurements vary a great deal under apparently identical conditions, some of the measurements must have large errors, and therefore must be inaccurate.

8.8 Signal Conditioning

Some functions of signal conditioners are amplification of the sensor signals, conversion of currents to voltages, supply of (ac or dc) excitations to the sensors so that changes in resistance, inductance, or capacitance are converted to changes in voltage, and filtering to eliminate noise or other unwanted signal components. Signal conditioners are often specific to particular applications. For example, a signal conditioner for a diode thermometer may not be appropriate for use with a thermocouple.

Single-Ended versus Differential Amplifiers

Often, the signal from the sensor is very small (one millivolt or less) and an important step in signal conditioning is amplification. Thus, the sensor is often connected to the input terminals of an amplifier. In an amplifier with a **single-ended input**, one of its input terminals is grounded as shown in [Figure 8.17\(a\)](#), and the output voltage is a gain constant A times the input voltage.

An amplifier with a **differential input** is shown in [Figure 8.17\(b\)](#). Differential amplifiers have non-inverting and inverting input terminals as indicated in the figure, and ideally, the output is the differential gain A_d times the difference between the input voltages.

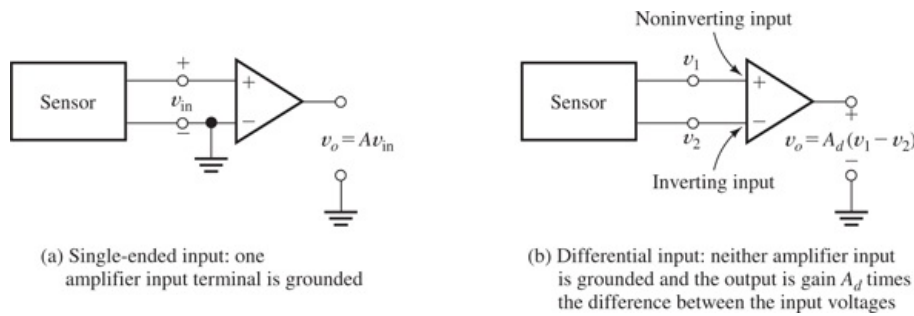


Figure 8.17

Amplifiers with single-ended and differential input terminals.

A model for the voltages produced by a typical sensor connected to a differential amplifier is shown in [Figure 8.18](#). The difference between the amplifier input voltages is the **differential signal**:

$$v_d = v_1 - v_2 \quad (8.5)$$

Sometimes, a large **common-mode signal** is also present, which is given by

$$v_{cm} = \frac{1}{2} (v_1 + v_2) \quad (8.6)$$

"

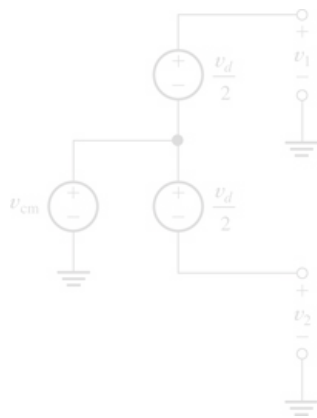


Figure 8.18

Model for a sensor with differential and common-mode components.

Almost always, the differential signal is of interest, and the common-mode signal represents unwanted noise. Thus, it is often very important for the differential amplifier to respond only to the differential signal. Great care must be taken in designing amplifiers that reject large common-mode signals sufficiently. A measure of how well a differential amplifier rejects the common-mode signal is the **common-mode rejection ratio** (CMRR). When large common-mode signals are present, it is important to select a differential amplifier with a large CMRR. **Instrumentation amplifiers** are very good in this respect. Differential amplifiers, CMRR, and instrumentation amplifiers are discussed at length in [Chapters 10](#) and [13](#).

When large common-mode signals are present, it is important to select a differential amplifier having a large CMRR specification.

Ground Loops

Often, the sensor and the signal-conditioning unit (such as an amplifier or current-to-voltage converter) are located some distance apart and are connected by a cable. Furthermore, the voltage (or current) produced by the sensor may be very small (less than one millivolt or one microampere). Then, several problems can occur that reduce accuracy or, in extreme cases, totally obscure the desired signal.

One of these problems is known as **ground loops**. If we have a single-ended amplifier, one of its input terminals is connected to a ground wire of the electrical distribution system. These ground wires eventually lead to the ground bus in the electrical distribution panel, which in turn is connected to a cold-water pipe or to a conducting rod driven into the earth. In instrumentation systems, we often have several pieces of equipment that are connected to ground through different wires. Ideally, the ground wires would have zero impedance, and all of the ground points would be at the same voltage. In reality, because of currents flowing through small but nonzero resistances of the various ground wires, small but significant voltages exist between various ground points.

Consider [Figure 8.19](#), in which we have a sensor, an amplifier with a single-ended input, and a cable connecting the sensor to the amplifier. The cable wires have small resistances denoted by R_{cable} . Several ground wires are shown with their resistances R_{g1} and R_{g2} . The current source I_g represents current flowing to ground. Typically, I_g originates from the 60-Hz line voltage through power-supply circuits of the instruments. If we connect both the sensor and the amplifier input to ground, part of I_g flows through the connecting cable, and the input voltage is the sensor voltage minus the drop across R_{cable} :

$$V_{\text{in}} = V_{\text{sensor}} - I_{g1}R_{\text{cable}} \quad (8.7)$$

When the sensor voltage is very small, it can be totally obscured by the drop across R_{cable} .

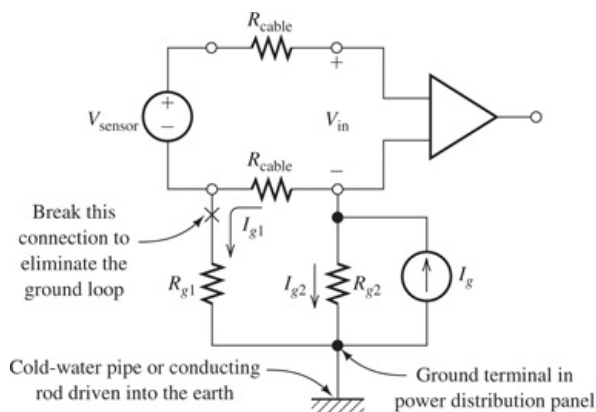


Figure 8.19

Ground loops are created when the system is grounded at several points.

On the other hand, if we break the sensor ground connection so only the amplifier is grounded, I_{g1} becomes zero and the input voltage is the sensor voltage as desired. Thus, in connecting a sensor to an amplifier with a single-ended input, we should select an ungrounded or **floating** sensor.

In connecting a sensor to an amplifier with a single-ended input, we should select a floating sensor.

If you have connected several audiovisual components, such as VCRs, TVs, radio tuners, CD players, stereo amplifiers, and so forth, you have probably encountered ground loops, which cause an annoying 60-Hz hum to be produced by loudspeakers.

Alternative Connections

Figure 8.20 shows four combinations of sensors and amplifiers. As we have seen, we need to avoid the combination of grounded sensor and single-ended input shown in part (a) of the figure because of ground loops. Any of the other three connections can be used. However, for a floating sensor with a differential amplifier as shown in part (d), it is often necessary to include two high-valued (much greater than the internal impedance of the sensor to avoid loading effects) resistors to provide a path for the input bias current of the amplifier. (Input bias current is discussed further in [Section 10.12](#).) If the resistors are not included, the common-mode voltage of the source can become so large that the amplifier does not function properly. In part (c) of the figure, the ground connection to the sensor provides a path for the bias current.

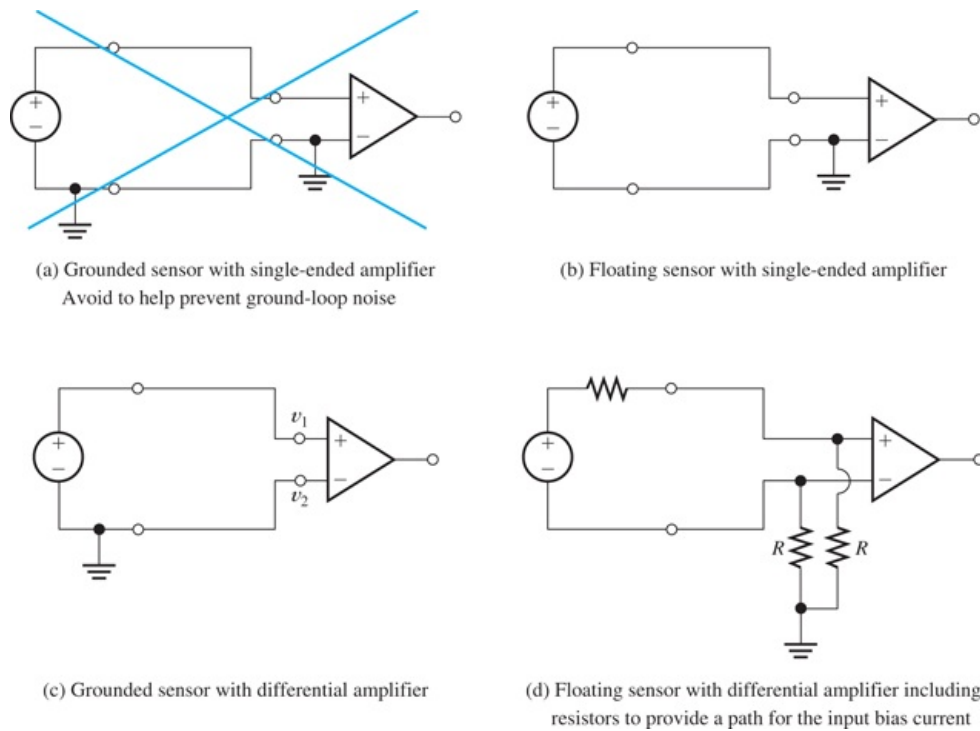


Figure 8.20

Four sensor amplifier combinations.

Noise

Another problem that can occur in connecting sensors to signal-conditioning units is the inadvertent addition of noise produced by the electric or magnetic fields generated by nearby circuits. (Computers are infamous for creating high-frequency electrical noise.) Electric field coupling can be modeled as small capacitances that are connected between nearby circuits and the cables, as shown in [Figure 8.21](#). Currents are injected into the cable through these capacitances. This is particularly a problem with unshielded cables and when the sensor impedance is large. A shielded cable, in which an outer conductor in the form of metallic foil or braided wire encases the signal conductors, can eliminate much of the noise caused by electric fields. The shield is connected to ground, providing a low-resistance path for the capacitive currents. This is illustrated in [Figure 8.22](#).

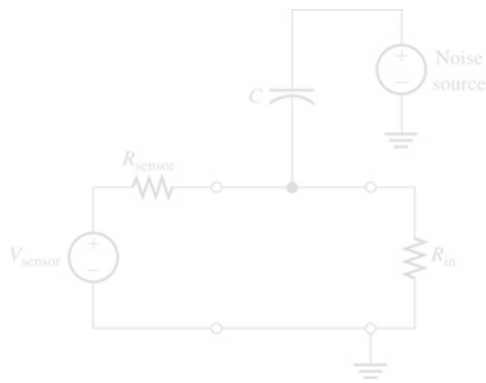


Figure 8.21

Noise can be coupled into the sensor circuit by electric fields. This effect is modeled by small capacitances between the noise source and the sensor cable.

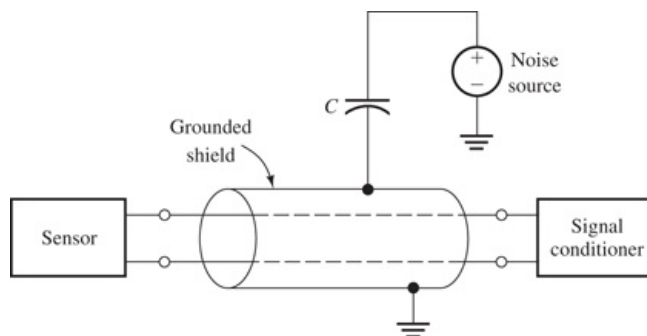


Figure 8.22

Electric field coupling can be greatly reduced by using shielded cables.

Electric field coupling of noise can be reduced by using shielded cables.

Noise problems can also occur due to magnetic coupling. Many circuits, particularly power-supply transformers, produce time-varying magnetic fields. When these fields pass through the region bounded by the cable conductors, voltages are induced in the cable. Magnetically coupled noise can be greatly diminished by reducing the effective area bounded by the conductors. Twisted-pair and coaxial cables (see [Figure 8.23](#)) are two good ways to accomplish this. Because the center lines of the conductors in coaxial cable are coincident, the effective bounded area is very small.

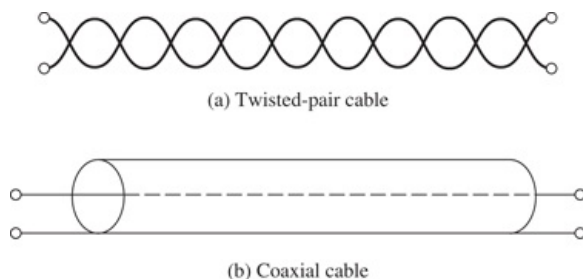


Figure 8.23

Magnetic field coupling can be greatly reduced by using twisted-pair or coaxial cables.

Magnetically coupled noise is reduced by using coaxial or twisted-pair cables.

Exercise 8.12

The voltages produced by a sensor are $v_1 = 5.7 \text{ V}$ and $v_2 = 5.5 \text{ V}$. Determine the differential and common-mode components of the sensor signal.

Answer $v_d = 0.2 \text{ V}$; $v_{cm} = 5.6 \text{ V}$.

PRACTICAL APPLICATION

8.2



The Virtual First-Down Line

In American football, the team on offense must gain 10 yards in a series of four plays to retain possession of the ball. Thus, a line marking the needed advance is of constant interest to fans viewing a game.

On September 27, 1998, in a Sunday Night Football game on ESPN, Sportvision introduced their “1st and Ten” system for electronically drawing the first-down line on television images. The system has been enthusiastically accepted and even won an Emmy award for technical innovation. In the 2003 season, 18 crews covered about 300 NCAA and NFL games.

While the concept of drawing a virtual line on a television image sounds simple, there are some formidable problems that need to be overcome to produce a result that appears to be painted on the field. Typically, three main cameras are situated above and back from the 50-yard line and both of the 25-yard lines. Each camera pans, tilts, zooms, and changes focus rapidly during the game. The virtual line needs to change its position, orientation, and width as the camera view changes. In addition, some football fields are not flat—they are crowned to ensure drainage and the yard lines are not exactly straight. If the virtual line does not closely match the curvature of the lines on the field, it will not look natural. Furthermore, the line needs to be drawn thirty times a second, once for each video frame. Of course, for realism, part of the line needs to disappear when a player, an official, or the ball moves across it. An impressive array of sophisticated electronic and computer technology has been employed by Sportvision engineers to meet these demands.

To set up the system on a given field, Sportvision starts by using laser surveying instruments to measure the elevation at a number of points along each 10-yard line. A computer uses this data to produce a virtual three-dimensional model of the field.

Sensors attached to each of the cameras measure pan, tilt, zoom, and focus. This data is fed into a computer that alters the model to match the perspective for a given camera, and a virtual map is drawn in blue lines over the image of the field seen by the camera. Finally, the virtual map is tweaked to match the real image for many combinations of pan, tilt, and zoom as illustrated in [Figure PA8.2](#). The resulting calibration data is saved for use by the system during an actual game.



Tilt, pan, zoom, and focus sensors on the camera provide information about which part of the field is in view.



During system calibration, computers use the field survey data to draw the virtual map (in gray) of the field on the TV screen.



Then the virtual map is tweaked to match the actual field. This is repeated for many combinations of tilt, pan, zoom, and focus for each of the three main cameras.

FIGURE

PA8.2

Computers use a virtual map based on surveying the football field to draw the 1st and Ten line on the television screen in real time.

A technique known as “chroma keying” has been around for a long time and is widely used in televised weather reports. A meteorologist forecasting weather stands in front of a light blue wall. Computers substitute weather maps and graphics for all the pixels (i.e., picture elements) that are light blue. Thus, the forecaster seems to stand in front of the weather map. The same kind of technology is used to allow officials, players, and the ball to seem to move over the virtual first-down line. However, discerning which pixels are players and which are part of the field is much more difficult than separating a forecaster from a blue wall. Weather forecasters usually avoid wearing clothing that matches the color of the wall, and the wall is all the same color. On the other hand, the field can be many different shades of white (painted yard lines on the field), green (grass or artificial turf), or brown (grass or mud). Part of the field may be sunlit while other parts are in shadow. Football teams, such as the Green Bay Packers, have uniforms that are partly green, which is especially difficult to distinguish from sunlit artificial turf. Other colors, such as brown, can also be difficult.

By constant recalibrating, the 1st and Ten system can keep track of which colors are part of the field and which are not, so the virtual down line is not drawn over players.

During a game, a team of four people operates the system, which contains five computers. The “spotter” is in the stadium and radios the location of the down line to a truck containing the equipment and two of the other team members. The “line position technician” enters the location data into the computers, monitors line position, and makes any needed adjustments. Another operator monitors changes in field colors so the chroma keying is properly accomplished. Finally, a troubleshooter looks for problems and solves them.

One of the computers receives and processes the pan, tilt, and zoom data from the cameras. Another keeps track of which camera is “on air.” A third displays the on-air video and superimposes the virtual map of the field including the current down line. Another computer discerns which parts of the image are field and which are players or officials. Finally, the fifth computer places the virtual down line on the broadcast image while avoiding any superimposed graphics that the network may place on the screen.

More information about the 1st and Ten system as well as similar technology for other sports can be found at www.sportvision.com.

8.9 Analog-to-Digital Conversion

As discussed starting on page 331, analog signals are converted to digital form by a two-step process. First, the analog signal is sampled (i.e., measured) at periodic points in time. A code word is then assigned to represent the approximate value of each sample. The sampling rate and the number of bits used to represent each sample are two very important considerations in the selection of a DAQ system.

Sampling Rate

The rate at which a signal must be sampled depends on the frequencies of the signal's components. (All signals can be considered to be sums of sinusoidal components that have various frequencies, amplitudes, and phases.) If a signal contains no components with frequencies higher than f_H , all of the information contained in the signal is present in its samples, provided that the sampling rate is selected to be more than twice f_H .

If a signal contains no components with frequencies higher than f_H , all of the information contained in the signal is present in its samples, provided that the sampling rate is selected to be more than twice f_H .

Aliasing

Sometimes, we may only be interested in the components with frequencies up to f_H , but the signal may contain noise or other components with frequencies higher than f_H . Then, if the sampling rate is too low, a phenomenon called **aliasing** can occur. In aliasing, the samples of a high-frequency component appear to be those of a lower frequency component and may obscure the components of interest. For example, [Figure 8.24](#) shows a 7-kHz sinusoid sampled at 10 kHz. As illustrated by the dashed line, the sample values appear to be those of a 3-kHz sinusoid. Because the sampling rate (10 kHz) is less than twice the signal frequency (7 kHz), the samples appear to be those of an alias frequency (3 kHz). (Notice that from the samples it is impossible to determine whether a 3-kHz or a 7-kHz signal was sampled.)

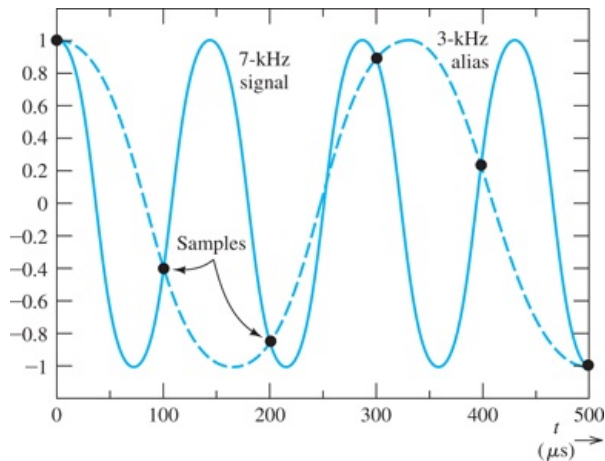


Figure 8.24

When a 7-kHz sinusoid is sampled at 10 kHz, the sample values appear to be those of a 3-kHz sinusoid.

[Figure 8.25](#) shows the alias frequency as a function of the signal frequency f . When the signal frequency f exceeds one-half of the sampling frequency f_s , the apparent frequency of the samples is different from the true signal frequency.

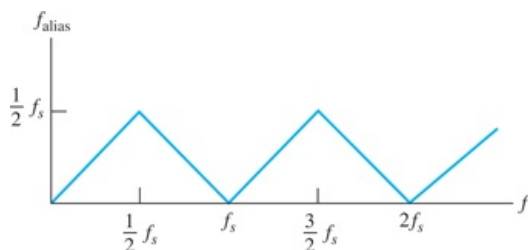


Figure 8.25

Alias or apparent frequency versus true signal frequency.

One way to avoid aliasing is to pick the sampling frequency high enough so that the alias frequencies are higher than the frequencies of interest. Then, the computer can remove the unwanted components by processing the samples with digital-filtering software. However, when high-frequency noise is present, it can result in a sampling rate that exceeds the ability of the computer to process the resulting data. Then, it is better to use an analog **antialias filter** ahead of the ADC to remove the noise above the highest frequency of interest. Typically, this is a high-order Butterworth filter implemented with operational amplifiers, such as those discussed in [Section 13.10](#) starting on page 681. Because real filters are unable to sufficiently reject components slightly above their cutoff frequencies, it is usually necessary to select the sampling frequency at least three times the highest frequency of interest. For example, the highest audible frequency is about 15 kHz, but in CD technology a sampling rate of 44.1 kHz is used.

Quantization Noise

A second consideration important in converting analog signals to digital form is the number of amplitude zones to be used. Exact signal amplitudes cannot be represented, since all amplitudes falling into a given zone have the same codeword. Thus, when a DAC converts the codewords to form the original analog waveform, it is possible to reconstruct only an approximation to the original signal the reconstructed voltage is in the middle of each zone, which was illustrated in [Figure 6.46](#) on page 333. Hence, some **quantization error** exists between the original signal and the reconstruction. This error can be reduced by using a larger number of zones, which requires a longer codeword for each sample. The number N of amplitude zones is related to the number of bits k in a codeword by

$$N = 2^k \quad (8.8)$$

Analog-to-digital conversion is a two-step process. First, the signal is sampled at uniformly spaced points in time. Second, the sample values are quantized so they can be represented by words of finite length.

Therefore, if we are using an 8-bit ($k = 8$) ADC, we find that there are $N = 2^8 = 256$ amplitude zones. The resolution of a computer-based measurement system is limited by the word length of the ADC. In compact-disc technology, 16-bit words are used to represent sample values. With that number of bits, it is very difficult for a listener to detect the effects of quantization error on the reconstructed audio signal. In the telephone system, 8-bit words are used and the fidelity of the reconstructed signal is relatively poor.

The effect of finite word length can be modeled as adding quantization noise to the reconstructed signal. It can be shown that the rms value of the quantization noise is approximately

$$N_{\text{rms}} = \frac{\Delta}{2\sqrt{3}} \quad (8.9)$$

where Δ is the width of a quantization zone.

The effect of finite word length can be modeled as adding quantization noise to the reconstructed signal.

In the example that follows, we illustrate how the various factors we have discussed are used in selecting the components of a computer-based measurement system.

Example 8.6 Specifications for a Computer-Based Measurement System

Suppose that we have a single-ended (i.e., one terminal of the sensor is connected to power-system ground) piezoelectric vibration sensor that produces a signal of interest having peak values of $\pm 25 \text{ mV}$, an rms value of 3 mV , and components with frequencies up to 5 kHz . The internal impedance of the sensor is $1 \text{ k}\Omega$. We want the system resolution to be $2 \text{ }\mu\text{V}$ or better (i.e., smaller) and the accuracy to be ± 0.2 percent of the peak signal or better. (Note that the desired resolution is considerably better than the accuracy. This allows the system to discern changes in the signal that are smaller than the error.) The probe wiring is likely to be exposed to electric and magnetic field noise having components at frequencies higher than 5 kHz . An ADC having an input range from -5 V to $+5 \text{ V}$ is to be used. Draw the block diagram of the measurement system and give key specifications for each block.

Solution

Because one end of the sensor is grounded, we need to use an instrumentation amplifier with a differential input to avoid ground-loop problems. (See [Figure 8.20](#) on pg 443.)

To help reduce capacitively and inductively coupled noise, we should select a shielded twisted-pair or coaxial cable to connect the sensor to the system. To avoid ground loops, the shield should be grounded at the sensor end only. Furthermore, we should use an antialias filter to reduce noise above 5 kHz . The block diagram of the system is shown in [Figure 8.26](#).

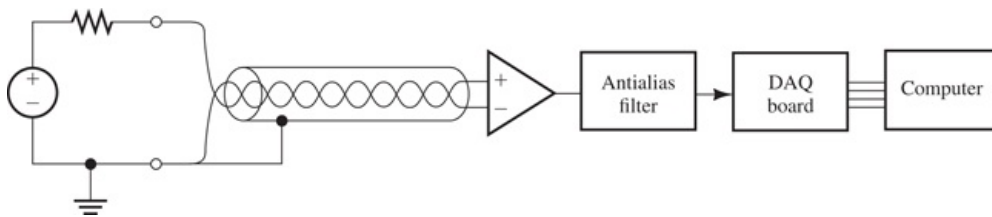


Figure 8.26

See [Example 8.6](#).

The voltage gain of the instrumentation amplifier/antialias filter combination should be $(5 \text{ V}) / (25 \text{ mV}) = 200$; so the sensor signal is amplified to match the range of the ADC. The input impedance of the amplifier should be very large compared with the internal sensor impedance; therefore, loading effects are insignificant. If we specify a minimum input impedance of $1 \text{ M}\Omega$, loading effects will reduce the signal by 0.1 percent, which is within the desired accuracy. (We need to allow for errors from other unknown sources.)

To achieve a resolution of $2 \text{ }\mu\text{V}$ for a $\pm 25\text{-mV}$ signal, we need an ADC with at least $(50 \text{ mV}) / (2 \text{ }\mu\text{V}) = 25,000$ amplitude levels. This implies an ADC word length of at least $k = \log_2(25,000) = 14.6$. Since word length must be an integer, $k = 15$ is the smallest word length that will meet the desired specifications. It turns out that DAQ boards with 16-bit ADCs are readily available, and to provide some design margin, that is the word length we should specify.

Because the highest frequency of interest is 5 kHz , we need a sampling frequency of at least 10 kHz . However, the antialias filter will not effectively remove the components that are only slightly above 5 kHz , so a greater sampling rate (20 kHz or greater) should be chosen. ■

Exercise 8.13

A certain 8-bit ADC accepts signals ranging from -5 V to $+5 \text{ V}$. Determine the width of each quantization zone and the approximate rms value of the quantization noise.





Answer $\Delta = 39.1 \text{ mV}$; $N_{\text{rms}} = 11.3 \text{ mV}$.


Exercise 8.14

A 25-kHz sinewave is sampled at 30 kHz. Determine the value of the alias frequency.

Answer $f_{\text{alias}} = 5 \text{ kHz}$.

Summary

1. A computer is composed of a central processing unit (CPU), memory, and input output (I/O) devices. These are connected together with bidirectional data and control buses. The CPU contains the control unit, the ALU, and various registers.
2. Memory is used to store programs and data. Three types of memory are RAM, ROM, and mass storage.
3. In von Neumann computer architecture, data and instructions are stored in the same memory. In Harvard architecture, separate memories are used for data and instructions.
4. Sensors are input devices that convert physical values to electrical signals. Actuators are output devices that allow the MCU to affect the system being controlled.
5. **Figure 8.5**  (on pg 415) shows the elements of a typical microcomputer used for process control.
6. Analog-to-digital converters (A/D) transform analog voltages into digital words. Digital-to-analog converters (D/A) transform digital words into analog voltages. Converters are needed to interface analog sensors and actuators with an MCU.
7. **Figure 8.6**  (on pg 418) shows the register set for the CPU12.
8. In a stack memory, data is added to or read from the top of the stack. It is a last-in first-out (LIFO) memory. The stack pointer is a register that contains the address of the top of the stack.
9. **Table 8.1**  (on pg 422) contains some of the instructions for the CPU12.
10. Six addressing modes are supported by the CPU12: extended addressing, direct addressing, inherent addressing, immediate addressing, indexed addressing, and program-relative addressing.
11. In writing programs for embedded microcontrollers, we often start by writing a source program using labels and mnemonics. An assembler converts the source program into an object program consisting of machine code that is loaded into the target system.
12. High costs are incurred in software development for MCUs. However, when the cost can be spread over many units, assembly language programming can be the best solution.
13. The block diagram of a typical computer-based instrumentation system is shown in **Figure 8.12**  on page 435.
14. When we need to sense the internal (open-circuit) voltage of a sensor, we should specify an amplifier having an input impedance that is much larger in magnitude than the Thévenin impedance of the sensor.
15. When we want to sense the current produced by a sensor, we need a current-to-voltage converter having a very small (ideally zero) input impedance magnitude compared to the Thévenin impedance of the sensor.
16. Bias errors are the same each time a measurement is repeated under the same apparent conditions. Bias errors include offset, scale error, nonlinearity, and hysteresis.
17. Random errors are different for each measurement and have zero average value.
18. The accuracy of an instrument is the maximum expected difference in magnitude between measured and true values (often expressed as a percentage of the full-scale value).
19. Precision is the ability of the instrument to repeat the measurement of a constant measurand. More precise measurements have less random error.
20. The resolution of an instrument is the smallest increment discernible between measured values.
21. Some of the functions of signal conditioners are amplification, conversion of current signals to voltage signals, supply of excitation to the sensor, and filtering to eliminate noise or other unwanted signal components.
22. One of the input terminals of a single-ended amplifier is grounded. Neither input terminal of a differential amplifier is grounded. The output of an ideal differential amplifier is its differential gain times the difference between the input voltages.

23. If the input voltages to a differential amplifier are v_1 and v_2 , the differential input signal is $v_d = v_1 - v_2$ and the common-mode signal is $v_{cm} = \frac{1}{2}(v_1 + v_2)$. Usually in instrumentation systems, the differential signal is of interest and the common-mode signal is unwanted noise.
24. When large unwanted common-mode signals are present, it is important to select a differential amplifier having a large CMRR (common-mode rejection ratio) specification.
25. Ground loops are created in an instrumentation system when several points are connected to ground. Currents flowing through the ground conductors can produce noise that makes the measurements less accurate and less precise.
26. To avoid ground-loop noise when we must connect a sensor to an amplifier with a single-ended input, we should select a floating sensor (i.e., neither terminal of the sensor should be grounded).
27. Shielded cables reduce the noise coupled by electric fields.
28. Coaxial or twisted-pair cables reduce magnetically coupled noise.
29. If a signal contains no components with frequencies higher than f_H , all of the information contained in the signal is present in its samples, provided that the sampling rate is selected to be more than twice f_H .
30. Analog-to-digital conversion is a two-step process. First, the signal is sampled at uniformly spaced points in time. Second, the sample values are quantized so they can be represented by words of finite length.
31. We model the effect of finite word length as the addition of quantization noise to the signal.
32. If a sinusoidal signal component is sampled at a rate f_s that is less than twice the signal frequency f , the samples appear to be from a component with a different frequency known as the alias frequency f_{alias} . Alias frequency is plotted versus the true signal frequency in [Figure 8.25](#)  on pg [448](#).
33. If a sensor signal contains high-frequency components that are not of interest, we often use an analog antialias filter to remove them so a lower sampling frequency can be used without aliasing.

Problems

Section 8.1: Computer Organization

P8.1.List the functional parts of a computer.

P8.2.What are tristate buffers? What are they used for?


P8.3.Give several examples of I/O devices.

P8.4.What is memory-mapped I/O?

P8.5.What is a bus? What is the function of the data bus? Of the address bus?

P8.6.What is an embedded computer?

***P8.7.**The address bus of a computer is 16 bits wide and the data bus is 32 bits wide. How many bytes does the memory potentially contain?

* Denotes that answers are contained in the Student Solutions files. See [Appendix E](#)  for more information about accessing the Student Solutions.

P8.8.Define the terms *microprocessor*, *microcomputer*, and *microcontroller*.

P8.9.Explain the difference between Harvard computer architecture and von Neumann computer architecture.

Section 8.2: Memory Types

P8.10.What is RAM? List two types. Is it useful for storing programs in embedded computers? Explain.

***P8.11.**What is ROM? List four types. Is it useful for storing programs in embedded computers? Explain.

P8.12.List three examples of mass-storage devices.

P8.13.Which type of memory is least expensive per unit of storage? (Assume that many megabytes of capacity are required.)

P8.14.How many memory locations can be addressed if the address bus has a width of 32 bits?

***P8.15.**Which type of memory would be best in the controller for an ignition system for automobiles?

P8.16.When might we choose EEPROM rather than mask-programmed ROM?

P8.17.What types of memory are volatile? Non-volatile?

Section 8.3: Digital Process Control

P8.18.List the elements that are potentially found in a microcomputer-based control application.

P8.19.What is a sensor? Give three examples.

P8.20.What is an actuator? Give three examples.

***P8.21.**Explain the difference between a digital sensor and an analog sensor. Give an example of each.

P8.22.List five common household products that potentially include an MCU.

P8.23.List two potential applications of MCU-based control or instrumentation in your field of specialization.

P8.24.What is an A/D? Why might one be needed in an MCU-based controller?

***P8.25.**What is a D/A? Why might one be needed in an MCU-based controller?

P8.26.What is polling? What is an interrupt? What is the main potential advantage of interrupts versus polling?

Section 8.4: Programming Model for the HCS12/9S12 Family

P8.27.What is the function of the A, B, and D registers of the CPU12?

P8.28.What is the function of the program counter register? Of the MCU?

***P8.29.**What is a stack? What is the stack pointer used for?

P8.30.What is a LIFO memory?

***P8.31.**Suppose that initially the contents of the registers are

A:07 B:A9 SP:004F X:34BF

and that memory locations 0048 through 004F initially contain all zeros. The commands PSHA, PSHB, PULA, PULB, PSHX are then executed in sequence. List the contents of the registers A, B, SP, and X, and the memory locations 0048 through 004F after each command is executed.

P8.32.Suppose that initially the contents of the registers are

A:A7 B:69 SP:004E Y:B804

and that memory locations 0048 through 004F initially contain all zeros. The commands PSHY, PSHB, PULY, PSHA are then executed in sequence. List the contents of the registers A, B, SP, and X and the memory locations 0048 through 004F after each command is executed.

***P8.33.**Write a sequence of push and pull commands to swap the high byte and low byte of the X register. After the sequence of commands is executed, the contents of the other registers should be the same as before.

Section 8.5: The Instruction Set and Addressing Modes for the CPU12

P8.34.For each part of this problem, assume that the X register contains 2000 and the A register initially contains 01. Name the type of addressing and give the content of A after each instruction listed next. The contents of memory are shown in [Figure P8.34](#).

- *LDDA \$2002**
- LDDA #\$43**
- *LDDA \$04**
- LDDA 6,X**
- *INCA**
- CLRA**
- *LDAA \$2007**
- INX**

0000	01	2000	37
0001	FA	2001	AF
0002	9B	2002	20
0003	61	2003	07
0004	9A	2004	20
0005	B6	2005	00
0006	73	2006	FF
0007	41	2007	F3

(a) (b)

Figure P8.34

P8.35. Suppose that the contents of certain memory locations are as shown in [Figure P8.34](#).

Furthermore, for each part of this problem, the initial contents of the CPU registers are:

(D) = \$0003, (X) = \$1 FFF, and (Y) = \$1000. Name the type of addressing and determine the contents of registers D, X, and Y after each of these instructions is executed:

- ADDB \$1002,Y
- LDAA B,X
- LDAB 7, +X
- LDX [\$1004,Y]
- LDAA [D,X]

P8.36.

- *Assume that the A register initially contains FF and that the program counter is 2000. What is the address of the instruction executed immediately after the branch command? The content of memory and the corresponding instruction mnemonics are shown in [Figure P8.36\(a\)](#).
- Repeat for [Figure P8.36\(b\)](#).
- Repeat for [Figure P8.36\(c\)](#).

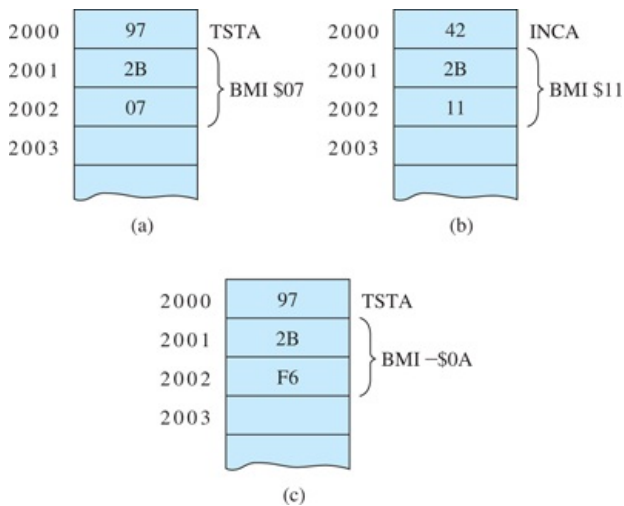


Figure P8.36

P8.37. Give the machine code for each of the following instructions:

- *CLRA
- *ADDA \$4A
- ADDA \$02FF
- BNE -\$06
- ADDA #\$0D

How many memory locations are occupied by each instruction?

P8.38. Find the content of the A register after each instruction as the following sequence of instructions is executed:

```
LDAA  #$01
ADDA  #$F1
CLRA
```

Is the N bit of the condition-code register set or clear? Is the Z bit of the condition-code register set or clear?

***P8.39.**

- a. Suppose that the content of A is 43. Find the content of A after the instruction ADDA #\$05 is executed.
- b. Suppose that the content of A is FA. Find the content of A after the instruction ADDA #\$0F is executed. (In this case, overflow occurs.)

***P8.40.** Assume that the content of A is \$A7 and that the content of B is \$20. Find the contents of A and B after the MUL instruction is executed. [Hint: The MUL instruction assumes that the contents of A and B are unsigned integer values.]

Section 8.6: Assembly-Language Programming

***P8.41.** Write an assembly-language program starting in location 200 that multiplies the content of the A register by 11_{10} and stores the result in memory locations \$FF00 and \$FF01, with the most significant byte in location \$FF00. The processor should then be stopped.

P8.42. Write an assembly-language program starting in location 0400 that stores 00 in location 0800, 01 in 0801, 02 in 0802, 03 in 0803, and then halts the processor.

***P8.43.** Write a subroutine called DIV3 that divides the content of A by three. Assume that the initial content of A is a positive integer in two's-complement form. On return from the subroutine, the quotient should reside in B and the remainder in A.

P8.44. Consider the following assembly-language code for the CPU12:

```
; PROBLEM 8.44
;
; ORG $0600
START LDAA #$07
      LDAB #$AF
      STD $0609
      STOP
      END
```

Describe the effect of each line of this code and list the contents of memory locations 0600 through 060A after the code has been assembled and executed.

P8.45. Write a subroutine called MUL3 that rounds the content of A to its nearest integer multiple of 3. Assume that the initial content of A is a positive integer in two's-complement form. Memory location \$0A can be used for temporary storage. Include comments in your source code to explain the program and its operation to human readers. [Hint: Repeatedly subtract 3 until the result becomes negative. If the result is -3 the original content of A was a multiple of 3 and should not be changed. If the result is -2 the original content of A was one plus an integer multiple of 3, and we should subtract one from the original number to obtain the nearest multiple of 3. If the result is -1 the original content of A is 2 plus an integer multiple of 3, and we should add 1 to the original number to obtain the nearest multiple of 3.]

P8.46. Suppose that register B contains a two-decimal-digit BCD number n . Write a subroutine called CONVERT that replaces the content of register B by its binary equivalent. The content of the other registers (except the program counter) should be unchanged at the completion of the subroutine. Memory locations \$1A, \$1B, and \$1C can be used for temporary storage. [Hint: We need to separate the upper nibble (four bits) of n from its lower nibble. This can be achieved by shifting n four bits to the left, with the result appearing in the D register. Shifting by four bits to the left is accomplished by multiplying by 2^4 .]

Section 8.7: Measurement Concepts and Sensors

P8.47. Name the elements of a computer-based instrumentation system.

***P8.48.** Draw the equivalent circuit of a sensor in which the open-circuit sensor voltage is proportional to the measurand. What are loading effects? How do we avoid them when we need to measure the Thévenin (i.e., open-circuit) sensor voltage?

P8.49. What signal-conditioner input impedance is best if the sensor produces short-circuit (Norton) current that is proportional to the measurand?

***P8.50.** A load cell produces an open-circuit voltage of $200 \mu\text{V}$ for a full-scale applied force of 10^4 N , and the Thévenin resistance is $1 \text{ k}\Omega$. The sensor terminals are connected to the input terminals of an amplifier. What is the minimum input resistance of the amplifier so the overall system sensitivity is reduced by less than 1 percent by loading?

***P8.51.** A certain liquid-level sensor has a Thévenin (or Norton) resistance that varies randomly from $10 \text{ k}\Omega$ to $1 \text{ M}\Omega$. The short-circuit current of the sensor is proportional to the measurand. What type of signal-conditioning unit is required? Suppose we allow for up to 1 percent change in overall sensitivity due to changes in the sensor resistance. Determine the specification for the input resistance of the signal conditioner.

P8.52. How are bias errors different from random errors?

P8.53. List four types of bias error.

***P8.54.** An instrumentation system measures distances ranging from 0 to 1 m full scale. The system accuracy is specified as ± 0.5 percent of full scale. If the measured value is 70 cm, what is the potential range of the true value?

P8.55. Explain how precision, accuracy, and resolution of an instrument are different.

***P8.56.** Three instruments each make 10 repeated measurements of a flow rate known to be $1.500 \text{ m}^3/\text{s}$ with the results given in [Table P8.56](#).

- Which instrument is most precise? Least precise? Explain.
- Which instrument has the best accuracy? Worst accuracy? Explain.
- Which instrument has the best resolution? Worst resolution? Explain.

Table P8.56

Trial	Instrument A	Instrument B	Instrument C
1	1.5	1.73	1.552
2	1.3	1.73	1.531
3	1.4	1.73	1.497
4	1.6	1.73	1.491
5	1.3	1.73	1.500
6	1.7	1.73	1.550
7	1.5	1.73	1.456
8	1.7	1.73	1.469
9	1.6	1.73	1.503
10	1.5	1.73	1.493

Section 8.8: Signal Conditioning

P8.57. List four or more functions of signal conditioners.

P8.58. How is a single-ended amplifier different from a differential amplifier?

P8.59. Suppose that the input voltages to an ideal differential amplifier are equal. Determine the output voltage.

***P8.60.** The input voltages to a differential amplifier are

$$v_1(t) = 0.002 + 5\cos(\omega t)$$

and

$$v_2(t) = -0.002 + 5\cos(\omega t)$$

Determine the differential input voltage and the common-mode input voltage. Assuming that the differential amplifier is ideal with a differential gain $A_d = 1000$, determine the output voltage of the amplifier.

P8.61. A sensor produces a differential signal of 6 mV dc and a 2-V-rms 60-Hz ac common-mode signal. Write expressions for the voltages between the sensor output terminals and ground.

***P8.62.** Suppose we have a sensor that has one terminal grounded. The sensor is to be connected to a DAQ board in a computer 5 meters away. What type of amplifier should we select? To mitigate noise from electric and magnetic fields, what type of cable should we use? Draw the schematic diagram of the sensor, cable, and amplifier.

P8.63. What is a floating sensor? When would we want to use a floating sensor?

***P8.64.** Suppose that the data collected from a sensor is found to contain an objectionable 60-Hz ac component. What potential causes for this interference would you look for? What are potential solutions for each cause of the interference?

Section 8.9: Analog-to-Digital Conversion

P8.65. In principle, analog-to-digital conversion involves two operations. What are they?

P8.66. What is aliasing? Under what conditions does it occur?

P8.67. What causes quantization noise?

***P8.68.** We need to use the signal from a piezoelectric vibration sensor in a computer-aided instrumentation system. The signal is known to contain components with frequencies up to 30 kHz. What is the minimum sampling rate that should be specified? Suppose that we want the resolution of the sampled values to be 0.1 percent (or better) of the full range of the ADC. What is the fewest number of bits that should be specified for the ADC?

***P8.69.** A 2-V-peak sinewave signal is converted to digital form by a 12-bit ADC that has been designed to accept signals ranging from -5 V to +5 V. (In other words, codewords are assigned for equal increments of amplitude for amplitudes between -5 V and +5 V.)

- Determine the width Δ of each quantization zone.
- Determine the rms value of the quantization noise and the power that the quantization noise would deliver to a resistance R .
- Determine the power that the 2-V sinewave signal would deliver to a resistance R .
- Divide the signal power found in part (c) by the noise power determined in part (b). This ratio is called the signal-to-noise ratio (SNR). Express the SNR in decibels, using the formula $\text{SNR}_{\text{dB}} = 10 \log(P_{\text{signal}}/P_{\text{noise}})$.

***P8.70.** We need an ADC that can accept input voltages ranging from 0 to 5 V and have a resolution of 0.02 V. How many bits must the codewords have?

***P8.71.** A 10-kHz sinewave is sampled. Determine the apparent frequency of the samples. Has aliasing occurred? The sampling frequency is

- 11 kHz;
- 8 kHz;
- 40 kHz.

P8.72. A 60-Hz sinewave $x(t) = A \cos(120\pi t + \phi)$ is sampled at a rate of 360 Hz. Thus, the sample values are $x(n) = A \cos(120\pi n T_s + \phi)$, in which n assumes integer values and $T_s = 1/360$ is the time interval between samples. A new signal is computed by the equation

$$y(n) = \frac{1}{2} [x(n) + x(n-3)]$$

- Show that $y(n) = 0$ for all n .
- Now suppose that $x(t) = V_{\text{signal}} + A \cos(120\pi t + \phi)$, in which V_{signal} is constant with time and again find an expression for $y(n)$.
- When we use the samples of an input $x(n)$ to compute the samples for a new signal $y(n)$, we have a **digital filter**. Describe a situation in which the filter of parts (a) and (b) could be useful.

Practice Test

Here is a practice test you can use to check your comprehension of the most important concepts in this chapter. Answers can be found in [Appendix D](#) and complete solutions are included in the Student Solutions files. See [Appendix E](#) for more information about the Student Solutions.

T8.1. First, think of one or more correct ways to complete each statement in [Table T8.1\(a\)](#). Then, select the best choice from the list given in [Table T8.1\(b\)](#). [Items in [Table T8.1\(b\)](#) may be used more than once or not at all.]

Table T8.1

(a)
<ul style="list-style-type: none">a. Tristate buffers . . .b. When I/O devices are accessed by the same address and data buses as data memory locations, we have . . .c. In an microcontroller, programs are usually stored in . . .d. The type of memory most likely to be volatile is . . .e. The type of memory most likely to be used for temporary data in an microcontroller is . . .f. Arithmetic operations are carried out in the . . .g. An microcontroller may be designed to respond to external events by . . .h. The registers of a CPU12 that hold one of the arguments and the results of arithmetic and logical operations are . . .i. The registers of a CPU12 that are used mainly for indexed addressing are . . .j. The TSTA instruction may change the content of . . .k. Stacks are . . .l. The type of addressing used by the ABA instruction is . . .m. The type of addressing used by the ADDA \$0AF2 instruction is . . .n. The type of addressing used by the ADDA #\$0A instruction is . . .o. The register that holds the address of the next instruction to be retrieved from memory is . . .p. The type of addressing used by the BEQ instruction is . . .
(b)
<ul style="list-style-type: none">1. are composed of switches that are always closed2. mass storage3. B and Y4. I/O devices have their own addresses and data buses5. are composed of open switches6. A and X7. control unit8. extended9. A, B, and D10. contain switches that open and close under the direction of the program counter11. facilitate the ability to transfer data in either direction over a bus12. the condition code register

13. ALU
14. polling
15. LIFO memories
16. inherent
17. memory-mapped I/O
18. A and D
19. interrupts
20. X and Y
21. ROM
22. direct
23. the program counter
24. dynamic RAM
25. the stack pointer
26. either interrupts or polling
27. static RAM
28. indexed
29. immediate
30. relative

T8.2. We have a CPU12 MCU. For each part of this problem, assume that the initial content of A is 00, the initial content of B is FF, the initial content of Y is 2004, and the initial content of selected memory locations is as shown in [Figure P8.34](#) on page 453. Name the type of addressing used and give the content of A (in hexadecimal form) after execution of the instruction:

- a. LDAA \$03;
- b. LDAA \$03,Y;
- c. COMA;
- d. INCA;
- e. LDAA #\$05;
- f. ADDD #A001.

T8.3. Suppose that initially the contents of the registers of a CPU12 microcontroller are

A:A6 B:32 SP:1039 X:1958

and that memory locations 1034 through 103C initially contain all zeros. List the contents of the registers A, B, SP, and X, and the contents of the memory locations 1034 through 103C after the sequence of commands, PSHX, PSHB, PULA, PSHX, has been executed.

T8.4. Name the four elements of a computer-based instrumentation system.

T8.5. Name four types of systematic errors in measurement systems.

T8.6. How are bias errors different from random errors?

T8.7. What causes ground loops in an instrumentation system? What are the effects of a ground loop?

T8.8. If a sensor must have one end connected to ground, what type of amplifier should we choose? Why?

T8.9. What types of cable are best for connecting a sensor to an instrumentation amplifier to avoid coupling of noise by electrical and magnetic fields?

T8.10. If we need to sense the open-circuit voltage of a sensor, what specification is important for the instrumentation amplifier?

T8.11. How do we choose the sampling rate for an ADC? Why?

