

1

Introduction to the 68HC12 Microcontroller

1.1 Objectives

After completing this chapter you should be able to:

- Define or explain the following terms: computer, processor, microprocessor, microcontroller, embedded system, hardware, software, cross assembler, cross compiler, RAM, DRAM, SRAM, ROM, PROM, EPROM, EEPROM, flash memory, byte, word, nibble, bus, KB, MB, mnemonic, opcode, and operand
- Explain the differences between the inherent, immediate, direct, extended, relative, indexed, and indirect indexed addressing modes
- Write a sequence of arithmetic and data transfer instructions to perform simple operations

1.2 Basic Computer Concepts

A computer is made up of hardware and software. The hardware consists of the following major components:

- *The processor.* The processor is the brain of a computer system. All computation operations are performed in the processor. A computer system may have one or multiple processors. A computer system that consists of multiple processors is called *multiprocessor* computer system.
- *The memory.* The memory is the place where software programs and data are stored. A memory system can be made of semiconductor chips, magnetic material, and/or optical discs.
- *Input device.* Input devices allow the computer user to enter data and programs into the computer so that computation can be performed. Switches, keypads, keyboards, mice, microphones, and thumb wheels are examples of input devices.
- *Output device.* The results of computation are displayed via output devices so that the user can read them and equipment can be controlled. Examples of output devices include CRT displays, LEDs, seven-segment displays, liquid-crystal-displays (LCD), printers, plotters, and so on.

As shown in Figure 1.1, the processor communicates with other components through a *common bus*. A **bus** is simply a group of conducting wires that allow signals to travel from one point to another. The common bus actually consists of three buses: address bus, data bus, and control bus. Each bus is a group of signals of the same nature.

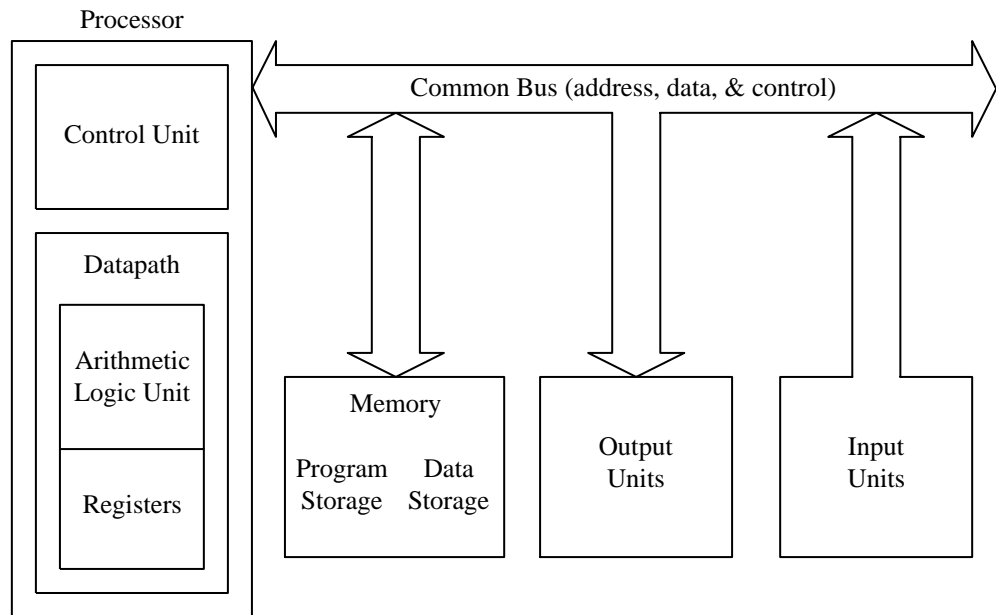


Figure 1.1 ■ Computer organization

1.2.1 The Processor

The processor, which is also called the central processing unit (CPU), can be further divided into two major parts:

Datapath. The datapath consists of a *register file* and an *arithmetic logic unit* (ALU). The register file consists of one or more registers. A register is a storage location in the CPU. It is used to hold data or a memory address during the execution of an instruction. Because the register file is small and close to the ALU, accessing data in registers is much faster than accessing data in memory outside the CPU. The register file makes program execution more efficient. The number of registers varies from computer to computer. All of the arithmetic computations and logic evaluations are performed in the ALU. The ALU receives data from main memory and/or the register file, performs a computation, and, if necessary, writes the result back to main memory or registers.

Control unit. The control unit contains the hardware instruction logic. It decodes and monitors the execution of instructions and also acts as an arbiter as various portions of the computer system compete for resources of the CPU. The system clock synchronizes the activities of the CPU. All CPU activities are measured by clock cycles. At the time of this writing, the highest clock rate of a PC has reached more than 2.0 GHz, where

1 GHz = 1 billion ticks (or cycles) per second

The control unit maintains a register called the *program counter* (PC), which controls the memory address of the next instruction to be executed. During the execution of an instruction, the presence of overflow, an addition carry, a subtraction borrow, and so forth, is flagged by the system and stored in another register called a *status register*. The resultant flags are then used by programmers for program control and decision-making.

WHAT IS A MICROPROCESSOR?

The processor in a very large computer is built from a number of integrated circuits. A *microprocessor* is a processor fabricated on a single integrated circuit. A *microcomputer* is a computer that uses a microprocessor as its CPU. Early microcomputers were quite simple and slow. However, many of today's desktop or notebook computers have become very sophisticated and even faster than many larger computers were only a few years ago.

One way to classify microprocessors is to use the number of bits (referred to as **word length**) that a microprocessor can manipulate in one operation. A microprocessor is 8-bit if it can only work on 8 bits of data in one operation. Microprocessors in use today include 4-bit, 8-bit, 16-bit, 32-bit, and 64-bit. The first 4-bit microprocessor is the Intel 4004, which was introduced in 1971. Since then many companies have joined in the design and manufacturing of microprocessors. Today 8-bit microprocessors are the most widely used among all microprocessors, whereas 64-bit microprocessors emerged only a few years ago and are mainly used in high performance workstations and servers.

Many 32-bit and 64-bit microprocessors also incorporate on-chip memory to enhance their performance. Microprocessors must interface to input/output devices in many applications. However, the characteristics and speed of the microprocessor and input/output devices are quite different. Peripheral chips are required to interface input/output devices with the microprocessor. For example, the integrated circuit i8255 is designed to interface a parallel device such as a printer or seven-segment display with the Intel 8-bit microprocessor 8085.

Microprocessors have been widely used since their invention. It is not an exaggeration to say that the invention of microprocessors has revolutionized the electronics industry. However, the following limitations of microprocessors have led to the development of microcontrollers:

- A microprocessor requires external memory to execute programs.
- A microprocessor cannot directly interface with I/O devices. Peripheral chips are needed.
- Glue logic (such as address decoders and buffers) is needed to interconnect external memory and peripheral interface chips with the microprocessor.

Because of these limitations, a microprocessor-based design cannot be made as small as might be desired. The development of microcontrollers not only eliminated most of these problems but also simplified the hardware design of microprocessor-based products.

WHAT IS A MICROCONTROLLER?

A microcontroller is a computer implemented on a single very large-scale integration (VLSI) chip. A microcontroller contains everything a microprocessor contains plus one or more of the following components:

- *Memory.* Memory is used to store data or programs. A microcontroller may incorporate certain amounts of SRAM, ROM, EEPROM, EPROM, or flash memory.
- *Timer.* Timer function is among the most useful and complicated functions in most microcontrollers. The timer function of most 8-bit and 16-bit microcontrollers consists of input capture, output compare, counter, pulse accumulator, and pulse width modulation (PWM) modules. They can be used to measure frequency, period, pulse width, and duty cycle. They can also be used to create time delay and generate waveforms.
- *Analog-to-digital converter (ADC).* The ADC is often used with a sensor. A sensor can convert certain non-electric quantities into an electric voltage. The ADC can convert a voltage into a digital value. Therefore, by combining an appropriate sensor and the ADC, a data acquisition system that measures temperature, weight, humidity, pressure, or airflow can be designed.
- *Digital-to-analog converter (DAC).* A DAC can convert a digital value into a voltage output. This function can be used in controlling DC motor speed, adjusting brightness of the light bulb, the fluid level, and so on.
- *Direct memory access (DMA) controller.* DMA is a data transfer method that requires the CPU to perform initial setup but does not require the CPU to execute instructions to control the data transfer. The initial setup for a DMA transfer includes the setup of source and destination addresses and transfer byte count. This mechanism can speed up data transfer by several times.
- *Parallel I/O interface* (often called parallel port). Parallel ports are often used to drive parallel I/O devices such as seven-segment displays, LCDs, or parallel printers.
- *Asynchronous serial I/O interface.* This transfer method transmits data bits serially without using a clock signal to synchronize the sender and receiver.
- *Synchronous serial I/O interface.* This interface transfers data bits serially and uses a clock signal to synchronize the sender and receiver.
- *Memory component interface circuitry.* All memory devices require certain control signals to operate. This interface generates appropriate control signals to be used by the memory chips.
- *Digital signal processing (DSP) feature.* Some microcontrollers (for example, the Motorola HC16) provide features such as the multiply-accumulate instruction that can execute in one clock cycle to support DSP computation.

The Motorola 68HC12 family of microcontrollers were introduced in 1996 as an upgrade for the 68HC11 microcontrollers. Members in this family implement the same instruction set and addressing modes but differ in the amount of on-chip memory and peripheral functions. The 68HC12 has the following features:

- 16-bit CPU
- Supports a standard 64KB address space
- Some members support a paged memory expansion scheme that increases the standard memory space by means of predefined windows in address space
- 768 bytes to 4KB of on-chip EEPROM
- 1KB to 12KB of on-chip SRAM
- 8-bit or 10-bit A/D converter
- 32KB to 256KB of on-chip flash or ROM memory
- Timer module that includes input capture, output compare, and pulse accumulator functions. This is the most complicated module in the 68HC12 microcontroller.
- Pulse-width modulation (PWM)
- Synchronous peripheral interface (SPI)
- Asynchronous serial communication interface (SCI)
- Byte data link communication (BDLC)
- Controller area network (CAN)
- Computer operating properly (COP) watchdog timer
- Single-wire background debug mode (BDM)
- Instructions for supporting fuzzy logic

A 68HC12 member may have from 80 to 112 pins. These signal pins are needed to support the I/O functions. A block diagram of the 912BC32 is shown in Figure 1.2.

For the time being, you may feel lost when seeing these names if you are new to microcontrollers. However, all of them will be explained in detail in the appropriate chapters.

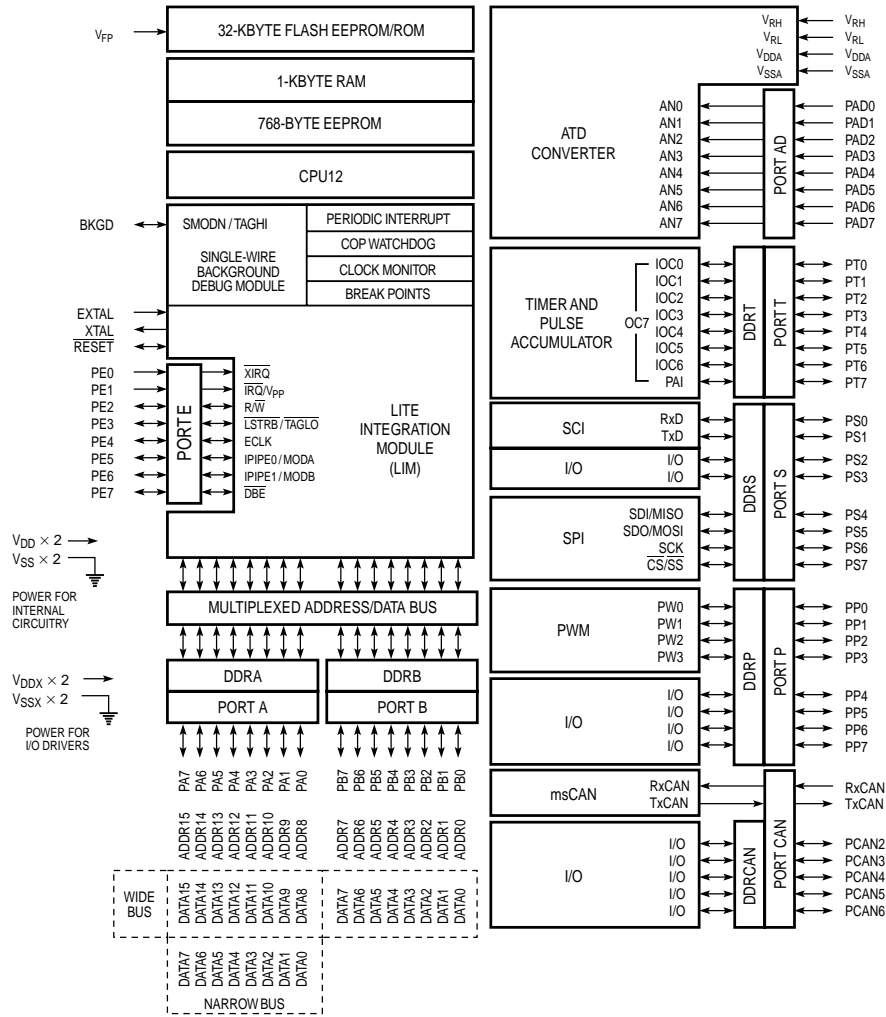


Figure 1.2 ■ Block diagram and pin assignment of the 68HC912BC32
(Redrawn with permission of Motorola)

EMBEDDED SYSTEMS

A microcontroller is designed to be used in a product to provide certain control functions. End users are interested in the features and functions of the product rather than the microcontroller. A product that incorporates one or more microcontrollers and has this nature is called an *embedded product* or *embedded system*. A cell phone is a good example of an embedded system. The cell phone contains a microcontroller that is responsible for making the phone call, accepting incoming calls, accessing Internet Web sites, displaying Web pages, handling all user input and output, keeping track of time, and so on.

Today's automobiles are also good examples of embedded systems. Most of today's new cars incorporate several microcontrollers. One microcontroller is responsible for controlling the instrument display. In order to carry out this function, the display microcontroller needs to col-

lect information using different sensors. These sensors are used to measure temperature, speed, distance, fuel level, and so on. Other microcontrollers are responsible for fuel injection control, cruise control, global positioning, giving warning when the car is too close to other cars or objects, and many other safety features available in high-priced vehicles. These microcontrollers may need to exchange information or even coordinate their operations. The Controller Area Network (CAN) bus is designed for this purpose and this bus will be discussed in detail in Chapter 12.

Another example of an embedded system is the home security system. A typical home security system consists of a microcontroller along with several sensors and actuators. Among the list of sensors are the temperature sensor, smoke detector, humidity sensor, motion detectors, and so on. When the security system detects the smoke, it may activate the alarm or even call the fire department. When the temperature is lower than a certain degree, the system may start the heater to warm the house. When the temperature is lower than a certain degree, it will start the air conditioner. When the house owner is away from home, the security system can be armed to call the neighbor or even the police department if the motion detector detects people trying to open the door. A home security system can be as sophisticated as you want.

1.2.2 Memory

Memory is the place where software programs and data are stored. A computer may contain semiconductor, magnetic, and/or optical memory. Only semiconductor memory will be discussed in this chapter. Semiconductor memory may be further classified into two major types: *random access memory*, or RAM, and *read-only memory*, or ROM.

RANDOM ACCESS MEMORY

Random access memory allows the processor to read from and write into any location on the memory chip. It takes about the same amount of time to perform a read or a write operation to any memory location. Random access memory is *volatile* in the sense that it cannot retain data without power.

There are two types of RAM technology: *dynamic RAM* (DRAM) and *static RAM* (SRAM). Dynamic random access memories are memory devices that require periodic refresh operations in order to maintain the stored information. *Refresh* is the process of restoring binary data stored in a particular memory location. The dynamic RAM uses one transistor and one capacitor to hold one bit of information. The information is stored in the capacitor in the form of electric charges. The charges stored in the capacitor will leak away over time; so periodic refresh operations are needed to maintain the contents of the DRAM. The time interval over which each memory location of a DRAM chip must be refreshed at least once in order to maintain its contents is called the *refresh period*. Refresh periods typically range from a few milliseconds to over a hundred milliseconds for today's high-density DRAMs.

Static random access memories are designed to store binary information without the need for periodic refreshes. Four to six transistors are used to store one bit of information. As long as power is stable, the information stored in the SRAM will not be degraded.

RAM is mainly used to store dynamic programs or data. A computer user often wants to run different programs on the same computer, and these programs usually operate on different sets of data. The programs and data must therefore be loaded into RAM from the hard disk or other secondary storage, and for this reason they are called dynamic.

READ-ONLY MEMORY (ROM)

ROM is *nonvolatile*. When power is removed from ROM and then reapplied, the original data will still be there. However, as its name implies, ROM data can only be read. If the processor attempts to write data to a ROM location, ROM will not accept the data, and the data in the addressed ROM memory location will not be changed.

Mask-programmed read-only memory (MROM) is a type of ROM that is programmed when it is manufactured. The semiconductor manufacturer places binary data in the memory according to the request of the customer. To be cost effective, many thousands of MROM memory units, each consisting of a copy of the same data (or program), must be sold. MROM is the major memory technology used to hold microcontroller application programs and constant data. Most people simply refer to MROM as ROM.

Programmable read-only memory (PROM) is a type of programmable read-only memory that can be programmed in the field (often by the end user) using a device called a PROM programmer or PROM “burner”. Once a PROM has been programmed, its contents cannot be changed. Because of this, PROMs are also called one-time programmable ROM (OTP). PROMs are fuse-based; in other words, end users program the fuses to configure the contents of the memory.

Erasable programmable read-only memory (EPROM) is a type of read-only memory that can be erased by subjecting it to strong ultraviolet light. The circuit design of EPROM requires us to erase the contents of a location before writing a new value into it. A quartz window on top of the EPROM integrated circuit permits ultraviolet light to be shone directly on the silicon chip inside. Once the chip is programmed, the window should be covered with dark tape to prevent gradual erasure of the data. If no window is provided, the EPROM chip becomes one-time programmable (OTP) only. Many microcontrollers incorporate on-chip one-time programmable EPROM to save cost for those users who do not need to reprogram the EPROM. EPROM is often used in prototype computers, where the software may be revised many times until it is perfected. EPROM does not allow erasure of the contents of an individual location. The only way to make changes is to erase the entire EPROM chip and reprogram it. The programming of an EPROM chip is done electrically using a device called an EPROM programmer. Today, most programmers are universal in the sense that they can program many types of devices including EPROM, EEPROM, flash memory, and *programmable logic devices*.

Electrically erasable programmable read-only memory (EEPROM) is a type of nonvolatile memory that can be erased by electrical signals and reprogrammed. Like EPROM, the circuit design of EEPROM also requires users to erase the contents of a memory location before a new value can be written into it. EEPROM allows each individual location to be erased and reprogrammed. Unlike EPROM, EEPROM can be erased and programmed using the same programmer. However, EEPROM pays the price for being very flexible in its erasability. The cost of an EEPROM chip is much higher than that of an EPROM chip of comparable density.

Flash memory was invented to incorporate the advantages and avoid the drawbacks of both the EPROM and EEPROM technologies. Flash memory can be erased and reprogrammed in the system without using a dedicated programmer. It achieves the density of EPROM, but it does not require a window for erasure. Like EEPROM, flash memory can be programmed and erased electrically. However, it does not allow the erasure of an individual memory location—the user can only erase a block or the entire chip. Today, more and more microcontrollers are incorporating on-chip flash memory for storing programs and static data.

1.3 The Computer’s Software

Programs are known as *software*. A *program* is a set of instructions that the computer hardware can execute. A program is stored in the computer’s memory in the form of binary numbers called *machine instructions*. For example, the 68HC12 machine instruction

```
0001 1000 0000 0110
```

adds the contents of accumulator B and accumulator A together and leaves the sum in accumulator A. The machine instruction

```
0100 0011
```

decrements the contents of accumulator A by 1. The machine instruction

```
1000 0110 0000 0110
```

places the value 6 in accumulator A.

Writing programs in machine language is extremely difficult and inefficient:

1. *Program entering.* The programmer will need to memorize the binary pattern of every machine instruction, which can be very challenging because a microprocessor may have several hundred machine instructions, and each machine instruction can have different length. Constant table lookup will be necessary during the program entering process. On the other hand, programmers are forced to work on program logic at a very low level because each machine instruction implements only a very primitive operation.
2. *Program debugging.* Whenever there are errors, it is extremely difficult to trace the program because the program consists of only sequences of 0s and 1s. A programmer will need to identify each machine instruction and then think about what operation is performed by that instruction. This is not an easy job.
3. *Program maintenance.* Most programs will need to be maintained in the long run. A programmer who did not write the program will have a hard time reading the program and following the program logic.

Assembly language was invented to simplify the programming job. An *assembly program* consists of assembly instructions. An *assembly instruction* is the mnemonic representation of a machine instruction. For example, in the 68HC12:

ABA stands for "add the contents of accumulator B to accumulator A." The corresponding machine instruction is 00011000 00000110.

DECA stands for "decrements the contents of accumulator A by 1." The corresponding machine instruction is 0100 0011.

A programmer no longer needs to scan through 0s and 1s in order to identify what instructions are in the program. This is a significant improvement over machine language programming.

The assembly program that the programmer enters is called *source program* or *source code*. A software program called an *assembler* is then invoked to translate the program written in assembly language into machine instructions. The output of the assembly process is called *object code*. It is a common practice to use a *cross assembler* to translate assembly programs. A cross assembler is an assembler that runs on one computer but generates machine instructions to be run on a different computer with a totally different instruction set. In contrast, a *native assembler* runs on a computer and generates machine instructions to be executed by machines that have the same instruction set. The freeware **as12** is a cross assembler that runs on an IBM PC and generates machine code that can be downloaded into a 68HC12-based computer for execution. The list file generated by an assembler shows both the source code and machine code (encoded in hexadecimal format). Here is an example:

line	addr.	machine code	source code	
1:		=00001000	org	\$1000
2:	1000	B6 0800	ldaa	\$800
3:	1003	BB 0801	adda	\$801
4:	1006	BB 0802	adda	\$802
5:	1009	7A 0900	staa	\$900
6:			end	

There are several drawbacks to programming in assembly language:

- The programmer must be very familiar with the hardware organization of the microcontroller on which the program is to be executed.
- A program (especially a long one) written in assembly language is extremely difficult to understand for anyone other than the author.
- Programming productivity is not satisfactory for large programming projects because the programmer needs to work on the program logic at a very low level.

For these reasons, high-level languages such as Fortran, COBOL, PASCAL, C, C++, and JAVA were invented to avoid the drawbacks of assembly language programming. High-level languages are very close to plain English and hence a program written in high-level language becomes easier to understand. A statement in high-level language often needs to be implemented by tens or even hundreds of assembly instructions. The programmer can now work on the program logic at a much higher level and achieve higher productivity. A program written in a high-level language is also called a *source program*, and it requires a software program called a *compiler* to translate it into machine instructions. A compiler compiles a program into *object code*. Just as there are cross assemblers, there are *cross compilers* that run on one computer but translate programs into machine instructions to be executed on a computer with a different instruction set.

High-level languages have their own problems. One of their problems is that the resulting machine code cannot run as fast as its equivalent in assembly language due to the inefficiency of the compilation process. For this reason, many time-critical programs are still written in assembly language.

The C language has been used extensively in microcontroller programming in the industry. The Web site at *Microcontroller.com* reported that C has been used to program 80% of the embedded systems, whereas assembly language is used in 75% of embedded systems. Both C and assembly language will be used throughout this text. The C programs in this book will be compiled using the ImageCraft ICC12 compiler and tested on Axiom evaluation boards.

1.4 The 68HC12 CPU Registers

The 68HC12 microcontroller has many registers. These registers can be classified into two categories: CPU registers and I/O registers. CPU registers are used solely to perform general-purpose operations such as arithmetic, logic, and program flow control. I/O registers are mainly used to configure the operations of peripheral functions, to hold data transferred in and out of the peripheral subsystem, and to record the status of I/O operations. The I/O registers in a microcontroller can further be classified into *data*, *data direction*, *control*, and *status registers*. These registers are treated as memory locations when they are accessed. CPU registers do not occupy the 68HC12 memory space.

The CPU registers of the 68HC12 are shown in Figure 1.3 and are listed below. Some of the registers are 8-bit and others are 16-bit.

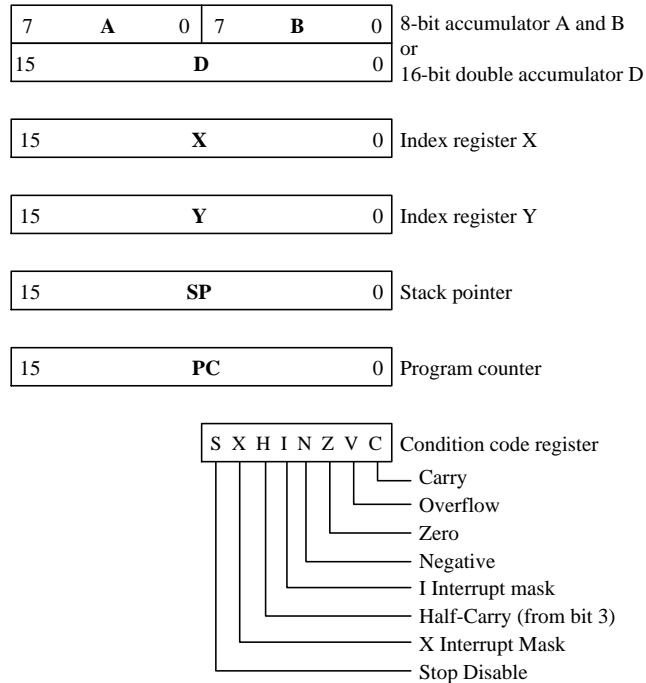


Figure 1.3 ■ MC68HC12 CPU registers

General-purpose accumulator A and B. Both A and B are 8-bit registers. Most arithmetic functions are performed on these two registers. These two accumulators can also be concatenated to form a single 16-bit accumulator that is referred to as the D accumulator.

Index registers X and Y. These two registers are used mainly in forming operand addresses during the instruction execution process. However, they are also used in several arithmetic operations.

Stack pointer (SP). A stack is a first-in-first-out data structure. The 68HC12 has a 16-bit stack pointer that points to the top byte of the stack (shown in Figure 1.4). The stack grows toward lower addresses. The use of stack will be discussed in Chapter 4.

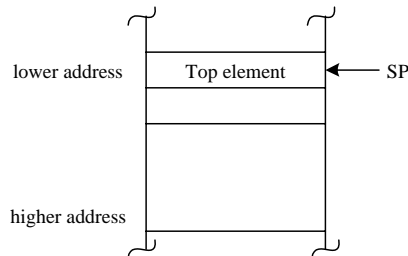


Figure 1.4 ■ 68HC12 stack structure

Program counter (PC). The 16-bit PC holds the address of the next instruction to be executed. After the execution of an instruction, the PC is incremented by the number of bytes of the executed instruction.

Condition code register (CCR). This 8-bit register is used to keep track of the program execution status, control the execution of conditional instructions, and enable/disable the interrupt handling. The contents of the CCR register are shown in Figure 1.3. The function of each condition code bit will be explained in later sections and chapters.

The 68HC12 supports the following types of data:

- Bits
- 5-bit signed integers
- 8-bit signed and unsigned integers
- 8-bit, 2-digit binary-coded-decimal numbers
- 9-bit signed integers
- 16-bit signed and unsigned integers
- 16-bit effective addresses
- 32-bit signed and unsigned integers

Negative numbers are represented in two's complement format. Five-bit and nine-bit signed integers are formed during addressing mode computations. Sixteen-bit effective addresses are formed during addressing mode computations. Thirty-two-bit integer dividends are used by extended division instructions. Extended multiply and extended multiply-and-accumulate instructions produce 32-bit products.

A multi-byte integer (16-bit or 32-bit) is stored in memory from most significant to least significant bytes starting from low to higher addresses. A number can be represented in binary, octal, decimal, or hexadecimal format. An appropriate prefix (shown in Table 1.1) is added in front of the number to indicate its base:

Base	Prefix	Example
binary	%	%10001010
octal	@	@1234567
decimal		12345678
hexadecimal (shorthand hex)	\$	\$5678

Table 1.1 ■ Prefixes for number bases

1.5 Memory Addressing

Memory consists of a sequence of directly addressable “locations.” A memory location is referred to as an *information unit*. A memory location can be used to store information such as data, instructions, and the status of peripheral devices. An information unit has two components: its *address* and its *contents*, as shown in Figure 1.5.

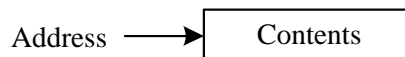


Figure 1.5 ■ The components of a memory location

Each location in memory has an address that must be supplied before its contents can be accessed. The CPU communicates with memory by first identifying the location's address and then passing this address on the address bus. This is similar to when a UPS delivery person needs an address to deliver a parcel. The data are transferred between memory and the CPU along the data bus (see Figure 1.6). The number of bits that can be transferred on the data bus at once is called the *data bus width* of the processor.

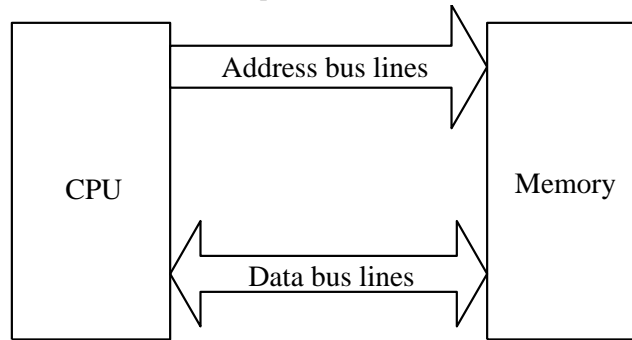


Figure 1.6 ■ Transferring data between CPU and memory

The size of memory is measured in bytes. A byte consists of 8 bits. A 4-bit quantity is called a *nibble*. A 16-bit quantity is called a *word*. To simplify the quantification of memory, the unit *kilobyte* (KB) is often used. K is given by the following formula:

$$K = 2^{10} = 1024$$

Another frequently used unit is *megabyte* (MB), which is given by the following formula:

$$M = K^2 = 2^{20} = 1024 \times 1024 = 1048576$$

The 68HC12 has a 16-bit address bus and a 16-bit data bus. The 16-bit data bus allows the 68HC12 to access 16-bit data in one operation. The 16-bit address bus enables the 68HC12 to address directly up to 2^{16} (65,536) different memory locations. Certain 68HC12 members use paging techniques to allow user programs to access more than 64KB.

In this book, we will use the notation **m[addr]** and **[reg]** to refer to the contents of a memory location at **addr** and the contents of the register **reg**, respectively. For example,

m[\$20]

refers to the contents of the memory location at \$20 and **[A]** refers to the contents of accumulator A.

1.6 68HC12 Addressing Modes

A 68HC12 instruction consists of one or two bytes of opcode and zero to five bytes of operand addressing information. The opcode byte(s) specifies the operation to be performed and the addressing modes to be used to access the operand(s). The first byte of a two-byte opcode is \$18.

Addressing modes determines how the CPU accesses memory locations to be operated upon. Addressing modes supported by the 68HC12 are summarized in Table 1.2.

Addressing mode	Source format	Abbre.	Description
Inherent	INST (no externally supplied operands)	INH	Operands (if any) are in CPU registers
Immediate	INST #opr8i or INST #opr16i	IMM	Operand is included in instruction stream. 8- or 16-bit size implied by context
Direct	INST opr8a	DIR	Operand is the lower 8 bits of an address in the range \$0000-\$00FF
Extended	INST opr16a	EXT	Operand is a 16-bit address
Relative	INST rel8 or INST rel16	REL	An 8-bit or 16-bit relative offset from the current PC is supplied in the instruction
Indexed (5-bit offset)	INST oprx5,xysp	IDX	5-bit signed constant offset from x,y,sp, or pc
Indexed (pre-decrement)	INST oprx3,-xys	IDX	Auto pre-decrement x, y, or sp by 1 ~ 8
Indexed (pre-increment)	INST oprx3,+xys	IDX	Auto pre-increment x, y, or sp by 1 ~ 8
Indexed (post-decrement)	INST oprx3,xys-	IDX	Auto post-decrement x, y, or sp by 1 ~ 8
Indexed (post-increment)	INST oprx3,xys+	IDX	Auto post-increment x, y, or sp by 1 ~ 8
Indexed (accumulator offset)	INST abd,xysp	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from x, y, sp, or pc
Indexed (9-bit offset)	INST oprx9,xysp	IDX1	9-bit signed constant offset from x, y, sp, or pc (lower 8-bits of offset in one extension byte)
Indexed (16-bit offset)	INST oprx16,xysp	IDX2	16-bit constant offset from x, y, sp, or pc (16-bit offset in two extension bytes)
Indexed-Indirect (16-bit offset)	INST [oprx16,xysp]	[IDX2]	Pointer to operand is found at 16-bit constant offset from (x, y, sp, or pc)
Indexed-Indirect (D accumulator offset)	INST [D,xysp]	[D,IDX]	Pointer to operand is found at x, y, sp, or pc plus the value in D
<p>Note</p> <p>1. INST stands for the instruction mnemonic. 4. opr8a stands for 8-bit address 2. opr8i stands for 8-bit immediate value. 5. opr16a stands for 16-bit address 3. opr16i stands for 16-bit immediate value. 6. oprx3 stands for 3-bit value (amount to be incremented or decremented).</p>			

Table 1.2 ■ M68HC12 addressing mode summary

1.6.1 Inherent Mode

Instructions that use this addressing mode either have no operands or all operands are in internal CPU registers. In either case, the CPU does not need to access any memory locations to complete the instruction.

For example, the following two instructions use inherent mode:

```
NOP          ; this instruction has no operands
INX          ; operand is a CPU register
```

1.6.2 Immediate Mode

Operands for immediate mode instructions are included in the instruction stream. The CPU does not access memory when this type of instruction is executed. An immediate value can be 8-bit or 16-bit depending on the context of the instruction. An immediate value is preceded by a # character in the assembly instruction.

For example:

```
LDA #55 ; A ← $55
```

places the value \$55 (hexadecimal) in accumulator A when this instruction is executed.

```
LDX #2000 ; X ← $2000
```

places the hex value \$2000 in index register X when this instruction is executed.

```
LDY #44 ; Y ← $0044
```

places the hex value \$0044 in index register Y when this instruction is executed. Only an 8-bit value was supplied in this instruction. However, the assembler will generate the 16-bit value \$0044 because the CPU expects a 16-bit value when this instruction is executed.

1.6.3 Direct Mode

This addressing mode is sometimes called zero-page addressing because it is used to access operands in the address range of \$0000–\$00FF. Since these addresses begin with \$00, only the eight low-order bits of the address need to be included in the instruction, which saves program space and execution time.

For example:

```
LDA $20 ; A ← m[$20]
```

fetches the contents of the memory location at \$0020 and puts it in accumulator A.

```
LDX $20 ; XH ← m[$20], XL ← m[$21]
```

fetches the contents of memory locations at \$0020 and \$0021 and places them in the upper and lower bytes (X_H and X_L) of the index register X, respectively.

1.6.4 Extended Mode

In this addressing mode, the full 16-bit address of the memory location to be operated on is provided in the instruction. This addressing mode can be used to access any location in the 64-KB memory map.

For example:

```
LDA $2000 ; A ← m[$2000]
```

copies the contents of the memory location at \$2000 into accumulator A.

1.6.5 Relative Mode

The relative addressing mode is used only by branch instructions. Short and long conditional branch instructions use relative addressing mode exclusively. Branching versions of bit manipulation instructions (BRSET and BRCLR) may also use the relative addressing mode to specify the branch target.

A short branch instruction consists of an 8-bit opcode and a signed 8-bit offset contained in the byte that follows the opcode. Long branch instructions consist of an 8-bit prebyte, an 8-bit opcode, and a signed 16-bit offset contained in two bytes that follow the opcode.

Each conditional branch instruction tests certain status bits in the condition code register. If the bits are in a specified state, the offset is added to the address of the next memory location after the offset to form an effective address, and execution continues at that address; if the bits are not in the specified state, execution continues with the instruction immediately following the branch instruction.

Both 8-bit and 16-bit offsets are signed two's complement numbers to support branching forward and backward in memory. The numeric range of short branch offset values is \$80 (-128) to \$7F (127). The numeric range of long branch offset values is \$8000 (-32768) to \$7FFF (32767). If the offset is zero, the CPU executes the instruction immediately following the branch instruction, regardless of the test result.

Branch offset is often specified using a label rather than a numeric value due to the difficulty of calculating the exact value of the offset. For example, in the following instruction segment:

```

minus .
      .           ; if N (of CCR) = 1
      .           ;           PC ← PC + branch offset
      bmi    minus ; else
      ...           ;           PC ← PC

```

The instruction **bmi minus** will cause the 68HC12 to execute the instruction with the label **minus** if the N flag of the CCR register is set to 1.

The assembler will calculate the appropriate branch offset when the symbol that represents the branch target is encountered. Using a symbol to specify the branch target makes the programming task easier.

1.6.6 Indexed Modes

There are quite a few variations for the indexed addressing scheme. The indexed addressing scheme uses a postbyte plus 0, 1, or 2 extension bytes after the instruction opcode. The postbyte and extensions implement the following functions:

- Specify which index register is used.
- Determine whether a value in an accumulator is used as an offset.
- Enable automatic pre- or post-increment or pre- or post-decrement.
- Specify the size of increment or decrement.
- Specify the use of 5-, 9-, or 16-bit signed offsets.

The indexed addressing scheme allows:

- the stack pointer to be used as an index register in all indexed operations
- the program counter to be used as an index register in all but autoincrement and autodecrement modes
- the value in accumulator A, B, or D to be used as an offset
- automatic pre- or post-increment or pre- or post-decrement by -8 to +8
- a choice of 5-, 9-, or 16-bit signed constant offsets
- indirect indexing with 16-bit offset or accumulator D as the offset

Table 1.3 is a summary of indexed addressing mode capabilities and a description of postbyte encoding. The postbyte is noted as **xb** in instruction descriptions.

Postbyte code (xb)	source code syntax	Comments	
		rr: 00 = X, 01 = Y, 10 = SP, 11 = PC	
rr0nnnn	r n, r -n, r	5-bit constant offset n = -16 to +15 r can be X, Y, SP or PC	
111r0zs	n, r -n, r	Constant offset (9- or 16-bit signed) z- 0 = 9-bit with sign in LSB of postbyte (s) -256 < n < 255 1 = 16-bit 0 < n < 65,536 if z = s = 1, 16-bit offset indexed-indirect (see below) r can be X, Y, SP, or PC	
111r011	[n, r]	16-bit offset indexed-indirect rr can be X, Y, SP, or PC	0 < n < 65,536
rr1pnnnn	n, -r n, +r n, r- n, r+	Auto pre-decrement/increment or auto post-decrement/increment; p = pre-(0) or post-(1), n = -8 to -1, +1 to +8 r can be X, Y, or SP (PC not a valid choice) +8 = 0111 ... +1 = 0000 -1 = 1111 ... -8 = 1000	
111r1aa	A, r B, r D, r	Accumulator offset (unsigned 8-bit or 16-bit) aa- 00 = A 01 = B 10 = D (16-bit) 11 = see accumulator D offset indexed-indirect r can be X, Y, SP, or PC	
111r111	[D, r]	Accumulator D offset indexed-indirect r can be X, Y, SP or PC	

Table 1.3 ■ Summary of indexed operations

5-BIT CONSTANT OFFSET INDEXED ADDRESSING

This indexed addressing mode adds a 5-bit signed offset that is included in the instruction postbyte to the base index register to form the effective address. The base index register can be X, Y, SP, or PC. The range of the offset is from -16 to +15. Although the range of the offset is short, it is the most often used offset. Using this indexed addressing mode can make the instruction shorter.

For example:

```
LDAA 0,X
```

loads the contents of the memory location pointed to by index register X (i.e., X contains the address of the memory location to be accessed) into accumulator A.

```
STAB -8,X
```

stores the contents of accumulator B in the memory location with the address, which is equal to the contents of index register X minus 8.

9-BIT CONSTANT OFFSET INDEXED ADDRESSING

This indexed addressing mode uses a 9-bit signed offset, which is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction. This gives a range of -256 through $+255$ from the value in the base index register.

For example:

```
LDAA $FF,X
```

loads the contents of the memory location located at the address, which is equal to 255 plus the value in index register X.

```
LDAB -20,Y
```

loads the contents of the memory location with the address, which is equal to the value in index register Y minus 20.

16-BIT CONSTANT OFFSET INDEXED ADDRESSING

This indexed addressing mode specifies a 16-bit offset to be added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction. This allows access to any location in the 64-KB address space. Since the address bus and the offset are both 16 bits, it does not matter whether the offset value is considered to be a signed or an unsigned value. The 16-bit offset is provided as two extension bytes after the instruction postbyte in the instruction flow.

16-BIT CONSTANT INDIRECT INDEXED ADDRESSING

This indexed addressing mode adds a 16-bit offset to the base index register to form the address of a memory location that contains a pointer to the memory location affected by the instruction. The square brackets distinguish this addressing mode from 16-bit constant offset indexing.

For example:

```
LDAA [10,X]
```

In this example, index register X holds the base address of a table of pointers. Assume that X has an initial value of \$1000, and that \$2000 is stored at addresses \$100A and \$100B. The instruction first adds 10 to the value in X to form the address \$100A. Next, an address pointer (\$2000) is fetched from memory at \$100A. Then the value stored in \$2000 is read and loaded into accumulator A.

AUTO PRE/POST DECREMENT/INCREMENT INDEXED ADDRESSING

This indexed addressing mode provides four ways to automatically change the value in a base index register as a part of instruction execution. The index register can be incremented or decremented by an integer value either before or after indexing takes place. The base index register may be X, Y, or SP.

Pre-decrement and pre-increment versions of the addressing mode adjust the value of the index register before accessing the memory location affected by the instruction—the index register retains the changed value after the instruction executes. Post-decrement and post-increment versions of the addressing mode use the initial value in the index register to access the memory location affected by the instruction, and then change the value of the index register.

The 68HC12 allows the index register to be incremented or decremented by any integer value in the ranges -8 through -1 , or 1 through 8 . The value need not be related to the size of the operand for the current instruction. These instructions can be used to incorporate an index adjustment into an existing instruction rather than using an additional instruction and increas-

ing execution time. This addressing mode is also used to perform operations on a series of data structures in memory.

For example:

```
STAA 1, -SP
```

stores the contents of accumulator A at the memory location with the address equal to the value of stack pointer SP minus 1. After the store operation, the contents of SP are decremented by 1.

```
LDX 2, SP+
```

loads the contents of the memory locations pointed to by the stack pointer SP and also increments the value of SP by 2 after the instruction execution.

ACCUMULATOR OFFSET INDEXED ADDRESSING

In this indexed addressing mode, the effective address is the sum of the values in the base index register and an unsigned offset in one of the accumulators. The value in the base index register itself is not changed. The base register can be X, Y, SP, or PC, and the accumulator can be either of the 8-bit accumulators (A or B) or the 16-bit D accumulator.

For example, the instruction:

```
LDAA B,X
```

loads the contents of the memory location into accumulator A with the address equal to the sum of the values of accumulator B and index register X. Both B and X are not changed after the instruction execution.

ACCUMULATOR D INDIRECT INDEXED ADDRESSING

This indexed addressing mode adds the value in accumulator D to the value in the base index register to form the address of a memory location that contains a pointer to the memory location affected by the instruction. The instruction operand does not point to the address of the memory location to be acted upon, but rather to the location of a pointer to the location to be acted upon. The square brackets distinguish this addressing mode from accumulator D offset indexing.

For example, the following instruction sequence implements a computed GOTO statement:

```

      JMP      [D, PC]
GO1  DC.W    target1  ; the keyword DC.W reserves two bytes to hold the
GO2  DC.W    target2  ; value of the symbol that follows
GO3  DC.W    target3  ;
      ...
target1...
      .
      .
target2...
      .
      .
target3...
      .
      .

```

In this instruction segment, the names (also called labels) *target1*, *target2*, and *target3* are labels that represent the addresses of the memory locations that the JMP instruction may jump to. The names GO1, GO2, and GO3 are also labels. They represent the memory locations that hold the values of the labels *target1*, *target2*, and *target3*, respectively.

The values beginning at GO1 are addresses of potential destinations of the jump instructions. At the time the **JMP [D, PC]** instruction is executed, PC points to the address GO1, and D holds one of the values \$0000, \$0002, or \$0004 (determined by the program some time before the JMP).

Assume that the value in D is \$0002. The JMP instruction adds the values in D and PC to form the address of GO2 and jumps to target2. The locations of target1 through target3 are known at the time of program assembly but the destination of the JMP depends on the value in D computed during program execution.

1.7 Addressing More than 64KB

Some 68HC12 devices (for example, 68HC912DG128) incorporate hardware that supports addressing a larger memory space than the standard 64KB. The expanded memory system is accessed by using the bank-switching scheme.

The devices with expanded memory treat the 16KB of memory space from \$8000 to \$BFFF as a program memory window. Expanded-memory devices also have an 8-bit program page register (PPAGE), which allows up to 256 16KB program memory pages to be switched into and out of the program memory window. This provides up to 4MB of paged program memory.

Accessing expanded memory will be discussed in later chapters.

1.8 68HC12 Instruction Examples

In this section we will examine several groups of instructions to develop a feel for the 68HC12 instruction set and learn some simple applications of these instructions.

1.8.1 The Load & Store Instructions

The load instruction copies the contents of a memory location or places an immediate value into an accumulator or a register. Memory contents are not changed.

Store instructions copy the contents of a CPU register into a memory location. The contents of the accumulator or CPU register are not changed. Store instructions automatically update the N and Z flags in the condition code register (CCR).

Table 1.4 shows a summary of load and store instructions.

All except for the relative addressing mode can be used to select the memory location or value to be loaded into an accumulator or a CPU register. All except for the relative and immediate addressing modes can be used to select the memory location to store the contents of a CPU register.

For example, the following instruction loads the contents of the memory location pointed to by index register X into accumulator A:

```
LDAA 0,X
```

The following instruction loads the contents of the memory location at \$1004 into accumulator B:

```
LDAB $1004
```

The following instruction copies the contents of accumulator A into the memory location at \$20:

```
STAA $20
```

Mnemonic	Function	Operation
LDAA	Load A	M) ⇒ A
LDAB	Load B	(M) ⇒ B
LDD	Load D	(M:M+1) ⇒ (A:B)
LDS	Load SP	(M:M+1) ⇒ SP
LDX	Load index register X	(M:M+1) ⇒ X
LDY	Load index register Y	(M:M+1) ⇒ X
LEAS	Load effective address into SP	Effective address ⇒ SP
LEAX	Load effective address into X	Effective address ⇒ X
LEAY	Load effective address into Y	Effective address ⇒ Y
Store Instructions		
Mnemonic	Function	Operation
STAA	Store A	A) ⇒ M
STAB	Store B	(B) ⇒ M
STD	Store D	(A) ⇒ M, (B) ⇒ M+1
STS	Store SP	(SP) ⇒ M, M+1
STX	Store X	(X) ⇒ M:M+1
STY	Store Y	(Y) ⇒ M:M+1

Table 1.4 ■ Load and store instructions

The following instruction stores the contents of index register X in memory locations at \$8000 and \$8001:

```
STX $8000
```

When dealing with a complex data structure such as a record, we often use an index register or the stack pointer to point to the beginning of the data structure and use the indexed addressing mode to access the elements of the data structure. For example, there is a record that contains the following four fields:

- ID number (unit *none*, size four bytes)
- height (unit *inch*, size one byte)
- weight (unit *pound*, size two bytes)
- age (unit *year*, size one byte)

Suppose this record is stored in memory starting at \$6000. Then we can use the following instruction sequence to access the weight field:

```
LDX #$6000          ; set X to point to the beginning of data structure
LDD 5, X            ; copy weight into D
```

1.8.2 Transfer & Exchange Instructions

A summary of transfer and exchange instructions are displayed in Table 1.5. Transfer instructions copy the contents of a register or accumulator into another register or accumulator. Source content is not changed by the operation. TFR is a universal transfer instruction, but other mnemonics are accepted for compatibility with the M68HC11. The TAB and TBA instructions affect the N, Z, and V condition code bits. The TFR instruction does not affect the condition code bits.

Transfer Instructions		
Mnemonic	Function	Operation
TAB	Transfer A to B	(A) ⇒ B
TAP	Transfer A to CCR	(A) ⇒ CCR
TBA	Transfer B to A	(B) ⇒ A
TFR	Transfer register to register	(A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP
TPA	Transfer CCR to A	(CCR) ⇒ A
TSX	Transfer SP to X	(SP) ⇒ X
TSY	Transfer SP to Y	(SP) ⇒ Y
TXS	Transfer X to SP	(X) ⇒ SP
TYS	Transfer Y to SP	(Y) ⇒ SP
Exchange Instructions		
Mnemonic	Function	Operation
EXG	Exchange register to register	(A, B, CCR, D, X, Y, or SP) ⇔ A, B, CCR, D, X, Y, or SP
XGDX	Exchange D with X	(D) ⇔ X
XGDY	Exchange D with Y	(D) ⇔ Y
Sign Extension Instructions		
Mnemonic	Function	Operation
SEX	Sign extend 8-bit operand	(A, B, CCR) ⇒ X, Y, or SP

Table 1.5 ■ Transfer and exchange instructions

Exchange instructions exchange the contents of pairs of registers or accumulators. For example:

```
EXG  A, B
```

exchanges the contents of accumulator A and B.

```
EXG  D,X
```

exchanges the contents of double accumulator D and index register X.

The SEX instruction is a special case of the universal transfer instruction that is used to sign-extend 8-bit two's complement numbers so that they can be used in

16-bit operations. The 8-bit number is copied from accumulator A, accumulator B, or the condition code register to accumulator D, index register X, index register Y, or the stack pointer. All the bits in the upper byte of the 16-bit result are given the value of the most significant bit of the 8-bit number. For example,

```
SEX  A,X
```

copies the contents of accumulator A to the lower byte of X and duplicates the bit 7 of A to every bit of the upper byte of X.

```
SEX  B,Y
```

copies the contents of accumulator B to the lower byte of Y and duplicates the bit 7 of B to every bit of the upper byte of Y.

Transfer instructions allow operands to be placed in the right register so that the desired operation can be performed. For example, if we want to compute the squared value of accumulator A, we can use the following instruction sequence:

```
TAB          ; B ← (A)
MUL          ; A:B ← (A) ? (B)
```

Applications of other transfer and exchange instructions will be discussed in Chapters 2 and 4.

1.8.3 Move Instructions

A summary of move instructions is listed in Table 1.6. These instructions move data bytes or words from a source ($M1, M: M+1_1$) to a destination ($M2, M: M+1_2$) in memory. Six combinations of immediate, extended, and indexed addressing are allowed to specify source and destination addresses ($IMM \Rightarrow EXT$, $IMM \Rightarrow IDX$, $EXT \Rightarrow EXT$, $EXT \Rightarrow IDX$, $IDX \Rightarrow EXT$, $IDX \Rightarrow IDX$).

Transfer Instructions		
Mnemonic	Function	Operation
MOVB	Move byte (8-bit)	$(M1) \Rightarrow M2$
MOVW	Move word (16-bit)	$(M:M+1_1) \Rightarrow M:M+1_2$

Table 1.6 ■ Move instructions

For example, the following instruction copies the contents of the memory location at \$1000 to the memory location at \$2000:

```
MOVB $1000, $2000
```

The following instruction copies the 16-bit word pointed to by X to the memory location pointed by Y:

```
MOVW 0,X, 0,Y
```

1.8.4 Add & Subtract Instructions

Add and subtract instructions allow the 68HC12 to perform fundamental arithmetic operations. A summary of Add and Subtract instructions is shown in Table 1.7.

Add Instructions		
Mnemonic	Function	Operation
ABA	Add B to A	$(A) + (B) \Rightarrow A$
ABX	Add B to X	$(B) + (X) \Rightarrow X$
ABY	Add B to Y	$(B) + (Y) \Rightarrow Y$
ADCA	Add with carry to A	$(A) + (M) + C \Rightarrow A$
ADCB	Add with carry to B	$(B) + (M) + C \Rightarrow B$
ADDA	Add without carry to A	$(A) + (M) \Rightarrow A$
ADDB	Add without carry to B	$(B) + (M) \Rightarrow B$
ADDD	Add without carry to D	$(A:B) + (M:M+1) \Rightarrow A:B$
Subtract Instructions		
Mnemonic	Function	Operation
SBA	Subtract B from A	$(A) - (B) \Rightarrow A$
SBCA	Subtract with borrow from A	$(A) - (M) - C \Rightarrow A$
SBCB	Subtract with borrow from B	$(B) - (M) - C \Rightarrow B$
SUBA	Subtract memory from A	$(A) - (M) \Rightarrow A$
SUBB	Subtract memory from B	$(B) - (M) \Rightarrow B$
SUBD	Subtract memory from D	$(D) - (M:M+1) \Rightarrow D$

Table 1.7 ■ Add and subtract instructions

Example 1.1

Write an instruction sequence to add 3 to the memory locations at \$10 and \$15.

Solution: A memory cannot be the destination of an ADD instruction. Therefore, we need to copy the memory content into an accumulator, add 3 to it, and then store the sum back to the same memory locations.

```
LDAA $10      ; copy the contents of $10 into accumulator A
ADDA #3       ; add 3 to A
STAA $10     ; store the sum back to $10
LDAA $15     ; copy the contents of $15 into accumulator A
ADDA #3       ; add 3 to A
STAA $15     ; store the sum back to $15
```

Example 1.2

Write an instruction sequence to add the byte pointed to by index register X with the following byte, and place the sum at the memory location pointed to by index register Y.

Solution: The byte pointed to by index register X and the following byte can be accessed by using the indexed addressing mode.

```
LDAA 0,X          ; put the byte pointed to by X in A
ADDA 1,X          ; add the following byte to A
STAA 0,Y          ; store the sum at the location pointed to by Y
```

Example 1.3

Write an instruction sequence to add the numbers stored at \$800 and \$801 and store the sum at \$804.

Solution: To add these two numbers, we need to put one of them in an accumulator:

```
LDAA $800        ; copy the number in $800 to accumulator A
ADDA $801        ; add the second number to A
STAA $804        ; save the sum at $804
```

1.9 Instruction Queue

The 68HC12 executes one instruction at a time and many instructions take several clock cycles to complete. When the CPU is executing the instruction, it does not need to access memory for the operand in every clock cycle. The design of the 68HC12 takes advantage of this fact to prefetch instruction bytes from the memory and put them in a queue to speed up the instruction execution process.

There are two 16-bit queue stages and one 16-bit buffer. Program information is fetched in aligned 16-bit words. Unless buffering is required, program information is first queued into stage 1, and then advanced to stage 2 for execution.

At least two words of program information are available to the CPU when execution begins. The first byte of object code is in either the even or odd half of the word in stage 2, and at least two more bytes of object code are in queue.

Queue logic manages the position of program information so that the CPU itself does not deal with alignment. As it is executed, each instruction initiates at least enough program word fetches to replace its own object code in the queue.

The buffer is used when a program word arrives before the queue can advance. This occurs during execution of single-byte and odd-aligned instructions. For instance, the queue cannot advance after an aligned, single-byte instruction is executed, because the first byte of the next instruction is also in stage 2. In these cases, information is latched into the buffer until the queue can advance.

1.10 Instruction Execution Cycle

In order to execute a program, the microprocessor or microcontroller must access memory to fetch instructions or operands. The process of accessing a memory location is called a *read*

cycle, the process of storing a value in a memory location is called a *write cycle*, and the process of executing an instruction is called an *instruction execution cycle*.

When executing an instruction, the 68HC12 performs a combination of the following operations:

- One or multiple read cycles to fetch instruction opcode byte(s) and addressing information.
- One or more read cycles to fetch the memory operand(s) (optional).
- The operation specified by the opcode.
- One or more write cycles to write back the result to either a register or a memory location (optional).

1.11 Summary

The invention of the microprocessor in 1971 resulted in the revolution of the electronics industry. The first microprocessor, the Intel 4004, implemented a simplified CPU onto an integrated circuit (IC). Following the introduction of the 4-bit 4004, Intel introduced the 8-bit 8008, 8080, and 8085 over three years. The introduction of the 8085 was a big success. The key to its success lies in its programmability. Through this programmability, many products can be designed and constructed. Other companies also joined in the design and manufacturing of microprocessors. Zilog, Motorola, and Rockwell are among the more successful ones.

The earliest microprocessors still needed peripheral chips to interface with I/O devices such as seven-segment displays, printers, timers, and so on. Memory chips were also needed to hold the application program and dynamic data. Because of this, the products designed with microprocessors could not be made as small as desired. Then came the introduction of microcontrollers, which incorporated the CPU, some amount of memory, and peripheral functions such as parallel ports, time instructions, and serial interface functions onto one chip. The development of microcontrollers has had the following impacts:

- I/O interfacing issue is greatly simplified.
- External memory is no longer needed for many applications.
- System design time is greatly shortened.

A microcontroller is not designed to build a desktop computer. Instead, it is used as the controller of many products. End users of these products do not care about what microcontrollers are used in their appliances. They only care about the functionality of the product. A product that uses a certain microcontroller as a controller and has this characteristic is called an *embedded system*. Cell phones, automobiles, cable modems, HDTVs, and home security systems are well-known embedded systems.

Over the last 20 years, we can see clearly how a microcontroller needs to incorporate some or all of the following peripheral functions in order to be useful:

- Timer module that incorporates input capture, output compare, real-time interrupt, and counting capability
- Pulse-width modulation function for easy waveform generation
- Analog-to-digital converter
- Digital-to-analog converter
- Temperature sensor
- Direct memory access (DMA) controller

- Parallel I/O interface
- Serial I/O interface such as UART, SPI, Microwire, I²C, and CAN
- Memory component interface circuitry

The 68HC12 from Motorola implements most of the above peripheral modules and the CPU onto one VLSI chip.

Memory is where software programs and data are stored. Semiconductor memory chips can be classified into two major categories: random-access memory (RAM) and read-only memory (ROM). RAM technology includes DRAM and SRAM. MROM, PROM, EPROM, EEPROM, and flash memory are read-only memories.

Programs are known as *software*. A program is a set of instructions that the computer hardware can execute. In the past, system designers mainly use assembly language to write microcontroller application software. The nature of assembly language forces an assembly programmer to work on the program logic at a relatively low level. This hampers programming productivity. In the last 10 years, more and more people have turned to high-level programming languages to improve their programming productivity. C is the most widely used language for embedded system programming.

Although system designers use assembly or high-level languages to write their programs, the microcontroller can only execute machine instructions. Programs written in assembly or high-level languages must be translated into machine instructions before they can be executed. The program that performs the translation work is called an *assembler* or *compiler*, depending on the language to be translated.

A machine instruction consists of *opcode* and *addressing information* that specifies the operands. Addressing information is also called *addressing mode*. The 68HC12 implements a rich instruction set along with many addressing modes for specifying operands. This chapter examines the functions of a few groups of instructions. Examples are used to explore the implementation of simple operations using these instructions.

The execution of an instruction may take several clock cycles. Because the 68HC12 does not access memory in every clock cycle, it performs an instruction prefetch to speed up the instruction execution. A two-word (16-bit word) instruction prefetch queue and a 16-bit buffer are added to hold the prefetched instructions.

1.12 Exercises

- E1.1 What is a processor?
- E1.2 What is a microprocessor? What is a microcomputer?
- E1.3 What makes a microcontroller different from a microprocessor?
- E1.4 How many bits can the 68HC12 CPU manipulate in one operation?
- E1.5 How many different memory locations can the 68HC12 access?
- E1.6 Why must every computer have some amount of nonvolatile memory?
- E1.7 Why must every computer have some amount of volatile memory?
- E1.8 What is source code? What is object code?
- E1.9 Convert 5K, 8K, and 13K to decimal representation.
- E1.10 Write an instruction sequence to swap the contents of memory locations at \$800 and \$801.
- E1.11 Write an instruction sequence to add 10 to memory locations at \$800 and \$801, respectively.
- E1.12 Write an instruction sequence to set the contents of memory locations at \$800, \$810, and \$820 to 10, 11, and 12 respectively.

E1.13 Write an instruction sequence to perform the operations equivalent to those performed by the following high-level language statements:

```
I:= 11;  
J:= 33;  
K = I + J - 5;
```

Assume variables I, J, and K are located at \$800, \$900, and \$1000, respectively.

E1.14 Write an instruction sequence to subtract the number stored at \$810 from that stored at \$800 and store the difference at \$805.

E1.15 Write an instruction sequence to add the contents of accumulator B to the 16-bit word stored at memory location \$800 and \$801. Treat the value stored in B as a signed number.

E1.16 Write an instruction sequence to copy four bytes starting from \$800 to \$900-\$903.

E1.17 Write an instruction sequence to subtract the contents of accumulator B from the 16-bit word at \$800-\$801 and store the difference at \$900-\$901. Treat the value stored in B as a signed value.

E1.18 Write an instruction sequence to swap the 16-bit word stored at \$800-\$801 with the 16-bit word stored at \$900-\$901.

E1.19 Give an instruction that can store the contents of accumulator D at the memory location with an address larger than the contents of X by 8.

E1.20 Give an instruction that can store the contents of index register Y at the memory location with an address smaller than the contents of X by 10.