6

MEMORY AND PROGRAMMABLE LOGIC

6-1 INTRODUCTION

A digital computer consists of three major units: the central processing unit, the memory unit, and the input-output unit. The memory unit is a device to which binary information is transferred for storage and from which information is available when needed for processing. When data processing takes place, information from the memory is first transferred to selected registers in the central processing unit. Intermediate and final results obtained in the central processing unit are transferred back to memory. Binary information received from an input device is first stored in memory and information transferred to an output device is taken from memory. A memory unit is a collection of binary cells which is capable of storing a large quantity of binary information.

There are two types of memories that communicate directly with the central processing unit: *random-access memory* (RAM) and *read-only memory* (ROM). Random-access memory can accept new information for storage to be available later for use. The process of storing new information in memory is referred to as a memory *write* operation. The process of transferring the stored information out of memory is referred to as a memory *read* operation. Random-access memory can perform both the write and read operations. Read-only memory can perform only the read operation. This means that suitable binary information is already stored inside the memory which can be retrieved or read at any time. However, the existing information cannot be altered because read-only memory can only read; it cannot write.

(a) Conventional symbol (b) Array logic symbol

FIGURE 6-1
Conventional and Array Logic Diagrams for OR Gate

Read-only memory is a *programmable logic device*. The binary information that is stored within a programmable logic device must be specified in some fashion and then embedded with the hardware. This process is referred to as *programming* the unit. The word programming here refers to a hardware procedure that specifies the bits that are inserted into the hardware configuration of the device.

Read-only memory (ROM) is one example of a programmable logic device (PLD). Other such units are the programmable logic array (PLA) and the programmable array logic (PAL). A programmable logic device is an integrated circuit with internal logic gates that are connected through electronic fuses. In the original state of the device, all the fuses are intact. Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function. In this chapter we introduce the three programmable logic devices and establish procedures for their use in the design of digital systems.

A typical programmable logic device may have hundreds of gates interconnected through hundreds of internal fuses. In order to show the internal logic diagram in a concise form, it is necessary to employ a special gate symbology applicable to array logic. Figure 6-1 shows the conventional and array symbols for a multiple input OR gate. Instead of having multiple input lines to the gate, we draw a single line to the gate. The input lines are drawn perpendicular to this line and are connected to the gate through internal fuses. In a similar fashion, we can draw the array logic for an AND gate. This type of graphical representation for the inputs of gates will be used throughout this chapter when drawing array logic diagrams.

6-2 RANDOM-ACCESS MEMORY (RAM)

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of the device. Memory cells can be accessed for information transfer to or from any desired random location and hence the name *random-access memory*.

A memory unit stores binary information in groups of bits called *words*. A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction, one or more alphanumeric characters, or any other binary coded information. A group of eight bits is called a *byte*. Most computer memories use words that are multiples of 8 bits in length. Thus, a 16-bit word contains two bytes, and a 32-bit word is made up of four bytes. The capacity of a memory unit is usually stated as the total number of bytes that it can store.

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify

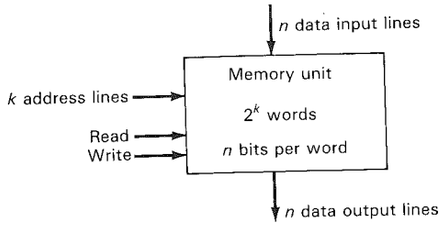


FIGURE 6-2
Block Diagram of Memory Unit

the direction of transfer. A block diagram of the memory unit is shown in Figure 6-2. The n data input lines provide the information to be stored in memory and the n data output lines supply the information coming out of memory. The k address lines specify the particular word chosen among the many available. The two control inputs specify the direction of transfer desired: The write input causes binary data to be transferred into the memory; and the read input causes binary data to be transferred out of memory.

The memory unit is specified by the number of words it contains and the number of bits in each word. The address lines select one particular word. Each word in memory is assigned an identification number called an address. Addresses range from 0 to $2^k - 1$, where k is the number of address lines. The selection of a specific word inside the memory is done by applying the k-bit binary address to the address lines. A decoder inside the memory accepts this address and opens the paths needed to select the word specified. Computer memories may range from 1024 words, requiring an address of 10 bits, to 2^{32} words, requiring 32 address bits. It is customary to refer to the number of words (or bytes) in a memory with one of the letters K (kilo), M (mega), or G (giga). K is equal to 2^{10}, M is equal to 2^{20}, and G is equal to 2^{30}. Thus, $64K = 2^{16}$, $2M = 2^{21}$, and $4G = 2^{32}$.

Consider, for example, the memory unit with a capacity of 1K words of 16 bits each. Since $1K = 1024 = 2^{10}$ and 16 bits constitute two bytes, we can say that the memory can accommodate 2048 or 2K bytes. Figure 6-3 shows the possible content of the first three and the last three words of this memory. Each word contains 16 bits which can be divided into two bytes. The words are recognized by their decimal addresses from 0 to 1023. An equivalent binary address consists of 10 bits. The first address is specified using ten 0's, and the last address is specified with ten 1's. This is because 1023 in binary is equal to 1111111111. A word in memory is selected by its binary address. When a word is read or written, the memory operates on all 16 bits as a single unit.

The 1K × 16 memory of Figure 6-3 has 10 bits in the address and 16 bits in each word. If we have a 64K × 10 memory it is necessary to include 16 bits in the address and each word will consist of 10 bits. The number of address bits needed in a memory is dependent on the total number of words that can be stored in the memory and is independent of the number of bits in each word. The number of bits in the address is determined from the relationship $2^k = m$, where m is the total number of words and k is the number of address bits.

Memory address		
Binary	Decimal	Memory content
0000000000	0	1011010101011100
0000000001	1	1010101110001001
0000000010	2	0000110101000110
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
1111111101	1021	1001110100010101
1111111110	1022	0000110100011110
1111111111	1023	1101111000100100

FIGURE 6-3
Contents of a 1024 × 16 Memory

Write and Read Operations

The two operations that a random-access memory can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into memory are as follows:

1. Transfer the binary address of the desired word to the address lines.
2. Transfer the data bits that must be stored in memory to the data input lines.
3. Activate the *write* input.

The memory unit will then take the bits from the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of memory are as follows:

1. Transfer the binary address of the desired word to the address lines.
2. Activate the *read* input.

The memory unit will then take the bits from the word that has been selected through the address and apply them to the output data lines. The content of the selected word does not change after reading.

Commercial memory components available in integrated circuit chips sometimes provide the two control inputs for reading and writing in a somewhat different configuration. Instead of having separate read and write inputs to control the two operations, some integrated circuits provide two control inputs: one input selects the unit and the other determines the operation. The memory operations that result from these control inputs are shown in Table 6-1.

The memory select (sometimes called chip select) is used to enable the particular memory chip in a multichip implementation of a large memory. When the memory

TABLE 6-1
Control Inputs to Memory Chip

Memory select	Read/write	Memory operation
0	X	None
1	0	Write to selected word
1	1	Read from selected word

select is inactive, the memory chip is not selected and no operation can be performed. When the memory select input is active, the read/write input determines the operation to be performed.

Standard Graphic Symbol

The standard graphic symbol for the RAM is shown in Figure 6-4. The numbers 16×4 that follow the qualifying symbol RAM designate the number of words in the memory and the number of bits per word. The common control block is shown with four address lines and two control inputs. Each bit of the word is shown in a separate section with an input and output data line. The address dependency A is used to identify the address inputs of the memory. Data inputs and outputs affected by the address are labeled with the letter A. The bit grouping from 0 through 3 provide the binary address that ranges from $A0$ through $A15$.

The operation of the memory is specified by means of the dependency notation (see Section 3-8). The RAM graphic symbol uses four dependencies: A (address), G (AND), EN (enable), and C (control). Input $G1$ is to be considered ANDed with $1EN$ and $1C2$ because $G1$ has a 1 after the letter G and the other two each has a 1 in their label. The EN dependency is used to identify an enable input that controls the data outputs. The dependency $C2$ controls the inputs as indicated by the $2D$ label. Thus, for a write operation we have the $G1 - 1C2$ dependency, the

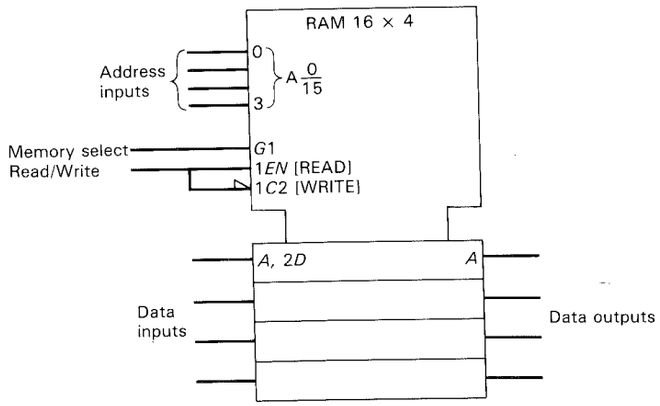


FIGURE 6-4
Standard Graphic Symbol for a 16 × 4 RAM

$C2 - 2D$ dependency, and the $An - A$ dependency, where n is the binary address in the four address inputs. For a read operation we have the $G1 - 1EN$ dependency and the $An - A$ dependency for the outputs. The interpretation of these dependencies results in the operation of the memory as listed in Table 6-1.

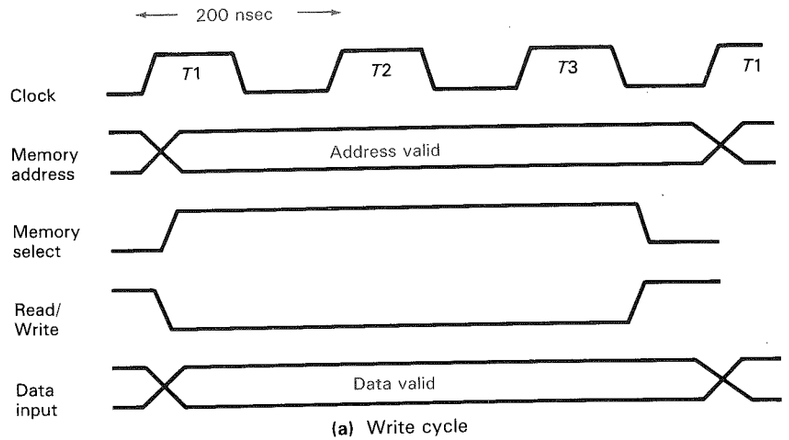
Timing Waveforms

The operation of the memory unit is controlled by an external device such as a central processing unit (CPU). The CPU is usually synchronized with its own clock pulses. The memory, however, does not employ internal clock pulses and its read and write operations are specified by the two control inputs. The *access time* of a memory is the time required to select a word and either read or write it. The CPU must provide the memory control signals in such a way as to synchronize its internal clocked operations with the read and write operations of the memory. This means that the access time of the memory must be within a time period equal to a fixed number of CPU clock pulse periods.

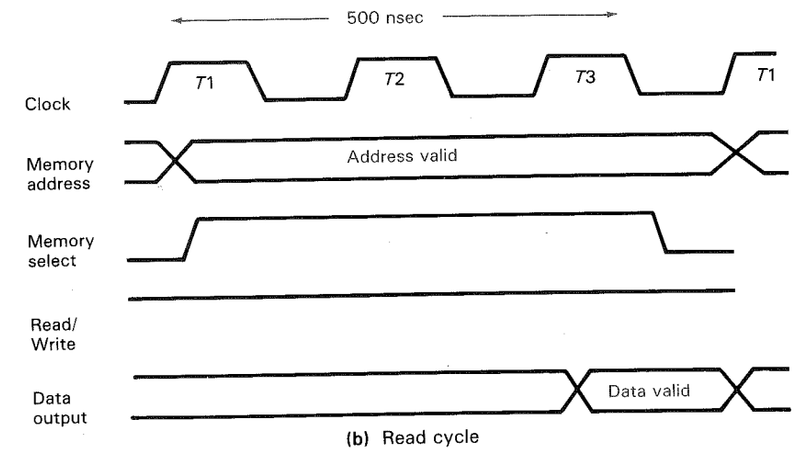
Assume, as an example, that a CPU operates with a clock frequency of 5 MHz, giving a period for one clock pulse of 200 nsec (1 nsec $= 10^{-9}$ seconds). Suppose now that the CPU communicates with a memory with access time that does not exceed 500 nsec. That means that the write cycle terminates the storage of the selected word within a 500 nsec interval, and that the read cycle provides the output data of the selected word within 500 nsec or less. Since the period of the CPU pulse is 200 nsec, it will be necessary to devote at least two and a half (possibly three) clock pulses to each memory request.

The memory cycle timing shown in Figure 6-5 is for a CPU with a 5 MHz clock and a memory with 500 nsec maximum access time. The write cycle in part (a) shows three 200 nsec pulses $T1$, $T2$, and $T3$. For a write operation, the CPU must provide the address and input data to the memory. This is done at the beginning of the $T1$ pulse. The two lines that cross each other in the address and data waveforms designate a possible change in value of the multiple lines. The memory select and the read/write signals must be activated after the signals in the address lines are stable to avoid destroying data in other memory words. The memory select signal switches to the high level and the read/write signal switches to the low level to indicate a write operation. The two control signals must stay active for at least 500 nsec. The address and data signals must remain stable for a short time after the control signals are deactivated. At the completion of the third clock pulse, the memory write operation is completed and the CPU can access the memory again with the next $T1$ pulse.

The read cycle shown in Figure 6-5(b) has an address for the memory which is provided by the CPU. The memory select and the read/write signals must be in their high level for a read operation. The memory places the data of the word selected by the address into the output data lines within a 500 nsec interval (or less) from the time that the memory select is activated. The CPU can transfer the data into one of its internal registers during the negative transition of the $T3$ pulse. The next $T1$ pulse is available for another memory request. Pulse $T1$ is also available for an internal CPU operation that uses the previous memory data word.



(a) Write cycle



(b) Read cycle

FIGURE 6-5
Memory Cycle Timing Waveforms

Types of Memories

The mode of access of a memory system is determined by the type of components used. In a random-access memory, the word locations may be thought of as being separated in space, with each word occupying one particular location. In a sequential-access memory, the information stored in some medium is not immediately accessible but is available only at certain intervals of time. A magnetic tape unit is of this type. Each memory location passes the read and write heads in turn, but information is read out only when the requested word has been reached. In a random-access memory, the access time is always the same regardless of the particular location of the word. In a sequential-access memory, the time it takes to access a word depends on the position of the word with respect to the reading head position and therefore, the access time is variable.

Integrated circuit RAM units are available in two possible operating modes, *static* and *dynamic*. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by *refreshing* the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. Dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip, but static RAM is easier to use and has shorter read and write cycles.

Memory units that lose stored information when power is turned off are said to be *volatile*. Integrated circuit RAMs, both static and dynamic, are of this category since the binary cells need external power to maintain the stored information. In contrast, a nonvolatile memory, such as magnetic disk, retains its stored information after removal of power. This is because the data stored on magnetic components is manifested by the direction of magnetization, which is retained after power is turned off. Another nonvolatile memory is the ROM discussed in Section 6-5.

A nonvolatile property is desirable in digital computers to store programs that are needed while the computer is in operation. Programs and data that cannot be altered are stored in ROM. Other large programs are maintained on magnetic disks. When power is turned on, the computer can use the programs from ROM. The other programs residing on the disks can be transferred into the computer RAM as needed. Before turning the power off, the user transfers the binary information from the computer RAM into a disk if this information must be retained.

6-3 MEMORY DECODING

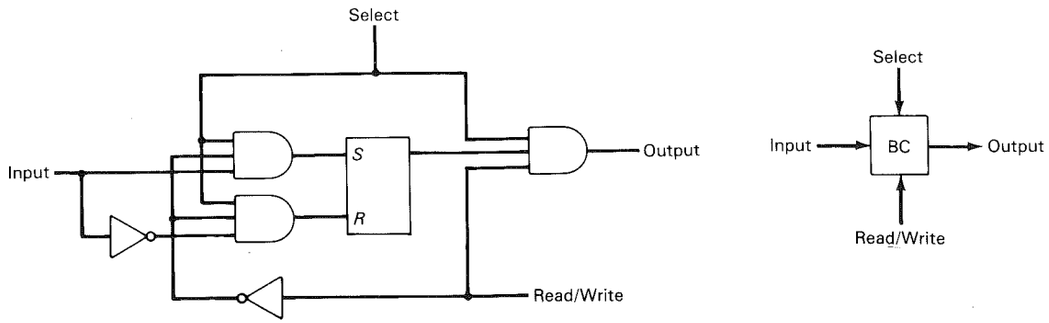
In addition to the storage components in a memory unit, there is a need for decoding circuits to select the memory word specified by the input address. In this section we present the internal construction of a random-access memory and demonstrate the operation of the decoder. To be able to include the entire memory in one diagram, the memory unit presented here has a small capacity of 16 bits arranged in 4 words of 4 bits each. We then present a two-dimensional coincident decoding arrangement to show a more efficient decoding scheme which is sometimes used in large memories.

In addition to internal decoders, a memory unit may also need external decoders. This happens when integrated-circuit RAM chips are connected in a multichip memory configuration. The use of an external decoder to provide a large capacity memory will be demonstrated by means of an example.

Internal Construction

The internal construction of a random-access memory of m words and n bits per word consists of $m \times n$ binary storage cells and associated decoding circuits for selecting individual words. The binary storage cell is the basic building block of a memory unit. The equivalent logic of a binary cell that stores one bit of information is shown in Figure 6-6. Although the cell is shown to include gates and a flip-flop, internally it is constructed with two transistors having multiple inputs. A binary storage cell must be very small so that as many cells as possible can be packed into the small area available in the integrated circuit chip. The binary cell stores one bit in its internal flip-flop. The select input enables the cell for reading or writing and the read/write input determines the cell operation when it is selected. A 1 in the read/write input provides the read operation by forming a path from the flip-flop to the output terminal. A 0 in the read/write input provides the write operation by forming a path from the input terminal to the flip-flop. Note that the flip-flop operates without a clock and is similar to an SR latch.

The logical construction of a small RAM is shown in Figure 6-7. It consists of four words of four bits each and has a total of 16 binary cells. Each block labeled BC represents a binary cell with its three inputs and one output as specified in Figure 6-6(b). A memory with four words needs two address lines. The two address inputs go through a 2×4 decoder to select one of the four words. The decoder is enabled with the memory select input. When the memory select is 0, all outputs of the decoder are 0 and none of the memory words are selected. With the memory select at 1, one of the four words is selected, dictated by the value in the two address lines. Once a word has been selected, the read/write input determines the operation. During the read operation, the four bits of the selected word go through OR gates to the output terminals. (Note that the OR gates are drawn according to the array logic established in Figure 6-1.) During the write operation, the data available in the input lines are transferred into the four binary cells of the selected word. The binary cells that are not selected are disabled and their previous binary values remain unchanged. When the memory select input that goes into the decoder



(a) Logic diagram (b) Block diagram

FIGURE 6-6
Memory Cell

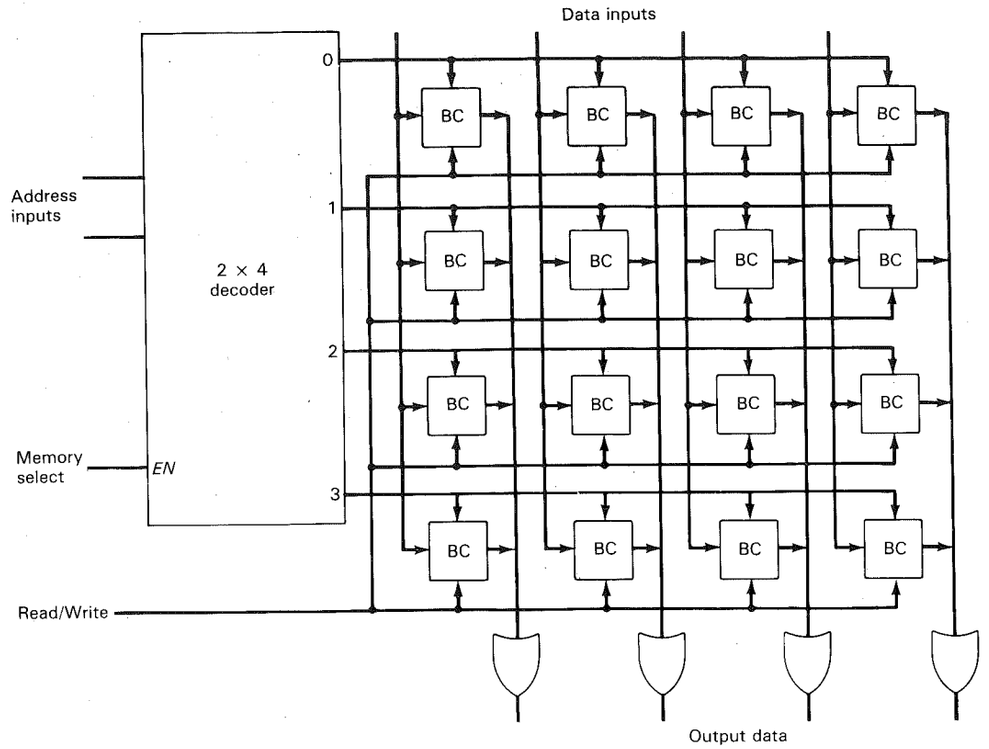


FIGURE 6-7
Diagram of a 4 × 4 RAM

is equal to 0, none of the words are selected and the contents of all cells remain unchanged regardless of the value of the read/write input.

Commercial random-access memories may have a capacity of thousands of words and each word may range from 1 to 64 bits. The logical construction of a large capacity memory would be a direct extension of the configuration shown here. A memory with 2^k words of n bits per word requires k address lines that go into a $k \times 2^k$ decoder. Each one of the decoder outputs selects one word of n bits for reading or writing.

Coincident Decoding

A decoder with k inputs and 2^k outputs requires 2^k AND gates with k inputs per gate. The total number of gates and the number of inputs per gate can be reduced by employing two decoders with a coincident selection scheme. In this configuration, two $k/2$-input decoders are used instead of one k-input decoder. One decoder performs the horizontal X-selection and the other the vertical Y-selection in a two-dimensional matrix selection scheme.

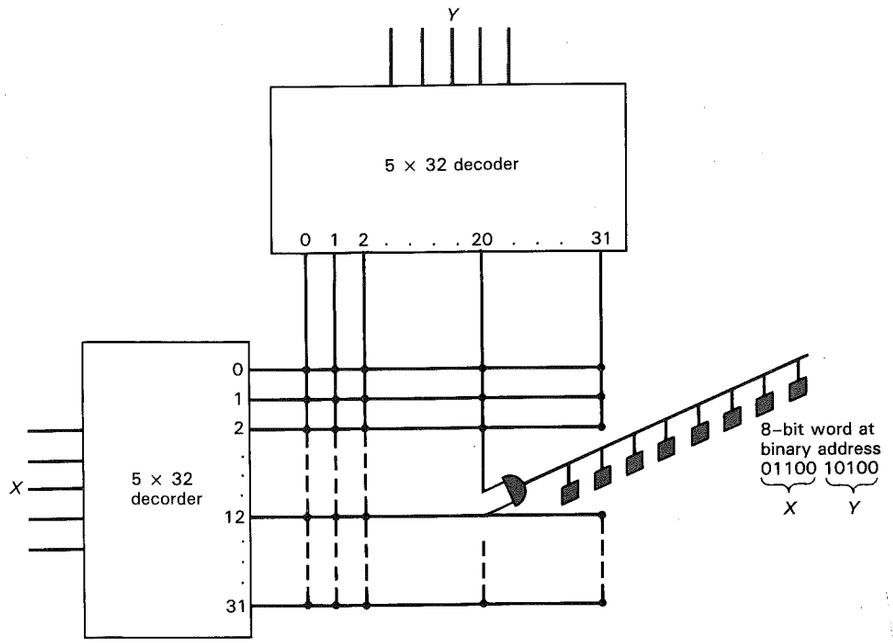


FIGURE 6-8
Two Dimensional Decoding Structure for a 1K × 8 RAM

The coincident selection pattern is demonstrated in Figure 6-8 for a 1K × 8 memory. Instead of using a single 10 × 1024 decoder, we use two 5 × 32 decoders. With the single decoder we would need 1024 AND gates with 10 inputs in each. With two decoders we only need 64 AND gates with five inputs in each. The five most significant bits of the address go to input X and the five least significant bits go to input Y. Each word of eight bits within the memory array is selected by the coincidence of one X line and one Y line. Thus, each word in memory is selected by the coincidence between one of 32 horizontal lines and one of 32 vertical lines for a total of 1024 words. As an example, consider the word with the address 404. The 10-bit binary equivalent of 404 is 01100 10100. This makes $X = 01100$ (binary 12) and $Y = 10100$ (binary 20). The 8-bit word that is selected lies in the X decoder output number 12 and the Y decoder output number 20. All eight bits of the word are selected simultaneously for reading or writing.

Array of RAM Chips

Integrated circuit RAM chips are available in a variety of sizes. If the memory unit needed for an application is larger than the capacity of one chip, it is necessary to combine a number of chips in an array to form the required memory size. The capacity of the memory depends on two parameters: the number of words and the number of bits per word. An increase in the number of words requires that we increase the address length. Every bit added to the length of the address doubles

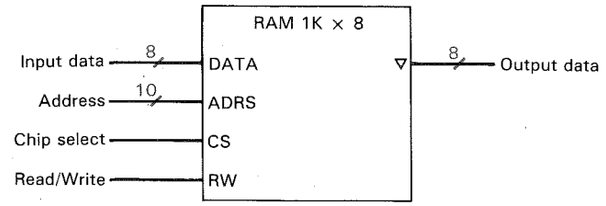


FIGURE 6-9
Block Diagram of a 1K × 8 RAM Chip

the number of words in memory. The increase in the number of bits per word requires that we increase the length of the data input and output lines, but the address length remains the same.

To demonstrate with an example, let us first introduce a typical RAM chip as shown in Figure 6-9. The capacity of the RAM is 1024 words of 8 bits each. It requires a 10-bit address and 8 input and output lines. These are shown in the block diagram by a single line and a number indicating the total number of inputs or outputs. The *CS* (chip select) input selects the particular RAM chip and the *RW* (read/write) input specifies the read or write operation when the chip is selected. The triangle symbol shown with the outputs is the standard graphic symbol for a *three-state* output. A three-state output may be in one of three possible states: a signal equivalent to binary 0, a signal equivalent to binary 1, or a high-impedance state. The high-impedance state behaves like an open circuit: it does not carry any signal and does not have a logic significance. The *CS* input of the RAM controls the behavior of the data output lines. When $CS = 0$, the chip is not selected and all its data outputs are in the high-impedance state. With $CS = 1$ and $RW = 1$, the data output lines carry the eight bits of the selected word.

Suppose that we want to increase the number of words in the memory by using two or more RAM chips. Since every bit added to the address doubles the binary number that can be formed, it is natural to increase the number of words in factors of two. For example, two RAM chips will double the number of words and add one bit to the composite address. Four RAM chips multiply the number of words by four and add two bits to the composite address.

Consider the possibility of constructing a 4K × 8 RAM with four 1K × 8 RAM chips. This is shown in Figure 6-10. The 8 input data lines go to all the chips. The three-state outputs can be connected together to form the common 8 output data lines. This type of output connection is possible only with three-state outputs. This is because only one chip select input will be active at any time while the other three chips will be disabled. The 8 outputs of the selected chip will contain 1's and 0's, and the other three will be in a high-impedance state with no logic significance to disturb the output binary signals of the selected chip.

The 4K word memory requires a 12-bit address. The 10 least significant bits of the address are applied to the address inputs of all four chips. The other two most significant bits are applied to a 2 × 4 decoder. The four outputs of the decoder are applied to the *CS* inputs of each chip. The memory is disabled when the enable input of the decoder is equal to 0. This causes all four outputs of the decoder to

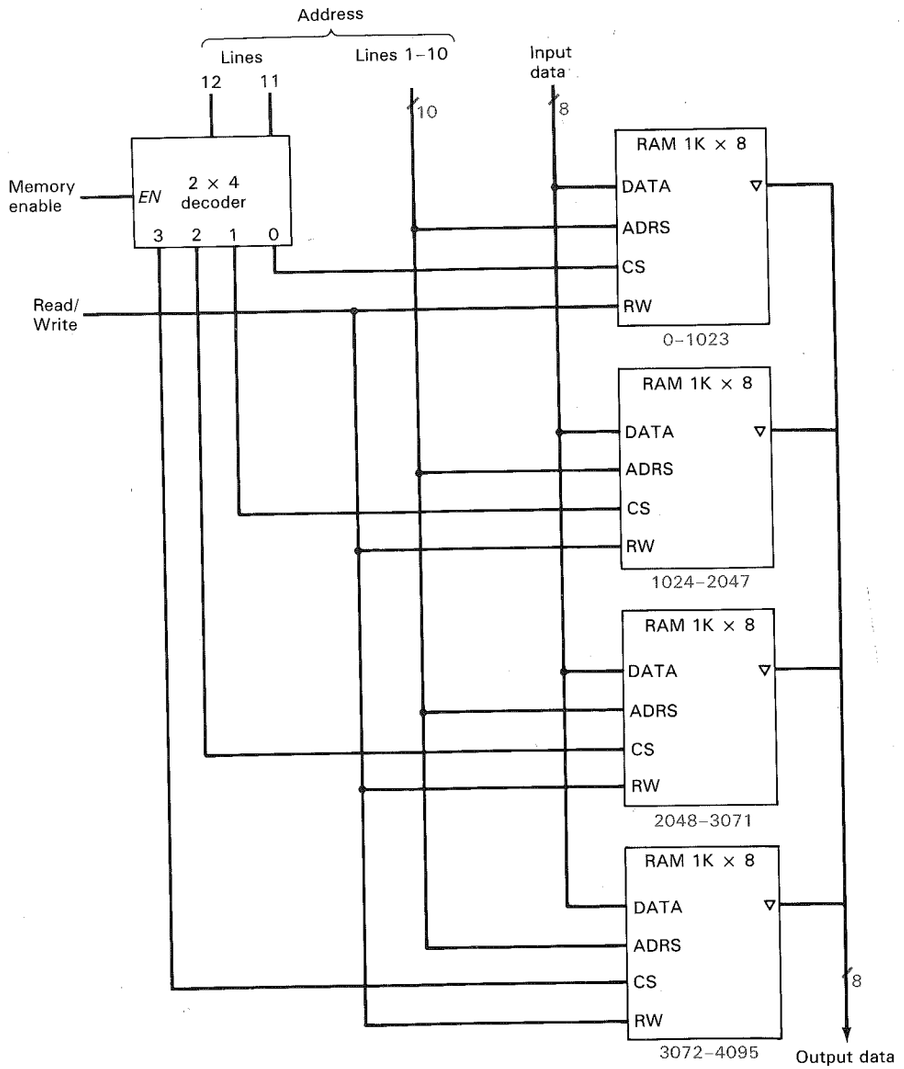


FIGURE 6-10
Block Diagram of 4K × 8 RAM

be in the 0 state and none of the chips are selected. When the decoder is enabled, address bits 12 and 11 determine the particular chip that is selected. If bits 12 and 11 are equal to 00, the first RAM chip is selected. The remaining ten address bits select a word within the chip in the range from 0 to 1023. The next 1024 words are selected from the second RAM chip with a 12-bit address that starts with 01 and follows by the ten bits from the common address lines. The address range for each chip is listed in decimal under its block diagram in Figure 6-10.

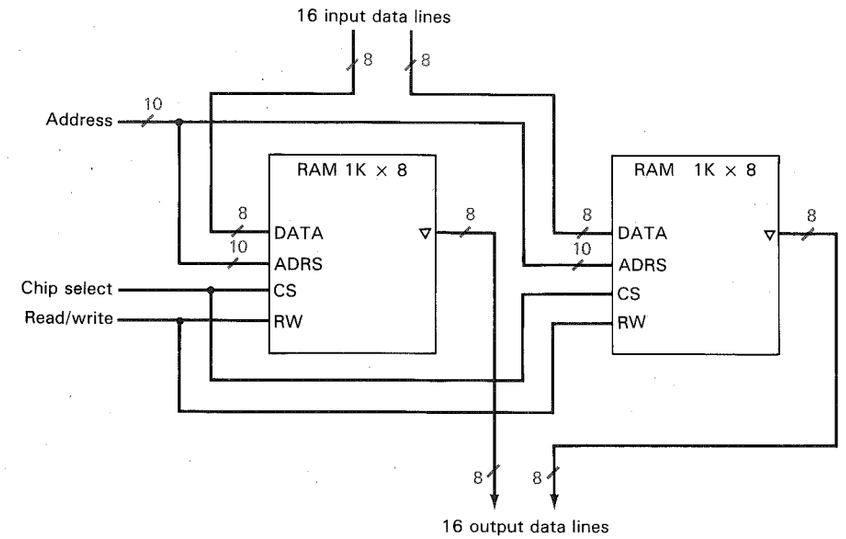


FIGURE 6-11
Block Diagram of 1K × 16 RAM

It is also possible to combine two chips to form a composite memory containing the same number of words but with twice as many bits in each word. Figure 6-11 shows the interconnection of two 1K × 8 chips to form a 1K × 16 memory. The 16 input and output data lines are split between the two chips. Both receive the same 10-bit address and the common *CS* and *RW* control inputs.

The two techniques just described may be combined to assemble an array of identical chips into a large capacity memory. The composite memory will have a number of bits per word that is a multiple of that for one chip. The total number of words will increase in factors of two times the word capacity of one chip. An external decoder is needed to select the individual chips from the additional address bits of the composite memory.

To reduce the number of pins in the package, many RAM integrated circuits provide common terminals for the input data and output data. The common terminals are said to be *bidirectional* which means that for the read operation they act as outputs and for the write operation they act as inputs. Bidirectional lines are constructed with three-state buffers and are discussed further in Section 7-4.

6-4 ERROR DETECTION AND CORRECTION

The complexity level of a memory array may cause occasional errors in storing and retrieving the binary information. The reliability of a memory unit may be improved by employing error-detecting and correcting codes. The most common error-detection scheme is the parity bit. (See Section 2-7.) A parity bit is generated and stored along with the data word in memory. The parity of the word is checked after reading it from memory. The data word is accepted if the parity sense is correct.

If the parity checked results in an inversion, an error is detected but it cannot be corrected.

An error-correcting code generates multiple check bits that are stored with the data word in memory. Each check bit is a parity over a group of bits in the data word. When the word is read back from memory, the associated parity bits are also read and compared with a new set of check bits generated from the read data. If the check bits compare, it signifies that no error has occurred. If the check bits do not compare with the stored parity, they generate a unique pattern called a *syndrome* that can be used to identify the bit in error. A single error occurs when a bit changes in value from 1 to 0 or from 0 to 1 during the write or read operation. If the specific bit in error is identified, then the error can be corrected by complementing the erroneous bit.

Hamming Code

One of the most common error-correcting codes used in random-access memories was devised by R. W. Hamming. In the Hamming code, k parity bits are added to an n-bit data word, forming a new word of $n + k$ bits. The bit positions are numbered in sequence from 1 to $n + k$. Those positions numbered as a power of 2 are reserved for the parity bits. The remaining bits are the data bits. The code can be used with words of any length. Before giving the general characteristics of the code, we will illustrate its operation with a data word of eight bits.

Consider for example the 8-bit data word 11000100. We include 4 parity bits with the 8-bit word and arrange the 12 bits as follows:

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
	P_1	P_2	1	P_4	1	0	0	P_8	0	1	0	0

The four parity bits P_1 through P_8 are in positions 1, 2, 4, and 8. The eight bits of the data word are in the remaining positions. Each parity bit is calculated as follows:

$$P_1 = \text{XOR of bits } (3, 5, 7, 9, 11) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 0 = 0$$

$$P_2 = \text{XOR of bits } (3, 6, 7, 10, 11) = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0$$

$$P_4 = \text{XOR of bits } (5, 6, 7, 12) = 1 \oplus 0 \oplus 0 \oplus 0 = 1$$

$$P_8 = \text{XOR of bits } (9, 10, 11, 12) = 0 \oplus 1 \oplus 0 \oplus 0 = 1$$

Remember that the exclusive-OR operation performs the odd function. It is equal to 1 for an odd number of 1's in the variables and to 0 for an even number of 1's. Thus, each parity bit is set so that the total number of 1's in the checked positions, including the parity bit, is always even.

The 8-bit data word is stored in memory with the 4 parity bits as a 12-bit composite word. Substituting the four P bits in their proper positions, we obtain the 12-bit composite word stored in memory.

Bit position	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	1	1	1	0	0	1	0	1	0	0

When the 12 bits are read from memory they are checked again for possible errors. The parity is checked over the same combination of bits including the parity bit. The four check bits are evaluated as follows:

$$C_1 = \text{XOR of bits } (1, 3, 5, 7, 9, 11)$$

$$C_2 = \text{XOR of bits } (2, 3, 6, 7, 10, 11)$$

$$C_4 = \text{XOR of bits } (4, 5, 6, 7, 12)$$

$$C_8 = \text{XOR of bits } (8, 9, 10, 11, 12)$$

A 0 check bit designates a even parity over the checked bits and a 1 designates an odd parity. Since the bits were stored with even parity, the result $C = C_8C_4C_2C_1 = 0000$ indicates that no error has occurred. However, if $C \neq 0$, the 4-bit binary number formed by the check bits gives the position of the erroneous bit. For example, consider the following three cases:

Bit Position	1	2	3	4	5	6	7	8	9	10	11	12	
	0	0	1	1	1	0	0	1	0	1	0	0	No error
	1	0	1	1	1	0	0	1	0	1	0	0	Error in bit 1
	0	0	1	1	0	0	0	1	0	1	0	0	Error in bit 5

In the first case, there is no error in the 12-bit word. In the second case there is an error in bit position number 1 because it changed from 0 to 1. The third case shows an error in bit position 5 with a change from 1 to 0. Evaluating the XOR of the corresponding bits we determine the four check bits to be as follows:

	C_8	C_4	C_2	C_1
No error	0	0	0	0
Error in bit 1	0	0	0	1
Error in bit 5	0	1	0	1

Thus, for no error we have $C = 0000$; with an error in bit 1 we obtain $C = 0001$; and with an error in bit 5 we get $C = 0101$. The binary number of C, when it is not equal to 0000, gives the position of the bit in error. The error can be corrected by complementing the corresponding bit. Note that an error can occur in the data word or in one of the parity bits.

The Hamming code can be used for data words of any length. In general, for k check bits and n data bits, the total number of bits $(n + k)$ that can be accommodated in a coded word is $2^k - 1$. In other words, the relationship $n + k = 2^k - 1$ must hold. This gives $n = 2^k - 1 - k$ (or less) as the number of bits for the data word. For example, when $k = 3$, the total number of bits in the coded word is $n + k = 2^3 - 1 = 7$, giving $n = 7 - 3 = 4$. For $k = 4$, we have $n + k = 15$, giving $n = 11$. The data word may be less than 11 bits but must have at least 5 bits; otherwise, only three check bits will be needed. This justifies the use of 4 check bits for the 8 data bits in the previous example.

The grouping of bits for parity generation and checking can be determined from a list of the binary numbers from 0 through $2^k - 1$. (Table 1-2 gives such a list).

The least significant bit is a 1 in the binary numbers 1, 3, 5, 7, and so on. The second significant bit is a 1 in the binary numbers 2, 3, 6, 7, and so on. Comparing these numbers with the bit positions used in generating and checking parity bits in the Hamming code, we note the relationship between the bit groupings in the code and the position of the 1-bits in the binary count sequence. Note that each group of bits starts with a number that is a power of 2, for example, 1, 2, 4, 8, 16, and so forth. These numbers are also the position number for the parity bits.

The Hamming code can detect and correct only a single error. Multiple errors are not detected. By adding another parity bit to the coded word, the Hamming code can be used to correct a single error and detect double errors. If we include this additional parity bit, the previous 12-bit coded word becomes $001110010100P_{13}$ where P_{13} is evaluated from the exclusive-OR of the other 12 bits. This produces the 13-bit word 0011100101001 (even parity). When the 13-bit word is read from memory, the check bits are evaluated and also the parity P over the entire 13 bits. If $P = 0$, the parity is correct (even parity), but if $P = 1$, the parity over the 13 bits is incorrect (odd parity). The following four cases can occur:

If $C = 0$ and $P = 0$	No error occurred
If $C \neq 0$ and $P = 1$	A single error occurred which can be corrected.
If $C \neq 0$ and $P = 0$	A double error occurred which is detected but cannot be corrected.
If $C = 0$ and $P = 1$	An error occurred in the P_{13} bit.

Note that this scheme cannot detect more than two errors.

Integrated circuits are available commercially that use a modified Hamming code to generate and check parity bits for a single-error-correction double-error-detection scheme. One that uses an 8-bit data word and a 5-bit check word is IC type 74637. Other integrated circuits are available for data words of 16 and 32 bits. These circuits can be used in conjunction with a memory unit to correct a single error or detect double errors during the write or read operations.

6-5 READ-ONLY MEMORY (ROM)

A read-only memory (ROM) is essentially a memory device in which permanent binary information is stored. The binary information must be specified by the designer and is then embedded in the unit to form the required interconnection pattern. ROMs come with special internal electronic fuses that can be "programmed" for a specific configuration. Once the pattern is established, it stays within the unit even when power is turned off and on again.

A block diagram of a ROM is shown in Figure 6-12. It consists of k inputs and n outputs. The inputs provide the address for the memory and the outputs give

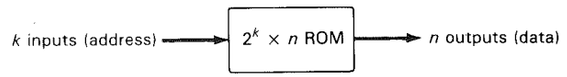


FIGURE 6-12
ROM Block Diagram

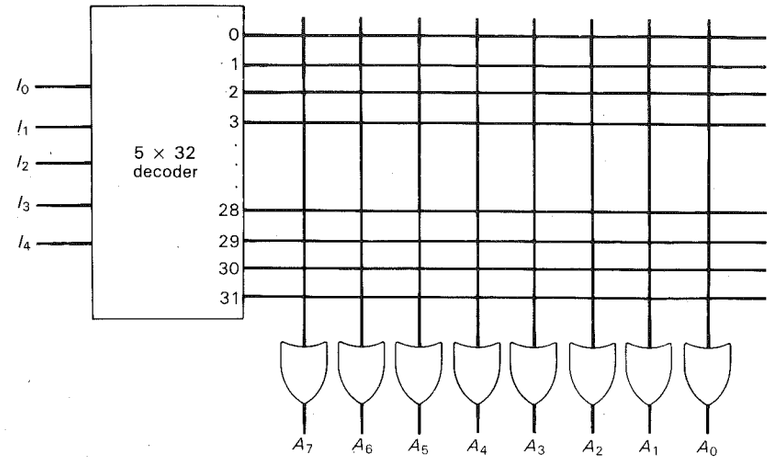


FIGURE 6-13
Internal Logic of a 32 × 8 ROM

the data bits of the stored word which is selected by the address. The number of words in a ROM is determined from the fact that k address input lines are needed to specify 2^k words. Note that the ROM does not have data inputs because it does not have a write operation. Integrated circuit ROM chips have one or more enable inputs and come with three-state outputs to facilitate the construction of large arrays of read-only memories.

Consider for example a 32 × 8 ROM. The unit consists of 32 words of 8 bits each. There are 5 input lines that form the binary numbers from 0 through 31 for the address. Figure 6-13 shows the internal logic construction of the ROM. The five inputs are decoded into 32 distinct outputs by means of a 5 × 32 decoder. Each output of the decoder represents a memory address. The 32 outputs of the decoder are connected through fuses to each of the eight OR gates. The diagram shows the array logic convention used in complex circuits (see Figure 6-1). Each OR gate must be considered as having 32 inputs. Each output of the decoder is connected through a fuse to one of the inputs of each OR gate. Since each OR gate has 32 internal fuses and there are 8 OR gates, the ROM contains $32 \times 8 = 256$ internal fuselinks. In general, a $2^k \times n$ ROM will have an internal $k \times 2^k$ decoder and n OR gates. Each OR gate has 2^k inputs which are connected through fuses to each of the outputs of the decoder.

The internal binary storage of a ROM is specified by a truth table that shows the word content in each address. For example, the content of a 32 × 8 ROM may be specified with a truth table similar to the one shown in Table 6-2. The truth table shows the five inputs under which are listed all 32 addresses. Each input specifies the address of a word of 8 bits whose value is listed under the output columns. Table 6-2 shows only the first four and the last four words in the ROM. The complete table must include the list of all 32 words.

The hardware procedure that programs the ROM results in blowing internal fuses according to a given truth table. For example, programming the ROM ac-

TABLE 6-2
ROM Truth Table (Partial)

Inputs					Outputs							
I_4	I_3	I_2	I_1	I_0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	1	0	0	0	1	1	1	0	1
0	0	0	1	0	1	1	0	0	0	1	0	1
0	0	0	1	1	1	0	1	1	0	0	1	0
		⋮						⋮				
1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	1	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	1	0	1	0
1	1	1	1	1	0	0	1	1	0	0	1	1

cording to the truth table given by Table 6-2 results in the configuration shown in Figure 6-14. Every 0 listed in the truth table specifies a fuse to be blown and every 1 listed specifies a path that is obtained by an intact fuse. As an example, the table specifies the 8-bit word 10110010 for permanent storage at input address 00011. The four 0's in the word are programmed by blowing the fuses between output 3 of the decoder and the inputs of the OR gates associated with outputs A_6, A_3, A_2, and A_0. The four 1's in the word are marked in the diagram with a cross to designate an intact fuse. When the input of the ROM is 00011, all the outputs of the decoder are 0 except output 3 which is at logic-1. The signal equivalent to logic-1 at decoder output 3 propagates through the fuses and the OR gates to outputs A_7, A_5, A_4,

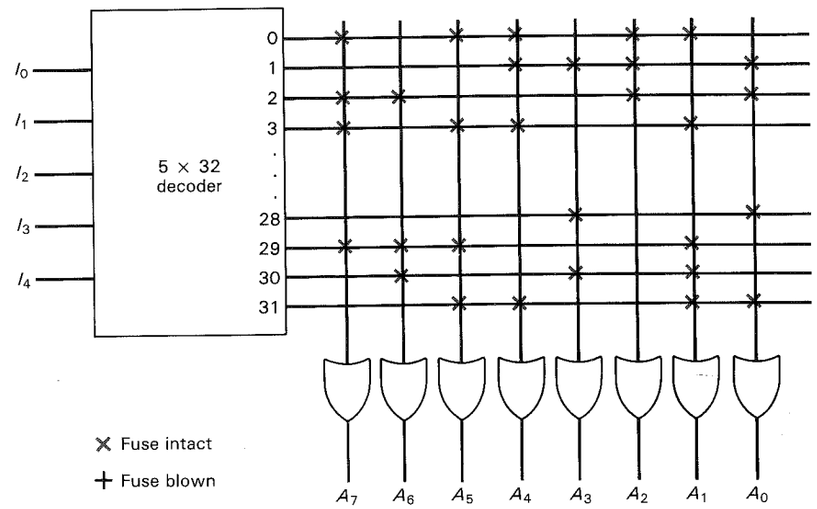


FIGURE 6-14
Programming the ROM According to Table 6-2

and A_1. The other four outputs remain at 0. The result is that the stored word 10110010 is applied to the eight data outputs.

Types of ROMs

The required paths in a ROM may be programmed in three different ways. The first is called *mask programming* and is done by the semiconductor company during the last fabrication process of the unit. The procedure for fabricating a ROM requires that the customer fill out the truth table he wishes the ROM to satisfy. The truth table may be submitted on a special form provided by the manufacturer or in a specified format on a computer output medium. The manufacturer makes the corresponding mask for the paths to produce the 1's and 0's according to the customer's truth table. This procedure is costly because the vendor charges the customer a special fee for custom masking the particular ROM. For this reason, mask programming is economical only if a large quantity of the same ROM configuration is to be ordered.

For small quantities, it is more economical to use a second type of ROM called a *programmable read-only memory* or PROM. When ordered, PROM units contain all the fuses intact giving all 1's in the bits of the stored words. The fuses in the PROM are blown by application of current pulses through the output terminals for each address. A blown fuse defines a binary 0 state and an intact fuse gives a binary 1 state. This allows the user to program the PROM in his own laboratory to achieve the desired relationship between input addresses and stored words. Special instruments called *PROM programmers* are available commercially to facilitate this procedure. In any case, all procedures for programming ROMs are hardware procedures even though the word programming is used. The hardware procedure for programming ROMs or PROMs is irreversible and, once programmed, the fixed pattern is permanent. Once a bit pattern has been established, the unit must be discarded if the bit pattern is to be changed.

A third type of ROM available is called *erasable PROM* or EPROM. The EPROM can be restructured to the initial value even though its fuses have been blown previously. When the EPROM is placed under a special ultraviolet light for a given period of time, the short wave radiation discharges the internal gates that serve as fuses. After erasure, the EPROM returns to its initial state and can be reprogrammed to a new set of words. Certain PROMs can be erased with electrical signals instead of ultraviolet light. These PROMs are called *electrically erasable PROM* or EEPROM.

Combinational Circuit Implementation

It was shown in Section 3-5 that a decoder generates the 2^k minterms of the k input variables. By inserting OR gates to sum the minterms of Boolean functions, we were able to generate any desired combinational circuit. The ROM is essentially a device that includes both the decoder and the OR gates within a single unit. By leaving intact the fuses of those minterms that are included in the function, the ROM outputs can be programmed to represent the Boolean functions of the output variables in a combinational circuit.

The internal operation of a ROM can be interpreted in two ways. The first interpretation is that of a memory unit that contains a fixed pattern of stored words. The second interpretation is of a unit that implements a combinational circuit. From this point of view, each output terminal is considered separately as the output of a Boolean function expressed as a sum of minterms. For example, the ROM of Figure 6-14 may be considered as a combinational circuit with eight outputs, each being a function of the five input variables. Output A_7 can be expressed as a sum of minterms as follows. (The three dots represent minterms 4 through 27 which are not specified in the figure.)

$$A_7(I_4, I_3, I_2, I_1, I_0) = \Sigma\, m(0, 2, 3, \ldots, 29)$$

An intact fuse produces a minterm for the sum and a blown fuse removes the minterm from the sum.

ROMs are widely used to implement complex combinational circuits directly from their truth table. They are useful for converting from one alphanumeric code, such as ASCII, to another, like EBCDIC. They can generate complex arithmetic operations such as multiplication or division, and in general, they are used in applications requiring a large number of inputs and outputs.

In practice, when a combinational circuit is designed by means of a ROM, it is not necessary to design the logic or to show the internal gate connections of fuses inside the unit. All that the designer has to do is specify the particular ROM by its IC number and provide the ROM truth table. The truth table gives all the information for programming the ROM. No internal logic diagram is needed to accompany the truth table.

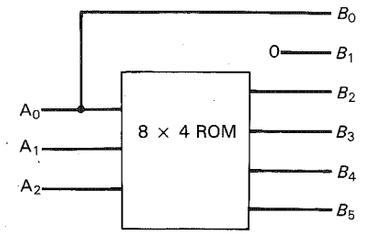
Example 6-1

Design a combinational circuit using a ROM. The circuit accepts a 3-bit number and generates an output binary number equal to the square of the input number.

The first step is to derive the truth table of the combinational circuit. In most cases this is all that is needed. In other cases, we can use a partial truth table for the ROM by utilizing certain properties in the output variables. Table 6-3 is the truth table for the combinational circuit. Three inputs and six outputs are needed to accommodate all possible binary numbers. We note that output B_0 is always

TABLE 6-3
Truth Table for Circuit of Example 6-1

Inputs			Outputs						Decimal
A_2	A_1	A_0	B_5	B_4	B_3	B_2	B_1	B_0	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	1	1
0	1	0	0	0	0	1	0	0	4
0	1	1	0	0	1	0	0	1	9
1	0	0	0	1	0	0	0	0	16
1	0	1	0	1	1	0	0	1	25
1	1	0	1	0	0	1	0	0	36
1	1	1	1	1	0	0	0	1	49



(a) Block diagram

A_2	A_1	A_0	B_5	B_4	B_3	B_2
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	1
0	1	1	0	0	1	0
1	0	0	0	1	0	0
1	0	1	0	1	1	0
1	1	0	1	0	0	1
1	1	1	1	1	0	0

(b) ROM truth table

FIGURE 6-15
ROM Implementation of Example 6-1

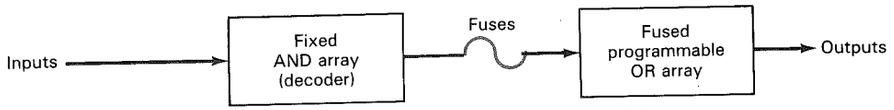
equal to input A_0; so there is no need to generate B_0 with a ROM. Moreover, output B_1 is always 0; so this output is a known constant. We actually need to generate only four outputs with the ROM; the other two are readily obtained. The minimum size ROM needed must have three inputs and four outputs. Three inputs specify eight words; so the ROM must be of size 8 × 4. The ROM implementation is shown in Figure 6-15. The three inputs specify eight words of four bits each. The truth table in Figure 6-15(b) specifies the information needed for programming the ROM. The block diagram of Figure 6-15(a) shows the required connections of the combinational circuit. ■

The previous example is too simple for actual implementation with a ROM. It is presented here for illustration purposes only. The design of complex combinational circuits with a ROM requires that we determine the number of inputs and outputs of the circuit. The size of ROM needed to implement the circuit is at least $2^k \times n$, where k is the number of inputs and n is the number of outputs. The truth table accompanying the ROM must list the 2^k binary addresses and provide the n-bit word for each address.

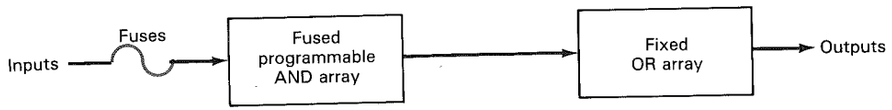
6-6 PROGRAMMABLE LOGIC DEVICE (PLD)

A programmable logic device (PLD) is an integrated circuit with an array of gates that are connected by programming fuses. The designer can specify the internal logic by means of a table or a list of Boolean functions. These specifications are then translated into a fuse pattern required to program the device. The gates in a PLD are divided into an AND array and an OR array to provide an AND-OR sum of products implementation. The initial state of a PLD has all the fuses intact. Programming the device involves the blowing of internal fuses to achieve a desired logic function.

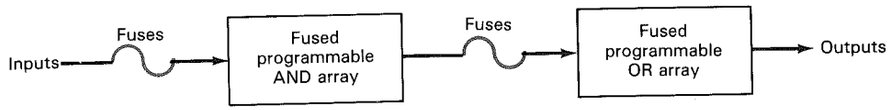
There are three major types of PLDs and they differ in the placement of fuses in the AND-OR array. Figure 6-16 shows the fuse locations of the three PLDs. The programmable read-only memory (PROM) has a fixed AND array constructed as a decoder and programmable fuses for the output OR gates. The PROM implements Boolean functions in sum of minterms. The programmable array logic (PAL) has a fused programmable AND array and a fixed OR array. The AND



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL)



(c) Programmable logic array (PLA)

FIGURE 6-16
Basic Configuration of Three PLDs

gates are programmed to provide the product terms for the Boolean functions which are logically summed in each OR gate. The most flexible PLD is the programmable logic array (PLA) where both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum of products implementation. The names PAL and PLA emerged from different vendors during the development of programmable logic devices. The implementation of combinational circuits with a PROM was demonstrated in the previous section. The design of combinational circuits with PLA and PAL is presented in the next two sections.

Some PLDs include flip-flops within the integrated circuit chip in addition to the AND and OR arrays. The result is a sequential circuit as shown in Figure 6-17. A PAL or a PLA may include a number of flip-flops connected by fuses to form the sequential circuit. The circuit outputs can be taken from the OR gates or from the outputs of the flip-flops. Additional programmable fuses are available to include the flip-flop variables in the product terms formed with the AND array. The flip-flops may be of the D or the JK type.

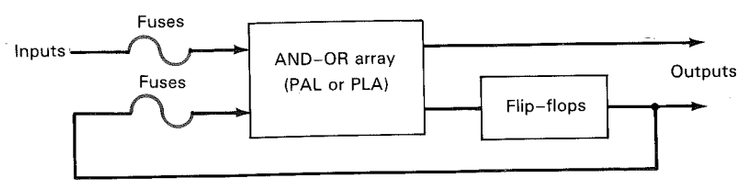


FIGURE 6-17
Sequential Programmable Logic Device

The advantage of using the PLD in the design of digital systems is that it can be programmed to incorporate a complex logic function within one integrated circuit. The use of programmable logic devices is an alternative to another design technology called VLSI (very large scale integration) design. VLSI design refers to the design of digital systems that contain thousands of gates within a single integrated circuit chip. The basic component used in VLSI design is the *gate array*. A gate array consists of a pattern of gates fabricated in an area of silicon which is repeated thousands of times until the entire chip is covered with identical gates. Arrays of one thousand to ten thousand gates can be fabricated within a single integrated-circuit chip depending on the technology used. The design with gate arrays requires that the designer specifies the layout of the chip and the way that the gates are routed and connected. The first few levels of fabrication process are common and independent of the final logic function. Additional fabrication levels are required to interconnect the gates in order to realize the desired function. This is usually done by means of computer aided design (CAD) methods. Both the gate array and the programmable logic device require extensive computer software tools to facilitate the design procedure.

6-7 PROGRAMMABLE LOGIC ARRAY (PLA)

The PLA is similar to the PROM in concept except that the PLA does not provide full decoding of the variables and does not generate all the minterms. The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product terms are then connected to OR gates to provide the sum of products for the required Boolean functions.

The internal logic of a PLA with three inputs and two outputs is shown in Figure 6-18. Such a circuit is too small to be available commercially but is presented here to demonstrate the typical logic configuration of a PLA. The diagram uses the array logic graphic symbols for complex circuits. Each input goes through a buffer and an inverter shown in the diagram with a composite graphic symbol that has both the true and complement outputs. Each input and its complement are connected through fuses to the inputs of each AND gate as indicated by the intersections between the vertical and horizontal lines. The outputs of the AND gates are connected by fuses to the inputs of each OR gate. The output of the OR gate goes to an XOR gate where the other input can be programmed to receive a signal equal to either logic-1 or logic-0. The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = \overline{X}$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$). The particular Boolean functions implemented in the PLA of Figure 6-18 are

$$F_1 = A\overline{B} + AC + \overline{A}B\overline{C}$$

$$\overline{F}_2 = AC + BC$$

The product terms generated in each AND gate are listed along the output of the gate in the diagram. The product term is determined from the inputs with intact fuses. The output of an OR gate gives the logic sum of the selected product terms.

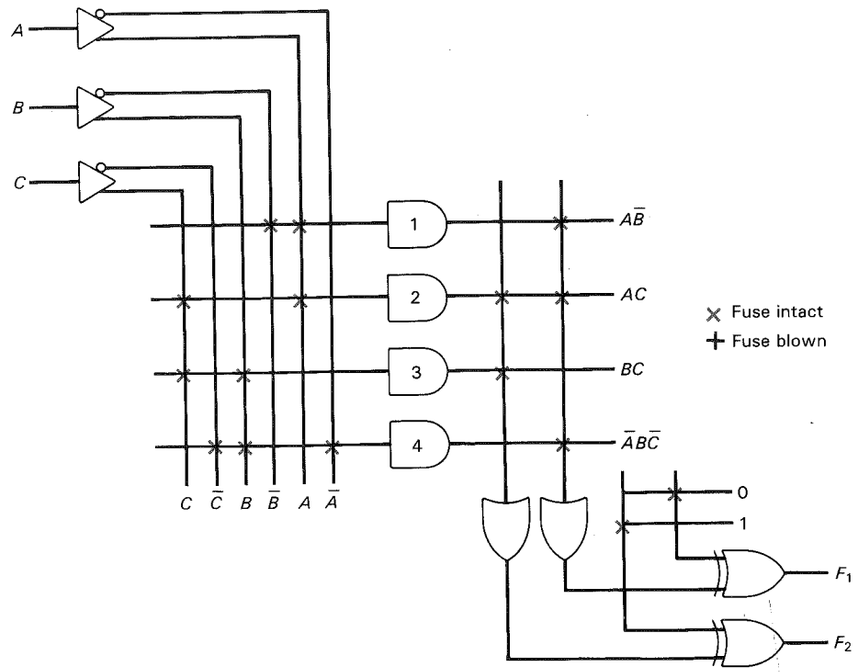


FIGURE 6-18
PLA with 3 Inputs, 4 Product Terms, and 2 Outputs

The output may be complemented or left in its true form depending on the fuses associated with the XOR gate.

The fuse map of a PLA can be specified in a tabular form. For example, the programming table that specifies the fusing of the PLA of Figure 6-18 is listed in Table 6-4. The PLA programming table consists of three sections. The first section lists the product terms numerically. The second section specifies the required paths between inputs and AND gates. The third section specifies the paths between the AND and OR gates. For each output variable we may have a T (for true) or C (for complement). The product terms listed on the left are not part of the table;

TABLE 6-4
PLA Programming Table

	Product term	Inputs			Outputs	
		A	*B*	*C*	(T) F_1	(C) F_2
$A\bar{B}$	1	1	0	—	1	—
AC	2	1	—	1	1	1
BC	3	—	1	1	—	1
$\bar{A}B\bar{C}$	4	0	1	0	1	—

they are included for reference only. For each product term, the inputs are marked with 1, 0, or — (dash). If a variable in the product term appears in its true form, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash.

The paths between the inputs and the AND gates are specified under the column heading *inputs* in the programming table. A 1 in the input column specifies an intact fuse from the input variable to the AND gate. A 0 in the input column specifies an intact fuse from the complement of the variable to the input of the AND gate. A dash specifies a blown fuse in both the input variable and its complement. It is assumed that an open terminal in the input of an AND gate behaves like a 1.

The paths between the AND and OR gates are specified under the column heading *outputs*. The output variables are marked with 1's for those product terms that are included in the function. Each product term that has a 1 in the output column requires a path from the output of the AND gate to the input of the OR gate. Those marked with a dash specify a blown fuse. It is assumed that an open terminal in the input of an OR gate behaves like a 0. Finally, a T (true) output dictates that the other input of the corresponding XOR gate be connected to 0, and a C (complement) specifies a connection to 1.

The size of a PLA is specified by the number of inputs, the number of product terms and the number of outputs. A typical integrated circuit PLA may have 16 inputs, 48 product terms, and 8 outputs. For n inputs, k product terms, and m outputs the internal logic of the PLA consists of n buffer-inverter gates, k AND gates, m OR gates, and m XOR gates. There are $2n \times k$ fuses between the inputs and the AND array; $k \times m$ fuses between the AND and OR arrays; and m fuses associated with the XOR gates.

When designing a digital system with a PLA, there is no need to show the internal connections of the unit as was done in Figure 6-18. All that is needed is a PLA programming table from which the PLA can be programmed to supply the required logic. As with a ROM, the PLA may be mask programmable or field programmable. With mask programming, the customer submits a PLA program table to the manufacturer. This table is used by the vendor to produce a custom-made PLA that has the required internal logic specified by the customer. A second type of PLA available is called a *field programmable logic array* or FPLA. The FPLA can be programmed by the user by means of certain recommended procedures. Commercial hardware programmer units are available for use in conjunction with FPLAs.

When implementing a combinational circuit with PLA, careful investigation must be undertaken in order to reduce the number of distinct product terms, since a PLA has a finite number of AND gates. This can be done by simplifying each Boolean function to a minimum number of terms. The number of literals in a term is not important since all the input variables are available anyway. Both the true and complement of the function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

Example 6-2 Implement the following two Boolean functions with a PLA.

$$F_1(A, B, C) = \Sigma\, m(0, 1, 2, 4)$$

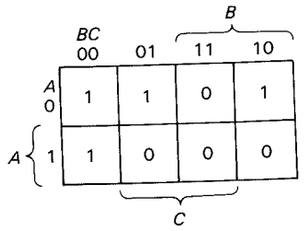
$$F_2(A, B, C) = \Sigma\, m(0, 5, 6, 7)$$

The two functions are simplified in the maps of Figure 6-19. Both the true and complement of the functions are simplified in sum of products. The combination that gives a minimum number of product terms is

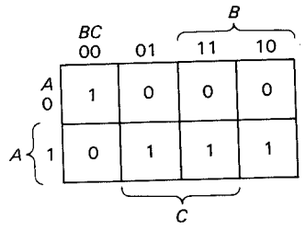
$$F_1 = \overline{AB + AC + BC}$$

$$F_2 = AB + AC + \overline{A}\,\overline{B}\,\overline{C}$$

This gives four distinct product terms: AB, AC, BC, and $\overline{A}\,\overline{B}\,\overline{C}$. The PLA programming table for this combination is shown in Figure 6-19. Note that output F_1 is the true output even though a C is marked over it in the table. This is because $\overline{F}_1$ is generated with an AND-OR circuit and is available at the output of the OR gate. The XOR gate complements the function to produce the true F_1 output.



$F_1 = \overline{A}\,\overline{B} + \overline{A}\,\overline{C} + \overline{B}\,\overline{C}$
$\overline{F}_1 = AB + AC + BC$



$F_2 = AB + AC + \overline{A}\,\overline{B}\,\overline{C}$
$\overline{F}_2 = \overline{A}C + \overline{A}B + A\overline{B}\overline{C}$

PLA programming table

	Product term	Inputs A	B	C	Outputs (C) F_1	(T) F_2
AB	1	1	1	–	1	1
AC	2	1	–	1	1	1
BC	3	–	1	1	1	–
$\overline{A}\,\overline{B}\,\overline{C}$	4	0	0	0	–	1

FIGURE 6-19
Solution to Example 6-2

The combinational circuit used in Example 6-2 is too simple for implementing with a PLA. It was presented here merely for demonstration purposes. A typical commercial PLA has over 10 inputs and 50 product terms. The simplification of Boolean functions with so many variables should be carried out by means of computer assisted simplification procedures. This is where computer software is of help in the design of complex digital systems. The computer aided design program must simplify each function and its complement to a minimum number of terms. The program then selects a minimum number of product terms that cover all functions in their true or complement form. The PLA programming table is then generated from which is obtained the required fuse map. The fuse map is applied to an FPLA programmer that goes through the hardware procedure of blowing the internal fuses in the integrated circuit.

6-8 PROGRAMMABLE ARRAY LOGIC (PAL)

The PAL is a programmable logic device with a fixed OR array and programmable AND array. Because only the AND gates are programmable, the PAL is easier to program but is not as flexible as the PLA. Figure 6-20 shows the logic configuration of a typical PAL. It has four inputs and four outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate. There are four sections in the unit, each composed of a 3-wide AND-OR array. This is the term used to indicate that there are three programmable AND gates in each section. Each AND gate has 10 fused programmable inputs. This is shown in the diagram with 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of the AND gate. One of the outputs is connected to a buffer-inverter gate and then fed back into the inputs of the AND gates through fuses.

Commercial PAL devices contain more gates than the one shown in Figure 6-20. A typical PAL integrated circuit may have eight inputs, eight outputs, and eight sections each consisting of an 8-wide AND-OR array. The output terminals are sometimes bidirectional which means that they can be programmed as inputs instead of outputs if desired. Some PAL units incorporate D-type flip-flops in the outputs. The outputs of the flip-flops are fed back through a buffer-inverter gate into the AND programmed array. This provides a capability for implementing sequential circuits.

When designing with a PAL, the Boolean functions must be simplified to fit into each section. Unlike the PLA, a product term cannot be shared among two or more OR gates. Therefore, each function can be simplified by itself without regard to common product terms. The number of product terms in each section is fixed, and if the number of terms in the function is too large, it may be necessary to use two sections to implement one Boolean function.

As an example of using a PAL in the design of a combinational circuit, consider the following Boolean functions given in sum of minterms.

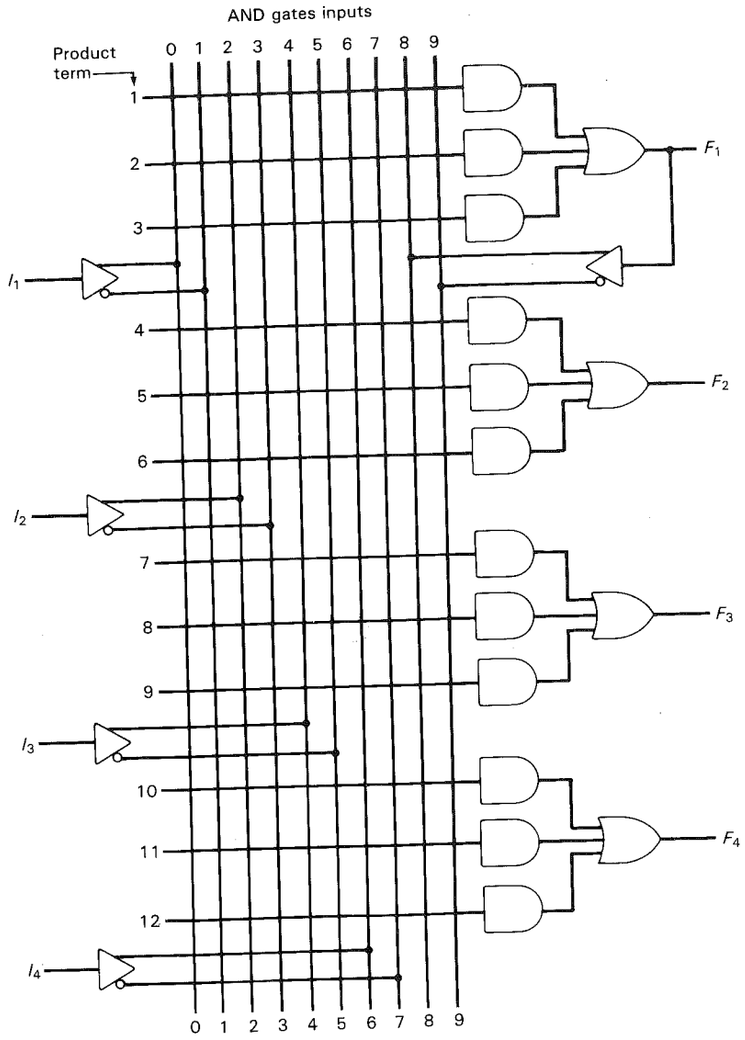


FIGURE 6-20
PAL with 4 Inputs, 4 Outputs, and 3-Wide AND-OR Structure

$$W(A, B, C, D) = \Sigma\, m(2, 12, 13)$$
$$X(A, B, C, D) = \Sigma\, m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$
$$Y(A, B, C, D) = \Sigma\, m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$
$$Z(A, B, C, D) = \Sigma\, m(1, 2, 8, 12, 13)$$

TABLE 6-5
PAL Programming Table

Product term	AND inputs A	B	C	D	W	Outputs
1	1	1	0	—	—	$W = A\overline{B}\overline{C}$
2	0	0	1	0	—	$+ \overline{A}\overline{B}C\overline{D}$
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$
5	—	1	1	1	—	$+ BCD$
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$
8	—	—	1	1	—	$+ CD$
9	—	0	—	0	—	$+ \overline{B}\overline{D}$
10	—	—	—	—	1	$Z = W$
11	1	—	0	0	—	$+ A\overline{C}\overline{D}$
12	0	0	0	1	—	$+ \overline{A}\overline{B}C\overline{D}$

Simplifying the four functions to a minimum number of terms results in the following Boolean functions:

$$W = A\overline{B}\overline{C} + \overline{A}\overline{B}C\overline{D}$$
$$X = A + BCD$$
$$Y = \overline{A}B + CD + \overline{B}\overline{D}$$
$$Z = A\overline{B}\overline{C} + \overline{A}\overline{B}C\overline{D} + A\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D$$
$$= W + A\overline{C}\overline{D} + \overline{A}\overline{B}\overline{C}D$$

Note that the function for Z has four product terms. The logical sum of two of these terms is equal to W. By using W it is possible to reduce the number of terms for Z from four to three.

The PAL programming table is similar to the one used for the PLA except that only the inputs of the AND gates need to be programmed. Table 6-5 lists the PAL programming table for the four Boolean functions. The table is divided into four sections with three product terms in each to conform with the PAL of Figure 6-20. The first two sections need only two product terms to implement the Boolean function. The last section for output Z needs four product terms. Using the output from W we can reduce the function to three terms.

The fuse map for the PAL as specified in the programming table is shown in Figure 6-21. For each 1 or 0 in the table we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash we mark the diagram with blown fuses in both the true and complement inputs. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the true and complement of each input variable, we have $A \cdot \overline{A} = 0$ and the output of the AND gate is always 0.

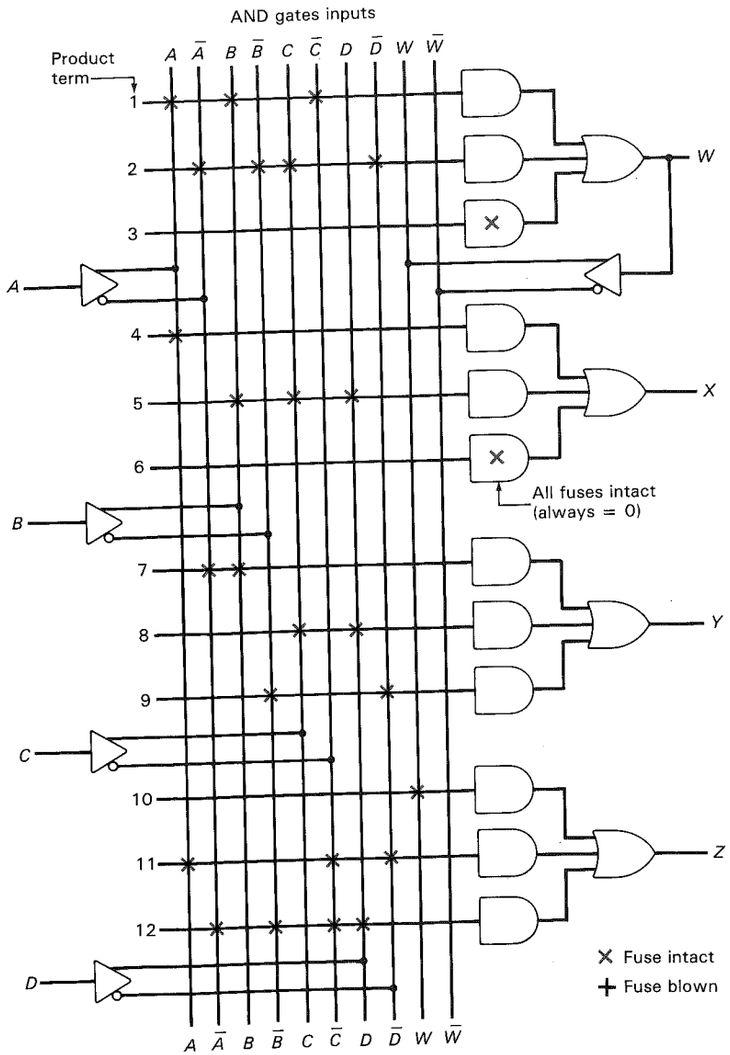


FIGURE 6-21
Fuse Map for PAL as Specified in Table 6-5

As with all PLDs, the design with PALs is facilitated by using computer aided design techniques. The blowing of internal fuses is a hardware procedure done with the help of special electronic instruments.

REFERENCES

1. PEATMAN J. B. *Digital Hardware Design*. New York: McGraw-Hill, 1980.
2. BLAKESLEE, T. R. *Digital Design with Standard MSI and LSI*. 2nd ed. New York: Wiley, 1979.
3. *The TTL Data Book*. Vol 4. Dallas: Texas Instruments, 1984
4. KITSON, B., ED. *Programmable Array Logic Handbook*. Sunnyvale, CA: Advanced Micro Devices, 1983.
5. HAMMING, R. W. "Error Detecting and Error Correcting Codes." *Bell System Tech. Jour.*, 29 (1950): 147–160.
6. LIN, S., AND COSTELLO, D. J., JR. *Error Control Coding*. Englewood Cliffs: Prentice-Hall, 1983.

PROBLEMS

6-1 The following memory units are specified by the number of words times the number of bits per word. How many address lines and input-output data lines are needed in each case?
(a) 2K × 16; (b) 64K × 8; (c) 16M × 32; (d) 96K × 12.

6-2 Give the number of bytes stored in the memories listed in Problem 6-1 (a), (b), and (c).

6-3 Word number 535 in the memory shown in Figure 6-3 contains the binary equivalent of 2209. List the 10-bit address and the 16-bit memory content of the word.

6-4 Draw the standard graphic symbol of a 256 × 1 RAM. Include the symbol for three-state output.

6-5 Show the memory cycle timing waveforms for the write and read operations. Assume a CPU clock of 2.5 MHz and a memory cycle time of 600 nsec.

6-6 Draw the standard graphic diagram of the 4 × 4 RAM of Figure 6-7 including three-state outputs. Construct an 8 × 8 memory using four 4 × 4 RAM units.

6-7 A 16K × 4 memory uses coincident decoding by splitting the internal decoder into *X*-selection and *Y*-selection.
(a) What is the size of each decoder and how many AND gates are required for decoding the address?
(b) Determine the *X* and *Y* selection lines that are enabled when the input address is the binary equivalent of 6,000.

6-8 (a) How many 128 × 8 RAM chips are needed to provide a memory capacity of 2048 bytes?
(b) How many lines of the address must be used to access 2048 bytes? How many of these lines are connected to the address inputs of all chips?
(c) How many lines must be decoded for the chip select inputs? Specify the size of the decoder.

6-9 A computer uses RAM chips of 1024 × 1 capacity.
(a) How many chips are needed and how should their address lines be connected to provide a memory capacity of 1024 bytes?
(b) How many chips are needed to provide a memory capacity of 16K bytes? Explain in words how the chips are to be connected.

6-10 An integrated circuit RAM chip has a capacity of 1024 words of 8 bits each (1K × 8).
(a) How many address and data lines are there in the chip?
(b) How many chips are needed to construct a 16K × 16 RAM?
(c) How many address and data lines are there in the 16K × 16 RAM?
(d) What size of decoder is needed to construct the 16K × 16 memory from the 1K × 8 chips? What are the inputs to the decoder and where are its outputs connected?

6-11 Given the 8-bit data word 01011011, generate the 13-bit composite word for the Hamming code that corrects single errors and detects double errors.

6-12 Given the 11-bit data word 11001001010, generate the 15-bit Hamming code word.

6-13 A 12-bit Hamming code word containing 8 bits of data and 4 parity bits is read from memory. What was the original 8-bit data word that was written into memory if the 12-bit word read out is
(a) 000011101010 (b) 101110000110 (c) 101111110100

6-14 How many parity check bits must be included with the data word to achieve single error correction and double error detection when the data word contains (a) 16 bits; (b) 32 bits; (c) 48 bits.

6-15 It is necessary to formulate the Hamming code for 4 data bits D_3, D_5, D_6, and D_7, together with three parity bits P_1, P_2, and P_4.
(a) Evaluate the 7-bit composite code word for the data word 0010.
(b) Evaluate the three check bits C_4, C_2, and C_1, assuming no error.
(c) Assume an error in bit D_5 during writing into memory. Show how the error in the bit is detected and corrected.
(d) Add a parity bit P_8 to include a double error detection in the code. Assume that errors occurred in bits P_2 and D_5. Show how the double error is detected.

6-16 Given a 32 × 8 ROM chip with an enable input, show the external connections necessary to construct a 128 × 8 ROM with four chips and a decoder.

6-17 A ROM chip of 4096 × 8 bits has two chip select inputs and operates from a 5-volt power supply. How many pins are needed for the integrated circuit package? Draw a block diagram and label all input and output terminals in the ROM.

6-18 The 32 × 6 ROM together with the 2^0 line as shown in Figure P6-18 converts a 6-bit binary number to its corresponding 2-digit BCD number. For example, binary 100001 converts to BCD 011 0011 (decimal 33). Specify the truth table for the ROM.

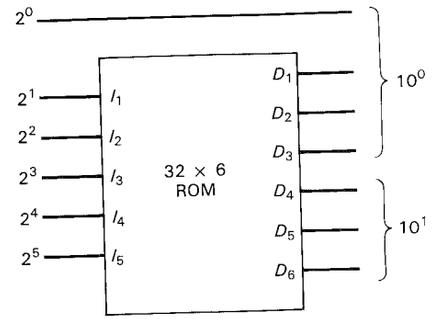


FIGURE P6-18
Binary-to-Decimal ROM Converter

6-19 Specify the size of a ROM (number of words and number of bits per word) that will accommodate the truth table for the following combinational circuit components.
(a) A binary multiplier that multiplies two 4-bit numbers.
(b) A 4-bit adder-subtractor (see Figure 3-12).
(c) A quadruple 2-to-1-line multiplexer with common select and enable inputs (see Figure 3-24).
(d) A BCD to seven segment decoder with an enable input.

6-20 Tabulate the truth table for an 8 × 4 ROM that implements the following four Boolean functions.

$$A(X, Y, Z) = \Sigma\, m(1, 2, 4, 6)$$
$$B(X, Y, Z) = \Sigma\, m(0, 1, 6, 7)$$
$$C(X, Y, Z) = \Sigma\, m(2, 6)$$
$$D(X, Y, Z) = \Sigma\, m(1, 2, 3, 5, 7)$$

6-21 Obtain the PLA programming table for the four Boolean functions listed in Problem 6-20. Minimize the number of product terms.

6-22 Derive the PLA programming table for the combinational circuit that squares a 3-bit number. Minimize the number of product terms. (See Figure 6-15 for the equivalent ROM implementation.)

6-23 List the PLA programming table for the BCD to excess-3 code converter whose Boolean functions are simplified in Figure 3-5.

6-24 Repeat Problem 6-23 using a PAL.

6-25 The following is a truth table of a 3-input, 4-output combinational circuit. Obtain the PAL programming table for the circuit and mark the fuses to be blown in a PAL diagram similar to the one shown in Figure 6-20.

Inputs			Outputs			
X	*Y*	*Z*	*A*	*B*	*C*	*D*
0	0	0	0	1	0	0
0	0	1	1	1	1	1
0	1	0	1	0	1	1
0	1	1	0	1	0	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1

6-26 Modify the PAL diagram in Figure 6-20 by including three clocked D-type flip-flops between the OR gates and outputs F_2, F_3, and F_4. The diagram should conform with the block diagram of a sequential circuit as shown in Figure 6-17. This will require three additional buffer-inverter gates and six vertical lines for the flip-flop outputs to be connected to the AND array through programmable fuses. Using the modified PAL diagram, show the fuse map that will implement a 3-bit binary counter with a carry output in F_1.

7

REGISTER TRANSFER AND COMPUTER OPERATIONS

7-1 INTRODUCTION

A digital system is a sequential logic system constructed with flip-flops and gates. It was shown in Chapter 4 that sequential circuits can be specified by means of state tables. To specify a large digital system with state tables is very difficult, if not impossible, because the number of states would be prohibitively large. To overcome this difficulty, digital systems are designed using a modular approach. The system is partitioned into modular subsystems, each of which performs some functional task. The modules are constructed from such digital devices as registers, counters, decoders, multiplexers, arithmetic elements, and control logic. The various modules are interconnected by common data and control paths to form the digital computer system.

Digital modules are best defined by the registers they contain and the operations that are performed on the binary information stored in them. Examples of register operations are shift, count, clear, and load. In this configuration, the registers are assumed to be the basic components of the digital system and the information flow and processing tasks among the data stored in the registers are referred to as *register transfer* operations. The register transfer operations of digital systems are specified by the following three basic components:

1. The set of registers in the system and their function.
2. The operations that are performed with the information stored in the registers.
3. The control that supervises the sequence of operations in the system.

A *register*, as defined in the register transfer notation, is a group of flip-flops that stores binary information and has the capability of performing one or more elementary operations. A register can load new information or shift the information to the right or the left. A counter is considered to be a register that performs the increment-by-one operation. A flip-flop standing alone is considered as a 1-bit register that can be set, cleared, or complemented. In fact, the flip-flops and associated gates of any sequential circuit are called a register by this method of designation.

The operations performed on the information stored in registers are called *microoperations*. A microoperation is an elementary operation that can be performed in parallel on a string of bits during one clock-pulse period. The result of the operation may replace the previous binary information in a register or may be transferred to another register. The digital functions introduced in Chapter 5 are registers that implement microoperations. A counter with parallel load is capable of performing the microoperations increment and load. A bidirectional shift register is capable of performing the shift right and shift left microoperations.

The control that initiates the sequence of operations consists of timing signals that sequence the operations in a prescribed manner. Certain conditions which depend on results of previous operations may determine the sequence of future operations. The outputs of the control logic are binary variables that initiate the various microoperations in the registers.

This chapter introduces the components of the register transfer with a symbolic notation for representing registers and specifying the operations on the contents of the registers. The register transfer method uses a set of expressions and statements that resemble the statements used in programming languages. This notation provides the necessary tools for specifying the prescribed set of interconnections among various digital functions.

Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers in conjunction with a common operational unit called an *arithmetic logic unit*, abbreviated ALU. To perform a microoperation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often performed in a separate unit. The shift unit is usually shown separately, but sometimes this unit is considered to be part of the overall arithmetic and logic unit.

A group of registers connected to a common ALU is called a *processor unit*. The processor unit is that part of a digital computer that implements the data processing operations in the system. The processor unit, when combined with a control unit that supervises the sequence of operations, is called the *central processing unit* abbreviated CPU. The second part of this chapter is concerned with the organization and design of the processor unit. The design of a particular arithmetic logic unit is undertaken to show the design process involved in implementing a complex digital circuit. The next chapter deals with the organization and design

of the control unit. Chapter 10 demonstrates the detailed design of a central processing unit.

7-2 REGISTER TRANSFER

The registers in a digital system are designated by capital letters (sometimes followed by numerals) that denote the function of the register. For example, the register that holds an address for the memory unit is usually called the memory address register and is designated by the name AR. Other designations for registers are PC, IR, $R1$, and $R2$. The individual flip-flops in an n-bit register are numbered in sequence from 0 through n-1, starting from 0 in the rightmost position and increasing toward the left. Figure 7-1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside as in Figure 7-1(a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol L (for low byte) and bits 8 through 15 are assigned the symbol H (for high byte). The name of the 16-bit register is PC. The symbol $PC(0\text{-}7)$ or $PC(L)$ refers to the low-order byte and $PC(8\text{-}15)$ or $PC(H)$ to the high order byte.

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register $R1$ into register $R2$. It designates a replacement of the contents of $R2$ by the contents of $R1$. By definition, the contents of the source register $R1$ do not change after the transfer.

A statement that specifies a register transfer implies that circuits are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability. Normally, we do not want the transfer to occur with every clock pulse, but only under a predetermined condition. A conditional statement is symbolized with an *if-then* statement

$$If\ (T_1 = 1)\ then\ (R2 \leftarrow R1)$$

where T_1 is a timing signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by

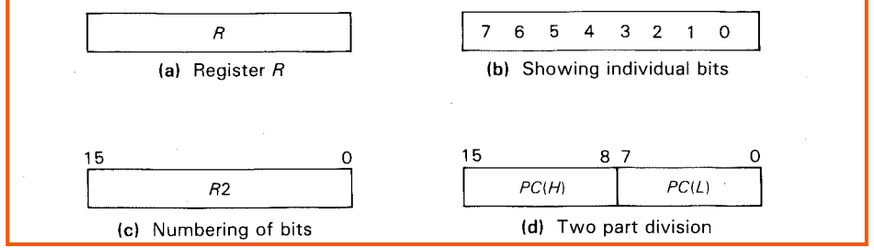


FIGURE 7-1
Block Diagram of a Register

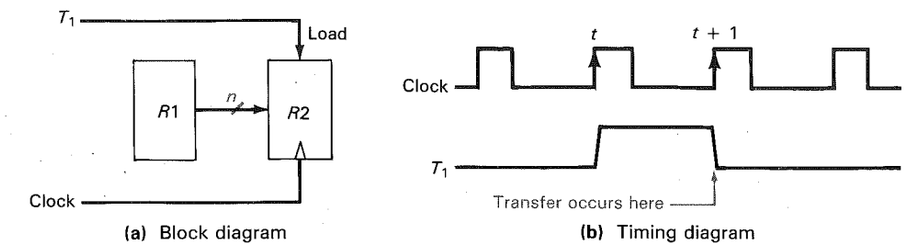


FIGURE 7-2
Transfer from $R1$ to $R2$ when $T_1 = 1$

specifying a *control function*. A control function is a Boolean variable that can be equal to 1 or 0. The control function is included in the statement as follows:

$$T_1\text{: } R2 \leftarrow R1$$

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only if $T_1 = 1$.

Every statement written in a register transfer notation implies a hardware construction for implementing the transfer. Figure 7-2 shows the block diagram that depicts the transfer from $R1$ to $R2$. The n outputs of register $R1$ are connected to the n inputs of register $R2$. The letter n will be used to indicate any number of bits for the register. It will be replaced by an actual number when the length of the register is known. Register $R2$ has a load control input which is activated by the timing variable T_1. It is assumed that the timing variable is synchronized with the same clock as the one applied to the register. As shown in the timing diagram, T_1 is activated by the rising edge of a clock pulse at time t. The next positive transition of the clock at time $t + 1$ finds $T_1 = 1$ and the inputs of $R2$ are loaded into the register in parallel. T_1 may go back to 0 at time $t + 1$ while timing variable T_2 becomes a 1 (see Figure 5-22).

Note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition. Even though the control condition such as T_1 becomes active at time t, the actual transfer does not occur until the register is triggered by the next positive transition of the clock.

The basic symbols of the register transfer notation are listed in Table 7-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by

TABLE 7-1
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	AR, $R2$
Parentheses ( )	Denotes a part of a register	$R2(0\text{–}7)$, $R2(L)$
Arrow $\leftarrow$	Denotes transfer of information	$R2 \leftarrow R1$
Comma ,	Separates two microoperations	$R2 \leftarrow R1$, $R1 \leftarrow R2$
Square brackets [ ]	Specify an address for memory	$DR \leftarrow M[AR]$

giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T_3\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided $T_3 = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

The square brackets are used in conjunction with memory transfer. The letter M designates a memory word, and the register enclosed inside the square brackets provides the address of the word in memory. This is explained in more detail in Section 7-4.

Multiplexer Selection

There are occasions when a register receives information from two different sources at different times. Consider the following conditional statement:

$$\text{If } (T_1 = 1) \text{ then } (R0 \leftarrow R1) \text{ else if } (T_2 = 1) \text{ then } (R0 \leftarrow R2)$$

The contents of register $R1$ are to be transferred to register $R0$ when timing variable T_1 occurs; otherwise, the contents of register $R2$ are transferred to $R0$ when T_2 occurs. The conditional statement may be broken into two parts using control functions.

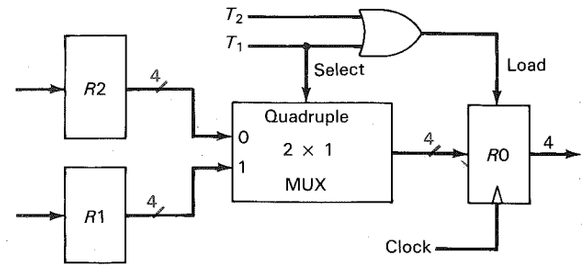
$$T_1\text{: } R0 \leftarrow R1$$
$$\overline{T}_1 T_2\text{: } R0 \leftarrow R2$$

This specifies a hardware connection from two registers, $R1$ and $R2$, to one common destination register $R0$. This type of operation requires a multiplexer to select between the two source registers according to the values of the timing variables.

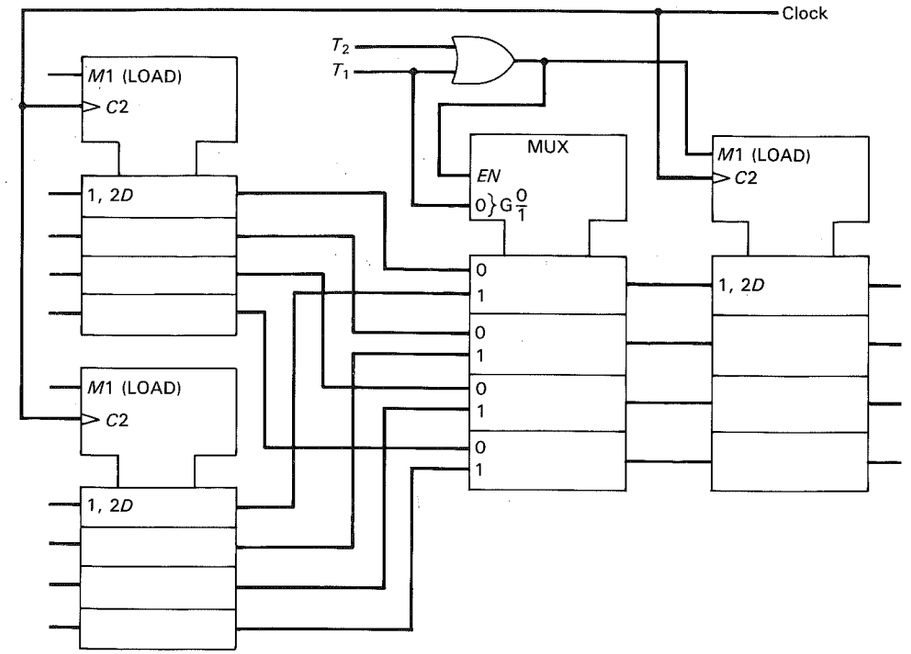
The block diagram of the circuit that implements the above two statements using 4-bit registers is shown in Figure 7-3(a). The quadruple 2×1 multiplexer selects between the two source registers. When $T_1 = 1$ register $R1$ is selected and when $T_1 = 0$, register $R2$ is selected by the multiplexer. Thus, when $T_1 = 1$, $R1$ is loaded into $R0$ irrespective of the value of T_2. When $T_2 = 1$ and $T_1 = 0$, $R2$ is loaded into $R0$. When both T_1 and T_2 are equal to 0, the multiplexer selects $R2$ for the inputs of $R0$ but the inputs are not loaded into the register because the load control input is equal to 0.

The detail logic diagram of the hardware implementation is shown in Figure 7-3(b). The diagram uses standard graphic symbols as presented in previous chapters. The graphic symbol for the registers is taken from Figure 5-3(b) and for the multiplexer from Figure 3-31(b). The enable input EN in the multiplexer is activated with the same condition as the load input of the destination register.

It is important to be able to relate the information given in a block diagram with the detail wiring connections in the corresponding logic diagram. In subsequent discussions we will present other block diagrams for various digital systems. In order to save space, the detailed logic diagram may be omitted. However, it should



(a) Block diagram



(b) Logic diagram with standard graphic symbols

FIGURE 7-3
Use of Multiplexers to Select Between Two Registers

be a straightforward procedure to obtain the logic diagram with detail wiring from the information given in the block diagram.

7-3 MICROOPERATIONS

A microoperation is an elementary operation performed with the data stored in registers. The type of microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numbers stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on contents of registers.

The register transfer microoperation was introduced in the previous section. This type of microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer. Among all possible operations that can exist in digital systems, there is a basic set from which all other operations can be obtained. In this section we define a set of basic microoperations, their symbolic notation, and the digital hardware that implements them.

Arithmetic Microoperations

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the following statement:

$$R0 \leftarrow R1 + R2$$

specifies an *add* microoperation. It states that the contents of register $R1$ are to be added to the contents of register $R2$ and the sum transferred to register $R0$. To implement this statement with hardware we need three registers and the digital component that performs the addition operation, such as a parallel adder. The other basic arithmetic microoperations are listed in Table 7-2. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R0 \leftarrow R1 + \overline{R2} + 1$$

$\overline{R2}$ is the symbol for the 1's complement of $R2$. Adding 1 to the 1's complement produces the 2's complement. Adding the contents of $R1$ to the 2's complement of $R2$ is equivalent to $R1 - R2$.

The increment and decrement microoperations are symbolized by a plus-one and minus-one operation, respectively. These microoperations are implemented with a combinational circuit or with a binary up-down counter.

The arithmetic operations multiply and divide are not listed in Table 7-2. The multiplication operation can be represented by the symbol * and the division be a /. These two operations are valid arithmetic operations but are not included in the basic set of microoperations. The only place where these operations can be considered as microoperations is in a digital system where they are implemented by means of a combinational circuit. In such a case, the signals that perform these operations propagate through gates, and the result of the operation can be transferred into a destination register by a clock pulse as soon as the output signal propagates through the combinational circuit. In many computers, the multiplication operation is implemented with a sequence of add and shift microoperations. Division is implemented with a sequence of subtract and shift microoperations. To specify the hardware in such a case requires a list of statements that use the basic microoperations of add, subtract, and shift.

TABLE 7-2
Arithmetic Microoperations

Symbolic designation	Description
$R0 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R0$
$R0 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R0$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$
$R0 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ (count up)
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ (count down)

There is a direct relationship between the statements written in a register transfer notation and the registers and digital functions that are required for their implementation. To illustrate with an example, consider the following two statements:

$$\overline{X}T_1\colon R1 \leftarrow R1 + R2$$

$$XT_1\colon R1 \leftarrow R1 + \overline{R2} + 1$$

Timing variable T_1 initiates an operation to add or subtract. If, at the same time, control variable X is equal to 0 then $\overline{X}T_1 = 1$ and the contents of $R2$ are added to the contents of $R1$. If $X = 1$, then $XT_1 = 1$ and the contents of $R2$ are subtracted from $R1$. Note that the two control functions are Boolean functions and reduce to 0 when $T_1 = 0$, a condition that inhibits the execution of either operation.

The implementation of the two statements is shown in block diagram form in Figure 7-4. An n-bit adder-subtractor (similar to the one shown in Figure 3-12)

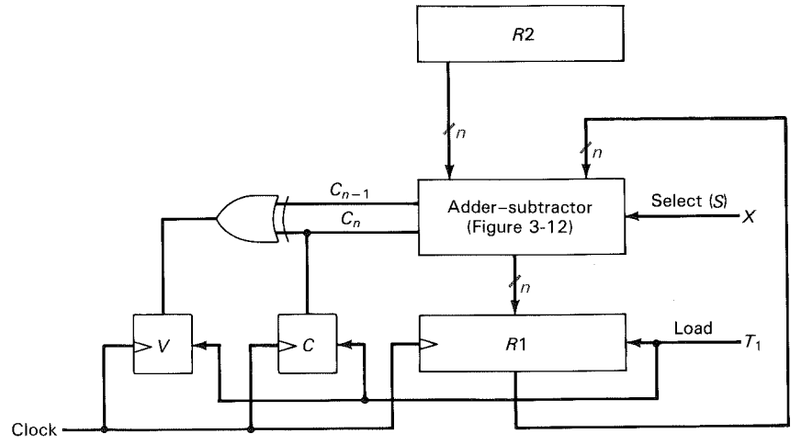


FIGURE 7-4
Implementation of Add and Subtract Microoperations

receives its input data from registers $R1$ and $R2$. The sum or difference is applied to the inputs of $R1$. The select input S in the adder-subtractor selects the operation in the circuit. When $S = 0$, the two inputs are added and when $S = 1$, $R2$ is subtracted from $R1$. Applying the control variable X to the select input provides the required operation. The output of the adder-subtractor is loaded into $R1$ if $\overline{X}T_1 = 1$ or if $XT_1 = 1$. This can be simplified to only T_1 since

$$\overline{X}T_1 + XT_1 = (\overline{X} + X)T_1 = T_1$$

Thus, control variable X selects the operation and timing variable T_1 loads the result into $R1$. The output carry C_n is transferred to flip-flop C. The function of flip-flop V is to detect for overflow as explained below.

Overflow

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. This is true for binary or decimal numbers whether signed or unsigned. When the addition is performed with paper and pencil, an overflow is not a problem, since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the length of registers is finite. A result that contains $n + 1$ bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign and negative numbers are in 2's complement form. When two signed numbers are added, the sign bit is treated as part of the number and the end carry does not indicate an overflow (see Section 1-5).

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result which is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary −128. Since the sum of the two numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

carries:	0	1		carries:	1	0
+70	0	1000110		−70	1	0111010
+80	0	1010000		−80	1	0110000
+150	1	0010110		−150	0	1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If,

however, the carry out of the sign bit position is taken as the sign bit of the result, then the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

An overflow condition can be detected by observing the carry *into* the sign bit position and the carry *out* of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

The addition and subtraction of two binary numbers with digital hardware is shown in Figure 7-4. If the numbers are considered unsigned, then the C bit detects a carry after addition or a borrow after subtraction. If the numbers are considered to be signed, then the V bit detects an overflow. If $V = 0$ after an addition or subtraction, it indicates that no overflow occurred and the answer in $R1$ is correct. If $V = 1$, then the result of the operation contains $n + 1$ bits. Only n bits of the number are in $R1$. The $(n + 1)$th bit is the sign bit and has been shifted out of position into the carry bit C.

Logic Microoperations

Logic microoperations are useful for manipulating the bits stored in a register. These operations consider each bit in the register separately and treat it as a binary variable. The symbols for the four basic logic microoperations are shown in Table 7-3. The complement microoperation is the same as the 1's complement and uses a bar on top of the register name. The symbol $\wedge$ is used to denote the AND microoperation and the symbol $\vee$ to denote the OR microoperation. By using these special symbols it is possible to differentiate between the add microoperation symbolized by a + and the OR microoperation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When this symbol occurs in a microoperation, it denotes an addition operation. When it occurs in a control or Boolean function, it denotes an OR operation. The OR microoperation will always use the $\vee$ symbol. For example, in the statement

$$T_1 + T_2: R1 \leftarrow R2 + R3,\ R4 \leftarrow R5 \vee R6$$

the + between T_1 and T_2 is an OR operation between two variables in a control (Boolean) function. The + between $R2$ and $R3$ specifies an add microoperation. The OR microoperation is designated by the symbol $\vee$ between registers $R5$ and $R6$.

TABLE 7-3
Logic Microoperations

Symbolic designation	Description
$R \leftarrow \overline{R}$	Complements all bits of register R
$R0 \leftarrow R1 \wedge R2$	Logic AND microoperation (clears bits)
$R0 \leftarrow R1 \vee R2$	Logic OR microoperation (sets bits)
$R0 \leftarrow R1 \oplus R2$	Logic XOR microoperation (complements bits)

The logic microoperations can be easily implemented with a group of gates. The complement of a register of n bits is obtained with n inverter gates. The AND microoperation is obtained from a group of AND gates, each of which receives a pair of bits from two source registers. The outputs of the gates are applied to the inputs of the destination register. The OR and Exclusive-OR microoperations require a similar arrangement of gates.

The logic microoperations can change bit values, clear a group of bits, or insert new bit values in a register. The following examples show how the bits of register $R1$ are manipulated by logic microoperations with a logic operand in $R2$.

The AND microoperation is used for clearing to 0 a bit or a selected group of bits in a register. The Boolean relationships $X \cdot 0 = 0$ and $X \cdot 1 = X$ dictate that a binary variable X when ANDed with 0 produces a 0 but when ANDed with 1 it does not change the value of X. A given bit or a group of bits in a register can be cleared to 0 if ANDed with 0. Consider the following example.

10101101 10101011	$R1$
00000000 11111111	$R2$
00000000 10101011	$R1 \leftarrow R1 \wedge R2$

The 16-bit logic operand in $R2$ has 0's in the high order byte and 1's in the low order byte. By ANDing this with the contents of $R1$, it is possible to clear to 0's the high order byte of $R1$ and leave the bits in the low order byte unchanged. Thus, the AND operation can be used to selectively clear bits of a register. This operation is sometimes referred to as *masking out* the bits because it masks or deletes all 1's from a selected portion of a register.

The OR microoperation is used to set a bit or a group of bits in a register. The Boolean relationships $X + 1 = 1$ and $X + 0 = X$ dictate that the binary variable X when ORed with 1 produces a 1 but when ORed with 0 it does not change the value of X. A given bit or a group of bits in a register can be set to 1 if ORed with 1. Consider the following example.

we just OR'ed them

10101101 10101011	$R1$
11111111 00000000	$R2$
11111111 10101011	$R1 \leftarrow R1 \vee R2$

The high order byte of $R1$ is set to all 1's by ORing it with all 1's in the $R2$ operand. The low order byte remains unchanged because it is ORed with 0's.

The XOR (exclusive-OR) microoperation is used to complement a bit or a group of bits in a register. The Boolean relationships $X \oplus 1 = \overline{X}$ and $X \oplus 0 = X$ dictate that when the binary variable X is XORed with 1 it is complemented but when XORed with 0 it remains unchanged. By XORing a bit or a group of bits in a register with 1, it is possible to complement the selected bits. Consider the example

10101101 10101011	$R1$
11111111 00000000	$R2$
01010010 10101011	$R1 \leftarrow R1 \oplus R2$

TABLE 7-4
Shift Microoperations

Symbolic designation	Description
$R \leftarrow$ shl R	Shift left register R
$R \leftarrow$ shr R	Shift right register R
$R \leftarrow$ rol R	Rotate left register R
$R \leftarrow$ ror R	Rotate right register R
$R \leftarrow$ asl R	Arithmetic shift left R
$R \leftarrow$ asr R	Arithmetic shift right R

The high order byte in $R1$ is complemented after the XOR operation with the operand in $R2$.

Shift Microoperations

Shift microoperations are used for serial transfer of data. They are also used in arithmetic, logic and control operations. The contents of a register can be shifted to the left or the right. There are no standard symbols for the shift microoperations. Here we will adopt the symbols *shl* and *shr* for shift left and shift right operations. For example

$$R1 \leftarrow \text{shl } R1,\ R2 \leftarrow \text{shr } R2$$

are two microoperations that specify a 1-bit shift to the left of register $R1$ and a 1-bit shift to the right of register $R2$. The register symbol must be the same on both sides of the arrow as in the increment operation.

While the bits of a register are shifted, the end bit position receives information from the serial input. The end position is the rightmost bit of the register during a shift left operation and the leftmost bit of the register during a shift right operation. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift, which is symbolized by shl or shr. In a *rotate* operation, the serial output is connected to the serial input and the bits of the register rotate without any loss of information. The symbol used for the rotate is shown in Table 7-4.

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift left multiplies a signed binary number by 2. An arithmetic shift right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2.

The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. Figure 7-5 shows a typical register of n bits. Bit R_{n-1} in the leftmost position holds the sign bit. R_0 is the least significant bit and R_{n-2} is the most

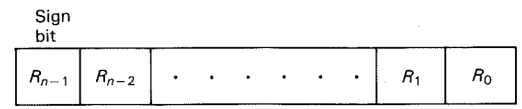


FIGURE 7-5
Defining Register R for Arithmetic Shifts

significant bit of the number. It is assumed that negative numbers have a 1 in the sign bit and are in 2's complement form.

The arithmetic shift right leaves the sign bit unchanged and shifts the number (including the sign bit) to the right. Thus, R_{n-1} remains the same, R_{n-2} receives the bit from R_{n-1} and so on for the other bits in the register. The bit in R_0 is lost.

The arithmetic shift left inserts a 0 into R_0, and shifts all other bits to the left. The initial bit of R_{n-1} is lost and replaced by the bit from R_{n-2}. A sign reversal occurs if the bit in R_{n-1} changes in value after the shift. This happens if the multiplication by 2 causes an overflow. An overflow occurs after an arithmetic shift left if R_{n-1} is not equal to R_{n-2} *before* the shift. An overflow flip-flop V_s can be used to detect an arithmetic shift overflow.

$$V_s = R_{n-1} \oplus R_{n-2}$$

If $V_s = 0$, there is no overflow, but if $V_s = 1$, there is an overflow and a sign reversal after the shift. V_s must be transferred into the overflow flip-flop with the same clock pulse that shifts the register.

7-4 BUS TRANSFER

A typical digital computer has many registers and paths must be provided to transfer information from one register to another. The number of wires will be excessive if separate lines and multiplexers are used between each register and all other registers in the system. A more efficient scheme for transferring information between registers in a multiple register configuration is a *bus* system. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals select which register will be the source and which will be the destination during each register transfer.

One way of constructing a common bus system is with multiplexers and a decoder. The multiplexers select one source register whose binary information is then placed on the bus. The decoder selects one destination register to accept the information from the bus. The construction of a bus system for four registers is shown in Figure 7-6. Each register has n bits numbered from 0 through n-1. The bits in the same significant position in each register are applied to a 4-to-1-line multiplexer to form one line of the bus. Only three multiplexers are shown in the diagram. The complete circuit must have n multiplexers numbered 0 to n-1. The n lines formed by the common bus system are routed to the n inputs of each register. The transfer of information from the bus into one destination register is accomplished by activating the load control input of the selected register. The particular load input is selected from the outputs of the decoder.

The MUX select inputs determine which register will place its contents on the bus. The MUX select inputs can be either 00, 01, 10, or 11, which select register $R0$, $R1$, $R2$, or $R3$, respectively. The destination select inputs to the decoder determine the register that receives the information from the bus. The destination select inputs can be either 00, 01, 10, or 11, and they select the destination register $R0$, $R1$, $R2$, or $R3$, respectively.

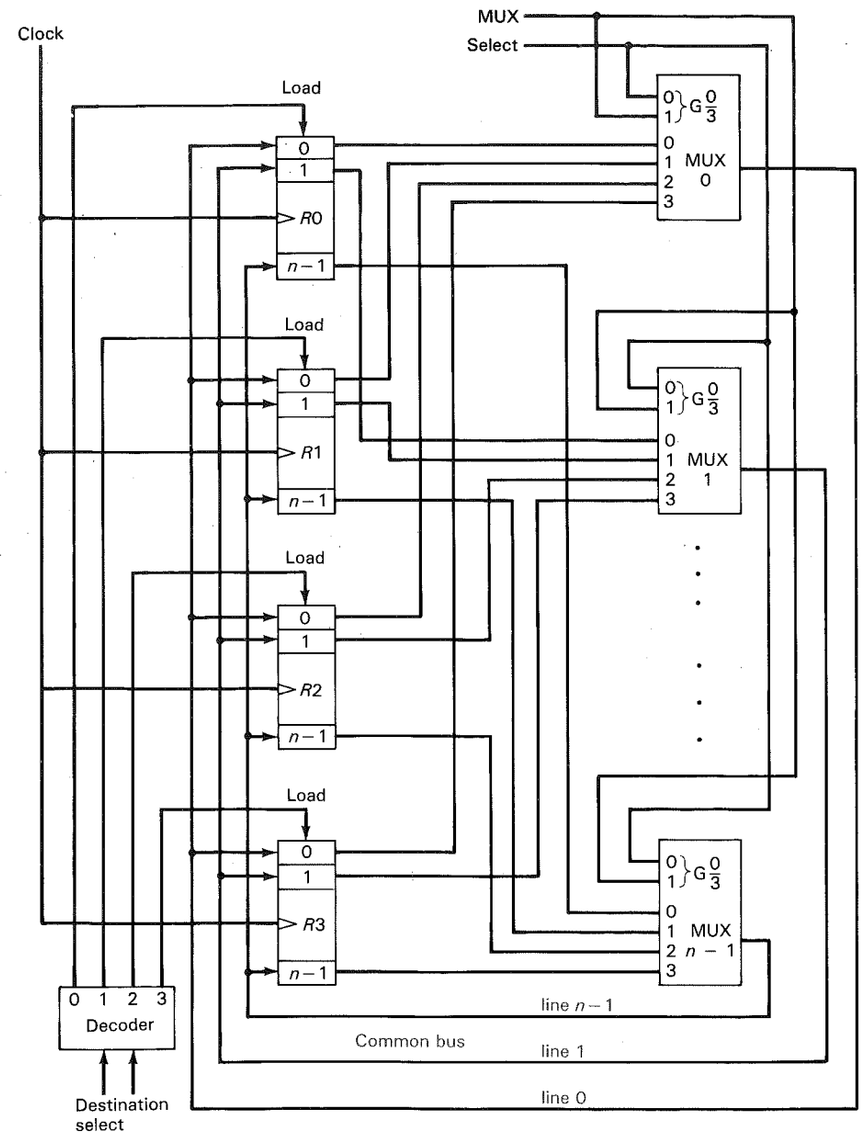


FIGURE 7-6
Bus System for Four Registers

To illustrate with a particular example, consider the transfer given by the following statement:

$$R2 \leftarrow R0$$

The control variables that enable this transfer must select register $R0$ as the source for the bus and register $R2$ as the destination. The multiplexer select inputs must be binary 00. This causes bit 0 of $R0$ to be applied to line 0 of the bus through

MUX 0. At the same time bit 1 of $R0$ is applied to line 1 of the bus through MUX 1. This repeats for all other bus lines up to line n-1 which receives bit n-1 of $R0$ through MUX n-1. Thus, the n-bit value of $R0$ is placed on the common bus lines when the MUX select is 00. The destination select inputs must be binary 10. This activates output 2 of the decoder which in turn activates the load input of $R2$. With the next clock transition, the contents of $R0$, being on the bus, are loaded into register $R2$ to complete the transfer.

Three-State Bus Buffers

A bus system can be constructed with three-state gates instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic-1 and logic-0 as in a conventional gate. The third state is a *high-impedance* state. The high-impedance state behaves like an open circuit which means that the output is disconnected and does not have a logic significance. Three-state gates may perform any conventional logic such as AND or NAND. However, the one most commonly used in the design of a bus system is the buffer gate.

The graphic symbol of a three-state buffer gate is shown in Figure 7-7. It is distinguished from a normal buffer by its small triangle symbol in front of the output terminal. The circuit has a normal input and a control input that determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input. The high-impedance state of a three-state gate provides a special feature not available in other gates. Because of this feature, a large number of three-state gate outputs can be connected with wires to form a common bus line without endangering loading effects.

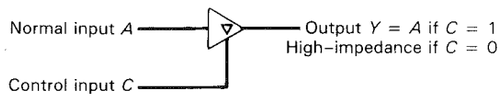


FIGURE 7-7
Graphic Symbol for a Three-State Buffer

The construction of a bus system with three-state buffers is demonstrated in Figure 7-8. The outputs of four buffers are connected together to form a single bus line. (This type of connection cannot be done with gates that do not have three-state outputs.) The control inputs to the buffers determine which one of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three-state buffer has access to the bus line while all other buffers are maintained in a high-impedance state.

One way to ensure that no more than one control input is active at any given time is to use a decoder as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance

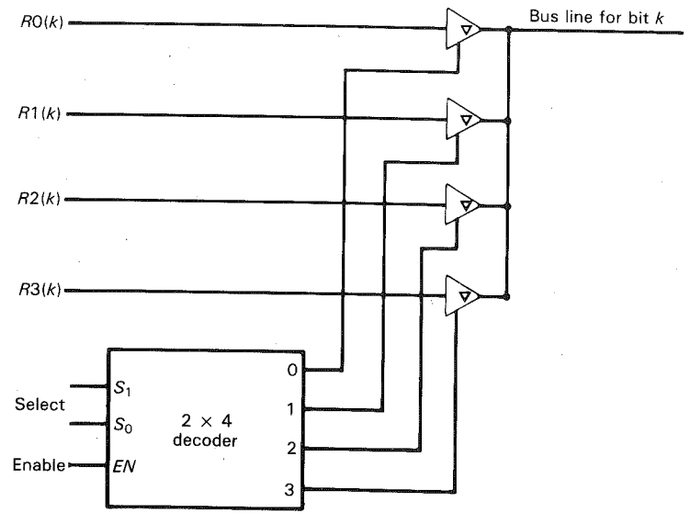


FIGURE 7-8
Bus Line with Three-State Buffers

state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active depending on the binary value in the select inputs of the decoder. Careful investigation will reveal that Figure 7-8 represents another way of constructing a 4 × 1 multiplexer. The diagram shows the inputs marked with a register name and the bit value in parenthesis. Thus, $R0(k)$ designates bit k in register $R0$. The circuit of Figure 7-8 can replace the multiplexer in Figure 7-6 that produces the bus line for bit k. To construct a common bus for four n-bit registers with three-state buffers requires n groups of four buffers each to produce the outputs for the n-bit bus. Only one decoder is necessary to select between the four registers.

There are occasions where it is necessary to employ a bidirectional bus system that can transfer information in both directions. A bidirectional bus allows the binary information to flow in either of two directions. A bidirectional bus can be constructed with three-state buffers to control the direction of information flow in the bus. One line of a bidirectional bus is shown in Figure 7-9. The bus control has two selection lines S_{in} for input transfer and S_{out} for output transfer. These selection lines control two three-state buffers connected back to back. When $S_{in} = 1$ and $S_{out} = 0$, the bottom buffer is enabled and the top buffer is disabled by going to a high-impedance state. This forms a path for input data coming from the bus to pass through the bottom buffer and into the input of a flip-flop register. When $S_{out} = 1$ and $S_{in} = 0$, the top buffer is enabled and the bottom buffer goes to a high-impedance state. This forms a path for output data coming from a register in the system to pass through the upper buffer and out to the bus line. The bus line can be disabled by making both control signals 0. This puts both buffers in a high-impedance state, which prevents any input or output transfer of information

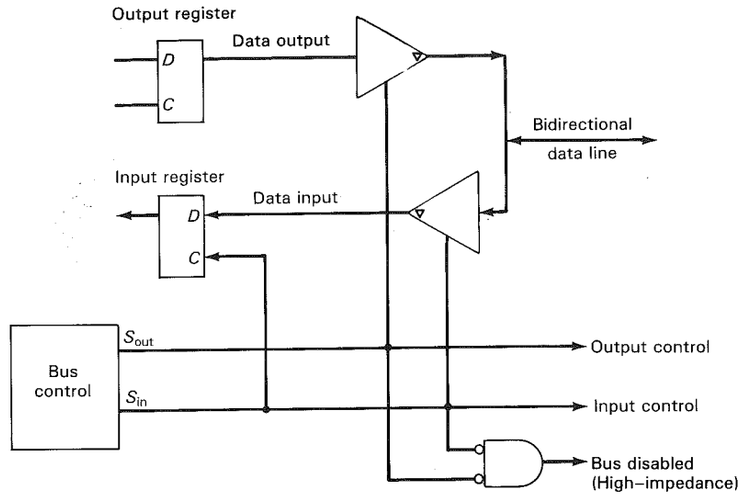


FIGURE 7-9
Bidirectional Bus Line with Three-State Buffers

through the bus line. This condition will exist when an external source is using the common bus to communicate with some other external destination. The two selection lines can be used to inform the external modules connected to the bus of the state which the bidirectional bus is at any given time.

Memory Transfer

The operation of a memory unit was described in Section 6-2. The transfer of information from a memory word to the outside environment is called a read operation. The transfer of new information into a memory word is called a write operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Consider a memory unit that receives the address from a register called address register symbolized by AR. The data is transferred to another register called data register symbolized by DR. The read operation can be stated as follows:

$$\text{Read: } DR \leftarrow M[AR]$$

This causes a transfer of information into DR from the selected memory word specified by the address in AR.

The write operation is a transfer from DR to the selected memory word M. This can be stated symbolically as follows:

$$\text{Write: } M[AR] \leftarrow DR$$

This causes a transfer of information from DR into the memory word selected by the address in AR.

In some systems, the memory unit receives address and data from many registers connected to common buses. Consider the case depicted in Figure 7-10. The address

for the memory comes from an address bus. Four registers are connected to this bus and any one may supply an address. The memory data is connected to a bidirectional data bus. The contents of the selected memory word during a read operation can go to any one of four registers which are selected by a decoder. The data word for the memory during a write operation comes from one of four registers selected by the data bus. The direction of information flow in the data bus is determined from the bus control inputs. For a read operation, the path is from memory to a data register. For a write operation the path is from a data register to memory. Each bidirectional line is controlled by a pair of three-state buffers as indicated in Figure 7-9.

A memory transfer statement in a multiple register system must specify the address register and the data register used. For example, the transfer of information from data register $D2$ to a memory word selected by the address in register $A1$ is symbolized by the statement

$$M[A1] \leftarrow D2$$

This is a write operation with $A1$ supplying the address. The statement does not specify the buses explicitly. Nevertheless, it implies the required selection inputs for the address bus to be 01 (for $A1$) and the register select for write inputs to be 10 (for $D2$). The bus control activates the write and output control signals.

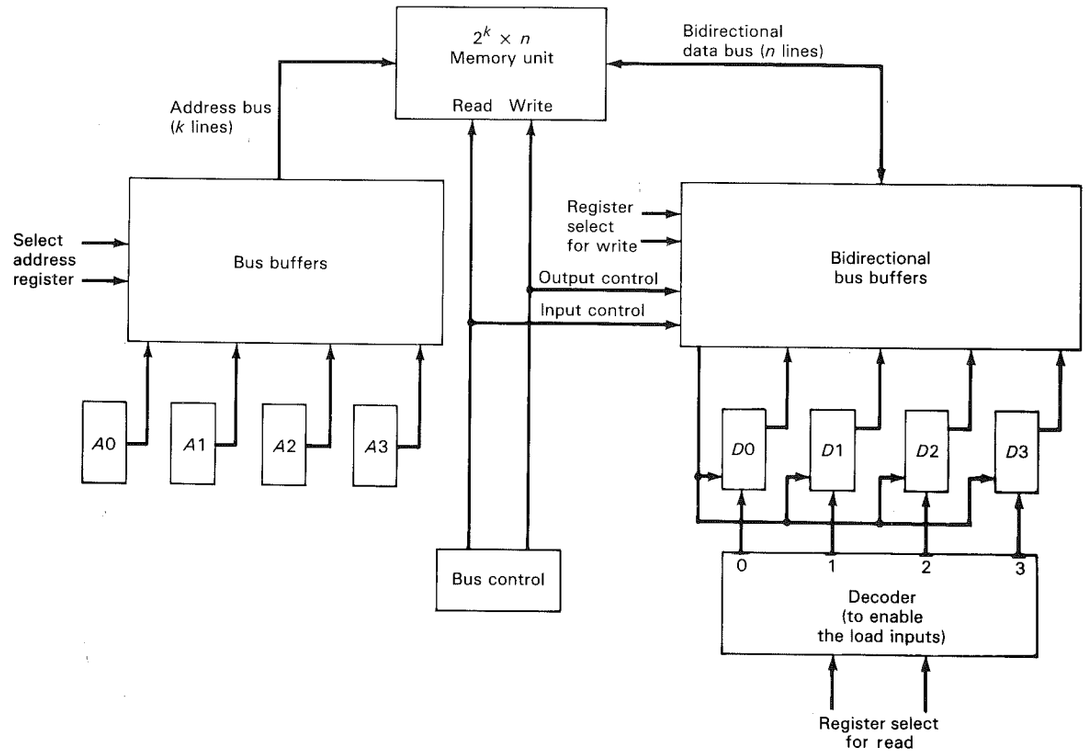


FIGURE 7-10
Memory Unit Connected to Address and Data Buses

The read operation in a memory with buses can be specified in a similar manner. The statement

$$D0 \leftarrow M[A3]$$

symbolizes a read operation from a memory word whose address is given by register $A3$. The information coming out of memory is transferred to data register $D0$. Again, this statement implies the required selection inputs for the address bus to be 11 (for $A3$) and the inputs to the decoder to be 00 (for $D0$) while the bus control activates the read and input control signals.

7-5 PROCESSOR UNIT

The processor unit is a central component in a digital computer system. It consists of a number of registers and the digital circuits that implement various microoperations. The processor part of the computer is sometimes referred to as the *data path* because it forms the paths for the operations among the registers. The various paths are said to be controlled by means of gates that form the required path for each particular operation. In a typical processor unit, the data paths are formed by means of buses and other common lines. The control gates that formulate the given path are essentially multiplexers and decoders whose selection lines specify the required path. The processing of information is done by one common circuit referred to as the arithmetic logic unit, abbreviated ALU.

When a large number of registers are included in a processor unit, it is most efficient to connect them through common buses. The registers communicate with each other not only for direct data transfer, but also while performing various microoperations. A bus organization with four registers and an ALU is shown in Figure 7-11. Each register is connected to two sets of multiplexers to form input buses A and B. The selection inputs in each set of multiplexers select one register for the corresponding bus. The A and B buses are applied to the inputs of a common arithmetic logic unit. The select inputs of the ALU determine the particular operation that is performed. The shift operations are implemented in the shifter. The output data from the ALU may be shifted to the right or to the left, or may go through the shifter without a change, depending on the shift select input. The result of the operation is placed on the output bus which in turn is connected to the inputs of all registers. The destination register that receives the result from the output bus is selected by a decoder. The decoder activates one register load input to provide the transfer path between the data in the output bus and the destination register.

The output bus has additional terminals for transferring data from the processor unit to an external device. External data can enter the processor unit through the input data terminals in one of the multiplexers.

It is sometimes convenient to supplement the ALU with a number of status bits. The status bits are useful for checking certain relationships between the values of A and B after an ALU operation. Four status bits are shown in Figure 7-11. The carry C and the overflow V are explained in conjunction with Figure 7-4. The zero

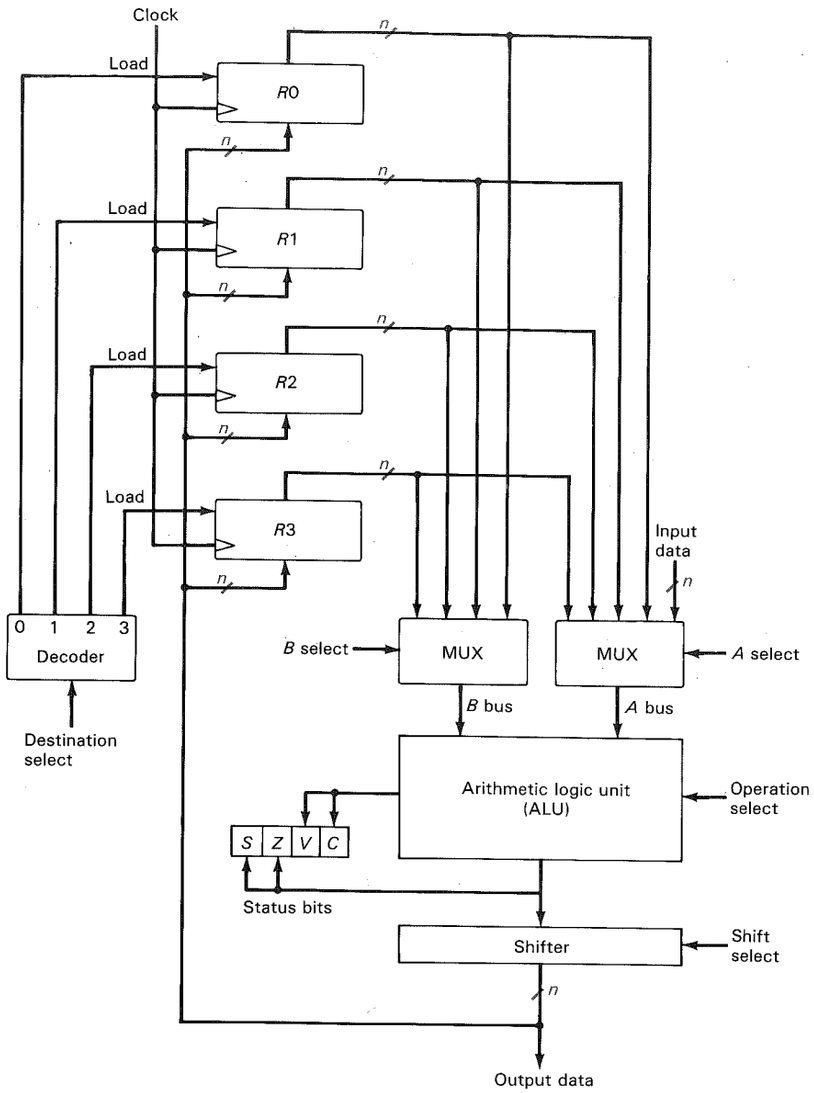


FIGURE 7-11
Block Diagram of a Processor Unit

status bit Z is set to 1 if the output of the ALU contains all 0's, and cleared to 0 otherwise. Thus, $Z = 1$ if the result of an operation is zero, and $Z = 0$ if the result is nonzero. The sign status bit S is set to the value of the sign bit of the result. The sign bit is always the leftmost bit out of the ALU.

A typical processor unit will have more than four registers. Computers with 16 or more registers are quite common. The construction of a bus system with a large number of registers requires larger multiplexers and ALU, otherwise, it is similar to the organization depicted in Figure 7-11.

The control unit that supervises the bus system in the processor must direct the information flow through the buses, the ALU, and the shifter by selecting the various components in the unit. For example, to perform the microoperation

$$R1 \leftarrow R2 + R3$$

the control unit must provide binary selection variables to the following selection inputs:

1. MUX A selector: to place the contents of $R2$ onto bus A.
2. MUX B selector: to place the contents of $R3$ onto bus B.
3. ALU operation selector: to provide the arithmetic operation $A + B$.
4. Shift selector: to provide a direct transfer from the outputs of the ALU onto the output bus (no shift).
5. Decoder destination selector: to load the contents of the output bus into $R1$.

The five sets of selection variables must be generated simultaneously and must be available in the corresponding terminals at the beginning of a clock pulse period. The binary data from the two source registers must propagate through the multiplexers, the ALU, the shifter, and into the inputs of the destination register, all during one clock pulse period. Then, when the next clock edge transition arrives, the binary information on the output bus is loaded into the destination register. To achieve a fast response time, the ALU is constructed with fast circuits and the shifter is implemented with combinational gates.

The operation of the multiplexers, the buses, and the destination decoder was presented in the previous section. The design of an ALU and shifter is undertaken in the next two sections. The control of the processor unit through its selection inputs is demonstrated in Section 7-8.

7-6 ARITHMETIC LOGIC UNIT (ALU)

An arithmetic logic unit (ALU) is a combinational circuit that performs a set of basic arithmetic and logic microoperations. The ALU has a number of selection lines used to select a particular operation in the unit. The selection lines are decoded within the ALU so that k selection variables can specify up to 2^k distinct operations.

Figure 7-12 shows the block diagram of a typical 4-bit ALU. The four data inputs from A are combined with the four data inputs from B to generate an operation at the F outputs. The mode select input S_2 distinguishes between arithmetic and logic operations. The two function select inputs S_1 and S_0 specify the particular arithmetic or logic operation to be generated. With three selection lines, it is possible to specify four arithmetic operations with S_2 in one state and four logic operations with S_2 in the other state. The input and output carries have meaning only during an arithmetic operation. The input carry C_{in} is quite often used as a fourth selection variable for arithmetic operations. In this way, it is possible to double the number of arithmetic operations from four to eight.

The design of a typical ALU will be carried out in three stages. First, the design of the arithmetic section will be undertaken. Second, the design of the logic section

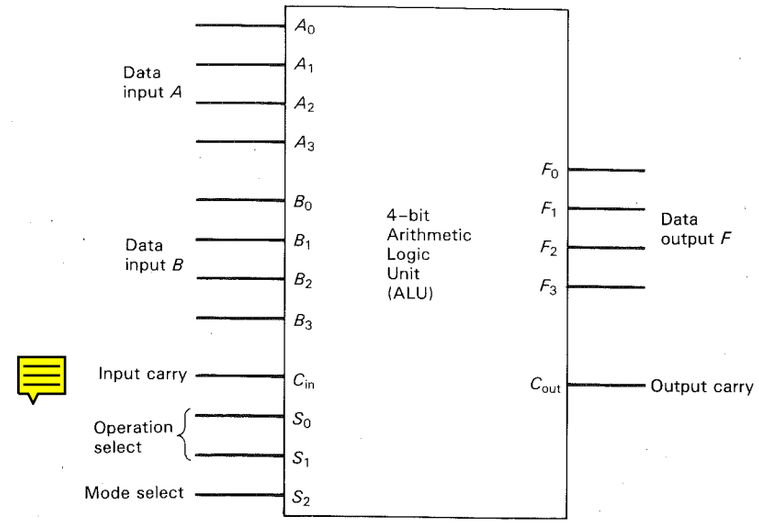


FIGURE 7-12
Block Diagram of a 4-Bit ALU

will be presented. The two sections will then be combined to form the arithmetic logic unit.

Arithmetic Circuit

The basic component of an arithmetic circuit is the parallel adder. A parallel adder is constructed with a number of full-adder circuits connected in cascade (see Figure 3-11). By controlling the data inputs to the parallel adder, it is possible to obtain different types of arithmetic operations. The block diagram of Figure 7-13 demonstrates a possible configuration where one set of inputs to the parallel adder is controlled by two selection lines S_1 and S_0. There are n bits in the arithmetic circuit with two inputs, A and B, and one output F. The n inputs from B go through a combinational circuit to the Y inputs of the parallel adder. The input carry C_{in} goes to the carry input of the full-adder circuit in the least significant bit position. The output carry C_{out} is from the full adder in the most significant position. The output of the parallel adder is calculated from the following arithmetic sum

$$F = A + Y + C_{in}$$

where A is the n-bit binary number at the X inputs and Y is the n-bit binary number at the Y inputs of the parallel adder. C_{in} is the input carry which can be equal to 0 or 1. Note that the symbol + in the above equation denotes an arithmetic plus.

By controlling the value of Y with the two selection inputs S_1 and S_0, it is possible to obtain a variety of arithmetic operations. This is shown in Table 7-5. If the inputs from B are ignored and we insert all 0's into the Y inputs, the output sum becomes $F = A + 0 + C_{in}$. This gives $F = A$ when $C_{in} = 0$ and $F = A + 1$ when $C_{in} = 1$. In the first case we have a direct transfer from input A to output F. In the second case, the value of A is incremented by 1. For a straight arithmetic addition, it is necessary to transfer the B inputs into the Y inputs of the parallel

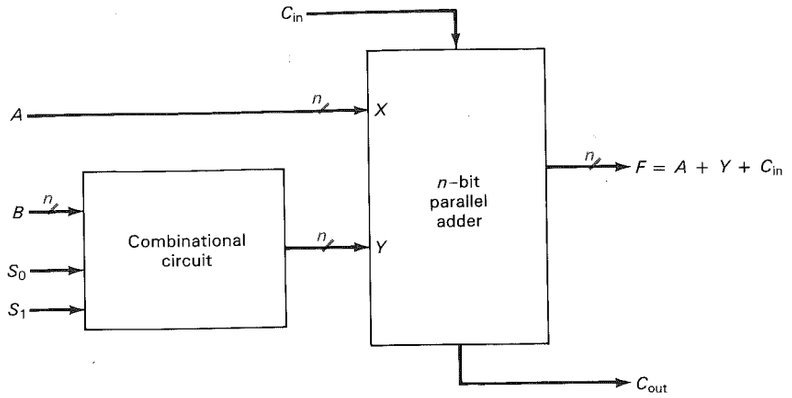


FIGURE 7-13
Block Diagram of an Arithmetic Circuit

adder. This gives $F = A + B$ when $C_{in} = 0$. Arithmetic subtraction is achieved when Y is made the complement of B to obtain $F = A + \overline{B} + 1$ when $C_{in} = 1$. This gives A plus the 2's complement of B which is equivalent to subtraction. Inserting all 1's into the inputs of Y produces the decrement operation $F = A - 1$ when $C_{in} = 0$. This is because a number with all 1's is equal to the 2's complement of 1. For example, the 2's complement of binary 0001 is 1111. Adding a number A to the 2's complement of 1 produces $F = A +$ 2's complement of $1 = A - 1$.

The combinational circuit in Figure 7-13 can be implemented with n multiplexers. The data inputs to each multiplexer in stage i for $i = 0, 1, 2, \ldots, n-1$ are: 0, B_i, $\overline{B}_i$, and 1, corresponding to selection values S_1S_0: 00, 01, 10, and 11, respectively. Thus the arithmetic circuit can be constructed with n multiplexers and n full adders.

The number of gates in the combinational circuit can be reduced if, instead of using multiplexers, we go through the logic design of one stage of the combinational circuit. This can be done as shown in Figure 7-14. The truth table for one typical stage i for the combinational circuit is listed in Figure 7-14(a). The inputs are S_1, S_0, B_i, and the output is Y_i. Following the requirements specified in Table 7-5, we let $Y_i = 0$ when $S_1S_0 = 00$, and similarly assign the other three values of Y_i for each of the combinations of the selection variables. Output Y_i is simplified in the map in Figure 7-14(b).

$$Y_i = B_iS_0 + \overline{B}_iS_1$$

S_1 and S_0 are common to all n stages. Each stage i is associated with input B_i and output Y_i for $i = 0, 1, 2, \ldots, n-1$.

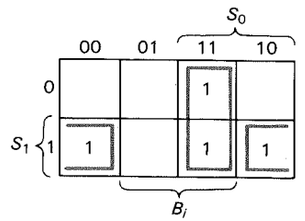
The logic diagram of a 4-bit arithmetic circuit is shown in Figure 7-15. The four

TABLE 7-5
Arithmetic Circuit Function Table

Select		Input	$F = A + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	all 0's	$F = A$ (transfer)	$F = A + 1$ (increment)
0	1	B	$F = A + B$ (add)	$F = A + B + 1$
1	0	$\overline{B}$	$F = A + \overline{B}$	$F = A + \overline{B} + 1$ (subtract)
1	1	all 1's	$F = A - 1$ (decrement)	$F = A$ (transfer)

Inputs			Output	
S_1	S_0	B_i	Y_i	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B}_i$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	

(a) Truth table



(b) Map simplification:
$Y_i = B_iS_0 + \overline{B}_iS_1$

FIGURE 7-14
Combinational Circuit for One Stage of Arithmetic Circuit

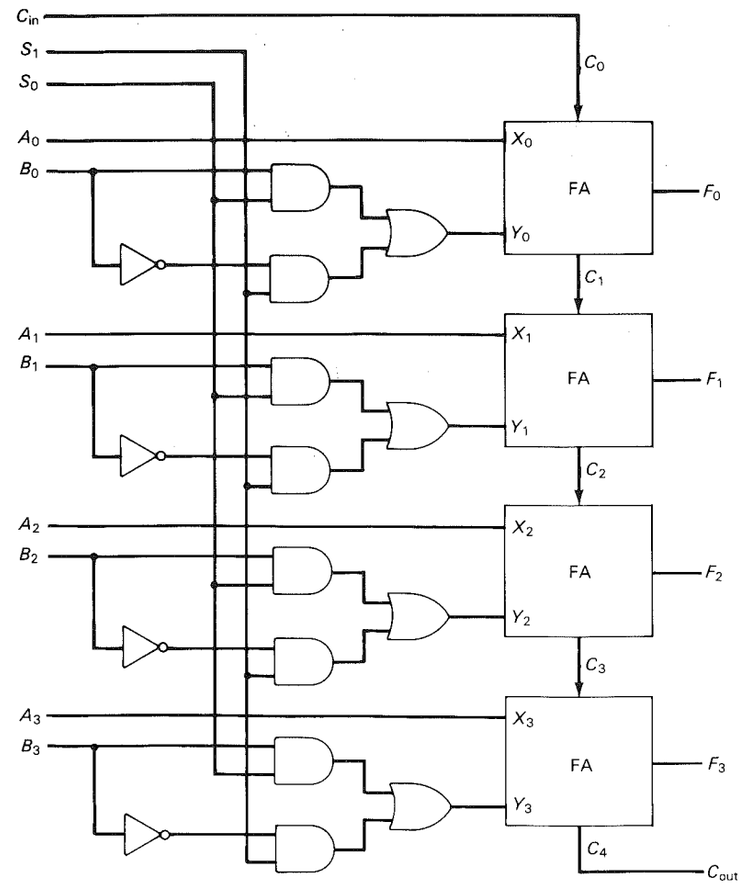


FIGURE 7-15
Logic Diagram of a 4-Bit Arithmetic Circuit

full-adder (FA) circuits constitute the parallel adder. The carry into the first stage is the input carry C_{in}. All other carries are connected internally from one stage to the next. The selection variables are S_1, S_0, and C_{in}. Variables S_1 and S_0 control all Y inputs of the full adders according to the Boolean function derived in Figure 7-14(b). C_{in} adds 1 to the sum when it is equal to 1. The eight arithmetic operations of the circuit as a function of S_1, S_0, and C_{in} are listed in Table 7-5.

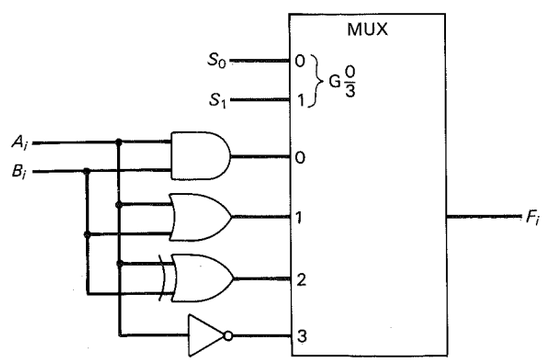
Logic Circuit

The logic microoperations manipulate the bits of the operands by treating each bit in a register as a binary variable. There are four basic logic operations from which all others can be derived. They are the AND, OR, XOR (exclusive-OR), and complement.

Figure 7-16(a) shows one stage of the logic circuit. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to a multiplexer with two selection variables S_1 and S_0. These selection variables choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for $i = 0, 1, 2, \ldots, n-1$. The selection variables are applied to all stages. The function table in Figure 7-16(b) lists the logic operations obtained for each combination of the selection variables.

Arithmetic Logic Unit

The logic circuit can be combined with the arithmetic circuit to produce an arithmetic logic unit. Selection variables S_1 and S_0 can be common to both circuits provided we use a third selection variable to differentiate between the two. The configuration of one stage of the ALU is illustrated in Figure 7-17. The outputs of the arithmetic and logic circuits in each stage are applied to a 2 × 1 multiplexer



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = \overline{A}$	Complement

(b) Function table

FIGURE 7-16
One Stage of a Logic Circuit

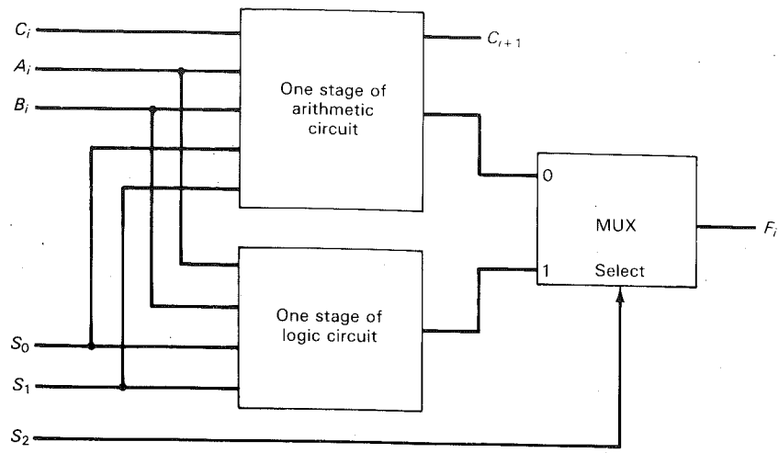


FIGURE 7-17
One Stage of ALU

with selection variable S_2. When $S_2 = 0$, the arithmetic output is selected, and when $S_2 = 1$, the logic output is selected. Note that the diagram shows just one typical stage of the ALU. The circuit of Figure 7-17 must be repeated n times for an n-bit ALU. The output carry C_{i+1} of a given arithmetic stage must be connected to the input carry C_i of the next stage in sequence. The input carry to the first stage is the input carry C_{in} which provides a selection variable for the arithmetic operations.

The ALU specified in Figure 7-17 provides eight arithmetic and four logic operations. Each operation is selected through the variables S_2, S_1, S_0, and C_{in}. The input carry C_{in} is used for selecting an arithmetic operation only.

Table 7-6 lists the 12 operations of the ALU. The first 8 are arithmetic operations and are selected with $S_2 = 0$. The next 4 are logic operations and are selected with $S_2 = 1$. The input carry has no effect during the logic operations and is marked with a 0 for convenience.

TABLE 7-6
Function Table for ALU

Operation Select					
S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	$F = A$	Transfer A
0	0	0	1	$F = A + 1$	Increment A
0	0	1	0	$F = A + B$	Addition
0	0	1	1	$F = A + B + 1$	Add with carry
0	1	0	0	$F = A + \overline{B}$	A plus 1's complement of B
0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	1	1	0	$F = A - 1$	Decrement A
0	1	1	1	$F = A$	Transfer A
1	0	0	0	$F = A \wedge B$	AND
1	0	1	0	$F = A \vee B$	OR
1	1	0	0	$F = A \oplus B$	XOR
1	1	1	0	$F = \overline{A}$	Complement A

7-7 SHIFTER UNIT

The shifter attached to the bus system transfers the output of the ALU onto the output bus. The shifter transfers the information by shifting it to the right or left. Provision must be made for a direct transfer with no shift from the ALU to the output bus. The shifter provides the shift operations commonly not available in the ALU.

An obvious choice for a shifter would be a bidirectional shift register with parallel load. Information from the ALU can be transferred to the register in parallel and then shifted to the right or left. In this type of configuration, a clock pulse is needed for loading the output of the ALU into the shift register, and another pulse is needed for the shift. These two pulses are in addition to the clock pulse required to transfer the information from the shift register to the selected destination register.

The transfer from a source register to a destination register through the ALU and shifter can be done with only one clock pulse when the ALU and shifter are implemented as combinational circuits. In a combinational circuit, the signals propagate through gates without the need of a clock pulse. Hence, the only clock needed in the processor bus system is for loading the data from the output bus into the selected destination register.

A combinational circuit shifter unit can be constructed with multiplexers as shown in Figure 7-18. The two selection variables H_1 and H_0 are applied to all four multiplexers to select the type of operation in the shifter. With $H_1H_0 = 00$, no

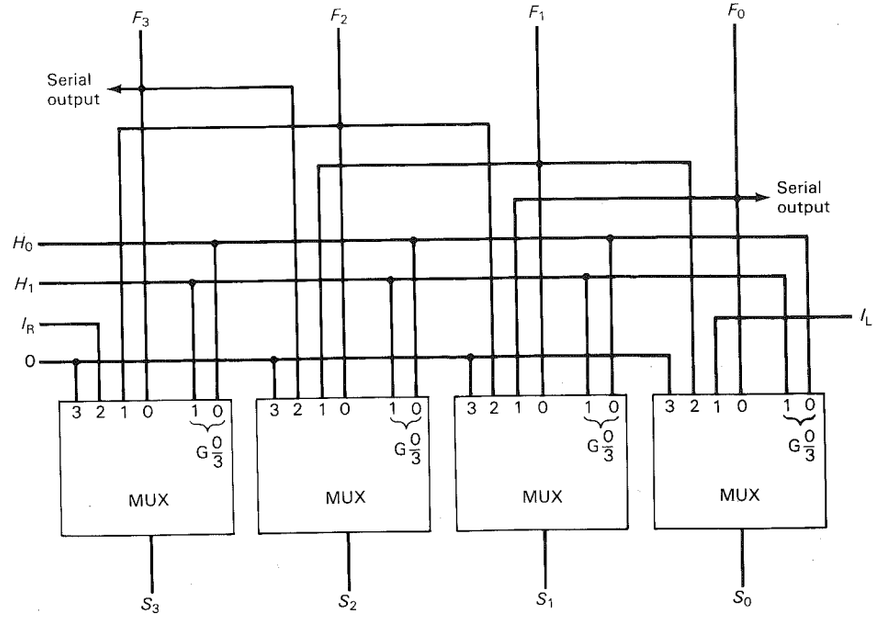


FIGURE 7-18
4-Bit Shifter

TABLE 7-7
Function Table for Shifter

H_1	H_0	Operation	Function
0	0	$S \leftarrow F$	Transfer F to S (no shift)
0	1	$S \leftarrow$ shl F	Shift left F into S
1	0	$S \leftarrow$ shr F	Shift right F into S
1	1	$S \leftarrow 0$	Transfer 0's into S

shift is executed and the inputs from F go directly to the shifter outputs in S. The next two values of the selection variables cause a shift left and a shift right operation. When $H_1H_0 = 11$, the multiplexers select the inputs attached to logic-0 and the S outputs are all equal to 0. Table 7-7 summarizes the operation of the shifter.

The diagram of Figure 7-18 shows only four stages of the shifter. The shifter, of course, must have n stages in a system with n parallel lines. Inputs I_R and I_L are the serial inputs for the shift right and shift left, respectively. Another selection variable may be employed to specify what goes into I_R and I_L during the shift. For example, a third selection variable, H_2, when in one state can select a 0 for the serial input during the shift. When H_2 is in the other state, the data can be rotated together with the value in the carry status bit. In this way, a carry produced during an ALU addition operation can be shifted to the right and into the most significant bit position of a register.

Barrel Shifter

In some processor applications the data must be shifted more than once during a single operation. A barrel shifter is a circuit that shifts the input data bits by a number of positions dictated by the binary value of a set of selection lines. The shift is a cyclic rotation which means that the input binary information is shifted in one direction and the bit coming from the most significant end of the shifter is brought back to the least significant position.

A 4-bit barrel shifter is shown in Figure 7-19. It consists of four multiplexers with two common selection lines S_1 and S_0. The selection variables determine the number of positions that the input data will be shifted to the left by rotation. When $S_1S_0 = 00$, no shift occurs and the inputs form a direct path to the outputs. When $S_1S_0 = 01$, the input binary information is rotated once with D_0 going to Y_1, D_1 going to Y_2, D_2 going to Y_3, and D_3 circulated back into Y_0. When $S_1S_0 = 10$, the input is rotated by two positions, and when $S_1S_0 = 11$, the rotation is by three bit positions.

Table 7-8 gives the function table for the 4-bit barrel shifter. For each binary value of the selection variables, the table lists the inputs that go to the corresponding output. Thus, to rotate three positions, S_1S_0 must be equal to 11; and D_0 goes to Y_3, D_1 goes to Y_0, D_2 goes to Y_1, and D_3 goes to Y_2.

A barrel shifter with 2^n input and output lines requires 2^n multiplexers each having 2^n data inputs and n selection inputs. The number of rotation positions is dictated by the selection variables and can be from 0 (no shift) to $2^n - 1$ positions.

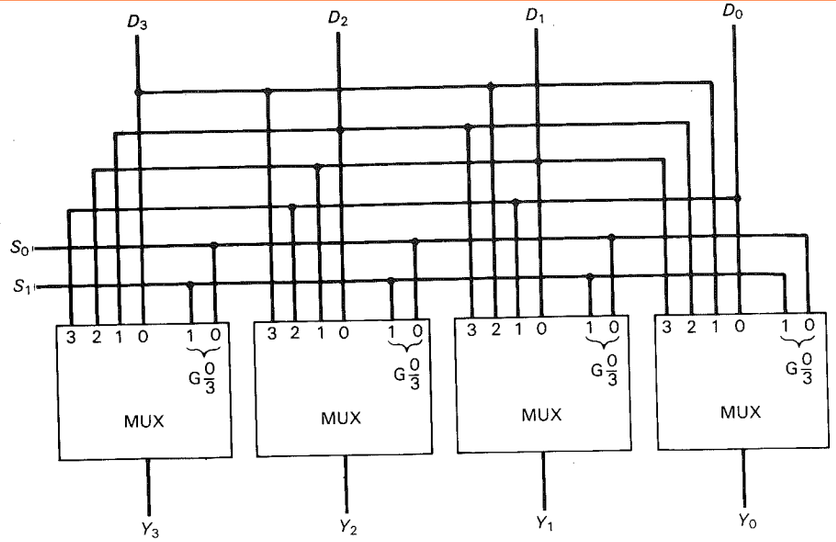


FIGURE 7-19
4-Bit Barrel Shifter

TABLE 7-8
Function Table for 4-Bit Barrel Shifter

Select		Output				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	Operation
0	0	D_3	D_2	D_1	D_0	No shift
0	1	D_2	D_1	D_0	D_3	Rotate once
1	0	D_1	D_0	D_3	D_2	Rotate twice
1	1	D_0	D_3	D_2	D_1	Rotate three times

7-8 CONTROL WORD

The selection variables in a processor unit control the microoperations executed within the unit during any given clock pulse transition. The selection variables control the buses, the ALU, the shifter, and the destination register. We will now demonstrate by means of an example how the control variables select the microoperations. The example defines a specific processor together with its selection variables. The choice of control variables for some typical microoperations will then be demonstrated.

A block diagram of a processor unit is shown in Figure 7-20. It has a register file of seven registers $R1$ through $R7$. The outputs of the seven registers go through two sets of multiplexers to select the inputs to the ALU. Input data from an external source (such as a memory unit) are selected by the same multiplexers. The output

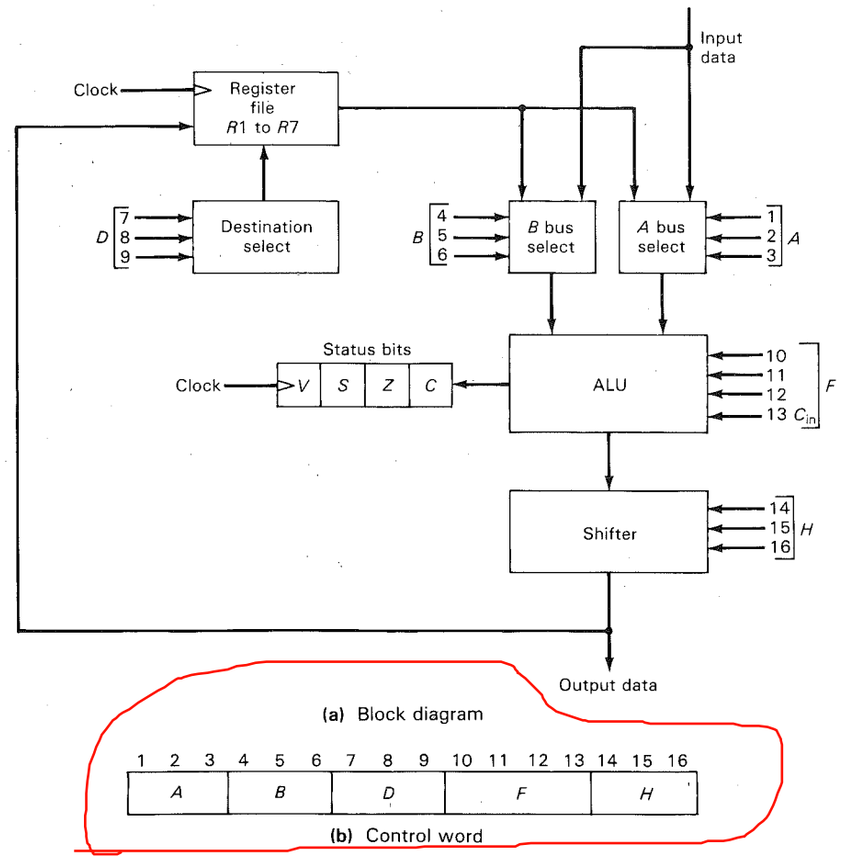


FIGURE 7-20
Processor Unit with Control Variables

of the ALU goes through a shifter and into the output bus. The output from the shifter can be transferred to any one of the registers and can also be directed to an external destination. The ALU provides the binary data for the four status bits: C (carry), Z (zero), S (sign), and V (overflow).

There are 16 binary selection inputs in the unit and their combined value specifies a *control word*. The 16-bit control word is defined in Figure 7-20(b). It consists of five parts called fields, with each field designated by a letter symbol. Four of the fields contain three bits each, and one field has four bits. The three bits of A select a source register for one input of the ALU. The three bits of B select a register for the second input of the ALU. The three bits of D select a destination register. The four bits of F select one of 12 operations in the ALU. The three bits of H select the type of shift in the shifter unit. The 16-bit control word, when applied to the selection inputs, specifies a particular microoperation.

TABLE 7-9
Encoding of Control Word for Processor Unit

Binary code	Function of selection fields					
	A	B	D	F with $C_{in} = 0$	F with $C_{in} = 1$	H
000	Input	Input	None	$F = A$	$F = A + 1$	No shift
001	$R1$	$R1$	$R1$	$F = A + \overline{B}$	$F = A + B + 1$	SHL
010	$R2$	$R2$	$R2$	$F = A + \overline{B}$	$F = A - B$	SHR
011	$R3$	$R3$	$R3$	$F = A - 1$	$F = A$	Bus = 0
100	$R4$	$R4$	$R4$	$F = A \wedge B$	—	—
101	$R5$	$R5$	$R5$	$F = A \vee B$	—	ROL
110	$R6$	$R6$	$R6$	$F = \overline{A} \oplus B$	—	ROR
111	$R7$	$R7$	$R7$	$F = \overline{A}$	—	—

The functions of all selection variables are specified in Table 7-9. The 3-bit binary code listed in the first column of the table specifies the binary code for each of the five fields. The register selected by fields A, B, and D is the one with the decimal number equivalent to the binary number in the code. When the A or B field is 000, the corresponding multiplexer selects the external input data. When D = 000, no destination register is selected but the contents of the output bus are available for the external outputs. The encoding of the F field is dependent on the value of the input carry C_{in}. The four bits in the F field for each operation are determined from the three bits in the corresponding first column of the table and the bit of C_{in}. The operations listed for the ALU are the same as the ones specified in Table 7-6.

The first four entries for the code in the H field specify the shift operations of Table 7-7. The H field has a third selection variable to distinguish between a logical shift and a shift by rotation. The SHL (shift left) and SHR (shift right) operations listed in the table are logical shifts, which means that 0 is applied to their corresponding serial inputs. When the third selection variable in the H field is equal to 1, it indicates a shift by rotation. This is designated in the table by the symbols ROL (rotate left) and ROR (rotate right). The ROL operation shifts the input data to the left and inserts the value of the serial output into the serial input. The same applies to the ROR operation except that the shift is to the right.

A control word of 16 bits is needed to specify a microoperation for the processor. The control word for a given microoperation can be derived from the selection variables. For example, the subtract microoperation given by the statement

$$R1 \leftarrow R2 - R3$$

specifies $R2$ for the A input of the ALU, $R3$ for the B input of the ALU, an ALU operation $F = A - B$, no shift for the shifter, and $R1$ for the destination register. Thus, the control word is specified by the five fields and the corresponding binary value for each field is obtained from the encoding listed in Table 7-9. The binary control word for the subtract microoperation is 0100110010101000 and is obtained as follows:

TABLE 7-10
Examples of Microoperations for the Processor

Microoperation	Symbolic designation					Control word				
	A	B	D	F	H	A	B	D	F	H
$R1 \leftarrow R2 - R3$	$R2$	$R3$	$R1$	$F = A - B$	No shift	010	011	001	0101	000
$R4 \leftarrow$ shr $(R5 + R6)$	$R5$	$R6$	$R4$	$F = A + B$	SHR	101	110	100	0010	010
$R7 \leftarrow R7 + 1$	$R7$	—	$R7$	$F = A + 1$	No shift	111	000	111	0001	000
$R1 \leftarrow R2$	$R2$	—	$R1$	$F = A$	No shift	010	000	001	0000	000
Output $\leftarrow R3$	$R3$	—	None	$F = A$	No shift	011	000	000	0000	000
$R4 \leftarrow$ rol $R4$	$R4$	—	$R4$	$F = A$	ROL	100	000	100	0000	101
$R5 \leftarrow 0$	—	—	$R5$	—	Bus = 0	000	000	101	0000	011

Field:	A	B	D	F	H
Symbol:	$R2$	$R3$	$R1$	$F = A - B$	No shift
Control word:	010	011	001	0101	000

The control word for this microoperation and few others are listed in Table 7-10.

The next example in the table is an add and shift microoperation given by the statement

$$R4 \leftarrow \text{shr } (R5 + R6)$$

It specifies an arithmetic addition for the ALU and a shift right operation for the shifter. The sum that is generated in the ALU is shifted to the right, dividing it by 2. The result of the operation is to produce the average value of the two unsigned binary numbers stored in registers $R5$ and $R6$ (provided there is no carry). From knowledge of the symbols in each field, the control word in binary is derived as shown in Table 7-10.

The increment and transfer microoperations do not use the B input of the ALU. For these cases, the B field is marked with a dash. For convenience, we assign 000 to any unused field when formulating the binary control word, although any other binary number may be used. To put the contents of a register into the output terminals, we place the contents of the register into the A input of the ALU, but none of the registers is selected to accept the data. The ALU operation $F = A$ places the information from the register into the output terminals. To shift or rotate the contents of a register without performing an operation in the ALU, we must specify an ALU operation $F = A$. This places the contents of the A input into the F output of the ALU without any change. To clear a register to 0, the output bus is made equal to all 0's with H = 011. The destination field D is made equal to the code for the register.

It is apparent from these examples that many other microoperations can be generated in the processor unit. The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a *control memory*. The output of the control memory is applied to the selection inputs of the processor. By reading consecutive control words from memory, it is possible to initiate the desired sequence of microoperations for the processor. This type of control organization, called *microprogramming*, is discussed in more detail in the next chapter.

REFERENCES

1. MANO, M. M. *Computer System Architecture*. Englewood Cliffs: Prentice-Hall, 1982.
2. BOOTH, T. L. *Introduction to Computer Engineering Hardware and Software Design*. 3rd ed. New York: Wiley, 1984.
3. DIETMEYER, D. L. *Logic Design of Digital Systems*. 2nd ed. Boston: Allyn & Bacon, 1978.
4. GOSLING, J. B. *Design of Arithmetic Units for Digital Computers*. New York: Springer-Verlag, 1980.
5. ERCEGOVAC, M., AND LANG, T. *Digital Systems and Hardware/Firmware Algorithms*. New York: Wiley, 1985.
6. KLINE, R. M. *Structured Digital Design*. Englewood Cliffs: Prentice-Hall, 1983.

PROBLEMS

7-1 Show the block diagram of the hardware that implements the following register transfer statement:

$$T_3\text{: } R2 \leftarrow R1,\ R1 \leftarrow R2$$

7-2 The outputs of four registers $R0$, $R1$, $R2$, and $R3$ are connected through multiplexers to the inputs of a fifth register $R5$. Each register is eight bits long. The required transfers are dictated by four timing variables T_0 through T_3 as follows:

$$T_0\text{: } R5 \leftarrow R0$$

$$T_1\text{: } R5 \leftarrow R1$$

$$T_2\text{: } R5 \leftarrow R2$$

$$T_3\text{: } R5 \leftarrow R3$$

The timing variables are mutually exclusive and only one variable can be equal to 1 at any given time while the other three are equal to 0. Draw a block diagram showing the hardware implementation of the register transfers. Include the connections necessary from the four timing variables to the selection lines of the multiplexers and to the load input of register $R5$.

7-3 Using two 4-bit registers $R1$ and $R2$, a quadruple 2-to-1-line multiplexer with enable, and four inverters, draw the circuit diagram that implements the following statements:

$T_1\text{: } R2 \leftarrow R1$	Transfer $R1$ to $R2$
$T_2\text{: } R2 \leftarrow \overline{R2}$	Complement $R2$
$T_3\text{: } R2 \leftarrow 0$	Clear $R2$ synchronous with the clock

The three timing variables T_1 through T_3 are mutually exclusive and only one variable can be equal to 1 at any given time. Use standard graphic diagrams as in Figure 5-3(b) for the registers and Figure 3-31(b) for the multiplexer.

7-4 Draw the block diagram for the hardware that implements the following statement:

$$XT_1 + YT_2\text{: } AR \leftarrow AR + BR$$

Where AR and BR are two n-bit registers, X and Y are control variables, and T_1 and T_2 are timing variables. Include the logic gates for the control function. (Remember that the symbol + designates an OR operation in a control or Boolean function, but it represents an arithmetic plus in a microoperation.)

7-5 Consider the following register transfer statements for the two 4-bit registers $R1$ and $R2$.

$$XT\text{: } R1 \leftarrow R1 + R2$$

$$\overline{X}T\text{: } R1 \leftarrow R2$$

When timing variable $T = 1$, the contents of $R2$ are added to the contents of $R1$ if $X = 1$, or the contents of $R2$ are transferred to $R1$ if $X = 0$. Draw a logic diagram showing the implementation of the two statements. Use standard graphic symbols for the 4-bit registers (Figure 5-3[b]), the 4-bit adder (Figure 3-28), and the multiplexer that selects the inputs to $R1$ (Figure 3-31[b]). In the diagram, show how the control variables X and T select the inputs to $R1$.

7-6 Using a 4-bit counter with parallel load as in Figure 5-19 and a 4-bit adder as in Figure 3-28, draw the logic diagram with standard graphic symbols that implements the following statements:

$T_1\text{: } R1 \leftarrow R1 + R2$	Add $R2$ to $R1$
$\overline{T}_1T_2\text{: } R1 \leftarrow R1 + 1$	Increment $R1$

7-7 A digital system has three registers: AR, BR, and PR. Three flip-flops provide the control variables for the system: S is a flip-flop which is set by an external start signal to start the system operation; F and R are two flip-flops used for sequencing the microoperations when the system is in operation. A fourth flip-flop D is set by the system when the operation is completed. The operation of the digital system is described by the following register transfer statements:

$$S\text{: } PR \leftarrow 0,\ S \leftarrow 0,\ D \leftarrow 0,\ F \leftarrow 1$$

$$F\text{: } F \leftarrow 0, \text{ if } (AR = 0) \text{ then } (D \leftarrow 1) \text{ else } (R \leftarrow 1)$$

$$R\text{: } PR \leftarrow PR + BR,\ AR \leftarrow AR - 1,\ R \leftarrow 0,\ F \leftarrow 1$$

(a) Show that the digital system multiplies the contents of registers AR and BR and places the product in register PR.
(b) Draw a block diagram of the hardware implementation. Include a start input signal to set flip-flop S and a done output signal from flip-flop D.

7-8 Assume that registers $R1$ and $R2$ in Figure 7-4 hold two n-bit unsigned numbers. When the select input X is equal to 1, the adder-subtractor circuit performs the arithmetic operation $R1$ + 2's complement of $R2$. This sum and the output carry C_n are transferred into $R1$ and C when $T_1 = 1$ and the clock goes through a positive transition.
(a) Show that if $C = 1$, then the value transferred to $R1$ is equal to $R1 - R2$. But if $C = 0$, the value transferred to $R1$ is the 2's complement of $(R2 - R1)$. (See Section 1-4 and example 1-9 for the procedure of subtracting unsigned numbers with 2's complement.)
(b) Indicate how the value in the C bit can be used to detect a borrow after the subtraction of two unsigned numbers.

7-9 Convert the following decimal numbers to signed binary with eight bits each (including the sign). Perform the operation with the signed binary numbers and check the last

two carries. State whether the carries indicate an overflow. Remember that 8-bit numbers have a capacity range from +127 to −128.
(a) (+65) + (+36) (c) (−36) + (−90)
(b) (+65) − (−90) (d) (−65) − (+90)

7-10 Perform the logic AND, OR, and XOR with the two 8-bit numbers 10011100 and 10101010.

7-11 Given the 16-bit value 01011010 11000011. What operation must be performed
(a) to clear to 0 the last eight bits?
(b) to set to 1 the first eight bits?
(c) to complement the middle eight bits?

7-12 The following logic microoperation is performed with registers AR and BR.

$$AR \leftarrow AR \wedge \overline{BR}$$ Complement BR and AND to AR

Determine how the bits in AR are manipulated by the bits in BR.

7-13 Starting from the eight bits 10110010, show the values obtained after each of the shift microoperations listed in Table 7-4.

7-14 Show that the statement

$$R1 \leftarrow R1 + R1$$

is the same as the logical shift left of register $R1$.

7-15 (a) Represent the number −26 as an 8-bit signed number. Show how the number can be multiplied and divided by 2 by means of the arithmetic shift microoperation. Indicate if there is an overflow.
(b) Repeat with the equivalent binary number −94 and +94.

7-16 Let S_1 and S_0 be the selection variables for the multiplexer in the bus system of Figure 7-6 and D_1 and D_0 be the selection variables for the destination decoder.
(a) Determine the transfer that occurs when the control variables $S_1S_0D_1D_0$ equal (1) 0001; (2) 0110; (3) 1111.
(b) Give the 4-bit selection for the following transfers: (1) $R0 \leftarrow R1$; (2) $R2 \leftarrow R1$; (3) $R3 \leftarrow R2$.

7-17 Draw a diagram of a bus system similar to the one shown in Figure 7-6 but use three-state buffers and a decoder instead of the multiplexers.

7-18 The following memory transfers are specified for the system of Figure 7-10.
(a) $D2 \leftarrow M[A3]$ (b) $M[A1] \leftarrow D0$
Specify the memory operation and determine the binary selection variables for the two bus buffers and the decoder.

7-19 A processor unit like the one in Figure 7-11 has 30 registers. How many selection lines are needed for each set of multiplexers and for the decoder?

7-20 Given an 8-bit ALU with outputs F_0 through F_7 and carries C_7 and C_8, show the logic circuit for setting the four status bits, C (carry), V (overflow), Z (zero), and S (sign).

7-21 Assume that the 4-bit ALU of Figure 7-12 is enclosed in one IC package. Show the connections among two such ICs to form an 8-bit ALU.

7-22 Design an arithmetic circuit with one selection variable S and two n-bit data inputs A and B. The circuit generates the following four arithmetic operations in conjunction with the input carry C_{in}. Draw the logic diagram for the first two stages.

S	$C_{in} = 0$	$C_{in} = 1$
0	$F = A + B$ (add)	$F = A + 1$ (increment)
1	$F = A - 1$ (decrement)	$F = A + \overline{B} + 1$ (subtract)

7-23 Show the logic diagram of a 4-bit arithmetic circuit that performs the operations specified in Table 7-5. Use four multiplexers and four full adders.

7-24 Design a four bit arithmetic circuit with two selection variables S_1 and S_0 that generates the following arithmetic operations. Draw the logic diagram of one typical stage.

S_1	S_0	$C_{in} = 0$	$C_{in} = 1$
0	0	$F = A + B$ (add)	$F = A + B + 1$
0	1	$F = A$ (transfer)	$F = A + 1$ (increment)
1	0	$F = \overline{B}$ (complement)	$F = \overline{B} + 1$ (negate)
1	1	$F = A + \overline{B}$	$F = A + \overline{B} + 1$ (subtract)

7-25 Design an arithmetic circuit with one selection variable S and two data inputs A and B. When $S = 0$, the circuit performs the addition operation $F = A + B$. When $S = 1$, the circuit performs the increment operation $F = A + 1$.

7-26 Input X_i and Y_i of each full-adder circuit in an arithmetic circuit has the digital logic specified by the following Boolean functions

$$X_i = A_i \qquad Y_i = \overline{B}_iS + B_i\overline{C}_{in}$$

where S is a selection variable, C_{in} is the input carry, and A_i and B_i are the input data in stage i.
(a) Draw the logic diagram of the 4-bit arithmetic circuit.
(b) Determine the four arithmetic operations performed when S and C_{in} are equal to 00, 01, 10, and 11.

7-27 Design a digital circuit that performs the four logic operations of exclusive-OR, exclusive-NOR, NOR and NAND. Use two selection variables. Show the logic diagram of one typical stage.

7-28 Add another multiplexer to the shifter of Figure 7-18 with two separate selection lines G_1 and G_0. This multiplexer is used to specify the serial input I_R during the shift right operation in the following manner:

G_1	G_0	Operation
0	0	Logical shift right ($I_R = 0$)
0	1	Rotate right ($I_R = F_0$)
1	0	Rotate right with carry ($I_R = C$, $C = F_0$)
1	1	Arithmetic shift right ($I_R = F_3$)

Show the connection of the multiplexer to the shifter.

7-29 The shift select H field as defined in Table 7-9 has three variables H_2, H_1, and H_0. The last two selection variables are used for the shifter as specified in Table 7-7. Design the circuit associated with selection variable H_2 that causes a rotation.

7-30 Obtain the function table of a 4-bit barrel shifter that rotates the bits to the right. (Table 7-8 assumes a rotation to the left.) Draw the circuit with four multiplexers.

7-31 Obtain the function table of an 8-bit barrel shifter that rotates the bits to the left.

7-32 Specify the control word that must be applied to the processor of Figure 7-20 to implement the following microoperations:

(a) $R2 \leftarrow R1 + 1$
(b) $R3 \leftarrow R4 + R5$
(c) $R6 \leftarrow \overline{R6}$
(d) $R7 \leftarrow R7 - 1$
(e) $R1 \leftarrow \text{shl } R1$
(f) $R2 \leftarrow \text{ror } R2$
(g) $R5 \leftarrow R3 + R1$
(h) $R6 \leftarrow R7$

7-33 Given the following 16-bit control word for the processor unit of Figure 7-20, determine the microoperation that is executed for each control word.

(a) 001 010 011 0101 000
(b) 110 000 001 0000 001
(c) 010 010 010 1100 000
(d) 000 001 000 1000 000
(e) 111 000 011 0000 110

8 CONTROL LOGIC DESIGN

8-1 INTRODUCTION

The binary information stored in a digital computer can be classified as either data or control information. Data are discrete elements of information that are manipulated to perform arithmetic, logic, shift, and other data processing tasks. These operations are implemented with digital components such as adders, decoders, multiplexers, counters, and shift registers. Control information provides command signals that supervise the various operations in the data section to accomplish the desired data processing tasks. The logic design of a digital system can be divided into two distinct parts. One part is concerned with the design of the digital circuits that perform the data processing operations. The other part is concerned with the design of the control circuit that supervises the operations and determines the sequence in which they are executed.

The timing for all registers in a synchronous digital system is controlled by a master clock generator. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The continuous clock pulses do not change the state of a register unless the register is enabled by a control signal. The binary variables that control the selection inputs of multiplexers and the load inputs of registers are generated in the control subsystem.

The relationship between the control logic and the data processor subsystems in a digital system is shown in Figure 8-1. The data processor part may be a general purpose processor unit with many registers and a common ALU, or it may consist of individual registers and associated digital circuits that perform various micro-

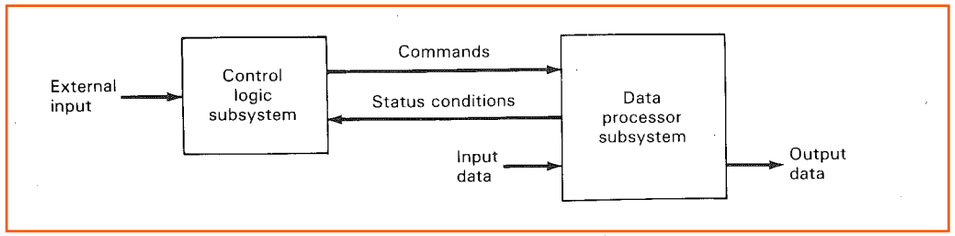


FIGURE 8-1
Control and Data Processor Interaction

operations. The control logic initiates properly sequenced command signals to the data processor. The control logic uses status conditions from the data processor to serve as decision variables for determining the sequence of the controlled operations.

The control logic that generates the signals for sequencing the operations in the data processor is essentially a sequential circuit with internal states that dictate the control commands for the system. At any given time, the state of the sequential control initiates a prescribed set of commands. Depending on status conditions and other external inputs, the sequential control goes to the next state to initiate other operations. The digital circuits that act as the control logic provide a time sequence of signals for initiating the microoperations in the data processor subsystem, and they also determine the next state of the control subsystem itself.

There are two major types of control organization: hardwired control and microprogrammed control. In the *hardwired* organization, the control logic is implemented with gates and flip-flops similar to a sequential circuit. It has the advantage that it can be optimized to produce a fast mode of operation. In the *microprogrammed* organization, the control information such as control functions or control words are stored as 1's and 0's in a special memory. A hardwired control, as the name implies, requires changes in the wiring among the various components if the design has to be modified or corrected. In the microprogrammed control, any required changes or modifications can be done merely by changing the contents of the memory.

In addition to providing the basic principles of both types of control organizations, this chapter presents specific examples that illustrate both methods. The microprogrammed organization is illustrated with an example that controls the operations of a general purpose processor unit. The various ways available for the design of a hardwired control are illustrated with an example of a binary multiplier. The last two sections in the chapter present the organization of a very simple computer and show how the computer can be designed by the hardwired method. The design of a computer central processing unit with a microprogrammed control organization is presented in Chapter 10.

8-2 MICROPROGRAMMED CONTROL

The purpose of the control unit in a digital system is to initiate a series of sequential steps of microoperations. During any given time, certain microoperations are to be initiated while others remain idle. The control variables at any given time can be represented by a string of 1's and 0's called a control word. As such, control

words can be programmed to initiate the various components in the data processor subsystem in an organized manner. A control unit whose binary control variables are stored in a memory is called a *microprogrammed control unit*. Each word in control memory contains within it a *microinstruction*. The microinstruction specifies one or more microoperations for the system. A sequence of microinstructions constitutes the microprogram. Since alterations of the microprogram are not needed once the control unit is in operation, the control memory can be a read only memory (ROM). The use of a microprogram involves placing all control variables in words of ROM for use by the control unit through successive read operations. The content of the word in ROM at a given address specifies the microoperations for the data processor subsystem.

A more advanced development known as dynamic microprogramming permits a microprogram to be loaded initially from the computer console or from an auxiliary memory such as magnetic disk. Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing (to change the microprogram) but is used mostly for reading. A memory which is part of a control unit, is referred to as a *control memory*.

The general configuration of a microprogrammed control unit is demonstrated in the block diagram of Figure 8-2. The control memory is assumed to be ROM, within which all control information is permanently stored. The control memory address register specifies the address of the microinstruction and the control data register holds the microinstruction read from memory. The microinstruction contains a control word that specifies one or more microoperations for the data processor. Once these operations are executed, the control must determine its next address. The location of the next microinstruction may be the next one in sequence, or it may be located somewhere else in the control memory. For this reason, it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the microoperations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction. Thus, a microinstruction contains bits for initiating microoperations in the data processor part and bits that determine the address sequence for the control memory itself.

The next address generator is sometimes called a microprogram *sequencer*, as it determines the address sequence that is read from control memory. The address of the next microinstruction can be specified in several ways depending on the sequencer inputs. Typical functions of a microprogram sequencer are incrementing the control address register by one, loading into the control address register an

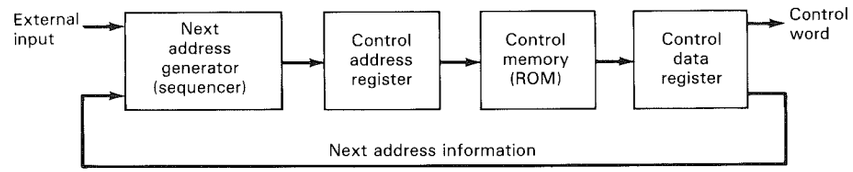


FIGURE 8-2
Microprogrammed Control Organization

address from control memory, transferring an external address, and loading an initial address to start the control operations.

The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is sometimes called a *pipeline register*. It allows the microoperations specified by the control word to be executed simultaneously with the generation of the next microinstruction. This configuration requires a two phase clock, with one phase applied to the address register and the other to the data register.

The system can operate without the control data register by applying a single phase clock to the address register. The control word and next address information are taken directly from the control memory. It must be realized that a ROM operates as a combination circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed, unlike random access memory. Each clock pulse will execute the microoperations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single phase clock, and therefore, we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components, the sequencer and the control memory, are combinational circuits and do not need a clock.

The main advantage of the microprogrammed control is the fact that, once the hardware configuration is established, there should be no need for further hardware or wiring changes. If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for the control memory. The hardware configuration should not be changed for different operations; the only thing that must be changed is the microprogram residing in control memory.

8-3 CONTROL OF PROCESSOR UNIT

To show the general properties of the microprogram organization, we will demonstrate a configuration that includes the control of a processor unit. A general purpose processor unit was developed in Chapter 7 and shown in block diagram in Figure 7-20. It has seven registers $R1$ through $R7$, an ALU, a shifter, and four status bits C, Z, S, and V. A microoperation is selected with a control word of 16 bits which is divided into 5 fields A, B, D, F, and H.

A microprogrammed unit for controlling the processor is shown in Figure 8-3. It has a control memory ROM with 64 words of 26 bits each, a control address register CAR, and two multiplexers MUX1 and MUX2. In order to select between 64 words of memory we need an address of 6 bits. To select among 8 multiplexer inputs we need 3 bits for MUX 2 select. One bit selects between an external address and the address field of the microinstruction with MUX1. Adding the 16 bits for the control word that selects a microoperation in the processor unit gives a total of 26 bits for the microinstruction. Thus, each microinstruction read from control

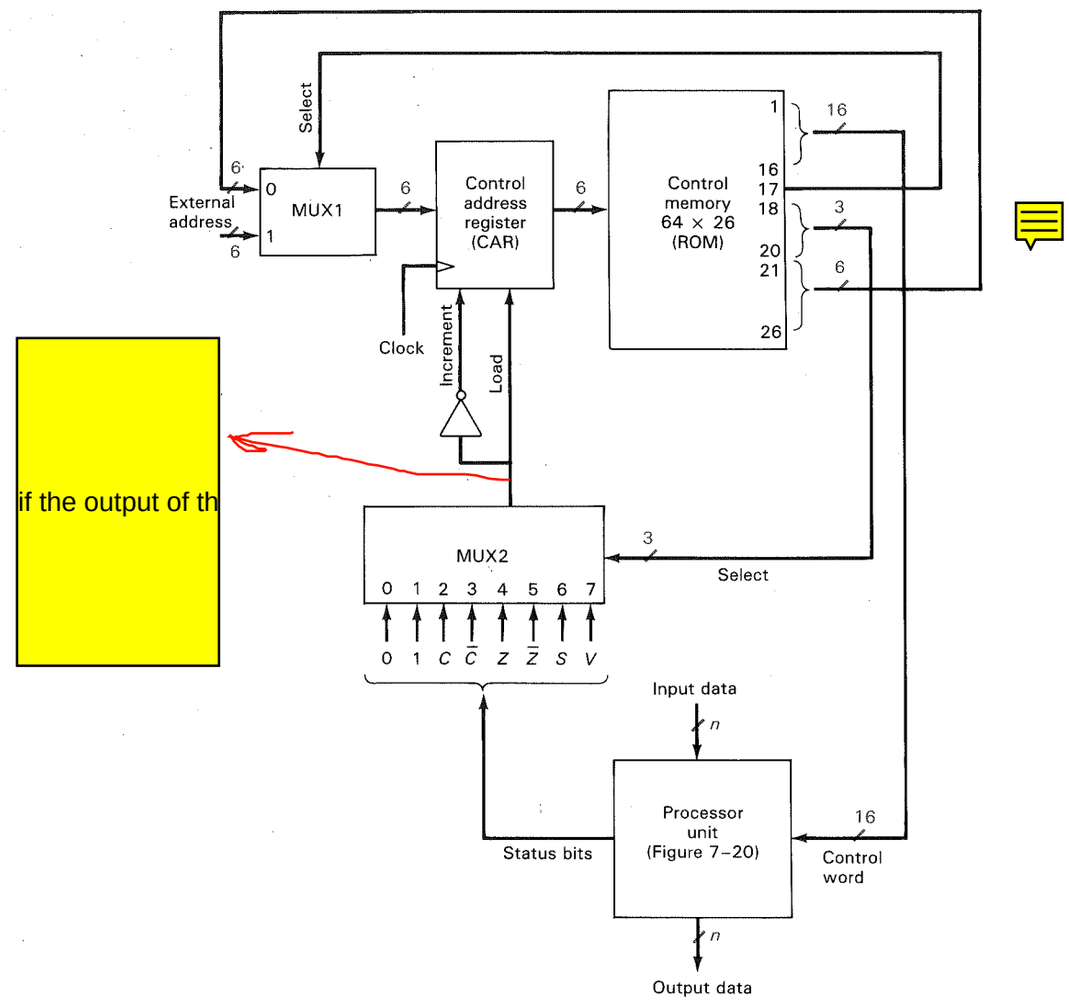


FIGURE 8-3
Microprogrammed Control for Processor Unit

memory contains the 16 bits of a control word for the processor and 10 bits for selecting the next address in the control address register, CAR.

The status bits from the processor unit are applied to the inputs of MUX2. If the selected input is equal to 1, then the load input of CAR is enabled, and the 6-bit address from MUX1 is loaded into CAR. If the selected input is equal to 0, CAR is incremented by one. Thus, bits 18, 19, and 20 of the present microinstruction select one of the inputs from MUX2 and the binary value of this input determines the operation performed on CAR.

Input 0 of MUX2 is connected to a binary constant which is always equal to 0. This input, when selected, generates a 0 output in MUX2 which enables the in-

crement operation in *CAR*. Thus, when MUX2 select bits are equal to 000, *CAR* is incremented by one. Similarly, input 1 of MUX2 is always equal to 1; so when MUX2 select bits are equal to 001, the load input to *CAR* is enabled and a new address is loaded into the register. This causes an unconditional branch to the address given by bits 21 through 26 of the present microinstruction (provided bit 17 is equal to 0). Both the true and complement inputs of *C* and *Z* status bits are applied to MUX2 inputs. In this way, it is possible to specify a branch to a new address depending on whether $C = 1$ or $C = 0$. The other two status bits, *S* and *V*, have only the true value.

The input into *CAR* is a function of bit 17 of the microinstruction. If bit 17 is equal to 0, the address field of the microinstruction is loaded into *CAR*. If bit 17 is equal to 1, an external address is loaded into *CAR*. The external address is for the purpose of initiating a new sequence of microinstructions which can be specified by the external environment.

The operation of the control unit is as follows. At any given clock pulse transition, the control address register *CAR* receives a new address. The 26-bit microinstruction at this address is read from control memory. The control word of the microinstruction specifies a microoperation in the processor and MUX1 and MUX2 determine the next operation for *CAR*. The signals propagate through the combinational circuits such as ROM, ALU, shifter, and multiplexers but the actual microinstruction is not terminated until a clock pulse transition occurs. The next transition of the clock transfers the result of the microoperation into the destination register in the processor, updates the required status bits, and puts a new address into *CAR*. The new address in *CAR* specifies a microinstruction that is read from control memory and is executed with the next transition of the clock. This process repeats for each clock transition. Note that the same clock pulse transition that executes the microoperation in the processor unit also transfers the next address to *CAR*.

Encoding of Microinstructions

The 26 bits of the microinstruction are divided into eight fields. The number of bits in each field and the symbolic name assigned to each is shown in Table 8-1. In order to facilitate writing microprograms we will assign a symbolic name to each binary combination of each field. Fields A, B, and D specify registers *R*1 through *R*7 in the processor unit. Bit combination 000 in fields A and B specifies an input data symbolized by INP. The destination field D does not select a register when its binary code is 000 and we will symbolize that with NONE. The encoding of bits and the corresponding symbols for fields F, H, and MUX2 are specified in the next three tables. MUX1 select can be either 0 or 1. We designate 0 by the symbol INT (for internal address) and 1 by EXT (for external address). The 6-bit address field is designated by the decimal equivalent of the binary address.

The ALU operations are listed in Table 8-2. Each operation is recognized by the 4-bit binary code assigned to it and its corresponding three letter symbol. The table also specifies how the status bits are affected by each operation. When the bit is not affected by the operation we mark it with an N in the table. A Y indicates that the operation updates the corresponding bit according to the result of the

TABLE 8-1
Microinstruction Format

Bits	Field	Symbols	Function
1–3	A	INP, *R*1 – *R*7	Right input of ALU
4–6	B	INP, *R*1 – *R*7	Left input of ALU
7–9	D	NONE, *R*1 – *R*7	Destination register
10–13	F	See Table 8-2	ALU operation
14–16	H	See Table 8-3	Shifter operation
17	MUX1	INT, EXT	MUX1 select (INT = 0, EXT = 1)
18–20	MUX2	See Table 8-4	MUX2 select
21–26	ADRS	Address number	Address field

INP: input data to processor; NONE: none of the registers are selected; INT: internal address field of microinstruction; EXT: external address.

operation. The *Z* (zero) and *S* (sign) bits are affected by all the operations except TSF. The *S* bit after a logic operation is considered as the leftmost bit of the result. The add and subtract operations also update the *C* (carry) and *V* (overflow) bits. There are two transfer operations in the ALU. The one symbolized by TSF (code 0000) does not change the status bits. The other symbolized by TRC (code 0111) resets the *C* bit to 0 and updates the *Z* and *S* bits.

The shifter operations are listed in Table 8-3. The no shift operation is symbolized by NSH and the all zeros output is recognized by the symbol ZERO. The other six operations specify shifts and rotates either to the left or right. SHL and SHR receive 0 in their corresponding serial inputs. ROL and ROR rotate the bit from the serial output into the corresponding serial input. RLC and RRC insert the value of the status bit *C* into the serial input and transfer the serial output bit into the *C* flip-flop. Note that the rotate with carry operations affect the carry status bit.

The binary select input to MUX2 and a symbolic name for each combination are listed in Table 8-4. The binary code 000 chooses a 0 input for the multiplexer which causes *CAR* to be incremented. We designate this condition by NEXT,

TABLE 8-2
ALU Operations (F Field)

Binary code	Symbol	Status bits*				Function
		Z	*S*	*C*	*V*	
0000	TSF	N	N	N	N	Transfer *A*
0001	INC	Y	Y	N	N	Increment *A* by one
0010	ADD	Y	Y	Y	Y	Add *A* + *B*
0101	SUB	Y	Y	Y	Y	Subtract *A* – *B*
0110	DEC	Y	Y	N	N	Decrement *A* by one
0111	TRC	Y	Y	0	N	Transfer *A* and reset carry
1000	AND	Y	Y	N	N	*A* AND *B*
1010	OR	Y	Y	N	N	*A* OR *B*
1100	XOR	Y	Y	N	N	*A* exclusive-OR *B*
1110	COM	Y	Y	N	N	Complement *A*

*N = status bit not affected; Y = status bit affected by the operation; 0 = reset to 0.

TABLE 8-3
Shifter Operations (H field)

Binary code	Symbol	Function
000	NSH	No shift
001	SHL	Shift left with serial input = 0
010	SHR	Shift right with serial input = 0
011	ZERO	All zeros in output of shifter
100	RLC	Rotate left with carry
101	ROL	Rotate left
110	ROR	Rotate right
111	RRC	Rotate right with carry

TABLE 8-4
Select Input to MUX2

Binary code	Symbol	Function
000	NEXT	Go to next address by incrementing CAR
001	LAD	Load address into CAR (branch unconditionally)
010	LC	Load on carry (branch if $C = 1$)
011	LNC	Load on no carry (branch if $C = 0$)
100	LZ	Load on zero (branch if $Z = 1$)
101	LNZ	Load on non zero (branch if $Z = 0$)
110	LS	Load on sign (branch if $S = 1$)
111	LV	Load on overflow (branch if $V = 1$)

branch = jump

meaning that the next microinstruction in sequence will be chosen. The binary code 001 chooses a 1 input for the multiplexer which causes the loading of the input address into *CAR*. This is designated by the symbol LAD and is essentially an unconditional branch operation. The other six inputs to MUX2 cause a branching to the specified address depending on a given status bit condition or its complement.

Examples of Microinstructions

Before going through an actual microprogram for the control unit it will be instructive to show examples of a few typical microinstructions. Table 8-5 lists six examples of microinstructions in symbolic form and also gives the corresponding 26-bit binary equivalent. The *CAR* address is the address where the microinstruction is assumed to be stored in control memory. The symbols on top of the table are the eight fields of the microinstruction format.

The first example is for a microinstruction stored in address 36. The ALU operation in the F field specifies a logical AND which is performed with the contents of register *R*1 (field A) and register *R*2 (field B). The result of the operation is transferred into register *R*1 (field D). MUX2 select has the symbol NEXT meaning that *CAR* will be incremented by one. The address field and the MUX1 select do not contribute to the selection of the next address and therefore, these two fields

example of some microistructions

TABLE 8-5
Examples of Microinstructions (Not a Microprogram)

CAR address	Control Word A	B	D	F	H	Select MUX1	MUX2	Address field
36	*R*1	*R*2	*R*1	AND	NSH	—	NEXT	—
	001	010	001	1000	000	0	000	000000
40	*R*3	—	*R*3	DEC	NSH	INT	LAD	43
	011	000	011	0110	000	0	001	101011
45	—	—	NONE	TSF	NSH	—	NEXT	—
	000	000	000	0000	000	0	000	000000
52	—	—	*R*4	TSF	ZERO	INT	LS	37
	000	000	100	0000	011	0	110	100101
56	*R*5	—	*R*5	TSF	SHL	INT	LNC	62
	101	000	101	0000	001	0	011	111110
59	*R*6	*R*7	*R*2	ADD	NSH	EXT	LAD	—
	110	111	010	0010	000	1	001	000000

are marked with a dash. The microinstruction can be specified with the following register transfer statement:

$$R1 \leftarrow R1 \wedge R2,\ CAR \leftarrow CAR + 1$$

The binary values for the microinstruction are listed under the symbols in each field. These values are obtained from Tables 8-1 through 8-4. The fields marked with a dash are not used, and therefore, any binary number can be substituted for the dash. For convenience and consistency we will mark these places with 0's. The 26-bit word that must be stored in control memory at address 36 is 001 010 001 1000 000 0 000 000000.

The second microinstruction in Table 8-5 resides in address 40. The equivalent register transfer statement for this microinstruction is

$$R3 \leftarrow R3 - 1,\ CAR \leftarrow 43$$

It decrements *R*3 and branches to the microinstruction at address 43. The branching to address 43 is specified by the LAD and INT symbols in MUX2 and MUX1 fields.

The microinstruction at address 45 is an example where the control word specifies no operation for the processor. The TSF in the ALU field does not change the status bits. The NONE in the D field results in no change in the processor registers. The NSH in the H field does no shifting in the shifter. If we choose 0's for the unused fields, then the control word will consist of 16 0's. This will not produce any operation in the processor but the input data will be applied to the output data. A binary microinstruction with 26 0's produces no operation in the processor and increments CAR by one. This is equivalent to a no operation microinstruction.

The microinstruction at address 52 performs the following operations:

$$R4 \leftarrow 0,\ \text{if } (S = 1) \text{ then } (CAR \leftarrow 37) \text{ else } (CAR \leftarrow CAR + 1)$$

The ZERO in the H field produces an all 0's in the shifter and this output is

transferred into $R4$. The TSF operation is required to make sure that none of the status bits are affected. The symbol LS in MUX2 selects the S (sign) status bit from the processor. If $S = 1$, then the address field of the microinstruction, being the binary equivalent of 37 in this case, is loaded into CAR. If $S = 0$, CAR is incremented. This is the way that all branch conditions operate on CAR.

The microinstruction at address 56 performs the following operations

$$R5 \leftarrow \text{shl R5, if } (C = 0) \text{ then } (CAR \leftarrow 62) \text{ else } (CAR \leftarrow CAR + 1)$$

and the one at address 59

$$R2 \leftarrow R6 + R7,\ CAR \leftarrow \text{External address}$$

The external address is selected by MUX1 through the symbol EXT. The LAD for MUX2 enables the load input of CAR. The address field of the microinstruction is not used.

8-4 MICROPROGRAM EXAMPLES

We now demonstrate by means of two examples how to write microprograms for control memory. We will assume that an external address initiates a sequence of microinstructions in control memory. This sequence constitutes a microprogram *routine*. The external address supplies the first address where the microprogram routine resides in control memory. The routine is terminated with a microinstruction that loads a new external address to start executing a different routine. Thus, the control memory is divided into several routines and an external control determines which routine should be executed. The branch to the required routine is accomplished by the application of the corresponding external address.

Microprogram Example

The first microprogram example has no specific application but it shows how the status bits are used for conditional branch microinstructions. In this example we will write a microprogram starting from address 20 that subtracts the contents of register $R2$ from the contents of register $R1$ and places the difference in register $R3$. If $R1 < R2$ we proceed to add $R1$ to $R4$. If $R1 > R2$ then $R2$ is added to $R4$. If $R1 = R2$ then $R4$ is increment by one. The contents of $R4$ are then applied to the output terminals of the processor and the microprogram routine terminates by branching to an external address. It is assumed that all binary numbers are unsigned. The operations to be performed can be summarized as follows:

1. $R3 \leftarrow R1 - R2$. This updates C and Z
2. If $R1 < R2$ (detected by $C = 0$) then $R4 \leftarrow R4 + R1$
   If $R1 > R2$ (detected by $C = 1$ and $Z = 0$) then $R4 \leftarrow R4 + R2$
   If $R1 = R2$ (detected by $Z = 1$) then $R4 \leftarrow R4 + 1$
3. Output $\leftarrow R4$ and branch to external address.

The condition for C after a subtraction of two unsigned binary numbers is presented in Section 1-4 and Example 1-9. The subtraction $R1 - R2$ is done by adding $R1$

TABLE 8-6
Register Transfer Statements for Example

Address	Microoperations and branch conditions
20	$R3 \leftarrow R1 - R2,\ CAR \leftarrow CAR + 1$
21	If $(C = 1)$ then $(CAR \leftarrow 23)$ else $(CAR \leftarrow CAR + 1)$
22	$R4 \leftarrow R4 + R1,\ CAR \leftarrow 26$
23	If $(Z = 0)$ then $(CAR \leftarrow 25)$ else $(CAR \leftarrow CAR + 1)$
24	$R4 \leftarrow R4 + 1,\ CAR \leftarrow 26$
25	$R4 \leftarrow R4 + R2,\ CAR \leftarrow CAR + 1$
26	Output $\leftarrow R4$, $CAR \leftarrow$ external address

to the 2's complement of $R2$. If $R1 \geq R2$ then $C = 1$, and if $R1 < R2$ then $C = 0$. To detect the condition $R1 > R2$ we must have $C = 1$, provided that $R1$ is not equal to $R2$, which is detected from $Z = 0$. When $R1 = R2$, the subtraction $R1 - R2$ is 0, and the Z bit is set to 1.

Before writing the symbolic microprogram it may be instructive to obtain the sequence of microoperations in terms of register transfer statements. This is shown in Table 8-6. At address 20 we place the microinstruction that subtracts $R2$ from $R1$. This operation updates the C and Z status bits. If $C = 1$ and $Z = 0$, we go to address 25 to add $R2$ to $R4$. If $C = 0$, we go to address 22 to add $R1$ to $R4$. If $Z = 1$ we go to address 24 to increment $R4$. After one of these operations is executed, the microprogram goes to address 26 to output the contents of $R4$ and branch to an external address.

The symbolic microprogram for the example is listed in Table 8-7. The transformation from the register transfer statements to the symbolic microprogram is a straightforward procedure. The microinstruction in address 21 specifies no operation for the processor and a branch condition depending on the value of C. Note that the value of C is determined from the result of the previous operation. Although the carry is available from the ALU during the subtraction operation in the previous microinstruction, it is not transferred into the status bit flip-flop until the next clock pulse. This same clock pulse also transfers the difference into $R3$ and increments CAR. Therefore, the value of C is not available in the status flip-flop until after CAR is incremented. The last microinstruction shows how the content of $R4$ is placed on the output terminals with the TSF and NSH operations. None of the

TABLE 8-7
Symbolic Microprogram for Example

CAR address	Control word					Select		Address field
	A	B	D	F	H	MUX1	MUX2	
20	R1	R2	R3	SUB	NSH	—	NEXT	—
21	—	—	NONE	TSF	NSH	INT	LC	23
22	R4	R1	R4	ADD	NSH	INT	LAD	26
23	—	—	NONE	TSF	NSH	INT	LNZ	25
24	R4	—	R4	INC	NSH	INT	LAD	26
25	R4	R2	R4	ADD	NSH	—	NEXT	—
26	R4	—	NONE	TSF	NSH	EXT	LAD	—

TABLE 8-8
Binary Microprogram for Example

ROM address	Memory content						
	A	B	D	F	H	MUX	ADRS
010100	001	010	011	0101	000	0 000	000000
010101	000	000	000	0000	000	0 010	010111
010110	100	001	100	0010	000	0 001	011010
010111	000	000	000	0000	000	0 101	011001
011000	100	000	100	0001	000	0 001	011010
011001	100	010	100	0010	000	0 000	000000
011010	100	000	000	0000	000	1 001	000000

processor registers receive this information although we could have transferred the data back to $R4$ without loss of information.

The binary microprogram is listed in Table 8-8. The binary values are obtained from the symbolic microinstructions and the corresponding binary codes given in Tables 8-1 through 8-4. When a ROM is used for the control memory, the binary microprogram provides the truth table for the hardware fabrication of the unit.

Counting the Number of 1's

The second microprogram example counts the number of 1's stored in register $R1$ and sets register $R2$ to that number. For example, if $R1 = 00110110$ the microprogram routine counts the four 1's in the register and sets $R2$ to the binary number 100. The numerical example assumes eight bits in $R1$ but the microprogram will count the number of 1's when the register has any other number of bits. The count is done by shifting each bit from $R1$ one at a time into the C status bit and incrementing $R2$ each time that C is equal to 1.

Although the microprogram can be derived directly from the statement of the problem, it may be instructive to construct a flowchart that shows the sequence of microoperations and decision paths. The flowchart for the microprogram routine is shown in Figure 8-4. We assume that the routine starts from address 8. Register $R2$ is initially reset to 0. The content of $R1$ is transferred through the ALU to update the Z bit and reset the carry bit C to 0. If $Z = 1$, it means that the content of $R1$ is zero which signifies that there are no 1's in the number. In that case, the microprogram routine terminates with $R2$ equal to 0. If the value of Z is equal to 0, it indicates that the content of $R1$ is not zero, and therefore, there are some 1's stored in it. $R1$ together with the carry are shifted by rotation as many times as necessary until a 1 is transferred into C. For every 1 detected in C, register $R2$ is incremented by one and the microprogram branches back to check if $R1$ is equal to 0. The program loop is repeated until all the 1's in $R1$ are counted. Note that the value of C is always 0 before the rotation with the content of $R1$.

The microprogram in symbolic form is listed in Table 8-9. The routine starts at address 8 by clearing $R2$ to 0. The microinstruction in address 9 uses the ALU operation TRC (transfer and reset carry) which transfers the contents of register $R1$ through the ALU, resets the carry to 0, and updates the Z and S bits (see Table 8-2). The microinstruction at address 10 checks the value of Z and if equal to 1,

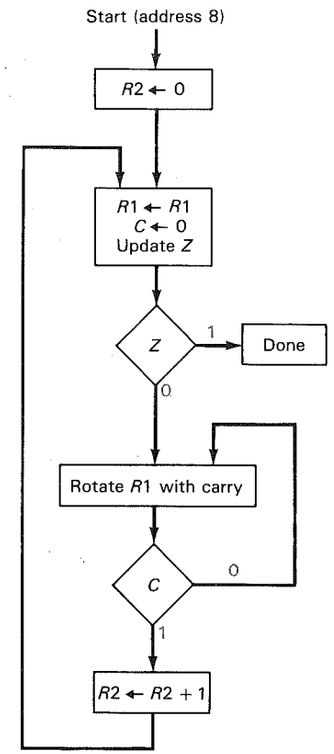


FIGURE 8-4
Flowchart for Counting the Number of 1's

it indicates that $R1$ contains all 0's and the routine is terminated by accepting a new external address. If Z is not equal to 1, the microprogram continues with the microinstruction at address 11. The RRC (rotate right with carry) operation places the least significant bit of $R1$ into C and shifts $R1$ to the right. The value of C is checked in the microinstruction at address 12. If it is 0, control goes back to address 11 to rotate again until C becomes a 1. When $C = 1$, control goes to address 13

TABLE 8-9
Microprogram for Counting the Number of 1's

CAR address	Control word					Select		Address field
	A	B	D	F	H	MUX1	MUX2	
8	—	—	$R2$	TSF	ZERO	—	NEXT	—
9	$R1$	—	$R1$	TRC	NSH	—	NEXT	—
10	—	—	NONE	TSF	NSH	EXT	LZ	—
11	$R1$	—	$R1$	TSF	RRC	—	NEXT	—
12	—	—	NONE	TSF	NSH	INT	LNC	11
13	$R2$	—	$R2$	INC	NSH	INT	LAD	9

to increment $R2$ and then returns to address 9 to check the content of $R1$ for an all 0's quantity.

The reader familiar with assembly language programming will realize that writing microprograms is very similar to writing assembly language programs for a computer. The microprogram concept is a systematic procedure for designing the control unit of a digital system. Once the microinstruction format is established, the design is done by writing a microprogram, which is similar to writing a program for a computer. For this reason, the microprogram method is sometimes referred to as *firmware* to distinguish it from the purely hardware method (which is also called hardwired control) and the software concept which constitutes a purely programming method.

8-5 DESIGN EXAMPLE: BINARY MULTIPLIER

The microprogrammed control organization is extensively used in the design of digital computers. However, the microprogram method is too expensive for the design of small digital systems and is not fast enough for high speed computers. A second alternative to control design is the hardwired control method. The hardwired control is essentially a sequential circuit whose state at any given time determines the microoperations to be executed in the data processor subsystem.

We will demonstrate the procedure for designing hardwired control by means of two examples. The first example is a circuit that multiplies two unsigned binary numbers. The second example is concerned with the design of a simple digital computer. A combinational circuit multiplier that uses many adders and AND gates was developed in Section 3-4. Here we design a sequential multiplier that uses only one adder and a shift register. The data processor part of the multiplier is developed in this section. The design of the control subsystem is presented in the next section. The design of the hardwired control of the simple computer is presented in Sections 8-7 and 8-8.

Binary Multiplier

The multiplication of two binary numbers is done with paper and pencil by successive additions and shifting. This process is best illustrated with a numerical example. Let us multiply the two binary numbers 10111 and 10011.

23	10111	Multiplicand
19	10011	Multiplier
	10111	
	10111	
	00000	
	00000	
	10111	
437	110110101	Product

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, 0's are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. Note that the product obtained from the multiplication of two binary numbers of n bits each can have up to $2n$ bits.

When the above procedure is implemented with digital hardware, it is convenient to change the process slightly. First, instead of providing digital circuits to store and add simultaneously as many binary numbers as there are 1's in the multiplier, it is convenient to provide circuits for the summation of only two binary numbers, and successively accumulate the partial products in a register. Second, instead of shifting the multiplicand to the left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Third, when the corresponding bit in the multiplier is 0, there is no need to add all 0's to the partial product, since this will not alter its value.

Equipment Configuration

The register configuration for the binary multiplier is shown in Figure 8-5. The multiplicand is stored in register B, the multiplier is stored in register Q, and the partial product is formed in register A. A binary adder is employed for adding the content of register B to the content of register A. The C flip-flop stores the carry after the addition. The P counter is initially set to hold a binary number equal to the number of bits in the multiplier. The counter is decremented after the formation of each partial product. When the content of the counter reaches zero, the product is formed in the double register A and Q and the process stops.

The control logic stays in an initial state until the start signal S becomes a 1. The system then performs the multiplication. The sum of A and B forms the partial product which is transferred to A. The output carry from the addition, whether 0 or 1, is transferred to C. Both the partial product in A and the multiplier in Q are

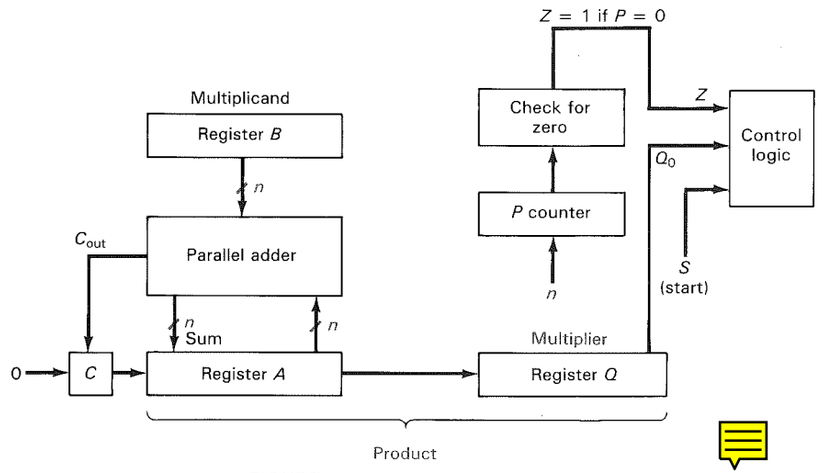


FIGURE 8-5
Block Diagram of a Binary Multiplier

shifted to the right. The least significant bit of A is shifted into the most significant position of Q; the carry from C is shifted into the most significant position of A; and 0 is shifted into C. After the shift right operation, one bit of the partial product is transferred into Q while the multiplier bits in Q are shifted one position to the right. In this manner, the least significant bit of register Q, designated by Q_0, always holds the bit of the multiplier that must be inspected next. The control logic receives this bit and also output Z from a circuit that checks P for zero. These two inputs are the status conditions from the data processor. The start input S is an external input. The outputs of the control logic initiate the required microoperations in the register.

Flowchart

A flowchart showing the sequence of operations in the binary multiplier is shown in Figure 8-6. Initially, the multiplicand is in B and the multiplier in Q. The multiplication process starts when S becomes 1. Register A and flip-flop C are reset to 0 and the sequence counter P is loaded with a binary number n, which is equal to the number of bits in the multiplier.

Next we enter a loop that keeps forming the partial products. The multiplier bit in Q_0 is checked, and if it is equal to 1, the multiplicand in B is added to the partial product in A. The carry from the addition is transferred to C. The partial product in A is left unchanged if $Q_0 = 0$. The P counter is decremented by one regardless of the value in Q_0. Registers A and Q are then shifted once to the right to obtain a new partial product. This shift operation is symbolized in the flowchart in compact form with the statement

$$AQ \leftarrow \text{shr } CAQ,\ C \leftarrow 0$$

CAQ is a composite register made up of flip-flop C and registers A and Q. If we use the individual register symbols, the shift operation can be described by the following microoperations:

$$A \leftarrow \text{shr } A,\ Q \leftarrow \text{shr } Q,\ A_{n-1} \leftarrow C,\ Q_{n-1} \leftarrow A_0,\ C \leftarrow 0$$

Both registers A and Q are shifted right. The leftmost bit of A, designated by A_{n-1}, receives the carry from C. The leftmost bit of Q, or Q_{n-1}, receives the bit from the rightmost position of A in A_0; and C is reset to 0. In essence, this is a long shift of the composite register CAQ with 0 inserted into the serial input which is at C.

The value in the P counter is checked after the formation of each partial product. If the content of P is not zero, status bit Z is equal to 0 and the process repeats to form a new partial product. The process stops when the P counter reaches 0 which causes control input Z to be 1. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in A and Q, with A holding the most significant bits and Q the least significant bits of the product.

The previous numerical example is repeated in Table 8-10 to clarify the multiplication process. The procedure follows the steps outlined in the flowchart.

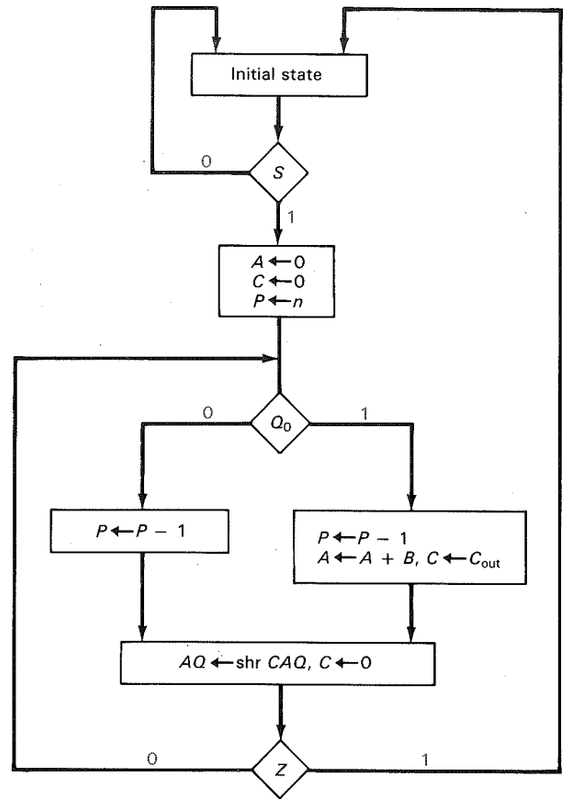


FIGURE 8-6
Flowchart for Binary Multiplier

TABLE 8-10
Numerical Example for Binary Multiplier

Multiplicand B = 10111

	C	A	Q	P
Multiplier in Q	0	00000	10011	101
$Q_0 = 1$; add B		10111		
First partial product	0	10111		100
Shift right CAQ	0	01011	11001	
$Q_0 = 1$; add B		10111		
Second partial product	1	00010		011
Shift right CAQ	0	10001	01100	
$Q_0 = 0$; shift right CAQ	0	01000	10110	010
$Q_0 = 0$; shift right CAQ	0	00100	01011	001
$Q_0 = 1$; add B		10111		
Fifth partial product	0	11011		000
Shift right CAQ	0	01101	10101	
Final product in AQ = 0110110101				

The type of registers needed for the data processor subsystem can be derived from the microoperations listed in the flowchart. Register A must be a shift register with parallel load to accept the sum from the adder and must have a synchronous clear capability to reset the register to 0. Register Q is a shift register. The counter P is a binary down counter with a facility to parallel load a binary constant. The C flip-flop must be designed to accept the input carry and have a synchronous reset capability. Registers B and Q will need a parallel load capability in order to receive the multiplicand and multiplier prior to starting the multiplication process.

8-6 HARDWIRED CONTROL FOR MULTIPLIER

The design of a hardwired control is a sequential circuit design problem. As such, it may be convenient to formulate the state diagram of the sequential control. A flowchart is very similar to a state diagram. The rectangular blocks that designate function boxes in the flowchart may be considered as states in a sequential circuit. The diamond shaped blocks that designate decision boxes in the flowchart determine the conditions for the next state transition in the state diagram. The microoperations that must be executed at a given state are specified within the function boxes. Although it is possible to formulate this relationship between a flowchart and a state diagram, the conversion from one form to the other is not unique. Consequently, different designers may produce different state diagrams for the same flowchart, and each may be a correct representation of the system.

We start the design by assigning an initial state T_0 to the sequential controller. We then determine the required transitions to other states T_1, T_2, T_3, and so on. For each state, we determine the microoperations that must be initiated by the control. This procedure produces the state diagram for the controller, together with a list of register transfer operations which are to be executed while the control circuit is in each state.

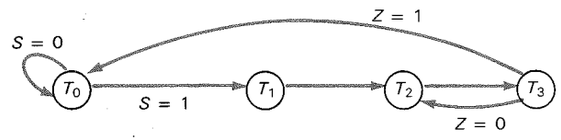
The control state diagram for the binary multiplier is shown in Figure 8-7(a). The information for the diagram is taken directly from the flowchart of Figure 8-6. The four states T_0 through T_3 are related to the rectangular function boxes in the flowchart. S and Z are taken from the diamond shaped decision boxes. The register transfer operations for each state as given in the flowchart are listed below the state diagram in Figure 8-7(b).

From the state diagram we see that the control stays in an initial state T_0 until S becomes 1. It then goes to state T_1 to initialize A, C, and P. Control then goes to state T_2. In this state, register P is decremented and the contents of B are added to A if $Q_0 = 1$; otherwise, A is left unchanged. The two register transfer statements associated with T_2 are

$$T_2\colon P \leftarrow P - 1$$

$$Q_0T_2\colon A \leftarrow A + B,\ C \leftarrow C_{\text{out}}$$

The first statement is always executed when $T_2 = 1$. The second statement is executed during state T_2 provided that Q_0 is also equal to 1. Thus the status bit variable Q_0 is included with timing variable T_2 to form a control (or Boolean)



(a) State diagram

T_0: Initial state

T_1: $A \leftarrow 0,\ C \leftarrow 0,\ P \leftarrow n$

T_2: $P \leftarrow P - 1$

$L = Q_0T_2$: $A \leftarrow A + B,\ C \leftarrow C_{\text{out}}$

T_3: $AQ \leftarrow \text{shr}\ CAQ,\ C \leftarrow 0$

(b) Register transfer statements

FIGURE 8-7
Control State Diagram and List of Microoperations

function Q_0T_2. Note that it is necessary to decrement P at state T_2 so that the decremented value can be checked at state T_3. Remember that all the operations are synchronized with the clock and are actually executed in response to a clock transition.

Control goes to T_3 after T_2. At state T_3, the composite register CAQ is shifted to the right and the contents of P are checked for zero. If $Z = 1$, the multiplication operation is terminated and control goes to the initial state T_0. If $Z = 0$, control goes to state T_2 to form a new partial product.

The block diagram of the control logic is shown in Figure 8-8. The inputs to the control logic are the external signal S and the two status conditions Z and Q_0. The outputs are T_0, T_1, T_2, T_3, and $L = Q_0T_2$. The AND gate that generates variable L is shown separately although it is part of the control logic.

The outputs of the control circuit must be connected to the data processor subsystem to initiate the microoperations in the registers. Output T_1 must be connected to the load input of P and to the reset inputs of A and C. Output T_2 must be connected to the decrement input of P. Output L must be connected to the load input of A and C to receive the sum and output carry from the adder. Output T_3 must be connected to the shift control input of registers A and Q and to the reset input of C. Output T_0 has no effect on the data processor since it only indicates that the system is in an initial state.

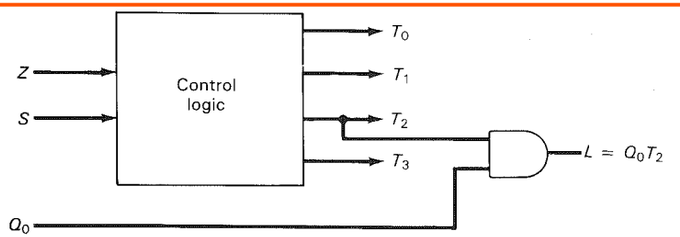


FIGURE 8-8
Control Block Diagram

Once the control sequence has been established, the sequential system that implements the control must be designed. Since the control is a sequential circuit, it can be designed by the sequential logic procedure as outlined in Chapter 4. However, in most cases this method is difficult to carry out because of the large number of states and inputs that a typical control circuit may have. As a consequence, it is necessary to use specialized methods for control logic design which may be considered as variations of the classical sequential logic method. We will now present two such design procedures.

Sequence Register and Decoder

The sequence register and decoder method, as the name implies, uses a register to sequence the control states and a decoder to provide an output for each state. A register with n flip-flops can have up to 2^n states and an n-to-2^n-line decoder will have up to 2^n outputs. An n-bit sequence register is essentially a circuit with n flip-flops together with the associated gates that effect their state transition.

The control state diagram for the binary multiplier has four states and two inputs. To implement it with a sequence register and decoder we need two flip-flops for the register and a 2-to-4-line decoder. Although this is a simple example, the procedure outlined below applies to more complicated situations as well.

The state table for the sequential control is shown in Table 8-11. It is derived directly from the state diagram of Figure 8-7(a). We designate the two flip-flops by G_1 and G_0 and assign the binary state 00, 01, 10, and 11, to T_0, T_1, T_2, and T_3, respectively. Note that the input columns have don't care entries whenever the input variable is not used. The outputs of the sequential circuit are designated by variables T_0 through T_3. The particular output variable that is equal to 1 at any given time is determined from the equivalent binary value of the present state. Thus, when the present state of the register is $G_1G_0 = 00$, output T_0 must be equal to 1 while the other three outputs remain at 0. Since the outputs are a function of only the present state, they can be generated with a decoder circuit having the two inputs G_1 and G_0 and the four outputs T_0 through T_3.

The sequential circuit can be designed from the state table by means of the classical procedure presented in Chapter 4. This example has a small number of states and inputs so we could use maps to simplify the Boolean functions. In most other control logic applications, the number of states and inputs is much larger. The application of the classical method requires an excessive amount of work to

TABLE 8-11
State Table for Control Circuit

Present state		Inputs		Next state		Outputs			
G_1	G_0	S	Z	G_1	G_0	T_0	T_1	T_2	T_3
0	0	0	X	0	0	1	0	0	0
0	0	1	X	0	1	1	0	0	0
0	1	X	X	1	0	0	1	0	0
1	0	X	X	1	1	0	0	1	0
1	1	X	0	1	0	0	0	0	1
1	1	X	1	0	0	0	0	0	1

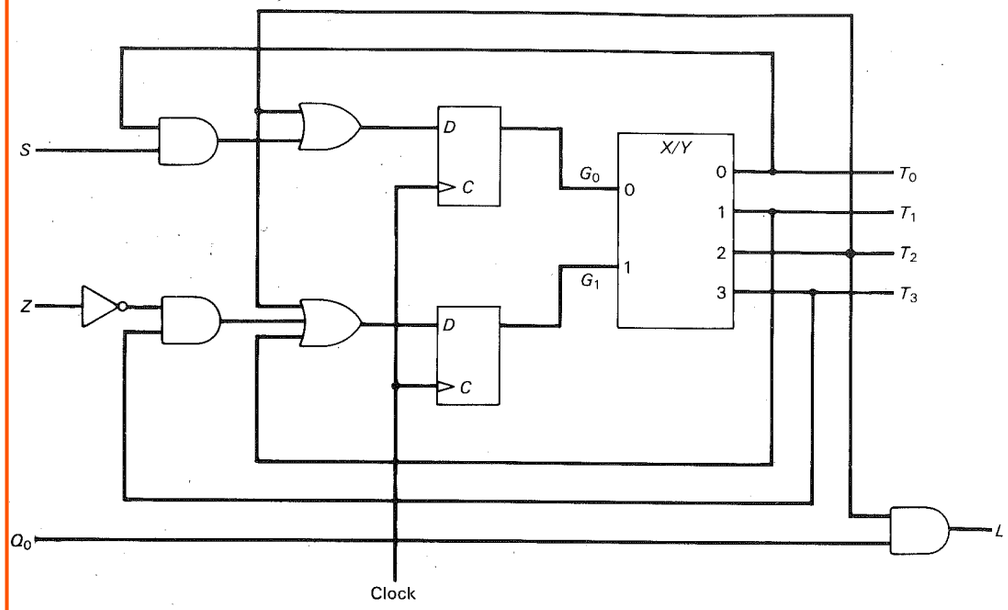


FIGURE 8-9
Logic Diagram of Control for Binary Multiplier Using a Sequence Register and Decoder

obtain the simplified input equations for the flip-flops. The design can be simplified if we take into consideration the fact that the decoder outputs are available for use in the design. Instead of using the flip-flop outputs as the present state conditions, we might as well use the outputs of the decoder to obtain this information. The outputs of the decoder are T_0 through T_3. These variables can be used to supply the present state conditions for the sequential circuit. Moreover, instead of using maps to simplify the flip-flop input equations, we can obtain them directly from the state table by inspection. Although this may not result in a minimal circuit, the possible waste of a few gates may be worth the time saved from the simplicity of the procedure. For example, from the next state conditions in the state table we find that the next state of G_1 is equal to 1 when the present state is T_1, or when the present state is T_2, or when the present state is T_3 provided that $Z = 0$. These conditions give the input equation for G_1.

$$D_{G1} = T_1 + T_2 + T_3\overline{Z}$$

Where D_{G1} is the D input of flip-flop G_1. Similarly, the input equation for flip-flop G_0 is

$$D_{G0} = T_0S + T_2$$

When deriving input equations by inspection from the state table, we cannot be sure that the Boolean functions have been simplified in the best possible way. For this reason, it is advisable to analyze the circuit to ensure that the derived equations do indeed produce the required state transitions as specified by the original state diagram.

The logic diagram of the control circuit is drawn in Figure 8-9. It consists of a register with two flip-flops G_1 and G_0 and a 2 × 4 decoder. The outputs of the

decoder are used to obtain the next state of the circuit according to the Boolean input equations listed above. The outputs of the controller should be connected to the data processor part of the system to initiate the required microoperations.

One Flip-Flop per State

Another possible method of control logic design is to use one flip-flop per state in the sequential circuit. Only one flip-flop is set to 1 at any particular time; all others are reset to 0. The single bit is made to propagate from one flip-flop to the other under the control of decision logic. In such a configuration, each flip-flop represents a state that is activated only when the control bit is transferred to it.

It is obvious that this method uses a maximum number of flip-flops for the sequential circuit. For example, a sequential circuit with 12 states requires a minimum of four flip-flops in a conventional sequential circuit. With the one flip-flop per state method, the control circuit uses 12 flip-flops, one for each state. At first glance, it may seem that this method would increase system cost since more flip-flops are used. But the method offers some advantages which may not be apparent at first. One advantage of this method is the simplicity with which it can be designed. This type of controller can be designed by inspection from the state diagram that describes the control sequence. No state or excitation tables are needed if D-type flip-flops are employed. This offers a savings in design effort, an increase in operational simplicity, and a possible decrease in the total number of gates since a decoder is not needed.

We will demonstrate the design procedure by obtaining the control circuit specified by the state diagram of Figure 8-7(a). Since there are four states in the state diagram, we choose four D flip-flops and label their outputs T_0, T_1, T_2, and T_3. The input equations for setting each flip-flop to 1 is determined from the present state and the input conditions along the corresponding directed lines going into the state. For example, flip-flop T_0 is set to 1 with the next clock pulse transition if the circuit is in present state T_3 and input Z is equal to 1 or if the circuit is in present state T_0 and S is equal to 0. These conditions are specified by the flip-flop input equation

$$D_{T0} = T_0\overline{S} + T_3Z$$

where D_{T0} denotes the D input of flip-flop T_0. In fact, the condition for setting a flip-flop to 1 is obtained directly from the state diagram from the condition specified in the directed lines going into the corresponding flip-flop state ANDed with the previous flip-flop state. If there is more than one directed line going into a state, all conditions must be ORed. Using this procedure for the other three flip-flops we obtain the input equations

$$D_{T1} = T_0S$$

$$D_{T2} = T_1 + T_3\overline{Z}$$

$$D_{T3} = T_2$$

The logic diagram of the controller is shown in Figure 8-10. It consists of four D flip-flops T_0 through T_3 and the associated gates specified by the input equations

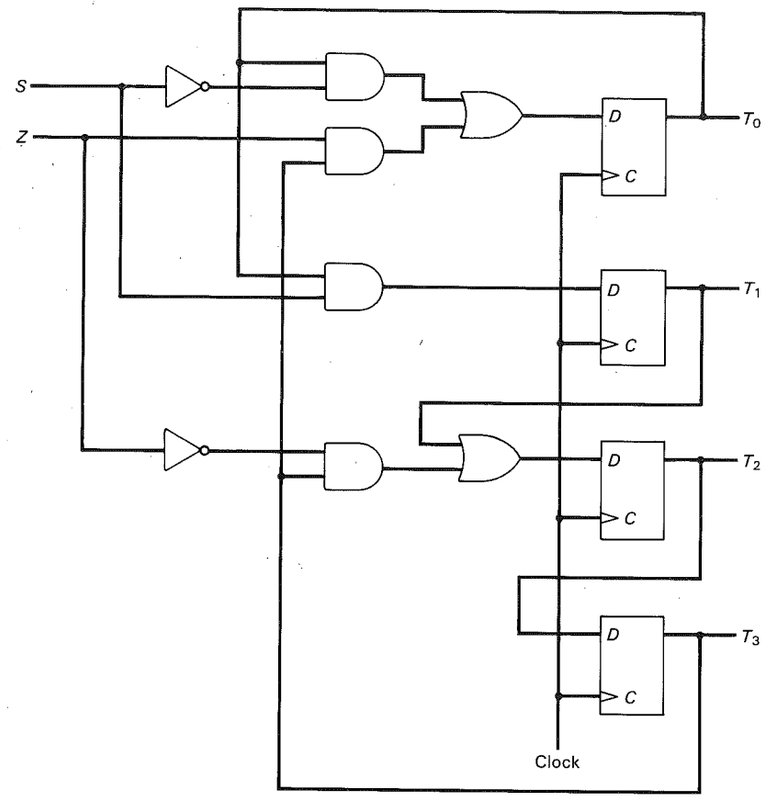


FIGURE 8-10
One Flip-Flop Per State Controller

listed above. The outputs of the flip-flops must be connected to the corresponding inputs in the registers of the data processor subsystem.

Initially flip-flop T_0 must be set to 1 and all other flip-flops must be reset to 0 so that the flip-flop representing the initial state is enabled. Once started, the one flip-flop per state controller will propagate from one state to the other in the proper manner. Only one flip-flop will be set to 1 with each clock pulse transition; all others are reset to 0 because their D inputs are maintained at 0.

Sometimes, the one flip-flop per state controller is implemented with a register that has a common asynchronous input for resetting all flip-flops initially to 0. In that case, it is possible to transfer the circuit into its initial state by modifying the input equation for T_0 as follows:

$$D_{T0} = T_0\overline{S} + T_3Z + \overline{T}_0\overline{T}_1\overline{T}_2\overline{T}_3$$

The third term in the equation sets flip-flop T_0 to 1 with the first clock pulse, just after all the flip-flops are reset to 0 asynchronously.

8-7 EXAMPLE OF A SIMPLE COMPUTER

In this and the next section we will introduce a simple digital computer and show how its operation can be defined by means of register transfer statements. We will then implement the computer by specifying the registers that are needed in the data processor subsystem and design the control subsystem by the hardwired method. The purpose here is to introduce the basic components of the digital computer and show another example of hardwired control. A more extensive study of the various concepts associated with digital computers will be covered in detail in the next chapter. The design of a more extensive digital computer is undertaken in Chapter 10.

Instruction Codes

A digital computer is a general purpose device capable of executing various operations and, in addition, can be instructed as to what specific operations it must perform. The user of a computer can control the process by means of a *program*. A program is a set of instructions that specifies the operations, operands, and the sequence in which processing is to occur. The data processing task may be altered by specifying a new program with different instructions or by specifying the same instructions with different data. Instruction codes, together with data, are stored in memory. The control reads each instruction from memory and places it in a control register. The control then interprets the instruction and proceeds to execute it by issuing a sequence of microoperations. Every computer has its own unique instruction set. The ability to store and execute instructions is the most important property of the general purpose computer.

An *instruction code* is a group of bits that instructs the computer to perform a specific operation. It is usually divided into parts, each having its own particular interpretation. The most basic part of an instruction code is its operation part. The *operation code* of an instruction is a group of bits that defines an operation, such as add, subtract, shift, or complement. The set of machine operations formulated for a computer depends on the processing it is intended to perform. The total number of operations obtained determines the set of machine operations.

The number of bits required for the operation code of an instruction is a function of the total number of operations used. It must consist of at least n bits for up to 2^n distinct operations. The designer assigns a bit combination (a code) to each operation. The control unit of the computer is designed to accept this bit configuration at the proper time in a sequence and to supply the proper commands to the data processor in order to execute the specified operation. As a specific example, consider a computer using 64 distinct operations, one of them an ADD operation. The operation code may consist of six bits, with a bit configuration 100110 assigned to the ADD operation. When the operation code 100110 is detected by the control unit, a command signal is applied to an adder circuit to add two binary numbers.

The operation part of an instruction code specifies the operation to be performed. The operation must be performed using data stored in computer registers or in memory. An instruction code, therefore, must specify not only the operation but

also the registers or memory words where the operands are to be found. The operands may be specified in an instruction code in two ways. An operand is said to be specified explicitly if the instruction code contains special bits for its identification. For example, an instruction code may contain an operation part and a memory address specifying the location of the operand in memory. An operand is said to be specified implicitly if it is included as part of the definition of the operation. For example, an operation that complements a register implies the register in the operation part of the code.

Instruction Code Formats

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction code are divided into groups that subdivide the instruction into parts called fields. Each field is assigned a task such as an operation code or a memory address. The various fields specify different functions for the instruction, and when shown together they constitute the instruction code format.

Consider, for example, the three instruction code formats depicted in Figure 8-11. The instruction format in (a) consists of an operation code which implies a register in the processor unit. It can be used to specify instructions such as "increment a processor register" or "complement a processor register." The register in this case is *implied* by the operation code. The instruction format in (b) has an operation code followed by an operand. This is called an *immediate operand* instruction because the operand follows immediately after the operation code. It can be used to specify instructions such as "add the operand to the present contents of a register" or "transfer the operand into a register" or it can specify any operation to be done between the given operand and a processor register. The instruction format specified in (c) gives the address of the operand in memory. In other words, the operation specified by the operation code is performed between a processor register and an operand which is stored in memory. The address of this operand is included as part of the instruction. This type of instruction is referred to as a *direct address* instruction.

The instruction formats shown in Figure 8-11 are three of many possible formats that can be formulated for a digital computer. They are presented here just as an example and should not be considered the only possibilities. In Chapter 9 we present and discuss other instructions and instruction code formats.

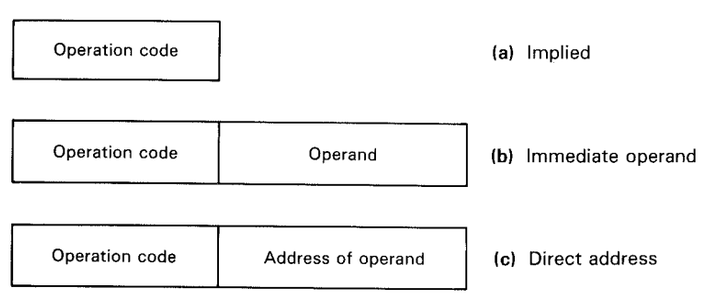


FIGURE 8-11
Three Possible Instruction Formats

Numerical Example

Assume that we have a memory unit with 8 bits per word and that an operation code contains 8 bits. The placement of the three instruction codes in memory in binary is shown in Figure 8-12. At address 25 we have an implied instruction that specifies an operation to increment register R. This operation can be symbolized by the statement

$$R \leftarrow R + 1$$

In memory address 35 and 36 we have an immediate operand instruction that occupies two 8-bit words. The first word at address 35 is the operation code for the instruction to transfer the operand to register R, symbolized by the statement

$$R \leftarrow \text{Operand}$$

The operand itself is stored immediately after the operation code at address 36.

In memory address 45 there is a direct address instruction that specifies the operation

$$R \leftarrow M[\text{address}]$$

The second word of the instruction at address 46 contains the address. Therefore, the operand to be transferred to register R is the one stored at this address. For the numerical example shown in the figure, we have the instruction symbolized by $R \leftarrow M[70]$. The symbol $M[70]$ signifies the contents of the memory word at address 70. Since this word contains the binary equivalent of 28, the result of the operation is the transfer of the 8-bit word 00011100 into register R.

Decimal address	Memory content		Operation
25	00000001	Operation code = 1	$R \leftarrow R + 1$
35	00000011	Operation code = 3	$R \leftarrow$ Operand
36	00101100	Operand = 44	
45	00000101	Operation code = 5	$R \leftarrow M$[Address]
46	01000110	Address = 70	
70	00011100	Operand = 28	

FIGURE 8-12
Memory Representation of Instructions

It must be realized that the placing of instructions in memory as shown in Figure 8-12 is only one of many possible alternatives. Only very small computers have 8-bit words. Large computers may have from 16 to 64 bits per word. In many computers, the entire instruction can be placed in one word, and in some, even two or more instructions can be placed in a single memory word.

At this point we must recognize the relationship between a computer *operation* and a hardware *microoperation*. An operation is specified by an instruction stored in computer memory. It is a binary code that tells the computer to perform a specific operation. The control unit in the computer retrieves the instruction from memory and interprets the operation code bits. It then issues a sequence of control functions to perform the required microoperations for the execution of the instruction.

Computer Block Diagram

The block diagram of a simple computer is shown in Figure 8-13. The computer consists of a memory unit, six registers, two decoders, and control logic gates. The memory has 256 words of 8 bits each, which is very small for a real computer, but sufficient for demonstrating the basic operations found in most computers. Instructions and data are stored in the memory, but all information processing is done in the registers. The six registers are listed in Table 8-12 together with a brief description of their function and the number of bits they contain.

The data register DR holds the contents of the memory word read from or written into memory. The address register AR holds the address of an operand

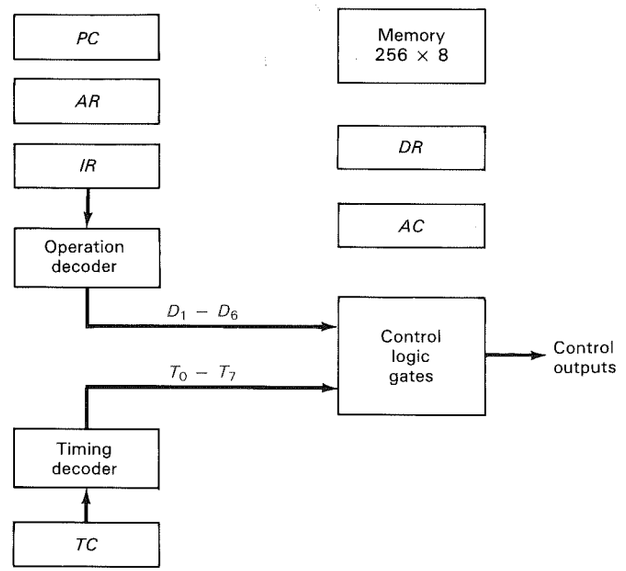


FIGURE 8-13
Block Diagram of a Simple Computer

TABLE 8-12
List of Registers for the Simple Computer

Register symbol	Number of bits	Register name	Function
DR	8	Data register	Holds contents of memory word
AR	8	Address register	Holds address of operand
AC	8	Accumulator	Processor register
IR	8	Instruction register	Holds operation code
PC	8	Program counter	Holds address of instruction
TC	3	Timing counter	Sequence register

read

when it has to be fetched from memory. The accumulator register is a general purpose register where the data processing is done. The instruction register *IR* receives the operation code of instructions. The decoder associated with this register supplies one output for each operation code encountered. Thus, decoder output $D_1 = 1$ if the operation code of the instruction is binary 1, output $D_2 = 1$ if the operation code is binary 2, and so on. The timing counter *TC* is also decoded to supply eight timing variables T_0 through T_7 (see Figure 5-22). The counter is incremented with every clock pulse, but it can be reset to 0 at any time to start a new sequence from T_0. The control logic gates receive the inputs from the decoders and generate the required control functions for the registers in the computer.

The program counter *PC* holds the current address of the instruction. The *PC* goes through a step-by-step counting sequence and causes the computer to read successive instructions previously stored in memory. To read an instruction, the content of *PC* is taken as the address for memory and a memory read cycle is initiated. The *PC* is then incremented by one; so it holds the next address in the sequence of instructions. An operation code read from memory into *DR* is transferred into *IR*. If the memory address part of the instruction is read into *DR*, this address is transferred into *AR* to access the operand.

Computer Instructions

Six instructions for the simple computer are listed in Table 8-13. We assume that there are eight bits in the operation code so it can be stored in one memory word. With eight bits it is possible to specify up to 256 operations. To simplify the presentation, we consider here only six operations. The symbolic names associated with the instructions are useful for listing programs in symbolic form instead of

TABLE 8-13
Six Instructions for the Simple Computer

Operation code	Symbolic name	Description	Function
00000001	INA	Increment *AC*	$AC \leftarrow AC + 1$
00000010	CMA	Complement *AC*	$AC \leftarrow \overline{AC}$
00000011	LDI OPRD	Load immediate operand	$AC \leftarrow$ OPRD
00000100	ADI OPRD	Add immediate operand	$AC \leftarrow AC +$ OPRD
00000101	LDA ADRS	Load to *AC*	$AC \leftarrow M$[ADRS]
00000110	STA ADRS	Store from *AC*	M[ADRS] $\leftarrow AC$

using binary codes. The first two instructions are implied instructions that increment and complement the accumulator register. The next two instructions are immediate type instructions. These instructions have two words, one for the operation code and the other for the immediate operand symbolized by OPRD. The OPRD following the symbol LDI stands for an actual operand that the programmer must specify. The instruction ADI OPRD adds the specified operand to the content of the accumulator.

The last two instructions in the table are direct address instructions. LDA is a symbol for the operation code and the ADRS following it stands for an address that the programmer must specify with this instruction. The load instruction transfers a memory operand into the accumulator register. The store instruction transfers the content of the accumulator into a memory word whose address is given by the instruction. The actual OPRD and ADRS values, together with their corresponding operation codes are stored in memory as shown in Figure 8-12.

To demonstrate how the instructions can be used to write a simple program, consider the arithmetic expression 83 − (52 + 25). The following program performs the computation and stores the result in memory at address 250. The subtraction is done by taking the 2's complement of (52 + 25) and adding it to 83.

```
LDI 52      Load 52 into the AC
ADI 25      Add 25 to the AC
CMA         Complement AC
INA         Increment AC
ADI 83      Add 83 to the AC
STA 250     Store the contents of AC in M[250]
```

To store this program in memory it is necessary to convert all symbolic names to their corresponding 8-bit operation codes and convert the decimal numbers to binary.

8-8 DESIGN OF SIMPLE COMPUTER

We will now show the procedure for designing the simple computer defined in the previous section. The registers of the computer are defined in the block diagram of Figure 8-13. The instructions of the computer are listed in Table 8-13. A computer with only six instructions is not very useful. We must assume that it has many more instructions even though only six of them are considered here. A program written for the computer is stored in memory. This program consists of many instructions, but once in a while the instructions used will be one of the six listed. We first consider the internal operations needed to execute the instructions that are stored in memory.

Instruction Fetch Phase

The program counter *PC* is initialized so it holds the address of the first instruction of the program stored in memory. When a start switch is activated, the computer sequence follows a basic pattern. An operation code whose address is in *PC* is read

from memory into DR. The operation code is then transferred from DR to IR and PC is incremented by one to prepare it for the address of the next word in memory. This sequence is called the *instruction fetch* phase, since it fetches the operation code from memory and places it in a control register. The timing variables T_0 and T_1 of the timing decoder are used as control functions for the fetch phase. This can be expressed with two register transfer statements as follows

$$T_0: DR \leftarrow M[PC]$$

$$T_1: IR \leftarrow DR,\ PC \leftarrow PC + 1$$

With timing variable T_0, the memory word specified by the address in PC is transferred into DR. This word is the operation code of an instruction. Timing variable T_1 transfers the operation code from DR into the instruction register IR and also increments PC by one. This prepares PC for the next read operation since it now contains the address of the next instruction word in memory. It is very important to realize that all the operations are synchronized with a common clock source. The same clock pulse transition that transfers the content of memory into DR when $T_0 = 1$, also triggers the timing counter TC and changes its contents from 000 to 001, which causes the timing decoder to go from T_0 to T_1.

The instruction fetch phase is common to all instructions. The microoperations and control functions that follow the instruction fetch are determined in the control section from the present operation code in IR. The output of IR is decoded by the operation decoder. This decoder has outputs D_1 through D_6, corresponding to the binary operation codes 00000001 through 00000110, assigned to the six instructions.

Execution of Instructions

During timing variable T_2, the operation code is in IR and only one output of the operation decoder is equal to 1. The control uses this output to determine the next microoperation in sequence. The INA (increment AC) instruction has an operation code 00000001 which makes decoder output $D_1 = 1$. The execution of this instruction can be specified by the following register transfer statement:

$$D_1T_2: AC \leftarrow AC + 1,\ TC \leftarrow 0$$

Thus, when $D_1 = 1$ at time T_2, the content of the AC is incremented by one and the timing counter TC is reset to 0. By resetting TC to 0, control goes back to timing variable T_0 to start the fetch phase and read the operation code of the next instruction in the program. Remember that PC is incremented during time T_1; so now it holds the address of the next instruction in sequence.

The LDI OPRD instruction has an operation code 00000011 which makes the output of the decoder $D_3 = 1$. The microoperations that execute this instruction are

$$D_3T_2: DR \leftarrow M[PC]$$

$$D_3T_3: AC \leftarrow DR,\ PC \leftarrow PC + 1,\ TC \leftarrow 0$$

The timing variable that follows the fetch phase when $D_3 = 1$ is T_2. At this time,

the operand symbolized by OPRD is read from memory and placed in DR. Since the operand is in memory following the operation code, it is read from memory using the address in PC. The operand read into DR is then transferred to the AC. PC is incremented once again to prepare it for the address of the next instruction. In fact, PC is incremented every time a word that belongs to the program is read from memory. This prepares PC for reading the next word in sequence. Moreover, at the completion of an instruction execution, the control always resets TC to 0 in order to return to the fetch phase.

The LDA ADRS instruction has an operation code that makes $D_5 = 1$. The microoperations needed to execute this instruction are

$$D_5T_2: DR \leftarrow M[PC]$$

$$D_5T_3: AR \leftarrow DR,\ PC \leftarrow PC + 1$$

$$D_5T_4: DR \leftarrow M[AR]$$

$$D_5T_5: AC \leftarrow DR,\ TC \leftarrow 0$$

The address of the operand, symbolized by ADRS, resides in memory right after the operation code of the instruction. Since PC was incremented during the fetch phase, it now holds the address where ADRS is stored in memory. The value of ADRS is read from memory at time T_2. At time T_3, the address is transferred from DR to AR, and PC is incremented by one. Since ADRS specifies the address of the operand, a memory read during time T_4 causes the operand to be placed in DR. The operand from DR is transferred to AC and control goes back to the fetch phase.

The sequence of microoperations for the simple computer are listed in the flowchart of Figure 8-14. The timing variables associated with the microoperations are indicated along the function boxes. The first two timing variables perform the fetch phase which reads the operation code into IR and decodes it. The microoperations that are executed during time T_2 depend on the operation code value in IR. This is indicated in the flowchart by six different paths that the control may take. Thus at time T_2 the AC is incremented if $D_1 = 1$, or the AC is complemented if $D_2 = 1$, or the operand is read from memory if D_3 or $D_4 = 1$, or the address part of the instruction is read from memory if D_5 or $D_6 = 1$. The particular microoperation executed at time T_2 is the one with the corresponding control function having a D variable equal to 1. The same can be said for the other timing variables.

The instruction ADI OPRD is executed similar to the way the LDI OPRD is executed except that the contents of DR is added to those of AC. The instruction STA ADRS is similar to the LDA ADRS except that the contents of AC are stored in memory.

A practical computer has many more instructions, and each instruction requires a fetch phase for reading its operation code from memory. The microoperations that execute the particular instruction are determined by the operation decoder output and sequenced by the timing variables. The list of control functions and microoperations for a practical computer would be much longer than the one shown in Figure 8-14. Obviously, the simple computer is not a practical device, but by using only six instructions, the basic function of a digital computer can be dem-

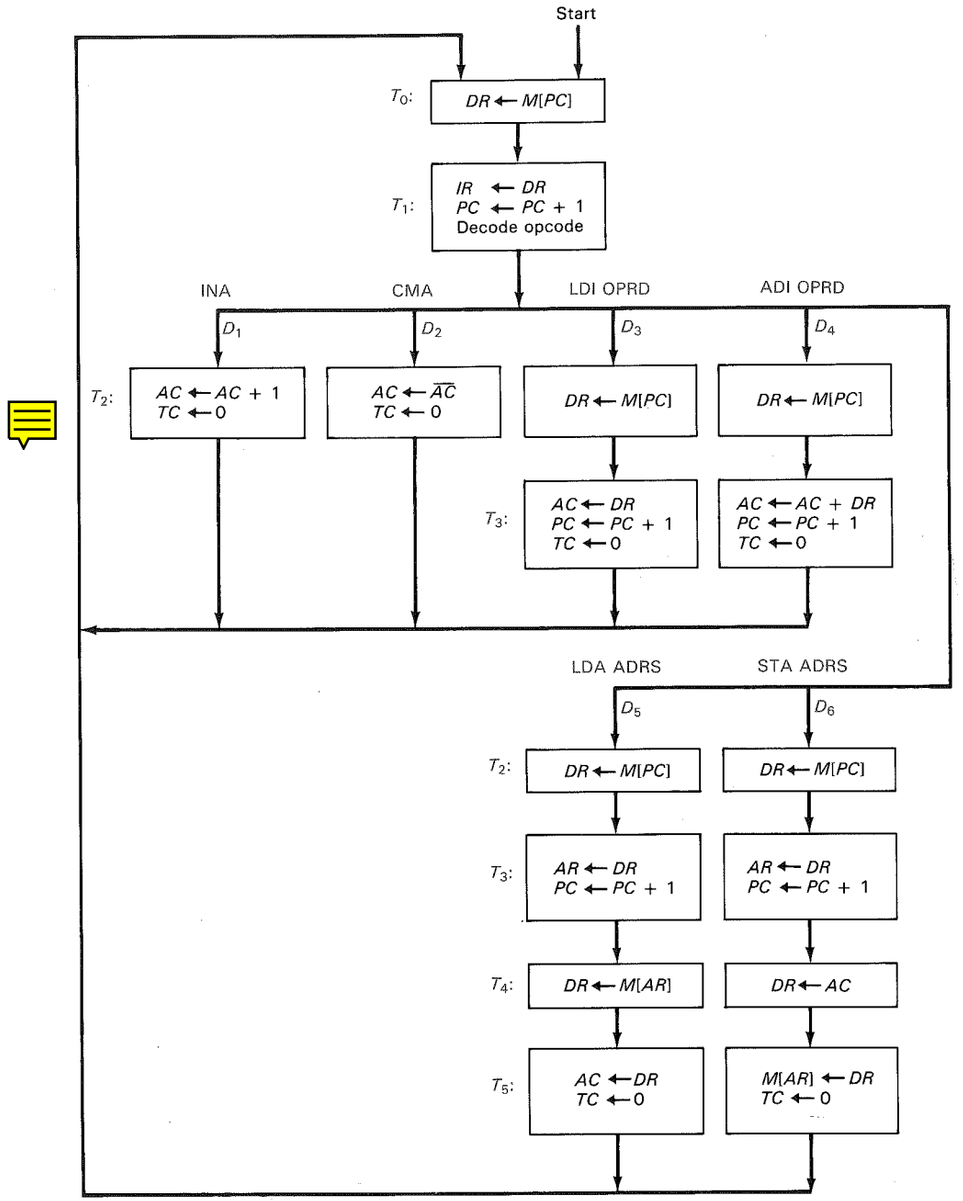


FIGURE 8-14
Microoperation Sequence for the Simple Computer

onstrated. The extension of this principle to a computer with more instructions and more processor registers should be apparent from this example.

Control Logic

It was shown that the register transfer notation is suitable for defining the operations specified by computer instructions and is a convenient method for specifying the sequence of internal operations in a digital computer. We will now show that the list of control functions and microoperations for a digital system is a convenient starting point for the design of the system. The list of microoperations specifies the type of registers and associated digital functions that must be incorporated into the data processing part of the system. The list of control functions specifies the logic gates required for the control unit. To demonstrate this procedure, we will go through the design of the simple computer based on the list of register transfer statements given in Figure 8-14.

The first step in the design is to scan the register transfer statements and retrieve all those statements that perform the same microoperation. For example, the microoperation $PC \leftarrow PC + 1$ is listed with timing variable T_1 and again with timing variable T_3 for the decoded instructions D_3, D_4, D_5, and D_6. The five conditions can be combined into one statement:

$$T_1 + (D_3 + D_4 + D_5 + D_6)T_3: PC \leftarrow PC + 1$$

Remember that a control function is a Boolean function. The symbol + between the control variables denotes a Boolean OR operation and the absence of an operator denotes a Boolean AND operation. The above statement combines all the control conditions for incrementing PC. The hardware implementation of this statement is depicted in Figure 8-15. When the output of the combination circuit is equal to 1, the next clock pulse transition increments PC by one. Thus PC must be a binary counter and the increment input is used for enabling the count.

There are 12 different microoperations listed in Figure 8-14. For each distinct microoperation, we accumulate the associated control functions and OR them together. The result is shown in Table 8-14. The combined control functions obtained for each microoperation are equated to a binary variable C_i for $i = 1, 2, 3, \ldots, 12$. The 12 variables can be easily generated with OR and AND (or NAND) gates to constitute the control logic gates in the control unit of the computer.

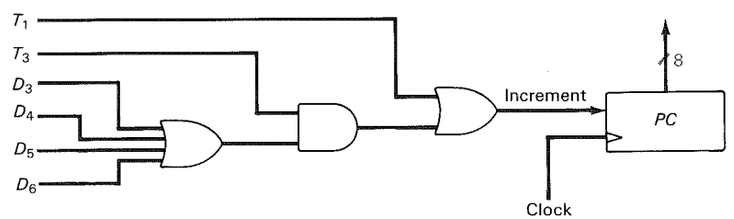


FIGURE 8-15
Implementation of $T_1 + (D_3 + D_4 + D_5 + D_6)T_3: PC \leftarrow PC + 1$

TABLE 8-14
Hardware Specification of Simple Computer

Control function	Microoperation
$C_1 = T_0 + (D_3 + D_4 + D_5 + D_6)T_2$:	$DR \leftarrow M[PC]$
$C_2 = T_1 + (D_3 + D_4 + D_5 + D_6)T_3$:	$PC \leftarrow PC + 1$
$C_3 = T_1$:	$IR \leftarrow DR$
$C_4 = D_1T_2$:	$AC \leftarrow AC + 1$
$C_5 = D_2T_2$:	$AC \leftarrow \overline{AC}$
$C_6 = D_3T_3 + D_5T_5$:	$AC \leftarrow DR$
$C_7 = D_4T_3$:	$AC \leftarrow AC + DR$
$C_8 = (D_5 + D_6)T_3$:	$AR \leftarrow DR$
$C_9 = D_6T_4$:	$DR \leftarrow AC$
$C_{10} = D_5T_4$:	$DR \leftarrow M[AR]$
$C_{11} = D_6T_5$:	$M[AR] \leftarrow DR$
$C_{12} = (D_1 + D_2)T_2 + (D_3 + D_4)T_3 + (D_5 + D_6)T_5$:	$TC \leftarrow 0$

Design of Computer

The design of the simple computer can be obtained from the information available in Table 8-14. The block diagram design is shown in Figure 8-16. It consists of the memory unit, six registers, two decoders, three multiplexers, an 8-bit adder, and control logic gates. The six registers are the ones listed in Table 8-12. The control logic gates receive their inputs from the two decoders and generate the 12 control functions C_1 through C_{12} according to the Boolean functions listed in Table 8-14. Any register that receives data from two sources needs a multiplexer to select between the two.

The control functions operate on the registers and memory to produce the required microoperations. For example, Table 8-14 shows that the control function C_1 must perform the microoperation $DR \leftarrow M[PC]$. This is a memory read operation with PC supplying the address and DR receiving the memory word. Therefore, C_1 selects the 8 bits from PC through MUX1, enables the read input of the memory, enables the load input of DR while MUX2 selects its inputs from the memory output data (because $C_9 = 0$).

The microoperations for the AC are determined from the following statements in Table 8-14.

C_4: $AC \leftarrow AC + 1$	Increment AC
C_5: $AC \leftarrow \overline{AC}$	Complement AC
C_6: $AC \leftarrow DR$	Transfer DR to AC
C_7: $AC \leftarrow AC + DR$	Add DR to AC

The AC can be implemented with a counter that has the capability of complementing its contents and loading external data. The counter produces the increment microoperation and the load control enables the input data. When $C_6 = 1$, C_7 must be equal to 0, and MUX3 selects the outputs of DR to be loaded into the AC. When $C_7 = 1$, the outputs of the 8-bit adder are loaded into the AC. The timing counter TC is incremented with every clock pulse except when $C_{12} = 1$, which resets the counter to 0.

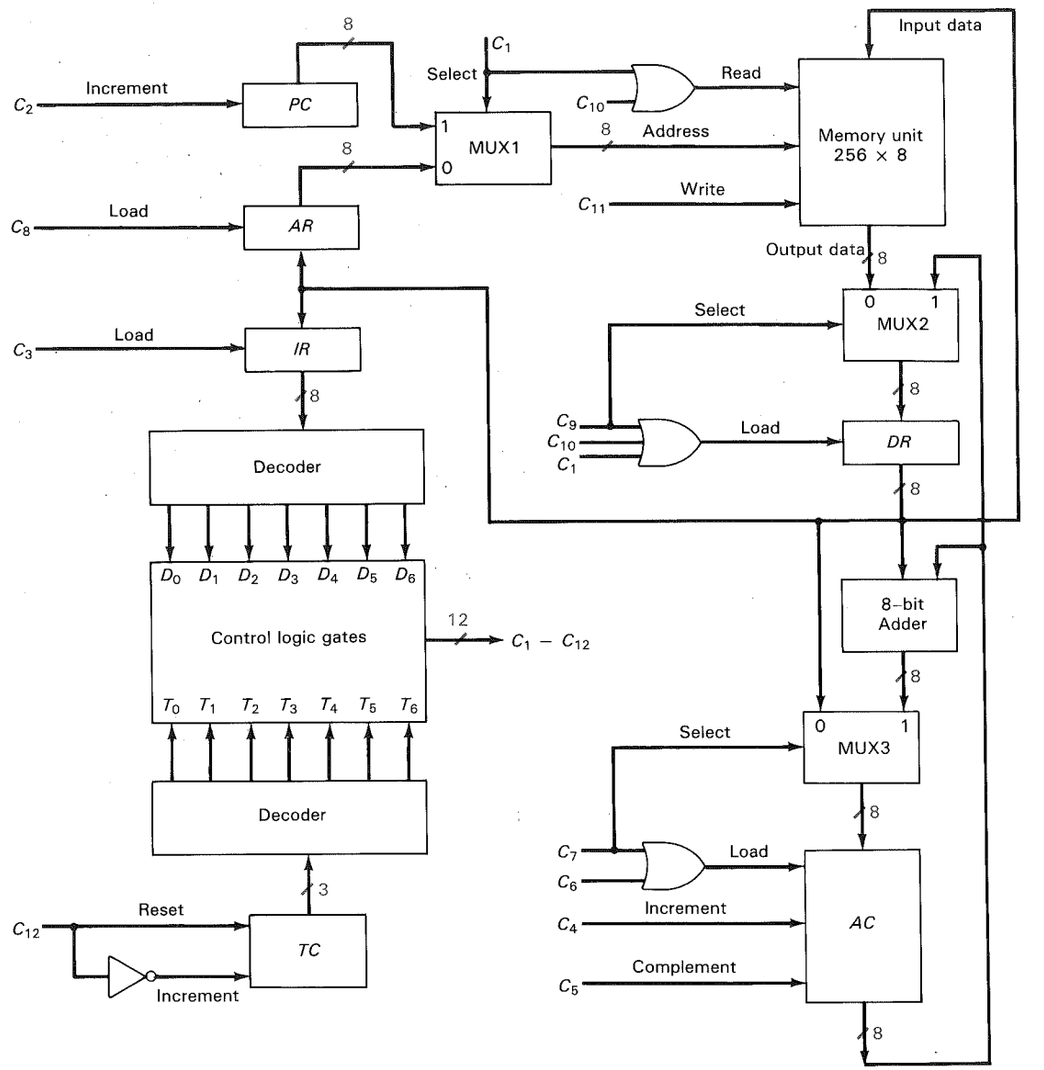


FIGURE 8-16
Hardware Design of the Simple Computer

The other control functions enable the appropriate registers and select the proper paths through the multiplexers. Note that all six registers must be connected to a common clock pulse generator for a synchronous operation. The clock inputs have been omitted from the block diagram in order to simplify the drawing.

The registers and other digital functions specified in Figure 8-16 can be designed

with integrated circuits. One can readily find commercial MSI circuits for all the registers, multiplexers, and decoders. The combinational circuit for the control logic gates can be constructed with individual gates. This part of the computer would be more efficiently implemented with a programmable logic device as explained in Chapter 6.

REFERENCES

1. MANO, M. M. *Computer System Architecture.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1982.
2. ERCEGOVAC, M. D., AND LANG, T. *Digital Systems and Hardware/Firmware Algorithms.* New York: Wiley, 1985.
3. AGRAWALA, A. K., AND RAUSCHER, T. G. *Foundations of Microprogramming.* New York: Academic, 1976.
4. MYERS, G. J. *Digital System Design with LSI Bit-Slice Logic.* New York: Wiley, 1980.
5. ANDREWS, M. *Principles of Firmware Engineering in Microprogram Control.* Potomac, MD: Computer Science Press, 1980.
6. FLETCHER, W. I. *An Engineering Approach to Digital Design.* Englewood Cliffs: Prentice-Hall, 1980.
7. RHYNE, V. T. *Fundementals of Digital Systems Design.* Englewood Cliffs: Prentice-Hall, 1973.
8. CLARE, C. R. *Designing Logic Systems Using State Machines.* New York: McGraw-Hill, 1973.
9. HILL, F. J., AND PETERSON, G. R. *Digital Logic and Microprocessors.* New York: Wiley, 1984.

PROBLEMS

8-1 Define the following terms in your own words: (a) data processor subsystem; (b) control subsystem; (c) hardwired control; (d) microprogrammed control; (e) control memory; (f) control word; (g) microinstruction; (h) microprogram; (i) microoperation.

8-2 A microprogram control unit is similar to the one shown in Figure 8-3 with the following differences: (a) MUX1 has four inputs for CAR; (b) MUX2 has 15 input status bits; (c) the control memory has 1024 words. Formulate the microinstruction format and specify the number of bits in each field. What should be the size of the control memory?

8-3 A microinstruction stored in address 35 in the control memory of Figure 8-3 performs the operation

$$R1 \leftarrow R1 + R2,\ CAR \leftarrow CAR + 1$$

Give the microinstruction in symbolic form using the symbol LAD for MUX2.

8-4 Give a symbolic microinstruction that resets register $R4$ to 0 and updates Z and S status bits without affecting the other two status bits. (Try an exclusive-OR operation.)

8-5 Give a symbolic microinstruction that resets register $R3$ to 0 and resets the C status bit to 0. The other three status bits can be anything.

8-6 List the symbolic and binary microinstructions similar to Table 8-5 for the following register transfer statements.
(a) $R3 \leftarrow R1 - R2,\ CAR \leftarrow 17$
(b) $R5 \leftarrow \text{shr}\ (R4 + R5),\ CAR \leftarrow CAR + 1$
(c) If $(Z = 0)$ then $(CAR \leftarrow 21)$ else $(CAR \leftarrow CAR + 1)$
(d) $R6 \leftarrow R6,\ C \leftarrow 0,\ CAR \leftarrow CAR + 1$ (update Z and S)

8-7 Given the following binary microprogram starting from address 16 (similar to Table 8-8)
(a) List the corresponding symbolic microprogram as in Table 8-7.
(b) List the corresponding register transfer statements as in Table 8-6.

ROM address	Binary microprogram
010000	001 010 011 0101 000 0000 000000
010001	000 000 000 0000 000 0010 010011
010010	000 000 001 0000 001 0001 010011
010011	011 000 011 0001 000 0000 000000
010100	000 000 000 0000 000 1101 000000
010101	010 010 010 1100 000 0001 010000

8-8 Translate the microprogram of Table 8-9 to binary.

8-9 Write a microprogram to compute the average value of four unsigned binary numbers stored in registers $R1$, $R2$, $R3$, and $R4$. The average value is to be stored in register $R5$. The other two registers in the processor can be used for intermediate results. Care must be taken when an output carry occurs.

8-10 Write a microprogram (starting from address 1) that checks the sign of the binary number stored in register $R1$. If the number is positive, it is divided by two. If negative, it is multiplied by two. If an overflow occurs after the multiplication, $R1$ is reset to 0.

8-11 Write a microprogram that compares two unsigned binary numbers stored in registers $R1$ and $R2$. The register containing the smaller number is then reset to 0. If the two numbers are equal, both registers are reset to 0.

8-12 Write a microprogram to multiply two unsigned binary numbers. The multiplicand is in register $R1$, the multiplier is in register $R2$, and the product is formed in $R2$ (least significant half) and $R3$ (most significant half). Register $R4$ holds a binary number equal to the number of bits in the multiplier. Derive the algorithm in flowchart form and from it, write the microprogram.

8-13 List the contents of registers A, Q, P, and C (as in Table 8-10) during the process of multiplying the two unsigned binary numbers 11111 (multiplicand) and 10101 (multiplier).

8-14 Determine the time it takes to process the multiplication operation in the digital system described in Section 8-5. Assume that the Q register has n bits and the interval between two clock pulses is t seconds.

8-15 Prove that the multiplication of two n-bit numbers gives a product of no more than $2n$ bits. Show that this condition implies that no overflow can occur in the multiplier designed in Section 8-5.

8-16 The control state diagram of Figure 8-7(a) does not use variable Q_0 as a condition for state transition. Q_0 is used instead as part of a control function in the list of register transfers in Figure 8-7(b) and therefore, is omitted from the state table in Table 8-11.

Redesign the control for the binary multiplier so that Q_0 appears as a condition in the state diagram and as an input in the state table.

8-17 Consider the block diagram of the multiplier shown in Figure 8-5. Assume that the multiplier and multiplicand consist of 16 bits each.
(a) How many bits can be expected in the product, and where is it placed?
(b) How many bits are in the P counter and what is the binary number that must be loaded into it initially?
(c) Design the combinational circuit that checks for zero in the P counter.

8-18 Design a digital system that multiplies two unsigned binary numbers by the repeated addition method. For example, to multiply 5×4, the digital system evaluates the product by adding the multiplicand four times: $5 + 5 + 5 + 5 = 20$. Let the multiplicand be in register BR, the multiplier in register AR, and the product in register PR. An adder circuit adds the contents of BR to PR. A zero detect circuit Z checks when AR becomes zero after each time that it is decremented. Design the control by the hardwired method.

8-19 Design a digital system with three 16-bit registers AR, BR, and CR to perform the following operations:
(a) Transfer two 16-bit signed numbers to AR and BR after a start signal S is enabled.
(b) If the number in AR is negative, divide the contents of AR by two and transfer the result to register CR.
(c) If the number in AR is positive but non-zero, divide the contents of BR by two and transfer the result to register CR.
(d) If the number in AR is zero, reset register CR to 0.

8-20 The state diagram of a control unit is shown in Figure P8-20. It has four states and two inputs Y and Z. Design the control by
(a) the sequence register and decoder method and
(b) the one flip-flop per state method.

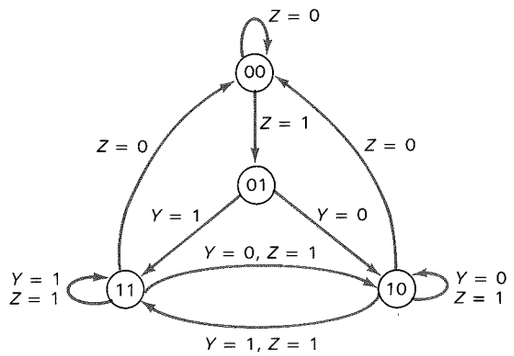


FIGURE P8-20
Control State Diagram for Problem 8-20

8-21 A digital computer has a memory unit with 24 bits per word. The instruction set consists of 150 different operations. There is only one type of instruction format with an operation code part and an address part. Each instruction is stored in one word of memory.
(a) How many bits are needed for the operation code part?

(b) How many bits are left for the address part of the instruction?
(c) How many words are there in memory and what must be the memory size?
(d) What is the largest unsigned binary number that can be accommodated in one word of memory?

8-22 A signed binary number is stored in address 153 of the simple computer. Write a program that will multiply this number by -1.

8-23 Derive the control functions listed in Table 8-14 from the information available in the flowchart of Figure 8-14.

8-24 Specify an instruction format for the simple computer that performs the following operation:

$$AC \leftarrow AC + M[\text{ADRS}] \qquad \text{Add to } AC$$

List the sequence of microoperations for executing this instruction and include it in the flowchart of Figure 8-14.

8-25 Repeat Problem 8-24 for the following instruction:

$$M[\text{ADRS}] \leftarrow M[\text{ADRS}] + AC \qquad \text{Add to memory}$$

8-26 Assume that the memory of the simple computer in Figure 8-13 has 64K $\times$ 8 words. This will require a 16-bit address to specify an 8-bit word in memory.
(a) What should be the number of bits in each of the six registers of the computer?
(b) How many words of memory are now required to store the instruction LDA ADRS (see Table 8-13)?
(c) List the sequence of microoperations needed to execute this instruction.

8-27 Show the path that the data takes in the hardware of the simple computer of Figure 8-16 when the control function C_7 is enabled. Include the effect of the clock. Assume that $DR = 10101010$ and $AC = 01011111$. What happens to DR and AC after the clock pulse transition when $C_7 = 1$?

9

COMPUTER INSTRUCTIONS AND ADDRESSING MODES

9-1 INTRODUCTION

The physical and logical structure of computers is normally described in reference manuals provided with the computer system. Such manuals explain the internal construction of the computer including the processor registers available and their logical properties. They list all hardware implemented instructions, specify their binary code format, and provide a precise definition of each instruction. A computer will usually have a variety of instructions and instruction code formats. It is the function of the control unit within the computer to interpret each instruction code and provide the necessary control signals needed to process each instruction.

A few simple examples of instructions and instruction code formats are presented in Section 8-7. We will now expand this presentation by introducing the most common instructions found in commercial computers. We will also investigate the various instruction formats that may be encountered in a typical computer.

The format of an instruction is depicted in a rectangular box symbolizing the bits of the instruction code. The bits of the binary instruction are divided into groups called *fields*. The most common fields found in instruction formats are

1. An *operation code* field that specifies the operation to be performed.
2. An *address* field that designates either a memory address or a code for choosing a processor register.
3. A *mode* field that specifies the way the address field is to be interpreted.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift type instruction or an operand field in an immediate type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift. The most common operations available in computer instructions are enumerated and discussed in Sections 9-5 through 9-8. The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address field. The various addressing modes that have been formulated for computers are presented in Section 9-3. The effect of including multiple address fields in an instruction is discussed in Section 9-2. Other topics covered in this chapter include stack organization and interrupt handling.

The various computer concepts introduced in this chapter are sometimes classified as being part of a broader field referred to as computer organization or computer architecture. They provide the necessary understanding of the internal organization and operation of digital computers. In the next chapter we use these topics to design a central processing unit (CPU).

9-2 ADDRESS FIELD

Operations specified by computer instructions are executed on some data stored in memory or in processor registers. Operands residing in memory are specified by their addresses. Operands residing in processor registers are specified by a register address. A register address is a binary code of n bits that specifies one of 2^n registers in the processor. Thus, a computer with 16 processor registers $R0$ through $R15$ will have in its instruction code a register address field of four bits. The binary code 0101, for example, will designate register $R5$.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends upon the internal organization of its registers. Most instructions fall in one of three types of organization:

1. Single accumulator organization.
2. Multiple register organization.
3. Stack organization.

An example of an accumulator type organization is the simple computer presented in Section 8-7. All operations are performed with the implied accumulator register. The instruction format in this type of computer uses one memory address field. For example, the instruction that specifies an arithmetic addition has only one address field symbolized by X.

ADD X

ADD is the symbol for the operation code of the instruction and X gives the address of the operand in memory. This instruction results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X.

An example of a processor unit with multiple registers is presented in Section 7-5. The instruction format in this type of computer needs three registers. Thus, the instruction for the arithmetic addition may be written in symbolic form as

use two or more processor register here, in the single accumulator we use an implied processor register it means we always use one spec

ADD R1, R2, R3

to denote the operation $R3 \leftarrow R1 + R2$. However, the number of register address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus, the instruction

ADD R1, R2

will denote the operation $R2 \leftarrow R2 + R1$. Registers $R1$ and $R2$ are the source registers and $R2$ is also the destination register.

Computers with multiple processor registers employ the MOVE instruction to symbolize the transfer of data from one location to another. The instruction

MOVE R1, R2

denotes the transfer $R2 \leftarrow R1$. Transfer type instructions need two address fields to specify the source operand and the destination of transfer.

Multiple register type computers employ two or three address fields in the instruction format. Each address field may specify a processor register or a memory address. An instruction symbolized by

ADD X, R1

will specify the operation $R1 \leftarrow R1 + M[X]$. It has two address fields, one for register $R1$ and the other for the memory address X.

The stack organization will be presented in Section 9-4. Computers with stack organization have instructions that require one address field for transferring data to and from the stack. Operation type instructions such as ADD do not need an address field because the operation is performed directly with the operands in the stack.

To illustrate the influence of the number of address fields on computer programs, we will evaluate the arithmetic statement

$$X = (A + B)(C + D)$$

using one, two, or three address instructions.

We will use symbols ADD, SUB, MUL, and DIV for the four arithmetic operations. The LOAD and STORE symbols will be used for the operation codes that transfer data to and from the AC register and memory. The symbol MOVE designates the operation code for transferring data in a multiple register type processor. We will assume that the operands are in memory addresses symbolized by the letters A, B, C, and D. The result is to be stored in memory at a location whose address is X.

Three Address Instructions

Computers with three address instruction formats can use each address field to specify either a processor register or a memory address for an operand. The program in symbolic form that evaluates $X = (A + B)(C + D)$ is shown below, together with an equivalent register transfer statement for each instruction.

ADD	A, B, R1	$R1 \leftarrow M[A] + M[B]$
ADD	C, D, R2	$R2 \leftarrow M[C] + M[D]$
MUL	R1, R2, X	$M[X] \leftarrow R1 * R2$

It is assumed that the computer has two processor registers, $R1$ and $R2$. The symbol $M[A]$ denotes the operand stored in memory at the address symbolized by A. The symbol $*$ designates multiplication.

The advantage of the three address format is that it results in short programs for evaluating arithmetic expressions. The disadvantage is that the binary coded instruction requires more bits to specify three addresses.

Two Address Instructions

Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory address. The program to evaluate $X = (A + B)(C + D)$ is as follows:

MOVE	A, R1	$R1 \leftarrow M[A]$
ADD	B, R1	$R1 \leftarrow R1 + M[B]$
MOVE	C, R2	$R2 \leftarrow M[C]$
ADD	D, R2	$R2 \leftarrow R2 + M[D]$
MUL	R2, R1	$R1 \leftarrow R1 * R2$
MOVE	R1, X	$M[X] \leftarrow R1$

The MOVE instruction moves or transfers the operands to and from memory and processor registers. The second operand listed in the symbolic instruction is assumed to be the destination where the result of the operation is transferred.

One Address Instructions

A computer with one address instructions uses an implied AC register. The program to evaluate the arithmetic statement is as follows:

LOAD	A	$AC \leftarrow M[A]$
ADD	B	$AC \leftarrow AC + M[B]$
STORE	T	$M[T] \leftarrow AC$
LOAD	C	$AC \leftarrow M[C]$
ADD	D	$AC \leftarrow AC + M[D]$
MUL	T	$AC \leftarrow AC * M[T]$
STORE	X	$M[X] \leftarrow AC$

All operations are done between the AC register and a memory operand. The symbolic address T designates a temporary memory location required for storing the intermediate result.

The program for evaluating the arithmetic expression using a stack will be given in Section 9-4.

9-3 ADDRESSING MODES

The operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced. Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

The availability of the addressing modes gives the experienced programmer a flexibility for writing programs that are more efficient in the number of instructions and the execution time.

Basic Computer Cycle

In order to understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer. The control unit of a computer is designed to execute each instruction in the program with a sequence of steps.

1. Fetch the instruction from memory into a control register.
2. Decode the instruction.
3. Locate the operands used by the instruction.
4. Fetch operands from memory (if necessary).
5. Execute the instruction in processor registers.
6. Store the results in the proper place.
7. Go back to step 1 to fetch the next instruction.

As explained in Section 8-8 there is one register in the computer called the program counter or *PC* that keeps track of the instructions in the program stored in memory. The *PC* holds the address of the instruction to be executed next and is incremented by one each time a word is read from the program in memory. The decoding done in step 2 is to determine the operation to be performed and the addressing mode of the instruction. The location of the operands in step 3 is determined from the addressing mode and the address field of the instruction. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

In some computers the addressing mode of the instruction is specified using a distinct binary code. Other computers use a common binary code that designates both the operation and the addressing mode of the instruction. Instructions may be defined with a variety of addressing modes and sometimes, two or more addressing modes are combined in one instruction.

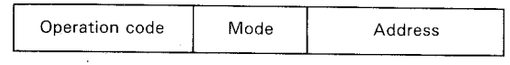


FIGURE 9-1
Instruction Format with Mode Field

An example of an instruction format with a distinct addressing mode field is shown in Figure 9-1. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a code for a processor register. Moreover, as discussed in the previous section, the instruction may have more than one address field. In that case, each address field is associated with its own particular addressing mode.

Implied Mode

Although most addressing modes modify the address field of the instruction, there is one mode that needs no address field at all. This is the implied mode. In this mode, the operand is specified implicitly in the definition of the operation code.

For example, the instruction "complement accumulator" is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, any instruction that uses an accumulator without a second operand is an implied mode instruction. Operation type instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

Immediate Mode

In the immediate mode, the operand is specified in the instruction itself. In other words, an immediate mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

Register and Register-Indirect Modes

It was mentioned previously that the address field of the instruction may specify either a memory address or a code for a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode. In this mode, the operands are in registers that reside within the processor unit of the computer. The particular register is selected from a register address field in the instruction format. An n-bit field can specify any one of 2^n registers.

In the *register-indirect* mode the instruction specifies a register in the processor whose content gives the address of the operand in memory. In other words, the selected register contains the memory address of the operand rather than the operand itself. Before using a register-indirect mode instruction, the programmer

we use this register in

must ensure that the memory address is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register-indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

An *autoincrement* or *autodecrement* mode is similar to the register-indirect mode except that the register is incremented or decremented after (or before) its address value is used to access memory. When the address stored in the register refers to an array of data in memory, it is convenient to increment the register after each access to the array. This can be achieved by using a register-increment instruction. However, because it is such a common requirement, some computers incorporate a special mode that automatically increments the content of the register after the memory data is accessed.

Direct-Addressing Mode

In the direct addressing mode the address field of the instruction gives the address of the operand in memory in a computational type instruction. It is the branch address in a branch type instruction.

An example of a computational type instruction is shown in Figure 9-2. The instruction in memory consists of two words. The first word, at address 250, has the operation code for "load to AC" and a mode field specifying a direct address. The second word, at address 251, contains the address field symbolized by ADRS and is equal to 500. The PC holds the address of the instruction. The instruction is brought from memory into two control registers and PC is incremented twice. The execution of the instruction results in the operation

$$AC \leftarrow M[\text{ADRS}]$$

Since ADRS = 500 and $M[500] = 800$, the AC receives the number 800. After the instruction is executed, the PC holds the number 252, which is the address of the next instruction in the program.

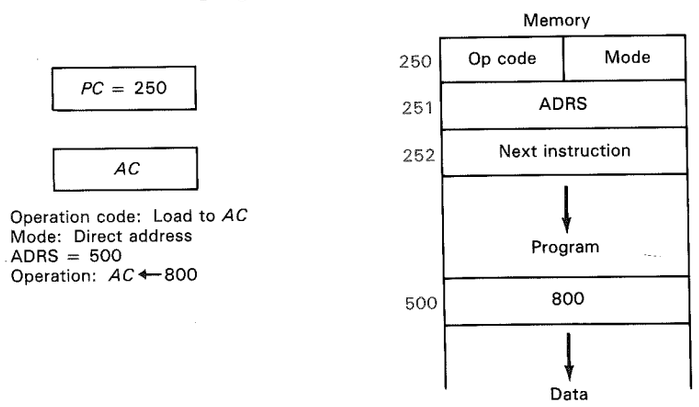


FIGURE 9-2
Example Demonstrating a Computational Type Instruction

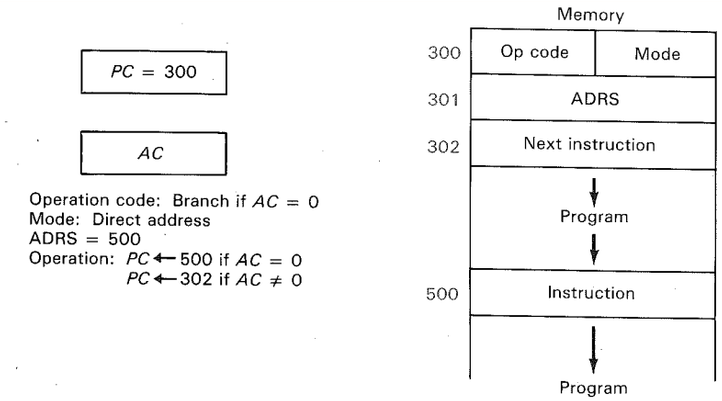


FIGURE 9-3
Example Demonstrating a Branch Type Instruction

Next consider a branch type instruction as shown in Figure 9-3. If the contents of AC is equal to 0, control branches to ADRS; otherwise the program continues with the next instruction in sequence. When $AC = 0$, the branch to address 500 is accomplished by loading the value of the address field ADRS into the PC. Control then continues with the instruction at address 500. When $AC \neq 0$, no branch occurs; and the PC, which was incremented twice after reading the instruction, holds the address 302, the address of the next instruction in sequence.

The address field of an instruction is used by the control unit of the computer to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction as given in the address field and the address used by the control when executing the instruction.

The *effective address* is defined to be the memory address obtained from the computation dictated by the addressing mode. The effective address is the address of the operand in a computational type instruction. It is the address where control branches, in response to a branch type instruction. In the direct-address mode, the effective address is equal to the address part of the instruction. In the register-indirect mode the effective address is the address stored in the selected register.

Indirect-Addressing Mode

In the indirect-addressing mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses the address part to access memory again to read the effective address.

Consider the instruction "load to AC" given in Figure 9-2. If the mode specifies an indirect address, the effective address is stored in $M[\text{ADRS}]$. Since ADRS = 500 and $M[\text{ADRS}] = 800$, the effective address is 800. This means that the operand

that is loaded into the *AC* is the one found in memory at address 800 (not shown in the figure).

Relative-Addressing Mode

Some addressing modes require that the address field of the instruction be added to the content of a specified register in the CPU in order to evaluate the effective address. In the relative-addressing mode the effective address is calculated as follows

Effective address = Address part of the instruction
+ Contents of *PC*

The address part of the instruction is considered as a signed number which can be either positive or negative. When this number is added to the contents of *PC*, the result produces an effective address whose position in memory is relative to the address of the next instruction in the program.

To clarify this with an example, let us assume that *PC* contains the number 250 and the address part of the instruction contains the number 500, as in Figure 9-2, with the mode field specifying a relative address. The instruction at location 250 is read from memory during the fetch phase and *PC* is incremented by one to 251. Since the instruction has a second word, control reads the address field into a control register and *PC* is incremented to 252. The effective address computation for the relative addressing mode is 252 + 500 = 752. The result is that the operand associated with the instruction is 752 locations relative to the location of the next instruction.

Relative addressing is often used in branch type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits than the number of bits required to designate the entire memory address.

Indexed-Addressing Mode

In the indexed-addressing mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the register. Any operand in the array can be accessed with the same instruction provided the index register contains the correct index value. The index register can be incremented to facilitate the access to consecutive operands.

Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when an index mode instruction is used. In computers with many processor registers, any one of the CPU registers can be used as an index register. In such a case, the index register must be specified explicitly with a register field within the instruction format.

A variation of the index mode is the *base-register* mode. In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to indexed addressing except that the register is called a base register instead of index register. The difference between the two modes is in the way they are used rather than in the way they are computed. An index register is assumed to hold an index number which is relative to the address field of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to the base address.

Summary of Addressing Modes

In order to show the differences between the various modes, we will investigate the effect of the addressing mode on the instruction shown in Figure 9-4. The instruction in addresses 250 and 251 is "load to *AC*" with the address field ADRS (or an operand NBR) equal to 500. The *PC* has the number 250 for fetching this instruction. The content of a processor register *R*1 is 400 and the *AC* receives the operand after the instruction is executed. In the direct mode the effective address is 500 and the operand to be loaded into the *AC* is 800. In the immediate mode the operand 500 is loaded into the *AC*. In the indirect mode the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 +

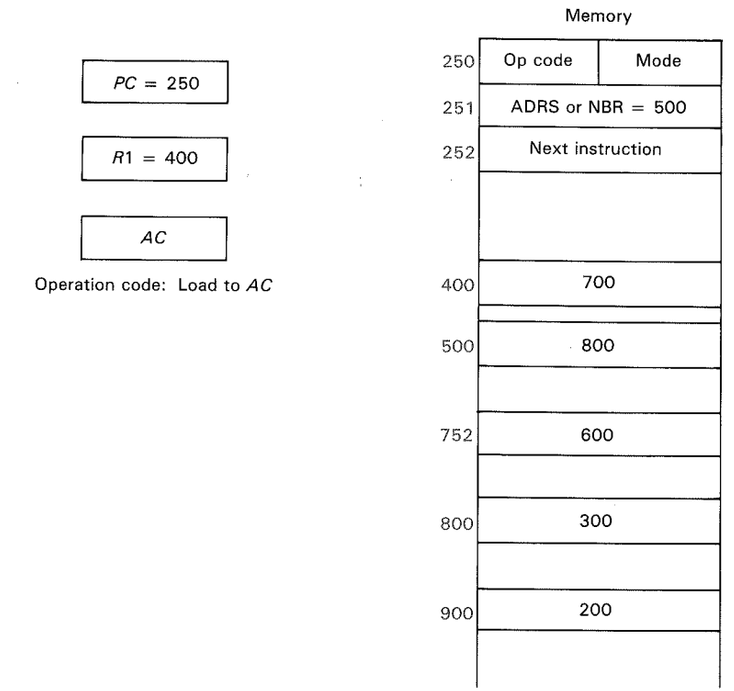


FIGURE 9-4
Numerical Example for Addressing Modes

TABLE 9-1
Symbolic Convention for Addressing Modes

Addressing mode	Symbolic convention	Register transfer	Refers to Figure 9-4: Effective address	Refers to Figure 9-4: Content of AC
Direct	LDA ADRS	$AC \leftarrow M[\text{ADRS}]$	500	800
Immediate	LDA #NBR	$AC \leftarrow \text{NBR}$	251	500
Indirect	LDA [ADRS]	$AC \leftarrow M[M[\text{ADRS}]]$	800	300
Relative	LDA $ADRS	$AC \leftarrow M[\text{ADRS} + PC]$	752	600
Index	LDA ADRS (R1)	$AC \leftarrow M[\text{ADRS} + R1]$	900	200
Register	LDA R1	$AC \leftarrow R1$	—	400
Register indirect	LDA (R1)	$AC \leftarrow M[R1]$	400	700

252 = 752 and the operand is 600. In the index mode the effective address is 500 + 400 = 900 assuming that $R1$ is the index register. In the register mode the operand is in $R1$ and 400 is loaded into the AC. In the register indirect mode the effective address is the contents of $R1$ and the operand loaded into the AC is 700.

Table 9-1 lists the value of the effective address and the operand loaded into the AC for seven addressing modes. The table also shows the operation with a register transfer statement and the symbolic convention for each addressing mode. LDA is the symbol for the load-to-accumulator operation code. In the direct mode we use the symbol ADRS for the address part of the instruction. The # symbol precedes the operand NBR in the immediate mode. The symbol ADRS enclosed in square brackets symbolizes an indirect address. Some computers use the symbol @ to designate an indirect address. The symbol $ before the address makes the effective address relative to PC. An index mode instruction is recognized by the symbol of a register that is placed in parentheses after the address symbol. The register mode is indicated by giving the name of the processor register following LDA. In the register indirect mode the name of the register that holds the effective address is enclosed in parentheses.

9-4 STACK ORGANIZATION

A very useful feature included in many computers is a memory stack, also known as a last-in first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack is sometimes compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.

The stack is useful for a variety of applications and its organization possesses special features that facilitate many data processing tasks. A stack is used in some electronic calculators and computers to facilitate the evaluation of arithmetic expressions. Its use in computers is mostly for handling of subroutines and interrupts as explained in Section 9-8.

A memory stack is essentially a portion of a memory unit accessed by an address that is always incremented or decremented after the memory access. The register

that holds the address for the stack is called a *stack pointer* (SP) because its value always points at the top item of the stack. The two operations of a stack are insertion and deletion of items. The operation of insertion is called *push* and it can be thought of as the result of pushing a new item onto the top of the stack. The operation of deletion is called *pop* and it can be thought of as the result of removing one item so that the stack pops out. However, nothing is physically pushed or popped in a memory stack. These operations are simulated by decrementing or incrementing the stack pointer register.

Figure 9-5 shows a portion of a memory organized as a stack. The stack pointer register SP holds the binary address of the item that is currently on the top of the stack. Three items are presently stored in the stack, A, B, and C in consecutive addresses 103, 102, and 101, respectively. Item C is on top of the stack, so SP contains 101. To remove the top item, the stack is popped by reading the item at address 101 and incrementing SP. Item B is now on top of the stack since SP contains address 102. To insert a new item, the stack is pushed by first decrementing SP and then writing the new item on top of the stack. Note that item C has been read out but not physically removed. This does not matter as far as the stack operation is concerned, because when the stack is pushed, a new item is written on top of the stack regardless of what was there before.

We assume that the items in the stack communicate with a data register DR. A new item is inserted with the push operation as follows:

$$SP \leftarrow SP - 1$$
$$M[SP] \leftarrow DR$$

The stack pointer is decremented so it points at the address of the next word. A memory write microoperation inserts the word from DR onto the top of the stack. Note that SP holds the address of the top of the stack and that $M[SP]$ denotes the memory word specified by the address presently in SP.

A new item is deleted with a pop operation as follows:

$$DR \leftarrow M[SP]$$
$$SP \leftarrow SP + 1$$

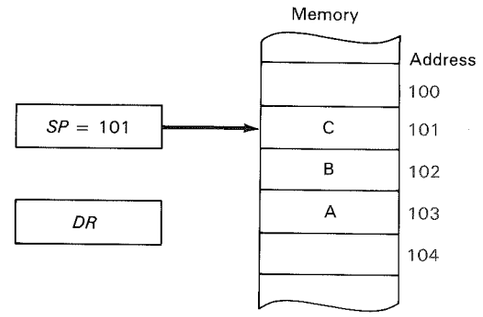


FIGURE 9-5
Memory Stack

The top item is read from the stack into *DR*. The stack pointer is then incremented to point at the next item in the stack.

The two microoperations needed for either the push or pop are access to memory through *SP* and updating *SP*. Which of the two microoperations is done first and whether *SP* is updated by incrementing or decrementing depends on the organization of the stack. In Figure 9-5 the stack grows by decreasing the memory address. The stack may be constructed to grow by increasing the memory address. In such a case, *SP* is incremented for the push operation and decremented for the pop operation. A stack may be constructed so that *SP* points at the next empty location above the top of the stack. In this case, the sequence of microoperations must be interchanged.

A stack pointer is loaded with an initial value. This initial value must be the bottom address of an assigned stack in memory. From then on, *SP* is automatically decremented or incremented with every push or pop operation. The advantage of a memory stack is that the processor can refer to it without having to specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation (RPN)

A stack organization is very effective for evaluating arithmetic expressions. The common mathematical method of writing arithmetic expressions imposes difficulty when evaluated by a computer. Conventional arithmetic expressions are written in the *infix* notation, with each operator written between the operands. Consider the simple arithmetic expression

$$A * B + C * D$$

The operator * denotes multiplication and is placed between the operands A and B and between C and D. The operator + denoting addition is between the two products. To evaluate the arithmetic expression it is necessary to compute the product $A * B$, store this product, compute the product $C * D$, and then sum the two products. From this simple example we see that to evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the sequence of operations that must be performed.

The Polish mathematician Jan Lukasiewicz proposed that arithmetic expressions be written in *prefix* notation. This representation, referred to as *Polish notation* places the operator before the operands. *Postfix* notation, referred to as *reverse Polish notation*, places the operator after the operands. The following examples demonstrate the three representations.

$A + B$	Infix notation
$+ A B$	Prefix or Polish notation
$A B +$	Postfix or reverse Polish notation

Reverse Polish notation, also known as RPN, is a form suitable for stack manipulation. The expression

$$A * B + C * D$$

is written in RPN as

$$A\ B * C\ D * +$$

and is evaluated as follows. Scan the expression from left to right. When an operator is reached, perform the operation with the two operands to the left of the operator. Remove the two operands and the operator and replace them with the number obtained from the operation. Continue to scan the expression and repeat the procedure for every operator until there are no more operators.

For the above expression we find the operator * after A and B. We perform the operation $A * B$ and replace A, B, and * by the product to obtain

$$(A * B)\ C\ D * +$$

where $(A * B)$ is a single quantity obtained from the product. The next operator is a * and its previous two operands are C and D; so we perform $C * D$ and obtain an expression with two operands and one operator:

$$(A * B)\ (C * D) +$$

The next operator is + and the two operands on its left are the two products; so we add the two quantities to obtain the result.

The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation. This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, then do multiplication and division, and finally, addition and subtraction. Consider the expression

$$(A + B) * [C * (D + E) + F]$$

To evaluate the expression we must first perform the arithmetic inside the parentheses and then evaluate the expression inside the square brackets. The multiplication of $C * (D + E)$ must be done prior to the addition of F. The last operation is the multiplication of the two terms between the parentheses and brackets. The expression can be converted to RPN by taking into consideration the operation hierarchy. The converted expression is

$$A\ B + D\ E + C * F + *$$

Proceeding from left to right, we first add A and B, then add D and E. At this point we are left with

$$(\mathrm{A} + \mathrm{B})\ (\mathrm{D} + \mathrm{E})\ \mathrm{C} * \mathrm{F} + *$$

Where $(A + \mathrm{B})$ and $(D + E)$ are each a single number obtained from the sum. The two operands for the next * are C and $(D + E)$. These two numbers are multiplied and the product added to F. The final * causes the multiplication of the last result with the number $(A + B)$. Note that all expressions in RPN are without parentheses.

The subtraction and division operations are not commutative, and the order of the operands is important. We define the RPN expression $A\ B -$ to mean $(A - B)$ and the expression $A\ B$ / to represent the division of A / B.

Stack Operations

Reverse Polish notation combined with a stack provides an efficient way to evaluate arithmetic expressions. This procedure is employed in some electronic calculators and also in some computers. The stack is particularly useful for handling long, complex problems involving chain calculations. It is based on the fact that any arithmetic expression can be expressed in parentheses-free Polish notation.

The procedure consists of first converting the arithmetic expression into its equivalent RPN. The operands are pushed onto the stack in the order that they appear. The initiation of an operation depends on whether we have a calculator or a computer. In a calculator, the operators are entered through the keyboard. In a computer they must be initiated by program instructions. The following operations are executed with the stack when an operation is specified: The two topmost operands in the stack are popped and used for the operation. The result of the operation is pushed into the stack, replacing the lower operand. By continuously pushing the operands onto the stack and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on top of the stack.

The following numerical example will clarify the procedure. Consider the arithmetic expression

$$(3 * 4) + (5 * 6)$$

In reverse Polish notation, it is expressed as

$$3\ 4\ *\ 5\ 6\ *\ +$$

Now consider the stack operations as shown in Figure 9-6. Each box represents one stack operation and the arrow always points to the top of the stack. Scanning the RPN expression from left to right, we encounter two operands. First the number 3 is pushed onto the stack, then the number 4. The next symbol is the multiplication operator *. This causes a multiplication of the two topmost numbers in the stack. The stack is popped twice for the operands and the product is pushed into the top of the stack in the lower position of the original operand. Next we encounter the two operands 5 and 6, and they are pushed onto the stack. The stack operation that results from the next * removes these two numbers and puts their product, 30, on the stack. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

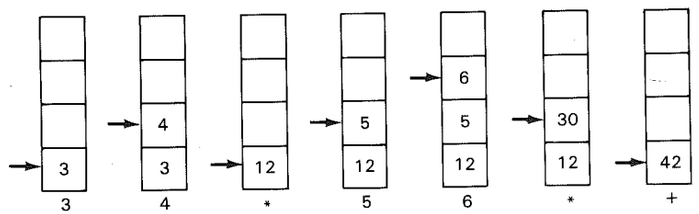


FIGURE 9-6
Stack Operations to Evaluate 3 * 4 + 5 * 6

Calculators that use RPN have an Enter key that functions as a push operation. When a number is keyed in following an Enter, the number is pushed onto the stack. When a number is keyed in following an operator, the calculator pushes the number on top of the stack and performs the operation with the two topmost numbers in the stack. Thus, the above expression is evaluated in an RPN calculator by using a sequence of keys as follows:

$$3\ \text{Enter}\ 4\ *\ 5\ \text{Enter}\ 6\ *\ +$$

Computer Stack

Most computers have a facility for a memory stack but only a few commercial computers have the appropriate instructions for evaluating arithmetic expressions with RPN. Such computers have a stack organized CPU with the top locations of the stack as registers. The rest of the stack is in memory. In this way, the operations that must be performed with the top two items of the stack are available in processor registers for manipulation with arithmetic circuits.

The PUSH and POP instructions require one address field to specify the source or destination operand. Operation type instructions for the stack such as ADD and MUL imply the two operands on top of the stack and do not require an address field in the instruction. The following program shows how the arithmetic statement

$$X = (A + B) * (C + D)$$

will be written for a computer stack (TOS stands for top of stack).

PUSH A	TOS $\leftarrow A$
PUSH B	TOS $\leftarrow B$
ADD	TOS $\leftarrow (A + B)$
PUSH C	TOS $\leftarrow C$
PUSH D	TOS $\leftarrow D$
ADD	TOS $\leftarrow (C + D)$
MUL	TOS $\leftarrow (C + D) * (A + B)$
POP X	$M[X] \leftarrow$ TOS

9-5 DATA TRANSFER INSTRUCTIONS

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction set of different computers differ from each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignment in the operation code field is different in different computers, even for the same operation. It may also happen that the symbolic name given to instructions is different in different computers, even for the same instruction. Nevertheless, there is a set of basic operations that most computers include among their

instructions. The basic set of instructions available in a typical computer is the subject that will be covered in the rest of this chapter.

Most computer instructions can be classified into three major categories:

1. Data transfer instructions.
2. Data manipulation instructions.
3. Program control instructions.

Data transfer instructions cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions perform arithmetic, logic, and shift operations. Program control instructions provide decision making capabilities and change the path taken by the program when executed in the computer. In addition to the basic instruction set, a computer may have other instructions that provide special operations for particular applications.

In this section we present a list of some typical data transfer instructions. In the next section we present the various data manipulation instructions. Program control instructions are discussed in Section 9-8.

Common Data Transfer Instructions

Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input and output, and among the processor registers themselves. Table 9-2 gives a list of eight typical data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. This is the assembly language abbreviation recommended by an IEEE standard (reference 6). It must be realized that different computers may use different mnemonics for the same instruction name.

The load instruction has been used to designate a transfer from memory to a processor register, such as an accumulator. The store instruction designates a transfer from a processor register into a memory word. The move instruction has been used in computers with multiple processor registers to designate a transfer from one register to another. It has also been used for data transfer between registers and memory and between two memory words. The exchange instruction exchanges information between two registers or a register and a memory word. The push and

TABLE 9-2
Typical Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOVE
Exchange	XCH
Push	PUSH
Pop	POP
Input	IN
Output	OUT

pop instructions transfer data between a memory stack and a processor register. The input and output instructions transfer data between processor registers and input and output devices.

Independent Versus Memory-Mapped I/O

The input and output (I/O) instructions are similar to the load and store instructions except that the transfer is from external registers instead of memory words. The computer is considered to have a certain number of input and output *ports* with one or more ports dedicated to communication with a specific input or output device. Each port is equivalent to an external register. The particular port is chosen by an address, similar to the way an address is used to select a word in memory. The input and output instructions include an address field in their format, for specifying the particular port selected for the transfer of input or output data.

There are two ways that the addresses are assigned for selecting both memory words and I/O ports. One method uses an independent I/O system and the other employs a memory-mapped configuration.

In the *independent I/O* system, the addresses assigned to memory and I/O ports are independent from each other. The computer has distinct input and output instructions with a separate address field that is interpreted by the control and used to select a particular I/O port. The independent I/O addressing method isolates memory and I/O selection so that the memory address range is not affected by the port address assignment. For this reason, this method is also referred to as an *isolated I/O* configuration.

The second alternative is to assign memory addresses to I/O ports. This configuration is referred to as *memory-mapped I/O*. There are no separate addresses for handling input and output transfers since I/O ports are treated as memory locations in one common address range. Each I/O port is regarded as a memory location, similar to a memory word. Computers that adopt the memory-mapped scheme have no distinct input or output instructions, because the same instructions are used for manipulating memory and for I/O data manipulation. For example, the load and store instructions used for memory transfer are also used for I/O transfer, provided that the address associated with the instruction is assigned to an I/O port and not to a memory word. The advantage of this scheme is that the same set of instructions can be used for reading and writing in memory and for the input and output of data.

9-6 DATA MANIPULATION INSTRUCTIONS

Data manipulation instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions.
2. Logical and bit manipulation instructions.
3. Shift instructions.

A list of data manipulation instructions will look very much like the list of microoperations given in Chapter 7. One must realize, however, that a microoperation is an elementary operation executed by the hardware of the computer. An instruction when executed in the computer must go through the instruction fetch phase to read the binary code from memory and the operands must be brought into processor registers according to the rules specified by the addressing mode. The last step is the execution of the instruction with one or more microoperations.

Arithmetic Instructions

The four basic arithmetic instructions are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and subtraction instructions. The multiplication and division must then be generated by means of special programs. The four basic arithmetic operations are sufficient for formulating solutions to any numerical problem when used with numerical analysis methods.

A list of typical arithmetic instructions is given in Table 9-3. The increment instruction adds one to the value stored in a register or memory word. A common characteristic of the increment operation when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts one from a value stored in a register or memory word. A number with all 0's when decremented, produces a number with all 1's.

The add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code. An arithmetic instruction may specify unsigned or signed integers, binary or decimal numbers, or floating-point data. The arithmetic operations with binary and decimal integers was presented in Chapter 1. The floating-point representation is used for scientific calculations and is presented in the next section.

The number of bits in any register is finite and therefore the results of arithmetic operations are of finite precision. Most computers provide special instructions to facilitate double precision arithmetic. A carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition with

TABLE 9-3
Typical Arithmetic Instructions

Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Subtract reverse	SUBR
Negate (2's complement)	NEG

two operands plus the value of the carry from the previous computation. Similarly, the "subtract with borrow" instruction subtracts two operands and a borrow which may have resulted from a previous operation. The reverse subtract instruction reverses the order of the operands and performs $B - A$ instead of $A - B$. The negate instruction performs the 2's complement of a signed number which is equivalent to multiplying it by -1.

Logical and Bit Manipulation Instructions

Logical instructions perform binary operations on strings of bits stored in registers or memory words. They are useful for manipulating individual bits or a group of bits that represent binary coded information. The logical instructions consider each bit of the operand separately and treat it as a Boolean variable. By proper application of the logical instructions, it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

Some typical logical and bit manipulation instructions are listed in Table 9-4. The clear instruction causes the specific operand to be replaced by 0's. The set instruction causes the operand to be replaced by 1's. The complement instruction inverts all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operand. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: A selected bit can be cleared to 0, set to 1, or be complemented. The three logical instructions are usually applied to do just that.

The AND instruction is used to clear to 0 a bit or a selected group of bits of an operand. For any Boolean variable X, the relationship $X \cdot 0 = 0$ and $X \cdot 1 = X$ dictates that a binary variable ANDed with a 0 produces a 0; but the variable does not change when ANDed with a 1. Therefore, the AND instruction is used to selectively clear bits of an operand by ANDing the operand with a string of bits that have 0's in the bit positions that must be cleared and 1's in the bit positions

TABLE 9-4
Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Set	SET
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC

that remain the same. The AND instruction is also called a *mask* because it masks (by inserting 0's) a selected portion of an operand.

The OR instruction is used to set to 1 a bit or a selected group of bits of an operand. For any Boolean variable X, the relationships $X + 1 = 1$ and $X + 0 = X$ dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0. Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with a string of bits with 1's in the bit positions that must be set to 1. The OR instruction is sometimes called a *bit set instruction*.

The XOR instruction is used to selectively complement bits of an operand. This is because of the Boolean relationships $X \oplus 1 = \overline{X}$ and $X \oplus 0 = X$. A binary variable is complemented when XORed with a 1 but does not change in value when XORed with 0. The XOR instruction is sometimes called a *bit complement instruction*.

A few other bit manipulation instructions are included in Table 9-4. Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Other status bits or flag bits can be set or cleared by program instructions in a similar manner.

Shift Instructions

Instructions to shift the content of an operand are provided in several variations. Shifts are operations in which the bits of the operand are moved to the left or the right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate type operations. In any case the shift may be to the right or to the left.

Table 9-5 lists four types of shift instructions. The logical shift inserts 0 into the end bit position after the shift. The end position is the leftmost bit for shift right and the rightmost positions for shift left. Arithmetic shifts conform to the rules for shifting signed numbers. The arithmetic shift right instruction preserves the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. The arithmetic shift left instruction inserts 0 into the end bit in the rightmost position and is identical to the logical shift left instruction. For this reason many computers do not provide

TABLE 9-5
Typical Shift Instructions

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right with carry	RORC
Rotate left with carry	ROLC

a distinct arithmetic shift left instruction when the logical shift left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost, as in a logical shift, but are rotated back into the other end. The rotate with carry instructions treats the carry bit as an extension of the register whose word is being rotated. Thus, a rotate left with carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit of the register into the carry, and shifts the entire register to the left.

Some computers have a multiple field format for the shift instruction. One field contains the operation code and the others specify the type of shift and the number of positions that an operand is to be shifted. A possible instruction format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

Here OP is the operation code field for specifying a shift; REG is a register address that specifies the location of the operand. TYPE is a 2-bit field that specifies one of the four types of shifts (logical, arithmetic, rotate, and rotate with carry); RL is a 1-bit field that specifies whether a shift is to the right or the left; and COUNT is a k-bit field that specifies up to $2^k - 1$ shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

9-7 FLOATING-POINT OPERATIONS

In many scientific calculations, the range of numbers is very large. The way to expand the range of numbers is to express them in floating-point notation. The floating-point number has two parts. The first part contains a *fraction* (sometimes called a *mantissa*) and the second part designates the position of the decimal point and is called the *exponent*. For example, the decimal number +6132.789 is represented in floating-point as

Fraction	Exponent
+.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+.6132789 \times 10^{+4}$.

Decimal floating-point numbers are interpreted as representing a number in the form

$$F \times 10^E$$

where F is the fraction and E the exponent. Only the fraction and the exponent are physically represented in computer registers. The base 10 and the decimal point of the fraction are assumed and are not shown explicitly. A floating-point binary number is represented in a similar manner except that it uses a base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as

Fraction	Exponent
01001110	000100

The fraction has a 0 in the leftmost position to denote a plus. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$F \times 2^E = +(.1001110)_2 \times 2^{+4}$$

A floating-point number is said to be *normalized* if the most significant digit of the fraction is nonzero. For example, the decimal fraction 0.350 is normalized but 0.0035 is not. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in both the fraction and exponent.

Floating-point representation increases the range of numbers that can be accommodated in a given register. Consider a computer with 48-bit registers. Since one bit must be reserved for the sign, the range of signed integers will be $\pm(2^{47} - 1)$ which is approximately $\pm 10^{14}$. The 48 bits can be used to represent a floating-point number with 36 bits for the fraction and 12 bits for the exponent. The largest positive or negative number that can be accommodated is

$$\pm(1 - 2^{-35}) \times 2^{+2047}$$

This number is derived from a fraction that contains a sign bit and 35 1's, and an exponent with a sign bit and eleven 1's. The maximum exponent is $2^{11} - 1$ or 2047. The largest number that can be accommodated is approximately equivalent to decimal 10^{615}.

Arithmetic Operations

Arithmetic operations with floating-point numbers are more complicated than with integer numbers and their execution takes longer and requires more complex hardware. Adding and subtracting two numbers requires that the decimal points be aligned since the exponent parts must be equal before adding or subtracting the fractions. The alignment is done by shifting one fraction and adjusting its exponent until it is equal to the other exponent. Consider the sum of the following floating-point numbers.

$$.5372400 \times 10^2$$
$$+.1580000 \times 10^{-1}$$

It is necessary that the two exponents be equal before the fractions can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When the fractions are stored in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the precision while the first method may cause an error. The usual alignment procedure is to shift the fraction that has the smaller exponent to the right

by a number of places equal to the difference between the exponents. After this is done, the fractions can be added.

$$.5372400 \times 10^2$$
$$+ \underline{.0001580 \times 10^2}$$
$$.5373982 \times 10^2$$

When two normalized fractions are added, the sum may contain an overflow digit. An overflow can be corrected by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros in the fraction as shown in the following example:

$$.56780 \times 10^5$$
$$- \underline{.56430 \times 10^5}$$
$$.00350 \times 10^5$$

A floating-point number that has a 0 in the most significant position is not normalized. To normalize the number, it is necessary to shift the fraction to the left and decrement the exponent until a nonzero digit appears in the first position. In the above example, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in normalized form.

Floating-point multiplication and division do not require an alignment of the fractions. The product can be formed by multiplying the two fractions and adding the exponents. Division is accomplished by dividing the fractions and subtracting the exponents.

In the examples shown we used decimal numbers to demonstrate the arithmetic operations of floating-point numbers. The same procedure applies to binary numbers with the exception that the base of the exponent is 2 instead of 10.

Biased Exponent

The fraction part of a floating-point number is usually in signed-magnitude representation. The exponent representation employed in most computers is known as a *biased representation*. The bias is an excess number which is added to the exponent so that internally all exponents become positive. As a consequence, the sign is removed from being a separate entity.

Consider for example the range of decimal exponents from −99 to +99. This is represented by two digits and a sign. If we use an excess 99 bias then the biased exponent e will be equal to $e = E + 99$, where E is the actual exponent. For $E = -99$, we have $e = -99 + 99 = 0$; and for $E = +99$, we have $e = 99 + 99 = 198$. In this way, the biased exponent is represented in a register as a positive number in the range from 000 to 198. Positive biased exponents have a range of numbers from 099 to 198. The subtraction of the bias 99 gives the positive values from 0 to +99. Negative biased exponents have a range from 098 to 000. The subtraction of 99 gives the negative values from −1 to −99.

The advantage of biased exponents is the fact that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. Another advantage is that the most negative exponent converts to a biased exponent with all 0's. The floating-point representation of zero is then a zero fraction and a zero biased exponent which is the smallest possible exponent.

Standard Operand Format

Arithmetic instructions that perform the operations with floating-point data use the suffix F. Thus, ADDF is an add instruction with floating-point numbers. There are two standard formats for representing a floating-point operand. The single precision data type consists of 32 bits and the double precision data type consists of 64 bits. When both types of data are available, the single precision instruction mnemonic uses an FS suffix and the double precision uses FL (for floating-point long).

The format of the IEEE standard (see reference 7) single precision floating-point operand is shown in Figure 9-7. It consists of 32 bits. The sign bit s designates the sign of the fraction. The biased exponent e contains 8 bits and uses an excess 127 number. The fraction f consists of 23 bits. The binary point is assumed to be immediately to the left of the most significant bit of the f field. In addition, an implied 1 bit is inserted to the left of the binary point which in effect expands the number to 24 bits representing a value from 1.0 to 2.0 (excluding 2). The component of the binary floating-point number that consists of a leading bit to the left of the implied binary point together with the fraction in the f field is called the *significand*. Some examples of f field values and the corresponding significands are shown below.

f Field	Significand	Decimal Equivalent
100 . . . 0	1.100 . . . 0	1.5
010 . . . 0	1.010 . . . 0	1.25
000 . . . 0	1.000 . . . 0	1.0

Even though the f field by itself may not be normalized, the significand is always normalized because it has a nonzero bit in the most significant position. Since normalized numbers must have a nonzero most significant bit, this 1 bit is not included explicitly in the format but must be inserted by the hardware during the arithmetic computations.

The exponent field uses an excess 127 bias value for normalized numbers. The range of valid exponents is from −126 (represented as 00000001) through +127 (represented as 11111110). The maximum (11111111) and minimum (00000000)

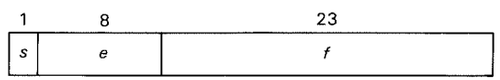


FIGURE 9-7
Standard Floating-Point Operand Format

TABLE 9-6
Evaluating Biased Exponents

Exponent *E* in decimal	Biased exponent $e = E + 127$	
	Decimal	Binary
−126	−126 + 127 = 1	00000001
−001	−001 + 127 = 126	01111110
000	000 + 127 = 127	01111111
+001	001 + 127 = 128	10000000
+126	126 + 127 = 253	11111101
+127	127 + 127 = 254	11111110

values that the e field can take are reserved to indicate exceptional conditions. Table 9-6 shows some values of biased exponents and their actual values.

Normalized numbers are numbers that can be expressed as floating-point operands where the e field is neither all 0's nor all 1's. The value of the number is derived from the three fields in the format of Figure 9-7 using the following formula:

$$(-1)^s 2^{e-127} \times (1.f)$$

The most positive normalized number that can be obtained has a 0 for the sign bit for positive sign, a biased exponent equal to 254 and an f field with 23 1's. This gives an exponent $E = 254 - 127 = 127$. The significand is equal to $1 + 1 - 2^{-23} = 2 - 2^{-23}$. The maximum positive number that can be accommodated is

$$+ 2^{127} \times (2 - 2^{-23})$$

The least positive normalized number has a biased exponent equal to 00000001 and a fraction of all 0's. The exponent is $E = 1 - 127 = -126$ and the significand is equal to 1.0. The smallest positive number that can be accommodated is $+2^{-126}$. The corresponding negative numbers are the same except that the sign bit is negative.

As mentioned before, exponents with all 0's or all 1's (decimal 255) are reserved for special conditions:

1. When $e = 255$ and $f = 0$, the number represents plus or minus infinity. The sign is determined from the sign bit s.
2. When $e = 255$ and $f \neq 0$, the representation is considered to be *not a number* or NaN, regardless of the sign value. NaNs are used to signify invalid operations such as the multiplication of zero times infinity.
3. When $e = 0$ and $f = 0$, the number denotes plus or minus zero.
4. When $e = 0$, and $f \neq 0$, the number is said to be *denormalized*. This is the name given to numbers with a magnitude less than the minimum value that is represented in the normalized format.

9-8 PROGRAM CONTROL INSTRUCTIONS

The instructions of a program are stored in successive memory locations. When processed by the control, the instructions are read from consecutive memory locations and executed one by one. Each time an instruction is fetched from memory,

TABLE 9-7
Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip next instruction	SKP
Call subroutine	CALL
Return from subroutine	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TEST

the program counter *PC* is incremented so that it contains the address of the next instruction in sequence. After the execution of a data transfer or data manipulation instruction, control returns to the fetch phase and uses *PC* to fetch the next instruction in sequence. In contrast, a program control instruction, when executed, may change the address value in *PC* and cause the flow of control to be altered. The change in *PC* as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature of digital computers since it provides control over the flow of program execution and a capability for branching to different program segments depending on previous computations.

Some typical program control instructions are listed in Table 9-7. The branch and jump instructions are used interchangeably to mean the same thing, although sometimes they are used to denote different addressing modes. The branch is usually a one address instruction. When executed, the branch instruction causes a transfer of the effective address into *PC*. Since *PC* contains the address of the instruction to be executed next, the next instruction will be fetched from the location specified by the effective address.

Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified effective address without any conditions. The conditional branch instruction specifies a condition for the branch such as a positive or negative result. If the condition is met, *PC* is loaded with the effective address and the next instruction is taken from this address. If the condition is not met, *PC* is not changed and the next instruction is taken from the next location in sequence.

The skip instruction does not need an address field. A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing *PC* during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds to the next instruction in sequence, where the programmer inserts an unconditional branch instruction. Thus, a conditional skip instruction followed by an unconditional branch instruction causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met.

The call and return instructions are used in conjunction with subroutine programs. Their performance and implementation is discussed later in this section.

The compare and test instructions are not program control instructions because they do not change the program sequence directly. They are listed in Table 9-7 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction performs a subtraction between two operands, but the difference is not retained. However, the status bit conditions are updated as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.

Conditional Branch Instructions

A conditional branch instruction is a branch instruction that may or may not cause a transfer of control depending on the state of the status bits in the processor. Each of these instructions tests different combination of status bits for a condition. If the condition is true, control is transferred to the effective address. If the condition is false, the program continues with the next instruction.

Table 9-8 gives a list of conditional branch instructions that depend directly on the status bits in the processor. Each instruction mnemonic is constructed with the letter B (for branch) and a letter for the name of the status bit. The letter N (for not) is included if the status bit is tested for a 0 condition. Thus BC is a branch if carry = 1, and BNC is branch if there is no carry (carry = 0).

The zero status bit is used to check if the result of an ALU operation is equal to zero or not. The carry bit is used to check the carry after addition or the borrow after subtraction of two operands in the ALU. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position. The sign bit reflects the state of the leftmost bit of the output from the ALU. $S = 0$ denotes a positive sign and $S = 1$, a negative sign. Therefore, a branch on plus checks for a sign bit of 0, and a branch on minus checks for a sign bit of 1. It must be realized that these two conditional branch instructions can be used to check the value of the leftmost bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations with signed numbers.

As stated previously, the compare instruction performs a subtraction of two operands, say $A - B$. The result of the operation is not transferred into a destination register, but the status bits are affected. The status bits provide information about the relative magnitude of A and B. Some computers provide special branch instructions that can be applied after the execution of a compare instruction. The specific conditions to be tested depend on whether the two numbers are considered to be unsigned or signed.

TABLE 9-8
Conditional Branch Instructions Relating to Status Bits

Branch condition	Mnemonic	Test condition
Branch if zero	BZ	$Z = 1$
Branch if no zero	BNZ	$Z = 0$
Branch if carry	BC	$C = 1$
Branch if no carry	BNC	$C = 0$
Branch if minus	BM	$S = 1$
Branch if plus	BP	$S = 0$
Branch if overflow	BV	$V = 1$
Branch if no overflow	BNV	$V = 0$

TABLE 9-9
Conditional Branch Instructions for Unsigned Numbers

Branch condition	Mnemonic	Condition	Status bits*
Branch if higher	BH	$A > B$	C or $Z = 0$
Branch if higher or equal	BHE	$A \geq B$	$C = 0$
Branch if lower	BL	$A < B$	$C = 1$
Branch if lower or equal	BLE	$A \leq B$	C or $Z = 1$
Branch if equal	BE	$A = B$	$Z = 1$
Branch if not equal	BNE	$A \neq B$	$Z = 0$

*Note that C here is a borrow bit

The relative magnitude of two unsigned binary numbers A and B can be determined by subtracting $A - B$ and checking the C and Z status bits. Most commercial computers consider the C status bit as a carry after addition and a borrow after subtraction. A borrow occurs when $A < B$ because the most significant position must borrow a bit to complete the subtraction. A borrow does not occur if $A \geq B$ because the difference $A - B$ is positive. The condition for borrow is the inverse of the condition for carry when the subtraction is done by taking the 2's complement of B. Computers that use the C status bit as a borrow after a subtraction, complement the output carry after adding the 2's complement of the subtrahend and call this bit a borrow.

The conditional branch instructions for unsigned numbers are listed in Table 9-9. It is assumed that a previous instruction updated status bits C and Z after a subtraction $A - B$. The words higher, lower, and equal are used to denote the relative magnitude between two unsigned numbers. The two numbers are equal if $A = B$. This is determined from the zero status bit Z which is equal to 1 because $A - B = 0$. A is lower than B and the borrow $C = 1$ when $A < B$. For A to be lower than or equal to B ($A \leq B$) we must have $C = 1$ or $Z = 1$. The relationship $A > B$ is the inverse of $A \leq B$ and is detected from the complement condition of the status bits. Similarly, $A \geq B$ is the inverse of $A < B$ and $A \neq B$ is the inverse of $A = B$.

The conditional branch instructions for signed numbers are listed in Table 9-10. Again it is assumed that a previous instruction updated the status bits S, V, and Z after a subtraction $A - B$. The words greater, less, and equal are used to denote the relative magnitude between two signed numbers. If $S = 0$, the sign of the difference is positive, and A must be greater than or equal to B, provided that $V = 0$ indicating that no overflow occurred. An overflow causes a sign reversal as discussed in Section 7-3. This means that if $S = 1$ and $V = 1$, there was a sign

TABLE 9-10
Conditional Branch Instructions for Signed Numbers

Branch condition	Mnemonic	Condition	Status bits
Branch if greater	BG	$A > B$	$(S \oplus V)$ or $Z = 0$
Branch if greater or equal	BGE	$A \geq B$	$S \oplus V = 0$
Branch if less	BL	$A < B$	$S \oplus V = 1$
Branch if less or equal	BLE	$A \leq B$	$(S \oplus V)$ or $Z = 1$

reversal and the result should have been positive, which makes A greater than or equal to B. Therefore, the condition $A \geq B$ is true if both S and V are equal to 0 or both are equal to 1. This is the complement of the exclusive-OR operation.

For A to be greater than but not equal to B ($A > B$), the result must be positive and nonzero. Since a zero result gives a positive sign, we must ensure that the Z bit is 0 to exclude the possibility of $A = B$. Note that the condition $(S \oplus V)$ or $Z = 0$ means that both the exclusive-OR operation and the Z bit must be equal to 0. The other two conditions in the table can be derived in a similar manner. The conditions BE (branch on equal) and BNE (branch on not equal) apply to signed numbers as well and can be determined from $Z = 1$ and $Z = 0$, respectively.

Subroutine Call and Return

A subroutine is a self contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the program. Each time a subroutine is called, a branch is made to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made again to return to the main program. A subroutine is also called a procedure.

The instruction that transfers control to a subroutine is known by different names. The most common names are call subroutine, call procedure, jump to subroutine, or branch to subroutine. The call subroutine instruction has a one address field. The instruction is executed by performing two operations. The address of the next instruction which is available in PC (called the return address) is stored in a temporary location and control is transferred to the beginning of the subroutine. The last instruction that must be inserted in every subroutine program is a *return* to the calling program. When this instruction is executed, the return address stored in the temporary location is transferred into PC. This results in a transfer of program control to the program that called the subroutine.

Different computers use different temporary locations for storing the return address. Some computers store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The advantage of using a stack for the return address is that when a succession of subroutines are called, the sequential return address can be pushed onto the stack. The return instruction causes the stack to pop, and the content of the top of the stack is then transferred to PC. In this way, the return is always to the program that last called the subroutine.

advantage of using st

A subroutine call instruction is implemented with the following microoperations:

$SP \leftarrow SP - 1$	Decrement the stack pointer
$M[SP] \leftarrow PC$	Store return address in stack
$PC \leftarrow$ Effective address	Transfer control to subroutine

The return instruction is implemented by popping the stack and transferring the return address to PC.

$PC \leftarrow M[SP]$	Transfer return address to PC
$SP \leftarrow SP + 1$	Increment stack pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in the memory stack. The programmer does not have to be concerned or remember where to return after the subroutine is executed.

we don't know when a interrupt in going to happen so it is necessary to store all the inforamatin as PC , status information and processor registers

9-9 PROGRAM INTERRUPT

The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence. Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an externally or internally generated request. Control returns to the original program after the service program is executed.

The interrupt procedure is in principle similar to a subroutine call except for three variations.

1. The interrupt is initiated by an external or internal signal rather than from the executions of an instruction.
2. The address of the service program that processes the interrupt request is determined by a hardware procedure rather than from the address field of an instruction.
3. In response to an interrupt it is necessary to store all the information that defines the state of the computer rather than storing only the program counter.

After the computer has been interrupted and the corresponding service program has been executed, the computer must return to exactly the same state that it was before the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing has happened. The state of the computer at the end of an execution of an instruction is determined from the contents of the program counter and other processor registers and the values of various status bits. The collection of all status bits is sometimes called the *program status word* or PSW. The PSW is stored in a separate register and contains the status information that characterizes the state of the computer. Typically, it includes the status bits from the last ALU operation and it specifies what interrupts are allowed to occur and whether the computer is operating in a user or system mode. Many computers have a resident operating system that controls and supervises all other programs. When the computer is executing a program that is part of the operating system, the computer is placed in a system mode. Certain instructions are privileged and can be executed in the system mode only. The computer is in a user mode when executing user programs. The mode of the computer at any given time is determined from special status bits in the PSW.

Some computers store only the program counter when responding to an interrupt. The program that performs the data processing for servicing the interrupt must then include instructions to store all register contents and the PSW. Other computers store the program counter and all status and register contents in response to an interrupt. In some cases, there exist two sets of processor registers within the computer. In this way, when the program switches from the user to the system mode in response to an interrupt, it is not necessary to store the contents of processor registers because each computer mode employs its own set of registers.

The hardware procedure for processing interrupts is very similar to the execution of a subroutine instruction. The state of the processor is pushed onto a memory stack and the address of the first instruction of the interrupt service program is loaded into *PC*. The address of the service program is chosen by the hardware. Some computers assign one memory location for the beginning address of the service program. The service program must then determine the source of the interrupt and proceed to service it. Some computers assign a separate memory location for each possible interrupt source. Sometimes, the interrupt source hardware itself supplies the address of the service routine. In any case, the computer must possess some form of hardware procedure for selecting a branch address for servicing the interrupt through program instructions.

Most computers will not respond to an interrupt until the instruction that is in the process of being executed is completed. Then, just before going to fetch the next instruction, the control checks for any interrupt signals. If an interrupt has occurred, control goes to a hardware interrupt cycle. During this cycle, the contents of *PC* (and sometimes the state of the processor) is pushed onto the stack. The branch address for the particular interrupt is then transferred to *PC* and control goes to fetch the next instruction. The service program is then executed starting from the address available in *PC*. The last instruction in the service program is a return from the interrupt instruction. When this instruction is executed, the stack is popped to retrieve the return address which is then transferred to *PC*.

Types of Interrupts

There are three major types of interrupts that cause a break in the normal execution of a program. They are classified as

1. External interrupts.
2. Internal interrupts.
3. Software interrupts.

External interrupts come from input or output devices, from timing devices, from a circuit monitoring the power supply, or from any other external source. Conditions that cause external interrupts are an input or output device requests a transfer of data, an external device has completed transfer of data, time out of an event has occurred, or power failure is pending. Time out interrupt may result from a program that is in an endless loop and thus exceeds its time allocation. Power failure interrupt may have as its service program a few instructions that transfer the complete state of the processor into a nondestructive memory such as a disk in the few milliseconds before power ceases.

Internal interrupts arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called *traps*. Examples of interrupts caused by internal conditions are register overflow, an attempt to divide by zero, an invalid operation code, memory stack overflow, and protection violation. These error conditions usually occur as a result of a premature termination of the instruction execution. The service programs that process the internal interrupts determine the corrective measure to be taken in each case.

External and internal interrupts are initiated by hardware conditions. A software

interrupt is initiated by executing an instruction. A software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program. The most common use of the software interrupt is associated with a system call instruction. This instruction provides means for switching from a user mode to a system mode. Certain operations in the computer may be assigned to the operating system which always operates in the system mode. For example, a complex input or output procedure must be done in a system mode. A program written by a user must run in user mode. When an input or output transfer is required, the user program causes a software interrupt, which stores the old computer state and brings in a new state that belongs to the system mode. The calling program must pass information to the operating system in order to specify the particular task that is being requested.

Processing External Interrupts

External interrupts may have single or multiple interrupt input lines. If there are more interrupt sources than there are interrupt inputs in the computer, two or more sources are ORed to form a common line. An interrupt signal may originate at any time during program execution. To ensure that no information is lost, the computer acknowledges the interrupt only after the execution of the current instruction is completed and only if the state of the processor warrants it.

Figure 9-8 shows a possible external interrupt configuration. The diagram shows four sources ORed to a single interrupt input signal. The CPU has within it an enable interrupt flip-flop (*EIF*) that can be set or reset with two program instructions. When *EIF* is reset, the interrupt signal is neglected. If *EIF* is set and the CPU is at the end of an instruction execution, the computer acknowledges the interrupt by enabling *INTACK* output. The interrupt source responds to *INTACK* by providing an interrupt vector address *IVAD* to the CPU. The program controlled *EIF* flip-flop allows the programmer to decide whether to use the interrupt facility or not. If an instruction to reset *EIF* has been inserted in the program, it means that the programmer does not want the program to be interrupted. An instruction to set *EIF* indicates that the interrupt facility will be active while the program is running.

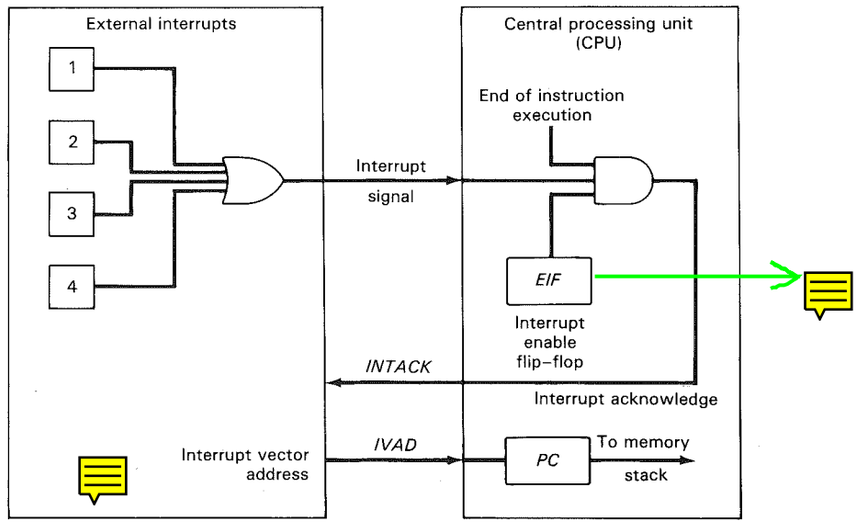


FIGURE 9-8
External Interrupt Configuration

The computer responds to an interrupt request signal if $EIF = 1$ and control has completed executing the present instruction. The microoperations that implement the interrupt cycle are as follows:

$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Store return address on top of stack
$EIF \leftarrow 0$	Reset enable interrupt flip-flop
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow IVAD$	Transfer interrupt vector address to *PC*
Go to fetch phase	

The return address available in *PC* is pushed onto the stack and *EIF* is reset to disable further interrupts. The program that services the interrupt can set *EIF* with an instruction whenever it is appropriate to enable other interrupts. The CPU assumes that the external source will provide an address in response to an acknowledge signal. This address is taken as the address of the first instruction of the program that services the interrupt. Obviously, a program must be written for that purpose and stored in memory.

The return from an interrupt is done with an instruction at the end of the service program similar to a return from a subroutine. The stack is popped and the return address is transferred to *PC*.

REFERENCES

1. MANO, M. M. *Computer System Architecture*. 2nd ed. Englewood Cliffs: Prentice-Hall, 1982.
2. TANENBAUM, A. S. *Structured Computer Organization*. 2nd ed. Englewood Cliffs: Prentice-Hall, 1984.
3. WAKERLY, J. F. *Microcomputer Architecture and Programming*. New York: Wiley, 1981.
4. LEWIN, M. H. *Logic Design and Computer Organization*. Reading, MA: Addison-Wesley, 1983.
5. HAYES, J. P. *Computer Architecture and Organization*. New York: McGraw-Hill, 1978.
6. *IEEE Standard for Microprocessor Assembly Language*. (IEEE Std 694-1985.) New York: The Institute of Electrical and Electronics Engineers.
7. *IEEE Standard for Binary Floating-Point Arithmetic*. (ANSI/IEEE Std 754-1985.) New York: The Institute of Electrical and Electronics Engineers.

PROBLEMS

9-1 Write a program to evaluate the arithmetic statement

$$X = (A + B * C) / (D + E * F - G * H)$$

(a) Use a general register computer with three address instructions.
(b) Use a general register computer with two address instructions.
(c) Use an accumulator type computer with one address instruction.

9-2 A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at $W + 1$) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated from the other addresses if the addressing mode of the instruction is
(a) direct; (b) indirect; (c) relative; (d) indexed.

9-3 A two-word relative mode branch type instruction is stored in memory at location 620 and 621 (decimal). The branch is made to an address equivalent to decimal 530. Let the address field of the instruction (stored at address 621) be designated by X.
(a) Determine the value of X in decimal.
(b) Determine the value of X in binary using 16 bits. (Note that the number is negative and must be in 2's complement. Why?)

9-4 How many times does the control unit refer to memory when it fetches and executes a two-word indirect addressing mode instruction if the instruction is (a) a computational type requiring an operand from memory; (b) a branch type.

9-5 What must be the address field of an indexed addressing mode instruction to make it the same as a register indirect mode instruction?

9-6 An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register $R1$ contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate; (c) relative; (d) register indirect; (e) indexed with $R1$ as the index register.

9-7 Convert the following arithmetic expressions from infix to reverse Polish notation.
(a) $A + B + C + D$
(b) $A * B + A * (B * D + C * E)$
(c) $A * B / C + D$
(d) $A + B * [C * D + E * (F + G)]$

9-8 Convert the following arithmetic expressions from reverse Polish notation to infix notation.
(a) $A\ B\ C\ D\ E\ *\ /\ -\ +$
(b) $AB\ *\ CD\ *\ +\ E\ F\ *\ +$
(c) $AB\ +\ C\ *\ D\ +$
(d) $AB\ +\ C\ *\ D\ +\ E\ *\ F\ +\ G\ *\ H\ +$

9-9 Convert the following numerical arithmetic expression into reverse Polish notation and show the stack operations for evaluating the numerical result.

$$(3 + 4)\ [10\ (2 + 6) + 8]$$

9-10 A first-in first-out (FIFO) memory has a memory organization that stores information in such a manner that the item that is stored first is the first item that is retrieved. Show how a FIFO memory operates with three counters. A write counter WC holds the address for writing into memory. A read counter RC holds the address for reading from memory. An available storage counter ASC indicates the number of words stored in FIFO. ASC is incremented for every word stored and decremented for every item that is retrieved.

9-11 Write a program to evaluate the arithmetic statement given in Problem 9-1 using computer stack instructions.

9-12 A computer with independent I/O system has the following input and output instructions.

IN ADRS

OUT ADRS

Where ADRS is the address of an I/O register port. Give the equivalent instructions for a computer with memory-mapped I/O.

9-13 Assuming an 8-bit computer, show the multiple precision addition of the two 32-bit unsigned numbers listed below using the add with carry instruction. Each byte is expressed as a 2-digit hexadecimal number.

6E C3 56 7A + 13 55 6B 8F

9-14 Perform the logic AND, OR, and XOR with the two binary strings 10011100 and 10101010.

9-15 Given the 16-bit value 1001101011001101. What operation must be performed in order to
(a) clear the first 8 bits to 0
(b) set the last 8 bits to 1
(c) complement the middle 8 bits.

9-16 An 8-bit register contains the value 01111011 and the carry bit is equal to 1. Perform the eight shift operations given by the instructions listed in Table 9-5. Each time start from the initial value given above.

9-17 Show how the following two floating-point numbers are to be added to get a normalized result.

$$(-.13567 \times 10^{+3}) + (+.67430 \times 10^{-1})$$

9-18 A 36-bit floating-point number consists of 26 bits plus sign for the fraction and 8 bits plus sign for the exponent. What are the largest and smallest positive quantities for normalized numbers?

9-19 A 30-bit register holds a floating-point decimal number in BCD. The fraction occupies 21 bits for five decimal digits and a sign. The exponent occupies 9 bits for two decimal digits and a sign. What are the largest and smallest positive quantities for normalized decimal numbers?

9-20 A 4-bit exponent uses an excess 7 number for the bias. List all biased binary exponents from $+8$ through -7.

9-21 The IEEE standard double-precision floating-point operand format consists of 64 bits. The sign occupies one bit, the exponent has 11 bits, and the fraction occupies 52 bits. The exponent bias is 1023. There is an implied bit to the left of the binary point in the fraction. Infinity is represented with a biased exponent equal to 2047 and a fraction of 0.
(a) Give the formula for evaluating normalized numbers.

(b) List a few biased exponents in binary as is done in Table 9-6.
(c) Calculate the largest and smallest positive normalized numbers that can be accommodated.

9-22 Prove that if the equality $2^x = 10^y$ holds, then $y = 0.3x$. Using this relationship, calculate the largest and smallest normalized floating-point numbers in decimal that can be accommodated in the single precision IEEE format.

9-23 It is necessary to branch to ADRS if the bit in the least significant position of the operand in a 16-bit register is equal to 1. Show how this can be done with the TEST (Table 9-7) and BNZ (Table 9-8) instructions.

9-24 Consider the two 8-bit numbers $A = 01000001$ and $B = 10000100$.
(a) Give the decimal equivalent of each number assuming that (1) they are unsigned; and (2) they are signed.
(b) Add the two binary numbers and interpret the sum assuming that the numbers are (1) unsigned; and (2) signed.
(c) Determine the values of the C (carry), Z (zero), S (sign), and V (overflow) status bits after the addition.
(d) List the conditional branch instructions from Table 9-8 that will have a true condition.

9-25 The program in a computer compares two unsigned numbers A and B by performing a subtraction $A - B$ and updating the status bits. Let $A = 01000001$ and $B = 10000100$.
(a) Evaluate the difference and interpret the binary result.
(b) Determine the values of status bits C (borrow) and Z (zero).
(c) List the conditional branch instructions from Table 9-9 that will have a true condition.

9-26 The program in a computer compares two signed numbers A and B by performing the subtraction $A - B$ and updating the status bits. Let $A = 01000001$ and $B = 10000100$.
(a) Evaluate the difference and interpret the binary result.
(b) Determine the value of status bits S (sign), Z (zero), and V (overflow).
(c) List the conditional branch instructions from Table 9-10 that will have a true condition.

9-27 The content of the top of a memory stack is 5320. The content of the stack pointer SP is 3560. A two-word call subroutine instruction is located in memory at address 1120 followed by the address field of 6720 at location 1121. What are the contents of PC, SP and the top of the stack
(a) Before the call instruction is fetched from memory.
(b) After the call instruction is executed.
(c) After the return from subroutine.

9-28 What are the basic differences between a branch instruction, a call subroutine instruction, and program interrupt?

9-29 Give five examples of external interrupts and five examples of internal interrupts. What is the difference between a software interrupt and subroutine call?

9-30 A computer responds to an interrupt request signal by pushing onto the stack the contents of PC and the current PSW (program status word). It then reads a new PSW from memory from the location given by the interrupt vector address ($IVAD$). The first address of the service program is taken from memory at location $IVAD + 1$.
(a) List the sequence of microoperations for the interrupt cycle.
(b) List the sequence of microoperations for the return from interrupt instruction.

10

DESIGN OF A CENTRAL PROCESSING UNIT (CPU)

10-1 INTRODUCTION

The central processing unit is the central component of a digital computer. Its purpose is to interpret instruction codes received from memory and perform arithmetic, logic, and control operations with data stored in internal registers, memory words, or I/O interface units. Externally, the CPU provides a bus system for transferring instructions, data, and control information to and from the modules connected to it.

A typical CPU is usually divided in two parts: the processor unit and the control unit. The processor unit consists of an arithmetic logic unit, a number of registers, and internal buses that provide the data paths for the transfer of information between the registers and the arithmetic logic unit. The control unit consists of a program counter, an instruction register, and timing and control logic. The control logic may be either hardwired or microprogrammed. If it is hardwired, registers, decoders, and a random set of gates are connected to provide the logic that determines the actions required to execute the various instructions. A microprogrammed control unit uses a control memory to store microinstructions and a sequencer to determine the order by which the microinstructions are read from control memory. The purpose of this chapter is to present a typical microprogrammed CPU and to show the detailed logic design of the unit including the microprogram.

The material in this chapter is presented mostly in the form of diagrams and tables. Background information from previous chapters is not repeated here. However, references are given to previous sections of the book where more detailed information can be found.

The design of the sample CPU is carried out in six parts in the following manner:

Section 10-2 shows the procedure for designing an arithmetic logic shift unit.
Section 10-3 presents the hardware of the processor unit.
Section 10-4 specifies a set of instruction formats for the CPU.
Section 10-5 specifies the microinstruction formats for the control memory.
Sections 10-6 through 10-8 show how to write the microprogram for the CPU.
Section 10-9 presents the hardware of the control unit.

The CPU can be designed from the hardware specifications given in Sections 10-3 and 10-9 and by loading the binary microprogram into control memory.

The position of the CPU among the other computer modules is shown in Figure 10-1. The memory unit stores the instructions, data, and other binary coded information needed by the processor. It is assumed that the access time of the memory is fast enough for the CPU to be able to read or write a word within one clock pulse period. Even though this restriction is not always applicable in a computer system, we assume it here in order to avoid complicating the design with a wait

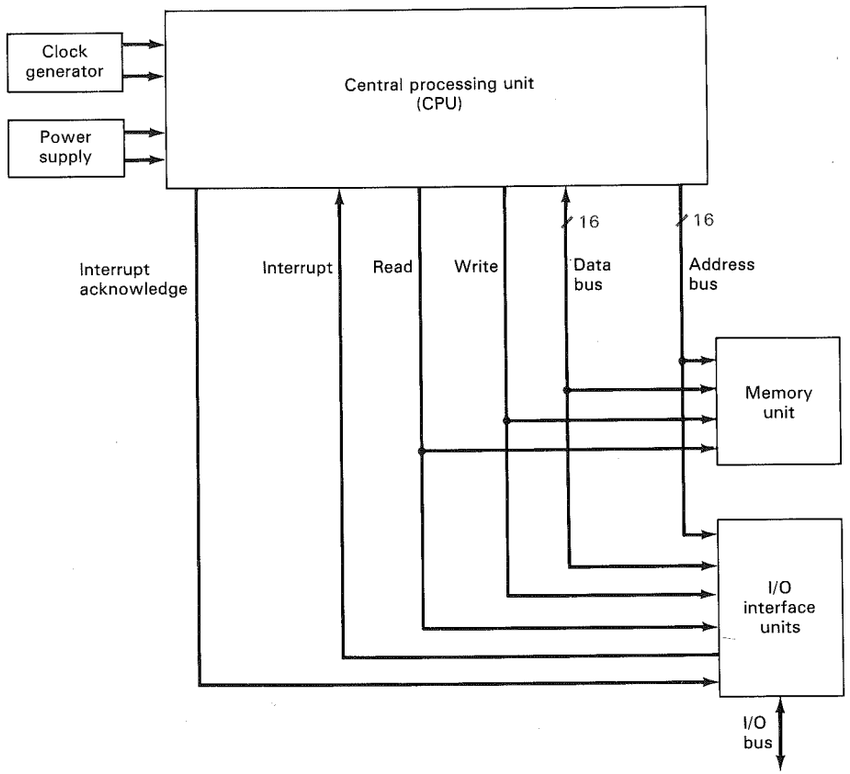


FIGURE 10-1
Block Diagram of the Sample Computer

period that stops the operations within the CPU until the memory cycle is completed.

The interface units provide a path for transferring information between the CPU and external input or output devices connected through the I/O bus. We assume a memory-mapped I/O method of addressing between the CPU and the registers in the interface units. By this method, there are no input or output instructions in the computer because the CPU can manipulate the data residing in interface registers with the same instructions that are used to manipulate memory locations. Each interface unit is organized as a set of registers that respond to the read or write signals in the normal address space of the 16-bit address. Typically, the bulk of the address space is assigned to memory words and a small segment is reserved for interface registers.

The communication between the CPU and external modules takes place via the address and data buses. The address bus consists of 16 lines and is unidirectional from the CPU to the other units. The data bus also consists of 16 lines but is bidirectional, allowing the information to flow in either direction. The read and write control lines from the CPU specify the direction of transfer in the data bus. The read signal informs the selected memory word or interface register to transfer data into the CPU. The write signal informs the selected unit that a data word is available for transfer from the CPU to the selected memory word or interface register.

Two other control lines are shown in Figure 10-1 for the interrupt and interrupt acknowledge signals. Their function is explained in conjunction with Figure 9-8. (The interrupt vector address IVAD shown in Figure 9-8 is transferred through the data bus.) Including the clock input and power supply lines, there are 40 external terminals in the CPU.

10-2 ARITHMETIC LOGIC SHIFT UNIT (ALSU)

The arithmetic logic shift unit (ALSU) is a combinational circuit that performs a number of arithmetic, logic, and shift microoperations. The control unit activates the microoperations when it executes the instructions in the program. The ALSU has a number of selection lines to select a particular operation in the unit. The selection variables are decoded by means of multiplexers so that k selection variables can specify up to 2^k distinct operations.

Figure 10-2 shows the general configuration of the ALSU. It is divided into four distinct subunits each of which is designed separately. There are 16 data lines for input A and another 16 data lines for input B. Each of the four subunits receives one or both sets of input data lines and produces a 16-bit output. In addition, all four subunits receive the two selection variables S_1 and S_0 and they respond to these two selection variables by generating one of four internal microoperations. The other two selection variables S_2 and S_3 are used to select one, and only one, of the four output functions from the subunits. This is then applied to the common 16-bit output F of the ALSU. Thus, an arithmetic operation is selected when $S_3S_2 = 00$, a logic operation is selected when $S_3S_2 = 01$, and a shift right or left operation is selected when $S_3S_2 = 10$ or 11, respectively.

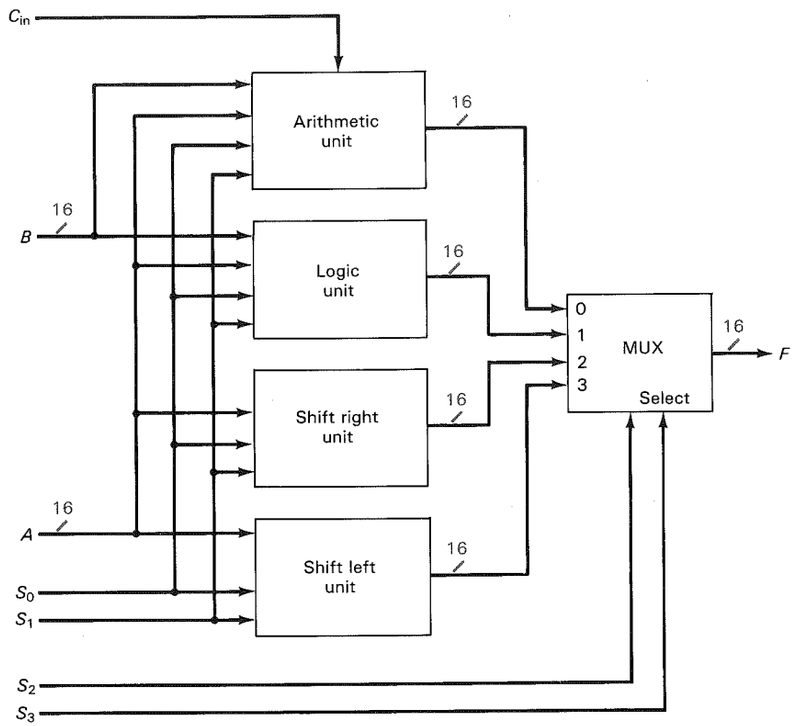


FIGURE 10-2
Arithmetic Logic Shift Unit (ALSU)

The detailed design of the arithmetic and logic units are described in Section 7-6. The input C_{in} acts as a third selection variable for the arithmetic unit to provide a total of eight arithmetic microoperations. The other three units each generate four microoperations. Thus there are a total of 20 microoperations in the ALSU. The list of the 20 microoperations is given in Table 10-1. Input C_{in} has a meaning only during an arithmetic operation. During logic operations it has no effect and can be assumed to be equal to 0. The first 12 operations are taken from Table 7-6 in Section 7-6. The last eight operations are for the two shift units whose internal construction is described here.

Shift Unit

The shift right unit is shown in Figure 10-3. The 16 inputs of A are designated A_0 through A_{15} and the 16 outputs of F as F_0 through F_{15}. The shift right operation is accomplished by connecting each input A_i to an output F_{i-1} for $i = 1, 2, 3, \ldots, 15$. The rightmost bit in position A_0 is shifted out and stored in the carry flip-flop for future inspection if necessary. This is done only during the shift right operation when $S_3S_2 = 10$. The output bit from the left most position in F_{15} comes from the output of a multiplexer whose selection inputs S_1 and S_0 determine the type of

TABLE 10-1
ALSU Function Table

S_3	S_2	S_1	S_0	$C_{in} = 0$		$C_{in} = 1$	
0	0	0	0	$F = A$	Transfer	$F = A + 1$	Increment
0	0	0	1	$F = A + B$	Add	$F = A + B + 1$	Add plus 1
0	0	1	0	$F = A - B - 1$	Subtract minus 1	$F = A - B$	Subtract
0	0	1	1	$F = A - 1$	Decrement	$F = A$	Transfer
0	1	0	0	$F = A \wedge B$	AND		
0	1	0	1	$F = A \vee B$	OR		
0	1	1	0	$F = A \oplus B$	XOR		
0	1	1	1	$F = \overline{A}$	Complement		
1	0	0	0	$F =$ Logical shift right A			
1	0	0	1	$F =$ Rotate right A			
1	0	1	0	$F =$ Rotate right A with carry			
1	0	1	1	$F =$ Arithmetic shift right A			
1	1	0	0	$F =$ Logical shift left A			
1	1	0	1	$F =$ Rotate left A			
1	1	1	0	$F =$ Rotate left A with carry			
1	1	1	1	$F =$ Arithmetic shift left A			

shift. When the selection inputs are equal to 00, the type of operation is a logical shift right which inserts a 0 into F_{15}. A rotate right operation when the selection inputs are 01 requires that $F_{15} = A_0$. F_{15} receives the input carry when the operation is a rotate right with carry. The previous carry may come from either an arithmetic operation or from a previous shift operation. With selection inputs equal to 11 we have an arithmetic shift right operation which requires that the sign bit in the leftmost position remain unchanged so that $F_{15} = A_{15}$.

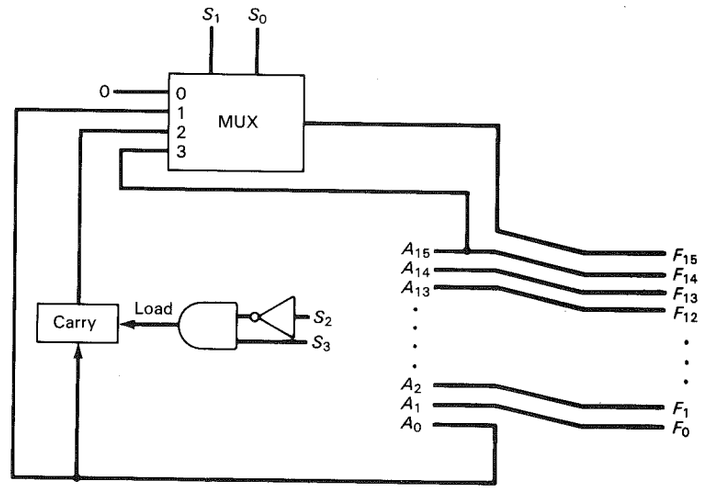


FIGURE 10-3
Shift Right Unit

The shift left unit is similar to Figure 10-3 except that the shift is to the left so that each input A_i is connected to an output F_{i+1}. Output F_0 receives a value from a 4×1 multiplexer with selection variables S_1 and S_0. The inputs of this multiplexer determine the type of shift much as the ones defined in Figure 10-3 did. Note that the arithmetic shift left operation is identical to the logical shift left with 0 insertion when the signed numbers are in 2's complement representation.

10-3 PROCESSOR UNIT

The block diagram of the processor unit is shown in Figure 10-4. It consists of a file of 14 registers, an ALSU, and three buses that provide the data path within the unit. Two sets of multiplexers select a register or the input data for the ALSU. A decoder selects a destination register by enabling its load input. All registers and the input and output data are 16 bits wide. The 17 bits of the control word that select a microoperation in the processor unit are divided into four fields: The *A* field specifies the input to the left side of the ALSU. The *B* field specifies the

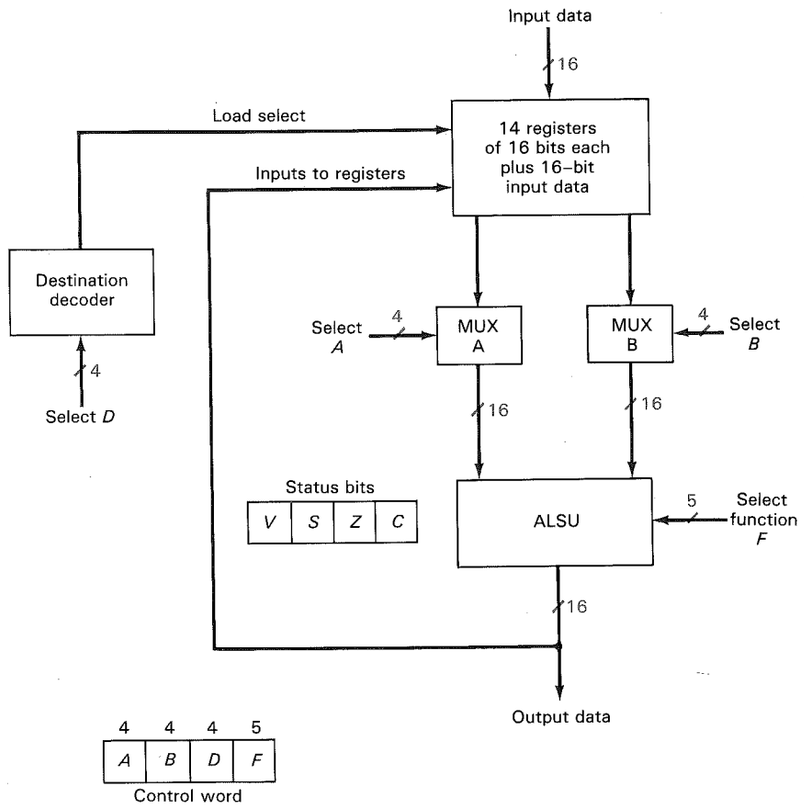


FIGURE 10-4
Processor Unit Hardware

input to the right side of the ALSU. The *D* field specifies the destination register. The *F* field designates the operation in the ALSU as given in Table 10-1. A detailed explanation of the processor unit is to be found in Sections 7-5 through 7-8.

The 14 registers in the processor unit are assigned special tasks by the CPU. The first eight registers, designated *R*0 through *R*7, are available to the user as general purpose registers that can be manipulated by program instructions. One register acts as a program counter and the other five registers are used by the control unit for storing temporary results. The status bits associated with the ALSU are symbolized by *C, Z, S,* and *V*. They are altered to reflect the results of the carry, zero, sign and overflow in the ALSU.

CPU Block Diagram

The position of the processor unit within the CPU is shown in Figure 10-5. The CPU consists of the processor unit, the control unit, four additional registers, and three-state buffers for two external buses. An address for memory or I/O interface is transferred to the address register *AR*. The data for a write operation is transferred to the data output register *DOR*. Information received after a read operation

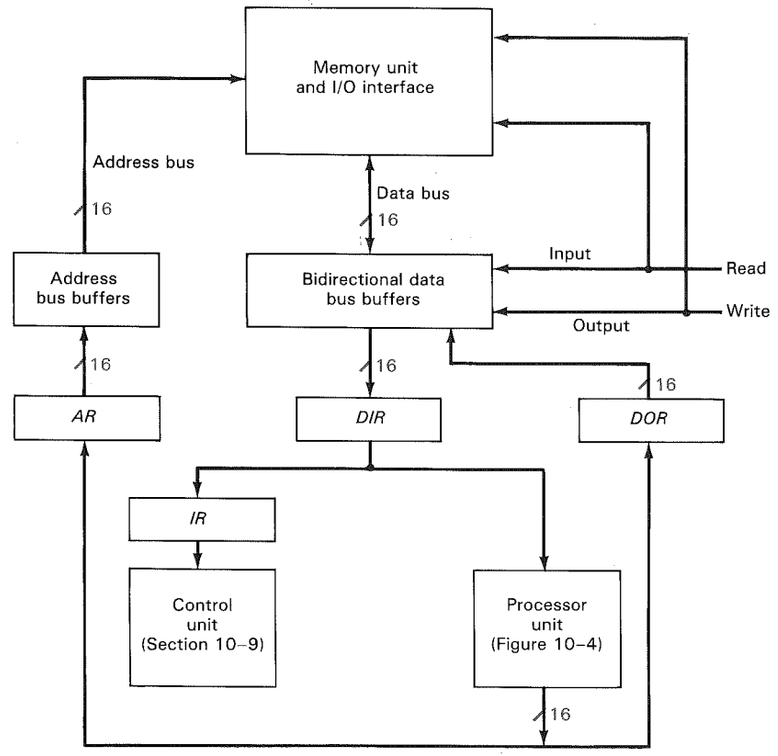


FIGURE 10-5
Block Diagram of CPU

is available in the data input register *DIR*. Instructions read during the fetch phase are transferred from *DIR* to the instruction register *IR*. The contents of *IR* determine the operations in the control unit. Data read from memory into *DIR* are applied to the data input lines in the processor unit and then manipulated in the ALSU. The construction of bus buffers for bidirectional transfer was covered in Section 7-4. The detailed construction of the control unit will be presented in Section 10-9.

The read and write signals are generated in the control unit by enabling corresponding flip-flops for one clock pulse period. It is assumed that the access time of the memory is sufficiently fast to be able to synchronize the operation of the memory with the clock pulses in the CPU. This means that the memory or interface responds to the read signal by placing the selected word in the data bus with enough time left to transfer it into the *DIR* prior to the next clock pulse from the one that enables the flip-flop read operation. In this way, the control can issue a read operation with one clock pulse and expect the data read from memory to be available in *DIR* at the next clock pulse. If we do not impose this requirement, the operations in the CPU must be delayed a number of clock pulses until the data word from memory is available in *DIR*. Note that this implies that the data input into *DIR* is not synchronized with the clock pulses in the CPU. *DIR* is actually a latch that follows the input data from the data bus as long as the input path is enabled in the bidirectional data bus by the read signal.

DIR is used only after a read operation to receive the data from the data bus. For a write operation, the data must be placed in *DOR* while the address is in *AR*. The write flip-flop when enabled by the control unit opens the output path through the bidirectional data bus buffers. The memory or interface stores the contents of the data bus in the word or interface register selected by the address bus within a time equal to one period of a CPU clock pulse.

Programmer Model

The computer as seen by the programmer and user is shown in Figure 10-6. Eight processor registers are used for manipulating data through program instructions. The last two registers are used as stack pointer and index register.The program counter can be changed by the programmer by means of branch type instructions. The status bits are affected by certain ALSU operations. The other parts of the computer of interest to the user are the ALSU, the memory unit, and the input and output addresses of external devices.

Even though the user and programmer can communicate with only 9 registers, there are actually 20 registers in the CPU. The other 11 registers are exclusively used by the control unit for internal operations.

CPU Registers

The 20 registers of the CPU can be divided into three groups. 14 registers are in the register file inside the processor unit. Three registers communicate with the data and address buses, and three registers are associated with the control unit. The following is a list of the 14 registers in the processor unit that communicate with the ALSU.

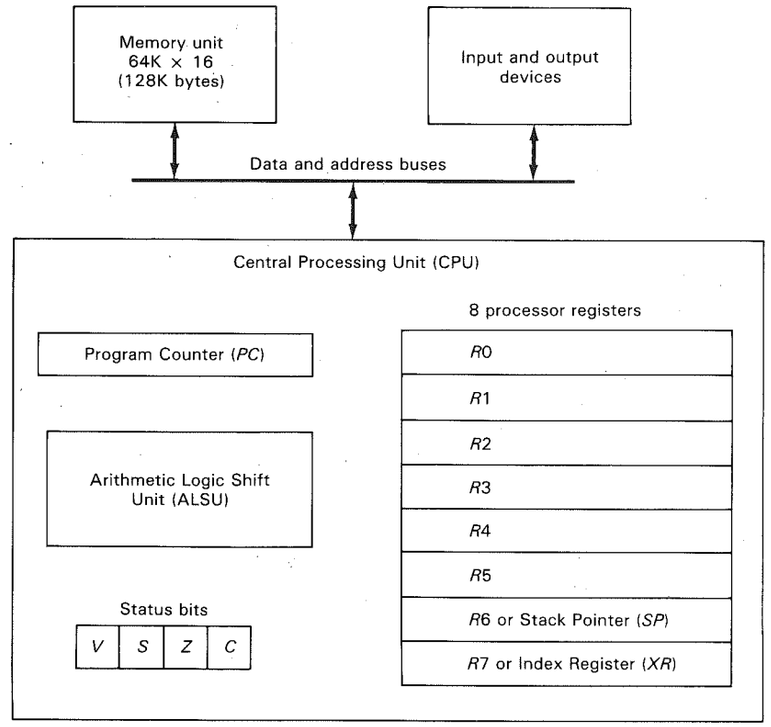


FIGURE 10-6
The Computer as Seen by the Programmer and the User

*R*0 through *R*5	Six general purpose registers
SP	Stack pointer
XR	Index register
PC	Program counter
SR	Source register
DR	Destination register
TR	Temporary register
ZR	Zero register (holds a constant 0)
NR	N = 16 register (holds a constant 16)

The last five registers are used by the control to store temporary results and necessary constants. The three registers that communicate with the address and data buses are

AR	Address register
DIR	Data input register
DOR	Data output register

The position of these registers is shown in Figure 10-5. The three registers associated with the control unit are

IR	Instruction register
CAR	Control address register
SBR	Subroutine register

CAR holds the address of the next microinstruction for control memory. *SBR* is used for storing the return address in conjunction with a subroutine call microinstruction. These two registers are presented in Section 10-9 together with the hardware of the control unit.

10-4 INSTRUCTION FORMATS

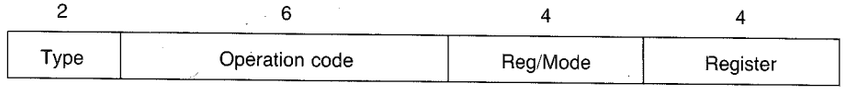
The sequence of microoperations that are incorporated in a CPU is determined from the instructions that are defined for the computer. The instructions are stored in memory as part of a program and are presented to the control unit in a binary form. Each binary coded instruction contains a number of fields that provide the pertinent information needed by the control to execute the instruction in the CPU. The binary coded form of the instruction is defined by its instruction format. Different computers have different instruction formats; this is one aspect that gives each computer its special hardware characteristic.

The instruction format chosen for the sample CPU is shown in Figure 10-7. An instruction code consists of 16 bits divided into four general fields. The first field contains 2 bits that specify the instruction format type. There are 4 instruction types as indicated in the figure. The second field contains 6 bits that specify the operation code of the instruction. With 6 bits we can formulate 64 distinct operations independent of the instruction type. If we include the 2-bit type field with the operation code, it will be possible to define up to 256 instructions. The next 2 fields of the instruction format depend on the instruction type. They specify either a processor register or the addressing mode of a memory word or, as in types 2 and 3, they always contain eight zeros.

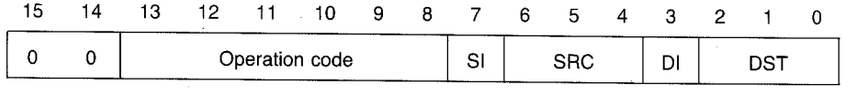
Type 0 instruction has two register fields in addition to the operation code field. It is used only in two or three operand type instructions such as move or add. It cannot be used for one operand type instruction such as increment or shift. The source register field and the destination register field each consists of three bits that specify one of the eight processor registers *R*0 through *R*7. Each register can be referred to directly or indirectly as specified by the indirect bit fields *SI* and *DI*. A direct reference means that the operand is in the register. An indirect register reference means that the register contains the address of the operand in memory. Note that the destination register is also the second source register in a three operand instruction. For example, an ADD operation adds the contents of the source register to the contents of the destination register and places the sum in the destination register. A two operand instruction, such as a MOVE, transfers the contents of the source register into the destination register.

Type 1 instruction format uses a register for one operand and a memory word for the second operand. It can be formulated for one or two operand instructions.

General format

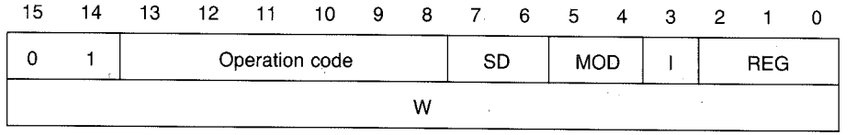


Type 0: Register-register. One-word instruction.



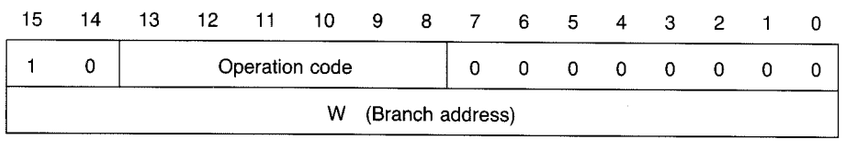
SRC —Source register
DST —Destination register
SI —Source indirect bit: =0 direct, =1 indirect
DI —Destination indirect bit: =0 direct, =1 indirect

Type 1: Memory-register. Two-word instruction (except for SD=11).



W —Memory word for address or immediate operand
REG —Register field
I —Register indirect bit: =0 direct, =1 indirect
MOD —Addressing mode for memory word W
  00 —Direct address mode
  01 —Immediate mode (only as a source)
  10 —Relative to PC mode
  11 —Index mode
SD —Source/destination and number of operands
  00 —2 operands. Source: memory word. Destination: register
  01 —2 operands. Source: register. Destination: memory word
  10 —1 operand specified by memory word W and MOD field
  11 —1 operand specified by REG and I fields (W not used)

Type 2: Branch type. Two-word instruction.



Type 3: Implied mode. One-word instruction.

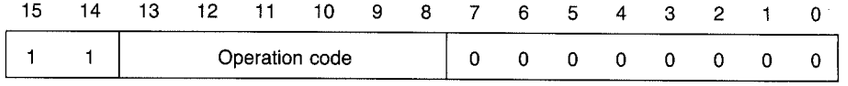


FIGURE 10-7
Instruction Formats for the Sample CPU

With one exception, type 1 instructions consist of two words. The second word is designated by *W* and follows the word of the instruction code. The register and indirect fields are similar to the corresponding fields in type 0. The *MOD* field specifies four addressing modes. In the direct address mode, the second word *W* contains the effective address defined as the address of the operand in memory. In the immediate mode, the value in *W* is the operand. In the relative mode, the value in *W* is added to the content of the program counter *PC* to obtain the effective address. In the index mode, the value in *W* is added to the content of the index register *XR* to obtain the effective address. The various addressing modes are discussed in more detail in Section 9-3.

The *SD* field specifies the source, the destination, as well as the number of operands in the instruction. When the first bit in the *SD* field is equal to 0, it specifies a two operand instruction, and when equal to 1, it specifies a one operand instruction. The second bit in the *SD* field specifies the source and destination operands. With $SD = 00$, the source operand is evaluated from the memory word *W* and the mode field *MOD*. The destination operand is determined from the *REG* and *I* fields. With $SD = 01$, the source and destination operand assignments are reversed. A one operand instruction, such as increment or shift, may have the operand either in a register or in memory. If in memory, the operand may be accessed through the addressing mode using *W* or the register indirect mode. Note that there are certain combinations in type 1 format that are not allowed. The immediate mode can be used only as a memory word source because it makes no sense to use it as a destination. Also when the *SD* field is 11, it specifies a one operand instruction residing in a register or in memory by means of a register indirect. This case does not use the memory word *W* and therefore, the instruction consists of only one word, and the *MOD* field is not used.

Type 2 format is dedicated exclusively to branch type instructions. The branch may be conditional or unconditional. The branch address *W* follows the instruction code. It is possible to modify this format to include a relative mode branch type instruction (see Problem 10-26). In this case, bits 0 through 7 are used for an offset number in the range of +127 through −128 to provide the relative address that must be added to *PC* when evaluating the effective address (instead of the second word *W*).

Type 3 format specifies an implied mode instruction such as set carry, return from interrupt, or no-operation. Bits 0 through 7 are not used and they are always equal to 0.

Computer Instructions

The complete design of the CPU must include a list of all the instructions for the computer together with the formats to which they belong. A complete list of instructions will be prohibitively long for our purpose here. However, a possible list of instructions that may be formulated can be found in Sections 9-5, 9-6, and 9-8. These three sections contain eight tables that list a set of instructions commonly found in computers.

In order to show the relationship between the instructions and their formats, we list some typical instructions in Table 10-2. Each instruction is assigned a symbolic name to be used by the programmer and a 6-bit binary code that is to be decoded by the control unit. The operation performed by the instruction determines the type of format to which it belongs. If an instruction has two or three operands, it can be formulated for both type 0 and type 1 formats. If it has one operand, it must be formulated using a type 1 format. Branch and call instructions are assigned to type 2 formats, and instructions that do not use an operand or an address field are assigned to format 3.

TABLE 10-2
Some Typical Instructions and Their Codes

Symbolic name	6-bit op-code	4-Digit hexadecimal code				Operation
		Type 0	Type 1	Type 2	Type 3	
ADD	001001	09xx	49xx	—	—	Addition
SUB	001010	0Axx	4Axx	—	—	Subtraction
MOVE	001011	0Bxx	4Bxx	—	—	Move data
INR	010011	—	53xx	—	—	Increment
DCR	010100	—	54xx	—	—	Decrement
PUSH	011110	—	5Exx	—	—	Push stack
BR	100001	—	—	A100	—	Branch
BRC	100010	—	—	A200	—	Branch on carry
CALL	100111	—	—	A700	—	Call subroutine
RET	110001	—	—	—	F100	Return from subroutine

Each instruction contains 16 bits which is equivalent to 4 hexadecimal digits. The 4-digit hexadecimal codes assigned to the ten instructions in Table 10-2 are controlled by the format type to which they belong. The 6-bit code of the instruction together with the 2-bit type constitute an 8-bit code which can be specified with two hexadecimal digits. For example the ADD instruction has a 6-bit operation code 001001. The first eight bits of the ADD type 0 instruction are 00001001 which is equivalent to hexadecimal 09. The corresponding ADD instruction for type 1 is binary 01001001, equivalent to hexadecimal 49. The two small x's in type 0 and 1 formats are digit values to be determined from the last eight bits of the instruction which specify the mode and the registers used.

In order to clarify the use of type 0 and 1 formats, it will be instructive to show by example all the possible combinations that can be formulated for a typical instruction. The four ADD instructions that can be formulated with type 0 format are listed in Table 10-3. It is assumed that the source register is *R*5 and the des-

TABLE 10-3
Example of Type 0 Instruction

Hex code	SI	DI	Op-code	Operands	Operation
0952	0	0	ADD	R5, R2	$R2 \leftarrow R2 + R5$
095A	0	1	ADD	R5, (R2)	$M[R2] \leftarrow M[R2] + R5$
09D2	1	0	ADD	(R5), R2	$R2 \leftarrow R2 + M[R5]$
09DA	1	1	ADD	(R5), (R2)	$M[R2] \leftarrow M[R2] + M[R5]$

TABLE 10-4
Example of Type 1 Instruction with Two Operands

Hex code	SD	MOD	Mode	Op-code	Operands	Operation
4902	00	00	Direct	ADD	W, R2	$R2 \leftarrow R2 + M[W]$
4912	00	01	Immediate	ADD	#W, R2	$R2 \leftarrow R2 + W$
4922	00	10	Relative	ADD	$W, R2	$R2 \leftarrow R2 + M[PC + W]$
4932	00	11	Index	ADD	W(XR), R2	$R2 \leftarrow R2 + M[XR + W]$
4942	01	00	Direct	ADD	R2, W	$M[W] \leftarrow M[W] + R2$
4962	01	10	Relative	ADD	R2, $W	$M[PC + W] \leftarrow M[PC + W] + R2$
4972	01	11	Index	ADD	R2, W(XR)	$M[XR + W] \leftarrow M[XR + W] + R2$

tination register is $R2$. There are four possible ways that the two indirect bits SI and DI can be combined, resulting in the four instructions listed in the table. The notation used for the addressing modes conforms with the notation introduced in Table 9-1.

Table 10-4 lists all the addressing modes that can be formulated for the type 1 ADD instruction. We assume a REG field 0010 corresponding to register $R2$. The third hexadecimal digit in the instruction code desginates the value in SD and MOD fields. When $SD = 00$, the destination operand is in register $R2$ and the MOD field specifies four ways to obtain the source operand from the second word W. When $SD = 01$, the source operand is in register $R2$ and the MOD field specifies three ways to obtain the destination operand. As mentioned previously, the immediate mode can be used only as a source operand and can be listed only with $SD = 00$.

Table 10-4 shows seven different ways to formulate a type 1 ADD instruction. This instruction can be used also with a register indirect mode by making the last hexadecimal digit equal to A instead of 2. This gives a total of 14 combinations that we can formulate a type 1 ADD instruction. Including the four instructions from Table 10-3, we have a total of 18 ways that we can formulate any 2 operand instruction.

Table 10-5 lists all the addressing modes that can be formulated for the type 1 INR (increment) instruction. The first two digits of the hexadecimal operation code for INR are 53 as listed in Table 10-2. The next digit of the instruction reflects the values of SD and MOD. The last digit is made arbitrarily equal to 0 when the register field is not used; it is equal to 2 when register $R2$ is used directly, and equal to hexadecimal A when register $R2$ is used indirectly. The last two entries in the table represent a one operand instruction that needs only a register; so the

TABLE 10-5
Example of Type 1 Instruction with One Operand

Hex code	SD	MOD	Mode	Op-code	Operand	Operation
5380	10	00	Direct	INR	W	$M[W] \leftarrow M[W] + 1$
53A0	10	10	Relative	INR	$W	$M[PC + W] \leftarrow M[PC + W] + 1$
53B0	10	11	Index	INR	W(XR)	$M[XR + W] \leftarrow M[XR + W] + 1$
53C2	11	XX	Register	INR	R2	$R2 \leftarrow R2 + 1$
53CA	11	XX	Reg indirect	INR	(R2)	$M[R2] \leftarrow M[R2] + 1$

MOD field is not used and the second word W is not needed. Table 10-5 shows five different ways that we can formulate a one operand type 1 instruction.

Type 2 and 3 instructions have only one possible combination. Some examples are

BR W	Branch to W
CALL W	Call subroutine starting from address W
RET	Return from subroutine

Type 2 instructions require an address field given by the memory word W. Type 3 instructions are one word instructions.

10-5 MICROINSTRUCTION FORMATS

The sequence of microoperations in the CPU is controlled by means of a microprogram stored in control memory. The microprogram consists of microinstructions that control the data paths and operations in the CPU. An example of a microprogram that controls the processor unit of a CPU was presented in Section 8-3. The microprogram to be presented here controls not only the microoperations in the processor unit but also all other data paths in the CPU. It is also capable of decoding computer instructions and obtaining the operands according to the specified addressing code.

The two microinstruction formats chosen for the control memory of the sample CPU are shown in Figure 10-8(a). Each format consists of 23 bits and is divided into six fields. Each field is identified with a symbolic name and the number of bits in each field is indicated on top of the symbolic name. The first field designated CS (Control Sequence) is used to distinguish between the two formats. When the 2-bit CS field is equal to 00, 01, or 10, it designates a microoperation type A format. When CS is equal to 11, it designates a jump or call type B format. The diagram also shows the convention for writing microinstructions in symbolic form. The actual symbols that can be assigned to each field are defined in detail in Tables 10-6 through 10-10.

Figure 10-8(b) lists all the binary codes and their corresponding symbolic names which are assigned to each field of the microinstruction. The five tables present a summary of all the components of the microinstruction and provide a reference for writing microprograms. When writing microprograms in symbolic form we will use special symbols for each binary value of the encoded fields. In this way, it will be possible to write microprograms with symbolic names instead of using the actual binary values for each field. Figures 10-8(a) and 10-8(b) and Table 10-6 are placed together for quick reference. The function of each field will be explained in greater detail in the discussion that follows.

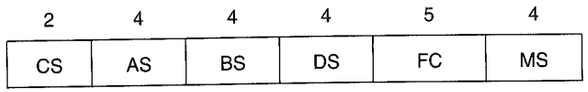
CS and BR Fields

The CS field is common to both microinstruction formats. It determines not only the format type but also the way the next microinstruction is selected from control memory. Table 10-6 lists the four ways that the 2-bit CS field makes this selection.

TABLE 10-6
Control Sequence and Branch Assignment

Binary code			
CS	*BR*	Symbol	Operation
00	—	NEXT	Use next address by incrementing *CAR*
01	—	RET	Return from subroutine
10	—	MAP	Map operation code into *CAR*
11	0	JUMP	Jump to *AD* if *ST* bit is satisfied
11	1	CALL	Call subroutine if *ST* bit is satisfied

Type A: Microoperation format.

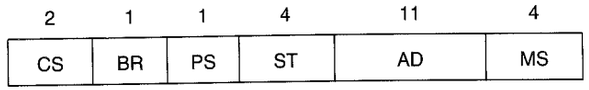


CS —Control sequence (must be 00, 01, or 10 for type A)
AS —Select register for A bus input to ALSU
BS —Select register for B bus input to ALSU
DS —Select register for D bus destination from ALSU
FC —Function control selection for the ALSU
MS —Miscellaneous microoperations.

Symbolic microinstruction convention:

CS	AS	BS	DS	FC	MS
Table 10-6	Table 10-7			Table 10-8	Table 10-9

Type B: Jump or Call format.



CS —Control sequence (must be 11 for type B format)
BR —Branch type: 0 for Jump, 1 for Call
PS —Polarity select: 0 for False, 1 for True
ST —Select status bit for test multiplexer
AD —11-bit address for control memory
MS —Miscellaneous microoperation (same as type A)

Symbolic microinstruction convention:

CS	BR	PS	ST	AD	MS
Table 10-6		T or F	Table 10-10	Symbolic address	Table 10-9

FIGURE 10-8(a)
Microinstruction Formats for Control Memory

Selection of ALSU Buses

Binary code	Symbols AS	BS	DS	Register selected
0000	*R*0	*R*0	*R*0	Processor Register 0
0001	*R*1	*R*1	*R*1	Processor Register 1
0010	*R*2	*R*2	*R*2	Processor Register 2
0011	*R*3	*R*3	*R*3	Processor Register 3
0100	*R*4	*R*4	*R*4	Processor Register 4
0101	*R*5	*R*5	*R*5	Processor Register 5
0110	*SP*	*SP*	*SP*	Stack Pointer
0111	*XR*	*XR*	*XR*	Index Register
1000	*PC*	*PC*	*PC*	Program Counter
1001	*SR*	*SR*	*SR*	Source Register
1010	*DR*	*DR*	*DR*	Destination Register
1011	*TR*	*TR*	*TR*	Temporary register
1100	*ZR*	*ZR*	*AR*	See Table 10-7
1101	*DIR*	*DIR*	*DOR*	See Table 10-7
1110	*R*(*D*)	*R*(*D*)	*R*(*D*)	Register selected by *IR*(0–2)
1111	*R*(*S*)	*NR*	NONE	See Table 10-7

ALSU Function Control

FC Code	Symbol	Microoperation
00000	TSF	$F = A$
00001	INC	$F = A + 1$
00010	ADD	$F = A + B$
00011	ADP	$F = A + B + 1$
00100	SBM	$F = A - B - 1$
00101	SUB	$F = A - B$
00110	DEC	$F = A - 1$
01000	AND	$F = A \wedge B$
01010	OR	$F = A \vee B$
01100	XOR	$F = A \oplus B$
01110	COM	$F = \overline{A}$
10000	SHR	F = Logical shift right A
10010	ROR	F = Rotate right A
10100	RRC	F = Rotate right A with carry
10110	ASR	F = Arithmetic shift right A
11000	SHL	F = Logical shift left A
11010	ROL	F = Rotate left A
11100	RLC	F = Rotate left A with carry
11110	ASL	F = Arithmetic shift left A

Status Bits for Test Multiplexer

ST code	Symbol	Name
0000	U	Always = 1 for unconditional transfer
0001	*C*	Carry status bit
0010	*Z*	Zero status bit
0011	*IR*3	Instruction register bit 3
0100	*IR*4	Instruction register bit 4
0101	*IR*5	Instruction register bit 5
0110	*IR*6	Instruction register bit 6
0111	*IR*7	Instruction register bit 7
1000	*S*	Sign status bit
1001	*V*	Overflow status bit
1010	EIF	Enable Interrupt flip-flop
1011	INTS	Interrupt input signal

Miscellaneous Microoperations

MS code	Symbol	Operation
0000	—	No operation
0001	READ	Enable READ flip-flop
0010	WRITE	Enable WRITE flip-flop
0011	LIR	Load *IR* from *DIR*
0100	SCF	Set carry status flip-flop
0101	RCF	Reset carry status flip-flop
0110	EST	Enable to update the four status bits
0111	ECB	Enable to update the *C* status bit
1000	ESZ	Enable to update the *S* and *Z* status bits
1001	SEIF	Set enable interrupt flip-flop
1010	REIF	Reset enable interrupt flip-flop
1011	IACK	Generate interrupt acknowledge

FIGURE 10-8(b)
Symbols and Binary Codes for Microinstruction Fields

The 1-bit *BR* (branch) field comes into play when *CS* = 11. It determines the type of branch chosen for the type B microinstruction format. For *CS* = 11 and *BR* = 0 we use the symbol JUMP to designate a conditional branch dependent on the value of the selected status bit in the *ST* field. The address for the jump is specified in the 11-bit *AD* field. When *BR* = 1, the microinstruction specifies a conditional CALL subroutine. (The jump and call microinstructions are similar to the branch and call instructions discussed in Section 9-8). The symbol NEXT is used when the microprogram chooses the next microinstruction in sequence.The symbol RET designates a return from subroutine condition.

When *CS* = 10, the control unit performs a mapping from the bits of the operation code into an address for control memory. The MAP condition provides a decoding of the binary instruction. For example, suppose that the type 0 ADD instruction has an 8-bit operation 00001001 (see Table 10-3). The binary instruction is in the instruction register *IR* after it is fetched from memory. Suppose that we place the microprogram routine that executes this instruction in control memory starting from address 23. The hardware decoding requires that a mapping or transformation be carried out from the binary operation code 00001001 into an 11-bit address 00000010111 for control memory. This mapping must be done for each instruction in order to obtain the corresponding address of its microprogram routine. The binary mapping can be specified in a table that gives the relationships between each 8-bit operation code and its corresponding 11-bit address for control memory. The truth table can be implemented with hardware using a programmable logic device (see Section 6-6).

Figure 10-9 shows a block diagram that may clarify the way that the *CS* and *BR* fields select the address of the next microinstruction. The control memory consists of 23 bits, but here we show only the 2-bit *CS* field, the 1-bit *BR* field, and the 11-bit *AD* (address) field. The control address register *CAR* supplies the address

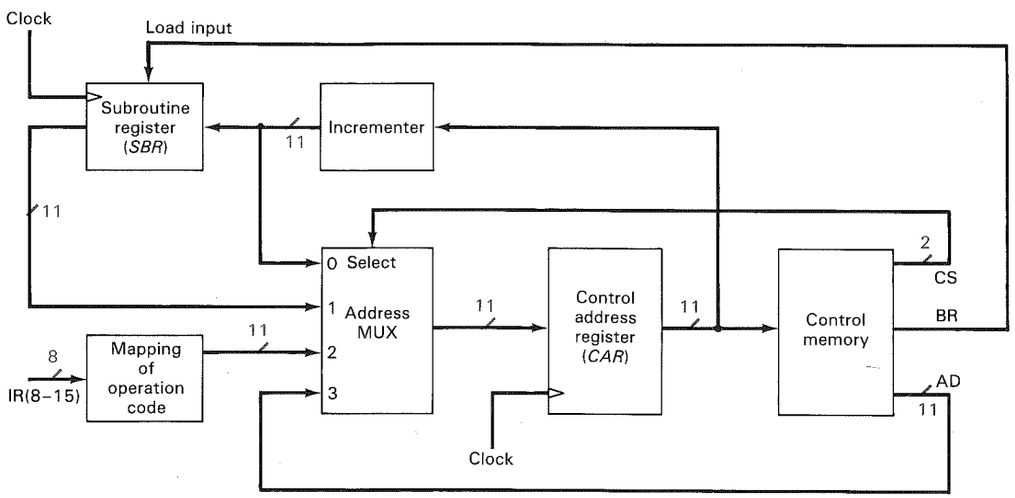


FIGURE 10-9
Selection of Next Address for Control Memory Using the CS and BR Fields

for control memory. The incrementer is a combinational circuit that adds one to the present value of *CAR*. The subroutine register *SBR* stores the return address when the microprogram calls a subroutine. For simplicity, we use here a single register for storing the return address instead of using a register stack.

While the present microinstruction is at the output terminals of the control memory, an address multiplexer selects the address for the next microinstruction. This address is loaded into *CAR* with the next clock pulse transition. The *CS* field is applied to the selection inputs of the address multiplexer. If *CS* = 00, *CAR* is incremented by one to proceed with the next microinstruction in sequence. A call subroutine microinstruction has *CS* = 11 and *BR* = 1. This causes the return address *CAR* + 1 to be loaded into the subroutine register *SBR* and the address in the *AD* field to be loaded into *CAR*. The return from subroutine condition transfers the content of *SBR* into *CAR*. An external address is transferred by mapping the operation code bits of the instruction in the instruction register *IR*(8-15) into an 11-bit address for *CAR*. A jump microinstruction branches to the address in *AD* without loading the subroutine register. The jump and call microinstructions shown in Figure 10-9 are unconditional. The conditional jump and call depend on the values of the *ST* and *PS* fields, as will be explained subsequently.

Figure 10-9 is a simplified version of the microprogram sequencer that is part of the control hardware. A more detailed block diagram of the sequencer can be found in Figure 10-18.

AS, BS and DS Fields

As mentioned previously, when the *CS* field is equal to 00, 01, or 10, the other fields of the microinstruction are taken from type A format. Fields *AS, BS,* and *DS* specify source and destination registers for the ALSU buses, and the *FC* field specifies the ALSU operation. The binary code and corresponding symbols for the ALSU buses are shown in Table 10-7. The *AS* field selects a source register for the *A* input of the ALSU. *BS* selects a second source register for input *B* of the ALSU, and *DS* selects a destination register. There are 17 registers in the processor unit that are controlled by the microprogram. The first eight registers *R*0 through *R*7 are general purpose registers, with *R*6 acting as the stack pointer *SP* and *R*7 as the index register *XR*. The other nine registers are the program counter *PC*, three control registers labeled *SR, DR,* and *TR,* two registers *ZR* and *NR* that hold the binary constants zero and 16, and three registers, *AR, DIR,* and *DOR,* that communicate with the memory.

Certain restrictions apply to some registers. *ZR* and *NR* provide constants for the ALSU and can operate only as a source and never as a destination. Therefore, they are not included in the destination select field *DS*. The three registers that communicate with the memory are also restricted in their use. The data input register *DIR* can have only a source operand and is included in the *AS* and *BS* fields but not in the *DS* field. The address register *AR* and the data output register *DOR* can function only as destination registers since they receive the address and data from the processor. These two registers are assigned to the *DS* field only.

In addition to a direct selection of the 17 registers, the microinstruction can select registers that are specified directly in the computer instruction. Remember

TABLE 10-7
Selection of ALSU Buses

Binary code	Symbols AS	BS	DS	Register selected
0000	*R*0	*R*0	*R*0	Processor register 0
0001	*R*1	*R*1	*R*1	Processor register 1
0010	*R*2	*R*2	*R*2	Processor register 2
0011	*R*3	*R*3	*R*3	Processor register 3
0100	*R*4	*R*4	*R*4	Processor register 4
0101	*R*5	*R*5	*R*5	Processor register 5
0110	*SP*	*SP*	*SP*	Stack pointer
0111	*XR*	*XR*	*XR*	Index register
1000	*PC*	*PC*	*PC*	Program counter
1001	*SR*	*SR*	*SR*	Source register
1010	*DR*	*DR*	*DR*	Destination register
1011	*TR*	*TR*	*TR*	Temporary register
1100	*ZR*	*ZR*	*AR*	See note below
1101	*DIR*	*DIR*	*DOR*	See note below
1110	*R*(*D*)	*R*(*D*)	*R*(*D*)	Register selected by *IR*(0-2)
1111	*R*(*S*)	*NR*	NONE	See note below

ZR—Holds 0's in all 16 bits and cannot be changed.
NR—Holds a constant (N = 16) and cannot be changed
R(*S*)—Register selected by the binary code in the source field in *IR*(4-6) as defined in Figure 10-7.
NONE—When *DS* = 1111 none of the registers is selected to receive the output from the ALSU.
AR, *DIR*, and *DOR* are registers that interface with the address and data buses (see Figure 10-5).

that registers *R*0 through *R*7 in the processor unit are available as general purpose registers and can be chosen by means of program instructions. Looking back at the instruction formats in Figure 10-7, we note that these registers are specified as a source in bits 4-6 and as destination in bits 0-2. These bits are part of the binary instruction that resides in the instruction register *IR* after the instruction is read from memory. When we assign to the *AS* field binary 1111, the hardware of the control unit will select a register for the A bus from the decoded value of the 3-bit code in *IR*(4-6). When writing symbolic microprograms, we designate this condition by the symbol *R(S)* meaning that the register chosen is given by the 3-bit code in the source field of the instruction. For example, if the 3-bit code in *IR*(4-6) is equal to 011 and the *AS* field of the microinstruction is 1111 (the binary value for the symbol *R(S)*), the control will provide a path from register *R*3 in the processor into the A bus of the ALSU.

Similarly, we use the symbol *R(D)* to designate a register chosen by the destination field of the instruction in *IR*(0-2). Note that *R(D)* can be chosen for all three fields, but *R*(*S*) is only available with the *AS* field.

When the *DS* field is specified with the symbol NONE, it is substituted with the binary value 1111. When this condition exists, none of the registers are selected to receive the data from the ALSU. This is equivalent to a "no operation" in the processor unit.

FC Field

Table 10-8 lists the microoperations that can be specified with the ALSU function control field *FC*. These operations are the same as the ones listed in Table 10-1. Each operation is assigned a symbolic name to be used when writing microprograms. The last bit in the 5-bit code for *FC* is the value of the input carry C_{in}. The input carry controls the arithmetic operations but has no effect on the logic and shift operations. The last bit of the binary code is chosen to be 0 for the logic and shift operations.

Table 10-8 gives a list of the status bits that have significance for each operation. All four status bits are important after an arithmetic operation of add and subtract. The other arithmetic and logic operations need only the sign and zero bits because they do not produce a carry or overflow. (The increment operation produces a carry at the same time that the number goes back to zero.) All shift operations affect the carry status bit. The arithmetic shift left operation may produce an overflow if the sign bit of the number changes after the shift. The status bits are affected after an arithmetic or logic operation only if they are enabled by a special code in the MS field. However, the carry bit is always updated with a shift operation as indicated in Figure 10-3. It will be assumed that the overflow bit will be automatically updated with an arithmetic shift left operation.

MS Field

Table 10-9 gives the list of the microoperations that can be generated with the 4-bit *MS* field. Note that this field is available in both microinstruction formats. When used with type A format, the *MS* field specifies a microoperation that can be generated simultaneously with the microoperation specified for the ALSU.

TABLE 10-8
ALSU Function Control

FC code	Symbol	Microoperation	Name	Status bits
00000	TSF	$F = A$	Transfer A	*S Z*
00001	INC	$F = A + 1$	Increment A	*S Z*
00010	ADD	$F = A + B$	Add	*S Z C V*
00011	ADP	$F = A + B + 1$	Add plus 1	*S Z C V*
00100	SBM	$F = A - B - 1$	Subtract minus 1	*S Z C V*
00101	SUB	$F = A - B$	Subtract	*S Z C V*
00110	DEC	$F = A - 1$	Decrement A	*S Z*
01000	AND	$F = A \wedge B$	AND	*S Z*
01010	OR	$F = A \vee B$	OR	*S Z*
01100	XOR	$F = A \oplus B$	Exclusive-OR	*S Z*
01110	COM	$F = \overline{A}$	Complement A	*S Z*
10000	SHR	F = Logical shift right A		*C*
10010	ROR	F = Rotate right A		*C*
10100	RRC	F = Rotate right A with carry		*C*
10110	ASR	F = Arithmetic shift right A		*C*
11000	SHL	F = Logical shift left A		*C*
11010	ROL	F = Rotate left A		*C*
11100	RLC	F = Rotate left A with carry		*C*
11110	ASL	F = Arithmetic shift left A		*C V*

TABLE 10-9
Miscellaneous Microoperations

MS code	Symbol	Operation
0000	—	No operation
0001	READ	Enable READ flip-flop
0010	WRITE	Enable WRITE flip-flop
0011	LIR	Load *IR* from *DIR*
0100	SCF	Set carry status flip-flop
0101	RCF	Reset carry status flip-flop
0110	EST	Enable to update the four status bits
0111	ECB	Enable to update the *C* status bit
1000	ESZ	Enable to update the *S* and *Z* status bits
1001	SEIF	Set enable interrupt flip-flop
1010	REIF	Reset enable interrupt flip-flop
1011	IACK	Generate interrupt acknowledge (Figure 10-1)

The operation symbolized by the READ symbol sets a flip-flop to 1. The output of this flip-flop provides the read signal for the memory unit. Thus, when the *MS* field in the microinstruction is equal to 0001, the next clock pulse sets the READ flip-flop and a read cycle is initiated. If the next microinstruction does not have 0001 in the *MS* field, the READ flip-flop is reset to 0 and the memory read cycle terminates. Similarly, the WRITE operation produces a memory write cycle when the corresponding WRITE flip-flop is set and then reset.

The operation symbolized by LIR is needed during the fetch phase. The instruction read from memory into the data input register *DIR* is transferred to the instruction register *IR* to be decoded by the control. The symbols SCF and RCF cause the setting or resetting of the carry status bit. This is useful when it is required to initialize the carry prior to a shift.

The *MS* field includes special conditions that enable the status bits in the processor to update them only when necessary. A microinstruction that specifies an arithmetic operation in the ALSU can update all the status bits by including the EST code in the *MS* field. If no status bits have to be updated, we must ensure that none of the three *MS* codes equivalent to EST, ECB, or ESZ is used. ECB updates only the carry bit and ESZ updates the sign and zero bits. The status bit conditions listed in Table 10-8 are given only as a suggestion. It is up to the designer who writes the microprogram to decide which status bits are to be updated after each instruction.

Other operations that are specified with the *MS* field include the setting and resetting of the EIF (enable interrupt flip-flop) and the generation of an interrupt acknowledge signal to an external device.

ST and *PS* Fields

The jump or call microinstruction in type B format is used for branching to a microinstruction out of sequence depending on a status bit condition. The status bits available for testing are listed in Table 10-10. The symbol *U* is used for the

TABLE 10-10
Status Bits For Test Multiplexer

ST code	Symbol	Name
0000	U	Always = 1 for unconditional transfer
0001	*C*	Carry status bit
0010	*Z*	Zero status bit
0011	*IR*3	Instruction register bit 3
0100	*IR*4	Instruction register bit 4
0101	*IR*5	Instruction register bit 5
0110	*IR*6	Instruction register bit 6
0111	*IR*7	Instruction register bit 7
1000	*S*	Sign status bit
1001	*V*	Overflow status bit
1010	EIF	Enable interrupt flip-flop
1011	INTS	Interrupt input signal

ST code 0000. This code, when applied to the select inputs of a multiplexer, makes the output of the multiplexer always equal to 1. The symbols *C*, *Z*, *S*, and *V* are for testing the status bits in the processor. Five bits from the instruction register *IR* are available for inspection by the microprogram. These bits specify the addressing modes of the instruction. Two status bits associated with the interrupt cycle are available for detection. The output of the EIF flip-flop indicates whether interrupts are allowed or prohibited. The interrupt input signal from external devices is checked by the control to find out if any interrupts are pending (see Figure 9-8).

The 1-bit field *PS* is associated with the *ST* field to determine the polarity of the tested bit. We can either test the true or false value of the status bit by letting *PS* be equal to 1 (symbolized by T) or to 0 (symbolized by F), respectively. If the tested bit polarity is satisfied, control branches to the address specified by the 11-bit *AD* field; otherwise, control continues with the next microinstruction in sequence. For example, a jump on carry, if true, will branch to *AD* if $C = 1$ but will continue with the next microinstruction if $C = 0$. A call on carry, if false, will branch to the subroutine at the address given by the *AD* field if $C = 0$ but will continue with next microinstruction if $C = 1$. An unconditional branch is achieved by specifying T for the *PS* field and *U* for the *ST* field. This condition will always be true because the status bit chosen with *ST* code of 0000 is always equal to 1.

10-6 EXAMPLES OF MICROINSTRUCTIONS

Before starting to write the microprogram for the CPU, it will be instructive to give a few examples of typical microinstructions. Table 10-11 lists nine examples of microinstructions in symbolic form. The symbols on top of the table are the six fields of type A format and the six fields of type B format. An entry in the label column is terminated with a colon and represents a symbolic address. The comments column explains the function of the microinstruction with register transfer statements and in words when appropriate.

TABLE 10-11
Examples of Symbolic Microinstructions

Label	CS	AS CSBR	BS PS	DS ST	FC AD	MS MS	Comments
1.	NEXT	SR	DR	DOR	ADD	EST	$DOR \leftarrow SR + DR$, enable status bits
2.	NEXT	DIR	XR	AR	ADD	—	$AR \leftarrow DIR + XR$
3. START:	NEXT	R(S)	—	SR	TSF	—	$SR \leftarrow R(S)$
4.	NEXT	PC	—	AR	TSF	READ	$AR \leftarrow PC$, set READ flip-flop
5.	RET	DIR	—	SR	TSF	SCF	$SR \leftarrow DIR$, set carry, return from subroutine
6.	MAP	—	—	NONE	—	—	Map operation code
7.		CALL	T	IR3	DEST	—	Call subroutine DEST if $IR3 = 1$
8.		JUMP	T	U	START	WRITE	Set WRITE flip-flop, jump to START
9. DEST:		JUMP	F	EIF	FETCH	—	Jump to FETCH if EIF = 0

The first six examples in the table have the symbols NEXT, RET, or MAP in the *CS* field. These are the symbols used with the type A microinstruction format. The CALL and JUMP symbols are used with the type B format. They are placed under the common symbol *CSBR* since they belong to both the *CS* and *BR* fields. Thus, a microinstruction is recognized as being of type A format if it has a symbolic entry under the *CS* field. The other symbols of the microinstruction belong to fields *AS*, *BS*, *DS*, *FC*, and *MS*. The type B microinstruction is recognized from the symbol JUMP or CALL under the combined *CSBR* fields. The other fields are *PS*, *ST*, *AD*, and *MS*.

The first microinstruction in the table specifies an ADD microoperation in the ALSU and an EST (enable status bits) condition in the *MS* field. This updates all four status bits *C*, *Z*, *S*, and *V* according to the sum produced in the output of the ALSU. The second microinstruction performs an addition in the ALSU but the status bits are not affected because the *MS* field is marked with a dash signifying a no operation condition.

The third microinstruction has the symbol START in the label column. This is a symbolic address that must also be present in the address field of a jump or call microinstruction as shown in example 8. The microinstruction transfers the content of *R(S)* (which is the processor register whose binary code is given by the source field of the instruction) into the control source register *SR*. The *BS* field is not used in this case and is marked with a dash. This is because the TSF (transfer) operation acts with the A bus data of the ALSU and neglects any value in the B bus.

The fourth example in the table shows a microinstruction that initiates a memory read cycle. This is done by transferring the address into the address register *AR* and setting the READ flip-flop to 1. The word read from memory at the address given by *AR* is assumed to be available in the data input register *DIR* for use with the next microinstruction in sequence.

The fifth and sixth microinstructions in the table show how to use the RET and MAP conditions for type A format. The other fields of the microinstruction may specify a microoperation for the ALSU or the *MS* field. If the *DS* field has the symbol NONE, then no operation is performed in the processor unit. The *MS* field performs no operation if it is marked with a dash.

The JUMP and CALL symbols are used with type B format. Example seven shows a call subroutine microinstruction that branches to the address symbolized by DEST if $IR3 = 1$. Example eight is an unconditional jump microinstruction that also sets the WRITE flip-flop to initiate a memory write cycle. Example nine shows a jump microinstruction that branches to the address symbolized by FETCH if the selected status bit is false, that is if EIF = 0. Note that the nine examples in Table 10-11 do not constitute a meaningful microprogram. Each microinstruction is independent of all others and is shown here only as an example for writing symbolic microinstructions.

Binary Microinstructions

Symbolic microinstructions are convenient when writing microprograms in a way that people can read and understand. But this is not the way that the microprogram is stored in control memory. The symbolic microinstructions must be translated to binary, either by means of a computer program called a microassembler or by the user if the microprogram is simple. The conversion from a symbolic microinstruction to its binary equivalent can be done using the encoding information listed in Figure 10-8(b).

Consider for example the first microinstruction in Table 10-11. It can be converted to binary as follows:

CS	AS	BS	DS	FC	MS
NEXT	SR	DR	DOR	ADD	EST
00	1001	1010	1101	00010	0110

The first line lists the symbolic names of the six fields of format A. The next line gives the corresponding symbols used in the microinstruction. From Figure 10-8(b) we extract the binary equivalent of each field to obtain the 23 bits of the binary microinstruction.

The microinstruction in example 6 in Table 10-11 is converted to binary in a similar fashion.

CS	AS	BS	DS	FC	MS
MAP	—	—	NONE	—	—
10	0000	0000	1111	00000	0000

The dashes in the *AS*, *BS*, and *FC* fields are assigned 0's, although any other binary value can be used since the information in these fields has no significance when none of the registers are affected. The dash in the *MS* field must be converted into 0000 to indicate a "no operation."

The microoperation in example 8 of Table 10-11 has a type B format. The symbolic address START must be assigned an 11-bit binary address value. Assum-

ing that START is equivalent to address 3, we form the binary microinstruction as follows:

CSBR	PS	ST	AD	MS
JUMP	T	U	START	WRITE
110	1	0000	00000000011	0010

The T in the *PS* field converts to 1 and the START symbol converts to an 11-bit address equivalent to binary 3. The total number of bits in the microinstruction is 23.

10-7 MICROPROGRAM FOR COMPUTER CYCLE

We are now ready to start writing the microprogram for the CPU. This will be done in parts by writing the microprogram for the fetch phase, the interrupt phase, and for the execution of some typical instructions. Microprograms for some other instructions will be left as exercises. The basic computer cycle that fetches and executes each instruction is explained in Section 9-3.

The first microprogram example to be developed will be introduced in three different forms. First, the general procedure will be shown in flowchart form using register transfer notation. From this we will write the microprogram in symbolic form showing the values for each field of the microinstruction. The last step is to translate the symbolic microprogram to binary. The ultimate form of a microprogram is always the binary content that must be stored in control memory. For convenience, people use symbolic representation to formulate the logical sequence of microinstructions and then use a computer program to translate the microprogram to binary.

It is assumed that a user program is stored in main memory starting at some known address. When the system is reset, the program counter *PC* is loaded with the beginning address of this program and the control memory address register *CAR* is cleared to 0. The first microinstruction of the fetch phase will be stored at address 0 in control memory.

The flowchart of Figure 10-10 shows the three phases of the computer cycle. The first box lists the microinstructions for fetching an instruction from memory and placing it in the instruction register *IR*. The address from *PC* is transferred to *AR* and the signal from the READ flip-flop causes the memory to place the instruction into the data bus from which it is transferred into *DIR*. The instruction in *DIR* is then transferred into *IR*, and *PC* is incremented by one. The microinstructions for the execute phase depend on the operation code found in *IR*(8-15). The 8-bit operation code is mapped into an address for *CAR*. This address points to the beginning of the microroutine that executes the instruction whose binary code is presently available in *IR*.

After the instruction is executed, the microprogram checks for a possible external interrupt. The computer cycle then repeats the fetch-execute-interrupt phases continuously. We will now demonstrate one complete computer cycle by choosing a specific instruction and including the microprogram for the interrupt phase.

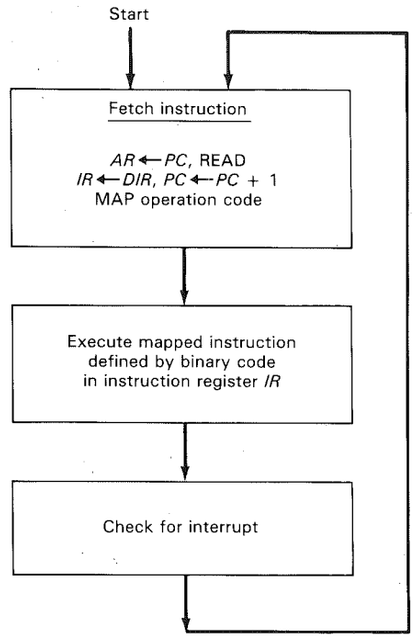


FIGURE 10-10
Computer Cycle

Type 0 ADD Instruction

Assume that the instruction transferred to *IR* at the end of the fetch phase is a type 0 ADD instruction. In order to follow the microprogram for this instruction it is necessary to review the type 0 format from Figure 10-7 and the four addressing modes listed in Table 10-3. The instruction code format is repeated in Figure 10-11 to provide a more convenient reference. The operation code in IR(8-15) is used to map into the 11-bit control memory address where the microprogram routine for this instruction resides. The selected source register is identified by the symbol $R(S)$ and the selected destination register is symbolized by $R(D)$. Bit 7 in *IR* is the source indirect bit and bit 3 is the destination indirect bit. When both these bits are equal to 0, the instruction specifies the microoperation

$$R(D) \leftarrow R(D) + R(S)$$

If an indirect bit is equal to 1, the operand must be retrieved from memory.

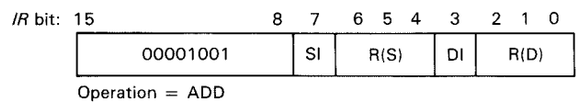


FIGURE 10-11
Type 0 ADD Instruction Format

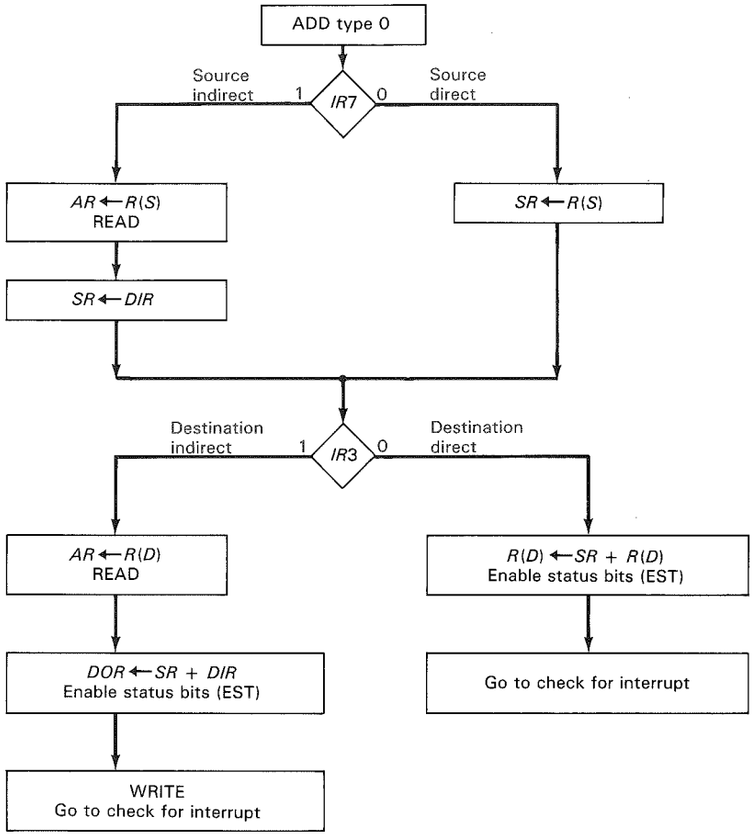


FIGURE 10-12
Microprogram Flowchart for Type 0 ADD Instruction

The sequence of microinstructions that execute the type 0 ADD instruction is shown in flowchart form in Figure 10-12. If $IR7 = 0$, the source operand is temporarily stored in control source register *SR*. If $IR7 = 1$, then the register specified by *R(S)* contains the address of the operand in memory. This address is transferred to the address register *AR* and a read cycle is initiated. The memory places the operand in *DIR* from which it is then transferred to *SR*.

The second part of the flowchart checks the bit in *IR*3. If it is equal to 0, then *SR* is added to *R*(*D*) and the sum transferred back to *R*(*D*). If $IR3 = 1$, *R*(*D*) has the address of the destination operand. The operand is first read from memory into *DIR*. Its contents are added to the contents of *SR* and the sum is transferred into the data output register *DOR*. A memory write cycle stores the contents of *DOR* into the memory word given by the address in *AR*. Remember that *AR* has the address that was originally in *R*(*D*) which is the address of memory where the sum must be placed. After the instruction is executed, control branches to the interrupt phase.

Interrupt Phase

The flowchart for the interrupt microprogram is shown in Figure 10-13. EIF (enable interrupt flip-flop) is an internal flip-flop that can be set or reset by program instructions or by the hardware. Only when this flip-flop is set to 1 will an interrupt signal be acknowledged by the CPU. The interrupt signal is detected from the presence of the status bit symbolized by INTS. If both EIF and INTS are equal to 1, the microprogram goes through an interrupt routine; otherwise it goes back to fetch the next instruction. The external interrupt procedure is explained in Section 9-9 in conjunction with Figure 9-8.

The interrupt routine consists of pushing the return address from *PC* onto the stack and resetting EIF to 0 to disable further interrupts (EIF can be set later by the program). The interrupt acknowledge *INTACK* signal is sent to the external device from which the device responds by placing an interrupt vector address on the data bus. The content of the data bus in *DIR* is transferred into *PC*. Control then goes to the fetch phase to start the interrupt service program at the vector address.

It was mentioned in Section 9-9 that the current contents of the status bits must be stored in the stack after an interrupt request. This can be done with an instruction in the interrupt service program (see Problem 10-27).

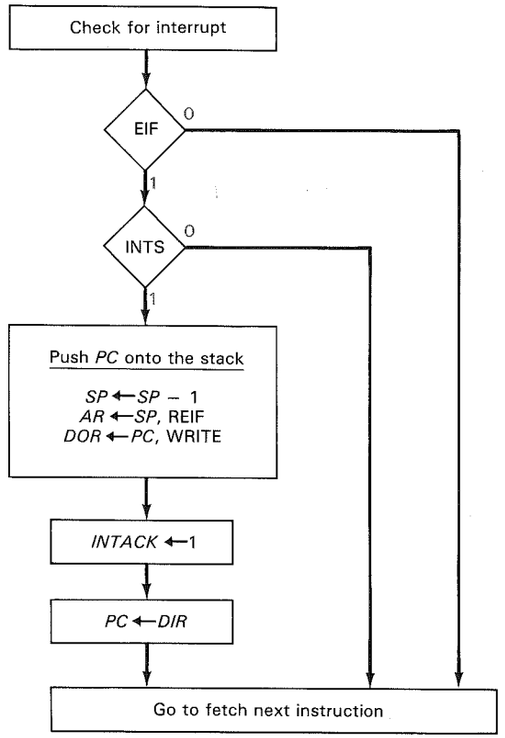


FIGURE 10-13
Flowchart for Interrupt Microprogram

TABLE 10-12
Symbolic Microprogram For One Computer Cycle

Label	CS	AS CSBR	BS PS	DS ST	FC AD	MS MS	Comments
Fetch instruction:							
FETCH:	NEXT	PC	—	AR	TSF	READ	$AR \leftarrow PC$, Read
	NEXT	PC	—	PC	INC	LIR	$PC \leftarrow PC + 1$, $IR \leftarrow DIR$
	MAP	—	—	NONE	—	—	Map operation code
Type 0 ADD instruction:							
ADDT0:	NEXT	R(S)	—	SR	TSF	—	$SR \leftarrow R(S)$
		CALL	T	IR7	REGSRC	—	Call REGSRC if $IR7=1$
		JUMP	T	IR3	DESADD	—	Go to DESADD if $IR3=1$
	NEXT	SR	R(D)	R(D)	ADD	EST	$R(D) \leftarrow SR + R(D)$, EST
		JUMP	T	U	INTRPT	—	Go to INTRPT
DESADD:	NEXT	R(D)	—	AR	TSF	READ	$AR \leftarrow R(D)$, Read
	NEXT	SR	DIR	DOR	ADD	EST	$DOR \leftarrow SR + DIR$, EST
		JUMP	T	U	INTRPT	WRITE	Write, go to INTRPT
REGSRC:	NEXT	R(S)	—	AR	TSF	READ	$AR \leftarrow R(S)$, Read
	RET	DIR	—	SR	TSF	—	$SR \leftarrow DIR$, Return
Check for interrupt:							
INTRPT:		JUMP	F	EIF	FETCH	—	Go to FETCH if EIF=0
		JUMP	F	INTS	FETCH	—	Go to FETCH if INTS=0
	NEXT	SP	—	SP	DEC	—	$SP \leftarrow SP - 1$
	NEXT	SP	—	AR	TSF	REIF	$AR \leftarrow SP$, reset EIF
	NEXT	PC	—	DOR	TSF	WRITE	$DOR \leftarrow PC$, Write
	NEXT	—	—	NONE	—	IACK	Interrupt acknowledge
	NEXT	DIR	—	PC	TSF	—	$PC \leftarrow DIR$
		JUMP	T	U	FETCH	—	Go to FETCH

Symbolic Microprogram

The symbolic microprogram for one computer cycle is listed in Table 10-12. The FETCH and INTRPT routines are common to all cycles, but the instruction that is executed depends on the operation code that is mapped into the control memory. Here we assume a mapping of the type 0 ADD instruction. The microprogram uses a subroutine symbolized by REGSRC to show how a subroutine is formulated and terminated with a RET control sequence. A dash in a column indicates that the field is not used.

The microprogram for the fetch phase follows the sequence dictated by the flowcharts of Figure 10-10 and consists of three microoperations. The type 0 ADD microroutine ADDT0 follows the flowchart of Figure 10-12. The source operand

is transferred into *SR* in the first microinstruction assuming that $IR7 = 0$. If $IR7 = 1$, the microprogram goes to subroutine REGSRC to read the operand from memory and place it in *SR*. *IR*3 is then checked in the third microinstruction, and if equal to 1, the microprogram jumps to DESADD. The two operands are added and the four status bits *C, Z, S,* and *V* are updated according to the result obtain in the sum. The microprogram then jumps to the interrupt routine.

The INTRPT routine follows the sequence of operations shown in the flowchart of Figure 10-13. If either EIF or INTS is equal to 0, the microprogram jumps back to FETCH to read and execute the next instruction in the program. If both are equal to 1, the microprogram stores the return address from *PC* onto the stack and places the vector address into *PC*. The computer cycle terminates by jumping back to the fetch phase.

Binary Microprogram

The binary form of the microprogram is listed in Table 10-13. It is assumed that the fetch phase starts from address 0 and the interrupt phase starts from address 64. The microroutine for each instruction will reside in a different part of control memory as specified by the address that is mapped into *CAR*. Here we assume that the routine for the type 0 ADD instruction starts from address 16. The binary codes for the various fields are obtained from Figure 10-8(b). A dash in the symbolic microprogram is converted to all 0's. The binary values of each symbolic address

TABLE 10-13
Binary Microprogram

Address	CS CS	AS BR	PS	BS ST	DS FC AD	MS MS	
0 (FETCH)	00	1000		0000	1100 00000	0001	
1	00	1000		0000	1000 00001	0011	
2	10	0000		0000	1111 00000	0000	
16 (ADDT0)	00	1111		0000	1001 00000	0000	
17	11	1	1	0111	00000011000	0000	(REGSRC = 24)
18	11	0	1	0011	00000010101	0000	(DESADD = 21)
19	00	1001		1110	1110 00010	0110	
20	11	0	1	0000	00001000000	0000	(INTRPT = 64)
21 (DESADD)	00	1110		0000	1100 00000	0001	
22	00	1001		1101	1101 00010	0110	
23	11	0	1	0000	00001000000	0010	(INTRPT = 64)
24 (REGSRC)	00	1111		0000	1100 00000	0001	
25	01	1101		0000	1001 00000	0000	
64 (INTRPT)	11	0	0	1010	00000000000	0000	(FETCH = 0)
65	11	0	0	1011	00000000000	0000	(FETCH = 0)
66	00	0110		0000	0110 00110	0000	
67	00	0110		0000	1100 00000	1010	
68	00	1000		0000	1101 00000	0010	
69	00	0000		0000	1111 00000	1011	
70	00	1101		0000	1000 00000	0000	
71	11	0	1	0000	00000000000	0000	(FETCH = 0)

is determined from its position in the label column. The total number of bits in each binary microinstruction is 23.

10-8 MICROPROGRAM ROUTINES

The complete design of the CPU requires that we formulate a set of instructions for the computer and write the microprogram routines for each instruction in the set. A possible set of instructions for the computer was developed in Chapter 9. The microprogram for the entire CPU will not be carried out here because it will take too much space. However, we will show examples of microprogram routines for the execution of some instructions with different format types. These examples will provide enough information for writing other microprogram routines for additional instructions.

Type 1 ADD Instruction

In order to write the microprogram that executes a type 1 ADD instruction it is necessary to review the type 1 format from Figure 10-7. This format is repeated in Figure 10-14 to provide a more convenient reference. The operation code is in *IR* bits 8 through 15. The register specified by the register field in bits 0, 1, and 2 is denoted by $R(D)$. *IR*3 specifies a direct or indirect mode for the register. *IR*4 and *IR*5 constitute the *MOD* field that specifies four addressing modes in conjunction with the address field of the instruction in the second word *W*. *IR*7 in the *SD* field is equal to 0 for a two operand instruction and is equal to 1 for a one operand instruction; and *IR*6 specifies the source and destination operands. In addition, it may be advisable to refer to Table 10-4 for a list of seven possible ways that a type 1 ADD instruction can be formulated with bit *I* in *IR*3 equal to 0.

Figure 10-15 shows a flowchart for the microprogram routine that executes the type 1 ADD instruction. Bit *IR*7 must be equal to 0 for this two operand instruction; so we can save time by not checking it unless we want to check for format errors. The flowchart uses two subroutines whose microprograms are shown in flowchart form in Figures 10-16 and 10-17.

The microprogram starts by checking the bit in *IR*6. When *IR*6 is 0, the *SD* field is 00 since *IR*7 is assumed to be 0. This specifies a source operand in memory and a destination operand that is determined from the register field. The source operand is retrieved from memory into source register *SR* by means of the subroutine MEMSRC. The microprogram then jumps to the third microinstruction of the ADDT0 routine in Table 10-12 to complete the execution of the instruction.

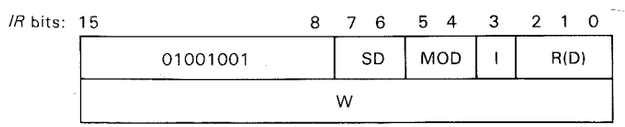


FIGURE 10-14
Type 1 ADD Instruction Format

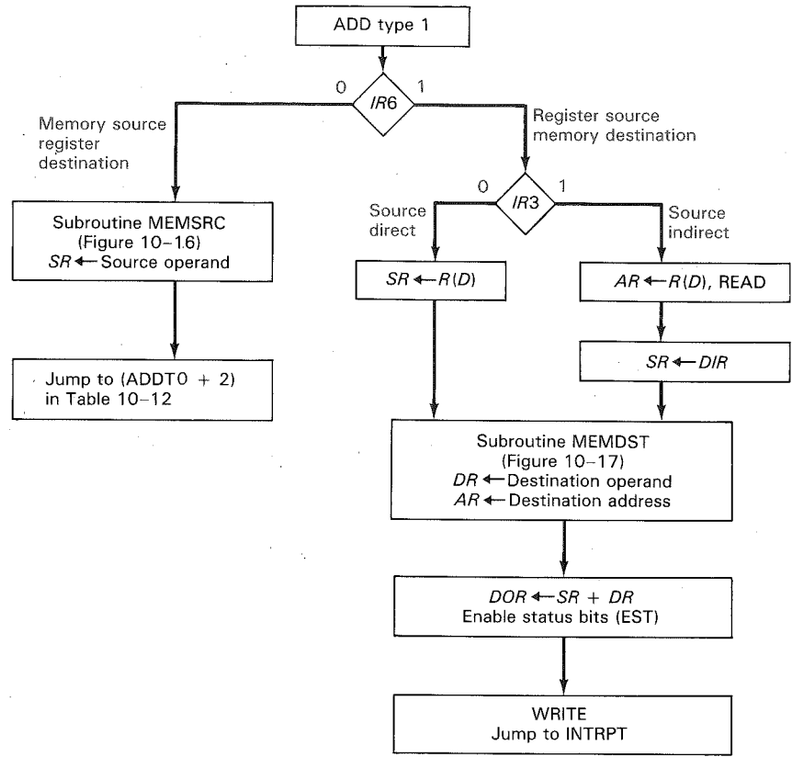


FIGURE 10-15
Microprogram Flowchart for Type 1 ADD Instruction

If *IR*6 = 1, then *SD* = 01. The source operand is evaluated from the register field and the destination operand is in memory. The indirect bit in *IR*3 determines if the register has the operand or the address of the operand. In either case, the source operand is transferred to *SR*. The microprogram then calls subroutine MEMDST to fetch the destination operand from memory and place it in the destination register *DR*. The address of the operand is available in *AR*. The two operands are added and stored in memory. The microprogram jumps to the interrupt routine to check for interrupts.

Subroutine MEMSRC shown in Figure 10-16 reads the source operand from memory and places it in the source register *SR*. The address in *PC* is transferred to *AR* in order to read the memory word *W* and place it in *DIR*. *PC* is then incremented by one. Bits *IR*5 and *IR*4 of the *MOD* field are checked to determine the addressing mode of the instruction. In the immediate mode, the operand *W* which is now in *DIR* is transferred into *SR*. In the other three modes, *DIR* holds the address field of the instruction. The effective address is computed and then transferred to *AR* to read the operand. The subroutine exits with the source operand in *SR*.

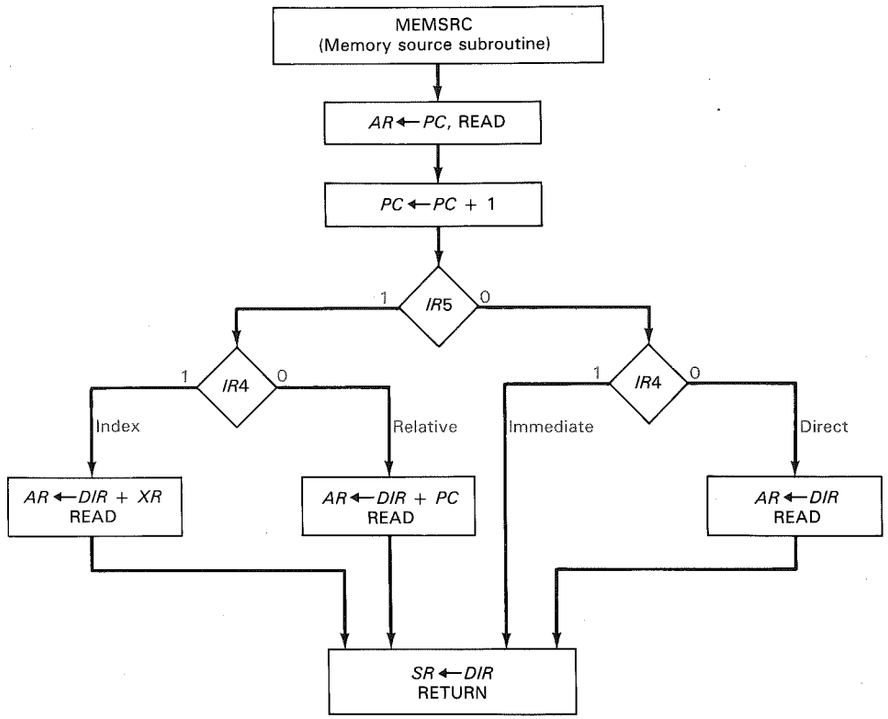


FIGURE 10-16
Flowchart for MEMSRC Subroutine

Figure 10-17 shows the flowchart for subroutine MEMDST. It reads the destination operand from memory and places it in the destination register *DR*. The immediate mode is not used as a destination, and if the instruction uses it by mistake, the microprogram treats it as a direct mode. The subroutine returns with the address of the destination in register *AR*.

The symbolic microprogram can be obtained from the three flowcharts. It will not be listed here but will be left as an exercise in the Problems section.

Type 1 Increment Instruction

The increment instruction must be of type 1 format with $IR7 = 1$ because it is a one operand instruction. Table 10-5 lists the five possible ways that this instruction can be formulated with the various addressing modes. The instruction format is as shown in Figure 10-14 except that the binary operation code is 01010011.

The symbolic microprogram routine for the increment instruction is listed in Table 10-14. If the bit in *IR*6 is equal to 0, the *SD* field is equal to 10, indicating that the operand is in memory. Subroutine MEMDST from Figure 10-17 is called to place the operand in the destination register *DR* and the address of the operand

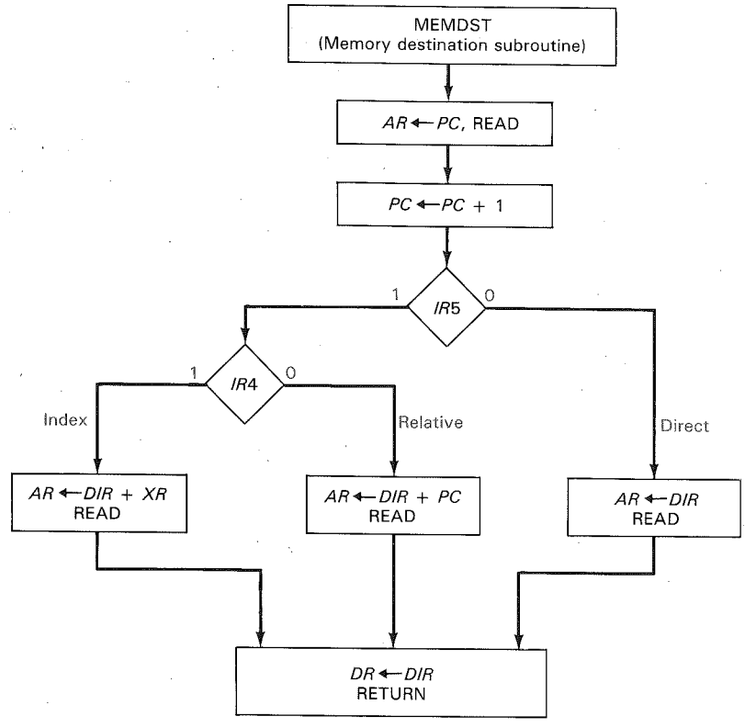


FIGURE 10-17
Flowchart for MEMDST Subroutine

TABLE 10-14
Microprogram for Type 1 Increment Instruction

Label	CS	AS CSBR	BS PS	DS ST	FC AD	MS MS	Comments
INR:		JUMP	T	IR6	REGINR	—	Reg operand if $IR6=1$
		CALL	T	U	MEMDST	—	Subroutine Figure 10-17
	NEXT	DR	—	DOR	INC	ESZ	$DOR \leftarrow DR + 1$, enable S, Z
		JUMP	T	U	INTRPT	WRITE	Store in memory
REGINR:		JUMP	T	IR3	INDINR	—	Indirect if $IR3=1$
	NEXT	R(D)	—	R(D)	INC	ESZ	$R(D) \leftarrow R(D) + 1$
		JUMP	T	U	INTRPT	—	Go to INTRPT
INDINR:	NEXT	R(D)	—	AR	TSF	READ	$AR \leftarrow R(D)$, Read
	NEXT	DIR	—	DOR	INC	ESZ	$DOR \leftarrow DIR + 1$
		JUMP	T	U	INTRPT	WRITE	Store in memory

in the address register *AR*. The number in *DR* is incremented and placed in *DOR*. The operand in *DOR* is then written into memory at the address given by *AR* and the microprogram jumps to check for interrupt.

If the bit in *IR*6 is equal to 1, then the *SD* field is equal to 11 and the operand is evaluated from the register field *R*(*D*). This is done in the microprogram beginning from the microinstruction at the symbolic address REGINR. If *IR*3 = 0, the operand in *R*(*D*) is incremented. If *IR*3 = 1, the instruction specifies a register-indirect mode and the microprogram jumps to INDINR. The operand is read from memory into *DIR*. It is then incremented and placed in *DOR*. The last microinstruction writes the operand in memory and jumps to check for interrupt. Note that the status bits *S* and *Z* are updated with this instruction.

Type 2 and 3 Instructions

Examples of symbolic microprogram routines for type 2 and 3 instructions are presented in Table 10-15. The branch unconditional instruction is executed with the BRNCH microroutine. The address field *W* is read from memory by using the address from *PC*. The value of *W* which is read into *DIR* is then transferred to *PC* in the second microinstruction.

The branch on carry instruction branches to the address in *W* if the carry bit is equal to 1. The microprogram routine BRC checks the status bit *C*, and if it is equal to 1, the microprogram jumps to BRNCH to execute the branch. If *C* is equal to 0, *PC* must be incremented by one in order to skip the second word of the instruction since it is not used. This prepares the *PC* for the address of the next instruction in the program.

The return from subroutine instruction pops the stack and transfers the contents of the top of the stack to *PC*. This is done by the routine RET. The stack pointer

TABLE 10-15
Microprogram Examples for Type 2 and 3 Instructions

Label	CS	AS CSBR	BS PS	DS ST	FC AD	MS MS	Comments
Branch unconditional microprogram (type 2 instruction):							
BRNCH:	NEXT	PC	—	AR	TSF	READ	$AR \leftarrow PC$, Read *W*
	NEXT	DIR	—	PC	TSF	—	$PC \leftarrow DIR$
		JUMP	T	U	INTRPT	—	Go to INTRPT
Branch on carry microprogram (type 2 instruction):							
BRC:		JUMP	T	C	BRNCH	—	If $C = 1$, go to BRNCH
	NEXT	PC	—	PC	INC	—	$PC \leftarrow PC + 1$
		JUMP	T	U	INTRPT	—	Go to INTRPT
Return from subroutine (type 3 instruction):							
RET:	NEXT	SP	—	AR	TSF	READ	$AR \leftarrow SP$, Read
	NEXT	DIR	—	PC	TSF	—	$PC \leftarrow DIR$
	NEXT	SP	—	SP	INC	—	$SP \leftarrow SP + 1$
		JUMP	T	U	INTRPT	—	Go to INTRPT

SP provides the address for the stack. The item on top of the stack is read from memory and placed in *DIR*. The content of *DIR* is transferred into *PC* and *SP* is incremented by one.

A careful study of the examples given in this section and the previous one should be helpful for writing microroutines for other instructions. The problems at the end of the chapter provide additional examples for writing microprogram routines.

10-9 CONTROL UNIT

The hardware of the control unit consists of the control memory that stores the microprogram, the address sequencing logic that chooses the next microinstruction address, and decoders and multiplexers that select the various components of the microoperation. The part of the control unit that determines the address of the next microinstruction is called the microprogram sequencer. In addition to the sequencer, the control must incorporate the hardware for decoding certain fields of the microinstruction.

Microprogram Sequencer

The block diagram of the microprogram sequencer together with the control memory are shown in Figure 10-18. The control memory has 2048 words of 23 bits each and requires an address of 11 bits. There are two multiplexers in the circuit. The address multiplexer selects an address from one of four sources and routes it into the control address register *CAR*. The new address loaded into *CAR* is then supplied to the control memory for reading the next microinstruction. The status multiplexer tests the value of a selected status bit and the result of the test is applied to an address selection logic circuit. The output of *CAR* is incremented and applied to one of the multiplexer inputs and to the subroutine register *SBR*. The other three inputs to the address multiplexer come from the address field *AD* of the present microinstruction, from the output of *SBR*, and from a mapping of the operation code in *IR*.

The mapping PLD is a programmable logic device (see Section 6-6) with eight inputs and 11 outputs. It maps or transforms the 8-bit operation code from its value in *IR*(8-15) to an 11-bit control memory address. The value of the address is determined from the microprogram routine that executes the instruction. For example, if the 8-bit operation code for the type 0 ADD instruction is 00001001 as chosen in Table 10-2, and the microprogram routine that implements this instruction starts from address 16 as shown in Table 10-13, the mapping PLD transforms the 8-bit operation code 00001001 to an 11-bit address 00000010000.

The incrementer circuit in the sequencer is not a counter with flip-flops but rather a combinational circuit constructed with gates. The circuit can be designed with 11 half adder circuits (see Figure 3-8) connected in cascaded stages. The output carry from one stage is applied to the input of the next higher order stage. The input carry in the first least significant stage must be equal to 1 to provide the increment by one operation.

The *ST* field of the microinstruction selects one of the status bits from the inputs of the status multiplexer. The polarity of the selected bit is determined from the

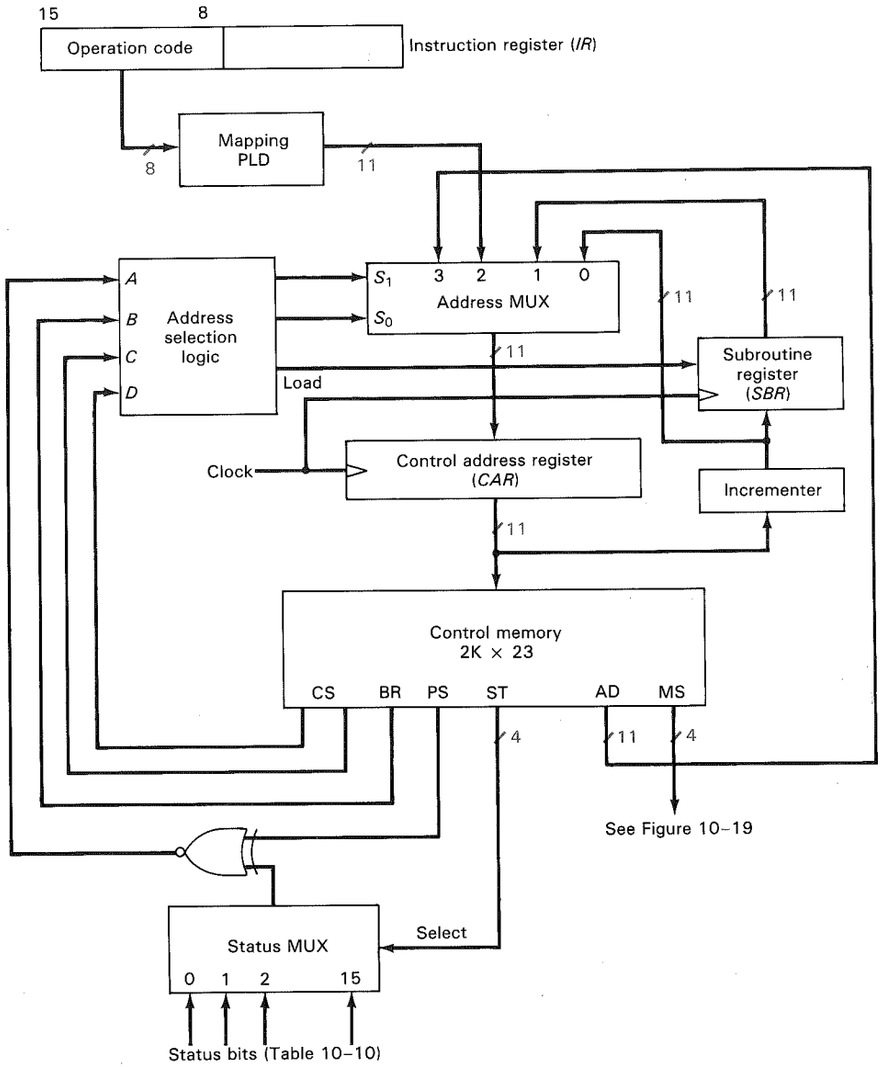


FIGURE 10-18
Microprogram Sequencer and Control Memory

bit value of the *PS* field. The exclusive-NOR gate produces an output of 1 if both of its inputs are equal to 1 or if both of its inputs are equal to 0. Thus the output of the gate is equal to 1 if the *PS* field bit is equal to the value of the selected status bit.

The address selection logic in the sequencer has four inputs and three outputs. The truth table for this circuit is shown in Table 10-16. Inputs *D* and *C* are identical to the bit values of the *CS* field and input *B* is equal to the bit value of the *BR* field. Input *A* receives the status of the selected bit from the status multiplexer. The values for S_1 and S_0 are determined from the required path in the address multiplexer. When *CS* is equal to 00, 01, or 10, inputs *D* and *C* go directly to

TABLE 10-16
Truth Table for Address Selection Logic

		Inputs				Outputs				
CS	BR	D	C	B	A	S_1	S_0	Load	Address MUX selection	
00	—	0	0	X	X	0	0	0	Increment *CAR*	(NEXT)
01	—	0	1	X	X	0	1	0	Subroutine register	(RET)
10	—	1	0	X	X	1	0	0	Mapping of op-code	(MAP)
11	0	1	1	0	0	0	0	0	Increment *CAR*	
11	0	1	1	0	1	1	1	0	*AD* field	(JUMP)
11	1	1	1	1	0	0	0	0	Increment *CAR*	
11	1	1	1	1	1	1	1	1	*AD* field, load *SBR*	(CALL)

outputs S_1 and S_0 and the other inputs of the address selection logic have no effect on the address multiplexer. When *CS* is equal to 11, the selected input to the address multiplexer depends on inputs *B* and *A*. When input *A* is equal to 0, it means that the selected bit in the status multiplexer does not have the required polarity. In this case, *CAR* is incremented by one. If input *A* is equal to 1, both S_1 and S_0 must be equal to 1 in order to transfer the address field *AD* through the address multiplexer. If input *B* is also equal to 1, it signifies a call subroutine microinstruction, and the load input to the subroutine register *SBR* is activated.

The truth table from Table 10-16 can be used to obtain the simplified Boolean functions for the address selection logic circuit. Using maps for the two functions we obtain the following simplified Boolean equations:

$$S_1 = ACD + \overline{C}D$$

$$S_0 = ACD + C\overline{D}$$

$$\text{Load} = ABCD$$

The combinational circuit has a common gate for the product term *ACD*. S_1 and S_0 can be implemented with three AND gates, two OR gates, and two inverters. The condition for storing the return address in *SBR* is generated by the load signal with an AND gate with inputs *A*, *B*, *C*, and *D*.

Decoding of Type A Microinstruction

The microprogram sequencer of Figure 10-18 implements the hardware necessary for the common field *CS* and the other fields of the type B microinstruction. The common field *MS* must be decoded separately and the other fields of type A microinstruction format need decoding circuits before they are applied to the buses in the processor unit.

Figure 10-19 shows the additional circuits necessary for interpreting the type A microinstruction. The *CS* and *MS* fields are common to both formats. The *FC* field is connected directly to the selection inputs of the ALSU to provide the specified operation. The *AS*, *BS* and *DS* fields designate the register to be applied to the CPU buses. The code assignment for the registers is given in Table 10-7. The *DS* field can choose any one of 14 destination registers. When *DS* = 1110, we use the

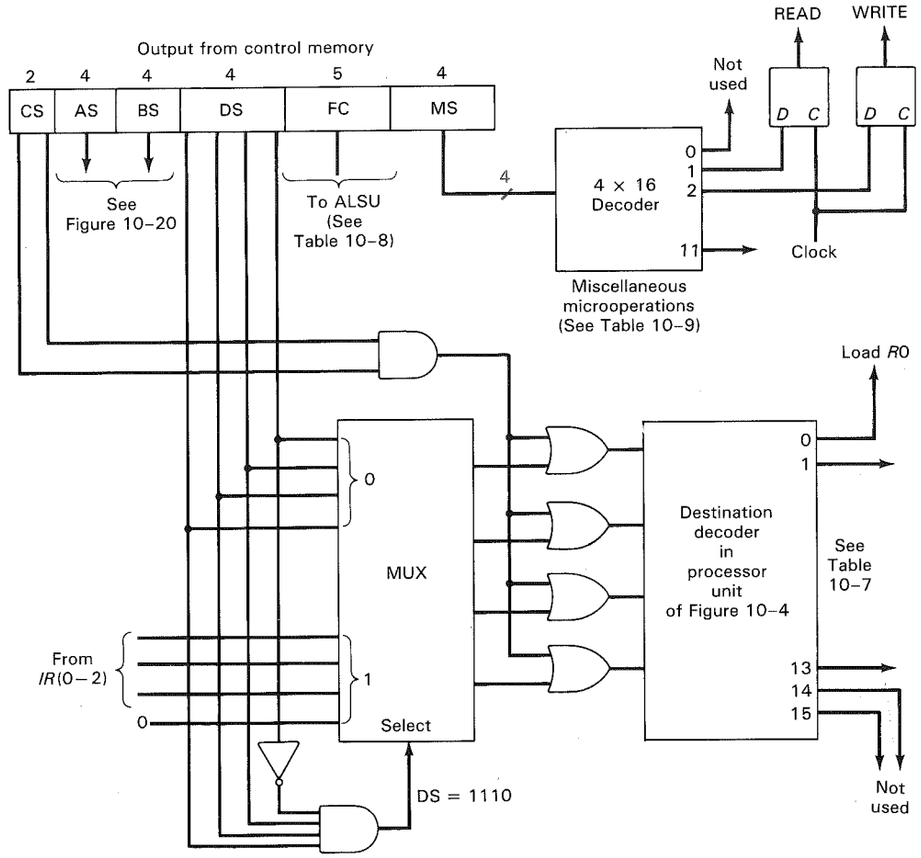


FIGURE 10-19
Decoding of a Type A Microinstruction

symbol $R(D)$ to designate a processor register $R0$ through $R7$ as specified by the 3-bit code in $IR(0\text{-}2)$. When $DS = 1111$, none of the registers are selected.

The multiplexer in Figure 10-19 implements the $R(D)$ symbolic designation. When DS is not equal to 1110, the multiplexer forms a path from its 0 inputs to the outputs. This transfers the 4-bit code from DS into the destination decoder in the processor unit. When $DS = 1110$, the multiplexer forms a path from its 1 inputs and transfers the bits from $IR(0\text{-}2)$ with a 0 in the fourth bit.

When the CS field is equal to binary 11 it designates a branch type B microinstruction. An ALSU operation must be inhibited when the CS field is equal to 11. This is done by forcing four 1's into the destination decoder in the processor unit. Output 15 of the decoder is not connected to any register load input and, therefore, none of the registers receives the output of the ALSU when the CS field is equal to 11.

The decoder for the MS field can provide up to 15 independent microoperations but only 11 microoperations are used and they are listed in Table 10-9. Output 0 of the decoder is not connected anywhere and provides a "no operation" condition. Outputs 1 and 2 of the decoder go to the D inputs of the READ and WRITE flip-

flops. These flip-flops are set to 1 when their D input is equal to 1 while a clock pulse transition occurs. The outputs of the flip-flops generate the corresponding read and write signals for the memory unit and also control the bidirectional bus buffers in the data bus of the CPU (see Figure 10-5). When the D input of a flip-flop is equal to 0, the next clock transition resets the output to 0.

The two fields AS and BS that select registers for the input buses of the ALSU go through multiplexers as shown in Figure 10-20. The multiplexer associated with

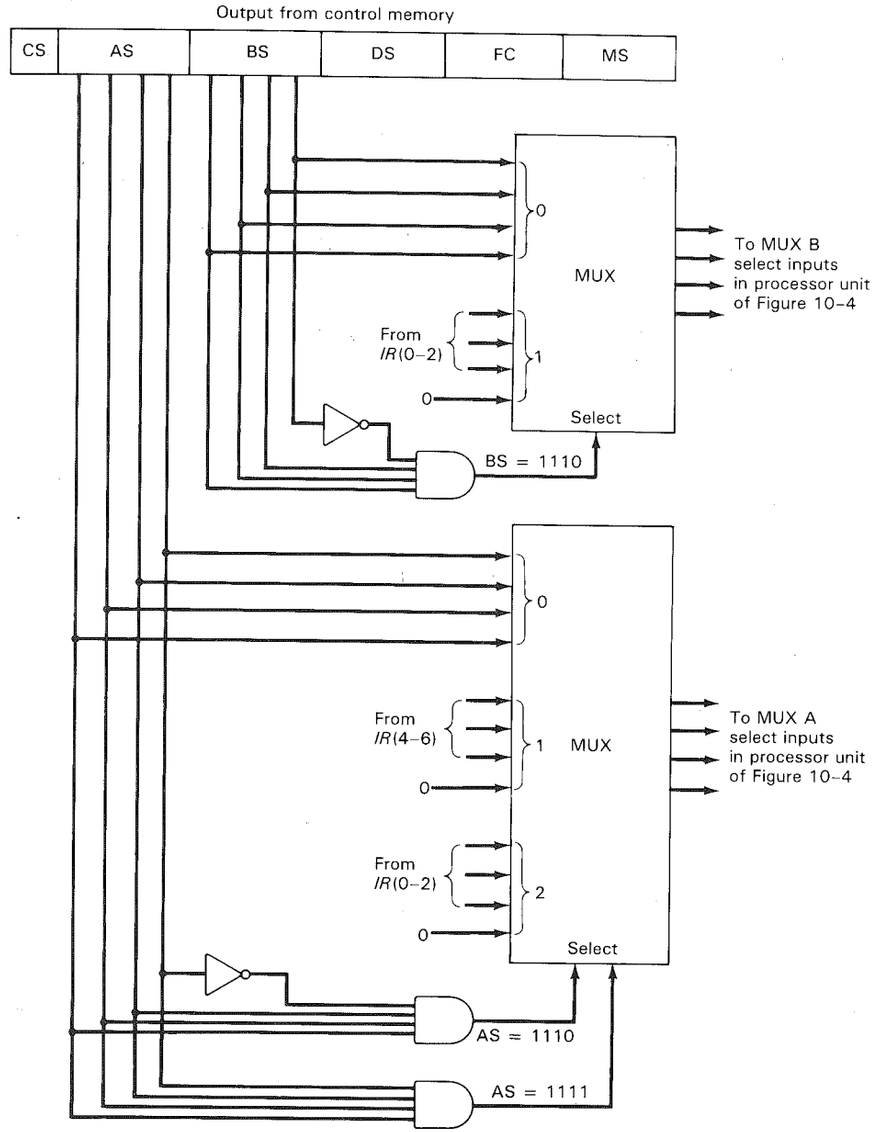


FIGURE 10-20
Selection of Registers in MUX A and MUX B of Processor Unit

BS implements the *R(D)* symbolic designation in a manner similar to the *DS* field. The output of this multiplexer is applied to the selection inputs of MUX B in the processor unit that selects the register for the *B* input of the ALSU. The multiplexer for *AS* implements the symbolic designations of $R(D)$ and $R(S)$ as defined in Table 10-7. The select inputs of the multiplexer are equal to 00 when the *AS* field is not equal to 1110 or 1111. This transfers the code from *AS* to the multiplexer output. The select inputs are equal to 01 when $AS = 1111$. This implements the $R(S)$ condition by transferring the code from the source field in $IR(4\text{-}6)$ into the select inputs of MUX A in the processor. The selection inputs are equal to 10 if $AS = 1110$. This implements the $R(D)$ condition by transferring the code from the destination register in $IR(0\text{-}2)$ into the select inputs of MUX A.

The control unit is synchronized with the same clock pulses that are applied also to the processor unit. Every clock pulse produces the following operations when applicable:

1. A microoperation is executed in the processor unit if *CS* is not equal to 11;
2. The microoperation specified in the *MS* field (if any) is executed;
3. The next microinstruction address is transferred into *CAR* and the corresponding microinstruction is read from control memory after a given time delay;
4. The return address is transferred into the subroutine register if a call instruction is satisfied.

All these operations occur simultaneously with the same common clock pulse transition. This is repeated for each clock pulse and each microinstruction that is read from control memory.

REFERENCES

1. MANO, M. M., *Computer System Architecture.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1982.
2. TANENBAUM A. S. *Structured Computer Organization.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1984.
3. LEWIN, M. H. *Logic Design and Computer Organization.* Reading, MA: Addison-Wesley, 1983.
4. HAMACHER, V. C., VRANESIC, Z. G., AND ZAKY S. G. *Computer Organization,* 2nd ed. New York: McGraw-Hill, 1984.
5. PROSSER, F. AND WINKEL, D. *The Art Digital Design.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1987.

PROBLEMS

10-1 Draw the diagram of the shift left unit similar to Figure 10-3.

10-2 A bidirectional bus line with three-state buffers is shown in Figure 7-9. Using this figure as a reference, show the construction of one line of the bidirectional data bus buffer in the CPU block diagram of Figure 10-5.

10-3 Interpret the following hexadecimal instructions using Figure 10-7 and Table 10-2 as reference. Express each instruction in symbolic form and in register transfer notation.
(a) 4A4B; (b) 0BC9; (c) 5E80; (d) 4B25

10-4 For each of the following symbolic instructions, specify the operation in register transfer notation and give the corresponding hexadecimal code (see Table 10-2 and Figure 10-7).
(a) SUB (R3), R5
(b) MOVE R3, (R0)
(c) MOVE #W, (R1)
(d) PUSH W(X)

10-5 Evaluate the 4-digit hexadecimal code of the instruction that executes the following operations (use Table 10-2 and Figure 10-7).
(a) $R5 \leftarrow M[R2]$
(b) $M[W] \leftarrow R3$
(c) $XR \leftarrow XR - 1$
(d) $PC \leftarrow M[SP]$, $SP \leftarrow SP + 1$

10-6 Repeat Table 10-4 using a register indirect mode with $R2$ as the selected register.

10-7 Each set of microoperations listed below can be specified with one microinstruction. Choose the symbolic name and binary value from Figure 10-8(b) for the fields of the microinstruction.
(a) $DR \leftarrow R(D)$, $CAR \leftarrow CAR + 1$
(b) $R0 \leftarrow R0 - 1$, $DIR \leftarrow M[AR]$, $CAR \leftarrow CAR + 1$
(c) $TR \leftarrow ZR + NR$, $M[AR] \leftarrow DOR$, $\text{CAR} \leftarrow SBR$

10-8 Convert each of the following 6-digit hexadecimal number to a 23-bit number. Let the binary number obtained from the conversion be a microinstruction as defined in Figure 10-8. Interpret the microinstruction and explain its operation.
(a) 1C1400; (b) 620000; (c) 334A26.

10-9 Table 10-7 lists a total of 17 registers in the three fields *AS*, *BS*, and *DS*.
(a) Identify the 14 registers inside the processor unit and the three registers that communicate with memory.
(b) Identify the nine registers that can be changed by program instructions.
(c) Identify three registers used by the control for temporary storage.
(d) Give examples where registers *ZR* and *NR* can be used. Why is it that these two registers are not listed under the *DS* field?
(e) What is the meaning of the symbols $R(S)$, $R(D)$, and NONE?

10-10 The *NR* register defined in Table 10-7 holds a constant for use in microprogram loops. Give the microinstruction that will transfer the content of *NR* to the temporary register *TR* for possible manipulation.

10-11 Modify the shift left circuit of Problem 10-1 by including the logic for updating the overflow status bit *V* during an arithmetic shift left microoperation.

10-12 The carry flip-flop in the status register is updated from the following four sources:
(a) From A_0 during a right shift (Figure 10-3).
(b) From A_{15} during a left shift (Problem 10-1).
(c) From the output carry C_{out} in the arithmetic unit when signals EST or ECB (Table 10-9) are enabled in the *MS* field.
(d) From the SCF and RCF microoperations in the *MS* field.

Design the gate logic for the input of the carry flip-flop using a JK type flip-flop.

10-13 Convert each of the nine symbolic microinstructions listed in Table 10-11 into a 23-bit binary equivalent. Assume that DEST = 9, START = 3, and FETCH = 0.

10-14 Determine the time it takes to fetch and execute a type 0 ADD instruction by going through the microprogram in Table 10-12. Assume that EIF = 1 but that there is no external interrupt. Assume that each microinstruction takes t nanoseconds. Evaluate the time for each of the four possible combinations of $IR(7)$ and $IR(3)$.

10-15 Write the symbolic microprogram for the type 1 ADD instruction from the information given in the flowchart of Figure 10-15.

10-16 Translate the microprogram listed in Table 10-14 into binary. Assume that the INR routine starts from address 32 and that INTRPT is at 64.

10-17 Draw a flowchart for the INR microroutine whose symbolic microprogram is listed in Table 10-14.

10-18 Write a microprogram for a type 0 MOVE instruction.

10-19 Write a microprogram for a type 1 MOVE instruction.

10-20 Write a microprogram for the PUSH stack instruction. Use the type 1 format with one operand. Note that all four modes are possible with the memory operand.

10-21 Write a microprogram for the type 1 NEGATE instruction. This instruction forms the 2's complement of the operand.

10-22 Write a microprogram for a type 2 CALL subroutine instruction.

10-23 Write a microprogram for the following type 2 instruction:

CALL subroutine if positive and non-zero

Assume that the S and Z status bits have been updated by a previous subtract instruction.

10-24 Write a microprogram for the logical shift right instruction. Use the type 1 instruction format (Figure 10-7) with two interpretations. When $IR(7) = 1$, the instruction has one operand and produces a single shift. When $IR(7) = 0$, then the REG and I fields of the instruction specify the operand to be shifted and W gives the number of times to shift. In the single shift case, the microprogram checks the SD and MOD fields to determine the operand. In the multiple shifts case, the SD and MOD fields are not used.

10-25 Write a microprogram for type 0 unsigned MULTIPLY instruction. The source and destination fields of the instruction specify the multiplier and multiplicand. The double length product is always left in registers $R0$ and $R1$.

10-26 Modify the type 2 instruction format of Figure 10-7 to include a relative mode branch instruction. This is done by using bit 13 of the instruction code to distinguish between the two addressing modes. If bit 13 is equal to 0, the interpretation remains the same as is shown in Figure 10-7, with W being the branch address. If bit 13 is equal to 1, bits 8-12 of the instruction specify the operation code and bits 0-7 contain a signed number in the range of +127 to −128 for the relative address. The second word W is not used in the relative mode because the branch address is computed by adding the contents of PC to the address field in $IR(0\text{-}7)$ with sign extension. The sign extension produces a 16-bit number with nine leading 0's if the sign bit $IR(7) = 0$ or a 16-bit number with nine leading 1's if the sign bit $IR(7) = 1$.

(a) Show any hardware addition that is needed in the CPU of Figure 10-5 to allow the 16-bit signed number with sign extension to enter the processor unit. (Try a multiplexer with IR and DIR as inputs going into the processor bus).

(b) What additions must be included in Table 10-9 and Table 10-10?

(c) Rewrite the two branch microprograms of Table 10-15 taking into account this new modification of the type 2 format.

10-27 The hardware of the processor unit as defined in Section 10-3 does not have a data path from the status bits to the memory unit. This path is necessary in order to push the status bits onto the stack after an interrupt or a call to subroutine. Similarly, a path must be provided to restore the status bits upon return from interrupt or subroutine. Modify the hardware of the CPU and write the microprogram routines for the instructions "push status bits" and "pop status bits." (One way that the hardware may be modified is to include two additional microoperations in Table 10-9 that transfer the status bits to and from the temporary register TR.)

10-28 Design the incrementer circuit shown in Figure 10-16 using 11 half adder circuits.

10-29 Derive the simplified Boolean functions for the address select logic from the truth table listed in Table 10-16.

10-30 Change the exclusive-NOR gate in Figure 10-16 to an exclusive-OR gate. Make any corrections in the entries of Table 10-18 to conform with this change and redesign the address selection logic.

10-31 Why is it that the processor unit must be disabled when the two bits of the CS field of the microinstruction are both equal to 1? Indicate how the hardware of the control unit fulfills this requirement.

11

INPUT–OUTPUT AND COMMUNICATION

11-1 INTRODUCTION

The input and output subsection of a computer provides an efficient mode of communication between the central processing unit (CPU) and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. Among the input and output devices that are commonly found in computer systems are keyboards, display terminals, printers, and magnetic disks. Other input and output devices encountered are magnetic tape drives, digital plotters, optical readers, analog-to-digital converters, and various data acquisition equipment. Computers can be used to control various processes such as machine tooling, assembly line procedures, and industrial control.

The input and output facility of a computer is a function of its intended application. The difference between a small and large system is partially dependent on the amount of hardware the computer has available for communicating with other devices and the number of devices connected to the system. Since each device behaves differently, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each device. However, there are certain techniques and procedures that are common to most devices.

In this chapter we present some of the common characteristics found in the input-output subsystem of computers. The various techniques available for transferring data either in parallel, using many conducting paths, or serially, through communication lines, are presented. A description of the different types of transfer modes is given, starting from a simple example of program control transfer to a configuration of systems with multiple processors.

11-2 INPUT–OUTPUT INTERFACE

Devices that are in direct control of the processing unit are said to be connected on-line. These devices transfer binary information into and out of the memory unit upon command from the CPU. Input or output devices attached to the computer on-line are called *peripherals*.

Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are

1. Peripherals are electromechanical devices and their manner of operation is different from the operation of the CPU and memory which are electronic devices. Therefore, a conversion of signal values may be required.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU. Consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals differ from each other and each must be controlled in a way that does not disturb the operation of other peripherals connected to the CPU.

To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called *interface* units because they interface between the processor bus and the peripheral device. In addition, each device has its own controller to supervise the operations of the particular mechanism of that peripheral. For example, the controller in a printer attached to a computer controls the paper motion, the print timing, and the selection of printing characters.

I/O Bus and Interface Units

A typical communication link between the CPU and several peripherals is shown in Figure 11-1. Each peripheral has associated with it an interface unit. The common bus from the CPU is attached to all peripheral interfaces. To communicate with a particular device, the CPU places a device address on the address bus. Each interface attached to the common bus contains an address decoder which monitors the address lines. When the interface detects its own address, it activates the path between the bus lines and the device that it controls. All peripherals with addresses that do not correspond to the address in the bus are disabled by their interface. At the same time that the address is made available in the address bus, the CPU provides a function code in the control lines. The selected interface responds to the function code and proceeds to execute it. If data has to be transferred, the interface communicates with both the device and the CPU data bus to synchronize the transfer.

The four I/O devices shown in Figure 11-1 are employed in practically any general purpose digital computer. The keyboard is an input device used for entering data

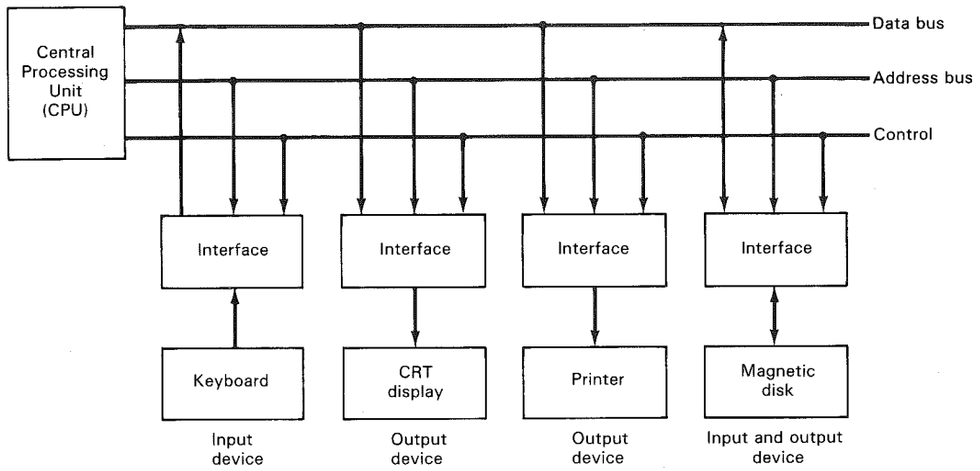


FIGURE 11-1
Connection of I/O Devices to CPU

into the computer. When the user depresses a key, the interface produces a binary code corresponding to the particular alphanumeric character in the keyboard. The most often used code is ASCII (American Standard Code for Information Interchange: see Table 1-6).

The CRT (cathode ray tube) display is often used as an output device for graphics or text entered from the keyboard and stored in computer memory. The CRT used in computer displays are the same type as in television sets. The graphics or text is displayed by positioning an electron beam in the tube and illuminating the surface at the proper time.

The printer is a useful output device for producing hard copy. Some printers print only one character at a time across a line. For faster printing, a high-speed line printer that can print an entire line at once is employed.

Magnetic disks are used for bulk storage of programs and data. The disk has a high-speed rotational surface which is coated with magnetic material. Access is achieved by moving a read/write mechanism to a track in the magnetized surface. When necessary, blocks of data are transferred to and from the disk and computer memory.

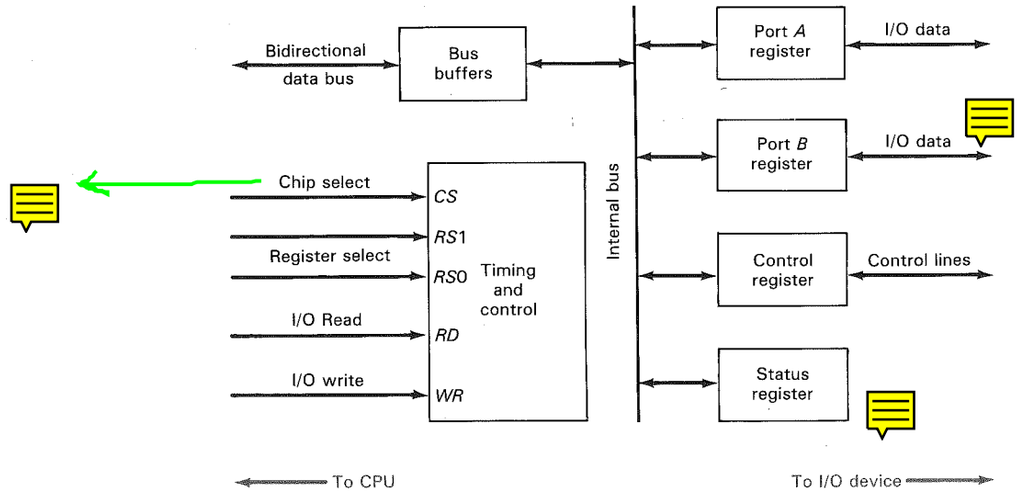
In addition to communicating with the I/O devices, the CPU of a computer must also communicate with the memory unit through an address and data bus. There are three ways that external computer buses communicate with memory and I/O. One method uses common data, address, and control buses for both memory and I/O. This configuration is referred to as memory-mapped I/O. The common address space is shared between the interface units and memory words, with each having a distinct address. Computers that adopt the memory-mapped scheme read and write from interface units as if they were assigned memory addresses.

The second alternative is to share a common address bus and data bus but use different control lines for memory and I/O. Such computers have separate read and write lines for memory and I/O. To read or write from memory, the CPU activates the memory read or memory write control. To input or output from an interface, the CPU activates the read I/O or write I/O control. In this way, the address assigned to memory and I/O interface units are independent from each other and are distinguished by separate control lines. This method is referred to as the isolated I/O configuration.

The third alternative is to have two independent sets of data, address, and control buses. This is possible in computers that include an I/O processor in the system in addition to the CPU. The memory communicates with both the CPU and I/O processor through a common memory bus. The I/O processor also communicates with the input and output devices through separate address, data, and control lines. The purpose of the I/O processor is to provide an independent pathway for the transfer of information between external devices and internal memory. The I/O processor is sometimes called a data channel.

Example of I/O Interface

A typical I/O interface unit is shown in block diagram form in Figure 11-2. It consists of two data registers called *ports*, a control register, a status register, a bidirectional data bus, and timing and control circuits. The function of the interface is to translate the signals between the CPU buses and the I/O device and to provide the needed hardware to satisfy the two sets of timing constraints.



CS	RS1	RS0	Register selected
0	X	X	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

FIGURE 11-2
Example of I/O Interface Unit

The I/O data from the device can be transferred into either port *A* or port *B*. The interface may operate with an output device or with an input device or with a device that requires both input and output. If the interface is connected to a printer, it will only output data; and if it services a card reader, it will only input data. A magnetic tape unit transfers data in both directions but not at the same time; so the interface can use only one set of I/O bidirectional data lines.

The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and device can be placed in a variety of operating modes. For example, port *A* may be defined as an input port only. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in the forward direction. The bits in the status register are used for status conditions and for recording errors that may occur during the data transfer. For example, a status bit may indicate that port *A* has received a new data item from the device. Another bit in the status register may indicate that a parity error has occurred during the transfer.

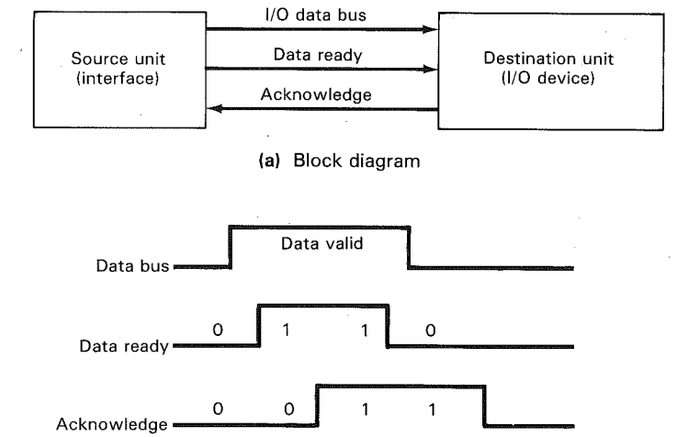
The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select input and the two register select inputs. A circuit must be provided externally (usually a decoder or a gate) to detect the address assigned to the interface registers. This circuit enables the chip select (*CS*) input when the interface is selected by the address bus. The two register select inputs *RS*1 and *RS*0 are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transferred into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

Handshaking

The control lines out of the interface may be used for a variety of applications depending on the type of I/O device used. One important application of these lines is to control the timing of data transfer between the I/O device and the interface. Two units, such as an interface and the controller of the I/O device to which it is attached, are designed with different control and clock generators and are said to be asynchronous to each other. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This kind of arrangement between two independent units is referred to as *handshaking*.

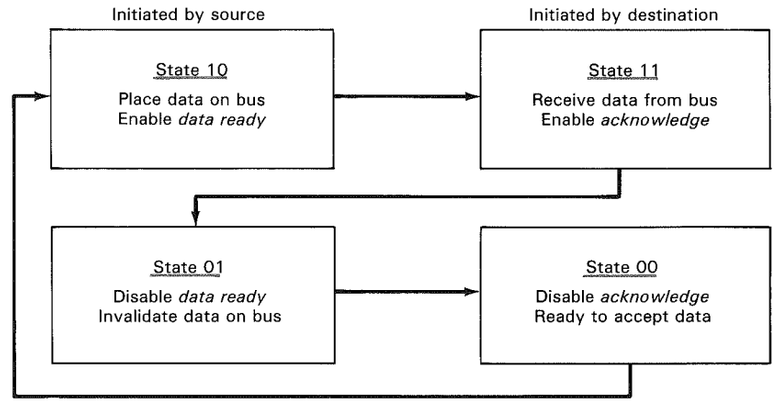
The basic principle of the two-wire handshaking procedure of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit that there is valid data on the bus. The other control line is in the other direction from the destination to the source. It is used by the destination unit to acknowledge the acceptance of data. In this way, each unit informs the other unit of its status and the result is an orderly transfer through the bus.

Figure 11-3 shows the data transfer procedure. The I/O data bus carries binary information from the source unit to the destination unit. Here we assume that the interface is the source and the I/O device is the destination, but the same procedure applies if the transfer is in the other direction. Typically, the bus has multiple data lines to transfer a given number of bits at a time. The two handshaking lines are *data ready*, generated by the source unit, and *acknowledge*, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The initial state is when both the *data ready* and *acknowledge* are disabled



(a) Block diagram

(b) Timing diagram



(c) State sequence of events

FIGURE 11-3
Asynchronous Transfer Using Handshaking

and in the 00 state. The subsequent states are 10, 11, and 01. The state sequence shown in part (c) of the figure lists the events that occur as the transfer goes through the four states. The source unit initiates the transfer by placing the data on the bus and enabling its *data ready* signal. The *acknowledge* signal is activated by the destination unit after it receives the data from the bus. The source unit disables its *data ready* signal in response to the destination's *acknowledge*. The destination unit then disables its *acknowledge* signal and the system goes to the initial state. The source does not send the next data item unless the destination unit shows its readiness to accept new data by disabling its *acknowledge* line.

The handshaking scheme provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units. If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a *timeout* mechanism, which produces an alarm condition if the data transfer is not completed within a predetermined time interval. The timeout is implemented by means of an internal clock that starts counting time when the unit enables one of its handshaking control signals. If the return handshake signal does not respond within a given time period, the unit assumes that an error occurred. The timeout signal can be used to interrupt the CPU and execute a service routine that takes appropriate error recovery action.

11-3 SERIAL COMMUNICATION

The transfer of data between two units may be performed in parallel or serial. In parallel data transfer, each bit of the message has its own path and the total message is transmitted at the same time. This means that an n-bit message is transmitted in parallel through n separate conductor paths. In serial data transmission, each bit in the message is sent in sequence, one at a time. This method requires the use of one pair of conductors or one conductor and a common ground. Parallel transmission is faster but requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but less expensive since it requires only one pair of conductors.

The way that computers and terminals remote from each other are connected is via telephone lines and other public and private communication facilities. Since telephone lines were originally designed for voice communication and computers communicate in terms of digital signals, some form of conversion may be needed. The converters are called *data sets*, *acoustic couplers*, or *modems* (from modulator-demodulator). A modem converts digital signals into audio tones to be transmitted over telephone lines and also converts audio tones from the line to digital signals for computer use. Various modulation schemes as well as different grades of communication media and transmission speeds are used.

Serial data can be transmitted between two points in three different modes: simplex, half-duplex, or full-duplex. A *simplex* line carries information in one direction only. This mode is seldom used in data communication because the receiver cannot communicate with the transmitter to indicate the occurrence of errors. Examples of simplex transmission are radio and television broadcasting.

A *half-duplex* transmission system is one that is capable of transmitting in both directions, but data can be transmitted in only one direction at a time. A pair of wires is needed for this mode. A common situation is for one modem to act as the transmitter and the other as the receiver. When transmission in one direction is completed, the roles of the modems are reversed to enable transmission in the opposite direction. The time required to switch a half-duplex line from one direction to the other is called the turnaround time.

A *full-duplex* transmission can send and receive data in both directions simultaneously. This can be achieved by means of a four-wire link, with a different pair of wires dedicated to each direction of transmission. Alternatively, a two-wire circuit can support full-duplex communication if the frequency spectrum is subdivided into two nonoverlapping frequency bands to create separate receive and transmit channels in the same physical pair of wires.

Serial transmission of data can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency rate, and bits are transmitted continuously at the adopted rate. In long distant serial transmission, the transmitter and receiver units are each driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clock frequencies in step with each other. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission where bits must be transmitted continuously to keep the clock frequencies in both units synchronized.

Asynchronous Transmission

One of the most common applications of serial transmission is for communication with a keyboard and serial printer. Each character consists of an alphanumeric code of eight bits with additional bits inserted at both ends of the character code. In the serial asynchronous data transmission technique, each character consists of three parts: the start bit, the character bits, and the stop bits. The convention is for the transmitter to rest at the 1-state when no characters are transmitted. The first bit, called the start bit, is always 0 and is used to indicate the beginning of a character. An example of this format is shown in Figure 11-4.

A transmitted character can be detected by the receiver from knowledge of the transmission rules. When a character is not being sent, the line is kept in the 1-state. The initiation of a character transmission is detected from the start bit, which is always 0. The character bits always follow the start bit. After the last bit of the

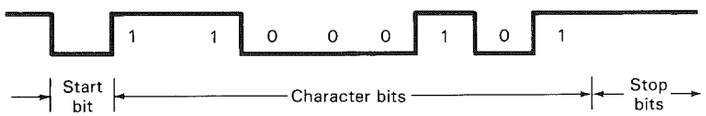


FIGURE 11-4
Format of Asynchronous Serial Transfer

character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time. Using these rules, the receiver can detect the start bit when the line goes from 1 to 0. A clock in the receiver examines the line at proper bit times. The receiver knows the transfer rate of the bits and the number of character bits to accept. After the character bits are transmitted, one or two stop bits are sent. The stop bits are always in the 1-state and frame the end of character to signify the idle or wait state.

At the end of the character, the line is held at the 1-state for a period of at least 1 or 2 bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time insures that a new character will not follow for 1 or 2 bit times.

As an illustration, consider the serial transmission of a terminal with a transfer rate of 10 characters per second. Each transmitted character consists of a start bit, 8 information bits, and 2 stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1 second for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 msec. The *baud* rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

The terminal has a keyboard and a printer. Every time a key is depressed, the terminal sends 11 bits serially along a wire. To print a character in the printer, an 11-bit message must be received along another wire. The terminal interface consists of a transmitter and a receiver. The transmitter accepts an 8-bit character from the computer and proceeds to send a serial 11-bit message to the printer line. The receiver accepts a serial 11-bit message from the keyboard line and forwards the 8-bit character code into the computer. Integrated circuits are available which are specifically designed to provide the interface between computer and similar interactive terminals.

Synchronous Transmission

Synchronous transmission does not use start-stop bits to frame characters. The modems used in synchronous transmission have internal clocks that are set to the frequency at which bits are being transmitted. For proper operation, it is required that the clock of the transmitter and receiver modems remain synchronized at all times. The communication line, however, carries only the data bits, from which the clock information must be extracted. Frequency synchronization is achieved by the receiving modem, from the signal transitions that occur in the received data. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock at the frequency of the incoming bit stream. In this way, the same rate is maintained in both the transmitter and receiver.

Contrary to asynchronous transmission, where each character can be sent separately with its own start and stop bits, synchronous transmission must send a

continuous message in order to maintain synchronism. The message consists of a group of bits that form a block of data. The entire block is transmitted with special control bits at the beginning and the end, in order to frame the entire block into one unit of information.

The communication lines, modems, and other equipment used in the transmission of information between two or more stations is called a *data link*. The orderly transfer of information in the data link is accomplished by means of a *protocol*. A data link protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of information. The purpose of the data link proctocol is to establish a connection between two stations, to identify the sender and receiver, to ensure that all messages are passed correctly without errors, and to handle all control functions involved in a sequence of data transfer. Protocols are divided into two major categories according to the message framing technique used. These are character-oriented protocol and bit-oriented protocol.

Character-Oriented Protocol

The character-oriented protocol is based on the binary code of a character set. The code most commonly used is ASCII. It is a 7-bit code with an eighth bit for parity check. The code has 128 characters, of which 95 are graphic characters and 33 are control characters. The list of ASCII characters can be found in Table 1-6. The control characters are used for the purpose of routing data, arranging the text in the desired format, and for the layout of the printed page. The characters that control the transmission are called *communication control* characters. These characters are listed in Table 11-1. Each character code has seven bits plus an even parity bit. It is referred to by a three-letter symbol. The role of each character in the control of data transmission is stated briefly in the function column of the table.

The SYN character serves as synchronizing agent between the transmitter and receiver. The 8-bit code 10010110 has the property that, upon circular shifting, it does not repeat itself until after a full 8-bit cycle. When the transmitter starts sending characters, it sends a few SYN characters first and then sends the actual message. The initial continuous string of bits accepted by the receiver is checked for a SYN character. In other words, with each clock period, the receiver checks

TABLE 11-1
ASCII Communication Control Characters

Code	Symbol	Meaning	Function
10010110	SYN	Synchronous idle	Establishes synchronism
10000001	SOH	Start of heading	Heading of block message
10000010	STX	Start of text	Precedes block of text
00000011	ETX	End of text	Terminates block of text
10000100	EOT	End of transmission	Concludes transmission
00000110	ACK	Acknowledge	Affirmative acknowledge
10010101	NAK	Negative acknowledge	Negative acknowledge
00000101	ENQ	Inquiry	Inquire if terminal is on
00010111	ETB	End of transmission block	End of block of data
10010000	DLE	Data link escape	Special control character

SYN	SYN	SOH	Heading	STX	Text	ETX	BCC

FIGURE 11-5
Typical Message Format for Character-Oriented Protocol

the last eight bits received. If they do not match the bits of the SYN character, the receiver accepts the next bit, rejects the previous high-order bit, and again checks the last eight bits received for a SYN character. This is repeated after each clock period and bit received until a SYN character is recognized. Once a SYN character is detected, the receiver has framed an 8-bit character. From here on, the receiver counts every eight bits and accepts them as the next character. Usually, the receiver checks two consecutive SYN characters to remove any doubt that the first one did not occur as a result of a noise signal on the line. Moreover, when the transmitter is idle and does not have any message characters to send, it sends a continuous string of SYN characters. The receiver recognizes these characters as a condition for synchronizing the line and goes into a synchronous idle state. In this state, the two units maintain bit and character synchronism even though no meaningful information is communicated.

Messages are transmitted through the data link with an established format consisting of a heading field, a text field, and an error checking field. A typical message format for the character-oriented protocol is shown in Figure 11-5. The two SYN characters assure proper synchronization at the start of the message. Following the SYN characters is the heading, which starts with an SOH (start of heading) character. The heading consists of an address and control information. The STX (start of text) character terminates the heading and signifies the beginning of text transmission. The text portion of the message is variable in length and may contain any ASCII characters except the communication control characters. The text field is terminated with the ETX (end of text) character. The last field is a block check character (BCC) used for error checking. It is usually either a longitudinal redundancy check (LRC) or a cyclic redundancy check (CRC). The LRC is an 8-bit parity over all the characters of the message in the frame. It is the accumulation of the exclusive-OR of all transmitted characters. The CRC is a polynomial code obtained from the message bits by passing them through a feedback shift register containing a number of exclusive-OR gates. This type of code is suitable for detecting burst errors occurring in the communication channel.

Bit-Oriented Protocol

The character-oriented protocol was originally developed to communicate with keyboard, printer, and display devices that use alphanumeric characters exclusively. As the data communication field expanded, it became necessary to transmit binary information that is not ASCII text. This happens, for example, when two remote computers send binary programs to each other over a communication channel. An arbitrary bit pattern in the text field becomes a problem in the character-oriented method. This is because any 8-bit pattern that corresponds to one of the communication control characters will be interpreted erroneously by the receiver.

Flag 01111110	Address 8 bits	Control 8 bits	Information Any number of bits	Frame check 16 bits	Flag 01111110

FIGURE 11-6
Frame Format for Bit-Oriented Protocol

The protocol that has been mostly used to solve this problem is the bit-oriented protocol.

The bit-oriented protocol is independent of any particular code. It allows the transmission of a serial bit stream of any length without the implication of character boundaries. Messages are organized in a specific format within a frame. In addition to the information field, a frame contains address, control, and error checking fields. The frame boundaries are determined from a special 8-bit number called a flag.

The frame format of the bit-oriented protocol is shown in Figure 11-6. A frame starts with the 8-bit flag 01111110 followed by an address and control sequence. The information field is not restricted in format or content and can be of any length. The frame check field is a CRC (cyclic redundancy check) sequence used for detecting errors in transmission. The ending flag indicates to the receiver station that the 16 bits just received are the CRC bits. The ending flag can be followed by another frame or a sequence of flags or a sequence of consecutive 1's. When two frames follow each other, the intervening flag is simultaneously the ending flag of the first frame and the beginning flag of the next frame. If no information is exchanged, the transmitter sends a series of flags to keep the line in the active state. The line is said to be in the idle state with the occurrence of 15 or more consecutive 1's. Frames with certain control messages are sent without information fields. A frame must have a minimum of 32 bits between two flags to accommodate the address, control, and the frame check fields. The maximum length depends on the condition of the communication channel and its ability to transmit long messages error-free.

To prevent a flag from occurring in the middle of a frame, the bit-oriented protocol uses a method called *zero insertion*. This requires that a 0 be inserted by the transmitting station after any succession of five consecutive 1's. The receiver always removes a 0 that follows a succession of five 1's. In this way, the bit pattern 01111110 is transmitted as 011111010 when it does not signify a flag. The received sequence is restored to its original value by the removal of the 0 following the five 1's. As a consequence, no pattern 01111110 is ever transmitted between the beginning and ending flags.

11-4 MODES OF TRANSFER

Binary information received from an external device is usually stored in memory for later processing. Information transferred from the central computer into an external device originates in the memory unit. The CPU merely executes the I/O

instructions and may accept the data temporarily, but the ultimate source or destination is the memory unit. Data transfer between the central computer and I/O devices may be handled in a variety of modes. Some modes use the CPU as an intermediate path; others transfer the data directly to and from the memory unit. Data transfer to and from peripherals may be handled in one of four possible modes:

1. Data transfer under program control.
2. Interrupt initiated data transfer.
3. Direct memory access (DMA) transfer.
4. Transfer through an I/O processor (IOP).

Program controlled operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from the CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the external device.

In the program controlled transfer, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it keeps the processor busy needlessly. It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. The CPU can proceed to execute another program. The interface, meanwhile, keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the data transfer, and then returns to the task it was originally performing.

Transfer of data under program control is through the I/O bus and between the CPU and a peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred, and then proceeds to execute other tasks. When the transfer is made, the interface requests memory cycles through the memory bus. When the request is granted by the memory controller, the interface transfers the data directly into memory. The CPU merely delays its operation to allow the direct memory I/O transfer. Since peripheral speed is usually slower than processor speed, I/O memory transfers are infrequent compared to processor access to memory. DMA transfer is discussed in more detail in Section 11-6.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP. I/O processors are presented in Section 11-7.

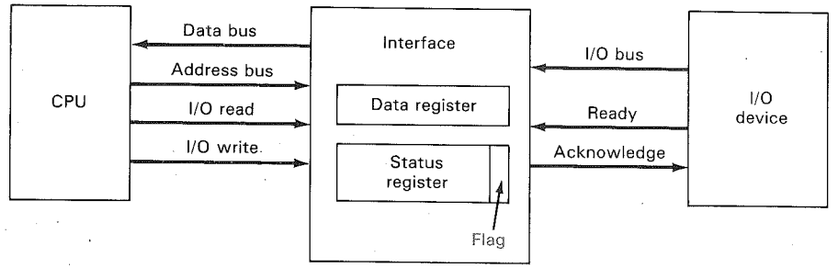


FIGURE 11-7
Data Transfer from Device to CPU

Example of Program Control Transfer

A simple example of data transfer from an I/O device through an interface into the CPU is shown in Figure 11-7. The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables the ready line. The interface accepts the byte into its data register and enables the acknowledge line. The interface sets a bit in the status register which we will refer to as a *flag*. The device can now disable the ready line but it will not transfer another byte until the acknowledge line is disabled by the interface. This is according to the handshaking procedure established in Section 11-2.

Under program control, the CPU must check the flag to determine if there is a new byte in the interface data register. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 either by the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the acknowledge line and the device can transfer the next data byte.

A flowchart of the program that must be written for the transfer is shown in Figure 11-8. The flowchart assumes that the device is sending a sequence of bytes which must be stored in memory. The program continually examines the status of the interface until the flag is set to 1. Each byte is brought into the CPU and transferred to memory until all of the data have been transferred.

The program control data transfer is used only in small computers or in systems that are dedicated to monitor a device continuously. The difference in information transfer rate between the CPU and the I/O device makes this type of transfer inefficient. To see why this is inefficient, consider a typical computer that can execute the instructions to read the status register and check the flag in 1 μsec. Assume that the input device transfers its data at an average rate of 100 bytes per second. This is equivalent to one byte every 10,000 μsec. This means that the CPU will check the flag 10,000 times between each transfer. The CPU is wasting time while checking the flag instead of doing some other useful processing task.

Interrupt Initiated Transfer

An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program, it does not check the

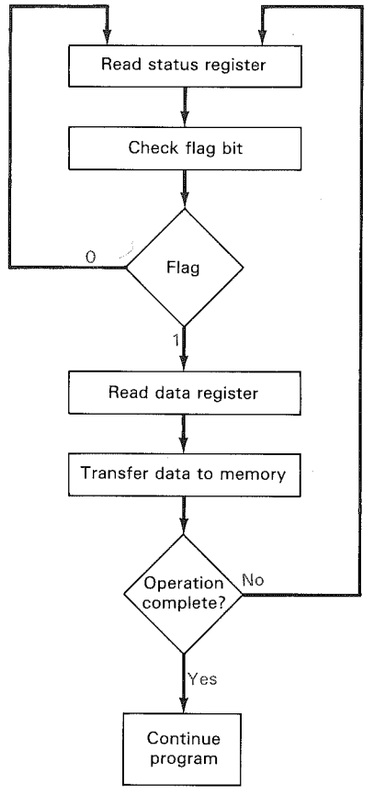


FIGURE 11-8
Flowchart for CPU Program to Input Data

flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from one unit to another. In principle, there are two methods for accomplishing this. One is called *vectored interrupt* and the other *nonvectored interrupt*. In a nonvectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the *vector address*. In some computers the vector address is the

first address of the service routine. In other computers, the vector address is an address that points to a location in memory where the first address of the service routine is stored. The vectored interrupt procedure is presented in Section 9-9 in conjunction with Figure 9-8.

11-5 PRIORITY INTERRUPT

A typical computer has a number of I/O devices attached to it with each device being able to originate an interrupt request. The first task of the interrupt system is to identify the source of the interrupt. There is also the possibility that several sources will request service simultaneously. In this case, the system must also decide which device to service first.

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher priority interrupt levels are assigned to requests which, if delayed or interrupted, could have serious consequences. Devices with high speed transfers such as magnetic disks are given high priority and slow devices such as keyboards receive the lowest priority. When two devices interrupt the computer at the same time, the computer services the device with the higher priority first.

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest priority source by software means. In this method, there is one common branch address for all interrupts. The program that takes care of interrupts begins at the branch address and polls the interrupt sources in sequence. The order in which they are tested determines the priority of each interrupt. The highest priority source is tested first and if its interrupt signal is on, control branches to a service routine for this source. Otherwise, the next lower priority source is tested, and so on. Thus, the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The particular service routine reached belongs to the highest priority device among all devices that interrupted the computer. The disadvantage of the software method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation, a hardware priority interrupt unit can be used to speed up the operation.

A hardware priority interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector address to access directly its own service routine. Thus, no polling is required because all the decisions are established by the hardware priority interrupt unit. The hardware priority function can be established either by a serial or parallel connection of interrupt lines. The serial connection is also known as the daisy chain method.

Daisy Chain Priority

The daisy chain method of establishing priority consists of a serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower priority devices in descending priority order to the lowest priority device, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Figure 11-9. The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU. When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU. This is equivalent to a negative-logic OR operation. The CPU responds to an interrupt request by enabling the interrupt acknowledge line. This signal is received by device 1 at its *PI* (priority in) input. The acknowledge signal passes on to the next device through the *PO* (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the *PO* output. It then proceeds to insert its own interrupt vector address (*VAD*) into the data bus for the CPU to use during the interrupt cycle.

A device with a 0 in its *PI* input generates a 0 in its *PO* output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its *PI* input will intercept the acknowledge signal by placing a 0 in its *PO* output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its *PO* output. Thus, the device with $PI = 1$ and $PO = 0$ is the one with the highest priority that is requesting an interrupt and this device places its *VAD* on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Figure 11-10 shows the internal logic that must be included within each device when connected in the daisy chain scheme. The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an

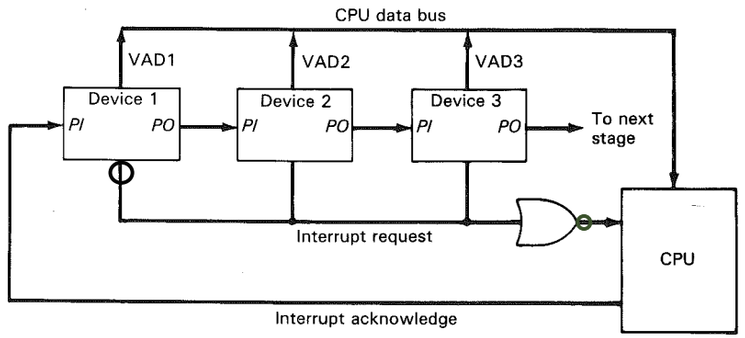
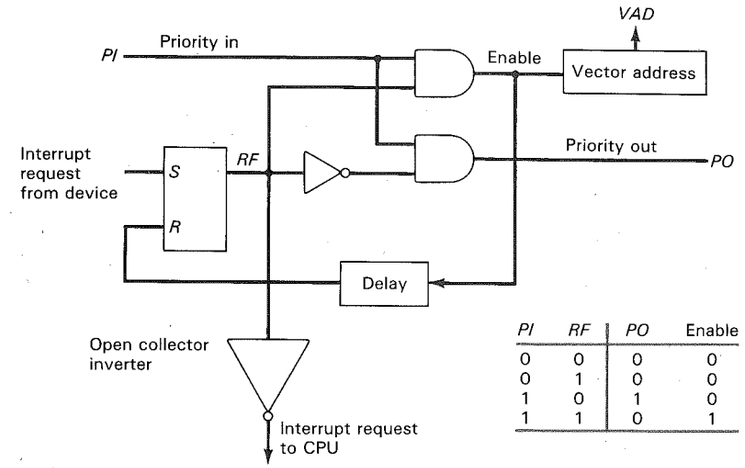


FIGURE 11-9
Daisy Chain Priority Interrupt



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

FIGURE 11-10
One Stage of the Daisy Chain Priority Arrangement

open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If $PI = 0$, both *PO* and the enable line to *VAD* are equal to 0, irrespective of the value of RF. If $PI = 1$ and $RF = 0$, then $PO = 1$ and the vector address is disabled. This condition passes the acknowledge signal to the next device through *PO*. The device is active when $PI = 1$ and $RF = 1$. This condition places a 0 in *PO* and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

Parallel Priority Hardware

The parallel priority interrupt method uses a register with bits set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register to control the status of each interrupt request. The mask register can be programmed to disable lower priority interrupts while a higher priority device is being serviced. It can also provide a facility that allows a high priority device to interrupt the CPU while a lower priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Figure 11-11. It consists of an interrupt register with individual bits set by external conditions and cleared by program instructions. Interrupt input 3 has the highest priority and input 0 has the lowest priority. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way, an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address which is transferred to the CPU via the data bus. Output *V* of the encoder is set to 1 if an interrupt that is not masked has occurred. This provides the interrupt signal for the CPU.

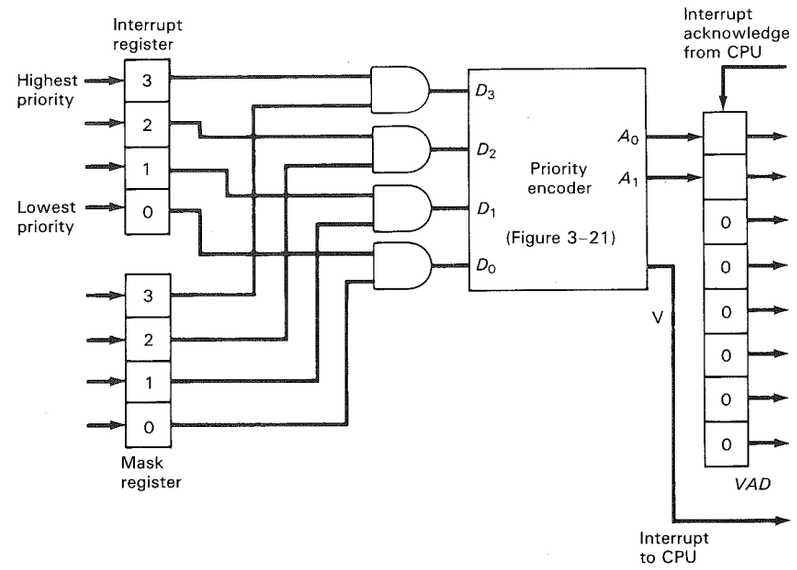


FIGURE 11-11
Priority Interrupt Hardware

The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that, if two or more inputs occur at the same time, the input having the highest priority takes precedence. The circuit of a four-input priority encoder can be found in Section 3-6 and its truth table is listed in Table 3-8. Input D_3 has the highest priority, and so regardless of the values of other inputs, when this input is 1, the output is $A_1A_0 = 11$. D_2 has the next priority level. The output is 10 if $D_2 = 1$ provided that $D_3 = 0$, regardless of the values of the other two lower priority inputs. The output is 01 when $D_1 = 1$ provided the two higher priority inputs are equal to 0; and so on down the priority levels. The interrupt output labeled V is equal to 1 when one or more inputs are equal to 1. If all inputs are 0, V is 0 and the other two outputs of the encoder are not used. This is because the vector address is not transferred to the CPU when $V = 0$.

The output of the priority encoder is used to form part of the vector address for the interrupt source. The other bits of the vector address can be assigned any values. For example, the vector address can be formed by appending six zeros to the outputs of the encoder. With this choice the interrupt vector for the four I/O devices are assigned the 8-bit binary numbers equivalent to decimal 0, 1, 2, and 3.

11-6 DIRECT MEMORY ACCESS (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer, the CPU is idle and has no control of the memory buses.

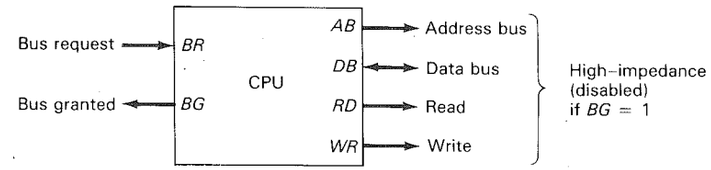


FIGURE 11-12
CPU Bus Control Signals

A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Figure 11-12 shows two control signals in a CPU that facilitate the DMA transfer. The *bus request* (*BR*) input is used by the DMA controller to request from the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of its present instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state. After this is done, the CPU activates the *bus granted* (*BG*) output to inform the external DMA that it can take control of the buses. As long as the *BG* line is active, the CPU is idle and the buses are disabled. When the bus request input is disabled by the DMA, the CPU returns to its normal operation, disables the bus granted output, and takes control of the buses.

When the bus granted line is enabled, the external DMA controller takes control of the bus system to communicate directly with the memory. The transfer can be made for an entire block of memory words, suspending the CPU operation until the entire block is transferred. This is referred to as *burst transfer*. The transfer can be made one word at a time between CPU instruction executions. Such transfer is called *single cycle* or *cycle stealing*. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to steal one memory cycle.

DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines are used for direct communication with the memory. The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Figure 11-13 shows the block diagram of a typical DMA controller. The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the *DS* (DMA select) and *RS* (register select) inputs. The *RD* (read) and *WR* (write) inputs are bidirectional. When the *BG* (bus granted) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When $BG = 1$, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus

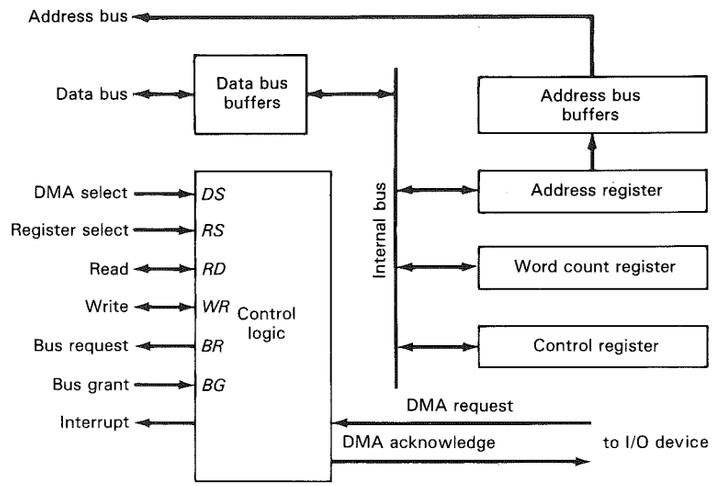


FIGURE 11-13
Block Diagram of a DMA Controller

and activating the *RD* or *WR* control. The DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus, the CPU can read from or write to the DMA registers under program control via the data bus.

The DMA is first initialized by the CPU. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transferred. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block.
3. Control to specify the mode of transfer such as read or write.
4. A control to start the DMA transfer.

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once

the DMA is initialized, the microprocessor stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

DMA Transfer

The position of the DMA controller among the other components in a computer system is illustrated in Figure 11-14. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address which activates the *DS* and *RS* lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control bit, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the *BR* line, informing the CPU to relinquish the buses. The CPU responds with its *BG* line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register onto the address bus, initiates the *RD* or *WR* signal, and sends a DMA acknowledge to the peripheral device.

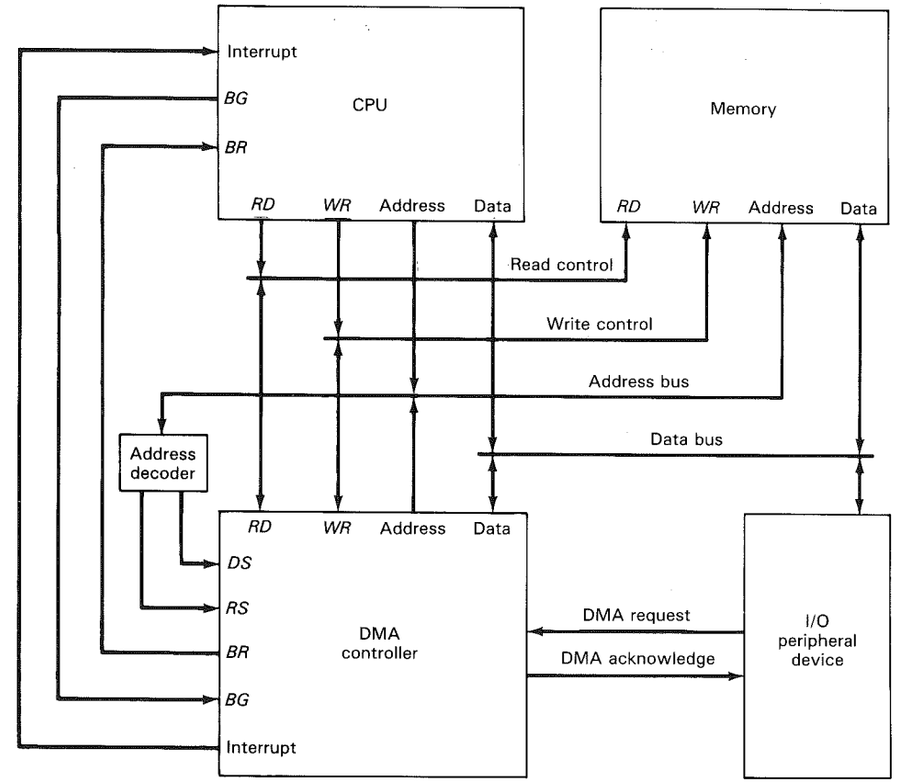


FIGURE 11-14
DMA Transfer in a Computer System

When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus, the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.

For each word that is transferred, the DMA increments its address register and decrements its word count register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high speed device, the line will be activated as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case, the DMA disables the bus request line so the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

If the word count reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count register. The zero value of this register indicates that all the words were successfully transferred. The CPU can read this register at any time, as well check the number of words already transferred.

A DMA controller may have more than one channel. In this case, each channel has a request and acknowledge pair of control signals that are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority.

DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of DMA transfer.

11-7 MULTIPLE PROCESSOR SYSTEMS

Instead of having each interface communicate with the CPU, a computer may incorporate one or more external processors and assign them the task of communicating directly with all I/O devices. An input-ouput processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. In this configuration, the computer system can be divided into a memory unit, and a number of processors comprised of the CPU and one or more IOPs. Each IOP takes care of input and output tasks, relieving the CPU of the housekeeping chores involved in I/O transfers. A processor that communicates with remote terminals over telephone and other communication media in a serial fashion is called a data communication processor (DCP).

A multiple processor system is an interconnection of two or more processors sharing a common memory and input-output equipment. The term processor includes CPU, IOP, DCP, and any other type of processor that the system may have. A multiple processor system may have multiple CPUs in addition to other processors.

Although large scale computers include two or more CPUs in their overall system configuration, it is the emergence of the microprocessor, or a CPU on a chip, that has been the major motivation for multiple processor systems. The fact that microprocessors take very little physical space and are very inexpensive brings about the feasibility of interconnecting a large number of microprocessors into a composite system. Very large scale integration circuit technology has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

The benefits derived from multiple processor organization is an improved system performance. It is achieved through partitioning an overall function into a number of tasks that each processor can handle individually. System tasks may be allocated to special purpose processors whose design is optimized to perform certain types of processing efficiently. An example is a computer system where one processor performs the computations for an industrial process control while others monitor and control the various parameters, such as temperature and flow rate. Another example is a computer where one processor performs high speed floating-point arithmetic computations and another takes care of routine data processing tasks. Performance also can be improved if a program can be decomposed into parallel executable tasks. The system function can be distributed among parallel concurrently-executing processors operating in parallel to reduce the overall execution time.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to operate correctly with, perhaps, some loss in efficiency.

Input-Output Processor (IOP)

The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks such as arithmetic, logic, branching, and code translation.

The block diagram of a computer with two processors is shown in Figure 11-15. The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit. The CPU is usually assigned the task of initiating the I/O program. From then on, the IOP

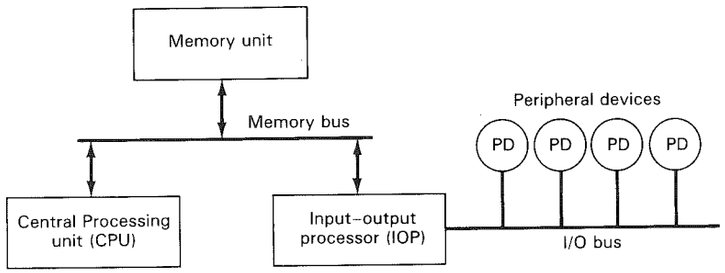


FIGURE 11-15
Block Diagram of a Computer with I/O Processor

operates independent of the CPU and continues to transfer data from external devices and memory.

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data is assembled into a memory word, it is transferred from IOP directly into memory by stealing one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity.

The communication between the IOP and the devices attached to it is similar to the program control method of transfer. Communication with the memory is similar to the direct memory access method. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In very large scale computers, each processor is independent of all others and any one processor can initiate an operation. In most computer systems, the CPU is the master, while the IOP is a slave processor. The CPU is assigned the task of initiating all operations but I/O instructions are executed in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. When an I/O operation is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

Instructions that are read from memory by an IOP are sometimes called *commands*, to distinguish them from instructions that are read by the CPU. An instruction and a command have similar functions. Commands are prepared by experienced programmers and are stored in memory. The command words constitute the program for the IOP. The CPU informs the IOP where to find the commands in memory when it is time to execute the I/O program.

The communication between CPU and IOP may take different forms depending on the particular computer considered. In most cases, the memory unit acts as a message center where each processor leaves information for the other. To appreciate the operation of a typical IOP, we will illustrate, by a specific example, the method by which the CPU and IOP communicate. This is a simplified example that omits many operating details in order to provide an overview of basic concepts.

The sequence of operations may be carried out as shown in the flowchart of Figure 11-16. The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program.

The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request

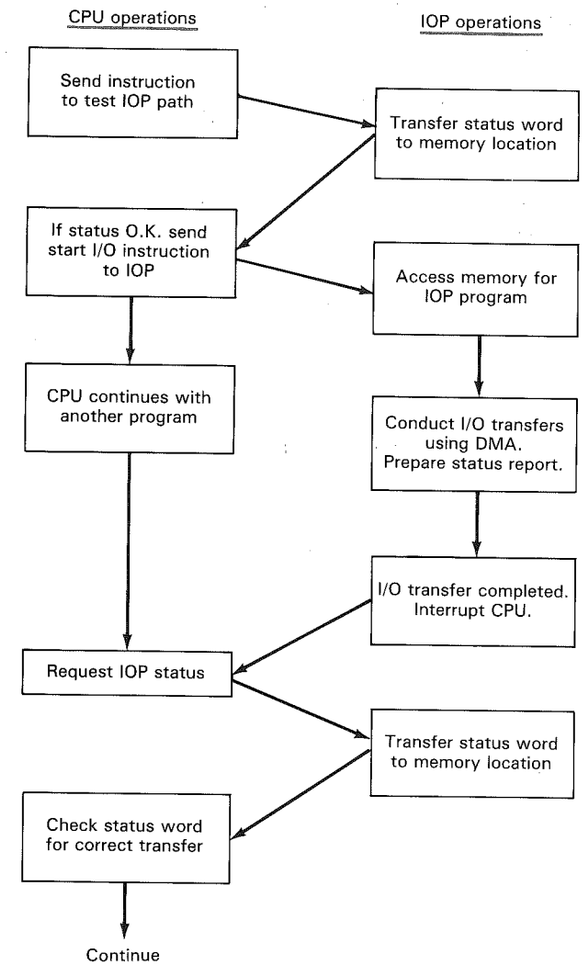


FIGURE 11-16
CPU-IOP Communication

to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or if any errors occurred during the transfer. From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily, without errors.

The IOP takes care of all data transfers between several I/O units and memory while the CPU is processing another program. The IOP and CPU compete for the use of memory so the number of devices that can be in operation is limited by the access time of the memory. It is not possible to saturate the memory by I/O devices in most systems, as the speed of most devices is much slower than the CPU. However, some very fast units, such as magnetic disks, can use an appreciable number of the available memory cycles. In that case, the speed of the CPU may deteriorate because it will often have to wait for the IOP to conduct memory transfers.

Interconnection Between Processors

The components that form a multiple processor system are CPUs, IOPs connected to input and output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations depending on the number of transfer paths that are available between the processors and memory.

One possible configuration employs a common bus system with all the processors connected through a common path to memory. A time-shared common bus for four processors is shown in Figure 11-17. Only one processor can communicate with the memory at any given time. Transfer operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the memory to initiate the transfer. The system may exhibit memory access conflicts since one common bus is shared by all processors. Memory contentions must be resolved with a bus controller that establishes priorities among the requesting processors.

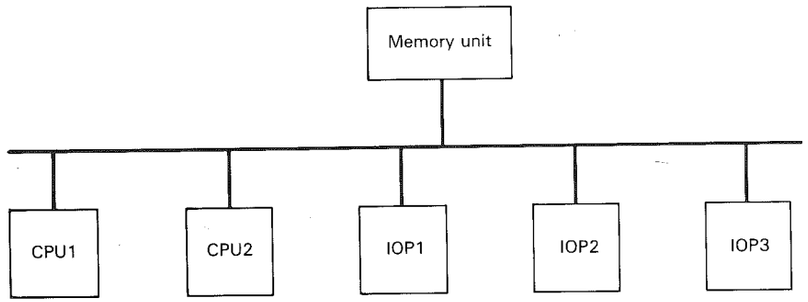


FIGURE 11-17
Time Shared Common Bus Multiple Processor Organization

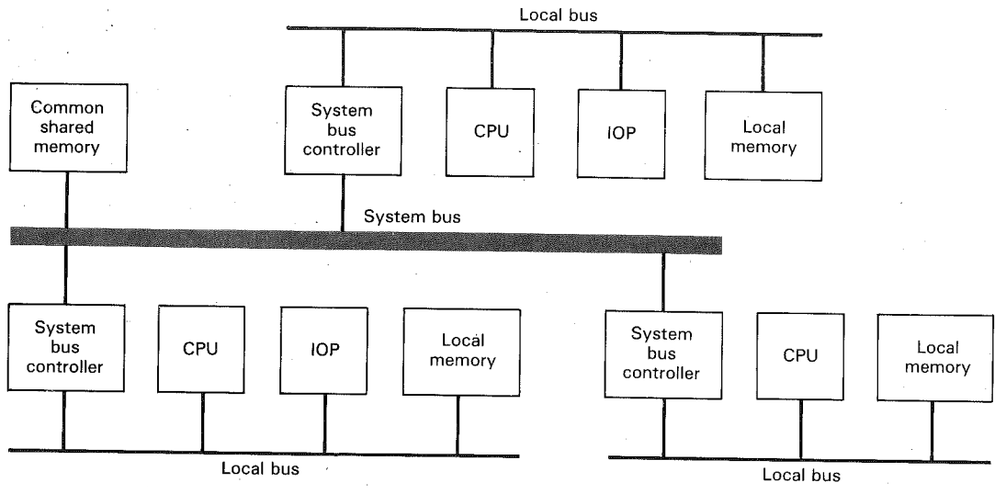


FIGURE 11-18
Dual-Bus Structure for Multiple Processors

A single common bus is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. The processors in the system can be kept busy more often through the implementation of a dual-bus structure as depicted in Figure 11-18. Here we have a number of local buses, each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to a system bus. The I/O devices attached to it may be made available to all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time. The other processors are kept busy communicating with their local memory and I/O devices.

Although the system shown in Figure 11-18 qualifies as a multiple processor system it can be classified more correctly as a multiple computer system. This is because a CPU, IOP, and memory, when connected together with a local bus, constitute a computer system in their own right. This type of multiple processor organization is the one most commonly employed in the design of multiple microprocessor systems.

Interprocessor Communication

The various processors in a multiple processor system must be provided with a facility for communicating with each other. The most common procedure is to set aside a portion of memory which is accessible to all processors. The primary use of the common memory is to act as a message center similar to a mailbox, where each processor can leave messages for other processors and pick up messages intended for it.

The sending processor structures a request, a message, or a procedure, and

places it in the memory mailbox. Status bits residing in common memory are generally used to indicate the condition of the mailbox, whether it contains meaningful information and for which processor intended. The receiving processor can check the mailbox periodically to determine if there are valid messages for it. The response time of this procedure can be time consuming, since a processor will recognize a request only when it performs the polling of messages. A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal. This can be accomplished through a software initiated interprocessor interrupt. It is done by an instruction in the program of one processor which when executed, produces an external interrupt condition in a second processor. This alerts the interrupted processor of the fact that a new message was inserted by the interrupting processor.

In addition to shared memory, a multiple processor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk. A communication path between two CPUs also can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an I/O device so that messages can be transferred through the I/O path.

Each processor in a multiple processor system must request access to common memory and other shared resources through a common bus system. If no other processor is currently using the common bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently using the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must be performed to resolve this multiple contention for the shared resources. Arbitration procedures service all processor requests on the basis of established priority. The bus arbitration technique bears a strong resemblance to the interrupt priority logic discussed in Section 11-5.

A multiple processor system is considered to be a *tightly coupled* system. This is characteristic of a system that has all its major components (such as CPU, IOP, and I/O devices) in close proximity. Computers that are interconnected with each other by means of remote communication lines form a *computer network*. The computers in the network may be close to each other or in geographically remote locations. Synchronous serial transfer is employed to send messages between the computers. Data and control are transmitted in packets and each packet follows a prescribed communication protocol. The distinction that is made between a tightly coupled computer system and a computer network is that the former uses shared memory to communicate between the processors. The computers in a network communicate with each other in a serial fashion through a communication medium.

REFERENCES

1. MANO, M. M. *Computer System Architecture.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1982.
2. LIPPIATT, A. G. AND WRIGHT, G. L. *The Architecture of Small Computer Systems.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1985.

3. LIU, Y. C. AND GIBSON, G. A. *Microcomputer Systems: The 8086/8088 Family.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1986.
4. HAMACHER, V. C., VRANESIC, Z. G. AND ZAKY, S. G. *Computer Organization.* 2nd ed. New York: McGraw-Hill, 1984.
5. HWANG, K. AND BRIGGS, F. A. *Computer Architecture and Parallel Processing.* New York: McGraw-Hill, 1984.

PROBLEMS

11-1 The addresses assigned to the four registers of the I/O interface of Figure 11-2 are equal to the binary equivalent of 12, 13, 14, and 15. Show the external circuit that must be connected between an 8-bit I/O address from the CPU and the *CS*, *RS*0, and *RS*1 inputs of the interface.

11-2 Six interface units of the type shown in Figure 11-2 are connected to a CPU that uses an I/O address of eight bits. Each one of the six chip select (*CS*) inputs is connected to a different address line. Thus, the high-order address line is connected to the *CS* input of the first interface unit and the sixth address line is connected to the *CS* input of the sixth interface unit. The two low-order address lines are connected to the *RS*1 and *RS*0 of all six interface units. Determine the 8-bit address of each register in each interface. (Total of 24 addresses.)

11-3 Derive a timing diagram and a sequence of events flow chart (as in Figure 11-3) for a transfer from the I/O device into the interface unit assuming that the interface initiates the transfer with a *request* handshake line. The other handshake is a *data ready* line from the device to the interface.

11-4 A commercial interface unit uses different names for the handshake lines associated with the transfer of data from the I/O device into the interface unit. The interface input handshake line is labeled STB (strobe), and the interface output handshake line is labeled IBF (input buffer full). A low-level signal on STB loads data from the I/O bus into the interface data register. A high-level signal on IBF indicates that the data item has been accepted by the interface. IBF goes low after an I/O read signal from the CPU when it reads the contents of the data register.

(a) Draw a block diagram showing the CPU, the interface, and the I/O device along with the pertinent interconnections between the three units.
(b) Draw a timing diagram for the handshaking transfer.
(c) Obtain a sequence of events flowchart for the transfer from the device to the interface and from the interface to the CPU.

11-5 How many characters per second can be transmitted over a 1200-baud line in each of the following modes? (Assume a character code of eight bits.)

(a) Synchronous serial transmission.
(b) Asynchronous serial transmission with two stop bits.
(c) Asynchronous serial transmission with one stop bit.

11-6 The address of a terminal connected to a data communication processor consists of two letters of the alphabet or a letter followed by one of the 10 numerals. How many different addresses can be formulated?

11-7 Sketch the timing diagram of the eleven bits (similar to Figure 11-4) that are transmitted over an asynchronous serial communication line when the ASCII letter C is transmitted with even parity.

11-8 What is the difference between synchronous and asynchronous serial transfer of information?

11-9 What is the minimum number of bits that a frame must have in the bit-oriented protocol?

11-10 Show how the zero insertion works in the bit-oriented protocol when a zero followed by the 10 bits that represent the binary equivalent of 1023 is transmitted.

11-11 Show the block diagram (similar to Figure 11-7) for the data transfer from the CPU to the interface and then to the I/O device. Determine a procedure for setting and clearing the flag bit.

11-12 Using the configuration established in Problem 11-11, obtain a flowchart (similar to Figure 11-8) for the CPU program to output data.

11-13 What is the basic advantage of using interrupt initiated data transfer over transfer under program control without an interrupt?

11-14 What happens in the daisy chain priority interrupt shown in Figure 11-9 when device 1 requests an interrupt after device 2 has sent an interrupt request to the CPU but before the CPU responds with the interrupt acknowledge?

11-15 Consider a computer without priority interrupt hardware. Any one of many sources can interrupt the computer and any interrupt request results in storing the return address and branching to a common interrupt routine. Explain how a priority can be established in the interrupt service program.

11-16 What should be done in Figure 11-11 to make the four *VAD* values equal to the binary equivalent of 76, 77, 78, and 79?

11-17 Design a parallel priority interrupt hardware for a system with eight interrupt sources.

11-18 (a) Obtain the truth table of an 8 × 3 priority encoder.
(b) The three outputs x, y, z, from the priority encoder are used to provide an 8-bit vector address in the form 101xyz00. List the eight addresses starting from the one with the highest priority.

11-19 Why are the read and write control lines in a DMA controller bidirectional? Under what condition and for what purpose are they used as inputs? Under what condition and for what purpose are they used as outputs?

11-20 It is necessary to transfer 256 words from a magnetic disk to a memory section starting from address 1230. The transfer is by means of the DMA as shown in Figure 11-14.
(a) Give the initial values that the CPU must transfer to the DMA controller.
(b) Give the step-by-step account of the actions taken during the input of the first two words.

11-21 What is the purpose of the system bus controller in Figure 11-18? Explain how the system can be designed to distinguish between references to local memory and references to commonly shared memory.

12

MEMORY MANAGEMENT

12-1 MEMORY HIERARCHY

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general purpose computers would run more efficiently if they are equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer installations accumulate and continue to accumulate large amounts of information. Not all accumulated information is needed by the processor at the same time. Therefore, it is more economical to use low cost storage devices to serve as backup for storing the information that is not currently used by the CPU. The memory unit that communicates directly with the CPU is called the *main memory*. Devices that provide backup storage are called *auxiliary memory*. The most common auxiliary memory devices used in computer systems are magnetic disks and magnetic tapes. Only programs and data currently needed by the processor reside in main memory. All other binary information is stored in auxiliary memory and transferred to main memory on a demand basis.

The total memory capacity of a computer can be visualized as being a hierarchy of components. The memory hierarchy system consists of all storage devices employed in a computer system from the slow but high capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed processing logic. Figure 12-1 illustrates the components

in a typical memory hierarchy. At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O processor. When programs not residing in main memory are needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data.

A special very high speed memory is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory included in Figure 12-1 is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. Processor logic is usually faster than main memory access time with the result that processing speed is mostly limited by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small memory between the CPU and main memory with an access time close to processor logic propagation delays. This type of memory is called a *cache* memory. It is used to store segments of programs, currently being executed in the CPU, and temporary data, frequently needed in the calculations. By making programs and data available at a rapid rate, it is possible to increase the performance rate of the processor. The organization of cache memories is presented in Section 12-4.

In a computer system where the demand for service is high, it is customary to run all programs in one of two modes: a *batch mode* or *time-sharing mode*. In the batch mode, the programs are prepared off-line without the direct use of the computer. An operator loads all programs into the computer where they are executed one at a time. The operator retrieves the printed output and returns it to the user. What makes the batch mode efficient is the fact that programs can be fed into the computer as fast as they can be processed. In this way it is ensured that the computer is busy processing information most of the time.

In the time-sharing mode, many users communicate with the computer via remote interactive terminals. Because of slow human response compared to computer speed, the computer can respond to multiple users at the same time. This is accomplished by having many programs reside in memory while the system allocates a time-slice to each program for execution in the CPU.

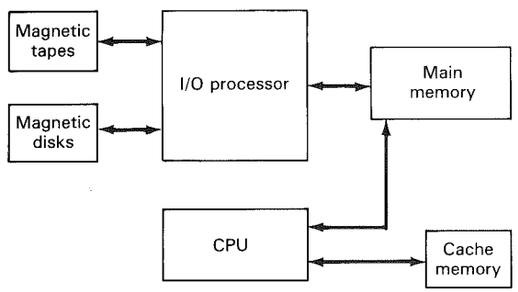


FIGURE 12-1
Memory Hierarchy in a Computer System

A major concept common to both batch and time-sharing modes is their use of *multiprogramming*. Multiprogramming refers to the existence of many programs in different parts of main memory at the same time. Thus, it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose a program is being executed in the CPU and an I/O transfer is required. The CPU initiates the I/O processor to start executing the transfer. This leaves the CPU free to execute another program. In a multiprogramming system, when one program is waiting for input or output transfer, there is another program ready to use the CPU.

With multiprogramming, the need arises for running partial programs, for varying the amount of main memory in use by a given program, and for moving programs around the memory hierarchy. Computer programs are sometimes too long to be accommodated in the total space available in main memory. Moreover, a computer system uses many programs and all the programs cannot reside in main memory at all times. A program with its data normally resides in auxiliary memory. When the program or a segment of the program is to be executed, it is transferred to main memory to be executed by the CPU. Thus, one may think of auxiliary memory as containing the totality of information stored in a computer system. It is the task of the operating system to maintain in main memory a portion of this information that is currently active. The part of the computer system that supervises the flow of information between auxiliary memory and main memory is called the *memory management system*. The hardware for a memory management system is presented in Section 12-5.

Other topics covered in this chapter are pipeline processing and associative memory. Pipeline processing is a technique for speeding the operations in a computer system. The associative memory is a content addressable type memory that is often used as part of a memory management system.

12-2 PIPELINE PROCESSING

Pipeline is a technique of decomposing a sequential process into suboperations with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments. The name pipeline implies a flow of information analogous to an industrial assembly line. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computations is made possible by associating a register with each segment in the pipeline. The registers provide isolation between segments so that all can operate on separate data simultaneously.

The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

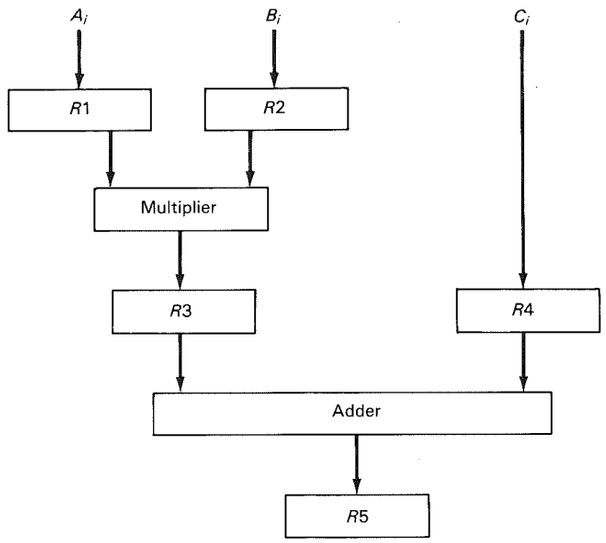


FIGURE 12-2
Example of Pipeline Processing

$$A_i * B_i + C_i \qquad \text{for } i = 1,2,3, \ldots, 7$$

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Figure 12-2. $R1$ through $R5$ are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits (see Section 3-4). The suboperations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

The five registers are loaded with new data every clock pulse. The effect of each clock pulse is shown in Table 12-1. It takes three clock pulses to fill up the pipe and retrieve the first output from $R5$. From there on, each clock pulse produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock pulses must continue until the last output emerges out of the pipeline.

Note that a pipeline processor performs simultaneous operations in each segment. No matter how many segments there are in the system, once the pipeline is full, it takes only one clock pulse to obtain an output, regardless of how many steps are required to execute the entire process. If the time it takes to process the suboperation in each segment is an interval t, and if there are k segments, then each complete computation is executed in $k \times t$ intervals. However, since successive

TABLE 12-1
Content of Registers in Pipeline Example

Pulse number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

operations are overlapped in the pipeline, the results are always delivered at every t interval after a setup time of $k \times t$ that it takes to fill up the pipeline.

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The procedure is efficient only in those applications where the same computation must be repeated on a stream of input data. Pipeline data processing has been applied mostly to floating-point arithmetic operations.

Instruction Pipeline

Pipeline processing can occur not only in the data stream but in the instruction stream as well. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. The only digression associated with such a scheme is that one of the instructions may cause a branch out of sequence. In that case, the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

An instruction pipeline can be implemented by means of a first-in first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack. Whenever the execute phase is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first-in first-out basis. Thus, an instruction stream can be placed in a queue, to wait for decoding and execution by the processor.

The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions. Whenever there is space in the FIFO buffer, the CPU executes an instruction fetch phase. The buffer acts as a queue from which control extracts instructions for execution. If the CPU executes an instruction that transfers control of a location out of normal sequence, the buffer is reset and the pipeline is declared empty. Control then fetches the instruction from the branch address and begins to refill the buffer from the new

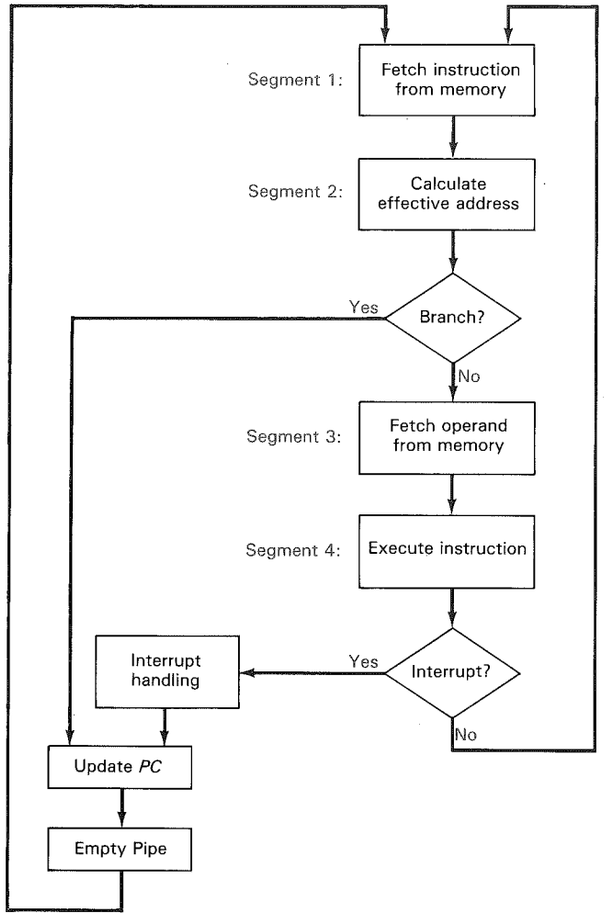


FIGURE 12-3
Four-segment CPU Pipeline

location. Often, branch instructions form a small fraction of the total number of instructions executed, so efficiency can be increased by this kind of instruction pipeline.

The instruction pipeline can be extended to include other phases in the CPU cycle. As an example, Figure 12-3 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline. While an instruction is being executed in the processor, the next instruction in sequence is busy fetching an operand from memory. The effective address may be calculated in a separate arithmetic circuit for the third instruction and, whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in a FIFO instruction queue. Thus, up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in process at the same time.

Once in a while, an instruction in the sequence may be a program control type that causes a branch out of normal sequence. In that case, the pending operations in the last two segments are completed and all information stored in the instruction

buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request, when acknowledged, will cause the pipeline to empty and start again from a new address value.

There are certain difficulties that will prevent the CPU pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory. Memory access conflicts are sometimes resolved by using a memory unit with multiple modules and storing the programs and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.

12-3 ASSOCIATIVE MEMORY

Many data processing applications require the search of items in a table stored in memory. The established way to search items in a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms have been developed to minimize the number of accesses while searching for an item in a random access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an *associative memory* or *content addressable memory*. This type of memory is accessed simultaneously and in parallel on the basis of data content. The memory is capable of finding an empty unused location to store a word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified. The memory locates the word that matches the specified content and marks it for reading.

Because of its organization, the associative memory is uniquely suited to do parallel searches by data association. Moreover, searches can be done on an entire word or on a specified field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

Hardware Organization

The block diagram of an associative memory is shown in Figure 12-4. It consists of a memory array and logic for m words of n bits per word. The argument register A has n bits, one for each bit of a word. The match register has m bits, one for

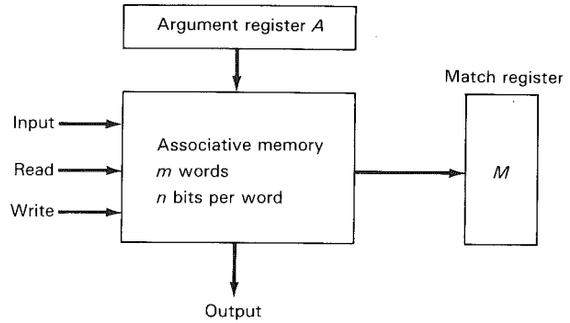


FIGURE 12-4
Block Diagram of Associative Memory

each word in memory. Each word is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. Reading is accomplished by a sequential access to memory for those words that have corresponding bits in the match register that have been set.

The internal organization of a typical binary cell consists of an elementary SR latch for storing the bit and the circuits for reading and writing as depicted in Figure 6-6 in Section 6-3. Additional circuits are needed for matching each word with the content of the argument register A. The match logic for each word can be derived from the comparison algorithm for two binary numbers. Let the bits of a word be designated by F_1, F_2, F_3, up to F_n, where n is the number of bits in the word. The content of the word will be equal to the argument in A if $A_i = F_i$ for $i = 1, 2, 3, \ldots, n$. Two bits are equal if both are equal to 1 or if both are equal to 0. The equality of two bits can be expressed by the exclusive-NOR function

$$X_i = A_iF_i + \overline{A}_i\overline{F}_i = \overline{A_i \oplus F_i}$$

where $X_i = 1$ if the pair of bits in position i are equal and $X_i = 0$ if the two bits are not equal.

For a word to be equal to the argument in A we must have all X_i variables equal to 1. This is the condition for setting the corresponding match bit M_j to 1. The Boolean function for this condition is

$$M_j = X_1X_2X_3 \ldots X_n$$

and constitute an AND operation of all pairs of matched bits in word j.

The circuit for matching two words of four bits in an associative memory is shown in Figure 12-5. A_1 through A_4 are the bits from the argument register that must be compared. Each bit of A is compared with the corresponding bit of the memory in the same significant position. BC stands for binary cell and the comparison is done with an exclusive-NOR gate. The outputs of all exclusive-NOR gates belonging to the same word go to the inputs of a common AND gate to generate the match signal M_j for $j = 1, 2, 3, \ldots m$, where m is the number of words in the memory.

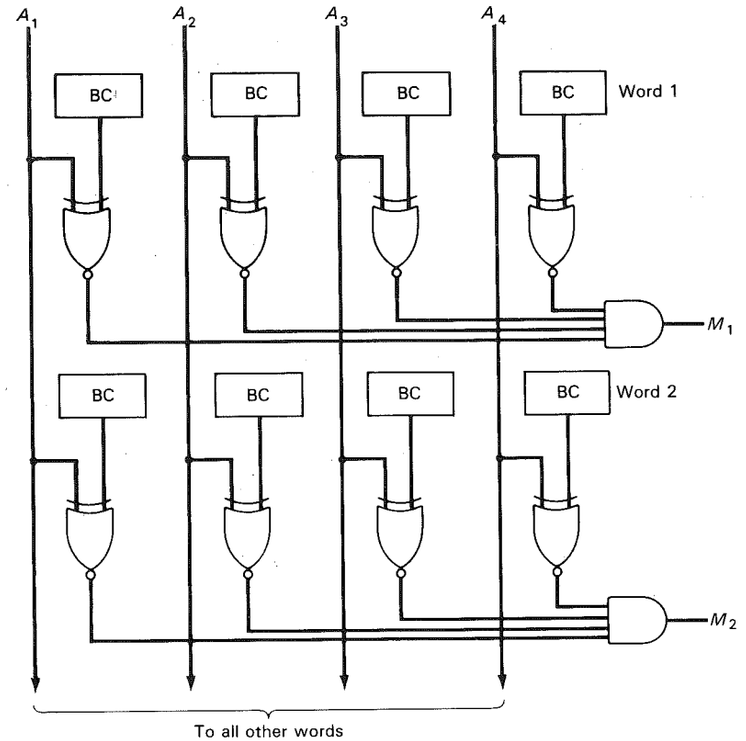


FIGURE 12-5
Match Logic for Two Words of Associative Memory

Read and Write Operations

If more than one word in the associative memory matches the argument value, all the matched words will have 1's in the corresponding bit position of the match register. It is then necessary to scan the bits of the match register one at a time. The matched words are read in sequence by applying a read signal to each word whose corresponding M bit is a 1.

In most applications, including applications in memory management hardware, the associative memory stores a table with no two identical items. In this case, only one word may match the argument value. By connecting output M_j directly to the read line of the same word (instead of the M register), the content of the matched word will be presented automatically at the outputs of the memory. Furthermore, if words having zero content are excluded, then an all zero output will indicate that no match occurred and that the searched item is not stored in memory.

An associative memory must have a write capability for storing the information to be searched. Since unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a *tag register*, must have as

many bits as there are words in memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by resetting its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location.

12-4 CACHE MEMORY

Analysis of a large number of typical programs has shown that references to memory at any given interval of time tends to be confined within a few localized areas in memory. This phenomenon is known as the *locality of reference*. The reason for this property may be understood considering that a typical computer program flows in a sequential fashion with program loops and subroutines encountered frequently. When a program loop is executed, the CPU repeatedly refers to the set of instructions in memory that constitute the loop. Every time a given subroutine is called, its set of instructions are fetched from memory. Thus loops and subroutines tend to localize the references to memory for fetching instructions. To a lesser degree, memory references to data also tend to be localized. Table lookup procedures repeatedly refer to a portion of memory where the table of items is stored. Iterative procedures refer to common memory locations and arrays of numbers confined within a local portion of memory. The result of all these observations is the locality of reference property which states that over a given interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly while the remaining memory is accessed relatively infrequently.

If the active portion of the program and data are placed in a fast, small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. As noted earlier, such a fast, small memory is referred to as a cache memory. It is placed between the CPU and main memory as illustrated in Fig. 12-1. The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory. The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory may find the required word in the fast cache.

The performance of cache memory is frequently measured in terms of a quantity called *hit ratio*. When the CPU refers to memory and finds the word in cache, it is said to produce a *hit*. If the word is not found in cache, then it is in main memory and it counts as a *miss*. The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the hit ratio. The hit ratio is measured

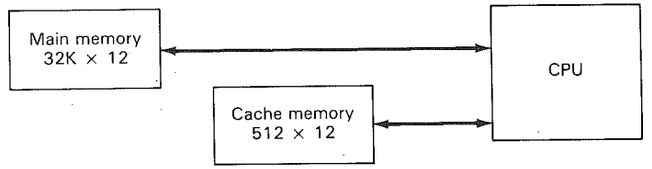


FIGURE 12-6
Example of Cache Memory

experimentally by running representative programs in the computer and measuring the number of hits and misses during a given interval of time. Hit ratios of 0.9 and higher have been reported. This high ratio verifies the validity of the locality of reference property.

The basic characteristic of cache memory is its fast access time. Therefore, very little time is wasted when searching for words in cache. The transformation of data from main memory to cache memory is referred to as a *mapping* process. Three types of mapping procedures are of practical interest in considering the organization of cache memory.

1. Associative mapping.
2. Direct mapping.
3. Set-associative mapping.

To help in the discussion of these three mapping procedures, we will use a specific example of a memory organization shown in Figure 12-6. The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory. The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12-bit data word from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

Associative Mapping

The fastest and the most flexible cache organization uses an associative memory. This organization is illustrated in Fig. 12-7. The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory . The diagram shows three words stored in cache. The address value of 15 bits is shown as a 5-digit octal number and its corresponding 12-bit data word is shown as a 4-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory. If the cache is full, an address-data pair must be displaced to make room for a pair that is needed but is not presently in the cache. The decision concerning what pair to replace is determined from the replacement algorithm that

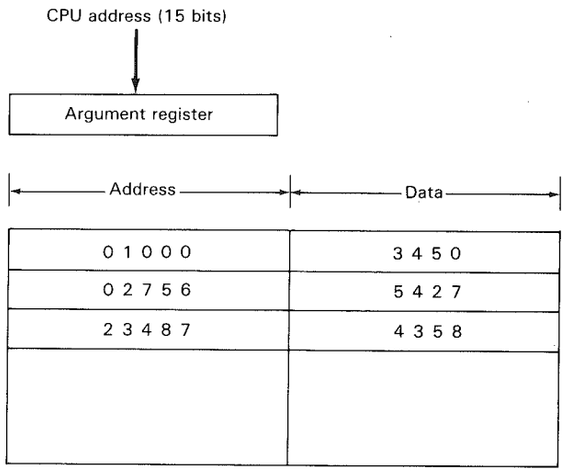


FIGURE 12-7
Associative Mapping Cache (All Numbers are in Octal)

the designer chooses for the cache. A simple procedure is to replace the words in cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

Direct Mapping

Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Figure 12-8. The CPU address of 15 bits is divided into two fields. The nine low-order bits constitute the *index* field and the remaining six bits form the *tag* field. The figure shows that main memory needs an address that includes both the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n in main memory. The n-bit memory address is divided into two fields: k bits for the index field and $n - k$ bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access cache. The internal organization of the words in cache memory is as shown in Figure 12-9(b). Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag field in the word read from cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory. It is then stored in the cache together with the new tag, replacing the previous value.

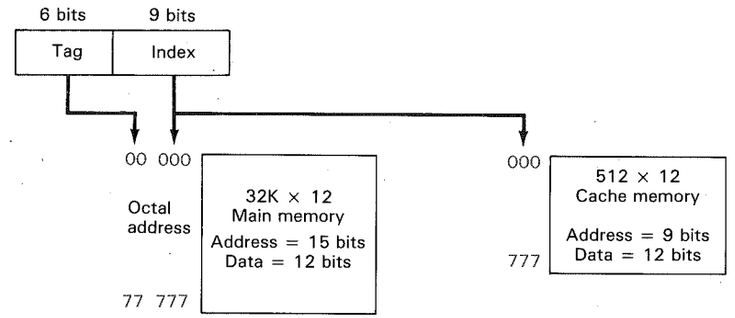
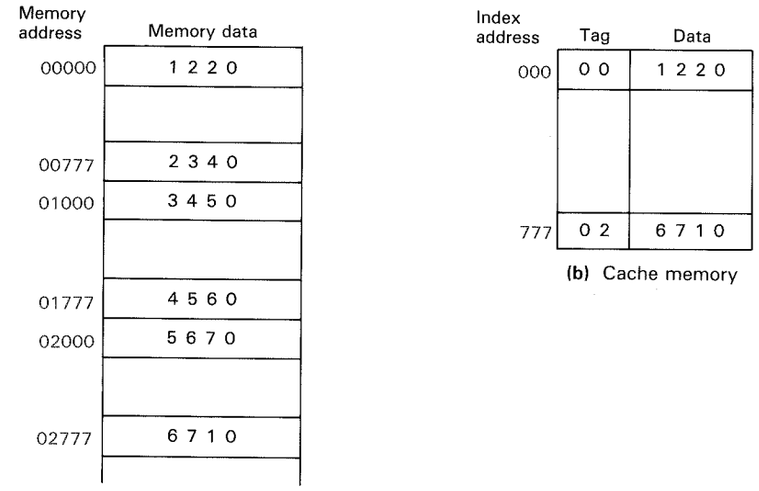


FIGURE 12-8
Addressing Relationships Between Main Memory and Cache

The disadvantage of direct mapping is that the hit ratio may drop considerably if two or more words with addresses having the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

To see how the direct mapping scheme operates, consider the numerical example shown in Figure 12-9. The word at address zero is presently stored in cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is used to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.



(a) Main memory

(b) Cache memory

FIGURE 12-9
Direct Mapping Cache Organization

Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

FIGURE 12-10
Set Associative Mapping Cache with Set Size of Two

Set-Associative Mapping

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index but with different tag values cannot reside in cache at the same time. A third type of cache organization, called set-associative mapping, is an improvement over the direct mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag, and the number of tag-data items in one word of cache is said to form a set. An example of a set-associative cache organization for a set size of two is shown in Figure 12-10. Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is $2(6 + 12) = 36$ bits. An index address of nine bits can accommodate 512 words. Thus, the size of cache memory is 512×36. It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

The octal numbers listed in Figure 12-10 refer to the main memory contents illustrated in Figure 12-9(a). The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777. When the CPU generates a memory request, the index value of the address is used to access the cache. The tag field of the CPU address is then compared with both tags in the cache to determine if a match occurs. The comparison logic is done by an associative search of the tags in the set similar to an associative memory search, hence the name set-associative. The hit ratio will improve as the set size increases because more words with the same index but different tags can reside in cache. An increase in the set size increases the number of bits in the words of cache and requires more complex comparison logic.

12-5 VIRTUAL MEMORY MANAGEMENT

In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or data are brought into main memory as they are needed by the CPU. *Virtual memory* is a concept used in some computer systems that permits the user to construct his program as though he had a large memory space, equal to the total of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual or logical

address to a physical address in main memory. Virtual memory is used to give the programmer the illusion that he has a very large memory at his disposal, even though the computer may have a relatively small main memory. A virtual memory system provides a mechanism for translating program generated addresses into corresponding main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of mapping tables.

In a multiprogramming environment where many programs reside in memory, it becomes necessary to move programs and data around in memory, to vary the amount of memory in use by a given program, and to prevent a program from changing other programs. The demands on computer memory brought about by multiprogramming have created the need for a memory management system. A memory management system is a collection of hardware and software procedures for managing the various programs residing in memory. The memory management software is part of an overall operating system available in the computer. Here we are concerned with the hardware unit associated with the memory management system.

Address Space and Memory Space

The address used by a program is called a virtual or *logical address*, and the set of such addresses is referred to as the *address space*. An address in main memory is called a location or *physical address*. The set of such locations is called the *memory space*. The address space is the set of addresses generated by the operating system for programs as they reference instructions and data. The memory space consists of the actual memory locations directly addressable for processing. In most computers the address and memory spaces are identical. The address space is allowed to be larger than the memory space in computers with virtual memory.

As an illustration, consider a computer with main memory capacity of 64K words. Sixteen bits are needed to specify a physical address in memory since $64K = 2^{16}$. Suppose that the computer has available auxiliary memory for storing $2^{20} = 1024K$ words. The auxiliary memory has a capacity of storing information equivalent to 16 memories since $64K \times 16 = 1024K$.

The address mapping from address space to physical space can be simplified if the addresses are divided into groups of fixed size. The logical memory is broken down into groups of equal size called *pages*. The term *block* or *page frame* refers to groups of equal size in physical memory. For example, if a page or block consists of 1K words then, using the previous example, address space is divided into 1024 pages and main memory is divided into 64 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space. The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page.

Memory management systems divide programs and data into logical parts called *segments*. A segment may be generated by the programmer or by the operating system. Examples of segments are subroutines, arrays of data, tables of items, or users' programs.

Dynamic Address Translation

In many applications, the logical address space uses variable length segments. The length of each segment is allowed to grow or contract according to the needs of the program being executed. One way of specifying the length of a segment is by associating with it a number of equal size pages. To see how this is done, consider the logical address shown in Figure 12-11. The logical address is partitioned into three fields. The segment field specifies a segment number. The page field specifies the page within the segment, and the offset field gives the specific word within the page. A page field of k bits can specify up to 2^k pages and a segment field of m bits can specify up to 2^m segments. Each segment may be associated with just one page or with as many as 2^k pages. The length of a segment may vary according to the number of pages that are assigned to it.

A numerical example may clarify the assignment of a logical address. Consider a logical address of 20 bits with 4 bits assigned to the segment field, 8 bits to the page field, and the remaining 8 bits to the offset field. The 4-bit segment can specify 16 different segments. The 8-bit page number can specify up to 256 pages, and the 8-bit offset implies a page size of 256 words. This configuration allows each segment to have any number of pages up to 256. The smallest possible segment will have one page of 256 words. The largest possible segment will have 256 pages for a total of $256 \times 256 = 64$K words.

The mapping of the logical address into a physical address is done by means of two tables, as shown in Figure 12-11. The segment number of the logical address specifies an address for the segment table. The entry in the segment table is a pointer address for a page table base. The page table base is added to the page

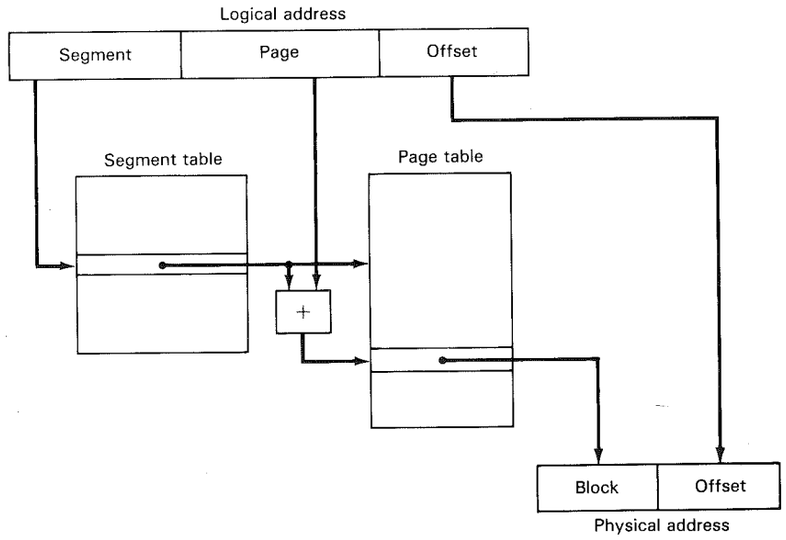


FIGURE 12-11
Address Translation in a Memory Management System

number given in the logical address. The sum produces a pointer address to an entry in the page table. The value in the page table provides the block number in physical memory. The concatenation of the block number with the offset produces the final physical mapped address.

The two mapping tables may be stored in two separate small memories or in main memory. In either case, a memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table, and one from main memory. This will slow the system significantly, compared to a conventional system that requires only one reference to memory. To avoid this speed penalty, a fast associative memory is used to hold the most recently referenced table entries. (This memory is sometimes called a translation lookaside buffer abbreviated TLB.) The first time a given block is referenced, its value together with the corresponding segment and page numbers are entered into the associative memory. Thus, the mapping process is first attempted by associative search with the given segment and page numbers. If it succeeds, the mapping delay is only that of the associative memory. If no match occurs, the slower table mapping procedure is used and the result transformed into the associative memory for future reference.

REFERENCES

1. MANO, M. M. *Computer System Architecture.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1982.
2. HANLON, A. G. "Content-Addressable and Associative Memory Systems: A Survey." *IEEE Trans. on Electronic Computers.* EC-15 (Aug. 1966): 509–521.
3. BELL, J., CASASENT, D., AND BELL, C. G. "An Investigation of Alternative Cache Organizations." *IEEE Trans. on Computers.* C-23(Apr. 1974): 346–351.
4. DENNING, P. J. "Virtual Memory." *Computing Surveys.* (Sept. 1970): 153–187.
5. HAMACHER, V. C., VRANESIC, Z. G., AND ZAKY, S. G. *Computer Organization.* 2nd ed. New York: McGraw-Hill, 1984.
6. TANENBAUM, A. S. *Structure Computer Organization.* 2nd ed. Englewood Cliffs: Prentice-Hall, 1984

PROBLEMS

12-1 Explain the need for auxiliary memory devices. How are they different from main memory and from other peripheral devices?

12-2 Explain the need for memory hierarchy. What is the main reason for not having a large enough main memory for storing all the available information in a computer system?

12-3 Define multiprogramming and explain the function of a memory management unit in computers that use the multiprogramming organization.

12-4 In certain scientific computations it is necessary to perform the arithmetic operation $(A_i + B_i)\ (C_i + D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. List the contents of all registers in the pipeline for $i = 1$ through 6.

12-5 Obtain the complement function for the match logic of one word in an associative memory. In other words, show that the complement of M_j is obtained from the logical sum of exclusive-OR functions. Draw the logic diagram for $\overline{M}_j$ and terminate it with an inverter to obtain M_j.

12-6 Obtain the Boolean function for the match logic of one word in an associative memory taking into consideration a tag bit that indicates whether the word is active or inactive.

12-7 A digital computer has a memory unit of 64K × 16 and a cache memory of 1K words. The cache uses direct mapping.
(a) How many bits are there in the tag and index fields of the address?
(b) How many bits are there in each word of the cache, and how are they divided into functions?

12-8 A set-associative mapping cache has a set size of 2. The cache can accommodate a total of 2048 words from main memory. The main memory size is 128K × 32.
(a) Formulate all pertinent information required to construct the cache memory.
(b) What is the size of the cache memory?

12-9 The access time of a cache memory is 100 nsec and that of main memory 1000 nsec. The hit ratio is 0.9. What is the average access time of the system?

12-10 An address space is specified by 24 bits and the corresponding memory space by 16 bits.
(a) How many words are there in the address space?
(b) How many words in the memory space?
(c) If a page consists of 2K words, how many pages and blocks are there in the system?

12-11 The logical address space in a computer system consists of 128 segments. Each segment can have up to 32 pages of 4K words in each. Physical memory consists of 4K blocks of 4K words in each. Formulate the logical and physical address formats.

12-12 Give the binary number of the logical address formulated in Problem 12-11 for segment 36 and word number 2000 in page 15.