CHAPTER ELEVEN

Input–Output Organization

IN THIS CHAPTER

11-1 Peripheral Devices

I/O

The input–output subsystem of a computer, referred to as I/O, provides an efficient mode of communication between the central system 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. A computer serves no useful purpose without the ability to receive information from an outside source and to transmit results in a meaningful form.

The most familiar means of entering information into a computer is through a typewriter-like keyboard that allows a person to enter alphanumeric information directly. Every time a key is depressed, the terminal sends a binary coded character to the computer. The fastest possible speed for entering information this way depends on the person's typing speed. On the other hand, the central processing unit is an extremely fast device capable of performing operations at very high speed. When input information is transferred to the processor via a slow keyboard, the processor will be idle most of the time while waiting for the information to arrive. To use a computer efficiently, a

large amount of programs and data must be prepared in advance and transmitted into a storage medium such as magnetic tapes or disks. The information in the disk is then transferred into computer memory at a rapid rate. Results of programs are also transferred into a high-speed storage, such as disks, from which they can be transferred later into a printer to provide a printed output of results.

Devices that are under the direct control of the computer are said to be connected on-line. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be part of the total computer system. Input or output devices attached to the computer are also called *peripherals*. Among the most common peripherals are keyboards, display units, and printers. Peripherals that provide auxiliary storage for the system are magnetic disks and tapes. Peripherals are electromechanical and electromagnetic devices of some complexity. Only a very brief discussion of their function will be given here without going into detail of their internal construction.

peripheral

monitor and keyboard

Video monitors are the most commonly used peripherals. They consist of a keyboard as the input device and a display unit as the output device. There are different types of video monitors, but the most popular use a cathode ray tube (CRT). The CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically. To produce a pattern on the screen, a grid inside the CRT receives a variable voltage that causes the beam to hit the screen and make it glow at selected spots. Horizontal and vertical signals deflect the beam and make it sweep across the tube, causing the visual pattern to appear on the screen. A characteristic feature of display devices is a cursor that marks the position in the screen where the next character will be inserted. The cursor can be moved to any position in the screen, to a single character, the beginning of a word, or to any line. Edit keys add or delete information based on the cursor position. The display terminal can operate in a single-character mode where all characters entered on the screen through the keyboard are transmitted to the computer simultaneously. In the block mode, the edited text is first stored in a local memory inside the terminal. The text is transferred to the computer as a block of data.

printer

Printers provide a permanent record on paper of computer output data or text. There are three basic types of character printers: daisywheel, dot matrix, and laser printers. The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and an energized magnet then presses the letter against the ribbon. The dot matrix printer contains a set of dots along the printing mechanism. For example, a 5×7 dot matrix printer that prints 80 characters per line has seven horizontal lines, each consisting of $5 \times 80 = 400$ dots. Each dot can be printed or not, depending on the specific characters that are printed on the line. The laser printer uses a rotating photographic drum

that is used to imprint the character images. The pattern is then transferred onto paper in the same manner as a copying machine.

magnetic tape

Magnetic tapes are used mostly for storing files of data: for example, a company's payroll record. Access is sequential and consists of records that can be accessed one after another as the tape moves along a stationary read–write mechanism. It is one of the cheapest and slowest methods for storage and has the advantage that tapes can be removed when not in use. Magnetic disks have high-speed rotational surfaces coated with magnetic material. Access is achieved by moving a read–write mechanism to a track in the magnetized surface. Disks are used mostly for bulk storage of programs and data. Tapes and disks are discussed further in Sec. 12-1 in conjunction with their role as auxiliary memory.

magnetic disk

Other input and output devices encountered in computer systems are digital incremental plotters, optical and magnetic character readers, analog-to-digital converters, and various data acquisition equipment. Not all input comes from people, and not all output is intended for people. Computers are used to control various processes in real time, such as machine tooling, assembly line procedures, and chemical and industrial processes. For such applications, a method must be provided for sensing status conditions in the process and sending control signals to the process being controlled.

The input–output organization of a computer is a function of the size of the computer and the devices connected to it. The difference between a small and a large system is mostly dependent on the amount of hardware the computer has available for communicating with peripheral units and the number of peripherals connected to the system. Since each peripheral behaves differently from any other, it would be prohibitive to dwell on the detailed interconnections needed between the computer and each peripheral. Certain techniques common to most peripherals are presented in this chapter.

ASCII Alphanumeric Characters

ASCII

Input and output devices that communicate with people and the computer are usually involved in the transfer of alphanumeric information to and from the device and the computer. The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters as shown in Table 11-1. The seven bits of the code are designated by b_1 through b_7, with b_7 being the most significant bit. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 characters that can be printed and 34 nonprinting characters used for various control functions. The printing characters consist of the 26 uppercase letters A through Z, the 26 lowercase letters, the 10 numerals 0 through 9, and 32 special printable characters such as %, *, and $.

The 34 control characters are designated in the ASCII table with abbrevi-

TABLE 11-1 American Standard Code for Information Interchange (ASCII)

	$b_7b_6b_5$							
$b_4b_3b_2b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

Control characters

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

ated names. They are listed again below the table with their functional names. The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of control characters: format effectors, information separators, and communication control characters. Format effectors are characters that control the layout of printing. They include

the familiar typewriter controls, such as backspace (BS), horizontal tabulation (HT), and carriage return (CR). Information separators are used to separate the data into divisions like paragraphs and pages. They include characters such as record separator (RS) and file separator (FS). The communication control characters are useful during the transmission of text between remote terminals. Examples of communication control characters are STX (start of text) and ETX (end of text), which are used to frame a text message when transmitted through a communication medium.

byte

ASCII is a 7-bit code, but most computers manipulate an 8-bit quantity as a single unit called a *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize 8-bit ASCII characters with the most significant bit set to 0. Additional 128 8-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font. When used in data communication, the eighth bit may be employed to indicate the parity of the binary-coded character.

11-2 Input–Output Interface

Input–output interface provides a method for transferring information between internal storage and external I/O devices. 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 and electromagnetic 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, and 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 are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

interface

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 may have its own controller that supervises the operations of the particular mechanism in the peripheral.

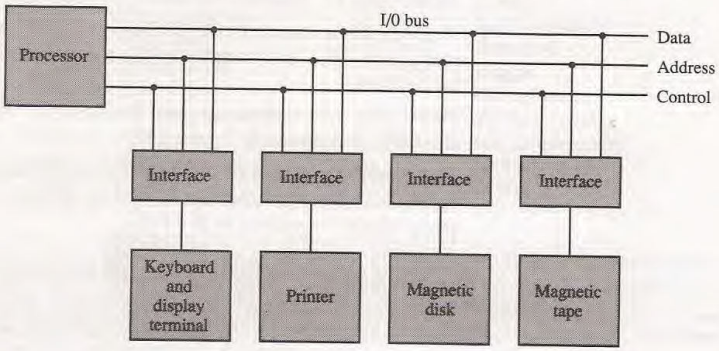
I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig. 11-1. The I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. It also synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. For example, the printer controller controls the paper motion, the print timing, and the selection of printing characters. A controller may be housed separately or may be physically integrated with the peripheral.

The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that 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 whose address does 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 lines, the processor provides a function code in the control lines. The interface

Figure 11-1 Connection of I/O bus to input-output devices.



selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. The interpretation of the command depends on the peripheral that the processor is addressing. There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

I/O command

control command

A *control command* is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

status

A *status command* is used to test various status conditions in the interface and the peripheral. For example, the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

output data

A *data output command* causes the interface to respond by transferring data from the bus into one of its registers. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command. The processor then monitors the status of the tape by means of a status command. When the tape is in the correct position, the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

input data

The *data input command* is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

IOP

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU). The memory communicates with both the CPU and the IOP through a memory bus. The IOP communicates also with the input and output devices through a separate I/O bus with its own address, data and control lines. The purpose of the IOP 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. In Sec. 11-7 we discuss the function of the IOP in more detail.

Isolated versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The *I/O read* and *I/O write* control lines are enabled during an I/O transfer. The *memory read* and *memory write* control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the *isolated I/O method* for assigning addresses in a common bus.

isolated I/O

In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line. This informs the external components that are attached to the common bus that the address in the address lines is for an interface register and not for a memory word. On the other hand, when the CPU is fetching an instruction or an operand from memory, it places the memory address on the address lines and enables the memory read or memory write control line. This informs the external components that the address is for a memory word and not for an I/O interface.

The isolated I/O method isolates memory and I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as *memory-mapped I/O*. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduces the memory address range available.

memory-mapped

In a memory-mapped I/O organization there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space. Typically, a segment of the total address space is reserved for interface registers, but in general, they can be located at any address as long as there is not also a memory word that responds to the same address.

Computers with memory-mapped I/O can use memory-type instructions to access I/O data. It allows the computer to use the same instructions for either input–output transfers or for memory transfers. The advantage is that the load and store instructions used for reading and writing from memory can be used to input and output data from I/O registers. In a typical computer, there are more memory-reference instructions than I/O instructions. With memory-mapped I/O all instructions that refer to memory are also available for I/O.

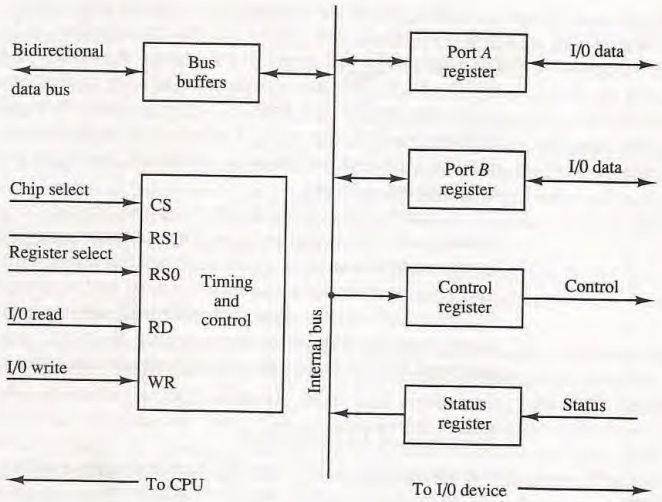
Example of I/O Interface

I/O port

An example of an I/O interface unit is shown in block diagram form in Fig. 11-2. It consists of two data registers called *ports*, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The I/O data to and 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 character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports *A* and *B* registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular interface register with which the CPU communicates.

The control register receives control information from the CPU. By loading appropriate bits into the control register, the interface and the I/O device attached to it can be placed in a variety of operating modes. For example, port *A* may be defined as an input port and port *B* as an output port. A magnetic tape unit may be instructed to rewind the tape or to start the tape moving in



CS	RS1	RS0	Register selected
0	×	×	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 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 I/O 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 and the two register select inputs. A circuit must be provided externally (usually, a decoder) 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 *RS1* and *RS0* 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 transfer 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.

11-3 Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

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 way of achieving this is by means of a *strobe* pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another 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 type of agreement between two independent units is referred to as *handshaking*.

strobe

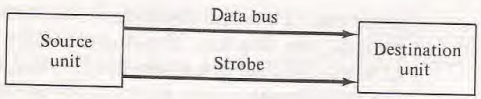
handshaking

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination. For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

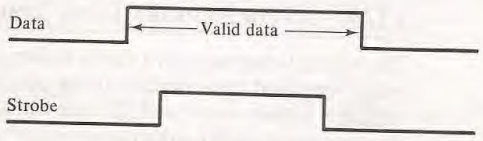
timing diagram

Strobe Control

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated by either the source or the destination unit. Figure 11-3(a) shows a source-initiated transfer.



(a) Block diagram



(b) Timing diagram

Figure 11-3 Source-initiated strobe for data transfer.

The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

As shown in the timing diagram of Fig. 11-3(b), the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. Actually, the source does not have to change the information in the data bus. The fact that the strobe signal is disabled indicates that the data bus does not contain valid data. New valid data will be available only after the strobe is enabled again.

Figure 11-4 shows a data transfer initiated by the destination unit. In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.

In many computers the strobe pulse is actually controlled by the clock pulses in the CPU. The CPU is always in control of the buses and informs the external units how to transfer data. For example, the strobe of Fig. 11-3 could be a memory-write control signal from the CPU to a memory unit. The source, being the CPU, places a word on the data bus and informs the memory unit,

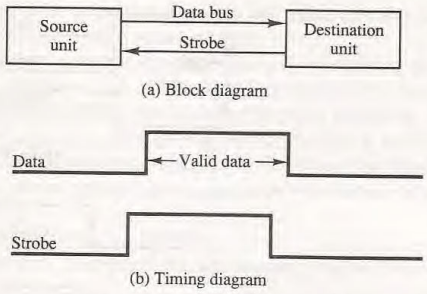


Figure 11-4 Destination-initiated strobe for data transfer.

which is the destination, that this is a write operation. Similarly, the strobe of Fig. 11-4 could be a memory-read control signal from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is the source, to place a selected word into the data bus.

The transfer of data between the CPU and an interface unit is similar to the strobe transfer just described. Data transfer between an interface and an I/O device is commonly controlled by a set of handshaking lines.

Handshaking

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer. The basic principle of the two-wire handshaking method 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 whether there are valid data in 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 inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

two-wire control

Figure 11-5 shows the data transfer procedure when initiated by the source. The two handshaking lines are *data valid*, which is generated by the source unit, and *data accepted*, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system can

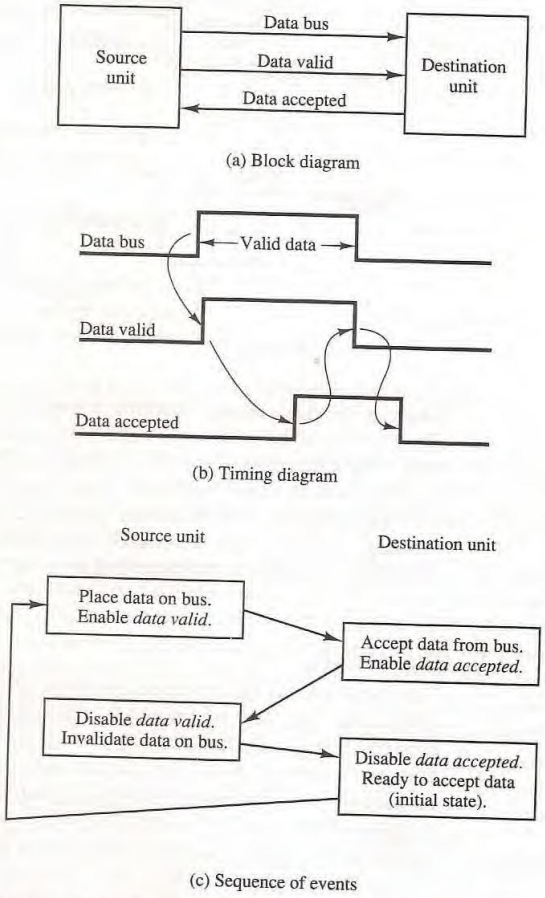


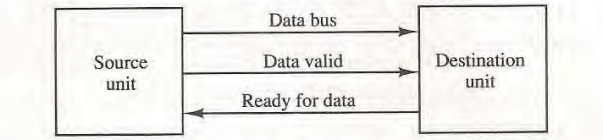
Figure 11-5 Source-initiated transfer using handshaking.

be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its *data valid* signal. The *data accepted* signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its *data valid* signal, which invalidates the data on the bus. The destination unit then disables its *data accepted* signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its *data accepted* signal. This scheme allows arbitrary delays from one state to the next

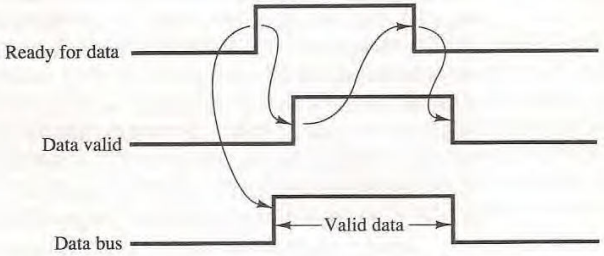
and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

The destination-initiated transfer using handshaking lines is shown in Fig. 11-6. Note that the name of the signal generated by the destination unit has been changed to *ready for data* to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the *ready for data* signal from the destination unit. From there on, the handshaking procedure follows the same pattern as in the source-initiated case. Note that the

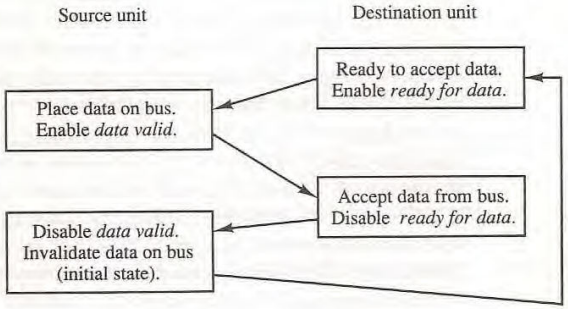
Figure 11-6 Destination-initiated transfer using handshaking.



(a) Block diagram



(b) Timing diagram



(c) Sequence of events

sequence of events in both cases would be identical if we consider the *ready for data* signal as the complement of *data accepted*. In fact, the only difference between the source-initiated and the destination-initiated transfer is in their choice of initial state.

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 if the data transfer is not completed within a predetermined time. 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 has occurred. The timeout signal can be used to interrupt the processor and hence execute a service routine that takes appropriate error recovery action.

timeout

Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, 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 must be transmitted 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 is less expensive since it requires only one pair of conductors.

Serial transmission can be synchronous or asynchronous. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks 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 frequency in both units synchronized with each other. Synchronous serial transmission is discussed further in Sec. 11-8.

synchronous

asynchronous

A serial asynchronous data transmission technique used in many interactive terminals employs special bits that are inserted at both ends of the character code. With this technique, each character consists of three parts: a start bit, the character bits, and stop bits. The convention is that the transmitter rests

start bit

at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in Fig. 11-7.

A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.
2. The initiation of a character transmission is detected from the start bit, which is always 0.
3. The character bits always follow the start bit.
4. After the last bit of the 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.

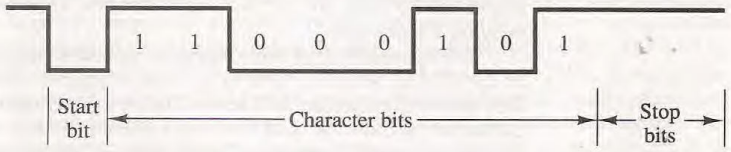
stop bit

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 the 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 one or two 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 ensures that a new character will not follow for one or two bit times.

As an illustration, consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists

Figure 11-7 Asynchronous serial transmission.



of a start bit, eight information bits, and two stop bits, for a total of 11 bits. Ten characters per second means that each character takes 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms. The

baud rate

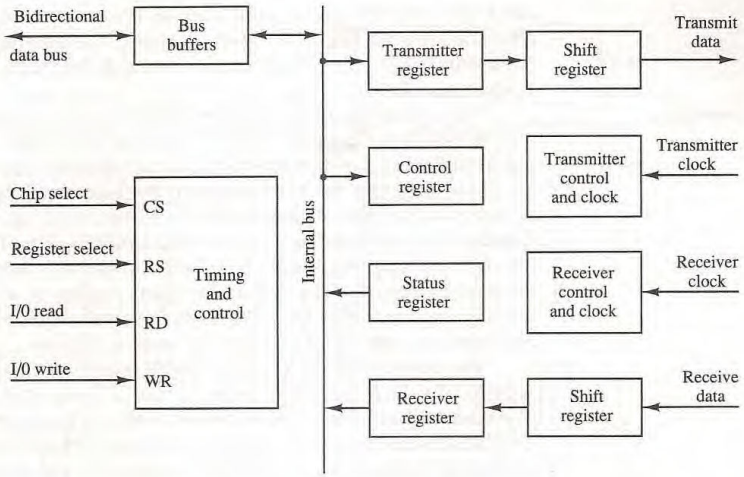
baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

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 into 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. Such a circuit is called an *asynchronous communication interface* or a *universal asynchronous receiver-transmitter* (UART).

Asynchronous Communication Interface

The block diagram of an asynchronous communication interface is shown in Fig. 11-8. It functions as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register. The transmitter register accepts a data byte from the CPU through the data bus. This byte is transferred to a shift register for serial transmission. The receiver portion receives serial information into another shift register, and when a complete data byte is accumulated, it is transferred to the receiver register. The CPU can select the receiver register to read the byte through the data bus. The bits in the status register are used for input and output flags and for recording certain errors that may occur during the transmission. The CPU can read the status register to check the status of the flag bits and to determine if any errors have occurred. The chip select and the read and write control lines communicate with the CPU. The chip select (CS) input is used to select the interface through the address bus. The register select (RS) is associated with the read (RD) and write (WR) controls. Two registers are write-only and two are read-only. The register selected is a function of the RS value and the RD and WR status, as listed in the table accompanying the diagram.

The operation of the asynchronous communication interface is initialized by the CPU by sending a byte to the control register. The initialization procedure places the interface in a specific mode of operation as it defines certain parameters such as the baud rate to use, how many bits are in each character, whether to generate and check parity, and how many stop bits are appended to each character. Two bits in the status register are used as flags. One bit is



CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Figure 11-8 Block diagram of a typical asynchronous communication interface.

used to indicate whether the transmitter register is empty and another bit is used to indicate whether the receiver register is full.

transmitter

The operation of the transmitter portion of the interface is as follows. The CPU reads the status register and checks the flag to see if the transmitter register is empty. If it is empty, the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit. The character is transferred in parallel from the transmitter register to the shift register and the appropriate number of stop bits are appended into the shift register. The transmitter register is then marked empty. The character can now be transmitted one bit at a time by shifting the data in the shift register at the specified

baud rate. The CPU can transfer another character to the transmitter register after checking the flag in the status register. The interface is said to be *double buffered* because a new character can be loaded as soon as the previous one starts transmission.

receiver

The operation of the receiver portion of the interface is similar. The receive data input is in the 1-state when the line is idle. The receiver control monitors the receive-data line for a 0 signal to detect the occurrence of a start bit. Once a start bit has been detected, the incoming bits of the character are shifted into the shift register at the prescribed baud rate. After receiving the data bits, the interface checks for the parity and stop bits. The character without the start and stop bits is then transferred in parallel from the shift register to the receiver register. The flag in the status register is set to indicate that the receiver register is full. The CPU reads the status register and checks the flag, and if set, it reads the data from the receiver register.

The interface checks for any possible errors during transmission and sets appropriate bits in the status register. The CPU can read the status register at any time to check if any errors have occurred. Three possible errors that the interface checks during transmission are parity error, framing error, and overrun error. Parity error occurs if the number of 1's in the received data is not the correct parity. A framing error occurs if the right number of stop bits is not detected at the end of the received character. An overrun error occurs if the CPU does not read the character from the receiver register before the next one becomes available in the shift register. Overrun error results in a loss of characters in the received data stream.

First-In, First-Out Buffer

FIFO

A first-in, first-out (FIFO) buffer is a memory unit that stores information in such a manner that the item first in is the item first out. A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input data and output data at two different rates and the output data are always in the same order in which the data entered the buffer. When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate. If the source unit is slower than the destination unit, the buffer can be filled with data at a slow rate and later emptied at the higher rate. If the source is faster than the destination, the FIFO is useful for those cases where the source data arrive in bursts that fill out the buffer but the time between bursts is long enough for the destination unit to empty some or all the information from the buffer. Thus a FIFO buffer can be useful in some applications when data are transferred asynchronously. It piles up data as they come in and gives them away in the same order when the data are needed.

The logic diagram of a typical 4×4 FIFO buffer is shown in Fig. 11-9. It consists of four 4-bit registers RI, $I = 1, 2, 3, 4$, and a control register with

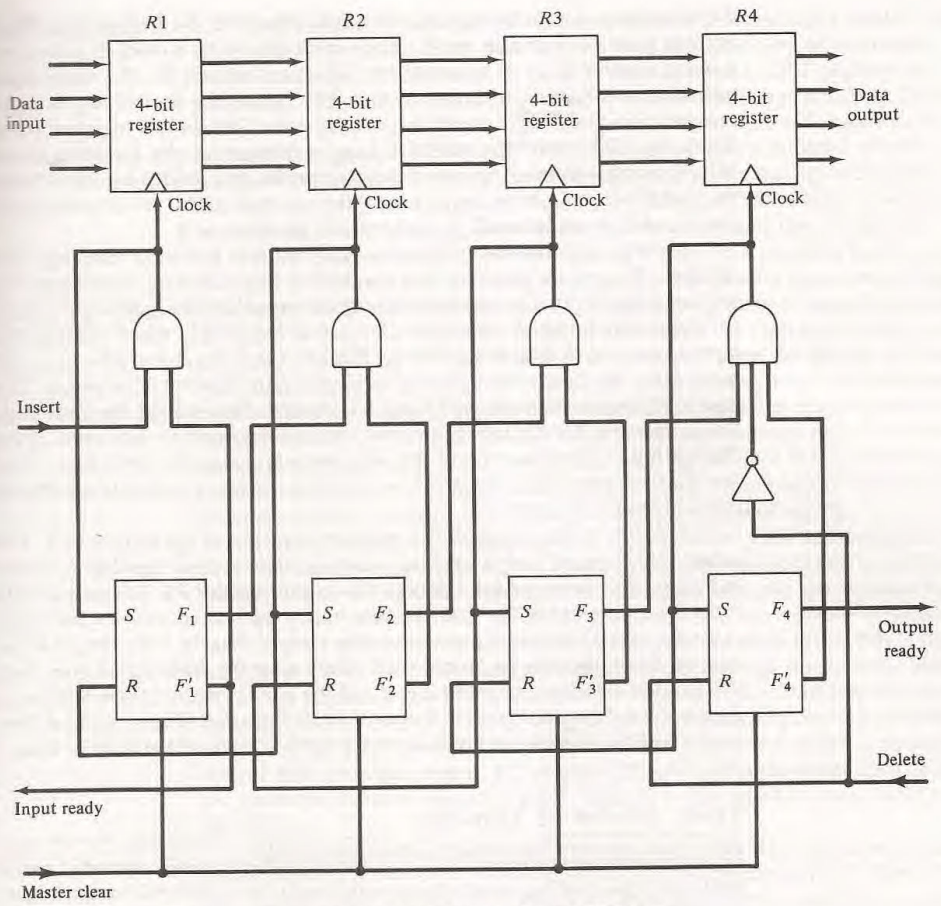


Figure 11-9 Circuit diagram of 4 × 4 FIFO buffer.

flip-flops F_i, $i = 1, 2, 3, 4$, one for each register. The FIFO can store four words of four bits each. The number of bits per word can be increased by increasing the number of bits in each register and the number of words can be increased by increasing the number of registers.

A flip-flop F_i in the control register that is set to 1 indicates that a 4-bit data word is stored in the corresponding register RI. A 0 in F_i indicates that the corresponding register does not contain valid data. The control register directs

the movement of data through the registers. Whenever the F_i bit of the control register is set ($F_i = 1$) and the F_{i+1} bit is reset ($F'_{i+1} = 1$), a clock is generated causing register $R(I + 1)$ to accept the data from register RI. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data. Data in the registers move down the FIFO toward the output as long as there are empty locations ahead of it. This ripple-through operation stops when the data reach a register RI with the next flip-flop F_{i+1} being set to 1, or at the last register $R4$. An overall master clear is used to initialize all control register flip-flops to 0.

Data are inserted into the buffer provided that the *input ready* signal is enabled. This occurs when the first control flip-flop F_1 is reset, indicating that register $R1$ is empty. Data are loaded from the input lines by enabling the clock in $R1$ through the *insert* control line. The same clock sets F_1, which disables the *input ready* control, indicating that the FIFO is now busy and unable to accept more data. The ripple-through process begins provided that $R2$ is empty. The data in $R1$ are transferred into $R2$ and F_1 is cleared. This enables the *input ready* line, indicating that the inputs are now available for another data word. If the FIFO is full, F_1 remains set and the *input ready* line stays in the 0 state. Note that the two control lines *input ready* and *insert* constitute a destination-initiated pair of handshake lines.

The data falling through the registers stack up at the output end. The *output ready* control line is enabled when the last control flip-flop F_4 is set, indicating that there are valid data in the output register $R4$. The output data from $R4$ are accepted by a destination unit, which then enables the *delete* control signal. This resets F_4, causing *output ready* to disable, indicating that the data on the output are no longer valid. Only after the *delete* signal goes back to 0 can the data from $R3$ move into $R4$. If the FIFO is empty, there will be no data in $R3$ and F_4 will remain in the reset state. Note that the two control lines *output ready* and *delete* constitute a source-initiated pair of handshake lines.

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 three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

programmed I/O

Programmed I/O 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 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 I/O device.

In the programmed I/O method, 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

interrupt

an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime 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 I/O transfer, and then returns to the task it was originally performing.

Transfer of data under programmed I/O is between CPU and peripheral.

DMA

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 DMA requests memory cycles through the memory bus. When the request is granted by the memory controller, the DMA transfers the data directly into memory. The CPU merely delays its memory access 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 Sec. 11-6.

IOP

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 Sec. 11-7.

Example of Programmed I/O

In the programmed I/O method, the I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig. 11-10. 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 its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register that we will refer to as an *F* or "flag" bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. This is according to the handshaking procedure established in Fig. 11-5.

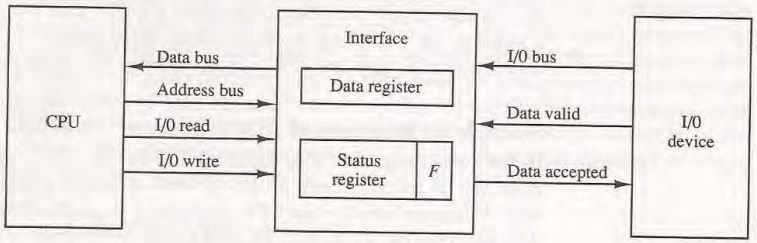
A program is written for the computer to check the flag in the status register to determine if a byte has been placed in the data register by the I/O device. This is done by reading the status register into a CPU register and checking the value of the flag bit. If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

A flowchart of the program that must be written for the CPU is shown in Fig. 11-11. It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words from an I/O device and store them in a memory buffer. A program that

Figure 11-10 Data transfer from I/O device to CPU.



F = Flag bit

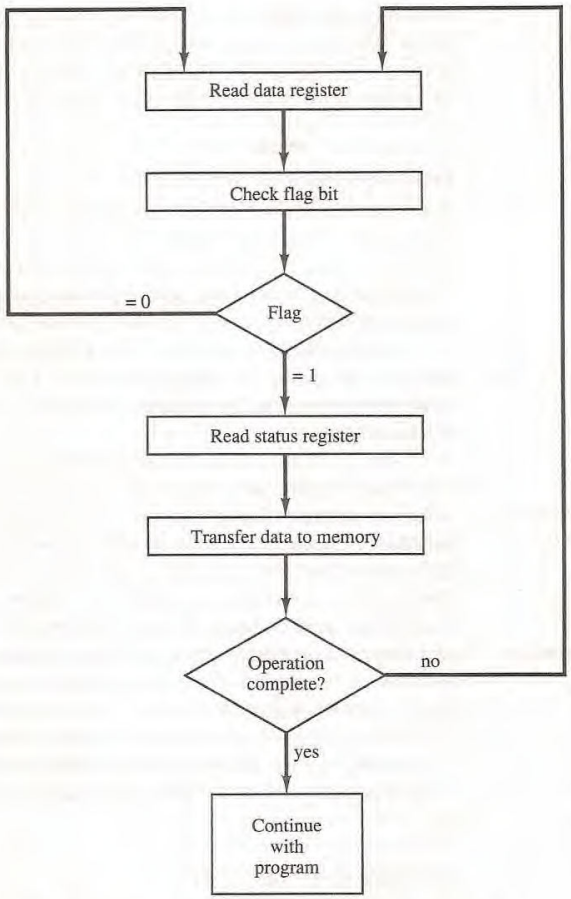


Figure 11-11 Flowchart for CPU program to input data.

stores input characters in a memory buffer using the instructions defined in Chap. 6 is listed in Table 6-21.

The programmed I/O method is particularly useful in small low-speed 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 two instructions that read the status register and check the flag in 1 μs. 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 μs. 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 I/O

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 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.

vectored interrupt

In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the *interrupt vector*. In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored. A system with vectored interrupt is demonstrated in Sec. 11-5.

Software Considerations

The previous discussion was concerned with the basic hardware needed to interface I/O devices to a computer system. A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Once ready, information is transferred item by item until all the data are transferred. In some cases, a control command is then given to execute a device function such as stop tape or print characters. Error checking and other useful steps often accompany the transfers. In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate the DMA channel to start its operation.

I/O routines

Software control of input–output equipment is a complex undertaking. For this reason I/O routines for standard peripherals are provided by the manufacturer as part of the computer system. They are usually included within the operating system. Most operating systems are supplied with a variety of I/O programs to support the particular line of peripherals offered for the computer. I/O routines are usually available as operating system procedures and the user refers to the established routines to specify the type of transfer required without going into detailed machine language programs.

11-5 Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU responds to the interrupt request by storing the return address from PC into a memory stack and then the program branches to a service routine that processes the required transfer. As discussed in Sec. 8-7, some processors also push the current PSW (program status word) onto the stack and load a new PSW for the service routine. We neglect the PSW here in order not to complicate the discussion of I/O interrupts.

In a typical application a number of I/O devices are attached to the computer, 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.

priority interrupt

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 low priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

polling

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 to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the daisy-chaining method.

Daisy-Chaining Priority

The daisy-chaining 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 up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Fig. 11-12. 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.

vector address (VAD)

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

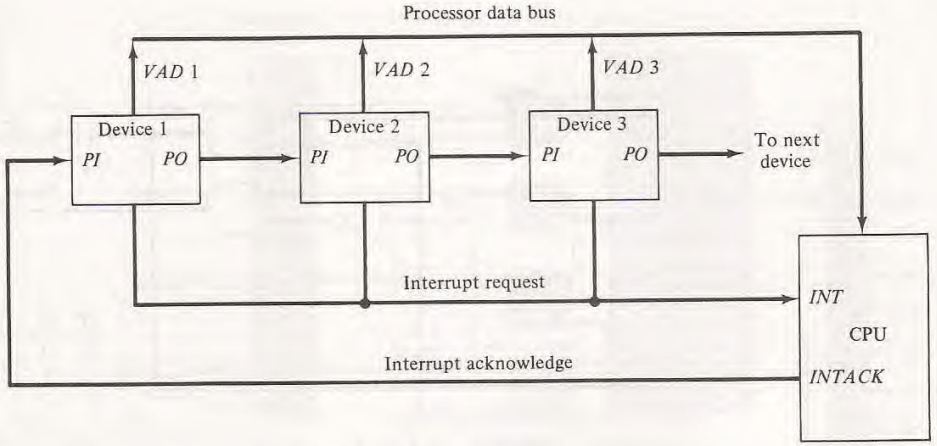


Figure 11-12 Daisy-chain priority interrupt.

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 11-13 shows the internal logic that must be included within each device when connected in the daisy-chaining 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 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 Interrupt

The parallel priority interrupt method uses a register whose bits are 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 whose purpose is to control the status of each interrupt request. The mask register can be programmed to disable

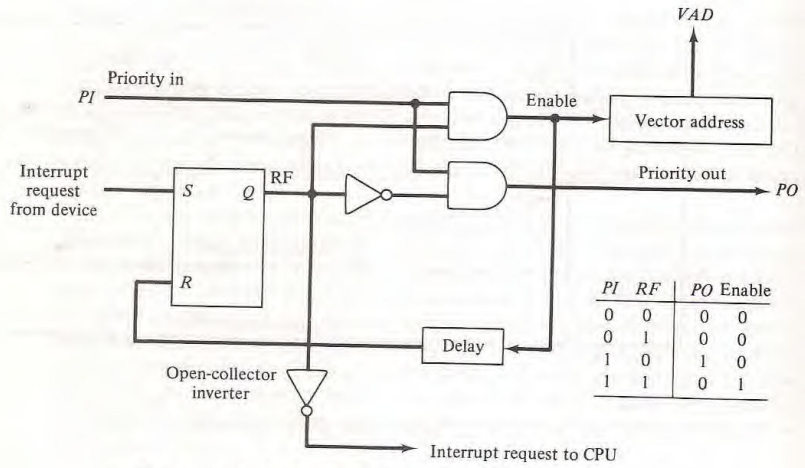


Figure 11-13 One stage of the daisy-chain priority arrangement.

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.

priority logic

The priority logic for a system of four interrupt sources is shown in Fig. 11-14. It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. 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.

Another output from the encoder sets an interrupt status flip-flop *IST* when an interrupt that is not masked occurs. The interrupt enable flip-flop *IEN* can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of *IST* ANDed with *IEN* provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

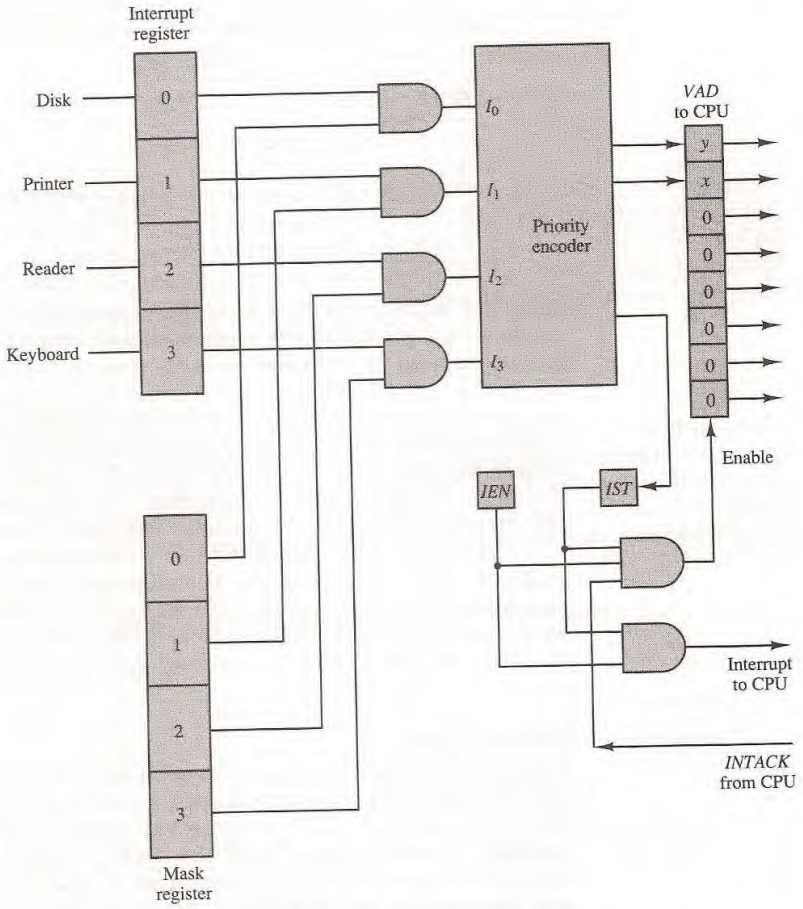


Figure 11-14 Priority interrupt hardware.

Priority Encoder

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 arrive at the same time, the input having the highest priority will take precedence. The truth table of a four-input priority encoder is given in Table 11-2. The ×'s in the table designate don't-care conditions. Input I_0 has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output $xy = 00$. I_1 has the next priority level. The output is 01 if $I_1 = 1$ provided

TABLE 11-2 Priority Encoder Truth Table

Inputs				Outputs			
I_0	I_1	I_2	I_3	x	y	IST	Boolean functions
1	×	×	×	0	0	1	
0	1	×	×	0	1	1	$x = I_0'I_1'$
0	0	1	×	1	0	1	$y = I_0'I_1 + I_0'I_2'$
0	0	0	1	1	1	1	$(IST) = I_0 + I_1 + I_2 + I_3$
0	0	0	0	×	×	0	

that $I_0 = 0$, regardless of the values of the other two lower-priority inputs. The output for I_2 is generated only if higher-priority inputs are 0, and so on down the priority level. The interrupt status *IST* is set only when one or more inputs are equal to 1. If all inputs are 0, *IST* is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when $IST = 0$. The Boolean functions listed in the table specify the internal logic of the encoder. Usually, a computer will have more than four interrupt sources. A priority encoder with eight inputs, for example, will generate an output of three bits.

The output of the priority encoder is used to form part of the vector address for each interrupt source. The other bits of the vector address can be assigned any value. For example, the vector address can be formed by appending six zeros to the x and y outputs of the encoder. With this choice the interrupt vectors for the four I/O devices are assigned binary numbers 0, 1, 2, and 3.

Interrupt Cycle

The interrupt enable flip-flop *IEN* shown in Fig. 11-14 can be set or cleared by program instructions. When *IEN* is cleared, the interrupt request coming from *IST* is neglected by the CPU. The program-controlled *IEN* bit allows the programmer to choose whether to use the interrupt facility. If an instruction to clear *IEN* has been inserted in the program, it means that the user does not want his program to be interrupted. An instruction to set *IEN* indicates that the interrupt facility will be used while the current program is running. Most computers include internal hardware that clears *IEN* to 0 every time an interrupt is acknowledged by the processor.

At the end of each instruction cycle the CPU checks *IEN* and the interrupt signal from *IST*. If either is equal to 0, control continues with the next instruction. If both *IEN* and *IST* are equal to 1, the CPU goes to an interrupt cycle. During the interrupt cycle the CPU performs the following sequence of microoperations:

$SP \leftarrow SP - 1$ Decrement stack pointer

$M[SP] \leftarrow PC$ Push *PC* into stack

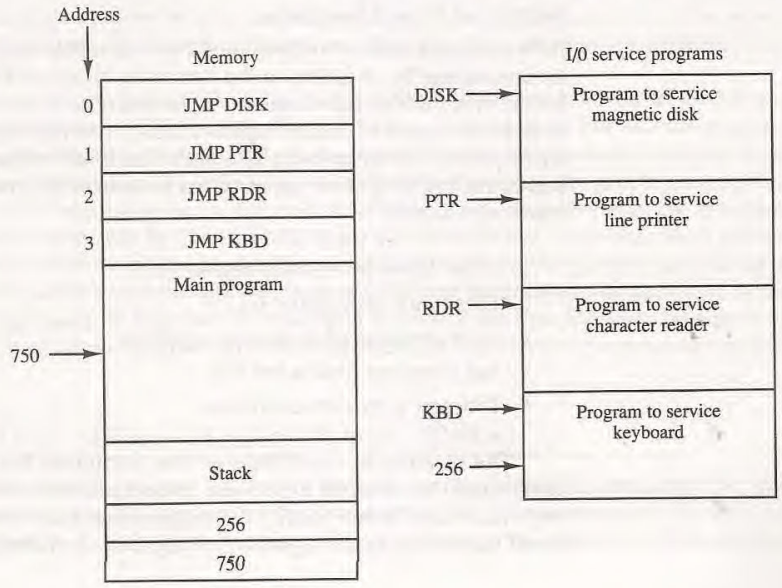
$INTACK \leftarrow 1$	Enable interrupt acknowledge
$PC \leftarrow VAD$	Transfer vector address to *PC*
$IEN \leftarrow 0$	Disable further interrupts
Go to fetch next instruction	

The CPU pushes the return address from *PC* into the stack. It then acknowledges the interrupt by enabling the *INTACK* line. The priority interrupt unit responds by placing a unique interrupt vector into the CPU data bus. The CPU transfers the vector address into *PC* and clears *IEN* prior to going to the next fetch phase. The instruction read from memory during the next fetch phase will be the one located at the vector address.

Software Routines

A priority interrupt system is a combination of hardware and software techniques. So far we have discussed the hardware aspects of a priority interrupt system. The computer must also have software routines for servicing the interrupt requests and for controlling the interrupt hardware registers. Figure 11-15 shows the programs that must reside in memory for handling the

Figure 11-15 Programs stored in memory for servicing interrupts.



service program

interrupt system. Each device has its own service program that can be reached through a jump (JMP) instruction stored at the assigned vector address. The symbolic name of each routine represents the starting address of the service program. The stack shown in the diagram is used for storing the return address after each interrupt.

To illustrate with a specific example assume that the keyboard sets its interrupt bit while the CPU is executing the instruction in location 749 of the main program. At the end of the instruction cycle, the computer goes to an interrupt cycle. It stores the return address 750 in the stack and then accepts the vector address 00000011 from the bus and transfers it to *PC*. The instruction in location 3 is executed next, resulting in transfer of control to the KBD routine. Now suppose that the disk sets its interrupt bit when the CPU is executing the instruction at address 255 in the KBD program. Address 256 is pushed into the stack and control is transferred to the DISK service program. The last instruction in each routine is a return from interrupt instruction. When the disk service program is completed, the return instruction pops the stack and places 256 into *PC*. This returns control to the KBD routine to continue servicing the keyboard. At the end of the KBD program, the last instruction pops the stack and returns control to the main program at address 750. Thus, a higher-priority device can interrupt a lower-priority device. It is assumed that the time spent in servicing the high-priority interrupt is short compared to the transfer rate of the low-priority device so that no loss of information takes place.

Initial and Final Operations

Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system. Remember that the interrupt enable *IEN* is cleared at the end of an interrupt cycle. This flip-flop must be set again to enable higher-priority interrupt requests, but not before lower-priority interrupts are disabled. The initial sequence of each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear lower-level mask register bits.
2. Clear interrupt status bit *IST*.
3. Save contents of processor registers.
4. Set interrupt enable bit *IEN*.
5. Proceed with service routine.

The lower-level mask register bits (including the bit of the source that interrupted) are cleared to prevent these conditions from enabling the interrupt. Although lower-priority interrupt sources are assigned to higher-numbered bits in the mask register, priority can be changed if desired since the

programmer can use any bit configuration for the mask register. The interrupt status bit must be cleared so it can be set again when a higher-priority interrupt occurs. The contents of processor registers are saved because they may be needed by the program that has been interrupted after control returns to it. The interrupt enable *IEN* is then set to allow other (higher-priority) interrupts and the computer proceeds to service the interrupt request.

The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear interrupt enable bit *IEN*.
2. Restore contents of processor registers.
3. Clear the bit in the interrupt register belonging to the source that has been serviced.
4. Set lower-level priority bits in the mask register.
5. Restore return address into *PC* and set *IEN*.

The bit in the interrupt register belonging to the source of the interrupt must be cleared so that it will be available again for the source to interrupt. The lower-priority bits in the mask register (including the bit of the source being interrupted) are set so they can enable the interrupt. The return to the interrupted program is accomplished by restoring the return address to *PC*. Note that the hardware must be designed so that no interrupts occur while executing steps 2 through 5; otherwise, the return address may be lost and the information in the mask and processor registers may be ambiguous if an interrupt is acknowledged while executing the operations in these steps. For this reason *IEN* is initially cleared and then set after the return address is transferred into *PC*.

The initial and final operations listed above are referred to as *overhead* operations or *housekeeping* chores. They are not part of the service program proper but are essential for processing interrupts. All overhead operations can be implemented by software. This is done by inserting the proper instructions at the beginning and at the end of each service routine. Some of the overhead operations can be done automatically by the hardware. The contents of processor registers can be pushed into a stack by the hardware before branching to the service routine. Other initial and final operations can be assigned to the hardware. In this way, it is possible to reduce the time between receipt of an interrupt and the execution of the instructions that service the interrupt source.

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

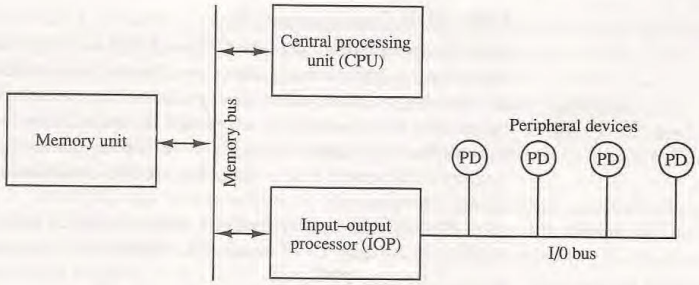


Figure 11-19 Block diagram of a computer with I/O processor.

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 are assembled into a memory word, they are 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.

commands

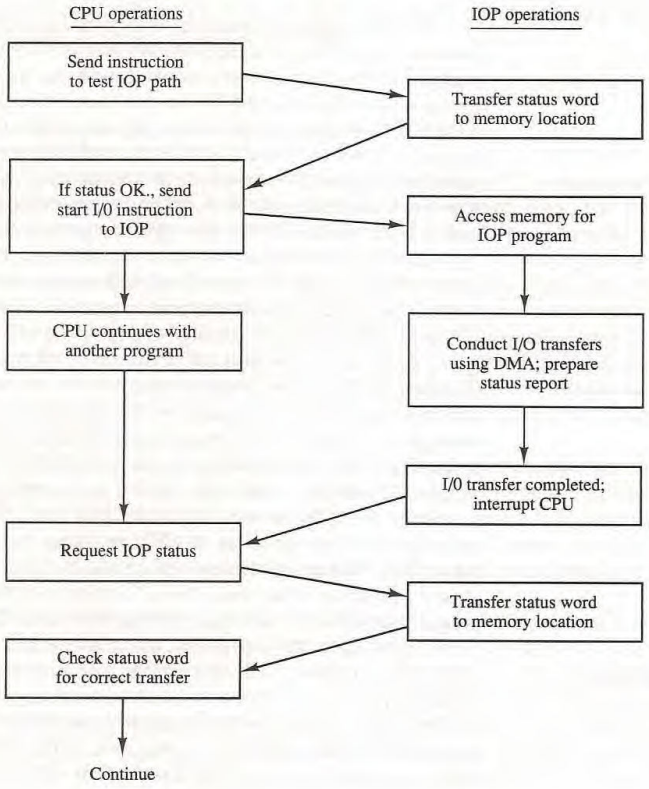
Instructions that are read from memory by an IOP are sometimes called *commands*, to distinguish them from instructions that are read by the CPU. Otherwise, an instruction and a command have similar functions. Commands are prepared by experienced programmers and are stored in memory. The command words 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.

CPU–IOP Communication

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 Fig. 11-20. The CPU sends an instruction to test the IOP path. The IOP

Figure 11-20 CPU-IOP communication.



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 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 the memory while the CPU is processing another program. The IOP and CPU are competing 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.

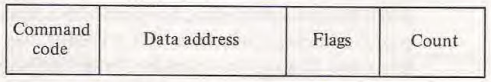
IBM 370 I/O Channel

The I/O processor in the IBM 370 computer is called a *channel*. A typical computer system configuration includes a number of channels with each channel attached to one or more I/O devices. There are three types of channels: multiplexer, selector, and block-multiplexer. The multiplexer channel can be connected to a number of slow- and medium-speed devices and is capable of operating with a number of I/O devices simultaneously. The selector channel is designed to handle one I/O operation at a time and is normally used to control one high-speed device. The block-multiplexer channel combines the features of both the multiplexer and selector channels. It provides a connection to a number of high-speed devices, but all I/O transfers are conducted with an entire block of data as compared to a multiplexer channel, which can transfer only one byte at a time.

The CPU communicates directly with the channels through dedicated control lines and indirectly through reserved storage areas in memory. Figure 11-21 shows the word formats associated with the channel operation.

(a) I/O instruction format

(b) Channel status word format



(c) Channel command word format

Figure 11-21 IBM 370 I/O related word formats.

The I/O instruction format has three fields: operation code, channel address, and device address. The computer system may have a number of channels, and each is assigned an address. Similarly, each channel may be connected to several devices and each device is assigned an address. The operation code specifies one of eight I/O instructions: start I/O, start I/O fast release, test I/O, clear I/O, halt I/O, halt device, test channel, and store channel identification. The addressed channel responds to each of the I/O instructions and executes it. It also sets one of four condition codes in a processor register called PSW (processor status word). The CPU can check the condition code in the PSW to determine the result of the I/O operation. The meaning of the four condition codes is different for each I/O instruction. But, in general, they specify whether the channel or the device is busy, whether or not it is operational, whether interruptions are pending, if the I/O operation had started successfully, and whether a status word was stored in memory by the channel.

The format of the channel status word is shown in Fig. 11-21(b). It is always stored in location 64 in memory. The key field is a protection mechanism used to prevent unauthorized access by one user to information that belongs to another user or to the operating system. The address field in the status word gives the address of the last command word used by the channel. The count field gives the residual count when the transfer was terminated. The count field will show zero if the transfer was completed successfully. The status field identifies the conditions in the device and the channel and any errors that occurred during the transfer.

The difference between the start I/O and start I/O fast release instructions is that the latter requires less CPU time for its execution. When the channel

receives one of these two instructions, it refers to memory location 72 for the address of the first channel command word (CCW). The format of the channel command word is shown in Fig. 11-21(c). The data address field specifies the first address of a memory buffer and the count field gives the number of bytes involved in the transfer. The command field specifies an I/O operation and the flag bits provide additional information for the channel. The command field corresponds to an operation code that specifies one of six basic types of I/O operations:

1. *Write*. Transfer data from memory to I/O device.
2. *Read*. Transfer data from I/O device to memory.
3. *Read backwards*. Read magnetic tape with tape moving backward.
4. *Control*. Used to initiate an operation not involving transfer of data, such as rewinding of tape or positioning a disk-access mechanism.
5. *Sense*. Informs the channel to transfer its channel status word to memory location 64.
6. *Transfer in channel*. Used instead of a jump instruction. Here the data address field specifies the address of the next command word to be executed by the channel.

An example of a channel program is shown in Table 11-3. It consists of three command words. The first causes a transfer into a magnetic tape of 60 bytes from memory starting at address 4000. The next two command words perform a similar function with a different portion of memory and byte count. The six flags in each control word specify certain interrelations between the command words. The first flag is set to 1 in the first command word to specify "data chaining." It results in combining the 60 bytes from the first command word with the 20 bytes of its successor into one record of 80 bytes. The 80 bytes are written on tape without any separation or gaps even though two memory sections were used. The second flag is set to 1 in the second command word to specify "command chaining." It informs the channel that the next command word will use the same I/O device, in this case, the tape. The channel informs the tape unit to start inserting a record gap on the tape and proceeds to read the next command word from memory. The 40 bytes of the third command

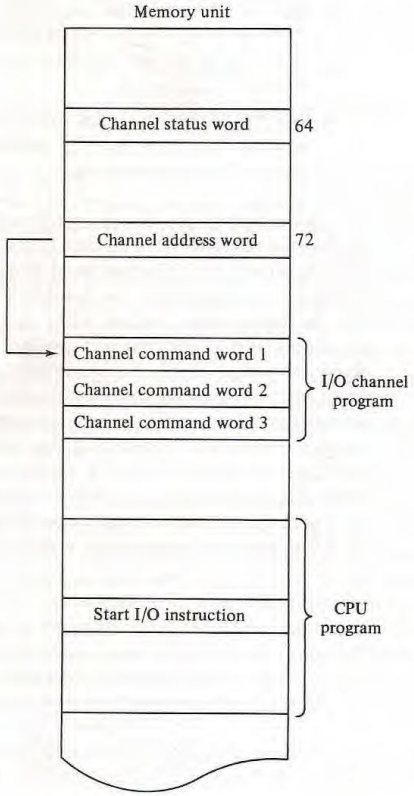
TABLE 11-3 IBM-370 Channel Program Example

Command	Address	Flags	Count
Write tape	4000	100000	60
Write tape	6000	010000	20
Write tape	3000	000000	40

word are then written on tape as a separate record. When all the flags are equal to zero, it signifies the end of I/O operations for the particular I/O device.

A memory map showing all pertinent information for I/O processing is illustrated in Fig. 11-22. The operation begins when the CPU program encounters a start I/O instruction. The IOP then goes to memory location 72 to obtain a channel address word. This word contains the starting address of the I/O channel program. The channel then proceeds to execute the program specified by the channel command words. The channel constructs a status word during

Figure 11-22 Location of information in memory for I/O operations in the IBM 370.



the transfer and stores it in location 64. Upon interruption, the CPU can refer to memory location 64 for the status word.

Intel 8089 IOP

The Intel 8089 I/O processor is contained in a 40-pin integrated circuit package. Within the 8089 are two independent units called *channels*. Each channel combines the general characteristics of a processor unit with those of a direct memory access controller. The 8089 is designed to function as an IOP in a microcomputer system where the Intel 8086 microprocessor is used as the CPU. The 8086 CPU initiates an I/O operation by building a message in memory that describes the function to be performed. The 8089 IOP reads the message from memory, carries out the operation, and notifies the CPU when it has finished.

In contrast to the IBM 370 channel, which has only six basic I/O commands, the 8089 IOP has 50 basic instructions that can operate on individual bits, on bytes, or 16-bit words. The IOP can execute programs in a manner similar to a CPU except that the instruction set is specifically chosen to provide efficient input–output processing. The instruction set includes general data transfer instructions, basic arithmetic and logic operations, conditional and unconditional branch operations, and subroutine call and return capabilities. The set also includes special instructions to initiate DMA transfers and issue an interrupt request to the CPU. It provides efficient data transfer between any two components attached to the system bus, such as I/O to memory, memory to memory, or I/O to I/O.

A microcomputer system using the Intel 8086/8089 pair of integrated circuits is shown in Fig. 11-23. The 8086 functions as the CPU and the 8089 as the IOP. The two units share a common memory through a bus controller connected to a system bus, which is called a "multibus" by Intel. The IOP uses a local bus to communicate with various interface units connected to I/O devices. The CPU communicates with the IOP by enabling the *channel attention* line. The *select* line is used by the CPU to select one of two channels in the 8089. The IOP gets the attention of the CPU by sending an interrupt request.

The CPU and IOP communicate with each other by writing messages for one another in system memory. The CPU prepares the message area and signals the IOP by enabling the channel attention line. The IOP reads the message, performs the required I/O functions, and executes the appropriate channel program. When the channel has completed its program, it issues an interrupt request to the CPU.

The communication scheme consists of program sections called "blocks," which are stored in memory as shown in Fig. 11-24. Each block contains control and parameter information as well as an address pointer to its successor block. The address of the control block is passed to each IOP channel during initialization. The busy flag indicates whether the IOP is busy or ready to perform

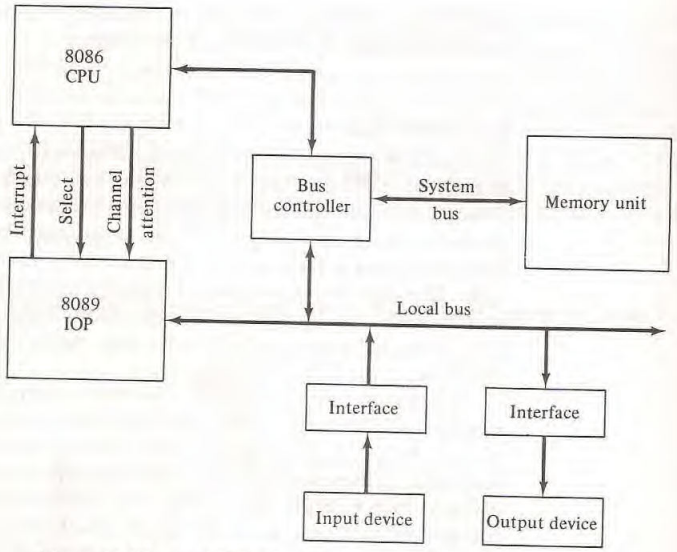


Figure 11-23 Intel 8086/8089 microcomputer system block diagram.

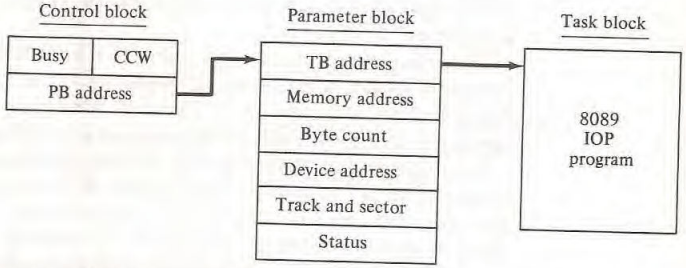


Figure 11-24 Location of information in memory for I/O operations in the Intel 8086/8089 microcomputer system.

a new I/O operation. The CCW (channel command word) is specified by the CPU to indicate the type of operation required from the IOP. The CCW in the 8089 does not have the same meaning as the command word in the IBM channel. The CCW here is more like an I/O instruction that specifies an operation for the IOP, such as start operation, suspend operation, resume operation, and halt I/O program. The parameter block contains variable data

that the IOP program must use in carrying out its task. The task block contains the actual program to be executed in the IOP.

The CPU and IOP work together through the control and parameter blocks. The CPU obtains use of the shared memory after checking the busy flag to ensure that the IOP is available. The CPU then fills in the information in the parameter block and writes a "start operation" command in the CCW. After the communication blocks have been set up, the CPU enables the channel attention signal to inform the IOP to start its I/O operation. The CPU then continues with another program. The IOP responds to the channel attention signal by placing the address of the control block into its program counter. The IOP refers to the control block and sets the busy flag. It then checks the operation in the CCW. The PB (parameter block) address and TB (task block) address are then transferred into internal IOP registers. The IOP starts executing the program in the task block using the information in the parameter block. The entries in the parameter block depend on the I/O device. The parameters listed in Fig. 11-24 are suitable for data transfer to or from a magnetic disk. The memory address specifies the beginning address of a memory buffer. The byte count gives the number of bytes to be transferred. The device address specifies the particular I/O device to be used. The track and sector numbers locate the data on the disk. When the I/O operation is completed, the IOP stores its status bits in the status word location of the parameter block and interrupts the CPU. The CPU can refer to the status word to check if the transfer has been completed satisfactorily.

11-8 Serial Communication

A data communication processor is an I/O processor that distributes and collects data from many remote terminals connected through telephone and other communication lines. It is a specialized I/O processor designed to communicate directly with data communication networks. A communication network may consist of any of a wide variety of devices, such as printers, interactive display devices, digital sensors, or a remote computing facility. With the use of a data communication processor, the computer can service fragments of each network demand in an interspersed manner and thus have the apparent behavior of serving many users at once. In this way the computer is able to operate efficiently in a time-sharing environment.

data communication processor

The most striking difference between an I/O processor and a data communication processor is in the way the processor communicates with the I/O devices. An I/O processor communicates with the peripherals through a common I/O bus that is comprised of many data and control lines. All peripherals share the common bus and use it to transfer information to and from the I/O processor. A data communication processor communicates with each terminal through a single pair of wires. Both data and control information are trans-

ferred in a serial fashion with the result that the transfer rate is much slower. The task of the data communication processor is to transmit and collect digital information to and from each terminal, determine if the information is data or control and respond to all requests according to predetermined established procedures. The processor, obviously, must also communicate with the CPU and memory in the same manner as any I/O processor.

The way that remote terminals are connected to a data communication processor is via telephone lines or other public or private communication facilities. Since telephone lines were originally designed for voice communication and computers communicate in terms of digital signals, some form of conversion must be used. 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 machine use. Various modulation schemes as well as different grades of communication media and transmission speeds are used. A communication line may be connected to a synchronous or asynchronous interface, depending on the transmission method of the remote terminal. An asynchronous interface receives serial data with start and stop bits in each character as shown in Fig. 11-7. This type of interface is similar to the asynchronous communication interface unit presented in Fig. 11-8.

modem

Synchronous transmission does not use start-stop bits to frame characters and therefore makes more efficient use of the communication link. High-speed devices use synchronous transmission to realize this efficiency. The modems used in synchronous transmission have internal clocks that are set to the frequency that bits are being transmitted in the communication line. For proper operation, it is required that the clocks in the transmitter and receiver modems remain synchronized at all times. The communication line, however, contains 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. The modem transfers the received data together with the clock to the interface unit. The interface or terminal on the transmitter side also uses the clock information from its modem. In this way, the same bit rate is maintained in both 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 transmitted sequentially as a block of data. The entire block is transmitted with special control characters at the beginning and end of the block. The control characters at the beginning of the block supply the information needed to separate the incoming bits into individual characters.

One of the functions of the data communication processor is to check for transmission errors. An error can be detected by checking the parity in each

character received. Another procedure used in asynchronous terminals involving a human operator is to *echo* the character. The character transmitted from the keyboard to the computer is recognized by the processor and retransmitted to the terminal printer. The operator would realize that an error occurred during transmission if the character printed is not the same as the character whose key he has struck.

block transfer

In synchronous transmission, where an entire block of characters is transmitted, each character has a parity bit for the receiver to check. After the entire block is sent, the transmitter sends one more character that constitutes a parity over the length of the message. This character is called a longitudinal redundancy check (LRC) and is the accumulation of the exclusive-OR of all transmitted characters. The receiving station calculates the LRC as it receives characters and compares it with the transmitted LRC. The calculated and received LRC should be equal for error-free messages. If the receiver finds an error in the transmitted block, it informs the sender to retransmit the same block once again. Another method used for checking errors in transmission is the cyclic redundancy check (CRC). This 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.

CRC

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 role of the modems is reversed to enable transmission in the reverse direction. The time required to switch a half-duplex line from one direction to the other is called the turnaround time.

full-duplex

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.

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 a data link is accomplished by means of a *protocol*. A data link control protocol is a set of rules that are followed by interconnecting computers and terminals to ensure the orderly transfer of

protocol

information. The purpose of a data link protocol is to establish and terminate 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 transfers. 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 (American Standard Code for Information Interchange). It is a 7-bit code with an eighth bit used for parity. The code has 128 characters, of which 95 are graphic characters and 33 are control characters. The graphic characters include the upper- and lowercase letters, the ten numerals, and a variety of special symbols. A list of the ASCII characters can be found in Table 11-1. The control characters are used for the purpose of routing data, arranging the test in a 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 11-4. Each character has a 7-bit code and 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.

SYN character

The SYN character serves as synchronizing agent between the transmitter and receiver. When the 7-bit ASCII code is used with an odd-parity bit in the most significant position, the assigned SYN character has the 8-bit code 00010110 which has the property that, upon circular shifting, it repeats itself only after a full 8-bit cycle. When the transmitter starts sending 8-bit characters, it sends a few 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 pulse, the receiver checks the last eight bits

TABLE 11-4 ASCII Communication Control Characters

Code	Symbol	Meaning	Function
0010110	SYN	Synchronous idle	Establishes synchronism
0000001	SOH	Start of heading	Heading of block message
0000010	STX	Start of text	Precedes block of text
0000011	ETX	End of text	Terminates block of text
0000100	EOT	End of transmission	Concludes transmission
0000110	ACK	Acknowledge	Affirmative acknowledgement
0010101	NAK	Negative acknowledge	Negative acknowledgement
0000101	ENQ	Inquiry	Inquire if terminal is on
0010111	ETB	End of transmission block	End of block of data
0010000	DLE	Data link escape	Special control character

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 pulse and bit received until a SYN character is recognized. Once a SYN character is detected, the receiver has framed a character. From here on the receiver counts every eight bits and accepts them as a single character. Usually, the receiver checks two consecutive SYN characters to remove any doubt that the first 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.

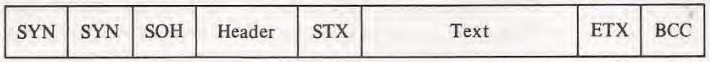
Messages are transmitted through the data link with an established format consisting of a header field, a text field, and an error-checking field. A typical message format for a character-oriented protocol is shown in Fig. 11-25. The two SYN characters assure proper synchronization at the start of the message. Following the SYN characters is the header, which starts with an SOH (start of heading) character. The header consists of address and control information. The STX character terminates the header and signifies the beginning of the 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 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 receiver accepts the message and calculates its own BCC. If the BCC transmitted does not agree with the BCC calculated by the receiver, the receiver responds with a negative acknowledge (NAK) character. The message is then retransmitted and checked again. Retransmission will be typically attempted several times before it is assumed that the line is faulty. When the transmitted BCC matches the one calculated by the receiver, the response is a positive acknowledgment using the ACK character.

Transmission Example

In order to appreciate the function of a data communication processor, let us illustrate by a specific example the method by which a terminal and the processor communicate. The communication with the memory unit and CPU is similar to any I/O processor.

Figure 11-25 Typical message format for character-oriented protocol.



A typical message that might be sent from a terminal to the processor is listed in Table 11-5. A look at this message reveals that there are a number of control characters used for message formation. Each character, including the control characters, is transmitted serially as an 8-bit binary code which consists of the 7-bit ASCII code plus an odd parity bit in the eighth most significant position. The two SYN characters are used to synchronize the receiver and transmitter. The heading starts with the SOH character and continues with two characters that specify the address of the terminal. In this particular example, the address is T4, but in general it can have any set of two or more graphic characters. The STX character terminates the heading and signifies the beginning of the text transmission. The text data of concern here is "request balance of account number 1234." The individual characters for this message are not listed in the table because they will take too much space. It must be realized, however, that each character in the message has an 8-bit code and that each bit is transmitted serially. The ETX control character signifies the termination of the text characters. The next character following ETX is a longitudinal redundancy check (LRC). Each bit in this character is a parity bit calculated from all the bits in the same column in the code section of the table.

The data communication processor receives this message and proceeds to analyze it. It recognizes terminal T4 and stores the text associated with the message. While receiving the characters, the processor checks the parity in each character and also computes the longitudinal parity. The computed LRC is compared with the LRC character received. If the two match, a positive acknowledgment (ACK) is sent back to the terminal. If a mismatch exists, a

TABLE 11-5 Typical Transmission from a Terminal to Processor

Code	Symbol	Comments
0001 0110	SYN	First sync character
0001 0110	SYN	Second sync character
0000 0001	SOH	Start of heading
0101 0100	T	Address of terminal is T4
0011 0100	4	
0000 0010	STX	Start of text transmission
0101 0010		
0100 0101	request	Text sent is a request to respond with the balance of
.	balance	account number 1234
.	of account	
.	No. 1234	
1011 0011		
0011 0100		
1000 0011	ETX	End of text transmission
0111 0000	LRC	Longitudinal parity character

negative acknowledgment (NAK) is returned to the terminal, which would initiate a retransmission of the same block. If the processor finds the message without errors, it transfers the message into memory and interrupts the CPU. When the CPU acknowledges the interrupt, it analyzes the message and prepares a text message for responding to the request. The CPU sends an instruction to the data communication processor to send the message to the terminal.

A typical response from processor to terminal is listed in Table 11-6. After two SYN characters, the processor acknowledges the previous message with an ACK character. The line continues to idle with SYN character waiting for the response to come. The message received from the CPU is arranged in the proper format by the processor by inserting the required control characters before and after the text. The message has the heading SOH and the address of the terminal T4. The text message informs the terminal that the balance is $100. An LRC character is computed and sent to the terminal. If the terminal responds with a NAK character, the processor retransmits the message.

While the processor is taking care of this terminal it is busy processing other terminals as well. Since the characters are received in a serial fashion, it takes a certain amount of time to receive and collect an 8-bit character. During this time the processor is multiplexing all other communication lines and

TABLE 11-6 Typical Transmission from Processor to Terminal

Code	Symbol	Comments
0001 0110	SYN	First sync character
0001 0110	SYN	Second sync character
1000 0110	ACK	Processor acknowledges previous message
0001 0110	SYN	Line is idling
.	.	
.	.	
.	.	
0001 0110	SYN	Line is idling
0000 0001	SOH	Start of heading
0101 0100	T	Address of terminal is T4
0011 0100	4	
0000 0010	STX	Start of text transmission
1100 0010		
1100 0001	balance	Text sent is a response from the computer giving the
.	is	balance of account
.	$100.00	
.		
.		
1011 0000		
1000 0011	ETX	End of text transmission
1101 0101	LRC	Longitudinal parity character

services each one in turn. The speed of most remote terminals is extremely slow compared to the processor speed. This property allows multiplexing of many users to achieve greater efficiency in a time-sharing system. This also allows many users to operate simultaneously while each is being sampled at speeds comparable to normal human response.

Data Transparency

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 which is not ASCII text. This happens, for example, when two remote computers send programs and data to each other over a communication channel. An arbitrary bit pattern in the text message becomes a problem in the character-oriented protocol. This is because any 8-bit pattern belonging to a communication control character will be interpreted erroneously by the receiver. For example, if the binary data in the text portion of the message has the 8-bit pattern 10000011, the receiver will interpret this as an ETX character and assume that it reached the end of the text field. When the text portion of the message is variable in length and contains bits that are to be treated without reference to any particular code, it is said to contain transparent data. This feature requires that the character recognition logic of the receiver be turned off so that data patterns in the text field are not accidentally interpreted as communication control information.

DLE character

Data transparency is achieved in character-oriented protocols by inserting a DLE (data link escape) character before each communication control character. Thus, the start of heading is detected from the double character DLE SOH, and the text field is terminated with the double character DLE ETX. If the DLE bit pattern 00010000 occurs in the text portion of the message, the transmitter inserts another DLE bit pattern following it. The receiver removes all DLE characters and then checks the next 8-bit pattern. If it is another DLE bit pattern, the receiver considers it as part of the text and continues to receive text. Otherwise, the receiver takes the following 8-bit pattern to be a communication control character.

The achievement of data transparency by means of the DLE character is inefficient and somewhat complicated to implement. Therefore, other protocols have been developed to make the transmission of transparent data more efficient. One protocol used by Digital Equipment Corporation employs a byte count field that gives the number of bytes in the message that follows. The receiver must then count the number of bytes received to reach the end of the text field. The protocol that has been mostly used to solve the transparency problem (and other problems associated with the character-oriented protocol) is the bit-oriented protocol.

Bit-Oriented Protocol

The bit-oriented protocol does not use characters in its control field and is independent of any particular code. It allows the transmission of serial bit stream of any length without the implication of character boundaries. Messages are organized in a specific format called 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. Examples of bit-oriented protocols are SDLC (synchronous data link control) used by IBM, HDLC (high-level data link control) adopted by the International Standards Organization, and ADCCP (advanced data communication control procedure) adopted by the American National Standards Institute.

Any data communication link involves at least two participating stations. The station that has responsibility for the data link and issues the commands to control the link is called the primary station. The other station is a secondary station. Bit-oriented protocols assume the presence of one primary station and one or more secondary stations. All communication on the data link is from the primary station to one or more secondary stations, or from a secondary station to the primary station.

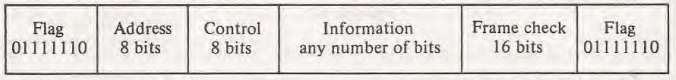
8-bit flag

The frame format for the bit-oriented protocol is shown in Fig. 11-26. 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 receiving station that the 16 bits just received constitute the CRC bits. The ending frame can be followed by another frame, another flag, 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 an information field. A frame must have a minimum of 32 bits between two flags to accommodate the address, control, and frame check fields. The maximum length depends on the condition of the communication channel and its ability to transmit long messages error-free.

zero insertion

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

Figure 11-26 Frame format for bit-oriented protocol.



by the transmitting station after any succession of five continuous 1's. The receiver always removes a 0 that follows a succession of five 1's. Thus the bit pattern 0111111 is transmitted as 01111101 and restored by the receiver to its original value by removal of the 0 following the five 1's. As a consequence, no pattern of 01111110 is ever transmitted between the beginning and ending flags.

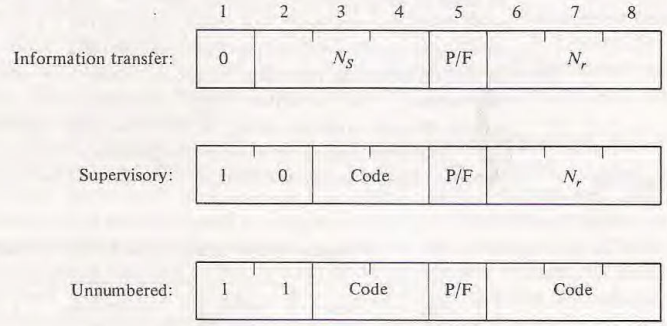
Following the flag is the address field, which is used by the primary station to designate the secondary station address. When a secondary station transmits a frame, the address tells the primary station which secondary station originated the frame. An address field of eight bits can specify up to 256 addresses. Some bit-oriented protocols permit the use of an extended address field. To do this, the least significant bit of an address byte is set to 0 if another address byte follows. A 1 in the least significant bit of a byte is used to recognize the last address byte.

control field

Following the address field is the control field. The control field comes in three different formats, as shown in Fig. 11-27. The information transfer format is used for ordinary data transmission. Each frame transmitted in this format contains send and receive counts. A station that transmits sequenced frames counts and numbers each frame. This count is given by the send count N_s. A station receiving sequenced frames counts each error-free frame that it receives. This count is given by the receive count N_r. The N_r count advances when a frame is checked and found to be without errors. The receiver confirms accepted numbered information frames by returning its N_r count to the transmitting station.

The P/F bit is used by the primary station to poll a secondary station to

Figure 11-27 Control field format in bit-oriented protocol.



N_s Send count P/F Poll/final
N_r Receive count Code Binary code

request that it initiate transmission. It is used by the secondary station to indicate the final transmitted frame. Thus the P/F field is called P (poll) when the primary station is transmitting but is designated as F (final) when a secondary station is transmitting. Each frame sent to the secondary station from the primary station has a P bit set to 0. When the primary station is finished and ready for the secondary station to respond, the P bit is set to 1. The secondary station then responds with a number of frames in which the F bit is set to 0. When the secondary station sends the last frame, it sets the F bit to 1. Therefore, the P/F bit is used to determine when data transmission from a station is finished.

The supervisory format of the control field is recognized from the first two bits being 1 and 0. The next two bits indicate the type of command. This follows by a P/F bit and a receive sequence frame count. The frames of the supervisory format do not carry an information field. They are used to assist in the transfer of information in that they confirm the acceptance of preceding frames carrying information, convey ready or busy conditions, and report frame numbering errors.

The unnumbered format is recognized from the first two bits being 11. The five code bits available in this format can specify up to 32 commands and responses. The primary station uses the control field to specify a command for a secondary station. The secondary station uses the control field to transmit a response to the primary station. Unnumbered-format frames are employed for initialization of link functions, reporting procedural errors, placing stations in a disconnected mode, and other data link control operations.

PROBLEMS

11-1. The addresses assigned to the four registers of the I/O interface of Fig. 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*1, and *RS*0 inputs of the interface.

11-2. Six interface units of the type shown in Fig. 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.

11-3. List four peripheral devices that produce an acceptable output for a person to understand.

11-4. Write your full name in ASCII using eight bits per character with the leftmost bit always 0. Include a space between names and a period after a middle initial.

11-5. What is the difference between isolated I/O and memory-mapped I/O? What are the advantages and disadvantages of each?

11-6. Indicate whether the following constitute a control, status, or data transfer commands.
- **a.** Skip next instruction if flag is set.
- **b.** Seek a given record on a magnetic disk.
- **c.** Check if I/O device is ready.
- **d.** Move printer paper to beginning of next page.
- **e.** Read interface status register.

11-7. 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 together with the pertinent interconnections among 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.

11-8. A CPU with a 20-MHz clock is connected to a memory unit whose access time is 40 ns. Formulate a read and write timing diagrams using a READ strobe and a WRITE strobe. Include the address in the timing diagram.

11-9. The asynchronous communication interface shown in Fig. 11-8 is connected between a CPU and a printer. Draw a flowchart that describes the sequence of operations in the transmitter portion of the interface when the CPU sends characters to be printed.

11-10. Give at least six status conditions for the setting of individual bits in the status register of an asynchronous communication interface.

11-11. How many bits are there in the transmitter shift register of Fig. 11-8 when the interface is attached to a terminal that needs one stop bit? List the bits in the shift register when the letter W is transmitted using ASCII with even parity.

11-12. 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.

11-13. Information is inserted into a FIFO buffer at a rate of m bytes per second. The information is deleted at a rate of n byte per second. The maximum capacity of the buffer is k bytes.
- **a.** How long does it take for an empty buffer to fill up when $m > n$?
- **b.** How long does it take for a full buffer to empty when $m < n$?
- **c.** Is the FIFO buffer needed if $m = n$?

11-14. The bits in the control register of the FIFO shown in Fig. 11-9 are $F_1 F_2 F_3 F_4 = 0011$. Give the sequence of internal operations when an item is deleted from the FIFO and then a new item is inserted.

11-15. What are the values of input ready and output ready and control bits F_1 through F_4 in Fig. 11-9 when:
- **a.** The buffer is empty?
- **b.** The buffer is full?
- **c.** The buffer contains two data items?

11-16. Show a block diagram similar to Fig. 11-10 for the data transfer from a CPU to an interface and then to an I/O device. Determine a procedure for setting and clearing the flag bit.

11-17. Using the configuration established in Prob. 11-16, obtain a flowchart (similar to Fig. 11-11) for the CPU program to output data.

11-18. What is the basic advantage of using interrupt-initiated data transfer over transfer under program control without an interrupt?

11-19. In most computers an interrupt is recognized only after the execution of the instruction. Consider the possibility of acknowledging the interrupt at any time during the execution of the instruction. Discuss the difficulty that may arise.

11-20. What happens in the daisy-chain priority interrupt shown in Fig. 11-12 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?

11-21. 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.

11-22. Using combinational circuit design techniques, derive the Boolean expressions listed in Table 11-2 for the priority encoder. Draw the logic diagram of the circuit.

11-23. Design a parallel priority interrupt hardware for a system with eight interrupt sources.

11-24. Obtain the truth table of an 8 × 3 priority encoder. Assume that the three outputs xyz from the priority encoder are used to provide a vector address of the form $101xyz00$. List the eight vector addresses starting from the one with the highest priority.

11-25. What should be done in Fig. 11-14 to make the four VAD values equal to the binary equivalent of 76, 77, 78, and 79?

11-26. What programming steps are required to check when a source interrupts the computer while it is still being serviced by a previous interrupt request from the same source?

11-27. 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?

11-28. 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 DMA as shown in Fig. 11-18.

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.

11-29. A DMA controller transfers 16-bit words to memory using cycle stealing. The words are assembled from a device that transmits characters at a rate of 2400 characters per second. The CPU is fetching and executing instructions at an average rate of 1 million instructions per second. By how much will the CPU be slowed down because of the DMA transfer?

11-30. Why does DMA have priority over the CPU when both request a memory transfer?

11-31. Draw a flowchart similar to the one in Fig. 11-20 that describes the CPU–I/O channel communication in the IBM 370.

11-32. 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.

11-33. List a possible line procedure and the character sequence for the communication between a data communication processor and a remote terminal. The processor inquires if the terminal is operative. The terminal responds with yes or no. If the response is yes, the processor sends a block of text.

11-34. A data communication link employs the character-controlled protocol with data transparency using the DLE character. The text message that the transmitter sends between STX and ETX is as follows:

```
DLE STX DLE DLE ETX DLE DLE ETX DLE ETX
```

What is the binary value of the transparent text data?

11-35. What is the minimum number of bits that a frame must have in the bit-oriented protocol?

11-36. 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 are transmitted.

REFERENCES

1. Gorsline, G. W., *Computer Organization: Hardware/Software*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1986.
2. Hays, J. F., *Computer Architecture and Organization*, 2nd ed. New York: McGraw-Hill, 1988.
3. Hill, F. J., and G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3rd ed. New York: John Wiley, 1987.
4. Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

5. Lippiatt, A. G., and G. L. Wright, *The Architecture of Small Computer Systems*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1985.
6. Patterson, D. A., and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.
7. Pollard, L. H., *Computer Design and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1990.
8. Rafiquzzaman, M., and R. Chandra, *Modern Computer Architecture*. St. Paul, MN: West Publishing, 1988.
9. Toy, W., and B. Zee, *Computer Hardware/Software Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1986.
10. Wakerly, J. F., *Microcomputer Architecture and Programming*. New York: John Wiley, 1981.
11. Ward, S. A., and R. H. Halstead, Jr., *Computation Structures*. Cambridge, MA: MIT Press, 1990.

CHAPTER TWELVE

Memory Organization

IN THIS CHAPTER

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 were 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 users accumulate and continue to accumulate large amounts of data-processing software. 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 a 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 tapes. They are used for storing system programs, large data files, and other backup information. Only programs and data currently needed by the processor reside in main memory. All

auxiliary memory

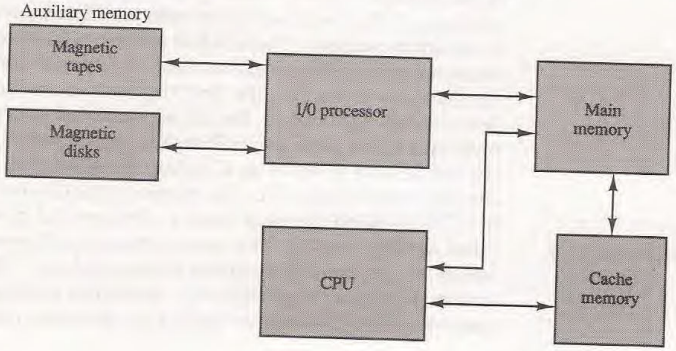
other information is stored in auxiliary memory and transferred to main memory when needed.

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.

cache memory

A special very-high-speed memory called a *cache* 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 is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory. A technique used to compensate for the mismatch in operating speeds is to employ an extremely fast, small cache between the CPU and main memory whose access time is close to processor logic clock cycle time. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.

Figure 12-1 Memory hierarchy in a computer system.



By making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer. The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed. Thus as the memory access speed increases, so does its relative cost. The overall goal of using a memory hierarchy is to obtain the highest-possible average access speed while minimizing the total cost of the entire memory system.

Auxiliary and cache memories are used for different purposes. The cache holds those parts of the program and data that are most heavily used, while the auxiliary memory holds those parts that are not presently used by the CPU. Moreover, the CPU has direct access to both cache and main memory but not to auxiliary memory. The transfer from auxiliary to main memory is usually done by means of direct memory access of large blocks of data. The typical access time ratio between cache and main memory is about 1 to 7. For example, a typical cache memory may have an access time of 100 ns, while main memory access time may be 700 ns. Auxiliary memory average access time is usually 1000 times that of main memory. Block size in auxiliary memory typically ranges from 256 to 2048 words, while cache block size is typically from 1 to 16 words.

multiprogramming

Many operating systems are designed to enable the CPU to process a number of independent programs concurrently. This concept, called *multiprogramming*, refers to the existence of two or more programs in different parts of the memory hierarchy at the same time. In this way it is possible to keep all parts of the computer busy by working with several programs in sequence. For example, suppose that 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 utilize 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 Sec. 12-7.

12-2 Main Memory

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips 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 tend 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. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles.

random-access memory (RAM)

Most of the main memory in a general-purpose computer is made up of RAM integrated circuit chips, but a portion of the memory may be constructed with ROM chips. Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed.

read-only memory (ROM)

Among other things, the ROM portion of main memory is needed for storing an initial program called a *bootstrap loader*. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. Thus when power is turned on, the hardware of the computer sets the program counter to the

bootstrap loader

computer startup

first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size. To demonstrate the chip interconnection, we will show an example of a 1024 × 8 memory constructed with 128 × 8 RAM chips and 512 × 8 ROM chips.

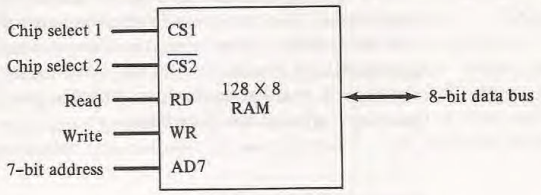
RAM and ROM Chips

bidirectional bus

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation, or from CPU to memory during a write operation. A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is shown in Fig. 12-2. The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit

Figure 12-2 Typical RAM chip.



(a) Block diagram

CS1	CS2	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

address and an 8-bit bidirectional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations of read or write.

The function table listed in Fig. 12-2(b) specifies the operation of the RAM chip. The unit is in operation only when CS1 = 1 and $\overline{CS2}$ = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When CS1 = 1 and $\overline{CS2}$ = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. 12-3. For the same-size chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and $\overline{CS2}$ = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

Memory Address Map

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established from knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a *memory address map*, is a pictorial representation of assigned address space for each chip in the system.

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips

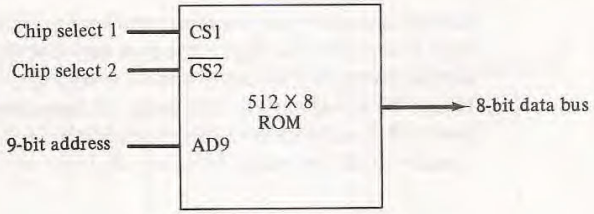


Figure 12-3 Typical ROM chip.

to be used are specified in Figs. 12-2 and 12-3. The memory address map for this configuration is shown in Table 12-1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to $2^9 = 512$ bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained from the information under the address bus assignment. The address bus lines are

TABLE 12-1 Memory Address Map for Microprocomputer

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

subdivided into groups of four bits each so that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

Memory Connection to CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in Fig. 12-4. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of Table 12-1. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2 × 4 decoder whose outputs go to the CS1 inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

12-3 Auxiliary Memory

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have a knowledge of magnetics, electronics, and electromechanical systems. Al-

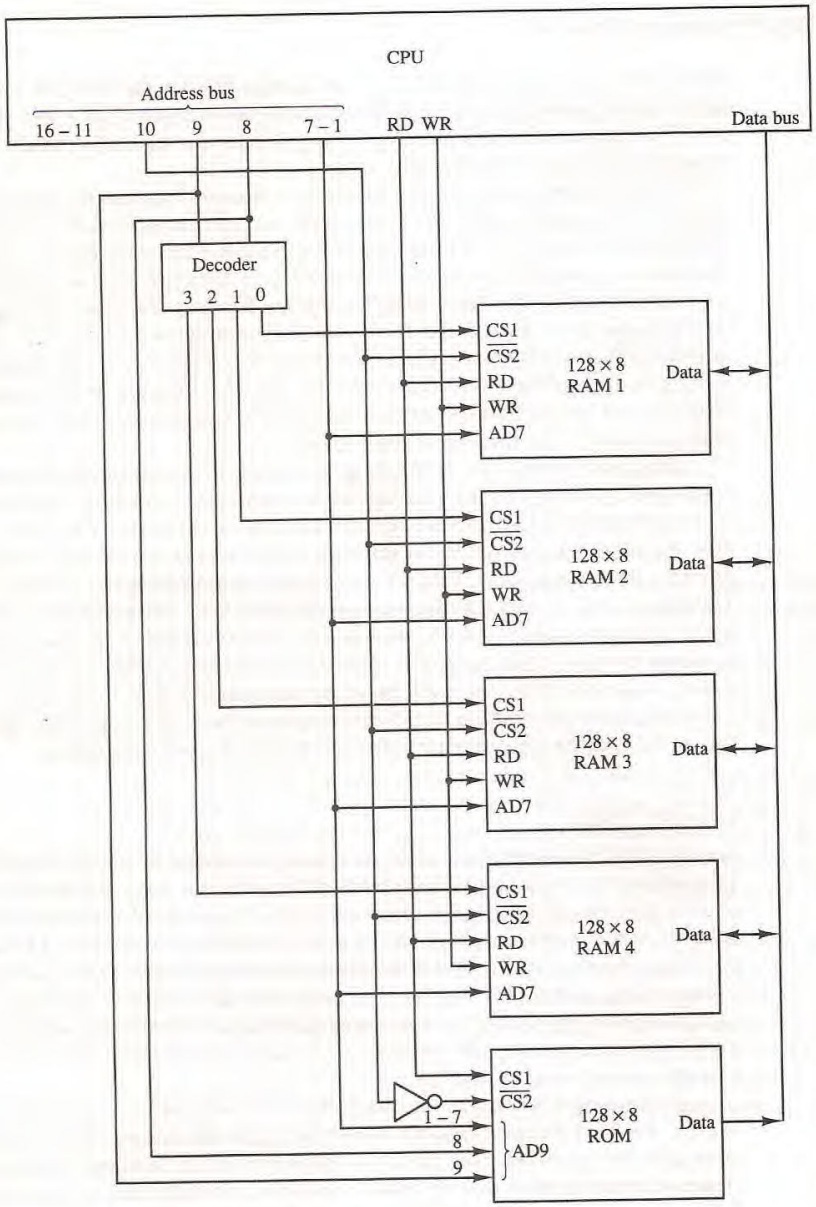


Figure 12-4 Memory connection to the CPU.

though the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a *seek* time required to position the read-write head to a location and a *transfer* time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks. A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. The recording surface rotates at uniform speed and is not started or stopped during access operations. Bits are recorded as magnetic spots on the surface as it passes a stationary mechanism called a *write head*. Stored bits are detected by a change in magnetic field produced by a recorded spot on the surface as it passes through a *read head*. The amount of surface available for recording in a disk is greater than in a drum of equal physical size. Therefore, more information can be stored on a disk than on a drum of comparable size. For this reason, disks have replaced drums in more recent computers.

Magnetic Disks

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started for access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. In most systems, the minimum quantity of information which can be transferred is a sector. The subdivision of one disk surface into tracks and sectors is shown in Fig. 12-5.

Some units use a single read/write head for each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

Permanent timing tracks are used in disks to synchronize the bits and

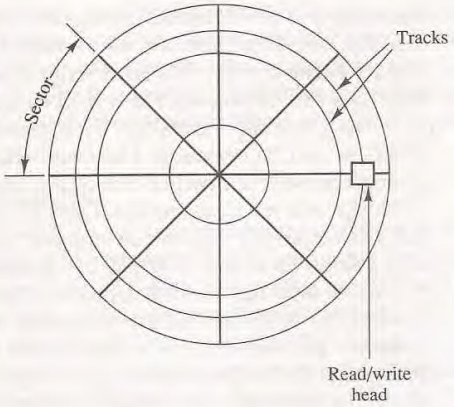


Figure 12-5 Magnetic disk.

recognize the sectors. A disk system is addressed by address bits that specify the disk number, the disk surface, the sector number and the track within the sector. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.

A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference. This equalizes the number of bits on all tracks of a given sector.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called *hard disks*. A disk drive with removable disks is called a *floppy disk*. The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

Magnetic Tape

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches the next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number and the number of characters in the record. Records may be of fixed or variable length.

12-4 Associative Memory

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search 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 or sequential access memory.

content addressable 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* (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the 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 all words which match the specified content and marks them 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

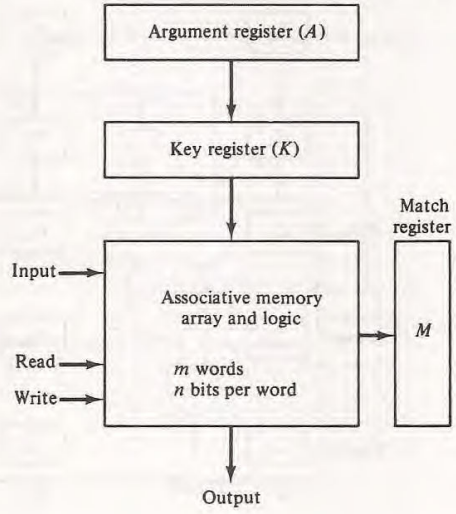
an entire word or on a specific field within a word. An associative memory is more expensive than a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument. For this reason, associative memories are used in applications where the search time is very critical and must be very short.

Hardware Organization

The block diagram of an associative memory is shown in Fig. 12-6. It consists of a memory array and logic for m words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word. The match register M has m bits, one for each memory word. Each word in memory 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. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which

Figure 12-6 Block diagram of associative memory.



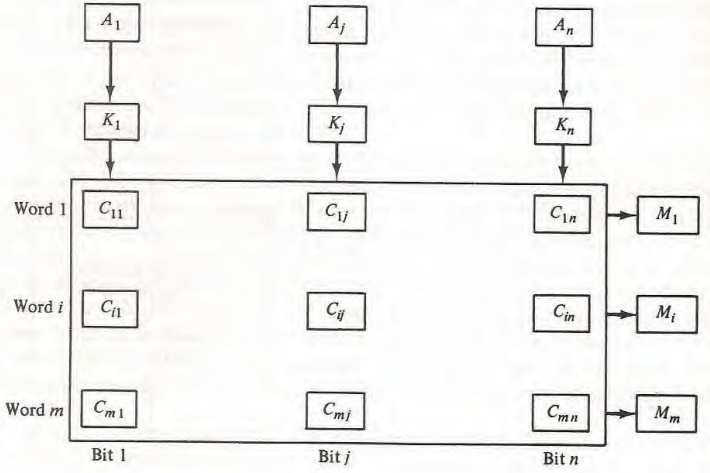
specifies how the reference to memory is made. To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.

A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in Fig. 12-7. The cells in the array are marked by the letter C with two subcripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that $K_j = 1$. This is done for all columns $j = 1, 2, \ldots, n$. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Figure 12-7 Associative memory of m word, n cells per word.



The internal organization of a typical cell C_{ij} is shown in Fig. 12-8. It consists of a flip-flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

Match Logic

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we *neglect* the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if $A_j = F_{ij}$ for $j = 1, 2, \ldots, n$. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

$$x_j = A_j F_{ij} + A_j' F_{ij}'$$

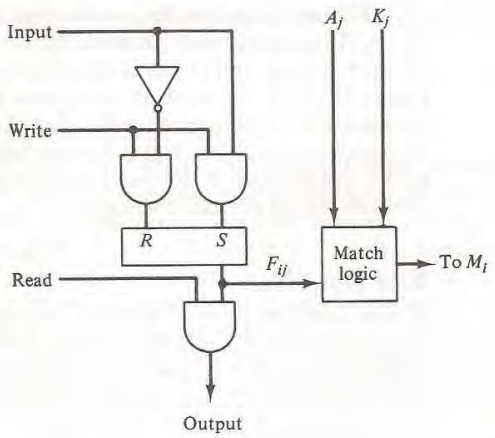
where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 x_2 x_3 \cdots x_n$$

and constitutes the AND operation of all pairs of matched bits in a word.

Figure 12-8 One cell of associative memory.



We now include the key bit K_j in the comparison logic. The requirement is that if $K_j = 0$, the corresponding bits of A_j and F_{ij} need no comparison. Only when $K_j = 1$ must they be compared. This requirement is achieved by ORing each term with K_j', thus:

$$x_j + K_j' = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ and $x_j + 1 = 1$. A term $(x_j + K_j')$ will be in the 1 state if its pair of bits is not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $K_j = 1$.

The match logic for word i in an associative memory can now be expressed by the following Boolean function:

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

Each term in the expression will be equal to 1 if its corresponding $K_j = 0$. If $K_j = 1$, the term will be either 0 or 1 depending on the value of x_j. A match will occur and M_i will be equal to 1 if all terms are equal to 1.

If we substitute the original definition of x_j, the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

where Π is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word $i = 1, 2, 3, \ldots, m$.

The circuit for matching one word is shown in Fig. 12-9. Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i. M_i will be logic 1 if a match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

Read Operation

If more than one word in memory matches the unmasked argument field, 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 line whose corresponding M_i bit is a 1.

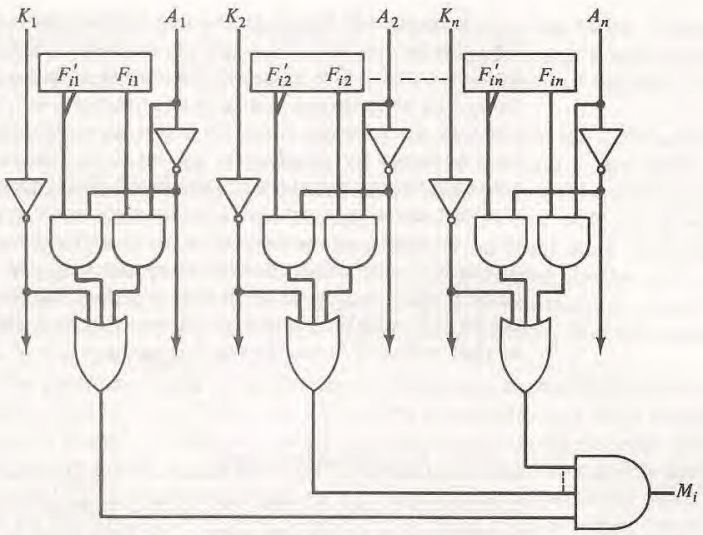


Figure 12-9 Match logic for one word of associative memory.

In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having a zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

Write Operation

An associative memory must have a write capability for storing the information to be searched. Writing in an associative memory can take different forms, depending on the application. If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where $m = 2^d$.

If 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*, would have as many bits as there are words in the 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 clearing 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. Moreover, the words that have a tag bit of 0 must be masked (together with the K_j bits) with the argument word so that only active words are compared.

12-5 Cache Memory

Analysis of a large number of typical programs has shown that the references to memory at any given interval of time tend to be confined within a few localized areas in memory. This phenomenon is known as the property of *locality of reference*. The reason for this property may be understood considering that a typical computer program flows in a straight-line fashion with program loops and subroutine calls 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 that portion in memory where the table is stored. Iterative procedures refer to common memory locations and array of numbers are confined within a local portion of memory. The result of all these observations is the locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively infrequently.

locality of reference

If the active portions 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. 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 fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory, the aver-

age memory access time will approach the access time of the cache. Although the cache is only a small fraction of the size of main memory, a large fraction of memory requests will be found in the fast cache memory because of the locality of reference property of programs.

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 find the required words in the fast cache memory.

hit ratio

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, 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 best measured 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 average memory access time of a computer system can be improved considerably by use of a cache. If the hit ratio is high enough so that most of the time the CPU accesses the cache instead of main memory, the average access time is closer to the access time of the fast cache memory. For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the 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 when considering the organization of cache memory:

mapping

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 as shown in Fig. 12-10. 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

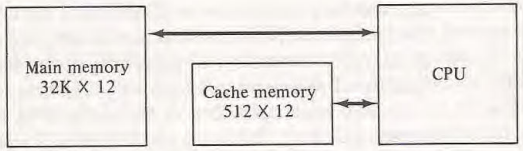


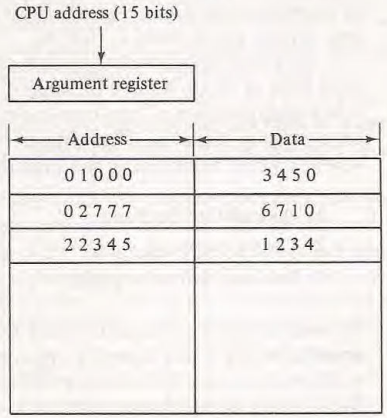
Figure 12-10 Example of cache memory.

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 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 most flexible cache organization uses an associative memory. This organization is illustrated in Fig. 12-11. 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 presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-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

Figure 12-11 Associative mapping cache (all numbers in octal).



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 and not presently in the cache. The decision as to what pair is replaced is determined from the replacement algorithm that the designer chooses for the cache. A simple procedure is to replace cells of the cache in round-robin order whenever a new word is requested from main memory. This constitutes a first-in first-out (FIFO) replacement policy.

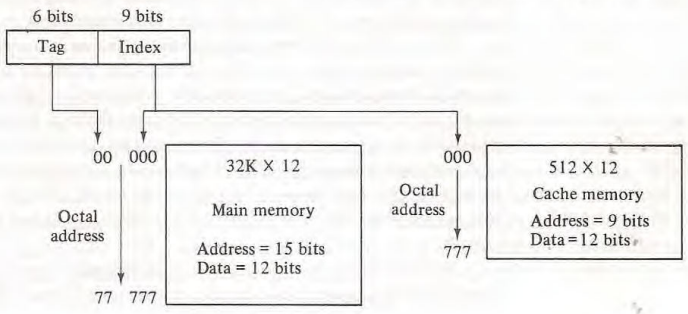
Direct Mapping

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 Fig. 12-12. The CPU address of 15 bits is divided into two fields. The nine least significant 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.

tag field

In the general case, there are 2^k words in cache memory and 2^n words 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 the cache. The internal organization of the words in the cache memory is as shown in Fig. 12-13(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

Figure 12-12 Addressing relationships between main and cache memories.



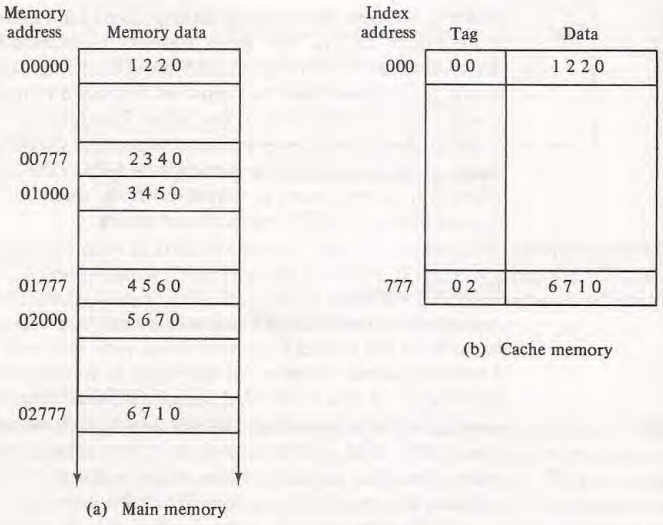


Figure 12-13 Direct mapping cache organization.

tag field of the CPU address is compared with the tag in the word read from the 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. The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have 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 organization operates, consider the numerical example shown in Fig. 12-13. The word at address zero is presently stored in the 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.

The direct-mapping example just described uses a block size of one word. The same organization but using a block size of 8 words is shown in Fig. 12-14.

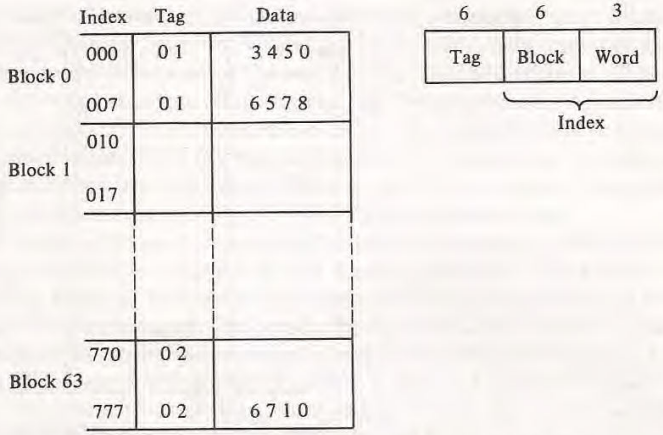


Figure 12-14 Direct mapping cache with block size of 8 words.

The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since $64 \times 8 = 512$. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field. The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

Set-Associative Mapping

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory 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 Fig. 12-15. 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.

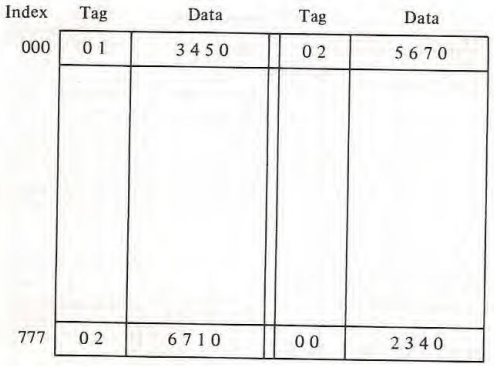


Figure 12-15 Two-way set-associative mapping cache.

The octal numbers listed in Fig. 12-15 are with reference to the main memory contents illustrated in Fig. 12-13(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: thus 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. However, an increase in the set size increases the number of bits in words of cache and requires more complex comparison logic.

replacement algorithms

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first-out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag–data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

Writing into Cache

An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

The simplest and most commonly used procedure is to update main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the *write-through* method. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

write-through

The second procedure is called the *write-back* method. In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache. It is only when the word is displaced from the cache that an accurate copy need be rewritten into main memory. Analytical results indicate that the number of memory writes in a typical program ranges between 10 and 30 percent of the total references to memory.

write-back

Cache Initialization

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, but in effect it contains some nonvalid data. It is customary to include with each word in cache a *valid bit* to indicate whether or not the word contains valid data.

valid bit

The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

12-6 Virtual Memory

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 large computer systems that permit the user to construct programs as though a large

memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct 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 a mapping table.

Address Space and Memory Space

address space
memory space

An address used by a programmer will be called a *virtual address*, and the set of such addresses 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*. Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main 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 a main-memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing $2^{20} = 1024K$ words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example $N = 1024K$ and $M = 32K$.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in Fig. 12-16. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long. (Remember that for

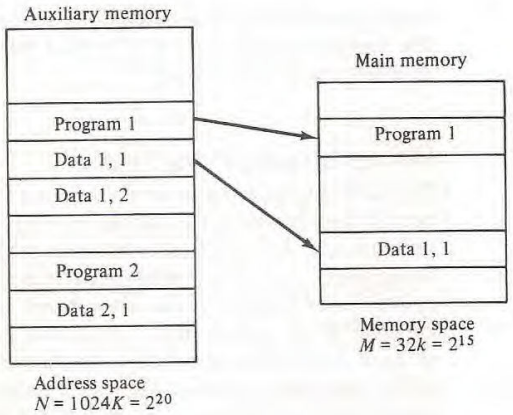
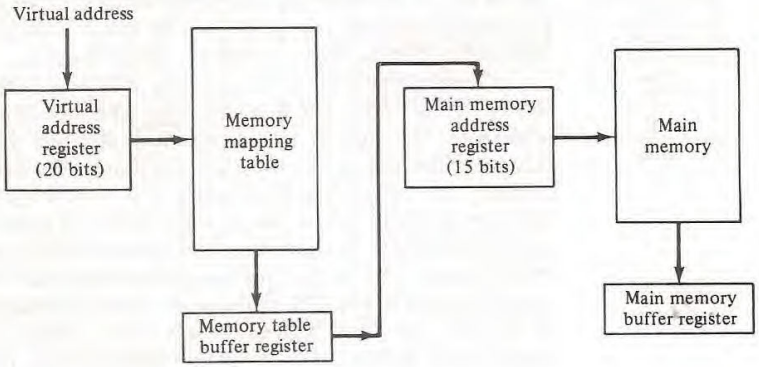


Figure 12-16 Relation between address and memory space in a virtual memory system.

efficient transfers, auxiliary storage moves an entire record to the main memory.) A table is then needed, as shown in Fig. 12-17, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation, which means that every address is translated immediately as a word is referenced by CPU.

The mapping table may be stored in a separate memory as shown in Fig. 12-17 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table

Figure 12-17 Memory table for mapping a virtual address.



takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

Address Mapping Using Pages

pages and blocks

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called *blocks*, which may range from 64 to 4096 words each. The term *page* refers to groups of address space of the same size. 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 32 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. The term "page frame" is sometimes used to denote a block.

page frame

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. 12-18. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig. 12-18, a virtual address has 13 bits. Since each page consists of $2^{10} = 1024$ words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

The organization of the memory mapping table in a paged system is shown in Fig. 12-19. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory. A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table. The content of the

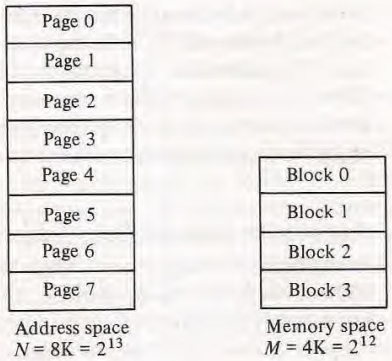
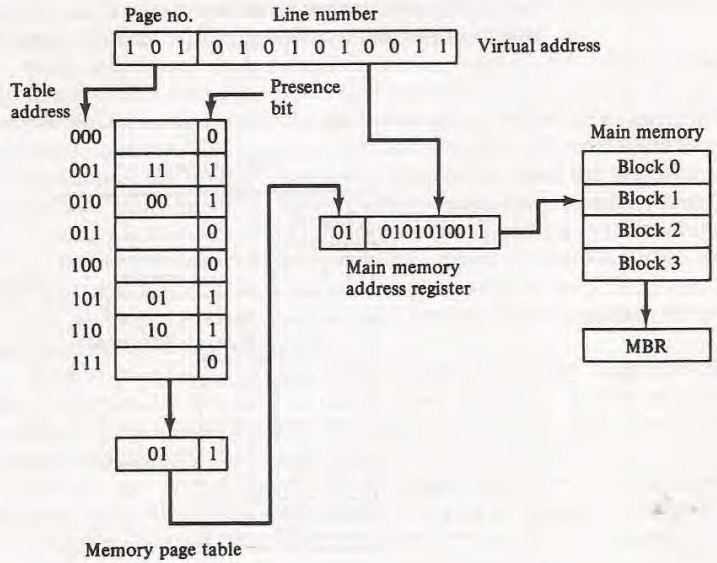


Figure 12-18 Address space and memory space split into groups of 1K words.

word in the memory page table at the page number address is read out into the memory table buffer register. If the presence bit is a 1, the block number thus read is transferred to the two high-order bits of the main memory address register. The line number from the virtual address is transferred into the 10 low-order bits of the memory address register. A read signal to main memory

Figure 12-19 Memory table in a paged system.



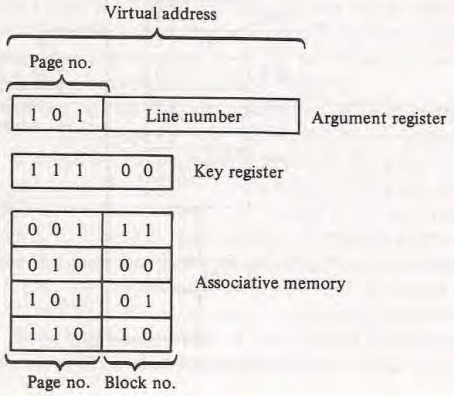
transfers the content of the word to the main memory buffer register ready to be used by the CPU. If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it into main memory before resuming computation.

Associative Memory Page Table

A random-access memory page table is inefficient with respect to storage utilization. In the example of Fig. 12-19 we observe that eight words of memory are needed, one for each page, but at least four words will always be marked empty because main memory cannot accommodate more than four blocks. In general, a system with n pages and m blocks would require a memory-page table of n locations of which up to m blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024K words and memory space of 32K words. If each page or block contains 1K words, the number of pages is 1024 and the number of blocks 32. The capacity of the memory-page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 locations will be empty and not in use.

A more efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding

Figure 12-20 An associative memory page table.



block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the example of Fig. 12-19. We replace the random access memory-page table with an associative memory of four words as shown in Fig. 12-20. Each entry in the associative memory array consists of two fields. The first three bits specify a field for storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument register are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

Page Replacement

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to main memory, and (3) where the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called *page fault*. When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, control is transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

page fault

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the *first-in, first-out* (FIFO) and the *least recently used* (LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks. When a new page must be loaded, the page least recently brought in is removed. The page to be removed is easily determined because its identification number is at the top of the FIFO stack. The FIFO replacement policy has the advantage of being easy to implement. It has the disadvantage that under certain circumstances pages are removed and loaded from memory too frequently.

FIFO

LRU

The LRU policy is more difficult to implement but has been more attractive on the assumption that the least recently used page is a better candidate for removal than the least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called *aging registers*, as their count indicates their age, that is, how long ago their associated pages have been referenced.

12-7 Memory Management Hardware

In a multiprogramming environment where many programs reside in memory it becomes necessary to move programs and data around the 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 many computers. Here we are concerned with the hardware unit associated with the memory management system.

The basic components of a memory management unit are:

1. A facility for dynamic storage relocation that maps logical memory references into physical memory addresses
2. A provision for sharing common programs stored in memory by different users
3. Protection of information against unauthorized access between users and preventing users from changing operating system functions

The dynamic storage relocation hardware is a mapping process similar to the paging system described in Sec. 12-6. The fixed page size used in the virtual

memory system causes certain difficulties with respect to program size and the logical structure of programs. It is more convenient to divide programs and data into logical parts called segments. A *segment* is a set of logically related instructions or data elements associated with a given name. Segments may be generated by the programmer or by the operating system. Examples of segments are a subroutine, an array of data, a table of symbols, or a user's program.

segment

The sharing of common programs is an integral part of a multiprogramming system. For example, several users wishing to compile their Fortran programs should be able to share a single copy of the compiler rather than each user having a separate copy in memory. Other system programs residing in memory are also shared by all users in a multiprogramming system without having to produce multiple copies.

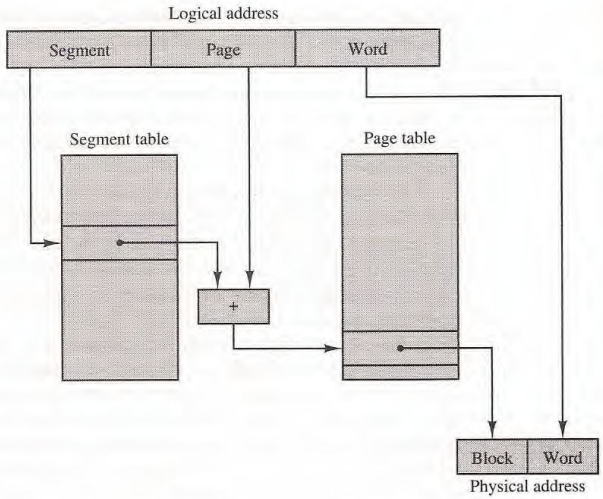
The third issue in multiprogramming is protecting one program from unwanted interaction with another. An example of unwanted interaction is one user's unauthorized copying of another user's program. Another aspect of protection is concerned with preventing the occasional user from performing operating system functions and thereby interrupting the orderly sequence of operations in a computer installation. The secrecy of certain programs must be kept from unauthorized personnel to prevent abuses in the confidential activities of an organization.

logical address

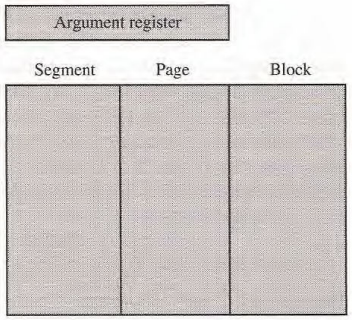
The address generated by a segmented program is called a *logical address*. This is similar to a virtual address except that logical address space is associated with variable-length segments rather than fixed-length pages. The logical address may be larger than the physical memory address as in virtual memory, but it may also be equal, and sometimes even smaller than the length of the physical memory address. In addition to relocation information, each segment has protection information associated with it. Shared programs are placed in a unique segment in each user's logical address space so that a single physical copy can be shared. The function of the memory management unit is to map logical addresses into physical addresses similar to the virtual memory mapping concept.

Segmented-Page Mapping

It was already mentioned that the property of logical space is that it uses variable-length segments. The length of each segment is allowed to grow and 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 Fig. 12-21. 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 word field gives the specific word within the page. A page field of k bits can specify up to 2^k pages. A segment number may be associated



(a) Logical to physical address mapping



(b) Associative memory translation look-aside buffer (TLB)

Figure 12-21 Mapping in segmented-page memory management unit.

with just one page or with as many as 2^k pages. Thus the length of a segment would vary according to the number of pages that are assigned to it.

The mapping of the logical address into a physical address is done by means of two tables, as shown in Fig. 12-21(a). The segment number of the logical address specifies the 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 number given in the logical address. The sum produces a pointer address to an entry in the page table. The value found in the page table provides the block number in physical memory. The concatenation of the block field with the word field produces the final physical mapped address.

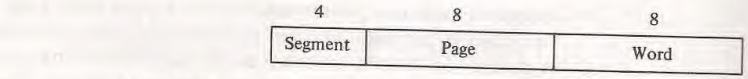
The two mapping tables may be stored in two separate small memories or in main memory. In either case, a memory reference from the CPU will require three accesses to memory: one from the segment table, one from the page table, and the third from main memory. This would slow the system significantly when 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 type of 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 as shown in Fig. 12-21(b). 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 of Fig. 12-21(a) is used and the result transformed into the associative memory for future reference.

Numerical Example

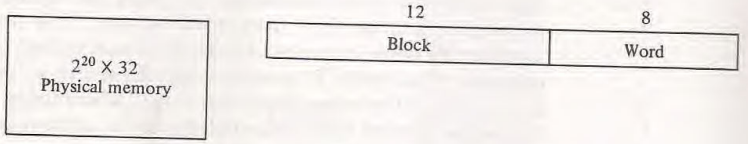
A numerical example may clarify the operation of the memory management unit. Consider the 20-bit logical address specified in Fig. 12-22(a). The 4-bit segment number specifies one of 16 possible segments. The 8-bit page number can specify up to 256 pages, and the 8-bit word field implies a page size 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 or 256 words. The largest possible segment will have 256 pages, for a total of 256 × 256 = 64K words.

The physical memory shown in Fig. 12-22(b) consists of 2^{20} words of 32 bits each. The 20-bit address is divided into two fields: a 12-bit block number and an 8-bit word number. Thus, physical memory is divided into 4096 blocks of 256 words each. A page in a logical address has a corresponding block in physical memory. Note that both the logical and physical address have 20 bits. In the absence of a memory management unit, the 20-bit address from the CPU can be used to access physical memory directly.

Consider a program loaded into memory that requires five pages. The operating system may assign to this program segment 6 and pages 0 through 4, as shown in Fig. 12-23(a). The total logical address range for the program is from hexadecimal 60000 to 604FF. When the program is loaded into physical memory, it is distributed among five blocks in physical memory where the operating system finds empty spaces. The correspondence between each memory block and logical page number is then entered in a table as shown in



(a) Logical address format: 16 segments of 256 pages each, each page has 256 words



(b) Physical address format: 4096 blocks of 256 words each, each word has 32 bits

Figure 12-22 An example of logical and physical addresses.

Fig. 12-23(b). The information from this table is entered in the segment and page tables as shown in Fig. 12-24(a).

Now consider the specific logical address given in Fig. 12-24. The 20-bit address is listed as a five-digit hexadecimal number. It refers to word number 7E of page 2 in segment 6. The base of segment 6 in the page table is at address 35. Segment 6 has associated with it five pages, as shown in the page table at addresses 35 through 39. Page 2 of segment 6 is at address 35 + 2 = 37. The physical memory block is found in the page table to be 019. Word 7E in block 19 gives the 20-bit physical address 0197E. Note that page 0 of segment 6 maps into block 12 and page 1 maps into block 0. The associative memory in Fig.

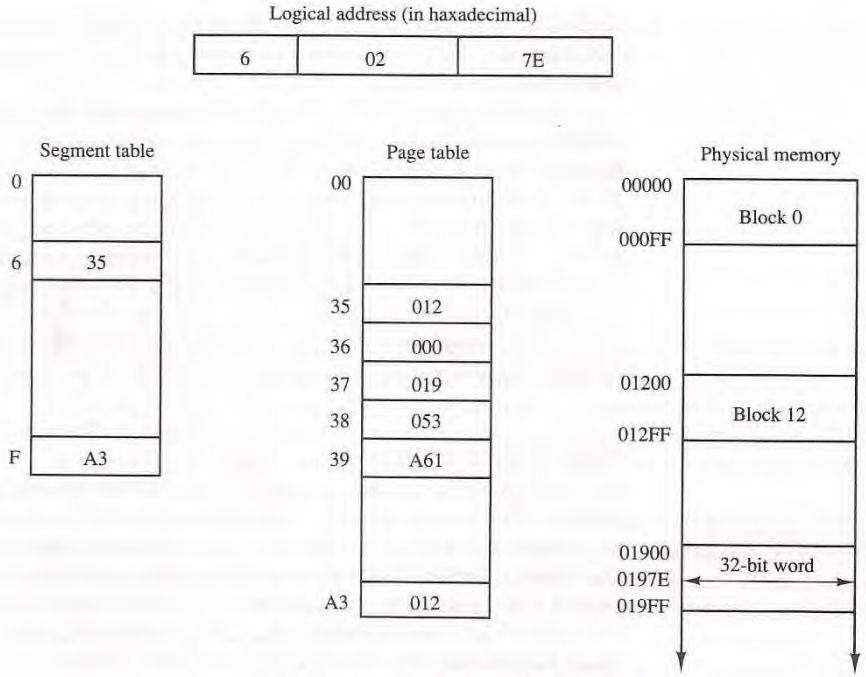
Figure 12-23 Example of logical and physical memory address assignment.

Hexadecimal address	Page number
60000	Page 0
60100	Page 1
60200	Page 2
60300	Page 3
60400 604FF	Page 4

(a) Logical address assignment

Segment	Page	Block
6	00	012
6	01	000
6	02	019
6	03	053
6	04	A61

(b) Segment-page versus memory block assignment



(a) Segment and page table mapping

Segment	Page	Block
6	02	019
6	04	A61

(b) Associative memory (TLB)

Figure 12-24 Logical to physical memory mapping example (all numbers are in hexadecimal).

12-24(b) shows that pages 2 and 4 of segment 6 have been referenced previously and therefore their corresponding block numbers are stored in the associative memory.

From this example it should be evident that the memory management system can assign any number of pages to each segment. Each logical page can be mapped into any block in physical memory. Pages can move to different blocks in memory depending on memory space requirements. The only updating required is the change of the block number in the page table. Segments can grow or shrink independently without affecting each other. Different segments can use the same block of memory if it is required to share a program by many users. For example, block number 12 in physical memory can be assigned a second logical address F0000 through F00FF. This specifies segment number 15 and page 0, which maps to block 12 as shown in Fig. 12-24(a).

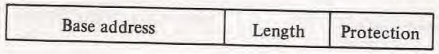
Memory Protection

Memory protection can be assigned to the physical address or the logical address. The protection of memory through the physical address can be done by assigning to each block in memory a number of protection bits that indicate the type of access allowed to its corresponding block. Every time a page is moved from one block to another it would be necessary to update the block protection bits. A much better place to apply protection is in the logical address space rather than the physical address space. This can be done by including protection information within the segment table or segment register of the memory management hardware.

The content of each entry in the segment table or a segment register is called a descriptor. A typical descriptor would contain, in addition to a base address field, one or two additional fields for protection purposes. A typical format for a segment descriptor is shown in Fig. 12-25. The base address field gives the base of the page table address in a segmented-page organization or the block base address in a segment register organization. This is the address used in mapping from a logical to the physical address. The length field gives the segment size by specifying the maximum number of pages assigned to the segment. The length field is compared against the page number in the logical address. A size violation occurs if the page number falls outside the segment length boundary. Thus a given program and its data cannot access memory not assigned to it by the operating system.

The protection field in a segment descriptor specifies the access rights available to the particular segment. In a segmented-page organization, each

Figure 12-25 Format of a typical segment descriptor.



entry in the page table may have its own protection field to describe the access rights of each page. The protection information is set into the descriptor by the master control program of the operating system. Some of the access rights of interest that are used for protecting the programs residing in memory are:

1. Full read and write privileges
2. Read only (write protection)
3. Execute only (program protection)
4. System only (operating system protection)

Full read and write privileges are given to a program when it is executing its own instructions. Write protection is useful for sharing system programs such as utility programs and other library routines. These system programs are stored in an area of memory where they can be shared by many users. They can be read by all programs, but no writing is allowed. This protects them from being changed by other programs.

The execute-only condition protects programs from being copied. It restricts the segment to be referenced only during the instruction fetch phase but not during the execute phase. Thus it allows the users to execute the segment program instructions but prevents them from reading the instructions as data for the purpose of copying their content.

Portions of the operating system will reside in memory at any given time. These system programs must be protected by making them inaccessible to unauthorized users. The operating system protection condition is placed in the descriptors of all operating system programs to prevent the occasional user from accessing operating system segments.

PROBLEMS

12-1. **a.** How many 128 × 8 RAM chips are needed to provide a memory capacity of 2048 bytes?
b. How many lines of the address bus must be used to access 2048 bytes of memory? How many of these lines will be common to all chips?
c. How many lines must be decoded for chip select? Specify the size of the decoders.

12-2. 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 to the address bus.

12-3. A ROM chip of 1024 × 8 bits has four select inputs and operates from a 5-volt power supply. How many pins are needed for the IC package? Draw a block diagram and label all input and output terminals in the ROM.

12-4. Extend the memory system of Fig. 12-4 to 4096 bytes of RAM and 4096 bytes of ROM. List the memory-address map and indicate what size decoders are needed.

12-5. A computer employs RAM chips of 256 × 8 and ROM chips of 1024 × 8. The computer system needs 2K bytes of RAM, 4K bytes of ROM, and four interface units, each with four registers. A memory-mapped I/O configuration is used. The two highest-order bits of the address bus are assigned 00 for RAM, 01 for ROM, and 10 for interface registers.
a. How many RAM and ROM chips are needed?
b. Draw a memory-address map for the system.
c. Give the address range in hexadecimal for RAM, ROM, and interface.

12-6. An 8-bit computer has a 16-bit address bus. The first 15 lines of the address are used to select a bank of 32K bytes of memory. The high-order bit of the address is used to select a register which receives the contents of the data bus. Explain how this configuration can be used to extend the memory capacity of the system to eight banks of 32K bytes each, for a total of 256K bytes of memory.

12-7. A magnetic disk system has the following parameters:

T_s = average time to position the magnetic head over a track

R = rotation speed of disk in revolutions per second

N_t = number of bits per track

N_s = number of bits per sector

Calculate the average time T_a that it will take to read one sector.

12-8. What is the transfer rate of an eight-track magnetic tape whose speed is 120 inches per second and whose density is 1600 bits per inch?

12-9. Obtain the complement function for the match logic of one word in an associative memory. In other words, show that M_i' is the sum of exclusive-OR functions. Draw the logic diagram for M_i' and terminate it with an inverter to obtain M_i.

12-10. 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.

12-11. What additional logic is required to give a no-match result for a word in an associative memory when all key bits are zeros?

12-12. **a.** Draw the logic diagram of all the cells of one word in an associative memory. Include the read and write logic of Fig. 12-8 and the match logic of Fig. 12-9.
b. Draw the logic diagram of all cells along one vertical column (column j) in an associative memory. Include a common output line for all bits in the same column.

c. If a page consists of 2K words, how many pages and blocks are there in the system?

12-20. A virtual memory has a page size of 1K words. There are eight pages and four blocks. The associative memory page table contains the following entries:

Page	Block
0	3
1	1
4	2
6	0

Make a list of all virtual addresses (in decimal) that will cause a page fault if used by the CPU.

12-21. A virtual memory system has an address space of 8K words, a memory space of 4K words, and page and block sizes of 1K words (see Fig. 12-18). The following page reference changes occur during a given time interval. (Only page changes are listed. If the same page is referenced again, it is not listed twice.)

4 2 0 1 2 6 1 4 0 1 0 2 3 5 7

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm used is (a) FIFO; (b) LRU.

12-22. Determine the two logical addresses from Fig. 12-24(a) that will access physical memory at hexadecimal address 012AF.

12-23. 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.

12-24. Give the binary number of the logical address formulated in Prob. 12-23 for segment 36 and word number 2000 in page 15.

REFERENCES

1. Baer, J. L., *Computer Systems Architecture*. Potomac, MD: Computer Science Press, 1980.
2. Dasgupta, S., *Computer Architecture: A Modern Synthesis*, Vol. 1. New York: John Wiley, 1989.
3. Gibson, G. A., *Computer Systems Concepts and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

4. Hamacher, V. C., Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, 3rd ed. New York: McGraw-Hill, 1990.
5. Hwang, K., and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.
6. Kain, R., *Computer Architecture: Software and Hardware*, Vol. 1. Englewood Cliffs, NJ: Prentice Hall, 1989.
7. Langholz, G., J. Francioni, and A. Kandel, *Elements of Computer Organization*. Englewood Cliffs, NJ: Prentice Hall, 1989.
8. Murray, W. D., *Computer and Digital System Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1990.
9. Patterson, D. A., and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1990.
10. Pollard, L. H., *Computer Design and Architecture*. Englewood Cliffs, NJ: Prentice Hall, 1990.
11. Stone, H. S. (ed.), *Introduction to Computer Architecture*, 2nd ed. Chicago: Science Research Associates, 1980.

CHAPTER 11

11-1

	$A_7 - A_2$	$A_1 A_0$
12 =	000011	00
13 =	000011	01
14 =	000011	10
15 =	000011	11

To CS, RS1, RS0

$CS = A_2 A_3 A_4' A_5' A_6' A_7'$

$RS1 = A_1$

$RS0 = A_0$

11-2

Interface	Port A	Port B	Control Reg	Status Reg
# 1	10000000	10000001	10000010	10000011
2	01000000	01000001	01000010	01000011
3	00100000	00100001	00100010	00100011
4	00010000	00010001	00010010	00010011
5	00001000	00001001	00001010	00001011
6	00000100	00000101	00000110	00000111

11-3

Character printer; Line printer; Laser printer; Digital plotter; Graphic display; Voice output; Digital to analog converter; Instrument indicator.

11-5

See text discussion in Sec. 11-2.

11-6

(a) Status command – Checks status of flag bit.

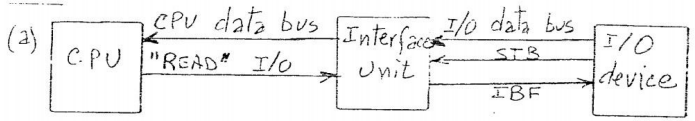
(b) Control command – Moves magnetic head in disk.

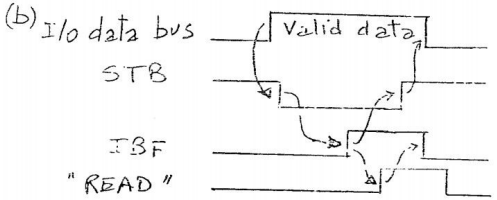
(c) Status command – Cheks if device power is on.

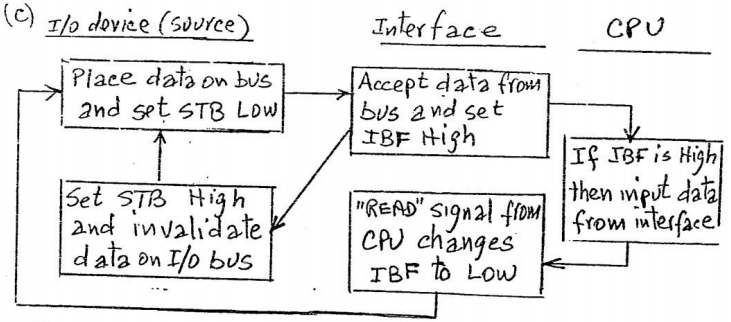
(d) Control command – Moves paper position.

(e) Data input command – Reads value of a register

11-7

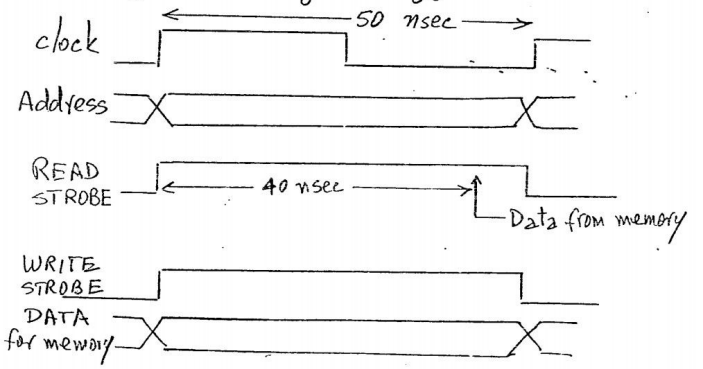




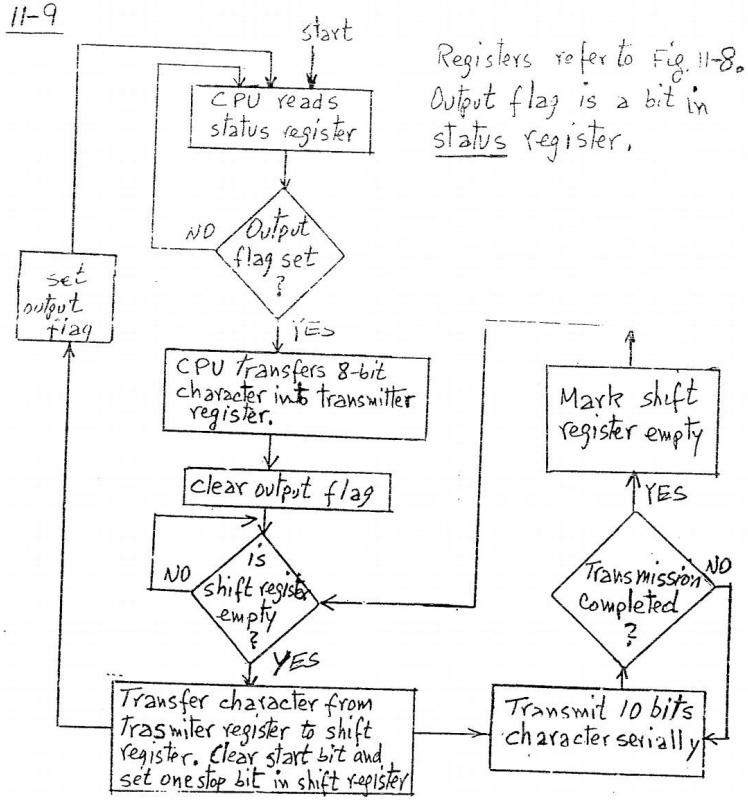


11-8

$20\,MHz = 20 \times 10^6\,Hz \quad T = \frac{10^{-6}}{20} = 50\ nsec.$



11-9



Registers refer to Fig. 11-8.
Output flag is a bit in status register.

11-10

1. Output flag to indicate when transmitter register is empty.
2. Input flag to indicate when receiver register is full.
3. Enable interrupt if any flag is set.
4. Parity error; (5) Framing error; (6) Overrun error.

11-11

10 bits: start bit + 7 ASCII + parity + stop bit.

From Table 11-1 ASCII W = 1010111

with even parity = 11010111

with start and stop bits = 1110101110

11-12

(a) $\frac{1200}{8} = 150$ characters per second (cps)

(b) $\frac{1200}{11} = 109$ cps

(c) $\frac{1200}{10} = 120$ cps

11-13

(a) $\frac{k \text{ bytes}}{(m-n) \text{ bytes/sec}} = \frac{k}{m-n}$ sec.

(b) $\frac{k}{n-m}$ sec. (c) No need for FIFO

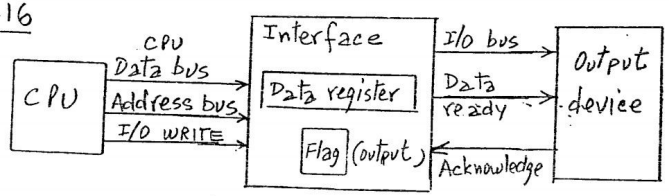
11-14

Initial	F = 0011	Output ← R4
After delete = 1	F = 0010	
After delete = 0	F = 0001	R4 ← R3
After insert = 1	F = 1001	R1 ← Input
(Insert goes to 0)	F = 0101	R2 ← R1
	F = 0011	R3 ← R2

11-15

	Input ready	output ready	F_1-F_4
(a) Empty buffer	1	0	0000
(b) Full buffer	0	1	1111
(c) Two items	1	1	0011

11-16

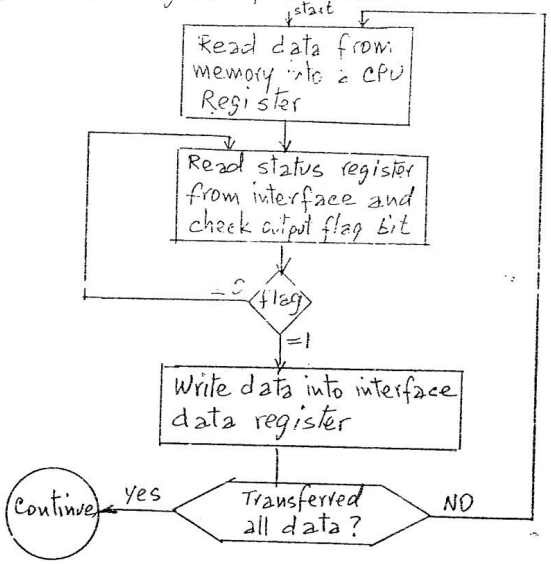


Flag = 0 if data register full (After CPU writes data)

Flag = 1 if data register empty (After the transfer to device)

When flag goes to 0, enable "Data ready" and place data on I/O bus. When "Acknowledge" is enabled, set the flag to 1 and disable "ready" handshake line.

11-17 CPU Program flow chart:



11-18

See text Section 11-4.

11-19

If an interrupt is recognized in the middle of an instruction execution, it is neeessary to save all the information from control registers in addition to processor registers. The state of the CPU to be saved is more complex.

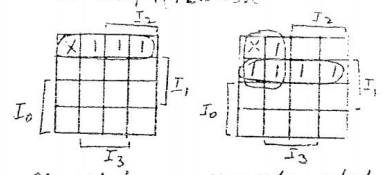
11-20

	Device 1	Device 2
(1) Initially, device 2 sends an interrupt request:	PI=0; PO=0; RF=0	PI=0; PO=0; RF=1
(2) Before CPU responds with acknowledge, device 1 sends interrupt request:	PI=0; PO=0; RF=1	PI=0; PO=0; RF=1
(3) After CPU sends an acknowledge, device 1 has priority:	PI=1; PO=0; RF=1 VAD enable=1	PI=0; PO=0; RF=1 VAD enable=0

11-22 Table 11-2

I_0	I_1	I_2	I_3	X Y	IST
1	x	x	x	0 0	1
0	1	x	x	0 1	1
0	0	1	x	1 0	1
0	0	0	1	1 1	1
0	0	0	0	x x	0

Map simplification



$X = I_0'I_1'$ $\quad Y = I_0'I_1 + I_0'I_2'$

11-23

Same as Fig. 11-14. Needs 8 AND gates and an 8×3 decoder.

11-24

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	X Y 3	IST
1	x	x	x	x	x	x	x	0 0 0	1
0	1	x	x	x	x	x	x	0 0 1	1
0	0	1	x	x	x	x	x	0 1 0	1
0	0	0	1	x	x	x	x	0 1 1	1
0	0	0	0	1	x	x	x	1 0 0	1
0	0	0	0	0	1	x	x	1 0 1	1
0	0	0	0	0	0	1	x	1 1 0	1
0	0	0	0	0	0	0	1	1 1 1	1
0	0	0	0	0	0	0	0	x x x	0

(e)

Binary	hexadecimal
1010 0000	A0
1010 0100	A4
1010 1000	A8
1010 1100	AC
1011 0000	B0
1011 0100	B4
1011 1000	B8
1011 1100	BC

11-25

$76 = (01001100)_2$

Replace the six 0's by 010011, xy

11-26

Set the mask bit belonging to the interrupt source so it can interrupt again.

At the beginning of the service routine, check the value of the return address in the stack. If it is an address within the source service program, then the same source has interrupted again while being serviced.

11-21

The service routine checks the flags in sequence to determine which one is set. The first flag that is checked has the highest priority level. The priority level of the other sources corresponds to the order in which the flags are checked.

11-27

When the CPU communicates with the DMA controller, the read and write lines are used as inputs from the CPU to the DMA controller.

When the DMA controller communicates with memory, the read and write lines are used as outputs from the DMA to memory.

11-28

(a) CPU initiates DMA by transferring:

256 to the word count register.

1230 to the DMA address register.

Bits to the control register to specify a write operation.

(b)

1. I/O device sends a "DMA request."
2. DMA sends BR (bus request) to CPU.
3. CPU responds with a BG (bus grant).
4. Contents of DMA address register are placed in address bus.
5. DMA sends "DMA acknowledge" to I/O device and enables the write control line to memory.
6. Data word is placed on data bus by I/O device.
7. Increment DMA address register by 1 and decrement DMA word count register by 1
8. Repeat steps 4-7 for each data word transferred.

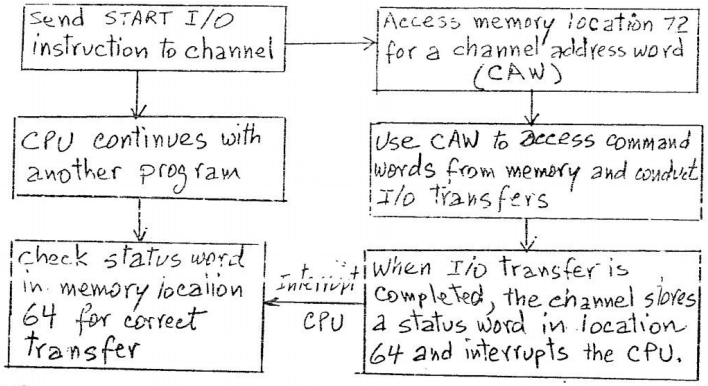
11-29

CPU refers to memory on the average once (or more) every 1 μsec. ($1/10^6$). Characters arrive one every $1/2400 = 416.6$ μsec. Two characters of 8 bits each are packed into a 16-bit word every $2 \times 416.6 = 833.3$ μsec. The CPU is slowed down by no more than $(1/833.3) \times 100 = 0.12\%$.

11-30

The CPU can wait to fetch instructions and data from memory without any damage occuring except loss of time. DMA usually transfers data from a device that cannot be stopped since information continues to flow so loss of data may occur.

11-31



11-32

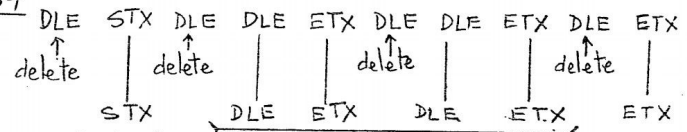
There are 26 letters and 10 numerals.

$26 \times 26 + 26 \times 10 = 936$ possible addresses.

11-33

The processor transmits the address of the terminal followed by ENQ (enquiry) code 00000101. The terminal responds with either ACK (acknowledge) or NAK (negative acknowledge) or the terminal does not respond during a timeout period. If the processor receives an ACK, it sends a block of text.

11-34



11-35

32 bits between two flags; 48 bits including the flags.

11-36

Information to be sent (1023): 0 11111111111

After zero insertion, information transmitted: 0 111110 111110 (delete)

Information received after 0's deletion: 0 11111 11111

CHAPTER 12

12-1 (a) $\frac{2048}{128} = 16$ chips

(b) $2048 = 2^{11}$ — 11 lines to address 2048 bytes
$128 = 2^7$ — 7 lines to address each chip
4 lines to decoder for selecting 16 chips

(c) 4×16 decoder

12-2

(a) 8 chips are needed with address lines connected in parallel.

(b) $16 \times 8 = 128$ chips. Use 14 address lines ($16K = 2^{14}$)
10 lines specify the chip address.
4 lines are decoded into 16 chip-select inputs.

12-3

10 pins for inputs, 4 for chip-select, 8 for outputs, 2 for power.
Total of 24 pins.

12-4

$4096/128 = 32$ RAM chips ; $4096/512 = 8$ ROM chips.
$4096 = 2^{12}$ — There 12 common address lines + 1 line to select between RAM and ROM.

Component	Address	16 15 14 13	12 11 10 9	8 7 6 5	4 3 2 1
RAM	0000-0FFF	0 0 0 0	↔ 5×32 decoder	x x x	x x x x
ROM	1000-1FFF	0 0 0 1	↔ 3×8 decoder	x	x x x x x x x x

to $\overline{CS2}$

12-5

RAM $2048/256 = 8$ chips ; $2048 = 2^{11}$; $256 = 2^8$
ROM $4096/1024 = 4$ chips ; $4096 = 2^{12}$; $1024 = 2^{10}$
Interface $4 \times 4 = 16$ registers ; $16 = 2^4$

Component	Address	16 15 14 13	12 11 10 9	8 7 6 5	4 3 2 1
RAM	0000-07FF	0 0 0 0	↔ 3×8 decoder	x x x x	x x x x
ROM	4000-4FFF	0 1 0 0	↔ 2×4 decoder	x x	x x x x x x x x
Interface	8000-800F	1 0 0 0	0 0 0 0	0 0 0 0	x x x x

12-6

The processor selects the external register with an address 8000 hexadecimal. Each bank of 32K bytes are selected by addresses 0000-7FFF. The processor loads an 8-bit number into the register with a single 1 and 7 0's. Each output of the register selects one of the 8 banks of 32K bytes through a chip-select input. A memory bank can be changed by changing the number in the register.

12-7 Average time = $\frac{1}{2}$ time for half revolution + time to read a sector.

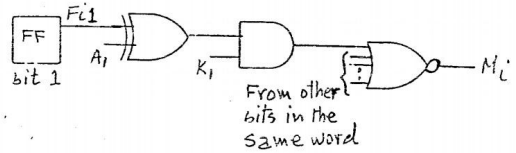
$$T_a = T_s + \frac{1}{2R} + \frac{n_s}{N_t} \times \frac{1}{R}$$

12-8 An eight-track tape reads 8 bits (one character) at the same time. Transfer rate = $1600 \times 120 = 192{,}000$ characters/s

12-9

From Sec. 12-4: $M_i = \prod_{j=1}^{n} [(A_j \oplus F_{ij})' + K_j']$

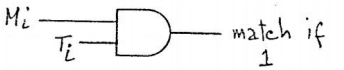
$$M_i' = \sum_{j=1}^{n} (A_j \oplus F_{ij}) K_j$$



12-10

A match occurs if $T_i = 1$

match = $M_i T_i$

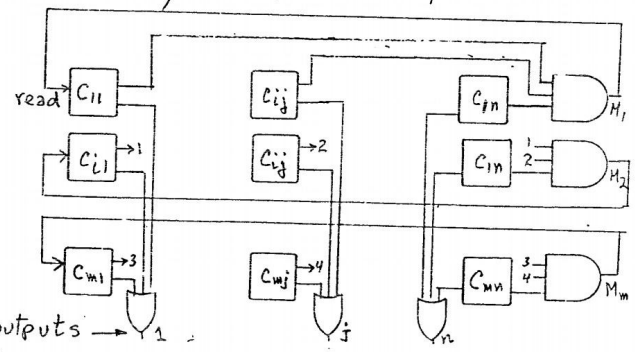


12-11

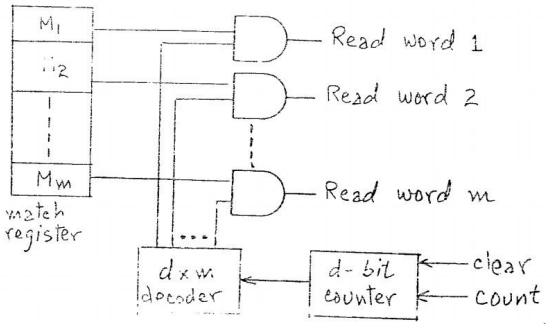
$$M_i = \left(\prod_{j=1}^{n} A_j F_{ij} + A_j' F_{ij}' + K_j' \right) \cdot (K_1 + K_2 + K_3 + \cdots + K_n)$$

At least one key bit K_i must be equal to 1

12-12(c)



12-13



A d-bit counter drives a d-to-m line decoder where $2^d = m$ (m = No. of words in memory). For each count, the M_i bit is checked and if 1, the corresponding read signal for word i is activated.

12-14

Let $x_j = A_j F_{ij} + A_j' F_{ij}'$ (argument bit = memory word bit)

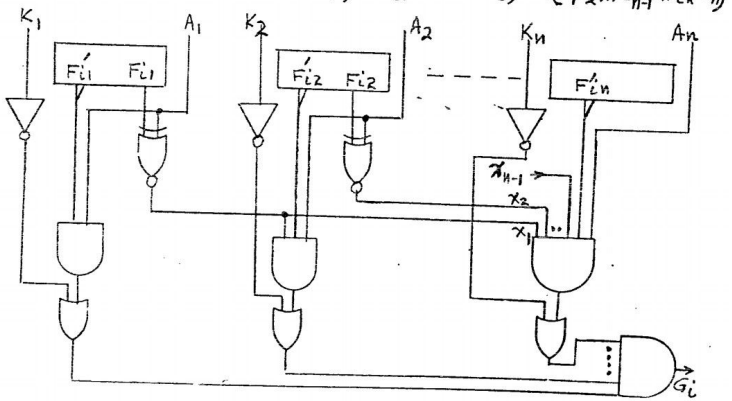
Output indicator $G_i = 1$ if:

$A_1 F_{i1}' = 1$ and $K_1 = 1$ (First bit in A = 1 while $F_{i1} = 0$)

or if $x_1 A_2 F_{i2}' = 1$ and $K_2 = 1$ (First pair of bits are equal and second bit in A = 1 while $F_{i2} = 0$)

... etc.

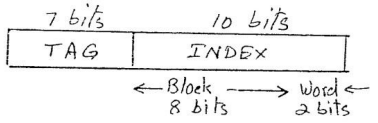
$$G_i = (A_1 F_{i1}' + K_1')(x_1 A_2 F_{i2}' + K_2')(x_1 x_2 A_3 F_{i3}' + K_3') \ldots (x_1 x_2 \ldots x_{n-1} A_n F_{in}' + K_n')$$



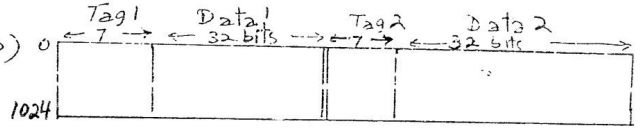
12-15

$128K = 2^{17}$; For a set size of 2, the index address has 10 bits to accomodate $\frac{2048}{2} = 1024$ words of cache.

(a)



(b)



Size of cache memory is $1024 \times 2(7+32)$
$= 1024 \times 78$

12-16

(a) $0.9 \times \underbrace{100}_{\text{cache access}} + 0.1 \times \underbrace{11000}_{\text{cache+memory access}} = 90 + 110 = 200$ nsec.

(b) $0.2 \times \underbrace{1000}_{\text{write access}} + 0.8 \times \underbrace{200}_{\text{read access from (a)}} = 200 + 160 = 360$ nsec.

(c) Hit ratio $= 0.8 \times 0.9 = 0.72$

12-17

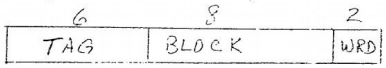
Sequence: A B C D B E D A C E C E

	LRU → 3	2	1	0
Count value =				
Initial words =	A	B	C	D
B is a hit	A	C	D	B
E is a miss	C	D	B	E
D is a hit	C	B	E	D
A is a miss	B	E	D	A
C is a miss	E	D	A	C
E is a hit	D	A	C	E
C is a hit	D	A	E	C
E is a hit	D	A	C	E

12-18

64K × 16 : 16 bit address; 16-bit data.

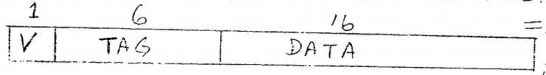
(a)



=16 bit address

INDEX = 10 bit cache address.

(b)



=23 bits in each word of cache

(c) 2^8 = 256 blocks of 4 words each

12-19

(a) Address space = 24 bits 2^{24} = 16M words

(b) Memory space = 16 bits 2^{16} = 64K words

(c) $\frac{16M}{2K}$ = 8K pages $\frac{64K}{2K}$ = 32 blocks

12-20

The pages that are not in main memory are:

Page	Address	address that will cause fault
2	2K	2048 - 3071
3	3K	3072 - 4095
5	5K	5120 - 6143
7	7K	7168 - 8191

12-21

4 2 0 1 2 6 14 0 10 2 3 5 7

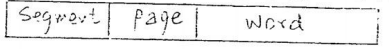
Page reference	(a) Pages in main memory	First-in Contents of FIFO	(b) Pages in memory	LRU	Most recently used
Initial	0124	4201	0124	4201	
2	0124	4201	0124	4012	
6	0126	2016	0126	0126	
1	0126	2016	0126	0261	
4	0146	0164	1246	2614	
0	0146	0164	0146	6140	
1	0146	0164	0146	6401	
0	0146	0164	0146	6410	
2	1246	1642	0124	4102	
3	2346	6423	0123	1023	
5	2345	4235	0235	0235	
7	2357	2357	2357	2357	

12-22

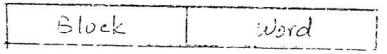
600AF and F00AF

12-23

Logical address:



Physical address:



12-24

Segment 36 = $(0100100)_2$ (7-bit binary)

Page 15 = $(01111)_2$ (5-bit binary)

Word 2000 = $(011111010000)_2$ (12-bit binary)

Logical address = 010 0100 01111 011111010000

(24-bit binary)