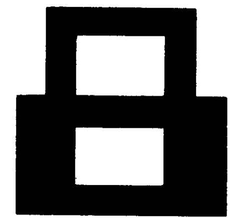


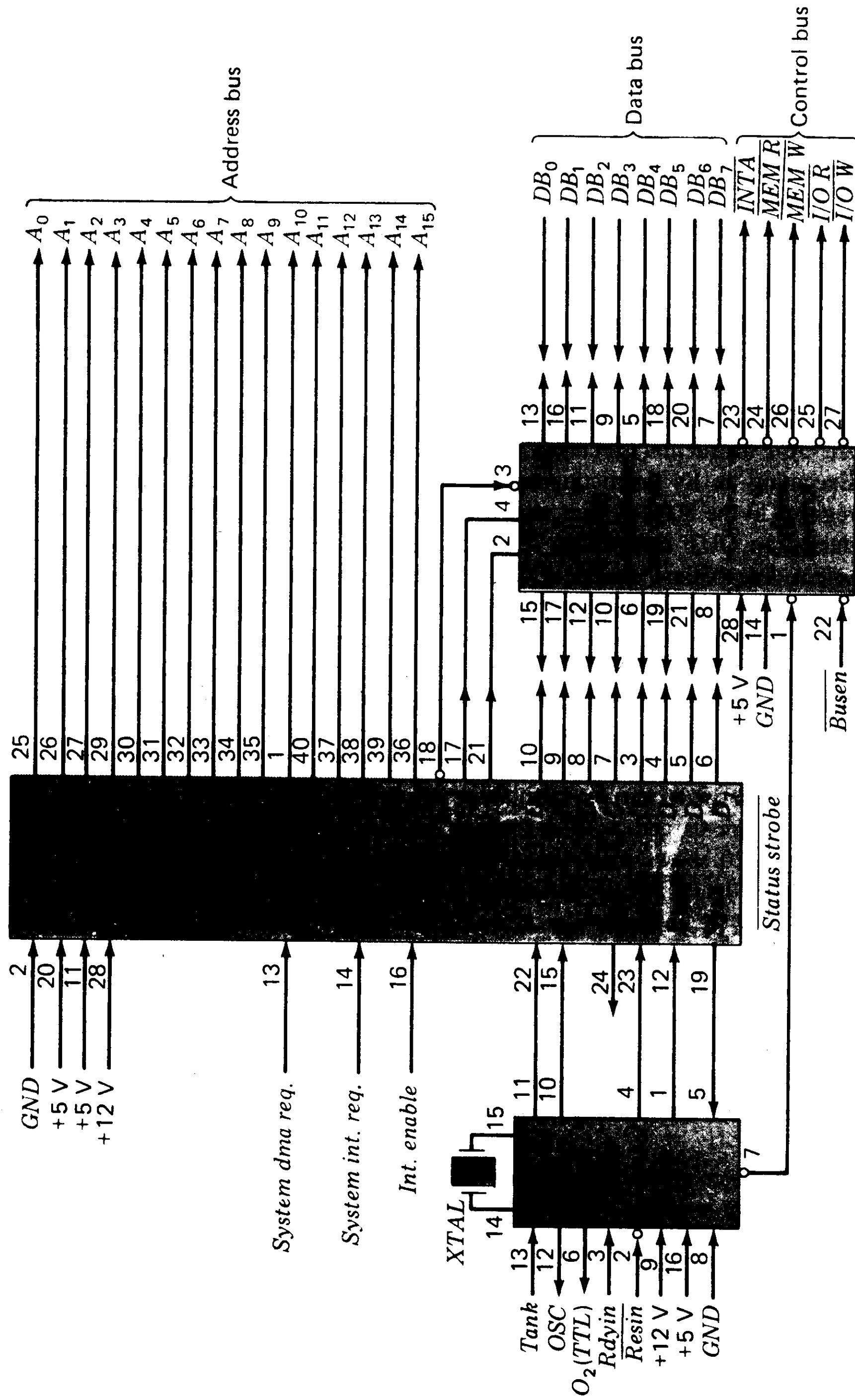
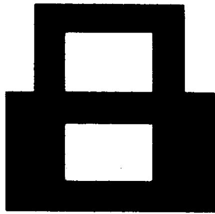


FIGURE 8.19

8080 bus. (*Intel Corp.*)



The 8080 system selects an input-output device as follows:

1 The device number of the selected device is placed on address lines A_7 to A_0.

2 If the device is to be read from by the 8080 bus, $\overline{\text{I/O R}}$ (which is normally 1) is made a 0. While $\overline{\text{I/O R}}$ is a 0, the selected device to be read from places its data on D_7 to D_0. When $\overline{\text{I/O R}}$ goes back to its normal 1 state, the selected device removes the data from lines D_7 to D_0.[13]

If the 8080[14] wishes to output data to a device, it places the device's number on A_7 to A_0. Then it places the data to be output on D_7 to D_0 and makes $\overline{\text{I/O W}}$, which is normally 1, a 0. The selected device then reads these data from the bus.

The reading and writing operations for the 8080 are under program control. An OUT instruction executed by the 8080 causes the outputting of data to a device. Executing an IN instruction causes a device to be read from. The accumulator register in the 8080 system receives data during an IN instruction and sends data during an OUT instruction. If an IN instruction is executed, the data from the selected device are read onto D_7 to D_0 and from there into the accumulator. If an OUT instruction is executed, the data are read from the 8080 system's accumulator onto D_7 to D_0, and then the selected device accepts the data on D_7 to D_0. (This accumulator is the same accumulator used for arithmetic operations such as those described in Chap. 5. The internal operation of the 8080 microprocessor is covered in Chap. 10. Section 10.11 covers program operation.)

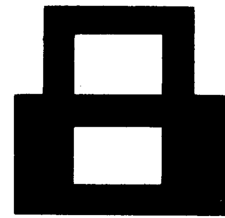
An interface design for the keyboard of Fig. 7.18 is shown in Fig. 8.20. The keyboard is given the device number 1, or binary 00000001. Therefore the lines A_7 to A_1 are 0s and A_0 is a 1 when the keyboard is selected. The NAND gate in Fig. 8.20 shows these inputs to be NANDed along with I/O R. Now, when I/O R is a 1($\overline{\text{I/O R}}$ a 0), the 8080 bus is saying, "Place the selected device's data on D_7 to D_0." In this design, if A_7 to A_0 contain 00000001 and $\overline{\text{I/O R}}$ is a 0, then the output of the NAND gate becomes a 0. This enables the tristate drivers connected to K_7 to K_0, the keyboard output from the flip-flops in Fig. 7.19. As a result, the values of K_7 to K_0 are placed on bus lines D_7 to D_0 where the 8080 bus can read them (into its accumulator).

Notice that the output of the NAND gate is normally a 1, which disables the tristate drivers so that they have high impedance and write nothing on bus lines D_7 to D_0.

A major question now arises: At any given time the operator of the keyboard may or may not have depressed a key, so that the keyboard may or may not have new information for the 8080. If the keyboard is simply read, the 8080 cannot tell whether the character supplied is new or old. (The same key could be pressed twice in succession.) To compensate for this, a system is used in which a *keyboard status word* can be read by the 8080 bus which will tell whether a new character

[13]This $\overline{\text{I/O R}}$ line corresponds to the $\overline{\text{READ}}$ line in Fig. 8.13; the $\overline{\text{I/O W}}$ line corresponds to the $\overline{\text{WRITE}}$ line in that figure.

[14]We refer to the 8080 microprocessor chips as simply the 8080, as is common practice.



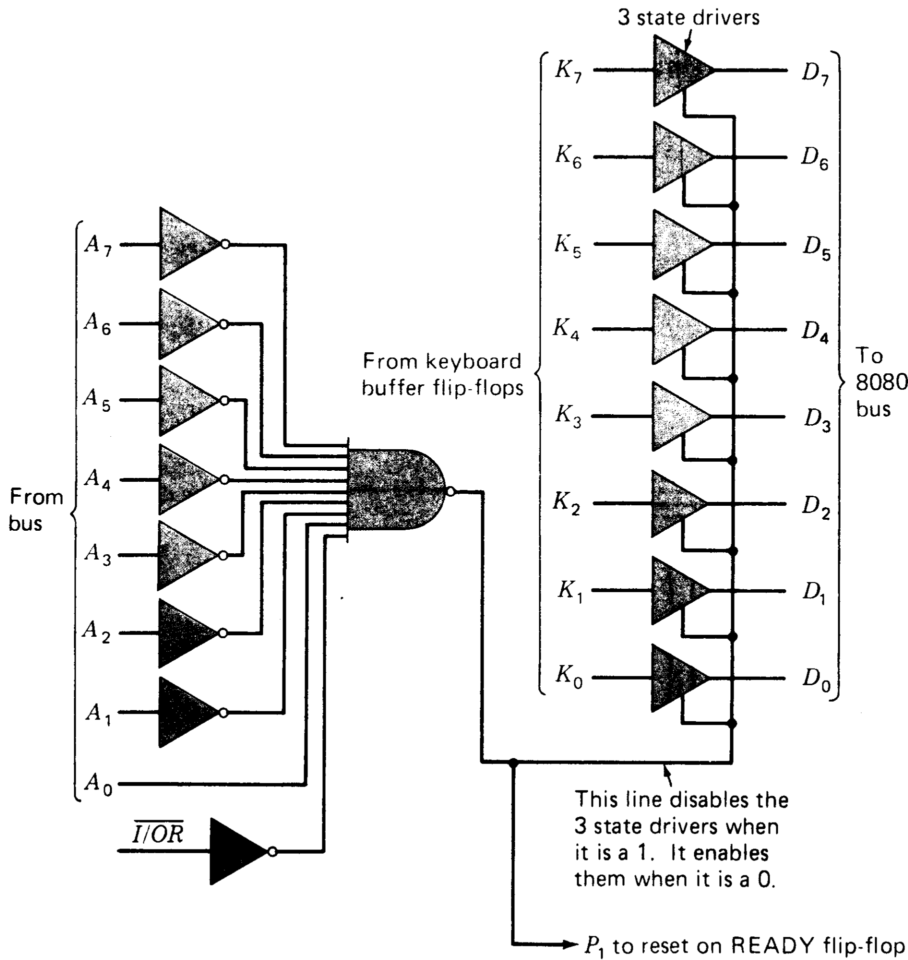


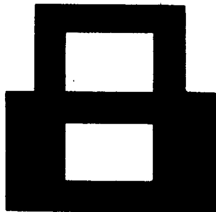
FIGURE 8.20

Interfacing a keyboard to the 8080 microprocessor.

is ready to be read from the keyboard. The scheme used here is the one most used for this kind of interface.

Figure 8.21 shows the *status word generator* interface for the keyboard. We have given this keyboard status word generator the device number 2. The keyboard status word is used as follows. If a new character is available from the keyboard, the keyboard status word will have a 1 in the D_7 position. If there is no new keyboard character, a 0 will be in the D_7 position. The remaining D_0 to D_6 of the keyboard status word will always be 0s.

The interface operates as follows. The program in the 8080 system reads the status word (an IN instruction is executed). The accumulator now contains the status word, and the program sees whether it has a 1, in which case the keyboard should be read. If the status word is all 0s, the program goes on to other programs or devices or, if it has nothing else to do, simply continues to read the status word until a 1 is found.



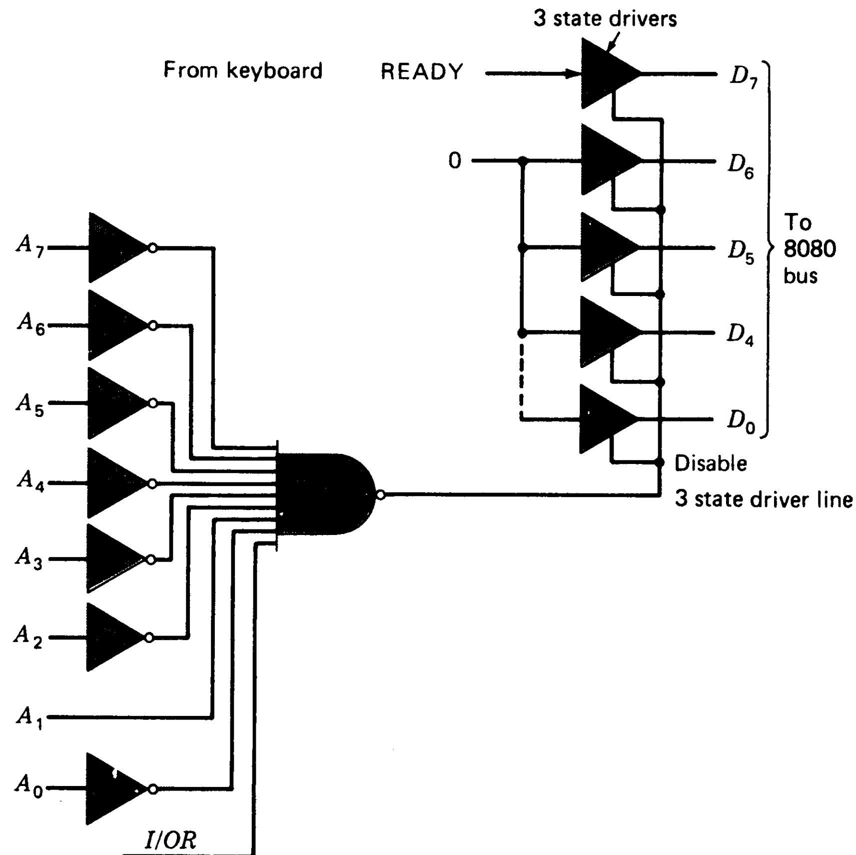


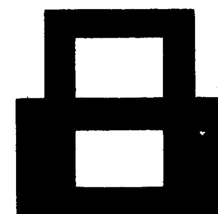
FIGURE 8.21

Keyboard status word generator.

The operation of the keyboard status word interface is shown in Fig. 8.21. When a key is depressed, the READY flip-flop is set to a 1, as shown in Fig. 7.19. Therefore when the $\overline{\text{I/O R}}$ is made a 0, indicating a device read, and the device number on A_7 to A_0 is 00000010, the NAND gate output in Fig. 8.21 goes to a 0, enabling the tristate devices so that a 10000000 is placed on D_7 to D_0, indicating that the keyboard is ready to be read.

When the keyboard is read, the READY flip-flop is cleared (reset) by the signal generated in Fig. 8.20. Therefore, if keyboard status words are read in the interval between when the keyboard has been read and when a key is depressed, the output on D_7 to D_0 will be all 0s.

The described use of a status register in the interface circuitry to give the status of an input-output device to the CPU is the most widely used technique for interfacing of this sort. In more complicated input-output devices, such as disk memories, there are more status bits in the status word which have a meaning, and these bits are set and reset by the processor and disk controller as operations are sequenced.



PROGRAM CONTROL OF KEYBOARD INTERFACE

***8.6**[15] The interface design for the keyboard is intended to be under program control. Thus a section of the program in the microprocessor will examine the keyboard status register to see whether the keyboard has data, if it does, data are read from the keyboard.

Table 8.1 shows a section of program for the 8080 microprocessor which will read from a keyboard. The 8080 system has an 8-bit *byte* at each address in memory. Each OP (operation) code, which tells what the instruction is to do, is a single byte in memory. There is an IN instruction with OP code 11011011 (binary), which tells the microprocessor to read from an input-output device. The number of the device (device code) immediately follows the IN instruction's OP code in the next byte.

In Table 8.1 the presentation of the program listing is arranged as follows. The program in assembly language is to the right. The program as actually stored is in the two left columns, which lists addresses in memory followed by the contents of each address in hexadecimal. The Label column lists names for locations in the memory, enabling programs to use names in memory instead of actual numeric addresses.

For example, this program starts at location 030 in memory. At this location is the value DB, the OP code for the IN instruction. The comments (to the right) are always preceded by a slash; the assembler ignores these comments.

The location 030 in memory is given the name KEYSTAT in the Label column.

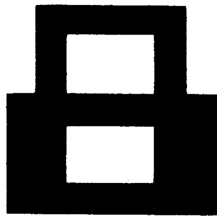
In location 031 there is the device number 2; therefore, the microprocessor will read location 030, find the IN instruction OP code, and read location 031, finding in it the device number 2. The microprocessor will then place the value 2 on the address lines and issue an input-output device read sequence on the bus.

This will result in the status register interface placing 00000000 on the data lines if there is no character to be read from the keyboard and placing 10000000

[15]Sections with asterisks can be omitted without loss of continuity.

TABLE 8.1

LOCATION IN MEMORY	CONTENTS	ASSEMBLY LANGUAGE LABEL	OP CODE	OPERAND	COMMENTS
030	DB	KEYSTAT	IN	2	/READ STATUS WORD
031	02				/INTO ACCUMULATOR
032	E6		ANI	80H	/AND ACCUMULATOR BITS
033	80				
034	CA		JZ	KEYSTAT	/JUMP BACK IF ZERO
035	30				
036	00				
037	DB		IN	1	/READ KEYBOARD
038	01				



if there is a character. This value will be read by the microprocessor into its accumulator, completing the instruction.

The next instruction is an ANI instruction with OP code E6. The ANI instruction performs a bit-by-bit AND of the byte following the instruction, in this case 10000000 (binary), with the accumulator. If the keyboard is ready to be read, this will result in a 1 in the leftmost position; if not, a 0.

The ANI instruction also sets a flip-flop called Z (for zero) in the 8080 to a 1 if the results of the AND contain a 1 and a 0 if not. Therefore, if a character is ready to be read, Z will contain a 1; if not, it will contain a 0.

The JZ is the OP code for a "jump-on-zero" instruction in the 8080. If the Z flip-flop is a 0, the microprocessor will take its next instruction word from the address given in the bytes[16] following the JZ; if Z is a 1, the instruction following these 2 bytes will be executed. As a result, if a character is ready to be read, the microprocessor will read the IN instruction 037 next; if no character is ready, the microprocessor will jump back to location 030. Notice that the programmer has used the label KEYSTAT instead of giving the numeric value in the address part of the instruction, but the actual address appears in the Contents column. (The assembler determined the location.) Also, note that a complete address in the 8080 requires 2 bytes (2^{16} words can be used in memory). The lower-order (least significant) bits come first in an instruction word, followed by the higher-order bits.

When the keyboard is to be read, the instruction word beginning at location 037 will be executed. This is an IN instruction, but the device number is 1, so the keyboard itself will be read from.

When the instruction is executed, the 8080 will place the device number 1 on its address lines and then generate a device read sequence of control signals. As a result, the keyboard interface will place the character in the keyboard buffer register on the data lines, and this character will be read into the 8080 accumulator, ending the read process.

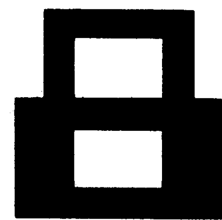
INTERFACING A PRINTER

8.7 The preceding sections have detailed the reading of data from a keyboard into a microprocessor (CPU). We now examine outputting characters from a microprocessor into a printer.

We assume that the printer uses an ASCII character in 8-bit parallel form to cause the printing of a single character. In 8080 interfacing, first the printer is selected. To do this, since different output devices may be connected to the microprocessor, the printer is given a unique device number, and we assume that the number is 3 (decimal). When the printer is selected, this number will appear on the microprocessor address lines A_7 to A_0 in binary.

Figure 8.22 shows an interface design. A NAND gate and six inverters are connected so that the NAND gate will have a 0 output only when the number 3 appears on A_7 to A_0 and $\overline{\text{I/O W}}$ is a 0. This NAND gate's output is used as a GO signal, which ultimately causes the printer to print the character on data lines D_7 to D_0. The $\overline{\text{I/O W}}$ signal is pulled negative (to a 0) when the character to be printed

[16]The 8080 system address has 2^{16} words of memory; thus 2 bytes are required for a complete address.



From 8080 bus: D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7

D CL

C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7 To printer

From bus: A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0

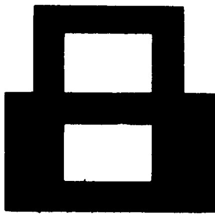
$\overline{I/O\ W}$ From 8080 bus

1 D CL Character Ready To printer

Printer must reset this flip-flop after character is printed

FIGURE 8.22

Interfacing a printer.



is available on D_0 to D_7 and the device address (3 in this case) is on lines A_7 to A_0.

A flip-flop called *character ready* is used to signal the printer that a character is ready to be printed. The printer must read this flip-flop and then print the character.

The program instruction which causes this character transfer in the 8080 is called an OUT instruction. The OUT instruction occupies two 8-bit bytes in memory, with the second byte containing the device number. When the OUT instruction is executed, the contents of the accumulator are placed on D_0 to D_7. Execution of the OUT instruction causes the printer to print a character corresponding to whatever code was stored in the accumulator.

The above implies that the computer program in the 8080 memory has previously stored the ASCII character for the character to be printed in the accumulator. (A load-accumulator instruction, described in Chap. 10, will effect this. For now we restrict our discussion to the interface strategy.)

There is a basic problem with the above scheme. A printer is a very slow electromechanical device, and the microprocessor, because of its high speed, is capable of flooding the printer with characters which it cannot possibly print. An attempt to print only after a pause between each character will be difficult to implement because the printer may require different time intervals to respond to different characters.

There are two basic solutions to this problem. One is to have the microprocessor examine the printer at regular intervals to see when a new character can be printed. If the printer can print, it "raises a flag" (turns on a flip-flop) which the microcomputer reads. If the flag is a 1, the microcomputer outputs a character to be printed; if the flag is a 0, the microcomputer goes back to what it was doing and then examines the printer again at a later time. (The computer may simply continue to examine the flag until it goes on.)

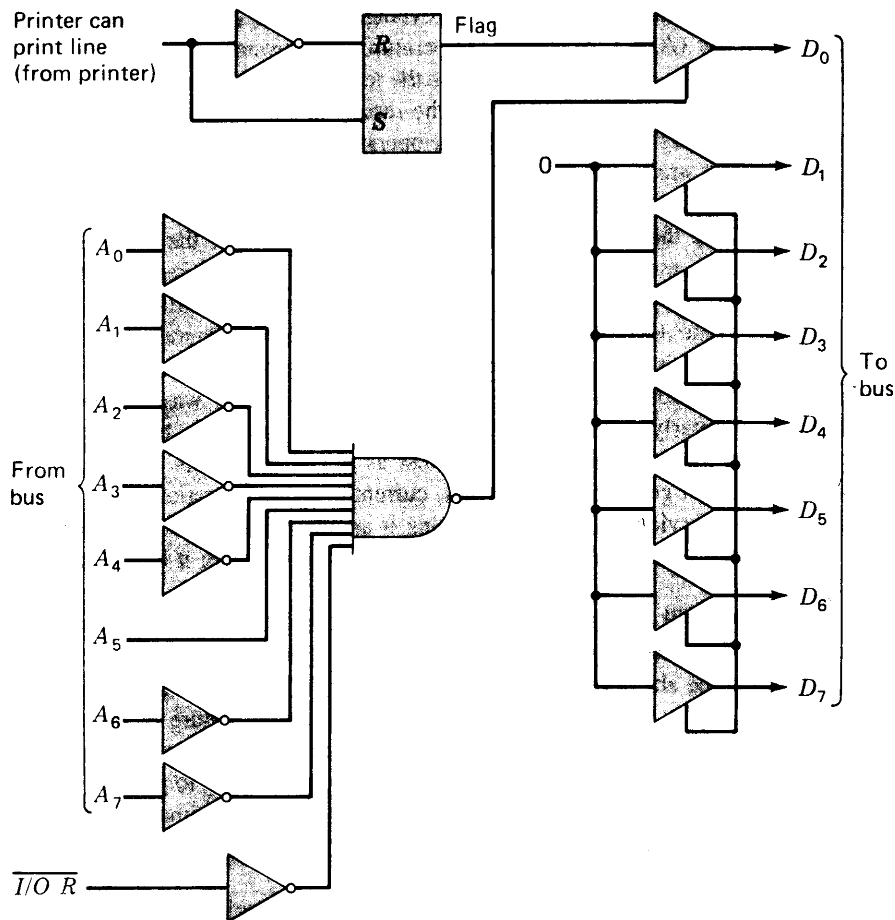
The other solution is to have the printer signal the computer with an INTERRUPT line whenever it is able to print. The computer then services this interrupt by feeding the printer a character.

We use the first technique in our example and explain interrupts in the following section.

All that is required to respond to a query from the 8080 microprocessor is shown in Fig. 8.23. When the printer is clear and able to handle a character, it sets the flag flip-flop on. The flag is then made a bit in a status register of 8 bits.

The program step to read the flag involves transferring an entire 8-bit character placed on the data lines from the status registers into the accumulator. The status register is given device number 4. When an IN instruction with device number 4 is executed by the microprocessor, the number 4 comes up on A_7 to A_0, and finally the $\overline{\text{I/O R}}$ line is brought low. This causes the transfer of the flag and its associated 0s into the microprocessor accumulator. Another instruction must then examine the accumulator to see whether it is all 0s or contains a 1. If the accumulator sign bit is a 1, the printer is ready for a character; if not, the computer must wait.

The above interfacing technique is widely used because of its simplicity of implementation. Using a flag (or several flags) to determine an output device's status, placing the flag(s) in a status register, and then reading the status register using a program are a standard computer interface technique.



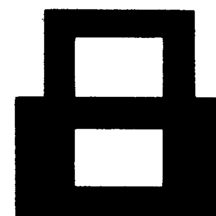


FIGURE 8.23

Printer status generator.

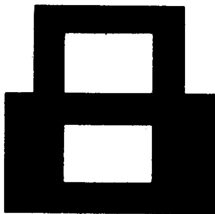
INTERRUPTS IN INPUT-OUTPUT SYSTEMS

8.8 The preceding examples showing how to interface a keyboard and a printer demonstrated a technique in which the program was used to examine flags in status registers to see whether an input-output device either had information or could accept information. This technique is widely used, particularly in microcomputer and minicomputer systems where not too many external devices are to be interfaced and sufficient time is available that the program can continually test the devices to see whether they are ready. Stepping from status register to status register by using a program to see which device is ready is called *polling* the I/O devices.

In many cases, however, there will be too many devices for this scheme to be successful. Or there will be a great amount of computation to be performed, so that continually taking time out to examine the status of input-output devices cannot be tolerated.

To deal with this problem, computers have *interrupt systems* for input-output devices, in which a given device can cause the program operation to be interrupted long enough for the input-output device to be serviced.

The operation of such a system can best be shown by an example. Suppose



that we have a computer system with a keyboard, a printer, and an input from an A-to-D converter (ADC) measuring temperature in a physics experiment. A great deal of computation is required to process the temperature reading from the ADC. The operator of the keyboard examines the results of the computation, which are printed on the printer, and occasionally the operator comments, using the keyboard. These comments are to be printed by the printer along with the temperature and the results of the calculation.

In this case the keyboard inputs are made infrequently, the printer is kept quite busy, and we assume that the A-to-D inputs are read at fairly frequent intervals.

The interrupt system works as follows. The computer normally is processing the inputs from the ADC. Each time a key on the keyboard is depressed, however, an interrupt signal is generated by the keyboard, the program in operation is interrupted, the keyboard is serviced, and the program which was interrupted is returned to. Similarly, a short list of characters to be printed may be stored in the computer, and the program adds to this list as it gathers results. Whenever the printer can print, it generates an interrupt, current program operation is interrupted long enough to service the printer by giving it another character to print, and the original program operation then continues at the point at which it had been interrupted.

The ADC will also generate interrupts which must be serviced by reading the output, and the readings are processed as soon as time is available.

To effect the above, there are some features an interrupt system should have. For instance, it may be necessary to turn off the interrupt feature of the printer, since when there is nothing to print, the printer would simply generate many time-consuming interrupts. (It can always print when there is nothing to print.) It might be necessary to turn off the entire interrupt system for a short time, since during servicing of the keyboard, an interrupt from the printer might cause an interrupt of an interrupt.

In order to examine the interrupt feature more closely, we note that the following things must be done each time an interrupt is generated:

1 The state of the program in operation when the interrupt is executed must be saved. Then the program can be reentered when the interrupt servicing program is finished.

2 The device that generated the interrupts must be identified.

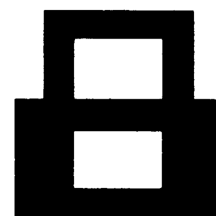
3 The CPU must jump to a section of the program that will service the interrupt.

4 When the interrupt has been serviced, the state of the program which was interrupted must be restored.

5 The original program's operation must be reinitiated at the point at which it was interrupted.

Discussion of how the interrupted program is handled and how returns are made to this program is deferred to Chap. 10 since more information is required about program execution. The mechanism for interrupt generation and identifying the device that wishes to be serviced can be dealt with here, however.

The interrupts are initiated by a device placing a 1 on an interrupt wire in



the bus. This notifies the CPU that a device wishes to be serviced. The CPU then completes the instruction it is executing and transfers control to a section of program designed to service the interrupt.

In the 6800 microprocessor and in the 6150, for example, the various devices are polled by examining the status registers, each in turn, until the interrupting device is located. This device is then serviced.

In the 8080 microcomputer and in the PDP-11 minicomputer, for example, the place in memory where the service program is located for the particular device that generated the interrupt is read into the CPU by the interrupting device. This is called a *vectored interrupt*. In effect, the device tells the CPU "who did it" and does not wait to be asked.

There can be a problem when several devices generate a 1 signal on the interrupt wire at the same time. If the devices are polled, the polling order determines who gets serviced first, and a device not serviced will continue to interrupt until it is serviced. For the vectored interrupt, however, if two devices attempted to write their identifier into the CPU at the same time, they might overwrite each other, so a scheme must be devised by which only one device tells the CPU whom to service. This is accomplished by chips[17] external to the CPU, which set a priority on the devices that can interrupt and handle only the highest priority device with its interrupt on.

More details on interrupts are given in the sections on particular computers in Chap. 10.

When microcomputer systems become larger and more peripheral devices are used, the interface design problem increases. To be efficient and use the microprocessor CPU chip to its utmost ability, it is necessary to use an interrupt system for peripheral devices so that the CPU is not burdened with polling peripherals continuously. To remove the load of servicing peripherals from the CPU, several microprocessor manufacturers produce separate chips that are I/O processors and that work closely with the CPU in handling peripheral servicing. Other important chips now produced to facilitate peripheral handling convert serial input signals (like the ASCII signals in Fig. 7.21) to parallel and place the parallel form on the bus as well as converting from parallel to serial (to drive some printers and modems). Chips are also produced to aid in interrupt processing, including the selection of the highest-priority peripheral demanding service, etc.

Figure 8.24 shows a block design of a system based on the 8086 microprocessor chip.[18] (This system is similar to IBM's personal computer system which uses the 8088.) Examination of this layout reveals how chips are assembled to interface devices in larger systems.

Figure 8.24 shows the parts of the 8086 microprocessor: (1) An 8234 clock generator generates the clock signal for the 8086 and is involved in resets. (2) The 8081 I/O processor (IOP) handles interrupts for the 8086 (the 8086 can be interfaced without this chip, it simply takes some I/O processing load from the 8088).

---

[17]The IC packages used range from gate arrays which examine and allocate priority to programmable interface controllers which contain ROMs with programs for the specific interfaces to be implemented.

[18]Intel (and others) also make an 8086 microprocessor chip which would interface like this chip except that the 8086 handles data 16 bits at a time while the 8088 has 8-bit data paths, and so a 16-bit data bus would be used for the 8086.

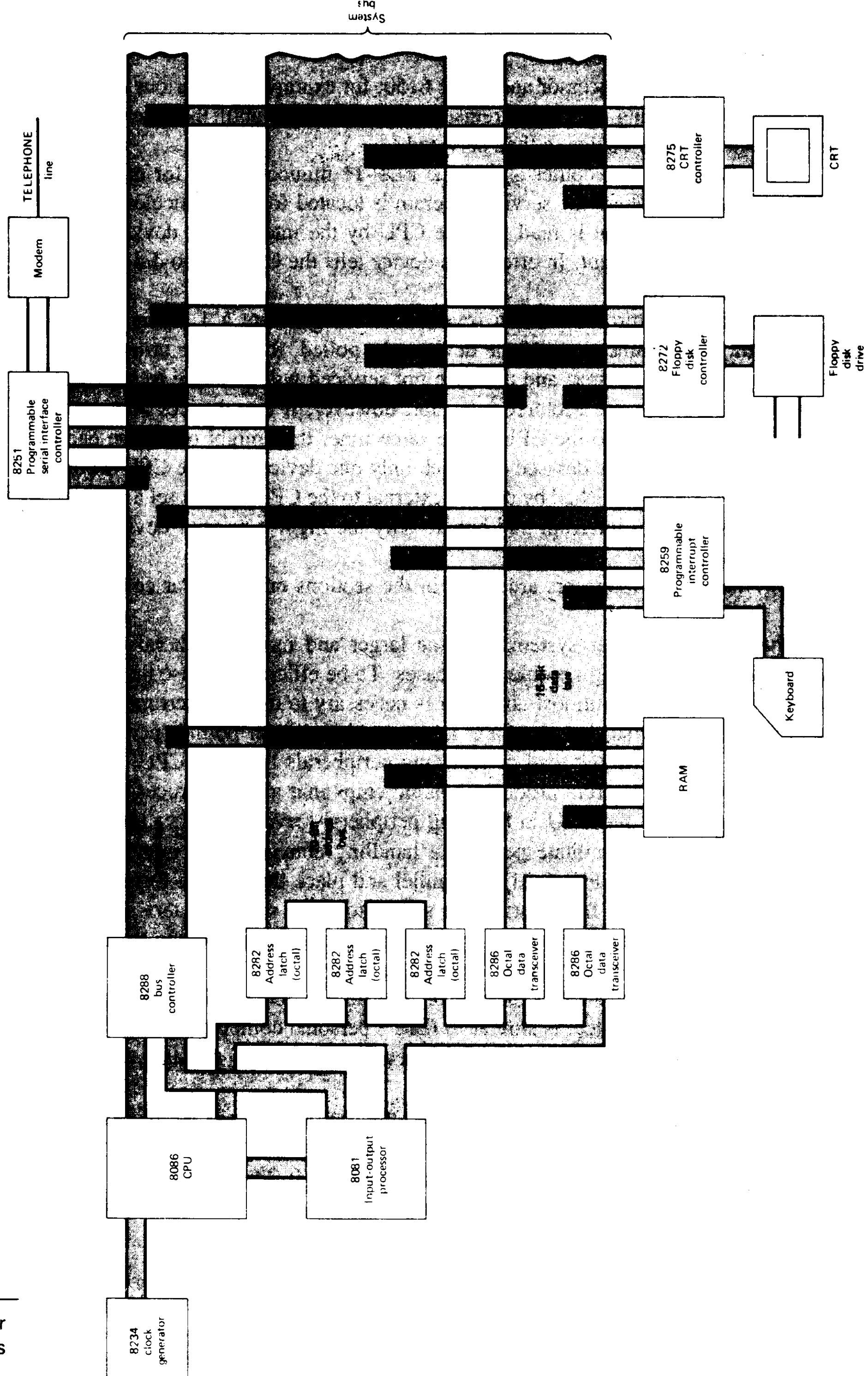
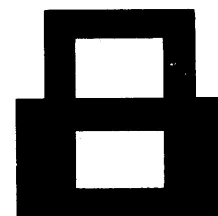


FIGURE 8.24

8086 microcomputer system showing bus layout.



(3) The 8288 bus controller buffers control signals and handles the time multiplexing of the control signals (the 8086 uses the same lines for address lines and control lines, producing control signals to tell which is being output at a given time). (4) The 8282 latches simply hold the address information and provide tristate drivers for the bus. This is necessary for two reasons: First, the 8086 outputs are not capable of driving many other circuits; second, the 8086 time-multiplexes its address signals, and the latches are used to hold the address while control signals are output on the same lines. (5) The 8286 transceivers simply provide tristate drivers and considerable drive capability for the data bus and receivers to read from the Data bus. (6) The 8259 is a programmable interrupt handler which examines demands for service from the peripherals, determines the highest-priority demand, and then interrupts the 8086-8089 combination and outputs a "vector" telling which peripheral requires service. (7) The 8259 programmable parallel interface controller handles keyboard interrupts. The term *programmable* means that the input-output configuration and data-handling functions of the chip are set up by means of data transfers from the 8086 under program control. (8) The 8251 programmable serial interface controller is used to handle serial data and to provide control signals for modems. This chip is programmable and such parameters as (*a*) the number of stop bits in a character transmission (refer to Fig. 7.21, the RS.232C interface it supports), (*b*) the speed of transmission and reception (controlled by a clock and a divide action), (*c*) whether parity is odd or even, etc., are all controlled by a register in the 8251 which is loaded from a program over the data lines by the 8086. (9) The floppy disk and CRT controllers are special interface chips made to service particular devices.

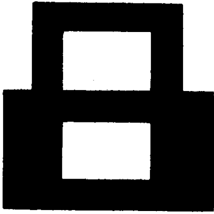
A STANDARD BUS INTERFACE

8.9 There has been some attempt to standardize on buses, particularly by such organizations as the IEEE and the National Bureau of Standards. Most buses which have become standards have been developed by computer and electronic concerns and have been used and adopted by several manufacturers before the standards organization have developed an official document.

Note that these bus standards do not simply have pin connections and line operation procedures, but also specify connectors and printed-circuit board sizes. Thus a system built around one of the buses can add printed-circuit boards containing more memory, interfacing for I/O devices, etc., as long as the board and bus operation for the board meet the standard specifications. So when the organization in Fig. 7.20 is used, the connectors, and boards, and interfaces are all prescribed by the standard.

A widely used bus and its protocol, which has been developed for interfacing instruments and microcomputers, are briefly outlined here. This bus, often called the *general-purpose interface bus* (GPIB), is described in IEEE Standard 488-1978, which is a microcomputer bus standard.

Figure 8.25 shows the basic interface and bus lines which can be used to interconnect a number of modules. Each bus line performs at least one interface function, depending on the interface capabilities. The specifications for cable construction and connectors are given in the standard.



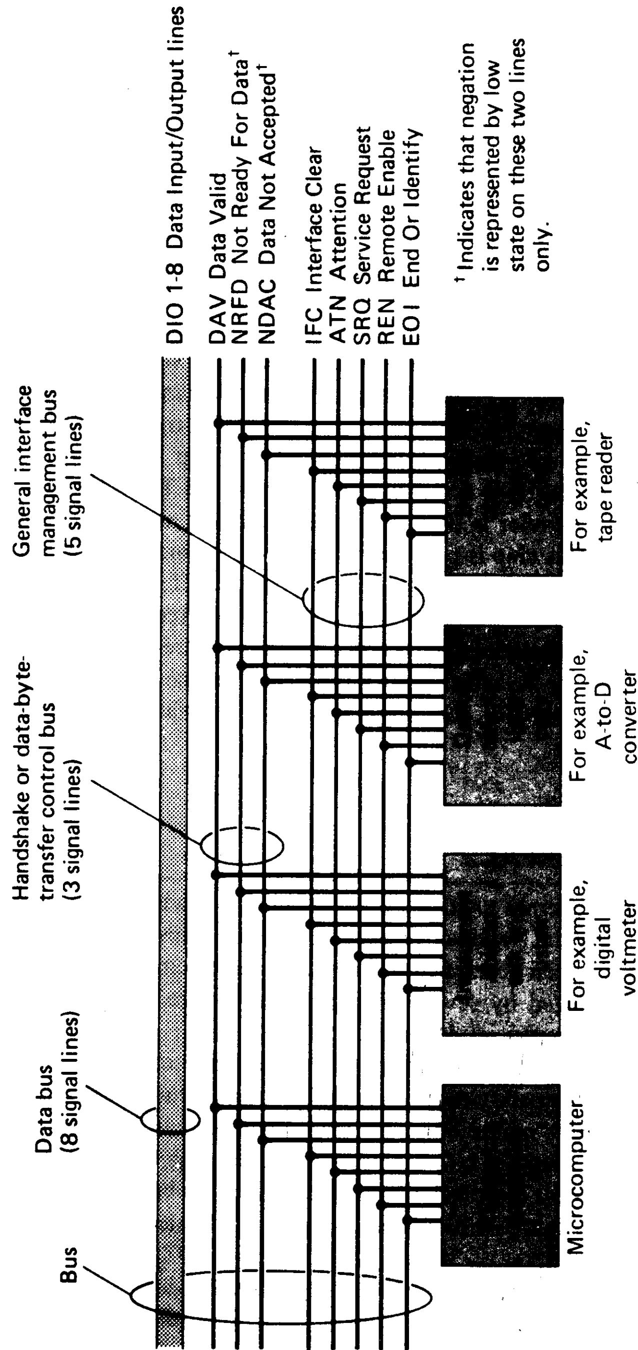
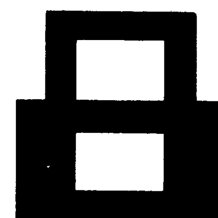


FIGURE 8.25

International standard bus.



At a given time, any particular module connected to the bus may be idle, monitoring the activity on the bus or functioning as (1) a talker, (2) a listener, or (3) a controller. As a talker, a module sends data over the bus to a listener (or listeners). As a listener, a module receives such data. As a controller, it directs the flow of data on the bus, mainly by designating which modules are to send data and which are to receive data.

Notice that the bus consists of 16 signal lines, grouped functionally into three component buses. The *data bus* (eight lines) is used to transfer data in parallel from talkers to listeners; it also transfers certain commands from the controller to subordinate modules. The *transfer bus* (three lines) is used for the handshaking process by which a talker or controller can synchronize its readiness to receive data. The *general interface management bus* (five lines), as its name suggests, is used principally by the controller.

Each system must have one module to designate listeners and talkers, and this module is called a *controller*. The controller uses a group of commands, referred to as *interface messages,* to direct the other modules on the bus in carrying out their functions of talking and listening.

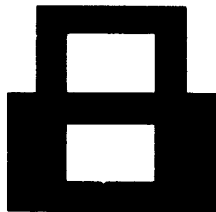
Normally the controller would be the CPU of a computer, and this unit would generate the command signals on the bus to the other modules, which would then respond. Because this interface is designed to handle a large number of different types of modules, the specification is reasonably complicated and general. The basic procedure for generating a transfer of data on this bus is as follows. First, the controller designates a listener by placing the listener's address (each listener is given a 5-bit address) on the data bus and raising the appropriate control lines. Then a talker is designated by placing the talker's address on the data lines and raising the appropriate control lines. Finally, the talker and listener are told to proceed, and the talker places data, 8 bits at a time, in parallel on the data lines.

In the transfer of data from talker to listener, certain basic problems arise in the operation of every bus.[19] These problems are solved by a handshaking procedure whereby talker and listener interact using the control lines. It is convenient to describe this procedure with a flowchart, as shown in Fig. 8.26. This diagram shows that three control lines, called DAV, NRFD, and NDAC (defined in Fig. 8.25), are used to control each data byte transfer. The talkers and receivers each raise and lower the control signals, as shown by the flowchart, and the talker places data on the data bus at the appropriate time.

The necessary control circuitry to implement this handshaking and the other required functions must be provided by each module's interface circuitry. It is possible to design a line of input-output equipment including instruments, tape punches, etc., and to interface each to the same bus by using the interface specification. IC manufacturers often furnish single chips made to provide the necessary logic for an interface.

---

[19]These problems concern how the listener knows when data are on the bus and how the talker knows when the listener has received the data. The bus described here is an *asynchronous bus*. Microprocessors and minicomputers often use synchronous buses where one wire in the bus contains a clock and the clock signal is used to time data transfers. In these systems, the talker must place the data on the data wires, and the listener must be ready to receive data when the clock edge arrives. A synchronous system is faster and simpler but less flexible.



8.6 Explain how a keyboard with an 8-bit output would be interfaced to the 68000 bus.

8.7 Draw waveforms for the 68000 bus for an 8-bit transfer of data to a printer.

8.8 The 8086 configuration in Fig. 8.24 is fairly complex. Discuss the benefits of such a configuration for a personal computer, and contrast it with a microprocessor used as a traffic light controller or an automobile ignition control system.

8.9 Explain the handshake on an asynchronous bus.

8.10 Design an interface for a 256-word 8-bit memory using the chip in Fig. 6.10 for the bus timing in Fig. 8.14.

8.11 Design an interface for a 256-word 8-bit memory using the chip in Fig. 6.10 for the bus timing in Fig. 8.13.

8.12 In the status register scheme used to interface a microprocessor to a keyboard, only 1 bit is used to determine the status of the keyboard. A status register could have several status bits, however, each with a different meaning. Discuss the use of the AND instruction to test various bits in conjunction with the JUMP instruction for the 8080.

8.13 The single status bit used in the printer interface status register is set on and off by the printer. It could be controlled by the printer and the interface. Explain how the interface would work in this case.

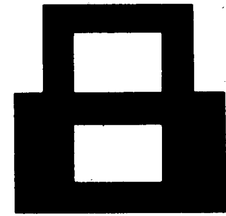
8.14 In an interface such as a printer there is a question as to how the interface should notify the printer when the character to be printed is on the signal wires, and how long the signal should be held there. There are two approaches:

(1) The printer must read the information within a stipulated time. In this case signals with data are placed on the interface wires (the interface device address having already been placed there) and are always held for some fixed time which is acceptable to all the interface circuitry used.

(2) The device being read into notifies the interface when it has received the characters. In this case another interface wire is used, and a signal is placed on this wire by the device being read into when it has accepted the input data. This is a handshake procedure where the interface device address is placed on the wires, data are then placed on the wires, and a wire to the device is raised which says, "The data are on the lines." The interface device then raises another wire, saying, "The data have been accepted."

IBM uses the first technique in its 3081 interfaces, whereas the IEEE (and several other standards organizations and computer manufacturers) use the handshaking technique. Discuss the advantages and disadvantages of each technique.

8.15 With the IEEE 488 interface it is possible to read into several devices at the same time. In this case the system controller places the data on the wires and then raises the wire, showing that the data are there. In responding, the devices accepting data use the *open-collector* circuit shown in Appendix C so that if any single device has not yet accepted the data, the response wire will be set to low.



Show how the open-collector circuit in Fig. C.5(*a*) ANDs and can only indicate acceptance of data by moving the line to 1.

8.16 Show how the circuitry in Fig. 8.20 can be modified to interface a keyboard with address 6 (device number 6).

8.17 Show how the program in Table 8.1 would be modified to service a keyboard with device number 8 and status register number 7.

8.18 Write a sequence of instructions which will read a keyboard and then print the characters read on a printer. Give the keyboard device number 5 and the printer device number 7. Number the status registers as you please.

8.19 Design an interface that will accept serial bit strings using ASCII and the teletypewriter serial format shown in Fig. 7.21. The interface should buffer this bit string of characters into the 8080 microprocessor.

8.20 Design an interface that will take a parallel data byte from an 8080 microprocessor bus and convert it to serial for a teletypewriter.

8.21 Explain handshaking on a bus when data are transferred from a sender to a receiver. How can this be used to prevent errors due to signal skew caused by signals on different wires arriving at different times (skewed) because of the differences in line length and characteristics and differences in delays through IC line drivers, etc.?

8.22 For the standard instrument interface, draw the signals DAV, NRFD, and NDAC for a data transfer from a talker to a listener. Assume that there are no problems in transferring data, and indicate who is raising and lowering each signal.

8.23 For the standard instrument interface, indicate how the controller selects a talker and a listener.

8.24 How is signal skew handled on the standard instrument interface?

8.25 Explain how peripheral devices interrupt a computer with a single bus organization.

8.26 Explain the meaning of *direct memory access* (DMA) and why it is desirable in some cases.

8.27 Can you think of any problems that might arise in multiprocessor systems?

8.28 If devices and status registers are numbered 1, 2, 4, 8, ... and only a few are used (less than or equal to the number of address wires), the gate to determine which device is selected in an interface can be simplified (or omitted). Show why.

8.29 Show in flowchart form the procedure for testing and finding which switch is closed, if any, for the encoding scheme described in Question 7.43.

8.30 How many steps would it take to scan an entire keyboard for the ASCII given in Fig. 7.7, using the two-dimensional keyboard scheme?

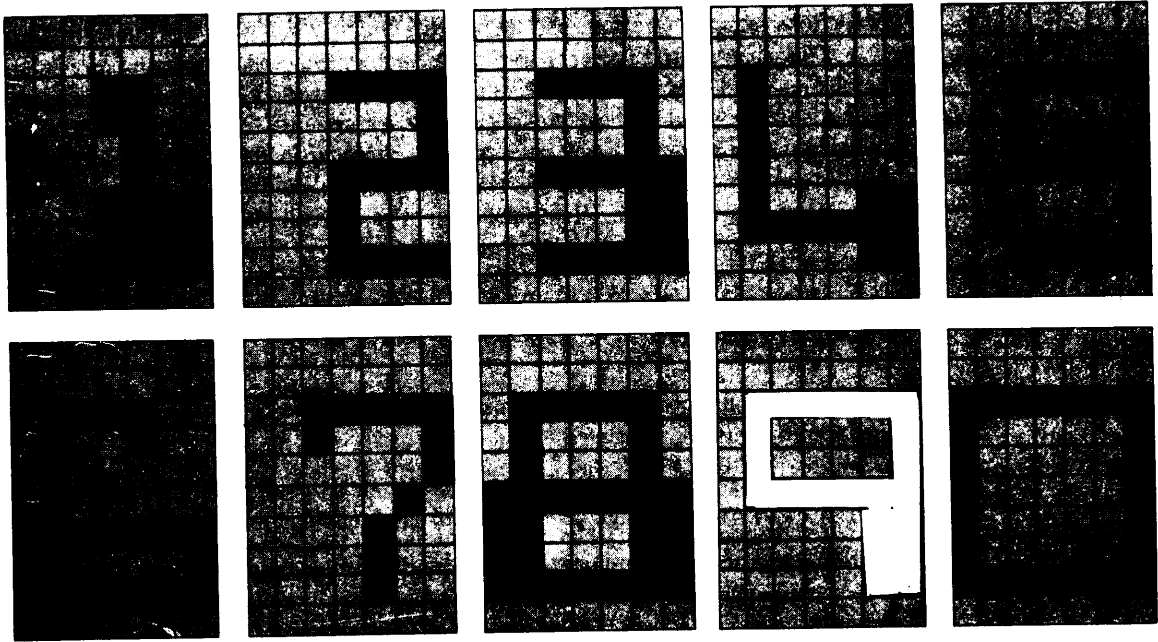
8.31 When the encoding scheme in the preceding questions is used, if the switches bounce, that is, if a closure of the switch is not constant but goes on and off when

a key is depressed, this complicates the encoding. Explain some problems that might arise from contact bounce if the above technique is used.

8.32 How would you suggest smoothing the bounce from the contacts for the encoding scheme from Question 8.31

8.33 Draw an encoder matrix for three ASCII characters (not shown in the chapter) as in the section on keyboards.

8.34 The signal that strobes the values into the flip-flops which read from the encoder of Fig. 7.18 must be slightly delayed. Explain why.



THE CONTROL UNIT

This chapter describes the control sections of digital computers. Preceding chapters illustrated techniques whereby arithmetic and logical operations may be performed and information read into and from various memory devices. To utilize the speeds and information handling capabilities of these techniques and devices, it is necessary to sequence automatically the various operations which occur at speeds compatible with those of the rest of the machine. The control element, therefore, must be constructed of high-speed circuitry. The basic elements used in the control element of a digital computer are described in Chaps. 3 and 4, and most of the concepts underlying the functioning of the control element are presented in Chaps. 3 through 5.

The *control unit* may be defined as "the parts that effect the retrieval of instructions in proper sequence, the interpretation of each instruction, and the application of the proper signals to the arithmetic unit and other parts in accordance with this interpretation."[1]

The function of the control circuitry in a general-purpose computer is to interpret the instruction words and then sequence the necessary signals to those sections of the computer that will cause it to perform the instructions. Previous chapters have shown how the application of the correct sequence of control signals to the logic circuitry in the arithmetic element enables the computer to perform arithmetic operations, and how binary words may be stored and later read from

---

[1]From *IEEE Standard Dictionary of Electrical and Electronics Terms*, IEEE Standard 100–1977, Institute of Electrical and Electronics Engineers, Inc.

THE CONTROL UNIT

several types of memory devices. For the computer to function, the operation of its sections must be directed, and the control circuitry performs that function.

This chapter first presents some introductory material concerning computer instruction-word execution. Two general-purpose computers are used as examples. Then a small general-purpose computer's control circuitry is described. The basic ideas in the design of control circuitry are presented in these sections. Register transfer concepts are emphasized. The final sections describe microprogrammed computer control concepts, giving the basic ideas used in this class of computers.

OBJECTIVES

1 Instruction word formats and instruction repertoires for two general-purpose binary computers are presented.

2 The design of the control section for a small computer is shown with a description of overall design procedures for large and small computers.

3 The use of register transfer language in designing and maintaining computers is presented. The control section designs in the chapter are based on this language and associated procedures. These are the most widely used techniques for digital computer system descriptions.

4 The control structure of a computer can be implemented by using a ROM, and then the computer is said to be microprogrammed. The subject is explained along with how register operations are sequenced when microprogramming is used.

CONSTRUCTION OF INSTRUCTION WORD

9.1 A computer word is an ordered set of characters handled as a group. Basically all words consist of a set of binary digits, and the meaning of the digits depends on several factors. For instance, the bits 01000100 could represent the decimal number 68 in a pure binary computer and the decimal number 44 in a BCD computer which uses an 8, 4, 2, 1 code. Thus the meaning of a set of digits is sometimes determined by its usage. In addition, other interpretations are possible, for instruction words are stored just as are data words, and the digits could represent an instruction to the computer. Since memory locations can store either instruction words or data words, the programmers and system operators must see that the instruction words are used to determine the sequence of operations which the computer performs, and that reasonable meanings are assigned to the data words.

If we assume that each memory location can contain a single instruction word, then a computer will start with the word stored in some specified address, interpret the contents of this location as an instruction, and then continue taking instruction words from the memory locations in order, unless a HALT or BRANCH instruction is encountered. The data to be used in the calculations will be stored in another part of the memory. Since the computer can store either instructions or data in the same memory addresses, considerable flexibility of operation results.

CONSTRUCTION OF INSTRUCTION WORD

An instruction word in a digital machine generally consists of several sections. The number of divisions in the word depends on the type of computer. Because of its wide usage and simplicity, we describe what are called *single-address instruction words* in this and the following sections, leaving more complicated formats for later. The single-address instruction word is widely used in microcomputers and minicomputers, as well as in many of the larger computers; it serves as a good basis for introducing control unit operations. Basically each single-address instruction word contains two sections: the *operation code* (OP code), which defines the instruction to be performed, such as addition, subtraction, etc.; and the *address part,* which contains the location of the number to be added or subtracted or otherwise used (the operand).

As an example, we now examine the Harris 6100 microprocessor. This microprocessor IC chip is used in the DEC word processors, one of their personal computers, and several other items including disk drive controllers, printer controllers, etc. The instruction word format, instruction repertoire, and general architecture originated in the DEC PDP-8 series, which was DEC's first "big winner" in the minicomputer area and the largest-selling minicomputer for some years. The Harris 6100 is widely used in DEC products, among others, and is generally available.

The 6100 has a basic memory word and instruction word of 12 bits.[2] The instruction word comprises two sections, an OP-code part and an address part, as shown in Fig. 9.1(*a*). There are only 3 bits in the OP-code part, and so only eight basic instruction types are possible. In this section we describe only three of these, leaving the remainder for Chap. 10. The instructions we study are the TAD (2s complement add), the DCA (deposit and clear), and the JMP (jump) instructions.

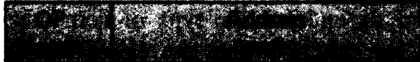
The TAD instruction [Fig. 9.1(*b*)] has an OP code of 001 (in binary). It tells the computer to add the number located in memory at the address given in the address part of the instruction to the number currently in the accumulator and to place the sum in the accumulator. Thus if the address part of the instruction were 000100110, this would reference the number at address 38 (decimal) in memory. The computer instruction word that will cause the 12-bit number at address 38 (decimal) memory to be added to the number in the accumulator will be 001000100110. Words are generally written in octal in the 6100, and this word would be 1046 in octal.

The DCA instruction has OP code 011 in binary. This instruction tells the CPU to deposit or store the present contents of the accumulator at the address given by the address part of the instruction. Thus the instruction word 011000001101 tells the CPU to store the current contents of the accumulator at location 13 in the memory. The DCA instruction also clears the accumulator to all 0s.

Let us now examine two program steps, a DCA followed by a TAD. Let these two instruction words be at memory locations 41 and 42 (octal). Let the DCA refer to location 50 (octal) and the TAD to location 51. The arrangement is as follows:

[2]This is a good size for a word processor because a character plus underscore, overbar, and other options in word processors can be stored in the 12 bits at each location.

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

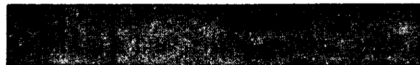


(*a*)

OP code for TAD instruction is 001

001000000111 Example: This instruction word tells computer to add word at location 7 in memory into the accumulator

(*b*)



OP code for DCA instruction is 011

011000001101 Example: This instruction word tells computer to deposit the contents of the accumulator at the address in memory given in the address section which is 13_{10}

(*c*)

FIGURE 9.1

6100 instruction words. (*a*) Instruction word format. (*b*) TAD instruction format. (*c*) DCA instruction format.

LOCATION IN MEMORY (OCTAL)	MEMORY CONTENTS (OCTAL)	MEMORY CONTENTS (BINARY)
--	----	------------
41	3050	011000101000
42	1051	001000101001
--	----	------------
50	0222	000010010010
51	0243	000010100011

We now analyze the action of the computer as it executes these two instructions. Suppose that the accumulator contains 0102 (octal) when the instruction at 41 is executed. Then the value 0102 will be deposited (stored) at location 50, overwriting or destroying the value 0222 which was in location 50. The accumulator will be cleared to all 0s.

Next, the instruction at location 42 in memory will be executed. This instruction will add the value at location 51, which is 0243 (octal), to the current value in the accumulator.

Therefore, when execution is begun on the instruction word at location 43 (not shown), the accumulator will contain 0243, and the contents of memory location 50 will be 0102.

TABLE 9.1 SECTION OF 6100 PROGRAM

ADDRESS IN MEMORY (OCTAL)	CONTENTS (OCTAL)	ASSEMBLY LANGUAGE			
		LABEL	OP CODE	ADDRESS	COMMENTS
0041	3051		DCA	LOC1	/CLEARS ACC
0042	1052		TAD	LOC2	/LOADS 0200
0043	1053		TAD	LOC3	/ADDS 212
0044	3054		DCA	LOC4	/STORES AT 54
0045	5071		JMP	71	/GO TO 71
...	...		...		
0051	0600	LOC1	0600		
0052	0200	LOC2	0200		
0053	0212	LOC3	0212		
0054	0310	LOC4	0310		

Another instruction in the 6100's repertoire is the JMP instruction with OP code 101. This instruction causes a jump in memory to the address (location) given in the address part of the instruction word. For example, suppose the value at location 71 (octal) in memory is 101001000011 (binary) or 5103 (octal). When the CPU reads this as the instruction word JMP 0103, it will cause the next instruction to be taken from location 103 in memory and not from location 72.

Table 9.1 shows the three instructions so far introduced, combined into a five-instruction-word section of program. Assembly language and octal values are both shown in this table.

The operation of these instructions by a CPU would be as follows. When location 41 is read, the DCA instruction stores the current contents of the accumulator, which is then cleared to 0s. The next instruction word is TAD LOC2, which causes the number 0200 at location 52 to be added to the accumulator, giving 0200 in the accumulator.

When the TAD LOC3 instruction is read, it causes the number 0212 at location 53 in memory to be added to the number 0200 in the accumulator, giving 0412 in the accumulator. The CPU then executes the instruction DCA LOC4, causing the value in the accumulator, which is 0412, to be stored at address 54 in the memory. The CPU then reads the JMP 71 instruction, causing it to fetch the next instruction word from location 71 in the memory (and not from location 46).

After this section of the code has been executed, the sum of the numbers at locations 52 and 53 is stored in location 54, and the CPU jumps to location 71 in memory.

The 6100 has several addressing features which are discussed in the next chapter. Also, because of the short OP code (3 bits), several of the other instructions are very clever (and somewhat tricky). More details of this also are given in Chap. 10.

INSTRUCTION CYCLE AND EXECUTION CYCLE ORGANIZATION OF CONTROL REGISTERS

9.2 A digital computer proceeds through the execution of a program with a basic rhythm or pattern in its sequence of operation which is produced by the necessity of drawing both instructions and operands from the same memory.

THE CONTROL UNIT

The basic sequence of operations for most instructions in a digital computer of the single-address type consists of an alternation of a time period called the *instruction cycle* and a period called the *execution cycle*. During the instruction cycle, an instruction word is obtained from the memory and interpreted, and the memory is given the address of the operand to be used. During the execution cycle, the memory obtains the operand to be used (for instance, the multiplier if the instruction is a multiplication or the augend if the instruction is an addition), and then the operation called for by the instruction word is performed upon this operand.

Most computers now being made use an IC memory for the storing of both instruction words and operands or data. The cycle time for the memory is fixed. Once we tell the memory that we wish to read from it or write into it, a certain time will elapse before we can instruct the memory that we are again ready to read or write. If we are reading from the memory, the selected word will be delivered a short time after the memory has been given the address of the word to be read and has been instructed to read.

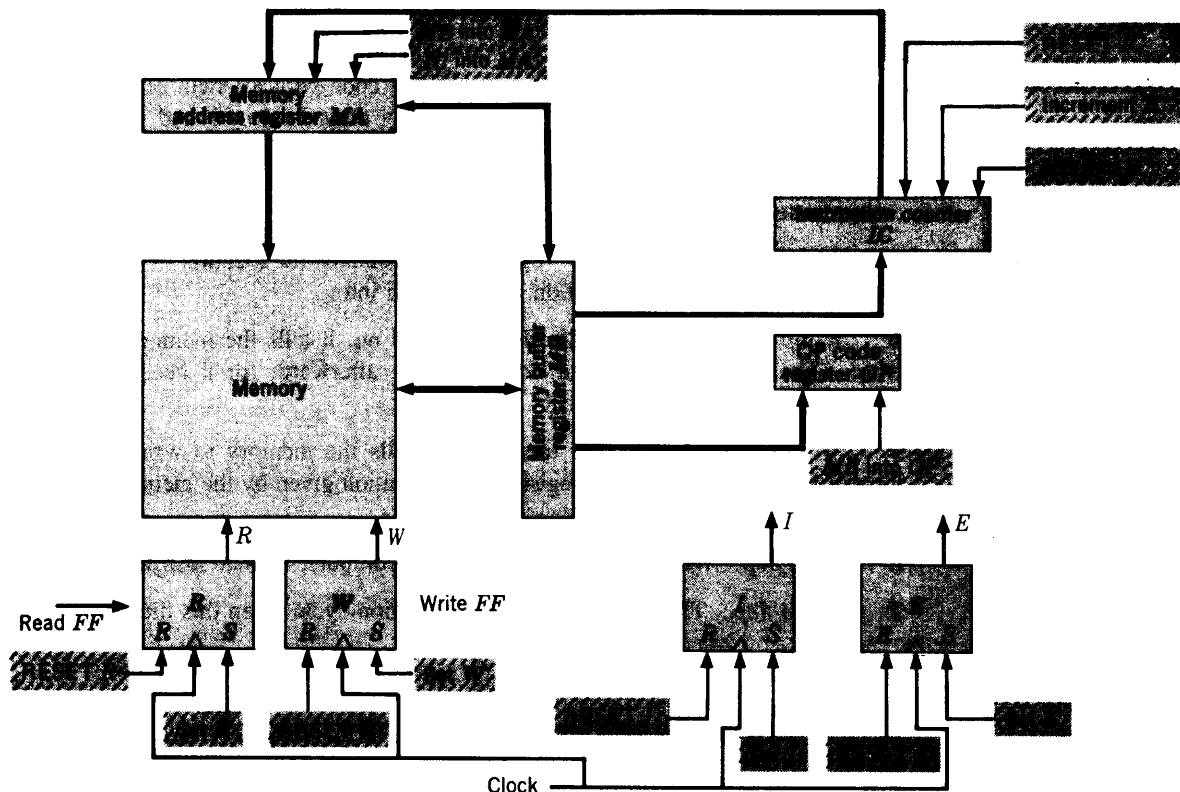
If the memory is to be written into, the word to be written as well as the address at which we wish to write it must be given to the memory. A WRITE signal also must be given to write this word at the location or address which we have given. As discussed in Chap. 6, the address which we write into or read from in the memory will be put into a *memory address register* and the word to be written into the memory put into the *memory buffer register*. When we read from the memory, the word read from the memory is delivered to the memory buffer register.

During each instruction cycle, the instruction word is transferred by the memory into the memory buffer register. To obtain this word, we must tell the memory to read and give the memory the address to read from. During the instruction cycle, the instruction word which was read into the memory buffer register is interpreted, and the address of the operand to be used is delivered to the memory address register. For many instructions this will be the address part of the instruction word which was read from the memory during the instruction cycle. During the execution time or execution cycle, an operand is obtained from or written into the memory, depending on the instruction word which was interpreted during the previous instruction time period.

For example, if the instruction being interpreted is an ADD instruction, the location of the augend is given in the address part of the instruction word, and this address must be given the memory address register. The memory then obtains the desired word and puts it into the memory buffer register. The computer must add this word to the word already in the accumulator. Afterward the computer must give to the memory the address of the next instruction word to be used and command the memory to read this word.

Note that the machine alternates between instruction cycles and execution cycles. Also note that during an execution cycle we must store somewhere in our control circuitry the OP code of the instruction word which was read from the memory, the address of the operand to be used (which was a part of the instruction word read from the memory), and the address of the next instruction word to be read from the memory and used.

As a result, there are several registers which are basic to almost every digital computer. These are shown in Fig. 9.2 and are described as follows:



Note: Control signals are shaded thus

FIGURE 9.2

Control registers.

1 *Instruction counter*[3] This counter is the same length as the address section of the instruction word. The counter can be either reset or incremented. A typical logic diagram for the instruction counter could consist of the counter shown in Fig. 4.14(*a*), having a RESET line and an INCREMENT or ENABLE line. This counter keeps track of the instructions to be used in the program, so that normally, during each instruction time, the counter is incremented by 1, which will give the location of the next instruction word to be used in the program. If, however, the instruction is a BRANCH or JUMP instruction, we may wish to place part of the *B* register's contents into this counter, and the MB INTO IC line does this. The counter can be reset to 0 when a program is started.[4]

It must also be possible to transfer the contents of this counter into the memory address register, which is used to locate a word in memory. Normally the instruction counter is increased by 1 during the performance of each instruction, and the contents of the counter are transferred into the memory address register at the beginning of each instruction time.

---

[3]In some computers the instruction counter is called the *program counter*.

[4]Most computers make it possible to load a selected address into the operation counter and thereby start the machine at that selected address.

2 *OP-code register* When an instruction word is read from the memory, the OP-code section of this word must be stored in order to determine what instruction is to be performed. If the computer has an OP code with a length of five binary digits, the operation register will be five binary digits in length and will contain the OP-code part of the instruction word which is read from the memory. Therefore we must be able to transfer a section of the memory buffer register into the OP-code register during the instruction time period.

3 *Memory address register* This register contains the location of the word in memory to be read or the location to be written into.

4 *R flip-flop* When this flip-flop is turned on, it tells the memory to read a word. (The flip-flop can be turned off shortly afterward, for it need not be on during the entire memory cycle.)

5 *W flip-flop* Turning on this flip-flop tells the memory to write the word located in the memory buffer register at the location given by the memory address register.

6 *I flip-flop* When this flip-flop is on, the computer is in an instruction cycle.

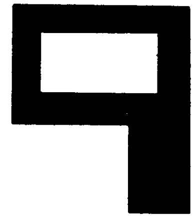
7 *E flip-flop* The computer is in an execution cycle when this flip-flop is on.

SEQUENCE OF OPERATION OF CONTROL REGISTERS

9.3 Let us consider further the construction of the control circuitry of a digital computer, again using the block diagram of the control registers, memory, memory address register, and memory buffer register shown in Fig. 9.2.

The control signals necessary to the operation of this small single-address computer are shown on the diagram and are as follows. There is a RESET IC line which clears the instruction counter to 0. (This is often connected to a pushbutton which clears the counter when the program is to be started.) There is an MB INTO IC control signal which causes the contents of the memory buffer register to be transferred into the instruction counter, and an INCREMENT IC control signal causes the instruction counter to be incremented by 1. Another control signal is MB INTO OP, which transfers the first five digits of the memory buffer register that contains the OP code of an instruction word into the five flip-flops in the operation register. The memory address register has two control signals. The IC INTO MA control signal causes the contents of the instruction counter to be transferred into the memory address register, and the MB INTO MA control signal causes the last 16 digits of the memory buffer register (which constitute the address part of an instruction word) to be transferred into the memory address register.

During each instruction cycle of the computer, we must turn on the READ flip-flop and at the same time (or earlier) transfer the contents of the instruction counter into the memory address register. The memory will now read an instruction word into the memory buffer register, after which time we can enable the MB INTO OP line, transferring the OP-code section of the instruction word into the OP-code register. The next actions that the computer will take are now dependent upon the contents of the OP-code register.



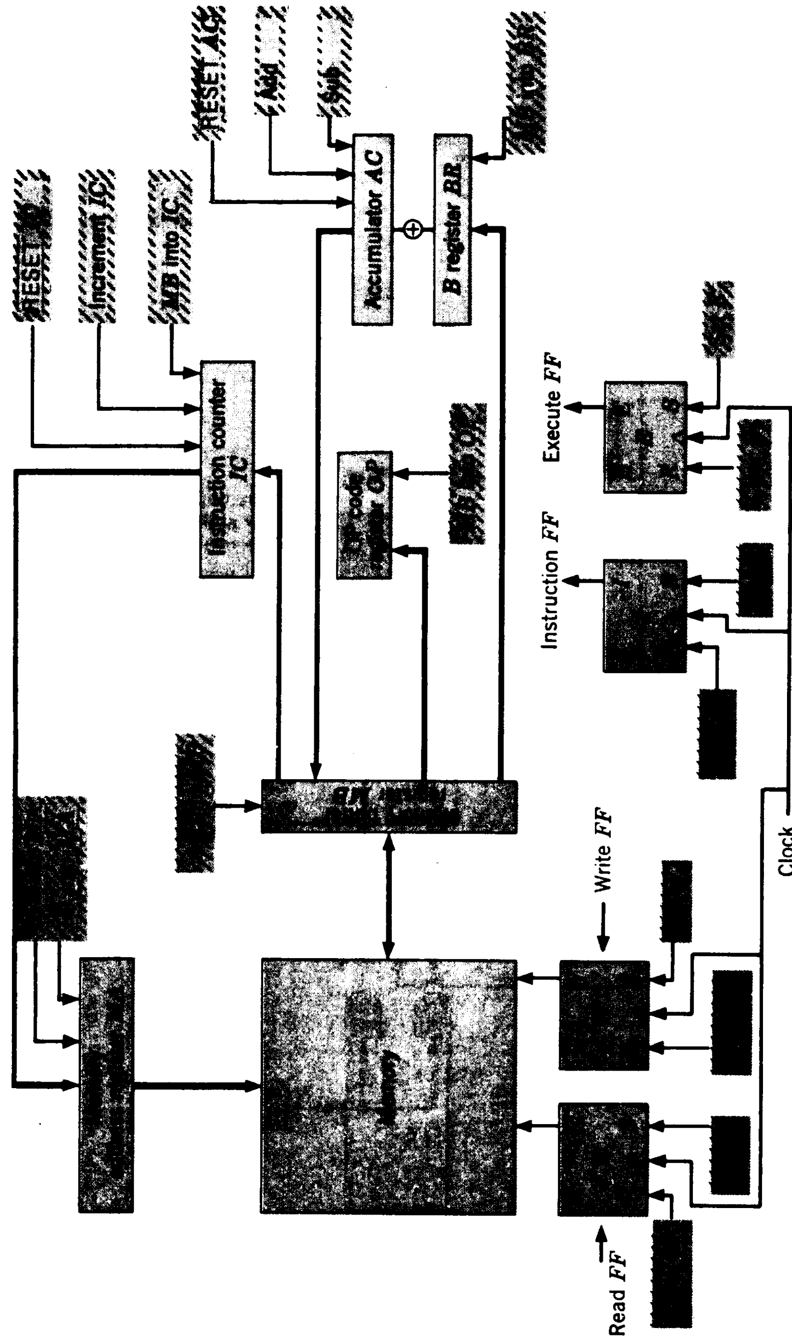


FIGURE 9.3

Control registers and arithmetic registers.

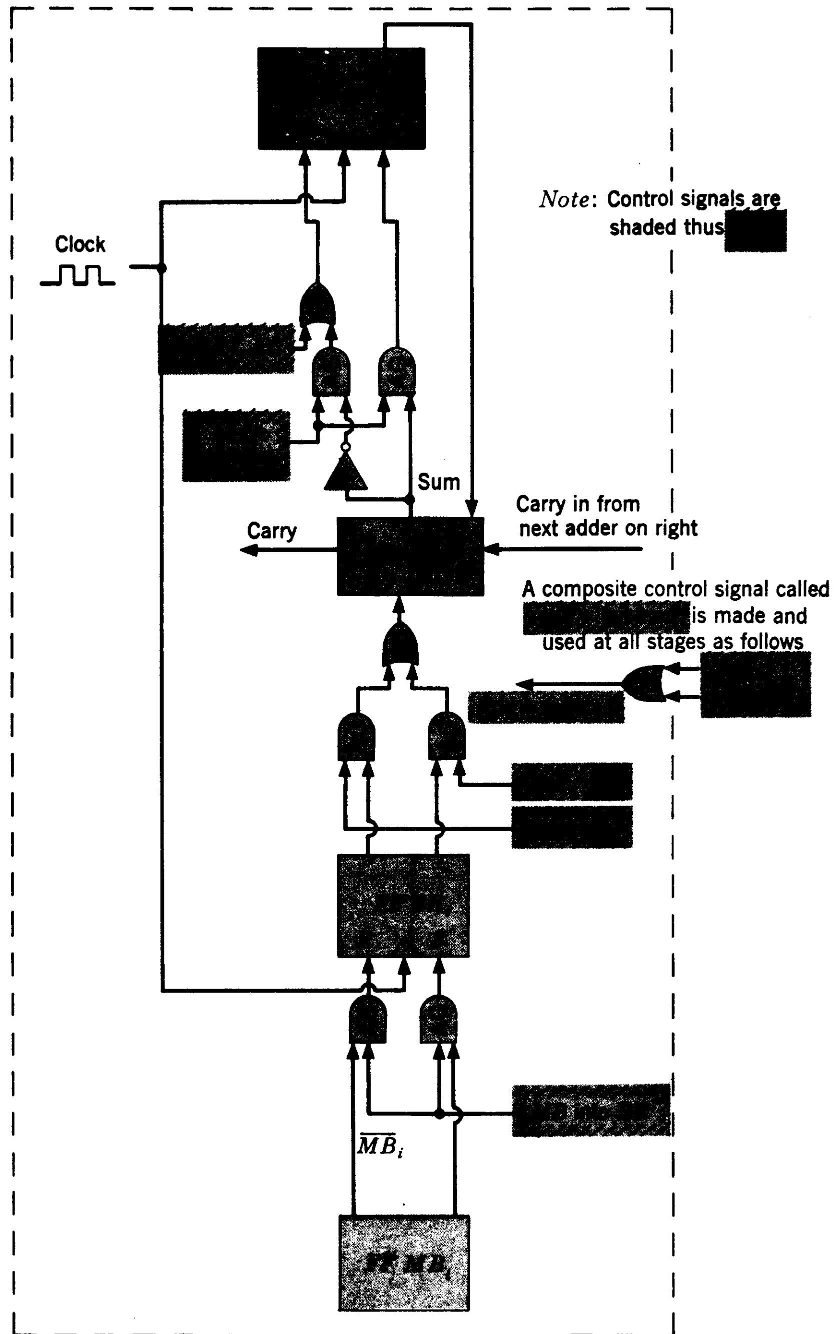


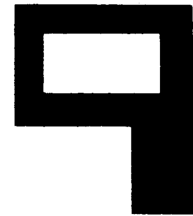
FIGURE 9.4

Accumulator flip-flop and B register flip-flop with control signals.

CONTROLLING ARITHMETIC OPERATIONS

9.4 Consider the problem of directing the arithmetic element as it performs an instruction word. Let us add an accumulator and a B register to the registers shown in Fig. 9.2, thus forming the block diagram shown in Fig. 9.3. Five more control

signals are required to perform such instructions as ADD, SUBTRACT, CLEAR AND ADD, and STORE:



1 *RESET ACC* This signal sets all the flip-flops in the accumulator to 0.

2 *ADD* This signal causes the B register to be added to the accumulator and the sum transferred into the accumulator.

3 *SUBTRACT* This signal causes the B register to be subtracted from the accumulator and the difference to be placed in the accumulator.

4 *MB INTO BR* This signal transfers the memory buffer register into the B register.

5 *AC INTO MB* This causes the contents of the accumulator to be transferred into the memory buffer register.

Figure 9.4 shows a single accumulator flip-flop and a single B register flip-flop, along with the control signals and gates required for these operations. The accumulator and B register are basically composed of as many of these blocks as there are bits in the basic computer word. (The carry into the least significant bit is connected to the SUBTRACT signal when 2s complement addition is used and to the carry-out of the sign digit when the 1s complement system is used.)

One further thing is needed. We must distribute our control signals in an orderly manner. Some sort of a time base, which will indicate where we are in the sequence of operations to be performed, is required. To do this, each memory cycle is broken into four equal time periods, the first of which we call T_0, the second T_1, the third T_2, and the fourth T_3. If we are in the first time period, we need a signal which will tell us that it is now time T_0; during the second period, we need a signal which will tell us that it is time T_1; etc.

Figure 9.5 shows a way of generating such timing signals. There is a clock signal input, and the clock is assumed to be running so that during a memory cycle

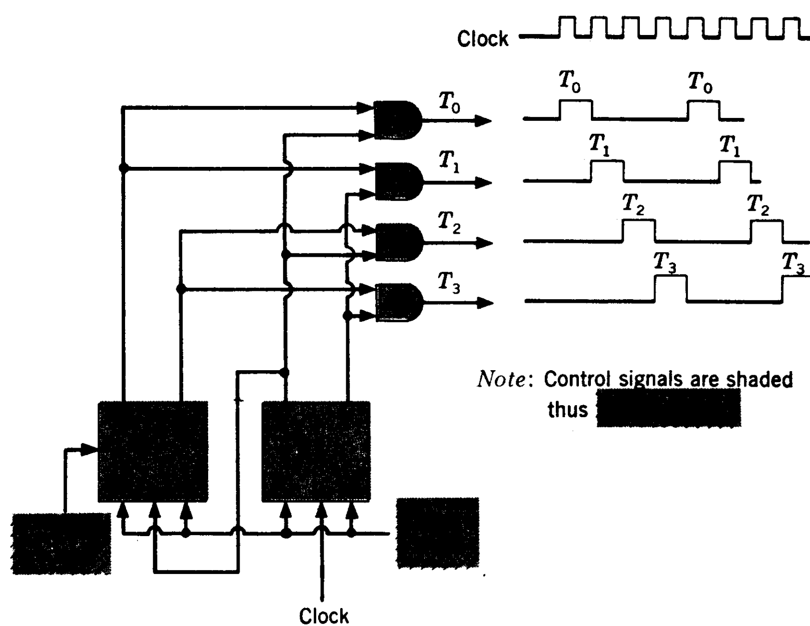


FIGURE 9.5

Timing-signal distributor.

TABLE 9.2 SEQUENCING OF CONTROL SIGNALS

INSTRUCTION	LIST OF CONTROL SIGNALS TO BE TURNED ON	COMMENTS
ADD		
I and T_0	SET R	Tells memory to read instruction word.
I and T_1	MB INTO OP, RESET R	Transfers OP-code part of instruction word into OP-code register; turns off READ flip-flop.
I and T_2	INCREMENT IC	Adds 1 to the instruction counter, preparing for the next instruction.
I and T_3	MB INTO MA, RESET I, SET E	Transfers the address part of the instruction word (which is in the memory buffer register) into the memory address register. Puts the computer in the execution cycle.
E and T_0	SET R	Turns on the READ flip-flop telling the memory to read a word.
E and T_1	MB INTO BR, RESET R	Transfers the contents of the memory buffer register into the B register. Since the memory buffer register now contains what was read from the memory, the addend is transferred into the B register; also turns off READ flip-flop.
E and T_2	ADD	The contents of the B register are added to the accumulator, and the sum is placed in the accumulator.
E and T_3	IC INTO MA, SET I, RESET E	The contents of the instruction counter are transferred into the memory address register, giving the location of the next instruction word to the memory. The instruction cycle flip-flop is turned on, and the execution cycle flip-flop turned off.
CLEAR AND ADD		
I and T_0	SET R	Tells memory to read instruction word.
I and T_1	MB INTO OP, RESET R	Transfers OP-code part of instruction word into OP-code register; turns READ flip-flop off.
I and T_2	INCREMENT IC	Adds 1 to the instruction counter, preparing for the next instruction.
I and T_3	MB INTO MA, RESET I, SET E	Transfers the address part of the instruction word (which is in the memory buffer register) into the memory address register.
E and T_0	SET R	Turns on the READ flip-flop, telling the memory to read a word.
E and T_1	MB INTO BR, RESET AC, RESET R	Transfers the memory buffer register into the B register and also clears the accumulator, so if the B register is now added to the accumulator, the accumulator will contain the word read from memory.
E and T_2	ADD	The contents of the accumulator are added to the B register, and the sum is placed in the accumulator.
E and T_3	IC INTO MA, SET I, RESET E	The contents of the instruction counter are transferred into the memory address register, giving the location of the next instruction word to the memory. The instruction cycle flip-flop is turned on, and the execution cycle flip-flop is turned off.

TABLE 9.2 SEQUENCING OF CONTROL SIGNALS (*continued*)

INSTRUCTION	LIST OF CONTROL SIGNALS TO BE TURNED ON	COMMENTS
SUBTRACT		
I and T_0	SET R	Tells memory to read instruction word.
I and T_1	MB INTO OP, RESET R	Transfers OP-code part of instruction word into OP-code register; turns off READ flip-flop.
I and T_2	INCREMENT IC	Adds 1 to the instruction counter, preparing for the next instruction.
I and T_3	MB INTO MA, RESET I, SET E	Transfers the address part of the instruction word (which is in the memory buffer register) into the memory address register. Puts the computer in the execution cycle.
E and T_0	SET R	Turns on the READ flip-flop, telling the memory to read a word.
E and T_1	MB INTO BR, RESET R	Transfers the contents of the memory buffer register into the *B* register. Since the memory buffer register now contains what was read from the memory, the subtrahend is transferred into the *B* register; also turns off READ flip-flop.
E and T_2	SUB	The contents of the *B* register are subtracted from the accumulator, and the difference is placed in the accumulator.
E and T_3	IC INTO MA, SET I, RESET E	The contents of the instruction counter are transferred into the memory address register, giving the location of the next instruction word to the memory. The instruction cycle flip-flop is turned on, and the execution cycle flip-flop is turned off.
STORE		
I and T_0	SET R	Tells memory to read instruction word.
I and T_1	MB INTO OP, RESET R	Transfers OP-code part of instruction word into OP-code register; turns off READ flip-flop.
I and T_2	INCREMENT IC	Adds 1 to the instruction counter, preparing for the next instruction.
I and T_3	MB INTO MA, RESET I, SET E	Transfers the address part of the instruction word (which is in the memory buffer register) into the memory address register.
E and T_0	SET W, AC INTO MB	Transfers word to be read into memory from accumulator into the memory buffer register.
E and T_1	RESET W	Turns off WRITE flip-flop.
E and T_2		Contents of memory buffer register are written into memory.
E and T_3	IC INTO MA, SET I, RESET E	The contents of the instruction counter are transferred into the memory address register, giving the location of the next instruction word to the memory. The instruction cycle flip-flop is turned on, and the execution cycle flip-flop is turned off.

we obtain four clock pulses. If it requires 1 μs to read into or write from the memory, a clock pulse should be generated every $\frac{1}{4}$ μs. Therefore the clock will run at a rate of 4 MHz.

The circuit has four output lines, designated T_0, T_1, T_2, and T_3. When the computer is in time period T_0, the output line T_0 will carry a 1 signal, and T_1, T_2, and T_3 will be 0s; at time T_1 only, line T_1 will have a 1 signal on it, etc.

Let us now write, in a short table, the sequence of operations which must occur during each of the ADD, SUBTRACT, CLEAR AND ADD, and STORE instructions. Notice that when the instruction cycle flip-flop is on, the operations during times T_0 and T_1 are always the same. In Table 9.2 the control signal to be turned on (or made a 1) is listed to the left, and what the signal does is listed to the right.

From this table of operations it is possible to design the control section of this small computer. The inputs are the OP code stored in the OP-code register, the timing-signal distributor, and the I and E flip-flops.

Notice, for instance, that when it is time T_0 and we are in an instruction cycle, we always turn on the READ flip-flop, telling the memory to read the instruction word located at the address in the memory address register. Then we assume that the memory places this word in the memory buffer register before time T_1, so at time T_1 we transfer the OP-code part of the instruction word into the OP-code register. These two facts tell us that we should logically AND the output line T_0 from the timing-signal distributor with the 1 output of the I flip-flop and then connect the $T_0 \cdot I$ signal to the set input of the READ flip-flop. Next we should connect a $T_1 \cdot I$ signal to the control line that transfers the first 5 bits of the memory buffer register into the OP register. This is shown in Fig. 9.6.

What happens next is always dependent on the OP-code register. We now connect a decoder with $2^5 = 32$ outputs to that register (assuming that we will use all the combinations by adding more instructions). We then have a set of signal lines, so that line 00000 = ADD will carry a 1 signal when we are adding (since the operation code for ADD is 00000); 00001 = SUB will carry a 1 if and only if we are subtracting, since the OP code for subtract is 00001; and 00010 = CLA will be a 1 only when we clear and add. We combine these lines and the timing-signal distributor lines and the I and E flip-flop lines to give us all the control signals needed to run the computer. Figure 9.6 shows the complete control circuitry required. A comparison of this figure with the timing and control signal chart in Table 9.2 will show how the control circuitry works and signals are manufactured when they are needed.

More instructions can be added by adding to the timing and control signal chart and by adding the required gates to the control circuitry. Analyzing the computer in this way, we can readily see how the control circuitry directs the operations performed in the machine, alternating the acquisition of instructions from the memory and the performance of the instructions.

TYPICAL SEQUENCE OF OPERATIONS

9.5 It is instructive to analyze the control circuitry in Fig. 9.6 during both an ADD instruction and a STORE instruction. Each instruction is started with the I (instruction cycle) flip-flop on and with the timing-signal distributor having an

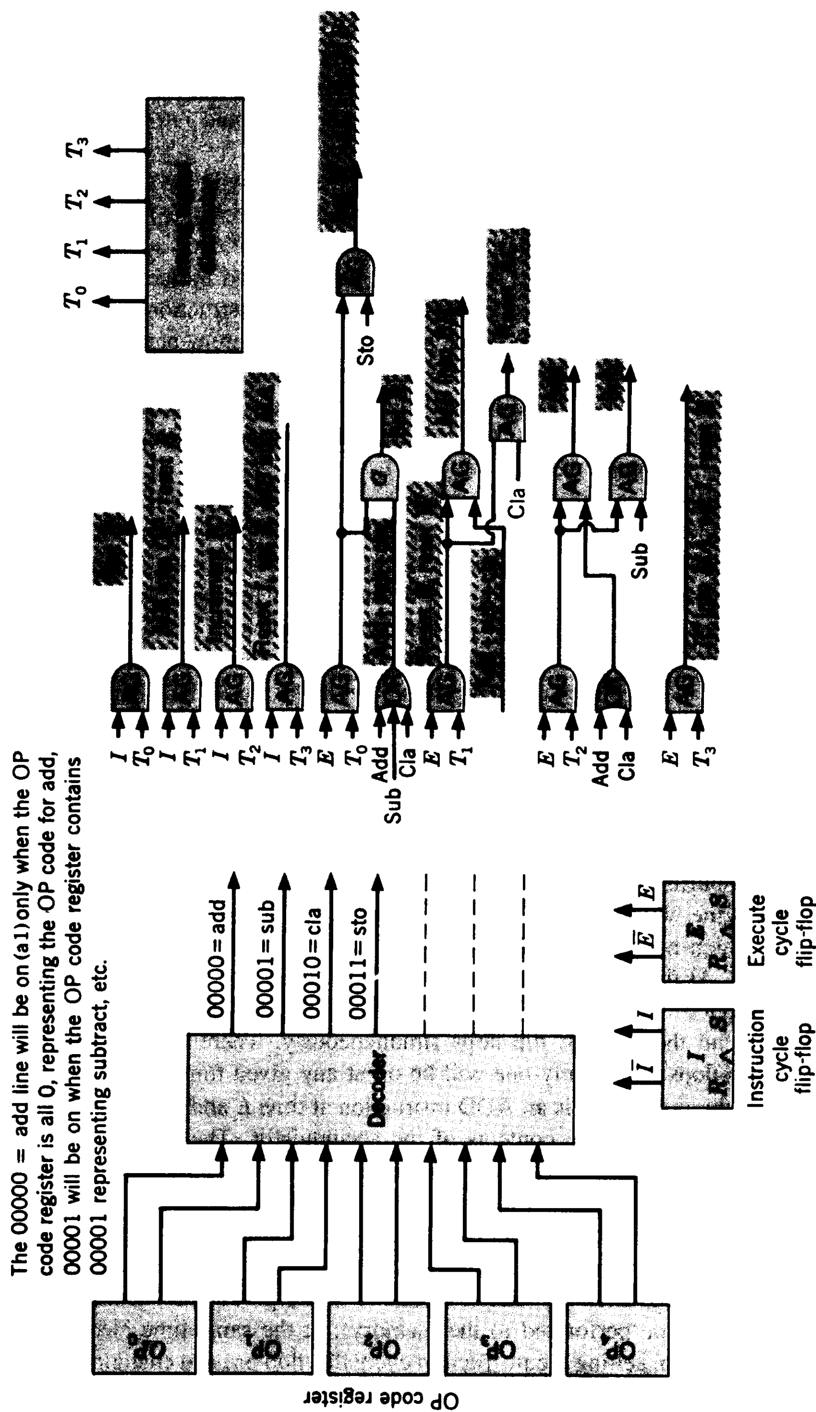


FIGURE 9.6

Control circuitry for four-instruction computer.

output on line T_0. The AND gate at the upper right of the figure will therefore be turned on by I and T_0, thus setting the READ flip-flop to the 1 state and initiating a READ from the memory. At this time, the memory address register is assumed to have the address of the instruction that will be read into the memory buffer register.

By time T_1 the word read from the memory will have been read into the memory buffer register, so that when we have the control state I and T_1, the contents of the memory buffer register which constitute the OP-code section of the instruction will be transferred into the OP-code register, and the computer will be in a position to decode the OP code and determine what instruction is to be performed.

At time I and T_2 the instruction counter is incremented by 1, so that the instruction counter now contains the address of the next instruction to be read from the memory. The AND gate connected to the I and T_2 input signals is used in turn on the INCREMENT IC control signal, and its output is designated by the name of the control signal.

Similarly, at time I and T_3 the memory buffer register is transferred into the memory address register by the MB INTO MA signal, thus transferring the address part of the instruction word into the memory address register. The next word read from or written into the memory will then be at the address designated by the address part of the instruction word which was just read from the memory.

At the same time, the instruction cycle flip-flop is cleared by the RESET I signal, and the execution cycle flip-flop is set on by the SET E signal, thus changing the state of the computer from an instruction cycle to an execution cycle.

At time E and T_0, then, during an ADD instruction, we set the R flip-flop on, thus telling the memory to read the word at the address currently in the memory address register. In this case, this address is the address part of the instruction word that is being executed. Then at time E and T_1 we transfer the contents of the memory buffer register into the B register. The memory buffer register at that time contains the word which has been read from the memory, so that we now have the word which has been addressed by the instruction word in the B register for the addition. At the same time we reset the READ flip-flop.

Notice that the RESET R and RESET W lines are used to reset both the READ and the WRITE flip-flops simultaneously. There is no harm in resetting both flip-flops, since only one will be on at any given time.

If the instruction is an ADD instruction at time E and T_2, we add the contents of the B register to the contents of the accumulator. The B register contains the word which has been read from the memory, and the accumulator has not been changed; so their sum will be transferred into the accumulator. Thus the sum of the word read from the memory and the previous contents of the accumulator will be placed in the accumulator. Then, at time E and T_3, we transfer the instruction counter into the memory address register (thus giving the address of the next instruction to be performed to the memory), at the same time clearing the EXECUTE flip-flop, setting the instruction cycle flip-flop on, and changing the computer from an execution cycle to an instruction cycle.

Since the I flip-flop is on and it is time T_0, the SET R control line will go high, thus telling the memory to read a word. The next instruction word will be read from the memory and can then be interpreted.

Let us now examine the operation of the STORE instruction. When the instruction flip-flop is on and we are in an instruction cycle, the R flip-flop will be

set on when time T_0 arrives, telling the memory to read just as for an addition, subtraction, or clear and add. Since at time I and T_1 the memory buffer register flip-flops contain the OP code of the instruction these will be transferred into the OP-code register.

At time I and T_2 we increment the instruction counter so that the address of the next instruction in memory now lies in the instruction counter; and at time I and T_3 we reset the instruction flip-flop and turn on the execution cycle flip-flop, thus putting the computer in an execution cycle.

At time E and T_1, if the instruction is a STORE instruction, we set the WRITE flip-flop on, thus initiating a WRITE into the memory. We also transfer the contents of the accumulator into the memory buffer register, so that the word written into the memory will be the current contents of the accumulator register, and so that after the WRITE cycle has been terminated, the accumulator will have been written into the memory at the address that was given by the instruction word.

At time E and T_1 we reset the WRITE flip-flop, since we have already told the memory to write, nothing need be done at E and T_2, for we are now writing the word into the memory. At time E and T_3 the instruction counter is transferred into the memory address register by the IC INTO MA control signal, thus giving the address of the next instruction to the memory. The instruction cycle flip-flop is turned on and the execution cycle flip-flop turned off, thus turning the computer to the instruction cycle state. The machine will now execute an instruction cycle by reading the next instruction word from the memory, interpreting it, and continuing the program.

The preceding example demonstrates how it is possible to design a computer that will execute a given sequence of operations and thereby cause it to perform each instruction word that is read from the memory. Although only four instructions are demonstrated in this particular example, more instructions can be added in exactly the same manner by simply writing what must be done when an instruction word is read from the memory, listing the operations that must be performed, and providing gates which will generate the control signals necessary to the performance of each instruction. Subsequent sections discuss shifting instructions and branching instructions. All these may be incorporated into the computer shown by simply adding gates to the control circuitry and providing for the additional gates necessary for the transfers and operations between registers.

The general form of the control signal generating scheme is shown in Fig. 9.7. It shows a timing-pulse distributor with eight different time divisions, in which case each time period is one-eighth of the memory cycle time.

BRANCH, SKIP, OR JUMP INSTRUCTIONS

9.6 The BRANCH, SKIP, or JUMP instruction varies from the normal instruction in several ways.[5] For single-address machines only one word, the instruction word, must be located in memory. Also, the contents of the instruction counter may be modified instead of being simply increased by 1. There are two types of

[5]A survey indicates that some manufacturers call these instructions BRANCH instructions, others call them TRANSFER instructions, and still others call them SKIP or JUMP instructions. All are the same thing.

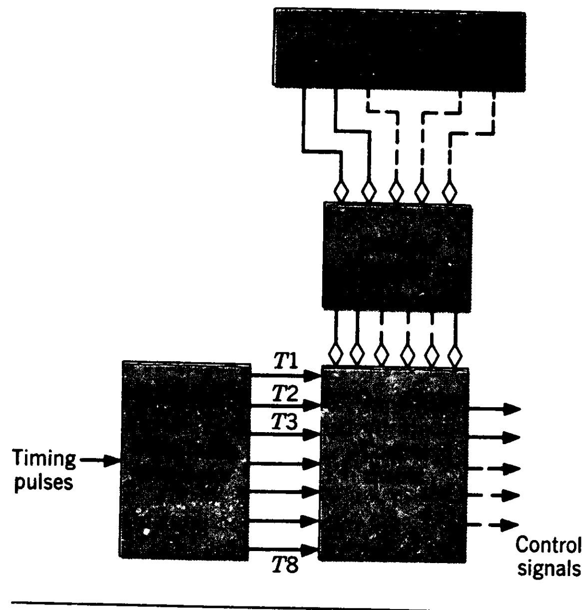


FIGURE 9.7

General configuration of control circuitry.

BRANCH instructions: conditional and unconditional. For an *unconditional* BRANCH instruction, the contents of the address portion of the memory buffer register are always transferred into the instruction counter. The next instruction performed will be the instruction at the location indicated by the address section of the instruction word. In a *conditional* branch instruction the branch may or may not occur depending on some condition. For example, in single-address computers the *conditional* BRANCH-ON-MINUS instruction will cause the machine to branch only if the number stored in the accumulator register of the arithmetic element is negative. If the number in the accumulator is positive, the contents of the instruction counter will simply be increased by 1, and the next instruction will be taken in the normal order.

As can be seen, during a conditional BRANCH-ON-MINUS instruction, the sign bit of the accumulator of the arithmetic element must be examined by control circuitry. If the sign bit is a 1, the number stored is negative, and so the number in the address part of the instruction word is transferred into the instruction counter. If the sign bit is a 0, a 1 is added to the instruction counter, and the computer proceeds.[6]

To demonstrate how a typical BRANCH-ON-MINUS (BRM) instruction operates in a single-address computer, we modify the control circuitry shown in Fig. 9.6 so that the small machine will include a BRM instruction. Let us give the OP code 00100 to BRM, so that the line beneath the 00011 = STO line will be high from the decoder attached to the OP-code register in Fig. 9.6 when a BRM instruction is in the register.

[6]Many computers have a set of *status bits* (flip-flops) which are set and reset depending on the results of operations performed. Jumps or transfers are then taken based on these flip-flops. The Questions and Chap. 10 cover this in detail.

TABLE 9.3

BRANCH ON MINUS	LIST OF CONTROL SIGNALS TO BE TURNED ON	COMMENTS
I and T_0	SET R	Tells memory to read instruction word.
I and T_1	MB INTO OP, RESET R	Transfers OP-code part of instruction word into OP-code register; turns off READ flip-flop.
I and T_2 and $\overline{AC_N}$	INCREMENT IC	If the sign digit of the accumulator AC_N is a 0, we want to increment the instruction counter and use its contents as the address of the next instruction.
I and T_2 and AC_N	MB INTO IC	If the sign digit of the accumulator is a 1, the accumulator is negative, and we want to use the address in the instruction word as the address of the next instruction word.
I and T_3	IC INTO MA	This transfers the instruction counter into the memory address register. Notice that the E flip-flop is not turned on as we are ready to read another instruction word; an EXECUTE cycle is not needed.

The first two time periods of the instruction cycle are the same for all instructions. First the memory is told to read, and then the instruction word is read from memory into the memory buffer register. The OP-code part is transferred into the OP-code register, so that, after time T_1 and the beginning of time T_2, the line 00100 = BRM will be high and all the other output lines from the decoder will be low. Now let us make a small table for a BRM instruction, showing what must be done to carry out this instruction. Table 9.3 shows the steps that must be taken.

If at time T_2 during the instruction cycle a BRM instruction OP code is in the OP-code register, one of two things must happen. Either we wish to increment the instruction counter and give this number as the address of the next instruction to be taken from the memory, or we wish to transfer the contents of the address portion of the instruction word into the instruction counter. Which choice we make depends on the sign bit of the accumulator, called AC_N. If the accumulator contains a negative number, it will have a 1 in AC_N; and if it contains a positive number, it will have a 0 in flip-flop AC_N. Therefore, if I AND T_2 AND $\overline{AC_N}$, we want to increment the instruction counter. If I AND T_2 AND AC_N happens to be the case, we wish to transfer the memory buffer register into the instruction counter. This is shown in the table. During time T_3 of this instruction cycle we want to transfer the instruction counter into the memory address register. We do not need to put the machine in an execution cycle, but can simply continue to another instruction cycle, taking the word at the address which has been transferred into the memory address register as the next instruction. Therefore, we do not clear the instruction cycle flip-flop or put a 1 in the execution cycle flip-flop; we simply transfer the instruction counter into the memory address register.

The control circuitry which will implement these operations is shown in Fig. 9.8. This means that the two particular AND gates in Fig. 9.6, which are connected to the I, T_2 and I, T_3 inputs, will be replaced with the two circuits shown in Fig.

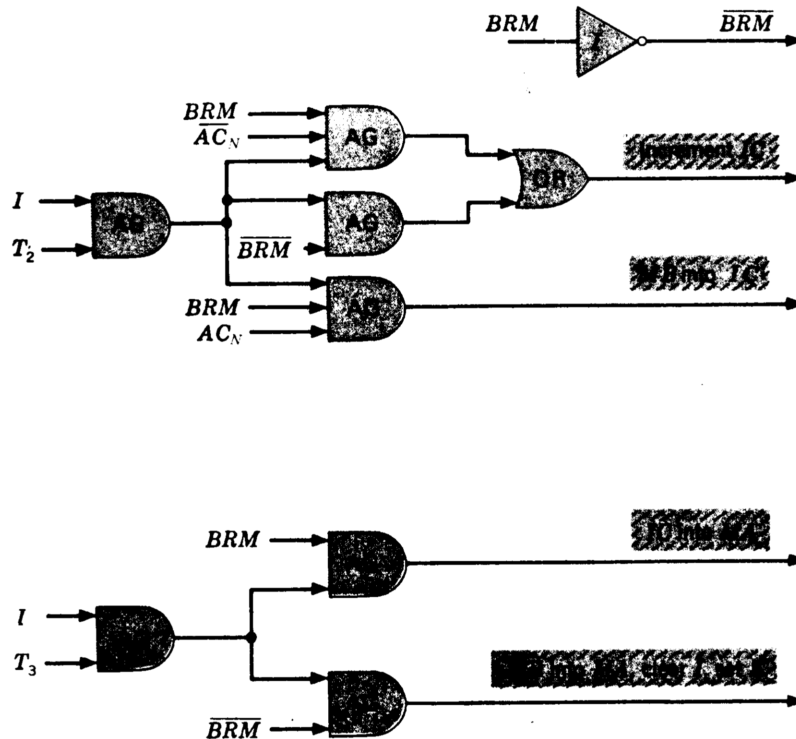


FIGURE 9.5

Modification of control circuitry for BRANCH instruction.

9.8. Notice that this logical circuitry, plus the circuitry in Fig. 9.6, is all that is needed to generate the control signals required for the BRM instruction.

Notice that the BRM instruction requires only one access to memory and thus only one instruction cycle for its execution.

SHIFT INSTRUCTIONS

9.7 The instructions we have examined to date were always performed within a basic fixed number of memory cycles. That is, the ADD, SUBTRACT, CLEAR AND ADD, and STORE instructions were performed within exactly two memory cycles, and the TRANSFER and BRANCH instructions required only one memory cycle. Several types of instructions may require more time than two memory cycles. Typical of these are multiplication and division, which generally require more time than two memory cycles. Similarly, an instruction such as SHIFT RIGHT or SHIFT LEFT could conceivably be performed in a single memory cycle, since the operand is in the accumulator when the instruction word is obtained. However, if the instruction calls for a large number of shifts, more than one memory cycle may be needed. In this case we could not initiate another memory cycle until we had finished shifting the requisite number of times. Similarly, for multiplication and division we could not initiate another memory cycle until we had finished our multiplication and division process.

To implement these types of instructions, we turn over our control of the computer to a simple control element which is dominated by a counter. This counter will sequence and count the number of steps that must be performed until the instruction has been completed, and then it will put the computer in an instruction cycle and tell the memory to read the next instruction word. We illustrate with a shift right instruction.

The SHIFT RIGHT instruction word consists of two parts: an OP code and an address part. The OP code of 00101 tells the machine to shift the word in the accumulator to the right the number of times given in the address part. So if we write 00101 for the OP code in an instruction word and then write 8 in binary form in the address part, the computer has been instructed to shift the binary number in the accumulator to the right 8 binary digits.

Assuming that we have an accumulator with gates so that we can shift the accumulator digits to the right, as explained in Chap. 5, all we need is to apply eight consecutive SHIFT RIGHT control signals to the accumulator and we will have shifted the number to the right eight places. Since there are only four pulses per memory cycle, we will not want to use the memory until we have completed our shifting. If, for instance, the instruction said SHIFT RIGHT 1, we could finish in one pulse time and start the next instruction cycle immediately after. But if the instruction word said SHIFT RIGHT 4, 5, or 15 or more times, we would have to wait until we had completed shifting before we could initiate another instruction cycle and fetch the next instruction word from the memory.

To do this, we first prepare the computer for the shifting operation by incrementing the instruction counter so that the next word obtained will contain the address of the next instruction word. To count the number of shifts that we perform, we add another register, called a *step-counter register*, which counts downward from a given number to 0. We then transfer the address part of the memory buffer register into the step counter, so the step counter contains the number of shifts to be performed. Then each time we shift, we decrement the counter by 1; so when the counter reaches 0, we will have performed the requisite number of shifts.

Figure 9.9 shows two stages of a decrementing counter and the gates necessary to transfer the memory buffer register contents into the step counter, designated *SC*. The two rightmost, or least significant, digits of the counter are shown (SC_1 and SC_0), as are the two rightmost digits of the memory buffer register (MB_1 and MB_0).

The actual number of stages in the step counter is determined by the maximum number of shifts which the machine must ever make and, since we will also use the same counter for multiplication and division, by the maximum number of steps that will ever be required to multiply or divide. For a computer containing 21 binary digits in the basic computer word, the counter might well contain five flip-flops. For a computer with a basic computer word of perhaps 35 or 36 binary digits, the step counter might well contain six or even seven flip-flops:

Consider a sequence of operations for a SHIFT instruction. Times I AND T_0 and I AND T_1 are as usual. At I AND T_2 we increment the instruction counter and transfer the count into the step counter. At I AND T_3, we set a flip-flop called *SR* (shift right) on, which tells the computer to start shifting. At the same time we clear the I flip-flop, so that the machine is in neither an instruction nor an execution cycle, although it is actually executing an instruction. Thus we do not initiate

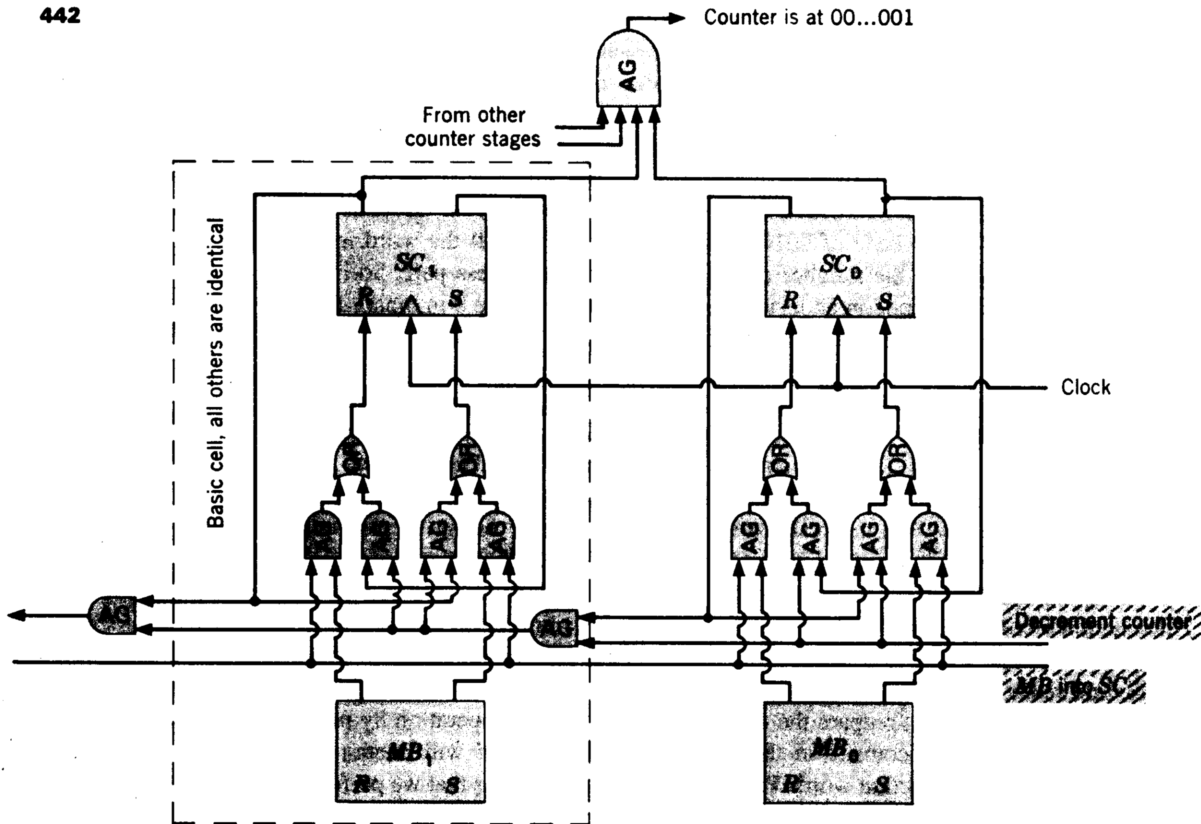


FIGURE 9.9

Two stages of decrementing counter and transfer network.

subsequent memory cycles, and the machine effectively freezes in the shifting state until the step counter has counted to 0, indicating that the requisite number of shifts has been performed. (Actually the step counter counts only to binary 1 rather than to 0 before the order to stop counting is given, for counting when the counter is at 0 would introduce an extra shift.) If we turn off the counter *SR* flip-flop when the output of the counter is at the 00 . . . 001 signal, and if at the same time we turn on the *I* or instruction cycle, the computer will proceed to the next instruction cycle, fetching the next instruction word from the memory and performing it. Table 9.4 shows this.

When the *SR* flip-flop is on, it will be necessary to stop the timing-signal distributor. Thus we arrange to disable this circuit, using the *SR* flip-flop's output for this purpose.

Implementation of the above procedure is straightforward. A three-input AND gate with inputs I, T_3, and 00101 = SHR (the output from the decoder in Fig. 9.6) can be used to turn on an *SR* flip-flop, the STOP output from the step counter to turn it off, also turning on I. The input to the clock can be turned off when *SR* is on.

TABLE 9.4

SHIFT RIGHT	CONTROL SIGNAL TURNED ON	COMMENTS
I and T_0	SET R	Tells memory to read.
I and T_1	MB INTO OP, RESET R	OP code of instruction word is transferred into OP-code register. READ flip-flop is turned off.
I and T_2	INCREMENT IC, MB INTO SC	The instruction counter is prepared to obtain the next instruction word. The address part of the instruction word is transferred into the step counter.
I and T_3	RESET I, SET SR	The instruction cycle flip-flop is turned off. The SHIFT RIGHT flip-flop is turned on.

REGISTER TRANSFER LANGUAGE

9.8 The preceding design showed how to generate a sequence of control signals which would cause instruction words read from a memory to be executed. The control signals were named so that the function of each was indicated. For example, the control signal INCREMENT IC causes the IC (instruction counter) to be incremented; RESET W causes W to be reset; MB INTO BR causes the contents of MB to be transferred into the BR register, etc.

COLLEGE OF P.G. STUDIES SRINIVAS MANGALORE 575 001.

To document a design, it is convenient to have a notational technique for representing these operations on and between registers. The most used way to organize and write register operations is called *register transfer language*, and was invented by I. S. Reed.[7] Currently, manufacturers' design efforts and their manuals documenting computer designs all use some version of register transfer language.

An example of a transfer between registers in register transfer language is

$$A \rightarrow B$$

This says, "Transfer the contents of register A into register B." A control signal to effect this might be conveniently called A INTO B.

Another example of register transfer language is

$$0 \rightarrow D$$

This says, "Set D to a 0." If D is a flip-flop, this simply means to reset D, and an appropriate control signal name might be RESET D.

---

[7]The book *Theory and Design of Digital Machines* by T. C. Bartee, I. L. Lebow, and I. S. Reed, McGraw Hill, New York, first presented this design technique in detail.

Here is another example:

$$A + B \rightarrow A$$

This says, "Add the numbers in A and B and place the sum in A." A control signal for this might well be called ADD or ADD AB.

The above operations on and between registers are sometimes called *micro-operations,* particularly if the computer is microprogrammed, as will be discussed.

An interesting statement in register transfer language is

$$A + 1 \rightarrow A$$

This says, "Add 1 to A and place the sum in A." A name for the corresponding control signal might be INCREMENT A.

In some cases, register transfers or operations affect only parts of registers. An example was shown in preceding sections where the first 5 bits in the memory buffer register MBR were transferred into the OP register. Subscripts are generally used to indicate specific bits, and an example transfer can be written

$$B_{0\text{–}4} \rightarrow P_{0\text{–}4}$$

This assumes that the B register flip-flops have been named B_0, B_1, . . . , B_N; and this means B_0, B_1, B_2, B_3, and B_4 will be transferred into P_0, P_1, P_2, P_3, and P_4, respectively.

A specific bit in a register also can be transferred. Consider

$$A_3 \rightarrow B_2$$

This transfers A_3 of register A into B_2 of register B.

Sometimes operations and transfers are dependent on certain conditions. This is indicated as follows:

$$R = 0 : A \rightarrow B$$

This statement means, "If R has value 0, transfer A into B." Here is another example:

$$R \cdot T_2 : IC + 1 \rightarrow IC$$

This statement says, "If R is a 1 and T_2 is a 1, then increment the IC register." The colon is used to indicate a conditional operation.

Here is the CLEAR AND ADD instruction in Table 9.2 rewritten in register transfer language. We will use the CLEAR AND ADD control signal CLA from the decoder in Fig. 9.6.

$$
\begin{aligned}
I \cdot T_0 &: 1 \rightarrow R \\
I \cdot T_1 &: MB_{19-23} \rightarrow OP_{0-4}, \\
& \quad 0 \rightarrow R \\
CLA \cdot I \cdot T_2 &: IC + 1 \rightarrow IC \\
CLA \cdot I \cdot T_3 &: MB_{0-18} \rightarrow MA_{0-18}, \\
& \quad 0 \rightarrow I,\ 1 \rightarrow E \\
CLA \cdot E \cdot T_0 &: 1 \rightarrow R \\
CLA \cdot E \cdot T_1 &: MB \rightarrow BR, \\
& \quad 0 \rightarrow AC,\ 0 \rightarrow R \\
CLA \cdot E \cdot T_2 &: A + B \rightarrow A \\
CLA \cdot E \cdot T_3 &: IC \rightarrow MA \\
& \quad 1 \rightarrow I,\ 0 \rightarrow E
\end{aligned}
$$

The above assumes 24 bits in the A and MB registers and 19 bits in the memory address register. So MB_{0-18} is the address part of an instruction word and gets transferred into MA_{0-18}.

Also notice that the entire circuitry for generating the gates for control signals can be read directly from the table. The final statement,[8] for example, says this: "If you AND CLA, E, and T_3, then the output from the AND gate can be used to initiate the transfer IC → MA and 1 → I and 0 → E." This means the output from the AND gate can be connected to (perhaps being ORed with other signals) the control signals IC INTO MA, SET I, and RESET E.

Many register transfer languages have been designed and used by different individuals and companies. One frequent variation is in making transfers move from right to left (which is more like programming practice). In this variation, we find

$$B \leftarrow A$$

instead of

$$B \rightarrow A$$

Sometimes equals signs are used. Then

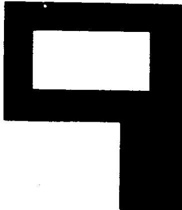
$$B = A$$

says to transfer A into B. Occasionally we see

$$B := A$$

This also says to transfer A into B in some variations.

[8]Notice the control signal CLA from the decoder in Fig. 9.6 can be used after time T_1 because the OP code is in OP after that time.



It is generally not hard to read one of the register transfer languages once a basic one has been understood.

The wide success and usage of register transfer languages to describe the internal operations of a computer is primarily due to the facility with which a design can be organized and the direct way a design can be translated from register transfer language into the control gating structure once the control signals have been named. Register transfer language is also widely used in giving the details of instruction repertoires, as we will see in Chap. 10.

MICROPROGRAMMING

***9.9** In the preceding sections, the control signals which sequence the operations that are performed to execute computer instructions were generated by using gates. There is another method, called *microprogramming,* which is also used to generate the control signals in an orderly fashion. This method generally involves use of a ROM to effectively store the control signals in a manner that will be described.

When a computer is microprogrammed, the individual operations between and on registers are called *microoperations*. For instance, transferring the program counter's contents into the memory address register is a microoperation. Similarly, incrementing the program counter is a microoperation, as is transferring the accumulator's contents into the memory buffer register.[9] In each case a microoperation is initiated by raising a single control signal and sequencing microoperations involves sequencing the appropriate control signals.

Figuring out a sequence of microoperations to do something is called *microprogramming*. The microprogrammer generally writes the list of operations, or *microprogram,* using a special language. Quite often a computer program is used to translate this microprogram into a listing describing the appropriate contents for a ROM, which will be used to store the microprogram. The statements that the microprogrammer writes are in a microprogramming language. This language can be very primitive or very complex.

To explain microprogramming, we use the computer layout and instructions given in the previous sections and redo the design, using a ROM to store the control signals. Therefore, the registers and control signals in Fig. 9.3 are used in the design. First we note that the basic list of microoperations needed is shown in Table 9.5. Each microoperation is described by using a symbolic notation (in effect, a microprogramming or register transfer language), and the corresponding control signal which will cause this operation to occur is also shown.

For instance, the microoperation MB → BR, which says to transfer the contents of the memory buffer register (MB) into the *B* register (BR), is made to occur by raising the control signal MB INTO BR. Notice the considerable similarity between the description in the microprogramming language and the control signal's name. This is a convenient practice, although the control signals could be named X_1, A_1, or anything desired. A 16-bit instruction word is assumed.

[9]Microoperations are just the same as register operations. The term *microoperations* is used in this area along with *microprogramming* for historical reasons. Microoperations and register transfers and operations are physically realized using the same control signals and associated gating structures.

TABLE 9.5 MICROOPERATIONS

MICROOPERATION	CONTROL SIGNAL NAME	BIT IN READ-ONLY CONTROL MEMORY
$0 \rightarrow IC$	RESET IC	C_7
$IC + 1 \rightarrow IC$	INCREMENT IC	C_8
$MB \rightarrow IC$	MB INTO IC	C_9
$0 \rightarrow AC$	RESET AC	C_{10}
$AC + BR \rightarrow AC$	ADD	C_{11}
$AC - BR \rightarrow AC$	SUBTRACT	C_{12}
$1 \rightarrow W$	SET W	C_{13}
$0 \rightarrow W$	RESET W	C_{14}
$1 \rightarrow R$	SET R	C_{15}
$0 \rightarrow R$	RESET R	C_{16}
$0 \rightarrow AC$	CLEAR AC	C_{17}
$MB_{10-0} \rightarrow MA$	MB INTO MA	C_{18}
$IC \rightarrow MA$	IC INTO MA	C_{19}
$AC \rightarrow MB$	AC INTO MB	C_{20}
$MB_{15-11} \rightarrow OP$	MB INTO OP	C_{21}
$MB \rightarrow BR$	MB INTO BR	C_{22}
$IAR + 1 \rightarrow IAR$	INCREMENT IAR	C_{23}
$C_{0-6} \rightarrow IAR$	C INTO IAR	C_{24}
$OP + IAR + 1 \rightarrow IAR$	ADD OP TO IAR	C_{25}
$0 \rightarrow IAR$	RESET IAR	C_{26}

Figure 9.10 shows a block diagram for the control system as it will be implemented. There is a ROM with 64 locations and 30 bits per address and an address register for this memory called *IAR* (microinstruction address register). Each output bit from the ROM is a control signal which will generate a microoperation, and these control signals are named C_0 to C_{26}. Seven of these outputs are special because they are *next addresses* which can be loaded into the IAR and which will be used to sequence the IAR in several cases. (This ROM is often called a *control memory*.)

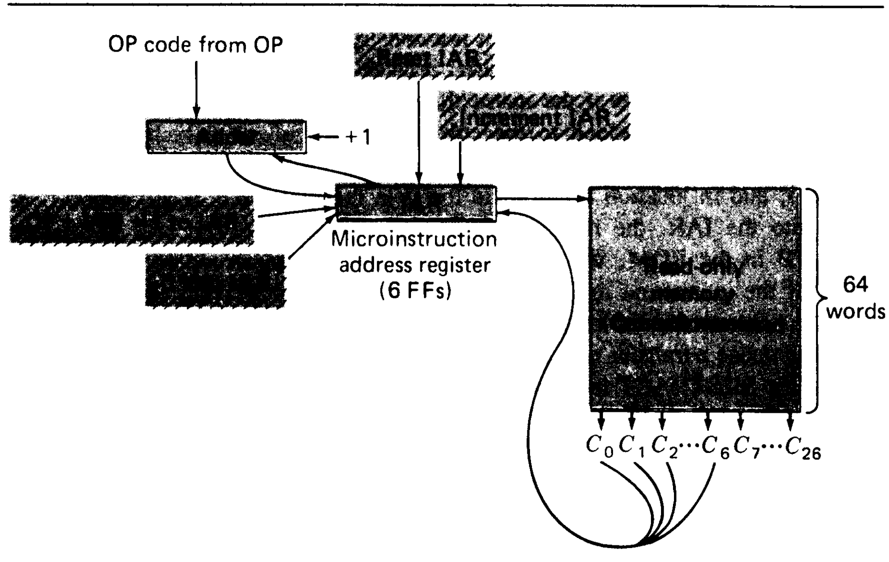
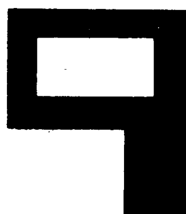


FIGURE 9.10

Block diagram for control system.



The following operations can be performed on this control unit. A 1 can be added to the IAR (in microprogramming language, IAR + 1 → IAR, the control signal is called INCREMENT IAR), and the output bits from the control memory labeled C_0 to C_6 can be transferred into the IAR. It is also possible to add the value in OP (see Fig. 9.3) plus 1 to the current contents of the IAR.

Now the basic scheme is this: The control signals to generate a given computer instruction, say ADD, are stored in a section of the control memory. The IAR sequences through this section, and at each location the outputs from the control memory will comprise the control signals. These ROM outputs then replace the control signals generated by the gates in Fig. 9.6.

The first problem is that the IAR must be set to the correct address at the beginning of that section in control memory which contains the bits storing the control signals for the instruction to be executed. To do this, we must examine the OP-code register's contents after we have read the instruction word from memory and then moved the OP-code section from the memory buffer register into the OP-code register. The complete microprogram for the control memory is shown in Table 9.6. Notice that the first microoperations performed are as follows:

LOCATION IN CONTROL MEMORY	MICROPROGRAM	
0	1 → R	IAR + 1 → IAR
1	MB_{15-11} → OP	IAR + 1 → IAR
2	OP + IAR + 1 → IAR	
3	C_{0-6} → IAR	
4	C_{0-6} → IAR	
5	C_{0-6} → IAR	
6	C_{0-6} → IAR	

The operation here is as follows. First the memory is told to read. (The prior instruction has loaded the memory address register with the location of the instruction word.) The instruction word is then in the memory buffer register when the next microoperation is performed. This microinstruction loads the OP-code register with the first 5 bits in the memory buffer register. Next this value is added to the IAR register plus 1. Now if the instruction is an ADD instruction with OP code 00000, then 1 will be added to the current IAR's contents (which will give 3 decimal). Thus the next word in the control memory to be accessed will be at location 3, and in location 3 the value for C in the first 7 bits is 20. When C is loaded into the IAR, the next microinstruction word addressed will be that at address 20 in the ROM, which contains the first microinstruction in the ADD section. If the instruction in OP was a SUBTRACT, the OP code will be 00001, and so the next word in the control memory to be used will be at location 4 decimal, which will cause a transfer to location 25, which in turn contains the microinstructions for the SUBTRACT instruction.

Therefore, an ADD instruction will cause a jump to location 20 (decimal) in the control memory, and a SUBTRACT will cause a branch to location 25. In each case these locations begin the section of memory containing the microinstructions which will cause the instruction to be executed.

At the end of each microprogram section which causes an instruction to be

TABLE 9.6	MICROPROGRAM FOR FOUR-INSTRUCTION COMPUTER	
LOCATION IN CONTROL MEMORY	**MICROPROGRAM**	**COMMENTS**
0	1 → R, IAR + 1 → IAR	Tell memory to read, increment [illegible]
1	MB[illegible] → OP, IAR + 1 → IAR	Place OP code in [illegible]
2	OP + IAR + 1 → IAR	Add OP code to 3; [illegible] IAR to be used.
3	C_{0-6} → IAR, C_{0-6} has value 20 decimal	Instruction was ADD; go to location [illegible]
4	C_{0-6} → IAR, C_{0-6} has value 25 decimal	Instruction was SUBTRACT, go to location [illegible] ROM.
5	C_{0-6} → IAR, C_{0-6} has value 30 decimal	Instruction was [illegible], go to location [illegible]
6	C_{0-6} → IAR, C_{0-6} has value 35 decimal	Instruction was STO, go to location 35 in ROM.
7–19	Left blank to add more instructions. NOTE: IAR + 1 → IAR occurs in every following line except 24, 29, 34, and 39.	
20	0 → I, 1 → E, MB_{10-0} → MA	Begin ADD instruction microoperations; place address of augend in memory address register.
21	1 → R, IC + 1 → IC	Read augend from memory; increment instruction counter.
22	MB → BR, 0 → R	Place augend in *B* register.
23	AC + BR → AC	Add and place sum in accumulator.
24	IC → MA, 1 → I, 0 → E, 0 → IAR	Set up for next instruction by placing instruction counter in memory address register and going to location 0 in control memory.
25	MB_{10-0} → MA, 0 → I, 1 → E	Begin SUBTRACT instruction microoperations.
26	1 → R, IC + 1 → IC	
27	MB → BR, 0 → R	Place subtrahend in *B* register.
28	AC − BR → AC	Subtract and place difference in accumulator.
29	IC → MA, 1 → I, 0 → E, 0 → IAR	Place address of next instruction word in memory address register and go to 0 in control memory.
30	MB_{10-0} → MA, 0 → I, 1 → E	Begin CLA instruction operations.
31	1 → R	
32	MB → BR, 0 → AC, 0 → R	Reset accumulator.
33	AC + BR → AC	Add *B* register to accumulator.
34	IC → MA, 1 → I, 0 → E, 0 → IAR	End CLA instruction; place address of next instruction in memory address register and go to 0 in control memory.
35	0 → I, 1 → E, MB_{10-0} → MA	Begin STO instruction.
36	1 → W, AC → MB	Place accumulator's contents in memory buffer register so that it can be stored; tell memory to write.
37	0 → W	
38	IC + 1 → IC	Set up for next instruction.
39	IC → MA, 1 → I, 0 → E, 0 → IAR	End of instruction, place address of next instruction in memory address register and go to 0 in control memory.

executed, the IAR is set to 0, which is the starting point for the operations that lead to reading in the next instruction and branching to the correct section in the control memory to cause the instruction to be executed.

VARIATIONS IN MICROPROGRAMMING CONFIGURATIONS

9.10 Figure 9.11 shows the microprogram of Table 9.6 stored in a memory. The implementation here has the control memory in Fig. 9.10 with its contents, as shown in Fig. 9.11. This basic configuration is used in most modern microprogrammed conputers. There are many variations on this idea, however, and there are many microprogramming languages. The references contain further descriptions and information in this area.

The microprogramming configuration shown in Fig. 9.11 has an output bit from the memory for each control signal. This is called *horizontal microprogramming*. For larger computers there may be many control signals, and thus there would be many bits in the control memory. (In general, the number of control signals varies from about 60 for small computers to about 3000 for the largest machines.) Since this would involve too large a control memory, the control signals are examined, and an attempt is made to reduce the number of outputs from the

FIGURE 9.11

Microprogram in memory. Control address C_0 has 0s in all positions.

ADDRESS IN MEMORY

CONTROL ADDRESS

$C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15} C_{16} C_{17} C_{18} C_{19} C_{20} C_{21} C_{22} C_{23} C_{24} C_{25} C_{26}$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39

Note: Only 1s are shown; remaining positions are 0s.